31. Write a program that launches an application using the vfork() system call.

```
BEGIN

    DECLARE pid as process ID

    // 1. Create a new child process using vfork()
    pid = vfork()
    IF pid < 0 THEN
        PRINT "vfork failed"
        EXIT

    // 2. CHILD PROCESS LOGIC
    IF pid == 0 THEN
        PRINT "Child process: launching application..."

        // Replace child process image with new program
        CALL execlp("/bin/ls", "ls", "-l", NULL)

        // If execlp fails
        PRINT "execlp failed"
        EXIT

    // 3. PARENT PROCESS LOGIC
    ELSE
        PRINT "Parent process: waiting for child to complete..."
        CALL wait(NULL)
        PRINT "Parent process: child has finished."

END
```

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

int main() {
    pid_t pid;

    // 1. Create a new child process using vfork()
    pid = vfork();

    if (pid < 0) {
        perror("vfork failed");
        exit(EXIT_FAILURE);
    }

    // 2. CHILD PROCESS
    else if (pid == 0) {
        printf("Child process: launching application...\n");

        // Launch an application (e.g., list files)
        execlp("/bin/ls", "ls", "-l", NULL);

        // If execlp fails
        perror("execlp failed");
        _exit(1);  // Use _exit() in vfork child to avoid corrupting parent memory
    }

    // 3. PARENT PROCESS
    else {
        printf("Parent process: waiting for child to complete...\n");
        wait(NULL);
        printf("Parent process: child has finished.\n");
    }

    return 0;
}
```

32. Demonstrate the use of wait() with fork() by writing a program that shows parentchild synchronization.

```
BEGIN

    DECLARE pid as process ID

    // 1. Create a new child process using fork()
    pid = fork()
    IF pid < 0 THEN
        PRINT "Fork failed"
        EXIT

    // 2. CHILD PROCESS LOGIC
    IF pid == 0 THEN
        PRINT "Child: Starting execution..."
        SLEEP for 3 seconds
        PRINT "Child: Execution complete."
        EXIT

    // 3. PARENT PROCESS LOGIC
    ELSE
        PRINT "Parent: Waiting for child to finish..."
        CALL wait(NULL)
        PRINT "Parent: Child process has completed. Continuing execution."
        PRINT "Parent: Execution complete."

END
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;

    // 1. Create child process
    pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }

    // 2. CHILD PROCESS
    else if (pid == 0) {
        printf("Child: Starting execution...\n");
        sleep(3);  // Simulate some work
        printf("Child: Execution complete.\n");
        exit(0);
    }

    // 3. PARENT PROCESS
    else {
        printf("Parent: Waiting for child to finish...\n");
        wait(NULL);  // Waits until the child finishes
        printf("Parent: Child process has completed. Continuing execution.\n");
        printf("Parent: Execution complete.\n");
    }

    return 0;
}
```

33. Write a program to illustrate different variants of the exec() family of system calls.

```
BEGIN

   DECLARE pid as process ID

   // 1. Create a new child process using fork()
   pid = fork()
   IF pid < 0 THEN
      PRINT "Fork failed"
      EXIT

   // 2. CHILD PROCESS LOGIC
   IF pid == 0 THEN
      PRINT "Child: Demonstrating exec() family calls..."

      // 2.1 Using execl()
      PRINT "Using execl() to run 'ls -l'"
      CALL execl("/bin/ls", "ls", "-l", NULL)

      // 2.2 If execl() fails, demonstrate execlp()
      PRINT "Using execlp() to run 'date'"
      CALL execlp("date", "date", NULL)

      // 2.3 Using execv()
      DECLARE args as array of strings = {"/bin/echo", "Hello from execv()", NULL}
      CALL execv("/bin/echo", args)

      // 2.4 Using execvp()
      DECLARE args2 as array of strings = {"echo", "Hello from execvp()", NULL}
      CALL execvp("echo", args2)

      // If all exec calls fail
      PRINT "All exec calls failed!"
      EXIT

   // 3. PARENT PROCESS LOGIC
   ELSE
      PRINT "Parent: Waiting for child to complete..."
      CALL wait(NULL)
      PRINT "Parent: Child finished executing exec() examples."

