CHAPTER 16

THE USER VIEW OF OPERATING SYSTEMS

© The New Yorker Collection. 1983 Chast from cartoonbank.com
16.0 INTRODUCTION

In Chapter 15, we introduced you to two different views of the role of the operating system as part of the overall computer architecture. Specifically, we looked at the operating system both as a means of delivering services to the user and as a way of controlling and operating the system facilities. In this chapter, we take a closer look at the operating system from the perspective of service to the user.

Much of the material in this chapter is at least superficially very familiar to you. You have worked closely with at least one type of computer system and quite possibly with more than one. You are familiar with some of the tasks, services, and capabilities that are provided for you by the system or systems that you have worked with. You are familiar with the different types of interfaces that you have used to perform those tasks and with the commands and the command structure that are built into the system for you.

In this chapter we are interested in two aspects of the operating system as it pertains to the user. First, we will consider the services that are provided to the user, and second, we will consider the medium for delivery of those services, namely, the type and appearance of the user interface that the system provides. You will see the standard tasks that a user interface is expected to perform, various ways in which those tasks can be implemented, and the advantages and disadvantages of different implementations. You will be able to observe that the services provided are relatively independent of the means used to access them.

As for the interface, we are more interested in the concepts of a user interface than in the specific commands, syntax, appearance, and usage of a particular interface. You will understand that different design approaches to the interface meet different goals and achieve different ends and are often aimed at different classes of users. You will see additional features that are frequently built into an operating system to make the system more “user-friendly”, or more powerful, or more efficient. Some of these represent additional services; many are simply ways to make access to the services easier, or more powerful, or more efficient. You will see that one interface may not be powerful enough for some tasks, while another interface requires too much effort to accomplish the common tasks that are required on a regular basis.

User services are a fundamental purpose for the existence of an operating system, and the user interface is essential to the access of those services. Nonetheless, some systems elect to view the user interface, and even many user services, as outside the realm of the operating system. Instead, these services and the user interface are treated as a shell that itself interfaces the operating system. There are strong arguments for this point of view. It makes possible different shells, each with their own services, capabilities, and work styles. If you don’t like the shell that is provided, you simply exchange it for another. As we noted in Chapter 15, operating systems based on UNIX and its variants are the strongest proponents of this view—operating systems such as Linux are routinely supplied with several different shells offering different capabilities; in many cases, the user can change shells with a single command. The counterargument to this point of view is that building the user interface...
and services into the operating system provides standardization, consistency, and much improved integration of services. Apple Macintosh systems take this approach.

There is a third approach to the user interface, namely to hide the system user interface and to use a Web browser model to serve as the interface for the applications that the user works with. Limited access to system utilities can also be supported through the Web interface.

This chapter takes a detailed look at the issues we’ve just raised. It explains and justifies the different types of user services that are provided with an operating system. It discusses and illustrates and shows the rationale for the various types of user interfaces, and it considers the trade-offs between them. It shows how user services are accomplished with each type of interface. Our primary goal in this chapter is to expand your ability to use your systems effectively and to understand the alternative methods available to you for using the operating system to achieve higher productivity. We hope that the chapter will also provide you with a better understanding of what happens internally within the system when you use your computer.

16.1 PURPOSE OF THE USER INTERFACE

The primary purpose of the user interface is to make the facilities of the computer system accessible to the user by providing the necessary services and commands and means of access to allow the user to get her work done conveniently and efficiently. We emphasize that it is not the intention of the user to interact with the operating system per se. Rather, the operating system exists to help the user to use the computer system productively. In modern operating systems, a secondary purpose has arisen that is almost as important: the operating system provides user interface services to application programs that assure that different programs have user interfaces that operate in the same way. This simplifies use of different applications on the system and reduces the user’s learning curve for new programs. We identify programs that use the operating system to provide similar interfaces as having the same (common) look and feel.

Although the operating system can support a common look and feel across the applications on a particular type of system, there is an important, steady trend toward the use of Web browsers to provide a common look and feel for applications across all types of systems. Because the World Wide Web is familiar to a large range of users, an organization can simplify the training requirements and enhance its productivity by using Web-based application interfaces as the predominant means of communication with its personnel. A Web-based approach is also attractive to the programmers that create the applications, because Web page creation is well understood and reasonably well standardized across different computer platforms. The use of internal corporate Internet-like intranets to provide information resources throughout an organization is an example of this trend, as is the capability of location-independent Web-based access to e-mail. In addition to the processing and display of data, documents, images, audio, and video, there is even a growing use of Web-based productivity tools such as word processors and spread sheet applications.

We note in passing that the Web interface is particularly effective when the “user” is not a person, but another machine. The standard languages shared across the Web, particularly XML and HTML, make it relatively easy to create Web-based interfaces that can coordinate work between systems.
A well-designed interface can enhance the user’s experience of the system and make use of the computer system a pleasure. This will allow the system to provide maximum benefit to its users. Conversely, a system with a poor user interface will be used reluctantly, and its potential value to its users will be diminished. Different classes of users are likely to define the concept of a good interface differently.

The operating system provides a variety of services to the user and to the user’s programs. The user interface provides access to these services using three different approaches. These are:

- A command interface of some type that accepts commands in some form directly from the user interface. Most commonly, the interface is either graphical (GUI) or command line (CLI).
- A command language that accepts and executes organized groups of commands as a form of program. Most command languages include capabilities for branching and looping, prompted user input, and passed arguments. Command languages are also referred to as scripting languages.
- An interface that accepts and performs requests for operating system services directly from the user’s programs (the API).

Modern operating systems provide all three of these capabilities. There are even a number of scripting languages that support portability between different operating systems. The user services provided by an operating system typically include:

- Loading and execution of program files
- Retrieval, storage, and manipulation of files
- User I/O services, in the form of disk commands, printer spooling, and so on
- Security and data integrity protection
- Interuser communication and shared data and programs, on multiuser and networked systems
- Information about the status of the system and its files
- I/O and file services plus other specialized services for user programs

Many systems also provide utilities that can be used in place of programs to manipulate the data within files and programs. These utilities can be used to sort data and to retrieve data selectively from within files. Frequently, utilities can be combined into “programs” using the command programming language to perform powerful and useful tasks. Linux is particularly strong in this regard. The choice of user services provided is dependent on the original focus and goals of the operating system designers.

Finally, modern systems expand on the concept of I/O service to provide libraries of specialized service routines that can be used by programs to generate graphics, control a mouse, create and manipulate user windows, generate and control menus, and perform other sophisticated functions. These make it easy for application programmers to supply a common-look-and-feel interface to their programs.

The difference in skills and interests among the various users on a system affects the design of an operating system user interface in two major areas:

- It affects the choice of services to be provided. For example, powerful programming services may not be needed by the typical user, but may be
extremely useful to a system programmer. Conversely, tools that allow the end
user easier access to the system may actually hinder the system programmer.

■ It affects the design of the actual interface. The sophisticated user may be more
comfortable with a more powerful, but difficult to use, interface. The typical user
does not want to, and should not have to, learn a special and difficult operating
system lingo just to use the computer.

The operating system must ultimately serve both groups of users, but a particular
operating system may be tailored toward one or the other. An operating system that was
designed with the goal of supporting engineers may be difficult for a graphic layout artist
or secretary to use. Conversely, an engineer may not be able to work effectively on a system
that the secretary finds ideal. An alternative is to provide two (or even more) different
interfaces, intended for different user groups. If the command interface is implemented as
a shell independent of the remainder of the operating system, this is easy to do. The normal
user can work with an interface that provides a menu or windowing interface. The more
technically sophisticated user can use the GUI for typical tasks, but bypass the windowing
shell and enter commands directly to a command interface, when necessary or convenient.

16.2 USER FUNCTIONS AND PROGRAM SERVICES

In Section 16.1 we listed seven major groups of user functions and program services that are
provided by most operating systems. Now we’ll consider these functions more specifically.

