**Chapter Two**

* 1. **User Operating System interface**

There are several ways for users to interface with the operating system. Here, we discuss two fundamental approaches. One provides a command-line interface, or command interpreter that allows users to directly enter commands to be performed by the operating system. The other allows users to interface with the operating system via a graphical user interface, or GUI.

* + 1. **Command Interpreter**

Some operating systems include the command interpreter in the kernel. Others, such as Windows XP and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as shells. For example, on UNIX and Linux systems, a user may choose among several different shells, including the *Bourne shell, C shell, Bourne-Again shell, Korn shell,* and others. Third-party shells and free user-written shells are also available. Most shells provide similar functionality, and a user's choice of which shell to use is generally based on personal preference. Figure 2.1 shows the Bourne shell command interpreter

being used on Solaris 10. The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The MS-DOS and UNIX shells operate in this way. These commands can be implemented in two general ways.

In one approach, the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code. An alternative approach -used by UNIX, among other operating systems -implements most commands through system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. Thus, the UNIX command to delete a file

rm file.txt

would search for a file called rm, load the file into memory, and execute it with the parameter file. txt. The function associated with the rm command would be defined completely by the code in the file rm. In this way, programmers can add new commands to the system easily by creating new files with the proper

****

Figure 2.1: The Bourne shell command interpreter in Solaris I 0.

names. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

2.1.2 Graphical User Interfaces

A second strategy for interfacing with the operating system is through a user friendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window and menu system characterized by a desk metaphor. The user moves the mouse to position its pointer on images, or icon, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory-known as a folder-or pull down a menu that contains commands.

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. As a very general rule, many UNIX users prefer command-line interfaces, as they often provide powerful shell interfaces. In contrast, most Windows users are pleased to use the Windows GUI environment and almost never use the MS-DOS shell interface. The user interface can vary from system to system and even from user to user within a system. It typically is substantially removed from the actual system structure. The design of a useful and friendly user interface is therefore not a direct function of the operating system.

**2.2 System Calls**

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly), may need to be written using assembly-language instructions.

Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is for the program to ask the user for the names of the two files. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/0 system calls. Once the two file names are obtained, the program must open the input file and create the output file. Each of these operations requires another system call. There are also possible error conditions for each operation. When the programmers to open the input file, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should print a message on the console (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the termin.al) whether to replace the existing file or to abort the program.

Now that both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors, depending on the output device (no more disk space, printer out of paper, and so on).

Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call). As we can *see1* even simple programs may make heavy use of the operating system.

Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail however. Typically application developers design programs according to an application programming interface (API). The API specifies a set of functions application programmer/ including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Win32 API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux/ and Mac OS X), and the Java API for designing programs that run on the Java virtual machine. Each operating system has its own name for each system call.

Behind the scenes/ the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Win32 function Create Process () (which unsurprisingly is used to create a new process) actually calls the NT Create Process () system call in the Windows kernel. Why would an application programmer prefer programming according to an API rather than invoking actual system calls? There are several reasons for doing so. One benefit of programming according to an API concerns program portability: An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API (although in reality/ architectural differences often make this more difficult than it may appear). Furthermore actual system calls can often be more detailed and difficult to work with than the API available to an application programmer. Regardless/ there often exists a strong correlation between a function in the API and its associated system call within the kernel.

The run-time support system (a set of functions built into libraries included with a compiler) for most programming languages provides a system-call interface that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values. The caller need know nothing about how the system call is implemented or what it does during execution. Rather it need only obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library.

System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call. For example, to get input, we may need to specify the file or device to use as the source, as well as the address and length of the memory buffer into which the input should be read. Of course, the device or file and length may be implicit in the call.

Three general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in *registers.* In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a *block,* or table, in memory, and the address of the block is passed as a parameter in a register (Figure 2.2).

****

Figure 2.2: Passing of parameters as a table.

This is the approach taken by Linux and Solaris. Parameters also can be placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system. Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.