

## Q. NO:- 1

### a. Explain with an example the system software.

**Ans:-** System software encompasses various components that manage and facilitate the operation of a computer system. One prominent example is the operating system (OS).

Let's consider Linux as an example of system software. Linux serves as the kernel, the core component of the operating system, responsible for managing hardware resources and providing essential services to applications.

### b. Discuss the program loading and execution in systems programming.

**Ans:-** Program loading and execution in systems programming involve several steps:

**1. Compilation:-** The source code of a program is written in a high-level programming language like C or C++. This code is then compiled into machine-readable instructions by a compiler, producing an executable file.

**2. Linking:-** If the program relies on external libraries or modules, the linker combines the compiled code with these libraries to create a single executable file.

**3. Loading:-** When a user initiates the execution of a program, the operating system loads the executable file into memory. This involves allocating memory space for the program's code, data, and stack.

**4. Execution:-** Once the program is loaded into memory, the CPU begins executing its instructions sequentially. The program interacts with the operating system and hardware devices as needed, performing its intended tasks.

**5. Termination:-** When the program completes its execution or encounters an error, it terminates. The operating system deallocates the memory resources used by the program and releases any other system resources it may have acquired.

### c. Explain the following core subsystems of the Linux kernel.

**Ans:-** The core subsystems of the Linux kernel include:-

**1. Process Scheduler:-** The process scheduler is responsible for determining which processes (programs) should run and for how long they should be allowed to execute on the CPU. It manages the CPU's time-sharing among multiple processes, ensuring efficient utilization of system resources and responsiveness to user interactions.

**2. Memory Management Unit (MMU):-** The MMU handles memory allocation and virtual memory management. It translates virtual memory addresses used by processes into physical memory addresses, allowing the operating system to manage memory efficiently. The MMU also implements features like memory protection, which prevents processes from accessing memory regions allocated to other processes.

**3. Virtual File System (VFS):-** The VFS provides an abstraction layer that allows the Linux kernel to support various file systems, such as ext4, NTFS, and FAT32, in a uniform manner. It presents a consistent interface for interacting with files and directories regardless of the underlying file system type. This abstraction simplifies file system management and facilitates interoperability between different types of storage devices and file systems.

d.

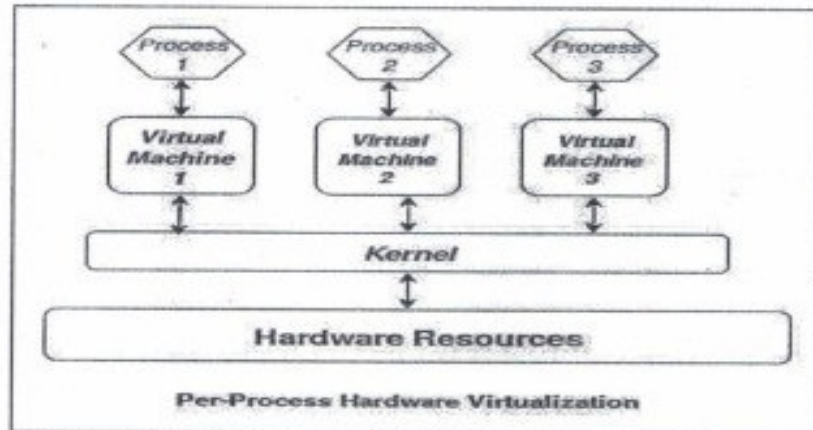


Figure 1

Based on the Figure 1, discuss how the Linux kernel works.

Ans:

## Linux Kernel

- To run a *predefined sequence of instructions*, known as a **program**.
- A program under execution is often referred to as a **process**. Now, most special purpose computers are meant to run a single process, but in a sophisticated system such a general purpose computer, are intended to run many processes simultaneously.
- Any kind of process requires hardware resources such are Memory, Processor time, Storage space, etc.
- The kernel virtualizes the common hardware resources of the computer to provide each process with its own virtual resources. This makes the process seem as it is the sole process running on the machine. The kernel is also responsible for preventing and mitigating conflicts between different processes.

## Q. No:- 2

### a. Describe the Linux Shell with an example.

**Ans:-** The Linux shell is a command-line interface (CLI) that interprets user commands and executes them. It provides a way for users to interact with the operating system by typing commands and receiving immediate feedback. One of the most commonly used Linux shells is Bash (Bourne Again Shell).

Here's an example of using the Linux shell:

```
$ ls -l
```

In this example, the `ls` command lists the contents of the current directory, and the `-l` option specifies the long listing format. When you execute this command in the shell, it displays detailed information about the files and directories in the current directory.

