1

INTRODUCTION TO THE OPERATING SYSTEM

In every computer system, the collection of programs that manages each and every piece of hardware and existing software is sometimes referred to as the executive, monitor, supervisor or control program. A much broader term is the Operating System (OS). OS varies widely in purpose and design, from a very simple system supporting a single user on a personal computer (PC) to extremely complex systems supporting many concurrent users with various sophisticated hardware and software resources.

In this chapter, we will describe

(i) what is an operating system
(ii) how the operating system works, and
(iii) how it has evolved as it exists today.

1.1 OS FUNCTIONS

Operating system functions are broadly classified into:

(i) a user/computer interface,
(ii) an efficient resource manager,
(iii) an efficient upgrader,
(iv) an efficient security manager.

1.1.1 OS Acts as a User/Computer Interface

The OS can be intuitively understood and appreciated by anyone who uses the computer. It is a collection of programs that act as an intermediary between a user and a computer hardware. The

programs allow users to share the computer simultaneously, protect data from unauthorized access, and keep dozens of independent devices operating correctly. All this is done at an incredible speed. One basic concept about the OS is that when a computer is executing a user's program, the OS is inactive. Thus, it is often said that the OS provides an environment where other programs can do useful work. This will be more clearly demonstrated in the ensuing discussions.

It is true that there is no completely adequate and universally accepted definition of an OS. The OS is better defined by what it does rather than what it is. The primary goal of a computer is to execute user programs and solve the problems efficiently. Towards this goal, the computer hardware is built. The electronic circuits of a digital computer can recognize and directly execute a limited set of simple instructions in terms of 0s and 1s, which are rarely more complicated than:

1. adding two binary numbers;
2. checking whether a number is 0 or 1; and
3. moving a piece of data from one place to another.

However, it is difficult and tedious to communicate with the computer using this type of language. The hardware alone is very difficult to use. To circumvent the problem, different translator programs and other utilities were gradually developed known as *system programs*.

Now the question arises who is going to control these software and the machine? The answer is certainly the OS, as it implements certain frequently used functions that assist in program creation, management of files, and control of I/O devices. In this context, a computer without an OS may be compared to a bus without a driver and conductor. Such a bus can still be run, but the passengers might fight over who will drive it and where it should go. Similarly, the users of a computer system without an OS might fight over the right to use the computer resources.

As this text proceeds, we will see how crude the hardware is and how much control of the software is required to manage all the system resources, viz. the hardware and the system programs. An OS need not be confined by a single type of hardware. It is capable of hiding low-level details of the real machine and providing high-level services, such as reading a file or printing a record. In short, an OS may provide services in the following areas:

1. **Program creation:** Support programs, such as editors and translators are not considered part of an OS. These programs use the system facilities in the same way as user programs do. Thus, the services of the support programs are actually accessed through the OS.
2. **Program execution:** A number of subsidiary tasks are handled by the OS to execute a program, such as initialization of the relevant I/O devices scrutinizing files, and loading of instructions and data into the main memory.
3. **Access to I/O devices:** Each I/O device requires its own sets of instructions or control signals for operations. The OS takes care of all these. The programmer/user can now think in terms of simple reads and writes. It would be a waste of time and effort if every user had to write his/her own I/O routine.
4. **Controlled access to files:** Here, in addition to the peculiar nature of the I/O device (disk drive, tape drive, etc.), the OS must deal with the file format of the storage medium. Also, in the case of a system with multiple, simultaneous users, the OS can provide protection mechanism to control access to different user files.

5. **Controlled access to any resource:** It is the function of the OS to provide protection to resources from unauthorized users, and resolve conflict arising out of competition of different users in claiming the same resource simultaneously.
6. **Error detection and recovery:** For hardware errors, such as a memory error, device failure, or software errors, such as arithmetic overflow, the attempt to access forbidden memory location, inability of the OS to grant request of an application, the OS softens the error condition, usually with the least impact on the running of the computer system. The response from the OS may be termination of the program that caused the error, retrying the operation, or simply reporting the error condition to the user.
7. **Accounting:** An OS may keep the usage statistics for various resources and monitor the performance parameters such as response time and turnaround time. These are useful for future enhancements. For multiuser system, these can be used for billing purposes.