Program Execution

The most obvious user function is the execution of programs. Most operating systems
also allow the user to specify one or more operands that can be passed to the program as
arguments. The operands might be the name of data files, or they might be parameters that
modify the behavior of the program.

To the typical end user, the smooth loading and execution of programs are nearly the
sole purpose for the operating system. Many operating systems treat program execution
the same as they treat nonresident operating system commands. The name of the program
is treated as a command; loading and execution begin when the command is typed or,
equivalently for a windowing system, when the mouse double-clicks on the graphical icon.
Alternatively, the user may click on a data file icon. The program associated with the data
file is executed with the data file as an operand.1

Since the operating system treats application and user programs in the same way as it
treats nonresident commands, it is conveniently impossible to tell the difference. Most of
the programs that you have used are not part of the operating system, but since they are
initiated the same way, you cannot tell the difference. This provides a consistency that is
convenient to the user. Microsoft Excel, Quicken, Firefox, and Adobe Acrobat, to name

---

1If you are not used to graphical system terminology, an icon is a small graphical representation of a program
or data file. “Double-clicking” involves clicking a button on the mouse twice in rapid succession.
just a few, are all independent, nonoperating system programs that share this common behavior, look, and feel.

Application programs perform their operations on a user-specified data file. Spreadsheet programs, for example, require a file of spreadsheet data; word processors use a file of formatted text. The command interface provides a method for specifying the data file to be used when the program is executed. In a command line system, the data file may be specified as an operand typed on the same line with the command. In a graphical system, data files may be associated with a particular application. This association is set up automatically by the operating system when the data file is created, or it may be manually established by the user. Once the association is set up, the application can be initiated automatically by selecting the data file. On most computers, for example, each data file has an icon associated with it; the application is launched with the particular data file by double-clicking the mouse on the data file icon. In Microsoft Windows, the same result can also be achieved by double-clicking the mouse on the name of the data file within the Windows Explorer.

To expedite the execution of programs, the system also provides a means for moving around the system between different peripheral devices and between different storage areas on those devices. Most operating systems embed these operations in a logical device and directory structure and provide commands for moving around the structure. In a command line system, commands provide the ability to attach to different devices and to change one’s attachment from one directory to another with a command such as `cd`, for change directory.

Graphical interfaces provide file folders as the equivalent to achieve the same purpose. Although you are probably most familiar with running your programs interactively, most operating systems also allow a program or sequence of programs to be run noninteractively, in a batch mode. The operating system allows the user to specify conditions under which the execution is to take place, for example, the priority of the programs, the preferred time when they should be executed, the stored location of the programs and the particular data files that are to be used. For example, the system can be told to perform a partial backup at a given time every night, with a full backup on Sundays.

**File Commands**

The second, and most familiar, category of user services are commands for the storage, retrieval, organization, and manipulation of files.

From the perspective of the user, the file management system is what “makes it all possible”. Four factors account for the importance of the file management system to the user:

- The ability to treat data and programs by logical file name, without regard to the physical characteristics of the file or its physical storage location
- The ability of the file management system to handle the physical manipulation of the files and to translate between logical and physical representations
- The ability to issue commands to the operating system that store, manipulate, and retrieve files and parts of files
- The ability to construct an effective file organization utilizing directories or file folders to organize one’s files in a meaningful way
The file management system is so important that we have devoted the entirety of Chapter 17 to it. Of interest to us here, as users, is the fact that most of the user commands in the operating system are directly used to manipulate files and file data. This is evident if you consider the commands that you use in your regular work with the computer. The brief partial list of Windows and UNIX/Linux CLI commands in Figure 16.1 typifies the commands that you would probably consider to be most important to you. Other operating systems provide essentially identical commands, although the commands might appear quite different, depending on the user interface. Graphical user interfaces provide equivalent operations for each of these commands. On a Macintosh computer, for example, you move a file by dragging its icon with the mouse from its current location to the desired location. You create a new directory by moving an empty file folder into the window that represents the desired attachment point.

Many additional features built into the command structure reflect the importance of a flexible file structure to the user. These include

- The ability to change from one device and one directory or subdirectory to another without otherwise modifying the file
- The ability to redirect input and output to different devices and files from their usual locations

### Disk and Other I/O Device Commands

In addition to the file commands, the operating system provides commands for direct operation on various I/O devices. There are commands for formatting and checking disks, for copying entire disks, for providing output directly to the screen or to a printer, and for other useful I/O operations. Some systems also require the mounting and unmounting of devices. This effectively attaches and detaches the directory structure of a device to the already existent directory structure as a means of adding devices to the system.

Most operating systems also provide a queuing system for spooling output to a printer. The printer is generally much slower than other computer facilities. The spooler works by copying the output text into a buffer in memory and then printing as a separate task. This allows programs to proceed as though printing had already taken place.
Security and Data Integrity Protection

Every operating system provides security protection for files. Generally, individual provisions are made to protect files from being read, written to, or executed. Some operating systems also provide protection from deletion. A few operating systems provide additional security, requiring a correct password to be typed before a disk can be used in the system, or a keyboard unlocked.

Systems with network multiple user access, of course, must provide much more protection. The system as a whole must be protected from unauthorized access. Files must be protected, so that the owner of the file can control who has access to the file. The owner can also protect the file from himself or herself by specifying read-only or execute-only access.

Many operating systems also allow controlled access from other users. UNIX provides three levels of security for each file, consisting of read, write, and execute access privileges for the owner, for associates of the owner (known as groups), and for anybody with access to the system. Windows offers share privileges to control file and device access between users on a network. Many large systems also provide access control lists, or ACLs (pronounced ack-ulls), that allow the system administrator to control access to program and data files on an individual user/file pair basis. In addition to file protection, every operating system with multiple user access provides a login procedure to limit access on the system to authorized users. Commands exist to allow the user to modify the access rights on a file and to change the user’s password. Networks also require the use of a login procedure that limits the user to the appropriate computers and facilities.

Interuser Communication and Data Sharing Operations

Modern systems generally provide means for multiple users to share data files and programs. Most systems also provide a means to pass data between programs and to communicate between users. Application programs like Lotus Notes, instant messaging, and videoconferencing can extend this capability of users to work collaboratively in a powerful way to both small and large networks of computer systems.

The simplest form of program sharing on a single system is to place the shared programs in a common memory area where all users can reach them. This is done for editors, compilers, general utilities, and other system software that is not part of the operating system. Many operating systems even allow several different levels of control over such shared programs. For example, the Little Man Computer simulator used at Bentley College is accessible to all computer majors, but other users must have permission to access it.

Data file sharing is an important resource when using databases, because it makes it possible for multiple users to access the same data in a way that the integrity of the data is protected. Needless to say, the system must provide tight security to limit data file access to those who should have it. An additional use for data file sharing is when two or more users work as a group on a document. All these users have access to the document. Some can only read it, while others can modify it. Some systems provide means for attaching notes to the document for other users to see. As with program sharing, it is possible to set several levels of data file sharing.
Modern networks routinely provide operating system message passing services in the form of e-mail and newsgroup support, file transfer (ftp), simple terminal facilities for connecting to a different system (telnet or ssh), Web support (http), instant messaging, and audio and video conferencing. Some systems also provide an internal messaging service for rapid communication between users logged on to the system, either directly or through a network.

Operating systems also provide internal services that allow programs to communicate with each other during execution. Modern systems go one step farther and extend this concept to allow the user to control interprogram communication as a means of extending the capabilities of individual programs.

The simplest example of this is the pipe command available in many systems for taking the output from one program and using it as the input to another. More sophisticated techniques allow a user to link two programs together so that a spreadsheet, for example, can appear in a word processing document. Double-clicking the mouse on the spreadsheet actually launches the spreadsheet program from within the word processor program, so that the spreadsheet can be modified. The most sophisticated systems actually allow the user to work with different application programs transparently, that is, without even being aware that an operation has launched another program. For example, modification of the spreadsheet would take place right within the word processing document; the user would not even be aware that a spreadsheet application program was being executed at the time.

