Chapter 3
User and Program Interfaces

Revised: Jan. 24, 2011

3.1 INTRODUCTION

If a computing system is to perform useful services, it must support communication with the system's users---the people who use it to support their application programs. This communication involves a two-way transfer of information: a user informs the system when specific services are required, and the system provides the user with the results of those services. The communication may be direct, such as through a typed dialog at an interactive terminal, or indirect, through jobs or programs submitted for batch execution. The operating system components responsible for managing this communication are the **user interface** and the **program interface**.

The user interface is the gateway for interaction and communication between a computing system and its users. To many users, the acceptability of an entire computing system is largely determined by its user interface. Yet many operating systems seem to do a poor job of user-interface design, and many OS texts do not consider the subject. Although there is no widespread agreement on the form an ideal interface should take, some common ideas have taken shape that have improved the quality of many existing user interfaces and can help in designing better ones in the future. These ideas are the motivation for this chapter.

The user interface directs the processing of a set of programs by providing high-level communication between the user and the operating system. Although many radically different modes of communication are possible, this interface typically takes one of two forms: a **command line interface (CLI)** or a **graphic user interface (GUI).** The CLI is so named because users normally type or submit a series of **commands** to tell the OS what actions to perform. This is the form used by most Unix and Linux systems, by Microsoft MSDOS, and by the Command Window available in most versions of Microsoft Windows. In contrast, communication with a GUI involves actions such as selecting graphic objects from the screen by means of a pointing device. This is the form used, for example, by the Macintosh OS, Linux desktops, and the various versions of Windows.

Significant variations that would still be classified as GUIs are increasingly common today on mobile devices like smartphones and tablet computers.

The program interface manages running programs and provides interaction between these programs and system-controlled resources and services. Most of these services are provided in a form that is generally known as an **application program interface (API).**

The program and user interfaces are usually related, as shown in Figure 3-1. The user interface is implemented by a program unit, often a system process. Like other programs, the user interface uses the program interface to access system services. Moreover, in some operating systems the program interface includes a method to invoke the user interface (especially a CLI). This feature allows programs to construct commands and pass them to the command interface for execution.

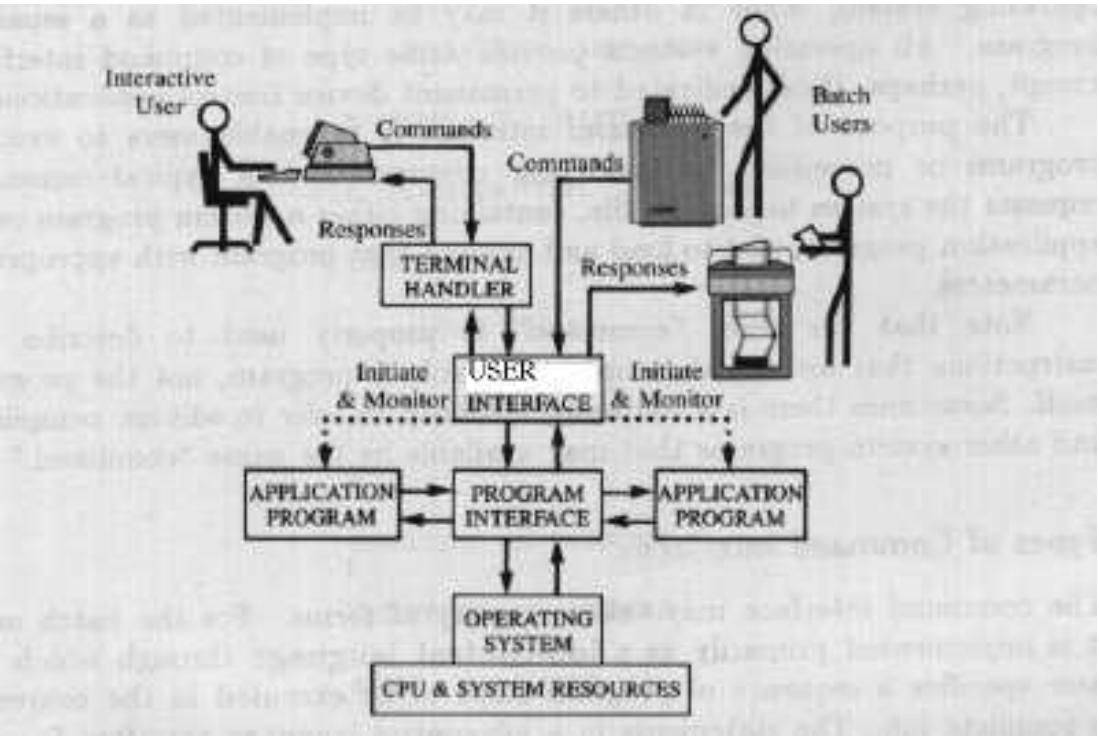

Figure 3-1: The User Interface

3.2 USER INTERFACE TYPES

The type of user interface provided for a given OS is determined in large part by the types of interactive devices available. We can identify the following fundamental classes:

1. **Batch**: stream input and output only. No interaction.

2. **Teletype**: Interactive using keyboard input and typewriter ("hardcopy") output. Display in lines with no reversal possible. One limited character set, 80 chars per line.

3. **Alphanumeric**: Interactive using keyboard input and video display output limited to characters from a single, fixed-size font. Full ASCII character set. 80 chars by 24 lines. Concept of a cursor position. 2-D cursor movement possible. Selective changes and erasing possible. Vertical scrolling usually supported, usually downward only.

4. **Graphic**: Interactive using keyboard input plus position selection by mouse or similar device. Display on "bit-mapped" screen. Size and resolution vary widely. Capable of displaying at least as much text as alphanumeric, usually more. Multiple character types and sizes on display with proportional spacing. Simple graphic images including line drawings, icons, windows, boxes, etc. Often (not always) supports color or shading, using many different models.

5. **Multimedia**: Interactive using sound and animation in addition to the capabilities of the graphic class. Generally sound output (high quality stereo) and sound input (with limited voice-recognition capabilities) are available. Video images can be displayed on the screen, but are usually limited to a single small window and update rates of a few images per second.

6. **Handheld**: Interactive within the constraints of a computing device small enough to hold in a hand or place in a pocket. Input is performed by a limited or “soft” keypad and/or by touch using fingers or a stylus. Output appears on a miniature display screen.

Teletype interfaces may be considered essentially obsolete, and batch interfaces occur only in limited situations. Multimedia and handheld interfaces are rapidly emerging concepts. The great majority of today's interface mechanisms, though, are alphanumeric or graphic.

Command line interfaces are usually based on the alphanumeric class. However, they originated with the teletype class, and sometimes still show some of its limitations. Simple menu interfaces are also based on the alphanumeric class.

Graphical user interfaces are based on the graphic class of display mechanisms. These have become much more widespread in recent years. We will discuss GUIs later in this chapter. User interfaces for multimedia and handheld systems will also be considered briefly.

3.3 COMMAND LINE INTERFACES

The command line interface is widely used by timesharing systems such as UNIX or Linux, and was formerly the norm on PCs running MS-DOS. Further, current desktop systems with

-

GUIs, such as Windows and Macintosh, incorporate command line interfaces that are preferred for some purposes. Such an interface provides a mechanism for humanly understandable, two-way communication between an operating system and computer users when full graphics is not necessarily available, when high overhead cannot be tolerated, or when a concise and powerful language is required.

Whatever its actual form, this communication makes use of one or more languages. A language by which a user conveys instructions to the operating system may be called an **Operating System Command Language (OSCL)**. A language by which the OS returns information to the user is called an **Operating System Response Language (OSRL)**. Languages that integrate both these functions may be referred to as **Operating System Command and Response Languages (OSCRLs)** [Beech 1980].

A CLI has two fundamental tasks:

1. Interpret commands in the OSCL and convey instructions to the rest of the operating system to carry out those commands.

2. Accept messages for the user from the operating system and present them using the OSRL.

In some systems, the CLI is an integral part of the operating system, while in others it may be implemented as a separate program.

The purpose of the CLI is to enable users to execute programs or procedures by specifying commands. A typical command requests the system to locate a file, containing either a system program or an application program, and to load and execute that program with appropriate parameters.

Note that the term "command" is properly used to describe the instructions that are typed in order to invoke a program, not the program itself. Sometimes there is a confusing tendency to refer to editors, compilers, and other system programs that may be available by the name "command."

Types of CLIs

The command-line interface may take a variety of forms. For the batch user, it is implemented primarily as a job-control language through which the user specifies a sequence of program steps to be executed in the course of a complete job. The statements in a job-control language are often focused primarily on describing the characteristics of a job step, such as the specific files and other resources to be used.