1.1.2 Interaction with OS

There are two ways in which one can interact with the OS:

- By means of System Calls (SC) in a program (obviously, this is an OS call).
- Directly by means of OS commands.

System calls

System calls provide the interface to a running program and the OS. The user program receives the OS services through a set of system calls. Earlier, these calls were available in assembly language instructions. Nowadays, however, these assembly language instructions are generally avoided for system programming. The use of system calls in C or Pascal programs very much resembles predefined functions or subroutine calls.

As an example of how system calls are used, let us consider a simple program where we have to copy data from one file to another. In an interactive system, the following system calls will be generated by the OS:

1. Prompt messages for inputting two file names and reading it from terminal.
2. Open source and destination file.
3. Prompt error messages in case the source file cannot be opened because it is protected against access, or destination file cannot be created because there is already a file with this name.
4. Read the source file.
5. Write into the destination file.
6. Display status information regarding various read/write error conditions. For example, the program may find that the end of the file has been reached or that there was a hardware failure.
7. The write operation may encounter various errors, depending on the output device (no more disk space, physical end of tape, printer out of paper and so on).
8. Close both files after the entire file is copied.

The following paragraphs help in understanding Program P1.1.

To convert program's logical requests into channel or I/O programs, the file manager must know the location and structure of the file from the directory and the 'file information tables'. The

directory relates logical file names to their physical locations and may give some basic information about the file (owner, date of creation, etc.). The 'file information table' gives additional information, such as file organization, record length and format, and indexing technique, if any.

To begin processing of a file, the file manager searches the directory and locates the appropriate file information table. The file manager may also create buffer areas to receive the blocks being read or written. This initialization procedure is known as *opening* the file. After processing of the file, the buffers and any other work areas and pointers are deleted.

Program P1.1 copies the contents of the source file into the destination file. The program is run (after compilation) by issuing the syntax with two arguments, such as <source file> and <destination file> along with the executable made after compilation, such as a.out in the command prompt (here the UNIX environment is considered). The format of the operation is as follows:

```
a.out <source file> <destination file>
```

The two file names are taken as arguments to the main( ). *First*, the source file is opened using 'OPEN' System Call. *Second*, 'CREAT', the new file with proper access permissions. *Third*, arrange to read the source file and write into the destination file.

The concerned program listing is as follows:

P1.1 Copy from the source to destination file

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

main(int argc, char *argv[])
{
     int finput, foutput;
     int i,j;
     char c;

     /*
checking for the correct no of arguments
*/
     if(argc != 3)
     {
         printf("The usage is 'a.out from_file to_file'");
         exit(1);
     }

/*
     opening the input file
*/
     if((finput = open(argv[1], O_RDONLY, 0444)) == -1)
     {
         printf("opening error for file %s in read only mode", argv[1]);
         exit(1);
     }
```

```
/*
creating the output file with read and write permissions to owner group and others
*/
     if((foutput = creat(argv[2], 0666)) == -1)
     {
         printf("opening error for file %s", argv[2]);
         exit(1);
     }

