EX.NO:1	USAGE OF TOOLS- UNIX SHELL COMMANDS

Unix shell commands are essential tools for interacting with the operating system and managing files and processes efficiently. Here are some common and useful Unix shell commands along with their typical usage:

Unix Shell Commands

1. **ls** - List directory contents
   - ls: List files and directories in the current directory.
   - ls -l: List files in long format (includes details like permissions, owner, size, etc.).
   - ls -a: List all files, including hidden ones (those starting with a dot).
2. **ps** - Report a snapshot of current processes
   - ps: Display information about processes running in the current terminal session.
   - ps aux: Display all processes running on the system with detailed information.
   - ps -ef: Another way to display all processes with more detailed information.
3. **top** - Display Linux tasks
   - top: Interactive process viewer for system monitoring. It shows a list of processes and their resource usage in real-time.
   - top -u username: Display processes owned by username only.
   - Press q to quit top.

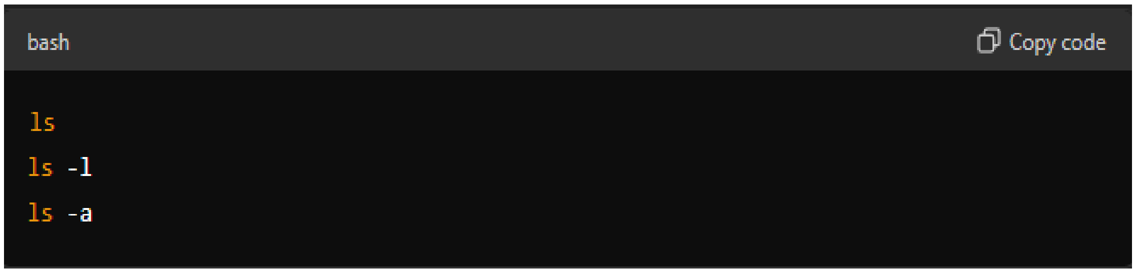
Text Editors

1. **nano** - Simple text editor for Unix-like systems
   - Open a file: nano filename
   - Save a file: Ctrl + O, then Enter
   - Exit nano: Ctrl + X
2. **vi / vim** - Highly configurable text editor
   - Open a file: vi filename
   - Switch to edit mode: Press i for insert mode.
   - Save and exit: Press Esc to exit insert mode, then type :wq and Enter.
3. **gedit** - GNOME text editor for Linux
   - Open a file: gedit filename

   - Save a file: Use the menu or Ctrl + S
   - Close gedit: Use the menu or Ctrl + Q
4. **emacs** - Extensible, customizable text editor and more
   - Open a file: emacs filename
   - Save a file: Ctrl + X, Ctrl + S
   - Exit emacs: Ctrl + X, Ctrl + C

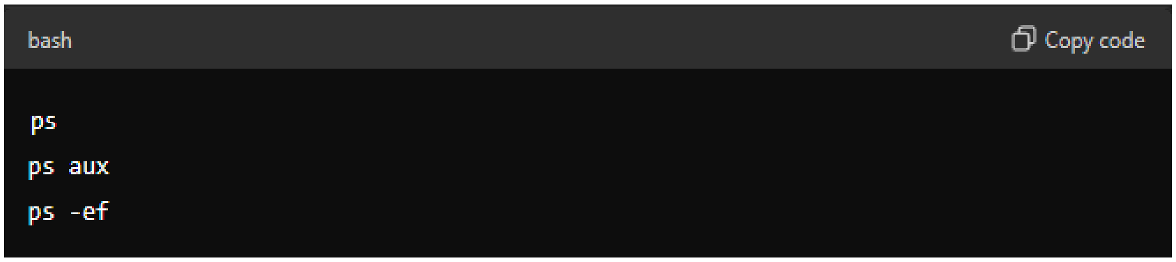
Example Usage Scenarios

- **File Commands**:
  - Listing files in a directory:



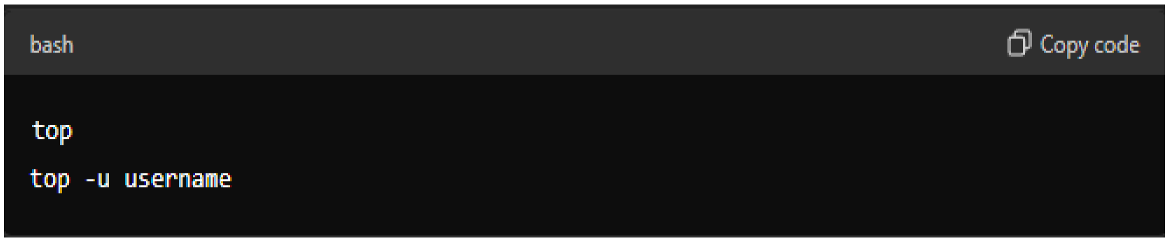
```bash
ls
ls -l
ls -a
```

  - Checking processes:



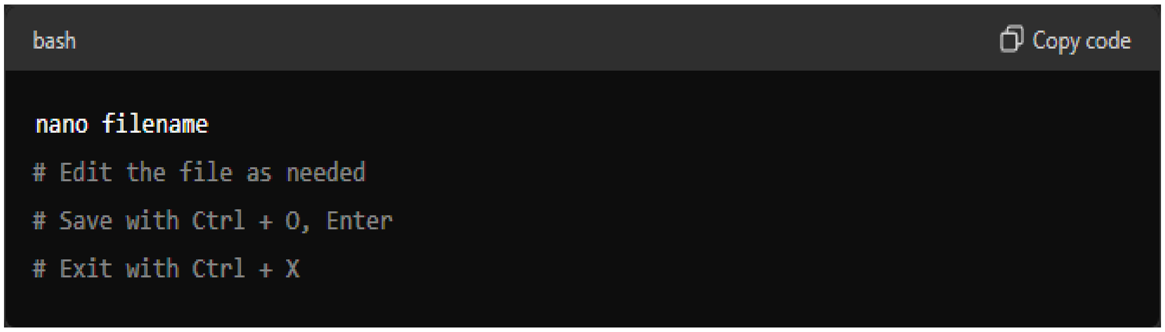
```bash
ps
ps aux
ps -ef
```

  - Monitoring system resources:



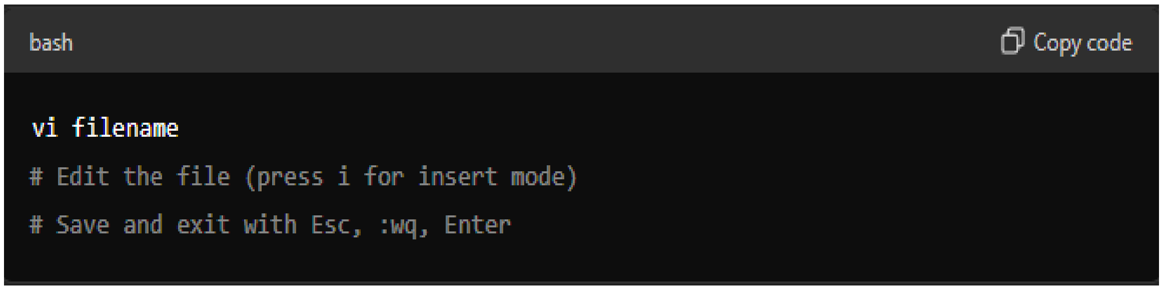
```bash
top
top -u username
```

- **Text Editors**:
  - Editing a file using nano:



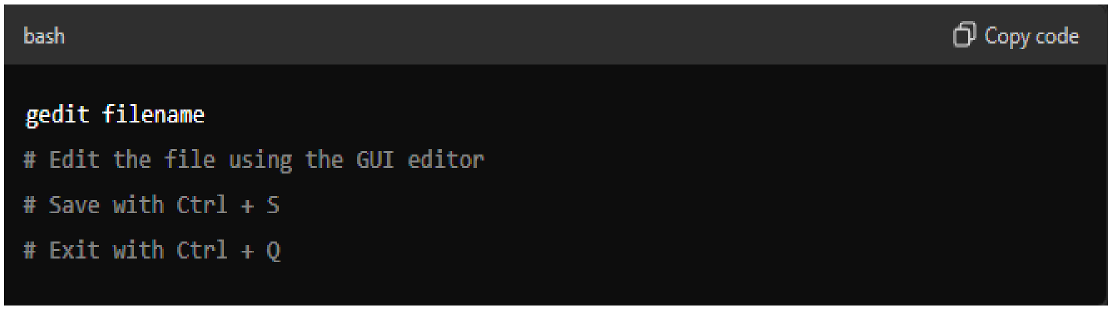
```bash
nano filename
# Edit the file as needed
# Save with Ctrl + O, Enter
# Exit with Ctrl + X
```

- Editing a file using vi:



```bash
vi filename
# Edit the file (press i for insert mode)
# Save and exit with Esc, :wq, Enter
```

- Editing a file using gedit:



```bash
gedit filename
# Edit the file using the GUI editor
# Save with Ctrl + S
# Exit with Ctrl + Q
```

- Editing a file using emacs:

  emacs filename

  # Edit the file

  # Save with Ctrl + X, Ctrl + S

  # Exit with Ctrl + X, Ctrl + C

OTHER UNIX -SELLL COMMANDS:

These commands and text editors are fundamental tools for navigating, managing, and editing files in a Unix-like environment, offering both efficiency and flexibility depending on the task at hand

1. **cd** - Change directory
   - cd directory_name: Change the current working directory to directory_name.

- cd ..: Move up one directory level.
- cd ~: Change to the home directory of the current user.

2. **pwd** - Print working directory
   - pwd: Display the current working directory path.
3. **mkdir** - Make directory
   - mkdirdirectory_name: Create a new directory with the name directory_name.
4. **rm** - Remove files or directories
   - rm file_name: Delete the file file_name.
   - rm -r directory_name: Recursively delete the directory directory_name and its contents.
5. **cp** - Copy files or directories
   - cp source_filedestination_file: Copy source_file to destination_file.
   - cp -r source_directorydestination_directory: Recursively copy source_directory and its contents to destination_directory.
6. **mv** - Move or rename files or directories
   - mv source destination: Move source to destination. Can also be used to rename files or directories by specifying different source and destination names.
7. **grep** - Search for patterns in files
   - grep pattern file_name: Search for pattern in file_name.
   - grep -r pattern directory_name: Recursively search for pattern in all files under directory_name.
8. **chmod** - Change file permissions
   - chmod permissions file_name: Change the permissions of file_name according to permissions (e.g., chmod 755 file_name).
9. **chown** - Change file owner and group
   - chownnew_ownerfile_name: Change the owner of file_name to new_owner.
   - chown -R new_owner:group_namedirectory_name: Recursively change the owner and group of directory_name and its contents.
10. **echo** - Display a line of text
    - echo "Hello, World!": Output Hello, World! to the terminal.
11. **cat** - Concatenate and display files
    - cat file_name: Display the contents of file_name in the terminal.

- cat file1 file2: Concatenate file1 and file2 and display their contents.

12. **head** and **tail** - Output the first or last part of files
    - head file_name: Display the first 10 lines of file_name.
    - tail file_name: Display the last 10 lines of file_name.
    - Use -n option to specify a different number of lines (e.g., head -n 20 file_name
13. **sort** - Sort lines of text files
    - sort file_name: Sort the lines of file_name alphabetically.
    - sort -n file_name: Sort the lines numerically.
14. **wc** - Print newline, word, and byte counts for each file
    - wcfile_name: Output the number of lines, words, and bytes in file_name.

