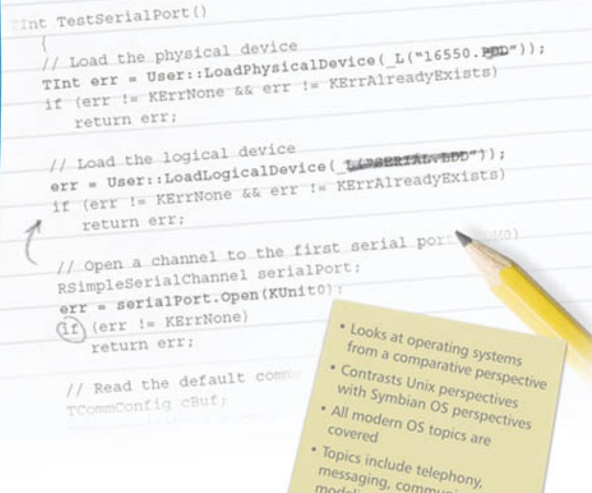
WILEY

Smartphone Operating System Concepts with Symbian OS

A Tutorial Guide



```
TInt TestSerialPort()
    {
    // Load the physical device
    TInt err = User::LoadPhysicalDevice(_L("16550.PDD"));
    if (err != KErrNone && err != KErrAlreadyExists)
        return err;

    // Load the logical device
    err = User::LoadLogicalDevice(_L("SERIAL.LDD"));
    if (err != KErrNone && err != KErrAlreadyExists)
        return err;

    // Open a channel to the first serial port
    RSimpleSerialChannel serialPort;
    err = serialPort.Open(KUnit0);
    if (err != KErrNone)
        return err;

    // Read the default comm
    TCommConfig cBuf;
```

- Looks at operating systems from a comparative perspective
- Contrasts Unix perspectives with Symbian OS perspectives
- All modern OS topics are covered
- Topics include telephony, messaging, communications modeling...
- Includes online lab manuals

symbian Academy

Michael J. Jipping

Smartphone Operating System Concepts with Symbian OS

A Tutorial Guide

Michael J. Jipping

Reviewed by

Attila Vamos, Chris Notton, Freddie Gjertsen, Gema Gomez-Solano, Ian McDowall, Jason Parker, Jonathan Yu, Kostyantyn Lutsenko, Matthew O'Donnell, Phil Spencer, Rahul Singh, Ricky Junday, Roy Ben Hayun

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb

BICENTENNIAL 1807 WILEY 2007 BICENTENNIAL

John Wiley & Sons, Ltd

Smartphone Operating System Concepts with Symbian OS

A Tutorial Guide

Smartphone Operating System Concepts with Symbian OS

A Tutorial Guide

Michael J. Jipping

Reviewed by

Attila Vamos, Chris Notton, Freddie Gjertsen, Gema Gomez-Solano, Ian McDowall, Jason Parker, Jonathan Yu, Kostyantyn Lutsenko, Matthew O'Donnell, Phil Spencer, Rahul Singh, Ricky Junday, Roy Ben Hayun

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb

BICENTENNIAL 1807 WILEY 2007 BICENTENNIAL

John Wiley & Sons, Ltd

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 6045 Freemont Blvd, Mississauga, Ontario, L5R 4J3, Canada

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Anniversary Logo Design: Richard J. Pacifico

Library of Congress Cataloging-in-Publication Data

Jipping, Michael J.
Smartphone operating system concepts with Symbian OS : a tutorial guide / Mike Jipping ; reviewed by Attila Vamos … [et al] ; head of Symbian Press, Freddie Gjertsen ; managing editor, Satu McNabb.
p. cm.
Includes index.
ISBN 978-0-470-03449-1 (pbk. : alk. paper)
1. Cellular telephones – Computer programs. 2. Symbian OS (Computer file)
3. Pocket computers – Computer programs. I. Title.
TK6570.M6J56 2007
621.3845′6 – dc22

2006102031

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 978-0-470-03449-1 (PB)

Typeset in 10/12pt Optima by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by TJ International, Padstow, Cornwall
This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Contents

Author's Acknowledgements

There are many people whose patience and forbearance meant that this book could be written. My family – my wife and three children – put up with me long enough to enable me to complete it.

There are many folks at Symbian who were instrumental in getting this book out. In particular, Satu McNabb and Freddie Gjertsen were key to this book's publication. I especially laud Satu's patience with me.

This book needed software to make it work. Kathleen Ludewig's Informant software was key to parts of the laboratory section. In addition, the creators of TaskSpy and SMan software contributed to Informant.

Symbian Press Acknowledgements

Symbian Press wishes to thank the author, Mike Jipping, for all his hard work, dedication and enthusiasm with this book and also all technical reviewers within Symbian who contributed by sharing their expertise and dedicating their time – many thanks.

Introduction

It is amazing to realize how important operating systems are to computers. So it is that the study of operating systems is of great importance as part of studying computers. Operating systems support access and innovation; they allow the complicated inner workings of hardware to be used with ease. As computers are developing rapidly, so are operating systems.

It is interesting to note that despite the change and evolution which operating systems undergo, they also remain constant. The underlying concepts of operating systems change much more slowly than the ways to adapt those concepts to new computer systems. As an example, the idea of a file has been implemented on computers for many years and will continue to be used for many years to come.

This book is written as an introduction to operating systems, with a focus on mobile phones and, specifically, Symbian OS. There are many textbooks that describe most aspects of operating systems, but most bypass mobile phone operating systems. Symbian OS is a unique and comprehensive mobile phone operating system and any complete examination of operating systems should include it.

This book is targeted at junior or senior undergraduate students. In addition to simply presenting and discussing operating system concepts, this book is accompanied by exercises that can be performed in the context of laboratory or experimental assignments. These assignments can be assigned and worked on in the classroom or a student's own time. Hands-on experience can be very important in cementing various concepts.

The Contents of this Book

- Chapters 1 to 3 provide an introduction to operating systems. They explain the history of operating systems and how operating systems came to be. They describe what an operating system is, how operating systems are designed and what their structure is. The explanation is replete with examples and sets the context for the more detailed information contained in later chapters.
- Chapters 4 to 6 describe the concepts of processes and threads and define the ways that these constructs organize a computer system. In addition to definitions, these chapters describe how processes and threads interact, providing a look at system concurrency. They cover issues involved in scheduling, communication, synchronization and handling of deadlocks.
- Chapter 7 describes how operating systems manage a computer's memory. The focus here is on how main memory is managed during process execution. It is here that differences between mainframes, desktops and mobile computers are evident. This chapter describes and discusses these differences.
- Chapter 8 discusses files and the ways operating systems have been invented for presenting files to users. There are several different systems for managing files, but the file concept itself is the same across platforms. This chapter describes the classic file-management algorithms as well as the ways files are handled on different platforms.
- Chapters 9 and 10 discuss computer input and output (I/O) and how important the management of I/O is to the running of a computer system. I/O management is probably the most crucial to a computer system because it deals with the slowest components of a computer system. Chapter 9 describes I/O in depth, considering design, interfaces and internal structure. Chapter 10 extends the ideas from Chapter 9 to apply to a communications network.
- Chapters 11 to 13 describe and exemplify how operating systems deal with communication. It is in these chapters that Symbian OS begins

to shine, because it is designed expressly for device communication. Chapter 11 deals with communication models in general, describing and discussing ways that operating systems build models for communications. Chapter 12 applies the concepts from Chapter 11 to telephony; Chapter 13 applies these concepts to messaging facilities.

- Chapter 14 deals with system security. Security extends to all major areas of a computer system and this chapter discusses the application of security ideas to processes through files and into communications.
- Chapter 15 provides a case study of how the operating system concepts of the previous chapters can be applied to an interesting new area of development. Virtual machines provide an area that needs management – through an operating system – but in special ways that adapt to its unique implementation. This chapter describes virtual machines and how operating systems address them.

The Laboratory Exercises

The laboratory exercises (which can be found at ***www.wiley.com/go/jipping***) is designed to get students to experiment in the design and implementation of operating systems. We focus on Symbian OS, but to do this we compare and contrast Symbian OS with other operating systems. In many cases, we compare Unix/Linux, Microsoft Windows and Symbian OS.

To follow the laboratory experiments, you need an implementation of Unix. This is for two reasons: Unix provides many comparisons to Symbian OS and other operating systems and there are concepts that Symbian OS does not address that are neatly exemplified in Unix.

Almost any implementation of Unix will do for this: Linux is the most widely used and works well. There are many 'live CD' implementations that do not require you to spend money (they are free) or to dedicate computing resources. You can boot a live-CD implementation directly from a CD-ROM and it runs completely in memory. It does not affect the PC hard drive and any installed software is not affected.

The implementation we use in this manual is Knoppix version 4.0. You can download a CD image of Knoppix from ***www.knoppix.org***. You can also use Ubuntu Linux, which can be found at ***www.ubuntu.org.***

1

Introduction to Mobile Phone Systems

The phrase 'viewing the world through rose-tinted glasses' finds its origins in literature at least as far back as 1861. The phrase implies that 'viewers' have a different – usually optimistic – view of the world from the 'standard' view, as if they are seeing it through a set of nicely tinted lenses. Computer operating systems are like tinted glasses, allowing the viewer to see a collection of hardware and software – memory, disk drives, CPU chips, Bluetooth transmitters, email programs and telephony applications in an ordered and controllable way: as a set of resources that can be harnessed to accomplish various tasks. An operating system is the model through which a computer's hardware and software can work together and the structure that provides controlled access between them.

Consider the many different sets of 'tinted glasses' that are in use today for manipulating computing resources. Many of today's hardware platforms are used by multiple operating systems. For example, Intel-based hardware, such as the Pentium family of CPUs, can support several different operating systems. The Microsoft Windows family of operating systems represents a set of many different operating systems – from Windows 95 to Windows XP – that run on the same hardware platform. The Linux operating system and BeOS provide other examples. These different systems form a set of different models of resource allocation and usage that operate on the same hardware. These operating systems are very different in how they view a computer system, but they are very much the same in many respects.

This book takes a close look at the variety of operating systems with a focus on a specific type of operating system: that of mobile phones. Mobile phone operating systems must embrace conventional system components as well as additional components crucial to mobile phones: communications and interface design. We look at each of these additional components. To be more specific, this book looks at mobile phone operating systems by examining Symbian OS. Symbian OS is an operating system that was designed from its beginnings to be implemented on mobile phones. Its design comprises conventional operating system modeling, employs a strong communications model and has a very flexible user interface model. Its origins are found in handheld computing and its usage on mobile platforms is growing dramatically. (It is predicted that, by 2008, half of all mobile phones will have a full-featured operating system, such as Symbian OS, running them.)

It is difficult to study mobile phone operating systems, even given the plethora of mobile phones, without also looking at conventional operating systems. We examine operating systems that power servers and desktop systems. We compare Symbian OS to these conventional systems, especially by comparing it to Linux.

In order to study operating systems, we must first define what an operating system is and understand the divide between an operating system and a hardware device. This chapter defines operating systems and the components that make them up. It then looks at the history of operating systems, including a history of Symbian OS. It finishes by looking at how operating systems fit onto various computing platforms.

1.1 What Is an Operating System?

There are many definitions of an operating system. All definitions agree on several points. First, an operating system is a software program. No matter where it is stored – on a hard drive, in ROM, on compact flash storage – an operating system is eventually loaded into a computer's memory and its instructions are executed just like any other software program.

Secondly, an operating system is a resource model. Operating systems are designed to present the various hardware resources of a computer to software and to a user. An operating system builds a model, a system, of how to deal with the resources of a computer. Software must work with this model to access and use those resources. The model provides a lens through which users view resources such as the communications system and the user interface.

Thirdly, an operating system binds the hardware and the software together. Because it presents the hardware to the software, an operating system is the glue that holds the two sides together. The software sees and accesses the hardware as it is presented through the operating system model. The hardware deals with the software through the same operating system model. A good operating system is based on an intuitive model that allows effective communication between the software and the hardware.

Finally, an operating system is essential. Without an operating system, a computer would not function. Its software could not be executed; its hardware would not be utilized. Any general-purpose computer has an operating system in some form. Thus, learning about operating systems means learning about an essential part of the computer.

The Operating Environment

To understand operating systems as the glue between hardware and software, let us examine these two elements and how they relate through the operating system.

Hardware is the physical part of the computer. It is the set of all the tangible components that provide the operational foundation for the software. Software is the set of programs and applications that execute their instructions on the hardware. A software program must use hardware in some way – for input, for output or to operate the hardware somehow.

Consider the example of a message manager application running on a mobile phone (see Figure 1.1). It collects text messages as they arrive, analyzes each message and responds to certain ones as the application's user has specified. The hardware receives radio signals and notifies the operating system that data is arriving. The operating system engages the sending source by working with the radio hardware to receive a text message using the appropriate data protocol. Once the complete message has arrived correctly, the operating system stores the message and notifies the message manager. The message manager application uses the operating system to access the stored message – which requires the operating system to interact with the hardware. The manager reviews the message and takes some kind of action, perhaps deleting the message or making an automatic reply. The automatic reply again requires the operating system to create a new message and to access the hardware for storage and transmission of the new message.

It is important to realize here that neither the hardware nor the software sees an operating system. The hardware is following a prescribed set of

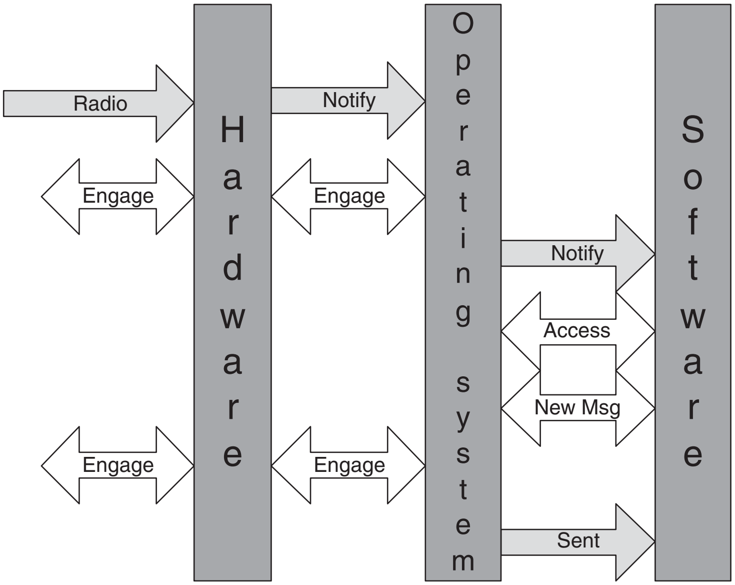


Figure 1.1 The relationship between hardware, operating system and software

instructions built into its memory. The software is using an application programming interface (API) to manipulate text from storage and to compose and send a message. Both sides see a different picture, yet both sides are drawn together and work to accomplish a joint goal. The operating system acts as the go-between and provides an operational picture to both sides.

The focus of a mobile phone is in the software that enables a user to use it. It is software that enables a user to make a phone call, send a message, set an alarm, or write on the display with electronic 'ink'. The user of the phone realizes that the hardware exists – it is in his hand, after all – but is most likely not aware of the operating system. A good operating system is transparent, allowing the user to use the software to interact with the hardware without showing its own face.

A Resource Model

The focus of an operating system is on providing ways for the software to use the hardware to do what the user wants. It is the goal of an operating system to make this happen seamlessly and transparently. Essentially, the operating system must provide the software with an accessible model of the hardware. The hardware must become a set of resources to be operated by the software. Management of that hardware resource is the job of the operating system.

Software manipulates hardware resources through an application programming interface. APIs can be provided by the operating system designer or by a third party. Software does not usually work with hardware directly, but manipulates resources by communicating with the operating system through a function call interface. The operating system builds a model of the hardware and provides system function calls that access that hardware model in specific ways (see Figure 1.2).

Consider the previous message manager example. The operating system has many choices to make as it works with messages. It could, for example, store the message text in a file and give an application a way to find the file name and to work with that file directly. The application would have to open the file (again, through the operating system resource model) and process the raw message data. Another way to present the message would be to store the message in a file, but present an application with an abstract object called a 'text message' that the application could work with. The application would make function calls that the operating system would intercept, deriving information about the message and returning that information. The application would not be aware of where the object was stored. These are two models of message handling: one

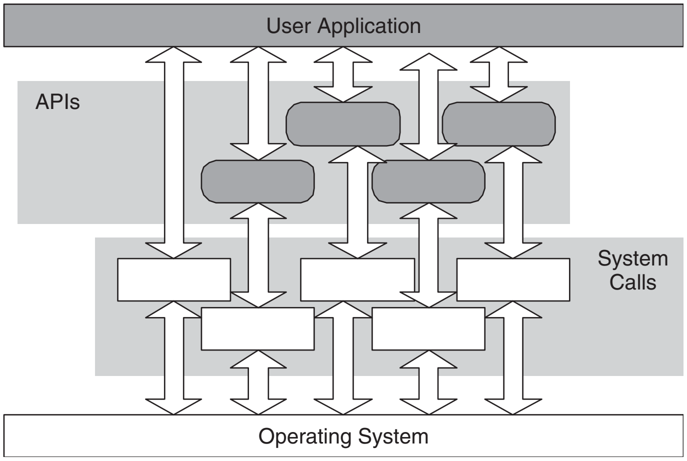


Figure 1.2 Structure of access to an operating system

more raw and direct, the other more abstract and object-oriented. The choice that the operating system makes about which one to use builds the character of the operating system.

A good system model is one that effectively and transparently provides software with an intuitive way to access system resources. System models are often based on *abstraction*. Abstraction involves the hiding of irrelevant data and the presentation of only useful, relevant information. We often label abstractions as 'objects'. For example, system resources are the abstract objects that the operating system presents to the software. They might represent a resource as a hardware object, with the detail abstracted away, or as a set of functions that can use the hardware. A file is an abstract object that represents a way to use hardware storage. A text message is an abstract object that represents a way to use both software and hardware resources to access that message. These are concepts built and supported by the operating system and provided to applications.

A good operating system has more goals than simply providing a useful model to software applications.

- *Robustness*: a good operating system is reliable and tolerates problems well. The system does not stop working due to isolated hardware or software errors and fails gracefully if it must deal with several errors at the same time. Robust operating systems provide services to software unless the hardware fails.

- *Scalability*: a good operating system incorporates resources as they are added to the system. This can be transparent to the user – the best way – or can involve some kind of user interaction. The plug-and-play concepts of Microsoft Windows – where devices are discovered and installed automatically – is an example of good scalability. On the other hand, old versions of Linux used to require recompilation of the operating system when new devices were added. This is an example of bad scalability.

- *Extensibility*: the operating system should be designed to adapt to new technologies that extend the operating system beyond the point at which it was implemented. For example, it should be able to adapt to new forms of file storage without a complete redesign of the operating system.

- *Throughput (the work that a processor can complete in a specific time period)*: an operating system must perform well and achieve high throughput. A good operating system minimizes the time spent providing services while maximizing throughput.

- *Portability*: a good operating system should be portable, that is, able to be run on many different hardware platforms.
- *Security*: an operating system must be secure. It must prevent unauthorized users and processes from accessing stored data and system services.

Many Operating Systems Fit the Bill

Even though the list of criteria for a good operating system looks a bit daunting, many operating systems have been created over the years that meet these criteria. In addition, many operating systems did some of these very well and steered the industry in one particular area. A list of operating systems can be found at ***http://en.wikipedia.org/wiki/List_of_operating_systems***.

Many operating systems are not very portable. They are specifically designed to run on a single platform. In addition, you will note that 'popularity' is not an item on the criteria list. Most operating systems were not popular, yet were designed to address a specific system model.

1.2 History of Operating Systems

Operating systems are the heart of every general-purpose computer. Since 1957, operating systems have been an essential component of computers. This section outlines a brief history of operating systems, highlighting the history of Symbian OS.

General-Purpose Operating Systems

The earliest computers did not have operating systems. They were dedicated computing devices that performed a single task, thereby needing only one 'program' to execute. From the ancient Incas in Central America to the Difference Engine constructed by Charles Babbage in 1847 to the early days of modern computing (the ENIAC in 1946, the Mark I in 1948), early computers focused on single tasks that had direct access to hardware and no operating system.

Operating systems were invented when it became clear that access to 'the system' needed to be standardized. Until the mid-1950s, programmers wrote their own routines for accessing resources, particularly system input and output. Patterns of programming were beginning to emerge, such as repeated use of certain mathematical functions. The need for basic, standardized operating system functionality, including device drivers and execution libraries, was becoming apparent. Critical mass was reached as computer systems were designed to allow queuing of jobs, or programs, to run one after the other.

The first operating system was released in 1957. Called BESYS, this operating system was implemented by Bell Labs to handle the execution of many short programs, queued up so that the operators did not have to load each program just prior to its execution. BESYS shared CPU time between several jobs at once, thus making it the first multitasking operating system.

Operating system research and implementation moved very fast in the 1960s. Two influential examples were OS/360, released by IBM in 1964, and MULTICS, released by Bell Labs, MIT and General Electric in 1965.

OS/360 was influential because it combined a powerful command language with the ability to run many jobs at once. The command language controlled job execution and specified how each job was to access resources. In addition, OS/360 worked on various computer models; it became the standard among batch processors.

MULTICS was influential because it took a very different approach from OS/360: it allowed users to use the operating system directly. It had a unique structure – using a central core of software called a 'kernel' – and allowed users to extend the operating system through software based on the kernel. Based on the foundational ideas introduced in MULTICS, Unix was invented at Bell Labs by a man named Ken Thompson in 1972. Thompson teamed with Dennis Ritchie, the author of a programming language called 'C', to produce the source code of the Unix operating system in that language. Unix was distributed almost free of charge and, in the 1970s, it spread to many platforms.

Since the spread of Unix, there have been many developments in operating systems. One of the biggest was brought about by a development in computers: the personal computer. The ideas invented by MULTICS and honed by Unix were streamlined to fit into a personal computer with the introduction of MS-DOS in 1981. MS-DOS ran on an IBM PC using the Intel 8088 chipset. Its first version was indeed primitive, but as hardware resources were improved upon and faster processors with more memory were packaged as desktop computers, MS-DOS evolved into Microsoft Windows and has taken on many of the foundational concepts embedded in Unix.

As we look at the evolution of operating systems, it is interesting to see the progression of computer resources that also evolved:

- computers started by running one task at a time and have progressed to running many tasks at the same time
- storage hardware has evolved from needing a large physical size for only 100 KB of data to packing 100 GB into a matchbox-sized disk
- electronic storage has made access much faster
- memory has progressed from only a few kilobytes to many gigabytes; even handheld and mobile phone platforms sport 128 MB (and larger) memories
- communication has gone from none to a large collection of possibilities: wired and wireless, serial and parallel, radio and infrared.

Operating systems have developed to take advantage of all of these aspects of computer hardware.

Symbian OS

Handheld devices were developed in the late 1980s as a way to capture the usefulness of a desktop device in a smaller, more mobile package. Although the first attempts at a handheld computer (for example, the Apple Newton) were not met with much excitement, the handheld computers developed in the mid-1990s were better tailored to the user and the way that they used computers 'on the go'. By the turn of the 21st century, handheld computers had evolved into smartphones – a combination of computer technology and mobile phone technology. Symbian OS was developed specifically to run on the smartphone platform.

The heritage of Symbian OS begins with some of the first handheld devices. The operating system began its existence in 1988 as SIBO (an acronym for '16-bit organizer'). SIBO ran on computers developed by Psion Computers, which developed the operating system to run on small-footprint devices. The first computer to use SIBO, the MC laptop machine, died when it was barely out of the gate, but several successful computer models followed the MC. In 1991, Psion produced the Series 3: a small computer with a half-VGA-sized screen that could fit into a pocket. The Series 3 was followed by the Series 3c in 1996, with additional infrared capability; the Sienna in 1996, which used a smaller screen and had more of an 'organizer' feel; and the Series 3mx in 1998, with a faster processor. Each of these SIBO machines was a great success, primarily for three reasons: SIBO had good power management, included light and effective applications, and interoperated easily with other computers, including PCs and other handheld devices. SIBO was also accessible to developers: programming was based in C, had an object-oriented design and employed application engines, a signature part of Symbian OS development. This engine approach was a powerful feature of SIBO; it made it possible to standardize an API and to use object abstraction to remove the need for the application programmer to worry about data formats.

In the mid-1990s, Psion started work on a new operating system. This was to be a 32-bit system that supported pointing devices on a touch screen, used multimedia, was more communication-rich, was more object-oriented, and was portable to different architectures and device designs. The result of Psion's effort was the introduction of EPOC Release 1. Psion built on its experience with SIBO and produced a completely new operating system. It started with many of the foundational features that set SIBO apart and built up from there.

EPOC was programmed in C++ and was designed to be object-oriented from the beginning. It used the engine approach pioneered by SIBO and expanded this design idea into a series of servers that coordinated access to system services and peripheral devices. EPOC expanded the communication possibilities, opened up the operating system to multimedia, introduced new platforms for interface items such as touch screens, and generalized the hardware interface. EPOC was further developed into two more releases: EPOC Release 3 (ER3) and EPOC Release 5 (ER5). These ran on new platforms such as the Psion Series 5 and Series 7 computers.

As EPOC was being developed, Psion was also looking to emphasize the ways that its operating system could be adapted to other hardware platforms. From mobile phones to Internet appliances, many devices could work well with EPOC. The most exciting opportunities were in the mobile phone business, where manufacturers were already searching for a new, advanced, extensible and standard operating system for its next generation of devices. To take advantage of these opportunities, Psion and the leaders in the mobile phone industry – for example, Nokia, Ericsson, Motorola and Matsushita (Panasonic) – formed a joint venture, called Symbian, which was to take ownership of and further develop the EPOC operating system core, now called Symbian OS.

Symbian OS was explicitly targeted at several generalized platforms. It was flexible enough to meet the industry's requirements for developing a variety of advanced mobile devices and phones, while allowing manufacturers the opportunity to differentiate their products. It was also decided that Symbian OS would actively adopt current, state-of-the-art key technologies as they became available. This decision reinforced the design choices of object orientation and a client–server architecture.

1.3 Computer Systems and their Operating Systems

In addition to following computers and their history, a different way to appreciate the relationship between operating systems and hardware is to look at them from a system perspective. Each type of computer system has an operating system that was designed for it – to take advantage of its unique features.

Mainframe Systems

Mainframe systems are characterized by a large central computer with a large number and wide variety of possible peripherals. These types of computers were the first to be used to run scientific and commercial applications.

Initially, mainframe systems needed to run only a single program at a time. The operating system would accept *jobs* – packages consisting of control commands, program code and data. The control commands dictated how to compile the program, how much memory it would take, what other resources would be used, etc. Operating systems for these types of computers could be quite simple. An operating system needed to read in the job, use the control commands to configure how the program would be loaded up and executed, and manage the program's access to resources and data. When a program executed, the operating system would remain in memory, tucked away in its own section. The BESYS operating system was created in this environment.

Mainframe systems became more complex for two reasons. First, running multiple jobs in sequence became desirable. A sequence of jobs – called a *batch* – would be sorted into groups based on what resources would be used. Often, using a resource required that the resource be on and configured in a certain way. Secondly, disk technology developed to the point where jobs could be placed on a disk drive rather than recorded on punched cards. This was a great step forward, because mistakes were easier to correct, and jobs could be submitted and processed more rapidly. Once disk access was available, an operating system could sort the jobs and choose which was most appropriate to run at a given time. This type of *job scheduling* allowed more efficient use of computer resources in addition to faster turnaround time for program execution.

In this kind of environment, idle time becomes an issue. There was a large difference between the speed of the CPU processor and the I/O speed of each device connected to the computer. Therefore, as the CPU accesses a device, much waiting is involved. This problem was exacerbated by the fact that older mainframes would run a single job at a time.

Eventually, mainframes and their operating systems came to embrace two more concepts: *multiprogramming* and *time-sharing*. To take advantage of the waiting time of a CPU, operating systems were designed to schedule multiple jobs at once. These several jobs would share the CPU: when one job caused the CPU to wait, another job took its place and executed on the CPU. This type of multiprogramming – where multiple programs ran on a single CPU – extended the idea of job scheduling to include *CPU scheduling*. This multiuse environment has several implications for memory and for I/O.

Time-sharing is an extension of CPU scheduling. If you consider a user interacting with a computer as just another job, then multiple users can interact with the computer at the same time. Time-sharing refers to the way that users share the CPU with other tasks, both other users and other jobs. OS/360 was implemented to support this kind of environment: a time-sharing, job-scheduling computing environment. Users would interact with the computer by creating jobs through terminals, saving them, then submitting them online to the computer. Output from these jobs was eventually generated and delivered to the user for consideration.

Mainframe systems shrank in size and eventually became small enough to put into a room with very little cooling equipment. The user interaction software evolved as well. The job-control program eventually became a *command shell*, a program that accepted commands interactively from a user, executed those commands and placed the output back on the screen. MULTICS was created in this environment and Unix perfected the use of this kind of interaction. Multiprogramming was the norm in these operating systems and all 'jobs' – including the user-command shell – competed for system resources, especially the CPU. Issues that affected performance – such as which job got priority and algorithms to effectively schedule all usage of the CPU – became very important and widely discussed.

Desktop Systems

Computers continued to shrink until it was feasible to combine a monitor, a CPU and a keyboard into a single package that could occupy a desktop. These systems distributed computing power to users, rather than having users access the computing power of a single machine.

IBM constructed the first personal computer; MS-DOS was the operating system that was used for this first PC. Initially, MS-DOS was a single-job operating system. Like the old mainframes, it ran a single job (now called a *process*) at a time and the operating system made choices about which job to run and how to manage resources. Hardware systems grew faster and supported more peripherals; operating systems, like those supporting mainframes, grew and added features to support these hardware systems. MS-DOS eventually incorporated multiprogramming and could support multiple processes using the CPU. As graphical user interfaces became more widely used to interact with the computer (in the place of a command shell), MS-DOS was upgraded to become Microsoft Windows and other operating systems, such as MacOS from Apple, emerged.

Desktop systems now support multiprogramming, time-sharing, networking and many types of peripherals. These systems assume that they exist in an environment that is shared by multiple PCs and multiple users. The operating systems embrace many users at once and encourage users to venture out over networks to share resources from other computers.

Distributed Systems

A distributed system is an extension of multiple connected stand-alone systems. These systems depend on each other to varying degrees. Some distributed systems simply share a few resources – such as printers and disk drives – while others share many resources – such as CPU time and input devices. Distributed systems assume that they are connected by some sort of communication network.

There are several models of distributed systems that operating systems have taken advantage of. The *client–server* model views some computers as servers, that is, providing a service of some sort, and some computers as clients that ask for and receive a service. Web browsing is a distributed activity that is based on the client–server model. Browsers are clients that ask servers for pages. *Peer-to-peer distribution* is a model in which computers are both servers and clients, using some and being used by others. The *interdependent* model is a peer-to-peer model where peers are tightly interconnected, such that they cannot operate if other peers are not also functioning. In the interdependent model, each peer has functions that are crucial to the entire network's operations.

There are several examples of operating systems for distributed computing systems. Good examples of the client–server model are the many distributions of Linux. The appeal of Gentoo Linux is that it is solely based on the Internet for its distribution. It uses the Internet for upgrading itself, for installing itself and for updating its applications. For these uses, the operating system is a client, communicating with one of many Gentoo servers.

For an example of an interconnected distributed operating system we have to go back to the 1980s. During those years, an operating system called Domain/OS was implemented that ran on computers made by the Apollo company. Domain/OS was a version of Unix that was truly distributed between computers on a network. The execution of a command or program might occur on the local computer a user was connected to or it might occur on another computer in the network. No matter where the command was executed, the results – text or graphics – appeared on the local screen. The decision about which specific computer executed any given command was based on an algorithm, which made the location decision based on factors such as load and network performance.

Handheld Systems

As computers inexorably shrank in size, handheld devices became feasible. These computers – usually fully fledged systems with all the peripherals and issues of desktop systems – fit into and can be used with one hand. At first glance, these systems look as if they could simply take on the operating systems of their bigger siblings, but they pose some unique challenges.

First, the internal environment is more restrictive. Less memory, less storage space and slower processors all dictate that the operating system must be tailored for a handheld environment, not just shrunk. Often, memory becomes 'disk' space: memory space is shared between a storage system and memory used by the system to run programs. The early Palm handhelds had 2 MB of memory for operating system space and file storage. In the face of these restrictions, the conventional models of operating systems change to accommodate the different environment.

Secondly, resources must be handled with more care. The resources on a handheld platform are more fragile – in the sense that a restricted environment puts more of a load on a resource. A restricted environment leaves less room for software to protect a resource. This means that an operating system must have a good model in place for dealing with resource access from multiple sources.

Thirdly, power restraints are crucial. While desktop systems are always connected to AC power, handheld systems are almost always run on batteries. Extensive running of hardware resources drain battery life dramatically. And power loss must be handled gracefully.

These considerations mean that an operating system must be written specifically for a handheld device. It faces many pressures; it must support the multiprogramming of a desktop system in a (sometimes severely) restricted environment that must sip battery power while coordinating access to many resources. This is a considerable task, but operating systems have risen to handle it. Linux has been scaled to fit on several handheld devices. Microsoft Windows has also been fitted for handheld platforms. The early versions of Symbian OS were designed for a handheld environment.

Mobile Phone Systems

As even handheld devices got smaller, it became possible to fuse a handheld device with a mobile phone. All the considerations of a handheld platform are multiplied when a handheld device becomes a communications tool. All the restrictions and issues are present while the system requirements take on communication issues as well. The resource model of the handheld platform is now augmented with communications and the functionality that comes with those communications.

On a mobile phone, the environment restrictions can be even more severe than on a handheld device. The data requirements of multimedia communication – text messages, phone calls, photographs, video clips and MP3s – are tremendous, yet must fit onto a restricted storage space. A mobile phone now has even more resources that must be carefully dealt with. And power is even tighter than normal, as the power requirements of a mobile phone are much higher than that of a handheld device.

In the face of even tighter constraints, operating systems have risen to the challenge. Several operating systems, such as Symbian OS, have been tailored for mobile phones.

Real-time Systems

A real-time system is a special-purpose computer system where rigid time requirements have been placed on either the processor or input/output operations. These time constraints are well-defined and system failure occurs when they are not met.

Real-time systems come in two varieties. Hard real-time systems guarantee that time constraints are met. Soft real-time systems place a priority on time-critical processes. In both cases, real-time systems have a specific structure. Any time-consuming task or device is eliminated and real-time service often comes from a dedicated computer. Disk drives or slow memory cannot be tolerated. All system services – hardware or software – must be *bounded*; that is, they must have specific response-time boundaries or they cannot be used.

In a sense, some mobile phone functions are real-time functions. The service of a phone call, for example, is a real-time service. But most functions of a mobile phone can be carried out by a non-dedicated, general-purpose operating system designed for the mobile phone platform.

Symbian OS was not initially a real-time operating system but the latest versions (Symbian OS v9 onwards) are powered by a real-time kernel.

1.4 Summary

This chapter has introduced the idea of an operating system and its relationships to both hardware and software. We defined what an operating system is and discussed the modeling that an operating system does for both hardware and software. We examined the operating systems from a historical perspective and an operational perspective.

The next chapter considers the character of operating systems. It discusses some of the common features of operating systems as they exist today and makes some working definitions that we use throughout the book. We also take a much closer look at the central operating system of this book: Symbian OS.

2
The Character of Operating Systems

Like humans, operating systems have a character. The character of an operating system is the collection of design ideas, software components and usage policies that you find in its implementation. This collection gives an operating system identifying marks and is the reason that people can celebrate certain facets or commiserate about features they struggle with.

The character of an operating system can be found in how it is implemented on various types of hardware. That character evolves over time, especially as the operating system takes various shapes through versions of its implementation.

This chapter looks at the various aspects of an operating system's character by introducing operating system concepts. We begin by looking at how operating systems evolve over time and see how operating systems view certain concepts – from ideas of disk storage to software protection. We then take a look at several examples: specific operating systems and their character.

2.1 The Evolution of Operating Systems

As we saw in Chapter 1, the first operating system was released in 1957. This operating system, BESYS, was closely matched to the hardware it ran on. Since this first introduction, operating systems have evolved as the hardware they run on has evolved. New concepts have been designed and implemented; some have caught on and some have died out.

Operating system design begins with a conceptual model of computer structures. Each operating system embodies a model of the hardware on which it is running. Good operating systems weave this model throughout their design. They set up and implement abstract concepts and let various implementations put those concepts to use on the hardware platform on which the implementations run.

A good example here is the concept of a *server*. The idea of a server was initially developed as a provider of service to other computers. Operating system designers needed a way to protect a system resource while providing easy access to it. In addition, this access needed to be provided *abstractly*, that is, in a modular way that hid the server's implementation. So the idea of a server was an application that would protect resources while providing access to those resources by answering message-based requests. This is an interesting implementation. Certainly, this could have been done using different concepts and in different ways, but the server has proven to be an effective conceptual model of resource management. Servers are used in many operating systems to work with all kinds of resources.

As another example, consider two different approaches to communications. The Unix operating system uses a file-oriented model to provide access to communication resources. In a Unix system, if a user needs access to a serial port, for example, she might 'open' a 'file' called `/dev/ttya`. A Unix operating system builds device software (called *drivers*) into its model of files; the 'open' system call executes a device driver if the user is opening a device for access. Microsoft Windows creates an API for each device – usually *layers* of APIs for each device. Windows has some of the ideas of Unix; it uses the nomenclature of files to address device ports. But it also designs the access to devices through unique APIs, not file I/O mechanisms. By contrast, consider the approach that Symbian OS takes to communication. In Symbian OS, servers are used to allow access to communication resources. While a Symbian OS user would still use an 'open' call (again using the Unix file nomenclature) to gain access to a serial port, she would first have to connect to the server that provides access to that port before she could open the port.

In all models, concepts of abstraction and modularity are preserved. Actual implementations are not specified; the abstraction of the conceptual model is the important part at this stage. Much research and opinion has been dedicated to the question of which model is best; both models have held up well under such scrutiny.

The evolution of operating systems is usually spurred by changes in hardware and models that address these changes. The roots of Symbian OS give a good example of this evolution. As we discussed in Chapter 1, Symbian OS finds its roots in EPOC, an operating system developed for handheld computers.[1] Mobile phones were a burgeoning technology and EPOC's designers wanted to address that technology. EPOC, however, had no models to address telephony and therefore did not extend well to phone-based devices. EPOC evolved into Symbian OS to address mobile phones. As evidence of this evolution, one can spot much code in Symbian OS that has been derived from EPOC. New models addressing new technology had to be developed, naturally, and one can see how new hardware and new technology drove operating system development.

All Operating Systems Evolve

All operating systems evolve. Some go extinct; some survive. There are many references for operating system evolution. Check out ***www.levenez.com/unix*** for a 'genealogical' look at the Unix operating system. The evolution of Microsoft Windows is documented by Microsoft at ***www.microsoft.com/windows/WinHistoryIntro.mspx***. The evolution of operating systems that run on Apple computers can be found at ***www.kernelthread.com/mac/oshistory***. Evolution of MS-DOS, the Microsoft operating system built for early PCs, continued until Microsoft Windows arrived. MS-DOS formed a foundation for Microsoft Windows until Microsoft Windows 2000, but was then relegated to extinction.

2.2 Computer Structures

As we consider how operating systems address computer systems, we should first outline what structures those systems are built from and how

[1] Of course, you can think of the mobile phones on which Symbian OS runs as handheld computers. By definition, mobile phones are indeed handheld computers. However, we distinguish handheld computers from mobile phones by defining handheld computers as a generic term describing computers that do not use telephony. Mobile phones are, then, handheld computers that use telephony.

they are used. The character of an operating system is determined, in part, by the structures it has to address.

System Structure and Operation

Kernel Structures

The core programs and data of an operating system together comprise the *kernel*. The kernel consists of the code that runs the operating system on a CPU and the data – typically organized in tables – that are used to keep track of how things are running on an operating system. The kernel is where the access to hardware is done and the kernel implements the operating system's design model.

There are several types of kernel. *Monolithic kernels* are found on some general-purpose computers; they implement all operating system functions and hardware abstractions within the kernel itself. This type of kernel (see Figure 2.1) usually comprises large amounts of code and large amounts of memory for system tables.

Microkernels provide only a small set of system function and hardware models. Much of the remaining functionality that might be found in a monolithic kernel is provided by server applications that run outside a microkernel (see Figure 2.2). Servers in Symbian OS provide this type of functionality.

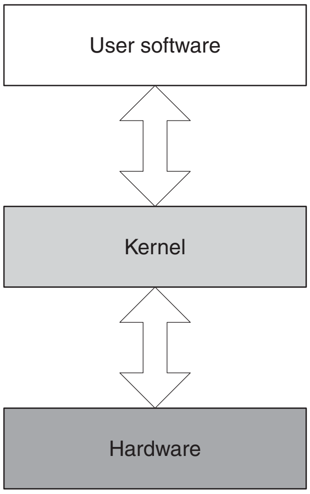


Figure 2.1 Monolithic kernel structure

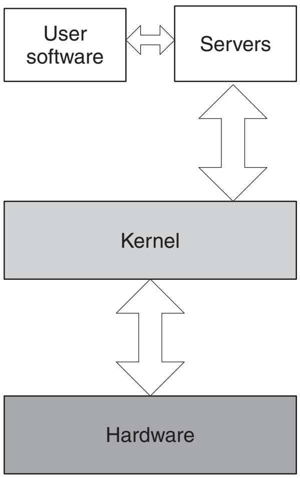


Figure 2.2 Structure of a microkernel

Hybrid kernels are like microkernels, except that some of the external application function is implemented in the kernel for performance reasons (see Figure 2.3).

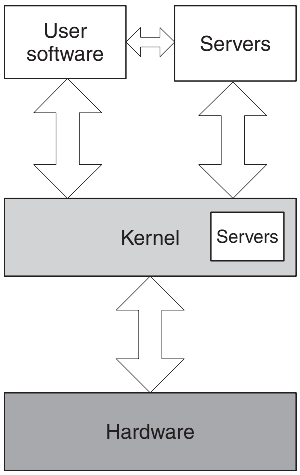


Figure 2.3 Hybrid kernel structure

Linux is typically considered a monolithic-kernel operating system. Most system functions are implemented in 'kernel space' (by the code and within the memory of the kernel). Symbian OS is implemented via a microkernel. The example in Section 2.1 of defining and opening a communication device serves well here. The implementation of devices and how they are accessed in Linux is built into the kernel. To change the implementation, one would have to change kernel code and recompile the entire kernel. In Symbian OS, devices are implemented by server – not kernel – functionality. To change the way communication devices are implemented in Symbian OS, one would have to change the code to the server and recompile it. No changes would have to be made to the microkernel itself.

Most modern systems are based on hybrid kernels. The most effective arguments against monolithic kernels are that small changes to the system require changes to the entire kernel and that errors in the kernel can cause an entire system to crash. Monolithic kernels are also larger and may not be suitable for devices with limited memory or systems that make good use of virtual memory. Hybrid systems work around these problems by pushing many kernel functions to servers and by taking extreme care to make the functions in the kernel modular and abstract.

Monolithic systems have implemented several features to help them be more flexible. For example, Linux implements the use of *modules*, which are code libraries loaded at run time that implement support features of the operating system. If the system Linux is running on has a USB port, Linux can load the USB module to drive the port. Note however, that while this allows flexibility and implementation outside the kernel core, once a module has been loaded, its operation and data become part of the kernel, adding to its monolithic character.

Interrupts

Modern computer systems are typically built from components which communicate with each other over a bus structure (see Figure 2.4).

Notice that each device in Figure 2.4 is connected to the system bus through a controller. These controllers are specific to each device and communicate with each other, sharing and competing for bus access. Controllers act as a liaison between devices and a communication medium.

In this system, the CPU must be the primary controlling device. Hence, across the bus, there is a hierarchy of device priorities and a way for devices to work with this priority system. Device controllers can communicate with any device sharing the bus and their communication can be pre-empted by other devices with higher priority.

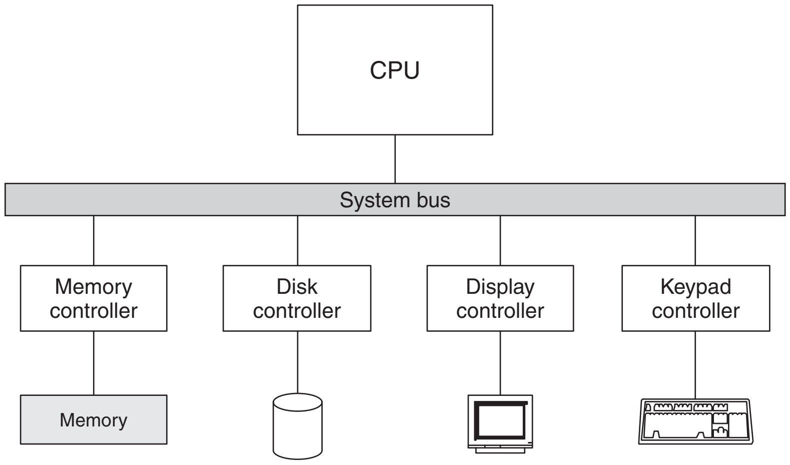


Figure 2.4 Structure of a generic computer system

In a bus-based system, it would be a waste of time to continuously check or listen to the bus to see if any device is communicating. Imagine stopping to pick up and listen to the telephone every several seconds to see if someone wants to talk to you. Instead, the bus system is driven by *interrupts*. An interrupt is like the ringing of a telephone: it is an event that is designed to get the attention of hardware, software or both. Normally, a device is intent on doing a specific task and does that task until its attention is drawn away elsewhere – for example, it finishes its task or has a problem. The device can raise an interrupt to alert the CPU. When interrupted, the CPU records what it was doing and services the interrupt, returning to its previous task when the interrupt service has been completed.

Device communication is thus designed around this interrupt mechanism. In fact, such communication is typically based on a system of interrupts. Interrupts are serviced by *interrupt service routines* (ISRs) via a table of vectors. These vectors are addresses of ISR-handling functions that a device is directed to execute upon the receipt of an interrupt. Since there are many different devices with many different ways to communicate, there are many interrupt vectors built into a system, with many different interrupts to go with them. As with devices, interrupts have priorities to organize them; during the handling of one interrupt, the ISR may ignore lower-priority interrupts to prevent them from running during the handling of an interrupt.

Operating systems embrace this interrupt system. Operating systems are interrupt-driven. They typically do very little on their own, but instead wait for interrupts to drive them to do their varied tasks. Operating systems have many services that can be used and many ways to use these services, but only offer them in response to requests. So operating systems have their own system of 'interrupt vectors' and these 'vectors' are implemented using system calls into software implementations. Upon receipt of an interrupt, an operating system stops what it was doing, saving the spot for its return, and finds a software implementation to service that interrupt. When the interrupt service routine has completed, the operating system returns to where it left off.

Interrupts make a great notification system, but sometimes notifications need to be turned off or ignored. This is facilitated in operating systems by *masking*. This terminology comes from the idea of using a bitstring to represent all possible interrupts. By constructing a second bitstring with 1s representing the interrupts to be enabled, this second bitstring can be ANDed with the bitstring of interrupts to produce only those interrupts which are enabled and functioning. This operation of masking is used to turn interrupts on and off. (In other situations, where a mask of bits is not used, the operation is still called masking.) Turning interrupts off allows the operating system to handle higher-priority interrupts without being distracted by other – probably lower-priority – interrupts.

This model of interrupt-driven operation is so useful that software interrupts have been worked into operating systems just like hardware interrupts. Software interrupts take several forms. There are interrupts that are triggered when errors occur (for example, reading beyond the end of a file), interrupts that cause the operating system to do certain things (for example, when a system timer goes off), and interrupts that have no established service routines (these are usually set up and driven by specific software applications). Interrupts can be sent explicitly (for example, Unix allows 'signals' to be sent to the operating system through special system calls) or they can be generated transparently by making function calls (many Symbian OS system calls generate software interrupts).

Since operating systems are passive software systems, there must be a way to get them started listening for and servicing interrupts. This is typically done by a *bootstrap program* in a fixed storage location. The computer's hardware is designed to find this program and start its execution. The bootstrap program is usually quite small and is designed to locate and start a much larger program. This second program is the operating system implementation. The bootstrap program is usually stored in read-only memory (ROM) supplied with the computer system. The second program, or the kernel, is the system that sets up the computing environment and waits for interrupts to occur.

Processes

The programs that run on a computer also work with the interrupt system. In modern operating systems, several programs execute at once, sharing the computing resources. These concurrent programs are called *processes* once they begin running on the CPU. Obviously, if a single process ran to completion before another began to operate, a computer would run extremely slowly. Instead, processes run at the same time and rely on interrupts to stop their execution, returning control to the operating system. The *scheduler* is the part of the operating system responsible for deciding which process should next execute on the CPU.

An operating system that allows multiple processes to run in this manner is said to support *multitasking*. Multitasking shares the CPU according to policies developed by the operating system designers and perhaps the operating system users. One such policy is the time period for which a program uses the CPU, called a *time slice*. Note that it almost certainly takes more than a single time slice for a program to execute to completion, since the period of time is in the order of milliseconds. This means that there are several programs, each using the processor for a time slice and each suspended while the operating system allows other programs to share the processor. This procedure of moving processes into and out of execution on the CPU is called a *context switch*; there is much housekeeping to be done during each switch that provides each program a context in which to run.

Operating systems may also support *multithreading*. Multithreading is different from multitasking in that multiple threads of control execute within the memory space of a single process. Multitasking refers to switching between processes; multithreading occurs within a specific process. Threads provide for code execution in a lighter form than with processes. For example, context-switching between threads uses a similar concept to that of switching between processes but is quicker since context information about the memory space does not need to change.

Device I/O

A computer system without devices is not very useful. How an operating system implements communication with a device is important for many reasons – including the performance of the computer and the ease with which it is programmed. From the previous section, we already know that device I/O is driven by interrupts, but exactly how those interrupts are generated and serviced determine how efficient the operating system is. The general sequence of servicing I/O requests is depicted in Figure 2.5.

The request made by an application is fielded by the operating system through one of its APIs. The operating system uses the device driver specific to the device being accessed and passes the request on to the hardware device (note that operating system interrupts are not needed to pass this data on). The hardware receives the request and services it, passing the results back up through the system. The device interrupts the operating system through the device driver and the operating system delivers the results to the application.

Notice that the scenario depicted in Figure 2.5 requires a lot of waiting. While the operating system is working with the application's request, the application is waiting for it to be completed. This is not unusual; application programs typically wait for devices. However, if the operating system were to wait for the results from the device, no other operating system duties would be performed. All other activities in the computer would therefore wait as well.

Consider an example in which an application tries to send a text message. After setting up the message data, the application initiates the

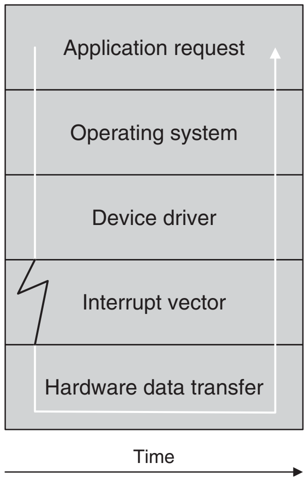


Figure 2.5 The control pathway for synchronous device I/O

transfer by signaling the mobile phone device to transfer the message. This request goes through an operating system API, which communicates through this level to the device driver and on to the hardware to send the message. It might be acceptable for the application to wait until the message is sent. However, if the operating system was forced to wait for the message, it would have to suspend all other services. That would mean that alarms would not be displayed and incoming phone calls would be ignored. If the message took a lengthy period of time to send, the phone would just freeze up until the message was finally on its way. Obviously, this is not a good situation.

The method of device communication that waits through the communication cycle is called *synchronous* communication. Synchronous communication causes all stages in the process to wait. This type of communication is good for real-time systems, where the system is dedicated to I/O and processing of received data, but not very useful for general-purpose systems.

Most general-purpose I/O is *asynchronous*. That is, other operations can continue while waiting for I/O to complete. An I/O sequence like that in Figure 2.6 must occur.

The hardware should signal that the transfer has begun and signal again when the results of the I/O request are in. Using this method, the operating system is free to process other requests and the application

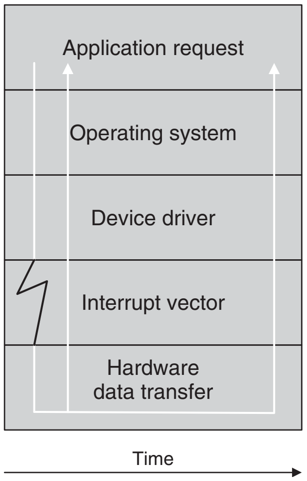


Figure 2.6 The control pathway for asynchronous device I/O

can even go on to do other things. (Often this method of I/O is best for applications that must work with a graphical user interface, which must usually be updated as the data request is being processed.)

The use of asynchronous device I/O means that an operating system must keep track of the state of devices. If the operating system is going to 'get back' to handling a device after it has serviced an I/O request, it has to keep track of what was happening with that device and where it was when it last worked with it. This record-keeping function of an operating system is a very important one, one that keeps an operating system busy much of the time and one that potentially takes up a lot of the memory needed to run an operating system.

In the quest to minimize the involvement of the operating system in device I/O, more I/O functionality can be placed on the device with the addition of more interrupts to enable communication. Taken to an extreme, a device could do all I/O by itself, filling a specific area in shared memory with data and signaling the operating system only when data transfer is complete. This method of I/O is called *direct memory access* (DMA) and is extremely useful in freeing up operating system and application time. DMA is a form of asynchronous I/O, but differs from the generic form. Asynchronous I/O is fine-grained: it signals the CPU whenever there is even a small amount of data to transfer. DMA is very coarse-grained and assigns all data operations to the device. The operating system starts the I/O operation and is only notified when it is complete.

There are, then, three modes of device communication: synchronous, asynchronous and DMA.

- A handheld Linux device that plays video is likely to use synchronous communication between the video driver and the operating system. Display of video is a real-time application and most real-time applications require synchronous I/O.

- Computers with windowing systems use asynchronous I/O to monitor GUI devices such as a mouse. When a mouse moves, it generates interrupts that cause the operating system to read the mouse events. When the mouse does not move, the operating system can safely ignore it and move on to other duties.

- Computers use DMA for larger I/O tasks. Consider reading from a disk drive. It is enough that an operating system would send a disk drive a command to read a block of data, along with the parameters needed to complete the transfer. Reading program code from a disk to execute, for example, is usually a task that is executed using DMA.

Each I/O method carries with it implications for system performance. With synchronous I/O, the operating system spends all its time monitoring and servicing devices. This means that performance and response to users and other services is slower than with other methods. Asynchronous I/O relieves the operating system from constant monitoring and, therefore, performance and system response increases. DMA frees the operating system from almost all device I/O responsibilities and therefore produces the fastest system service and response time. Most operating systems use a combination of methods to gain an efficient design.

Storage Structures

Along with central computer operation and device I/O, storage makes a third essential component of a computer system. The ability to record information and refer to it again is foundational to the way modern computer systems work. A system without storage would not even be able to run a program, since modern systems are based on stored programs.[2] Even if it was able to run instructions (perhaps asking the user for each instruction), input could not be stored and output could only be generated one byte at a time.

The core computing cycle is very dependent on storage. This core computing cycle, often referred to as the 'fetch–execute' cycle, fetches an instruction from memory (storage), puts the instruction in a register (more storage), executes that instruction by possibly fetching more information (more storage), and storing the results of the execution in memory (even more storage). This basic computing cycle is part of a design first developed by John von Neumann, who built it into a larger computing system based on sequential computer memory and external storage devices.

The many storage mechanisms of a computer system can be viewed as a hierarchy, as shown in Figure 2.7. Viewing these systems together allows us to consider their relationships with one another.

[2] Certainly, computers without disk storage are used every day. But note that even these computers have memory for storage – sometimes large amounts of it. Any computer system has storage at least in the form of memory or registers accessible by the CPU. Most systems build their storage requirements from there.

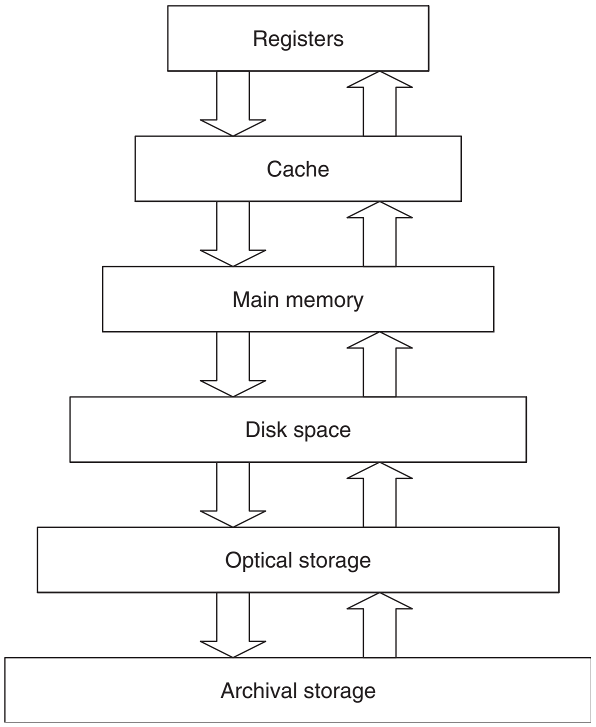


Figure 2.7 Storage hierarchy

- *Registers* are at the top of the hierarchy. This collection represents the fastest memory available to a computer system and the most expensive. Depending on how a processor is constructed, there may be a small or large set of these memory cells. They are typically used by the hardware only, although an operating system must have access to (and therefore knowledge of) a certain set of them. Registers are volatile and therefore represent temporary storage.

- Storage *caches* represent a buffer of sorts between fast register storage and slower main memory. As such, caches are faster and more expensive than main memory, but slower and cheaper than register memory. On a typical computer system, the caching subsystem is usually broken into sublevels, labeled 'L1', 'L2' and so forth. The hierarchy continues to apply to these sublevels; for example, L1 caches are faster and more expensive than L2 caches. Caches represent a method to free up the hardware from waiting for reads or writes to main memory. If an item being read exists in cache, then the cached version is used. If data needs to be written, then the cache controller takes care of the writing and frees up the CPU for more program execution. Caches are volatile and therefore also represent temporary storage.

- *Main memory* represents the general-purpose temporary storage structure for a computer system. Program code is stored there while the program is executing on the CPU. Data is stored in the main memory temporarily while a program is executing. The I/O structures, discussed in the previous section, use main memory as temporary storage for data. This type of memory is usually external to the CPU and is sometimes physically accessible by the user (for example, on desktop systems, users can add to main memory or replace it).
- *Secondary storage* is a slower extension of main memory that holds large quantities of data permanently. Secondary storage is used to store both programs and data. The first – and still most common – form of secondary storage is magnetic disks. These store bits as small chunks of a magnetic medium, using the polarity of magnetic fields to indicate a 1 or a 0. Faster storage has evolved more recently in the form of electronic disks, large collections of memory cells that act as a disk. Formats such as compact-flash cards, secure-digital cards and mini-SD cards all provide permanent storage that can be accessed in a random fashion. These are used in the same way as magnetic media to manipulate file systems.
- *Tertiary, or archival, storage* is meant to be written once for archival purposes and stored for a long period of time. It is not intended to be accessed often, if ever. Therefore, it can have slow access times and slow data-retrieval rates. Examples here are magnetic tape and optical storage such as compact discs (CD-ROMs). CD-ROMs can be thought of as lying between secondary and tertiary storage, because access time on CDs is quite good.

There are several concepts built into this storage hierarchy that affect how an operating system treats each storage medium. The first, and most basic, is the model used to access storage. The idea of a file as a group of data having a specific purpose has been the model of access used since almost the invention of permanent storage. If many files can be stored on a medium, there is also the need for organization of those files. Ideas such as directories and folders have been developed for this organization. The way that these concepts have been implemented is called a *file system*. The design and appearance of file systems differs across operating systems while the concepts of files and the structure of directories remain constant.

The concept of *access rights* has proven useful in implementing secure storage. In some systems, access to storage is granted to any process requesting that access. In other systems, processes requesting access to storage must present identification along with the request and are only granted the access that the identification gives them. In these types of systems, there is typically an owner of a unit of storage and the owner sets up how others may access that unit. Note that this requires that the system using these access rights establish a method of user- or process-identification. For example, Symbian OS establishes identification based on a process's function within the operating system. There are system processes and non-system (user) processes; in addition, there are other processes that have more privileges than users but not the complete privileges of the system. Access to storage on Symbian OS is granted based on these classifications of processes.

Another concept that has evolved from the hierarchy of storage is *caching*. As shown in Figure 2.7, speed of storage access decreases as you work down the hierarchy. Caches were developed as a way to shield devices from slower storage. Cache management has become an important issue. For example, if a cache is full and the CPU needs to write more data to it, some data already in the cache is overwritten. If the cache is managed carefully, the 'relevant' data is kept in the cache and the rest is written to the next level. However, the meaning of 'careful management' is different depending on the design of the operating system.

The idea of *virtual storage* is a concept that works across the storage hierarchy. Storage is virtual when it is larger or has more attributes than it physically has. Virtual storage is implemented as an extension of one layer in the hierarchy on lower layers. Main memory can be thought of as virtual cache storage. When cache fills up, it extends into main memory. Likewise, virtual main memory is implemented on disk space. When space in main memory runs out, it overflows onto secondary storage. As with caching, virtual storage involves management: it must be organized so that portions of it can be moved back and forth to the next storage layer.