/*
Now reading from the input file one character at a time until the end of the file is
reached (read() == 0) or an error encountered while reading from the input file (read()
== -1) and writing it to the output file while checking if the character was written to
the output file properly or not.
*/
while((i=read(finput, &c, 1)) > 0)
     {
     if((j=write(foutput, &c, 1)) == 1)
     printf("written output %c\n", c);
     }
}
```

As we can observe, a user program makes substantial use of the OS. All interactions between the user program and its operating environment must occur as a result of a request from the program to the OS.

1.1.3 Operating System Commands

Apart from system calls, users may interact with the OS directly by means of its commands.

For example, if we want to list files or sub-directories in MS DOS, we invoke 'dir' command. In UNIX, the similar command is '1s'. In either case (the system calls or the OS commands), the OS acts as an interface between the user and the hardware of a computer system.

The following paragraphs will help the reader in conceiving the ideas prevalent in UNIX. Then Program P1.2 can be better understood.

Every process (to know about the process, which is a live program, please refer to Chapter 3) except process 0 is created when another process executes the FORK system call. In addition, every process has one parent process but may have many child processes. The OS identifies each process by its process number, called the *process id* (or PID). Process 0 is a special process that is created *by hand* when the system boots. Conventionally, this process is called the *swapper* or *scheduler process*. After forking, a child process (process with PID 1) known as INIT is formed while process 0 becomes the swapper process (Figure 1.1).

Bootstrap sequence

Boot procedures vary from system to system but the goal is common to all machines; to get a copy of the OS into the main memory and to start executing it. This is usually done through various stages and, hence, the name bootstrap.

By pushing of a single button the machine gets instruction to load a bootstrap program in the main memory or directly execute a bootstrap program in ROM.

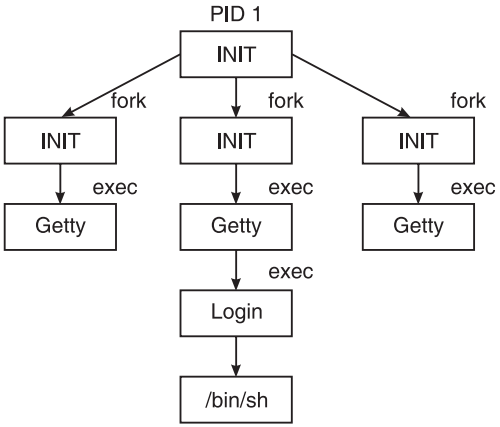

Figure 1.1 The INIT and start sequence.

In UNIX, a bootstrap procedure eventually reads the boot block (block 0) of track zero of a disk and loads it into the main memory. The program contained in the boot-block loads the OS from the file/UNIX. After the OS is loaded in the main memory, the boot program transfers the control to the starting address of the OS and the latter starts executing.

The OS initializes the internal data structures. For example, it constructs the linked lists of free buffers and inodes (internal representation of a file is given by an inode, which contains a description of the disk layout of the file data and other information, such as file owner, access permission and access times), constructs hash queues for buffers and inodes, initializes region structures, page table entries and so on. Among other initialization tasks, it mounts the root file system onto the root ("/") and creates an environment for process 0. Next process 0 invokes the fork algorithm directly from the OS because it is executing in the OS mode. The new process (the INIT) running in the OS mode creates its user-level context by allocating a data region and attaching it to its address space. The INIT process copies code from the OS address space to the new region. This code can now be termed as user-level context of process 1. Thus, process 1 is a user-level process as opposed to process 0, which is an OS-level process that executes in the OS mode. The context for process 1, copied from the OS, consists of a call to execute system calls to execute the program/etc/INIT. Process 1 is commonly called INIT because it is responsible for initialization of a new process. The INIT process runs with a user id of zero, so that it has super user privileges.

Now for each of the terminals to be activated, the INIT process forks a copy of itself and then each child process executes the/etc/getty program (because of invoking the execute system calls). Figure 1.1 explains this.

The getty program naturally

1. sets the terminal speed;
2. outputs a greeting message; and
3. waits for the user to enter his login name.

After getting the login name, the getty executes the program /bin/login, passing its login name as an argument. The login program then looks up the login name in the /etc/passwd file, and

prompts the user for a password. Note that so far INIT, getty and login programs were run with a user id and effective user id of zero (the super user). In addition, the PID does not change when an execution occurs, so all the procedures starting from the forked copy of INIT have the same PID and are as follows:

1. It sets the current working directory to the login directory field from the password file entry. This is done by calling the 'chdir' system calls.