These are just a few examples of Unix shell commands; there are many more available with various options and capabilities to suit different tasks and workflows.

RESULT:

Thus The Unix Shell Commands Was Executed Sussesfully

EX.NO:02[a]	**C PROGRAMMING LANGUAGE REFRESHER ARGUMENTS PASSING WITHOUT POINTER**

AIM:

To refresh some key aspects of C programming related to header files, compilation and linking using GCC, program execution, functions, argument passing, structures, pointers, and file handling.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Declare two integer variables a and b.

Step - 3: Assign values to a and b.

Step - 4: Create a function swap that takes two integer arguments.

Step - 5: Inside the swap function, use a temporary variable to swap the values of the arguments.

Step - 6: Call the swap function with a and b as arguments.

Step - 7: Print the values of a and b after calling the swap function.

Step - 8: End the program.

SOURCE CODE:

```
#include <stdio.h>
Void swap(inta, intb) {
  Int temp = a;
  a = b;
  b = temp;
}
 Int main()
{
  Int a = 10, b = 20;
  swap(a, b);
  printf("Values after swap function are: %d, %d",
         a, b);
  return0; }
```

OUTPUT:

```
Values before swap function are: 10, 20

Values after swap function are: 20, 10
```

RESULT:

The program for argument passing – without pointer is executed successfully

EX.NO:02[b]	C PROGRAMMING LANGUAGE EX REFRESHER ARGUMENTS PASSING – WITH POINTER

AIM:

To refresh some key aspects of C programming related to header files, compilation and linking using GCC, program execution, functions, argument passing, structures, pointers, and file handling.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Declare two integer variables a and b.

Step - 3: Assign values to a and b.

Step - 4: Create a function swap that takes two integer pointers as arguments.

Step - 5: Inside the swap function, use a temporary variable to swap the values of the arguments.

Step - 6: Call the swap function with a and b as arguments.

Step - 7: Print the values of a and b after calling the swap function.

Step - 8: End the program.

SOURCE CODE:

```
#include <stdio.h>
voidswap(int* a, int* b)
{
  inttemp;
  temp = *a;
  *a = *b;
  *b = temp;
}
intmain()
{
  inta = 10, b = 20;
  printf("Values before swap function are: %d, %d\n",
         a, b);
  swap(&a, &b);
  printf("Values after swap function are: %d, %d",
      a, b);
```

OUTPUT:

```
Values before swap function are: 10, 20
 Values after swap function are: 20, 10
```

RESULT:

The program for argument passing – with pointer is executed successfully.

EX.NO:02[c]	C PROGRAMMING LANGUAGE EX REFRESHER STRUCTURES IN C

AIM:

To refresh some key aspects of C programming related to header files, compilation and linking using GCC, program execution, functions, argument passing, structures, pointers, and file handling.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Use the struct keyword to define a structure.

Inside the structure, declare the member variables with their respective data types.

Step - 3: Declare variables of the structure type.

Step - 4: Assign values to the structure members.

Step - 5: Use the dot operator (.) to access and manipulate the structure members.

Step - 6: Print the values of the structure members to verify the initialization.

Step - 7: End the program.

EXAMPLE PROGRAM:

```
#include <stdio.h>
#include <string.h>
struct str1 {
   int i;
   char c;
   float f;
   char s[30];
};
struct str2 {
   int ii;
   char cc;
   float ff;
} var3;

int main() {
   struct str1 var1 = {1, 'A', 1.00, "GeeksforGeeks"};
   struct str1 var2; // Changed to the same type as var1

   // Copy the contents of var1 to var2
   var2 = var1;
   // Initialize var3 with specific values
   var3.ii = 5;
   var3.cc = 'a';
   var3.ff = 5.00;
   // Print contents of var1 and var2
   printf("Struct 1:\n\ti = %d, c = %c, f = %f, s = %s\n",
          var1.i, var1.c, var1.f, var1.s);
   printf("Struct 2:\n\ti = %d, c = %c, f = %f, s = %s\n",
          var2.i, var2.c, var2.f, var2.s); // var2 is now compatible with var1

   // Print contents of var3
   printf("Struct 3:\n\ti = %d, c = %c, f = %f\n",
          var3.ii, var3.cc, var3.ff);
  return 0;
}
```

Output:

```
Struct 1:
    i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 2:
    i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 3
    i = 5, c = a, f = 5.000000
Struct 1:
    i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 2:
    i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 3
    i = 5, c = a, f = 5.000000


Struct 1:
    i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 2:
    i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 3
    i = 5, c = a, f = 5.000000
```

RESULT:

the program for structures in c is executed successfully

EX.NO:02[d]	C PROGRAMMING LANGUAGE EX REFRESHER FILE HANDLING

AIM:

To refresh some key aspects of C programming related to header files, compilation and linking using GCC, program execution, functions, argument passing, structures, pointers, and file handling.

ALGORITHM:

```
Step - 1:  Begin the program.
Step - 2: Start by including the necessary header files.
Step - 3: You need to specify the file name and the mode in which you want to open the file (e.g., “r” for reading, “w” for writing, “a” for appending). If the file cannot be opened, handle the error appropriately.
Step - 4: Use the fprintf() function to write formatted data to the file.
Step - 5: Use the fgets() function to read a line from the file.
Step - 6: Use the fclose() function to close the file.
Step - 7: End the program.
```

SOURCE CODE:

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

int main() {
    // File pointer
    FILE *fptr;

    // Open the file for writing
    fptr = fopen("file.txt", "w");

    // Check if the file was successfully opened
    if (fptr == NULL) {
        printf("The file could not be opened. The program will exit now.\n");
        exit(0);
    } else {
        printf("The file was created successfully.\n");
    }

    // Close the file
    fclose(fptr);

    return 0;
}
```

Output

```
The file is created Successfully.
```

RESULT:

The program for file handling in c is executed successfully.

EX.NO:03	**USAGE OF TOOLS – GCC, GDB, Objdump, Shell scripts**

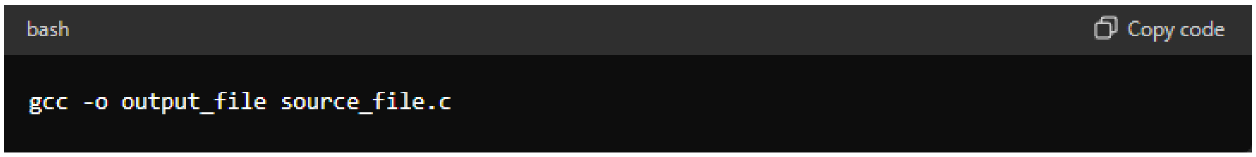
AIM:

Using tools like GCC, GDB, Objdump, and shell scripts can significantly enhance your development and debugging workflow in C programming. Here's a brief overview of how each tool is used:

GCC (GNU Compiler Collection):

GCC is a powerful compiler suite for various programming languages, including C. It's essential for compiling C source code into executable programs.

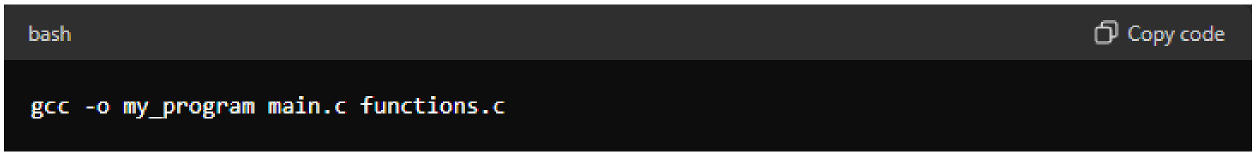
Usage:



```bash
gcc -o output_file source_file.c
```

- `-o output_file`: Specifies the name of the output executable.
- `source_file.c`: The C source file to compile.

Example:

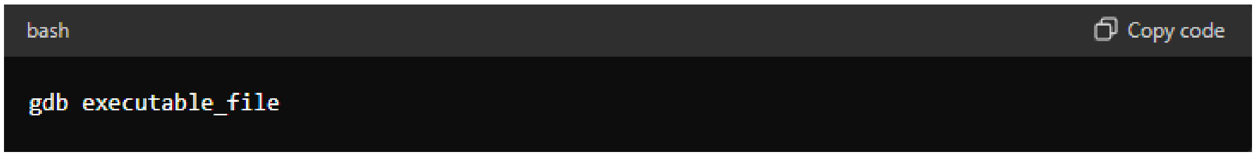


```bash
gcc -o my_program main.c functions.c
```

GDB (GNU Debugger)

GDB is a debugger that allows you to inspect what a program is doing at a given point in time and helps you find and fix bugs.

Usage:



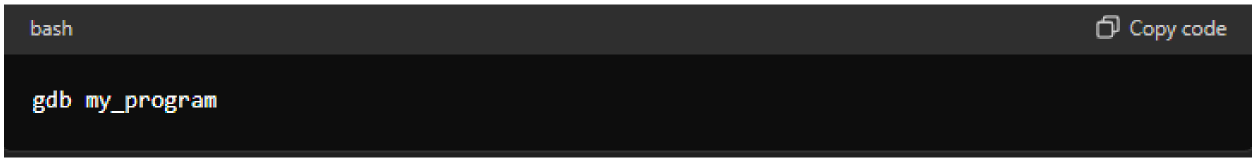
```bash
gdb executable_file
```

- `executable_file`: The compiled executable file you want to debug.

Basic Commands:

- `run`: Start executing the program.
- `break`: Set a breakpoint at a specified line or function.
- `print`: Print the value of a variable.
- `step`: Execute the next line of code.
- `continue`: Resume execution until the next breakpoint.

Example:

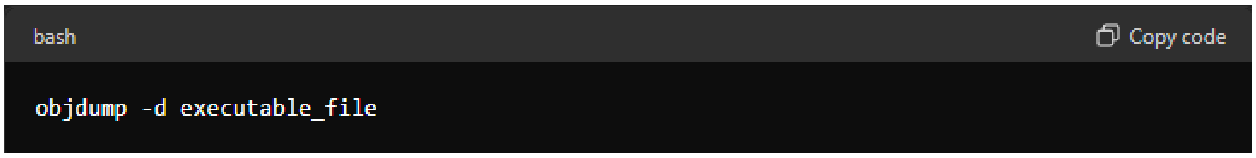


```bash
gdb my_program
```

Objdump

Objdump is a utility that displays information about object files and executable files.

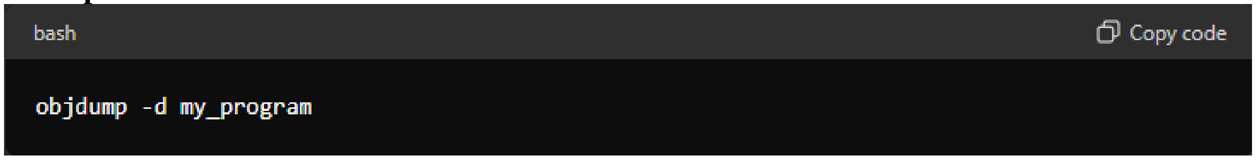
Usage:



```bash
objdump -d executable_file
```

- `-d`: Disassembles the executable file.

Example:



```bash
objdump -d my_program
```

This command will disassemble `my_program` and display its assembly code.

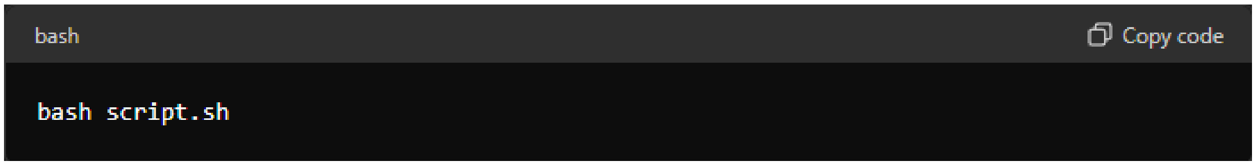
Shell Scripts

Shell scripting allows you to automate tasks and perform various operations in the Unix/Linux shell environment.

Usage:

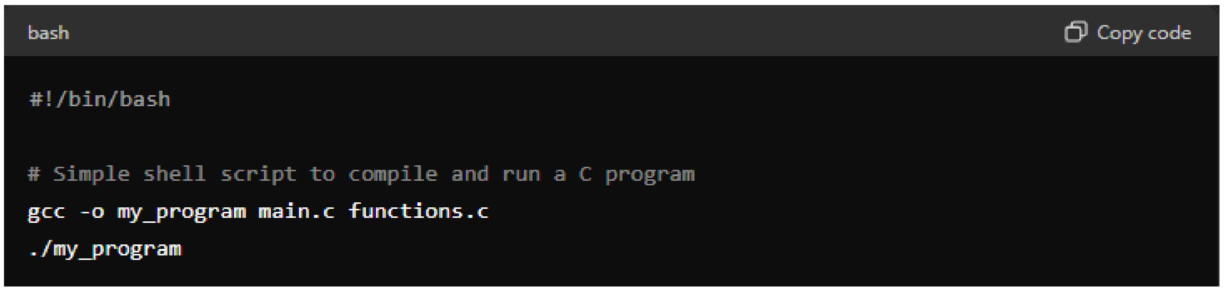
1. **Create a Shell Script:**
   - Create a new file with a `.sh` extension (e.g., `script.sh`).
   - Add the necessary shell commands and script logic inside this file.

2.Run a Shell Script:



```bash
bash script.sh
```

Example:

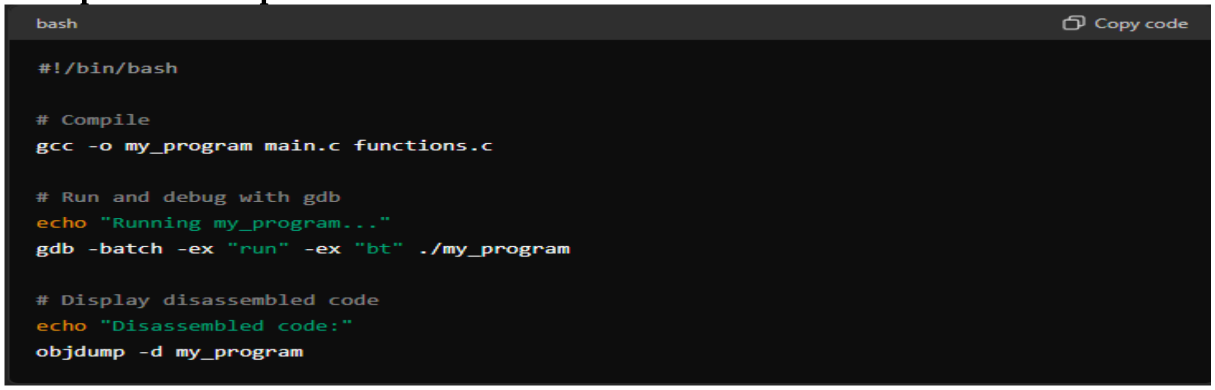


```bash
#!/bin/bash

# Simple shell script to compile and run a C program
gcc -o my_program main.c functions.c
./my_program
```

Combining Tools

You can combine these tools in various ways to streamline your development process:

- Use a shell script to automate the compilation (`gcc`), execution (`./my_program`), and debugging (`gdb`).
- Analyze the disassembled output (`objdump`) to understand low-level details of your program's execution.
- **Example Shell Script:**



```bash
#!/bin/bash

# Compile
gcc -o my_program main.c functions.c

# Run and debug with gdb
echo "Running my_program..."
gdb -batch -ex "run" -ex "bt" ./my_program

# Display disassembled code
echo "Disassembled code:"
objdump -d my_program
```

RESULT:

Thus The Unix Shell Commands Was Executed Sussesfully.

EX.NO:4	SIMPLE STRACE USAGE TO SHOWCASE DIFFERENT INTERFACES (stdlib, system call)

AIM:

Strace is a powerful diagnostic tool used to monitor interactions between a process and the Linux kernel, showcasing various system calls and library calls made during its execution. Here's a simple example to demonstrate how strace can capture and display different types of interfaces: standard library calls (stdlib) and system calls.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Install a C Compiler (if not already installed).

Step - 3: Write the code and Save the Program.

Step - 4: Compile the Program.

Step - 5: Run the Executable

Step - 6: End the program.

EXAMPLE FOR GETPPID():

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
   pid_t ppid = getppid();
   printf("Parent Process ID: %d\n", ppid);
   return 0;
}
```

OUTPUT:

```
Parent Process ID: 350
```

EXAMPLE FOR FORK():

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

```
#include <sys/types.h>
#include <unistd.h>
int main() {
    pid_t p = fork();
    if (p < 0) {
        perror("fork fail");
        exit(1);
    }
    printf("Hello world!, process_id(pid) = %d \n", getpid());
    return 0;
}
```

Output:

```
Hello world!, process_id(pid) = 15901
```

RESULT:

The program for getppid() and fork() system calls executed and verified successfully

EX.NO.05	TOOLS USAGE – ps, pstree, top

AIM:

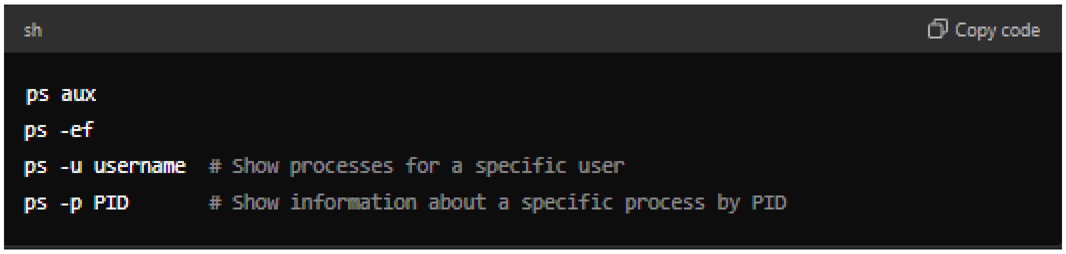
The tools ps, pstree, and top are commonly used in Unix-like operating systems to monitor and manage processes.

1. `ps` (Process Status):

The `ps` command displays information about active processes. It's often used to check the status of processes, find process IDs (PIDs), and see which user is running a process.**Common Options:**

- `ps aux`: Displays a detailed list of all running processes.
- `ps -ef`: Similar to `ps aux`, but uses different options for formatting.

Examples:


sh Copy code

```sh
ps aux
ps -ef
ps -u username  # Show processes for a specific user
ps -p PID       # Show information about a specific process by PID
```

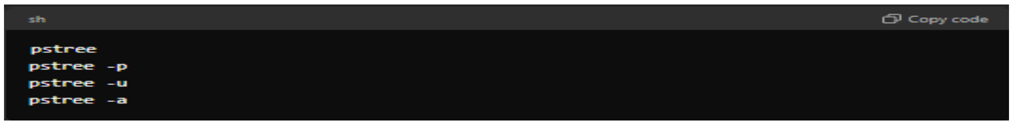
2. `pstree` (Process Tree)

The `pstree` command shows processes in a tree format, illustrating the parent-child relationship between processes. This is useful for understanding the hierarchy and dependencies of processes.

Common Options:

- `pstree`: Display the process tree for all users.
- `pstree -p`: Show PIDs in the tree.
- `pstree -u`: Show usernames in the tree.
- `pstree -a`: Show command line arguments.

Examples:


sh Copy code

```sh
pstree
pstree -p
pstree -u
pstree -a
```

3. `top` (Task Manager)

The `top` command provides a real-time, dynamic view of system processes. It displays CPU and memory usage, process IDs, user names, and more. It's interactive, allowing you to sort and filter processes.

Common Options:

- `top`: Start `top` in interactive mode.
- Press `q`: Quit `top`.
- Press `h`: Display help within `top`.
- Press `P`: Sort by CPU usage.
- Press `M`: Sort by memory usage.

Examples:

```sh
top
```

In `top`, you can interact with the display by pressing various keys:

- `k`: Kill a process.
- `r`: Renice a process (change its priority).
- `1`: Show CPU usage per core.

These tools are essential for system administrators and power users to monitor and manage system performance and resources.

RESULT:

The program for getppid() and fork() system calls executed and verified successfully.

EX.NO.06[a]	**USAGE OF PROCESS CONTROL SYSTEM CALLS TO IDENTITY PROCESS IDENTIFIERS, CREATE PROCESS HIERARCHIES, LAUNCH NEW EXECUTABLES, CONTROL EXIT SEQUENCE OF PARENT AND CHILD PRO-CESSES.**

AIM:

Process control in Unix-like operating systems involves several system calls that allow programs to create, manage, and terminate processes. Here's an overview of some key process control system calls, their usage, and how they relate to process identifiers, process hierarchies, launching new executables, and controlling the exit sequence of parent and child processes.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Identifying Process Identifiers.

Step - 3: Creating Process Hierarchies

Step - 4: Launching New Executables

Step - 5: compile the program

Step - 6: run the program.

Step - 7: End the program.

MUTEX:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
pthread_t tid[2];  // Array to store thread IDs
int counter;       // Shared counter variable
pthread_mutex_t lock;  // Mutex lock for synchronizing threads
void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);  // Lock the mutex before modifying shared resources
    unsigned long i = 0;
    counter += 1;  // Increment the shared counter
    printf("\n Job %d has started\n", counter);
    for (i = 0; i < (0xFFFFFFFF); i++);
   printf("\n Job %d has finished\n", counter);
  pthread_mutex_unlock(&lock);  // Unlock the mutex after work is done
 return NULL;
}
int main(void)
{
    int i = 0;
    int error;
    // Initialize the mutex lock
    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n Mutex init has failed\n");
        return 1;
    }
    // Create two threads
    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0) {
            printf("\nThread can't be created :[%s]", strerror(error));
        }
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

OUTPUT:

```
Job 1 started

Job 1 finished

Job 2 started

Job 2 finished
```

RESULT:

The program for getppid() and fork() system calls executed and verified successfully.

EX.NO.6[b]	**USAGE OF PROCESS CONTROL SYSTEM CALLS TO IDENTITY PROCESS IDENTIFIERS, CREATE PROCESS HIERARCHIES, LAUNCH NEW EXECUTABLES, CONTROL EXIT SEQUENCE OF PARENT AND CHILD PRO-CESSES.**

AIM:

Process control in Unix-like operating systems involves several system calls that allow programs to create, manage, and terminate processes. Here's an overview of some key process control system calls, their usage, and how they relate to process identifiers, process hierarchies, launching new executables, and controlling the exit sequence of parent and child processes.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Identifying Process Identifiers.