Hardware Access and Protection

In the early days of computing, before operating systems were used on computer systems, a single program ran to completion on a computer, using its resources as it saw fit. As computer usage evolved, operating systems were used to provide a consistent and standard interface to computing resources. This meant that access to a single resource – the system clock or the graphics display – had to be coordinated by the operating system and shared with other applications vying for those resources.

As all children learn, sharing is good. Sharing resources means that they can be used more efficiently and more completely. If resources are shared, all applications can appear to execute at the same time and are presented with the illusion that they are the only application running on the computer system. Consider, for example, sharing a network connection between two browsers, as shown in Figure 2.8.

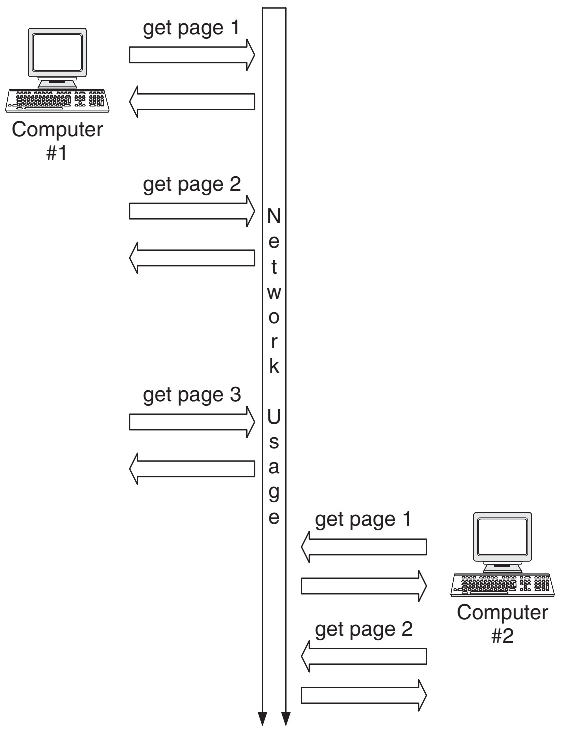


Figure 2.8 Sharing a network serially

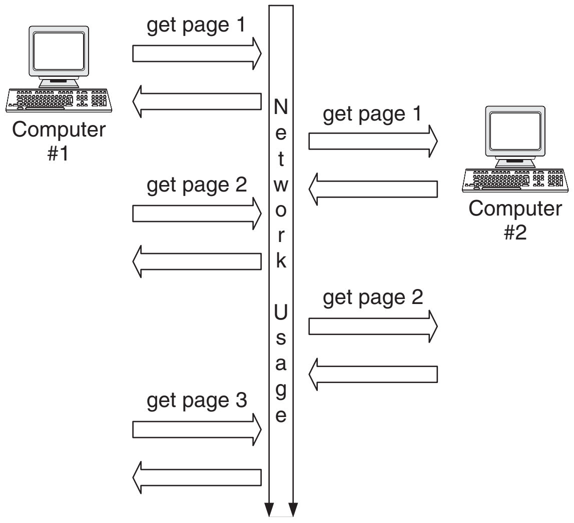


Figure 2.9 Sharing a network concurrently

In Figure 2.8, Computer 1 is served three pages from the web and Computer 2, which needs two pages, is forced to wait until Computer 1 is done using the network. Time is depicted from top to bottom. This kind of 'one at a time' serial usage is certainly the safest way to share a resource, but not the fastest. Consider the sharing scenario in Figure 2.9, where Computer 2 can use the network while Computer 1 is waiting between page fetches. The time is shorter but the information gathered from the network is the same. As long as care is taken to make sure that one computer's actions does not change the other's, then sharing can be done in a safe manner.

However, sharing can also be bad. If done poorly, the mechanism used for sharing can ruin the illusion – each concurrent program would feel the effects of other programs. If it is not done correctly, data might be manipulated by the wrong program. A browser might receive data that another browser had requested.

As we consider how to protect system resources from concurrent access, we also need to remember that the operating system is a competitor in this area. We have said that the operating system is simply another program running on the computer. In this sense, the operating system competes for resources just like other programs that run. However, the operating system is a bit different – a bit more privileged – than 'normal' programs because it must manage the other programs (as well as itself).

Let's consider how protection is addressed in operating system design. To do this, we must consider how to protect programs from each other and how to protect resources such as memory and the CPU. This not only applies to organized access to resources, but it also keeps errant or malicious code from accessing resources in ways that could jeopardize the operations of other programs.

Protection modes

We must protect programs from each other; this includes protecting the operating system from other programs. We need at least two separate ways of operating: the operating system needs a *privileged mode* and other programs need a *user mode* of operation. User-mode operation is restricted to tasks that all programs may perform. This includes mundane tasks such as arithmetical computation or executing statements in program code. Privileged-mode operation allows a program to do tasks only the operating system should do. These tasks include working with system devices or managing which program should be run.

Managing these modes efficiently and rapidly requires hardware support. Most hardware architectures include at least the two modes we have defined and they sometimes support multiple user modes. The hardware may implement this by adding a bit, or a set of bits, to indicate the current mode of operation. By setting the mode bit, the mode of the instruction being executed can be determined easily. In addition to mode bits, there are certain instructions that are considered privileged instructions. It is assumed that only processes with privileged access execute privileged instructions and the hardware enforces this by checking the mode bits before executing these instructions.

Consider some situations where modes are important. When a computer starts up at system-boot time, the hardware starts out in privileged mode. This makes sense because the operating system is initializing itself and the system resources. Once the operating system is running and starts applications, each application executes in user mode. When an interrupt occurs and the operating system must service a need from a device or resource, the operating system is running and the hardware is placed in privileged mode. Since the interrupts drive when the operating system takes over the management of the computer, the operating system is always in privileged mode when it controls the processor. Whenever control is given to another program, the mode is switched to user mode.

As we have seen, there are many times when a user-mode program needs to access a system resource that only the operating system can manage. Since a user-mode program cannot change to privileged mode by itself, it must ask the operating system to perform the privileged-mode operation. User-mode programs do this by making a *system call* into operating system code. Control is passed to the operating system and the operating system handles the privileged-mode action as it sees fit. When it is done with the operation, the operating system passes control back to the user-mode program, changing the protection mode in the process. This method of using system calls means that all privileged-mode operations are still handled by the operating system.

What if There Is no Hardware Support for Protection Modes?

Early architectures such as the Intel 8088 architecture (on which MS-DOS was implemented) did not have mode bits built into the hardware, which meant that there were no privileged instructions. No privileged instructions meant that any process could manipulate any system resource. In early versions of MS-DOS, for example, user programs could manipulate operating system tables and change operating system code!

Protecting memory

Programs must be protected from each other, as we saw in the previous section. This includes restricting memory-space usage to only those programs that should use it. Since there are many ways to use memory, this kind of protection must guard against several types of usage.

We must protect user programs from changing the memory of other programs. This means that, while multiple users have data in memory at the same time, users' memory spaces must be protected from each other. In addition, programs might be able to corrupt operating system memory or even change interrupt vectors. User code needs to be hemmed in – cordoned off from the rest of the memory.

While we can build memory protection into APIs or into operating system code, this is not enough because there are multiple ways to corrupt memory. To properly protect memory, we need to determine the range of addresses that a process or interrupt vector can use, protect memory *outside* the address range and ensure that nothing can run outside the control of the operating system. Instead of setting up protection to keep other usages out, we set up protection to keep each process usage in.

This is typically done in conjunction with hardware. When a particular piece of code is executing, the operating system sets two registers: a *base register* holds the lowest address that can be used by the executing code and a *limit register* holds the number of memory addresses that can be addressed. Setting these registers is reserved only for the operating system; it does this through the use of privileged instructions. Working with these registers is part of the work of the operating system as it manipulates processes to share the CPU.

Protecting the CPU

The many programs that run at the same time on a computer share the CPU; this sharing includes the operating system. As we structure the way that this sharing is done, we must make sure that the operating system always gets control of the CPU back from a program – even if that program has bugs or goes into an infinite loop.

Consider what happens if we do not protect the CPU in this way. A program that gets into an infinite loop may never relinquish control of execution and the computer would be frozen. If this happened on a mobile phone while a conversation was going on, the phone would simply freeze and the conversation could not continue. Even worse, consider if *system code* had a bug that caused a user program in privileged mode to start writing to protected memory, corrupting operating system tables.

To prevent this from happening, operating systems often use two concepts. First, we can use a *timer* to cause an interrupt that transfers control back to the operating system. That timer is set when the program begins using the CPU and interrupts the program's execution after a specific period has elapsed. While timers can be fixed, they are usually capable of using a variable time period. If an operating system uses a variable program timer then certain programs can run for longer than others. Note that timer interrupts can happen at any time and that sometimes they happen at inconvenient times. For example, these timer interrupts can be made during system calls, interrupting the kernel during a system call service.

A second way of protecting the operating system from freezing up is to combine the use of variable timers with a control concept of processor sharing. This combination enables operating systems to implement concepts of process management, including time slices and context switches.

There are many housekeeping details that must be done with timers. For example, consider the manipulation of the program timer. At each context switch, this timer must be reset for the next execution. If the timer is a variable timer, then a new value must be derived for the next program's time slice. This is quite probably in a table and should be looked up, but it provides a good example of the complex duties of the operating system. In addition, processes can voluntarily give up their time on the processor by *yielding* the processor to other processes.

Keeping the Current Time

Using timers to protect the CPU also provides operating systems with a mechanism to keep the current time. If timer interrupts occur at regular intervals, we can compute the time of day based on the last accurate time.

However, this method is actually quite inaccurate. We must depend on the fact that there are no interrupts during a timer interrupt and that servicing the timer interrupt itself is instantaneous. Since these assumptions are usually not correct, the system time starts to drift if we implement it this way. Operating systems typically use hardware time-of-day clocks to implement system time.

Communication Structures

A fourth essential component of modern computers is communication. The structures that have been developed to handle communication issues parallel very closely those that address I/O. Communication is a special case of I/O and special attention – resulting in specialized APIs and operating system structures – has been given to this area by operating system designers. In some cases, for example Symbian OS, an operating system has been developed around communication issues.

At the lowest level, communication is raw binary data moving through physical I/O devices connected to a computer. As discussed above, operating systems take a variety of approaches to implementing device I/O and presenting the I/O interface to the other APIs. In the various implementations of communication structures, operating systems treat the physical communication devices as they would other I/O devices.

On top of the physical I/O device, operating systems implement an interface between software and hardware through the use of device drivers.

On top of the hardware–software interface, operating systems place a layer that allows users to use the hardware through the privileged instructions of the kernel. As we have seen earlier, there are many ways to manipulate communication hardware, from file-like interfaces in Linux to file-server applications in Symbian OS. The implementation of this access layer incorporates the design model of the operating system.

Most communication requires protocols to be run through specific device interfaces. A protocol is an exchange of data that follows a specific prearranged format. There are many different ways of communicating through computer devices and these different ways follow different protocols. For example, one could pass a file between computers using a TCP/IP local area network or using Bluetooth technology. While the end result is the same – a file gets from one computer to another – the protocols that are used to exchange that file's data are very different between the two media.

With the advent of the Internet and the overwhelming use of TCP/IP, operating systems typically abstract away the details of communication protocols by implementing communication through an abstraction called a *socket*. A socket is a connection with two endpoints – two sides of the communication channel – with an implementation of a communication protocol in between. The abstraction of a socket works well because the protocol implementation is hidden and the methods of data exchange are kept the same, regardless of the protocol being implemented. So a program can use the `write()` system call to send data over a socket and not be concerned about whether the socket is connected over TCP/IP or Bluetooth.

The idea of abstraction can play out further in a fashion similar to the model of memory. In this extended abstraction, each protocol is built on the underlying services of the layer below. This 'stack' of communication protocols is nicely implemented as a stack of implementations, where each implementation represents a certain functionality and each one passes its data to layers above or below. We discuss these issues further in Chapter 10.

2.3 Different Platforms

We have given an overview of the concepts and structures that characterize modern operating systems. This section gives some examples of how these concepts and structures are used in different implementations.

OS/360 and MVS

The IBM OS/360 line of mainframe computers was developed ahead of the operating systems that were to run on them. Thus, the hardware existed while software developers were scrambling to get an operating system that made that hardware useful to run. As operating systems developed for the 360 line, operating concepts were also evolving. The operating system grew over time as concepts of multiprogramming and multitasking were developed.

- The first version of OS/360 was the simplest: a sequential scheduler called the primary control program (PCP). PCP performed only one task at a time. Control returned to the operating system only when the task was completed. I/O was processed synchronously and caused programs to stop while they waited for I/O to complete.
- The next version of OS/360 introduced multiprogramming with a fixed number of tasks (MFT). MFT could (eventually) run up to 15 tasks at once and could reschedule tasks while they waited for (synchronous) I/O to complete.
- The last version of OS/360 allowed a variable number of tasks to be run concurrently – theoretically, any number could be concurrent. Multitasking with a variable number of tasks (MVT) also supported rescheduling of synchronous I/O-bound tasks.
- A new version of the operating system, the single virtual storage (SVS) version, was developed. SVS implemented multitasking, but forced all processes to occupy the same memory space. Context-switching was expensive but memory protection was simple. The only memory that could be violated was the operating system memory, as all programs shared that space.
- The most popular version of the operating system was called multiple virtual storage (MVS). MVS put each process in its own address space in memory, allowing memory to grow as needed by adding virtual memory on the disk. Memory protection was now more complicated, because process memory moved in and out of physical memory and multiple applications could be resident in memory at the same time.

MVS saw implementations in OS/370 and OS/390 for various incarnations of IBM mainframes. Notice how the concepts of multiprogramming and multitasking evolved, carrying with them ideas of virtual memory and device I/O. Memory-protection issues also evolved: protection was a lot easier in PCP, where only one process ran at a time, than in MVS, where multiple processes were concurrent.

Unix and Linux

As we discussed in Chapter 1, Unix evolved from MULTICS and used many of its ideas. The character of Unix has a very loosely connected feel: all of its components build on each other through the use of fixed APIs and its approach to software design is to build the operating system by interconnecting simpler tools. In Unix, simpler is better.

There are many examples of this design philosophy. Commands on Unix are quite simple and can be combined to build more complex commands. Files on Unix are not structures in any way but are considered only as a sequence of bytes, to be interpreted by individual applications.

- The kernel is a monolithic kernel. Any changes to kernel operations – such as the change in serial-port implementation – requires a change in kernel source code and a recompilation and reinstallation of the entire kernel.
- Unix is multitasking and supports multithreading. It supports configurable policies regarding scheduling of processes. Unix is a multi-user system, where multiple users can be accessing the same computer and sharing the resources.
- Devices are implemented as files and access to devices is achieved by opening the file representation of a device. Devices can be 'opened' by multiple users for reading and by only one user for writing.
- Unix uses virtual memory and uses memory mapping to avoid user applications from accessing memory from other applications. Memory mapping automatically translates any memory reference into the area reserved for the process.
- Unix supports many kinds of file systems and communication methods through implementations of dynamically loaded implementation modules and device drivers.

Linux is an open-source version of Unix. The fact that Linux is open source has been both a blessing and a curse: allowing the source code to the operating system to be shared has fostered much innovation but has also allowed people to exploit weaknesses.

Symbian OS

Symbian OS is unique among operating systems in the sense that it was designed from its inception[3] with smartphones as the target platform. It is not a generic operating system shoehorned into a smartphone nor is it an adaptation of a larger operating system for a smaller platform. As we saw in Chapter 1, Symbian OS has a history of evolving design (from SIBO to EPOC to Symbian OS) specifically targeted at smartphones for its implementation.

The precursors to Symbian OS have given their best features. The operating system is *object-oriented,* inherited from EPOC. This means that systems calls involve system, or kernel-side, objects and that the idea of abstraction permeates system design. Where an operating system such as Unix might create a file descriptor and use that descriptor as a parameter in an `open` call, Symbian OS would create an object of the `RFile` class and call the `open()` method tied to the object. In Unix, it is widely known that file descriptors are integers that index a table in the operating system's memory. In Symbian OS, one really has no idea how the file object is implemented; one simply creates an `RFile` object and uses its methods.

Symbian OS has other inherited features. It is a multitasking and multithreaded operating system. Many processes can run concurrently, they can communicate with each other and utilize multiple threads that run internal to each process. The operating system has a file system compatible with Microsoft Windows (technically, a FAT32 file system); it supports other file-system implementations through a plug-in interface. It uses TCP/IP networking as well as several other communication interfaces, such as serial, infrared and Bluetooth.

Symbian OS has some unique features that come from its focus on the smartphone platform. Because of limited (or, in most cases, no) disk storage, no virtual memory is implemented. Symbian OS has a pluggable messaging architecture – one where new message types can be invented and implemented by developing modules that are dynamically loaded by the messaging server.

Consider the way system calls work in Symbian OS. There are two types of system call. An *executive call* makes a request for the kernel to execute

[3] Note that the origins of Symbian OS can be found in EPOC (as stated in Chapter 1) and EPOC was not designed for smartphones. However, when Symbian OS was designed as a replacement for EPOC, it was indeed intended for smartphones and was designed with this target platform in mind.

an operation in privileged mode on behalf of the user-space requestor. An executive call causes a software interrupt, which is serviced by branching the operation into kernel code. The interrupt is serviced and control is passed back to the user. Executive calls can modify kernel-space objects but cannot create or delete them. Operations such as memory allocation or thread creation need to be done by *kernel-server requests*. There is a server that protects kernel resources and requests to manipulate those resources need to go through that server. Server requests are themselves executive calls.

- The kernel structure of Symbian OS has a microkernel design. Minimal system functions and data are in the kernel with many system functions spread out into user-space servers. The servers get their jobs done by making executive calls into the kernel when necessary.
- Symbian OS supports the use of virtual machines: the implementation of a 'computer within a computer'. The implementation of the Java programming language and the run-time environment needed to run Java is done through this mechanism.
- Communication structures in Symbian OS are easily extended. Modules can be written to implement anything from user-level interfaces to new protocol implementations to new device drivers. Because of the microkernel design, these new modules can be introduced and loaded into the operation of the system dynamically.
- Symbian OS has been designed at its core with APIs specialized for multimedia. Multimedia devices and content are handled by special servers and by a framework that lets the user implement modules that describe new and existing content and what to do with it.

2.4 Summary

This chapter has been about the concepts and structures that make up the character of an operating system. Operating systems evolve over time as the hardware they run on and the needs of users evolve.

We discussed several concepts that are implemented in operating systems. We looked at system structures, including kernels, the interrupt system and how applications become processes running on a CPU. We looked at the different kinds of device I/O and how interrupts are used to implement them. We had an overview of storage structures, including the storage hierarchy and the ideas involved in caching and file systems. We looked at system protection strategies, from protection modes to ways of protecting memory and CPU usage. We reviewed communication structures, implemented by sockets.

The chapter concluded by taking examples of operating system character: we looked at IBM OS/360, Unix and Symbian OS.

The next chapter begins our closer look at these operating system components by looking at processes and scheduling.

Exercises

1. Consider the following services and classify them as taking place in the kernel or outside the kernel. Do this for both microkernel and hybrid kernel systems.
   a. Opening and closing files
   b. Writing to a register
   c. Reading a memory cell
   d. Receiving a text message
   e. Playing a sound bite.
2. Software interrupts are useful for many things. We discussed timers as an example of software interrupts. Think of other examples in a computer system of software interrupts. (Hint: Think of software interrupts as events.)
3. With software interrupts, what form does the interrupt vector take? Where is it stored?
4. Context-switching is expensive because of the 'context' that is switched. Try to identify as many parts of this context as you can.
5. Often people think of 'protection' as 'security'. We discussed ways to protect user programs and the operating system from each other. In what ways could the protection mechanisms we discussed be a form of security?
6. We gave a few examples of device I/O types. Can you develop more examples for each I/O type? Explain your answers.

7. Which of the following operations should be done in the kernel as privileged mode?
    a. Reading and writing files
    b. Sending a text message
    c. Taking a picture with the camera
    d. Notifying a program about data received
    e. Switching processes on the CPU
    f. Reading from a memory cell.
8. What would a system look like without a privileged mode? Could such an operating system be useful? How would it be possible to implement protection?
9. Systems typically have multiple caches built into the hardware; we called them L1 and L2. Why is multilevel caching useful?
10. Survey the concepts and structures we discussed in this chapter. In what ways would hardware be useful to help implement a specific structure? For example, how could hardware help with context-switching or memory protection?
11. How would you classify a kernel that implemented the various forms of the OS/360 operating systems?
12. As we discussed, Symbian OS uses two levels of kernel space operations: the executive call and the kernel-server request. Why would this be necessary? Why should executive calls be barred from creating objects in the kernel? (Hint: think about the usefulness of the servers in a microkernel.)
13. Symbian OS has an object-oriented design. Survey our discussion of Symbian OS and flag places where object orientation would complement the design strategies.
14. Symbian OS is designed for use on smartphones. Consider a smartphone platform and identify the forms of communication that it would use. For each of these forms of communication, identify the type of device I/O that could be used to implement that form.
15. Consider your answer to the previous question. Must Symbian OS be a real-time operating system? Are there portions of system operation that do not have to operate in real time? How does the microkernel design of Symbian OS help reduce the need for real-time operation?

3
Kernel Structure

A computer network is functioning at its best when no one notices it. When it is working properly, computers communicate with each other with ease and users pay no attention to how their web browser works or that email must travel long distances to get to their computers. In reality, there are many subsystems that co-operate to make a working network function properly. When a web browser requests a web page and it simply appears, it is easy to ignore the complicated layered structure that underlies the ease of a network's function.

The same can be said of the kernel structure of an operating system. When it works well, it is easy to ignore that it is even there. However, the structure of an operating system's kernel is at center of its character and essential to its proper function. It is useful to examine what a kernel is comprised of and how a kernel works. This chapter examines a kernel in several ways. We discuss how a kernel is put together, that is, what parts make up a kernel. We then discuss how system calls interact with kernel code and what paths a system call might make through the kernel. We follow with a similar discussion about interrupts. We wrap up the chapter by taking a hard look at an example: the Symbian OS kernel.

3.1 How a Kernel Is Put Together

The design of a kernel is very important to the performance of the computer it runs on. We discussed kernel design – especially monolithic kernels and microkernels – and we noted how certain kernel designs work better on specific types of computer platform. For example, we saw how kernels with a microkernel design work better on smartphone devices. Symbian OS is an operating system that has a microkernel architecture. It is great example of a kernel that has a layered structure. We provide an overview of that structure in this section.

It is interesting to look at kernel design as a set of pieces. Only some of those pieces are actually running at any given time. From this perspective, there are two types of component that a kernel is built from: *active* components and *passive* components. We examine them in this section.

Kernel Structure

A kernel is built in layers. The layers of a kernel structure reflect the functionality of that part of the kernel. Inner layers implement basic, primitive functions in such a way that these basics execute very quickly. Innermost layers are also the most privileged layers, able to access all components of the operating system whenever they need to. As you look from inner to outer layers, the functions of the layers get less primitive and privileges are taken away; you move out toward user-mode applications requiring fewer kernel-mode privileges and functionality. Figure 3.1 shows the general Symbian OS kernel structure.

- The *nanokernel* provides some of the most basic functions in Symbian OS. Simple threads operating in privileged mode implement services

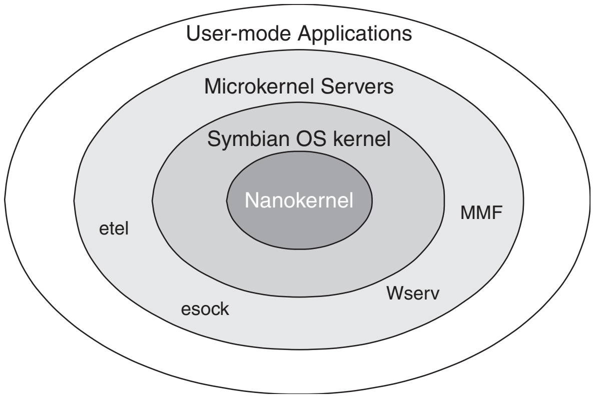


Figure 3.1 Layers in the Symbian OS kernel

that are very primitive. Included among the implementations at this level are scheduling and synchronization operations, interrupt handling and synchronization objects called *mutexes* and *semaphores* (we discuss these later). Most of the functions implemented at this level can be pre-empted. Functions at this level are so primitive (so that they are fast) that the nanokernel must not implement any kind of complicated operation, such as dynamic memory allocation.

- The *Symbian OS kernel layer* provides kernel functions needed by the rest of the operating system. Each operation at this level is a privileged operation and combines the primitive operations of the nanokernel to implement more complex tasks. Complex object services, user-mode threads, process scheduling and context switching, dynamic memory, dynamically loaded libraries, complex synchronization objects and interprocess communication are just some of the operations implemented by this layer. This layer is fully pre-emptible and interrupts can cause this layer to reschedule any part of its execution – even in the middle of context-switching!
- The *server layer* is typical of microkernel architectures. Operations that do not require complete privileged operations or that have a complex implementation are pushed out to servers. Server-based functions typically govern specific areas of functionality, such as handling the display or working with sockets, and usually run as user-mode services. These areas of functionality require kernel-based operations only sporadically and therefore can sit outside the Symbian OS kernel layer.
- The *user-mode applications* run almost completely in user mode and perform kernel-based operations either by interacting with servers or by making system calls that activate kernel-mode activity.

It is instructive to look at the kernel structure from another perspective. It is easy to think of the kernel as an always-on, executing set of code that runs alongside application programs. This is not the case. Only part of the kernel runs constantly; much of the kernel is implemented passively, set into operation by system calls and interrupt handlers.

Active Kernel Components

Active kernel components are those parts of the kernel that execute along with other processes in the operating system. These kernel processes typically have higher priorities and high levels of protection. They are usually multithreaded to allow for multiple threads of access from threads of execution in various processes.

Active kernel components are active so that they can monitor the system in real time. They field requests for kernel services, service those requests, load and unload system modules (the passive kernel components), and perform all the bookkeeping that needs to be done. Active components assist with the working of passive components as they field requests and implement the requests in kernel mode. Consider some examples.

- Two processes want to communicate. One of their implementation choices is to pass data through global memory from one process to another. This global memory is maintained by the kernel and access is gained through kernel requests. Each process makes a system call that sends a request to the kernel process. As we see in Chapter 6, this request requires a mechanism called a *semaphore* that coordinates how each process accesses the global data. All of this access, from semaphores to global memory reading and writing, must be maintained by active kernel components.
- An application begins execution by building a context, then switching into and out of contexts as the CPU is multiplexed between processes. Context-switching is managed by active kernel components in response to a voluntary relinquishing of the CPU, some kind of I/O blocking or a timer event. Again, bookkeeping must be done and memory must be managed to make sure processes switch contexts properly. In between process switches, other system duties must take place and the active components implement these as well.
- An application wishes to manipulate an I/O device, for example, sending data to an IR port. Active kernel components are built with varying degrees of peripheral support. Monolithic kernel structures typically have a built-in set of device implementations. Microkernel architectures, by contrast, are typically independent of peripheral implementations. In all cases, it is highly likely that code is loaded dynamically to implement various portions of device I/O. The active components of the kernel are involved in module loading and unloading and do these tasks as required by service requests. In Symbian OS, for example, sending data through an IR port results in the loading of several I/O drivers in sequence, in response to several kernel requests.

Let's consider this last example a little more closely. In a freshly booted system, the active components of a microkernel-based operating system have a very small memory footprint. For example, the executing kernel in Symbian OS v8 uses about 200 KB of memory when it starts up. A request to use the IR port on a device causes a series of events, orchestrated and implemented by the executing kernel code. The executing kernel needs to add layers of code to implement the data exchange over IR. None of these layers are initially in memory and all of these layers represent an initial performance hit as the kernel code initializes device drivers and starts communication.

The size of the executing kernel components have a direct impact on system performance through the use of memory and the time it takes to load additional modules that implement various features. A monolithic kernel structure minimizes the loading of implementation layers. Much of the functionality of device I/O, for example, is built into a monolithic kernel. Response time is initially quicker because modules do not need to be loaded. However, because many modules are loaded that are perhaps unneeded, memory requirements go up dramatically. Unix operating systems typically boot an active kernel that requires 10 MB to 60 MB at boot time. Microsoft Windows has a smaller requirement, but it is typically at least 8 MB of memory.

Monolithic structures come preloaded with many of the implementations required to run an operating system. The structure is very static, because support for various hardware is built in. On the other side of the size spectrum, microkernel structures take up much less memory upon boot and their structure is more dynamic. Microkernels usually support a 'pluggable' architecture with support for hardware that can be loaded as needed and 'plugged into' the kernel. Thus, microkernels are more flexible – code to support new hardware can be loaded and plugged in any time – but monolithic structures are faster – they avoid the overhead of the pluggable interface).

One way to enhance the performance of all types of active kernel components is to give multithreaded implementations. Remember that the active part of the kernel is an executing process like the other processes in the system. Therefore, it can have multiple threads of execution running inside a single context. The benefit of multithreading to kernel implementation is that each thread can execute a request for kernel service, resulting in multiple requests being serviced at the same time. This is especially helpful when multithreading is implemented for system modules – such as microkernel servers – and for user code. When all of these threads of control are capable of requesting kernel services, the kernel must be multithreaded to support them.

Consider a user-mode application that requests a kernel-mode operation, for example, to load a set of data from a flash memory card. As the kernel is doing this, a phone call comes in and must be serviced. A kernel with a single thread would have to select one of these requests to work on and queue the other request for later service. The system could prioritize the requests by making the phone call (a real-time operation) more important and thereby making the user-mode request wait. However, a multithreaded kernel could service both operations at the same time, handling device interrupts with a thread separate from user-mode requests. While the CPU must still be shared between these two requests, the end result is that service is faster because both operations are in memory at the same time.

Passive Kernel Components

Passive kernel components are those parts of the kernel that are not continually executing but are available for execution on behalf of service requests. These components are present in the form of libraries and dynamically loaded modules that contain code that implements system calls and interrupt service routines. It is through these components that user-level code can get kernel-level tasks done on their behalf.

These elements of the kernel are called *passive* because they do not execute on their own. They are spurred into execution when a system call is made or an interrupt is generated. They contain code that either operates on its own in kernel mode or communicates with the active components of the kernel. There are several examples of this type of kernel component.

- *Device drivers* are loaded dynamically by some kernel implementations when devices are used. In some operating systems – for example, Symbian OS – device drivers themselves are broken down into components that are loaded individually. Symbian OS, for example, uses logical drivers that implement more abstract properties of a device (for example, operations such as read and write, on and off) and physical drivers that handle the specific implementation of the logical operations with specific devices.

- *Microkernel servers* are usually run only when needed and terminate when their services are no longer required. Consider, for example, a smartphone whose user was exercising many applications that use many different servers. As use continued, more servers would be started to service needs. These servers might only be required for a short time – to coordinate the use of Bluetooth, for example – and can afterwards be terminated. This keeps the tables of the kernel cleaner and emptier.
- Passive behavior can sometimes be used to enhance performance. For some microkernel implementations, servers are started at boot time and run without shutting down. Because they only react to requests, they do not poll and are not actively executing at other times. Therefore, there is no cost to the CPU in leaving these servers running (although there is a memory cost because they consume memory resources permanently). Symbian OS implements servers in this manner.
- Situations where the type of information can change dramatically often require dynamic modules. For example, wireless messages for a smartphone are of very different types. Diverse message types are handled by dynamically loaded libraries. In Symbian OS, these are known as *message type modules,* or MTMs. There are special MTMs for email messages and SMS messages. There are many abstractions that are implemented as passive kernel components.

3.2 System Calls and the Kernel

We have seen processes that run in user mode and how processes and libraries can cause execution in kernel mode. The interface between these two modes is provided by *system calls*. These are function calls that cause requests to be made to the kernel and the kernel to execute on behalf of those requests.

System calls constitute an extra layer between applications and inner layers of the kernel structure. This extra layer has several advantages: it provides a layer of abstraction that makes programming easier, freeing users from knowing low-level details about the system and hardware; it increases efficiency and security, because the kernel can coordinate access to hardware and protect processes from each other; it makes programming code more portable, since the code works on any system that supports the same set of interfaces.

System calls are implemented as software interrupts. A system call is typically implemented with a type of hardware instruction that causes a special interrupt handler to be invoked. The implementation either identifies the system call with an index into a table maintained by the operating system or causes a jump to specific handler code. In either case, the implementation causes the system to go into privileged-mode operation and implement a preprogrammed system function. Obviously, there needs to be many of these specially-handled operations; for example, the ARM processor reserves 24 bits to identify which system handler to invoke.

It is important to point out that the use of kernel mode and user mode is enforced by the operating system as a way to protect resources. It is easy to think that you can perhaps manipulate a device better than the kernel can and that anyone could summon up operations in kernel mode. However, the kernel's role is to coordinate access and it only has a certain number of operations that are done in kernel mode. Most systems do not allow application code to perform kernel-mode actions, that is, to execute the instruction necessary to turn on kernel mode. However, various systems enforce this in very different ways.

That last point is especially true in Symbian OS. Since Symbian OS v9.1, Platform Security has implemented *capabilities*: any request to the kernel must be accompanied by the capability to make that request. This type of security makes sure that, while a user-mode program cannot simply make the kernel do anything, even the fixed number of tasks in the kernel are protected. Security issues and operating system protection are discussed in Chapter 14.

3.3 Interrupt Implementation

As we have mentioned before, an interrupt is a signal of some sort, typically generated by hardware, that is used to inform the kernel of some condition that needs attention. Typical interrupts are asynchronous device I/O notifications and initiation of device changes (such as allocation of memory or user-initiated read or write). Because interrupt service involves a change in system resources, servicing an interrupt is a kernel-mode operation.

The servicing of an interrupt is much like a kernel request from user-side code. The interrupt itself – the signal – can be seen as a request for service. There are two important differences, however, between interrupts and user requests. Interrupts must have a high priority and the servicing of interrupts is defined by a routine in the kernel's memory supplied by the device doing the interrupting.

Interrupts are typically designed to have some indication of priority. This is a recognition that some interrupts are more important that others. For example, an interrupt from a timer to force a context switch is probably more important than an interrupt from a keyboard that a character has been typed. A device's interrupt priority is typically selected based on two criteria: its *latency* requirements and its interrupt *execution time*. The latency requirement is the maximum time within which an interrupt must be serviced. (Usually, if it is not serviced in this time, some event is lost or performance is degraded seriously.) The interrupt execution time is the time required to service the interrupt. Interrupts with a short interrupt latency time must have a short interrupt service time.

Interrupts are serviced in a kernel by an *interrupt service routine* (ISR). There are many ISRs (because there are many types of interrupt) and only a few need to be recognized by the kernel at any given time. Therefore, most kernel designs maintain a vector table that contains the interrupt and a pointer to its ISR. This table is finite and is coordinated with the number of possible interrupt sources. When an interrupt occurs, the address of the ISR is looked up in the vector table and that ISR is called.

It is important to note that, while an ISR runs on the kernel side of an operating system, the *context* – or state – of the system cannot be assumed. An interrupt can occur at any time and the system could be in any state when the kernel is interrupted. This inability to access any kind of context information restricts what can be done in an ISR.

Interrupts are typically implemented by an operating in several phases:

1. The preamble phase saves the context of the executing process and prepares to execute the ISR.
2. The second phase determines which code to execute on behalf of the interrupt request (it *dispatches* the interrupt). This is either a built-in routine (in the case of a system class) or an external piece of code.
3. The third phase is the execution of the system call or ISR code, handled by privileged-mode code. Typically, this phase is itself interruptible.
4. The last phase implements the closure of the process (the 'postamble' phase). This typically amounts to a reversal of the preparatory phase:

the context is switched back to the interrupted process or storage is restored and execution resumes where it left off.

3.4 Completing the Kernel Design in Symbian OS

The smartphone platform is a unique one. It requires many real-time services, but also must provide an environment that is similar to a desktop system in its richness. In order to respond to both of these requirements, the Symbian OS kernel has a more complicated structure than the one outlined earlier in this chapter. In this section, we expand our look at the structure of the Symbian OS kernel by fleshing out a complete kernel structure.

The kernel structure is shown in Figure 3.2. It is organized in relationship to how system calls are made, that is, the path a user-mode program must travel to execute privileged code in the kernel.

The Symbian OS model starts by working with the peripheral hardware. Several kernel components communicate directly with a smartphone's hardware.

- *Device drivers*, the interface for program code to work with devices (see Chapter 2), are split into two pieces in Symbian OS: the *physical device driver* (PDD) interfaces directly with the hardware and

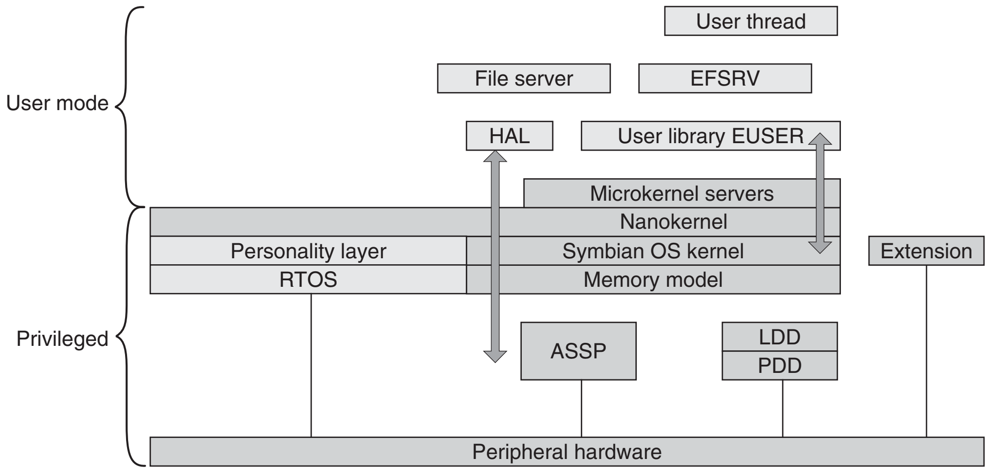


Figure 3.2 Structure of the Symbian OS kernel

the *logical device driver* (LDD) presents an interface to upper layers of software. In addition, the kernel can interact directly with hardware through the *application-specific standard product* (ASSP), which implements a number of components through a standard interface (so a specific driver is not needed). Finally, real-time components of the operating system – those specifically involved in phone calls – can also interact directly with the phone hardware when they run in a special mode (called the 'personality layer').

- The *memory model* used by the operating system is a model of how memory is organized on a device and how the operating system works with it. We deal with memory management and memory models in Chapter 7. Several memory models are possible in Symbian OS and these are implemented by the Symbian OS kernel.
- The *Symbian OS kernel* relies on the nanokernel, but is separate from the real-time portions of the operating system. It implements the various memory models that platforms require.
- The *nanokernel* implements the most basic and primitive parts of Symbian OS and is used by the phone part of the operating system as well as the larger kernel layer.
- The *real-time OS and personality layers* are specifically designed to implement phone functionality. The RTOS implements the GSM functions of a smartphone in direct connection with the hardware. The personality layer allows a smartphone manufacturer to use a different implementation of phone function (say, analog functionality) by using the implementation from another operating system or device and using a personality layer to connect that implementation to the GSM functionality of the smartphone. The personality layer then acts as an interpreter, translating the non-GSM implementation into an implementation the smartphone can understand.
- *User-mode layers* include microkernel servers as well as user applications. As we have discussed before, these interact with the Symbian OS kernel to request and initiate kernel-mode operations.

There are many paths through the Symbian OS kernel structure. A user-mode application might go through the file server, which would make a Symbian OS kernel request, which would require device I/O, which would make use of the nanokernel. A phone call might initiate functions in the RTOS, which would interact directly with the hardware.

An application might simply cause arithmetic instructions to execute and might not use any kernel functions at all.

Note that we did not mention the *extension* portion of the kernel structure. Extensions are device drivers that are loaded and executed when a phone boots up. They interact with the kernel and can cause kernel-mode operation. However, they represent layers in the kernel that extend functionality, but do not directly interact with user-mode applications. For example, the ASSP layer is an extension.

3.5 Summary

This chapter has been about how kernels are structured and how the various parts of a kernel interact with each other and with user-mode code. We began with a general look at kernel components from a layered perspective and the perspective of active and passive components. We then defined system calls and interrupts in relation to the kernel. We completed the chapter by taking a fresh and complete look at the Symbian OS kernel structure, from the hardware to user-mode threads.

In the next chapter, we begin to look at memory models and how memory must be organized to use it effectively.

Exercises

1. Consider the following services (seen in Chapter 2) and classify them as to where their implementation would take place in the kernel structure.
    a. Opening and closing files
    b. Writing to a register
    c. Reading a memory cell
    d. Receiving a text message
    e. Playing a sound bite.
2. Reconsider the following question from Chapter 2 and pinpoint the place these operations should happen in the kernel structure. Which of the following operations should be done in the kernel as privileged mode?

   a. Reading and writing files
   b. Sending a text message
   c. Taking a picture with the camera
   d. Notifying a program about data received
   e. Switching processes on the CPU
   f. Reading from a memory cell.

3. Consider a software timer that would be used by software as a 'wake-up device' or an alarm that would send a software interrupt when the timer goes off. Place a priority on the timer interrupt. Name some events that are more important than a timer event. Name some events that are not as important.

4. Should a timer be a real-time or a system-time object? In other words, should it be implemented by the RTOS or by the system kernel? Explain your answer.

5. Consider the phases of interrupt implementation (Section 3.3). We mentioned that ISR execution is pre-emptible. Should the other steps be pre-emptible? Give reasons for your answer.

6. Consider again the diagram in Figure 3.2. Why is the nanokernel on top of the Symbian OS kernel, which is on top of the memory model? According to Figure 3.1, the nanokernel is the innermost layer. Can you describe why the diagram in Figure 3.2 is accurate?

7. Symbian OS is an extensible operating system. If someone wanted to, they could write code that would run completely in kernel mode for each of its operations. Explain how this could happen – especially when we described system calls as built into the operating system.

8. Describe why you might want to replace passive components of an operating system with components that you could write. What could happen if this were done maliciously?

9. Consider how platform security in Symbian OS might be effective. Describe how forcing a system call to present capabilities before it is serviced might protect a smartphone from harmful effects of software.

4

Processes and Threads

Many people enjoy the circus. One act that I remember seeing as a child is a man who kept plates spinning on sticks. He would start one plate spinning, then add more and more until he had an incredible number of plates going at the same time. He would spend his time running from plate to plate, making sure all were spinning and none were rotating too slowly. While he was running himself ragged, all the plates amazingly stayed in the air. In a sense, this circus performer is a shared resource, powering all the plates in the operating environment. If he spent his time getting a single plate to spin perfectly, none of the other plates would get their turn. But if we accept the fact that plates do not spin perfectly and we allow plates to slow down a bit, we can get an environment where all plates are spinning.

Computer operating systems are like the plate spinner. They have a limited set of CPUs (usually only one) that are to be used by many processes at once. If an operating system were to let one process run to completion before others were allowed to use the CPU, the result would be a slow system where very little would get done. If all programs that wanted the CPU were allowed to try to grab it, there would be chaos and very little would get done. Therefore, we must force all programs to cooperate and share the CPU in an orchestrated, organized fashion. In doing so, we realize that a single program might take longer, but overall, all programs can use the processing power, in what looks like a parallel fashion.

The concept of a *process* is part of this orchestrated system. A process is a program that is in a state of execution. It forms the center point of a system that shares the CPU and grants access to all programs that need it. This chapter examines this system of sharing the CPU and the components that make it up. We start by looking at the big picture and giving an overview of the components of the process model. We then focus on processes and how they can be manipulated – both by the operating system and by users. We conclude the chapter by looking specifically at how the process model works on mobile phones based on Symbian OS.

4.1 An Overview of the Process Model

Before we discuss how the process model applies to various architectures, we should first define the components of the model. The discussion of processes sometimes suffers from what to call these components. If we consider all processes, we can define several different types that could run on a system. Batch systems run *jobs* that complete without interruption. In Chapter 2, we defined *user programs* or *applications* as programs that interact with users and run on time-sharing systems. We will refer to every executing program on a system as a process; a process may execute on behalf of a user or the operating system. Thus, any executing code (a job, a program or an application) is characterized as a process.

Processes

As we stated previously, a process is an executing program. A process is different from the program that defines it in several ways. First, a process is in a state of execution. This means that its code is being executed (or is waiting to be executed) by the CPU. Second, a process is obviously made up of more than just program code. A process is defined by code that executes, called the *text section*, but it is also characterized by a set of data (usually program variables), called the *data section* of the process, and the state of other hardware components it needs to run. These hardware components include the program counter (which holds the address of the instruction being executed in the process), temporary registers used to execute the process's instructions, and the program stack (containing data required to run the program from a language point of view: parameters, return addresses and local variables).

The difference between a program and a process demonstrates itself in many ways. For example, a program is a passive entity; a process is an active entity. A program can be thought of as the *definition* of a process; several processes that derive their definitions from the same program may be running on a computer. Each of those running processes, although associated with the same program, is different and unique. These processes would have the same text section, but their data sections would be different.

Process state

As it executes, a process is said to be in one of several *states* (see Figure 4.1). The state of a process is defined by what the process is doing at any given moment. We define process states as follows:

- *new*: a process being created is in the new state – its text section is constructed from the code in a program; its data section and stack are allocated in memory; and hardware components are initialized
- *ready*: a process in the ready state is available for running on a processor but waiting for execution
- *running*: a process in the running state is executing on a processor – it is manipulating the hardware allocated to it and the system is being altered according to its instructions
- *waiting*: a process in the waiting state is suspended and waiting for some external event to occur; the external event could be anything from receipt of an interrupt to the completion of an I/O request
- *terminated*: a process in the terminated state has completed its execution on a processor; when a terminated process is removed from system tables and data storage, it ceases to be a process.

Notice that the arcs in Figure 4.1 are labeled with the operation that is performed to move a process between the states at either end. For example, a process moves from 'new' to 'ready' through the process of creation. Notice, too, that the diagram describes the path taken between states. A process cannot move from the 'new' state immediately to a 'running' state. Likewise, a process cannot be waiting for an external event and move directly to the 'terminated' state. Finally, it is important to realize that, while only one process can be running on a single processor at any given moment, many processes can be in the 'waiting' and 'ready' states.

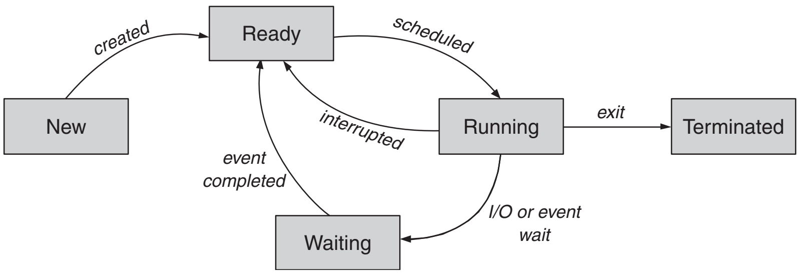


Figure 4.1 Process states

Process control block

There are several system components associated with a running process. These components are recorded by the operating system in a *process control block* (PCB), as shown in Figure 4.2. The PCB contains and records the various pieces of information that represent a process to the operating system.

The components of a PCB can be described as follows:

- *process state*: the current state of the process as it is manipulated by the operating system
- *process ID*: an identifier – usually an integer – that uniquely identifies the process in the system
- *program counter*: the program instruction being executed
- *CPU registers*: other registers used by the executing program
- *parent ID*: the identifier of the process's parent process
- *children IDs*: the identifiers of the process's child processes
- *scheduling information*: information pertinent to how often the process can use the processor
- *memory management information*: this information is important for the protection of memory areas; it includes the values of the base and limit registers, page table identifiers, etc.
- *accounting information*: timing information used by the operating system, including the amount of time used by the process and the limits on execution
- *I/O status*: the status of I/O devices that are being used by the process.

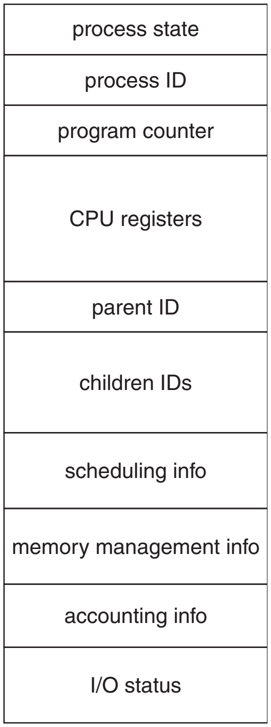


Figure 4.2 A process control block

The PCB represents all facets of a process to the operating system. As information about processes is stored by the operating system, the PCB serves as the unit of storage. Each process, therefore, has its own PCB and, implied by looking at a PCB, its own set of registers, memory space, accounting entries, I/O interactions, and so forth.

Process scheduling

A process moves through all the states in Figure 4.1 while executing on a system. However, as we stated before, a process shares the CPU with all other processes that are in the ready state. The operating system *scheduler* is the element in an operating system that takes a process from the ready queue and allows it to execute for a while. The act of moving a process from state to state and eventually to termination is called *process scheduling*.

As processes are created and enter the ready state, they enter a queue called the *ready queue*. The job of the scheduler is to take processes from this queue and allow them to execute for a time. This queue is represented in the operating system as a linked list of PCBs (we can think of each PCB as being augmented to include pointers to implement this linked list). All processes in the list that represent the ready queue are ready to execute.

The act of scheduling is represented by the removal of the head of the ready queue. Once a process is finished executing on the CPU, it is removed from execution. There are several ways to be removed from execution: a process could have exhausted the time slot it has been given or it could be blocked while waiting for an I/O event. The process's PCB is placed in the appropriate queue to await more processing. There is, therefore, more than one queue in a system. Processes waiting for external events are placed in a device queue. Each device has its own queue. Processes could be waiting for a system event not tied to a device, since there is an event queue for these processes.

So moving between states in an operating system amounts to moving PCBs between queues. A scheduler, then, moves processes from the ready queue to execution and from execution to one of the queues on the system. Scheduling requires working with many aspects of the operating system many times over. Chapter 5 deals with the intricacies of scheduling in more detail.

Implementation concepts

Figure 4.1 shows that processes must be created to enter the system. When it is created, a process is given a unique identifier, a process ID. The process ID, usually an integer, makes this process accessible in the table of processes running on the system.

A process must be created by the operating system – operating in kernel mode – or by another process. This relationship between processes is often characterized as a parent–child relationship. Parents create child processes, which go on to create other child processes, and so on. The entire collection forms a kind of family tree.

This hierarchical relationship is exploited in a number of ways. For example, it is typical that if a process receives an interrupt, then its children also receive that interrupt. It is also typical that a parent process cannot terminate until its children have terminated (while this behavior can be changed by system calls, it is the default behavior).

A *zombie process* is ready to terminate but for some reason cannot inform its parent of its termination. This could happen if the parent process was aborted for some reason. The child process tries to inform the parent of its termination and waits for the parent to respond. Since no response is forthcoming, the child process stays forever in the waiting state. It cannot terminate, but it cannot move to the ready state to be executed. Such processes are not unusual on large systems with much activity.

Threads

As an executing program, a process has a rather large 'footprint' on a computer system. It commands a system's resources and acts like it is the only program being run on a computer. Its PCB might take up quite a bit of process-table space and it might take a lot of time to move the process into and out of the running state. One of the components of a process is the thread of control – the set of instructions being executed on behalf of the program. This thread is orchestrated by the program counter and represents an executing program.

Consider the situation where a process has multiple threads of control. The other parts of the process remain the same: one PCB governing the process's information, one memory space, one set of accounting information, and so forth. Now, within a single structure, a process could run multiple tasks at one time with these multiple threads of execution. These tasks share the resources represented in a process's structure and would have to do so carefully. Such multithreaded processes have potential for executing faster than processes with a single thread.

Not all components of a process are shared. Threads do have some components they keep private (see Figure 4.3). For multiple threads to have their own execution space, each thread must command its own program counter and its own set of registers. Each thread becomes an entity that can be scheduled and has a set of interrupts that it responds to.

Multithreading communications

Let's say that you are using the Internet and begin to browse some web pages. Your browser program executes on your behalf by starting a process and moving that process through process states until it begins to share the CPU with the other processes on the system. You type a URL into the 'Address' field in the browser and click the 'Go' button. You now expect your browser to go and fetch a web page from the Internet and display it.

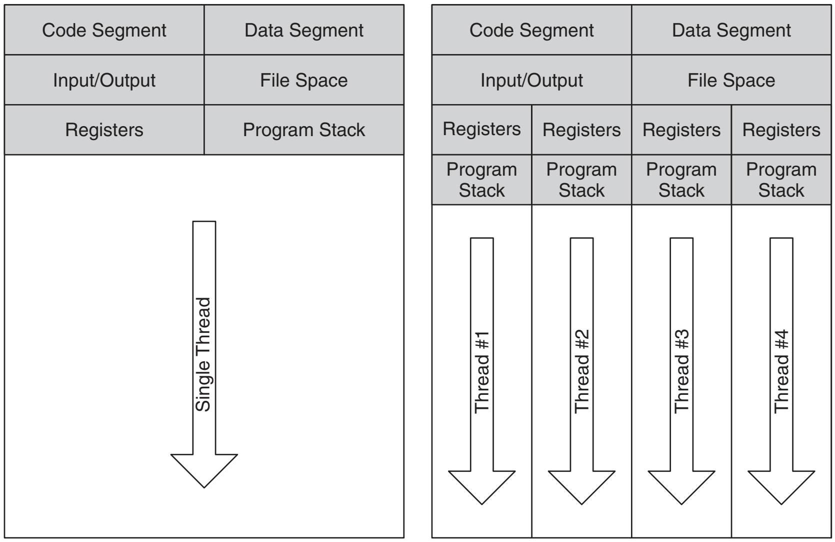


Figure 4.3 Singlethreaded and multithreaded processes

Consider what would happen at this point if your browser used a single thread of control. The browser interface would freeze while the process fetched and displayed the web page you requested. In fact, because web pages often result in many files to be fetched (for example, image and text files), the page would render quite slowly, because only one file would be fetched, then rendered, at a time.

Most web browser implementations are multithreaded. You can demonstrate this for yourself. Try viewing a complex website – one with many graphics – and use the menu system on your browser. Try downloading a web page then download it again when the first time is only halfway through. Even simple things show multithreading: roll your mouse over a link in a web page while it is downloading and you will usually see the cursor change. All of these activities would not be possible without a multithreaded implementation.

Consider the web server that answers the requests that web browsers make for web pages. If a web server used only a single thread, only one web page request would be serviced at a time. Requestors would have to wait a long time while the components of a single web page were delivered.

After considering these illustrations, you might think that multiple threads are the best way to program in applications. This is usually true.

However, there are times when a single thread is more beneficial. This is usually the case when a resource, such as a device or special sections of memory, is shared but must be accessed carefully – usually by one thread at a time. To do otherwise would corrupt the resource or the data derived from it. We discuss concurrency in operating systems more in Chapter 6.

Benefits of multithreading

Processes benefit from multithreading in a number of ways:

- *Sharing of resources*: threads share the memory space, the operating system resources and even the code of the process to which they belong.
- *Saving resources*: because threads share resources amongst themselves and with the process that spawned them, expensive and time-consuming allocation of new resources is not required for each thread; threads often require the same kind of attention that processes do but they require fewer resources which makes operations such as creating and scheduling a thread much faster.
- *Interacting with users*: multithreaded applications can interact with users while doing other things; if there are operations that require waiting or computing for a long period of time, multiple threads allow an application to attend to these long operations while still making the user feel as if she is in control. Because multiple operations can be going on at the same time, applications feel quicker and more responsive.
- *Accessing multiple processors*: if a computing system has multiple processors, multithreaded application can take advantage of this; multiple threads, like multiple processes, can each be scheduled to run on a separate processor. True parallelism can be achieved when separate threads of a single application run on separate processors.

User and kernel threads

When an application is multithreaded, it is composed of *user threads*. These threads run on behalf of an application, which is itself running at the user level. Threads that run on behalf of users at the user level consume user resources and interact with the system at the user level. The kernel is not needed until system calls are made.

If an operating system supports user-level multithreading, the threading of the kernel affects how multithreaded applications execute. As an analogy, consider a multilane highway that merges to a single lane as it goes through a small town. When the highway is busy, merging all those lanes of traffic onto the single road causes huge backups as cars in each lane wait their turn to enter the single road.

In a similar manner, some kernels that support multithreading at the user level support only a single thread at the kernel level. This is called a *many-to-one model* of kernel threading (see Figure 4.4a). Many user threads compete for a single thread of kernel control. As with the highway example, the result is a backlog of kernel requests and much waiting on behalf of user threads.

In a many-to-one system, applications still get some user-level benefit from multithreading but when they interact with the kernel there is no

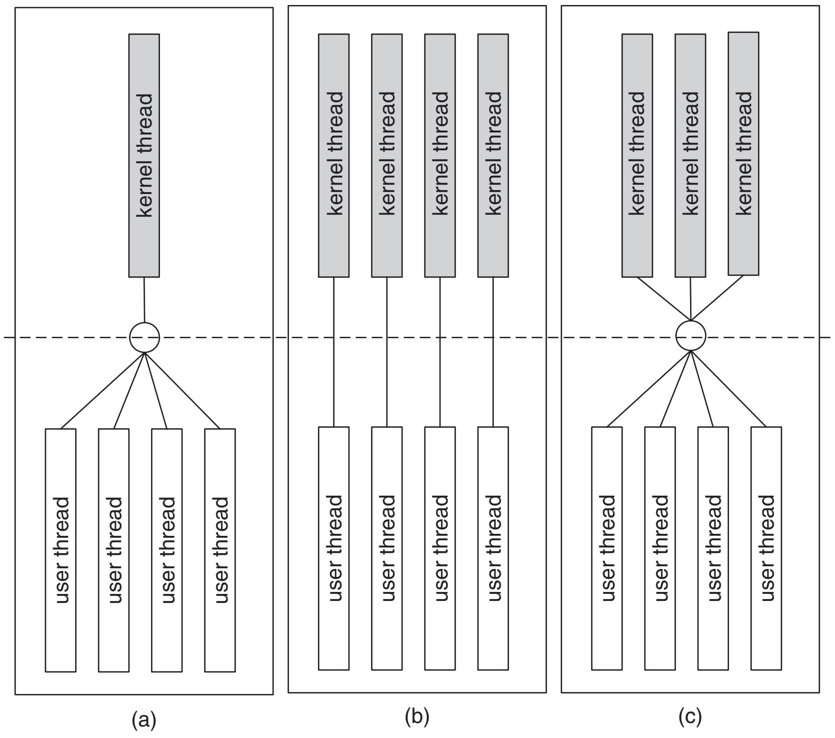


Figure 4.4 Kernel thread models

additional benefit. In fact, many systems that use singlethreaded kernels (for example, Microsoft Windows NT) restrict the number of user threads that can be used at any one time. Note that, in this model, a single thread can block an application. If multiple threads need the kernel, only one gets through and the rest block.

An alternative to a many-to-one model is a model where the kernel spawns a new thread for each user thread that makes a kernel-level request (see Figure 4.4b). This is called a *one-to-one model* of kernel threading. Each kernel thread services the requests from a single user thread. This increases concurrency and eliminates the backlog of requests. In the one-to-one model, each thread in an application could be serviced, which means an application does not block when a single thread needs kernel services. In this model, threads are created and destroyed based on the requests that come in. The overhead of creating kernel threads can start to affect the performance of applications. If an application requires many kernel services, it also bears the burden of creation and destruction of kernel threads. For this reason, operating systems that support this model (e.g., Windows 2000) restrict the number of threads supported by the system.

To get around constant kernel creation, most operating systems use a *many-to-many model* of kernel threading, shown in Figure 4.4c. In the many-to-many model, threads are created once and remain active. Many threads are created on system startup, but not the number needed for a one-to-one match with user threads that make kernel requests. These threads are kept around in a *thread pool* to service many requests. The result is some backlogging of requests as with the many-to-one model, but more efficient and concurrent processing of those requests as with the one-to-one model. The many-to-many model is better adapted for multiprocessor systems, as the number of kernel threads can be tuned for the number of processors. Many Unix implementations support the many-to-many model.

Issues with multithreading

There are many issues that surface when we expand our concept of processes and scheduling to include threads.

- *Thread management*: threads must be managed and scheduled; they go through states similar to processes (see Figure 4.1). Applications control the creation of threads but the actual creation and movement through states is done by the kernel. Note that this adds more complexity to the idea of scheduling a process and complicates decisions about scheduling. If a scheduler gives equal time to all processes, it must choose one thread to run on behalf of a process or give a tiny amount of time to each thread in a process, so that the total time on the threads equals the time spent on a process.

- *Process creation*: should a thread be allowed to create a new process or just other threads within a process? If we consider an application that does not create threads to be an application with a single thread, then we cannot deny any thread the ability to create a process (since such an application has no real 'main' thread). If any thread can create a process, then what does the new process look like? As we see later in this chapter, processes typically create other processes by cloning themselves. Do we clone a process, including created threads? The answer to this is usually yes and no. Systems that support process *forking* (the act of cloning a process) typically give that support in two versions: with and without all the created threads. This complicates programming with process creation.
- *Thread cancellation*: terminating a thread is referred to as *thread cancellation*. Because threads operate inside the larger context of a process, cancellation is not as traumatic to the system as process termination. In *asynchronous cancellation*, a thread may immediately cancel another thread. However, concurrency issues arise when threads are cancelled. For example, if a thread is updating data shared by other threads in a process, what happens to the data when a thread is cancelled? Often data is corrupted when such an event occurs. This is usually remedied by using *deferred cancellation*. In this situation, a thread cancels itself when the system tells it to terminate but waits for the appropriate time, for example, when no data is being manipulated.