This approach relies heavily on the operating system to support communication between different programs in a fashion invisible to the user. The user is not even aware of which application is executing at any given instant. This technique views the document as the center of focus, instead of the applications that are being executed.

**System Status Information**

As previously seen in Chapter 15, most operating systems provide status information that can be useful to a user who knows how to interpret the data. This data is usually more important to the people who operate and maintain the computer system, but sometimes status information can be useful to programmers and users in optimizing their work.

Commands commonly exist to determine the amount of available disk space, the amount of available memory, the number of users on the system and who they are, the percentage of time that the CPU and I/O channels are busy, and many other statistics.

As an example of how this data might be useful, consider an application program that does the billing for a large electric utility company. Such a program might require many hours of CPU time to complete a month’s billing. A small change in the program could cut the CPU time by a significant percentage. The measurement of CPU time used is essential to assess the improvement. This data might also be important to a user who is being billed for her time.

The names of other users on the system might be used to establish a phone-type conference or to send a quick message to another user.

Many systems provide a logging facility that maintains a file of all keyboard and screen I/O. With a log file the user can determine at a later time what commands were typed and what modifications made to programs and data.
The examples given here are only a few of the possible uses for system status and information. There are many other possibilities. Status information can be particularly important to personal computer users who must maintain their own systems. For example, status information allows the user to ascertain the condition of a disk: to determine the number of bad blocks on the disk or to analyze and reduce the fragmentation that is present on the disk. For example, Windows provides the `scandisk` and `defrag` commands for this purpose.

**Program Services**

One of the most important user functions provided by the operating system is actually invisible to the user. The operating system provides a variety of services directly to the user’s programs. The most important of these are the I/O and file services. Also important are requests for the use of system resources that a program needs, such as additional memory or larger blocks of time and GUI services that support “common look and feel”.

Typical program services can be used to retrieve a file, to save a file, or to locate the individual blocks of a file. The program services can read one or a few particular blocks of a file into a designated user area of memory. Most program service facilities can keep track of several files at once.

Typical program services also allow an application program to bypass the file system under certain conditions and to perform I/O directly to disk and other I/O devices. For example, the program could request a service that retrieves a particular block on the disk. This type of service might be important to a program that attempts to repair damaged data files, for example. Most operating systems control these services in such a way as to prevent damage to the system or to other users’ files and resources. As another example, output to a screen is performed directly to I/O, rather than through the file system, on many computer systems. These I/O services allow programs to write text and graphics to the screen without compromising the system.

Services to programs also include services that are less apparent to the user, but important nonetheless. Interprocess message passing allows programs to synchronize their actions to protect data integrity, to share and exchange data, and even to distribute program processing among different machines on a network. DCOM (Distributed Component Object Model), its extension .NET, and CORBA (Common Object Request Broker Architecture) are two standards that allow programs to locate and share objects, either on a system or across a network. Remote Procedure Call (RPC) allows a program to call a procedure at a remote location across a network.

.NET and other equivalent operating system tools also ease a programmer’s task at creating new programs by providing commonly required program objects and allow computers on a network to share the processing load. Particularly in modern systems, where elegant user interfaces and graphics are the norm rather than the exception, the operating system provides a library of powerful service routines for implementing user interfaces and graphics for individual programs. On most modern workstations and personal computers, it is only necessary for a program to call a series of these routines to maintain control of windows, drop-down menus, dialog boxes, and mouse events. The routines also include powerful system calls for drawing complex graphic shapes on the screen.
To use the program service routines, the user’s program makes requests to the operating system through the **application programming interface (API)**. In most systems, the API consists of a library of service functions that may be called by a program. The call and required parameters are passed to the selected service function, using whatever method is implemented for the given machine. Most commonly, a simple call is used, with a stack used to pass the parameters. The service function is responsible for communication with appropriate routines within the operating system that perform the requested operation. A software interrupt or service request instruction is used for this purpose. The service routine returns required results, if any, to the calling program. On some systems, the calling program uses a software interrupt directly to access the API. Windows win32, the standard API on every Microsoft Windows system, provides hundreds of service functions to programs.

The use of operating system program services provides convenience for a program developer. More important, providing a single gateway through which all I/O must pass assures the integrity of files and other I/O. If each program placed its files on the disk independently, there would be no way to assure that the files didn’t destroy each other. The integrity of the system is even more important if the system is multiprogramming and/or multiuser. As an example of what could go wrong, consider a printer being addressed directly by two different programs. The output would be a garbled mix of the output to be printed from the two programs, obviously useless to anyone. Early, single-user systems allowed programs to bypass the operating system and do their own I/O. Doing so increases the risk that data may be destroyed. Furthermore, bypassing the I/O services of the operating system may make the program incompatible with other, similar (such as clone) computers that may use different internal addresses for the I/O device functions. Finally, the I/O services provided are convenient. They make it possible to simplify the task of writing programs by providing the difficult I/O and file functions automatically.

A properly designed multiprogramming operating system will not allow programs to perform their own I/O; the operating system services must be used. It is the role of the system services to properly queue I/O requests and to perform them in such a way as to protect the data. No current machine allows a user program to bypass operating system services.

### 16.3 Types of User Interface

There are two types of user interface in common use. One of these is the **command line interface (CLI)**, which is seen on a wide variety of operating systems, including the command prompt in Windows. Although this is historically the most common interface, the **graphical user interface (GUI)** has supplanted the CLI for most routine day-to-day use. GUls are seen on the Apple Macintosh, Windows-based PCs, Sun workstations, most Linux desktop systems, and a variety of other devices such as PDAs and mobile phones.

---

2. The concept of an API also applies to applications that allow other applications to “piggyback” or “plug in” to their services. For example, Web browsers provide API services to their plug-ins, as do several Google tools.

3. Many Linux servers have only a CLI, no GUI, because the GUI consumes so many computer resources that might be better used to provide a higher level of client services.
Although Web browsers are not actually part of the operating system, there are many computer people who believe that a Web browser (together with its tools) is viable as an adjunct or alternative to the standard GUI in many circumstances. It is possible to use the Web browser to perform many common tasks, such as file management. (As an experiment, try using file:///C: as a URL on any Web browser in Windows and notice that you can drag a file to the desktop, for example. Or double-click on a file icon in the browser to launch it.) There is even a Linux shell that promotes the use of a customized Web browser for most operations.

As already noted, the type of interface seen by the user depends on the focus of the operating system. A batch system requires a different interface than a system primarily intended for interactive use. A system designed primarily for inexperienced end users will differ from one designed for sophisticated technical users. Today, most users are reasonably comfortable at a computer screen, the purely batch system is in declining use, and the primary user interface with a computer is interactive, using a keyboard, mouse, and video display. Furthermore, the graphical user interface and Web browser are rapidly becoming the predominant user interfaces for most work.

The Command Line Interface

The command line interface is the simplest form of user-interactive interface. The operating system command shell provides a prompt; in response, the user types textual commands into the keyboard. The command line is read serially, character by character, into a keyboard buffer, where it is interpreted by the command interpreter and executed. Commands are entered and executed one line at a time, although most interpreters provide a means for extending a command onto multiple lines. The command line interpreters for most operating systems use a standard format for their commands. The command itself is followed by operands appropriate to the particular command, as shown:

```
command operand1 operand2...
```

The operands are used to specify parameters that define the meaning of the command more precisely: the name of a particular file to be used with the command, a particular format for listing data, or a detail about how the command is to be performed. Many times, some or all of the operands are optional; this simply means that a default condition is to be used if the operand is not specified.

In some circumstances, the command itself may be preceded by a logical path name that specifies the particular device or file location where the command is to be found. However, most operating systems maintain an internal list where most commands can be found. This list is often referred to as path variable.