END
```

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

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;

    pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }

    // CHILD PROCESS
    else if (pid == 0) {
        printf("Child: Demonstrating exec() family calls...\n");

        // 1. execl() - Uses a full path and explicit arguments
        printf("\n[execl()] Running 'ls -l'...\n");
        execl("/bin/ls", "ls", "-l", NULL);

        // 2. execlp() - Uses PATH environment variable to find program
        printf("\n[execlp()] Running 'date'...\n");
        execlp("date", "date", NULL);

        // 3. execv() - Passes argument vector array with full path
        char *args1[] = {"/bin/echo", "Hello from execv()", NULL};
        printf("\n[execv()] Running '/bin/echo'...\n");
        execv("/bin/echo", args1);

        // 4. execvp() - Uses PATH and argument vector array
        char *args2[] = {"echo", "Hello from execvp()", NULL};
        printf("\n[execvp()] Running 'echo'...\n");
        execvp("echo", args2);

        // If none of the exec calls work
        perror("All exec calls failed");
        exit(EXIT_FAILURE);
    }

    // PARENT PROCESS
    else {
        printf("Parent: Waiting for child to complete...\n");
        wait(NULL);
        printf("Parent: Child finished executing exec() examples.\n");
    }

    return 0;
}
```

34. Create a program that demonstrates exit() combined with wait() and fork() (showing how children terminate and how parents collect status).

```
BEGIN

    DECLARE pid as process ID
    DECLARE status as integer

    // 1. Create a new child process
    pid = fork()
    IF pid < 0 THEN
        PRINT "Fork failed"
        EXIT

    // 2. CHILD PROCESS LOGIC
    IF pid == 0 THEN
        PRINT "Child: Starting execution..."
        SLEEP for 2 seconds
        PRINT "Child: Exiting with status code 5"
        CALL exit(5)

    // 3. PARENT PROCESS LOGIC
    ELSE
        PRINT "Parent: Waiting for child to terminate..."
        CALL wait(&status)

        IF WIFEXITED(status) THEN
            PRINT "Parent: Child exited normally."
            PRINT "Parent: Exit status = WEXITSTATUS(status)"
        ELSE
            PRINT "Parent: Child did not terminate normally."
        END IF

        PRINT "Parent: Execution complete."

END
```

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

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;
    int status;

    // 1. Create a child process
    pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }

    // 2. CHILD PROCESS
    else if (pid == 0) {
        printf("Child: Starting execution...\n");
        sleep(2);  // Simulate work
        printf("Child: Exiting with status code 5\n");
        exit(5);   // Terminate with exit code 5
    }

    // 3. PARENT PROCESS
    else {
        printf("Parent: Waiting for child to terminate...\n");

        wait(&status);  // Collect child's termination status

        if (WIFEXITED(status)) {
            printf("Parent: Child exited normally.\n");
            printf("Parent: Exit status = %d\n", WEXITSTATUS(status));
        } else {
            printf("Parent: Child did not terminate normally.\n");
        }

        printf("Parent: Execution complete.\n");
    }

    return 0;
}
```

35. Write a program that uses kill() to send signals between two unrelated processes.

```
BEGIN

    DECLARE pid as process ID
    DECLARE choice as integer

    PRINT "Choose process role:"
    PRINT "1. Sender process"
    PRINT "2. Receiver process"
    READ choice

    // 1. RECEIVER PROCESS LOGIC
    IF choice == 2 THEN
        DECLARE signal_handler for SIGUSR1

        PRINT "Receiver: My PID is", GETPID()
        REGISTER signal_handler for SIGUSR1

        LOOP forever
            PAUSE()  // Wait for signal
        END LOOP

    // 2. SENDER PROCESS LOGIC
    ELSE IF choice == 1 THEN
        DECLARE target_pid as integer
        PRINT "Enter receiver's PID: "
        READ target_pid

        PRINT "Sender: Sending SIGUSR1 to PID", target_pid
        CALL kill(target_pid, SIGUSR1)

        IF kill() fails THEN
            PRINT "Error: Failed to send signal"
        ELSE
            PRINT "Signal sent successfully."
        END IF

    ELSE
        PRINT "Invalid choice"