We referred to interrupts that operate between processes as *signals*. One process can signal another to indicate a particular condition. When an application has multiple threads, it raises the question about which thread receives a signal when one arrives. All threads could receive the signal, or one or more threads could indicate they are waiting to receive a signal. Usually, all threads receive a signal sent to a process, although only certain threads deal with it.

Active Objects in Symbian OS

Threads in an operating system can be viewed as *lightweight processes*. As a lightweight process, a thread has many properties that a process does: it has an execution path, it can be scheduled, it uses resources, etc. Also, like a process, it requires bookkeeping: recording of where a thread is in its execution must occur so that the thread may be restarted when it gets a chance to execute on the CPU. So, while a thread is lightweight, it still has an effect on a running system.

On a system with restricted resources, such as a smartphone, the weight of threads has an effect. Much care is taken in restricted environments to be as light as possible. In these environments, processes are kept to a minimum and even threads are scrutinized as to their necessity. There is typically one main thread that controls most aspects of an application. When there are multiple threads in an application, they are usually spawned from the main thread and meant to deal with situations that cause waiting, for example, network communication or device I/O, so as to free up the main thread for other functionality.

It is possible to make a process – a kind of lightweight thread – that has less of an effect; it follows a fixed kind of behavior and is lighter on a restricted system. In Symbian OS, this lightweight thread is called an *active object*. Active objects are specialized threads that have some unique characteristics.

- Each active object is specifically focused on an event that causes a thread or process to block: communication, device I/O, and so on.
- Each active object works with a single active scheduler that listens for events for which the object is waiting. All active objects within a process use the same scheduler; different schedulers can be used between processes.
- An active object defines a specific entry point in its code that is used by the active scheduler when an event is generated. The entry point is represented by a standard function call that is defined for each active object.
- While waiting for an event, all active objects within a single process, because they are maintained by the same scheduler, act as a single thread to the system.

Active objects in a single process, therefore, can be coordinated by a single scheduler implemented in a single thread. By combining code into one thread that would otherwise be implemented as multiple threads, by building fixed entry points into the code, and by using a single scheduler to coordinate their execution, active objects form an efficient and lightweight version of standard threads. See Section 4.2 for some coding examples using active objects.

Contexts and Context-switching

A process has a *context*, that is, the collection of system resources that it is using represented by the process's PCB. These resources include the register set the process is using, the program counter, memory contents that implement data in its code, open files, identifying information such as a process identifier, and so on.

When a process moves to or from the running state to execute on a processor, several things must happen to the context. The context of the process that preceded the current process must be extracted from the processor and the context of the process to be executed must be installed on the processor. This act of moving contexts is called a *context switch*.

When switching contexts, the old context must be written to that process's PCB and the PCB must be stored. Then the PCB from the new process is read and restored. This procedure takes time and its efficiency is dependent on the amount of data in the PCB and how much the hardware assists. For example, some hardware architectures provide for multiple sets of registers and switching contexts simply switches a register set without copying (this is the case, for example, on UltraSPARC architectures). Context-switching incurs enough overhead that research into how to minimize it is worth the effort.

Processes on Linux

Execution on Linux systems is organized around processes. An executing program on Linux is a process, characterized by a PCB and given some memory area in which to store data. In Linux, PCBs are called *process descriptors* and these descriptors can be very complex.

A process in Linux can be in one of the states shown in Figure 4.1. In addition to these states, Linux defines two other process states: an *uninterruptible* state, where a process cannot be interrupted by other processes or operating system calls, and a *zombie* state, where a process is waiting for a parent process to receive notification of the child's termination. The uninterruptible state is rarely used, but is useful when a process initiates an action that must run to completion. For example, if a device driver takes a long time to initialize a device, it might need to run to completion and ignore any interruption (which would cause it to lose contact with the device it is initializing).

Processes in Linux are identified by a 32-bit integer. Even though process IDs (PIDs) are stored as 32 bits, Linux maintains backwards compatibility with older versions and only uses PIDs up to 32 767 (i.e., the largest 16-bit integer). When PIDs reach 32 768, the system starts the numbering process over and finds an unused PID starting at 0.

In Linux, processes not only record parents and children, but also siblings. Relationships much like those shown in Figure 4.5 exist. Sibling relationships make it much easier to pass around notification of events, such as interrupts and termination events.

Processes in various states in Linux are grouped together in queues. The ready queue holds the processes in the ready state that are waiting to be executed. There are also wait queues that hold the processes in the waiting and uninterruptible states. Processes in the zombie state do not need a queue, since there is no order to their movement out of the zombie state. Wait queues are split into subclasses of processes based on the event for which they are waiting. Those waiting for disk I/O, for example, are put together in their own queue separate from processes waiting for keyboard I/O.

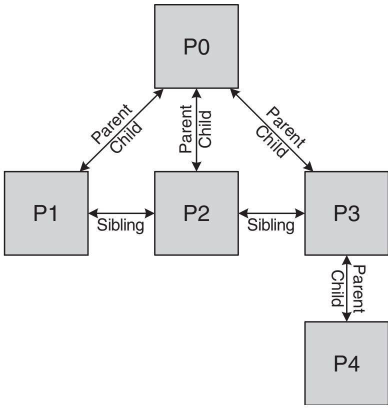


Figure 4.5 Process relationships in Linux

Context-switching in Linux often uses hardware to make the switching faster. Since Linux has been focused on the Intel hardware architecture, it takes advantage of hardware context-switching to automatically save contexts. However, using hardware to do this means that the operating system cannot ensure the validity of the process data. In addition, Linux has branched out from its Intel beginnings into other hardware. For these reasons, software context-switching using kernel-mode procedures is implemented in the most recent versions of Linux.

One final note should be made about the process hierarchy. Every family tree has a root and the Linux process hierarchy has a process with ID 0. This process is a process started by the boot sequence and it has the duty of initializing all operating system tables and of starting process 1, or the 'init' process. The init process has the job of starting all other processes that are needed to run a Linux system.

Threads in Linux follow a standard called Pthreads. Support for Pthreads follows the portable operating system interface (POSIX) standard and defines an API for thread creation and synchronization. This standard accompanies an older standard also supported by Linux. POSIX is a set of standard APIs for system calls and was invented as a way to smooth out the differences between Unix implementations. Most versions of Unix and Linux adhere to the POSIX standards, as do many other operating systems. Since support is voluntary, many operating systems – including Microsoft Windows and Symbian OS – provide support for POSIX in selected areas of API definition. POSIX is supported by the Open Group (for more information, see their web site at ***www.opengroup.org***).

Processes, Threads and Active Objects in Symbian OS

Symbian OS favors threads and is built around the thread concept. A process is seen by the operating system as a collection of threads with a PCB and memory space. Thread support in Symbian OS is based in the nanokernel with nanothreads.

Recall that the nanokernel is the basic, lowest level of the kernel. It provides very simple thread support in the form of *nanothreads*. The nanokernel provides for nanothread-scheduling, synchronization (communication between threads) and timing services. Nanothreads run in privileged mode and need a stack to store their run-time-environment data. Nanothreads do not run in user mode; this fact means that the operating system can keep tight control over each thread. Each thread needs a very minimal set of data to run, basically the location of its stack and how big that stack is. The operating system keeps control of everything else, for example, the code each thread uses, and stores a thread's context on its run-time stack.

Nanothreads have thread states in the same way as processes have states. The model used by the Symbian OS nanokernel adds a few states to the basic model in Figure 4.1. In addition to the basic states, nanothreads can be in the following states:

- *suspended*: different from the waiting state – a thread is blocked by some upper layer object (e.g., a Symbian OS thread)
- *fast Semaphore Wait*: waiting for a fast semaphore – a type of sentinel variable – to be signaled (see Chapter 6)
- *DFC Wait*: waiting for a *delayed function call* (DFC) to be added to the DFC queue; DFCs implement an interrupt service in Symbian OS, allowing a function call to be processed after the interrupt is serviced
- *sleeping*: waiting for a specific amount of time to elapse
- *other*: a generic state that is used when developers implement extra states for nanothreads, to extend the nanokernel functionality for new platforms (called *personality layers*); the developer must also implement how states are transitioned to and from their extended implementations.

Compare the nanothread idea with the conventional idea of a process. A nanothread is essentially an ultra-lightweight process. It has a mini-context that gets switched as nanothreads get moved into and out of the processor. Each nanothread has a state, as do processes. The keys to nanothreads are the tight control that the nanokernel has over them and the minimal data that make up the context of each one.

Symbian OS threads build upon nanothreads; the kernel adds support beyond what the nanokernel provides. User-mode threads that feature in standard applications are implemented by Symbian OS threads. Each Symbian OS thread contains a nanothread and adds its own run-time stack to the stack the nanothread uses. Symbian OS threads can operate in kernel mode via system calls. Symbian OS also adds exception handling and exit signaling to the implementation of Symbian OS threads.

Symbian OS threads implement their own set of states on top of the nanothread implementation to reflect the new ideas built into Symbian OS threads. Symbian OS adds seven new states for threads, focused on special blocking conditions that can happen to a Symbian OS thread. These special states include waiting and suspending on semaphores, mutex variables and condition variables (see Chapter 6). Remember that, because of the implementation of Symbian OS threads on top of nanothreads, these states are implemented in terms of nanothread states, mostly by using the suspended state in various ways.

Processes, then, are Symbian OS threads grouped together under a single PCB structure with a single memory space. There may be only a single thread of execution or there may be many threads under one PCB. Concepts of process state and process scheduling have already been defined by Symbian OS threads and nanothreads. Scheduling a process, then, is really implemented by scheduling a thread and choosing the right PCB to use for its data needs.

Symbian OS threads organized under a single process are connected in several ways:

- a main thread is marked as the starting point for the process
- threads share scheduling parameters; changing parameters for the process changes the parameters for all threads
- threads share memory-space objects, including device and other object descriptors
- when a process is terminated, the kernel terminates all threads in the process.

Active objects are specialized forms of threads and are implemented in a special way to lighten their burden on the operating environment. Remember that active objects are organized so that when they are brought back from a blocked state, they have a specific entry point into their code that is called. Since they run in user space, active objects are Symbian OS threads. As Symbian OS threads, active objects have their own nanothreads and can join with other Symbian OS threads to form a process to the operating system.

As active objects are Symbian OS threads, one can ask what the advantage is of using them. The key to active objects is in scheduling, which we cover in Chapter 5. It is important to realize, however, where active objects fit into the Symbian OS process structure. When a thread makes a system call that blocks its execution while in the waiting state, the operating system still needs to check the thread. The operating system spends time between context switches checking waiting processes to determine if they need to move to the ready state. Active objects wait for a specific event. The operating system does not need to check them but moves them when their specific event occurs. The result is less thread checking and faster performance.

4.2 Programming with Processes

How we think about processes affects how we program those processes. A key element, then, to understanding how processes work is to understand how they are programmed. This means studying the system calls and the programming patterns that are used when processes and threads are manipulated.

There are several types of programming methods for processes. An older, more traditional way focuses on processes. Another method incorporates threads into this conventional programming model. A third way of programming uses specialized thread objects such as Symbian OS active objects. These are all illustrated in this section.

The Conventional Model

The conventional model works at the process level. Processes are created by an operation called a fork, which creates a new process by building a new PCB and placing the new process in the ready queue. The fork operation is complemented by a 'join' or 'wait' operation. As shown in Figure 4.6, the fork operation splits the execution of one process into several and the join operation joins the process executions back into one. In Figure 4.6, at the dotted line labeled 'A', there are four processes executing. Join operations cause the subprocesses to terminate and combine their executions. By the end of Figure 4.6, there is a single process executing again.

In Linux, for example, a process creates a new process by calling the `fork()` system call. This system call creates a new process by cloning the current process's PCB. The result is two processes that look identical;

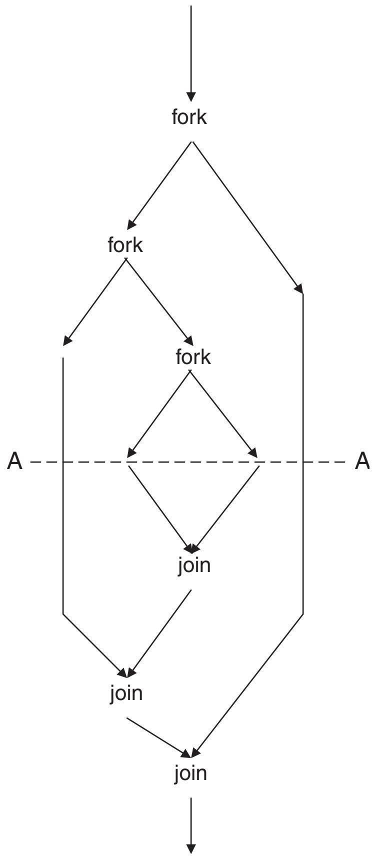


Figure 4.6 The interaction between fork and join operations

the new one is an exact copy of the old one. Both processes continue execution at the point of the `fork()` system call. The child knows it is a child because the `fork()` call returns a zero; the parent knows it is the parent because the `fork()` call returns a non-zero result, which is the process ID of the child process.

As an example, consider the following simple code:

```
#include <stdio.h>

void main()
  {
```

```
int pid, status;

pid = fork();
if (pid == 0)
  {
  printf("This is the child!\n");
  sleep(60);
  }
else
  {
  printf("This is the parent! Child's PID = %d\n", pid);
  wait(&status);
  printf("Child terminated...status = %d\n", status);
  }
}
```

The first executable code line is:

```
pid = fork();
```

Before this line, there is a single process. After this line executes, there are two processes. For one process, the child, `pid` has the value 0. For the other process, the parent, `pid` has a non-zero value that is the process ID of its child. In the example, the child simply prints a message, sleeps for 60 seconds, and terminates. The parent executes the line:

```
wait(&status);
```

which implements a join operation in Linux. A `wait()` call moves the caller process to the ready queue until the process for which it is waiting terminates. In this case, the call is waiting for the termination of any process the parent created.

Processes terminate in many ways. The normal way for a program to end is to simply run out of code. Most programming languages provide for some kind of call to `exit()`. In addition to allowing programs and processes to use the `exit()` system call, processes can also terminate their children. Signals that tell a process to terminate can be sent arbitrarily between processes, but ones sent from a parent to a child process are especially hard to ignore.

In this conventional process model, processes share very few resources. Each newly created process works in its own environment with its own PCB. This means each process has its own variables, its own access to resources and its own access to the kernel. What is shared are system resources such as input/output devices and memory. As we have discussed before, operating systems are designed to protect resource access and to share system resources in a very orchestrated manner, so that multiple accesses do not corrupt resources. This means that even though a child is created by a parent, they are peers from the point of view of the resource.

This conventional approach to multiprogramming is very resource intensive. The operating system must keep track of each process by recording and manipulating its PCB. Switching context between processes is expensive, because PCB data must be recorded and moved into and out of memory. Creating processes is also expensive because of PCB creation and the cloning of the parent context.

Sharing data between processes is quite difficult. The operating system puts up a barrier between processes and protects each process from others. Data can be shared in Linux by creating global, shared memory through system calls or by creating data channels called pipes that a process can share with children that it creates. However, these methods are special cases and are themselves expensive to the operating system.

Consider a sorting algorithm. In a quicksort implementation, an array of data elements is split up into pieces and a recursive call is made to the quicksort algorithm to work on each piece. The quicksort algorithm can be implemented in a parallel manner, where each recursive call is replaced by a new process. If we use the conventional model, we can get many processes involved. These processes can easily get started on the sorting process, because each has a copy of the data array (as the PCB is cloned). However, when it comes time to put the sorted pieces back together, the conventional model makes implementation very difficult. Somehow, only pieces of the data array must be communicated to a single process, which must put everything back together. Parallel versions of quicksort are rarely implemented with processes.

Programming with Threads

Threads solve many of the difficulties with processes. Threads are easily created: the creation process is lightweight because it is not resource-intensive, no PCBs are cloned. Ideas of forking execution and joining threads together still exist for threads, because they are truly parallel entities. However, working with threads is much easier.

Again consider Linux, so that we can compare it to the examples above. The following code contains a simple program to create and join threads of execution in a program.

```
#include <synch.h>
#include <thread.h>
#include <unistd.h>

void *child (void* args)
  {
  printf("This is the child!\n");
  sleep(60);
  }

void main()
  {
  thread_t yarn;

  thr_setconcurrency(2);
  thr_create(NULL, NULL, child, NULL, 0, &yarn);

  printf("This is the parent!\n");

  thr_join(yarn, &yarn, NULL);
  }
```

Notice that each thread of execution is implemented by its own function and concurrent threads are defined by function definitions. The code produces a child that prints a message and terminates, because its definition terminates. The parent creates the thread through a call to `thr_create()`. The parent waits to join its execution thread with the child's by calling `thr_join()`. This call blocks until the child specified by the `yarn` descriptor terminates.

Threads help to mitigate the difficulties found in the conventional process model. Thread creation is easy and lightweight; thread context-switching happens within a PCB. The operating system still has to keep track of threads, but bookkeeping requires less data. In addition, threads allow sharing of all resources of the parent.

The quicksort algorithm lends itself easily to a thread-based implementation, because recursive function calls can be replaced by thread creation. Like functions, threads share memory and the quicksort algorithm marks off portions of the same data array for each function or thread to work on. Joining the pieces back together is irrelevant, because each thread worked on a portion of the same array.

Programming in Symbian OS

Symbian OS supports processes, threads and active objects, the special case of threads focused on external events. Symbian OS supports standard and conventional interfaces, which allow processes and threads to be used in much the same way as on other systems such as Linux. Symbian OS supports the concepts of forking and joining processes, but does not clone a PCB when forking (it implements a 'fresh' executable image, not duplicated from the parent). There are a few other differences, but they do not affect the global concepts of process creation. Symbian OS also supports the POSIX thread manipulation APIs.

Active objects are unique to Symbian OS. Recall that active objects are Symbian OS threads that multitask cooperatively, that is, they are designed to facilitate asynchronous requests that wait for external events, usually tied to device I/O. The keys to using an active object are that each active object must release control to the operating system when it is ready to wait for an external event and that each active object must keep track of its internal state, because execution is restarted at the same place every time it restarts.

In Symbian OS programming, active objects are derived from the `CActive` class, which provides access to a `CActiveScheduler` object. An active object must implement at least two functions:

```
void RunL()
void DoCancel()
```

The `RunL()` function is the heart of an active-object implementation. Upon construction, the active object creates and initializes anything it needs. To inform the operating system that an asynchronous request has been submitted (that the active object must wait for), the active object calls the function `SetActive()`. This suspends the active object thread, turning control of the asynchronous operation over to a scheduler. When this operation completes and generates an event the active object has registered for, the scheduler calls the `RunL()` function for the active object.

The `DoCancel()` function must be implemented to cancel the actions implemented by the active object. On receiving a request to cancel the operations for an active object, the system first checks whether there is an outstanding request for this object, then calls `DoCancel()`.

There must be a way to start the active object. This should be a function that initializes the active object and executes the first asynchronous request. This function then passes control to the scheduler by calling `SetActive()`.

Consider an example that sends an object over the serial port of a Symbian OS device. We could start this sending process by calling the following function:

```
void TodoXferSerialAO::SendItem(CAgnEntry *aEntry)
  {
  iEntry = aEntry;

  CParaFormatLayer *iParaFormatLayer = CParaFormatLayer::NewL();
  CCharFormatLayer *iCharFormatLayer = CCharFormatLayer::NewL();

  // Set up to-do item data
  iTodoItem = CAgnTodo::NewL(iParaFormatLayer, iCharFormatLayer);
  iTodoItem = (CAgnTodo *)(aEntry->CastToTodo());
  priority = iTodoItem->Priority();
  duedate = iTodoItem->DueDate();

  // Start the protocol
  iAppUi->SetProgress(_L("Starting the protocol"));
  buffer.Copy(KReady);
  Send(buffer);

  iSendingState = ESendingState1;
  }
```

The `SendItem()` function receives a to-do list entry (the parameter from the `CAgnEntry` class), prepares it for sending, then calls a `Send()` function to send the object through the serial port.

The `Send()` function is defined as follows:

```
void TodoXferSerialAO::Send(const TDes8& aText)
  {
  TInt len;
  TBuf8<100> buffer;

  len = aText.Length();

  // Send the length
  buffer.SetLength(0);
  buffer.Append((TChar)(48+len));
  buffer.Append(aText);
  commPort.Write(status, buffer, len+1);

  SetActive();
  }
```

There are some interesting parts of this implementation. First, note the structure of the code: it sets up some parameters for serial communication and then calls `Write()`. This is an asynchronous call that returns immediately, beginning the communication process in parallel. Secondly, we use a variable – `status` – to remember the state we are in. Finally, notice the use of `SetActive()` at the end of the code. When this system call is performed, the active object's execution is terminated and the thread is placed on a waiting queue.

Once we have started an active object and turned over control to the scheduler, we need a way to continue its execution whenever I/O operations complete. This is found in the implementation of the `RunL()` function. The most common pattern embraced by this function is essentially to use one big `switch` statement, based on the value of the state variable. Each `case` in the `switch` statement looks something like this:

```
case ESEr1:
  if (status == KErrNone)
    {
    if (buffer.equals(KOk))
      {
      iSendingState = ESError; // indicate an error
      }
    else
      {
      iAppUi->SetProgress(_L("Sending the priority"));
      buffer.Format(KPriorityFormat, priority);
      Send(buffer);
      iSendingState = EStateXfer1;
      }
    }
  else
    {
    iSendingState = ESError; // indicate an error
    iAppUi->SetProgress(KRcvErrMessage);
    }
  break;
```

At the beginning of each case, we can assume that the previous I/O operation has completed with a value stored in a status variable. We check the status variable to determine if the operation completed successfully. If it was successful, we process the results, engage another I/O operation, and change the state variable for the next invocation of `RunL()`. If it has not completed successfully, we must deal with it somehow – perhaps aborting the active object or resetting the communication stream. At the end of the `case`, if we want to continue, we must call `SetActive()` again.

4.3 Summary

This chapter discussed the concepts of working with processes and threads in an operating system. We began the chapter by examining the process concept and how much of a process's conceptual and actual implementations are derived from the process control block. We defined how the operating system handles processes and how that handling has some difficulties in implementation. We then defined threads and showed how threads relate to and improve upon processes. User-level threads have implications about kernel threads and we defined how these relate to each other. We also discussed the Symbian OS concept of active objects as special versions of threads.

We discussed concepts of implementation and programming of processes, threads and active objects. Processes use concepts of forking and joining and we discussed how to program this type of activity in Linux. We also showed how threads can be programmed. We concluded the chapter by discussing some programming issues with active objects in Symbian OS.

We have left the topic of how processes and threads are scheduled to Chapter 5.

Exercises

1. Give examples of multithreading from the applications that you use every day. Describe how you think each example would work if a single thread of control were to be used.
2. Consider situations where multithreaded applications would not be more useful than singlethreaded applications. Describe two examples of this and give your explanation as to why there is no performance improvement.
3. Give two situations where kernel multithreading would definitely be an improvement over singlethreading.
4. Give two situations where kernel multithreading would not be an improvement over singlethreading.
5. Compare context-switching between user-level threads with context-switching between kernel-level threads. Where are they the same? Where are they different?

6. Consider how a process is created. Compare the procedure with how a thread is created. How are resources used differently?
7. Compare the creation of active objects with the creation of threads. How do you think these procedures are different?
8. Give two situations where active objects would not be better than threading. Explain your thinking.

5

Process Scheduling

We introduced the last chapter with a circus performer: a man that I remember from childhood who kept plates spinning on sticks. He could spin many plates at the same time. While his performance seemed to be focused on the spinning plates, I suspect that his real skill lay in the choice he made after he paid attention to a single plate. In the split second where he ran from one plate to another, keeping each spinning on those long sticks, he had to make a choice as to the plate that needed his attention most. If he chose poorly, at least one plate would begin to wobble and eventually fall off its stick and break. If he chose wisely, he kept all the plates spinning.

We can think of this circus performer as a scheduler. He needs to make important choices that schedule plates for 'spin maintenance'. Some plates probably need his attention more than others and he needs to make his choices wisely, according to the needs of the plates.

Computer operating systems are like that. They have a limited set of CPUs (usually only one) that are to be used by many processes at once. As the operating system shares the computing resources, choices must be made. How long should a process operate on a CPU? Which process should run next? How often do we check the system?

The concept of scheduling a CPU is very important to keeping a computer running quickly and efficiently. This chapter introduces the basic ideas of CPU scheduling and presents several scheduling algorithms. We also examine how these concepts and algorithms apply to various types of operating system architectures.

5.1 Basic Concepts

The concepts involved with scheduling a CPU seem simple on the outside but are really quite difficult upon closer inspection. The idea of multiprogramming is essentially a simple one: several processes share processing time on a CPU. The idea of sharing is an easy concept to grasp. However, it is the mechanics of this sharing that is difficult. Processes must be started appropriately and stopped at the right time, allowing another process to take over the CPU. What is appropriate? How long does the process have the CPU? What is the next process to take over? These questions make sharing a difficult concept indeed.

Concepts of Sharing

We need to be clear on how the CPU is shared. The act of scheduling is the act of moving processes from the ready state to the running state and back again. Recall that processes in the ready state are waiting in the ready queue. This ready queue is not necessarily a FIFO queue: processes do not necessarily enter and leave in a fixed order. In fact the choice of which process to move from the ready queue to running is at the heart of process scheduling.

The way CPU sharing is controlled is important. Methods of sharing should accommodate the way that a process works. For example, we could allow processes to share a processor at their own discretion. This would mean that sharing would be dependent on each process – dependent on when each process decided to give up the processor. This makes it easy for a process to hog the CPU and not give it up. Obviously, this makes the operating system a bit simpler, but would not be a great way to equitably share things on a general-purpose computer. We could also move scheduling decisions away from each process and give them to a third party – perhaps the operating system. This would make scheduling less dependent on the whim of each process and more dependent on policies implemented by a central controller.

When a process moves from the running state to the ready state without outside intervention, we call the scheduling mechanism a non-pre-emptive mechanism. Many movements from the running state are non-pre-emptive. When a process moves to the waiting state or a process terminates, it does so by its own choice. In non-pre-emptive scheduling, a process may hang on to the CPU for as long as it wants (or needs) to.

By contrast, *pre-emptive scheduling* allows the operating system to interrupt a process and move it between states. Pre-emptive scheduling is usually used for general-purpose operating systems because the mechanism used can be fairer and processes can be simpler. However, pre-emptive scheduling can have costs associated with it. It requires more hardware support: timers must be implemented to support the timing criteria for processes and ways of switching between processes must be supported by registers and memory. The operating system must also provide secure ways of sharing information between processes. Consider two processes sharing data between them. If one is pre-empted as it is writing data and the second process is then run on the CPU, it might begin to read corrupted data that the first process did not completely write. Mechanisms must be in place that allow the processes to communicate with each other to indicate that such conditions exist.

Pre-emptive scheduling affects how the operating system is designed. The kernel must be designed to handle interrupts for a context switch at any time – even the most inopportune times. For example, if a process makes a system call that causes the kernel to make system changes but it is pre-empted, what happens to the changes made by the kernel? This is complicated by the chance that the next process might depend on the changes made by the previous process's system call. Corruption of system data is likely if this is not handled correctly. In a case like this, a Linux system would force the context switch to wait until the kernel mode changes were completed or an I/O call is made. This method ensures that processes sharing one CPU serialize access to system resources. Even this way of coordinating access to system resources is not sufficient when there are multiple CPUs or the operating system supports real-time processing.

Most modern operating systems use pre-emptive schedulers but there are several examples of non-pre-emptive kernels. Microsoft Windows 3.1 used a non-pre-emptive scheduler. Applications could give up control in several ways. They could give it up knowingly or they could give it up through certain system calls or I/O functions.

Early Apple Macintosh operating systems were also non-pre-emptively scheduled. Early systems were based on the Mach kernel, an open source design developed at Carnegie Mellon University to support operating system research, primarily distributed and parallel computation. In version 10, MacOS was based on FreeBSD (technically the XNU kernel), which is pre-emptively scheduled.

The part of the operating system that actually performs the pre-emptive context switch is called the *dispatcher*. The dispatcher is comprised of the set of functions that moves control of the CPU from one process to another. The dispatcher must enter kernel mode, interrupt the process that is currently running on the processor, move that process to the ready queue, choose the next process to use, activate that process's code, switch to user mode, and cause the processor to begin execution at the appropriate point in the new process. This is a tall order; there are many procedures to be performed. While the dispatcher must run as fast as possible, there is overhead involved with doing its job and therefore there is a certain latency that is experienced when the dispatcher is called in to do a context switch. This *dispatch latency* is inherent in the system.

Scheduling Criteria

The dispatcher is supposed to make decisions about when to remove a process from the CPU and which process to assign the CPU to next. The dispatcher makes its decisions using several criteria. There has been much research devoted to the best way to schedule a CPU.

The CPU must be kept as busy as possible. *CPU utilization* is a criterion that measures the percentage of time the CPU is busy; we want this percentage as high as possible. Because of the reality of executing programs, CPU utilization is rarely at 100 percent, but a well-managed system can achieve high CPU utilizations of 75 to 90 percent.

Another measure of CPU activity is the amount of work done over a period of time. Called *CPU throughput*, this measure can be calculated in several different ways. For example, the number of jobs per day is a coarse measure. A finer measure is the number of processes completed in a time unit. Short database transactions could be measured in processes per second while longer computations might be measured in processes per hour or per day.

Another issue in scheduling is fairness, i.e., a measure of how much time each process spends on the CPU. We want to make an effort to make the times fair. Note here that 'fair' does not mean 'equal' in all situations. Sometimes, certain processes need to spend more time on the CPU than others.

Turnaround time is yet another criterion upon which we can base scheduling decisions. Turnaround time refers to the amount of time a process takes to execute. This is measured from the time of definition to the operating system (i.e., the time it left the create state) to the time of termination (the time it entered the terminate state). Turnaround time looks at all the activities of a process – including all time spent running on the CPU, all time waiting for I/O, all time in the ready queue, etc.

Another criterion is the amount of *waiting time* a process does. Waiting time is the total amount of time waiting in the ready queue. Waiting time does not include the amount of time waiting for the system – such as I/O time – because these times are not affected by CPU scheduling.

Finally, *response time* is a criterion often used in making scheduling decisions. Response time is most often used in scheduling for interactive systems. In interactive systems, measures such as turnaround time are not as important as how a process responds to requests. The time from submitting a request to receiving a response is how response time is defined.

In general, the goal of any scheduling strategy is to maximize CPU usage and throughput while minimizing turnaround time, waiting time, and response time. Over the running of an operating system, it is often necessary to consider averages of these measures or the minimum or maximum of them.

Scheduling strategies often use different criteria for different types of systems. Batch-mode systems concentrate on receiving tasks and getting them done with little or no human interaction. For these systems, response time is of little interest, but minimizing turnaround time is very important. Similarly, a server-based system would want to maximize response time as it is based on request and response pairings. A desktop system would want to maximize response time and minimize waiting time.

Microkernels use an interesting set of criteria. Microkernels are used for both general-purpose operating systems and for specialized systems such as mobile phones. Because of this, different criteria are applied for different situations. A microkernel usually moves scheduling from servers and makes it a kernel-mode activity, thus acting like a general-purpose operating system. However, a microkernel must pay close attention to fairness, because many of the processes in user space implement system functions. In addition, there is a fair amount of overhead in a microkernel system, because system processes communicate with each other by passing messages (rather than through kernel memory). Keeping CPU utilization high, the communication overhead low and scheduling fair is a difficult thing for microkernels.

A mobile phone system is a microkernel with a mixture of system types. It has elements of real-time systems and elements of interactive systems. A phone-based operating system would want to minimize response time and waiting time, but turnaround time is not very important as there are few applications that are started to do short, specific tasks. Throughput is hard to measure on a phone-based system, because processes are designed to service requests in the long-term and they stay running for relatively long periods of time.

5.2 Scheduling Strategies

We have just discussed the concepts involved in scheduling processes to share a CPU and the criteria used to make decisions about CPU usage. There are several strategies that use scheduling concepts and criteria to implement CPU sharing. As we discuss these strategies, note that we focus on the problems of deciding which process should use the CPU and when a process should be removed from using the CPU.

First-Come-First-Served Strategy

Perhaps the easiest way to schedule a CPU is on a first-come-first-served (FCFS) basis. This scheduling strategy allows the first process that requests a CPU to use the CPU until the process is completed. When one process is using the CPU, other processes that need the CPU simply queue up in the ready queue. This allows the head of the ready queue to be used as the next process to be scheduled. This scheduling strategy is non-pre-emptive. Processes are removed from the CPU only when they are in the waiting state or they have terminated.

Consider the following set of processes that arrive to use a CPU in the order stated:

Process	Time Needed
P_1	29
P_2	5
P_3	15
P_4	4

An FCFS scheduler would schedule them as shown in Figure 5.1.

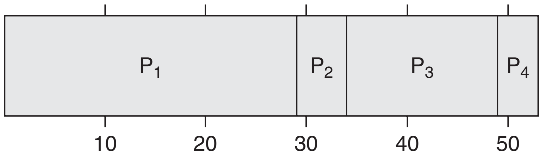


Figure 5.1 Processes scheduled using a first-come-first-served strategy

We can state the turnaround time and the waiting time for the processes in the following table:

Process	Turnaround Time	Waiting Time
P_1	29	0
P_2	34	29
P_3	49	34
P_4	53	49

The throughput of our imaginary system is 4 processes in 53 time units, or 0.075 processes per time unit.

It is also useful to consider average measures, such as the *average waiting time*. In our example, the average waiting time is 28 time units.

The order in which the processes are granted requests makes a large difference to the measurements we take. Consider a different ordering of processes, shown by the time bar in Figure 5.2.

In this situation, we can measure time in the following way:

Process	Turnaround Time	Waiting Time
P_2	5	0
P_4	9	5
P_3	24	9
P_1	53	24

In this scenario, the throughput of our system is the same: 4 processes in 53 time units. But the waiting time is much better: the average waiting time for this example is 9.5 time units.

It is easy to see how a FCFS strategy does not guarantee minimal criteria and measures may vary substantially depending on process execution

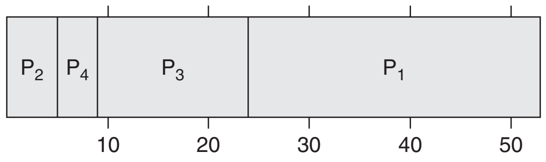


Figure 5.2 Processes scheduled using a FCFS strategy in a different order

times and the order of execution. Fairness issues hurt the consideration of this scheduling strategy. FCFS is inherently unpredictable and may very likely produce unfair schedules.

Shortest-Job-First Strategy

Another non-pre-emptive strategy can be invented from the examples above. The first example ran the longest process first (because it requested first) and, in doing so, worsened the measurements that we took (an average wait time of 28 time units rather than the 9.5 time units when we scheduled the shorter processes first). If we always chose the process with the shortest running time first, it would seem that we could improve the measurements we are watching.

As an example, consider a new set of processes:

Process	Time Needed
P_1	20
P_2	3
P_3	26
P_4	7

The *shortest-job-first* (SJF) scheduling strategy is illustrated in Figure 5.3.

This ordering produces the following measurements:

Process	Turnaround Time	Waiting Time
P_2	3	0
P_4	10	3
P_1	30	10
P_3	56	30

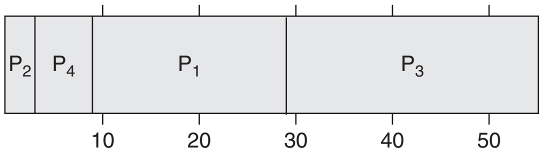


Figure 5.3 Processes scheduled using a shortest-job-first strategy

This order of processing has an average wait time of 10.75 time units. If we had used the FCFS strategy, the average waiting time would be 12.25 time units.

While it is possible to prove that an SJF strategy is optimal for average times, the strategy has several issues. First, it penalizes long processes simply for being long. Secondly, it becomes possible to starve a process. Starvation occurs when a process is waiting in the ready queue but never makes it to the running state. As long as processes enter the queue with running times shorter than it, that process is never run on the CPU.

Finally, the hardest thing about an SFJ strategy is very basic: knowing how long a process will take to run. Processes typically do not enter the ready state having determined in advance their running time. In fact, an interactive process – say, a user application with a GUI – may never terminate ('never' is, of course, a relative term; read that as 'not until the CPU stops functioning'). We can make estimates based on past behavior or estimate running time based on the process type. Typically, if some kind of prediction is to be made of running time, it is a weighted average, using, for example, a binomial distribution:

$$T_{n+1} = aT_n + (1 - a)T_{n-1}$$

In this calculation, *a* represents the weight we want to place on more recent timing measures. This type of estimate might take into consideration a long history of timing. This technique is called *aging* and is applicable to many situations where we must estimate some property in the system.

Round-Robin Strategy

Both FCFS and SJF are usually used as non-pre-emptive strategies. However, we still have the criterion of fairness to consider. If we schedule processes to run to completion or we depend on processes to give up the CPU when they can, we can make measurements but we can make no statement about fairness. Fairness can only be assured when we use a pre-emptive strategy.

One of the oldest and simplest pre-emptive strategies is the *round-robin* scheduling strategy. In a round-robin strategy, all processes are given the same time slice and are pre-empted and placed on the ready queue in the order they ran on the CPU. The ready queue becomes a simple list of processes that are ready to run and each process is placed on the end of that list when pre-empted from the CPU.

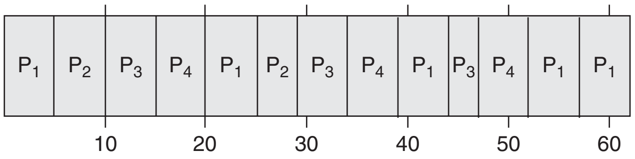


Figure 5.4 Processes scheduled using a round-robin strategy

For example, consider the processes below:

Process	Time Needed
P_1	25
P_2	9
P_3	13
P_4	15

Let's say the time slice in this system is 5 time units. A round-robin scheduling strategy would produce a timeline like that shown in Figure 5.4.

There is very little to manage about a round-robin strategy. The only variable in the scheme is the length of the time slice – the amount of time spent on the processor. Setting this time to be too short – say close to the time it takes to perform a context switch – is counterproductive. It lets the context-switching time dominate performance and lowers CPU efficiency. However, making a time slice too long gets away from the benefits of a round-robin strategy. Response time decreases for short requests.

In addition, the round-robin strategy (in common with the FCFS and SJF strategies) ignores a very important concept: some processes are more important than others and should be run more often. Kernel processes are usually more important than user processes to the maintenance of an operating system. Even within kernel processes, some are more important to efficient use of operating system resources and some are less important. A round-robin strategy puts all processes on an equal footing.

Priority Strategy

A *priority-scheduling strategy* takes into account that processes have different importance placed upon them. In priority scheduling, the process in the ready queue with the highest priority is chosen to run. This type of scheduling can be either pre-emptive or non-pre-emptive, as it is the choice of the next process that defines a priority-scheduling strategy.

Priority scheduling makes certain requirements of the operating system. First, and most obviously, the operating system must employ the concept of *process priority*. The priority is an attribute of a process that measures the importance of the process in relationship to others and allows the operating system to make decisions about scheduling and the amount of time to keep a process on a processor. In a pre-emptive scheduling environment, process priority is a very useful attribute to assign to a process. The priority of the process is usually given by a number, which is kept in the process's PCB.

It is usually required that the operating system be allowed to manipulate priorities somehow. Priorities are set by the user or the process creator but the operating system is usually allowed to change priorities dynamically. This is done to reflect the properties of the processes. For example, if a process spends most of its time waiting for I/O (is I/O-bound) then when it returns from I/O requests, it is usually considered fair play to give it the processor when it wants it. The operating system requires the ability to adjust the priority of such a process as it moves through its execution.

Priorities themselves usually take the form of numbers, quantities that can be easily compared by the operating system. Microsoft Windows assigns values from 0 to 31; various Unix implementations assign negative as well as positive values to reflect user (negative) and system (positive) priority assignment. As that shows, there is no general agreement on assigning priority values.

Consider an example of priority scheduling. Let's say that requests for the processor are made in the following order:

Process	Time Needed	Priority
P_1	11	5
P_2	4	4
P_3	26	11
P_4	7	5
P_5	10	20

For the purposes of this example, higher numbers mean higher priority.

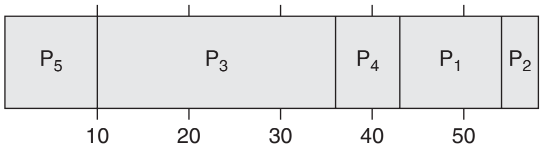


Figure 5.5 Processes scheduled using a priority strategy

If a priority scheduler is used, then the scheduling of processes in a non-pre-emptive scheduling environment could look similar to Figure 5.5.

At each stage, the process with the highest priority is chosen to replace the process leaving the CPU. The average waiting time is 36.6 time units, which is not better than the other strategies, but the more important processes do get executed sooner.

Priority scheduling seems to address the reality of scheduling processes in an appropriate manner. However, there are a few issues that we must attend to in order to implement priority scheduling correctly. The first is *process starvation,* an issue we have addressed with other strategies. It is possible to construct a sequence of process-scheduling requests where there is a process that is never scheduled, because each new request has a higher priority. In this scenario, the process at the lowest priority level is starved and never gets the CPU. This is an issue especially for heavily loaded systems, where there are many processes and the likelihood of a high-priority process is great.

The solution to starvation is to include the concept of *aging* into priority scheduling. The idea is to change the priority of a process as time goes by. This can be done by reducing the priority of higher-priority processes or by raising the priority of starved processes. Usually, higher-priority processes are reduced in priority because that has a more rapid effect.

Consider the big picture of processes: which process should have the lowest priority? On most operating systems, the process with the lowest priority is the process that does the most menial of tasks. It is the process that, should it *never* get scheduled, no harm is done.

On Microsoft Windows, this process is called the ‘System Idle Process’ and is exactly what its name implies: an idle process. When the system is idle, this process is eventually run. It does nothing – just takes up CPU cycles until something else needs the processor.

In a Unix system such as Solaris, this idle process is the scheduler itself. When the system boots, it starts all processes by starting the scheduler. This scheduler starts the system initializer, called ‘init’, that spawns all other processes. When all processes are idle, the scheduler schedules itself and runs code that burns up CPU time without doing anything.

Multiple-Queuing Strategy

We have described priority scheduling as a matter of choice: choosing the process with the highest priority to schedule on the processor. In many operating systems, processes do not have unique priorities. There may be many processes with the same priority. Many system processes have the same high priority. This means that the scheduler is eventually going to have to choose between processes with the same priority.

Priority scheduling is often implemented with *priority queues.* A priority queue holds processes of a certain priority value or range of values. The idea is that, if returning a process to the ready queue places that process in one of several priority queues, scheduling is simply a matter of taking the head of the next nonempty queue and placing that process on the processor. The result is a faster scheduling choice when looking for a process to run on the CPU, but a slower return of that process to the ready queue (or queues).

We can generalize on this idea. If we think about grouping processes into various classes, each class can have its own scheduling queue. This *multiple-queuing scheduling strategy* could even use multiple strategies: different scheduling strategies for different queues. Processes are either permanently assigned to a specific queue, based on their characteristics upon entering the system, or they can move between queues, based on their changing characteristics as they are executed in the system. Note that this assumes that certain characteristics are either derivable from a process or that the process states its characteristics as it enters the system. In turn, allowing a process to change queues implies that a process can communicate with the operating system about its changing requirements (know as *multilevel feedback*) or that the operating system can somehow derive that a process's needs have changed.

Real-time Strategy

As we briefly mentioned in Chapter 1, real-time systems can be classified as one of two different system types, each with different scheduling needs. Hard real-time systems guarantee that time constraints are met. In hard real-time systems, there is usually a specific amount of time that is specified along with the process-scheduling request. The system guarantees that the process is run in the specified amount of time or that the process is not run at all. The system first responds by either accepting the process for scheduling or rejecting the request as not possible. If the system accepts a hard real-time process-scheduling request, it must base this decision on its knowledge of the request and its characteristics matched against its knowledge of the system upon which it is running and its resource characteristics. In order for a system to know itself this well, specialized software and hardware are typically needed.

Soft real-time systems place a priority on time-critical processes and are less restrictive. Soft real-time systems do not guarantee performance; they give real-time processes favorable treatment and keep certain latency times to a minimum. A real-time operating system must be able to assume that scheduling overhead is restricted to a certain time. Specifically, two benchmarks must be bounded for the operating system to able to schedule in real time: the time from an interrupt to a user thread and the time from an interrupt to a kernel thread. If these times are accurately predictable, even a general-purpose operating system can support soft real-time scheduling.

Soft real-time systems are usually scheduled using one of two methods. A process has a fixed amount of time in which it can execute, called its *deadline*. If an operating system can manipulate process priority such that real-time processes are scheduled in increasing deadline order, and before non-real-time processes, then it can be shown that this abides by the rules of real-time scheduling. This method is known as *static, monotonic scheduling*. It does not produce optimal scheduling, but it is ‘good enough’ for many real-time needs. It is simple and efficient, suitable for memory-limited systems.

When it is not sufficient to be ‘good enough’, an operating system must make scheduling choices by paying closer attention to deadlines. In *deadline-driven scheduling* schemes, choices are made based on the priority of the process in addition to some consideration of deadlines. For example, in an earliest-deadline-first scheduling strategy, choices are made by computing how close a process is to its deadline. The choice of the next process to schedule is the process closest to its completion.

Real-time scheduling is a complex issue. Much research work has been done on this topic that proves complicated scheduling can be accomplished by adopting some simpler policies. [Leung and Whitehead 1982] and [Liu and Layland 1973] are well worth reading for some more information on this topic.

5.3 Scheduling in Linux

Process scheduling in Linux tries to accommodate all kinds of needs. It is a time-sharing system, giving general-purpose processes a time slice and multiplexing the CPU based on those time slices. This means that the Linux process scheduler is a pre-emptive scheduler. Understanding the importance of certain processes over others, Linux also uses a dynamic process-priority strategy. In addition, Linux uses a real-time scheduling strategy with deadline scheduling for real-time processes. Thus, Linux handles all types of processes, from interactive processes through batch processing, even real-time processes.

The Linux scheduling algorithm divides CPU time into *epochs*. At the beginning of each epoch, the time slice of every process waiting to be scheduled is recomputed. This means that different processes generally have different sizes of time slice. If a process does not exhaust its time slice (for example, it goes into the wait state before its time slice is complete), it can be rescheduled on the CPU for the remainder of the time slice. An epoch is complete when all processes have run for their time slice.

Priority values affect which process with remaining time in its time slice is chosen next for the CPU. As stated previously, Linux uses dynamic priorities for conventional process scheduling. Linux computes the dynamic priority as the sum of the time slice quantity and the amount of time left until the end of the time slice.

Real-time scheduling in Linux is performed by raising the priority of real-time processes and by tuning the operating system to make bounds on system overhead. If a real-time process is running, pre-emption should not allow a lower-priority process to take over the CPU ahead of any other real-time process.

Process scheduling in Linux is relatively straightforward. The epoch method ensures that all processes are run in a certain time period, but that more important processes are executed first.

- This algorithm is good for a variety of uses – interactive, batch mode – that do not stretch a system to the limit. In this environment, the Linux algorithm can run all the processes, avoid process starvation and even service real-time processes.
- If the number of processes is large, recalculating the process time slices before each epoch is quite inefficient. System responsiveness depends on the average time-slice duration of processes in the ready state. For systems under high loads, choosing this time-slice quantity

can be tricky and the time slice chosen by the Linux scheduling algorithm is often too large.

- Linux has a strategy of dynamically raising the priority for I/O-bound processes. This ensures a short response time for interactive processes and provides an aging strategy to avoid starvation. However, processes that wait for I/O but do not require user interaction also have their priority artificially boosted. Thus, if a system has many I/O-bound processes, all processes – even those with user interaction and little I/O – suffer.
- Real-time support is based on the fact that real-time processes are scheduled often and that all system latencies are predictable. These criteria are supported in Linux (providing the operating system is pre-emptively scheduled) but there are other issues. It is possible, for example, for a lower-priority process to block a real-time process. This can occur when the lower-priority process is running when the real-time process enters the ready state. This phenomenon is known as *priority inversion*. In addition, a real-time process could need a system service that is servicing the request of a lower-priority process. This problem is known as *hidden scheduling*. Linux allows both priority inversion and hidden scheduling and thus has weak support for real-time processes.

5.4 Scheduling in a Microkernel Architecture

Recall that a microkernel architecture is an attempt to minimize the size of kernel-level structures and to push as much functionality as possible to the user level. The question with regard to scheduling in microkernels is where to place the scheduler. Is process scheduling a kernel-level or a user-level function?

Placing scheduling at the user-level has a certain appeal. Scheduling policies can change more easily – even at a user's discretion. Systems can be customizable – users can set their own scheduling strategies and applications can alter how scheduling is done.

Placing scheduling functionality at the kernel level has a few advantages. Essentially, scheduling relies on kernel information. A scheduler must know, for instance, how long a process has been using the CPU and what the priorities of processes are. A scheduler must also access kernel-level structures, including process queues and PCBs, and handle kernel-level events, such as interrupts. Because a scheduler must access so many kernel-level objects, scheduling is typically a kernel-level function, even in a microkernel. Because the overhead of making lots of kernel requests is high, placing scheduling at the user level would hurt an implementation. Allowing a scheduler access to information is much faster than making many requests for the same information.

This means that scheduling is one of the basic functions of an operating system that is kept at the kernel level by a microkernel. Whatever the structure of an operating system – monolithic to microkernel – the scheduler is built into the kernel.

5.5 Scheduling in Symbian OS

Symbian OS is a mobile phone operating system that is intended to have the functionality of a general-purpose operating system. It can load arbitrary code and execute it at run time; it can interact with users through applications. At the same time, the operating system must support real-time functionality, especially where communication functions are concerned. This combination of requirements makes scheduling interesting.

Because of the real-time requirements, Symbian OS is implemented as a real-time operating system. It is built to run on multiple phone platforms, without specialized hardware, so the operating system is considered to be a soft real-time system. It needs enough real-time capabilities to run the protocols for mobile protocol stacks, such as GSM and 3G (not to mention future protocols). In fact, Symbian OS considers scheduling to be such a basic service that the nanokernel provides it.

The combination of general-purpose functionality with real-time system requirements means that the best choice for implementation is a system that uses a static, monotonic scheduling strategy, augmented by time slices. Static, monotonic scheduling is a simple strategy to use – it organizes processes with the shortest deadline first – and the introduction of time slices means that processes with the same deadline (or no deadline) can be assigned time slices and scheduling using a priority-scheduling scheme. There are 64 levels of priority in Symbian OS.

As we discussed before, a key to soft real-time performance is predictable execution time. If an operating system can predict how long a process will run, then a static, monotonic scheduling strategy will work, since it makes some big assumptions about run time. Predicting execution time is based on the process and several system characteristics. There are several important characteristics that must be predictable, including:

- *latency times*: an important benchmark is the latency of handling interrupts: the time from an interrupt to a user thread and from an interrupt to a kernel thread
- *the time to get information about processes*: for example, the time it takes to find the highest priority thread in the ready state
- *the time to move threads between queues and the CPU*: manipulating scheduling queues – for example, moving processes to and from the ready queue – must be bounded. This functionality is used all the time and it must have a bound on it or the system cannot predict performance.

Predicting these quantities is important and is reflected in the design of the scheduler. For example, in order for Symbian OS to predict the time for finding the highest-priority thread, the operating system uses 64 separate queues, one for each priority level. In addition, there is a 64-bit mask, where a bit being on in the mask indicates that there are processes in the corresponding queue. This means that to choose a process from a queue the operating system scans the mask and chooses the first process in the first available queue, instead of searching over a single queue with an unknown number of processes in it.

5.6 Summary

In this chapter, we have looked at the requirements of sharing a computer's CPU between processes. We began the chapter by outlining what we mean by 'sharing' and what criteria can be used to assess how good a sharing strategy is. We then examined several different strategies that are used to schedule a CPU to be used by multiple processes. Finally, we described three scheduling implementations, for Linux, general microkernels and Symbian OS.

In this chapter, we have examined how to build the 'illusion' of supporting multiple processes executing at the same time on a single processor. The next chapter examines how to continue this illusion by discussing how concurrently running processes can communicate with each other.

Exercises

1. We discussed pre-emptive and non-pre-emptive scheduling. List computing environments and state whether pre-emptive or non-pre-emptive scheduling would be best used. Give environments that cannot use either pre-emptive or non-pre-emptive scheduling.
2. Describe why non-pre-emptive scheduling should not be used in a soft real-time environment.
3. How should an I/O call be used by a non-pre-emptive scheduler?
4. Consider the following set of processes, listed with the time needed and scheduling priority requested. Assume that 1 is the highest priority and that the requests arrive in the order P_1 to P_5. Assume that a time slice is 2 time units.

Process	Time Needed	Priority
P_1	21	2
P_2	19	4
P_3	3	3
P_4	10	6
P_5	13	5

   a. Draw time bars to indicate the scheduling sequence for these processes under an FCFS, an SJF, a round-robin and a pre-emptive priority scheduling scheme.
   b. Compute the turnaround time for each process for each of the scheduling strategies.
   c. Compute the average waiting time for each process for each of the scheduling strategies.
   d. Which of the scheduling scenarios has the best turnaround time?
   e. Which of the scheduling scenarios has the least waiting time?

5. Suppose a new scheduling strategy, called *least processor time first* (LPTF), is invented. In an LPTF strategy, the next process chosen for the CPU is the one that has used the least processor time. Why

does this favor I/O-bound processes? Is it effective in eliminating starvation?

6. How would you tune the Linux scheduling strategy to better support real-time computing? Give at least three suggestions.

7. Suppose a new Linux strategy is invented that looks for higher-priority processes between the scheduling of each process. If one is waiting, it is scheduled for an extra time slice. Why does this method encourage starvation?

8. Should interrupts be disabled during a context switch? Describe how disabling and enabling affects a scheduling algorithm.

9. How does queue maintenance affect a scheduling algorithm? In other words, explain how careful placing of a process's PCB in a queue *after* removing the process from the CPU affects a scheduling algorithm.

6

Process Concurrency and Synchronization

As children, most people are taught to take turns. Taking turns is a way to share something between two or more people in an organized fashion so that everyone has time to use the object, but no one can exclusively take it as their own. Taking turns is a great way to share, but it is often not done correctly, especially with children. If turns are unfair or someone is seen as taking more than her turn, several bad things can happen. Usually, the result is a fight among the children sharing the object or damage to the object being shared. Often an adult must step in to monitor the situation.

When processes must share objects in a computer system, the same careful attention must be given to proper sharing. If an object – say a device or a portion of memory – is not shared correctly, several bad things can happen. The shared object could be corrupted or the processes involved could deadlock as they fight for access. In all cases, things are best handled when the operating system steps in and provides some kind of supervision.

This chapter deals with how to share objects in an operating system between processes. We have established in previous chapters that processes can be seen as executing concurrently, even when they share a single processor. We examine what it takes to get those concurrent processes to coordinate with respect to shared objects and communicate between processes within an operating environment. We also examine how to avoid pitfalls of sharing, specifically deadlocks.

6.1 Concepts and Models for Concurrency

In an operating system, multiple processes run virtually at the same time and share all the resources of a computer system. It is the nature of modern operating systems to support applications that work this way. Modern operating systems are built assuming that processes share memory space, devices and other resources with each other. Much of the time, dwelling in this shared world is easy and requires no special action. However, there are times when processes must cooperate to properly share something between them. This section introduces the concepts and ideas necessary to discuss sharing and cooperation.

Understanding the Environment

Let's begin by looking at the environment in which cooperating processes execute. A usual execution environment contains any number of cooperating sequential processes, which are running in parallel with each other. Each process is running sequential code within its process space. Even though other processes are also executing concurrently, each process is unaware of other actions taken by other processes.

For example, consider a process that reads data from a memory buffer that is generated by a second process. This could occur, for example, if one process is producing timing data and the other process is reading that data and displaying it. Consider, for example, the code below:

```
while (true)
  {
  while (timing_count == 0) ;  // do nothing
  timing_data = time_buffer[time_out];
  time_out = (time_out + 1) % buffer_size;
  timing_count --;
  display(timing_data);
  }
```

The consumer is waiting – doing nothing – until a variable named `timing_count` is greater than zero, which indicates how much timing data there is in the buffer to be read. This timing data is produced by a data producer that executes code similar to that below:

```
while (true)
  {
```

```
  while (timing_count == buffer_size) ; // do nothing
  timing_data = timer();
  time_buffer[time_in] = timing_data;
  time_in = (time_in + 1) % buffer_size;
  timing_count ++;
  }
```

These two code fragments run in parallel. Note that each process shares the `time_buffer`, which is filled and emptied of time data. Each process keeps its own idea of how much is in each buffer. Finally, note that the count `timing_count` is also shared between processes. The variables `time_out` and `time_in` are private variables and are meant to keep track on a circular basis. The shared `timing_count` variable is meant to indicate how much data is in the buffer that should be displayed.

It is easy to demonstrate how these code sections would work together well. For example, if a producer section is run before a consumer section, all objects are shared correctly, as in the sequence of code below (the time-data producer is in italics and indented):

```
  timing_data = timer();
  time_buffer[time_in] = timing_data;
  time_in = (time_in + 1) % buffer_size;
  timing_count ++;
timing_data = time_buffer[time_out];
time_out = (time_out + 1) % buffer_size;
timing_count --;
display(timing_data);
```

Even certain interleavings of the code work correctly:

```
timing_data = time_buffer[time_out];
time_out = (time_out + 1) % buffer_size;
  timing_data = timer();
  time_buffer[time_in] = timing_data;
  time_in = (time_in + 1) % buffer_size;
timing_count --;
  timing_count ++;
display(timing_data);
```

These are 'macro-style' interleavings. That is, they interleave entire statements, which themselves are comprised of instructions. Interleaving process execution ultimately happens at the instruction level. Consider what happens if we interleave the instructions that these statements comprise a bit differently. Let's assume that the execution of `timing_count ++` and `timing_count --` are implemented like this:

```
load register from timing_count location
add (or subtract) 1 to (or from) register
store register into timing_count location
```

Now consider the following interleaving of instructions:

```
  load register from timing_count location
  add 1 to register
load register from timing_count location
subtract 1 from register
  store register into timing_count location
store register into timing_count location
```

If the value of `timing_count` is 10 at the beginning of this sequence, then the producer sets `timing_count` to 11 and the consumer sets it to 9. In this case, both values are wrong; the correct result of this interleaving should leave `timing_count` at 10.

We can see, therefore, that certain interleavings of statements are correct and others leave corrupted data. Our goal is to derive ideas about parallel execution that allow us to ensure correct manipulation of data all the time.

The Goal: Serializability

It is clear that, without proper precautions, data manipulation can only be guaranteed to be correct when processes are not concurrent. That is, resource corruption cannot occur when only one process at a time is using the resource. This is a dilemma, however, because, as we saw in Chapter 5, running processes one at a time does not make sense for the performance of a system. The goal, then, is to make shared access to resources look as if it is done sequentially. This property is called *serializability*.

We must invent ways for processes to cooperate and be coordinated that allows concurrency, yet makes access to shared resources look like serial access. We start to do this by identifying the code sections that access shared resources. We call these code sections *critical sections*. We must coordinate the access these critical sections have to shared resources so as to adhere to these criteria:

- *mutual exclusion*: this is a guarantee that, when a process is executing inside a critical section, it is the only one in the critical section accessing the shared resource
- *no starvation*: if a process wants to enter its critical section and no other processes are in their critical sections, then it is eventually allowed to do so
- *bounded waiting*: there must be a limit to the number of times other processes can enter their critical sections between the time a process requests to enter its critical section and the time that request is granted.

In other words, we must ensure that our serialization mechanism is effective, with no starvation and no indefinite postponement.

While we consider ways to guarantee these criteria, we must continue to remind ourselves that statements in a program that make up a process's critical section are actually instructions. At some point, as we drill down to the machine language making up a program, we have to assume that something executes without interruption. (If any process can be interrupted at any time, we can say very little about guaranteeing the criteria above.) We assume that machine language instructions execute without interruption – *atomically*. This means that the execution of one instruction completes before the execution of another instruction from another process begins.

Synchronization of Two Processes

Let us start our consideration of process synchronization by considering the case of only two processes executing concurrently. We branch out from this view in the next section, but restricting ourselves to two processes allows us to examine the issues more closely.

If we are only looking at two processes, a first solution to sharing a resource might be to take turns by using a `turn` variable that indicates whose turn it is. When the `turn` variable has a process's ID, then it is that process's turn. The following code shows how this might work:

```
while (true)
  {
  while (turn != myID) ;

  // critical section
  turn = nextID;

  // whatever else needs to be done
  }
```

The process IDs of the two processes involved are stored in `myID` and `nextID`. This method ensures some of our criteria but not all of them. For only two processes, this method does indeed ensure mutual exclusion: the `turn` variable can only have one value at a time and changes only when the critical section is complete. There is also a bound on waiting: when a process is ready to enter its critical section, the other process can enter only once. However, the rule against no starvation is violated: if a process is ready to enter its critical section, and the other process is not, the current process can only enter if it is its turn. A process can starve another process simply by not taking its turn.