As an example, the Linux command

```
ls -lF pathparta/pathpartb
```

consists of the command ls and the two operands -lF and pathparta/pathpartb. This command requests a directory listing from the subdirectory with path name pathparta/pathpartb. Without the optional operand pathparta/pathpartb, the command lists the current directory (or file folder, if you prefer), wherever that happens to be. An additional operand might redirect the output to a file or printer, for example, instead of to
the screen. The Linux command to store the directory list in putfilea (presumably for later printing) would look like this:

```bash
ls -lF pathparta/pathpartb >putfilea
```

The equivalent command at a Windows command prompt would look like this:

```cmd
DIR PATHPARTA\PATHPARTB>PUTFILEA
```

In each of these cases, many other optional operands are possible; these operands would be used to modify the facts listed and the format of the directory listing. The additional operand `-F` in the Linux command tells the system to list the directory in a specific "long" format, one file to a line (the "l"), with subdirectories indicated (the "F").

Operands are either **keyword** or **positional**. In some systems, the operands may be both. Positional operands require the operand to be located in a particular position within the line. For example, in the Windows command

```
COPY SOURCE-FILE DESTINATION
```

the first operand, `SOURCE-FILE`, positionally specifies the path name of the file to be copied. The second operand, `DESTINATION`, which is optional, specifies either a new name for the file or the path name of a directory to which the file is to be copied. If the second operand is absent, the directory to which the user is currently “attached” is used. The importance of the position of these positional operands is obvious: some older operating systems specified the destination operand first; reversing the position could destroy the file to be copied.

Keyword operands are identified by the use of a particular keyword. In many systems, the keyword is accompanied by a modifier symbol that identifies the operand as a keyword as opposed to a file name. The keyword identifies the purpose of the operand. Keyword operands are frequently used as optional operands, sometimes with a particular positional value attached. In some systems, keyword operands and modifiers can be placed anywhere after the command without affecting the positions of positional operands. In other systems, the keyword operands, if present, must be placed in a particular position. The slash mark `(/)` in Windows and hyphen `(-)` in Linux are examples of modifier symbols. Keyword operands are sometimes known as **switches** or **modifiers**.

The Windows command

```
MODE COM1 BAUD=2400 PARITY=N DATABITS=8
```

uses the positional operand `COM1` to identify a particular communications port or other device. `BAUD`, `PARITY`, and `DATABITS` are all examples of keyword operands. Each has its own positional operand that selects a particular option, but the order of the keywords is immaterial.

Similarly, the command

```
DIR /P/A:DH PATHNAME
```

uses the `/P` and `/A` switches to specify that files are displayed on a screen one page at a time, and to modify the list of path name directory files that will be displayed. Command line interpreters also include other provisions designed to increase the flexibility of the command. The most important provisions include the ability to redirect input and output, the ability to combine commands using pipes, and the ability to combine commands into
shell scripts or, as they are sometimes incorrectly called, **batch programs**. (You are already
aware of the correct usage of this term. In this book we will always differentiate between
shell scripts and batch programs.) Another important capability is the use of the wild card,
a character symbol or symbols that can substitute for one or more unspecified letters in
an operand. Use of a wild card in a command can make possible a search, or can cause a
command to be repeated with several different arguments.

Although they work somewhat differently, both Linux and Windows use the question
mark symbol (?) to replace a single character and an asterisk (*) to replace a group of 0 or
more characters. Linux has additional wild card possibilities, shown in the Linux case in
Supplementary Chapter 2. For example, the Linux command

```
ls -l boo.*
```

would search the current directory for all files that have the name boo. This command
might result in the following output:

```
-r--r--rwx  1  irv  cisdep  221  May 16  7:02  boo.dat
---x---rwx  1  irv  cisdep  5556  May 20  13:45  boo.exe
-r--rw-r-- 1  irv  cisdep  20  Jun 5  2:02  boo.hoo
```

In the Windows command,

```
COPY ABC* B:
```

all files whose names begin with ABC are copied to the drive B: master directory. In this
case, the wild card is used to expand the command to repeat the copy process for several
different files.

In addition to **wild card** provisions, some operating systems allow the user to back
up and repeat a previous command using the cursor keys on the keyboard. Such systems
usually allow the user to edit the command, as well.

Command line interfaces are well suited to experienced users who are comfortable
with the system and who want the power and flexibility that a CLI offers them. Command
line interfaces are generally the hardest to learn. The range of possibilities and options
that accompany many commands often make it difficult to figure out the particular syntax
that is required for a desired operation. Manuals and online help are particularly useful in
working with command line interfaces. Online help is available for all Linux commands
using the “man commandname” command.

## Batch System Commands

Batch systems use an interface that is similar to the command line interpreter in many
respects, but the purpose is different. Commands specify the location of programs to be
executed and data to be used, using a Job Control Language. Job control commands use a
format similar to that of the command line interpreter:

```
command operand1 operand2...
```

Batch command operands also are either of keyword or positional type. The most
familiar language of this type is IBM zOS/Job Control Language. Batch jobs consisting of
one or more programs are “submitted” to a system for execution and are generally executed
with no human interaction. Since direct human interaction with the batch system is not
possible, all steps must be carefully planned out, including actions to be taken when errors occur. Batch programs are well suited for routine transaction processing applications, such as credit card billing and payroll.

As we noted above, command line interfaces also provide the ability to “batch” commands together to form pseudoprograms, known as shell scripts, or just scripts, that will be executed as a unit. These are not true batch programs, as they are still intended for interactive use. Nonetheless, Windows users sometimes refer to these programs as batch programs or, more commonly, as bat files. Most command line interfaces provide additional commands that are intended especially for the creation of powerful scripts. The overall command structure is then referred to as a command language, or scripting language. The topic of command languages is considered further in Section 16.5. There is an example of a JCL procedure in Supplementary Chapter 2.

**Graphical User Interfaces**

The mouse-driven, icon-based graphical user interface, or GUI (pronounced gooey) has, for all practical purposes, replaced the command line interface as the prevalent interface between user and computer. The GUI has been implemented in many forms. The best-known implementations are the user interfaces present on the Apple Macintosh computer and on IBM-type PCs that are equipped with the Microsoft Windows interface. Most other computer systems offer a similar interface. Figure 16.2 is a picture of a typical Windows XP screen. The screen of a Linux computer running the KDE shell appears in Figure 16.3. Notice the similarities between the two. The graphical user interface provides the convenience of a desktop metaphor. The user can arrange the desktop to his or her own preferences, can easily move around the desktop to perform different tasks, and can see the results in WYSIWYG (what-you-see-is-what-you-get) form.

Windowing systems from different vendors take on different appearances, but share similar elements. Normally, a graphic interface consists of one or more screens or desktops, each of which contains one or more windows. A window is a portion of the screen that is allocated to the use of a particular program, document, or process. Windows contain gadgets or widgets for resizing the windows, for moving the windows around the screen, for scrolling data and images within a window, and for moving windows in front of or behind other windows on the screen. Windows usually also contain a title bar that identifies the window. There is also at least one menu bar on the screen. On some systems, a single menu bar on the screen is always associated with the active window (discussed shortly). On other systems, each window has its own menu bar. Each item on the menu bar can be used to activate a hierarchical set of pull-down menus, used for selecting options within the program being executed. Windows and Linux screens also provide a task bar for rapid program start-up, task switching, and status information.

On many modern systems, Windows can be configured to look and act in different ways. As we already noted, one important option is the Web interface, which provides the look and feel of a Web browser for all operations within the window.

Many systems allow windows to be iconified, tiled, overlapped, or cascaded. Iconifying a window means reducing it to the size of an icon, even though the window is still open.
This allows screen space to be saved. The window is reexpanded by clicking on it. Tiled windows are lined up on the screen in such a way that they do not overlap and use all the available space on the screen. Overlapping windows is the normal situation, with windows located where placed by the system or by the user. Cascaded windows are a version of overlapped windows where the windows are overlapped in such a way that the title bar and one other border of each window can be seen. Some of the newest systems use transparent windows and miniaturized windows to ease the user’s task of keeping track of everything on her desktop. A less common, but creative, approach used on certain systems “rolls up” the windows like a windows shade, showing only the title bar.