Step - 3: Creating Process Hierarchies

Step - 4: Launching New Executables

Step - 5: compile the program

Step - 6: run the program.

Step - 7: End the program.

Pthread:

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
void * threadFunc(void * arg)
{
   std::cout << "Thread Function :: Start" << std::endl;
   sleep(2);
   std::cout << "Thread Function :: End" << std::endl;
   return NULL;
}
```

```
int main()
{
    pthread_t threadId;
  // Create a thread that will function threadFunc()
    int err = pthread_create(&threadId, NULL, &threadFunc, NULL);
    if (err)
    {
   std::cout << "Thread creation failed : " << strerror(err);
    return err;
    }
    else
        std::cout << "Thread Created with ID : " << threadId << std::endl;
    std::cout << "Waiting for thread to exit" << std::endl;
    err = pthread_join(threadId, NULL);
    // check if joining is sucessful
    if (err)
    {  std::cout << "Failed to join Thread : " << strerror(err) << std::endl;
        return err;
    }
  std::cout << "Exiting Main" << std::endl;
return 0;
}
```

Output:

```
Thread Created with ID : 140054120654592
Waiting for thread to exit
Thread Function :: Start
Thread Function :: End
Exiting Main
```

RESULT:

The program for mutex and pthread() was executed and verified successfully

EX.NO.7	FAMILIARITY WITH FILES IN THE / proc / pid / directory

Aim:

The /proc directory in Unix-like operating systems is a virtual filesystem that provides an interface to kernel data structures. It is commonly used to access information about processes, system resources, and hardware. Within the /proc directory, each running process has its own subdirectory named by its process ID (PID). These subdirectories contain files that provide information about the specific process.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Identify the PID of the processes.

Step - 3: Navigate to the /proc directory.

Step - 4: Use the ls command to list the files and directories within the /proc/<PID> directory.

Step - 5: Examine specific files

Step - 6: Read the files

Step - 7: compile and run the program.

EXAMPLE FOR GETPID():

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

int main() {
pid_tpid = getpid();
printf("Process ID: %d\n", pid);
   return 0;
}
```

OUTPUT:

```
Process ID: 1234
```

RESULT:

The /proc/[pid]/ directory was executed and verified successfully

EX.NO.08	(VIRTUAL)ADDRESS OF VARIABLE AND INITIALIZED POINTERS

AIM:

The "address of a variable" refers to the memory address where a variable is stored in the computer's memory. Every variable in a program occupies a certain amount of memory, and this memory location has a unique address.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Type the program and save.

Step - 3: Open Terminal.

Step - 4: Navigate to the Directory.

Step - 5: Compile the Program.

Step - 6: Run the program.

EXAMPLE:

```
#include <stdio.h>
void geeks()
{ int var = 10;
   int *ptr;
   ptr = &var; // Assign the address of a variable to a pointer
 printf("Value at ptr = %p \n", (void *)ptr);
   printf("Value at var = %d \n", var);
   printf("Value at *ptr = %d \n", *ptr);
}int main() {
   geeks();
   return 0;}
```

OUTPUT:

```
Value at ptr = 0x7ffcc0d9438c
Value at var = 10
Value at *ptr = 10
```

RESULT:

The program for initialized pointers was executed and verified successfully.

EX.NO:9	USE OF MALLOC() AND DEMONSTRATION OF PER-PROCESS VIRTUAL ADDRESSES

AIM:

The use of malloc() and how it ties into per-process virtual addresses in C or C++.

UNDERSTANDING MALLOC():

In C and C++, malloc() is a function defined in <stdlib.h> (or <cstdlib> in C++) that dynamically allocates memory during runtime. It stands for "memory allocation". When you call malloc(size), it attempts to allocate a block of memory of size bytes from the heap and returns a pointer to the beginning of that block if successful. If it fails to allocate memory (e.g., due to insufficient memory available), it returns NULL.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Type the program and save.

Step - 3: Open Terminal.

Step - 4: Navigate to the Directory.

Step - 5: Compile the Program.

Step - 6: Run the program.

EXAMPLE:

```
#include <stdio.h>

#include <stdlib.h>

int main() {

   // This pointer will hold the base address of the block created

   int *ptr;

   int n, i;

   // Get the number of elements for the array

   printf("Enter number of elements: ");

   scanf("%d", &n);
```

```
printf("Entered number of elements: %d\n", n);

// Dynamically allocate memory using malloc()

ptr = (int *)malloc(n * sizeof(int));

// Check if the memory has been successfully allocated by malloc or not

 if (ptr == NULL) {

    printf("Memory not allocated.\n");

    exit(0); }

    else {

    // Memory has been successfully allocated

    printf("Memory successfully allocated using malloc.\n");

    // Get the elements of the array

    for (i = 0; i < n; ++i) {

        ptr[i] = i + 1;

    }

    // Print the elements of the array

    printf("The elements of the array are: ");

    for (i = 0; i < n; ++i) {

        printf("%d, ", ptr[i]);}

       printf("\n"); // Add a newline for better output formatting

   }

   free(ptr);

   return 0;
```

```
}
```

OUTPUT:

```
Enter number of elements:7
Entered number of elements: 7
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7,
```

RESULT:

The program for malloc() was executed and verified successfully.

EX.NO:10	TOOLS USAGE- STRACE, FREE,TOP,HTOP,VMSTAT,/PROC/PID/MAPS

AIM:

The usage and purpose of each of these tools commonly used in Linux environments for various system monitoring and debugging tasks.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Start with empty lists for keys and values.

Step - 3: When you need to add or update a pair, first check if the key exists.

Step - 4: If the key exists, update its value. If it doesn't, add a new key-value pair.

Step - 5: To retrieve a value, look up the key. If found, return the value; if not, return -1..

Step - 6: Print all keys and values whenever needed.

Step - 7: compile and run the program.

EXAMPLE FOR MAPS:

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

#define MAX_SIZE 100 // Maximum number of elements in the map

int size = 0; // Current number of elements in the map
char keys[MAX_SIZE][100]; // Array to store the keys
int values[MAX_SIZE]; // Array to store the values

// Function to get the index of a key in the keys array
int getIndex(char key[])
{
    for (int i = 0; i < size; i++) {
        if (strcmp(keys[i], key) == 0) {
            return i;
        }
    }
    return -1; // Key not found
}

// Function to insert a key-value pair into the map
void insert(char key[], int value)
{
    int index = getIndex(key);
    if (index == -1) { // Key not found
        strcpy(keys[size], key);
        values[size] = value;
        size++;
    } else { // Key found
        values[index] = value;
    }
}

// Function to get the value of a key in the map
int get(char key[])
{
    int index = getIndex(key);
    if (index == -1) { // Key not found
        return -1;
    } else { // Key found
        return values[index];
    }
}

// Function to print the map
void printMap()
{
    for (int i = 0; i < size; i++) {
        printf("%s: %d\n", keys[i], values[i]);
    }
}

int main()
{
    insert("Geeks", 5);
    insert("GFG", 3);
    insert("GeeksforGeeks", 7);

    printf("Value of complete Map: \n");
    printMap();

    printf("\nValue of GFG: %d\n", get("GFG"));
    printf("Index of GeeksforGeeks: %d\n", getIndex("GeeksforGeeks"));

    return 0;
}
```

OUTPUT:

```
Value of complete Map:
Geeks: 5
GFG: 3
GeeksforGeeks: 7
Value of GFG: 3
Index of GeeksforGeeks: 2
```

RESULT:

The program for maps was executed and verified successfully.

EX.NO:11	Free memory statistics correlated with malloc(). Number of system calls and malloc() usage.

AIM:

The calloc function in C is used to allocate memory for an array of objects and initializes all bytes in the allocated storage to zero.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Declare a pointer (ptr).

Step - 3: Set the number of elements (n).

Step - 4: Allocate memory dynamically.

Step - 5: Check if memory was allocated

Step - 6: If memory is allocated, Print the values and free the memory.

Step - 7: End the program.

EXAMPLE FOR CALLOC():

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

int main() {

    // This pointer will hold the base address of the block created
    int *ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int *)calloc(n, sizeof(int));
    // Check if the memory has been successfully allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    } else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array (initialize the values)
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;  // Storing values 1 to n
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
        printf("\n");
    }
// Free the allocated memory
    free(ptr);
 return 0;
}
```

OUTPUT:

```
Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,
```

RESULT:

The program for calloc() was executed and verified successfully.

Ex.no:12	Implement a custom memory allocator using system calls.

AIM:

Implementing a custom memory allocator using system calls requires creating a basic memory management system that bypasses the standard malloc() function and directly interacts with the operating system's memory allocation primitives, like brk() and mmap().

Custom Allocator: A simplified memory allocator that provides basic malloc() and free() functionality

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Type and save the code.

Step - 3: Open Terminal.

Step - 4: Navigate to the Directory

Step - 5: Compile the Code

Step - 6: Run the Executable

Step - 7: End the program.

EXAMPLE FOR CUSTOM MEMORY ALLOCATOR:

```
#include <iostream>
#include <vector>
using namespace std;
// Custom memory allocator class
template <typename T>
class myClass {
public:
 typedef T value_type;
 // Constructor
  myClass() noexcept {}
 // Allocate memory for n objects of type T
  T* allocate(std::size_t n) {
   return static_cast<T*>(::operator new(n * sizeof(T)));
    }
  // Deallocate memory
    void deallocate(T* p, std::size_t n) noexcept {
    ::operator delete(p);
    }
   };
    int main() {
    // Define a vector with the custom allocator
    vector<int, myClass<int>> vec;
    for (int i = 1; i <= 5; ++i) {
        vec.push_back(i);
    }

    // Print the elements
    for (const auto& elem : vec) {
        cout << elem << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
1 2 3 4 5
```

RESULT:

The program for custom memory allocation was executed and verified successfully.

EX.NO:13	USER MODE PROGRAMS TO DEMONSTRATE LDE

AIM:

To C programs that demonstrate concepts related to Localized Dynamic Energy (LDE), complete with sample outputs for better understanding.