An example of a job control sequence for ATLAS is shown in Figure 3-2. This example is based on Barron and Jackson [1972, p.146]. The statements in this figure specify characteristics of a job to be run on the ATLAS system.

-

```
JOB
U123456 JAMES MOONEY
INPUT
1  SPECIAL DATA
OUTPUT
0  LINEPRINTER  5000 LINES
3  CARDS  250  LINES
7  FIVEHOLE  TAPE  2  BLOCKS
TAPE
1  MALCOLM  LANE
TAPE  COMMON
3
COMPUTING  10  MINUTES
EXECUTION  15  MINUTES
STORE  71  BLOCKS
COMPILER  FORTRAN
***T
```

Figure 3-2: Example of ATLAS JCL

The most widely known job-control language is that used on IBM's OS/360 series systems. Figure 3-3 shows an example of IBM JCL for a simple FORTRAN program based on Barron and Jackson [1972, p.149]. The statements in this figure instruct an IBM system to compile, link, and run a FORTRAN program. The first card identifies the user and specifies job attributes. The next group runs the compiler (most cards are concerned with defining data files). The next group runs the link program. The final group runs the compiled program itself.

Interactive users have a similar facility, commonly called the **command language**. However, in the interactive environment the interface can be much more complex. The language itself may have an elaborate structure. Various types of prompting and typing assistance commands may be available. The interface may include conventions for returning system messages to the user. Other features might include intelligent terminal handling, immediate commands, help facilities, command logs, and packaging of commands into command files (or

scripts). These features are made available when a user communicates with the operating system itself. In some cases they are supported for communication with other programs as well.

```
//MOONEY   JOB(1023,20,47),LANE,MSGLEVEL=2,PRTY=6,CLASS=B
//COMP     EXEC  PGM=IEYFORT,PARM='SOURCE'
//SYSPRINT DD    SYSOUT=A
//SYSLIN   DD    DSNAME=SYSL.UT4,DISP=OLD,
//               DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//SYSIN    DD    *

                   ...
                (Source Program Cards)
                   ...

//GO       EXEC  PGM=FORTLINK,COND=(4,LT,C)
//SYSPRINT DD    SYSOUT=A
//SYSLIN   DD    DSNAME=*.COMP.SYSLIN,DISP=OLD
//SYSLIB   DD    DSNAME=SYSL.FORTLIB,DISP=OLD
//FT03F001 DD    SYSOUT=A,DCB=(RECFM=FA,BLKSIZE=133)
//FT05F001 DD    DDNAME=SYSIN
//FT06F001 DD    SYSOUT=A,DCB=(RECFM=FA,BLKSIZE=133)
//FT07F001 DD    UNIT=SYSCP

//GO.SYSIN DD    *

                      ...
                (Program Data Cards)
                      ...
```

Figure 3-3: Example of IBM JCL

A basic interactive command language provides stream-oriented communication, in which commands and messages are processed as lines of text. This type of powerful, concise language, generally most suitable for an experienced user, is illustrated in Figure 3-4.

Screen-oriented command interfaces, such as menu systems, provide an alternative approach in which the user chooses from a set of options offered by the system. This method,

which can be helpful to novice users because they need remember no commands, is especially appropriate in certain applications, such as text editing. However, because menu systems restrict the user to a limited number of options at any time and require time for the display of menus, they are not usually favored by experienced users. A menu system is illustrated in Figure 3-5.

```
>dir
        MYPROG1.PAS
        MYPROG2.PAS
        YOURPROG.FOR

>delete MYPROG2.PAS

>pascal MYPROG1

>link MYPROG1 SYSLIB

>run MYPROG1

        This is a test
>
```

Figure 3-4: A Stream-oriented Command System

```
AVAILABLE OPTIONS

1. Compile FORTRAN program
2. Compile Pascal program
3. File services
4. Print services
5. Terminal emulation
6. Run an application program
7. View additional options

Select number (? for help):
```

Figure 3-5: A Menu-oriented Command System

This type of menu is, of course, similar in functionality to the familiar "point and click" menus provided by many GUIs. However, no graphic terminal or input device is required.

CLI Structure

The next several sections concentrate on stream-oriented interactive command languages (as distinguished from batch languages and from menu systems). The term CLI will be assumed to refer to this type of an interface.

The CLI is implemented by a program module called a **command handler** or **shell** (a term denoting the way in which this interface encloses the rest of the OS from the viewpoint of a user). The term "shell" was introduced by MULTICS and has become widely known through its use on UNIX systems.

The command handler or shell may be implemented in a variety of ways. For example, it may be:

- an integral part of the OS, as in OS/MVT or VAX/VMS.
- a distinct module of the OS that can be modified or replaced, as in MS-DOS or CP/M.
- an ordinary program that may be easily replaced, as in UNIX.

These structures are contrasted in Figure 3-6. In some of these cases it is possible to substitute a customized command interface for the standard one. In a multiuser UNIX system, each user may even have a different interface at the same time.

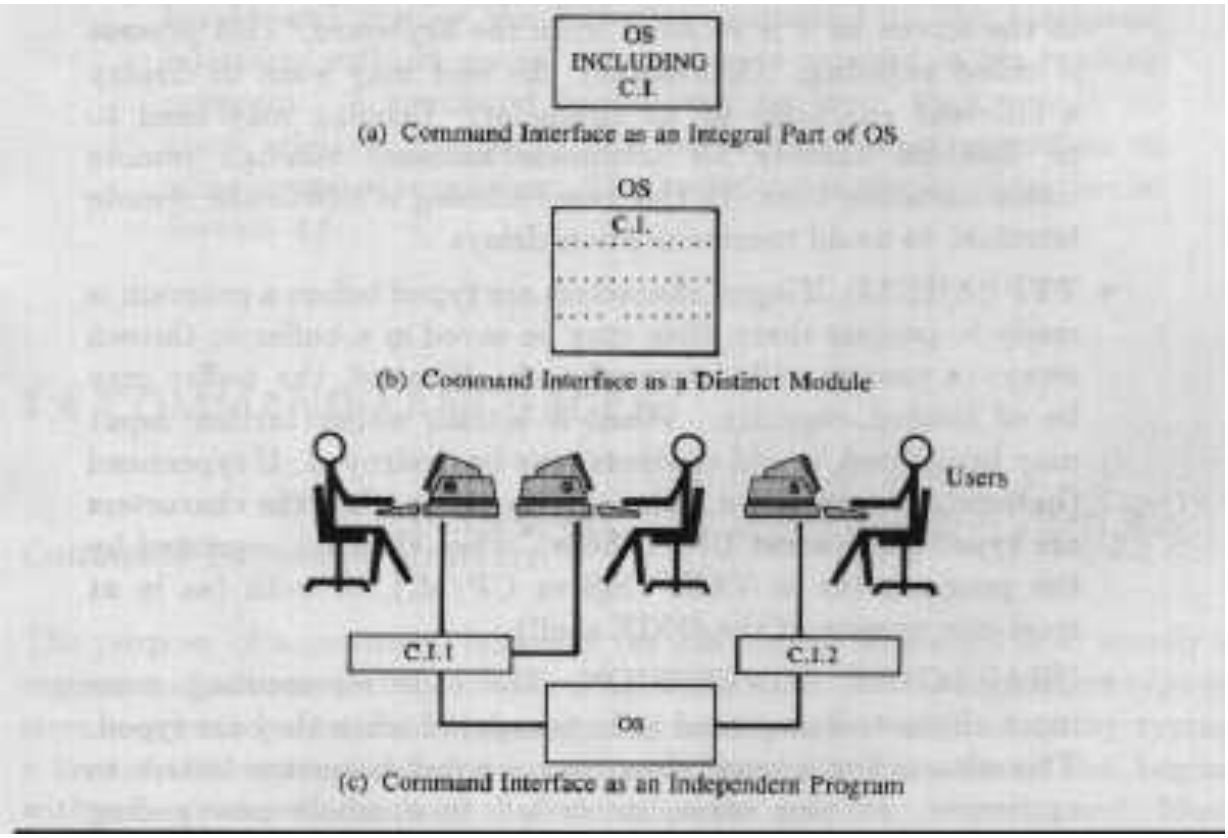

Figure 3-6: Structuring Techniques for the User Interface

3.4 THE TERMINAL HANDLER

The user of a CLI communicates with the OS through an I/O device. Originally this device was likely to be a teletype; after some time alphanumeric display terminals became more common. Today the most likely device supporting this user interface is a desktop computer; if the OS is on a different computer, the desktop pretends to be an alphanumeric terminal.

Input is transmitted to the system primarily by means of a keyboard. In some cases, a pointing device such as a mouse may also be present. Output is presented on a screen that is able to display a number of lines of text, and in some cases also offers graphic capabilities. The terminal device is managed by a software component known as the **terminal handler**.

Although the terminal is managed by techniques similar, in part, to those used for other devices, its central role in the user interface gives it a special status. This role is evident from the terminal handler's position in Figure 3-1. Decisions made by the terminal handler can have a substantial effect on the overall command interface presented to the user, and can determine the form of communication available between users and application programs as well. For this reason, the terminal handler must be considered an integral component of the user interface.