2. Then by calling setgid and setuid system calls, it sets the groupid and userid values specified in the password file entry.
3. Next, when the login program knows who has logged in, it executes the shell program specified by the password file entry or /bin/sh if one is not specified. The settings of the group id and user id are to lower the privilege of the process (if the login name is root, then it is super user) than that of the super user.

The function of a shell in an OS is essentially command interpreter, that is, to get the next command and execute it, which can be seen from Figure 1.2.

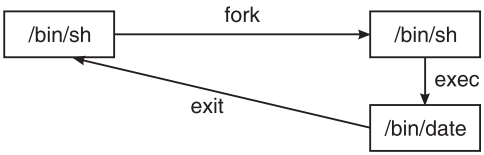

Figure 1.2 Function of Sh.

Some of the functions of the shell are as follows:

1. Shell processes the command-line and extracts the command and argument(s) separately.
2. The shell forks a copy of itself and waits for the child process to terminate.
3. The child uses the path environment variable to look for an executable file having the same name as the command. The environment variable can have any of the three paths:
   (i) /usr1/pabitra/bin,
   (ii) /usr/local/bin or
   (iii) /usr/bin.
   The shell looks in the directory /usr1/pabitra/bin, then in the directory /usr/local/bin, and so on. Assuming the executable program is located in one of these directories, the child process executes the program.
4. When this executable command program is finished (after operating on desired argument(s) if any) it calls exit with its return status which terminates the child process and allows the wait to return to the parent process, as illustrated in the figure.

A simulated environment to demonstrate UNIX-like behaviour of any multiuser OS is noted here. Using the system calls of UNIX and standard C language functions and library calls, this program creates a shell and provides commands, such as 'cat', 'cp', 'date' to the user. However, to run this program, the OS should be UNIX, and an ANSI C compiler is required to compile this program.

In the execution of this program, a login prompt appears with password protection. The information (database) regarding the user account and the password information is stored in a flat file called alogin.dat. In the program, there are many printf statements containing "\033[01;33m"

like code which is provided in the UNIX clones, such as Linux and Sun/Solaris to change the colour of the console text. Here, it is used in the program to make the password input (at the time of login) by the user invisible (by making the foreground and the background colour black). The number of login attempts is also restricted here. With the help of these two features, we are going to demonstrate the security implementations in an OS such as UNIX. Security implementation and restriction of user are also demonstrated by the use of commands such as 'adduser' (user account creation) which is implemented in the program in such a way that it only works when the user logs on the simulated UNIX environment as a 'root'. (*Note:* Being a simulation, the password is not encrypted before storing.)

Command-line inputs are taken into an array, and then is parsed to distinguish between command and its parameters.

Note that the UNIX-like behaviour can be easily implemented by the user-level program design, as shown in Pl. 2.

P1.2 Unix-like behaviour in the program design form

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>

#define TRUE 1
#define FALSE 0
#define SIZE 256

int ttyOpenFlag =TRUE, aLoginFlag =FALSE, aLoginCount;
char vLoginName[10], fLoginName[10], vPassword[10];
char fPassword[10];
FILE *fpLoginFile;

/*********************************************
            Variables required for shell
*********************************************/
char consoleInput[256], com[256], argument1[256];
char argument2[256];
int validCommandLineFlag;
int consoleCode, status;
int i, j, registerSpace[256];

/*********************************************
            variables required for function Cat
*********************************************/
char catBuffer[SIZE];
int catStatus;
int fpCatSource;
```

```
/*********************************************
            variables required for function Cp
*********************************************/
char cpBuffer[SIZE];
int cpStatus;
int fpCpSource, fpCpDestination;

/*********************************************
            variables used by functionDate
*********************************************/
int tm;

/*********************************************
            variables used by functionAddUser
*********************************************/
char addUserName[10], addUserOldPassword[10];
char addUserNewPassword[10];
FILE *fpAddUser;

main( )
{
     system("tput clear");
     while(0)
     {
         printf("Here the simulated INIT process has pid
         %d", getpid());
         alogin();
     }
}

/*********************************************
```