On systems that allow multiple screens, a group of windows is attached to a particular screen. Individual screens, together with their associated windows, can be minimized or can be moved forward or behind other screens, but are not usually resizable. When multiple screens are allowed, each screen represents a separate user interface. Four desktops are indicated in the task bar in Figure 16.3. A new, somewhat gimmicky, approach places the different screens on the sides of a cube that can be rotated to reveal the desired workspace. The future of this approach is (hopefully) uncertain.
To the user, the window is a box that is used for the input of text and commands to a program and the graphical or text output resulting. Placing multiple windows on the screen provides a convenient way to implement a multiprogramming interface with separate input and output for each program. At any given time, one window is active, meaning that it will respond to the keyboard and mouse. The color or appearance of the title bar is frequently used to indicate which window on the screen is currently active. On some systems, moving the mouse cursor into a window activates the window. This method is known as **mouse focus**. In other systems, the window must be activated by clicking the mouse while the cursor is inside the window or by opening the window. This is known as **click to focus**.

Depending on the GUI design, data and program files can take the form of text or of graphical symbols called icons. On most systems, icons can be animated. Icons that are in use change shape or color. Icons can be moved around the screen with a mouse. Some systems also allow the use of a light pen or a tablet or a pressure pen, or even, a finger, for this purpose. Pressure pens are commonly used as the pointing device on personal digital assistants such as the Palm Pilot. The pressure pen itself is just a piece of plastic. The actual pointing signal is produced when the system detects the physical pressure at a particular location on the screen or senses the position and movement of a finger near the screen.
Many commands are issued to the operating system by moving the mouse and manipulating a button on the mouse at the appropriate time. For example, a program is initiated by pointing the mouse cursor at the icon for the program and clicking the mouse button twice. A copy command involves holding the mouse button down while the user drags the icon from its original position to a position on top of the desired destination directory icon or window, then drops it by releasing the button. A delete command is performed by dragging the icon to an icon that represents the picture of a trash can. Other commands use the mouse together with pop-up or pull-down menus.

Both program and data files are stored in folders, and folders may be nested. Double-clicking on a folder opens a window that shows the contents of the folder. In other words, folders are equivalent to a file directory structure. The active window corresponds to the current directory attachment point.

Requestor boxes can be used for commands that require textual input, such as a new file name. Additional gadgets, such as push buttons and sliders, exist for other types of control of the interface.

Most window-interfaced operating systems also allow the use of special keys on the keyboard to duplicate commands that use the mouse-icon-menu method.

**EXAMPLE**

The Macintosh OS X interface consists of a single screen known as the desktop. The desktop can be used to hold various items, such as a trashcan, folders, and data that is being worked on. A typical Macintosh OS X screen is shown in Figure 16.4. The desktop consists of icons representing each volume on the system, a menu bar, the dock, and a trash icon. A volume consists of a disk or partition of a disk.

Clicking the mouse on a volume or dock icon opens a window. On the Macintosh, icons can represent folders, applications, and documents. Clicking on an application will launch (load and execute) the program. Clicking on a document icon will open the associated application program and load the indicated data. Some windows may also be opened from the menu bar.

The dock serves as a convenient receptacle for applications, folders, documents, files, and volumes that are accessed often. It also holds icons for minimized open applications. Mac OS X does not distinguish between applications which are open and minimized from those that are permanently placed on the dock for convenience. If you look at the figure carefully, you will see that the dock is divided by a vertical line into two sections. The left section holds applications. The right side of the dock holds folders, documents, and the like.

Names for the icons on the dock are hidden; they appear when the mouse pointer is positioned on the icon. Some dock icons also sprout pop-up menus. Drag-and-drop is used to add icons to the dock. Icons are removed by dragging them out of the dock. The dock can be hidden, shrunk, or rotated into a new position on the desktop.

Two windows are open in the figure. The open windows provide a title bar that contains the usual gadgets, known on the Macintosh as controls. These are used to close the window, to expand the window to full screen size, to minimize the window, to resize the window to a desired size, and to provide scrolling left and right, up and down to expose the underlying
FIGURE 16.4

A Typical Macintosh OS X Screen

The Macintosh desktop uses a single menu bar that is shared by the active window and the desktop. The mouse can be moved over the menu bar to cause pull-down menus to appear. Many of the items on pull-down menus are standard no matter what the user is doing. Those that are inappropriate at a particular time are “grayed out”, that is, represented in a lighter color. These items do not respond to the mouse. This standardization makes the interface easier to learn and to use. Some menu bar items change to represent possible actions that are unique to a particular situation.

Numerous windows can be open at once, each with its own work, and it is possible to cut or copy and paste data between different windows by marking the desired data using...
the mouse and using the cut, copy, and paste menu functions. Only one window is active at a time. The user selects the active window by clicking the mouse anywhere inside it. Even though windows can overlap, the active window will always be brought to the front and displayed in full. Once a window is active, the user can manipulate the window and the data within it. During program execution, buttons, dialog boxes, and pull-down menus are used to control the program and to enter data, simplifying interaction between the user and the program.

In the figure, iPhoto is an application. The other open window is a special Finder window. The Finder window is used to navigate the system. It contains a tool bar that works similarly to a web browser and panels that represent a hierarchy of folders and their contents. The tool bar can be modified for a particular user’s preferences. Applications can be launched and documents opened directly from the Finder.

Besides ease of use, what is important about the Macintosh interface is its consistency. Throughout the interface, every operation works the same way. This enhances ease of use and quick learning, as well as user comfort. A powerful library of graphics software routines within the operating system is used to enforce this consistency.

It should be noted that the graphical interface is totally committed to the user end of the user interface. Internally, the commands are executed by the operating system in essentially the same way as that of any other interface. We mention this so that you can see the value of separating the various layers of the operating system conceptually. Modification or change of one layer does not have to affect other layers, as long as the interface between layers is self-consistent.

Trade-offs in the User Interface

It might seem obvious to you that the ease of use of a graphical interface makes this the ideal interface for most users. For the typical end user, the graphical interface is indeed an attractive choice. It is easy to learn and easy to use. Little training is required, and the basic operations are intuitive. It therefore meets the most important criterion of the user interface: it is effective in allowing the user to get work done. The graphical interface has a second, less obvious, advantage as well. With a graphical interface, it is easy to implement a multitasking system in which the user can control every task by placing each executing task in a separate window. Although some command line systems provide a way to execute programs “in the background”, the method is much more awkward: switching between tasks is not convenient, displayed output is mixed together, and it is difficult to separate and interact with both programs.

In addition, the GUI reflects current computer usage. Modern computers are used routinely to display graphics, photos, and video. The interface is simply more consistent with this type of usage than the CLI would be. The common use of the Web browser on devices such as cell phones, in addition to its use as a primary application tool on computers, provides further support for the GUI as a primary interface for the average user.

The graphical interface is not without disadvantages, however. The graphical interface is much harder to implement and much more demanding in its hardware and software requirements. This interface works best with a powerful graphic video capability. It requires
a lot of memory, just to store the pictures as well as to hold the programs. The software
is complex, although visual and object-oriented languages and API services simplify the
coding of such programs.

In contrast, the command line interface is simple and straightforward. It is text
oriented, and input to the command interpreter can be treated as a simple serial character
stream. The command line interface also has more inherent flexibility and power. Many
experienced users consider the graphical interface to be slow and clumsy. They prefer to
type a command to get their work done. Arguments and operands are easy to use and to
specify. It is easier to work with wild card commands when an operation is to be repeated
many times or when a specialized search is to take place. It is more difficult to combine
commands or to use piping techniques using a graphical interface.

Even though the graphical I/O built into user programs is easy to use, the development
of graphical I/O for user programs is more difficult, and the programs are larger and slower,
because of the numerous details that must be handled by service routines. It is obviously
easier to read and write a stream of text than it is to draw windows, handle menus, and
identify mouse movements and actions.

Finally, it is more difficult to combine a series of graphical actions into a working
script of commands, especially when branches and loops are required, although Windows
is attempting to move in that direction with Windows PowerShell. One of the powers of
the command line interface is the ability to “program” the commands.