Some issues that directly affect the command interface must be resolved by the terminal handler. They include:

- **Echoing**. The terminal handler normally sends each character to the screen as it is received from the keyboard. This process is called echoing. Occasionally the user may want to display a different character or no character. Echoing may need to be disabled entirely for terminals accessed through remote communication lines. In this case, echoing is left to the remote terminal to avoid communication delays.

- **Typeahead**. If input characters are typed before a program is ready to process them, they may be saved in a buffer or thrown away---a process called typeahead. If saved, the buffer may be of limited capacity. When it is full, either further input may be ignored, or old contents may be destroyed. If typeahead (buffering) is permitted, echoing may occur when the characters are typed (as in most UNIX shells), when they are requested by the program (as in VMS), or both (as in some versions of the UNIX shell).
- **Character conversion**. The code representing some input characters may need to be translated when they are typed. This character conversion may convert lowercase letters to uppercase, or vice versa, or switch to a whole new coding system. Conversion may also be required before echoing, such as converting tab characters to a suitable number of spaces.
- **Line buffering**. The terminal interface may hold input until a full line is typed before sending it to the program. This line buffering technique offers users a valuable opportunity to check, edit, or even cancel a line, as long as the end-of-line key has not been pressed. However, some programs are designed to take immediate action when individual characters are typed, and must receive their input characters one by one.
- **Break characters**. Most interactive command interfaces reserve a few special characters, called break characters, to cause immediate action. On many systems, for example, typing CTRL-C aborts the current program and reinitializes the command handler. Break characters must be recognized and acted on immediately, even if other input typed before them is still in a buffer.
- **Command heralds**. The terminal interface forms a constant layer between the user and all programs that communicate through the terminal, including editors and other interactive programs as well as the command interface. A command herald [Rayner 1980] is a special character or sequence, recognized universally when typed as a prefix for a special command sequence. The terminal handler can recognize such a command herald and convey the following command to the command interface, without special arrangements required in the running program. A command herald may be used, for example, to allow selective logging of any interactive dialog, regardless of what program is running.

-

3.5 COMMAND LANGUAGES

Command Language Structure

The purpose of a command language (or job control language) is to specify a sequence of actions to be performed by the OS. In the most common type of stream-oriented command language, the user specifies each action by typing a line of instructions called a command line. Each command line begins with a single-word command that identifies the action to be performed. Most commands cause a program to be run that carries out the requested action.

For many commands, you may need to specify additional details, such as the names of files to be used for this particular run. You can make this specification through command arguments, which are typed on the command line following the command itself. You can compare a sequence of commands to a computer program in which each statement is a procedure call with appropriate arguments. Figure 3-7 details some typical commands for several command interfaces.

A typical command language defines a set of commands to specify the desired actions, together with the arguments required for each command. The simplest possible command language would include a single command:

```
RUN progname (arguments)
```

This command causes the execution of a specified program. The program name forms the first argument for the command RUN. (The general form of arguments will be discussed later.) This is analogous to a programming language providing only a call statement. Since any possible action desired can be embodied in a program, this language has the necessary power; however, it has many practical weaknesses, and no real OS uses such a language.

An extreme alternative is to eliminate the RUN command and make the name of every program a separate command. This approach, which would correspond to procedure calls in a language with no explicit CALL keyword (such as C or Java), may require searching many large file directories for each command. This approach is used in the standard UNIX command interfaces.

A compromise between these methods is to provide a limited set of standard commands, plus a RUN command to invoke additional programs from files. The standard commands cause execution of a system-controlled set of programs, which may be parts of the OS itself or may be programs stored and maintained in a special way. This approach, used in various systems, provides a fixed and visible distinction between system programs and user programs.

In still other cases the OS maintains a basic set of commands, but the user can install additional commands. Thus, some user programs can be accessed using their own name as a command, while others are executed using a RUN command. This technique is used by VMS. Unfortunately, the installation process is complex.

```
UNIX:

     ls                    list current directory
     who                   show who is logged on
     grep "ABC" file1      list lines containing "ABC"
                             from file1
     pr *.txt | lpr        print all files whose names
                             end with "*.txt"
     sh proc1 3 7 "Hi there" execute a command file
                            named "proc1"
     myprog                run a program named "myprog"

MSDOS:

     copy file1 file2      copy file1 to file2
     dir c:                list (current) directory of
                               disk drive C
     del filea.txt         delete a file
     proc1                 execute a command file named
                              "proc1.bat"
     myprog               run a program named
                            "myprog.exe" or "myprog.com"

VMS:

     cc myprog /list=listfile
                         compile a C program with
                                 selected options
     run myprog           run a program named
                                 "myprog.exe"
     define filename $logname
                          define an alternate name for a
                          file
     set default [newdir.proj1]
                         set a new default directory
     type file1.pas;3   display a file on the screen
     @proc1             execute a command file named
                         "proc1.com"
```

Figure 3-7: Some Typical Commands

Common Commands

Command languages that provide a fixed or initial set of commands normally include commands to access the programs considered most important in the total computing environment. These programs vary widely. In a typical general-purpose computing environment, commands will probably exist for at least the following activities:

- Display a list of the files in a directory
- Copy, rename, or delete files
- Run editing programs to create and edit text files
- Compile and link programs in various languages
- Display information about the system or the user's environment

A particular system may support many additional commands. Although common operations occur in most systems, there is little similarity between the names or argument formats used. For example, a command to change the name of a file would take the following form on several well-known systems:

```
CP/M: REN newname oldname
MS-DOS: RENAME oldname newname
UNIX: mv oldname newname
VMS: RENAME /OLDNAME=oldname /NEWNAME=newname
CMS: rename oldname * * newname = *
```

or, in IBM JCL:

```
//RENAME   EXEC  PGM=IEHPROGM
//SYSPRINT DD    SYSOUT=A
//SYS1     DD    UNIT=3330, VOL=SER=SYS001
//VOL1     DD    DISP=OLD, UNIT=3330,
VOL=SER=ABC123
//SYSIN    DD    *
RENAME
DSNAME=oldname,VOL=3330=ABC123,NEWNAME=newname
/*
```

Early attempts to establish standard commands and command languages, especially by international researchers, are described in Unger [1975], Beech [1980], and Hopper and Newman [1986]. These efforts met with limited acceptance.

Command Arguments

To provide the necessary information for each specific use, commands may be augmented by arguments. The alternative is a separate command for each possible variation, a highly impractical approach. However, a tradeoff is possible between having many separate commands with few arguments, or a few commands with many arguments. The first method offers concise commands but allows fewer variations, and a large command set is difficult to remember. The second technique is most flexible but may require more typing for each command.

There are two common approaches to the syntax of arguments:

- **Positional arguments**. Positional arguments are a sequence of strings separated by blanks or commas. Their meaning is determined by position, as in subroutines in most programming languages. This method is concise and simple, but the user must remember the rules for each command. It is difficult to leave arguments out. Giving arguments in the wrong order can lead to unexpected-possibly disastrous-results.

- **Keyword arguments**. Each keyword argument is identified by a specific name and may or may not also have a data string. More typing is required, but more variations are possible; and each command, if recorded in a log or command file, is better documented.

An example of a command using positional arguments to invoke a C++ compiler is:

```
CPP  prog1.cpp prog1.obj
```

In this example, the first argument is expected to specify the file containing the source program, while the second argument names the file to receive the object code. If the arguments are accidentally reversed, the source program may be destroyed.

A similar example using keyword arguments is:

```
CPP  /SRC=prog1.CPP  /OBJ=prog1.obj /NOOPTIMIZE
```

Operating systems emphasizing simplicity, such as UNIX, often use positional arguments. Those favoring a more powerful command structure support the keyword approach. Some systems combine elements of both approaches.

If the program knows what types of arguments to expect, another method of obtaining them is to interactively ask the user for the necessary information. If information about expected arguments is stored with each program, the OS can automatically query for missing arguments. Relying exclusively on the dialog approach, however, can slow down system use, and makes it difficult to run programs in non-interactive environments.

-

Command Abbreviation and Completion

If commands are designed to be descriptive, they can be fairly long. It is tedious for experienced users to type frequently used commands in their entirety. Because of this, many command interfaces provide for **command abbreviation**. With this technique, certain abbreviations are recognized to represent complete commands. The OS may define specific abbreviations that will be accepted for selected commands, such as `e` for edit, `dir` for directory, and so on. Alternatively, the command handler may be designed to accept any abbreviation, provided that it forms a unique prefix for exactly one recognized command. This approach is used by VMS; `dir`, `dire`, and `direct` are all acceptable abbreviations for the directory command. However, `di` is ambiguous, since several commands start with these two letters. As a further refinement, in some cases users or the OS can specify which choice should be used for ambiguous abbreviations, such as `e` for edit.