The method failed because we did not know enough information about processes. If we knew whether a process was ready to enter its critical section, we might be able to fix this. The following code shows a way around this:

```
while (true)
  {
  ready[myID] = true;
  while (ready[nextID]) ;

  // critical section

  ready[myID] = false;

  // whatever else needs to be done
  }
```

In this method, we establish two flags – an array of Booleans – that indicate whether a process is ready to enter its critical section. By setting a flag and checking the other process's flag, a process can implement the idea of taking turns while avoiding starvation.

This still does not satisfy all our criteria. It does indeed satisfy mutual exclusion and goes some way towards meeting the starvation requirement. However, it does not completely work: since the `ready` flags are set in separate statements, a second process could set its `ready` flag between the two statements before entering the critical section. That is, we could have:

```
ready[myID] = true;
  ready[nextID] = true;
while (ready[nextID]) ;
  while (ready[myID]) ;
```

Now both processes are stuck in their waiting loops and each process starves.

The correct solution lies in combining the ideas of both of the previous methods: take turns, but only if the other process is not ready. For this solution, we need both `turn` variables and `ready` arrays:

```
while (true)
  {
  ready[myID] = true;
  turn = nextID;
  while (ready[nextID] && turn == nextID) ;

  // critical section

  ready[myID] = false;

  // whatever else needs to be done
  }
```

Synchronization of Multiple Processes

We have seen a two-process solution; now we need to generalize to a method that works in a multiple-process environment. There are many methods of multiple-process synchronization; this topic remains fodder for many research projects and doctoral dissertations. Examples include the colored ticket algorithm, clock algorithms, and many unnamed mutual exclusion algorithms. For further reading, [Lamport 1987] is an excellent paper that surveys synchronization methods.

We outline the *bakery method* – sometimes called the grocery store or shop method. The bakery method is based on an algorithm used in bakeries and shops. In these situations, customers organize themselves by taking a number and waiting their turn until their number is called. The customer with the lowest number is the next to be served. We simulate this type of behavior:

```
do
  {
```

```
choosing[pid] = true;
number[pid] = max(number[0], ..., number[n-1]) + 1;
choosing[pid] = false;

for (i=0; i<num_processes; i++)
  {
  while (choosing[i]) ;   // do nothing
  while ( (number[i] != 0) &&
          ( (number[pid], pid) < (number[i], i) ) ) ;
  }

// critical section

number[pid] = 0;

// whatever
} while (true);
```

We start by picking a number – which we assume is more than all the other numbers chosen. However, we cannot guarantee that multiple processes do not choose the same number. Then we check all processes, waiting for each to finish choosing and for each to return their number if it is less. The notation `(number[i],i)` is meant to convey that either the number chosen is less or the process ID is less (if the number is the same). To understand that this works, consider that if a process P_i is in its critical section and P_j (where i is not the same as j) is waiting with an already-selected number then it must be true that `(number[i],i) < (number[j],j)`.

In this algorithm, we can see that mutual exclusion is adhered to by observing that when P_i is in its critical section, `number[i] != 0`. When a second process P_j wants to enter its critical section, then `(number[i],i) < (number[j],j)`. So the algorithm makes P_j wait until P_i is done and the statement `number[pid] = 0` is executed.

Notice that processes enter critical sections when they need to, on a first-come-first-served basis. This is enough to show that no starvation can occur and waiting is bounded.

6.2 Semaphores

As we saw in the previous sections, using algorithms to guarantee our three criteria – mutual exclusion, no starvation and bounded waiting – is complicated and clumsy. They also require a bit of overhead to use. The complexity of the algorithms has developed because the atomicity of operations cannot be ensured. For example, interleaving of statements can cause problems with the implementation. If the statements from one process could be executed atomically – without interleaving – we would be in much better shape.

We can get around these methods by the use of a tool called a *semaphore*. A semaphore is a data object – often a simple integer – that is supplied by the operating system and guaranteed to be manipulated atomically. Using a semaphore to make processes cooperate takes the place of using the methods from Section 6.1 for critical sections.

Semaphores are accessed through two operations: `wait()` and `signal()`. These two operations can be thought of in the following generic fashion:

```
wait(S)
  {
  while (S <= 0) ;
  S--;
  }

signal(S)
  {
  S++;
  }
```

These methods require an initial value for the semaphore `S`, which is usually considered to be the number of processes that can access a particular resource. These methods also assume that operations on `S` are atomic and cannot be interrupted. This includes arithmetic operations as well as testing.

To see how we can use semaphores, consider this code sequence:

```
while (true)
  {
  wait(CS);

  // critical section

  signal(CS);

  // whatever else needs to be done
  }
```

In this example, processes share a semaphore, `CS`, initialized to 1 (there can only be one process in a critical section at a time). As a process wants to enter its critical section, it waits for `CS` to be equal to 1, then decrements it, all in one uninterruptible action. When it is finished with its critical section, the process increments the semaphore in an atomic manner, thereby signaling to other processes that they may enter the critical section.

Consider the implementation of semaphores. The methods we outlined in Section 6.1 were all based on *busy waiting*. Busy waiting occurs when a process is waiting for something that it needs to check constantly. This type of waiting is extremely inefficient because it requires CPU time. This wastes CPU time that another process might be using. Instead, semaphores are implemented more like devices: waiting for a semaphore causes movement of the process from the running queue to the waiting queue. This means that a process is blocked while it waits for a semaphore to be available, but it does not consume CPU cycles. It is the use of the `signal()` operation by some other process that unblocks the waiting process and moves it to the ready queue again. The result is an efficient method of coordination that uses no CPU time and abides by our synchronization criteria.

Binary semaphores are a special case of semaphore. They have one of two values: 0, indicating no availability, or 1, indicating availability. This gives the user a means of indicating 'taken' and 'available' – or 'lock' and 'unlock' as we see in Section 6.3.

6.3 Locks, Monitors and Other Abstractions

For some people, using a semaphore is too 'close' to system operations. That is to say, semaphores are not a sufficiently abstract idea. Other, more abstract, concepts have been invented to hide the use of semaphores.

A *lock* is an abstraction that masks a semaphore. Locks are usually associated with data items, not sections of code. The operation of *binary locking* is usually done on a data item and guarantees that all manipulations on that data item by the owner of the lock are done atomically, without interruption. Trying to lock a locked data item usually causes the lock operation to block until the data item is unlocked. Locks can be of other types than binary and can sometimes allow certain operations to happen concurrently. For example, *read locks* are usually distinct from *write locks*. If a read lock is held on a data object, then only read operations – from any process – can proceed in parallel. Write locks guarantee mutual exclusion: the owner of a write is guaranteed atomicity with respect to the locked data item.

Locks can be implemented by semaphores. Binary semaphores are used, with the lock operation implemented by `wait()` and the unlock operation implemented with `signal()`.

A *critical region* is a programming language construct designed to ensure access to critical sections by focusing on shared variables. Critical regions require two types of special syntax, for variables and for critical sections of code. For example, if we were to design a concurrent queue, we might declare that queue as a concurrent C++ struct as follows:[1]

```
struct concurrentQueue
  {
  int queue[];
  int head, tail;
  }

time_buffer: shared struct concurrentQueue;
```

Notice the last line that declares the `time_buffer` variable to be a shared concurrent queue. We can specify a consumer that uses this shared queue as in the following code:

```
region time_buffer when (timing_count > 0)
  {
  timing_data = time_buffer[time_out];
  time_out = (time_out + 1) % buffer_size;
  timing_count --;
  display(timing_data);
  }
```

The code's syntax incorporates a guard condition – in our case this is `timing_count > 0` – and focuses the critical section of code on the shared resource. The intent of this kind of syntax is to guard against programmer errors associated with manipulating semaphores. This abstraction takes away the semaphore manipulation and focuses on syntax.

Another form of abstraction that hides semaphores is a *monitor*. A monitor is a programming language construct similar to a class or other object-oriented abstraction. In object-oriented languages, classes are an

[1] This and other examples of code are written in a fictitious enhancement to C++. There is no 'shared' keyword, for example, but I explain the syntax and it should be clear what the code is used for.

encapsulation of data definitions and the procedures that operate on those definitions. A monitor is an extended definition of a class where any use of a monitor's operation guarantees atomicity with respect to the data items defined in the monitor. Consider the following specification of a monitor:

```
public monitor concurrentQueue
  {
  int queue[];
  int head, tail;

  concurrentQueue() { // constructor code }

  void enqueue(int aQueueItem) { ... }
  int dequeue() { ... }

  int length() { ... }
  }
```

This example is certainly not that mysterious; you could replace the keyword 'monitor' with the keyword 'class' and have a class specification written in Java.

A monitor is implemented with semaphores and critical sections. Use of a monitor's methods implies locking the data items defined in the monitor's declaration first, then making the method call. The data items need to be unlocked just before the method returns. And locking, as we stated previously, can be implemented with semaphores. So again we have semaphores as the basis for higher-level abstractions.

The concepts we have just discussed – such as critical regions and monitors – were developed as experimental concepts by researchers trying to understand how best to program with concurrent processes in an operating system. Some of the first researchers in this field were Per Brinch Hansen and C.A.R. Hoare. These scientists invented new languages or augmented existing ones so that they could experiment with how these new constructs worked. Languages such as Concurrent Pascal represented existing languages (Pascal) that were augmented and Mesa and CSP (and later, Occam) were new languages influenced by their work.

6.4 The Dining Philosophers: A Classic Problem

A classic problem in concurrency has to do with five philosophers sitting around a table. These five philosophers have nothing to do except eat

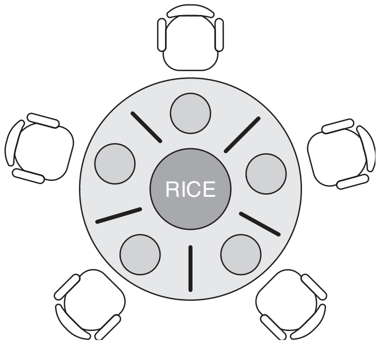


Figure 6.1 The philosophers' dining table

and think. They can think on their own, but to eat, they need a shared bowl of rice and two chopsticks. The table is built so that each of the five philosophers has a single chopstick to her left and right, shared by the philosophers to the left and right (see Figure 6.1).

The problem is that a philosopher cannot eat unless she has two chopsticks. This has several implications. If a philosopher is eating, then her neighbors cannot eat. If a philosopher picks up one chopstick, but her neighbor has also picked up one, then she cannot eat. The goal in this problem is to allow every philosopher to go through eat–think cycles without starvation.

The obvious solution is shown in the following code:

```
while (true)
  {
  think();
  take_chopstick(i);
  take_chopstick(i+1 % 5);
  eat();
  drop_chopstick(i);
  drop_chopstick(i+1 % 5);
  }
```

The call to `take_chopstick(i)` waits until the specified chopstick is available and then takes it. This is obvious … but wrong. There are several reasons why this obvious solution does not solve the problem. First, consider the example we just mentioned. If the first several statements were interleaved with each other, it is possible that `take_chopstick(i)` is interleaved with all the others before the next statement is executed. The result is deadlock, as all the philosophers wait for each other to give up a chopstick. Secondly, consider a scenario where philosophers are able to eat and take chopsticks during the time their neighbors are thinking. Then, while their neighbors wait, they put down the chopsticks, think and try to eat again. The result is that several philosophers starve. This is definitively not a desirable situation.

We can see starvation in another solution. Let's say that each philosopher, after thinking and taking her left chopstick, checks to see if the right chopstick is available. If it is, she takes the chopstick and eats. If it is not, she puts down the left chopstick, waits for a time and repeats the process. This repeats until a chopstick is available. Unfortunately, starvation is even more probable in this scenario. If the neighbors can eat and think faster than the philosopher's checking time, then the philosopher never gets to eat and starves.

The proper solution to this problem uses semaphores to define a critical section. To start with, we can define a semaphore – call it 'utensils' – that is used to enter a critical section. Before a philosopher can start acquiring chopsticks, she performs `wait(utensils)` and when she is done with the chopsticks, she performs `signal(utensils)`. This protects the critical section comprised of taking chopsticks, eating and replacing the chopsticks and ensures that no philosopher starves and that everyone eats. The act of eating is now serializable.

Unfortunately, this is a bad solution with regard to performance. With this solution, only one philosopher eats at a time, even though others are ready and could eat without bothering each other. The proper solution adds a semaphore for each chopstick. The chopsticks must still be checked and taken in a critical section, however. This ensures that starvation does not occur. A proper specification for this last version is given in Section 6.5.

There are several classic problems that have developed around process concurrency. The Dining Philosophers' problem is one of the most famous. However, there are others worth thinking about.

The *reader–writer problem* is a problem where a data object is shared among several processes. Some processes read from the data object and some write to the data object. The object of the system is to allow arbitrary reads and writes without corrupting the data.

The *producer–consumer problem* (otherwise known as the *bounded-buffer problem*) is a problem where a process fills a buffer with data and another process empties that same buffer. The buffer is bounded, that is, it can only contain a specific number of data items. We must guard the buffer so that consumers cannot read an empty buffer and producers cannot write to a full buffer.

6.5 An Example in Unix

Standard Unix supports semaphores among other concurrency constructs. Let's consider what a solution to the Dining Philosophers might look like on a Unix platform, Solaris. We start with the following main program:

```
#include <stdio.h>
#include <synch.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
#include <sys/time.h>

#define N        5
#define LEFT     (i-1)%N
#define RIGHT    (i+1)%N
#define THINKING 0
#define HUNGRY   1
#define EATING   2

/* NOTE the redefinitions of wait and signal -- with the same semantics.
 * And "sema_t" will serve as a semaphore.
 */

#define wait(S)      sema_wait(S)
#define signal(S)    sema_post(S)
#define semaphore    sema_t

semaphore mutex, s[N];
int state[N];

main ()
  {
  int phil[N], status;
  int i;
  struct timeval tp;

  // Set things up by seeding the random number generator

  gettimeofday(&tp,&tp);
  srand(tp.tv_sec);

  // Next we initialize the semaphore.

  status = sema_init(&mutex, 1, USYNC_PROCESS, NULL);
  for (i=0; i<5; i++) sema_init(&s[i], 1, USYNC_PROCESS, NULL);
// Now, we create N forks by using a for loop.

for (i=0; i<=N-1;  i++)
  {
  if ( (phil[i] = fork()) == 0 )
    {
    philosopher(i);
    break;
    }
  }

 // Finally, we pause to wait for termination of all forks.

if (i == 5)
  {
  for (i=0; i<5; i++) wait(&status);
  }
}
```

Semaphores are defined by the data type of `sema_t`. The wait and signal functions are implemented by `sema_wait()` and `sema_post()`. We use two types of semaphores (as outlined in Section 6.4): one to guard a critical section and a set to govern the taking of the chopsticks. The function `sema_init()` is used to give each semaphore an initial value. Each one is a binary semaphore.

Unix implements the creation of processes through the `fork()` system call. The `fork()` call clones the current process's PCB into two PCBs that are virtually identical. The only difference between them is that the call to `fork()` in the parent returns the process ID for the child process; in the child, it returns 0. So the code above creates five processes with the following fragment.

The call to the `philosopher()` function occurs only if the process is a child, when the `fork()` call returns 0. The following code defines `philosopher()`. The definition follows what we outlined in the previous section: a philosopher thinks, picks up chopsticks, eats, and puts the chopsticks back down. Eating and thinking amount to sleeping for random periods of time.

```
/* THINK and EAT -- and sleep random amounts of time (thinking and
 * eating is hard business).
 */
```

```
void think(int i)
  {
  printf("Philosopher #%d is thinking...\n", i);
  sleep(rand() / 6553);
  }

void eat(int i)
  {
  printf("Philosopher #%d is eating...\n", i);
  sleep(rand() / 6553);
  }

// The REAL philosophy business.  Think and eat forever.

void philosopher (int i)
  {
  int times=0;

  while (1)
    {
    think(i);
    take_chopsticks(i);
    eat(i);
    put_chopsticks(i);
    }
  }
```

The real meat of the solution is the implementation of `take_chopsticks()` and `put_chopsticks()`:

```
// Test to see if neighbors are NOT eating.

void test (int i)
  {
  if ( (state[i] == HUNGRY) &&
       (state[i-1%N] != EATING) &&
       (state[i+1%N] != EATING) )
    {
    state[i] = EATING;
    signal(&s[i]);
    }
  }

/* Take the forks correctly -- if neighbors are NOT eating. Note the
 * "down" call as the last line.
 */
void take_forks (int i)
  {
  wait(&mutex);
  state[i] = HUNGRY;
  test(i);
  signal(&mutex);
  wait(&s[i]);
  }

/* Put forks down correctly. Change the state to thinking. And enable
 * the neighbors if, by replacing forks, they can now eat.
 */

void put_forks (int i)
  {
   wait(&mutex);
   state[i] = THINKING;
   test(i-1%N);
   test(i+1%N);
   signal(&mutex);
  }
```

We implement a 'state' of a philosopher as a way to indicate a philosopher's desire. A philosopher can be `EATING`, `HUNGRY` or `THINKING`. `HUNGRY` is a desire to be `EATING` (naturally). The procedure `test()` is very important. If the philosopher to the right is not `EATING`, the current philosopher is `HUNGRY`, and if the philosopher to the left is not `EATING`, then the current philosopher may eat. The semaphores make the entire solution work.

6.6 Concurrency in Symbian OS

As we have stated in previous chapters, Symbian OS supports concurrency between processes. We have also seen how the Symbian OS kernel is supported by the Symbian OS nanokernel. In addition, the Symbian OS architecture is essentially that of a microkernel. Therefore, we can expect synchronization primitives to be implemented in the kernel.

For Symbian OS, however, this is not as simple as it may seem. Because of nanokernel support, there are multiple kinds of semaphores, implemented at both levels in the kernel.

The most primitive objects are in the nanokernel. The nanokernel's support for synchronization takes the form of two types of objects: *mutexes* and *semaphores*. A mutex is essentially a binary semaphore: it has only two states and is designed to implement mutual exclusion between two processes. A semaphore is a more general form of a mutex; it can hold values greater than 1, allowing mutual exclusion between multiple processes. Both blocking and nonblocking mutexes and semaphores are supported. The recommended way of using nanothread synchronization is through the `NKern` class, which allows blocking calls. The `FMWait()` and `FMSignal()` methods implement blocking synchronization for mutexes; `FSWait()` and `FSSignal()` implement such functionality for semaphores. Nonblocking use of these synchronization objects is provided through other classes; nanokernel mutexes are implemented in the `NFastMutex` class and semaphores are implemented in the `NFastSemaphore` class. When access is nonblocking, only one process may acquire access through a mutex and all others requesting access (say, through a `wait()` call) are rejected, but not forced to wait.

Waiting is expensive to implement and the nanokernel is designed to be as fast as possible. However, using nonblocking synchronization means that the kernel needs to be locked as the synchronization object is checked. It also means that if a process wants to implement waiting with nanokernel primitives, it must implement its own wait cycle. This means that nonblocking calls are expensive as well. The safest route is to let the kernel handle waiting.

Kernel objects in Symbian OS are built on top of nanokernel objects. Thus, the kernel has analogous synchronization primitives: mutexes and semaphores. Kernel mutexes are binary semaphores and implement some of the semaphore properties that nanokernel mutexes do not. For example, the versions of `wait()` and `signal()` that are implemented for kernel mutexes allow for blocking and queuing: multiple processes may call `wait()` on a mutex and they block while they wait their turn. It is possible to hold several kernel mutexes simultaneously. Counting semaphores are also implemented by the Symbian OS kernel. As with mutexes, these semaphores are blocking and processes may hold multiple semaphores at once. Mutexes in the kernel are implemented by the `RMutex` class and semaphores by the `RSemaphore` class.

There is an interesting issue that applies to synchronization primitives in Symbian OS: process priority. Process priority and mutexes provide an interesting dilemma. Symbian OS has the property that if processes with different priorities are waiting for a mutex, then the process with the highest priority should be next to acquire the mutex when it is released. However, if a lower-priority process holds the mutex, then it can delay a higher-priority process. This is quite undesirable and the designers of Symbian OS have installed some mechanisms to keep it from happening. First, mutexes are not obtained until the last possible moment, to give other processes as much time to get in the waiting queue as possible. Secondly, Symbian OS uses *priority inheritance*. In the case of the low-priority process holding a mutex, the operating system raises the priority of the low-priority process to that of the highest process waiting for the mutex. This is to ensure that no other process, whose priority is higher than the low-priority process, would be running before it and postpone its releasing of the mutex sooner. Finally, there are special queues for processes that are suspended while waiting for a mutex. The operating system does not give mutexes to suspended processes because the mutex might be acquired for some undetermined time.

6.7 Interprocess Communication

One can certainly consider the use of synchronization primitives as a form of communication between processes. Processes that are synchronized over semaphores do indeed communicate the need for mutual exclusion. Often, however, more information needs to be exchanged between processes and therefore a more complicated set of semantics is required. Interprocess communication (IPC) builds on the ideas developed for process synchronization but adds concepts of data transfer and more complex exchange semantics.

Concepts

Cooperating processes can share information between them in several ways. One of the more obvious ways is by expanding the ideas of semaphores into full-blown shared-memory environments. If a kernel can implement shared objects such as semaphores, then it surely can expand to accommodate other kinds of shared-memory models. The idea of a shared-memory environment requires the kernel to provide memory resources, implemented as variables or buffers inside applications, to processes upon request. Multiple processes can request the same memory area and any changes to that memory affects all processes.

Using shared memory, however, is a lot like using global variables in a program. Shared memory must be used carefully; effects are immediate and parallel usage must be synchronized. There are other, more abstract, ways to communicate between processes. These other ways have synchronization built-in and do not require sharing memory.

Interprocess communication is best provided through the use of message passing. As with all ways of exchanging information, message passing requires a sender and a receiver. The way that they work together to exchange a message results in two models for passing information between concurrent units: the *mail model* and the *phone model*. The difference between these two models lies in whether or not the receiving unit needs to attend to the message before the sending unit may proceed.

The mail model

Information is sent by process P_1 to process P_2 and placed in a mailbox. Process P_1 may then proceed with its execution and process P_2 may come and retrieve the message at a later time. If more than one message is sent to the same mailbox, the messages are usually queued up within the mailbox so that process P_2 may successively retrieve messages until the mailbox is empty.

There are several ways that the mailbox might be identified, providing different versions of the mail model. These are distinguished by the way in which communication takes place. The *many-to-one version* is analogous to the way a typical post office mailbox operates, with messages arriving from any of a number of processes, but only destined for one specific process. Therefore, the sender specifies the receiver of the message, but the receiver retrieves messages without needing to specify the sender.

The *one-to-one version* accepts messages from one sender. Here the sender must not only specify the receiver, but when the receiver retrieves the message, it must specify the identity of the sender as well. A given mailbox is then identified with both the sending and the receiving processes. This is similar to a mailbox used to pass information from a boss to a secretary where all messages come from the same sender and all go to the same receiver.

The *many-to-many version* accepts messages from many processes and these messages may be retrieved by many processes. A sending process therefore places the message in the mailbox without specifying who the receiver is to be. The next process to retrieve from that mailbox is the receiving process. This is similar to a mailbox in an office with many bosses and many secretaries where a boss puts a job to be done in a mailbox and the next available secretary retrieves the message from the box and does the job specified.

The phone model

Under the mail model, the sender simply sends the message and does not wait for message receipt. The second model for passing information, the phone model, requires that the sending unit wait for the receiving unit to accept the message before proceeding. This is analogous to placing a phone call where the caller must wait for the person called to respond before the message can be sent. The phone model is also known as the rendezvous model, where two people meet together at a prearranged location to pass information. By its nature, the phone model also synchronizes the two processes since they must wait to make a simultaneous contact for the message to be sent.

There are two forms of the phone model, each with different views of waiting. In the first form, the sender waits only for notification from the receiver that the message has been received and, upon this notification, both processes continue with their execution. In the second form, the sender waits for both message receipt and message processing. This second version of the phone model is similar to a procedure call; the caller calls the procedure, sending parameters, and waits until this procedure returns, possibly with modified parameters. The analogy is so strong, in fact, that this second form of the phone model is typically referred to as a *remote procedure call*.

As with the mail model, the phone model might be one-to-many, one-to-one, or many-to-many.

Sockets

Sockets were invented by the designers of Berkeley Unix and were first used as a way to access network protocols. In the Berkeley terminology, a socket is an 'endpoint for communication'. By itself, as an endpoint, a socket is not very useful. But when connected to another socket on another computer, the pair become a communication channel that uses a protocol to transfer data. You can think of sockets as two ends of a conversation and the protocol as the translator.

Sockets require both a client and a server. The client connects to its end of the socket and makes a request to the server for connection. The server either replies with no connection or connects to its end and replies positively. Then data is exchanged across the socket.

The beauty of the socket model is in its abstractness and its translation abilities. The abstractness of the model can be seen in how it is used: each side simply writes data to and reads data from a socket as if it was any other local I/O device. Each side really does not know (or care) how the other side reads or processes the data. In fact, the socket may implement translation of data, again without each side knowing (or caring). The translation is implemented by the operating system and occurs as the data is transferred between the endpoints. These translations may be as simple as little-endian to big-endian or as complicated as using the Bluetooth protocol.

Sockets have certain properties, chosen based on how they are used. A socket is either *connected* or *connectionless*. A connected socket maintains a *virtual connection* between the two endpoints. This means that address information for the remote endpoint needs to be given only once, and that each access to the connection can be done without specifying this information again. A connectionless socket forces the application to specify the remote endpoint information each time it is used and has no virtual connection. A connected socket is easier to use and is more reliable yet requires higher overhead in its implementation. Connected sockets use mechanisms to ensure that data arrives at the remote endpoint in the exact order they were sent and that they arrive error-free or do not arrive at all. Connectionless sockets make no such guarantees about data arrival or reliability.

Connected sockets are implemented with *streams*. A stream is a logical connection between two endpoints that implements the following properties:

- *reliability*: with a stream, data is delivered accurately (as they were sent) without error or it is not delivered at all; if there is no data delivery, this is detected and the socket owner is notified
- *error control*: errors are detected automatically and the remote endpoint is usually asked to retransmit the data packet that had the error; maintaining error control usually involves using checksums on data packets and forcing remote endpoints to acknowledge when they receive packets
- *ordered delivery*: data that flows between two endpoints can be broken up and sent as fragments; a stream makes sure those fragments arrive at their destination in the order in which they were sent and that the larger data packets are reassembled correctly.

The reliability of connected sockets comes at a price. There are more protocol layers involved and hence more protocol overhead. There is more communication between endpoints and hence more data traffic.

Remote Procedure Calls

Remote procedure calls (RPCs) describe a very commonly used set of IPC semantics. Using the phone model of IPC, if we force the sender to identify the receiver, and block the sender until the receiver is done processing the message that was sent, we then have semantics that mirror those of procedures in a programming language. The act of sending is much like the act of calling a procedure, except that the procedure is on a remote process.

To complete the analogy, we need a few more concepts. When a procedure call is made, data is transferred to the called procedure in the form of parameters and passed back to the caller in the form of a return value. We can use these ideas for RPC: the RPC call transfers data from sender to receiver and the receiver can send data back through a return value abstraction. These facilities need the same 'translation' abstraction as sockets: the movement of data between sender and receiver assumes that the receiver can read what the sender has written. Issues of endianness of the architecture or error control are to be worked into the implementation of RPC. All the sender has to worry about is calling and returning.

RPC mechanisms are common in networked environments. The abstraction is similar to the socket abstraction, except that data does not flow back and forth arbitrarily. Data is sent in a single message and received when the remote procedure is done. RPC mechanisms are useful in situations where short messages must be exchanged but control is required. RPC is analogous to the stream mechanism.

IPC in Symbian OS

IPC is central to the implementation of Symbian OS. Since it has a microkernel design, much of the operating system's functionality is pushed into servers running at the user level. These servers communicate with each other, with the kernel and with user applications on a socket-based, client–server-oriented basis.

User-level objects in Symbian OS use a socket-based system to communicate. A user-level server is an active object (remember active objects from Chapter 4?) that waits for connections with requests and services them. Data is exchanged through the sockets via objects from the `RMessage2` class. Remote procedure calls are not used in Symbian OS.

6.8 Managing Deadlocks

When processes wait for each other, either directly or in some circular manner, we call the situation a *deadlock*. Consider a simple situation of a bank transfer. Let's say that two processes – say, two people at two automated teller machines – desire to transfer funds between the same two accounts. The processes that they go through are shown in Figure 6.2.

In the simple scenario where Process A executes its first statement, followed by Process B executing its first statement, deadlock ensues when Process A executes its second statement and Process B executes its second statement. This is because Process A is waiting for Process B, which is waiting for Process A.

For a deadlock situation to occur, four conditions must be present:

- at least one resource must be acquired for exclusive use by a process
- that process must be waiting to acquire another resource in the system
- a circular waiting set must exist, where each process is dependent on resources held by another process
- pre-emption cannot be allowed to free resources.

Deadlock situations require that the operating system detects and recovers from deadlocks. Deadlock detection is a matter of analyzing the relationships between processes. The operating system should maintain a table of structure resource requests. When a process requests access to a resource that is allocated to another process, an entry in the structure should indicate the waiting relationship. The result is a graph

Process A

```
lock (account A)
lock (account B)
decrement $100 from account A
increment account B by $100
unlock (account B)
unlock (account A)
```

Process B

```
lock (account B)
lock (account A)
decrement $100 from account B
increment account A by $100
unlock (account A)
unlock (account B)
```

Figure 6.2 Process definition for transferring money

of relationships between processes. This graph needs to be checked periodically for cycles. When a cycle exists, a deadlock situation has occurred.

Recovering from a deadlock situation can be a difficult operation. There are two ways to break a deadlock: process termination and resource pre-emption. To break a deadlock by process termination, the operating system needs to terminate one or more processes that are involved in the cycle detected in the resource allocation graph. Terminating all processes is a simple solution; finding and terminating only one crucial process can be difficult. The single-process approach is a decision based on many factors, including the priority of the process, the length of time the process has been executing, how many other resources are used by the process, and how many resources are still needed by the process.

To break a deadlock by resource pre-emption, the operating system must choose the resource and the process to pre-empt. Sometimes this choice is simple: a single resource may be the logjam and it may be obvious which process has that resource. However, there must be computable ways to make this choice. In addition, once the deadlock is broken, it may occur again and we must somehow ensure that the same process accessing the critical resource is not always chosen again (that would cause a starvation situation). Another, more gentle, approach is to *rollback* a process to a safe state. The process under examination is not terminated but reset to a state where the critical resource can be allocated to another process. Most of the time this actually means restarting the process, because determining a safe state usually is not possible.

6.9 Summary

This chapter has been devoted to issues surrounding the concurrency of processes in an operating system. We first introduced what happens when process interleave their statements and instructions. We then described the goal of serializability and ways that we could algorithmically synchronize processes to share resources properly. We introduced semaphores as a way to make synchronization easier. We developed other abstractions that built on and expanded semaphores. We then worked through the Dining Philosophers' Problem and demonstrated semaphores in Linux. We then discussed concurrency in Symbian OS. We introduced interprocess communication via message passing, sockets, and remote procedure calls. We finished the chapter by discussing ways to manage deadlocks.

Exercises

1. We discussed busy waiting as a special case of waiting in an operating system. What other kinds of waiting can go on in an operating system?
2. How is it possible to execute only half of a statement before a context switch is made?
3. We stated that making processes be truly serial – executing one after the other – would have a bad effect on performance. Does serializable access have the same performance hit? Explain your answer.
4. Prove that the following algorithm (the final solution to synchronizing two processes shown in Section 6.1) does indeed adhere to the three criteria of mutual exclusion, no starvation and bounded waiting.

```
while (true)
  {
  ready[myID] = true;
  turn = nextID;
  while (ready[nextID] && turn == nextID) ;

  // critical section

  ready[myID] = false;

  // whatever else needs to be done
  }
```

5. Show that if the manipulation of semaphores through `wait()` and `signal()` were not atomic, that mutual exclusion may be violated.
6. Should interrupts be disabled during the manipulation of semaphores? Explain.
7. The following code shows the critical region from the discussion about locks in Section 6.3. Rewrite it using semaphores.

```
region time_buffer when (timing_count > 0)
  {
```

```
timing_data = time_buffer[time_out];
time_out = (time_out + 1) % buffer_size;
timing_count --;
display(timing_data);
}
```

8. Implementation of monitors can restrict the way semaphores are obtained and released. Explain why a `signal()` call must be the last call for a monitor implementation.
9. Explain why a mutex is necessary in Symbian OS. Would a semaphore with a value of 1 also work?
10. Explain why the two-tier implementations of mutexes and semaphores (in the nanokernel and the kernel) is necessary in Symbian OS.
11. Are sockets based on the mail model or the phone model of IPC? Explain your answer.
12. Symbian OS does not implement remote procedure calls. Can RPC behavior be implemented with sockets? Explain.
13. Why does an operating system need multiple types of locks?

7
Memory Management

In the last several chapters, we have discussed how the CPU can be shared as a resource among processes. By proper scheduling and using concurrency, we can use the CPU more efficiently and increase the performance of the operating system. There are other resources in a computer system that also require sharing; after the CPU, a computer's memory is one of the most crucial. Proper sharing of memory also affects an operating system's efficiency and performance.

In this chapter, we discuss memory management. We develop the background and concepts necessary for this discussion and discuss management techniques. Many of these management concepts apply to desktop computers and servers, but some do not work with handheld units and smartphones. So we spend some time discussing systems that do not use all memory-management schemes. We use Symbian OS as an example of smartphone memory management.

Before we get started, the type of memory we are concerned with should be made clear. We are not concerned with what would normally be called *secondary storage*, such as hard disk space. Neither are we concerned with fast, on-chip storage, such as registers or caches. We are concerned with memory used for execution of programs – which could be main memory connected by bus to the CPU or RAM storage. The main qualifier is that the memory be used for program execution.

7.1 Introduction and Background

Like the CPU in a computer, memory is a resource that every process in a system must use. Like context switching on a CPU, proper sharing of memory by processes affects the entire computer system's performance.

Consider a scenario where a context switch means clearing memory and initializing it with the incoming process's data. This scenario would have much overhead built into it: in addition to a context switch (already a costly procedure), this scenario would have an operating system taking the time to save the execution environment for the process, wipe out memory, and pull in the memory image for the new process. The memory images – from the previous process and the incoming process – would have to be either saved or restored from a backing store – probably a hard disk. Hard disks are slow and I/O time would become a bottleneck.

Clearly, memory cannot be used exclusively by one process at a time. It must be shared. Sharing memory – without constant movement of memory blocks – means that multiple programs (we referred to these as processes in Chapter 4) occupy memory at the same time. Further, this implies that programs might be in arbitrary locations in memory – and probably not the same locations each time a process is brought onto the CPU to execute. This presents a tricky situation. Each program cannot know where it will be placed in memory and therefore is written believing it alone is using that memory. On top of all this, we must also be able to structure the environment so processes cannot trespass on each other's memory areas.

So our situation is complex: processes must share memory, but cannot know ahead of time what memory they will be using. Processes must believe they have all memory to use, but in reality are cordoned off into memory sections that cannot stray into each other. Processes must read or write data using locations they cannot know ahead of time. This is indeed a situation in need of some simplifying.

From Source Code to Memory

A process takes many forms as it moves from textual source code to a binary, executing memory image. Consider the steps toward execution as they are pictured in Figure 7.1. There are several stages in this process where data and instructions can be bound to memory addresses.

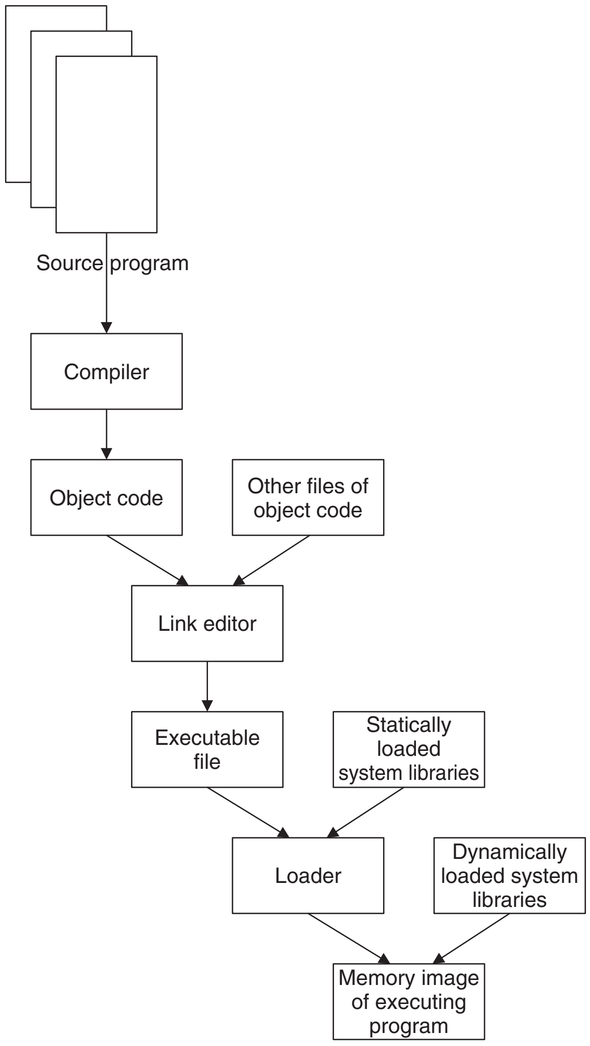

Source program
Compiler
Object code
Other files of object code
Link editor
Executable file
Statically loaded system libraries
Loader
Dynamically loaded system libraries
Memory image of executing program

Figure 7.1 From source code to executing program

Life for a process begins as a source program written in a programming language. The source code usually goes through a compiler to be translated into machine language for execution. Sometimes programs are translated directly into the form that is executed, but it is most likely that executing programs are built from several different modules.

Program modules represent pieces of programs that are built individually and then combined to form the final executing unit. These other object modules are built by the programmer or contributed from other sources.

This compiler stage is one place where components of a process can be bound to memory addresses. *Absolute binding* is the only type of address binding possible at compile time. Address references in the machine code can only be bound to actual addresses if the programmer knows the addresses at compile time. This is a situation that almost never happens now, but could happen for older operating systems. In early version of MS-DOS, for example, when single programs ran to completion without context switching, the beginning address for memory references was known and could be part of the compilation process – no memory sharing was going on. Note that if the starting address of a program in memory changes, then absolutely bound code must be recompiled.

Whether there is one module or many, everything must be combined for loading. This is done by the link editor. The link editor combines all the modules together into a single image. This image is composed of program modules only; no system libraries have been loaded at this time. System libraries are combined as needed by the loader.

While absolute binding is possible at load time, *relocatable binding* is most often used. When the programmer does not know at compile time where the program will start in memory, code is generated in such a way that it can be relocated easily. This can affect how programs are written as well as how the code is generated. Assembly code written for relocatable execution cannot reference absolute addresses. For example, the assembler for the SPARC architecture abides by this rule by forcing programs to use only labels (not even relative offsets) when referring to program code addresses or data locations.

Code is relocated and bound at execution time. At this stage, system libraries can be loaded into memory and their addresses correctly assigned. Note that there are two types of assignments going on here. Program code is relocatable and bound when a binary image is loaded into memory. In addition, libraries are loaded (if needed: they may already be in memory) and their address references are correctly bound within the program code. These two bindings represent *execution-time binding*.

Execution-time binding is the most flexible type of address binding. As processes are context-switched, the program code moves in and out of memory and may change locations often. In addition, library code becomes unused as processes are context-switched and may be removed, only to be loaded into different memory locations as they are needed. All this chaotic activity requires flexible execution-time binding.

Determining Module Dependencies

It is not obvious from running software what modules it depends on. You can determine this using an analysis program. On Solaris and Linux, the `ldd` command helps with this.

```
ldd /usr/bin/ls
```

For example, running the command above on a Solaris system gives the following output, showing three library dependencies:

```
libc.so.1 =>     /usr/lib/libc.so.1
libdl.so.1 =>    /usr/lib/libdl.so.1
/usr/platform/SUNW,Ultra-4/lib/libc_psr.so.1
```

On Microsoft Windows, you need third-party software, but you can list dependencies. For example, the screenshot in Figure 7.2 shows the dependencies for a program called `depends.exe`.

Logical and Physical Addressing

Issues of address binding lead us to the difference between *logical* and *physical addresses*. In a fully shared memory, where programs may

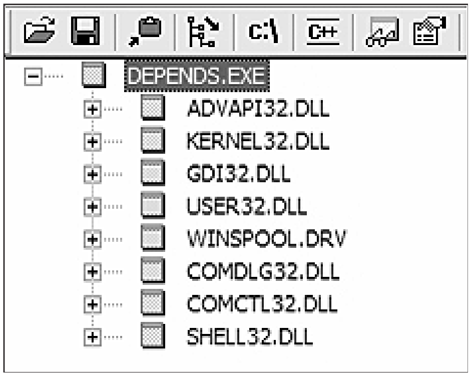

DEPENDS.EXE
ADVAPI32.DLL
KERNEL32.DLL
GDI32.DLL
USER32.DLL
WINSPOOL.DRV
COMDLG32.DLL
COMCTL32.DLL
SHELL32.DLL

Figure 7.2 Dependencies for a program

be executing from several different locations in memory during their execution, there are two types of memory addresses. Logical addresses are used by programmers in code and issued by the executing program during execution. Physical addresses are the actual addresses of real memory words. Logical addresses represent the program's concept of code and data. Physical addresses represent the actual address – more than likely relocated from where the program thinks it is – of that program code and data.

The translation between logical and physical addresses is something the operating system does, assisted by computer hardware. Only the operating system knows, at any specific moment in time, where code and data are located in memory. This means that executing code must make memory requests (for reading or writing) using logical addresses and the operating system translates them into physical addresses.

Memory-Management Units

The operating system's job is made a lot easier by a *memory-management unit* (MMU), a special hardware processor whose job it is to help the operating system manage memory. One of its functions is to translate between logical and physical addresses. This is done by the operating system setting a *relocation register* in the MMU and funneling all address references through MMU translation using this relocation register. The relocation register contains the first address in memory where the process's memory space is placed. The MMU uses this scheme by adding the value of the relocation register to memory references that go through it. Figure 7.3 depicts this process.

It is important to realize that the process that is executing knows only logical addresses and therefore uses only logical addresses. All references by the process are made to logical addresses. It is the operating system working with the MMU that translates the references correctly. We call the address space that the executing program references *logical address space*. The address space of real memory is called *physical address space*.

The truth is that memory management is a complicated process. Here we have just started to dig into what memory management is; we will add to it in the rest of this chapter. In the midst of all this complexity, MMUs are essential components.

MMUs are basic processors with memory for operating system tables and circuitry for fast searching and address computation. They do extensive bit-manipulation on addresses and lots of offset computation. They

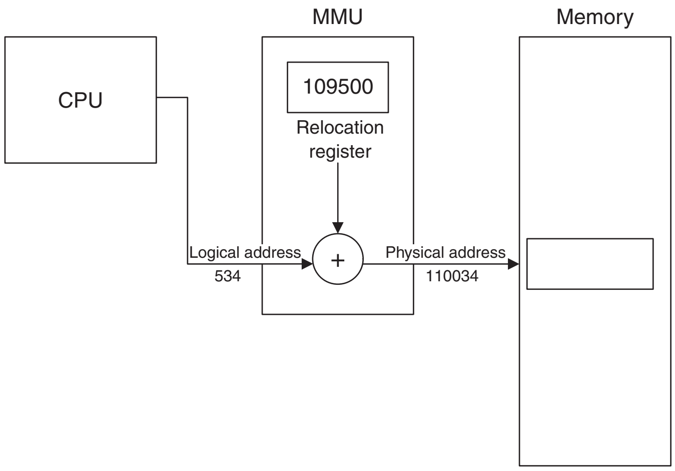


Figure 7.3 Translating a logical address to a physical address

do not have to be very powerful with respect to computation but they do have to be fast. Some of their memory is very fast (for example, translation look-aside buffers).

MMUs are so essential to some operating systems that these operating systems are not implemented on systems without an MMU. Microsoft Windows CE and Linux, for example, assume the presence of an MMU.

Dynamic Loading and Linking

It is tempting to see the amount of memory that modern desktop computers use and not worry about how much memory is used by an application. However, reducing the memory required for programs is still a rewarded endeavor, especially on smaller devices. Dynamic loading and linking are techniques that can be used to reduce memory requirements.

Dynamic loading can be used to distribute load costs and to eliminate memory waste. When using dynamic loading, a process only loads the portions of a program that it needs into memory. Often a program is broken up by function definitions, but it can also be in larger units – such as classes or groups of code. When the operating system starts a program that uses dynamic loading, only the main program is loaded and executed. The main program loads classes or functions before using them. The advantages here are that code that is never called is not loaded into memory. In addition, the operating system does not have to go into kernel mode. Dynamic loading is almost always a technique that user programs implement.

Using *overlays* is also a technique to save memory or to work in memory-restricted environments. As with dynamic loading, a program that uses an overlay method is broken up into pieces. These pieces are loaded into memory by the program itself in such a way that they overlay the memory of the currently executing program. The overlaid code occupies the same memory space as the program code that loaded. This, in effect, rewrites the program's code. The newly overlaid code is then executed.

As with dynamic loading, overlaying code requires no help from the operating system. No translation of addressing is required. The space taken up by the executing code is constant – the overlaid code takes the same space as the code it replaces. However, this method suffers from the performance delays of I/O. It was used often for older operating systems, in the days when 640 KB of memory was the norm for personal computers. It was not uncommon to be using an application, only to have the application freeze up while the overlay was read and installed.

A variation of dynamic loading is called *dynamic linking*. Linking is the action of joining modules of compiled code – including those supplied as libraries by the operating system – together with user object modules so that a complete executable can be run. Some systems employ *static linking*, which combines the libraries with user code before execution time. Dynamic linking links system libraries with user code when they are needed during execution and allows the system libraries to be linked from memory where they are loaded. If a library is already in memory from linking with another application, then it stays in memory where it is and is linked from that location. When dynamic linking is used, something has to alert the operating system to perform the dynamic library link. A small bit of code – called a *stub* – is statically linked in place of the actual library implementation of the system call. This code stub performs the dynamic link and replaces itself with the address of the dynamically loaded implementation.

There are several advantages to this scheme. First, libraries are not loaded until they are needed. Secondly, loading cost is distributed throughout the life of the executing application. Thirdly, libraries are loaded only once and it is possible that there is no load cost for many applications (especially for a commonly used library). Finally, all these advantages mean that program size will be smaller. Since libraries are not statically linked, programs carry fewer bytes of code when they are loaded for execution. When Sun Microsystems went from using statically linked code in its operating system to dynamically linked code, the size of its executable programs dropped by 60%.

Modern operating systems use dynamic linking as a way to minimize memory space used by applications. With the copious amounts of memory that come with computers, overlays and dynamic loading mechanisms are not needed.

7.2 Swapping and Paging

As shown at the end of Figure 7.1, a process's code must be in memory for the computer to execute it. However, every process that has been in the running state also needs code in memory. This means that processes must share memory in the same way as they share the CPU.

Swapping Memory Areas

One obvious way to do this that we have already discussed would be to store the image currently in memory, erase memory, and move the memory image of the new process into memory before execution. This sequence is called *swapping* and is at the heart of how processes actually share memory. Swapping requires some kind of storage in which to store the memory images between context switches. Typically, this is done using fast hard disk space.

A process that is swapped out to a storage area (such as a hard disk) by this method can sometimes be swapped back from that storage area to the same spot in memory. This can be advantageous, especially for certain methods of address binding. If absolute binding is used, set up at compile time, then a process's memory image is required to go in exactly the same memory location – dictated by the compilation process – each time. However, if relocatable binding is used, then a process can go anywhere in memory, because physical addresses are always calculated from logical addresses for each memory reference.

Notice that with swapping time added, context switching becomes a very expensive activity. Switching time is composed of process switching – the movement of PCBs through queue data structures – and the movement of memory areas.

Let's say that a process requires 2 MB of memory and that the disk drive used for storage has a transfer rate of 10 MB per second. The transfer rate of this memory swap is equal to

$$2\,\text{MB}/10\,\text{MB per second} = 1/5\,\text{second} = 200\,\text{milliseconds}$$

If we assume a generic disk drive with an average latency of 9 milliseconds, our swap time is 208 milliseconds for one swap. Since there are two swaps – one out and one in – the swap time is 416 milliseconds. This is in addition to other costs of process switching.

Obviously, we want to reduce swap time as much as possible. One way to do this is to focus on the storage medium. If we could increase the transfer rate of memory data, we could lower the cost, as measured by the amount of time needed to perform a context switch. Using solid-state memory is an option: compact flash memory can transfer as fast as 66 MB per second and Secure Digital cards can transfer data up to 133 MB per second. While this is an option, using flash memory is an expensive way to reduce swap time on a general-purpose machine.

Another way to reduce swap time is to reduce swapping itself. If we could reduce the amount of swapping that was required – perhaps eliminate the need to swap altogether – this would obviously be a great reduction in context switching overhead. This is something that operating systems try very hard to do. To understand this, we must develop ideas of memory paging.

Memory Paging

The idea of paging has its roots in the availability of physical memory. When the physical memory available on a computer is many times the requirements of a process's memory space, most of the large memory is wasted. If, however, we allow several memory images to occupy physical memory at once, not only can we use more memory but we might also be able to avoid swapping altogether (if all our memory images can fit into physical memory).

To implement this idea, we need some definitions. Physical memory is usually divided into blocks of fixed size called *frames*. Logical memory is analogously divided by the operating system into *pages*, which are also blocks of fixed size. To facilitate fast swapping, the storage medium is usually also divided into blocks, with each block the same size as a physical-memory frame. Similarly, it is best that the page size be at least some multiple of the frame size.

Now, when a process needs memory space, its code and data are brought from storage into memory in pages that are placed in physical-memory frames. When other processes need their pages to be brought into memory, these pages are loaded into physical-memory frames that are unoccupied. When a process terminates, its memory pages are removed from physical memory.

Now let's free up our ideas of swapping. If the operating system can translate addresses and can keep track of where pages are in memory, then there really is no need for process pages to be contiguous. By breaking up logical memory into pages, we can scatter those pages all over memory as needed. The operating system needs a way to keep track of all this; a *page table* is used to keep of track where pages are (both in memory and on page storage) and which process they belong to. By doing this, we can utilize as much memory as is available.

We should note here that some operating systems do not actually keep a specific table called a 'page table'. For some, the page data is stored with the PCB data in the process table. The information gleaned from all PCBs forms the 'page table' we speak of here. In other cases, page tables are kept, but they are kept on a per-process basis. Again, these process page tables together form the system page table we are discussing here.

The page table is used in address calculation. Each logical address used by programs on the CPU is composed of two parts: the page number of the memory block being addressed and the offset within the page of the memory location. The page table, then, is a mapping between logical-memory pages and physical-memory frames. Let's take an example as shown in Figure 7.4. In this example, logical-memory pages are 1 KB, as are physical-memory frames. In logical memory, addresses are (obviously) in sequence. The page table holds the physical-memory frame for each logical page, when that page exists in physical memory.

The memory-page table is not the whole story here. There is also a page table for pages stored in secondary storage. As with the memory-page table, this is usually kept in one place, but it could be distributed to process-page tables. It is usually managed by either the operating system or a memory-management unit on the hardware.

Memory Allocation Patterns

When we implement the carving up of process memory into logical pages and mapping those pages to physical-memory frames, we have a coarse-grained allocation of physical memory to the total memory needs of a process. Within memory pages, actual memory use takes place as memory is allocated and de-allocated by processes. There are patterns of memory allocation that emerge from fine-grained use of memory.

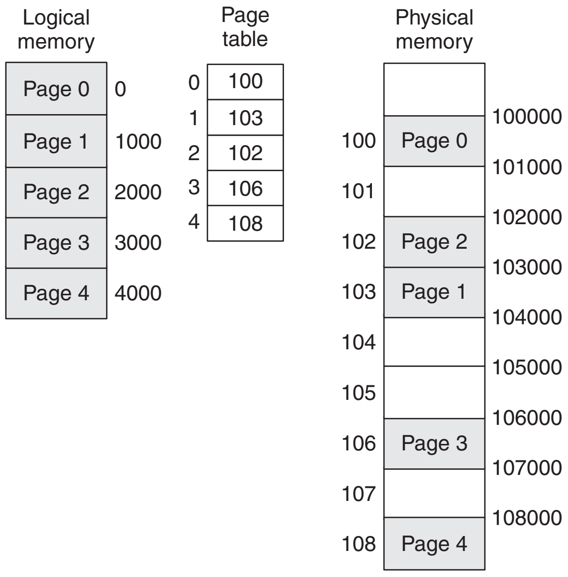


Figure 7.4 Using a page table to support memory paging

Memory is a chaotic place. In addition to referring to memory locations as data storage, processes cause the structures that the operating system has forced onto memory to change rapidly. When a process is moved from the ready queue to the running queue, it requires its memory pages to be in memory. If some pages are in memory and some are not, those that are resident are used as needed. Eventually, there is probably a need for pages to be swapped in, but that work is postponed as long as possible.

Within memory pages, memory is allocated in both static and dynamic ways. Static allocations result from fixed or predictable memory needs. These needs include space for the object code that defines a program and space for the fixed data requirements in a program. Declared variables are good examples of fixed data requirements: their space needs can be determined from parsing the source code. Dynamic memory needs are those that arise during program execution. Examples of dynamic-memory allocation are the creation of data objects using the `new` operator in C++ or using the `malloc()` call in C to create memory areas. Dynamic memory allocations are usually granted as continuous memory spaces (this reduces the need for swapping).

Even dynamic memory requirements are usually serviced from a memory area that is allocated in a static manner. Dynamic memory allocations come from a structure called a *heap*; heaps are located in allocated memory space like any other memory requirement. Heaps are typically allocated in pieces of a fixed size, which allows dynamic memory to fit into a memory-paging scheme.

Configuring pages can be a challenge because of the variety of ways that allocations take place. A big challenge in page configuration is the determination of proper memory page size. There are several problems that can result from inaccurate choice of page size.

Consider the inside of a memory page. If the memory area being used inside a page does not utilize the entire page, there is a certain amount of memory wasted. For example, if the program code or use of the heap does not completely fill a page, there is free memory inside a page that cannot be reused for other pages. This gets worse with heaps. As memory is allocated as a contiguous unit and deallocated in a program, dynamic memory areas develop ‘holes’: areas of unallocated memory are dispersed among areas of allocated memory. As time goes on and memory is allocated and deallocated, these holes get spread out around the heap. This creation of free but unusable memory is called *fragmentation*. The type of fragmentation that occurs inside a memory page is called *internal fragmentation*.

The way dynamic storage is allocated can aggravate fragmentation. Dynamic memory needs are typically serviced in one of three ways:

- *first fit*: a search is conducted through memory and the first area of free memory is allocated for the memory request; searching usually starts at the beginning of memory each time
- *best fit*: a search is conducted through memory and the area of free memory that is closest in size to the request is allocated (this type of allocation is best done if a table of free space – location and size – is kept; then the table is searched, not memory)
- *worst fit*: a search is conducted and the largest block of memory is allocated to the request; this approach allows a large amount of memory to be left behind as free space.

If space is wasted between blocks of allocated space, we call that *external fragmentation*. Internal fragmentation wastes memory that cannot be recovered, since the memory has already been allocated, but external fragmentation can be recovered if memory allocation breaks down. Consider a memory allocation pattern like that in Figure 7.5.

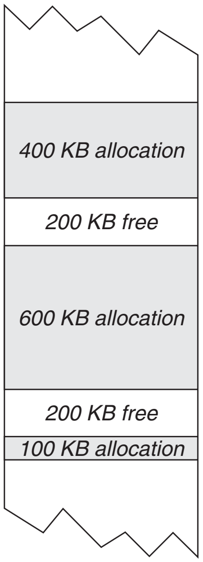


Figure 7.5 Example of external fragmentation

All memory has been allocated except the fragments in the figure. Together, the two free blocks amount to 400 KB of space, but they are not contiguous. The largest request that can be serviced is 200 KB – even when the total free space is 400 KB. When memory is contiguous, more allocation requests can be serviced.

Sometimes fragmentation can be avoided; other times it cannot. Fragmentation is caused by blocks of memory which cannot fit requests by themselves. Joining fragments that are next to each other might alleviate some of the problem; checking to see if fragments can be joined is usually done on memory deallocation. Another way of relieving the problems is to move fragments around so that they are next to each other and can be joined. These methods can be quite costly but can result in more memory being used.

On-demand Paging and Replacement

In previous sections, we have discussed how memory is manipulated by a program that is continually moved in and out of the running state. We have not, however, discussed the big picture: the execution environment has many processes, all of which are being moved in and out of the running state, each of which requires memory pages. What happens when the number of pages required by all processes exceeds the number of pages available in memory?

We can partially alleviate this problem by using *on-demand paging*. This method brings pages into memory only when they are required, much like dynamic loading of program code. For example, an application's program code might require three pages of memory, but only one page is required to start the program. For this situation, on-demand paging would only bring in the first page as the program starts execution and bring the rest of the pages as the code or data in them is needed. The operating system watches for *page faults*, which are events that are triggered when memory is referenced from a page that is not in memory, as determined by referencing its address through the page table.

Bringing one page into memory at a time might be too time-consuming. Therefore, an operating system might group certain pages into a *working set* of pages that are brought in together. For example, the first page of an application's code could be included in a working set with the memory pages for its statically declared variables and the first page for the application's heap. Bringing in a working set cuts down on the number of page faults and the amount of time to service the faults. The working set is determined by the operating system; a standard working set is typically used and adjusted as an application runs.

One other method that is used to save memory is *page replacement*. Memory fills up with pages rather quickly as a computer boots up and starts initial processes running. When memory is full of pages and an operating system brings in a page from the disk that needs to be placed (as indicated by a page fault), it chooses an existing page to replace. These pages are then *swapped* – the old page is removed and placed on the disk and the new page takes its place. There is often special space, called *virtual memory*, on the backing store allocated for this kind of page replacement. Virtual memory extends actual memory onto the chosen backing store. Because of the number of executing processes on a computer system, the size of virtual memory is often several times that of actual memory.[1]

As pages are replaced in memory, the criteria that are used to select which pages get replaced often affect a computer's performance.

[1] It is interesting to note here that even virtual memory is limited by a computer's memory word size. Virtual memory is referenced through addresses stored in memory words, so the size of memory words determine the amount of addressable virtual memory. For a 32-bit memory work, that number is approximately 4 GB.

Excessive page swapping – a condition called *thrashing* – is very bad for performance: the entire time slice devoted to a process can be taken up with disk I/O. The choice of page to be replaced can be done many ways, including the few examples below:

- *oldest first*: the page that has been in memory the longest is chosen for replacement; while this may make some intuitive sense, this is often a poor choice because it ignores how often memory in the page used; often the oldest pages in memory are those of shared libraries that are used by all processes
- *least frequently used (LFU)*: a page that has not had much use is chosen for replacement; the assumption is that the page will continue not to get much use; the operating system must also keep track of the length of time that a page has been in memory and be careful not to select pages that have just been added
- *least recently used (LRU)*: the page that was used longest ago is chosen for replacement; this assumes that a page that has not been used for a long time will continue not to be used.

Hardware can be relied on in many ways to assist with paging and virtual memory. First, while the page table is usually kept in memory, the MMU usually keeps a pointer to the page table in a special register called the *page-table base register*. Maintaining this register allows the operating system to place the page table anywhere in memory (in keeping with paging) and the MMU to track the page table. With paging hardware, memory is accessed as shown in Figure 7.6. The logical-page portion is replaced by the physical-page portion from the page table. Even with hardware assistance, the sequence shown in Figure 7.6 requires two physical-memory accesses for every one logical-memory reference. This doubles the memory access time.

Translation look-aside buffer

As a solution to this problem, MMUs often employ the use of a *translation look-aside buffer* (TLB). The TLB is a piece of very fast, associative memory, capable of searching many areas of memory simultaneously. This means that many table entries can be searched at the same time for a logical-page entry. This type of memory is very expensive which means that not much of it is used; MMUs usually use TLBs with between 64 and

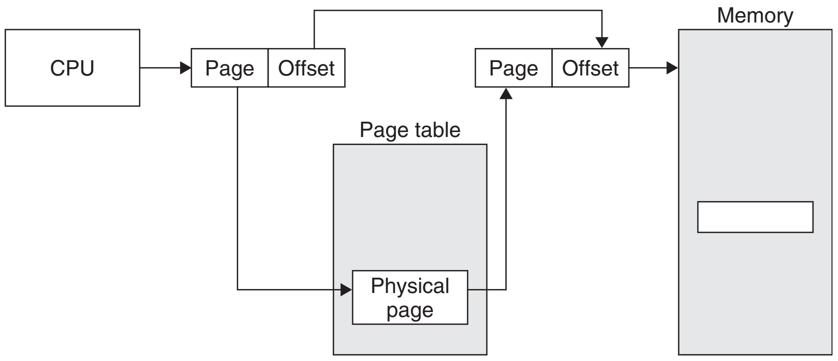


Figure 7.6 Paging hardware assembling a physical address

1024 entries. The TLB fills up with recent searches; when a search is not successful, the entries are added. Note that TLBs store only part of a page table's entries, because the memory is expensive and limited.

A new page table causes all the entries stored in the TLB to become useless. When a new page table is used – for example, on a context switch – the TLB entries must be erased to make sure the new process's logical-address space maps to the old process's physical-address space.