Despite these difficulties, the graphical user interface is convenient and useful for most
users in most circumstances. It is the primary interface on most personal computer systems
and is relatively common on mainframe computers as well.

Gradually, too, the disadvantages of this interface are being solved. Most systems
now provide an alternative command line interface, such as the command prompt in
Windows, for example, for use in situations where the graphical interface is inconvenient
or weak. Application programs now exist to help the program developer with the creation
of windows and other tasks required for the program interface. And of course the Web
browser offers simple tools to create application interfaces that are both inherently graphical
and easy to develop.

Standards exist that allow different computers and terminals to share the same
graphical interface, even though the individual hardware and software is different. Such
capability is important in networked and distributed computing environments, where
display elements from one system must be displayed faithfully on a different system. As we
have already noted, one obvious choice for this capability is the use of Web technology—
Java applets, scripting languages, HTML, and XML and the like—to create the required
display.

An attractive alternative in many instances is X Window, which allows various
computers that use various forms of UNIX, Linux, and certain other operating systems
to work together graphically. X Window provides a language that describes the graphical
interface; each individual computer implements the language within its own operating
system to produce the desired result. The X Window system was developed at MIT in
1986 and has been accepted by most manufacturers as a way of furthering the idea of
a standard graphical interface regardless of hardware. X Window is discussed further in
Section 16.4. In many instances, Web browsers can serve in this role, using Java applets,
scripting languages, HTML, and XML to create the required display.
Software Considerations

The programs that control the user interface must perform two primary functions:

- Maintain the appearance of the user interface on the screen
- Translate user requests into user services and initiate the programs that will provide those services

Of course, if the interface is a command line, maintaining the appearance of the interface on the screen is trivial, since it is necessary only to print a prompt and wait for a response. CLI interfaces are text based; therefore, any necessary remote display over a network, for example, is not a problem.

Similarly, translation of CLI commands into the appropriate services is simple. It is necessary only to compare the text typed by the user to that of known commands or file names and execute the command as typed by the user. If the command is internal, it is executed within the operating system. If it is external, it is loaded and executed. The operands on the line are passed to the command procedure as arguments.

Windowing interfaces are more difficult. The windowing software is responsible for drawing and maintaining the appearance of the screen; for creating pull-down menus and dialog boxes; for reacting to requests made by the user in the form of mouse clicks; for maintaining the positions of different objects on the screen; for opening, closing, moving, and resizing windows when requested; and for accomplishing many other tasks.

Even a task as conceptually simple as moving an object on the screen, say, the cursor or an icon or a window or a slider control, requires a considerable effort in terms of programming. (Picture trying to write the program to do it using the Little Man Computer machine language!) As the user moves the mouse, the mouse generates interrupts to the CPU. The mouse interrupt program determines the direction and distance that the mouse has moved. It calculates the new X and Y coordinates of the object on the screen by geometrically adding the move to the present position of the object. Then it redraws the object at the new location, by storing a picture of the object in display memory. When it does so, it must also store the shadow of the image stored “behind” the new position of the object and restore the image of whatever was hidden by the object in its previous position. This operation is depicted in Figure 16.5.

(We note that in modern desktop computer systems and workstations, the software required to perform tasks such as the image shadowing just described is actually built into some graphics display controllers. The division of labor between the controller and the operating system is established by the
device driver software used with a particular graphics controller. Either way, the task must be performed.)

In addition to display maintenance and handling, the interface program must also interpret the commands and request the appropriate services. Double-clicking on a document icon, for example, requires the software to identify the icon from its location on the display screen, determine the associated application program, load and execute the application, and load the document data. The command interpreter is thus somewhat more complicated for a windowing system. Note that requesting a service will also require use of the display services, since it will be necessary for the new application to open one or more windows of its own, set up its own menus, and so on.

Finally, it is useful to consider the display of objects within a window, such as images in a photograph processing application, or formatted text in a word processor, or a page in a Web browser. Responsibility for each of these displays lies within the particular applications. The output to be displayed in a window will take the form of objects and bitmaps, as described in Chapter 4. The applications use the API facilities within the operating system to produce the actual display. In turn, the operating system will then create the overall screen display using a combination of its own software with that of the graphics display controller.

Overall, you can see that the graphical user interface software is considerably more complex than the corresponding CLI or menu interface software.

16.4 X WINDOW AND OTHER GRAPHICS DISPLAY METHODOLOGIES

Graphical user interfaces are attractive, convenient, and relatively straightforward when the computer and display are located together, such as in a personal computer or workstation. When the display terminal is separated by a distance from the computer the graphical interface is more difficult to achieve. Such a situation might occur, for example, if a user is trying to operate her computer across a network, using the local display and mouse facilities on a remote computer. The difficulty is the large amount of data that must be transmitted from one location to the other to transmit graphic images. In Chapter 9 we observed that a single bitmapped graphical image might contain thousands or millions of bytes of information. Clearly, it is not practical to transmit the screen display as a bitmapped image across the network on a continual basis.

The X Window standard represents one successful attempt to solve this problem. X Window works by separating the software that actually produces the display image on the screen from the application program that creates the image and requests the display into a somewhat unusual client-server arrangement. The program that produces the image on the screen is known as a display server. We remind you that in data communications terminology a server is a program that provides services for other programs. In this case, the server provides display services for one or more client application programs. (We have assumed that the client application programs are running on computer systems located remotely from the display, although, as you will see shortly, this is not a necessary assumption.) The display server is located at the display terminal, computer, or workstation where the image is to appear. The display server can draw and control windows. It provides gadgets, dialog boxes, and pull-down and pop-up menus. It can create and display various
fundamental shapes, such as points, rectangles, circles, straight and curved lines, icons, a cursor, and text fonts. In conjunction with a mouse and keyboard located on the same terminal the display server can move, resize, and otherwise control these windows. The mouse can also move the cursor under display server control, that is, local to the terminal. It has only to notify the application as to the final position of the cursor.

Thus, much of the work in creating a graphical window interface is performed local to the display itself and does not have to be transmitted from the computer system that is running the application program. The application program uses the display services to produce its desired images by interacting with the display server. Again, in data communication terminology, the application program acts as a client, requesting display services that it needs from the display server. The program may request that a pull-down menu be displayed, for example; the display server draws the menu on the screen at the appropriate location, as determined by its knowledge of the size and location of the window. If the user at the terminal clicks a mouse on a particular menu entry, the server notifies the application that this event has occurred. Figure 16.6 illustrates the operation of an X Window application with a display server.

Although it is unavoidable that the application must still transmit actual image data to the display, the amount of data to be transmitted is considerably reduced. Essentially, the display server can perform many of the basic display operations with very little communication required. WYSIWYG text, for example, requires only that the choice of font, the

---

**FIGURE 16.6**

The X Window Client-server Relationship

The X server accepts requests from X client 1, running on remote host "boris.

It also accepts requests from X client 2, running locally on "natasha.

The X server program running on "natasha" manages its display hardware and provides the display resource that each client needs.

display location, and the actual text data be transmitted. The font data is stored at the display server. The server also provides a library of all the basic tools and widgets to draw windows, provide drop-down menus, present control buttons, respond to mouse clicks, and many other functions. This method requires far less data communication between the program and the display than if the transmission of actual text images and windows were required.

X Window places no restrictions on the location of the client application; therefore, the application program could reside on the same computer system as the display server or remotely on a different system. Furthermore, the display server can process requests from several different client applications concurrently, each in its own window. You would expect this to be true, since a graphical user interface can have several open windows at one time. This leads to an interesting and exciting possibility: that different windows on a single display could be communicating with application programs on different machines! Indeed, this is the case. The picture in Figure 16.7 illustrates this situation. The window in the upper-left corner is communicating with a program located on the same PC with the display. The PC is running an X Window server under Microsoft Windows. The other windows on the screen are connected to various systems located remotely: a (now ancient) VMS system connected to a local area network and a Sun UNIX workstation connected

**FIGURE 16.7**
A Multicomputer X Window Display
via telephone and modem. This image was taken many years ago, but the X Window technology that created it has changed only in minor details.