Command abbreviation requires that the user know the full set of potential commands, and may require a lot of searching when this set is large. For this reason it is not used by some standard UNIX shells. Another technique, however, called **command completion**, is used by certain UNIX shells. Introduced by TENEX, the command completion method allows a user to type an abbreviation for a command, followed immediately by a special control character, such as ESC. If the control character is typed, the command handler will determine if the characters typed so far form a valid, unambiguous abbreviation. If so, it will accept the command and display the rest of its name on the terminal. The screen will appear just as though the user had typed the entire command. If the abbreviation matches no command or more than one command, the interface will ring a bell. In this case the user is free to back up or to type additional letters.

A few interfaces have used a variation of command completion in which the interface watches each character typed and automatically completes the command as soon as the user provides a unique abbreviation. However, this technique is usually undesirable because of its high **astonishment factor** --- users do not like to be surprised by sudden and unexpected output while typing a word.

Command Recall and Editing

A few command interfaces retain a record of some number of recently used commands. If desired, users can recall commands from this record, edit them, and use them again. The UNIX C-shell is an example of an interface that supports this mechanism. If a complicated command or short series of commands is to be repeated, the recall mechanism allows the original commands to be reentered, avoiding retyping and the likelihood of error. This facility is especially useful when a long command has been typed with a slight error. Usually, the error is not detected until the entire line has been typed, and it is necessary to start over to correct it. With a recall feature, the almost-correct command can be corrected and reused.

Wild Cards

A different type of abbreviation often assists users in abbreviating file names. This technique, known as **wild cards**, also allows identification of a whole set of files with similar

names by a single character string. In a typical wild card mechanism, a special character such as "*" may be typed to represent any string of characters, including none. Thus the string j*.obj may stand for any (and all) file names which start with j and end with .obj, including jim.obj, joe.obj, j.obj, and so on. Usually a second special character is defined, such as "?," which may be used to stand for exactly one unspecified character.

Wild cards are a powerful and desirable mechanism for users, but they present problems for a command interface. If the command handler interprets the abbreviations, it must search for all possible matches and then present this information to programs in a reasonable form. This can be difficult and time-consuming if there are many matches. Alternatively, each program can interpret wild cards in its own arguments, and perhaps handle them more intelligently using standard subroutines. However, wildcards will then be available only when invoking programs that choose to deal with them.

Quality in Command Languages

Evaluation of the good and bad features of a specific command language is a subjective judgment that can lead to heated arguments among users. However, Schneider et al.[1980] have identified some characteristics which seem desirable in almost any good command interface. These include:

- **Consistency**. Consistency becomes an issue when we ask questions such as: Do the same terms and structures always mean the same thing? Is the same activity usually specified in the same way? Some common commands, such as COPY or LIST, mean very different things in different systems or environments. LIST may mean display a directory on the terminal, display a file on the terminal, print a file on a printer, and so on. Often compilers for various languages are invoked in very different ways, or similar command arguments are handled inconsistently by different commands.

- **Naturalness**. Do the terms and structures used seem "right" to each user? Does the system behave as expected, that is, have a low astonishment factor? Since all users will not agree on naturalness, this goal may be attainable only by systems that allow users to customize their own command interface.

- **Reasonableness**. Examples illustrating reasonableness are shown in Figure 3-8. Does the system seem cooperative, impose no rules that seem unnecessary, and provide help and alternative actions? An interface that requires commands to be uppercase only, or in fixed column positions on a line, is not reasonable. A system that demands input the user may not understand, with no obvious means of help or escape, is unreasonable; so is one whose responses are rude or witty rather than helpful and polite.

- **Completeness**. The concept of completeness requires mechanisms available in one context to be usable in other appropriate contexts. Can the user do everything that "seems like it should work"? For example, if wild cards are sometimes allowed in file names, this facility should be available for all commands.

- **Soundness**. The soundness concept imposes a burden on an interface that strives for completeness. It requires that all commands that are syntactically correct will cause predictable and reasonable behavior.

Several additional factors contributing to a good command language are also discussed by Schneider et al.[1980]. One factor is the need to adapt the interface to the level of sophistication of each user. Novice or occasional users may require detailed prompts or menus and extensive help messages. Experienced users prefer brief prompts, concise messages, and the ability to abbreviate commands. The CMS command handler, for example, can be switched between **verbose mode**, which provides detailed messages and prompts, and **brief mode**, in which displayed information is kept short.

One other important concept is the principle of **force**, which says that the effort required to specify a command should be appropriate to the effect of that command. In most cases, commands should require little force. Abbreviations should be accepted, defaults provided for missing arguments, and so on. However, on a UNIX system it is too easy to type

```
rm *
```

which results in deletion of all files (in the current directory). Such a drastic result should require more force. One solution is used by VMS, which requires file deletion commands to include the entire filename. For example:

```
delete myprog.pas;7
```

is the form necessary to delete even a single file, and

```
delete *.*;*
```

makes it slightly harder to delete all of them.

In many environments, a drastic request like "delete all files" will also result in a request for confirmation from the OS. These confirmation messages, however, should not be too extreme (see Figure 3-8).

To be fair, we should observe that some of the quality measures we have discussed conflict with each other. Moreover, real consistency is possible only if all parts of the user environment, including all compilers, editors, and other programs invoked by commands, are designed to conform to a common user interface philosophy. The operating system cannot meet this requirement alone. However, the command interface is the most central component. It must establish a standard of consistency and quality to make these attributes a possibility for the total environment.

a. User cannot get further information or escape:

```
DO YOU WANT THE MASTER FILE INVERTED?
>?
INVALID RESPONSE
>no
INVALID RESPONSE
>quit
INVALID RESPONSE
>help
INVALID RESPONSE
>y
INVERTING THE MASTER FILE ...
```

b. System requires excessive confirmation:

```
>delete myfile
DO YOU REALLY WANT TO DELETE MYFILE?
>yes
ARE YOU SURE?
>yes
IF I PROCEED, THE INFORMATION IN MYFILE WILL BE
DESTROYED.  THIS CANNOT BE REVERSED.  SHALL I
PROCEED ANYWAY?
>yes
...
```

c. Rude and unhelpful responses:

```
>print file1 on lpr2
THERE'S NO SUCH PRINTER, YOU DUMMY!
```

d. Error detected, no further explanation:

```
>run prog1
BUS ERROR
```

Figure 3-8: Some Examples of Unreasonableness

3.6 RESPONSE LANGUAGES

As mentioned earlier, an operating system response language is a language by which an OS communicates information back to the user. This information comprises various messages, either in response to a user's direct request, or to provide the user with information about the current conditions and activities within the system. The types of responses that need to be considered by an OSRL are discussed here.

Messages

A command interface is a two-way street. Besides listening to the user, the system must return messages for various reasons. The interface may assist in the processing of messages from other interactive programs as well, helping to assure a consistent treatment.

Messages from the system serve a number of purposes. The following are the most frequent types:

- **Prompt messages** let the user know the OS is waiting for instructions or information. If they are not provided, the user may be unsure if the system is ready for input, or what type of input is expected.

- **Help messages** provide the user with information to help decide what to do next.

- **Progress messages** tell the user that a long activity is still progressing, and possibly how much has been done. Relatively long activities, such as compiling, can benefit the user with periodic progress messages. They provide reassurance that the work is progressing and may aid in estimating how much longer it will continue.

- **Termination messages** tell the user that the current activity has finished. This category includes error messages.

Each of these categories plays an important role in an effective command interface. Help facilities, which are especially valuable and require careful consideration, are discussed in the next section.

Help Messages

Help messages provide information on request about what can be done and how to do it. Help information is vital to the confused user who needs guidance. However, it can be verbose and should not be imposed on the user who doesn't need it. Some guidelines that may apply to a good help system are:

- Most help should be available on request, and successive requests should lead to more details.

- Some help should always be available when input is requested, ideally in any environment, and should correctly explain the current options to the user.
- The "standard" amount of help should be adjustable to give more to the inexperienced user and less to the experienced user.

A good example of a help facility that supports these principles is the hierarchical help system of VMS. In this system, typing the command HELP with no arguments causes a list of available commands to be displayed, as shown in Figure 3-9. This list is generally adapted to the user's current environment; for example, a list of editing commands may appear if help is requested while an editor is in use.

