3.2 Operating-System Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. These operating-system services are provided for the convenience of the programmer, to make the programming task easier.

Program execution: The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

I/O operations: A running program may require I/O. This I/O may involve a file or an I/O device. For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

A File-system manipulation: programs need to read and write files. Programs also need to create and delete files by name.

Communications: In many circumstances, one process needs to exchange information with another process. Such communication can occur in two major ways. The first takes place between processes that are executing on the same computer; the second takes place between processes that are executing on different computer systems that are tied together by a computer network. Communications may be implemented via shared memory, or by the technique of message passing, in which packets of information are moved between processes by the operating system.

Error detection: The operating system constantly needs to be aware of possible errors. Errors may occur in the CPU and memory hardware, in I/O devices and in the user program. For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

In addition, another set of operating-system functions exists not for helping

the user, but for ensuring the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

Resource allocation: When multiple users are logged on the system or multiple jobs are running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors. There might also be routines to allocate a tape drive for use by a job.

Accounting: We want to keep track of which users use how many and which kinds of computer resources.

Protection: The owners of information stored in a multiuser computer system may want to control use of that information. When several disjointed processes execute concurrently, it should not be possible for one process to interfere with the others, or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with each user having to authenticate himself to the system, usually by means of a password, to be allowed access to the resources.

3.3 System Calls

System calls provide the interface between a process and the operating system. These calls are generally available as assembly-language instructions.

Certain systems allow system calls to be made directly from a higher level language program, in which case the calls normally resemble predefined function or subroutine calls. As an example of how system calls are used, consider writing a simple program to read data from one file and to 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 window-and-menu 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 similar window can be opened for the destination name to be specified. 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 and may encounter possible error conditions. When the program tries to open the input file, it may find that no file of that name exists or that the file is protected against access. In these cases, the program should print a message (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 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). In an interactive system, another option is to ask the user (a sequence of system calls to output the prompting message and to read the response from the keyboard) 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. Finally, after the entire file is copied, the program may close both files (another system call), write a message (more system calls), and finally terminate

normally (the final system call). As we can see, even simple programs may make heavy use of the operating system.

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, and the address and length of the memory buffer

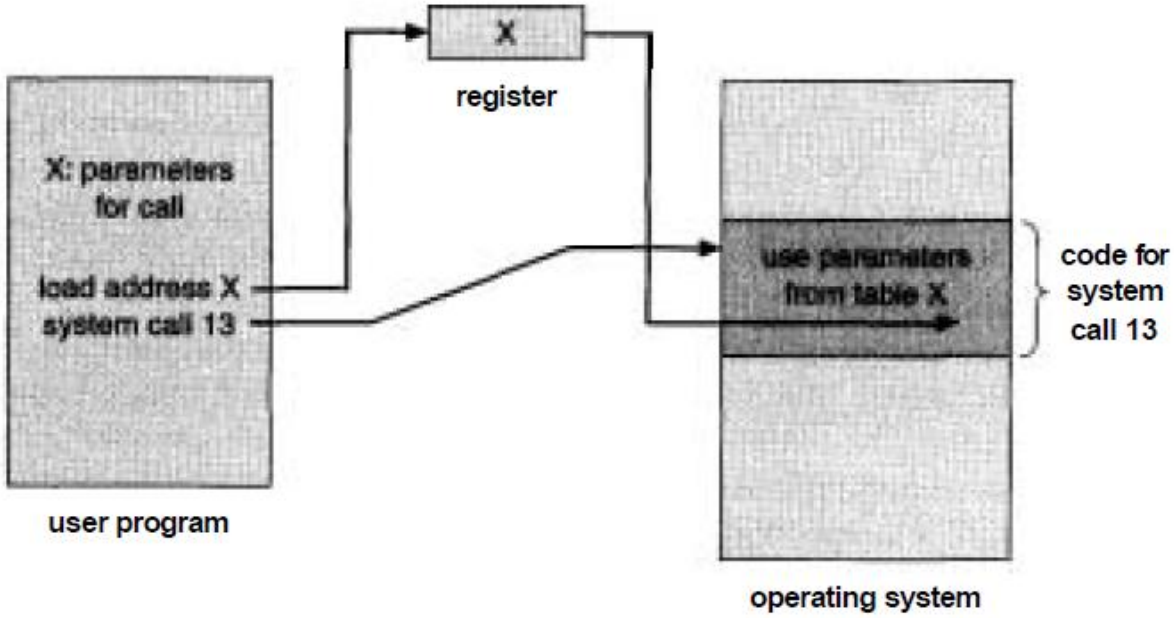

Fig. 3.1 Passing of parameters as a table.