### b. Explain FOUR (4) ways of variable declaration in Shell script.

**Ans:- Four ways of variable declaration in Shell script:**

**1. Implicit Declaration:-** Variables can be declared implicitly by assigning a value to them without explicitly declaring their data type.

For example:

```
myVar="Hello"
```

**2. Explicit Declaration:-** Variables can be explicitly declared using the `declare` or `typeset` built-in commands. This allows specifying attributes such as data type and scope.

For example:

```
declare -i num=5 # Declares an integer variable
```

**3. Using `read` Command:-** Variables can be declared interactively by prompting the user to input a value using the `read` command.

For example:-

```
echo "Enter your name:"  
read name
```

**4. Command Substitution:-** Variables can be assigned the output of a command using command substitution.

For example:-

```
currentDate=$(date +%Y-%m-%d)
```

c. Write a Shell script to get three integers from the user and give output as a sum of three values.

Ans:- Shell script to get three integers from the user and output the sum:-

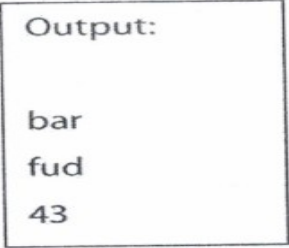
```
#!/bin/bash

# Prompt the user to enter three integers
echo "Enter three integers: "
read num1
read num2
read num3

# Calculate the sum of the three integers
sum=$((num1 + num2 + num3))

# Display the sum
echo "The sum of $num1, $num2, and $num3 is: $sum"
```

d.



```
Output:
bar
fud
43
```

Figure 2

Write a Shell script for the output in Figure 2.

Ans:-

```
#!/bin/bash

# Assigning values to variables
var1="bar"
var2="fud"
num="43"

# Printing the values
echo "$var1"
echo "$var2"
echo "$num"
```

### Q. No:- 3

a.

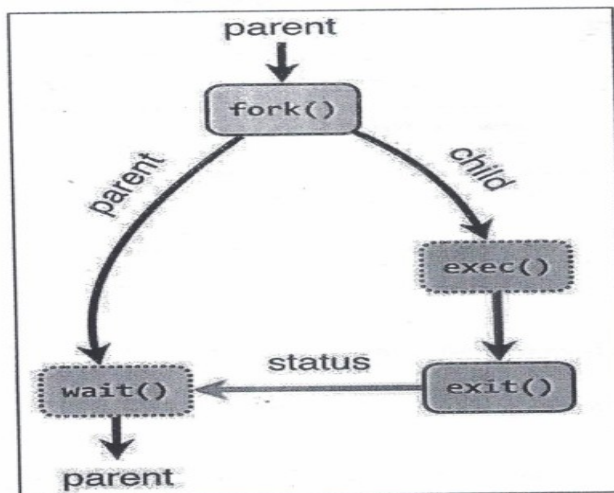


Figure 3

Describe the **FOUR (4)** system calls in Figure 3.

**Ans:-** The following system calls are used for basic process management.

**Fork:-** A parent process uses fork to create a new child process. The child process is a copy of the parent. After fork, both parent and child executes the same program but in separate processes.

**Exec):-** Replaces the program executed by a process. The child may use exec after a fork to replace the process' memory space with a new program executable making the child execute a different program than the parent.

**Exit):-** Terminates the process with an exit status.

**Wait):-** The parent may use wait to suspend execution until a child terminates. Using wait the parent can obtain the exit status of a terminated child.

### b. Explain the two(2) ways of process termination in operating system.

**Ans:-** In an operating system, processes can terminate in two main ways:

**1. Normal Termination:-** A process terminates normally when it completes its execution and exits voluntarily. This could happen when the process reaches the end of its code, explicitly calls an exit system call, or returns from the main function. Upon normal termination, the operating system typically performs cleanup tasks associated with the process, such as releasing allocated memory and closing open files.

**2. Abnormal Termination:-** A process terminates abnormally when it encounters an error or exception during its execution. This could occur due to various reasons such as division by zero, accessing invalid memory, or encountering a critical error condition. When a process terminates abnormally, the operating system may generate an error message or perform other actions depending on the specific circumstances. Additionally, the operating system might attempt to clean up any resources associated with the terminated process to prevent resource leaks or other issues.

c. Output:

```
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!
```

Figure 4

Write a C program that prints the output as shown in Figure 4.

Ans:- Here's a C program that prints "Hello World!" eight times, similar to the output in Figure 4:

```
#include <stdio.h>  
  
int main() {  
    int i;  
  
    // Loop to print "Hello World!" eight times  
    for (i = 0; i < 8; i++) {  
        printf("Hello World!\n");  
    }  
  
    return 0;  
}
```

d. Write a C Program that prints the process you're running, the process another user is running , or all the processes on the system.