If the user types HELP followed by a specific command name as an argument, an explanation of that command is displayed, in addition to a list of possible arguments, variations, or related topics. The user may obtain more information on these topics by typing HELP with additional arguments.

```
DIRECTORY

Provides a list of files or information about a file or
group of files.

Format:

DIRECTORY [file-spec[,...]]

Additional information available:

Parameters  Command_Qualifiers
/ACL       /BACKUP    /BEFORE     /BRIEF    /BY_OWNER
/COLUMNS    /CREATED
/DATE      /EXCLUDE   /EXPIRED    /FILE_ID  /FULL
/GRAND_TOTAL
/HEADING   /MODIFIED  /OUTPUT     /OWNER    /PRINTER
/PROTECTION
/SECURITY  /SELECT    /SINCE      /SIZE     /TOTAL
/TRAILING   /VERSIONS
/WIDTH
Examples

DIRECTORY Subtopic?
```

Figure 3-9: HELP on the VMS System

Help systems can often make effective use of a menu style, especially in environments that support rapid screen updating. An example is found in later versions of MS-DOS (see Figure 3-10.) In this example, a request for help from the normal command level results in a display list of all available commands. Users can select a specific command by moving the cursor to its name. A summary of the meaning of each selected command appears immediately at the bottom of the screen. If the user types another special key, more detailed help is displayed for the selected command.

Later versions of MSDOS often provided a help screen in more of a GUI style.

A straightforward approach taken by UNIX systems is to provide help by displaying pages from the system reference manual directly on the screen. This idea is conceptually simple and attractive, but results in compromises in the manual structure that make it of limited usefulness in either displayed or printed form. The problem of designing help facilities that are equally effective whether displayed online or printed as manuals is a difficult but important design problem.

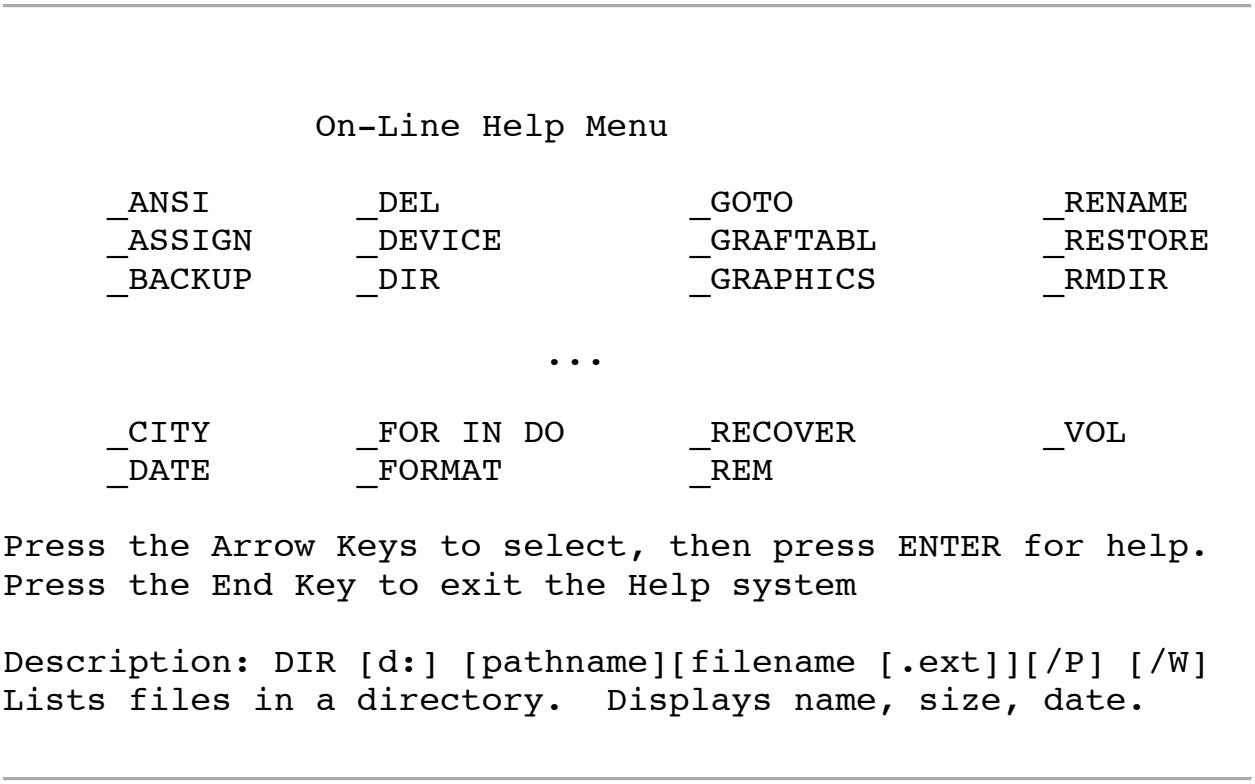

Figure 3-10: HELP on an MSDOS System

3.7 COMMAND FILES

Many command interfaces allow commands to be written in advance and stored in a file. The commands can then be executed by invoking the file by a special shorthand. Such a file has many names, such as submit files (CP/M), exec files (CMS), batch files (MSDOS), command files (VMS), or scripts (UNIX). We will refer to them generically as **command files**.

A command file mechanism provides a way of packaging sets of commands into named files. The file name may then be used as a command or command argument to invoke the entire set of commands at once. Command files are often thought of as batch files, but in a well-designed interface they can be much more powerful: command files can be a tool for constructing programs, and these programs, quite unlike batch jobs, can interact with the user as they progress. The commands in a command file, as well as the programs invoked by those commands, can receive input from either the terminal or the command file itself, and can have their output directed either to the terminal or to another destination.

A powerful command file facility can offer many desirable features. The following list gives some possibilities:

- accept arguments like standard commands, to be substituted in their text like macro arguments.
- allow command files to invoke other command files in a nested structure.
- allow input to come from either the terminal or the command file.
- allow optional display of the commands executed on the terminal, and optional logging to a file or printer.
- allow comments in command files for documentation purposes.
- allow the user to intervene when an error is detected, and then resume the command file at a suitable point.

The ability of command files to pass input to programs makes them a powerful mediator of prepackaged communication between users and all programs. Use of a command herald can play the same role here as in the terminal handler. Features such as those listed above can be made available to all programs. This mechanism is illustrated by the version of the DOS-99 operating system discussed by Mooney [1979]. A list of capabilities available through the command herald structure on this system (called **immediate commands**) is shown in Figure 3-11. In the system outlined by the figure, the ampersand (&) serves as a command herald. It is recognized in any interactive context, and the characters immediately following are given special meaning. The system uses three modes. In addition to normal (interactive) and command file modes, escape mode allows temporary interactive input while command files are being processed. This was especially useful for error correction.

```
COMMAND HERALD:  &

COMMANDS USEFUL IN BOTH COMMAND FILES AND INTERACTIVE
MODE:

&;    Begins a comment line
&<RET> Continues to a new line without telling the program
&S    Suppress logging
&R    Resume logging
&&    Enter a single ampersand

COMMANDS USEFUL IN COMMAND FILE AND ESCAPE MODES ONLY:

&n    Substitute the nth parameter passed to the command
file
&\    Toggle between command file mode and escape mode
&\n   Switch to escape mode for exactly n lines
&*    Resume command file mode
&#    Terminate the current command file (only one if
nested)
&!    Terminate all command files
&I    Ignore errors
&E    Switch to escape mode if errors are found
```

Figure 3-11: Immediate Commands in DOS-99

3.8 COMMAND VS. PROGRAMMING LANGUAGES

The notion of a command file makes possible the use of control flow statements, such as branches, conditionals, and loops, in a sequence of commands, allowing command languages to take on many of the aspects of a programming language. In many cases, such command languages can be used to write useful programs without using a programming language at all. A major example of such programs "interpreted" by the command processor is the UNIX Shell. Figure 3-12 contains an example of a program written for the C shell, a standard shell supplied with most UNIX systems.

This program generates an appropriate greeting depending on the current time of day. The first line ensures that the C shell will be invoked to process the script. The date command is first executed, which determines the current date and time. This information is filtered by a command called awk to extract the hour. A suitable phrase is selected depending on the hour, and a simple editor is used to place this phrase at the end of a file containing the "message of the day," a message which is displayed for all users when they log on. Similar facilities are provided in the EXEC processor of CMS, and in VMS DCL.

```
#! /bin/csh -f
set file=/tmp/med_$$
cat <<'fin' >$file
g/Good morning!/d
g/Good afternoon!/d
g/Good evening!/d
g/What are you doing on at this time of night?/d
$-1a
'fin'
set time = 'date | awk 'printf $4' | awk -F: 'print
$1''
if ($time < 6) then
    echo 'What are you doing on at this time of night?'
>>$file
else if ($time < 12) then
    echo 'Good morning\!' >>$file
else if ($time < 18>) then
    echo 'Good afternoon\!' >>$file
else
    echo 'Good evening\!' >>$file
endif
cat <<'fin' >>$file
```

Figure 3-12: A UNIX Shell Program

Programmability is also a feature of some older command languages, such as that of MULTICS. Colijn [1976, 1981] has demonstrated the use of this language to solve the Tower of Hanoi problem and compute Ackermann's Function. Each of these are popular programming exercises, often used to illustrate recursion.