Swap-space configuration

The configuration of swap space and how big it should be are challenging issues. In Unix systems, it is typical to set up an entire partition of the disk for virtual memory. In addition, should the initial partition not be enough, Unix allows files to be added as swap space. Microsoft Windows sets up an area (like a file) on a disk drive and constantly monitors the space for the user. If that space needs to be increased, Microsoft Windows does it automatically – up to a certain boundary.

While it is hard to set rules for swap-space size, system administrators typically use a rule of thumb that swap space should be at least three times the size of memory.

Protection

With all the paging and replacing that goes on, it might be easy to forget that there needs to be protective barriers thrown up around memory pages. We need to prevent code from straying beyond the boundaries of its pages. Our scheme must embrace the swapping and paging ideas we have developed.

Protection in a paged environment must focus on physical-memory frames. Since every memory reference goes through the page table to reference frames, we can add protection bits for frames and store them in the page table. These bits can give certain properties for frames: read-only or read–write. For further protection, we can add an execute bit. To support paging, we can add a valid–invalid bit, which indicates if the page being accessed is actually in memory.

Note that these protection schemes are focused on the page table. Property bits are the easiest way to provide protection. When logical-to-physical translation is taking place, the nature and owner of the request is also analyzed. If the process that issues the address is not the process that owns the page or if the requested operation is something that is not permitted by the access bits, the attempt is ruled illegal and the operating system is notified.

The valid–invalid bit is set when paging occurs. If the bit is set to invalid, this means the page has been swapped to virtual memory and a page fault should be triggered.

7.3 Systems Without Virtual Memory

Many computer systems do not have the facilities to provide virtual memory. Consider a smartphone. The only storage available to the operating system is memory; most phones do not come with a disk drive. Because of this, most smaller systems – from PDAs to smartphones to higher-level handheld devices – do not utilize virtual memory in their memory-management strategy.

Consider the memory space used in most small-platform devices. Typically, these systems have two types of storage: RAM and flash memory. RAM stores the operating system code (to be used when the system boots); flash memory is used for both operating memory and permanent storage. Often, it is permissible to add extra flash memory to a device – such as a Secure Digital card, for example – and this memory is used exclusively for permanent storage.

In most cases, the absence of virtual memory does not mean the absence of memory management. In fact, most smaller platforms are built on hardware that includes many of the management features of larger systems. This includes features such as paging, address translation, and logical–physical address abstraction. The absence of virtual memory simply means that pages cannot be swapped. The abstraction of memory pages is still used; pages are replaced, but the page being replaced is discarded and not recorded.

Because of this, the absence of virtual memory does mean that care must be taken to preserve memory and to optimize its use. There are steps that smaller systems take to ensure that memory is efficiently managed.

- *Management of application size*: memory management begins with applications. The size of an application – from the code that the application needs to the memory areas that must be allocated for their use – have a strong effect on how memory is used. It requires skill and discipline to create small software and the attitude of developers is an obstacle. The push to use object-oriented design can also be an obstacle here (more objects means more dynamic-memory allocation which means larger heap sizes). Most operating systems for smaller platforms heavily discourage static linking of modules.

- *Heap management*: the heap – the space for dynamic memory allocation – must be managed very tightly on a smaller platform. Heap space is typically bounded on smaller platforms to force programmers to reclaim and reuse heap space as much as possible. Venturing beyond the boundaries results in errors in memory allocation.

- *Execution in-place*: platforms with no disk drives usually support execution in-place. Applications execute in memory without being moved from storage to operating memory. Since permanent storage *is* memory, this can be accomplished easily. Performance benefits from zero loading time, but also from the contiguousness of memory pages. In addition, there is no need to swap or replace memory pages.

- *Loading of DLLs*: the choice of when to load DLLs can affect the perception of system performance. Loading all DLLs when an application is first loaded into memory, for example, is more acceptable than loading them at sporadic times during execution. Users better accept lag time in loading an application than delays in execution. Note that DLLs may not necessarily need to be loaded. This might be the case if they are already in memory or they are contained on external flash storage (in which case, they can be executed in place). If they are in internal memory, they may be executed in-place.

- *Offloading of memory management to hardware*: if there is an available MMU, it is used to its fullest extent. In fact, the more functionality that can be put into an MMU, the better the system performance.

Even with the execution in-place rule, small platforms still need memory that is reserved for operation. This memory is shared with permanent storage and is typically managed in one of two ways. First, a very simple approach is taken by some operating systems and memory is not paged at all. In these types of systems, context switching means allocating operating space – heap space, for instance – and sharing this operating space between all processes. This method uses little or no protection between process memory areas and trusts processes to function well together. Palm OS takes this simple approach to memory management.

The second method takes a more disciplined approach. In this method, memory is sectioned into pages and these pages are allocated to operating needs. Pages are kept in a 'free list' managed by the operating system and are allocated as needed to both the operating system and user processes. In this approach, because there is no virtual memory, when the free list of pages is exhausted, the system is out of memory and no more allocation can take place. Symbian OS is an example of this second method and Section 7.5 gives a more detailed examination of its memory-management policies.

7.4 Segmentation

We have discussed two views of memory: a logical view and a physical view. The logical view can be described as the view the program has while the physical view is the way memory is really used. One of the duties of the operating system is to connect the two: to map the logical view onto the physical view.

Programmers and users typically view an application or program as having several parts or *segments*. A program contains code – a main program and user-defined functions. The system provides libraries of callable code. Data must be stored somewhere in memory. Operating space – heap and stack – is also needed. Each of these conceptual objects is usually viewed as a segment. Individual components of a segment are usually seen as accessible from the beginning of that segment as an offset.

This separation into segments is an idea reinforced by compilation and assembly tools. In addition, the execution format of an executable file also supports segments. Consider the format of executable files that have the ELF format (e.g., from the Solaris operating system), shown in Figure 7.7. The format supports a large number of segments as dictated by compilers and assemblers. The assembler for the SPARC architecture forces programmers to use at least two sections: code and data. This is typical and compilation tools are free to expand on this.

From previous discussions, we know that this logical view of segments of an application is not actually the way things are done. Memory is divided by pages and program memory is placed into these pages, which may be arbitrarily scattered throughout physical memory. *Segmentation*, then, is a way to keep the user view of memory segments intact even when the implementation in physical memory is quite different.

As with virtual memory, we must deal with logical-segment addresses and physical-segment addresses – and with the translation between them. Segments are typically numbered and this number can be used as part of the logical address. For example, we might use an address format such as <*segment number, offset*>, where the segment number is given as the leftmost part of the address and the offset within the segment is given as the rightmost part of the address.

Access to segments is typically assisted by hardware. As with virtual memory and paging tables, MMUs keep a segment table to help access segments. This table aids physical-address calculation through the process

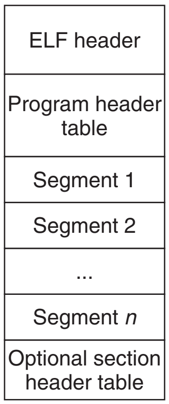


Figure 7.7 ELF executable file format

outlined in Figure 7.8. Like page tables, segment tables change with every context switch.

Figure 7.9 shows four segments: the main program, the static data area, the heap and library functions. Each of these segments is numbered. The segment table shown in Figure 7.9 maps each segment to a memory page.

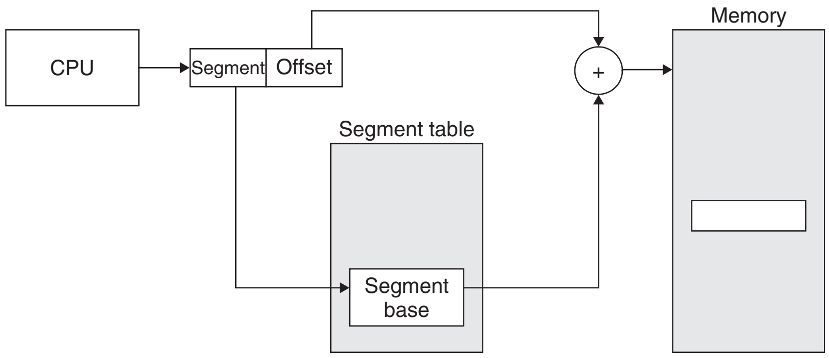


Figure 7.8 Segmentation hardware-address calculation

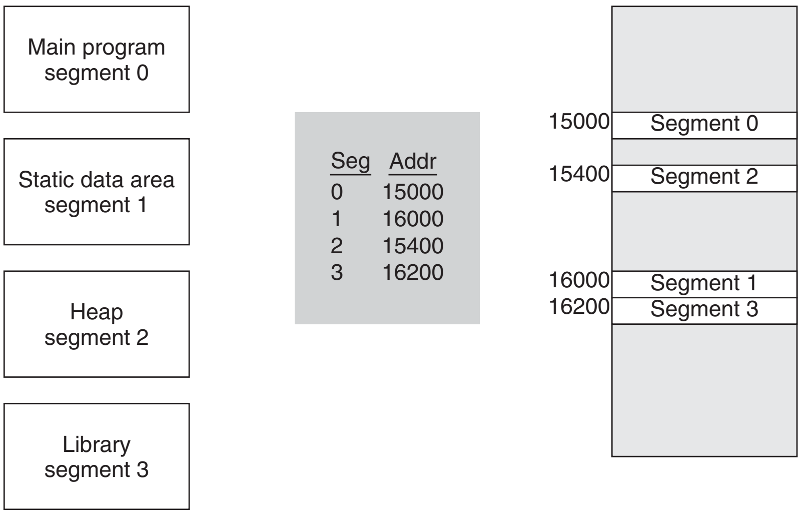


Figure 7.9 Example of segment tables

It is possible to combine segmentation and virtual paging. This amounts to a double table calculation: the segmentation table points to a virtual address, which is translated through a page table to a physical address. This is quite a bit of overhead in address calculation, so many machine architectures provide special registers to hold portions of the segment table. For example, the Intel 80386 had six segment registers, supporting programs with up to six segments. The advantage to using both segmentation and paging is in using a fully functional set of memory models. Paging can be used for system performance and segmentation can be used to support programmer modeling.

7.5 Memory in Symbian OS

Symbian OS provides a great example of a system that does not use a virtual-memory–swap-space model for its memory management. It does, however, use most other mechanisms we have discussed for managing its memory, including hardware MMUs.

Symbian OS is a 32-bit operating system, which means addresses can range up to 4 GB. However, it employs the same abstractions as larger systems: programs must use logical addresses, which are mapped by the operating system to physical addresses. Programs may be placed at arbitrary locations in memory. At any given time, a program does not know where exactly it is in memory, so the use of logical addresses is important.

As with most systems, Symbian OS divides memory into logical pages and physical frames. Frame size is usually 4 KB, but is variable. Since there can be 4 GB of memory, a frame size of 4 KB means a page table with over a million entries. With limited sizes of memory, Symbian OS cannot dedicate 1 MB to the page table. In addition, the search and access times for such a large table would be a burden on the system.

To solve this, Symbian OS adopts a two-level page-table strategy, as shown in Figure 7.10. The first level, the *page directory*, provides a link to the second level and is indexed by a portion of the logical address (the first 12 bits). This directory is kept in memory and is pointed to by the *translation table-base register* (TTBR). A page-directory entry points into the second level, which is a collection of page tables. These tables provide a link to a specific page in memory and are indexed by a portion of the logical address (the middle 8 bits). Finally, the page in memory is indexed by the last portion of the logical address (the last 12 bits).

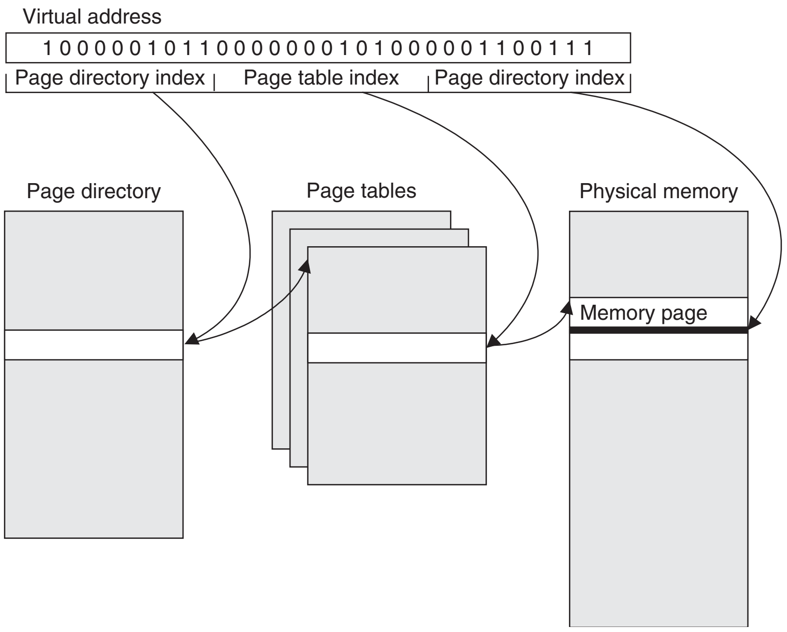


Figure 7.10 Paging and virtual addresses in Symbian OS

Hardware assists in this logical-to-physical address-mapping calculation. While Symbian OS cannot assume the existence of any kind of hardware assistance, most of the architectures on which it is implemented have MMUs. The ARM processor, for example, has an extensive MMU, complete with a TLB to assist in address computation.

What happens when a page is not in memory? In Symbian OS, this represents an error condition, because all application-memory pages should be loaded when the application is started. Remember that DLLs are pulled into memory by small stubs of code linked into the application executable, not by a page fault on a missing memory page. Since Symbian OS can address 4 GB of memory and it is unlikely that a smartphone would have such a large amount of physical memory, there might be situations when a page is referenced that is not in memory. Such a reference would cause an 'unhandled exception' error, terminating the user's application. Users of Symbian OS who have experienced a `KERN-3 EXEC` error when running an application have seen this happen.

Despite the lack of swapping, memory is very dynamic in Symbian OS. Applications are context-switched through memory and, as we stated, are loaded into memory when they start execution. The memory pages each application requires can be statically requested from the operating system upon loading into memory. Dynamic space – e.g., for the heap – is bounded, so static requests can be made for dynamic space as well. Memory frames are allocated to pages from a list of free frames; if no free frames are available, then an error condition is raised. We cannot replace memory frames that are used with pages from an incoming application, even if the frames are for an application that is not executing currently. This is because there is no swapping in Symbian OS and there is no place to which it can copy the displaced pages.

There are four different versions of the memory implementation model that Symbian OS uses. Each model was designed for certain types of hardware configuration.

- *The moving model* was designed for early ARM architectures (ARM version 5 and before). The page directory in the moving model is 4 KB long and each entry holds 4 bytes, giving the directory a size of 16 KB. Memory pages are protected by access bits associated with memory frames and by labeling memory access with a 'domain'. Domains are recorded in the page directory and the MMU enforces access permissions for each domain. While segmentation is not explicitly used, there is an organization to the layout of memory: there is a data section for user-allocated data and a kernel section for kernel-allocated data.

- *The multiple model* was developed for versions 6 and later of the ARM architecture. The MMU in these versions differs from that used in earlier versions. For example, the page directory requires different handling, since it can be sectioned into two pieces, each referencing two different sets of page tables: user-page tables and kernel-page tables. The new version of the ARM architecture revised and enhanced the access bits on each page frame and deprecated the domain concept.

- *The direct model* assumes that there is no MMU at all. This model is rarely used and is not allowed on real smartphones. The lack of an MMU would cause severe performance issues. This model is useful for development environments where the MMU must be disabled for some reason.

- *The emulator model* was developed to support the Symbian OS emulator on Microsoft Windows. As one might expect, the emulator has a few restrictions in comparison to a real target CPU. The emulator runs as a single Microsoft Windows process, therefore the address space is restricted to 2 GB, not 4 GB. All memory provided to the emulator is accessible to any Symbian OS process and therefore no memory protection is available. Symbian OS libraries are provided as Microsoft Windows DLLs and, therefore, Microsoft Windows handles the allocation and management of memory.

7.6 Memory Use in Linux

Linux is designed to run on the Intel architecture and, therefore, it supports the memory models designed into the Intel 80x86 line of processors.

Linux is a 32-bit operating system and therefore can access up to 4 GB of memory. The processor MMU supports segmentation and memory frames; Linux supports frame sizes of 4 KB.

Linux implements a free list of pages and supports placement of application pages anywhere in memory. It uses a 'buddy system' algorithm to allocate page frames. The goal of this system is to allocate contiguous page frames as often as possible and it does this by maintaining as many adjacent free page frames as possible. The buddy system keeps multiple lists of free blocks of various sizes, joining blocks, and thus moving them between lists, whenever blocks are freed up next to other free blocks. The intent is that blocks are allocated faster because the right blocks at the right size are found by checking the correct list (instead of searching the entire list).

Like Symbian OS, Linux uses a page-directory–page-table structure for translating logical addresses into physical ones. As we have discussed, this two-level approach to address translation saves memory but doubles the time for address calculation. Hardware helps with this situation; the Intel architecture provides address computation in its MMU as well as a TLB to find pages quickly.

Linux implements the idea of *reserved-page frames*. These frames represent an area of memory reserved for the Linux kernel and its data structures. This area can never be relinquished or allocated to a user process. Linux tries as much as possible to keep its page frames contiguous; it even chooses an area of memory that is unlikely to be used to load itself into at boot time. The kernel typically starts at the 2 MB mark and reserves as many frames as it needs. Because it cannot predict how many dynamically assigned pages it needs, these are used where they can be found.

Linux implements a swapping strategy for memory pages and most often uses disk space for virtual memory. Swap space is implemented as either a partition on a disk drive or as a large file. Within this space, Linux keeps pages for each process as close to each other as possible. This minimizes disk I/O time. Pages are moved to swap space when the number of free memory pages drops below a predefined threshold (Linux does not wait for the number to become zero).

The selection of pages to remove from memory is interesting. The general rule for Linux is to choose one of the pages held by the process with the most pages in memory. Within this collection of pages, LRU is used. A counter is added to the page table that stores the amount of time that has elapsed since the last access to the page. The page with the largest amount of elapsed time is chosen.

Linux uses several types of segmented memory. It supports code and data segments for the kernel, code and data segments for user processes, and global and local segment tables. Segments are shared by all processes. The segment tables contain entries that direct the operating system to segments for individual processes. Any hardware properties that support segmentation are exploited.

Protections are designated inside page or segment tables. Each page or segment is coded with a specific access value and the type of access is always compared to the access value of the page or segment on which the operating is done.

Linux has been adapted to run on 64-bit machines, which affects several parts of memory management. The address space expands from 4 GB to 16 EB,[2] making much more memory possible. Larger memory means larger swap space allocation. More memory means larger paging and segment tables. Rather than take up huge amounts of reserved operating system memory for larger page tables, 64-bit Linux implementations typically use a three-level paging scheme, which extends the two-level scheme used in 32-bit systems.

7.7 Summary

This chapter has discussed how operating systems manage memory. Like a computer's CPU, memory must be shared between the processes

[2] That is, 16 exabytes. An exabyte is slightly more than one billion gigabytes. The new address space is 4 billion times the size of the former 4 GB address space.

that execute on a computer. We began by introducing the terminology and background of memory management, including how a program progresses from source code to an in-memory executing image, the difference between logical and physical addressing, and the mechanisms used for dynamic loading and linking. We then discussed paging and how paging is used to manage memory. Issues surrounding paging include the use of virtual memory, how to keep track of pages and how memory and page allocation affect fragmentation. We then discussed how the lack of support for virtual memory on smaller computer platforms affects operating system design. We introduced segmentation and saw how the concept affected design. We concluded the chapter with discussions of memory management in Symbian OS and Linux.

Exercises

1. Can a logical address also be a physical address? If not, why not? If so, what type of address binding would be used for this?
2. When physical memory is allocated as frames, can external fragmentation exist? Explain.
3. Let's say that a program makes many requests for dynamic memory, but they are all small (10 bytes). Which memory allocation scheme is best for this pattern of memory allocation?
4. How are relocatable binding and execution-time binding different?
5. Consider a configuration with a logical address space that has 4096 words, mapped onto a physical memory with 64 pages.
   a. How many bits should the logical address have?
   b. How many bits should the physical address have?
6. Consider a new page-replacement algorithm, Not Recently Used (NRU), that favors keeping pages which have been recently used. This algorithm works on the following principle: when a page is referenced, a bit is set for that page, marking it as referenced. Similarly, when a page is modified (written to), a modified bit is set. By using a mix of these bits, the system can guess about recent usage (e.g. referenced but not modified or modified but not referenced). How is this strategy different from Least Recently Used?

7. Give at least three situations in which memory space needs to be protected from programs.
8. Smartphones are being designed with hard disks (e.g., the Nokia N91). What would the effect be if these devices started using virtual memory?
9. Why is loading all application pages and dependent modules right away important for a smartphone device?
10. How many page tables are possible in Symbian OS?
11. How many memory frames are possible in Symbian OS?
12. What is the alternative to the reserved page frames that Linux uses? What effect does the alternative have on an operating system?

8

File Systems and Storage

We introduced operating systems in Chapter 1 saying that using one was like viewing hardware through 'rose-tinted glasses'. Operating systems present a view of the system resources to users and enable applications to use those resources in a constrained and organized way.

File systems are perhaps the greatest example of this abstracted view of system resources. A file system takes a collection of bits and bytes from a storage medium and creates an organized hierarchy of files and directories, enabling complete use of this abstract concept. All storage media has the idea of files and directories and it is an incredibly useful concept.

This chapter explores how this abstraction is built and used by operating systems. We start with the basic building blocks of file systems – names, attributes and operations – and move on to file system implementation, giving examples in several forms. We then look at file systems on mobile devices, using Symbian OS as an example. Finally, we examine security issues involved in file system implementation.

8.1 Files and Directories

Computers store information in many ways on many different media. There are magnetic tapes, CD-ROMs, disk drives, flash memory and others. One of an operating system's jobs is to provide a consistent, convenient interface to all these storage media. The idea of a file was invented as a structural unit to help provide this interface. A file is a logical concept that is translated onto a physical device. This section introduces files and other organizational concepts, along with their properties.

Basic Concepts

A *file* is a collection of information formed by bits of data on a storage medium. It is the foundation unit of storage in a computer system. As users work with storage, files are the units of manipulation they use.

The information held in a file is at the discretion of the file's owner. Files can contain any kind of information; they are simply a collection of bytes. The same file may be interpreted in several different ways, depending on the application that reads the information it contains. In fact, files can have many different kinds of information kept inside, organized however the creator wishes. Again, the interpretation of the bits contained within a file depends on the application that reads the information.

Files can have a specific type of information with certain properties. Files of a specific type are said to have a specific *structure*. Some file structures are quite common and recognizable. Text files have characters in them that are organized in a line-by-line format. A file is typically called a *binary* file if the contents are not readable by humans (i.e., not a text file). There are many structured binary files, for which the structure is documented by some standard body. Image files contain picture data in many different formats: Jpeg, Gif and PNG, to name a few. Files can contain execution data of a compiled program that can be transferred bit-for-bit to memory and executed. A binary file could be a database, with its contents readable by a database server application.

A *directory* is an organizational tool that groups files together in a hierarchy. A directory is analogous to a folder: a directory can hold files or other directories and folders. Because they hold other directories, a directory structure forms a tree of directories, with a specific directory at the top of the tree called the *root*. The navigation through the directory tree from root to file is called a *path*; paths through a directory tree have unique names – path names – that reflect the unique path through the directory structure.

Another way to look at a directory is as a table of contents. It is a table, whose contents are files and directories, kept on the storage medium. It is through this table that access to files is allowed. The directory is the only way to find the files it holds; no data on those files is kept elsewhere. In fact, all searches for files begin at the root of a file system, and proceed through the file system by looking up directories in the path to a file, opening those directories, and finding the next file or directory in the path.

Disk drives are also organized by partitions. A *partition* – often called a *volume* or a *minidisk* – is a subsection of a disk drive that is regarded as its own contained space. A disk drive may be a host to a number of different file systems, each of which is usually created within an individual disk partition. Files and directories cannot span partition boundaries. When a partition is full, other partitions cannot contribute file space. In essence, each partition is its own storage device. Partitions are often enforced by being implemented by the storage medium hardware. There is at least one partition per medium; if not otherwise split up, all storage is kept on one partition.

What if There Were no Directories?

The concept of files has been around almost as long as the concept of external storage. One of the first uses of a file was in 1956, when IBM introduced the IBM 350 disk file. At the time, the entire disk was a file.

Soon, the idea began to catch on that multiple files could exist on a storage medium. However, the idea of files preceded the invention of directories. Files started to proliferate and their organization started to get confusing. Short names were used, but names were also used to reflect organization.

Directories were invented to aid organization. But imagine how all the files on a computer were grouped without the benefit of directories. Without directories, there would only be one flat file space, with all files organized by the names they were given. When you consider the number of files on a typical modern computer, be grateful for directories.

Attributes

Files and directories have many properties – often called *attributes* – that help identify and organize them. Certain attributes also help with managing various aspects of file systems, such as security.

Files are identified by their *names*. Names are mostly a convenience for humans, but they are used by operating systems as well. A name is composed of alphanumeric characters; the case of letters is sometimes significant, depending on the operating system. The length of a name is sometimes bound; for example, early versions of MS-DOS had the 8.3 limitation: eight characters in the name of the file and three characters in the extension, separated by a period. The file *extension* is the suffix on a file and is often used to categorize the type of file it is: `.doc` files might be document files while `.txt` files might be text files.

The file extension often classifies the file *type*. A file's type determines the kind of data contained within it. Various operating systems support file types in various ways. Microsoft Windows, for example, associates an application with a file type (`.doc` files are usually opened with Microsoft Word). Linux, on the other hand, almost always completely ignores file extensions, depending instead on the contents of the file to determine how it is supported.

A file has a *size*, i.e., the number of bytes it contains. A file's size is usually determined by its contents, but sometimes a file may take up space on a storage medium but be empty. An empty file is usually a zero-length file, but a file could be filled with meaningless data, thereby having a size but still being considered 'empty'. This is a good example of the difference between physical size and logical size, where a file's physical size is marked by bytes occupied on a storage medium and its logical size is marked by usable content.

A file often has an *owner* and also has a *group* identification. These two attributes are used by operating system security features to regulate which processes and users may have access to the contents of a file. These attributes are given by the operating system upon a file's creation and can be changed throughout a file's lifetime. The owner of a file is the user ID of the user on whose behalf the file was created. The group ID identifies a group of users that may access this file.

In addition to owner and group IDs, a file may also have additional protection attributes. These attributes inform the access control mechanisms of an operating system about how a file may be used. This access control information often specifies one of three access methods – reading, writing or executing – in one of several categories, for example, owner, group and 'other' (users who are not the owner or in a file's group).

The date and time of access is usually an attribute of a file. The access type is usually recorded with the date and time. It is common to record time of creation, time of reading and time of modification.

The information about a file is kept in a directory structure on the storage medium (see Figure 8.1). The directory – the table of contents – holds the name of the file and the location where the file starts on the disk. A file's starting point is usually not the start of its contents, but rather a data structure holding the rest of the file's information. Part of this information is the starting location of the file's contents.

Much of this same information is stored for directories themselves. Directory information includes name, owner, group, size, access information and access times. In addition, as indicated above, they contain a table of contents that point to locations on the storage medium where file information is held.

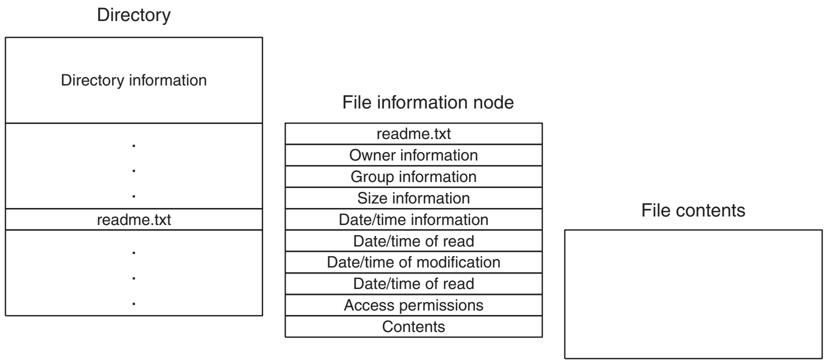


Figure 8.1 Storage of directory and file information

Names

The naming of files and file paths is an interesting issue. Obviously, file names need enough characters so that the name can be descriptive. However, directory information space is limited, so names cannot have infinite length.

File names typically consist of letters and numbers, with some punctuation. File names are typically restricted to some limit, say 256 characters, with a suffix of some sort. While the historical convention is to have a suffix of three characters, suffices can be of any length.

The pathname of a file is constructed as follows:

```
<root> <directory list> <filename>
```

The root of a file-system tree is designated in many ways. In Linux, the root is always the same, designated simply as '/'. In Microsoft Windows and Symbian OS, the root is dependent on the storage medium, which is designated by a letter followed by a colon, and a '\' character to designate the root. The listing of the directory path differs between operating systems. So the path to 'readme.txt' might be designated `/software/graphics/readme.txt` on Linux and `C:\software\graphics\readme.txt` on Symbian OS.

Structure

The internal structure of a file must match the structure expected by the application that uses it. In addition, the file structure must match what the operating system expects, if the operating system is to understand that structure.

Some operating systems put very little emphasis on file structure. Linux has very few expectations of a file: it treats all files as a set of bytes without any particular structure. Microsoft Windows has a similar philosophy. Operating systems like these associate applications with certain files, but that association is not determined by how the data inside a file is structured.

Other operating systems, on the other hand, do indeed pay attention to file structure. If a file structure is recognized, these operating systems have special operations that can be performed on the file. For example, DEC's VMS operating system recognized special file structures. This support was woven throughout the operating system, even into the APIs that supported programming languages. Note that if an operating system is to support structured files, it must implement recognition of structured files through the system.

The association of applications with certain files is sometimes implemented through file structure. Linux has the notion of 'magic cookies' for this association: the first bytes of a file are matched against a database of applications. If a match is found, that file is associated with the application from the database.

Automatically deriving the meaning of a file can be problematic. The problem is that all such systems – including extensions and magic cookies – assume that the identified file characteristics uniquely specify a file type. But this system is very easy to fool. In Microsoft Windows, for example, simply changing a file's extension is enough to make the operating system change the association. Changing a file's suffix from `.txt` to `.pdf`, for example, causes Microsoft Windows to open the file with a PDF reader instead of a file editor. Unfortunately, there is no foolproof way to judge the type of a file.

Operations

Operating systems define eight basic operations that can be done on files. These operations are the ones that can be accessed through user interfaces and programming APIs.

- *Creation*: it takes two steps to create a file. First, there must be *space allocation*. Even when an empty file is created, a portion of disk space is allocated to it, if only for file information. The second step that must take place is that the file must be entered into the directory. The information in the directory records the file's name and the location of the space allocated for that file on the disk.

- *Opening*: to access a file's contents, that file must be opened. The file name is the main input parameter to this operation. When requested to open a file, the operating system does several things. It checks to see if the file exists. If it does not exist, the system may create it (or give an error, depending on the system call used). Information about the file is then recorded in internal tables. This information includes the location of the file on the storage medium and the position of the next byte to be read. This system information is used by the operating system for other operations. As a result of an open operation, a file handle is created by the operating system within system memory and that handle is passed back to the client. The file handle is then used to identify the file during subsequent operations (instead of passing the file name again and again).
- *Reading*: information in a file would be useless if we did not read it. Reading a file causes the operating system to attempt to retrieve bytes from an open file at that file's current position. If the file still has data at its current position, those data bytes are retrieved and supplied to the system caller, and the position pointer is moved to reflect the amount of data that was read. Otherwise, an error is returned to the caller.
- *Writing*: a write operation adds data to a file. Data is typically written into a file's space at the currently recorded position within the file. That data might overwrite existing data or it could be inserted into the file. The current-position pointer for the file is then advanced to reflect the data that was written to the file. Note that writing data to a file usually makes the file grow in size and the operating system might be forced to allocate new disk space to accommodate that growth.
- *Repositioning*: the current-position pointer can be reset. Repositioning this pointer will affect where the next data byte is read from or written to.
- *Closing*: closing a file does the reverse of opening. It updates the file's information in the system directory and removes the file's entry from the system's open-file table. Any subsequent read or write operations are likely to fail because the file handle in question is not in the system table.

- *Truncating*: truncating a file means removing data that exists after the current-position pointer for the file. Often, this operation is used to empty a file's contents without deleting the file.
- *Deleting*: to delete a file, the two steps for file creation need to be reversed. First, the file is removed from the directory with which it is associated. Secondly, the file's space is reclaimed from the storage medium. Most file-system implementations permit a client to delete a file without a need to open it first.

There are other common operations for a file that might combine some of the basic operations above. A file can be appended to, for example, by repositioning the current-position pointer for the file to the end of the file and performing a write operation. Files can be renamed or cut from one directory and pasted into another. A file can be copied by creating a new file, opening the source file, and reading and writing between the two.

Sometimes multiple position pointers are kept for a file in the file table. It is sometimes advantageous to maintain a position pointer for reading and one for writing. This has several implications for working with a file; for example, a read operation does not change where a write operation takes place. In fact, it is possible that these operations may happen simultaneously because one does not affect the other.

The system's open-file table is obviously central to manipulating files. The file table holds several pieces of information for each file that is opened. We have pointed out several of these: name, current-position pointer (or pointers) and location on the storage medium. In addition, the *access rights* for the file are usually stored in the table.

Types of Access

While the operations of reading and writing seem straightforward, there are actually several different methods for accessing files. Much of the difference between these methods focuses on how the operating system I/O implementation moves the current-position pointer in a file.

Sequential access is probably the simplest method for file access. The data in a file is accessed in a linear order from beginning to end. The current-position pointer moves forward in a file after every read or write operation. Sequential access is usually the default access method for file operations.

Direct or random access is a method where any part of a file is accessible. Using direct access, a system call can place the current-position pointer anywhere in the file before reading or writing. There are two types of direct access: arbitrary access and fixed-length access. Arbitrary direct access of files views a file only as a set of bytes and allows the current-position pointer to point to any byte in the file. Fixed-length direct access views a file as a set of blocks of fixed-length. System calls that use fixed-length direct access move the current file pointer to specific blocks.

Direct access is useful when access to information needs to be immediate. This is the case for implementations of database systems, where fixed-length direct access is used to read records in the database. This is also the case when reading a file means skipping to arbitrary positions within that file, such as reading digital music data or processing video.

Another way of accessing files is by *indexed access*. The index of a file is analogous to the index of a book: data within the file is recorded along with its position. This means that there are typically two files used for indexed access: an index file and a data file. Once pertinent data is found in the index file – say a keyword or product ID – the position within the data file is obtained and used directly to retrieve the information. Indexed access thus builds on top of direct access methods. Modern operating systems have moved away from implementing indexed access; it is offered as an add-on method through special libraries rather than a core operating system service.

8.2 Implementation of a File System

The implementation of a file system is a lesson in abstraction. A file system must support the concepts of files and hierarchical directory structures. Operating systems typically provide the same interface – file creation, opening a file, reading, writing, etc. – no matter where a file is located. However, a file system on a CD-ROM is implemented in a very different manner from a file system in the memory of a smartphone. But to a program, both file systems look the same.

We explore how this abstraction is implemented in this section. We first explore the concepts an implementation must use. Then we cover several different file-system implementations.

A Generic View

It is useful to review the components of a file system and the ways that we use those components to build the file-system abstraction. There are some assumptions we make about file systems and some file and directory

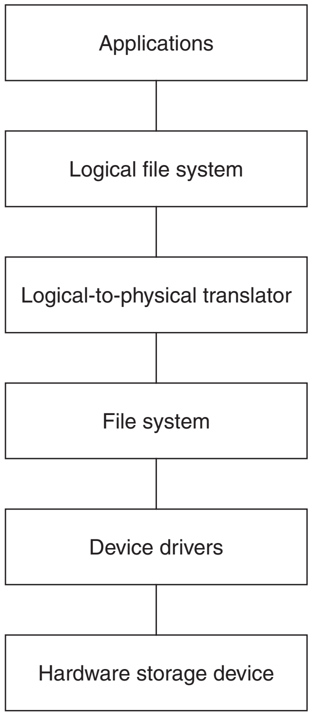


Figure 8.2 Generic file-system implementation layers

components that we can describe so that we can better discuss specific implementations later.

The file-system abstraction is built up in layers. Each layer adds some functionality and represents an area for implementation. Consider the diagram in Figure 8.2 as a depiction of the layers of a file system.

Storage devices

At the lowest layer lies the storage device. Physical storage can be viewed as a linear sequence of bits or bytes, usually organized into larger units called blocks. Blocks are typically the unit of transfer between a storage device and the operating system. There are many different types of storage device that are supported by operating systems and each communicates in a different way. The I/O-interface controller communicates effectively with the bus to which it is connected, but it also has its own command interface.

Storage devices, like memory, give addresses to their data space. For some devices, addresses can be as simple as block number. For these devices, a read request must be accompanied by a single number, indicating the block to read. Other devices can have a more complicated addressing method. Hard disk drives, for example, are usually organized in three dimensions. Disks are made of concentric rings of magnetic media; one of these rings is called a *track*. Tracks are usually divided into smaller storage portions called *sectors*. *Cylinders* are formed by mapping the same track of each disk platter vertically. A read request for a hard disk drive, then, must be accompanied by three numbers: cylinder, track and sector. Some file-system implementations group several sectors into one single logical unit called a *cluster*, so a request to read data from a disk can address only a cluster, but not an individual sector. This technique allows manipulation of the size of the smallest addressable storage block on big storage devices.

By virtue of their size and their functionality, smartphones have some specific requirements when it comes to storage media. These devices all come with onboard flash memory, which is used by the operating system as operating memory and by users to store files. Other storage is either built-in or removable; built-in secondary storage usually takes the form of a hard disk drive and removable storage is usually a form of flash memory in a small form factor. Flash memory can be accessed faster, but wears out more quickly; hard-disk space can be greater.

Device drivers

The level above physical hardware is the device-driver level. Device drivers act as translators. They communicate with both the operating system and the storage device, translating operating system requests into the command interface of the storage device. In addition, device drivers include interrupt handlers that implement various methods of reading from the device – from real-time, byte-level access to direct-memory access. Device drivers can access the hardware of both the storage medium and the computer's CPU to transfer data and commands.

File systems

The next level is the file-system layer. This layer implements the basics of a file system. It knows hardware information such as block sizes on storage media but it also knows the logical addressing system used by upper layers. This layer receives logical addresses and translates into the physical addresses of the storage medium. This layer usually tracks the free-space on a storage medium and runs space management algorithms. This layer will implement file system management methods that are generic enough to be used by any higher-level implementation.

Logical file systems

The logical file-system layer implements specific types of file systems. Concepts such as security, access control, and file typing are different from file system to file system and these implementations would be built on top of a generic file system. Implementations from this layer provide the APIs for operations common to files: opening, reading, writing, closing, etc. This layer keeps track of files via a file table that uses *file control blocks* (FCBs) as entries. Like process control blocks, FCBs record information about files in use. The information recorded in an FCB is dependent on the file-system implementation.

Applications

Applications interface with the logical file system either through the APIs provided by it or by communicating with file servers. Even when applications need raw, byte-level access to storage, they must work through these methods, which pass system calls through the hierarchy. It is important to remember that system calls are the way that applications get kernel-level access to system resources and file systems are a resource that must be coordinated by the kernel.

Storage Medium Structure

There is a generic structure to the space on a storage medium. In general, there are four components that organize the bits on a storage medium to produce a file system.

The first area on a storage medium is typically the *master boot record* (MBR). This is an area that contains information necessary to boot the operating system and get the computer up and running. Early on, operating system kernel code used to occupy the MBR, but as operating systems grew in size and the need to change operating systems became more obvious, this code was kept somewhere on the storage medium and the *address* of this location was stored in the MBR. On most computing systems, the MBR is the first set of bytes on the storage medium; this assumption is made by many operating systems.

The next area of the storage medium is called the *partition-label control block* (PLCB). This area contains information about storage medium partitions. Such information can include the size of each partition, the position at which each partition starts and ends on the storage medium, and the number and location of free blocks of storage. Most partitions are defined here; there can be special partitions (areas of the disk) that are defined elsewhere.

The next area on a storage medium is the *directory structure*. This area contains information to organize and find files; we detail this structure in the next section.

The next area is used for file storage. Both FCBs and actual file contents are stored in this area. We detail the structure of this space in a later section.

In addition to the structure of the storage media, the operating system also keeps track of information for each file system. The structures maintained in this way reflect and augment the storage structures on the storage medium. The operating system usually maintains a partition table, a table defining the directory structure, and open-file tables. This last structure records where open files are in the operating system by placing FCBs in a table, kept in memory. Tables of open files are usually also kept on a per-process basis.

Directory structure

In general, a directory must maintain several pieces of information. A directory is the central point of file-system information and is the only way to find where a file begins on a storage medium. Therefore, it must maintain information on all files and where to find them. Sometimes this information is resident in the directory and sometimes directories simply point to this information on a storage medium.

Directories are really just files of data stored in a file system. A directory usually holds the following information:

- *directory name*: directories have names, just like files, and these names need to be stored with the directory
- *directory size*: the number of files and directories must be stored
- *FCB information*: each file and directory has a control block and these control blocks are accessible through the directory. Either the complete FCB is stored with the directory structure or the address of where the FCB resides is stored.

As we stated previously, each directory is accessed by other directories. The chain of directories forms a tree. There is, therefore, a root to the directory tree, a single directory at which paths to all others start. This root directory is not accessible from other directories, because it is at the top. The root directory location is either stored directly in the MBR (as a storage medium address in a specific location) or it is stored on a specific place on the storage medium. The former method is the most flexible and allows root directory information to be stored and duplicated for backup purposes.

Since all storage in Unix is a set of bytes and all storage can be referenced as a file, it should not be surprising that Unix treats directories as file space that can be opened by applications and manipulated. Obviously, directory storage in Unix has a specific structure but, as with all data, Unix does not organize the structure, but leaves that job to the parts of the kernel that implement a file system. It is possible to read the data from a directory and look at it in any way. You can even look at the contents of a directory by using the 'cat' command.

File structure

Files are generally stored in pieces on the storage medium. As we discussed previously, each piece is the size of a block on the storage medium and these pieces are strung together to make a complete file. Depending on the method of storage, performance may be adversely affected by how a file is stored.

The structure of a file's FCB is an important start to efficient storage. How much or how little information is stored makes a difference to how fast file information is accessed. A generic structure of an FCB is shown in Figure 8.3.

How space is allocated for files plays a large part in the performance of file I/O. Contiguous allocation of all the pieces of a file is the best way to store them for performance reasons. Performance is directly affected by how hard the mechanical aspects of the storage medium must work. Storing file blocks contiguously minimizes the movement of the read head on disk drives and thus renders the best performance. Fragmenting a file into many pieces flung widely across a hard disk make the hard disk's reading mechanism move more over the disk surface and slows I/O performance.

However, contiguous allocation of file blocks is rarely possible. Since, in most cases, it is impossible to accurately predict how much space a file needs, it is also impossible to allocate enough contiguous space for a

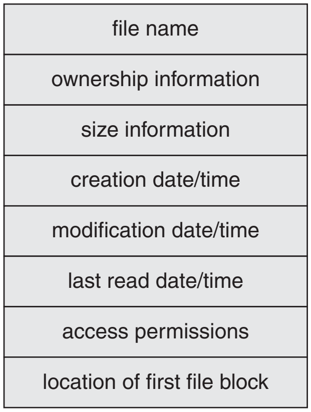


Figure 8.3 A typical file-control-block format

file to grow into the space allocated for it. It is, rather, more prudent to place pieces of a file wherever they may fit and link the pieces together somehow so the file-system implementation can put them together while reading data from the pieces.

Working with linked file space requires having access to the linkage information. In some systems, the linkage information can be kept with the file table. Other systems store link information with each piece of a file on a storage medium. In these cases, the file looks like a linked list, and working through the contents of a file means traversing a linked list from front to back.

Sometimes a compromise is made between contiguous allocation and linked allocation. File blocks can be put contiguously together in clusters and the clusters can be strung together to allow flexibility in file growth and allocation. A cluster is usually a group of disk sectors, which contain the file blocks. The size of a cluster is determined when the file system is created and remains constant over that file system.

A slight variation on the method of keeping link information in a file is the indexed approach. In the indexed approach, the address of the first pieces of a file is stored in the file table. The address on the storage medium of each piece of a file is stored in the file table as an index, an address relative to the base address of the file. Sometimes, if the file is large, the index entries are stored as the first file block on the storage medium and the entry in the FCB points to this block.

We must deal with fragmentation issues when we split the storage into fixed-size blocks. All the above schemes suffer from internal fragmentation: files are seldom exactly the size that fits into a specific number of fixed blocks. There is usually space within a block that is wasted. External fragmentation is the space wasted outside the collection of blocks. If a fixed-block scheme is used, external fragmentation is eliminated as the storage medium is carved up into blocks that fit exactly.

From time to time, it is a good idea to *defragment* storage media. Defragmenting has nothing to do with internal or external fragmentation; it is focused on the wide dispersion of the blocks belonging to file content across a storage medium. The closer a file's blocks are, the less mechanical wear occurs from trying to move all over a medium to read the blocks. Defragmentation moves the blocks of all files so that they are as close to each other as possible, hopefully adjacent to one another.

Free space and bad blocks

In addition to file content, there are two other types of blocks on a storage medium. Free space is space available for allocation to files and is made available when files are deleted. Because of block-allocation patterns, free space comes available in blocks that are scattered all over the medium. These blocks must be accessible when space is needed and therefore are usually linked together using the same methods used to link file blocks together. Contiguous or linked methods apply to free blocks as well.

Occasionally, a block 'goes bad' – becomes damaged in some way so that it is unusable. A bad block can simply be avoided on a storage medium. As with file blocks and free space, bad blocks are usually linked together so they can be found and avoided.

FAT and VFAT File Systems

When Microsoft developed the first operating system to run on IBM hardware, it needed to invent a file system for the computer's hard drives. In 1977, the FAT file system debuted on IBM PCs using the Microsoft Disk Basic system. This first file system – called the FAT file system for its use of a file-allocation table – is still in use, in more evolved forms, in modern versions of Microsoft Windows. The FAT file system is also use for most mobile media storage (for example, compact flash or multimedia cards).

The initial version of the FAT file system is called FAT12. This first version was very simple and restricted: no support for hierarchical directories, disk addresses were 12 bits long and the disk size was stored as a 16-bit count of disk sectors, which limited the size to 32 MB. The maximum size of a partition was 32 MB.

The release of MS-DOS 2.0 occurred at the beginning of 1983. This version of FAT12 introduced hierarchical directories. The use of directories allowed FAT12 to store many more files on the hard disk, as the maximum number of files was no longer constrained by the root directory size. This number could now be equal to the number of clusters (or even greater, using zero-sized files). The format of the FAT itself did not change. The 10 MB hard disk on the PC XT had 4 KB clusters. If a 20 MB hard disk was later installed and formatted with MS-DOS 2.0, the resultant cluster size would be 8 KB, the boundary at 15.9 MB.

In 1988, with the release of MS-DOS 4.0, the FAT16 file system was finalized. In this file system, the disk address was now 16 bits and the maximum partition size jumped to 2 GB. The maximum cluster size in a FAT16 file system is 32 KB.

FAT12 and FAT16 file systems had what is known as the 8.3 limitation. Filenames on the system were limited to eight characters with a three-character suffix. A variant of the FAT16 file system allowed longer filenames to be used. This variant was known as the VFAT file system after the Microsoft Windows 95 VxD device driver.

The FAT32 file system was introduced in 1996 and is still in use today. The FAT32 file system uses 32 bits for disk addressing and for clusters. Only 28 of the 32 bits are used to address clusters, but even this allows for 2^{28} clusters, which allows FAT32 to support media sizes up to 2 TB. Unfortunately, limitations in other Microsoft utilities mean that the file-allocation table is not allowed to grow beyond 2^{22} clusters, which supports media sizes up to 124 GB. Because of the 32-bit disk address, files can grow to a maximum size of 4 GB. Also the long filenames from VFAT were implemented.

Each of the FAT file-system variants shares common characteristics. They are implemented using a *file-allocation table* with file pieces put together using a linked list of clusters. There is one file-allocation table per disk volume. The FAT has a common structure, illustrated by Figure 8.4.

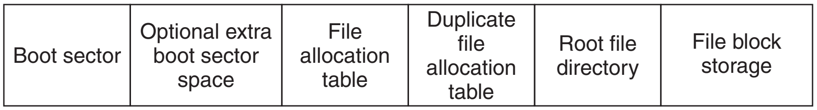


Figure 8.4 The generic format of a FAT file system

The first section of the FAT is known as the *boot sector*. This section of the file system contains the operating system's boot-loader implementation. This is the place that the computer goes to retrieve code that boots the operating system. In some implementations, this is code that initializes the system for operating system execution. In others, the operating system code is stored elsewhere on the disk and the boot sector only contains an address where this code can be found.

There is a section of this format that is made up of optional reserved sectors. This section of the file system is usually used by extensions of the basic boot-sector format. Other versions of the operating system that expand upon Microsoft Windows but are compatible with it use this optional section for an expanded boot sector. The boot sector for Microsoft Windows NT is bigger than that for FAT16, for instance, but since it uses this optional section, the rest of the format can be the same.

The next two areas are identical copies of the file-allocation table for the partition. These tables are maps of the file-block storage areas of the storage medium. They contain one table entry per disk block and each entry holds the address of the next disk block of the file. This chain is started by a directory entry, which indexes filenames and the blocks they start at. The chain is terminated with a special end-of-file value. Free blocks have a 0 value in the table. File-allocation tables also use special values to indicate bad or reserved clusters.

The next area was used by FAT12 and FAT16 file systems and contained the root directory, used as starting point for all traversals through the file system. In FAT32, this directory was located on the disk and an address to it was stored in the FAT.

Finally, the rest of the storage medium is used for file block storage. Blocks are simply laid next to each other and adjacent blocks could easily be from different files. The only way to connect blocks is through the FAT.

Let's take an example. A user wants to open a file and read the first 1000 bytes from a file called `\book\readme.txt`. If we were using a FAT16 file system, we would start our journey by accessing the root directory, which is stored in the boot sector, and we would look up the directory name `book`. This entry would have the block number in the block-storage region of the disk. We would read that block and assume it contained directory information. We would look up the name `readme.txt` (see Figure 8.5) and get the block number of the first file block. Assuming a standard cluster size of 512 bytes, we would need two blocks. We would read the first block from the disk and display its

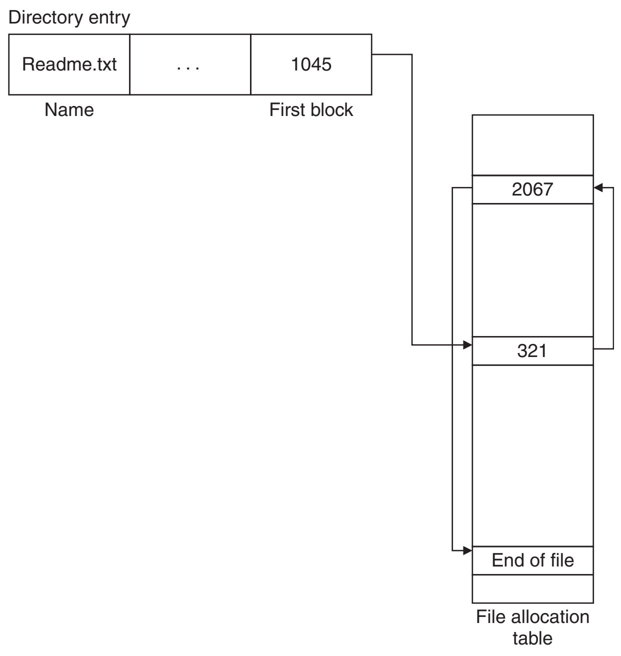


Figure 8.5 Files stored with the file-allocation table

contents. We then need the table entry for the first block to get the second block. We would read the second block and display the remaining bytes.

Many storage media support some version of the FAT file system. FAT12 is used for floppy disks; FAT16 is used on most removable media (USB flash drives, for example). FAT32 is compatible with storage media used by Microsoft Windows 2000 and XP.

NTFS

The New Technology File System – otherwise known as NTFS – debuted with Microsoft Windows NT. It supports many innovations over the FAT16 system used by previous Microsoft Windows versions: compression, file-level security, larger partitions and RAID. In addition, NTFS supports encryption of file-system data. One of the main features of NTFS is exceptional fault tolerance, because it is a transactional file system.

NTFS does away with the file-allocation table and completely changes the way partitions are formatted. A master file table (analogous to the FAT) is stored on disk and an address to it is stored in the boot sector. The boot sector contains code that starts the boot-up process. Most interestingly, the boot sector has as its first bytes a jump instruction that enables a jump to where the bootstrap code is located in the boot sector. This means that data can be large or small and the operating system can always find boot code.

NTFS is the favored file system on Microsoft Windows installations.

Unix File Systems: VFS and UFS

Unix generally uses two different kinds of file system. A *virtual file system* (VFS) is an abstract file system, with abstract interfaces to commonly used system calls. Programmers and applications make VFS calls to access file system facilities. The VFS calls are then implemented by real file-system implementations. Two examples of real Unix file systems are the Unix File system (UFS) and the Network File system (NFS). VFS is effective because Unix is designed to use many different file-system implementations. Implementations of abstract VFS calls are loaded dynamically, and calls are made using a specific type of implemented file system.

The UFS, also known as the Berkeley Fast File system, is used by many Unix implementations. Each partition starts with space reserved for *boot blocks*, addresses that point to operating system code that resides on the storage medium. The next space is called a *superblock*; it contains numbers that identify the partition and parameters that can be altered to tune the behavior of the partition. The remainder of the storage space is organized into groups of fixed size. Each group contains a duplicate copy of the superblock; a group header, containing statistics, pointers to free blocks and tunable parameters; a number of *inodes* that contain information about files; and blocks of data that contain file content.

Inodes are the heart of a UFS implementation. Inodes are a combination of file information – name, ownership, times of access, etc. – and pointers to blocks of file content. In fact, inodes are directly analogous to FCBs. Inodes implement file data pointers in an interesting way. They store 12 *direct blocks*, which point directly to blocks that hold file content. The 13th entry is an *indirect block*, which points to a file block on the storage medium that holds the addresses of other file blocks. The 14th entry is a *double indirect block*, which points to a file block whose addresses point to indirect blocks. The last entry is a *triple indirect block*, which has three levels of indirection built into it.

The indirection built into the design of an inode needs a bit of explaining. If we assume each file block holds 15 pointers, then each inode can access 3 627 file blocks.[1] Consider the space required if each inode held 3 627 file block addresses: 14.5 KB of space to access each file. If the file system were to be filled with predominantly small files, then many of the addresses would go unused. In the inode scheme, with a standard 4 096-byte block, files up to 48 KB in size can be accessed directly from the inode. Files that are between 48 KB and 108 KB take only one more block in the inode, but require an extra file access to read the file block with the actual disk addresses. This scheme favors smaller files, allowing access to large files – up to 4 GB – to have a slight penalty.

Remote File Systems

Servers usually have file systems that they set up for other computers to use over a network. This type of file service allows client computers to use the file systems of servers as if they were local.

A server file system is (obviously) local to the server it is hosted on. It is exported via network protocols to other computers. Two of the most widely used protocols are the Network File Service (NFS) and Server Message Block (SMB) file systems. The former originated on Sun Microsystems SunOS operating system and has been implemented for most other operating systems. The latter originated with Microsoft for its operating systems and has also been implemented for many other operating systems.

The goal of using remote file systems is to allow the user or application not to see the difference between a locally implemented file system and a remote file system. This is where the abstraction of using file APIs is most important. Consider VFS from Unix. Under VFS, the same API is used no matter if the underlying file system is UFS or NFS. The `open()` function is called no matter what. The underlying file system provides a specific implementation of the `open()` function.

Other Interesting File System Implementations

There are many file systems that have been developed over the years that operating systems have used. There have been several recent innovations that are in use. *Log-structured file systems* write every file-system

[1] This is computed as follows: 12 direct blocks; one indirect block, which adds 15 more; one doubly indirect block, which adds 15 indirect blocks, each of which adds 15 more; and finally one triply indirect block, adding 15 x15 x15 blocks. The result is 12 + 15 + 225 + 3 375 = 3 627.

modification to a file, allowing them to be replayed and analyzed. A variant of this type of file system – the journaling file system – is in use in Linux devices (the ext3 file system). Other file systems in use today are the Universal Disk Format system for DVDs and CD-R/RWs and the Hierarchical File System used by MacOS.

8.3 File Systems on Mobile Phones

In terms of file systems, mobile phone operating systems have many of the requirements of desktop operating systems. Most are implemented in 32-bit environments; most allow users to give arbitrary names to files; most store many files that require some kind of organized structure. This means that a hierarchical directory-based file system is desirable. And while designers of mobile operating systems have many choices for file systems, one more characteristic influences their choice: most mobile phones have storage media that can be shared with a Microsoft Windows environment.

If mobile phone systems did not have removable media, then any file system would be usable. In systems that use flash memory, there are special circumstances to consider. Block sizes are typically from 512 bytes to 2 048 bytes. Flash memory cannot simply overwrite memory; it must erase first, then write. In addition, the unit of erasure is rather coarse: individual bytes cannot be erased; entire blocks must be erased at a time. Erase times for flash memory is relatively long.

To accommodate these characteristics, flash memory works best when there are specifically designed file systems that spread writes over the media and deal with the long erase times. The basic concept is that when the flash store is to be updated, the file system writes a new copy of the changed data over to a fresh block, remaps the file pointers, then erases the old block later when it has time.

One of the earliest flash file systems was Microsoft's FFS2 for use with MS-DOS in the early 1990s. When the PCMCIA industry group approved the Flash Translation Layer specification for flash memory in 1994, flash devices could look like a FAT file system. Linux also has specially designed file systems, from the Journaling Flash File System (JFFS) to the Yet Another Flash Filing System (YAFFS).

However, mobile phone platforms must share their media with other computers, which demands that some form of compatibility be in place.

Most often, FAT file systems are used. Specifically, FAT16 is used for its shorter allocation table (than FAT32) and for its reduced need for long files.

Being a mobile smartphone operating system, Symbian OS needs to implement at least the FAT16 file system. Indeed, it provides support for FAT16 and uses that file system for most of its storage media. However, the Symbian OS file-server implementation is built on an abstraction much like Unix's VFS. Object orientation allows objects that implement various operating systems to be plugged into the Symbian OS file server, thus allowing many different file-system implementations to be used. Different implementations may even co-exist in the same file server.

Implementations of NFS and SMB file systems have been created for Symbian OS.

8.4 Security

Security is very important for file systems. Since files are the basic units of storage, it is extremely important that they remain secure and protected from malicious access. The remainder of the file system structure is also vulnerable.

Chapter 14 is dedicated to security. However, security issues are so important that we discuss them here as well – as they pertain to file systems. In this section, we look at the issues with security and outline the attacks and protections that file systems can have.

General Security Issues

Access to a file system and the files it contains needs to be controlled. Allowing any and all access would be a mistake, because it invites malicious activity. However, too many restrictions make file systems cumbersome to use. In addition to the proper security, we also need to decide what elements need security restrictions imposed on them.

When an access is made to a file system, the fundamental assumption is that the access is authorized or permissible. A fine-grained security system would request authorization before each access. A coarse-grained security system would make a single validation that would verify all access. Somewhere between the two extremes lies a system with enough security and a tolerable amount of overhead.

Authorization implies identification. File-system access cannot be authorized for users if those users are not identified. Identifying users is usually done by allowing them to log into a system or otherwise giving a user name or ID. Files are usually tagged with this user ID and specific permissions are given to authorized user IDs.

There are certain users that have all permissions to all files. Most often, these are termed *superusers* or *root users*. These users have all permissions by design (note the assumption that the user has been identified and authorized).

Using remote file systems can be a security issue. Consider the following scenario: user X is authorized to access a collection of files. When that collection of files is shared remotely to another computer system, what happens when user X does not exist on the remote system? Or worse, what happens when user X does exist on the remote system, but is a different user with the same user name?

Typically, identification is verified on the system that the file system comes from before access is granted. This means that identical user names on two different systems would not result in an infraction of security, because verification of the user name (called 'authorization' in Chapter 14) is done on the computer the file system comes from. That verification is unique and done in one place. When there are multiple servers serving up file systems, a centralized server for authentication is often preferred. This can happen through a designated computer on the network; this computer often runs an identity server to verify users.

Security Failures: Flaws and Attacks

Security advances often come from learning by mistakes or finding lapses in security. There have been many security failures since file systems were implemented. An overview of some of these is appropriate before we discuss mechanisms used to protect files.

Operating systems have long allowed access without user identification. In this type of system, there is no specific owner of a file and all access to all files is implicitly granted. With no user identification, there is essentially a single user of the computer. That user controls all system resources, including all files and file access. Most early operating systems – including early versions of Microsoft Windows – were implemented with this type of access.

It is this environment that enabled the creation of viruses. Viruses are fragments of data that are typically added to programs in such a way that they can be executed when the program is executed. This 'infection' is passed from program to program by the executing virus code. Such infection is easy and permitted when no user identification is required to access files.

Sometimes operating systems verify user identification but do not use that identification to regulate access to files. These types of systems are 'gatekeepers': once a user is validated – or passed through the gate – that user may do anything to the system and its data. In these systems, user validation usually serves to personalize the environment for users but is often diluted for security. Recent versions of Microsoft Windows – through to Microsoft Windows 2000 – would set up access to files in such a way that the default access rights would grant all permissions to all users. The result was that, no matter what user you were, you could still access all files and have all privileges.