Ans:-

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <dirent.h>
#include <string.h>

int main() {
    DIR *dir;
    struct dirent *entry;

    // Open the /proc directory
    dir = opendir("/proc");
    if (dir == NULL) {
        perror("opendir");
        exit(EXIT_FAILURE);
    }

    // Read each entry in the /proc directory
    while ((entry = readdir(dir)) != NULL) {
        // Check if the entry is a directory and represents a process ID
        if (entry->d_type == DT_DIR && atoi(entry->d_name) != 0) {
            char filename[256];
            FILE *fp;
            char buf[1024];

            // Construct the file path for the process status file
            sprintf(filename, "/proc/%s/status", entry->d_name);

            // Open the process status file
            fp = fopen(filename, "r");
            if (fp == NULL) {
                perror("fopen");
                continue;
            }

            // Read the process name from the status file
            while (fgets(buf, sizeof(buf), fp) != NULL) {
                if (strncmp(buf, "Name:", 5) == 0) {
                    printf("Process ID: %s, Name: %s", entry->d_name, buf + 6);
                    break;
                }
            }
        }

        // Close the process status file
        fclose(fp);
    }
}
```

```

}

// Close the /proc directory
closedir(dir);

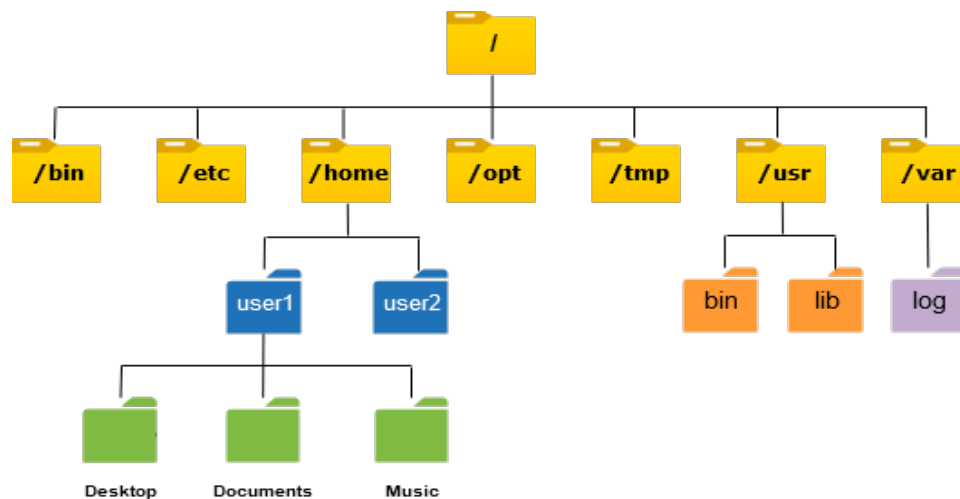
return 0;
}

```

**Q. NO:- 4**

**a. Discuss the concept of files in Linux file system with an example of the files system hierarchy.**

**Ans:-** In the Linux file system, files are fundamental units of data storage that contain information organized into bytes. Each file is identified by a unique name within its directory and is associated with various attributes such as permissions, ownership, size, and timestamps. The Linux file system follows a hierarchical structure, starting from the root directory ("/") and branching into subdirectories. Directories can contain files and other directories, forming a tree-like structure.



**Root directory ("/"):-** The root directory is the top-level directory in the file system hierarchy. It serves as the parent directory for all other directories and files.

**Subdirectories:-** Directories within the file system hierarchy contain files and possibly more subdirectories. For example, the "bin", "etc", "home", and "var" directories are subdirectories of the root directory.

**Files:-** Files contain data organized into bytes. They can be of various types, such as text files, executable files, directories, symbolic links, and device files. For instance, "ls", "mkdir", "passwd", and "hosts" are files within their respective directories.

**Directory structure:-** Directories can have nested structures, containing subdirectories and files. For example, the "home" directory contains subdirectories for different users ("user1", "user2"), each of which contains files specific to that user ("file1.txt", "file2.txt").



**b. Explain the usage of the following initial permissions parameters in Linux file system**

**Ans:-** The initial permission parameters in Linux file systems are used to define the initial access permissions for files and directories when they are created. These parameters are typically used with functions like `open()` or `mkdir()` to specify the permissions for the file or directory. Here's an explanation of each parameter:

- 1. S\_IRUSR:-** This parameter represents read permission for the owner (user) of the file. When set, it allows the owner to read the contents of the file.
- 2. S\_IWUSR:-** This parameter represents write permission for the owner (user) of the file. When set, it allows the owner to modify or delete the file.
- 3. S\_IXGRP:-** This parameter represents execute permission for the group associated with the file. When set, it allows members of the group to execute the file if it is a program or script.
- 4. S\_IWOTH:-** This parameter represents write permission for others (users not in the owner group or not the owner). When set, it allows users who are not the owner or members of the group to modify or delete the file.
- 5. S\_IXOTH:-** This parameter represents execute permission for others (users not in the owner group or not the owner). When set, it allows users who are not the owner or members of the group to execute the file if it is a program or script.

**c. Explain the following library functions for file system**

**Ans:-**

**1. fopen()**

**Purpose:-** `fopen()` is used to open a file and associate it with a stream, allowing further operations like reading or writing to the file.

**Syntax:-** `FILE *fopen(const char *filename, const char *mode);`

**filename:-** A string representing the name of the file to be opened.

**mode:-** A string indicating the mode in which the file should be opened, such as "r" for reading, "w" for writing (creating a new file or overwriting existing content), "a" for appending, etc.

**Return Value:-** If successful, `fopen()` returns a pointer to a `FILE` object representing the opened file. If an error occurs, it returns `NULL`.

**Example:-**