The Tower of Hanoi is a puzzle that begins with stacks of rings of varying size on two spindles, with larger rings on the bottom and smaller on top. The goal is to transfer a complete stack to a third spindle by moving one ring at a time, never placing a larger ring on a smaller ring. Ackermann's function is a well-known mathematical function of two parameters which rapidly increases in value.

Colijn's MULTICS programs for these problems are shown in Figure 3-13. Lines beginning with "&" followed by a space are comments. The statement &command_line off suppresses displaying the commands on the screen. ec is the Multics "execute" command for command procedures.

In applications where one programming language is used exclusively, if the language is suitable for interpretive use, it is sometimes possible to make it serve as the command language as well. This has been done, for example, with versions of LISP and BASIC.

3.9 OTHER CLI ISSUES

User Environments

In the course of interacting with a computing system, a user is presented with a user environment that establishes the type of commands or other input that may be meaningful at any given moment. The user environment is frequently changing. At any moment, the user environment is determined by the operating system, the program being executed, and the current conditions within that program.

An important element of the user environment is the name space, which is available to the user at any time. It defines the names that will be recognized by commands and programs for files and other objects. Some command interfaces provide the user with various capabilities to influence this name space. VMS allows the definition of alternate names called logical names for files, which can be reassigned at will to real file names or portions of file names. The UNIX C-Shell provides for the definition of various types of commands as aliases for other commands, allowing abbreviations for commonly used commands and arguments.

Another aspect of the user environment may be the establishment of various options that affect how commands are processed or the behavior of certain programs. A wide variety of options may be maintained by various user interfaces. Examples include the extent of the prompt message that may be displayed between commands (CMS EXEC); the number of previous commands to be saved for possible recall (UNIX C-Shell); default format for printer output (DOS-99); treatment of special terminal input and display (many systems).

Any program that conducts a dialog creates a user environment. The command interface provides a command environment in which commands are the expected input. An interactive editor creates an editing environment with different expectations. Other types of environments may be produced by other interactive programs. Often the overall characteristics of the environment for an interactive session or job can be set by a special command file called a profile, which is executed automatically when the job or session is started.

a) TOWER OF HANOI: This command procedure prints the transfer sequence for the Tower of Hanoi, calling itself recursively as required.

```
&       Hanoi.ec - solve the Tower of Hanoi Puzzle
&
&       Calling sequence:  ec Hanoi n "s" "i' "d"
&       where n=no. of discs, and s, i, and d are the
&       names to be printed for the three stacks.
&
&command_line off
&if [greater &1 1] &then ec Hanoi [minus &1 1] &2 &4 &3
&print move disc &1 from &2 to &4
&if [greater &1 1] &then ec Hanoi [minus &1 1] &3 &2 &4
```

b) ACKERMANN'S FUNCTION: This procedure computes ta value for Ackermann's Function. Three subordinate procedures are used. The main procedure and p0 are concerned principally with setting up a numerical variable. p1 and p2 do the actual calculations.

```
&       Ackermann.ec - compute Ackermann's function
&
&       Calling sequence:  ec Ackermann m n
&                          n+1              if m = 0
&       A(m,n)  =          A(m-1,1)         if m>0, n=0
&                          A(m-1,A(m,n-1))  if m>0, n>0
&command_line off
value&set_seg whocares
ec p1 &1 &2
value&dump A
&command_line on
```

```
p0.ec:  &command_line off
        value$set A $1
```

```
p1.ec:  &command_line off
        &if [nequal &1 0] &then ec p0 [plus &2 1]
        &if [nequal &1 0] &then &quit
        &if [ngreater &2 0]
        &then ec p2 &1 [minus &2 1]
        &else ec p1 [minus &1 1] 1
```

```
p2.ec:  &command_line off
        ec p1 &1 &2
        ec p1 [minus &1 1] [value A]
```

Figure 3-13: MULTICS Command Language Programs

The notion of different environments raises several issues:

- Does the command interface behavior, such as prompts or help facilities, suit the current environment?
- Is it possible to easily change between environments?
- How can command structures and names be kept as consistent as possible even in different environments?
- Is it possible or desirable to have only one environment in which system commands, file editing, or other activities are equally possible at any time?

The possibility of a single environment has been explored on systems as diverse as MVS, Macintosh, and various LISP programming environments.

To some extent these issues transcend the design of an OS command interface alone. They can be resolved only by a unified philosophy that establishes a recommended format for all types of user interaction. A system with such a unified philosophy uses a consistent approach in its command interface, standard editor, and other interactive system programs, and recommends the same philosophy to application programs as well. The philosophies of UNIX, VMS, or Macintosh are markedly different. A word processor that fits comfortably into one of these environments would not be well suited for another without significant changes.

Filters

In some operating systems, programs that follow certain conventions may be usable as filters (programs that read a data file and produce a modified version, which may be processed in turn by a series of other programs, as in an assembly line). Filters play an important role in the UNIX philosophy. They are well matched to the UNIX communication structure called "pipes," which allow output from one program to be directly routed into the input of another concurrently running program.

Programs intended for use as filters must adhere to a suitable philosophy in their design. They are expected to read data from a single input stream, and produce a single output stream without extraneous information. For example, a filter to list the files in a directory should produce a listing with no headings or extra lines. These characteristics are not as suitable when programs are to be executed alone at a terminal. Because a filter program may also need to be run interactively, it may be designed to determine whether it is being run as a filter or not, and adjust its behavior as appropriate.

3.10 GRAPHICAL USER INTERFACES

Overview

The CLI interface examples considered so far are based on line-by-line interaction using text characters and a standard character set. A simple menu-based interface may make use of a few screen-oriented operations, such as clearing the screen and explicitly positioning the cursor. At best, however, these interfaces treat the display screen as a small fixed array of characters from a limited character set.

For interactive computer systems that were in widespread use up to the mid 1980s, this model for the user interface represented an appropriate use of the generally available mechanisms. Timesharing systems were usually accessed by video terminals using relatively slow serial connections that could exchange no more than about 100 bytes of information per second. Most terminals were capable of displaying only standard two-dimensional text; but even if the terminals had extensive graphic capabilities, the slow communication speed would have made a more graphical communication model impossible.

Early personal computers replaced timesharing with dedicated single-user interaction, but conventional terminals and serial lines were still used. Although communication speeds became higher, they were still a limitation, and the power of early microprocessors was not sufficient to support a more ambitious model for interaction.

The display capabilities of today's computing hardware, however, have improved dramatically. Workstations such as the PC and Macintosh now fully support the graphic display class described above. These systems manage displays using bit-mapped graphics. This means that the display is represented by memory information that directly describes the appearance of each pixel (picture element). The memory is directly accessible by the CPU for updating; at the same time special hardware scans this memory to construct the actual picture. Memories have also become larger, processors have become more powerful, and greatly improved display devices have been developed. It is no longer unusual to encounter personal systems with full-color displays having resolutions of thousands of pixels per square inch. A great deal of information can be displayed on this type of system in a detailed graphical form.

In addition, most of these systems are now equipped with some form of graphical input device, which allows the positioning of cursors and other objects on the screen by direct pointing. The most common type of device is the mouse; other graphical input devices include joysticks, trackballs, trackpads, special pens, etc.

The problem of improved display models has remained more difficult for remotely-accessed timesharing systems. In this case bitmapped graphics is not possible. The processor is physically distant from the display, and no single memory can be accessible to both. A second problem exists because the central computer has limited knowledge about the characteristics of the display device. In a personal system, a single display device is built in. With a timesharing system, many different kinds of display devices may be connected, varying greatly in their

capabilities. The system cannot make strong assumptions about the capabilities of the display or how to control it.

Again, the problems are being overcome, but more slowly. Very high speed serial communication devices are becoming more common, which can exchange many thousands of bytes per second even over long-distances. Extensive local processing power is becoming more normal in terminal devices; most "terminals" today are actually PCs or workstations running terminal-emulation software. Finally, the diversity of terminals is being addressed by widespread use of standard models. Most terminals and emulators now support a common model for screen-oriented alphanumeric communication based on the VT-100 terminal of Digital Equipment Corporation. More ambitious graphics is supported by the X-Window specification developed by M.I.T.

The graphical input and display capabilities of many current systems have made possible a different model for user communications, the graphical user interface (GUI) model. A GUI represents both objects and action choices as pictures on the screen, called icons, rather than text. Selections are then made using a "point-and-click" approach rather than by typing a command. Other elements commonly associated with GUIs include:

- Multiple windows which can be displayed on the screen, each acting as a smaller, independent screen on which text or data can be displayed;
- Hierarchical menu systems, in which menus can be displayed or hidden by mouse operations or keyboard commands;
- A variety of boxes, controls, indicators, etc. which can be used to select options, display messages, input text, or display specialized information;
- Text display in a variety of sizes and typefaces.