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 3.1). This is the approach

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

System calls can be grouped into five major categories: process control, file management, device management, information maintenance, and communications. Figure 3.2 summarizes the types of system calls normally provided by an operating system.

3.3.1 Process Control

A running program needs to be able to halt its execution either normally (end) or abnormally **(abort).** If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a debugger to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error. In a batch system, the command interpreter usually terminates the entire job and continues with the next job. Some systems allow control cards to indicate special recovery actions in case an error occurs. If the program discovers an error in its input and wants to terminate abnormally, it may also want to define an error level.

Process control
o end, abort
o load, execute
o create process, terminate process

o get process attributes, set process attributes
o wait for time
o wait event, signal event
o allocate and free memory

File management

o create file, delete file
o open, close
o read, write, reposition
o get file attributes, set file attributes

Device management

o request device, release device
o read, write, reposition
o get device attributes, set device attributes
o logically attach or detach devices

Information maintenance

o get time or date, set time or date
o get system data, set system data
o get process, file, or device attributes
o set process, file, or device attributes

Communications

o create, delete communication connection
o send, receive messages
o transfer status information
o attach or detach remote devices

Fig. 3.2 Types of system calls.

A process or job executing one program may want to **load** and execute another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command, the click of a mouse, or a batch command. An interesting question is where to return control when the loaded program terminates. This question is related to the problem of whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program. If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new job or

process to be multiprogrammed. Often, system calls exists specifically for this purpose (create process or submit job).

If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (get process attributes and set process attributes). We may also want to terminate a job or process that we created (terminate process) if we find that it is incorrect or is no longer needed. Having created new jobs or processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time (wait time); more likely, we may want to wait for a specific event to occur (wait event). The jobs or processes should then signal when that event has occurred (signal event).

Another set of system calls is helpful in debugging a program. The trap is usually caught by a debugger, which is a system program designed to aid the programmer in finding and correcting bugs. The MS-DOS operating system is an example of a single-tasking system, which has a command interpreter that is invoked when the computer is started (Figure 3.3(a)). Because MS-DOS is single-tasking, it uses a simple method to run a program and does not create a new process. It , loads the program into memory, writing over most of itself to give the program as much memory as possible (Figure 3.3(b)). It then sets the instruction pointer to the first instruction of the program. The program then runs and either an error causes a trap, or the program executes a system call to terminate. In either case, the error code is saved in the system memory for later use. Following this action, the small portion of the command interpreter that was not overwritten resumes execution. Its first task is to reload the rest of the command interpreter from disk. Once this task is accomplished, the command

interpreter makes the previous error code available to the user or to the next program.

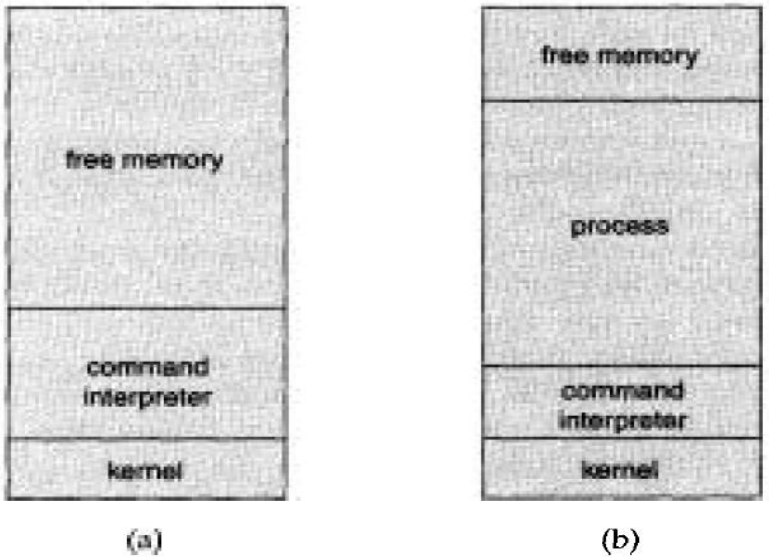

Fig. 3.3 MS-DOS execution. (a) At system startup. (b) Running a program

FreeBSD is an example of a multitasking system. When a user logs on to the system, the shell (or command interpreter) of the user's choice is run. This shell is similar to the MS-DOS shell in that it accepts commands and executes programs that the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure **3.4).** To start a new process, the shell executes a **fork** system call. Then, the selected program is loaded into memory via an **exec** system call, and the program is then executed. Depending on the way the command is issued, the shell then either waits for the process to finish, or runs the process "in the background." In the latter case, the shell immediately requests another command. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files, or through a window-and-menu

interface. Meanwhile, the user is free to ask the shell to run other programs, to monitor the progress of the running process, to change that program's priority, and so on. When the process is done, it executes an **exit** system call to terminate, returning to the invoking process a status code of 0, or a nonzero error code. This status (or error) code is then available to the shell or other programs.