END
```

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

```
#include <unistd.h>
#include <signal.h>

// Signal handler function
void signal_handler(int sig) {
    if (sig == SIGUSR1) {
        printf("Receiver: Received SIGUSR1 signal!\n");
    }
}

int main() {
    int choice;

    printf("Choose process role:\n");
    printf("1. Sender process\n");
    printf("2. Receiver process\n");
    printf("Enter choice: ");
    scanf("%d", &choice);

    if (choice == 2) {
        // Receiver process
        printf("Receiver: My PID is %d\n", getpid());
        signal(SIGUSR1, signal_handler);

        printf("Receiver: Waiting for signal...\n");
        while (1) {
            pause(); // Wait indefinitely for signals
        }
    }
    else if (choice == 1) {
        // Sender process
        pid_t target_pid;
        printf("Enter receiver's PID: ");
        scanf("%d", &target_pid);

        printf("Sender: Sending SIGUSR1 to PID %d...\n", target_pid);

        if (kill(target_pid, SIGUSR1) == -1) {
            perror("Error sending signal");
        } else {
            printf("Sender: Signal sent successfully!\n");
        }
    }
    else {
        printf("Invalid choice.\n");
    }

    return 0;
}
```

36. Write a program that uses kill() to send signals between related processes (created with fork()).

```
BEGIN

    DECLARE pid as process ID

    // 1. Create a child process using fork()
    pid = fork()
    IF pid < 0 THEN
        PRINT "Fork failed"
        EXIT

    // 2. CHILD PROCESS LOGIC
    IF pid == 0 THEN
        DECLARE signal_handler for SIGUSR1
        REGISTER signal_handler for SIGUSR1

        PRINT "Child: My PID is", GETPID()
        PRINT "Child: Waiting for signal from parent..."
        LOOP forever
            PAUSE()   // Wait for signals
        END LOOP

    // 3. PARENT PROCESS LOGIC
    ELSE
        SLEEP for 3 seconds  // Give time for child to set up
        PRINT "Parent: Sending SIGUSR1 to child..."
        CALL kill(pid, SIGUSR1)

        IF kill() fails THEN
            PRINT "Parent: Failed to send signal"
        ELSE
            PRINT "Parent: Signal sent successfully."
        END IF

        SLEEP for 1 second
        PRINT "Parent: Terminating now."
END
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

// Signal handler for the child process
void handle_signal(int sig) {
    if (sig == SIGUSR1) {
        printf("Child: Received SIGUSR1 signal from parent!\n");
    }
}

int main() {
    pid_t pid;

    // 1. Create a child process
    pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }

    // 2. CHILD PROCESS
    else if (pid == 0) {
        signal(SIGUSR1, handle_signal);  // Register signal handler
        printf("Child: My PID is %d\n", getpid());
        printf("Child: Waiting for signal from parent...\n");

        while (1) {
            pause();  // Wait indefinitely for a signal
        }
    }

    // 3. PARENT PROCESS
    else {
        sleep(3);  // Give child time to initialize
        printf("Parent: Sending SIGUSR1 to child (PID %d)...\n", pid);

        if (kill(pid, SIGUSR1) == -1) {
            perror("Parent: Error sending signal");
        } else {
            printf("Parent: Signal sent successfully.\n");
        }

        sleep(1);
        printf("Parent: Terminating now.\n");

        // Optionally wait for child termination (if we terminate it later)
        // wait(NULL);
    }

    return 0;
}
```

37. Implement a program that uses alarm() and signal handling to require user input within a specified time limit.

```
BEGIN

    DECLARE signal_handler for SIGALRM

    // 1. REGISTER signal handler
    REGISTER signal_handler for SIGALRM

    // 2. PROMPT user for input
    PRINT "You have 5 seconds to enter your name:"

    // 3. SET alarm timer
    CALL alarm(5)

    // 4. READ input from user
    READ input_string

    // 5. CANCEL alarm if input is received in time
    CALL alarm(0)

    PRINT "Hello,", input_string
    PRINT "Input received before timeout."

END

// SIGNAL HANDLER FUNCTION
signal_handler(SIGALRM):
    PRINT "Time’s up! No input received."
    EXIT program
```

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

// Signal handler for SIGALRM
void timeout_handler(int sig) {
    printf("\nTime’s up! No input received within the limit.\n");
    exit(1);  // Terminate program after timeout
}

int main() {
    char name[50];

    // 1. Register signal handler
    signal(SIGALRM, timeout_handler);

    // 2. Prompt for input
    printf("You have 5 seconds to enter your name: ");

    // 3. Start timer
    alarm(5);

    // 4. Attempt to get user input
    if (fgets(name, sizeof(name), stdin) != NULL) {
        // 5. Cancel alarm if input is received
        alarm(0);
        printf("Hello, %sInput received before timeout.\n", name);
    }

    return 0;
}
```