Several typical GUIs containing many of these elements are shown in Figure 3-14.

Origins

The GUI concept was originated by Alan Kay in the 1970s at the Xerox Palo Alto Research Center, and was first implemented on an experimental workstation called the Alto. Soon afterwards Xerox introduced a GUI-based commercial workstation, the Star, intended for office automation. Because the Star was intended to model the normal activities performed at an office desk, the screen represented a desktop, and icons on the screen represented paper-based information files such as folders and documents. Star also introduced the mouse and the concept of "dragging" objects around the screen to accomplish tasks.

The Star was not a commercial success, due apparently to its high cost and rather specialized application, but it became the forerunner of a number of GUIs that appeared in personal systems in the next several years. GUIs were developed as alternative user interfaces for early CP/M systems and for Apple IIs. Apple then introduced a new small computer, the Lisa,

which was GUI based. The future of the GUI approach was finally assured by the introduction in 1984 of the improved successor to the Lisa, the Apple Macintosh.

In the years since the Macintosh appeared, most new PCs and workstations began offering some type of GUI for the user interface. Early examples include the Commodore Amiga, IBM's OS/2, NeXT computers, and the Sun workstations. The IBM-PCs, based originally on the MS-DOS command interface, have now moved entirely to Microsoft Windows. For UNIX-based timesharing and network environments, several GUIs based on X-Windows have been developed, as well as bit-mapped alternatives for Linux workstation environments.

The differences between these many GUIs are often found more in their detailed appearance and behavior (termed "look and feel") than in their functionality. These differences may be seen in the examples of Fig. 3-14, which include Macintosh OSX ("Aqua"), Windows XP, Gnome for Linux, Symbian on a Nokia cellphone, and OSX on the iPhone. A detailed discussion of specific interfaces is beyond the scope of this text.

Figure 3-14: Some Typical GUI Displays

3.11 GUIs vs CLIs

This section considers briefly the relative advantages and tradeoffs of GUIs and of the traditional stream-based interactive interfaces discussed previously. We will consider only the user interface used for invoking programs and communicating with the operating system itself. Somewhat different issues may arise in the interfaces to programs which perform user interaction themselves.

It is often felt that command-line interfaces are preferred by experienced users, while other types such as menu-based or GUIs are most suitable to inexperienced or occasional users. Indeed, the CLI offers power and efficiency, allowing many complex operations to be invoked by relatively terse interactions. However, the command language must be learned by the user, and the commands themselves must be memorized. Mistakes are easily made and not always easily detected. The CLI user remains in apparent control of the system.

The GUI, on the other hand, uses menus and icons to offer a limited set of choices in an intuitively natural way. No language need be learned, and the available choices are always visible (assuming the icons are labeled or clearly understood). On the negative side, much more information must be displayed for a single operation, choices are limited, and it is more difficult to specify complicated options and parameters.

The issue of communication efficiency was originally quite critical. A single line of text such as a command line could be transmitted very quickly over the slowest lines; only a brief prompt (if any) was needed as a message from the system. The time required to display a text-based menu was excessive, even though the input could be shortened. The cost of transmitting all the data required to create or update a GUI display would be completely unacceptable.

These issues may still be significant when remote communication is required, but they have largely disappeared for bit-mapped systems. Most GUI displays can be maintained at nearly instantaneous speeds, and the operation of selection by mouse is quicker and more efficient than typing a command on a keyboard.

The problem of limited operation choices is generally overcome by providing techniques for displaying selected directories in a hierarchical file system. Since most possible operations correspond to files, a complete set can be invoked by these techniques.

The remaining disadvantages of a GUI, from the standpoint of an experienced user, are based on functional limitations. First, it may be very difficult to specify the parameters and options that can be included routinely in command lines. It may be possible to "preselect" a set of files for a program to process, if order is not important, but more complicated options usually require a special dialogue after the program has begun. Concepts like I/O redirection and pipes do not map in any obvious way to a GUI.

The second functional limitation is the lack of a method to support command files, or scripts. The concept of specifying in advance a sequence of operations (with parameters), and placing that specification in a file for repeated use, is foreign to the GUI. One solution that is sometimes used involves "recording" a series of GUI inputs, including both mouse and keyboard

operations, and playing them back when desired. This method provides a basic script, but makes arguments and other advanced features impossible. The only general solution (not usually seen) is to define a command language as an adjunct to the GUI, with command-line equivalents for every possible GUI operation.

If these limitations can be overcome, the GUI model may become more universally adopted by occasional users and professionals alike.

3.12 MULTIMEDIA AND TOUCH-BASED INTERFACES

The Graphical User Interface displays information by means of pictures, possibly detailed and colored, but usually small and static. Input is provided by a conventional keyboard or by selection operations with a simple pointing device. This represented the limitation of display capabilities through the end of the 1980s.

Hardware and software developments in the 1990s began to make feasible a variety of new communication forms, including:

- High quality sound output, and rapidly improving voice recognition systems for sound input;
- Animated graphics and moving video displays in screen windows;
- Pen input devices and software enabling recognition of handwritten text.

These developments lead to the emergence of multimedia systems and handheld systems, as described above. Most workstations today, even low-end models, have full multimedia capabilities. Tablet PCs, Palm and Windows PDAs, and smart cellphones such as iPhone and Blackberry are all examples of the handheld category.

Multimedia systems may support user interfaces in which input commands (as well as system messages) can be spoken. Current sound-recognition systems require a strict format and have a limited vocabulary, but a computer that speaks and listens as well as we do is frequently said to be just around the corner.

Pen and touch input are popular for many mobile devices. These systems are usually too small to support a keyboard, although limited soft keyboards may be used. The primary input device is the stylus or the finger (or thumb). User interfaces for such systems are being built around touch input for commands, and output on a miniature display screen. Sometimes special touch operations, such as "crossing out" for deletion or spreading for zoom, form a new type of user interface paradigm.

-

3.13 THE PROGRAM INTERFACE

The program interface is the means by which an OS provides services to, and communicates with, a running program. It is used by all user programs and often by components of the operating system as well, in particular the command handler. One responsibility of the program interface is to load and set up a program that is ready to begin. In addition, the interface must offer the proper response to normal or abnormal termination. While a program is in execution, the program interface accepts requests for system services and resources, and communicates them to the resource managers of the OS. This structure consists primarily of system calls by which programs communicate with the operating system.

System calls form a special type of procedure call, usually implemented by a distinct machine instruction. In addition to providing an efficient mechanism for invoking OS subprograms, this instruction performs a switch into privileged mode. System calls are viewed by a program as low-level procedures, directly accessible only from assembly language. Most programs written in high-level languages will access these calls through subprogram libraries, where the subprograms carry out the system calls and may perform other functions as well.

A well-designed OS will provide calls for simple, efficient access to all the services it can perform. In addition, it is desirable that many common services are accessible in a similar way in various systems so that programs can be easily moved from one OS to another-such as agreeing on a standard set of system calls.

Initialization and Termination

Part of the responsibility of the program interface is to load and initialize programs for execution. In a simple, single-user OS, this function might be performed by a loader, always resident in memory, which reads program code from a file, copies it directly into a standard area of memory, and transfers it to a standard starting location. More commonly, the program in the file is not an exact copy of memory, and the storage area is not fixed. The loader may have to perform relocation, initialize data areas and so on. If the loader is complex, it may itself be a transient part of the OS, copied into memory when needed.

The program loader works closely with other components of the OS, such as job management and storage management. These modules identify programs to be started, assign the required memory, and invoke the loader to perform initialization. If the program is moved or suspended in the course of execution, the storage manager is responsible for this activity.

Program termination may be invoked by a system call or may occur automatically at the end of the program. Use of a system call allows the program to return information, such as a status code to another process or to the user through the command interface. The responsibilities of the program interface include managing this returned information and returning the program's resources to the available pool. In particular, this responsibility means properly closing any files left open, and reclaiming all memory used for the program's code and data.

-

System Services

The purpose of system calls is to request system services. Many of these request direct use of system resources, such as access to files or input or output on various devices. Other calls request permission for future use of a resource, such as opening a file or allocating a region of memory. Additional system services may provide information, such as system load conditions, time of day, or the present status of an I/O activity. Some of the types of services provided by system calls in typical systems are summarized below:

- USER INTERFACE. User interface services allow a program to find out information about its environment, such as resource limits, standard I/O channels, or command language variables. They also allow a program to access the command arguments specified when the program was invoked, making it possible for a program to operate in a variety of environments. We consider this category in the next section.

- PROCESS MANAGEMENT. Services in this category support management of multiple processes and communication between processes.

- DEVICE MANAGEMENT. These services manage physical I/O devices in ways appropriate to their structure and purpose. They also include various services for communication with remote systems.

- TIME MANAGEMENT. This group includes services to time events and to determine the actual time of day, and to manage explicit timers and timed events.