The following function simulates the login process which is executed by INIT when getty() succeeds by successfully opening the tty-line. In our simulated UNIX, it is considered that tty-line open is successful and is denoted by making ttyOpenFlag = TRUE.

```
*********************************************/
alogin()
{
     /*************************************************
         Getty process executes.
         Set process group (setpgrp()) .
         Open tty-line. If the tty-line is successfully opened, then it further proceeds.
*************************************************/
if (ttyOpenFlag == TRUE)
     {
     /*************************************************
     EXECs login program.
```

The user will be offered for login until 'aLoginFlag' becomes true and the number of unsuccessful logins is checked by 'aLoginCount'. When 'aLoginCount' becomes greater than 10, no more offer of login is granted and the system is shutdown.

```
    ************************************************/
printf("\nYou may use login name 'root' and password 'root'\n");
        while((aLoginFlag == FALSE)&&(aLoginCount<=10))
        {
            fpLoginFile = fopen("alogin.dat", "w");
            if (fpLoginFile == NULL)
            {
    /************************************
```

For the sake of this simulation, we are creating the alogin.dat file whenever it is not found or cannot be opened (e.g. when the program is run for the first time or the file was accidentally deleted). However, in a real situation, the system should generate an error.

```
    ******************************************/
            fpLoginFile = fopen("alogin.dat", "a+");
                if (fpLoginFile == NULL)
                {
printf("\nSystem Error. Can't Create ALOGIN.DAT File\n");
                    exit();
}
strcpy(addUserName,"root" );
strcpy(addUserOldPassword,"root");
fprintf(fpLoginFile, "%s %s\n", addUserName, addUserOldPassword);
                printf("\nPassword File is updated.");
                fclose(fpLoginFile);
                fpLoginFile = fopen("alogin.dat", "r");
            }
        printf("\nLogin: ");
            printf("\033[01;33m");
            scanf("%s", vLoginName);
        printf("\033[m");
            printf("Password: ");
            printf("\033[30m");
        scanf("%s", vPassword);
            printf("\033[m");
            aLoginFlag = FALSE;
        while(1)
            {
                if(feof(fpLoginFile))
break;
fscanf (fpLoginFile, "%s %s\n", fLoginName, fPassword);
            if (strcmp(vLoginName, fLoginName) == 0)
                {
```

```
                    if (strcmp(vPassword, fPassword) == 0)
                    {
                        aLoginFlag = TRUE;
                        break;
                    }
            }
            }

            if (aLoginFlag == TRUE) /*login successful*/
            {
printf("\033[01;36mLogin successful\033[m \n");
                /* INIT process exec-ed shell from here */

ashell();
                aLoginFlag = FALSE;
                aLoginCount = 0;
            }
            else
            {
                /* login is not successful */
                printf("\033[01;31mLogin incorrect\033[m
\n");
                aLoginCount ++;
            }
        } /* while(1) loop ends here */

        if (aLoginCount > 10)
        {
            printf("\nNumber of Login Attempts
  exceeded.");
            printf("\nSystem is Locked for possible
        Intrusion..\n");
            printf("\nSending all the process term-end
        signal........");
            printf("\nSimulated UNIX has Shutdown.....");
            kill(0, SIGKILL);
        }
        fclose(fpLoginFile);
    }
}

/*
```

The following function simulates the shell for the simulated UNIX.

```
*/
ashell()
```

```
{
     printf("Simulated INIT process has exec-ed shell
     having pid %d", getpid());
     while (1)
     {
printf("\n$>");
     /* to parent to ignore the death of any child */
     signal(SIGCLD, SIG_IGN);
     parseInput();
     extractCommand();

/************************************************
```

If 'shutdown' command is entered, all the processes, which form the child of the simulated INIT process and the simulated INIT, are killed and the simulated UNIX environment comes to an end.

```
************************************************/
     if (strcmp(com, "shutdown") == 0)
     {
             printf("\nSending all the process term-end
         signal........");
             printf("\nSimulated UNIX has Shutdown.....");
             kill(0, SIGKILL);
     }

/************************************************
```