Most operating systems in use today verify user identification and use that identification for file access. These systems identify the type of access allowed for various classes of users by identifying the user. These systems are as vulnerable as their verification process. If a user can enter a computer system with another user name, for example, then security on files is meaningless. Access assumes authorization; if authorization is compromised, so is file access.

Unix has long had this type of security implementation. As the next section describes, Unix file systems have the notion of user classes, which include 'owner', and users are classified by their user ID when they validate themselves to the system. This allows Unix to classify users further, in groups or as 'other'. Each of these classes has security settings that allow for file read, write and execute operations, with a file owner being able to grant access to its file to particular user classes.

Establishing and enforcing ownership on files is a great way to thwart virus infection. When only an owner can modify a file, then a virus can only infect a file if its executing process is identified as the owner. Sometimes, the owner can grant others 'write' permission by using ACLs or by giving groups of users access. Viruses can infect files in these cases if the executing process is identified as a user having permission to write to a file. Because owners are usually carefully controlled, Unix systems are rarely infected with viruses.

It is useful to note that some computer systems cannot establish user identification and therefore must work to provide other forms of security systems. Smartphones are a great example of this situation. It would be terribly inconvenient for the smartphone user to identify herself before each use (imagine 'logging in' to a smartphone to answer a call).

Protection Mechanisms

Protection of files starts with something outside of a file system: authentication. Authentication is the verification of user identity. As Chapter 14 points out, verification of identity can happen in a number of ways; appropriate verification uniquely and correctly identifies users to the operating system.

Most protection mechanisms verify that file access is permissible by recording several pieces of information with each file. These pieces usually include:

- *ownership*: the user ID associated with the process that created the file; this assumes that the operating system identifies users and can relay that information to files
- *permission specification*: if the owner of a file is recorded, then ownership permission is also recorded; other types of access can be recorded: access for non-owners or for members of the group that the owner belongs to
- *access control*: when permissions are too broad to properly secure a file, access control lists (ACLs) can be used to specify users (rather than groups) and specific permissions can be associated with certain users
- *capabilities*: when access control lists are too broad or require too much detail, users and applications are assigned a specific type of capability (for example, 'file read' and 'file write'). The capability is always matched against the type of access requested. Capabilities are usually more detailed than permissions or access control. For example, 'file delete' might be a capability that is usually not assigned as a permission.

There are many variations on these security mechanisms. Linux uses permissions for file security. In a Linux system, there are three types of user: owners, members of the owner's groups, and others. There are several types of file access in Linux. The most useful access types are 'read', 'write' and 'execute'. The permissions can be used like bits in a number and are assigned to each of the three types of user. For example, a file may have the following listing:

```
-rwx-----x 1 jipping 11001344 Jul 18 15:49 Presentation.ppt
```

The first entry shows the permissions: owners (the `rwx` part) have all access rights, group members (the `---` part) have no access rights and 'others' (the `--x` part) have only execute permissions.

Security on Symbian OS

Smartphone security is an interesting variation on general computer security. There are several aspects of smartphones that make security a challenge. Symbian OS has made several design choices that differentiate it from general-purpose desktop systems and other smartphone platforms.

Consider the environment for smartphones. They are single-user devices and require no user identification. A phone user can execute applications, dial the phone and access networks all without identification. In this environment, using permissions-based security is challenging, because the lack of identification means only one set of permissions is possible – the same set for everyone.[2]

Instead of user permissions, security often takes advantage of other types of information. In Symbian OS v9 and later, applications are given a set of capabilities when they are installed. (The process that grants capabilities to an application is covered in Chapter 14.) The capability set for an application is matched against the access that the application requests. If the access is in the capability set, then access is granted; otherwise, it is refused. Capability matching requires some overhead – matching occurs at every system call that involves access to a resource – but the overhead of matching file ownership with a file's owner is gone. The tradeoff works well for Symbian OS.

There are some other forms of file security on Symbian OS. There are areas of the Symbian OS storage medium that applications cannot access without special capability. This special capability is only provided to the application that installs software onto the system. The effect of this is that installed applications are protected from non-system access (meaning that

[2] This does not mean that applications are barred from requesting user identification. This discussion only addresses system identification.

non-system malicious programs, such as viruses, cannot infect installed applications).

For Symbian OS, the use of capabilities has worked as well as file ownership for protecting access to files.

8.5 Summary

This chapter has examined how operating systems implement file systems and files to store data. We began by examining the basic concepts of files, directories and partitioning. We looked at attributes of files, the name and structure of files and directories, and the operations that can be performed on them. We then looked at how file systems are implemented and gave several examples – from FAT file systems used in Microsoft Windows to the file system used in Unix. We gave an overview of some issues with using file systems on mobile phones and finished the chapter by looking at file-system security.

Exercises

1. Explain the difference between a text file and a binary file on a Symbian OS device.
2. Consider a file that is 2000 bytes long stored on a medium with 512-byte blocks. Assuming that the FCB is in memory, how many storage I/O operations are required to read the entire file in each of the following systems?
   a. a contiguous storage scheme
   b. a linked storage scheme
   c. a FAT implementation scheme
   d. a UFS implementation scheme.
3. Consider a UFS design where there are 12 direct pointers, an indirect pointer, a double-indirect point and a triple-indirect pointer as described in Section 8.2. What is the largest possible file that can be supported with this design?
4. Why is it better to store items from the boot sector – root directory FCB, operating system boot code, etc – on a storage medium rather than in the boot sector?

5. Consider a file system that uses linked allocation for file blocks. What benefits are gained by making these blocks adjacent when the allocation scheme remains as linked allocation?
6. Is it beneficial to locate free space blocks next to each other?
7. When defragmentation is attempted on a storage medium, there are often areas of the medium that cannot be moved. Characterize these areas and describe why they cannot be moved.
8. Consider the situation when a smartphone shares RAM between operating system memory and a file system.
    a. What file-system implementation is likely to be used? Explain.
    b. How can a file system adapt to dynamically changing storage sizes (as operating system needs change, the file space grows and shrinks)?
9. Why could Microsoft Windows 98, which used a FAT16 file system, only support partitions with a maximum size of 2 GB?
10. If VFS and UFS were to be implemented for Symbian OS, would a flash memory card that uses this combination have more or less space than a card formatted with a FAT16 file system?
11. If a new file system were to be invented for the next version of Symbian OS, what improvements could be made over the FAT16 file system?
12. Is fragmentation a problem on the storage medium used by Symbian OS? Why or why not?

9
Input and Output

A computer can do all the computing in the world, but it would be a useless device without input and output. I think there are many people like this. Sometimes people are thinkers; others like to talk (output) a lot without listening (input); still others hear you (input) but do not talk (output) to you. The most pleasant people to be with are typically those who listen, consider what you are saying and reply. Thus it is with computers. While there are few computers that take input while producing absolutely no output and there are no computers that produce only output (even computer-driven clocks need to be set with input), the most useful computers are those with general input and output capabilities.

Operating systems must balance the needs for general computing with the needs to process input and generate output. It is very easy to get these tasks out of balance; proper techniques are required to do all of them at the same time. Management and control of input and output can be a difficult thing.

This chapter discusses what is required to bring a balance to input and output. We have discussed related topics in other chapters, but here we bring the pieces together. We first give an overview of I/O components. We then review I/O hardware and give examples of the concepts needed to manage I/O. Then we look at software issues connected to I/O. We discuss I/O in Symbian OS and conclude with some Symbian OS examples.

9.1 I/O Components

An operating system must manage input and output just like it manages the other resources in a computer system. Input and output are harder resources to manage, however, because of their variety in function and speed. While the CPU of a computer remains the same, any number of I/O devices can be connected and disconnected. Consider a mouse, a CD-ROM drive and a printer. These devices are all different in speed and in the way they are used, yet they must all be managed by the same operating system.

We typically manage complexity by using abstraction. Throughout this book, we have emphasized how operating systems try to deal with the relevant details of something through a standard interface, while ignoring much of the complexity that is not relevant to management. That is the method used here: standard interfaces handled in a standard manner allow I/O technology to expand and differentiate without forcing the operating system to change at the same rapid pace. However, innovation around I/O advances rapidly and standard interfaces are often outpaced by the needs of new devices – thus resulting in new 'standard' interfaces. This, too, must be managed and allowed to happen in a controlled setting.

There are three components that are important to an operating system's management of I/O. Device controllers on the hardware side are linked to device drivers on the software side by buses (see Figure 9.1).

Hardware Interface: Device Controllers

A *device controller* is a hardware component that represents the hardware interface to the operating system. A controller has the ability to operate the device to which it is attached. A controller provides an interface that translates between the complex working of an I/O device and the operating system.

A controller is really a device in its own right. It is typically made up of a processor, some memory and some microcode that enables it to process interactions with an operating system. It is programmed to interact with the operating system using standard interfaces and to translate the abstract interaction it receives through its interfaces into specific operations that accomplish on a device what the operating system is asking. A controller can operate a simple device, such as a serial device, or it can be in charge of a complex system, such as storage media. The controller is responsible

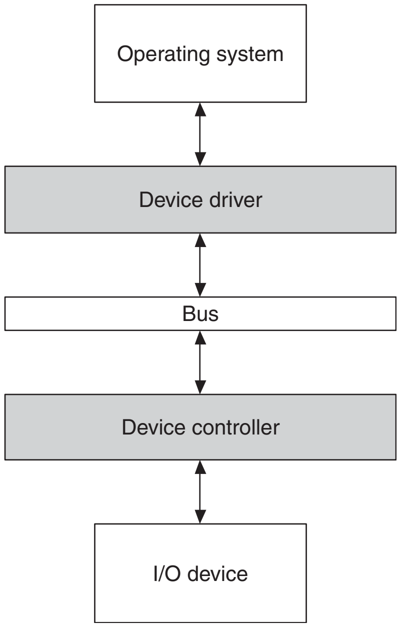


Figure 9.1 Relationship between device drivers and device controllers

for performing tasks that the operating system requires, such as reading, writing, formatting or mapping bad sectors.

Sometimes a controller is not directly connected to its device using an I/O bus system such as USB or Firewire. A controller must sometimes communicate with many devices at once across an I/O bus connection. In all cases, a controller represents the hardware side of the I/O abstraction.

Software Interface: Device Drivers

The software side of the I/O abstraction is a collection of device drivers. A device driver is a piece of software that interacts with the operating system using a standard interface and then interacts with device controllers, providing the software side of the translation between the operating-system and the device.

Since devices are invented and evolve rapidly, supplying device drivers that work in each operating system is typically the responsibility of a hardware manufacturer.[1] When a user attaches a device, that device usually comes with a driver that can be (or sometimes has already been) installed and can handle the new addition. When the operating system needs something done, it communicates with the device driver, which communicates with the device controller, which, in turn, operates the device itself.

Bus Connections

The glue that holds device controllers to device drivers is an I/O bus. At the hardware level, a bus is a set of wires that communicates data – both usable data and control data – between connections. These physical connections tie the device driver to the device controller. At the software level, a bus is seen as a connector that allows messages, requests, and transfer of data between the operating system and an I/O device.

The communication between a driver and a controller happens according to a *protocol*. Protocols are the language of request and service, providing methods to communicate that provide for accurate and error-controlled delivery of data. A protocol can be seen as an agreement between two parties: when one side does something, the other side knows how to respond because both are abiding by a set of rules.

The Small Computer Systems Interface (SCSI) bus in a computer system is a good example here. The SCSI bus is connected to several SCSI devices at the same time. Each device has a SCSI controller that knows how to use the SCSI protocol to communicate with the SCSI device driver for the operating system. This SCSI protocol is very complicated. The SCSI bus has a certain type of design that accommodates this complexity. This design differentiates it from the other buses in a computer system.

It is interesting to note that, as with the SCSI example, some device drivers are tuned to the bus protocol rather than the device they are controlling. In the SCSI example, all devices that connect to the SCSI bus work with the SCSI protocol. This means that device drivers can work with that protocol too, rather than with each individual device. This can be called a physical-device abstraction to logical device.

[1] Sometimes the operating system provides the driver. This happens when standards are clear and devices adhere to these standards. The USB mass storage driver of Microsoft Windows is an example of this.

9.2 I/O Hardware Issues

A computer works with a large number of devices. These generally fit into several categories: storage devices, communication devices, interface devices and display devices. Devices that do not fit into these categories tend to be specialized, such as data-gathering equipment or automobile-monitoring devices. It is typical of a computer to control these devices using a standard interface and a set of generalized commands. Even with all this variety in hardware, just a few hardware concepts are needed to understand how hardware interacts with an operating system.

General Device Communication

As discussed in Section 9.1, operating systems communicate with devices through buses that connect components from device driver to device controller. A general bus structure is shown in Figure 9.2.

A device is connected to a computer system through a bus, but can be connected to that bus in several ways. A device could be connected

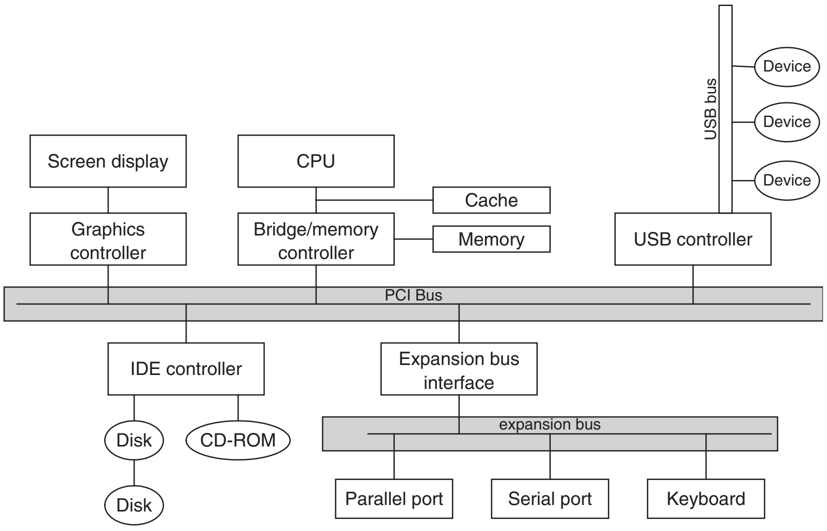


Figure 9.2 Generic I/O bus structure

directly by being plugged into the bus through a slot on the computer's motherboard. This is typical of desktop and server systems. Other devices are connected through a cable, plugged into a port, or open receiver, on the computer. Others are connected wirelessly, for example, through Bluetooth technology. Sometimes devices can even be *daisy-chained* together, when one device is connected to another, which is connected to a third and so forth. Eventually one device in the chain must be connected to a computer.

Communication is initiated by a controller and destined for the operating system or by the operating system and destined for a device through a controller. A simple way that these communications are passed on is through registers. Communication bits and bytes are passed through a variety of registers:

- the *status register* contains a set of bits that are read by both the computer and the bus controller; the bits indicate states such as whether a data transfer is completed or whether there has been a device error
- the *control register* is used to control the data exchange process; when it contains data, that data is in the form of a command which usually amounts to 'read' or 'status', but bits can also be set to indicate how data is to be transferred
- the *data-in register* is read by the operating system to get input data from devices
- the *data-out register* is written to by the operating system to pass data to a device.

Data registers are typically between one and four bytes long. Some device controllers can hold buffers full of data waiting to be sent through these registers to the operating system. The registers are situated either in dedicated I/O space in the CPU or in the main memory. This latter situation is called *memory-mapped I/O*. In this implementation, the CPU writes or reads data to or from the dedicated address or range of addresses of the main memory space.

Device controllers also send data to operating systems through memory-mapped I/O. Memory-mapped I/O uses the register idea but, instead of registers, memory on the processor is used. When the CPU sends an I/O request, it uses standard registers to issue the request but the data ends up in main memory.

These two methods are adequate for small amounts of data but, for larger amounts of data, they require too much movement of data once the data has left the bus. The *direct-memory access* (DMA) approach allows the bus controller to access memory directly and to signal to the operating system when the I/O operation is complete. All I/O functions happen through memory: from initiation of an I/O operation to the arrival of data and the signaling of operation completion. DMA allows the operating system to service other needs and only attend to data when it is signaled.

DMA works with system resources and shares them with the CPU. It accesses main memory just as the processor does. Occasionally, DMA data transfer takes over a resource – for data transfer over the bus, for example, or for depositing data in memory. In these cases, the CPU cannot use the resource while it is in use for DMA. This artifact of DMA is called *cycle stealing* and it can slow down access to computer resources. In the first, conventional DMA cycle, the data transfer uses the bus to transfer all data from or to memory without CPU attention. Since the system bus is in use, the CPU does no external operations. In the cycle-stealing DMA transfer, the system can use those CPU cycles where the CPU does not transfer any data to or from the memory on the system bus. The bus cycles are stolen from the CPU and used by the DMA controller to transfer data.

Polling

To facilitate the transfer of data, the operating system must pay attention to the transfer system. For systems using registers and memory access, this attention amounts to a constant monitoring, a checking that continually watches components in the system and reacts to changes in system states. This constant monitoring is called *polling*.

Polling happens with the register method of data transfer by monitoring the bits in the status register. There is usually a bit, called the *busy bit*, that indicates that data is in the process of being transferred. When that bit clears, the data is completely transferred. Note that, because the processor operates at a speed faster than the data transfer, it is likely that the operating system does a lot of checking before the data transfer is complete. Polling happens on both sides with the register method. The bus controller polls the status register and reacts to various settings. For example, there is typically a *command-ready* bit that indicates that a command is waiting in the control register for the bus controller to use.

In the memory-mapped I/O method, polling still takes place. One might think that the use of memory would mean that registers are not used. However, only the data itself is placed in memory, not the command and status registers. Therefore, the operating system must continually poll the register set.

Polling is usually detrimental to system performance. The constant checking of the operating system must be woven into cycles that the operating system goes through to switch the context of processes. Checking is done regardless of the state of the registers and regardless of whether a data transfer is actually taking place. This constant checking, including useless checking of empty registers, drags down the performance of a system.

Direct-memory access avoids the polling penalty. An operating system using DMA does not have to poll but instead waits on devices to inform it when data is ready. This frees up the operating system to attend to other things and operations such as context switching get more attention and, hence, run faster. Data transfer is attended to when the bus controller signals that data is ready.

Interrupts

Several of the operations mentioned above rely on some sort of signaling between devices and the operating system. This interrupt mechanism is an important part of operating system implementation.

Interrupts usually work through a dedicated hardware wire called the *interrupt-request line*. The CPU checks this line at the end of the fetch–execute cycle, after it has completed an instruction execution. When an interrupt is signaled, the interrupt-request line has voltage on it. When this condition is detected, the CPU diverts execution to an interrupt-service routine (or routines). We discussed the handling of interrupts in Section 3.3.

As they pertain to I/O, interrupts save the CPU from polling. Using interrupts as a way to alert the operating system to when to process data is a major contribution to performance.

9.3 I/O Software Issues

It is the job of system software to shape the view of the hardware for the user and programmer. As usual, this shape is derived by the use of abstraction: the interfaces to hardware are defined so that access is standardized and the implementation of standard access is left to system designers and device-driver writers.

Unfortunately, there is a large variety of 'standard' interfaces, depending on the operating system being used. Device manufacturers must supply a number of device drivers that adapt to the various systems in use. Devices can be characterized in a number of ways, depending on the system:

- *character-stream or block*: character-stream devices deliver data as characters, byte by byte; block devices deliver data in blocks of bytes
- *sequential or random-access*: sequential devices deliver data in a fixed order as specified by the device; random-access devices can deliver data in any order requested by the operating system
- *synchronous or asynchronous*: synchronous devices transfer data in predictable time periods, often coordinated by the system clock; asynchronous devices are more unpredictable, delivering data when it is ready rather than in fixed intervals
- *sharable or dedicated*: sharable devices can be accessed and used concurrently by several processes; dedicated devices communicate with one process at a time
- *speed of operation*: device speeds vary widely, from mere bytes per second to gigabytes per second.
- *read-only, write-only and read–write*: some devices perform both reading and writing operations, while others implement only one of these.

Many times, operating systems hide these differences and present only a few general types to the user. In addition, some operating systems provide a kind of backdoor way to access all devices. So while abstraction is in wide use, there are ways to skirt the abstraction and directly address each device. In Linux, for example, the `ioctl()` system call is a way to pass command data directly to a device, rather than using the standard interfaces.

Kernel I/O Structure

Kernels provide many types of service related to I/O, usually devoting an entire subsystem to handling I/O.

I/O scheduling is important to efficient I/O handling. The order in which applications issue requests for I/O is not usually the best order for the most efficient operations. If we allow an operating system to rearrange the ordering of I/O requests, better performance and better sharing can take place. The kernel usually maintains a queue of requests for each device. When an application issues an I/O system call, that request is queued. The kernel chooses to issue the requests in an order that, for example, minimizes the movement of the hard-disk read arm or can be serviced by a single block read rather than several smaller reads.

Kernels often use buffers to transfer data and smooth the differences between device speeds. Buffering allows the operating system to compensate for the speed difference between the CPU and devices. Data from slower devices accumulates in a buffer until the buffer is full, at which time the faster receiver is notified. Kernels often use *double buffering* by processing the first buffer and allowing the device to fill up a second one. (Displays benefit from double buffering because they can display images without the flickering caused by new buffer retrieval.) Buffers allow data from devices with different transfer sizes to be exchanged. Multiple blocks of data can fill a buffer and then be transferred using different block sizes and the data is removed from the buffer. Buffers also support differences in the concepts that applications have toward data models. One application may model its data one way and that influences how data is copied, read and written. Another application may have a different model and different ways to move data. When these applications exchange data though the kernel, buffering helps to allow each application to use its model without trauma. Finally, buffering also helps with simple data storage. Rings or circle buffers allow producers and consumers of data to go about their functions at different speeds without stopping to ensure the other side is keeping up.

Operating system kernels usually maintain a cache with copies of data used. As we have discussed before, caches are fast memories that are positioned between the kernel and external devices. They act as buffers to make access to data faster than actually retrieving it from the device. Disk caches are a good example: reads from a disk drive look for the data first in a disk cache. If the data is in the cache, the read operation can happen much more rapidly. Write operations happen this way too. Writes to a cache are buffered, with the cache performing the slower write operation and the kernel going on to do other things. The combination of buffering and caching is a powerful way to remedy the disparity of speeds and performance between devices and the CPU.

Sometimes, buffering happens so often that it gets a special name. A *spool* is a buffer that holds the output for a device – usually a printer – that cannot work with interleaving data streams. Printers can only serve one job at a time, but several applications may want to print at the same time. This issue is solved by buffering the jobs waiting to print in the printer spool. The operating system moves every print request to the spool and the printing system polls the spool. This type of system is used for every device that cannot multiplex between concurrent requests. Tape drives also use spooling.

Sometimes operating systems set up other ways of dealing with multiple requests for the same device. *Device reservation* is a technique that is used, where an application asks the kernel for a device and waits until it can have exclusive access. In this method, polling is used to wait for the device (just the condition spooling was designed to avoid). However, there are times when an application must have more direct control over a device and reservation ensures that the application can get this kind of access.

Errors happen often in the handling of devices. Devices are not always connected in the tight, controlled fashion that other components of a computer are connected. This means that data errors can creep into communication. Other factors, including bus overloading and controller failure, also cause errors. Errors in device communication are usually handled by the kernel and signaled to applications making system calls. Errors usually result in system-call failure, indicated, in same way, to the caller by the return value of the system call. Some systems return a value of '0' if a system call succeeds and '1' if it fails; other systems set external variables (e.g., `errno` in Unix) to reflect the error status of a communication operation. Devices often maintain detailed logs of errors and the reasons for them; kernels can access these logs by using further system calls.

The kernel often keeps a lot of data on the state of devices. Data structures used in the kernel often form a set of tables that maintain state and access information. These tables require a large amount of memory to maintain and much of the memory consumed by the kernel as it is running is filled with data tables keeping track of devices.

Timers

There are many types of devices used with operating systems. Clocks and timers are devices – although they are not considered as such. Most computers have hardware clocks and timers available for applications to use and they are typically used and manipulated in the same way as external devices. These devices are very useful and provide three basic functions:

- the current time of day
- elapsed time
- triggers or alarms that cause interrupts to fire when they expire.

The operating system and applications use these capabilities extensively.

Kernels and applications depend on *interval timers* for their operations. Interval timers can be set and interrupts are triggered when time runs out. Timers can be programmed to automatically reset themselves once they expire and this capability is used to generate periodic interrupts. The system scheduler uses mechanisms such as these to do context-switching. Timers are used to signal periodic flushing of data caches. Device time-outs are managed by the kernel using interval timers.

Usually a computer system has only one hardware clock, but software *virtual clocks* allow an operating system to use many different timers and provide them to applications. To implement virtual clocks, the kernel maintains a sorted list of requested timer events and continually resets the timer for the next queued event requested. When a timer event occurs, the kernel resets the timer for the next earliest time.

Sometimes, applications – or the kernel – need to keep track of time in very fine units. Microseconds are very important in the maintenance of some applications. Computer clocks do not usually keep time in very fine increments, often providing only coarse-grained resolution. It is typical for hardware clocks to provide only up to 60 ticks per second. Sometimes higher-resolution hardware clocks are provided, but they are available only for sampling; they are not integrated into the timer/interrupt system as the standard system clock.

Blocking and Nonblocking I/O

Kernels deal with the speed disparity between CPUs and devices in many ways (we have already discussed several). One way is to provide a choice to the user: should access to a device block while data is transferred or remain unblocked and asynchronous?

A blocking system call suspends the process making the call until the system call can complete. Completing the system call means retrieving or writing the appropriate amount of data from or to the device being used. Blocking system calls remove a process from the running or ready queues and place it into the waiting queue: the process is blocked while waiting for a device to complete an operation. Once the system call has completed, the process is returned to the running or ready queue (depending on the semantics of the operating system) and execution resumes. Most operating systems use blocking I/O calls for applications; it is easier to understand and deal with blocking application code.

A nonblocking system call is implemented by checking the status of the devices referenced in the call. If the device is ready to perform the operation, the call executes that operation and returns. If the device is not ready, the call returns immediately with an error code.

Another version of a nonblocking system call is an asynchronous call. The call results in a request made to the device being used, but returns immediately. When the requested operation is complete, the process is somehow notified. This notification could take the form of a software interrupt or event or could result in the operating system calling a specific function defined within the process.

Blocking calls are typical when dealing with reading and writing from files. The delays are not burdensome and the semantics of reading and writing make more sense if blocking I/O is used. On the other hand, working with user interface devices – such as a touch screen or a mouse – is most effective with nonblocking I/O calls. Notifying a process when the screen is touched means the system can work on other things and deal with user input only when that input is ready.

Multithreaded applications are most effective with I/O calls. While the system waits for I/O to happen, other parts of the application can continue executing.

9.4 I/O in Symbian OS

As an example of I/O architecture, let's consider the way Symbian OS handles input and output. The goal here is not to give you a complete tour through the I/O structure of Symbian OS, but rather to give examples of the discussion above.

Device Drivers

In Symbian OS, device drivers execute as kernel-privileged code and give user code access to system-protected resources. As we discussed, device drivers represent software access to hardware.

A device driver in Symbian is split into two levels of drivers: a *logical device driver* (LDD) and a *physical device driver* (PDD). The LDD presents an interface to the upper layers of software while the PDD interacts directly with the hardware. In this model, the LDD can remain consistent over a specific class of devices, while the PDD changes for each device. Sometimes, if the hardware is fairly standard or common, Symbian OS also supplies a PDD.

Consider an example of a serial device. Symbian OS defines a generic serial LDD (`ECOMM.LDD`) that defines the user side API for accessing the serial device. The LDD represents an interface to the PDD, which provides the interface to serial devices. The PDD implements buffering and the flow control mechanisms necessary to help regulate the differences in speed between the CPU and serial devices. A single LDD – the user interface – can connect to any of the PDDs that might be used to run serial devices.

LDDs and PDDs can be dynamically loaded by user programs if they are not already existing in memory. Programming facilities are provided to check to see if loading is necessary.

Kernel Extensions

Kernel extensions are device drivers that are loaded by Symbian OS at boot time. Because they are loaded at boot time, they are special cases that need to be treated differently from normal device drivers.

Kernel extensions are built into the boot procedure. These special device drivers are loaded and started after the scheduler starts. They implement functions that are crucial to operating systems: DMA services, LCD management, bus control to peripheral devices (e.g., the USB bus). These are provided for two reasons. First, it matches the object-oriented design abstractions we have come to see as characteristic of microkernel design. Secondly, it allows the separate platforms that Symbian OS runs on to run specialized device drivers that enable the hardware for each platform without recompiling the kernel.

Hardware Abstraction Layer

Symbian OS uses abstraction in many ways; the hardware-abstraction layer (HAL) is a great example of this. The HAL is a set of variables and functions that access system configuration and attributes.

The API consists of a series of C++ enumerations and handler functions that manage the group attributes. Some HAL groups correspond to specific hardware devices, such as displays or keyboards, while other groups access general platform parameters, such as media-driver information and timers.

Direct-Memory Access

Device drivers frequently make use of DMA and Symbian OS supports the use of DMA hardware. DMA hardware consists of a controller that controls a set of DMA channels. Each channel provides a single direction of communication between memory and a device. Using the DMA hardware, bidirectional transmission of data requires two DMA channels. At least one pair of DMA channels is dedicated to the screen LCD controller. In addition, most platforms provide a certain number of general DMA channels. Once data has been transferred, a system interrupt is triggered.

The DMA service provided by DMA hardware is used by the PDD – the part of the device driver that interfaces with the hardware. Between the PDD and the DMA controller, Symbian OS implements two layers of software. There is the software DMA layer and a kernel extension that interfaces with the DMA hardware. The DMA layer is itself split into a platform-independent layer and a platform-dependent layer. As a kernel extension, the DMA layer is one of the first device drivers to be started by the kernel during the boot procedure.

Support for DMA is complicated for a special reason. Symbian OS supports many different hardware configurations and no single DMA configuration can be assumed. The interface to the DMA hardware is standardized across platforms, and is supplied in the platform-independent layer. The platform-dependent layer and the kernel extension are supplied by the manufacturer, thus treating the DMA hardware in the same way that Symbian OS treats any other device: with a device driver in LDD and PDD components. Since the DMA hardware is viewed as a device in its own right, this way of implementing support makes sense because it parallels the way Symbian OS supports all devices.

Storage Media

Media drivers are a special form of PDD that are used exclusively by the file server to implement access to storage media devices. Because smartphones can contain both fixed and removable media, the media drivers must recognize and support a variety of storage. Symbian OS support for storage media includes a standard LDD and an interface API for users. The file server in Symbian OS can support up to 26 different drives at the same time. Local drives are distinguished by their drive letter.

Some Last Notes

Symbian OS deals with blocking I/O in an interesting way. The designers realized that the weight of all threads waiting on I/O events affects the other threads in the system. To alleviate this, and to enable other things to be done, Symbian OS provides active objects. These specialized threads, covered in detail in Chapter 4, allow blocking I/O calls to be handled by the operating system, rather than the process itself. Active objects can be coordinated by a single scheduler implemented in a single thread. By combining code, which would otherwise be implemented as multiple threads, into one thread, by building fixed entry points into the code, and by using a single scheduler to coordinate their execution, active objects form an efficient and lightweight version of standard threads.

When the active object uses a blocking I/O call, it signals the operating system and suspends itself. When the blocking call completes, the operating system 'wakes up' the suspended process and that process continues execution as if a function had returned with data. The difference is one of perspective for the active object. It cannot call a function and expect a return value. It must call a special function and let that function set up the blocking I/O, but return immediately. The operating system takes over the waiting.

Removable media poses an interesting dilemma for operating system designers. When a Secure Digital card is inserted in its reader slot, it is a device just like all others. It needs a controller, a driver, a bus structure, and probably communicates with the CPU through DMA. However, the fact that the user can remove the card is a serious problem to this device model: how does the operating system detect insertion and removal and how should the model accommodate the absence of a media card? To get even more complicated, some device slots can accommodate more than one kind of device (e.g., an SD card, a miniSD card (with an adapter) and a MultiMediaCard all use the same kind of slot).

Symbian OS starts its implementation of removable media with their similarities:

- all removable media can be inserted and removed

- all removable media can be removed 'hot', that is, while it is being used
- each medium can report its capabilities
- incompatible cards must be rejected
- each card needs power.

To support removable media, Symbian OS provides *software controllers* that control each supported card. The controllers talk to device drivers for each card, also in software. A socket object is created when a card is inserted and this object forms the channel over which data flows. To accommodate the changes in the card's state, Symbian OS provides a series of events that occur when state changes happen. Figure 9.3 shows the states that are possible with media cards. Device drivers are configured like active objects to listen for and respond to these events.

9.5 Summary

This chapter has provided an overview of issues for operating systems regarding input and output. We started by looking at three basic units: controllers, device drivers and buses. We took a hard look at hardware issues, then software issues. We wrapped up the chapter by looking at how Symbian OS deals with devices.

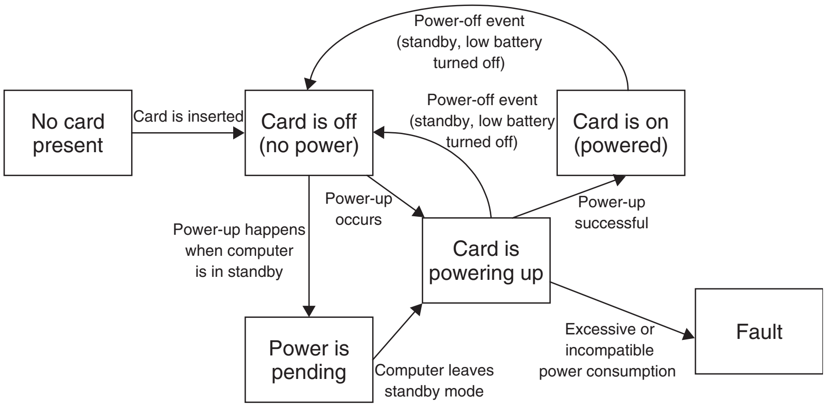

Power-off event (standby, low battery turned off)
No card present
Card is inserted
Card is off (no power)
Power-off event (standby, low battery turned off)
Card is on (powered)
Power-up occurs
Power-up successful
Power-up happens when computer is in standby
Card is powering up
Fault
Excessive or incompatible power consumption
Power is pending
Computer leaves standby mode

Figure 9.3 Possible power states of removable media on Symbian OS

Exercises

1. Evaluate the advantages of using the three-part structure we discussed: device driver, controller and bus. Give at least two advantages and two disadvantages.
2. Consider the following I/O scenarios and describe the role of buffering, caching and spooling in each of them:
   - mouse with a graphical user interface
   - keyboard
   - hard disk drive with user files
   - USB flash drive
   - graphics card directly plugged into a bus.
3. Describe the sequence that takes place when an operating system wants to write one 1024-byte block to a disk drive. Use the three methods we discussed: direct, memory-mapped and direct-memory access.
4. Give three specific scenarios when blocking I/O should be used.
5. Give three specific scenarios when blocking I/O should not be used.
6. Why would you not want to use polling to manage a keyboard?
7. We mentioned in the chapter that interrupts are polled in the fetch–execute cycle of the processor. Is this the only place for them? Could you poll for interrupts after context switching?
8. What would be the advantage of ignoring interrupts from devices? Would we ever want to do this?
9. Does DMA increase or decrease system concurrency? Why?
10. In Symbian OS, kernel extensions are special device drivers. What is special about them? Why must they be treated specially?

10

Networks

Communication is a powerful tool. People who seem very productive alone are empowered when they communicate with others. Communication enhances and expands the ways people work. The ability to talk and interact with others is a very effective tool.

It is like this with computers and networks. By themselves, computers are powerful and can do many effective tasks. When they communicate with other computers over a network, however, the potential of computers grows and communication enhances and extends their effectiveness. Operating systems that embrace networking extend their reach and enable users to use more resources to their advantage.

In this chapter, we examine how networks extend the effectiveness of operating systems with respect to CPU processing, memory, file systems, and I/O. We conclude the chapter by taking a specific example from Symbian OS.

10.1 Opening a Closed Environment

In a closed environment, with no network, computers depend only on their own components and have no opportunities to use or cooperate with other computers.

The components that a computer uses are the ones we have discussed in this book as needing management by an operating system. The CPU and execution capability, memory, data storage media, and I/O all need management if they are to be shared between users and processes that each want exclusive access. We have spent much time so far in this book on describing ways to adequately share these resources.

Let's review the needs that we have seen for these components. Managing the CPU has required us to invent the concept of a process and to build the idea of movement through process states. We have discussed various features of processes, including the ability to communicate between processes and the need for processes to share the CPU through the use of a scheduler.

The use of memory has brought us ideas of how to share that memory with processes. Concepts of logical and physical memory, memory pages, and demand-driven virtual-memory paging are all attempts to share memory appropriately and fairly. We have discussed the protection of memory areas as necessary when more than one process leaves its data in memory.

The need for and use of data storage has caused designers to invent concepts for its management. Files were invented as the basic unit of storage; directories were invented to organize files on a storage medium. File systems have evolved as ways to organize and maintain the information about files and directories and to manage access to those files and directories. File systems are implemented by enabling direct access to the storage medium.

Input/output concepts have developed to address the need for computers to accept, produce and display data. The sheer number of I/O devices and the speed differential between these devices and the CPU has made abstraction a necessary tool in the handling of I/O. Device drivers and device controllers allow us to isolate the important workings of a device into a standardized interface while implementing that interface in ways tailored to each device.

As we have developed ways for operating systems to address these components, there are several common threads that have stood out. Abstraction is a concept used on many levels in an operating system. The presentation of only those details that are needed, coupled with the hiding of implementations that do not address the user's needs has been a hallmark of operating system design. Enabling the various components to be shared is another aspect of computer systems that is common for all components. Finally, communication has been prevalent among all the components involved in running a computer system.

Communication networks have extended the CPU processing, memory, data storage, and I/O capabilities of computers. As operating systems have been developed to address such extensions, they have used abstraction, sharing and communication to facilitate design.

10.2 Extending Computers in a Connected Environment

Networks have allowed the extensions of computers in many ways. The nature of these extensions is heavily influenced by the interconnections that are possible over the network. We consider the way that communication networks operate first, then apply those networks to the components discussed in Section 10.1.

The Nature of Interconnection

There are many ways to communicate electronically and many types of network that can facilitate communication between computers. Despite this variety, interconnections have many things in common.

Networks start with a communications medium. This medium is the method used to connect the computers within the network. Many possibilities exist today. Local area networks (LANs) are networks contained in small geographic areas, such as a single building or a neighborhood. LANs usually are connected with wires through twisted-pair or fiber-optic cables. Wireless LANs have grown in popularity, connecting computers through a central transmitter-access point. Bluetooth has also recently emerged as a radio-based, wireless communication medium. Bluetooth is a protocol that enables communication between radio transmitters.

Networks have a *topology*, which is the set of characteristics that describe how computers in the network are physically connected. These topologies (see Figure 10.1) are influenced by the communication medium and can be characterized by the following criteria:

- *connections*: the number of physical connections and the number of computers connected; connections contribute to issues such as routing of messages and the way that storage is managed
- *installation*: topologies can be vastly different in the ways they are physically connected and the initial effort required to make those connections
- *cost of communication*: the time and money it takes to communicate over the network influences and characterizes the topology
- *availability*: topologies can make data more susceptible to failures or can make access more resilient.

Networks are characterized by the *connection strength* between computers. *Loosely coupled* connections offer networks where not all computers are required for the network to function and the connectivity of computers on the network and between themselves can come and go at will. Examples of loosely coupled connections include mobile devices that share data across mobile networks and web browsers using web services. *Tightly coupled* networks are those with computers who are closely dependent on each other and the operating system on each is dependent on the presence of others. Examples of tightly coupled systems include mobile phone networks and the Internet itself. Loosely coupled networks tolerate failure and reconfiguration much better than tightly coupled networks.

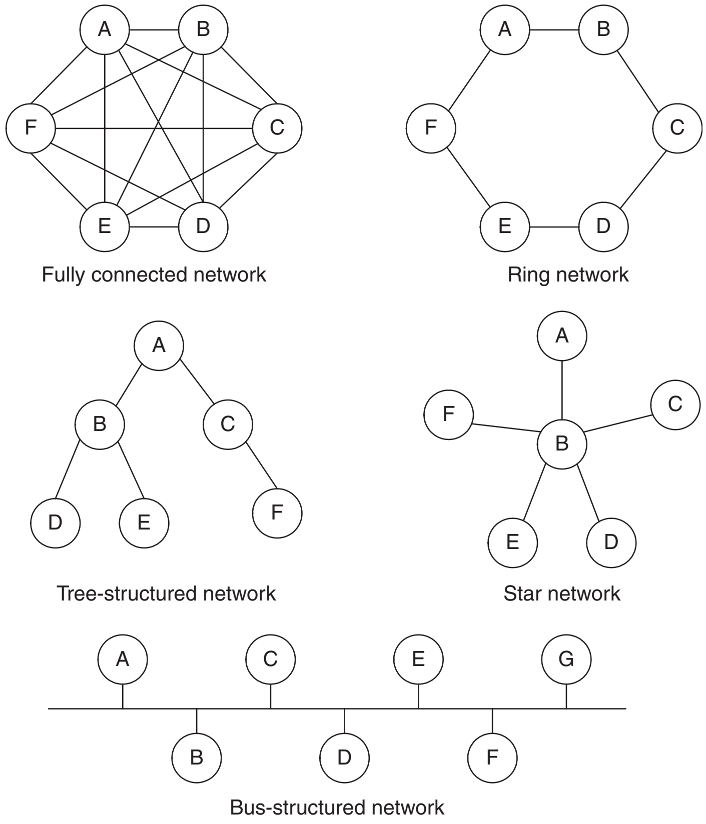


Figure 10.1 Network topologies

The protocols that run over network connections are part of what makes networks unique. A protocol is the exchange of meta-information over a network that makes it possible to send real content. Protocols are as varied as the types of network. The token-ring protocol is designed for circular networks, such as the ring network shown in Figure 10.1. TCP/IP is a network protocol for LANs, which was originally designed to run over a bus topology as shown in Figure 10.1, but has been run over other topologies as well. The Bluetooth protocol is designed to allow Bluetooth peer-to-peer communication. The topology of a network deeply influences the choice of protocol.

Protocols can co-exist on many types of network. Where networks are passive – that is, they transfer any and all data sent over them – multiple protocols can co-exist because the medium tolerates any kind of data. Where networks are active or highly specialized, the base protocol can run other protocols by treating those protocols as applications that are simply sending data. This *protocol encapsulation* approach has several applications.

The choice of protocol for a network also dictates how computers are addressed across a network. Addresses are required on a network so that computers can specifically address other computers. Addressing differs from network to network, depending on the needs of the protocol. TCP/IP networks have IP addresses in the form 192.168.1.250 while Bluetooth addresses are of the form 01:33:FA:72:45:1B. The protocols can decipher the meaningful parts of the address and use those parts to make sure information arrives at its destination.

The combination of addressing and unique protocols provides networks with ways to route data. Token rings pass data between computers by sending the data around the ring, allowing it to pass through intermediate computers before arriving at the destination. TCP/IP routing is a combination of local area broadcasting and 'next-hop' routing, where data not destined for the local network is channeled through a special computer – called a router – that sends the data on to either the next network or the next router.

Distributed Processing

Networks extend a computer's CPU power by enabling distributed processing. Processing can be distributed across a network by distributing the operating system, distributing the computing power, or both.

Distributing the operating system can be as simple as having multiple computers with their own operating system on a network or as complicated as having a single operating system that governs many different computers. While these approaches sound the same, their implementations are quite different. Cooperating network-based computers are quite common and usually share their resources in loosely coupled ways. It is not uncommon for a computer on a network to share its file or printer resources. In this type of loose distributed environment, the operating systems must extend the implementation of resources while the abstraction of resources remains the same.

One way multiple computers share CPU resources is through *clustering*. A computing cluster is a collection of computers which have their own operating systems but implement message passing between network nodes. This message passing includes the allocation of workloads to each computer. In this scenario, each computer is independent but works with the others. Each accepts commands from a central computer in addition to accepting workload allocation from other nodes in the network. In order for clusters to work well, the interconnection mechanisms must be fast and reliable. Each computer on the network is independent and relies on the communication medium to make cooperation work.

One popular way to cluster computers for distributed processing is a Beowulf[1] cluster, a group of desktop computers (usually) running a Unix operating system. They are networked into a LAN and have libraries and programs installed which allow processing to be shared among them. Beowulf clusters originated in technology created by the United States National Aeronautics and Space Administration (NASA). There is no particular piece of software that defines a cluster as a Beowulf. Commonly used parallel processing libraries include the message-passing interface (MPI) and parallel virtual machine (PVM) APIs. Both of these permit the programmer to divide a task between networked computers and collect the results of processing.

In some configurations, computers are governed by a single operating system. This type of implementation involves tight interdependence on each other. In these environments, copies of the operating system run on all computers and these copies are coordinated by messages sent between them. This cooperation orchestrates the sharing of workload and manipulates the computers in such a way that computation is divided amongst all of them. A good example here is a now-defunct

[1] The name comes from the main character in the Old English epic *Beowulf*.

operating system called DomainOS. DomainOS was implemented across a collection of computers, each of which ran the entire operating system. Each computer, by itself, could act independently when alone, but shared workload when it sensed others on a network with it. Even a simple command might run on another computer in the network.

Computers on a network interact as servers and clients. The most typical way for computers to cooperate on a network is via a client–server relationship, where the client makes requests that the server receives and services. Most distributed computing is done via message-passing, where each computer acts as a *peer* – that is, as both client and server. Each sends requests and each can receive requests.

Distributed systems can work together in several ways:

- *single instruction, multiple data (SIMD)* occurs when multiple computers run the same instructions in the same sequence and in synchronization with each other; each computer works on a different data set or data stream; this is a tightly coupled, very dependent environment
- *multiple instructions, single data (MISD)* environments are usually called *vector processors*; each processor works on data and passes the stream to the next processor, which does its work and passes the stream to the next processor, and so forth; MISD processors each run a minimal operating system and accept their multiple instructions from a central source
- *multiple instructions, multiple data (MIMD)* environments are loosely coupled environments where the processors cooperate yet work quite independently; networks and clusters represent this type of environment.

There is, of course, the SISD environment: the single instruction, single data set environment. From a distributed processor point of view, this is a single computer without any distribution at all.

Sharing Memory

Communication can also expand an operating system's view of memory. Shared memory can take several forms, each expanding operating systems in different ways.

One way to share memory is to have a large area of RAM accessible by multiple processors. In this situation, the processors probably also use caches to speed up access to recently manipulated data, but now this cached data needs to be flushed much more often, as it needs to be accessible to all computers in the environment. Unless caches are flushed rapidly, coherence problems spring up: processors might not be working on the same instance of information. Additional hardware is required to resolve this race condition.

Another way to distribute memory is to allow each computer in the network to have its own memory, but all processors can access a large, conceptually shared memory. This shared memory can be a physical-storage medium or can be distributed amongst all the processors. In the former case, cache flushing and synchronized access become very important. In the latter case, the abstraction of having a large memory is implemented by passing requests for memory between processors.

In all cases of shared memory, *memory coherence* is an important issue. Memory coherence has the same issues that multiple processes have when updating memory: computers that access shared, distributed memory must be properly synchronized. In the distributed case, however, proper synchronization involves a lot of message passing to ensure memory updates.

Networked File Systems

Networks extend file systems by implementing *distributed file systems* (DFS). A distributed file system adds a new implementation under the typical abstraction of a file system. A DFS is certainly implemented differently from a local file system but it looks the same to the user of the file system.

In order to be shared, a file system must be implemented and housed on a computer. This *file server* would be the source of the files and would provide the interface through which other operating systems would access them. Other operating systems that need to access such shared file systems are the clients in this arrangement. As we have discussed before, abstraction is heavily used here, which enables the file server to make file service available from a number of different sources. Any number of file systems – with any number of local implementations – could be shared through a file-service interface.

To maximize the abstract qualities of distributed file systems, naming is an important issue to consider. There are two properties of name mappings for a DFS that we need to pay attention to:

- *location transparency:* the name of a file does not reveal the location of the file's physical storage
- *location independence:* the name of the file does not need to be changed when a file's physical location changes.

Most DFS implementations provide for location transparency but not for location independence. Consider, for example, the way that Microsoft Windows maps a file. Windows associates a drive letter with a file system from a local storage medium or a DFS, providing location transparency. It is impossible to tell from the name whether `E:\Program Files\Warrant\playit.exe` comes from a local drive or a file server over the network. However, for a file to change its location – that is, its drive letter – a user must disconnect the drive letter from the file system, then remap the drive letter to a new file system. It is not possible for a file to migrate between storage locations in Windows and not notice that migration in the DFS implementation.

Storage-free computers

Distributed file systems make it possible to get files from servers housed remotely on a network. What would happen if a system were to get all its files remotely from file servers?

Such diskless computers are in wide use today. They have no disk-based storage of their own; they get all their storage from servers on a network. This configuration has ramifications for operating systems. For example, where is the operating system stored if there is no local storage? Usually there is a small amount of boot code stored locally in the device's ROM that directs the system to find the remainder of the operating system code on a file server. Security also becomes a very important issue here. Not only is user security important but machine security is important. If a computer can masquerade as another on the network, it can boot as the other computer and access its files.

As with memory, caching is a way to speed up the relatively slow-paced storage I/O. And, as with memory, caching poses a problem with shared storage: caching must be flushed rapidly to provide consistency between all the computers that are sharing data. This flushing is controlled by the operating system, which can adopt any of a number of different policies. *Write-through caching* is the simplest approach, where data is written to the storage medium immediately it is placed in the cache. This is a reliable approach but has poor writing performance, as cache writes must endure the latency of writing to a remote file server. *Delayed-write caching* delays writing to the remote file server until multiple writes can happen or latency can otherwise be minimized. Delayed writes are better in that writes can happen more quickly (on average), but these schemes can have reliability problems, since unwritten data can be lost if the local computer crashes. *Write-on-close caching* writes data when the file is closed. This speeds up access even more, allowing for recent writes to overwrite old data in a cache and saving all slower access for times after a file is not being accessed. However, files that are open for longer periods of time suffer coherency problems from this scheme.

File replication is a way for remote file servers to increase performance. When file servers replicate files between many locations, multiple clients can choose where to get files from (presumably the closest server for fastest access). Again, abstraction of replication details away from the user means that the user has no idea where a file comes from. Allowing the operating system to make the choice adds overhead to the implementation as well. The operating system must keep track of performance measures, such as 'distance' of a file (measured in time taken to obtain it) and response time of the file server. In addition, updating a file in this environment is even more of a problem than with caching. Replicating a file means that the version of a file written to a file server must immediately be replicated and sent to all file servers holding duplicate copies. This action takes time, yet the abstraction used means that file updating is immediate. This situation is often remedied by influencing clients to get updated copies of a file from the file server on which it was last updated until all the copies have propagated.

A *stateful file service* is one where the file server traces each file being accessed by a client, keeping track of file positioning and how much data has been read from that file. A *stateless file service* is one where any block on a storage medium can be requested and sent without any further semantic understanding as to the organization of the block in a file. There are two big differences between these two services. First, this determination affects which side keeps track of the file position for operations. If the file server maintains all the information about a file and which data in the file has been read or written, the client does not have to and the protocol used can be minimized. In a stateless environment, the client is responsible for maintaining an understanding of files and how much data has been read. The other difference is in reliability. If a stateful server crashes, all information about the file and its updates are lost.[2] If a stateless server crashes, minimal information is lost and service can simply pick up where it left off when the server comes back online.

While stateless service might seem to be more robust, stateful service might be warranted under certain situations. For example, some implementations of file replication implement cache writes and validation upon the server's request. In this case, stateless service cannot be used, since the server would maintain knowledge of the files being updated and would have to determine when to request cache writing.

Implementations of DFS

There are several DFS implementations available for modern operating systems. By far, the two most widely used are Server Message Block (SMB) and Network File Services (NFS).

SMB is a protocol for sharing files, printers, serial ports and communications abstractions such as named pipes and mail slots between computers. It debuted in 1985 as a sharing protocol from IBM and was further developed by Microsoft as the Microsoft Networks/OpenNET-File Sharing Protocol from 1987. SMB has seen several different implementations: Samba is an open-source implementation and CIFS is the most recent implementation of the protocol by Microsoft.

NFS is a protocol originally developed by Sun Microsystems, debuting around 1984. NFS is one of many protocols built on the Open Network Computing Remote Procedure Call system (ONC RPC).

Other DFS implementations include the Andrew File System (AFS), designed for the Andrew distributed-computing system at Carnegie Mellon University and AppleTalk, a sharing protocol designed for MacOS.

Communicating with Networked Devices

Networks have extended input/output mechanisms by allowing remote devices to communicate in abstract ways and to share any kind of information. There have been several extensions to generalize I/O using networks.

General communication has been enhanced by the use of the socket network abstraction. As we described in Chapter 6, sockets were invented by the designers of Berkeley Unix as 'endpoints for communication'. As an

[2] There are systems that can prevent a stateful crash from losing data. Fault-tolerant systems implement continuous update and refreshing of file systems.

endpoint, a socket is not very useful. When connected to another socket on another computer, the pair becomes a communication channel that uses a protocol to transfer data. Sockets are two ends of a conversation and the socket protocol is the translator.

The beauty of the socket model is in its abstractness and its translation abilities. The abstractness of the model can be seen in how it is used: each side simply writes data to and reads data from a socket as if it were any other local I/O device. The socket may implement translation of data, without each side knowing. The translation is implemented by the operating system and occurs as the data is transferred between the endpoints.

The socket, then, provides an abstract extension to I/O by implementing a generic I/O device. Specific implementations demonstrate this. A network socket, for example, is used for many different protocols. One can open a TCP layer socket that exchanges data with a web server or a UDP socket that communicates with an NTP server. Both TCP and UDP layers are handled automatically and the network properties are preserved in the implementation. Bluetooth sockets work in the same way; Bluetooth sockets can implement connections that look like serial cable connections or some generic data transfer connection.

10.3 Connectivity in Symbian OS

Like many other general-purpose operating systems, Symbian OS embraces networks and allows implementations that use networking to expand its core functionality. Symbian OS supports many kinds of communication. Since it is a smartphone operating system, it naturally supports telephony. It also supports many other communication technologies, including Bluetooth, wireless networking, Ethernet, infrared, and messaging protocols. In addition, it allows for the implementation of new technologies that might be added in the future.

Distributed computing is supported by Symbian OS in the sense that the operating system supports general user applications. There is no real distributed computing built into Symbian OS, but the operating system can execute any type of general application, including those that implement clustering or other forms of distribution. In fact, Beowulf clusters have been implemented using Symbian OS phones using Ethernet protocols over GPRS networks.

Symbian OS does not specifically support shared memory models and is unlikely to do so any time soon. Symbian OS is targeted to smartphones which are designed to work in isolation.

When it comes to file systems, Symbian OS has some unique features that allow the expansion of file-system implementation. The file server that Symbian OS implements to protect file resources utilizes a plug-in architecture to recognize file systems. Since Symbian OS has a wide variety of communications choices, implementing a distributed file system is a matter of writing the proper plug-in for the file server and allowing Symbian OS to recognize it. Many experiments have been performed on file systems for Symbian OS; NFS is among the implementations that have been used for the operating system.

Symbian OS has a rich set of implementations for network-based I/O. One of its basic communication structures is the socket, which is implemented for each of the communication possibilities it supports. Through the use of sockets, Symbian OS supports many protocols, including TCP/IP, Bluetooth, WAP, HTTP and many others.

10.4 Summary

This chapter has discussed the ways that network communication can be used to extend the functionality of operating systems. We introduced the facilities important to a closed computer system and discussed how networking could extend a closed system. Specifically, we discussed four areas of extension: CPU processing, memory sharing, file service, and I/O. We concluded the chapter by briefly looking at the extensions that networks can provide to Symbian OS.

Exercises

1. List at least five different aspects of operating systems that access networks. Include the functionality of the operating system feature that is extended by network access.
2. Consider the network topologies in Figure 10.1. Which topologies would you judge to be the most reliable? Justify your statements.
3. List at least three different communication media that computers around you use. For each one, consider how it fulfils the criteria

from Section 10.2 (connections, installation, communication cost and availability).

4. For each communication medium listed in exercise 3, characterize the connection strength between the nodes on the network.
5. Give examples of each of the distributed-system models.
6. Can handheld computers or smartphones be considered 'storage-free'? While most do not have hard disk drives, most use storage. Do they use network storage as well? Find examples of some that do.
7. Why is caching such an important aspect of distributed file systems?

11

Modeling Communications

The game of ‘telephone’ (sometimes called ‘Chinese Whispers’) is an interesting children’s game. This game is played by telling a child a sentence and allowing that child to tell the sentence to another child, who tells it to another child, and so forth. After a number of message transfers, the final message is revealed. This final message is almost always an unrecognizable version of the first message.

In Chapter 9, we took a generic look at I/O modeling and implementations in operating systems. In this book, we are considering smartphone operating systems and therefore we should focus the generic discussion of I/O on communications. This chapter considers generic communications models and specific implementations in various operating systems.

The model of communications resembles the ‘telephone’ game. A popular, effective way to model communications streams is to break the communications implementation into stages where each stage implements a specific kind of functionality. As a message passes through these stages, various levels of functionality are added until the final message bears only a small resemblance to the original. However, in this case, this is a good thing. It allows the message to be passed correctly from computer to computer. We examine how this happens in this chapter.

We should point out that communications are not solely the property of smartphones and operating systems such as Symbian OS. Linux has been used to implement communications facilities; wired network communications saw its beginning on Unix and Linux platforms. All modern operating systems today implement some kind of model for the communications they support – from serial communications to Bluetooth to wireless networking.

We begin with an overview of general communications models – from the general-purpose operating systems model to more focused models used in handheld platforms. Then we look specifically at the communications model used in Symbian OS. We expand on this and finish the chapter by showing how the Symbian OS model works on other computer platforms.

11.1 Communications Models

In order to discuss how operating systems approach communications, we need to apply the I/O concepts from Chapter 9 to communications. Once we understand how general I/O fits with communications, we can then examine how communications is tailored to various needs.

It is important to keep in mind the obvious point that communications involves more than one computer. While we can look at the process model of running programs on a computer as a contained environment on a single computer, developing a communications model means addressing how multiple computers interact. This complicates things because we cannot accurately predict how all components of communications work together. This is especially true when dealing with computers in a heterogeneous environment. For example, Symbian OS might have a precise model for getting web pages, but it will fail if a Windows-based server sends pages in a way that Symbian OS is not expecting. This means that models must expect errors and prepare for flexibility. It also means that models must embrace standards; adhering to standards is the best way to work with other devices.

General Communications Concepts

A general communications model, one that addresses all facets of communications, is an expression of design goals and criteria.

- A communications model must support all communications applications of which the computer is capable. This criterion is perhaps obvious, but it should be stated. Users of communications devices expect a level of functionality from their devices. The communications model must support this functionality and be able to address the future gracefully.

- Communications model components must be exceptionally flexible to cope with the variety of devices that are possible for each platform. Users of computers demand a mix of connectivity methods and communications platforms. The communications architecture must easily adapt itself to the changing requirements of configuration and connectivity. Even on-the-fly reconfiguration, for example, from a dial-up connection to a wireless network, must be easy and straightforward.
- Communications components must be organized to accommodate the constant restructuring and rebuilding of communications technology. The communications architecture must be built in a modular fashion so that pieces can be replaced as technology evolves without upsetting the entire structure. In fact, the structure should be able to accommodate the coexistence of old and new pieces.
- The communications components must adapt to the RAM and CPU constraints of their intended target platforms. While they must do a great amount of work, the components of the communications architecture must not consume a burdensome amount of resources. The resources of a communications device are to be targeted at an application, not consumed by communications methods.

This is a difficult job: support all functionality possible in a flexible, modular fashion in what can be a limited computing environment!

Clients and Servers

The way computers interact – especially in a network environment – is often characterized as a client–server type of relationship. The client is the computer taking advantage of a service; the server is the computer providing that service. The client typically utilizes the service by sending requests to the server; the server provides the service, typically by responding to those requests. The requests and responses take the form of messages sent back and forth between client and server.

We have seen this relationship in many places already. The client–server model describes a relationship like that on which a system call depends, where the ‘client’ is a user-level application and the ‘server’ is the kernel, providing a set of services for the kernel-level request. The client–server model adequately describes the server approach taken by the design of microkernels, where taking advantage of a service means connecting to a server and making a request.

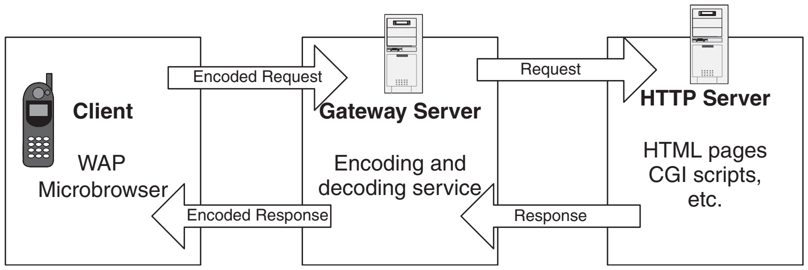


Figure 11.1 A client–server relationship

A good communications example is a micro-browser in a WAP-enabled mobile phone. Figure 11.1 shows how this works. The phone calls up a WML page for viewing by sending an encoded request to a computer at the company that provides the service. This computer is called a *gateway server*, because it provides a gateway to the Internet for the mobile phone. This server, in turn, becomes a client by passing the request from the mobile phone on to the actual Internet server that houses the WAP page in question. This server happens to be providing a HTTP service and answers the request by sending a web page to the gateway server. The gateway server then translates the HTML page into WML and encodes the response using the encryption scheme the mobile phone expects. Finally, the gateway server sends the response to the mobile phone and the micro-browser displays the WML page on the phone's display.