- MEMORY MANAGEMENT. These services support dynamic allocation of memory and access to memory status information.

- FILE MANAGEMENT. These services manage file space and file access, and may also provide access to I/O devices treated as special files.

- EXCEPTION HANDLING. These are services that establish procedures for the later handling of errors or exceptional conditions that may arise due to program actions or events external to the program.

- SYSTEM MANAGEMENT. Services in this category provide suitably privileged system managers with overall control of system operation, such as scheduling control, setting clocks, initialization and shutdown, adding and removing resources, performance monitoring, or maintaining user records and system logs.

- OTHER CATEGORIES. Other possible types of system calls include program initialization and termination; control of specific I/O devices and device types, such as terminals and printers; special services to support high reliability, real-time system requirements, and distributed operation.

3.14 USER INTERFACE OPERATIONS

A few system calls provided by various OSs directly access and control the user interface. We consider some examples in this section.

The command and arguments by which the program was invoked may be accessed in several different ways. In UNIX the individual arguments (but not the original line) are available as named variables in the user environment. The original command line may also be available uninterpreted.

Some program interfaces allow programs to present lines to the command interpreter to be parsed, or processed, as commands. Although UNIX does not allow a program to call the shell through a system call, any process can create a new process with a private shell that can be used in a similar way. Many OSs, including CP/M, MS-DOS, VMS, and UNIX, provide system calls or other mechanisms to aid in parsing and interpreting file names. A variety of system calls may exist to obtain information about the user environment or the system itself, or to make changes to the environment, such as setting options.

The proposed system calls related to the user interface in IEEE [1985a] provided mechanisms to obtain basic information about the OS; obtain the original command line; obtain arguments one by one from the command line; and determine information about some "standard" input and output channels.

FOR FURTHER READING

A number of books have given comprehensive treatment to user interface issues, focusing especially on interactive command interfaces of various types. An early broad survey was provided by Martin [1973]. A later review of many issues in user interface design is provided by Shneiderman [1987]. Some recent ideas in UI design are presented by Raskin [2000].

The evolution of job control languages is described by Barron and Jackson [1980]. An early icon-based command interface, the Xerox STAR, is described by Smith et al.[1982]. A series of conferences has focused on command language issues and possible standard command languages. Some interesting ideas may be found in the proceedings of these conferences [Unger 1975; Beech 1980; Hopper & Newman 1986].

Quality issues in command languages are considered by Schneider et al.[1980]. Rayner [1980] describes command heralds, and their use in DOS-99 is explained in Mooney [1979]. The use of a command language (the UNIX shell) as a programming language is studied by Dolotta and Mashey [1980]. A user interface structure to support logging and command files is described by Mooney [1982]. Sakamura [1987c] describes the novel "man-machine interface" developed by the BTRON project.

Frank and Theaker [1979b] describe the user interface on the portable MUSS system. A good presentation of the standard UNIX shell is contained in Kernighan and Pike [1984].

Aspects of the KRONOS and MULTICS command languages are discussed by Colijn [1976, 1981].

A good and consistent treatment of the program interface is given by Milenkovi¿[1987]. A standard set of system calls suitable for portable applications is proposed in IEEE [1985a]. The UNIX program interface is described by Bach [1986], and a standard form for this interface is proposed in IEEE [1986]. Other useful treatments of specific program interfaces include Kenah and Bate [1984] for VMS, Popek and Walker [1985] for LOCUS, Ready [1986] for VRTX, Ohkubo et al.[1987] for CTRON, Monden [1987] for ITRON, and many of the presentations in Zarella [1981, 1982, 1984].

REVIEW QUESTIONS

1. Explain briefly three different measures of the quality of a command language.

2. Explain briefly three features of a command language that are important primarily because of their usefulness in command files.

3. State one advantage to use of a RUN command in a command language.

4. State one advantage to use of keyword arguments rather than positional arguments.

5. Why would it be important for a program to know whether it was invoked by a terminal command or a command file?

6. What is the principal difference between submitting a batch job and running a command file?

7. Describe two desirable qualities of a command interface and illustrate each with a specific example.

8. List and describe briefly five characteristics of a command language that are desirable primarily to support command files.

9. List and briefly describe three possible criteria that you can use to evaluate the quality of a command language.

10. List three examples of common system calls unrelated to file management.

-

ASSIGNMENTS

1. Suppose a terminal handler supported none of the capabilities described in Section 3.4. Instead, it passed every typed character directly to the command handler or application program as received, without buffering, interpretation, or echoing. Which of the missing features could be provided by the program? What difficulties would be encountered?

2. Suppose a stream-oriented command interface recognizes commands from a predefined set. The interface is expected to match characters to commands incrementally as typed and signal when an unambiguous prefix has been typed. Design a data structure for representing command names. What steps are required to add new commands to your structure?

3. Consider a command interface as described in Assignment 2, except that any file name may be used as a command. What data structures could be used in this case? What additional difficulties would be encountered?

4. Suppose that a command interface has no knowledge of the specific arguments expected by each command. How could the arguments, actually received, be organized and made available to the program? Consider both positional syntax and keyword syntax.

5. How could a program supply information to the command handler about the specific arguments expected? Suggest a data structure to represent this information.

6. Outline an algorithm for a procedure that determines if a particular file name matches a wildcard specification. Let the character "?" represent any single character, and let the character "*" represent any number of characters (including zero).

7. Consider any specific command interface with which you are familiar. List five characteristics of this interface which have a high astonishment factor.

8. Collect several examples of unreasonableness, inconsistency, and similar problems found in a command interface of your choice. Suggest design changes to correct these problems.

9. List some important ways in which the requirements of an effective on-line help system differ from those of a printed manual. Discuss how you might organize a set of information intended both for on-line help and for printing of manuals on demand.

10. An example of a command file might allow a user to type

```
pascal    file1, file2, file3
```

to compile a series of any number of programs in Pascal. Suppose that such a command should be translated as

```
pascom    file1.pas
pascom    file2.pas
pascom    file3.pas
link file1.obj, file2.obj, file3.obj
```

Suppose further that if any file name is invalid or not recognized, the user should have an opportunity to correct the name without restarting the entire procedure.

List the capabilities required in such a command file system. Select a command interface which supports command files. Can the described command file be written on your system? If not, what capabilities are missing?

11. Investigate how the environment created by your favorite command interface can be customized to suit the preferences of each user. Prepare a profile command file for this system, specifying the environment as completely as possible. List some changes a user might want to make that are not possible.

12. As discussed in Section 3.14, it is sometimes useful for an application program to invoke the command interface to parse or execute program-generated commands. Discuss how this might be done with each of the organizations shown in Figure 3-6.

13. Select a command interface that you have used for a computing system. State and explain two strengths and two weaknesses of this interface. Give concrete examples to support your answers. Concentrate on the command interface itself, *not* on system tools, such as editors or compilers.

14. Suppose you could design a command interface to your own tastes, with no constraints and with no need to keep it similar to any other interface. Outline the characteristics of this interface. What features discussed in the text would you include? Give reasons for your choices.

PROJECT

This section describes a programming project to construct a command interface for a simple multiprogramming operating system. You should read the general description in this text in conjunction with the more specific descriptions given in your project manual. The interface to be constructed is an important component of the multiprogramming executive (MPX) project described later in the text.

The command interface is called COMHAN (short for command handler). Its role is to *read* a command entered via the system console keyboard, to *analyze* the command, and to *execute* the command. The important characteristics of consistency, naturalness, reasonableness, completeness, and soundness are important considerations in the design of COMHAN.

Commands will be added to COMHAN by other projects in this book so the necessary system commands to control the MPX operating system will be present when needed. The initial set of commands which COMHAN should support is as follows:

1. VERSION: Print the version number (including date) for COMHAN and MPX.

2. DATE: Set the date in COMHAN for use later by MPX. This command is also used to display the current date.

3. DIRECTORY: Print a directory of the programs on disk which are loadable under MPX. (Support software is provided to read the directory entries from disk.)

4. STOP: Stop execution of COMHAN and MPX, and returns to the normal operating system of the computer system on which COMHAN is being run.

5. HELP: Print help information about all commands or a specific command.

The above commands should be supported by COMHAN in a manner consistent with the characteristics desired in a command interface. You must decide what must be entered for a command and what the responses of the commands should be. The following questions should be considered while designing COMHAN:

- Are command abbreviations allowed?
- What wild cards (if any) are allowed?
- How is help information requested? When requested, what is the format of the help output?
- What is the display format of COMHAN going to be (a simple menu, a simple prompt, or a fancy selection screen using the display features available)?

Remember that commands will be added to COMHAN in the future. Hence, such commands must be easy to add. This implies that COMHAN should be table-driven using the appropriate data structures to provide for the easy addition of new commands.

The project manual provides the details of the support environment available to you, and of other specific requirements for completing this project within that environment.