If the 'logout' command is entered, the program control is taken outside of the 'while' loop and the subsequent login process is initialized. Please check out the control logic flow carefully.

```
************************************************/
     if (strcmp(com, "logout") == 0)
         break;

     if (strcmp(com, "date") == 0)
     {
         if (fork() == 0) functionDate();
         wait(&status);
     }

     if (strcmp(com, "ls") == 0)
     {
         if (fork() == 0) functionLs();
         wait(&status);
     }
```

```
if (strcmp(com, "cat") == 0)
     {
         if (fork() == 0) functionCat();
         wait(&status);
     }
if(strcmp(com, "cp") == 0)
         {
             if(fork() == 0) functionCp();
             wait(&status);
         }

         if(strcmp(com, "clear") == 0)
         {
             if(fork() == 0) functionClear();
             wait(&status);
         }

         if(strcmp(com, "man") == 0)
         {
             if(fork() == 0) functionMan();
             wait(&status);
         }

         if(strcmp(com, "adduser") == 0)
         {
             if(fork() == 0) functionAddUser();
             wait(&status);
         }
     }
}

/************************************************
```

The following four functions, i.e., 'parseInput()', 'extractCommand()', 'extractArgument1()', 'extractArgument2()' are used for parsing the console input of the command-line stored in 'consoleInput[]' array.

parseInput(): This function is used to parse through the command-line entered in the simulated UNIX shell, and register the position of occurrence of space in the command-line (as a demarcation between the command and arguments) to another array 'registerSpace[]'.

extractCommand(): This function is used to extract the command from the whole command-line stored in the array 'consoleInput[]' by the efficient use of variable 'registerSpace[0]'.

Here, 'register[0]' is used as it contains the first demarcation position located in the command-line 'consoleInput[]' and concerned command is transferred to 'com[]' as a string.

extractArgument1(): This function is used to extract the first argument from the 'consoleInput[]' to 'argument1[]' as a string, with the use of 'registerSpace[1]'. Here, the registerSpace[1] is used as it contains the position of the second demarcation in the command-line.

extractArgument2(): This function is used to extract second argument from the command-line. Its use is similar to that of extract Argument1().

```
**********************************************/
parseInput()
{
     /* read form the console */
     i=0;
     while((consoleInput[i] = getchar()) != '\n') i++;
     consoleInput[i]='\0';

     for(i=0,j=0;i<256;i++)
     {
         if(consoleInput[i]==' '||consoleInput[i]== '\0')
         {
             registerSpace[j] = i;
             j++;
         }
     }
}

extractCommand()
{
     for(i=0;i< registerSpace[0]; i++)
     {
         com[i] = consoleInput[i];
     }
     com[i] = '\0';
}

extractArgument1()
{
     for(i=(registerSpace[0]+1);i<registerSpace[1]; i++)
     {
         argument1[i-(registerSpace[0]+1)]=
 consoleInput[i];
     }
     argument1[i] = '\0';
}



extractArgument2()
{
     for(i=(registerSpace[1]+1);i<registerSpace[2]; i++)
     {
         argument2[i-(registerSpace[1]+1)] =
```

```
 consoleInput[i];
      }
      argument2[i] = '\0';
}

/************************************************
```

The following function simulates cat in the simulated UNIX environment when it is invoked by 'cat' command with or without valid argument in the simulated UNIX shell.

Please refer to the 'printf()' statement for understanding the process of execution of the cat function.

Here, there are two instances of cat command. When no argument is supplied with the cat command, the source file of the cat command is taken as standard input device. The corresponding code implemented here has a known bug. If the user wants to quit from this by ^z, then all the processes terminate abruptly.