38. Create an alarm clock program using alarm() and signal handlers

```
BEGIN

    DECLARE seconds as integer
    DECLARE signal_handler for SIGALRM

    // 1. REGISTER the signal handler for alarm signal
    REGISTER signal_handler for SIGALRM

    // 2. ASK user for alarm duration
    PRINT "Enter number of seconds for the alarm: "
    READ seconds

    // 3. SET alarm for the given duration
    PRINT "Alarm set for", seconds, "seconds..."
    CALL alarm(seconds)

    // 4. WAIT for the alarm signal
    LOOP forever
        PAUSE()  // Wait for signals
    END LOOP

END

// SIGNAL HANDLER FUNCTION
signal_handler(SIGALRM):
    PRINT "⏰ Alarm ringing! Time’s up!"
    EXIT program
```

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

// Signal handler for alarm
void alarm_handler(int sig) {
    printf("\n⏰ Alarm ringing! Time’s up!\n");
    exit(0);
}

int main() {
    int seconds;

    // 1. Register signal handler for SIGALRM
    signal(SIGALRM, alarm_handler);

    // 2. Ask user for alarm time
    printf("Enter number of seconds for the alarm: ");
    scanf("%d", &seconds);

    // 3. Set the alarm
    printf("Alarm set for %d seconds...\n", seconds);
    alarm(seconds);

    // 4. Wait for signal
    while (1) {
        pause();  // Wait for SIGALRM
    }

    return 0;
}
```

39. Write a program that reports file statistics using stat() (include important fields such as file access permissions, file type, etc.).

```
BEGIN

    DECLARE structure variable fileStat of type struct stat
    DECLARE filename as string

    // 1. ASK user to enter the filename
    PRINT "Enter the filename: "
    READ filename

    // 2. CALL stat() system call
    IF stat(filename, &fileStat) fails THEN
        PRINT "Error: Cannot access file."
        EXIT

    // 3. DISPLAY basic file information
    PRINT "File Size:", fileStat.st_size, "bytes"
    PRINT "Number of Links:", fileStat.st_nlink
    PRINT "File Inode:", fileStat.st_ino

    // 4. DETERMINE file type
    IF S_ISREG(fileStat.st_mode) THEN
        PRINT "File Type: Regular File"
    ELSE IF S_ISDIR(fileStat.st_mode) THEN
        PRINT "File Type: Directory"
    ELSE IF S_ISCHR(fileStat.st_mode) THEN
        PRINT "File Type: Character Device"
    ELSE IF S_ISBLK(fileStat.st_mode) THEN
        PRINT "File Type: Block Device"
    ELSE IF S_ISFIFO(fileStat.st_mode) THEN
        PRINT "File Type: FIFO/PIPE"
    ELSE IF S_ISLNK(fileStat.st_mode) THEN
        PRINT "File Type: Symbolic Link"
    ELSE IF S_ISSOCK(fileStat.st_mode) THEN
        PRINT "File Type: Socket"
    END IF
    // 5. DISPLAY access permissions (user, group, others)
    PRINT "File Permissions:"
    IF fileStat.st_mode & S_IRUSR THEN PRINT "r" ELSE PRINT "-"
    IF fileStat.st_mode & S_IWUSR THEN PRINT "w" ELSE PRINT "-"
    IF fileStat.st_mode & S_IXUSR THEN PRINT "x" ELSE PRINT "-"
    REPEAT same checks for group and others

    // 6. DISPLAY ownership and timestamps
    PRINT "Owner UID:", fileStat.st_uid
    PRINT "Group GID:", fileStat.st_gid
    PRINT "Last Access Time:", ctime(&fileStat.st_atime)
    PRINT "Last Modification Time:", ctime(&fileStat.st_mtime)

END
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>

int main() {
    struct stat fileStat;
    char filename[256];

    // 1. Get filename from user
    printf("Enter the filename: ");
    scanf("%s", filename);

    // 2. Call stat() system call
    if (stat(filename, &fileStat) < 0) {
        perror("Error accessing file");
        exit(EXIT_FAILURE);
    }

    // 3. Display file details
    printf("\nFile: %s\n", filename);
    printf("Size: %ld bytes\n", fileStat.st_size);
    printf("Number of Links: %ld\n", fileStat.st_nlink);
    printf("Inode: %ld\n", fileStat.st_ino);