As in other windowing systems, the location of the cursor is used to determine the active window.

Notice that the operating system at the display need not be the same as the operating systems where the application programs are running, as long as display server software is available for the particular operating system in use at the display terminal. X Window display server software is available for most operating systems, including UNIX, Linux, Windows, the Macintosh OS, and many others. In fact, an X Window server can even be built into a bare bones display terminal and used with a central processor located elsewhere.

Systems similar to the X Window system also exist at the application level for services such as the Web-based services mentioned previously. Although these systems operate somewhat differently, the concept is quite similar: to produce as much of the display as possible using software located at the display and to minimize the amount of image data to be transmitted. These services provide software, display standard formats such as PNG, PDF, and SVG, and communications protocols, particularly HTML and XML for this purpose, and also store commonly used images and display features at the display site.

16.5 COMMAND AND SCRIPTING LANGUAGES

In early batch systems it was necessary to submit all pieces of a program at one time, including the data and any other programs that were required as part of the operation. Generally, punched cards were used for this purpose.

Suppose you had a COBOL program (it would have been COBOL in those days!) that was to be compiled and executed. Your submittal to the computer required the cards that made up the COBOL compiler, the COBOL program itself, perhaps a library of binary math routines that will be called by the program when it is executed, and finally some data for the program. Your submittal also contained commands to the computer telling it what to do with all these cards. (Some of the commands were actually more informational, like “load the following group of cards, which are data for the program that you are running.”)

The entire submission was known as a job.

Later on, the COBOL compiler and math routines became part of the software that was permanently stored on disk within the computer system, but the system still needed to be told what to do, what programs (e.g., the COBOL compiler) to call from the disk, where to find the data, and where to print the results.

As noted, the commands to the computer took the form of punched cards that contained the required information. These cards were called job control cards because they told the computer system how to control the job. The different types of commands made up a sort of language, called job control language or, more commonly, JCL. In other words, a job consisted of a series of commands in JCL intermixed with the appropriate programs and data.

The best known of these languages is IBM zOS/JCL, but you should be aware that other vendors have their own JCL languages. Generally, there is no compatibility between JCL languages from different vendors, and in fact, there are several different incompatible versions of “IBM JCL”. (By the way, you might note that the expression “JCL language”, which is in common use, is redundant.) The use of JCL continues to this day. JCL statements
are entered into the system with a screen editor and are stored in the system as a file of **card images**, so called because each statement is still laid out as though it were stored on an 80-column card. These card images are usually batched together as a file of commands that is executed in the same way that a program is, except that each line of the “program” is actually a command to the operating system. Operands in each JCL command specify the appropriate files and other details that are required. Both positional and keyword operands are used.

The commands that you use on your computer are not very different from the ones that make up a job control language. After all, a computer is a computer, and the tasks that you do are not really very different from those that are done as part of a batch job. Although you commonly perform command operations one at a time, you can probably think of times when it would have been convenient to combine a group of commands that you could have done all at once. In fact, you may already be aware that there is a way to do this.

As we previously have noted, operating systems provide a way to group commands into a single file that can be executed as if it were a program. The file may itself even contain executable programs. In addition to the usual commands, the operating system provides additional commands that are used specifically for this purpose, commands that allow branching and looping, for example. Figure 16.8 is an example of a Windows command language script that prepares backup for a text file by copying an old version to a flash drive (E), creating a new backup on hard disk, and then opening the word processor program. The name of the text file is given as an operand when the command file is executed. “%1” is a placeholder for the operand.

Most modern operating systems provide command languages. Perhaps the most elegant implementation of this concept belongs to UNIX. In addition to the usual commands, the UNIX and Linux shells contain a set of powerful utility commands and other features that allow the creation of very sophisticated command programs. In UNIX and Linux, shell scripts are a fundamental way of getting work done. Shell scripts can be executed just as if they were programs, and in fact, due to the power of the shell script language, it is frequently possible to avoid writing a normal program by using a shell script instead.

**FIGURE 16.8**

Windows Program DOWP

```bash
@echo
if "%1" == '' go to error
if not exist %1 goto error
if not exist %1.txt goto newtxt
if exist %1.old copy %1.old e:\%.arc
copy %1.txt %1.old
wordproc %1.txt
goto end
:newtxt
echo This is a new file. Opening word processor....
wordproc
goto end
:error
echo proper command format is 'dowp filename'
echo with no extension.
:end
```
CHAPTER 16  THE USER VIEW OF OPERATING SYSTEMS

FIGURE 16.9

The Linux Airport Distance Shell Script

```
export city state lat long port
grep -i "$1 $2" townfile | read city state lat long port
if [ -z "$city" ] then
echo "this city is not in the file"
elif [ "$port" = "y" ] then
echo "$city $state has its own airport"
else
    awk 'BEGIN {close = 10000}
        $5 == "y" {dist = ($3 - "$lat"){($3 - $lat') + ($4 - "long") * ($4 - "long")}
            if (dist < close) {
                close = dist
                ccity = "$1"
                cstate = "$2" }
    ' townfile
fi
```

A typical line in townfile:

```
Boston MA 42.3333 71.083 y
```

You’ve already seen an example of a Windows command language script that assists the user in performing a routine computer task. Many Linux operating system operations are, in fact, shell scripts. Shell scripts often can be used in place of a conventional program, as well. The example shown in Figure 16.9 is a UNIX Bourne shell program that determines the nearest major airport to an arbitrary city entered by the user.

Scripting languages are expanded forms of command languages, often encompassing features well beyond those found in a standard command language. Some scripting languages even provide portability between different operating systems. Perl, Javascript, PHP, and Python are examples of portable scripting languages.

The Elements of a Command Language

Like any programming language, there are certain elements in a command language that affect the usefulness of the language. The choice of commands and utilities is an important consideration, but it is not the only consideration. There are other features that enhance the value of a language. These are some of the most important features of a command language:

- The ability to print messages on the screen and to accept input from the user into the script.
- The ability to specify variables and a method to assign and change the value for those variables.
The ability to branch and loop. Notice that the ability to change variable values is important as a means of controlling branches and ending loops.

The ability to specify arguments with the command that executes the program and to transfer those arguments into parameters and variables within the program. The command script in Figure 16.8 uses this technique to allow the user to specify the name of the file to be backed up.

The ability to detect an error that results from a command and recover from it in a graceful way. If the operating system attaches numerical values to its errors, the command program can make decisions and branch based on the level of error. This could be used to determine if a particular file or a particular hardware component exists, for example.

The Command Language Start-up Sequence Files

A major use for a command language is the system start-up file. Most modern operating systems allow the execution of specific command files at system start-up and also when a user logs in to the system. Two types of start-up files exist. One type is used to configure the basic operating system, as we discussed in Chapter 15. Start-up configuration files are only modified by the system administrator.

The second type of start-up file is used to tailor the system to the user’s preferences. User start-up commands can be used to set various parameters, such as the preferred places to look for files, the type of terminal that the user is working with, the selection of a command shell, and the appearance of the command line prompt. On a system shared by many users, the user start-up command file can be tailored to each individual user. It is executed whenever a user logs in. Login start-up files for a UNIX system depend on the default shell being used for the system. The Bourne shell start-up script is called \texttt{.login}; the C-shell script is called \texttt{.cshrc}. Since these files are text files, they can be easily communicated across a network to provide uniform capability to all users on the network. This allows a system administrator to change every user’s profile with the modification of a single file.

16.6 SERVICES TO PROGRAMS

Most of the discussion in this chapter has centered around the user interface and methods of controlling and using the interface, but we would be remiss if we did not say a few additional words about services provided by the operating system to application programs that support and affect the user interface. As we noted in Chapter 15, operating systems have long provided services to application programs in the areas of file management, I/O processing, and system resource management. A few operating systems, particularly the Macintosh OS, have provided services such as the ToolBox that extend these capabilities to include the graphical user interface. Services outside the operating system, but not quite applications either, such as X Window, extend graphical capabilities to other systems.