Note that in the example, there were two clients and two servers. The gateway server was a server to the mobile phone and a client to the HTTP server. This is an example of a *proxy server*, a server that represents its client on another network by becoming a client itself.

Communications Stacks

Another way to characterize communications in an operating system is through a *stack-based model*. A stack-based approach to communications recognizes that there are many different levels that analyze and use communications data. Each level has its own functionality and adds its own unique properties to the communications stream.

Let's use as an example the ISO networking model. This model is shown in Figure 11.2. Each layer in the model has a specific duty and

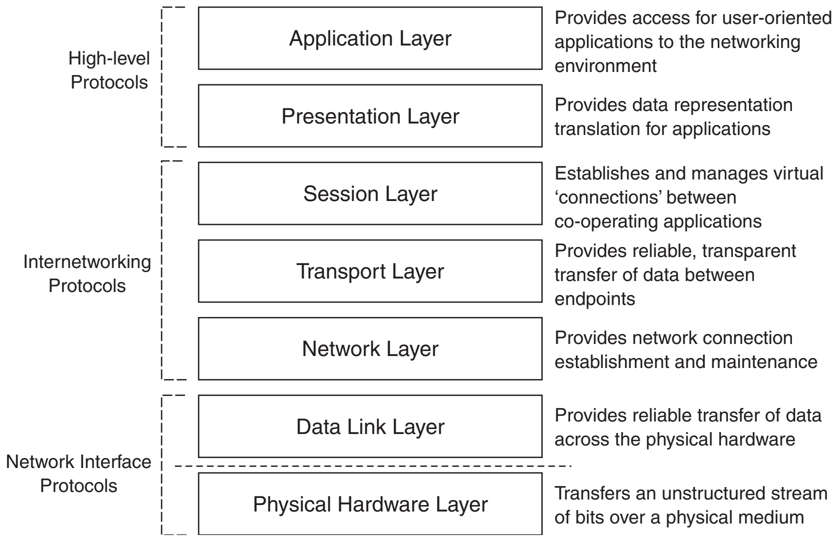

High-level Protocols
Internetworking Protocols
Network Interface Protocols
Application Layer
Presentation Layer
Session Layer
Transport Layer
Network Layer
Data Link Layer
Physical Hardware Layer
Provides access for user-oriented applications to the networking environment
Provides data representation translation for applications
Establishes and manages virtual 'connections' between co-operating applications
Provides reliable, transparent transfer of data between endpoints
Provides network connection establishment and maintenance
Provides reliable transfer of data across the physical hardware
Transfers an unstructured stream of bits over a physical medium

Figure 11.2 The ISO protocol stack model

adds data to the communications stream to carry out that duty. A packet of information at the application level is sent to the presentation layer. The presentation layer augments the data in some way, usually by adding information to the packet, and sends it on to the session layer. This continues until the packet reaches the bottom layer, the physical hardware. At this point, the data stream is sent to its destination and the physical hardware of the destination computer receives the stream. Now the process begins in reverse. As the data packet is received by each layer, that layer strips off the data it needs, using the data to perform some function. Then what is left is sent up the stack to the next layer. By the time the packet reaches the application layer, it is comprised of application data only. The effect of this type of data transfer is that each layer has the illusion that it is talking directly to its counterpart layer on the destination computer.

There are many examples of this way to characterize system communications. The Wireless Access Protocol (WAP) stack, shown in Figure 11.3, has a stack-based depiction. The Bluetooth protocol stack, shown in Figure 11.4, has this type of specification as well. In all cases, regardless of how complicated the protocol stack, the idea of moving through the stack and adding functionality to communications data applies.

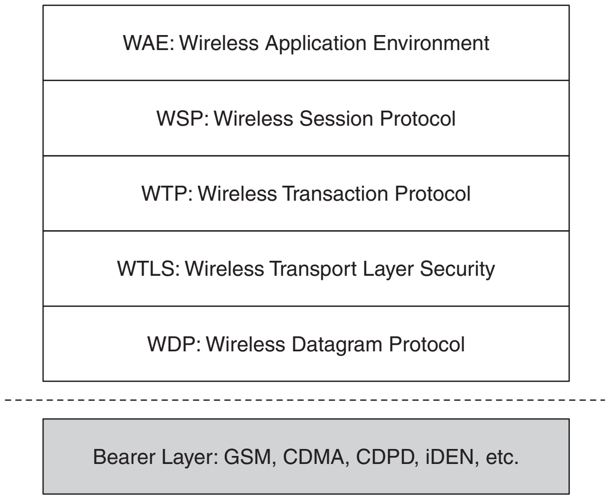

WAE: Wireless Application Environment
WSP: Wireless Session Protocol
WTP: Wireless Transaction Protocol
WTLS: Wireless Transport Layer Security
WDP: Wireless Datagram Protocol
Bearer Layer: GSM, CDMA, CDPD, iDEN, etc.

Figure 11.3 The WAP protocol stack

There are several advantages to this stack-based approach. The first is something we have just mentioned: each layer can maintain an illusion that it is communicating solely with the same layer on another computer.

Another advantage is modularity. Each layer in the communications model can be implemented by a module of some sort. That module can be built to serve only a specific function, implementing a specific layer in the stack. As with all modular software components, this type of design

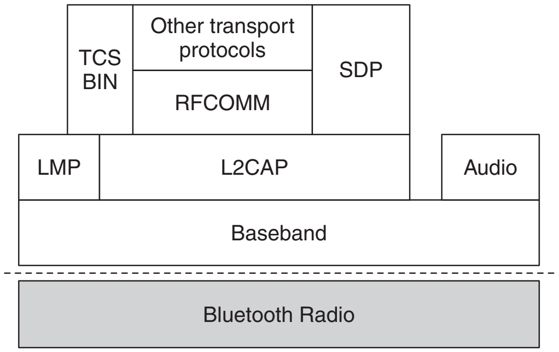


Figure 11.4 The Bluetooth protocol stack

enhances the ability to modify only certain parts of the communications stack without affecting other parts. If, for example, an operating system were to start using IPv6 instead of IPv4 in the implementation of Ethernet networking, the network layer could be removed and reworked, leaving the other layers alone.

Communications Abstractions

Another way to look at communications is to consider the abstractions that operating systems employ to address it. Abstractions are often used by operating systems to allow users to encapsulate many details into a concise model. This concise model, while hiding many details, helps to understand operating system functions better.

Sockets

We looked at sockets in Chapter 6 as a means of interprocess communications. Sockets are often used in general to depict the communications between computers.

By way of review, sockets were invented by the designers of Berkeley Unix and were first used as an 'endpoint for communications' to access network protocols. As an endpoint, a socket is not very useful. But when connected to a socket on another computer, the pair becomes a communications channel that uses a protocol to transfer data. You can think of sockets as two ends of a conversation and the protocol as the translator.

Sockets assume a client–server model. The client connects its end of the socket and makes a request to the server for connection. The server either connects up its end and replies positively or replies that no connection can be made. Then, if the socket has been successfully connected, data is exchanged across the socket.

The socket model is useful as a communications model because of its abstractness and its depiction of translation abilities. The abstractness of the model can be seen in how it is used: each side simply writes data to and reads data from a socket as if it were a local I/O device. Each side really does not know (or care) how the other side reads or processes the data. In fact, the socket may implement translation of data, again without each side knowing (or caring). The translation is implemented by the operating system and occurs as the data is transferred between the endpoints.

Event-driven Communications

Waiting is something built into the use of communications. Whether communications is with a device, another process, or another computer, the act of exchanging communications data means that some waiting must occur on the other side's response. As we discussed in Chapter 9, waiting for device I/O is dealt with by means of interrupts. Asynchronous events serve as software interrupts for communications to other computers.

The idea behind communications events is the same as with device I/O. We use the idea of an 'event' to represent the occurrence of something that an application might be waiting for in a communications exchange. There are many events that are specified to represent communications. A process registers an interest in certain communications events and the operating system takes care of receiving the communications and notifying the process when communications data is ready. This means that, like with device I/O, the process involved can do other tasks concurrently with the waiting for communications.

This asynchronous event model is useful in several ways. It keeps applications from freezing while they wait for an event. It allows the system to interleave instructions more efficiently from concurrent processes or threads. It also allows the system to put the entire device to sleep in certain cases (e.g., when all processes are waiting for events) to conserve battery power.

The Symbian OS concept of active objects is an excellent example of communications events. As we have discussed in previous chapters, an active object is a specialized object that allows requests to be made that would otherwise force waiting, going to sleep to avoid a polling loop and handling the event that is generated when the request gets a response by waking up and continuing execution. By building into Symbian OS the concept of an active object as a thread, the designers of Symbian OS built an object that can handle waiting for responses from communications and not prevent other code from running.

Consider an email application as an example of an active object. Email can be collected by sending a request to a server. A response is rarely immediate, especially if there are many messages to pick up. If an application were not to use active objects, the application would be likely to freeze up while waiting for all the messages to download. This situation would occur because the application would be concentrating on the data exchange with the email server and not on listening for stylus taps or updating the screen. However, if the application used an active object to communicate with the server, then it could respond to the user at the same time it was waiting for messages. Active objects enable more interaction with applications and a clean interface for handling situations that might arise during communications.

For many operating systems, it is possible not to use an event-driven method for communications. Waiting for communications data is also available for processes. If a process has no other tasks to attend to or has a need to react immediately when communications data arrives, then it can make the standard system calls that block until data arrives on a socket.

11.2 Communications on Symbian OS

Symbian OS provides a great example of our general communications model discussion. It was designed with specific criteria in mind and can be characterized by event-driven communications using client–server relationships and stack-based configurations.

Basic Infrastructure

A good way to start looking at the Symbian OS communications infrastructure is by examining its basic components. Let's start by considering the generic form of this infrastructure shown in Figure 11.5.

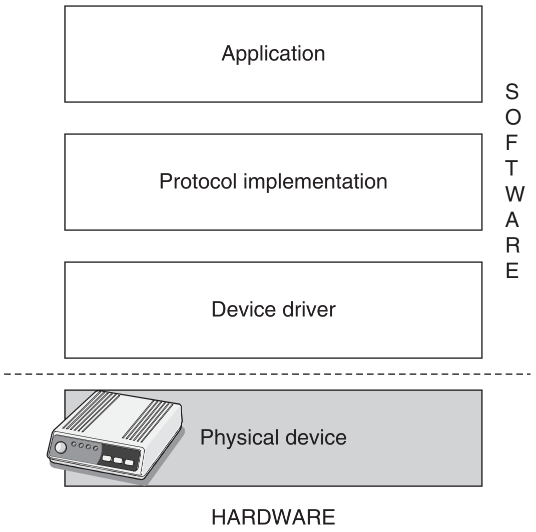


Figure 11.5 Basic depiction of Symbian OS communications infrastructure

Consider Figure 11.5 as a starting point for an organizational model. At the bottom of the stack is a physical device, connected in some way to the computer. This device could be a mobile phone modem or a Bluetooth radio-transmitter embedded in a communicator. Since we are not concerned with the details of hardware here, we treat this physical device as an abstract unit that responds to commands from software in the appropriate manner.

The next level up – the first level we are concerned with – is the device-driver level. The software at this level is concerned with working directly with the hardware via a fixed, standardized interface to the upper software layers. The software at this level is hardware-specific and every new piece of hardware requires a new software device driver to interface with it. Symbian OS comes with a set of device drivers for commonly used pieces of hardware (e.g., wireless Ethernet clients or Bluetooth transmitters). Different drivers are needed for different hardware units, but they must all implement the same interface to the upper layers. The protocol implementation layer expects the same interface no matter what hardware unit is used.

Standards play a major role with Symbian OS device drivers. Hardware is becoming increasingly standardized and sometimes one device driver can manage several pieces of hardware because they all abide by the same standard. For example, many serial devices – modems or IR ports – can be controlled by a single device driver. In addition, protocol implementations are increasingly assuming device-driver standards. Standards such as Bluetooth or wireless Ethernet, for instance, are becoming widely supported and therefore must be incorporated in the device-driver layer.

The next layer up is the protocol-implementation layer. This layer contains implementations of the protocols supported by Symbian OS. These implementations assume a device-driver interface with the layer beneath and supply a single, unified interface to the application layer above. This is the layer that implements the Bluetooth and TCP/IP protocol suites, for example, along with other protocols.

Finally, the application layer contains the application that must utilize the communications infrastructure. The application does not know much about how communications are implemented – however, it does do the work necessary to inform the operating system of which devices it use. Once the drivers are in place, the application does not access them directly, but depends on the protocol-implementation-layer APIs to drive the real devices.

A Closer Look at the Infrastructure

Now let's take a closer look at the layers in Symbian OS communications infrastructure. Figure 11.6 contains a new diagram based on the generic model in Figure 11.5. The blocks from Figure 11.5 have been subdivided into operational units that depict those used by Symbian OS.

The physical device

First, notice that the device has not been changed. As we stated before, Symbian OS has no control over hardware. Therefore, it accommodates hardware through this layered API design, but does not specify how the hardware itself is designed and constructed. This is an advantage to Symbian OS and its developers. By viewing hardware as an abstract unit and designing communications around this abstraction, the designers

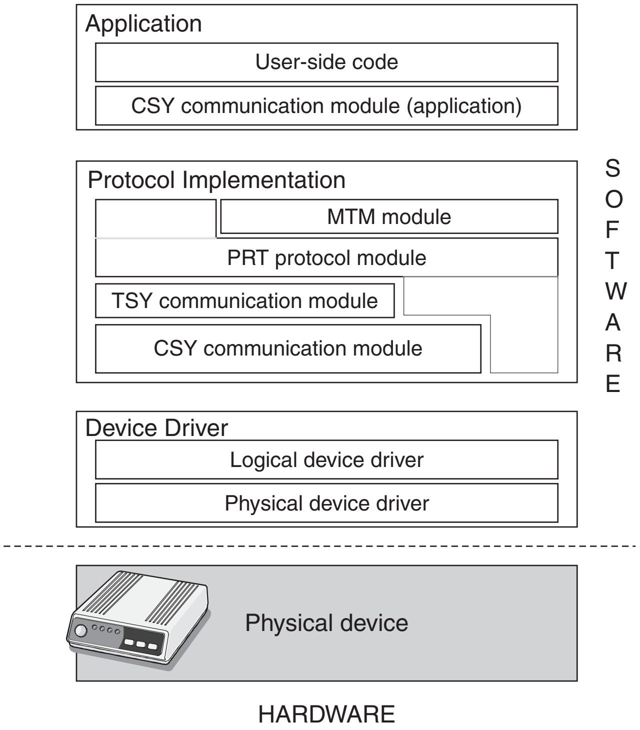


Figure 11.6 Detailed look at the Symbian OS communications infrastructure

have ensured that Symbian OS handles the wide variety of devices that are available now and that it can accommodate the hardware of the future.

The device-driver layer

The device-driver layer of Figure 11.5 has been divided into two layers in Figure 11.6. The *physical device-driver* (PDD) layer interfaces directly with the physical device, through a specific hardware port. The *logical device-driver* (LDD) layer interfaces with the protocol-implementation layer and implements Symbian OS policies as they relate to the device. These policies include input and output buffering, interrupt mechanisms and flow control. The division of these layers represents a division in implementation, where the PDD implementers can focus on an efficient and correct hardware interface and the LDD implementers can work to perfect the interface with the upper layers to maximize performance.

User code interfaces with the LDD through the `RBusLogicalChannel` class. This is a simple interface that all user code interactions go through. Note that this class is used no matter if the device is the display, a Bluetooth transmitter, or an infrared port. This provides a layer of abstraction that results in a consistent interface to the physical device. The PDD provides the connection to the physical device. It interfaces with the LDD using an interface designed by the LDD.

As an example of this division of responsibilities, consider the serial interface. There are several serial-device types that can be connected to a Symbian OS device. The IR port and the RS232 port are both serial devices and can use the same generic serial LDD, called `ECOMM.LDD`. These ports are serial ports and use the same policies with respect to issues such as flow control. However, their PDD modules are different: one services the RS232 port and one services the IR port. Other examples include the Ethernet driver (`ENET.LDD` and `ETHERNET.PDD`) and the sound driver (`ESOUND.LDD` and `ESDRV.PDD`).

Chapter 9 contains more information about device drivers and kernel extensions. There are many more details about these items that are not relevant here.

The protocol-implementation layer

Several sublayers have been added to the protocol-implementation layer. Four types of module are used for protocol implementation.

- *CSY modules*: the lowest level in the protocol implementation layer is the communications server. A CSY module communicates directly with the hardware through the PDD portion of the device driver, implementing the various low-level aspects of protocols. For instance, a protocol may require raw data transfer to the hardware device or it may specify 7-bit or 8-bit buffer transfer. These 'modes' would be handled by the CSY module. Note that CSY modules may use other CSY modules. For example, the IrDA CSY module that implements the IrCOMM interface to the IR PDD also uses the serial device driver, ECUART CSY module.

- *TSY modules*: telephony comprises a large part of the communications infrastructure and special modules are used to implement it. The telephony server (TSY) modules implement the telephony functionality. Basic TSYs may support standard telephony functions, e.g., making and terminating calls, on a wide range of hardware. More advanced TSYs may support advanced phone hardware, e.g., those supporting GSM functionality.

- *PRT modules*: the central modules used for protocol implementation, protocol (PRT) modules, are used by servers to implement protocols. A server creates an instance of a PRT module when it attempts to use the protocol. The TCP/IP suite of protocols, for instance, is implemented by the `TCPIP.PRT` module. Bluetooth protocols are implemented by the `BT.PRT` module.

- *MTMs*: as Symbian OS has been designed specifically for messaging, its architects built a mechanism to handle messages of all types. These message handlers are called *message type modules* (MTMs). Message handling has many different aspects and MTMs must implement each of these aspects. User-interface MTMs must implement the various ways users view and manipulate messages, from how a user reads a message to how a user is notified of the progress of sending a message. Client-side MTMs handle addressing, creating and responding to messages. Server-side MTMs must implement server-oriented manipulation of messages, including folder manipulation and message-specific manipulation.

These modules build on each other in various ways, depending on the type of communications that is being used. Implementations of protocols using Bluetooth, for example, use only PRT modules on top of device drivers. Certain IrDA protocols do this as well. TCP/IP implementations that use PPP use PRT modules, a TSY module and a CSY module. TCP/IP implementations without PPP typically do not use either a TSY module or a CSY module but link a PRT module directly to a network device driver. The WAP protocol stack uses a WAP PRT on top of an SMS PRT, which in turn is built on a GSM TSY and some kind of CSY (ECUART, IrCOMM, or RFCOMM).

Infrastructure modularity

We should take a moment to appreciate the modularity of the stack-based model used by the communications infrastructure design. The 'mix and match' quality of the layered design should be evident from the examples just given. Consider the TCP/IP stack implementation. A PPP connection can go directly to a CSY module or choose a GSM or regular modem TSY implementation, which in turn goes through a CSY module. When the future brings a new telephony technology, the existing structure works and we only need to add a TSY module for the new telephony implementation. In addition, fine-tuning the TCP/IP protocol stack does not require altering any of the modules it depends on; we simply tune up the TCP/IP PRT module and leave the rest alone. This extensive modularity means new code plugs into the infrastructure easily, old code is easily discarded and existing code can be modified without shaking the whole system or requiring any extensive reinstallations.

Finally, Figure 11.6 has added sublayers to the application layer. There are CSY modules that applications use to interface with protocol modules in the protocol implementations. While we can consider these as parts of protocol implementations, it is a bit cleaner to think of them as assisting applications. An example here might be an application that uses IR to send SMS messages through a mobile phone. This application would use an IRCOMM CSY module on the application side that uses an SMS implementation wrapped in a protocol-implementation layer. Again, the modularity of this entire process is a big advantage for applications that need to focus on what they do best and not on the communications process.

11.3 Communications on Other Operating Systems

Symbian OS is a great example of the way that most other operating systems model communications. Figure 11.6 is a good way to depict the application of I/O concepts to communications in other operating systems.

Device drivers are the way operating systems tie physical communications-hardware devices into the system. Every device is different, but every device must eventually be presented in a standard way. This means that devices must speak to the operating system via some kind of 'translator', and that translator is the device driver. Device drivers are typically loaded dynamically when their intermediary services are needed, much like the drivers for other types of device I/O (discussed in Chapter 9).

The protocol implementation for an operating system is a good example of the stack model of communications. Most implementations of communications form a stack, as we demonstrated with Symbian OS. As discussed in Chapter 9, abstraction plays a big role here. While the same API is in often place, the implementation functions are dynamically loaded when they are needed.

Consider, for example, the code below, which is for a generic Linux server that accepts a socket connection from a client and reads and processes the client's request across the socket.

```
void main(int argc, char **argv)
  {
  /* Declarations for sockets */
  int sockfd;
  int result, select;
  int readfds[32];
  struct sockaddr_in sin;
  int msgsock;
  char recv_buffer[256];

  /* Step 1: Create the socket. */
  sockfd = socket (AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    {
    (void) perror ("socket creation");
    exit(-1);
    }

  /* Step 2: Data setup */
  sin = (struct sockaddr_in *) & salocal;
  memset ((char *) sin, "\0", sizeof (salocal));
  sin->sin_family = AF_INET;
  sin->sin_addr.s_addr = INADDR_ANY;
  sin->sin_port = 1100;

  /* Step 3: Bind the socket to the network */
  result = bind (sockfd, & salocal, sizeof (*sin));
if (result < 0)
  {
  (void) perror ("socket binding");
  exit(-1);
  }

/* Step 4: Set up listening on the socket */
listen(sockfd,5);

for(;;) {
  FD_ZERO(&readfds);
  if(sockfd >= 0) FD_SET(sockfd, &readfds);

  /* Step 5: Wait for input on the socket */
  status = select(32, &readfds, NULL, NULL, NULL);
  if(status == -1)
    {
    if (errno == EINTR) continue;
    exit(101);
    }
  if((sockfd >= 0) && FD_ISSET(sockfd, &readfds))
    {
    DEBUG("Setting up the MSG SOCKET\n");

    /* Step 6: GOT input, accept it, creating a file descriptor. */
    msgsock = accept(sockfd, 0, 0);

    /* At this point, we have a file descriptor (msgsock) that we
       can read and write to. NOW we process the data that comes
       over the socket.
       Step 7: Process input. */
    result = read(msgsock, recv_buffer, (int) sizeof(recv_buffer));
    if(result > 0)
      {
      result = process_cmd(recv_buffer,msgsock);
      close(msgsock);
      }
    }
  }
}
```

Each system call addresses some part in the stack. Step 1 creates a socket, unbound but with a definition (it is a TCP/IP network socket, `AF_INET`, to be opened over TCP, `SOCK_STREAM`). Step 3 pushes a bit deeper into the stack, using the data set up in Step 2 to bind the file descriptor to an entry in a system table connected with networks. This `bind()` call implies that a read or write takes place and loads the code necessary to connect the read or write with the data on a network. Before `bind()`, a socket is just a file descriptor; after `bind()`, it is a descriptor tied to a specific implementation of read and write system calls. Step 4 activates network listening for requests, using `read()` and `write()` calls to engage in socket protocols with the other side of the network connection. At the end of this call, the device driver has been loaded and is actively working with the network port to process network data. Step 5 is a blocking call that waits until some file descriptor has data on it. Again, notice the abstraction in this call. The `select()` call takes a collection of file descriptors of all kinds and returns when at least one has data. That implies that all the descriptors must have the same interface (read/write). When the call returns, we assume that the socket we have set up as a file descriptor has data waiting. Step 6 accepts that data, deriving a regular two-way socket from the server socket. Step 7 begins receiving data.

This example illustrates the abstraction and modularity built into the stack model of communications. It combines a client–server approach with stack-oriented implementations and sockets to give us an effective way of describing network service. These models have proved so useful that they are built into most modern operating systems in use today.

11.4 Summary

This chapter has discussed the types of models that operating systems use to characterize communications. Abstraction is the key to using these models. We discussed how client–server relationships are built and how implementation stacks use abstraction to connect system calls together. We described how the ideas of sockets and event-driven communications provided mechanisms to understand implementation details. We then gave examples of these models from both Symbian OS and Linux.

Exercises

1. We stressed many different uses for abstraction in this chapter. Find them and describe why each one is useful.
2. Client–server relationships can be found in many places in an operating system. Find at least three such relationships in the operating system of your choice and report them.
3. In a mobile phone call, there are a series of client–server relationships that are established. Identify these and report them.

4. Describe how a web page is requested and delivered between a web browser and a website. Identify the client–server relationships in the sequence of events.

5. Invent a communications stack for a person who speaks French to get a message (spoken by him in French) to a colleague in Germany. The message must be received the same day and must be in German when it reaches the colleague.

6. In Symbian OS, Bluetooth communications is done primarily through sockets. Explain why this is preferred to just opening the Bluetooth device and reading from and writing to it.

7. An application needs to open a socket to a web server, send a request and read a web page. Describe the events that would be generated by that sequence of operations.

12 Telephony

The convergence of telephony and handheld computing devices has had a long history. Modems have been available commercially since the 1960s and, from that time, telephony services and computers have been linked. As integrated devices were developed, control over telephony functions became a more crucial issue for operating system designers. The support of telephony had to adhere to the communications models that have governed all computer communications. Telephony functionality became an important component of Symbian OS even before the first true Symbian OS phones (the Ericsson R380 and Nokia 9210 Communicator) shipped, with core functionality to support a 'two-box' connection (PDA to phone) appearing for the original Psion Series 5 range. These devices demonstrated how telephony communications and computing capability – integrating data and voice capability – can be mutually beneficial.

This chapter examines how operating systems can support telephony. Symbian OS obviously stands out as a great example of telephony support. A wide variety of telephony functions and uses are provided by the Symbian OS ETel telephony subsystem. This chapter gives an overview of telephony services and the operating systems components that implement them. We then take a look at the ETel model and its interfaces by examining the different situations it was designed to address and looking at the structure of the APIs. We dig into the details of ETel by looking at the main aspects in its design.

12.1 Modeling Telephony Services

Communication using a conventional analog telephone is usually split into two forms: data and voice communication. Data communication over a telephone is handled by a modem, a digital-to-analog converter. Digital communication in the form of binary data is converted to analog audio representation that can be sent between telephones over conventional phone systems. At the other end, the reverse process takes place.

The computer controls the modem by using a set of commands. The Hayes command set, for example, is by far the most widely used set of commands for modems. Hayes-format commands take the form of 'AT' commands: the character sequence 'AT' followed by characters to command the modem to do certain tasks. To initialize a modem and cause it to dial a number, for example, we might give it the following command string:

```
ATQ0E1DT555-3454
```

This string tells the modem to send result codes back to the computer in response to commands, echo back the commands sent to the modem and tone-dial the number `555-3454`.

Voice communication via both the public-switched-telephone network (PSTN) and the mobile phone network (for example, GSM) is effectively a peer-to-peer connection, where the 'address' of the remote device is the telephone number being dialed. The telephone number is accepted by the telephone-network-switching infrastructure and a connection is made between the local and remote devices, until one or both terminate the connection by 'hanging up'. A telephone may be able to handle several telephone lines; each line can typically multiplex several phone calls at once, by working appropriately with the telephone system.

Telephone networks can be wired ('landline' PSTN networks) or wireless (mobile phone networks). Of the possible equipment and switching network combinations, digital choices have emerged as the most promising for the future. Digital switching has replaced more antiquated step and cross-bar exchanges within the PSTN. A GSM phone is a good example of digital phone equipment using a digital connection network; GSM is a frame-based protocol that sends data frames of a fixed size for a fixed time interval, mixing these frames with others. The phone must also be a digital device to use the digital network, and may offer the capability of being controlled by an external computer to turn it into an advanced modem capable of using the digital network on behalf of another device. A Nokia E61, for example, has an infrared port that allows it to accept commands from an external device.

The communications model of an operating system can be extended to include telephony as a core feature. How well telephony works with an operating system depends on the operating system architecture and its history.

In a conventional operating system, working with telephony services means using system calls to interact with a telephony device. The operating system usually works with a telephony device in either of two extremes: a very abstract way or methods that are low-level and concrete. For example, Linux treats a telephony device as any other device and allows `ioctl()` calls – low-level system calls that can manipulate any device – to interact with device drivers and control the telephony device. On the other hand, Microsoft Windows Mobile treats the phone portion of handheld computer as an object and provides a way to work with it through the object interface. For example, in a Windows Mobile application that implements dialing a phone number, you might find code like this (written in C#):

```
Phone myPhone = new Microsoft.WindowsMobile.Telephony.Phone();
myPhone.Talk("555-7341\0");
```

Whether an interface uses high- or low-level manipulation, the control of the phone device is direct via a kernel-mode driver.

In microkernel architectures, devices are controlled by servers that provide a high level of abstraction to multiple clients, forcing manipulation of devices to take place through this abstraction. Thus, microkernels use telephony servers to provide telephony services through standard client–server relationships with other processes. Symbian OS, for example, implements an ETel server that provides access to telephony devices. Applications interested in making a phone call must connect with this server and send it requests.

By way of understanding telephony communication, we examine the Symbian OS telephony subsystem in more detail. This study serves as an example of a microkernel implementation of telephony as well as of Symbian OS design.

Telephony implementations demonstrate what we have been illustrating all through this book: various operating systems do things at various levels of abstraction. Linux is perhaps the least abstract while Symbian

OS is perhaps the most abstract. Because these abstraction levels are by design, we cannot simply ask which one is best. We have to ask which one is the best in its area of application and its audience of developers.

12.2 A Structural Overview

The communications model Symbian OS uses to implement telephony is abstract enough to provide the application programmer with a consistent, standard interface, no matter what kind of telephony device is being used.

The Symbian OS telephony subsystem closely models the real-world user experience of using telephones (Figure 12.1).

The model characterizes telephony as a collection of phones. Each phone is an abstraction of a telephony device (e.g., a modem or a GSM phone; usually only one exists per phone, which corresponds to that particular device's radio hardware setup such as GSM, UMTS or

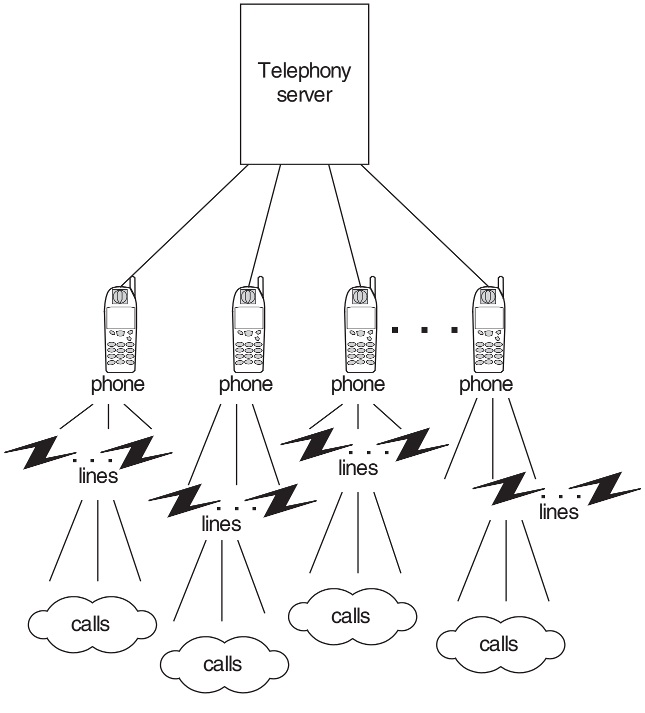


Figure 12.1 The ETel phone model

CDMA). Through this abstraction, we can access a device's status and capabilities and be notified if changes occur to a device's properties. A phone can have one or more lines. An application can access the status and capabilities of a line, as it can for a phone, and can be notified of any changes in these features. Lines usually correspond to specific telephony services (e.g. voice, fax, circuit-switched data, etc.).

The actual connection of a local endpoint through a circuit-switched network to the phone is designated as a call. A line can have zero or more active calls. A call can dial a number, wait on a line for an incoming call and be terminated. As with lines, an application can get status and capabilities information for a call and be notified of changes to a call's state.

These abstractions are central to the use of the Symbian OS telephony subsystem through the ETel APIs.

TSY Modules in Symbian OS

The heart of this model's implementation is in the TSY module. By integrating the specific implementation of this model for a particular phone type into a module, Symbian OS designers ensured that the API for this functionality would remain the same across different phones, and the application programmer is free from worrying about the implementation specifics for a particular phone. When Symbian OS is used with new phone hardware for the first time, a new TSY must be developed.

TSY modules are designed to plug into the telephony server and provide access to telephony functionality. Search your Symbian phone or emulator and look for TSY modules provided with the device. In Symbian OS v9, you should find several TSYs that are provided with the distribution, including generic phone TSYs and ones that implement CDMA.

The ETel Subsystem

The ETel subsystem has four key constituents (see Figure 12.1): the ETel server, the Phone abstraction, the Line abstraction, and the Call object.

ETel server

The ETel server manages access to the telephony system. It is accessed using functions from the `RTelServer` class. Before telephony can be used by an application, it must connect to the telephony server. This is done with the `Connect()` function. Using this function, applications

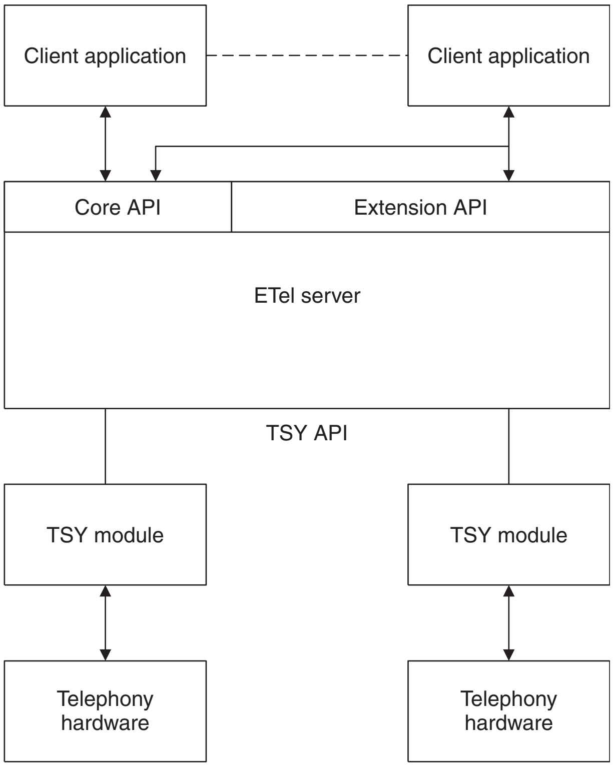


Figure 12.2 ETel structural diagram

connect to the telephony server, specifying how many *message slots* are needed. A single message slot is a communication channel in one direction. The default number of slots assigned is 32. The `RTelServer` class is a subclass of the `RSessionBase` class, and therefore inherits the `Close()` function, which is used to shut down an active telephony server session.

Once a connection is established, the TSY module that is needed should be loaded. TSY modules can be manipulated through the `Load-PhoneModule()` and `UnloadPhoneModule()` functions. The first function loads a TSY module, and the second function removes a TSY module. These modules are analogous to device drivers (especially in that they relate to a specific device – in this case, the particular baseband hardware in the phone) and are implemented with logical and physical components particular to the specific baseband. Recall from Chapter 11 that device drivers need to be loaded and unloaded; TSYs require the same handling.

Once the appropriate TSY module has been loaded, applications can make queries about its properties. These queries take the form of telephony server functions, such as the `Version()` or `GetPhoneInfo()` functions, and result in requests being sent to, and responses returning from, the ETel server. It is possible to obtain version information, the phone's name, the type of telephone network it uses, and other information. Information about the TSY itself is also available from the telephony server. All TSY modules are assumed to support a minimal set of telephony functionality, but by calling `IsSupportedByModule()` it is possible to determine exactly what ETel functions are supported.

Let's take an example: we have a `CPhoneCall` class that initializes a phone and makes a voice phone call. The definition for this class might look like this:

```
class CPhoneCall : public CActive
  {
  enum TCallState {EDialing, EDone, EError};
public:
  ~CPhoneCall();
public:
  // Static construction
  static CPhoneCall* NewLC();
  static CPhoneCall* NewL();
public:
  void MakeCall(TDesC& aTelephoneNumber);
private:
  CPhoneCall();
  void ConstructL();
  void InitL();
  void DoCancel();
  void RunL();

  RTelServer iTelServer;
  RPhone iGsmPhone;
  RLine iPhoneLine;
  RCall iPhoneCall;

  TRequestStatus iCallStatus;
  TCallState iCallState;
  };
```

Notice that this class is an active object that uses the `iCallState` variable to track its communication state and the `iCallStatus` variable to monitor its I/O progress.

Now, consider the definition of the `InitL()` function as it applies to the telephony server. Here, we want to deal with a voice call over a GSM phone:

```
void CPhoneCall::InitL()
  {
  RTelServer::TPhoneInfo phoneInfo;
  RPhone::TLineInfo lineInfo;
  RPhone::TCaps capabilities;
  RLine::TCaps lCapabilities;

  TInt result;
  TInt phones, lines, calls;
  TFullName name;

  // Connect to the telephony server
  result = iTelServer.Connect();
  User::LeaveIfError(result);

  // Load the right TSY
  _LIT(KTsyToLoad,"gsmbsc.tsy")
  result = iTelServer.LoadPhoneModule(KTsyToLoad);
  User::LeaveIfError(result);

  // Get information about phones from the server
  result = iTelServer.EnumeratePhones(phones);
  User::LeaveIfError(result);
  if (phones == 0) User::LeaveIfError(KErrNotSupported);

  // other code to init phones, lines and calls
  }
```

We connect to the telephony server and load the GSM TSY module. If no phones are supported (for example, if no GSM TSYs could be found), this code leaves with an error code.

The phone abstraction

After connecting to the ETel server, a phone supported by the telephony server should be selected. Phones are characterized by the `RPhone` class and are accessed through a subsession established by an `RPhone` object. As when establishing connections and sessions, we use `Open()` and `Close()` functions for this. After opening a phone subsession, notifications can be set up and the phone must be initialized. Changes to the phone's state and capabilities are reported to the client application using functions known as *notifications*. Generally, the client makes all the notification requests prior to calling any functions which may change the state of the telephony device. Initializing is allowed to be asynchronous, because it may take some time to set up the telephony device.

When a phone subsession is open and the device has been initialized, applications can use the other functions of the phone device or make queries of it thorough the `RPhone` class interface.

Let's continue the `CPhoneCall` class example. We need to expand the implementation of the `InitL()` function to encompass initializing phones. The result is below:

```
void CPhoneCall::InitL()
  {
  RTelServer::TPhoneInfo phoneInfo;
  RPhone::TLineInfo lineInfo;
  RPhone::TCaps capabilities;
  RLine::TCaps lCapabilities;

  TInt result;
  TInt phones,lines,calls;
  TFullName name;

  // code to initialize the telephony server connection

  // Get the information on the phone we need
  result = iTelServer.GetPhoneInfo(0, phoneInfo);
  User::LeaveIfError(result);
  name.Copy(phoneInfo.iName);

  // Open the phone and get its capabilities
  result = iGsmPhone.Open(iTelServer, name);
  User::LeaveIfError(result);
  result = iGsmPhone.GetCaps(capabilities);
  User::LeaveIfError(result);
  if ((capabilities.iFlags & RPhone::KCapsVoice) == 0)
    User::LeaveIfError(KErrNotSupported);

  // other code to init lines and calls
  }
```

Note that the phone is initialized when the first asynchronous request is sent. So we do not need to call `Initialize()` from this initialization code. In the code above, we retrieve the name of the first phone from the telephony server and open it up. On a Nokia 8290 phone (a GSM phone used in the United States), the name of this first phone is '`GsmPhone1`'. We conclude this code by making sure that the phone we obtained can indeed support voice capability.

The line abstraction

Once a subsession with a phone has been established, we may establish a subsession for a particular line. The line implementation is implemented by the `RLine` class. As with `RPhone` objects, `RLine` object subsessions are opened and closed with `Open()` and `Close()` functions.

As with `RPhone` objects, `RLine` objects can be notified when properties of a line change. There are many different properties that can change and this is reflected in the number of notification functions that are defined for the `RLine` class. There are four notification functions, each with its own cancellation function. Each is also asynchronous and requires a status variable for monitoring; notification functions are useful here.

Let's continue to flesh out the `CPhoneCall` class example. Initializing a line for a phone means getting its name and opening a subsession, as below:

```
void CPhoneCall::InitL()
  {
  RTelServer::TPhoneInfo phoneInfo;
  RPhone::TLineInfo lineInfo;
  RPhone::TCaps capabilities;
  RLine::TCaps lCapabilities;

  TInt result;
  TInt phones,lines,calls;
  TFullName name;

  // code to initialize the telephony server and phone

  // Get the info on the line we need -  we have hard-coded 2 to open
  // the 3rd line. In reality, one should use EnumerateLines() to
  // determine the required line on any particular phone
  result = iGsmPhone.GetLineInfo(2, lineInfo);
  User::LeaveIfError(result);
  name.Copy(lineInfo.iName);

  // Open the line and get its capabilities
  result = iPhoneLine.Open(iGsmPhone, name);
  User::LeaveIfError(result);
  result = iPhoneLine.GetCaps(lCapabilities);
  User::LeaveIfError(result);
  if ((lCapabilities.iFlags & RLine::KCapsVoice) == 0)
    User::LeaveIfError(KErrNotSupported);

  // code to initialize call
  }
```

This example chooses the third line available on the phone and checks its capabilities. On a Nokia 8290 phone, the third line is the voice line (the first two are fax and data lines) and the name of this line is `Voice`.

The call object

With a session established to the telephony server and phone and line subsessions now open, we can finally open and manage a call. Calls are implemented by the `RCall` class. Before we discuss how to use this class, we should point out a few things about calls.

- Calls have names, as with other telephony-server objects. The name of a call is generated by the operating system through the TSY and returned when a call subsession is opened. A 'fully qualified name' is one that includes call, line and phone information in the format:

  ```
  PhoneName::LineName::CallName
  ```

- Opening a call subsession does not connect a call. As with phones and lines, a call subsession must be opened before we can use a call. Opening a subsession allows the telephony server to allocate memory and resources for a call but does not manipulate the call in any way.
- Calls can be incoming as well as outgoing. In addition to instructing the telephony server to make a call, we can instruct the server to answer a call. Unlike the other layers in the model, a new subsession is opened with calls other than `Open()` and `Close()`. The `OpenNewCall()` function in its several forms creates a new call in an idle state. `OpenExistingCall()` is more usually used to open a call in an 'alerting' state. A new call can be opened by referencing an open telephony server session, a phone subsession or a line subsession. Subsessions can be opened with existing calls, i.e., calls in progress. This is done by applications that want to work with calls already received or started by other applications. For example, if one application placed a voice call, a second application could implement a call timer. To hang up after a certain time period, the timer application would have to open the existing call with the `OpenExistingCall()` function.

As an example, we can complete the `CPhoneCall::InitL()` function. Here, we simply open a new call subsession by referencing the line subsession:

```
void CPhoneCall::InitL()
  {
  RTelServer::TPhoneInfo phoneInfo;
  RPhone::TLineInfo lineInfo;
  RPhone::TCaps capabilities;
  RLine::TCaps lCapabilities;

  TInt result;
  TInt phones,lines,calls;
  TFullName name;

  // code to initialize server, phone and line

  // Open a new call
  result = iPhoneCall.OpenNewCall(iPhoneLine, name);
  User::LeaveIfError(result);
  }
```

On our Nokia phone, this call is assigned the name `VoiceCall1`.

Although it may seem like a long journey, eventually all sessions and subsessions are opened and initialized. At this point, we still have not made a call, but the system is ready for this next step.

Calls are made by instructing the Symbian phone to dial with a directory number or by connecting to an already dialed call. To dial a call, we use the `Dial()` function from the `RCall` class. Dialing functions come in synchronous or asynchronous varieties and can include call parameters.

To illustrate this, let's define the `MakeCall()` function from our `CPhoneCall` example. We have decided to make the `CPhoneCall` class an active object and we can use an asynchronous version of the `Dial()` function:

```
void CPhoneCall::MakeCall(TDesC& aNumber)
  {
  iPhoneCall.Dial(iCallStatus, aNumber);
  iCallState = EDialing;
  SetActive();
  }
```

Since we have already set up the telephony system, this is a simple implementation. The number is a string, and we return from this function right away while the system dials the call. When the call is dialed, the status variable changes state and the active object's `RunL()` function is called. We can implement this change by including the following code in the `RunL()` function:

```
switch (iCallState)
  {
  case EDialing:
    if (iCallStatus == KErrNone)
      {
      // handle the successful call
      }
    else
      {
      // handle the call error
      }
    break;
  ...
```

This is just like the active object code we have seen before.

If we have successfully created all the sessions and subsessions we need, answering an incoming call is straightforward. The `Answer-IncomingCall()` function – in synchronous and asynchronous versions – allows the phone to answer the call. Once a call has been answered, the application can monitor the call's state and perform various operations on it.

If it is a data call, the application might want to access the data port directly for a time. For example, if it is making a call with a modem to transfer some data, the application might want the telephony server to take care of dialing the phone number and connecting to the opposite side, but it will then want to take control to pass the data. This is done by 'loaning' the data port to the application using the `RCall::LoanDataPort()` function. Once the data port has been used for the transmission of data, it can be returned to the telephony server using the `RecoverDataPort()` function. While the port is loaned to the client it is possible that some ETel operations are not available.

For a call, there are a few notifications that the system can give an application. The hook status (a phone is 'on hook' when it is idle and 'off hook' when it is being used), the call phase state, and the call duration can be registered for notification. Remember that remote devices or the switching network can terminate a call at any time without asking for permission or giving prior warning. The call state status is a representation of the state of a call as it passes through its lifecycle: idle, dialing, ringing, answering, connecting, connected, or hanging up. The call duration is the time, in seconds, that the call has been active. Notification is sent every second (useful for making an indication to the user based on the duration of the call).

To avoid problems with multiple clients accessing the same call, Symbian OS designates a specific client as the *owner* of a call. This ownership is initially passed to the client that connected to a phone call first, but it can be transferred to another client. For example, one contact manager client may be responsible for setting up a data call while another client is responsible for data transfer.

Call termination, that is, 'hanging up' the phone, is accomplished through the `HangUp()` function. This function initiates termination of the call. Depending on the network, hang-up functions may behave differently. For example, for GSM networks, it takes time for the call to become idle (hence the hanging-up state) but it is not possible to re-connect a call once this request has been made. On some wired networks it is possible to hang up and then, if the remote party has not terminated the call, retrieve the call again.

12.3 Voice over IP Telephony

To fully appreciate the telephony model that Symbian OS implements, it is useful to consider a new form of telephony: voice over IP (VoIP). Unlike GSM (which is circuit-switched), VoIP calls are packet-based transfers of digital data, but the medium of transfer is a computer network. First, let's take a brief look at how VoIP works, as shown in Figure 12.3.

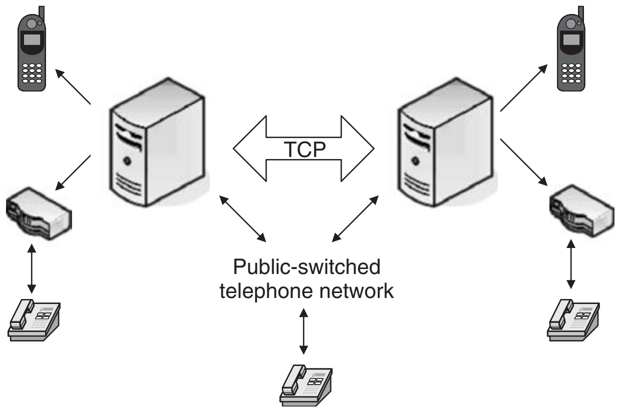


Figure 12.3 Simplified diagram of VoIP connections

Several different phone types can be used with VoIP. Conventional, analog phones need to have an analog telephone adapter (ATA) that connects to a network; mobile phones with network capability can connect through wireless networks; digital phones connect directly to a network.

In all cases, the phone unit connected to a network may begin the call by using Session Initiation Protocol (SIP). SIP is a protocol that creates sessions and handles many of the logistics of finding the endpoint (mobile phone or digital phone) to which the phone call will eventually connect. Usually, calls are initiated by engaging in SIP with a VoIP provider. This provider provides mapping services to map the locator information (e.g., a telephone number) to an IP address on the network or a destination on a public-switched telephone network (PSTN). If the service finds the destination on the network, it informs the caller where the destination is and the caller forms a peer-to-peer relationship with the destination, exchanging data packets that represent the voice conversation. If the destination is on a PSTN, the server becomes the destination and a proxy, relaying voice traffic from the network over the PSTN.

For our purposes, VoIP provides an illustration of how telephony models work. In a conventional operating system such as Linux, a completely different model must be used to implement VoIP. The old model was a low-level one: phone equipment was controlled by interacting with it over a wired connection with control functions. VoIP would be implemented as part of the networking communications stack, not with control functions. SIP is an application protocol that uses TCP to transport its packets. So a VoIP application interacts with sockets and TCP connections, sending SIP packets and receiving responses. While the concept of send-and-receive is the same, different media means different implementations.

Modular models, such as the one in Symbian OS, also require new coding to implement new technologies such as VoIP, but the changes happen in a modular fashion supplied by the system. In Symbian OS, VoIP can be implemented by a new TSY module. In fact, in our `CPhoneCall` example, the only change to our code would be to change which TSY module was loaded in `CPhoneCall::InitL()`. The remainder of the code can remain intact because of the structure of the Symbian OS telephony model.

Smartphones that have access to any packet-based network can provide VoIP implementations. Devices that support Symbian OS v9 and above have the capability of supporting wireless network access as well, which can be used to implement VoIP in addition to (or even instead of) access over a more traditional cellular network.

As an example, consider Nokia S60 3rd Edition phones, which run Symbian OS v9. The Nokia E61 implements VoIP in two parts: the SIP component that engages a VoIP provider's server and the network calling settings that determine which SIP settings to use. After these are configured, the phone can make calls over GSM or over a network medium.

Look for these types of implementation in Symbian OS phones with TCP/IP network access built-in.

12.4 Summary

In this chapter, we have taken a look at how operating systems support telephony. We summarized how conventional systems allow telephony to be manipulated as low-level devices and protocols. We then spent considerable time looking at how Symbian OS views telephony in terms of four key constituents: an ETel telephony server, a phone abstraction, phone line abstractions defined for a specific phone, and finally a call made over one such phone line. We reviewed how to set up the structure so that we can use phone calls and we looked at making and answering calls. We defined how applications can be notified when changes in the phone system and settings are made. We concluded the chapter by looking at voice over IP and how it fits into telephony models.

Exercises

1. If a phone is used as a modem, is that considered telephony? Does it fit into the telephony model?
2. Why does Linux force programmers to use such primitive methods to access system services?
3. Why does Symbian OS force programmers to use such high-level methods to access telephony services?
4. How might a Linux application address VoIP?
5. In Symbian OS, does a phone call take up a lot of memory? Does it take up a lot of CPU time?
6. Consider making a phone call in Symbian OS while using other parts of the operating system (such as making a calendar entry). Make a guess as to how much of the system resources the phone call would take up in comparison to other tasks.

13
Messaging

In addition to telephony, which we covered in Chapter 12, messaging is an area that smartphones do well. It is their basic functionality to allow communication: both voice and data. As we did with telephony, discussing messaging provides us with a way to compare and contrast operating system approaches while showing off areas in which certain operating systems shine.

Through experience, most people understand that messages can take many forms. There are verbal and non-verbal messages; messages are written on paper and heard via audio devices; messages can be notes passed in secret or signs on a billboard. Delivery of messages is an important but sometimes chancy thing (if you have ever had to rely on another person to deliver a message, you know what I mean). Even with this wide assortment of message types and delivery functions, humans are able to send and receive messages fairly easily. We have, in fact, developed a system that processes different message types using the same methods implemented by tools specialized to each message.

Electronic messaging is a very diverse area. There are many electronic message types and delivery takes many different forms. For example, email messages can be delivered over a wireless network connection and SMS messages can come through a GSM connection to a mobile phone. So it should come as no surprise that designing a single model that characterizes and works with all message forms is quite a challenge. Operating systems approach this area in different – yet predictable – ways.

This chapter gives an overview of the message framework implemented by operating systems, with a special emphasis on Symbian OS. We start with a survey of messaging components and requirements. We then look at Symbian OS and its messaging framework. We wrap up the chapter by comparing the approach of Symbian OS to that of Linux.

13.1 The Character of Messaging

Let's start by stepping back for a moment and taking an overview of messaging. Messaging systems need to put together a generic framework that can handle the components of many different message types. This framework is likely to view all messages as composed of generic components. Through the use of abstraction, each different message type can be handled by a separate implementation of those generic components. Then this abstract message-handling system needs to be built on top of existing models and systems.

As we look at messages and their components, it is important to note where these messages originate on the computer system. All devices have a central message store. This store contains the messages received by and created on the device on which it resides. In some operating systems (e.g., Symbian OS), this message file has a specialized, hierarchical format. In other systems (e.g., Linux), it is simply a text file that can be parsed in several different ways. This store can usually reside on any storage accessible to a device; message applications can usually change where this store resides.

Let's look at the messaging framework by pulling apart a message, so we can build a framework from its component parts.

Dissecting a Message

Messages are self-contained data objects sent between two devices. They are self-contained in the sense that they do not depend on the sender's or the receiver's environment. They are data objects because they may take one of a number of forms, and their definition is open-ended. Messages are typically used to relay specific pieces of information between machines, as well as humans.

Messages have several common characteristics.

- *A sender*: each message originates from somewhere, either from a person or a computing device. The identity of the sender is usually included with the message, although that identity can rarely be trusted.

The sender can be one of many entities. Messages can be original person-to-person communication or generated by a computer.

- *An intended destination*: messages can be sent to a single destination or to a group. At some layer in the transport system, messages are always sent device-to-device. The final destination can be a human reader or an application.
- *Timestamps*: a message is typically given information about the time and date when an operation is performed on it. A message can receive many timestamps and can have many operations performed on it. An email message, for example, may travel through several relay points before arriving at its final destination and bear timestamps from each relay point.
- *Content*: a message usually carries content information to its intended destination. While the content could be empty, it is still considered to be part of the message's definition.
- *Format*: different types of messages take different forms. However, messages tend to have a common organizational format: a header and a body. The message header contains information about the message itself, such as sender, destination, delivery options and timestamps. The body of a message contains its content, that is, the information the message was meant to convey to its recipient. Beyond this general structure, messages vary widely in how they represent or format each message section.

Consider the email in Figure 13.1. This email message is in the form that is exchanged between two devices (not necessarily in the final form a person might read).

This message has a typical format. The header is separated from the body by a blank line (about two-thirds of the way down the message). You can easily spot the sender and destination by the 'From:' and 'To:' lines. The header contains several timestamps. The 'Received:' fields display information about the relays the message went through. Note that there is a lot of information in the header, more than is typically useful. In this case, the body of the message is an ASCII-based textual message.

Message sending and receiving typically involves both servers and clients. A typical scenario is depicted in Figure 13.2. The sender composes the message on his or her local device. The message is then 'sent' – which means it is uploaded to a server, either over a conventional, wired network

```
Received: from mail.brainshareproject.com (mail.brainshareproject.com)
      by smaug.cs.hope.edu (8.9.3+Sun/8.9.1) with ESMTP id AAA09235
      for <jipping@cs.hope.edu>; Wed, 27 Jun 2006 00:33:11 -0400 (EDT)
Received: from debian (dialup-209.Dial1.Level10.net [192.245.239.242])
      by mail.brainshareproject.com (EL-8_9_3_3/8.9.3) with ESMTP id VAA14057
      for <jipping@cs.hope.edu>; Tue, 26 Jun 2006 21:32:46 -0700 (PDT)
Received: from jjones by debian with local (Exim 3.22 #1 (Debian))
      id 15F70M-0000UR-00
      for <jipping@cs.hope.edu>; Tue, 26 Jun 2006 23:32:42 -0500
Date: Tue, 26 Jun 2006 23:32:42 -0500
From: John Jones <jjones@brainshareproject.com>
To: jipping@cs.hope.edu
Subject: Infomercials.org
Message-ID: <20060626233241.A1876@brainshareproject.com>
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Disposition: inline
User-Agent: Mutt/1.3.18i
X-StarTrek-Quote: Make it so.
Sender: John Jones <jjones@brainshareproject.com>
Content-Length: 701

I want to bounce this site off you:
      http://www.infomercials.org/
and hear any opinions you have on the material it presents.  They seem to
have a very well-thought-out approach to managing large (or small) numbers
of customers.

John J.
```

Figure 13.1 Example email message

or a wireless network. This server, known as a message center or relay server, must now deliver the message.

There are also delivery methods that work on a peer-to-peer basis. These methods allow local message composition and delivery of the

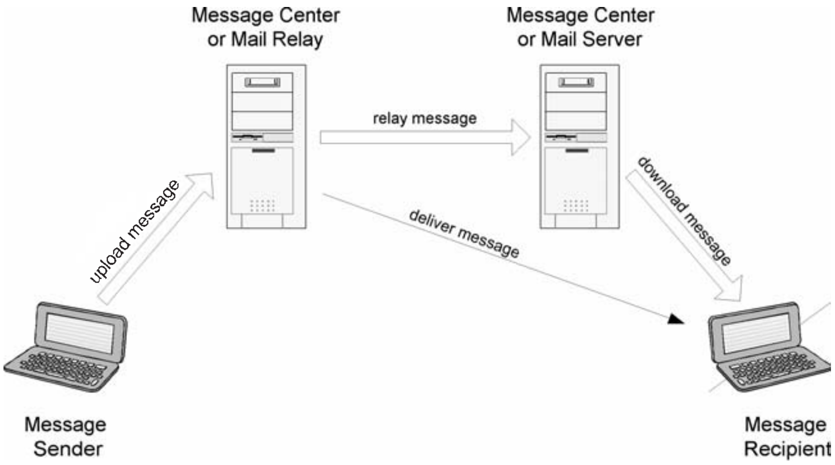


Figure 13.2 A typical message-delivery scenario

message directly to the recipient. The message-relay server is removed from the loop.

In general, there are two models for message delivery:

- *push model*: if it can, the server delivers the message directly; it contacts the destination device and pushes the message to it; this requires that the destination is ready to receive messages; SMS messages are examples of this type of delivery
- *pull model*: when a device is not usually connected to a wired or mobile network, the server stores the message for the recipient; the receiving device must contact the server and pull its messages from the server; often, in this model, the message-relay point is not the message-storage point; the storage server receives the message from the message relay and keeps it for the recipient; email messages are examples of this type of delivery system.

Note that the pull model may involve the push model. A device might pull its messages by notifying a server that it is online and ready. The server then uses push-model mechanisms to deliver content to that device.

Electronic mail

Electronic mail is one of the most common and widely used forms of messaging. It was originally developed as an electronic means of sending text-based messages – textual 'mail' for human consumption – between computers. As the value of sending messages was realized, people began to send other things as well. Email has evolved to encompass all types of objects, for example, programs, spreadsheets and word-processing documents.

The format of an email message is shown in Figure 13.1.[1] As we have stated, it has a header and a body. The header is comprised of a sequence of lines or fields, each of which is composed of a key and a value. Each field relays information about the message to the receiver. The sender is required to insert a 'From' field, a 'To' field, and a 'Posted-Date' field into the message header and is free to insert other fields. In addition, any field whose key value begins with the string 'X-' may be inserted by the user (note the `X-StarTrek-Quote` field in Figure 13.1).

[1] The most widely used format for email messages is specified by the Internet Engineering Task Force in a document called RFC 821.

The body of an email message contains the message content. This content is typically composed of a message in ASCII or Unicode text. However, the message body can also have *attachments*, which are data objects that accompany the message. These objects are included using a standard called Multipurpose Internet Mail Extensions (MIME). Objects included using MIME are each included in a message type format, with a header to identify the type of the data object and a body that contains the data object itself. Many objects can be included in a message.

Even when it includes MIME objects, an email message has a text-based representation. All email is sent using readable (ASCII/Unicode) characters. If a message contains attachments that are not comprised of text, then those attachments are encoded in a special way to derive text from them. The encoding method used, e.g., base 64, is included in the header for the data object, so the receiving software can decode the object. The delivery of email messages to the end user follows the pull model of message delivery.

There are several protocols that are used to send and receive email. By far, the most widely used sending protocol is Simple Mail Transfer Protocol (SMTP). There are two protocols used to receive or read email: Post Office Protocol (POP) and Internet Message Access Protocol (IMAP). These are TCP/IP-based protocols.

Look at Your Email

Look at your own email messages to verify the format we specify here. Save your email to a file and examine the file's contents. If you can, find a mail message with attachments and look at the contents. You may have to specify a 'save headers' or 'save all headers' property to your mail reader; often a mail reader saves only those header fields that it deems interesting. Use the headers to track the path your email message has gone through to reach you.

SMS messaging

Email messages are meant to be in a general form that adapts to most computer systems. They are text-based, so most devices can read and relay them. They are flexible enough to accommodate many different types of data objects. By contrast, SMS messages are very specific and targeted by their nature to a specific carrier technology.