    // 4. Determine file type
    printf("File Type: ");
    if (S_ISREG(fileStat.st_mode))  printf("Regular File\n");
    else if (S_ISDIR(fileStat.st_mode)) printf("Directory\n");
    else if (S_ISCHR(fileStat.st_mode)) printf("Character Device\n");
    else if (S_ISBLK(fileStat.st_mode)) printf("Block Device\n");
    else if (S_ISFIFO(fileStat.st_mode)) printf("FIFO/Pipe\n");
    else if (S_ISLNK(fileStat.st_mode)) printf("Symbolic Link\n");
    else if (S_ISSOCK(fileStat.st_mode)) printf("Socket\n");
    else printf("Unknown\n");

    // 5. File permissions
    printf("Permissions: ");
    printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n");

    // 6. Display owner, group, and timestamps
    printf("Owner UID: %d\n", fileStat.st_uid);
    printf("Group GID: %d\n", fileStat.st_gid);
    printf("Last Access: %s", ctime(&fileStat.st_atime));
    printf("Last Modification: %s", ctime(&fileStat.st_mtime));

    return 0;
}
```

40. Write a program that reports file statistics using fstat() (include important fields such as file access permissions, file type, etc.).

```
BEGIN

    DECLARE file descriptor fd as integer
    DECLARE structure variable fileStat of type struct stat
    DECLARE filename as string

    // 1. ASK user to enter the filename
    PRINT "Enter the filename: "
    READ filename
```

```
// 2. OPEN the file in read-only mode
fd = open(filename, O_RDONLY)
IF fd < 0 THEN
   PRINT "Error: Cannot open file."
   EXIT

// 3. CALL fstat() system call
IF fstat(fd, &fileStat) fails THEN
   PRINT "Error: Cannot get file status."
   CLOSE fd
   EXIT

// 4. DISPLAY basic file information
PRINT "File Size:", fileStat.st_size, "bytes"
PRINT "Number of Links:", fileStat.st_nlink
PRINT "File Inode:", fileStat.st_ino

// 5. DETERMINE file type
IF S_ISREG(fileStat.st_mode) THEN
   PRINT "File Type: Regular File"
ELSE IF S_ISDIR(fileStat.st_mode) THEN
   PRINT "File Type: Directory"
ELSE IF S_ISCHR(fileStat.st_mode) THEN
   PRINT "File Type: Character Device"
ELSE IF S_ISBLK(fileStat.st_mode) THEN
   PRINT "File Type: Block Device"
ELSE IF S_ISFIFO(fileStat.st_mode) THEN
   PRINT "File Type: FIFO/PIPE"
ELSE IF S_ISLNK(fileStat.st_mode) THEN
   PRINT "File Type: Symbolic Link"
ELSE IF S_ISSOCK(fileStat.st_mode) THEN
   PRINT "File Type: Socket"
END IF

// 6. DISPLAY file permissions (user, group, others)
PRINT "File Permissions:"
IF fileStat.st_mode & S_IRUSR THEN PRINT "r" ELSE PRINT "-"
IF fileStat.st_mode & S_IWUSR THEN PRINT "w" ELSE PRINT "-"
IF fileStat.st_mode & S_IXUSR THEN PRINT "x" ELSE PRINT "-"
REPEAT same checks for group and others

// 7. DISPLAY ownership and timestamps
PRINT "Owner UID:", fileStat.st_uid
PRINT "Group GID:", fileStat.st_gid
```

```
    PRINT "Last Access Time:", ctime(&fileStat.st_atime)
    PRINT "Last Modification Time:", ctime(&fileStat.st_mtime)

    // 8. CLOSE the file descriptor
    CALL close(fd)

END
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>

int main() {
    struct stat fileStat;
    char filename[256];
    int fd;

    // 1. Get filename from user
    printf("Enter the filename: ");
    scanf("%s", filename);

    // 2. Open the file
    fd = open(filename, O_RDONLY);
    if (fd < 0) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    // 3. Call fstat() on the file descriptor
    if (fstat(fd, &fileStat) < 0) {
        perror("Error getting file stats");
        close(fd);
        exit(EXIT_FAILURE);
    }