Windows provides similar services through the Win32 API. These services reflect the trend to expand the role of the operating system generally to include services and support to application programs and users that provide many capabilities that were formerly
within the applications themselves. As we noted earlier, these services enable the system to provide a standard look and feel for different applications extending, even, to the Web interface. They simplify and extend the graphic capabilities of application programs, they improve the capability of programs to communicate with each other and to pass data from one application to another, they provide the ability to launch an application program from within another, they provide e-mail and other communication capabilities, and they provide document and graphical storage services at a more sophisticated level than was found previously in traditional OS file management facilities.

By integrating these capabilities into the operating system, the system can assure that every application program responds in similar ways to user actions. Integration also provides smooth and seamless interaction between the different applications. Just as the file manager assures a consistent representation and interface for file manipulation across different devices, so these new services provide the user with a more powerful and easier to use way to access his program applications. The overall effect is an increased emphasis on the user interface and new ways of working that are more oriented toward the work to be accomplished and less to the launching and manipulation of application programs. Although many of these tools are found in a “shell”, they are more tightly integrated into the operating system than was true of previous shells.

The addition of new operating system services is intimately tied to improved programming methods, particularly object-oriented programming. These services commonly take the form of libraries of objects that application programs request via the usual call mechanisms. Several standards continue to evolve for these services. Standards are necessary so that applications may be developed with the assurance that the services required will be available. The most important of these standards include DCOM, .NET, and CORBA. The dividing line between the operating system and the application programs has become increasingly unclear, as the operating system moves into a realm that has traditionally been part of the application programs themselves. The use of the Web browser as a primary user interface provides a common look and feel that extends all the way from local file management to Web-based business applications to the services of the Web. Perhaps these services will be considered part of a new OS layer, called application program services, or perhaps the operating system itself will be divided differently, into user services and a kernel that provides just the basic internal services. Some researchers and operating system designers envision a model consisting of a small operating system extended by objects that support the user’s applications. This approach suggests the growing operating system emphasis on the user environment and on application integration.

Overall, the effect on the user’s interaction with the computer has changed dramatically in the last few years and promises to change even more profoundly in the future. Presently, the user performs her work by opening applications and working within those applications. The concepts of suites of applications, at the application program level, and of object linking, at the system level, extend this capability to allow the applications to communicate, to share data, and to perform operations within an application by launching another application. The additional capabilities envisioned for software at the system level, whether considered part of the operating system or another type of interface shell, will expand this process and ultimately can be expected to shift the user’s focus almost entirely to the document, data set, or other work entity, with applications launched invisibly as they are required to accomplish a particular task.
Modern operating systems provide an interface that is used by programs and by the user to interact with the system. The interface provides a variety of user and program services, including a user interface with command capability, program execution and control capability, I/O and file services for programs and for users, command languages for building programs of commands, system information, security features, and interuser communication and file sharing capability. An application programming interface (API) provides a standard set of services that application programs use to access the operating system and to provide a common look and feel to their user interfaces. In this chapter we considered each of these services.

Most systems are primarily interactive. For this purpose there are currently two primary types of user interface, the command line interface and the graphical user interface, each with its own advantages and disadvantages. Similar operations can be performed with each, although the method employed is different for each. For users with limited needs, a Web-based interface is often a more suitable and productive means of access to the system.

X Window is an important graphical display methodology, particularly in networked and distributed environments. X Window is an attempt to provide windowing capability while partially solving the difficulty of transmitting large quantities of graphical data from one location to another. X Window is built around a client-server architecture.

Command languages allow a user to build more powerful functions out of the command set. Most command languages provide looping and selection capability, as well as interactive input and output. Some command languages are intended for batch processing. IBM’s zOS/JCL is an important example of a batch language.

FOR FURTHER READING

A general book that discusses the user interface in great detail is Sharp [SHAR07]. Marcus and colleagues [MARC95] provide an easy-to-read discussion and comparison of graphical user interfaces. There are several others listed in the references, including Weinschenk [WEIN97], Tidwell [TIDW06], and Tufte [TUFT90]. There are also numerous books on Web design, a related topic not included here. The general operating system aspects of the user interface and program services can be found in any of the operating system texts identified in Chapter 15. The X Window system is introduced well in Christian and Richter [CHRI94] and presented in much more detail in many books, including Mansfield [MANS93] or Jones [JONE00].

KEY CONCEPTS AND TERMS

- access control list (ACL)
- active window
- application association
- application program interface (API)
- batch program
- card image
- click to focus
- command language
- command line interface (CLI)
- command shell
- common look and feel
- controls
- CORBA (Common Object Request Broker Architecture)
READING REVIEW QUESTIONS

16.1 What is the primary purpose for a user interface?
16.2 What effect does the quality of a user interface have on the use and productivity of a computer?
16.3 What is the purpose of a command language or scripting language?
16.4 What is the advantage of offering the same user interface for applications, user programs, and commands?
16.5 Discuss the major tradeoffs between a command line interface and a graphical user interface.
16.6 Describe the user interface provided by a CLI command shell.
16.7 Describe the format of a CLI command. What is an operand? What is the difference between a keyword operand and a positional operand?
16.8 Sketch or “print screen” a typical GUI screen. Label each of the major components on the screen.
16.9 In addition to the commands themselves, command languages provide additional capabilities that are important when a user is not directly involved in the control and execution of each command. Name at least three features that are necessary to make a command language useful.
16.10 Although Web-based user interfaces are somewhat limited when compared to CLIs and GUIs, their use in organizations has grown and continues to grow. What advantages do Web interfaces offer over CLIs and GUIs?

EXERCISES

16.1 List and explain some definite advantages to the use of a command line interface over other types of interfaces. Do the same for a graphical user interface. Do the same for a Web-based interface. What is the target audience for each type of interface?
16.2 Discuss the advantages and disadvantages of providing the user interface as a separate shell, rather than as an integral part of the operating system.
16.3 If you have access to two or more command line interface shells, such as Windows and Linux bash or Linux bash and tcsh, compare the major commands that are available on each. Note the similarities and differences between them, particularly in their capabilities and in the way the command task is performed.

16.4 Consider the major commands in a command line interface system such as the Linux bash shell. Explain how each task would be performed on a graphical user interface system such as Windows or the Macintosh.

16.5 There are some capabilities that are easy to achieve with a GUI, but much more difficult with a CLI. Describe a few of these capabilities.

16.6 Explain piping. What additional capability does piping add to a command language?

16.7 Explain the concept of redirection. Illustrate your answer with an example of a situation where redirection would be useful.

16.8 What purpose do arguments serve in a batch file or shell script?

16.9 Identify the name and purpose of each of the components of the GUI that you use.

16.10 Use the batch file or shell script capability of your system to build a menuing interface that implements the most common commands that you use.

16.11 Describe the difficulties that exist in providing a GUI at a location remote from the computer that is creating the display. Describe the methods used by X Window to partially overcome these difficulties. Why is it not possible for X Window to totally solve these problems?

16.12 Discuss the advantages that result from the client-server architecture of the X Window system.

16.13 When people describe client-server architecture, they are usually referring to a system in which a large server is serving a client on a PC. With X Window, the reverse is frequently the case. Explain.

16.14 The designers of the UNIX operating system described the ideal shell command language as one that is made up of a large set of simple commands, each designed to perform one specialized task well. They also provided various means to combine these simple commands to form more powerful commands.

   a. What tools are provided to combine the commands?

   b. What are the advantages of this approach over providing a smaller set of much more powerful commands? What are the disadvantages?

   c. If you know something about the UNIX or Linux command set, discuss the characteristics of UNIX/Linux commands that make it easier to combine these commands powerfully.

16.15 If you could design a “wild card” system with features beyond those normally provided in a CLI, what features would you add?