ALGORITHM:

Step 1: Setting Up the Environment

Step 2: Simulating Load Conditions

Step 3: Implementing the LDE Algorithm

Step 4: Implementing the LDE Algorithm

Step 5: Compile and Run the Program

Run the program:

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define ALLOC_SIZE (1024 * 1024) // 1 MB

#define NUM_ALLOCATIONS 1000

int main() {

    void **allocations = malloc(NUM_ALLOCATIONS * sizeof(void *));

    if (allocations == NULL) {

        perror("Failed to allocate memory for allocations");

        return 1;

    }


    for (int i = 0; i < NUM_ALLOCATIONS; i++) {

        allocations[i] = malloc(ALLOC_SIZE);
        if (allocations[i] == NULL) {
            perror("Failed to allocate memory");
            return 1;
        }
        usleep(10000); // Simulate processing time (10 ms)
    }

    printf("Allocated %d MB of memory.\n", NUM_ALLOCATIONS);

    // Deallocate memory
    for (int i = 0; i < NUM_ALLOCATIONS; i++) {
        free(allocations[i]);
    }
    free(allocations);

    printf("Deallocated memory.\n");
    return 0;
}
```

OUTPUT:

```
Allocated 1000 MB of memory.
Deallocated memory.
```

RESULT:

The program for custom memory allocation was executed and verified successfully.

EX.NO:14	DEMONSTRATION OF PROCESS EXECUTION INTERLEAVING IN DIFFERENT ORDERS

AIM:

The aim of demonstrating process execution interleaving is to show how different execution orders of concurrent processes impact system performance, resource utilization, and synchronization. This understanding helps in optimizing scheduling, managing resources, and designing robust systems.

ALGORITHM:

Step 1 : **Set Up Threads**: We'll create multiple threads, each representing a different process.

Step 2 : **Simulate Execution**: Each thread will perform a small task, and we'll use sleep to simulate time taken by each process.

Step 3 : **Join Threads**: We'll join the threads to ensure they complete before the main program exits.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define NUM_PROCESSES 3

// Structure to hold process information

typedef struct {

   int id; // Process ID

} Process;

// Function that each thread will execute

void *execute_process(void *arg) {

   Process *process = (Process *)arg;

   printf("Process %d: Starting execution.\n", process->id);
```

```
    for (int i = 0; i < 5; i++) {
        printf("Process %d: Executing step %d...\n", process->id, i + 1);
        sleep(1); // Simulate work being done by sleeping
    }
    printf("Process %d: Finished execution.\n", process->id);

    return NULL;
}

int main() {
    pthread_t threads[NUM_PROCESSES];
    Process processes[NUM_PROCESSES];

    // Create threads to simulate process execution
    for (int i = 0; i < NUM_PROCESSES; i++) {
        processes[i].id = i + 1; // Assign IDs to processes
        pthread_create(&threads[i], NULL, execute_process, (void *)&processes[i]);
    }

    // Wait for all threads to finish
    for (int i = 0; i < NUM_PROCESSES; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("All processes have completed execution.\n");
    return 0;
}
```

Output:

```
Process 1: Starting execution.
Process 2: Starting execution.
Process 3: Starting execution.
Process 1: Executing step 1...
Process 2: Executing step 1...
Process 3: Executing step 1...
Process 1: Executing step 2...
Process 2: Executing step 2...
Process 3: Executing step 2...
Process 1: Executing step 3...
Process 1: Executing step 4...
Process 2: Executing step 3...
Process 3: Executing step 3...
Process 2: Executing step 4...
Process 3: Executing step 4...
Process 1: Finished execution.
Process 2: Finished execution.
Process 3: Finished execution.
All processes have completed execution.
```

RESULT:

Thus program for Demonstration of process execution interleaving in different orders was executed and verified successfully.

EX.NO:15[a]	**Simulation based analysis of scheduling policies** **FIRST COME FIRST SERVE (FCFS)**

AIM:

The simulation will involve creating processes with arrival times and burst times, then applying each scheduling policy to determine metrics like average waiting time and average turnaround time.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Initialize the processes and burst times.

Step - 3: Calculate the waiting time for each process

Step - 4: Calculate the turnaround time for each process

Step - 5: Print the waiting time and turnaround time for each process

Step - 6: Calculate the average waiting time and Calculate the average turnaround time.

Step - 7: End the program.

EXAMPLE PROGRAM:

```
#include <bits/stdc++.h>

using namespace std;

class Process {

private:

   int at;

   int bt;

   int ct;

   int tat;

   int wt;

   int pid;

   public:

   int& operator[](string var)

   {
        if (var == "at")
            return at;
        if (var == "bt")
            return bt;
        if (var == "ct")
            return ct;
        if (var == "tat")
            return tat;
        if (var == "wt")
            return wt;
        return pid;
    }
    void update_after_ct()
    {
        tat = ct - at;
        wt = tat - bt;
    }
    void display()
    {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid, at, bt, ct,
        tat, wt);
    }
};
float average(vector<Process> P, string var)
{
    int total = 0;
    for (auto temp : P) {
        total += temp[var];
    }
    return (float)total / P.size();
}
int main()
{
    Input description.
    First line contains an integer n
    the next n lines contains 2 space separated integers
    containing values for arrival time and burst time for
    example:
    2
    0 3
    1 2
    */
    int n;
    cin >> n;
    int counter = 0;
    vector<Process> P(n);
    for (Process& temp : P) {
        temp["id"] = counter++;
        cin >> temp["at"] >> temp["bt"];
    }
    sort(P.begin(), P.end(),
         [](Process first, Process second) {
             return first["at"] < second["at"];
         });
    printf("pid\tat\tbt\tct\ttat\twt\n");
    P[0]["ct"] = P[0]["at"] + P[0]["bt"];
    P[0].update_after_ct();
    P[0].display();
    for (int i = 1; i < P.size(); i++) {
        if (P[i]["at"] < P[i - 1]["ct"]) {
            P[i]["ct"] = P[i - 1]["ct"] + P[i]["bt"];
        }
        else {
            printf("curr['at'] : %d, prev['ct'] : %d\n",
                   P[i]["at"], P[i - 1]["ct"]);
            P[i]["ct"] = P[i]["at"] + P[i]["bt"];
        }
        P[i].update_after_ct();
        P[i].display();
    }
    printf("Average waiting time : %f\n", average(P, "wt"));
    return 0;
}
```

OUTPUT:

```
PID   AT    BT    CT    TAT   WT
0     0     10    10    10    0
1     10    5     15    5     0
2     15    8     23    8     0
Avg_turnaround:7.666666666666667
Avg_Waitingtime:0.0
```

RESULT:

The program was executed and verified successfully.

EX.NO:15[b]	Simulation based analysis of scheduling policies SHORTEST JOB FIRST(SJF)

AIM:

To simulate one of the CPU scheduling algorithm, Shortest Job First scheduling using C program.

ALGORITHM:

Step1: Start the program.

Step 2: Read the number of process.

Step 3: Read the Execution time for each process.

Step 4: Sort the process according to their execution time in ascending order.

Step5: Calculate the waiting time of each processes and the average waiting time as, Avg. waiting time = (waiting time of all process)/number of processes.

Step6: Calculate the average turn around time as, Avg. turn around time = (Turn around time of all process)/number of processes.

Step7: Display the waiting time and turn around time for each process.

Step8: Display the Average waiting time and Average turn around time.

Step9: Display the result by using Gantt chart

Step10: End the program.

SOURCE CODE:

```
#include<stdio.h>
#define MAX 50
struct process
{
    int pro;    // Process number
    int brt;    // Burst time
    int tt;     // Turnaround time
    int wt;     // Waiting time
} pr[MAX];

// Main program
```

```
int main() {
    int i, j, n, t, totwt = 0, tottt = 0;
    float avgwt, avgtt;
    int prevtt = 0;

    printf("\n\t\t\t SHORTEST JOB FIRST");
    printf("\n\n Enter the number of processes: ");
    scanf("%d", &n);

    // Getting Process no. and Burst Time
    for(i = 1; i <= n; i++) {
        pr[i].pro = i;
        printf("\nProcess no: %d ", pr[i].pro);
        printf("\nEnter burst time: ");
        scanf("%d", &pr[i].brt);
    }
    // Sorting the Burst Time in Ascending order
    for(i = 1; i <= n; i++) {
        for(j = i + 1; j <= n; j++) {
            if(pr[i].brt > pr[j].brt) {
                // Swap process number
                t = pr[j].pro;
                pr[j].pro = pr[i].pro;
                pr[i].pro = t;

                // Swap burst time
                t = pr[j].brt;
                pr[j].brt = pr[i].brt;
                pr[i].brt = t;
            }
        }
    }
    // Calculating Waiting time and Turnaround Time
    prevtt = 0;
    for(i = 1; i <= n; i++) {
        pr[i].wt = prevtt;
        pr[i].tt = pr[i].wt + pr[i].brt;
        prevtt = pr[i].tt;
    }

    // Output table headers
    printf("\n PROCESS NO \t BURST TIME \t WAITING TIME \t TURNAROUND TIME");
    printf("\n ------------ \t ----------- \t ------------ \t ----------------");

    // Sum of Waiting time and Turnaround Time
    for(i = 1; i <= n; i++) {
        printf("\n\t%d\t\t%d\t\t%d\t\t%d", pr[i].pro, pr[i].brt, pr[i].wt, pr[i].tt);
        totwt += pr[i].wt;
        tottt += pr[i].tt;
    }

// Average of Waiting time and Turnaround Time
    avgwt = (float) totwt / n;
    avgtt = (float) tottt / n;
    printf("\n\n Average Waiting Time: %6.2f", avgwt);
    printf("\n\n Average Turnaround Time: %6.2f\n\n", avgtt);

    // Top Line for Gantt Chart
    for(i = 0; i <= pr[n].tt; i++) {
        printf("-");
    }
    printf("\n");
    // Gantt Chart
    printf("%d", 0);
    for(i = 1; i <= n; i++) {
        printf("%*d", pr[i].brt + 1, pr[i].tt);
    }
    printf("\n");

    // Bottom Line for Gantt Chart
    for(i = 0; i <= pr[n].tt; i++) {
        printf("-");
    }
    printf("\n");

    return 0;
}
```

OUTPUT:

```
 SHORTEST JOB FIRST
 Enter the number of processes: 3
Process no: 1
Enter burst time: 50
Process no: 2
Enter burst time: 60
Process no: 3
Enter burst time: 80

 PROCESS NO         BURST TIME          WAITING TIME       TURNAROUND TIME
 ------------       -----------         ------------       ----------------
        1               50                  0                   50
        2               60                  50                  110
        3               80                  110                 190

 Average Waiting Time:  53.33
 Average Turnaround Time: 116.67

-----------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------
0                                     50                                             110
190
```

RESULT:

Thus the program was executed successfully and the output was shown and verified.

EX.NO:15[c]	Simulation based analysis of scheduling policies ROUND ROBBIN

AIM:

To simulate one of the CPU scheduling algorithms, Round robin scheduling using C program.

ALGORITHM:

Step 1: Start the program.

Step 2: Get the total number of processes and time slice from the user.

Step 3: Get the burst time for every process from the user.

Step 4: Initialize the total turnaround time to 0.

Step 5: calculate the total turnaround time and remaining time of each process for every iteration using the below statements, tottt = tottt + ts; pr[i].remtime = pr[i].remtime - ts;

Step 6: Calculate the total waiting time and turnaround time and its average.

Step 7: Display average waiting time and average turnaround time to the user.

Step 8: Stop the program.