Otherwise, if an argument is supplied with cat command, then the concerned file is opened and read, and the output is written in the standard output device.

```
************************************************/
functionCat()
{
      printf("%s command is executed by child process
          (pid %d) of parent process (pid %d)",com, getpid(), getppid());
              extractArgument1();
      if (argument1[0] == '\0')
      {
/************************************************
```

The first argument, 0, in the read function, depicts the standard input device and the first argument, 1, in the write function depicts the standard output device.

```
************************************************/
          while((catStatus = read(0, catBuffer,
              sizeof(catBuffer))) > 0)
                  write(1, catBuffer, catStatus);
      }
      else
      {
          if((fpCatSource = open(argument1, 0)) != -1)
          {
              printf("\n");
              while((catStatus = read(fpCatSource,
      catBuffer, sizeof(catBuffer))) > 0)
                  write(1, catBuffer, catStatus);
          }
```

```
        else
     printf("\nERROR !! Unable to open file
        %s......", argument1);
     close(fpCatSource);
     }
     exit(0);
}

/***********************************************
```

The following function simulates the cp in the simulated UNIX environment when it is invoked by 'cp' command in the simulated UNIX shell.

Please refer to the 'printf()' statement for understanding of the process executing the cp function.

```
***********************************************/
functionCp()
{
     printf("%s command is executed by child process
        (pid %d) of parent process (pid %d)", com,
        getpid(), getppid());
     extractArgument1();
     extractArgument2();
     if ((fpCpSource = open(argument1, 0)) == -1)
     {
        printf("ERROR !! Can't open source %s",
        argument1);
        exit(0);
     }
     if((fpCpDestination = creat(argument2, 0666))==-1)
        if((fpCpDestination = open(argument2, 1)) == -1)
        {
                printf("ERROR !! Can't creat or overwrite
            destination file %s", argument2);
                exit(0);
        }
while((cpStatus = read(fpCpSource, cpBuffer,
     sizeof(cpBuffer))) > 0)
        write(fpCpDestination, cpBuffer, cpStatus);


printf("\nCopying from %s to %s is successful. ",
        argument1, argument2);
     exit(0);
}
```

```
functionClear()
{
     printf("%s command is executed by child process
             (pid %d) of parent process (pid %d)", com,
             getpid(), getppid());
         system("tput clear");
         exit(0);
}

/************************************************
```

The following function prints the date in the simulated UNIX environment when it is invoked by date command in the simulated UNIX shell. Refer to the 'printf()' statement for understanding of the process executing the date function.

```
************************************************/
functionDate()
{
     printf("%s command is executed by child process
         (pid %d) of parent process (pid %d)", com,
         getpid(), getppid());
     time(&tm);
     printf("\n %s", asctime(localtime(&tm)));
     exit(0);
}

/************************************************
```

The following function simulates 'ls' in the simulated UNIX environment when it is invoked by the date command in the simulated UNIX shell. Refer to the 'printf()' statement for understanding of the process executing the 'ls' function.

```
************************************************/
functionLs()
{
     printf("%s command is executed by child
     process (pid %d) of parent process (pid %d)",
     com, getpid(), getppid());
     system("ls");
     exit(0);
}
/************************************************
```

The following function simulates 'man' command in the simulated UNIX environment.

```
************************************************/
functionMan()
```

```
{
     printf("%s command is executed by child process
         (pid %d) of parent process (pid %d)", com,
         getpid(), getppid());
     printf("\n\nThe following Commands are available in
         the Simulated UNIX env");
     printf("\nclear date  ls  cat  cp  clear  man
         logout  shutdown\n");
     exit(0);
}

/***********************************************
```

The following function simulates the 'adduser' function in the simulated UNIX environment. This command is executed only when the user has login as a root. No validity checking is implemented in the following code if the user is already in the database.

```
***********************************************/

functionAddUser()
{
     if (strcmp(vLoginName, "root") == 0)
         {
         if((fpAddUser = fopen("alogin.dat","a+"))== NULL)
         {
             printf("\nERROR opening file alogin.dat'...");
         }
         else
         {
             printf("\nEnter Login Name: ");
             scanf("%s", &addUserName);
             printf("Enter Password: ");
             printf("\033[30m");
             scanf("%s", &addUserOldPassword);
             printf("\033[m");
             printf("ReEnter Password: ");
             printf("\033[30m");
             scanf("%s", &addUserNewPassword);
             printf("\033[m");
             if (strcmp(addUserOldPassword, addUserNewPassword) == 0)
             {
                     fprintf(fpAddUser, "%s %s\n", addUserName,
                 addUserOldPassword);
                     printf("\nPassword File is updated.");
             }
```

```
            else
            {
                printf("\nDisparity in password
            rechecking");
            }
        }
        fclose(fpAddUser);
    }
    else
    {
        printf("\nPermission Denied !! To use this
            command, login as root");
    }
    exit(0);
}
```