SMS messages are short messages, 160 characters or less, and are specifically targeted for mobile phones, usable on a wide range of networks. They are data messages which are not intended to be viewed until they are decoded and displayed. The sending of SMS messages adheres to the push model of message delivery. Messages are sent to a service center that relays the message and delivers it to its destination. The service center contacts the receiving device and keeps trying until it finds the device powered on and receiving messages.

An SMS message follows the standard message format in that there is a message header and a message body. The header contains information about the message and the body contains the message itself. The SMS message below contains the message 'hello':

```
07917283010010F5040BC87238880900F10000993092516195800AE83229BFD46
```

Note that the data is actually a stream of bits and written above in hexadecimal. If this message were to be received on a GSM phone, chances are that phone would display 'hello'. It is fairly obvious that we are dealing with a different type of message than we dealt with for email.

Let's examine this message and by this example examine the SMS standards for messaging. The table below analyzes the pieces of the example SMS message above.

Data Field	**Description**
07	The length of the service center information. In this case, the number is 7 octets.
91	The type of service center address. In this case, 91 means the phone number has an international format.
72 83 01 00 10 F5	The service center address in 'decimal semi-octets'. Although formatted like octets, the number reads in decimal digits. Because the service center address has an odd number of digits, it is padded with F (all ones) to pad out the octet. Here, the number is +27381000015.

Data Field	Description
04	Type of message. This is an SMS-DELIVER message.
0B	Sender address length. Here, 0B means the sender address is 11 digits.
C8	The type of the sender's address.
72 38 88 09 00 F1	The sender's address, in decimal semi-octets. Note the padding. Here the address is +27838890001.
00	A protocol identifier, establishing the way we send the rest of the messages.
00	The data coding scheme. SMS messages can be sent in many types of encoding; the most popular, used here, is 7-bit data.
99 30 92 51 61 95 80	This is the sending time stamp in semi-octets. The first 6 octets represent the date, the next 6 represent the time and the last two represent the time zone.
0A	Length of the message, in this case 5 octets.
E83229BFD46	The actual message, where 8-bit octets are used for 7-bit data.

Let's take note of a couple of things about this message. First, the semi-octet format of the addresses and the timestamp is odd, but readable. Note that the octets are swapped in this representation (shown in a little-endian manner). Second, this message is in 7-bit 'default alphabet' format. This is an alphabet of 127 characters that contains many of the most often used international characters. This is a GSM standard. Finally, to compress as much as possible, the 7-bit representation is encoded in the 8-bit quantities in a special way.

SMS messages can have many forms. In the GSM standard, the messages can, for example, be faxes or pages. The standard also allows email messages to be sent to GSM phones by way of SMS. In this latter case, the mobile device is able to treat the message as an email message and perform email operations on it (like replying to the message, for example). Adapting SMS messages to these other forms requires both a service provider that can perform such conversions and software on both the sender's and receiver's devices that can handle these adaptations. For example, the service provider probably has to provide mapping between an email address and a phone number and software must be used to cut large messages into smaller messages that can be sent over SMS.

BIO messaging

Bearer Information Object (BIO) messages are messages meant for the receiving device, not the user. These messages contain structured data objects of a known, predetermined format. They can be delivered using various transports, for example, email, SMS and IR.

Various data objects can be sent as BIO messages, including SMS configuration messages, configuration settings for various applications and application data objects.

A good example of a BIO message is one that contains ringtones. A ringtone is a tune that a mobile phone plays to alert its user of some condition – say, a call coming in. While mobile phones have many unique tones to use, many phone manufacturers also provide the ability to program ringtones and send them to the phone. You can have your favorite movie theme song play when you get a phone call. Ringtones are programmed using a textual 'language' that can be encapsulated in an SMS message and sent to a mobile device. Because of directives in the message header, the phone processes the message rather than display the message on its screen. Processing a ringtone means decoding the specification, storing the resulting tone and incorporating the tone in its list of tones.

Another good example of BIO messaging is the exchange of vCards. They are electronic versions of business cards, containing names and contact information. The specification of a vCard is textual and can be included as an object, for instance, in email. By sending a vCard to a device, by IR, for example, your contact information can be automatically inserted into the device's directory. By indicating in the message header that the vCard message is a special message, the device intercepts the vCard and records it, rather than displaying it.

vCard and vCalendar Objects

Virtual business cards and virtual calendar objects are a standardized way to exchange information about contacts and agenda items between devices. These can be attached to email or sent to another device via methods such as IR or Bluetooth. They are textual specifications and, as such, are flexible and adaptable to many different transports and devices.

The Internet Mail Consortium (IMC) governs the maintenance of the standards on these objects. The IMC is a group of computer companies that includes Symbian. Their website is at ***www.imc.org***.

Fax messaging

As a messaging technology, facsimile transmission – that is, the electronic transmission of images over phone lines – developed in parallel with computer communication. As standards were developed regarding digital messages and their transmission, fax standards were developed independently. In order to integrate a fax standard with the messaging standards we have discussed, some adaptation has been required.

A fax message is actually an image. Before computers were used to send faxes, fax machines were developed to scan a piece of paper into an image in the machine's memory, to transfer this image to other machines via a modem and to print the image from memory back to paper. As computers got involved in this process, they eliminated the need for paper, and the fax image could be converted from its native format to one of the more standard image formats that computers use (e.g., Gif or Jpeg formats). Modems have been adapted to include fax capability.

The faxing model is compatible with the messaging models we have discussed. Sending a fax follows the push model of messaging, where the sender keeps trying to send the fax until the intended recipient fax machine can receive it. As a message, a fax transmission has a body: the image that is transmitted. We have to stretch the model a bit to find a message header, however. For a fax message, implementations typically consider the cover page of the fax transmission to be its header. It contains information typically found in a message header – source and destination information, for example – and it precedes the message body.

The data format of a fax transmission is specified by the Comité Consultatif International Téléphonique et Télégraphique (CCITT), now known by the name of its parent organization, the International Telecommunication Union (ITU). The format of the graphics image is specified but is outside the scope of this book. It is the job of the sending machine to convert any text or images to that graphics format, to use the fax mode of an attached communications device and to push the graphics data stream through to the recipient.

Sending a fax is typically done through a modem with fax capability. There is also faxing capability built into the GSM standard. Mobile devices that use GSM therefore have the capability of sending fax messages through the GSM service.

The Fax Image Format

You can find more information about the fax image format on the web. However, CCITT Group 4 standards are a bit hard to find, because fax images are specified as TIFF class F images. More information on TIFF class F can be found in [Campbell 1990].

Message Modeling

Messages have several common characteristics. Figure 13.3 depicts these characteristics and adds a few more. Messages are composed of *delivery information* and *content* and are generally characterized by *message types*.

Delivery information is composed of sender information, destination information and timestamp information. Email messages, for example, contain all this information textually in the header of a message. Sender information is contained in the 'Received:' and 'From:' fields; destination information is kept in the 'To:' field; and timestamp information is found in the 'Date:' field.

Content has three parts: properties, message content and possibly a set of data objects or attachments. An email message can again serve as a straightforward example. There are pieces of message header that specify certain properties of the message; the `Content-Type:` and

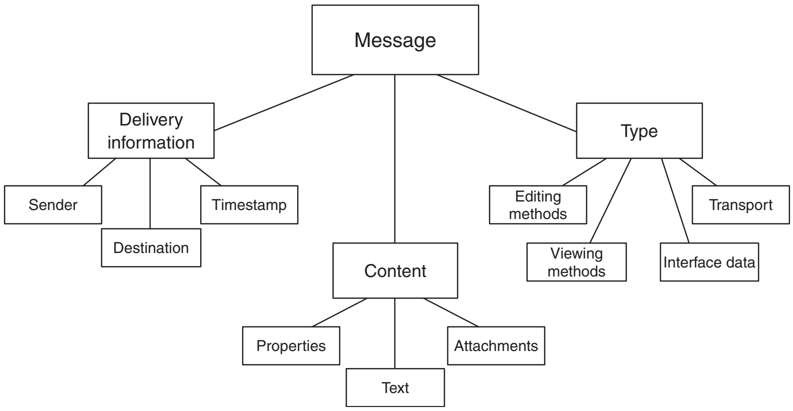


Figure 13.3 The relationships between component parts of a message

`Content-Length:` fields, for example, indicate the MIME properties and the length of the body. The message itself is contained in the body of an email message as text. In email, data objects can also appear in the message body as MIME attachments.

Each message also has a message type. The type of a message is the definition of a larger class of messages that describes general characteristics of that class. A message type cannot describe the specific contents of a message, but it can describe properties of the message class. These properties include:

- *the editing function*: this property is a description of how to edit a message of a particular message type; on a smartphone device, this 'description' takes the form of an implementation of a message-editor application
- *the viewing function*: this property describes how to view a message of a particular message type; on a smartphone, this 'description' can be an implementation of a message-viewer application
- *user-interface data*: there can be certain data associated with user interfaces that deal with a certain message type, including items such as icons and progress-dialog interfaces to display for the message type
- *the transport function*: messages of a certain type are transported in the same way to and from their destinations or repositories; the implementation of this transport is associated with the message type and not each individual message.

Let's look at email messages again. All messages that are of the 'email' class of messages might be viewed the same way – through an email viewer that can present the textual message with tools to view header information as well as any attachments to the email. There is a common composition (editing) interface that you can use to compose new messages or edit draft messages. There is a set of user-interface definitions – some icons for email applications to display, for example – that can be accessed. Finally, email is transported to a server via SMTP and received from a server via POP3 or IMAP4. (Since there are three transport functions, one could argue that there are actually three message types for email. Symbian OS views it this way as well.)

If we expand our view to other types of messages, we can see that the framework established for Symbian OS works for any type of message we have encountered. Fax, SMS and BIO messages all have this format, albeit with different implementations for each component.

13.2 The Symbian OS Messaging Model

We examine Symbian OS as an example of an operating system built to handle messaging in its basic design.

When building the messaging system, Symbian OS designers were guided by the need to put together a generic framework that could handle the components of many different message types. They built a framework that viewed all messages as composed of generic components. Through the use of object orientation, each message type is handled by a separate implementation of those generic components. Symbian OS message architecture brings these implementations together under a common messaging application and a common API.

Symbian OS uses *message type modules* (MTMs) to define message types. An MTM is composed of four classes that are used as base classes for specific message-handling implementations:

- the user-interface MTM is defined as the `CBaseMtmUi` class and defines user-interface capabilities, such as viewing and editing messages
- the client-side MTM defined as the `CBaseMtm` class handles the interface between the internal representation of a message's data and the user-interface MTM

- the user-interface-data MTM is represented by the `CBaseMtmUiData` class and provides access to user-interface-data properties (e.g., icons)
- the server-side MTM is defined in the `CBaseServerMtm` class and provides message-transport capability.

Each message type, then, has an implementation of each of these MTMs, subclassing their definitions from the classes above. Some message types can share MTMs; IMAP email and POP email, for example, can share the same user interface MTM while having different server-side and client-side MTMs.

Some Perspective on this Architecture

This way of modeling messages is powerful and very effective. Through various MTM implementations, Symbian OS supports its four main message formats (email, SMS, fax and BIO formats) as well as several formats that work behind the scenes to implement other protocols.

It is interesting to examine the disparity between the various forms of messaging under the MTM framework. Fax messaging, for instance, is vastly different from SMS messaging, yet both fit comfortably in this model. Although a fax has different delivery requirements from an SMS message and must be viewed as an image, where SMS can be viewed as text, both can still be integrated with the same messaging application on a Symbian OS device.

The large number of different forms of messaging results in a large number of MTM implementations. Any messaging application must be able to sift through these implementations to find the necessary MTMs as quickly as possible. Symbian OS helps in this endeavor by providing a registry of installed MTMs that are accessible to an application. This registry allows MTM components to be identified and instantiated quickly and easily.

Now consider this structure from a programmer's perspective. Imagine you are a programmer who wants to implement the sending of an SMS message as a small part of a larger application. It would be quite daunting for you to use the MTM framework in all its glory to send a 'simple' SMS message. In fact, you might just reject the idea as not worth the time and effort. The designers of the MTM model understood that a complicated structure might frustrate programmers who want to do simpler tasks, so they added something to the architecture called *send-as messaging*. Send-as messaging provides a simple interface that allows applications to create outgoing messages by using a single API. By using send-as messaging, a programmer uses one interface for any message and simply informs the OS what type of message to send; the OS implementation figures out which MTMs to use and how to send the message. This is a powerful idea and it reinforces the modularity of the MTM structure. It has proven to be an easy and effective interface. It is so effective, in fact, that many applications use it to transport data other than traditional messages. The Agenda engine, for example, uses send-as messaging to send vCalendar objects over the IR interface.

Message Server Functions

Symbian OS considers the data repository for messages as a resource that can be shared between processes. Therefore, following Symbian OS and microkernel standards, there is a server that protects this repository and manages access to it. Any application that wants to handle messages in some way must become a client and make requests to the message server. The message server implements two valuable functions.

- *Access to the message-data repository*: as clients request message access, the message server delegates temporary, exclusive access to message data. The message server must keep the message data repository correctly ordered and must cope with anomalous message events correctly. For example, access failures or incomplete sessions must not corrupt the message data.
- *Access to MTMs*: the message server must enable applications to identify requests, such as the sending of a message, that require protocol-specific functionality and load the appropriate MTM.

The message server accepts requests from client applications that require access to messages and MTMs. These requests are handled asynchronously, as they might require a large amount of time to perform (fax messages, for example, typically take much more time to process than email messages). Requests can include changing the structure or contents of the local folders, sending and receiving messages through different services, changing the structure or contents of remote mailboxes, and MTM-specific requests. Note that actions such as delivery and storage concepts such as local folders and remote mailboxes are all managed by the message server.

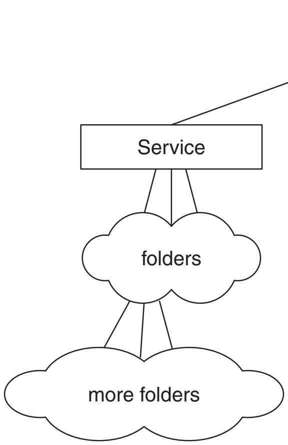
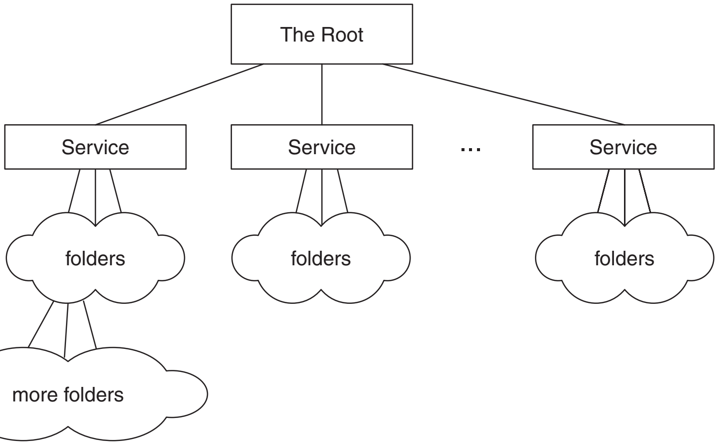



Figure 13.4 The Symbian OS message model structure

Storage and Message Structure

The Symbian OS message store has a specific structure. The diagram in Figure 13.4 shows an overview of this structure.

Messages are stored in folders, which are accessed as children of services. A service can be viewed as a message source: an ISP is a service, as is local, on-device storage. Folders can be user-defined or system-defined; for example, in and out boxes are defined by the system and are applicable under the local storage service. Folders can have subfolders, which can have subfolders, etc. This hierarchical organization of the storage structure means that access to messages requires a walk through the storage tree before access is granted to a message.

Messages themselves also have a structure, shown in the table below:

Message Component	**Type of data**
Generic message header	Message-index-entry data
MTM-specific information	MTM-stream data
Message body	Rich text data in message store
Attachments	Attachment files stored in the message's store

Message-index entries have a specific format. That format is quite large and we will not reproduce it here. Suffice to say that both delivery and content-header information is stored here and that this information varies by MTM. The MTM-stream data is specifically formatted for the type of MTM that is used for the message. Obviously, this information also varies by MTM. The message body is a formatted object that contains text as well as formatting information. Details about attachments are found in the message header – the index portion – but the data files themselves are stored in their own format in the message store after the message body.

Manipulating Messages

To appreciate the Symbian OS message architecture in motion, let's take an overview look at what it takes to send and receive messages.

To access messages, we must create and manipulate several Symbian OS objects. First, we must create a session with the message server. Next, we must initialize contact with the message-system registry in such a way that we can create MTMs later to handle messages. Finally, we access the message system by walking through the message structure tree: we access the system root, then find the message's service, then the message's folder (and possibly subfolders), and then we access the message.

A session with the message server is initiated by system calls from the client. Sessions can be opened synchronously or asynchronously.

After establishing a server session, the next step is to open the registry so that we can access MTM objects when we need them. There are three kinds of access into the registry: one to access client-side MTMs, one to access user-interface MTMs, and one to access user-interface-data MTMs. Each access is implemented by its own class and their own system calls. They are derived from the same base classes and have similar definitions.

Once we have a valid message-server session established and we have access to the registry, we can access the message-tree structure. Let's look at the characteristics of an entry in the message tree. An entry includes the following properties:

- an entry ID
- the header information
- an 'owning service', i.e., the parent of an entry
- a certain number of children (for the root and inner-tree nodes)
- file storage
- an MTM list.

Behind the Scenes

Stop for a moment and consider what is going on behind the scenes of remote-message transfer. The implementation of this is different for different message types.

Consider a session with a remote email server. The local device's copy and move operations access the server through the existing socket and pass commands to the server through this socket. These operations do not need additional connections as one already exists.

Finally there are implementations of different MTMs, which include functions that are specific to each MTM and do not generalize to all MTMs. For example, an email-message MTM that uses POP3 would need functions to connect to and disconnect from the POP3 server. A BIO-message MTM would not need a function to connect to a server because there is no server for BIO messages (they are pushed to a device). On the other hand, a BIO-message MTM needs a function to process the BIO-message content. This applies to SMS and fax messages as well. Each MTM implementation requires its own system calls.

The messaging architecture provides a way for MTM components to offer protocol-specific functionality not provided by base-class-interface functions. These MTM-specific functions are implemented in the MTM and assigned IDs that correspond to each protocol-specific operation offered.

Easy Sending of Messages: Send-As Messaging

By now you might be a little taken aback by the complexity of message handling and the message architecture of Symbian OS. While it is true that the architecture is big and complex – after all, it must handle messages as different as faxes and SMS messages – the designers of Symbian OS have streamlined the process we use to send messages. The sending process is relatively short and very straightforward. Let's review that process in this section.

The sending procedure is called *send-as messaging*. It is a generic process – i.e., one process for all message types – that uses a message type's MTM to guide the sending process. It is a powerful model that can be used to send messages of all types and even to transfer data in unique ways. The procedure follows the sequence we outlined in the last section.

Send-as messaging centers on the `CSendAs` class and is accomplished by the following steps:

1. Choose the MTM that the `CSendAs` object uses to send the message. This can be done in two ways: we can set the MTM directly or we can use a `CSendAs` object to search for the MTM we need.
2. Choose the service to use for the outgoing message.
3. Create the message (and all its parts) as a `CSendAs` object. Once the message for the `CSendAs` object has been created, we can create and modify its components: the recipients, the subject, contents of the body, and so forth.
4. Save and send the message means saving it into the appropriate service's message store.

Receiving Messages

Applications can work with the messaging system and message arrival. While the message server itself handles the listening and message-transport functions, applications can register themselves to be notified upon message arrival.

Registration occurs when a connection with the message server is established. Recall that connecting with the message server requires a parameter that is an object of the `MMsvSessionObserver` class.

The message server calls a specific function from the class object passed during the server connection whenever a 'message event' occurs. A code that signifies which event occurred is passed in the first parameter and up to three pointers to data areas that apply to the message event make up the rest of the function call's parameter list.

13.3 Message Handling in Linux

Where messaging may seem quite complex on Symbian OS, it is positively simple on Linux. As an illustration and contrast, let's overview how messages are handled by the Linux operating system.

Message Models in Linux

As we have seen before, Linux designers usually take the simplest route they can when dealing with complex situations. Messaging is no exception. There is little built-in support for messages or message types in the Linux kernel. Any message modeling or message structure must be built and configured by applications or by third-party developers designing modules for Linux. Email messages represent an exception: there is some small support for email in Linux.

Linux message structure is quite straightforward. Text files represent email messages; any other kind of messages are built and used by the applications that implement them. Even attachments to email require message-processing programs supplied outside of the operating system. Linux supplies the mechanism to receive messages and place them in a file. Any reading of messages must access the message file and process that file in a way appropriate to the message type.

In a sense, this approach to messages utilizes abstraction – except that it is not built into the operating system. Linux supplies the base operation and assumes that any further processing is provided by an outside source. In like manner, Symbian OS provides the base processing and assumes that further processing of a message is supplied by outside MTMs. The major difference between these two is the lack of a model in Linux. Linux simply assumes that applications are provided to process the text file message that has just been received.

Sending Messages

The communications medium over which messages travel is a shared resource in Linux. Whether it is a network or a Bluetooth connection, Linux usually provides a server that shares the resource between the processes that require it.

The reasoning for these Linux servers, however, is a bit different than the reasoning behind servers in a microkernel operating system like Symbian OS. For Linux, servers implement protocols. These protocols are complicated enough that a server is designed to handle the complex nature of the communication. The server implementation is an assistant to developers that want to use the resource, but that is all. The server does not protect the resource; in fact, if an application were to want to use the resource, the system would gladly allow the access. This means that, while sending messages in Linux is aided by opening a connection to a server, that connection is not technically necessary.

However, sending messages in Linux is helped by various message servers. Consider email. There are several email servers that run under Linux; a popular application for this service is the 'sendmail' server. A sendmail server receives an email message from a client, processes that message by reviewing the textual message contents and opens a connection to a remote server by way of delivering the message. There is no special format other than the format of an email message itself (which is public-domain specification).

There is nothing special about an email server. In fact, if an email client wanted to send its email directly to the receiver's email server, it could certainly do so. The network connection would be shared by the operating system through the abstraction of sockets. The 'sendmail' process could easily be bypassed.

Receiving and Delivering Messages

As you might expect from a Linux system, receipt and delivery of messages is very simple as well. The computers that are configured to receive messages run a server that listens for incoming messages, receives those messages and stores them. Again, a server is used because it is convenient not because it is managing the shared communication medium.

Email again makes a great example. The 'sendmail' server processes incoming messages as well as outgoing messages. It listens for connections – as it always does – and delivers messages to local as well as remote computers. Notice that this action is the same action we discussed in the last section. The server's actions are very straightforward – receive a message, determine the destination, and either send it on or store it locally. Local storage is as a text file.

Sometimes the simplicity of Linux is a liability. The simplicity of 'receive a message' and 'deliver a message' has degraded system security in the past. A practice called *relaying* is used to send an email from an outside party to an outside party. This would not be such a bad practice were it not for spam email or unsolicited email. Often, a relayed message comes into an email server with 100s of email addresses as destinations. The resulting delivery is not only a burden on the delivery server, but it also shields the sender's address. Relaying is a practice that is on the decline, mostly because of new verification mechanisms built into Linux email servers.

13.4 Summary

In this chapter, we have discussed the messaging frameworks of operating systems. Since messages vary widely in content but can be roughly categorized by form, frameworks have been designed to be generic enough to handle all forms of messages and to allow specialization of message handling.

We reviewed the components of a message and outlined the structure of message implementations for each message type. We then discussed the way that Symbian OS addresses messages, from the message server, the process that manages messaging stores and facilities, to the detailed ways that we interact with the message server to read and send email. We also discussed 'send-as messaging', which is a convenient way to send messages through a simple interface, using functions common to all message types. We then contrasted the Symbian OS approach with the Linux approach. We noted how a simple, usually text-based, approach to messages can be abstract, allowing for the same interfaces to work with many different message types, and also allow for the other implementations to co-exist with current servers.

Exercises

1. Why are email messages always in readable text?
2. Is sending a fax an example of the push or pull model of messaging?
3. Why does Symbian OS use message servers? Why are clients not allowed to access messages directly?
4. Let's say that a new message format is invented that is like the SMS format, but can be sent over a local area network. If you had to implement a receiver for this kind of message, would you rather use the approach of Linux or Symbian OS? Explain your answer.
5. Smartphones have powerful computers in them. Why would it not work to make a smartphone into a mail server?

14
Security

The Great Wall of China is an amazing feat of architecture. Started during China's Zhou dynasty around 600 BC, the Great Wall still stands today. It has been fortified and repaired and today runs for over 5000 kilometers through northern China. Its purpose was chiefly defensive in nature: it was a protective barrier between China and marauding tribes to the north. Protection – keeping those who do not belong out – was the reason the wall was built. When the wall was breached for the last time by the Manchurians in 1644, it is said that the wall was still doing its job and that a weakness in the government allowed others to take power.

While the Great Wall provided protection, it did not provide security. Protection is a barrier keeping everything out. Security allows certain things in. Security represents 'smart protection': protection is part of what it does but there are added elements that determine if entrance into a secure area is allowed.

In the context of an operating system, security has several facets. There are many levels that must be secure. There must be a consideration of the environment external to a computer system. Access to system elements must be protected and authorized access must be granted. Security needs to prevent malicious destruction and accidental misuse but allow permissible access. Protection means more than simply preventing access; security requires more than allowing entry into the system.

This chapter explores what security means to computer systems in general and smartphones in detail. We examine the ways that data can be misused and corrupted and present ways to guard against malicious misuse.

14.1 Understanding Security Issues

It has been said that the only truly secure computer is one without power – turn a computer off and it is fully secure. Security is difficult to implement correctly.

In fact, total security cannot be achieved. A system is secure if its resources are accessed and manipulated as intended in all circumstances. This implies a guarantee, something which cannot be given. Security violations occur because someone tried a way that was not blocked. Unfortunately, system designers are only human and the components of complex operating systems sometimes interact in unforeseen ways.

A classic security problem was revealed in Unix systems in the 1980s. Unix had (and still has) a command called 'finger' that queries another computer to see user information. Someone discovered that if you give the name of a user in just the right way, you can overflow an internal buffer (which was fixed at a static size) and push the overflowing data into the executable part of the program. By structuring the query to contain executable instruction data, a person could send a 'query' that was too large and force the finger server to execute the code that had overflowed into the executable data. And since the finger server ran with very high privileges, a person could run programs as the system administrator. All this because a programmer put a static bound on an array!

While we cannot guarantee the protection of a system, it is possible to make the cost of system access very high. Operating system security measures must secure all foreseen methods of accessing a system so that using an unforeseen method is very costly. To make this cost high, security must be exercised at four levels:

- *physical* access to the computer or device must be secured against intruders; this means securing room access or keeping track of a mobile phone
- *human* users must be screened to ensure that system access is done by trusted individuals and those that access a system are who they purport to be
- *network* access is implemented over wired lines or wireless connections, using Ethernet and mobile phone technologies; networks carry data and provide a way to break in
- *the operating system* must protect and secure itself; all access must be screened to determine if it is proper or not.

Both network and operating system security depend on a secure physical environment and access from trusted individuals. No matter how secure the operating system of your phone is, putting it on a table and walking away encourages someone to steal it and access your data. Allowing access to data to a person you think you trust who then gives that access to malicious users (perhaps for money) cannot be predicted or prevented by an operating system.

Because of the implications of an insecure system, it is worth considerable time and effort to make systems secure. Often this seems like a losing game. For example, designers that are working on securing Linux also publish the source code to the operating system. Such open source code is scrutinized by far more people than are working on the implementation. While hundreds might be working on security implementation, thousands might be using the source code to gain access.

The remainder of this chapter focuses on operating system security. The other areas of security, especially physical and human security, are beyond the scope of this book.

14.2 Authorization

When a system function is used or data is accessed, there is a fundamental assumption that the access is authorized. It is rare for an operating system to ask for authorization before performing these functions (but it does happen occasionally). We explore what authorization means in this section.

Authorization means 'to be given authority'. In turn, to be given authority implies two things: it did not exist before and it was given by some other authorized entity. So if you are authorized to do something, you probably had to ask for authorization. The person who authorized you verified your request and granted you authority. Often, authorization is demonstrated by a token or symbol that is recognizable. A police officer is usually authorized by his uniform; a plain-clothes officer requires a badge to show her authorization. Sometimes, however, authorization is not questioned. In this case, authorization is assumed or not required. For example, it is not typical to need to show authorization to enter a public library; the assumption is that anyone may use the library, so any use does not need authorization.

In an operating system, a process carries information about itself that can be used as tokens of authority. A process has a process ID and owner and group designations. It also records the date and time of its creation and which process created it. In most cases, this information set is enough to pick from. This information is assigned when a process is created, derived from the parent that created it.

Take, for example, a process hierarchy in a Unix system. Upon login, a user can be granted shell access. The shell is a process whose job it is to communicate with the user and execute commands on his behalf. The shell process has the owner and group information assigned to it by the operating system login process. Any process spawned by this shell derives owner and group information from the shell. If the shell is authorized to do something, a command spawned by the shell is authorized to do the same.

Sometimes authority in a computer system is given to any process. In Microsoft Windows 98 and earlier versions, authority was given to any process simply because they were running on the system. These versions of Windows did not require authorization to perform operating system tasks. For example, if you were using the computer, you could delete any and all files on the system's hard drive. Even in more recent versions of Windows, the permissions on files have been set to allow maximum access with minimal user security.

14.3 Authentication

Because authority is mostly assumed in an operating system, getting that authority is a function that must be administered carefully. If a process is to grant access to the computer system to someone, the identity of that person must be verified or *authenticated.* Authentication is the verification of identifying characteristics and is an extremely important part of security, because, as stated in the previous section, authority is often not verified. Authentication is usually user-based. A user must identify herself in a manner that the system can verify. Authentication is usually based on one or more of three elements: what you know, what you have or who you are.

What You Know

Authentication based on what you know usually takes the form of some kind of password or 'passalgorithm' system. It is very common to base security systems on passwords. Password systems usually ask for a user identifier and a password that has been assigned to that identifier as the basis for authentication. The user identifier is probably public knowledge but the password should be unique to a user.

Passwords work on many levels. They are most often used to gain permission to use a computer system. If system security is more fine-grained, passwords can be applied to system resources. The network device, for example, may be password-protected in many operating systems and its use forces the operating system to ask the user for the password. An even finer-grained approach could allow different passwords to reflect different access rights: one password would allow reading a file while another would allow reading and writing.

While passwords are common, they are not foolproof and have proven very vulnerable in the history of operating systems. The problem with passwords is that they must be remembered for the user to use them. This means that the temptation to make them easy to remember is very great. And if a password is easy to remember, it is also easy to guess. The most common type of attack against system passwords is called the *dictionary attack*, which simply walks through a dictionary and tries all the word and variants on those words as passwords. Such attacks are easily done and well documented (as are ways to foil such attempts).

Password storage is an issue that can make an operating system vulnerable. On open-source operating systems such as Linux, where the password-storage methods are available for all to examine, attacks are frequent and passwords are vulnerable. Often passwords are stored in a location separate from other user information, a location that can only be accessed by processes with high system privileges.

Password encryption is often used to safeguard storage. To make encryption difficult, encryption algorithms often use salts or keys to get started. These are character sequences that are used by the algorithms to offset encoding in special ways. Unfortunately, the salt or key must be known ahead of time to reproduce the same encryption – or to produce a decryption. This means that the salt or key must also be stored on the computer system and could be accessible by those seeking to break passwords.

Consider Unix password encoding. Unix uses a well-known encryption algorithm that needs a two-character salt to get started. The algorithm produces an output – the encoded password – which is combined with the salt that was used and stored. Anyone with access to the password would also have access to the salt and could decrypt the password. However, while Unix designers used a well-known encryption algorithm to encrypt, they did not provide the facilities to decrypt. And the well-known algorithm they used was a very complicated one. This means that Unix can easily encrypt passwords but not easily decrypt them. If you provide Unix with your password, the only way to verify that your password is correct is to encrypt it and to compare the encrypted versions. This is why a system administrator on a Unix system cannot tell you what your password is; she can only change it.

Passwords are not the only authentication method that focuses on what you know. *Passalgorithms* are algorithms used to derive a password or phrase. For example, a passalgorithm could be as simple as the current day of the week concatenated with the day of the month. So the password on Monday, June 26, becomes 'monday26'. Often these types of algorithms are used in challenge–response security systems. When a user wants entrance into a system, the system generates a random word or phrase and asks the user for the password, based on working that phrase through the passalgorithm.

What You Have

Sometimes, passwords are not unique or secure enough and passalgorithms are too complex for humans to work through them quickly. Then it becomes a matter of computing passwords or otherwise authenticating with some kind of device that a user must carry with him.

Often, when high security is required, people use devices to generate passwords based on the time of day or some complicated algorithm. The same methods are used by the computers that verify identity: the same algorithms generate the same output and the output is compared. These are often used with challenge–response systems where the challenge is a character string to be typed into a handheld device that produces some other character string that the user answers with.

Who You Are

One element that no one can duplicate is who you are. There are aspects of each person that are truly unique and very hard to duplicate. These make great sources for authentication.

One aspect of an individual is personal information. There are several pieces of personal information that each person possesses. These include name, gender, birth date, age, government identification numbers, etc. When combined, these can form a unique information set that can be used to identify people. Personal information is used on many websites to gain access. For example, many banking websites ask for your government identification number and your name or email address. Combining these produces information that is sufficiently private and difficult to guess.

Physical characteristics are another personal aspect that is very hard to duplicate. Fingerprints, voice characteristics and retinal blood vessel patterns are examples of unique human characteristics that can be sampled for user authentication. Biometrics, as this area of identification is called, are useful when the means of sampling them is inexpensive. Currently, fingerprints are easy to sample – fingerprint readers come standard on some computers. Retinal scans are expensive to take.

When No Authentication is Used

We mentioned older operating systems that did not use authentication. One would think that all modern operating systems use authentication. There are several cases, however, where authentication still is not used.

Consider a situation where a computer is a single-user device, as is the case for smartphones. For these devices, ownership is equivalent to authentication and possession of a device means that one should be granted access to it. Computers in public places, such as libraries, are also systems that would be difficult to use if authentication was to be enforced.

Security is still required in these situations. If authority is granted to any entity, the main concern is that any action that goes on is assumed to be authorized. Some systems, such as those in libraries, simply accept this fact and realize that systems can be modified at will. The administrators of systems of this type simply reinstall the computers at regular intervals to erase any malicious programs that might be on them. Other systems ask many permission questions. When software is to be installed, for example, the system might ask if installation is intended – and ask it several times. Still other systems tighten security around specific system functions such as writing data to a file or installing applications – while allowing all other functions to continue unchecked. Symbian OS falls into this last group.

14.4 System Threats

We go through a lot of trouble authenticating users and meticulously granting authority to perform specific functions. This is because the world outside a computer system usually contains some person or application that wants access. There are always attacks on computer systems and there are constant attempts to gain access to computers.

The security of passwords is threatened by many things including the people using them. Humans are fallible people and they develop habits regarding passwords that are helpful to them but harmful to security. People use obvious information in their passwords or invent phrases that are easily guessed. Dictionary attacks take advantage of these bad habits. People also write down their passwords and place them in desk drawers or stick them to monitors. Unless passwords are chosen very carefully, and protected well, they can – and will – be discovered.

Trojan horses and spyware are ways to gain access to systems. When a program or application masquerades as one type of program but actually has more than one function (especially functions that are not documented), that program is called a Trojan horse. For example, a program that listed your files, but deleted them as it was listing them, would be a Trojan horse. Even worse are programs that install other applications when they are executed. Spyware does this: it piggybacks on an application, which installs and runs the spyware. The spyware replicates itself this way and consumes system resources.

Buffer overflows are a common threat to system security. As we mentioned before, a buffer overflow is a condition that results from an application trying to store data in a buffer that is too small. The result is data that overwrites adjacent memory locations. This overflow could corrupt data or write sections of executable code, as in the finger example. Buffer overflows can cause a program to crash, can corrupt data or can be a security breach. Preventing buffer overflows can prove difficult.

A virus is a program that 'infects' another program by embedding executable code in it. The next time the infected program is run, it does different things because new, embedded code is now executed. Viruses may infect many programs and are usually designed to propagate themselves. Infected programs are detectable by virus-checking software because infections follow a specific pattern and are usually targeted at specific programs.

Worms are programs that are introduced on systems without any permission. This may be through bugs in an existing program or through well-intentioned, but exploitable, features. A good example of this is a worm that spread through Microsoft SQL Server 2000. This worm used a buffer overflow, which existed in the way SQL handles data sent to its Microsoft SQL monitor port. The worm would send data to the monitor port, which processed the data and executed code during the processing. The buffer overflow wrote code into the executable portion of SQL Server, and the data processing was the attacker's code. In addition, since Microsoft SQL Server 2000 runs with system administrator privileges, the attacker's code also ran with such privileges. Fortunately, the worm did not contain any additional malicious content; however, because of the nature of the worm and the speed at which it attempts to re-infect systems, it caused a denial-of-service attack against infected networks.

Denial-of-service attacks are threats to networks. An attacker might flood a network with data, filling the network to capacity and making any other kind of network traffic difficult. An attacker might target a single service – say, a web service – and send a flood of HTTP requests, thereby shutting down service to requests from any other source. Denial-of-service attacks become even more insidious when they come from multiple sources – called distributed denial-of-service attacks. In distributed denial-of-service attacks, many computers team up to bring down a computer or website.

There are many other types of threat that exist. The number of attacks on computers is limited only by human ingenuity.

Many operating systems are designed by concentrating on how to manage the computer system and laying on security after that initial design. The result is usually security that does not fit correctly with all computer use. Examples abound. In the early design of Unix system services such as trivial-file-transfer protocol (TFTP) were designed with no authentication in mind, allowing any file to be transferred anywhere. The invention and use of email spam initially resulted from a feature designed by trusting mail transport designers. These designers allowed one message to be sent to a site from anywhere and they allowed it to go to many destinations at once.

14.5 Security on Smartphones

Smartphones provide a difficult environment to make secure. They are single-user devices and require no user authentication to use basic functions. Even more complicated functions (such as installing applications) require authorization but no authentication.[1] However, they run on

[1] Bluetooth usage is an exception to this. The Bluetooth protocols specify a passkey be used to allow Bluetooth functionality between phones.

complex operating systems with many ways to bring data – including executing programs – in and out. Safeguarding these environments is complicated.

Symbian OS is a good example. Users expect Symbian OS smartphones to allow any kind of use without authentication – no logging in or verifying your identity. Yet, as we have found out in this book, an operating system as complicated as Symbian OS is very capable yet also susceptible to viruses, worms, and other malicious programs. Versions of Symbian OS prior to v9 offered a *gatekeeper* type of security: the system asked the user for permission for every installed application. The thinking in this design was that only user-installed applications could cause system havoc and an informed user would know what programs he intended to install and what programs were malicious. The user is trusted to use them wisely.

This gatekeeper design has a lot of merit. For example, a new smartphone with no user-installed applications would be a system that could run without error. Installing only applications that a user knew were not malicious would logically maintain the security of the system. The problem with this design is that users do not always know the complete ramifications of the software they are installing. There are viruses that masquerade as useful programs, performing useful functions while silently installing malicious code. Normal users are unable to verify the complete trustworthiness of all the software available.

This verification of trust is what prompted a complete redesign of platform security for Symbian OS v9. This version of the operating system keeps the gatekeeper model, but takes the responsibility for verifying software away from the user. Software developers are now responsible for verifying their own software through a process called *signing* and the system verifies the developer's claim. Not all software requires such verification, only those that access certain system functions. When an application requires signing, this is done through a series of steps:

1. The software developer must obtain a *vendor ID* from a certificate authority. These trusted parties are certified by Symbian.
2. When a developer has developed a software package and wants to distribute it, he must submit his package to an independent test house for validation. The developer submits his vendor ID, the software, and a list of ways that the software accesses the system.
3. The test house then verifies that the list of software access types is complete and that no other type of access occurs. If the test house

can make this verification, the software is signed by it. This means that the installation package has a special amount of information that details what it does to a Symbian OS system and that it may actually do that.

4. The installation package is sent back signed to the software developer and may now be distributed to users.

Note that this method depends on how the software accesses system resources. Symbian OS says that in order to access a system resource, a program must have the *capability* to access the resource. This idea of capabilities is built into the kernel of Symbian OS. When a process is created, part of its PCB records the capabilities granted to the process. Should the process try to perform an access that was not listed in these capabilities, the access would be denied by the kernel and a program error would result.

The result of this seemingly elaborate process to distribute signed applications is a trust system in which an automated gatekeeper built into Symbian OS can verify software to be installed. The installation process checks the signage of the installation package. If the signing of the package is valid, the capabilities granted to the software are recorded and these are the capabilities granted to the application by the kernel when it executes. The diagram in Figure 14.1 depicts the trust relationships in Symbian OS v9.

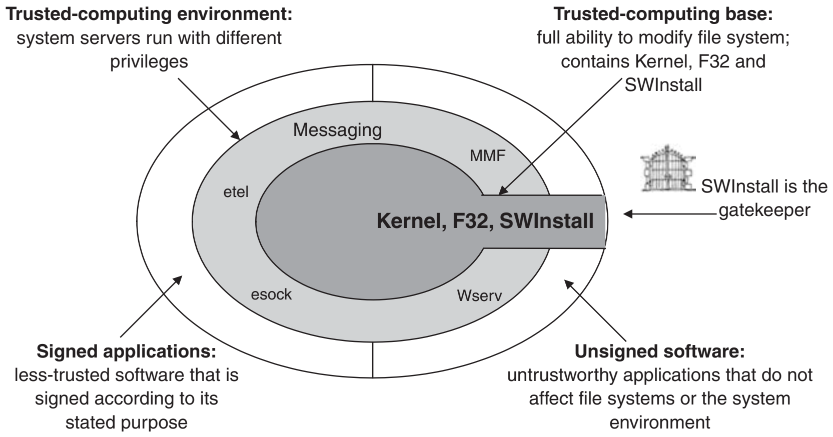

Trusted-computing environment:
system servers run with different privileges
Trusted-computing base:
full ability to modify file system; contains Kernel, F32 and SWInstall
Messaging
MMF
etel
Kernel, F32, SWInstall
esock
Wserv
SWInstall is the gatekeeper
Signed applications:
less-trusted software that is signed according to its stated purpose
Unsigned software:
untrustworthy applications that do not affect file systems or the system environment

Figure 14.1 The trust relationships in Symbian OS v9

Note here that there are several levels of trust built into the system. There are some applications that do not access system resources at all, and therefore do not require signing. An example of this might be an application that only displays something on the screen. These applications are not trusted, but they do not need to be. The next level of trust is made up of user-level signed applications. These signed applications are only granted the capabilities they need. The third level of trust is made up of system servers. Like user-level applications, these servers may only need certain capabilities to perform their duties. In a microkernel architecture such as Symbian OS, these servers run at the user-level and are trusted like user-level applications. Finally, there is a class of programs that requires full trust of the system. This set of programs has the full ability to change the system and is made up of kernel code.

There are several aspects to this system that might seem questionable. For example, is this elaborate process really necessary (especially when it costs money to do)? The answer is yes: the Symbian Signed system replaces users as the verifier of software integrity and real verification is done. This process might seem to make development difficult: does each test on real hardware require a new signed installation package? To answer this, Symbian OS recognizes the need for developer certificates. A developer must get a special signed digital certificate that is time limited (usually for six months) and specific to a particular smartphone. The developer can then build his own installation packages with the digital certificate.

In addition to this gatekeeping function in Symbian OS v9, Symbian OS also employs something called *data caging*, which organizes data into certain directories. Executable code only exists in one directory, for example, that is writable only by the software installation application. In addition, data written by applications can only be written into one directory, which is private and inaccessible from other programs.

14.6 Summary

This chapter has provided an overview of operating system security. We began by introducing general security concepts. We then discussed how authorization and authentication are used to ensure secure system access. We then outlined some of the threats that can jeopardize operating system access. We concluded by taking a look at the security of Symbian OS.

Exercises

1. Passwords can be problematic. Devise a few schemes that choose good passwords but allow them to be remembered.
2. Files have characteristics that can be used to see if they have been tampered with. Describe which characteristics of a file could reveal corruption.
3. One method used for system security on mobile phones is a passworded screen-saver application. A screen saver locks the screen after a period of idle time and any use of the phone makes the application ask for a password before allowing use. Discuss this as a method of security. How much security does this provide? Is it good enough for authentication?
4. Suppose a program advertised itself as installing a set of cool ringtones on your mobile phone. You download this program and, when Symbian OS asks you, you allow installation. You might get ringtones when the program is run, but you also get a process that waits for two weeks and then deletes all the files on your phone. How does Symbian OS v9 catch this type of Trojan horse?
5. In 1988, Robert Morris unleashed an infamous attack on the Internet with a worm of his design. It brought down thousands of computers and he eventually got a sentence of three years of probation, 400 hours of community service, a fine of $10 050 and the cost of his probationary supervision. Make an argument for or against this judgment.
6. What steps should an administrator of a computer hooked to the Internet take to secure his system?
7. What steps should a Symbian OS smartphone user take to secure her smartphone system?

15
Virtual Machines

Many science-fiction stories have a ‘virtual reality’ premise. A person or a community lives and works in an environment they believe is their whole world. One day, they stumble upon evidence that their world is not what they think it is. In fact, they discover that their ‘world’ is really just a contained environment within a larger world, usually much different from their own.

This is the idea behind a virtual machine. The applications hosted by the machine assume that it is a real computer, with operational and functional components that a real computer has. In reality, it is probably an emulated computer, with components also emulated by software or connected through a larger operating system to hardware. There are several reasons to use a virtual machine and the implementation of a virtual machine poses certain challenges to operating system design (both for the virtual machine itself and the host operating system).

This chapter discusses these issues in detail. This topic is a great way to review and apply the principles from this book. We discuss the basic concepts of virtual machines, including the need for them and the challenges they represent. We also pay specific attention to implementation of the Java virtual machine on Symbian OS.

15.1 Basic Concepts

We have discussed operating systems as layers or interfaces that provide programs a way to interact with hardware. Through various models and implementations, operating systems provide an environment for multiple programs to run at once and to access hardware in an organized and shared manner. The layers in the structure that is built by an operating system look like those in Figure 15.1.

Processes access the hardware by making system calls, which are requests that are serviced by the kernel and responded to as the kernel interacts with the computer's hardware. The kernel takes active steps that coordinate hardware use: scheduling the shared CPU, for example, or virtual memory techniques that allow memory to be shared.

What if programs carried this operating structure further to allow other programs to act as an operating system and execute other programs? For example, what if an operating system ran inside another operating system? This would mean one of the processes executed by an operating system would actually be another operating system that also ran programs. This idea is called a *virtual machine* and represents another layer to go through in accessing the computer's hardware. Figure 15.2 shows this type of environment.

In order for a program on a virtual machine to access hardware, there are now multiple layers to go through, each of which is implementing its own access mechanisms and its own view of hardware.

As you can imagine, discussing the same concepts for operating-systems-within-operating-systems can get quite confusing. We call the bottom kernel level the *host* operating system and the hosted virtual machine the *virtual* operating system.

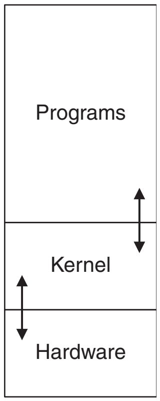


Figure 15.1 Relationships between hardware, operating system and programs

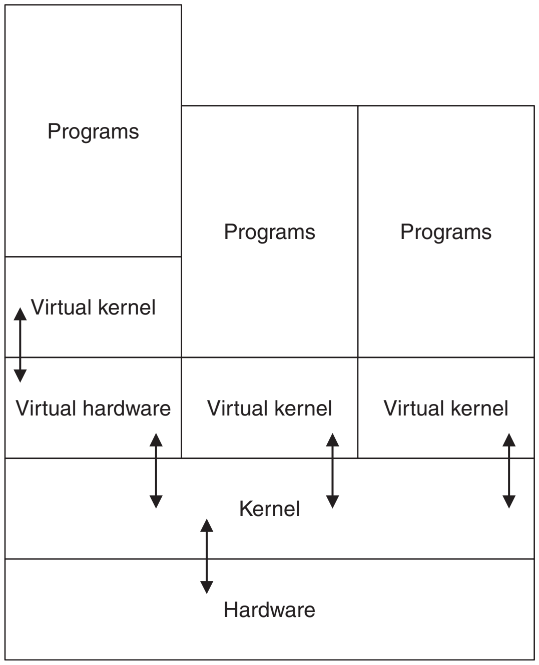


Figure 15.2 Relationships between hardware, operating system and virtual machines

The Need for Virtual Machines

There are several reasons why virtual machines are handy devices to use. Most of these reasons center on a virtual machine as a contained environment that controls access to the host operating system and hardware.

Sheltering system resources from accidental abuse is a prime advantage of virtual machines. A virtual machine is a contained environment, accessing the host operating system through fixed, constrained methods. Each virtual machine is isolated from all the others by the host operating system's memory management and protection mechanisms. A virtual machine is a great place to run untrusted applications – ones that have the potential to ruin an operating environment.

A protected and isolated environment is an excellent arena for development of new system software – particularly new operating systems. Operating system designers face a recursive problem: operating systems need hardware for design and testing, but hardware needs an operating system to allow design and testing. This catch-22 situation is remedied nicely by a virtual machine environment. Each operating system design can be done in its own virtual machine environment. The designer can control the parameters of the environment and the testing of the operating system. The host operating system is untouched.

This is especially useful when the hardware is also being designed and tested. New hardware requires a new layer – one that provides the operating system an emulated hardware environment. A virtual machine allows the hardware to be tested with real operating system software without expensive building and rebuilding of hardware. Note here that 'hardware' is a misnomer; the hardware *design* is tested by the software-emulated environment in a virtual machine.

Virtual machines also allow for software to be developed more easily for less-accessible systems. Symbian OS is a good example. The development environment for Symbian OS software has always provided an emulation environment: Symbian OS running on emulated hardware contained in a Microsoft Windows operating system. This virtual machine allows software to be developed using tools that run in Microsoft Windows while the software itself can be tested and executed by the Symbian OS emulator. Since Symbian smartphones are not designed to allow general-purpose software development, the virtual machine approach is the best way to facilitate new software.

Virtual machines are used to bridge system incompatibilities. Consider this scenario. For various reasons, you must upgrade your computer to Microsoft Windows XP but you have an essential software package that only runs in Microsoft Windows 2000. Rather than buying a new computer just for this software package or finding a new version of the package that runs in Windows XP, you could install a Windows 2000 on a virtual machine and run this essential software there. Virtual machine software from companies such as VMware and Moka5 provide this kind of virtual platform.

Virtual machines have recently been used as execution platforms. Increasingly, we see more interpreted programming languages which are designed to produce executable code that runs on a virtual machine that makes the code platform-independent. That is, when you compile a program in a language targeted to a virtual machine, that program executes on any computer that can run the virtual machine. The original program does not need to be recompiled. (The illusive 'write once, run everywhere' goal is achieved successfully to a certain extent, with realistic restrictions.)

Virtual Environments Are Everywhere

Virtual machines exist throughout an operating system. In fact, it is usually the operating system's goal to create a virtual machine for every program. The environment a process runs in is designed to make that process believe it is running alone on the computer's CPU and has all memory at its disposal – a virtual environment. Communication stacks employ abstraction to make virtual environments – the 'machine' at a certain layer in the stack is supposed to operate as if it is communicating with the corresponding 'machine' at the same layer on another computer. Abstraction is used in many areas of an operating system – from messaging to I/O – to make virtual environments out of the raw hardware interface.

Implementation

Virtual machines can be implemented in two ways. They can be designed to run on the same hardware as the host operating system or they can be designed to run on an entirely different hardware architecture. Both of these cases are illustrated in Figure 15.2.

If a virtual machine requires a hardware architecture that is different from the host architecture, then that hardware layer is usually implemented first. This is a difficult task, because the hardware must be emulated exactly, especially when I/O and hardware features (e.g., timers) are important. However, it is also a very liberating task. It frees designers to build an architecture of their own design, with unique capabilities that might be unavailable if the architecture were actually built. For example, the Java virtual machine features interesting architectural aspects – such as a memory pool for constants and typed registers – that have proven quite expensive to build yet work well in software.

The software layer represents the operating system on the virtual machine. This is generally software compiled for the hardware machine being emulated. Sometimes, depending on the way the virtual machine is implemented, the software needs to be compiled in a special manner. If it does, it generally cannot take advantage of all features of hardware – real execution and testing needs to be done on real hardware.

Symbian OS provides a good example of this. In the emulated, virtual-machine environment, it provides for software development: the system libraries are delivered as Windows DLL-format files and the software under development must be compiled specifically for the emulator environment. This is because the emulator does not execute a specific hardware's instructions. Since the hardware cannot be emulated exactly, to test various parts of it (e.g., timers and real-time components) software must be run on the actual smartphone target.

Sometimes there is no clear difference between the operating system layer and the hardware layer. For example, on the Java virtual machine, there is no operating system – just emulated hardware. Sometimes, because certain operating systems are designed for many different hardware platforms, only the operating system layer is present in the virtual machine, assuming that it will run on the native host architecture.

A virtual machine must emulate both user and privileged mode, but runs on the host machine in user mode. This means that system calls on the virtual machine must be translated into appropriate system calls on the host machine. User mode operation on a virtual machine can execute on the virtual machine. Any access to privileged hardware – almost any system resource – must go through the host system.

The Challenges of Virtual Machines

Virtual machines are interesting and effective in concept, but are difficult to implement in reality. The biggest challenge to implementation is the exact duplication required for emulation. The goal for implementation is to use exactly the same code for the operating system layer as in the target platform, but this is not always possible. If the hardware layer is needed, exact duplication is again required, but very difficult to render.

A good example is the Symbian OS emulator. Symbian OS is designed to run on multiple platforms. However, the Windows emulator for Symbian OS required a special implementation with special restrictions (Chapter 7 discussed the special memory model developed for Symbian OS when it ran on the emulator). For many programs, the emulated virtual machine looks identical to the target hardware. However, certain programs can only be tested on the target. Programs that work specifically with hardware, for instance, cannot be tested on the emulator. Also, programs that rely on processes cannot run on the emulator because the operating environment is one large process. Since Symbian OS v9.1, these restrictions have mostly been eased; the design of Symbian OS has since incorporated Windows as a target platform and the design of the emulator more closely resembles hardware.

Virtual machines also suffer in terms of performance. Virtual machines that are not optimized are basically interpreters: they translate actions by programs they are executing into actions that can be executed by the host machine. In a virtual-machine implementation with multiple layers, each layer is an interpreter. The Java Virtual Machine (JVM) is an example of this: Java is based on bytecode execution, but there is no computer system the JVM runs on that executes JVM instructions ('bytecode') directly. So each bytecode instruction must go through an interpretation, where actual hardware instructions that implement each bytecode instruction are eventually executed on that bytecode instruction's behalf. Each passage through an implementation layer represents a performance degradation.

One of the most complex challenges to virtual-machine implementation is the access to the host machine's system resources. A major issue that virtual-machine implementations must deal with is access to disk-file systems. Consider a situation where the host system is running Microsoft Windows with an NTFS file system but the virtual machine runs a Unix operating system with a UFS file system. Clearly, these are incompatible. Consider again a situation where the virtual machine requires access to a peripheral, say a printer, that is being coordinated by the host operating system. These situations all require special handling. Usually, virtual devices are connected with physical devices through special interfaces that merge the virtual machine with the host machine's device mechanisms. Disks, however, are a special case. These are usually handled by creating virtual disks that operate inside file space on physical disks. A Linux virtual machine might have a 10 GB virtual disk that is connected to a 10 GB file on a physical disk. The Symbian OS emulator can provide drives that are actually files on the host's system, as well as MMC emulation and an emulation of ROM.

15.2 The Java Virtual Machine and Symbian OS

The implementation of the JVM on Symbian OS provides an interesting case study in the implementation of virtual machines.

Before looking at implementation issues, we must clarify which JVM is implemented for the Symbian OS platform. Symbian OS has supported Java for quite a long time – since the days when Symbian OS was actually EPOC, implemented on handheld devices, not smartphones. Early Symbian OS versions supported two types of JVM: PersonalJava and Java Platform, Micro Edition (Java ME). PersonalJava was an early attempt to pare down the Standard Edition of Java to fit into smaller platforms. It has largely been abandoned in favor of Java ME implementations. Java ME is a general specification of how Java fits on small platforms. Java ME is subdivided into configurations, profiles and optional packages. For their implementations, Symbian OS designers implemented the Mobile Information Device Profile (MIDP) with the Connected Limited Device Configuration (CLDC). The CLDC defines the base set of application-programming interfaces and the Java Virtual Machine for resource-constrained devices such as mobile phones, pagers, and mainstream personal digital assistants. When put together with MIDP, it provides a Java platform for developing applications to run on devices with limited memory, processing power and graphical capabilities.

As we stated, the JVM is an interesting virtual machine in that it has no operating system. Its sole existence is to implement an operating environment for program execution. Java applications are implemented as a set of classes that are dynamically loaded as needed by the virtual machine. Java classes, in turn, are compiled to a sequence of bytecode', which is Java's terminology for assembly language. Java bytecode is designed to have short instructions that could be loaded from anywhere: from a disk drive or over a network.

The Java Virtual Machine is an abstract computer that provides an insulation layer between the Java program and the underlying hardware platform and operating system.

The JVM can be divided into several basic parts that are implemented in software to emulate a virtual underlying hardware layer: bytecode execution stack, a small number of registers, the object heap and the method area which stores streams of bytecode. The JVM also provides a mechanism of native access to the host operating system.

The JVM reads sequences of bytecode, which is a sequence of assembly instructions for the JVM. Each instruction consists of an opcode that instructs the JVM on what needs to be done and the following instruction operands provide the required information for the completion of the command. As well as having in-built support for several primitive types, the JVM bytecode set includes instructions that operate on operands as objects in order to invoke instance and static-class methods.

The JVM has its own architecture. While this architecture is a virtual architecture implemented in software, it nonetheless influences the instruction set design of JVM bytecode. The JVM is based on a stack design, which means that all data for computation must be pushed onto the stack, so it can be used in calculation, and that the JVM needs no registers for storing data. The combination of the emulated hardware architecture, bytecode design and class-file format make a unique design that is at a much higher level compared to real hardware. It has features that enable fast interpretation of instructions: class-file-constant pools that allow loading and storing of constants by using short assembly instructions, a typed data stack and registers that hold the program counter and manage the stack.

In addition to emulating virtual hardware, the JVM provides a mechanism of native access to the host operating system. This means that implementation areas and various capabilities that are out of the scope of the virtual machine role as a bytecode execution engine can be implemented using the operating system native APIs. Symbian OS takes advantage of this mechanism by providing its Java ME implementation access to its rich native APIs. For example, the `javax.microedition.lcdui.TextField` class is implemented as a native Symbian OS widget and so a Java program benefits from the underlying usage of the native `AVKON` widget implementation on a Nokia phone or the UIQ widget implementation on a Sony Ericsson phone. The two run the same Java program, based on the same interfaces and JVM, but with a different look derived from the manufacturer-dependent native implementation of the user interface. Access to system resources is granted through standard Java APIs such as the Bluetooth API and the Messaging API.

15.3 Summary

This chapter has given a broad overview of the concept of virtual machines and how virtual machines can be used by operating systems. We began by defining virtual machines and outlining their advantages and challenges. We concluded the chapter by giving an example of how the Java virtual machine is implemented on Symbian OS.

Exercises

1. Describe the sequence of actions or calls that must take place for a program in a virtual machine on virtual hardware to write to the virtual machine's disk drive.
2. Consider memory management on a virtual machine. Would the host or the virtual machine map the virtual machine's logical address requests to physical addresses? Would the host or the virtual machine do paging if an application on the virtual machine referenced a page that was not in memory?

3. What are the consequences of a poor emulation? Is it sufficient to simply state that a virtual machine ‘approximates’ an operating environment?

4. The JVM has an architecture that allows for very short bytecode instructions. Why is this advantageous for Java?

5. Does the design of Java have any implications for its performance on Symbian OS?

Appendix A

Web Resources

http://developer.symbian.com/main

http://en.wikipedia.org/wiki/List_of_operating_systems

www.imc.org

www.kernelthread.com/mac/oshistory

www.knoppix.org

www.levenez.com/unix

www.microsoft.com/windows/WinHistoryIntro.mspx

www.opengroup.org

www.pushl.com/taskspy

www.renegade-uiq.com

www.ubuntu.org

References

Campbell, J. (1990) ‘The Spirit of TIFF Class F’, Cygnet Technologies, 2560 9th., Suite 220, Berkeley, CA USA.

Lamport, L. (1987) ‘A fast mutual exclusion algorithm’, *ACM Transactions on Computer Systems*, 5:1, 1–11.

Leung, J.Y.T. and Whitehead, J. (1982) ‘On the complexity of fixed-priority scheduling of periodic, real-time tasks’, *Performance Evaluation (Netherlands)* 2:4 (December 1982), 237–50.

Liu, C.L. and Layland, J.W. (1973) ‘Scheduling algorithms for multi-programming in a hard real-time environment’, *Journal of the ACM* 20:1 (January 1973), 46–61.

Siberschatz, A., Galvin, P. and Gagne, G. (2003) *Operating System Concepts*, John Wiley & Sons.

Index