SOURCE CODE:

```
#include<stdio.h>
#define MAX 50
struct process {
    int pro;        // Process number
    int brt;        // Burst time
    int tt;         // Turnaround time
    int wt;         // Waiting time
    int remtime;    // Remaining time
} pr[MAX];

// Main program
int main() {
    int i, j, n, t = 0, totwt = 0, tottt = 0;
    float avgwt, avgtt;
    int ts;  // Time slice
    int totbrt = 0;   // Total burst time
```

```
// Input number of processes and time slice
printf("\n Enter the number of processes: ");
scanf("%d", &n);
printf("\n Enter the time slice: ");
scanf("%d", &ts);

// Getting Process no. and Burst Time
for(i = 1; i <= n; i++) {
    printf("\nEnter process no: %d\n", i);
    pr[i].pro = i;
    printf("Enter burst time: ");
    scanf("%d", &pr[i].brt);
    pr[i].remtime = pr[i].brt; // Initialize remaining time
    totbrt += pr[i].brt;       // Total burst time for all processes
}

// Calculating Waiting time and Turnaround Time
tottt = 0; // Reset total turnaround time counter
while (1) {
    if(tottt == totbrt) { // Break if total turnaround time equals total burst time
        break;
    }
    for(i = 1; i <= n; i++) {
        if(pr[i].remtime == 0) { // If the remaining time is zero, process is finished
            continue;
        }

        if(pr[i].remtime > ts) {
            tottt += ts; // Add time slice to total turnaround time
            pr[i].remtime -= ts; // Reduce remaining time by time slice
        } else {
                tottt += pr[i].remtime; // Add the remaining time to total turnaround time
                pr[i].remtime = 0; // Set remaining time to 0 (process finished)
            }

            pr[i].wt = tottt - pr[i].brt; // Calculate waiting time
            pr[i].tt = tottt;            // Set turnaround time
        }
    }

    // Output table headers
    printf("\n PROCESS NO \t BURST TIME \t WAITING TIME \t TURNAROUND TIME");
    printf("\n ------------ \t ----------- \t ------------ \t ----------------");

    // Sum of Waiting time and Turnaround Time
    totwt = 0;
    tottt = 0;
    for(i = 1; i <= n; i++) {
        printf("\n\t%d\t\t%d\t\t%d\t\t%d", pr[i].pro, pr[i].brt, pr[i].wt, pr[i].tt);
        totwt += pr[i].wt;
        tottt += pr[i].tt;
    }

    // Average of Waiting time and Turnaround Time
    avgwt = (float) totwt / n;
    avgtt = (float) tottt / n;
    printf("\n\n Average Waiting Time: %6.2f", avgwt);
    printf("\n\n Average Turnaround Time: %6.2f", avgtt);

    return 0;
}
```

OUTPUT:

```
 Enter the number of processes: 3
 Enter the time slice: 45
Enter process no: 1
Enter burst time: 65
Enter process no: 2
Enter burst time: 85
Enter process no: 3
Enter burst time: 35

 PROCESS NO       BURST TIME       WAITING TIME     TURNAROUND TIME
 ------------     -----------      ------------     ----------------
      1               65               80               145
      2               85               100              185
      3               35               90               125

 Average Waiting Time:  90.00

 Average Turnaround Time: 151.67
```

RESULT:

Thus the given program is executed successfully and the output was shown and verified

EX.NO:15[d]	SIMULATION BASED ANALYSIS OF PRIORITY SCHEDULING

AIM:

To simulate one of the CPU scheduling algorithms, Priority scheduling using C program.

ALGORITHM:

Step1: Start the program.

Step2: Read the number of processes.

Step3: Get the Execution time (Burst time) and priority for each process.

Step4: Sort the processes based on their priority.

Step5: Calculate the waiting time of each process and the Average waiting as Avg. waiting time = (waiting time of all processes)/number of processes.

Step6: Calculate the turn around time of each process and Average turn around time as Avg. turn around time = (Turn around time of all processes)/number of process.

Step7: Display the arrival, waiting time and turn around time for each process.

Step 8: Display the Average waiting time and average turn around time.

Step9: Display the result by using Gantt chart

Step 10: End processing.

EXAMPLE CODE:

```
#include<stdio.h>
#define MAX 50
struct process
{
    int pro;  // process number
    int brt;  // burst time
    int tt;   // turnaround time
    int wt;   // waiting time
    int pri;  // priority
} pr[MAX];

int main()
{
    int i, j, n, t, totwt = 0, tottt = 0;
```

```
float avgwt, avgtt;
  int prevtt = 0;

  printf("\n\t\t\t PRIORITY SCHEDULING\n");
  printf("\n\n Enter the number of processes: ");
  scanf("%d", &n);

  // Getting Process number, Burst Time, and Priority
  for(i = 0; i < n; i++)
  {
     pr[i].pro = i + 1;  // Process numbers start from 1
     printf("\nProcess no: %d", pr[i].pro);
     printf("\nEnter burst time: ");
     scanf("%d", &pr[i].brt);
     printf("Enter priority: ");
     scanf("%d", &pr[i].pri);
  }

  // Sorting the processes by priority (ascending order)
  for(i = 0; i < n; i++)
  {
     for(j = i + 1; j < n; j++)
     {
        if(pr[i].pri > pr[j].pri)
        {
           // Swap process number
           t = pr[j].pro;
           pr[j].pro = pr[i].pro;
           pr[i].pro = t;

           // Swap burst time
           t = pr[j].brt;
```

```
            pr[j].brt = pr[i].brt;
            pr[i].brt = t;

            // Swap priority
            t = pr[j].pri;
            pr[j].pri = pr[i].pri;
            pr[i].pri = t;
        }
    }
}

// Calculating Waiting Time and Turnaround Time
for(i = 0; i < n; i++)
{
    pr[i].wt = prevtt;
    pr[i].tt = pr[i].wt + pr[i].brt;
    prevtt = pr[i].tt;
}

// Displaying the result
printf("\nPROCESS NO \tBURST TIME \tWAITING TIME \tTURNAROUND TIME\n");
printf("----------- \t----------- \t------------ \t---------------\n");

for(i = 0; i < n; i++)
{
    printf("\t%d\t\t%d\t\t%d\t\t%d\n", pr[i].pro, pr[i].brt, pr[i].wt, pr[i].tt);
    totwt += pr[i].wt;
    tottt += pr[i].tt;
}

// Calculating and displaying averages
avgwt = (float)totwt / n;
```

```
    avgtt = (float)tottt / n;
    printf("\nAverage Waiting Time: %6.2f", avgwt);
    printf("\nAverage Turnaround Time: %6.2f\n\n", avgtt);

    // Displaying Gantt Chart
    printf("\nGantt Chart:\n");
    // Top line
    for(i = 0; i <= pr[n-1].tt; i++)
    {
        printf("-");
    }
    printf("\n");

    // Gantt chart process times
    j = pr[0].wt;
    printf("%d", j);
    for(i = 0; i < n; i++)
    {
        printf("%*d", pr[i].brt, pr[i].tt);
    }
    printf("\n");

    // Bottom line
    for(i = 0; i <= pr[n-1].tt; i++)
    {
        printf("-");
    }
    printf("\n");

    return 0;
}
```

OUTPUT:

```
 PRIORITY SCHEDULING
 Enter the number of processes: 3
Process no: 1
Enter burst time: 4
Enter priority: 6
Process no: 2
Enter burst time: 8
Enter priority: 9
Process no: 3
Enter burst time: 5
Enter priority: 7
PROCESS NO        BURST TIME        WAITING TIME      TURNAROUND TIME
-----------       -----------       ------------      ---------------
     1                 4                 0                 4
     3                 5                 4                 9
     2                 8                 9                 17
Average Waiting Time:   4.33
Average Turnaround Time:  10.00


Gantt Chart:
------------------
0   4   9    17
```

RESULT:

Thus the program was executed successfully and the output was shown and verified.

EX.NO:16	TOOLS USAGE — NICE/PROC/PID/STATUS

AIM:

To demonstrate the usage of nice, proc/[pid]/status, and how you can access process information in a C program, let's write a C program that:

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Identify the PID of the process.

Step - 3: Open the /proc/[pid]/status file using a file-handling mechanism

Step - 4: Read the file line by line

Step - 5: Extract the necessary information, like process name, memory usage, or state.

Step - 6: Close the file after reading the required data.

Step - 7: End the program.

EXAMPLE PROGRAM:

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

#define STATUS_FILE "/proc/%d/status"
#define BUFFER_SIZE 1024

// Function to read and print /proc/[pid]/status file
void print_proc_status(pid_t pid) {
    char filename[BUFFER_SIZE];
    snprintf(filename, sizeof(filename), STATUS_FILE, pid);
    int fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror("Failed to open status file");
        exit(EXIT_FAILURE);
    }

    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;
    while ((bytes_read = read(fd, buffer, sizeof(buffer) - 1)) > 0) {
        buffer[bytes_read] = '\0'; // Null-terminate the buffer
        printf("%s", buffer);
    }

    close(fd);
}

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        printf("Child process with PID %d started with nice value 10.\n", getpid());

        // Set a nice value for the child process
        if (nice(10) == -1) {
            perror("Failed to set nice value");
            exit(EXIT_FAILURE);
        }
        // Infinite loop to keep the process alive for demonstration
        while (1) {
            printf("Child process is running with PID %d and nice value 10.\n", getpid());
            sleep(5);  // Sleep to slow down the loop
        }
    } else {
        // Parent process
        printf("Parent process with PID %d waiting for child process %d to change nice value.\n",
getpid(), pid);

        sleep(1); // Wait for the child to set its nice value

        // Print the /proc/[pid]/status information
        printf("\nReading /proc/%d/status:\n", pid);
        print_proc_status(pid);

        // Wait for child process to finish
        wait(NULL);
    }

    return 0;
}
```

OUTPUT:

```
Parent process with PID 1234 waiting for child process 1235 to change nice value.

Reading /proc/1235/status:
Name:   proc_status_demo
State:  S (sleeping)
Tgid:   1235
Pid:    1235
```

```
PPid:   1234
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
...
Nice:   10
...
```

RESULT:

Thus the program was executed and verified successfully

EX.NO:17	CREATION OF THREADS USING THE PTHREAD API AND MODIFICATION OF SHARED VARIABLES WITH AND WITHOUT SYNCHRONIZATION.

AIM:

This program demonstrates how two threads modify a shared variable both without synchronization (leading to a race condition) and with synchronization (using a mutex).

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Include Headers #include <pthread.h> for thread functions.

Step - 3: Define Shared Variables and Create Threads

Step - 4: Modify Shared Variables

Step - 5: Synchronize Access

Step - 6: Compile and run the program.

EXAMPLE PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 2
#define ITERATIONS 1000000

int shared_variable = 0; // Shared variable

pthread_mutex_t mutex; // Mutex for synchronization
```

```
void* increment_without_sync(void* arg) {
    for (int i = 0; i < ITERATIONS; i++) {
        shared_variable++; // Increment shared variable without synchronization
    }
    return NULL;
}

void* increment_with_sync(void* arg) {
    for (int i = 0; i < ITERATIONS; i++) {
        pthread_mutex_lock(&mutex); // Lock the mutex
        shared_variable++; // Increment shared variable with synchronization
        pthread_mutex_unlock(&mutex); // Unlock the mutex
    }
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];

    // 1. Without Synchronization
    shared_variable = 0;
    printf("Without Synchronization:\n");

    // Create threads that modify the shared variable without synchronization
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment_without_sync, NULL);
    }

    // Wait for all threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    // Print the value of the shared variable
    printf("Final value of shared variable (without sync): %d\n", shared_variable);
    // 2. With Synchronization
    shared_variable = 0;
    pthread_mutex_init(&mutex, NULL); // Initialize the mutex
    printf("\nWith Synchronization:\n");
    // Create threads that modify the shared variable with synchronization
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment_with_sync, NULL);
    }
    // Wait for all threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    // Print the value of the shared variable
    printf("Final value of shared variable (with sync): %d\n", shared_variable);
    // Destroy the mutex
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

OUTPUT:

```
Without Synchronization:
Final value of shared variable (without sync): 1227011
With Synchronization:
Final value of shared variable (with sync): 2000000
```

RESULT:

Thus the program was executed and verified successfully

EX.NO:18[a]	Using spinlock, implement semaphores, barriers (using the threads API)

AIM:

Implementing semaphores and barriers using threads API in C can be done using synchronization primitives such as spinlocks.

ALGORITHM

STEP 1:pthread_cond_init: Initializes a condition variable

STEP 2:pthread_cond_wait: Waits on a condition variable, releasing and then reacquiring the mutex

STEP 3:pthread_cond_timedwait: Waits on a condition variable for a specified time period

STEP 4:pthread_cond_signal: Signals a condition variable, waking up at least one blocked thread

STEP 5:pthread_cond_broadcast: Signals a condition variable, waking up all blocked threads

STEP 6:pthread_cond_destroy: Destroys a condition variable and cleans up its resources

SORCE CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <stdatomic.h>

#include <unistd.h>


typedef struct {

    atomic_int lock; // Atomic variable to represent the lock

} Spinlock;
```

```
// Initialize the spinlock
void spinlock_init(Spinlock *spinlock) {
    atomic_store(&spinlock->lock, 0);
}


// Acquire the spinlock
void spinlock_lock(Spinlock *spinlock) {
    while (atomic_exchange(&spinlock->lock, 1) != 0) {
        // Busy wait (spin) until the lock becomes available
        while (atomic_load(&spinlock->lock) != 0);
    }
}
// Release the spinlock
void spinlock_unlock(Spinlock *spinlock) {
    atomic_store(&spinlock->lock, 0);
}
// Shared resource
int shared_resource = 0;
Spinlock spinlock;
// Thread function to increment the shared resource
void *increment(void *arg) {
    for (int i = 0; i < 100000; i++) {
        spinlock_lock(&spinlock);
        shared_resource++; // Critical section
        spinlock_unlock(&spinlock);
    }
    return NULL;
}
int main() {
    pthread_t threads[4];
    // Initialize the spinlock
    spinlock_init(&spinlock);
    // Create multiple threads to increment the shared resource
    for (int i = 0; i < 4; i++) {
        pthread_create(&threads[i], NULL, increment, NULL);
    }
    // Wait for all threads to finish
    for (int i = 0; i < 4; i++) {
        pthread_join(threads[i], NULL);
    }
    // Print the final value of the shared resource
    printf("Final value of shared resource: %d\n", shared_resource);
    return 0;
}
```

OUTPUT:

```
Final value of shared resource: 400000
```

RESULT:

Thus the program was executed and verified successfully.

EX.NO:18[b]	Using mutexes implement semaphores, barriers (using the threads API)

AIM:

Implementing semaphores and barriers using threads API in C can be done using synchronization primitives such as mutexes.

ALGORITHM

STEP 1:pthread_cond_init: Initializes a condition variable

STEP 2:pthread_cond_wait: Waits on a condition variable, releasing and then reacquiring the mutex

STEP 3:pthread_cond_timedwait: Waits on a condition variable for a specified time period

STEP 4:pthread_cond_signal: Signals a condition variable, waking up at least one blocked thread

STEP 5:pthread_cond_broadcast: Signals a condition variable, waking up all blocked threads

STEP 6:pthread_cond_destroy: Destroys a condition variable and cleans up its resources

SOURCE CODE

```
#include <iostream>

#include <thread>

#include <mutex>

using namespace std;

int flag[2];

int turn;

const int MAX = 1e9;

int ans = 0;

void lock_init()

{

    flag[0] = flag[1] = 0;

    turn = 0;
```

```
}
void lock(int self)
{
    flag[self] = 1;
    turn = 1 - self;
    while (flag[1 - self] == 1 && turn == 1 - self);
}
void unlock(int self)
{
    flag[self] = 0;
}
void func(int self)
{
    int i = 0;
    cout << "Thread Entered: " << self << endl;
    lock(self);
    for (i = 0; i < MAX; i++)
        ans++;
    unlock(self);
}
int main()
{  thread t1(func, 0);
    thread t2(func, 1);
    lock_init();
    t1.join();
    t2.join();
    cout << "Actual Count: " << ans << " | Expected Count: " << MAX * 2 << endl;
    return 0;
```

OUTPUT:

```
Thread Entered: 1
Thread Entered: 0
Actual Count: 2000000000 | Expected Count: 2000000000
```

RESULT:

Thus the program was executed and verified successfully.

EX.NO:18[c]	**Using condition variables to implement semaphores, barriers (using the threads API)**

AIM:

Implementing semaphores and barriers using threads API in C can be done using synchronization primitives such as condition variables.

ALGORITHM

STEP 1:pthread_cond_init: Initializes a condition variable

STEP 2:pthread_cond_wait: Waits on a condition variable, releasing and then reacquiring the mutex .

STEP 3:pthread_cond_timedwait: Waits on a condition variable for a specified time period .

STEP 4:pthread_cond_signal: Signals a condition variable, waking up at least one blocked thread .

STEP 5:pthread_cond_broadcast: Signals a condition variable, waking up all blocked threads .

STEP 6:pthread_cond_destroy: Destroys a condition variable and cleans up its resources .

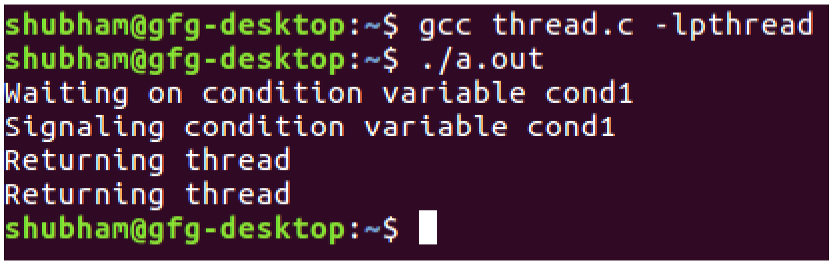
SOURCE CODE

```
#include <pthread.h>

#include <stdio.h>

#include <unistd.h>

 // Declaration of thread condition variable

pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;

 // declaring mutex

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

 int done = 1;

 // Thread function
```

```
void* foo()

{

    // acquire a lock

    pthread_mutex_lock(&lock);

    if (done == 1) {


        // let's wait on condition variable cond1

        done = 2;

        printf("Waiting on condition variable cond1\n");

        pthread_cond_wait(&cond1, &lock);

    }

    else {


        // Let's signal condition variable cond1

        printf("Signaling condition variable cond1\n");

        pthread_cond_signal(&cond1);

    }

  // release lock

    pthread_mutex_unlock(&lock);

   printf("Returning thread\n");

  return NULL;

}

// Driver code

int main()

{
    pthread_t tid1, tid2;
 // Create thread 1
    pthread_create(&tid1, NULL, foo, NULL);
 // sleep for 1 sec so that thread 1
    // would get a chance to run first
    sleep(1);
   // Create thread 2
    pthread_create(&tid2, NULL, foo, NULL);


    // wait for the completion of thread 2
    pthread_join(tid2, NULL);
     return 0;
}
```

OUTPUT:

-



```
shubham@gfg-desktop:~$ gcc thread.c -lpthread
shubham@gfg-desktop:~$ ./a.out
Waiting on condition variable cond1
Signaling condition variable cond1
Returning thread
Returning thread
shubham@gfg-desktop:~$
```

RESULT:

Thus the program was executed and verified successfully.

EX.NO:19[a]	IMPLEMENT SOLUTIONS TO THE PRODUCER-CONSUMER PROBLEMS USING THE DIFFERENT SYNCHRONIZATION PRIMITIVES PROGRAM.

AIM:

In the Producer-Consumer problem, one or more producer threads produce items and place them into a shared buffer, while one or more consumer threads take items from the buffer. We need to ensure that the producer and consumer threads do not access the buffer concurrently in a way that leads to inconsistencies.

ALGORITHM:

STEP 1 :Include the <pthread.h> header file

STEP 2 :Link with the -lpthread option

STEP 3 :Declare a variable of type pthread_cond_t

STEP 4 :Initialize the variable with PTHREAD_COND_INITIALIZER or pthread_cond_init

STEP 5 :Call pthread_cond_wait or pthread_cond_timedwait to wait on a condition variable

STEP 6 :Call pthread_cond_signal or pthread_cond_broadcast to signal or broadcast a condition variable

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>

#define BUFFER_SIZE 10
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 2
#define PRODUCE_COUNT 5

// Shared buffer
int buffer[BUFFER_SIZE];
int count = 0; // Number of items in the buffer
int in = 0;    // Index for the producer
int out = 0;   // Index for the consumer

// Mutex and condition variables
pthread_mutex_t mutex;
pthread_cond_t not_full;
pthread_cond_t not_empty;

// Producer function
void* producer(void* arg) {
for (int i = 0; i < PRODUCE_COUNT; i++) {
        pthread_mutex_lock(&mutex);

while (count == BUFFER_SIZE) {
            pthread_cond_wait(&not_full, &mutex);
        }

        buffer[in] = i; // Produce an item
printf("Producer %ld produced %d\n", (long)arg, buffer[in]);
        in = (in + 1) % BUFFER_SIZE;
        count++;

        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&mutex);
        sleep(rand() % 2); // Simulate variable production time
    }
returnNULL;
}

// Consumer function
void* consumer(void* arg) {
```

```
for (int i = 0; i < PRODUCE_COUNT; i++) {
        pthread_mutex_lock(&mutex);

while (count == 0) {
            pthread_cond_wait(&not_empty, &mutex);
        }

int item = buffer[out]; // Consume an item
printf("Consumer %ld consumed %d\n", (long)arg, item);
        out = (out + 1) % BUFFER_SIZE;
        count--;

        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
        sleep(rand() % 2); // Simulate variable consumption time
    }
returnNULL;
}

intmain() {
pthread_t producers[NUM_PRODUCERS];
pthread_t consumers[NUM_CONSUMERS];

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_full, NULL);
    pthread_cond_init(&not_empty, NULL);

// Create producer and consumer threads
for (long i = 0; i < NUM_PRODUCERS; i++) {
        pthread_create(&producers[i], NULL, producer, (void*)i);
    }
for (long i = 0; i < NUM_CONSUMERS; i++) {
        pthread_create(&consumers[i], NULL, consumer, (void*)i);
    }

// Wait for all threads to finish
for (int i = 0; i < NUM_PRODUCERS; i++) {
        pthread_join(producers[i], NULL);
    }
for (int i = 0; i < NUM_CONSUMERS; i++) {
        pthread_join(consumers[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_full);
    pthread_cond_destroy(&not_empty);
return0;
}
```

OUTPUT:

```
Producer 0 produced 0
Producer 1 produced 0
Consumer 1 consumed 0
Consumer 0 consumed 0
Producer 1 produced 1
Producer 0 produced 1
Consumer 1 consumed 1
Consumer 0 consumed 1
Producer 1 produced 2
Consumer 0 consumed 2
Producer 0 produced 2
Consumer 1 consumed 2
Producer 0 produced 3
Consumer 0 consumed 3
Producer 1 produced 3
Producer 0 produced 4
Consumer 1 consumed 3
Consumer 1 consumed 4
Producer 1 produced 4
Consumer 0 consumed 4
```

RESULT:

Thus the Producer Consumer   program was executed and verified successfully.

EX.NO:19[b]	IMPLEMENT SOLUTIONS TO THE READERWRITERS PROBLEMS USING THE DIFFERENT SYNCHRONIZATION PRIMITIVES PROGRAM

AIM:

In the Reader-Writer problem, multiple reader threads need to access a shared resource (such as a database) for reading, while writer threads need to write to it. Readers can work concurrently but writers need exclusive access.

ALGORITHM:

Step 1 : **Readers**: Can read the shared resource concurrently.

Step2 : Must have exclusive access to the shared resource when writing.

Step 3 : **Priority**: A decision must be made about how to prioritize readers and writers. Common strategies include

Step 4: **Readers First**: Allow readers to proceed as long as no writers are waiting.

Step 5:**Writers First**: Allow writers to proceed as soon as they are waiting, potentially starving readers.

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#define NUM_READERS 3
#define NUM_WRITERS 2
#define READ_COUNT 5
#define WRITE_COUNT 3
// Shared resource and synchronization primitives
int data = 0;
int read_count = 0;
pthread_mutex_t mutex;
pthread_mutex_t read_mutex;
```

```
pthread_cond_t write_cond;
// Reader function
void* reader(void* arg) {
for (int i = 0; i < READ_COUNT; i++) {
        pthread_mutex_lock(&read_mutex);
        read_count++;
if (read_count == 1) {
            pthread_mutex_lock(&mutex); // Lock the resource for reading
        }
        pthread_mutex_unlock(&read_mutex);

printf("Reader %ld reading data: %d\n", (long)arg, data);

        pthread_mutex_lock(&read_mutex);
        read_count--;
if (read_count == 0) {
            pthread_mutex_unlock(&mutex); // Unlock the resource if no readers
        }
        pthread_mutex_unlock(&read_mutex);

        sleep(rand() % 2); // Simulate variable read time
    }
returnNULL;
}
// Writer function
void* writer(void* arg) {
for (int i = 0; i < WRITE_COUNT; i++) {
  pthread_mutex_lock(&mutex);
  data = rand() % 100; // Write data
printf("Writer %ld writing data: %d\n", (long)arg, data);
 pthread_mutex_unlock(&mutex);
  pthread_cond_broadcast(&write_cond);
 sleep(rand() % 2); // Simulate variable write time
    }
returnNULL;
}
intmain() {
pthread_t readers[NUM_READERS];
pthread_t writers[NUM_WRITERS];
 pthread_mutex_init(&mutex, NULL);
 pthread_mutex_init(&read_mutex, NULL);
 pthread_cond_init(&write_cond, NULL);
// Create reader and writer threads
for (long i = 0; i < NUM_READERS; i++) {
        pthread_create(&readers[i], NULL, reader, (void*)i);
    }
for (long i = 0; i < NUM_WRITERS; i++) {
        pthread_create(&writers[i], NULL, writer, (void*)i);
    }
// Wait for all threads to finish
for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(readers[i], NULL);
    }
for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writers[i], NULL);
    }
  pthread_mutex_destroy(&mutex);
    pthread_mutex_destroy(&read_mutex);
    pthread_cond_destroy(&write_cond);
return0;
}
```

OUTPUT:

```
Reader 1 reading data: 0
Reader 0 reading data: 0
Reader 2 reading data: 0
Writer 0 writing data: 15
Reader 0 reading data: 15
Writer 1 writing data: 86
Writer 1 writing data: 49
Reader 1 reading data: 49
Reader 2 reading data: 49
Reader 0 reading data: 49
Writer 0 writing data: 59
Reader 1 reading data: 59
Reader 0 reading data: 59
Writer 1 writing data: 26
Reader 0 reading data: 26
Reader 1 reading data: 26
Reader 2 reading data: 26
Writer 0 writing data: 67
Reader 2 reading data: 67
Reader 1 reading data: 67
Reader 2 reading data: 67
```

RESULT:

Thus the Reader Writer program was executed and verified successfully.

EX.NO:20[a]	IMPLEMENTATION OF BINARY SEMAPHORE

AIM:

The goal is to implement synchronized access to a shared memory area across processes using semaphores. This ensures mutual exclusion and correct message passing between producer and consumer processes using shared memory.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Initialize the shared memory

Step - 3: Create and initialize the binary semaphore

Step - 4: Producer process (writing messages)

Step - 5: Consumer process (reading messages)

Step - 6: Compile and run the program.

Step - 7: End the program.

SOURCE CODE:

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

sem_t semaphore;

void* critical_section(void* arg) {
    sem_wait(&semaphore); // P operation
    printf("Entered critical section\n");
    // Critical section code
    printf("Exiting critical section\n");
    sem_post(&semaphore); // V operation
    return NULL;
}
```

```
int main() {
   pthread_t t1, t2;
   sem_init(&semaphore, 0, 1); // Initialize semaphore with value 1

   pthread_create(&t1, NULL, critical_section, NULL);
   pthread_create(&t2, NULL, critical_section, NULL);

   pthread_join(t1, NULL);
   pthread_join(t2, NULL);

   sem_destroy(&semaphore); // Destroy semaphore
   return 0;
}
```

OUTPUT:

```
Entered critical section
Exiting critical section
Entered critical section
Exiting critical section
```

RESULT:

The program was executed and verified successfully.

EX.NO:20[b]	IMPLEMENTATION OF COUNTER SEMAPHORE

AIM:

To implement a **counting semaphore** to synchronize access to a shared pool of resources, allowing multiple processes to acquire or release resources based on availability. This ensures efficient resource management in concurrent systems.

ALGORITHM:

```
Step - 1: Begin the program.
Step - 2: Create Shared Memory
Step - 3: Attach Shared Memory
Step - 4: Create Semaphore and Initialize Semaphore
Step - 5: Access Shared Memory - Lock the semaphore to ensure exclusive access
Step - 6: Perform operations on the shared memory and Unlock the semaphore
Step - 7: End the program.
```

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM_THREADS 5
sem_t semaphore;
void* thread_function(void* arg) {
    int thread_num = *((int*)arg);
    sem_wait(&semaphore); // Wait (P operation)
    printf("Thread %d is in the critical section\n", thread_num);
    sleep(1); // Simulate some work
    printf("Thread %d is leaving the critical section\n", thread_num);
    sem_post(&semaphore); // Signal (V operation)
    return NULL;
}
```

```
int main() {
   pthread_t threads[NUM_THREADS];
   int thread_nums[NUM_THREADS];
 // Initialize the semaphore with a value of 2
   sem_init(&semaphore, 0, 2);
   for (int i = 0; i < NUM_THREADS; i++) {
      thread_nums[i] = i;
      pthread_create(&threads[i], NULL, thread_function, &thread_nums[i]);
   }
   for (int i = 0; i < NUM_THREADS; i++) {
   pthread_join(threads[i], NULL);
   }
   sem_destroy(&semaphore);
   return 0;
}
```

OUTPUT:

```
Thread 0 is in the critical section
Thread 1 is in the critical section
Thread 0 is leaving the critical section
Thread 2 is in the critical section
Thread 1 is leaving the critical section
Thread 4 is in the critical section
Thread 2 is leaving the critical section
Thread 3 is in the critical section
Thread 4 is leaving the critical section
Thread 3 is leaving the critical section
```

RESULT:

The program was executed and verified successfully.

EX.NO:21	Implementation of file utilities(e.g.find,grep)usingthesystemcallAPI.

AIM:

To implement file utilities such as find and grep using system calls to perform operations like searching directories and file content filtering. This involves using system calls to handle files, directories, and I/O operations efficiently.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Include Headers

Step - 3: Implement Directory Traversal (find)

Step - 4: Implement File Search (grep)

Step - 5: Compile and Run the program.

Step - 6: End the program.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFFER 1024
int main(int argc, char *argv[]) {
    FILE *fp;
    char fline[BUFFER];
    int line = 1;
    if (argc != 3) {
        printf("Please use the following syntax:\n");
        printf("./a.out filename searchstring\n");
        printf("Example: ./a.out test1.c print\n");
        exit(1);
    }
// argv[1] is filename
    if (!(fp = fopen(argv[1], "r"))) {
        printf("grep: Could not open file: %s\n", argv[1]);
        exit(1);
    }
while (fgets(fline, BUFFER, fp) != NULL) {
        // argv[2] is search string
        if (strstr(fline, argv[2]) != NULL) {
            printf("Line number: %d text: %s\n", line, fline);
        }
        line++;
    }

    fclose(fp);
    return 0;
}
```

OUTPUT:

```
Please use the following syntax:
./a.out filename searchstring
Example: ./a.out test1.c print
```

RESULT:

The program was executed and verified successfully.

EX.NO:22	IMPLEMENTATION OF SIMPLE FILE SYSTEM THAT MANAGES FILES ON AN EMULATED DISK, FOCUSING ON FILE SYSTEM API, SUPERBLOCK, INODE, AND DATA BLOCK MANAGEMENT.

AIM:

To create a simple file system that manages files on an emulated disk, focusing on file system API, superblock, inode, and data block management. This involves initializing the file system, allocating inodes and blocks, and handling basic file operations.

ALGORITHM:

Step - 1: Begin the program.

Step - 2: Prepare the file system.

Step - 3: Find an Inode

Step - 4: Find a Block

Step - 5: Create the file and put its content into the block.

Step - 6: Run the Executable

Step - 7: End the program.

Source code:

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

#define BLOCK_SIZE 4096
#define NUM_BLOCKS 1024
#define NUM_INODES 128

typedef struct {
    int size;
    int direct_blocks[10];
```

```
} inode;

typedef struct {
    int num_blocks;
    int num_inodes;
    int free_blocks;
    int free_inodes;
    int inode_map[NUM_INODES];
    int block_map[NUM_BLOCKS];
} superblock;

superblock sb;
inode inodes[NUM_INODES];
char disk[NUM_BLOCKS][BLOCK_SIZE];

void init_fs() {
    sb.num_blocks = NUM_BLOCKS;
    sb.num_inodes = NUM_INODES;
    sb.free_blocks = NUM_BLOCKS;
    sb.free_inodes = NUM_INODES;
    memset(sb.inode_map, 0, sizeof(sb.inode_map));
    memset(sb.block_map, 0, sizeof(sb.block_map));
}
int allocate_inode() {
    for (int i = 0; i < NUM_INODES; i++) {
        if (sb.inode_map[i] == 0) {
            sb.inode_map[i] = 1;
            sb.free_inodes--;
            return i;
        }
    }
    return -1; // No free inodes
```

```
}
int allocate_block() {
    for (int i = 0; i < NUM_BLOCKS; i++) {
        if (sb.block_map[i] == 0) {
            sb.block_map[i] = 1;
            sb.free_blocks--;
            return i;
        }
    }
    return -1; // No free blocks
}

void create_file(const char *filename, const char *content) {
    int inode_index = allocate_inode();
    if (inode_index == -1) {
        printf("No free inodes\n");
        return;
    }

    inode *node = &inodes[inode_index];
    node->size = strlen(content);
    int block_index = allocate_block();
    if (block_index == -1) {
        printf("No free blocks\n");
        return;
    }
    node->direct_blocks[0] = block_index;
    strcpy(disk[block_index], content);
    printf("File '%s' created with inode %d and block %d\n", filename, inode_index, block_index);
}
int main() {
    init_fs();
    create_file("hello.txt", "Hello, World!");
    return 0;
}
```

OUTPUT:

```
File 'hello.txt' created with inode 0 and block 0.
```

RESULT:

The program was executed and verified successfully.