In Section 2.2, we will see that the above-mentioned interfacing between the user program and the hardware has been done only with the help of a particular type of interrupt (i.e. interrupting the CPU of the computer). See Section 2.5 for a discussion on 'Interrupts' under PC environments.

The reader may note that the OS always deals with allocation and deallocation of the system resources to the active (live) programs called *processes*. These will be discussed in the subsequent chapters. For the time being, we highlight the second function of OS.

1.1.4 Operating System as Efficient Resource Manager

An operating system is a collection of programs designed to manage the system resources, viz. memory, CPU, devices and information (program and data). All these resources are valuable and it is the function of the OS to see that they are used very efficiently. It is also the function of the OS to resolve the conflicts arising out of competition in claiming different resources by various users. The OS as a resource manager must do the following:

1. Keep track of the status of each resource.
2. Enforce policy that determines which process (A process is a computation or program that can be run concurrently with other processes—this concept will be made clearer in Chapter 3) gets what resource, when and how long.
3. Allocate the resource to a process.
4. Reclaim the resource from the process.

The OS acts as memory manager, processor manager, I/O device manager and information manager, which are discussed in the following sections.

Memory management

Memory management has following functions:

1. Keep track of the main memory. What locations are in use and by whom? In addition, what locations are not in use, i.e. free or unallocated?
2. Decide which process gets memory, when it gets it and how much.
3. Allocate the memory when a process requests it and Point 2 from the previous numbered list allows it.

4. Reclaim the resource when the process no longer needs it or has been terminated, for example, in a time-sharing environment, one user's program is executed in a time-slice by CPU until one of the following conditions exists:
   (a) Program is completed.
   (b) An error is detected.
   (c) An I/O operation is met.
   (d) The process is pre-empted.
   (e) In the event of any of the above, the process with the next highest priority is executed. In the event of (a) or (b), the memory may be removed from the process whereas, in the event of (c) or (d), the execution of the process may be suspended (i.e. removed from the main memory to the secondary memory) for the time being.

Processor management

Processor management has following functions:

1. Keep track of the processors and the status of the processes. The program that does this has been traditionally called the *traffic controller* (Saltzer 1974).
2. Decide which process will have a chance to use the processor; the process scheduler chooses one process from all the processes present in the system.
3. Allocate the processor to a process by setting up necessary hardware registers; this is often called *dispatcher*.
4. Reclaim the processor when the process relinquishes processor usage, terminates or exceeds the allowed amount of usage.
5. Regarding the decision as to which user's job or program gets into the system, many factors must be considered. For example, a job that is requesting more main memory or tape drives than the system has, should not be allowed into the system at all.

Here it is worth mentioning that 'job scheduler' program is not included as a part of memory management or device management primarily because the major consideration used in admitting a job into the system is the amount of processor time allocated to it. Further, the record keeping operations for job scheduling and processor scheduler are very similar.

Device management

The device management functions are:

1. Keeping track of the resource (devices, channels, control units). This is typically called the I/O traffic controller.
2. Deciding what is an efficient way to allocate the device. If the device is to be shared among the different processes, then decide who gets it and how much; this is called I/O scheduling.
3. Allocating the device and initiate the I/O operation.
4. Reclaiming the device when the I/O terminates.
5. Assessing has flexible the devices are so that one can decide how they should be allocated. Here, we must note the following factors:
   (a) Some devices cannot be shared (e.g. most of the input devices) and must, therefore, be dedicated to a process.