```
FILE *fp = fopen("example.txt", "r");
if (fp == NULL) {
    perror("Error opening file");
    exit(EXIT_FAILURE);
}
```

**2. fread()**

**Purpose:-** `fread()` is used to read data from a file into a buffer.

**Syntax:-** `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`

**ptr:-** A pointer to the buffer where the data will be read into.

**size:-** The size in bytes of each element to be read.

**nmemb:-** The number of elements to read.

**stream:-** A pointer to the `FILE` object representing the file to read from.

**Return Value:-** The number of elements successfully read. If an error occurs or the end of file is reached before any data is read, it returns zero or a value less than `nmemb`.

**Example:-**

```
char buffer[100];
size_t bytes_read = fread(buffer, sizeof(char), 100, fp);
```

### 3. fwrite()

**Purpose:-** fwrite() is used to write data from a buffer into a file.

**Syntax:-** size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

**ptr:-** A pointer to the buffer containing the data to be written.

**size:-** The size in bytes of each element to be written.

**nmemb:-** The number of elements to write.

**stream:-** A pointer to the `FILE` object representing the file to write to.

**Return Value:-** The number of elements successfully written.

**Example:-**

```
char buffer[] = "Hello, world!";
size_t bytes_written = fwrite(buffer, sizeof(char), strlen(buffer), fp);
```

### 4. fflush()

**Purpose:-** fflush() is used to flush the output buffer associated with a stream.

**Syntax:-** int fflush(FILE \*stream);

**stream:-** A pointer to the `FILE` object representing the stream to be flushed.

**Return Value:-** 0 If successful, `fflush()` returns zero. If an error occurs, it returns EOF.

**Example:-**

```
int result = fflush(fp);
```

**d. Write an example of the syntax for the library functions in Question 4(c).**

**Ans:**

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

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
int main() {
```

```
    // Open a file for writing
```

```
    FILE *fp = fopen("example.txt", "w");
```

```
    if (fp == NULL) {
```

```
        perror("Error opening file");
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

```
    // Write data to the file
```

```
    char data[] = "Hello, world!";
```

```
    size_t bytes_written = fwrite(data, sizeof(char), strlen(data), fp);
```

```
    if (bytes_written != strlen(data)) {
```

```
        perror("Error writing to file");
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

```
    // Flush the output buffer
```

```
    int flush_result = fflush(fp);
```

```
if (flush_result == EOF) {
    perror("Error flushing buffer");
    exit(EXIT_FAILURE);
}

// Close the file
int close_result = fclose(fp);
if (close_result != 0) {
    perror("Error closing file");
    exit(EXIT_FAILURE);
}

// Open the file for reading
fp = fopen("example.txt", "r");
if (fp == NULL) {
    perror("Error opening file");
    exit(EXIT_FAILURE);
}

// Read data from the file
char buffer[100];
size_t bytes_read = fread(buffer, sizeof(char), 100, fp);
if (bytes_read == 0) {
    perror("Error reading from file");
    exit(EXIT_FAILURE);
}

// Print the data read from the file
printf("Data read from file: %s\n", buffer);

// Close the file
close_result = fclose(fp);
if (close_result != 0) {
    perror("Error closing file");
    exit(EXIT_FAILURE);
}

return 0;
}
```