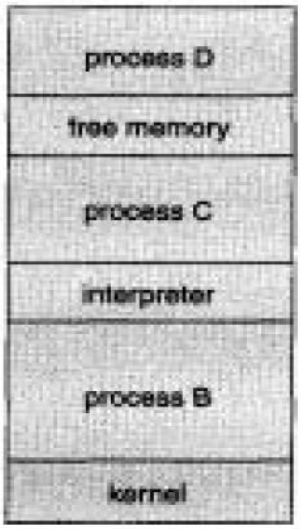

Fig. 3.4 UNIX running multiple programs.

3.3.2 File Management

Once the file is created, we need to open it and to use it. We may also read, write, or reposition. Finally, we need to close the file. We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, we need to be able to determine the values of various attributes, and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on. At least two system calls, get file attribute and set file attribute, are required for this function. Some operating systems provide many more calls.

3.3.3 Device Management

A program, as it is running, may need additional resources to proceed. Additional resources may be more memory, tape drives, access to files, and so on.

If the resources are available, they can be granted, and control can be returned to the user program; otherwise, the program will have to wait until sufficient resources are available. Files can be thought of as abstract or virtual devices. Thus, many of the system calls for files are also needed for devices. If the system has multiple users, however, we must first request the device, to ensure exclusive use of it. After we are finished with the device, we must release it. These functions are similar to the open and close system calls for files. Once the device has been requested (and allocated to us), we can read, write, and (possibly) reposition the device, just as we can with ordinary files.

3.3.4 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current time and date. Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

In addition, the operating system keeps information about all its processes, and there are system calls to access this information. Generally, there are also calls to reset the process information **(get process attributes** and **set process attributes).**

3.3.5 Communication

There are two common models of communication. In the message-passing model, information is exchanged through an interprocess-communication facility provided by the operating system. Before communication can take place, a connection must be opened. The name of the other communicator must be known. Each computer in a network has a host name, such as an IP name, by which it is commonly known. Similarly, each process has a process name, which is translated into an equivalent identifier by which the operating system can refer to it. The **get hostid** and **get processid** system calls do this translation. These identifiers are then passed to the general-purpose **open** and **close** calls provided by the file system, or to specific **open connection** and **close connection** system calls, depending on the system's model of communications. The recipient process usually must give its permission for communication to take place with an **accept connection** call. Most processes that will be receiving connections are special purpose daemons-systems programs provided for that purpose. They execute a **wait for connection** call and are awakened when a connection is made. The source of the communication, known as the *client,* and the receiving daemon, known as a *server,* then exchange messages by **read message** and **write message** system calls. The **close connection** call terminates the communication.

In the shared-memory model, processes use **map memory** system calls to gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process' memory. Shared memory requires that several processes agree **to** remove this restriction. They may then exchange information by reading and writing data in these shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control.

The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Both of these methods are common in operating systems, and some systems even implement both. Message passing is useful when smaller numbers of data need to be exchanged, because no conflicts need to be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared

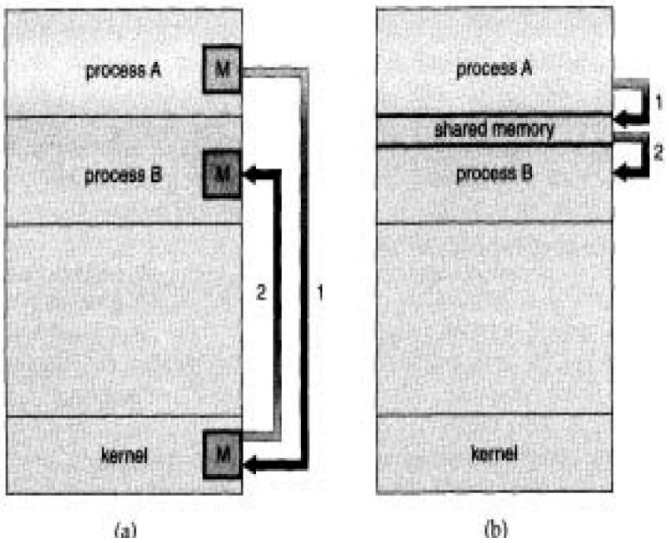

Fig. 3.5 Communications models. (a) Msg passing. (b) Shared memory.

memory allows maximum speed and convenience of communication. Problems exist, in the areas of protection and synchronization. The two communications models are contrasted in Figure **3.5.**