    // 4. Display file details
    printf("\nFile: %s\n", filename);
    printf("Size: %ld bytes\n", fileStat.st_size);
    printf("Number of Links: %ld\n", fileStat.st_nlink);
    printf("Inode: %ld\n", fileStat.st_ino);
    // 5. Determine file type
    printf("File Type: ");
    if (S_ISREG(fileStat.st_mode))  printf("Regular File\n");
    else if (S_ISDIR(fileStat.st_mode)) printf("Directory\n");
    else if (S_ISCHR(fileStat.st_mode)) printf("Character Device\n");
    else if (S_ISBLK(fileStat.st_mode)) printf("Block Device\n");
    else if (S_ISFIFO(fileStat.st_mode)) printf("FIFO/Pipe\n");
    else if (S_ISLNK(fileStat.st_mode)) printf("Symbolic Link\n");
    else if (S_ISSOCK(fileStat.st_mode)) printf("Socket\n");
    else printf("Unknown\n");

    // 6. Display file permissions
    printf("Permissions: ");
    printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n");

    // 7. Display ownership and timestamps
    printf("Owner UID: %d\n", fileStat.st_uid);
    printf("Group GID: %d\n", fileStat.st_gid);
    printf("Last Access: %s", ctime(&fileStat.st_atime));
    printf("Last Modification: %s", ctime(&fileStat.st_mtime));

    // 8. Close the file descriptor
    close(fd);

    return 0;
}
```

41. Develop a multithreaded chat application in Java or C.

```
BEGIN

    DECLARE integer SERVER_PORT
    DECLARE integer listen_fd
    DECLARE integer client_fd
    DECLARE array clients[MAX_CLIENTS] of integer
    DECLARE integer client_count
    DECLARE mutex clients_mutex
    DECLARE buffer[BUF_SIZE]

    // 1. Initialize server socket
    listen_fd = socket(AF_INET, SOCK_STREAM, 0)
    BIND listen_fd to SERVER_PORT
    LISTEN on listen_fd

    // 2. Accept loop
    LOOP forever
        client_fd = accept(listen_fd)
        IF client_fd < 0 THEN
            PRINT "accept failed" and CONTINUE
        END IF

        LOCK clients_mutex
        IF client_count < MAX_CLIENTS THEN
            ADD client_fd to clients
            INCREMENT client_count
            CREATE a detached thread to run handle_client(client_fd)
        ELSE
            CLOSE client_fd
        END IF
        UNLOCK clients_mutex
    END LOOP

    // Thread function handle_client(fd):
    BEGIN
        DECLARE local buffer[BUF_SIZE]
        LOOP while read from fd > 0
            READ message from fd into buffer
            LOCK clients_mutex
            FOR each client in clients DO
                IF client != fd THEN
                    SEND buffer to client
                END IF
            END FOR
            UNLOCK clients_mutex
        END LOOP

        // Client disconnected: remove from clients list
        LOCK clients_mutex
        REMOVE fd from clients
```

```
        DECREMENT client_count
        CLOSE fd
        UNLOCK clients_mutex
    END

END
```

Server.c

```c
// server.c
// Compile: gcc -pthread -o server server.c
// Run: ./server <port>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>

#include <pthread.h>

#define MAX_CLIENTS 100
#define BUF_SIZE 1024

static int clients[MAX_CLIENTS];
static int client_count = 0;
static pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;
static int listen_fd = -1;

static void broadcast_message(const char *msg, int exclude_fd) {
    pthread_mutex_lock(&clients_mutex);
    for (int i = 0; i < client_count; ++i) {
        int sockfd = clients[i];
        if (sockfd != exclude_fd) {
            ssize_t sent = send(sockfd, msg, strlen(msg), 0);
            (void)sent; // ignore partial-send complexity for simplicity
        }
    }
    pthread_mutex_unlock(&clients_mutex);
}
```

```
static void remove_client(int fd) {
    pthread_mutex_lock(&clients_mutex);
    int found = -1;
    for (int i = 0; i < client_count; ++i) {
        if (clients[i] == fd) {
            found = i;
            break;
        }
    }
    if (found != -1) {
        // shift left
        for (int j = found; j < client_count - 1; ++j)
            clients[j] = clients[j+1];
        client_count--;
    }
    pthread_mutex_unlock(&clients_mutex);
}

void *handle_client(void *arg) {
    int client_fd = *(int *)arg;
    free(arg);

    char buf[BUF_SIZE];
    ssize_t n;

    // Announce join
    snprintf(buf, sizeof(buf), "User %d joined the chat.\n", client_fd);
    broadcast_message(buf, client_fd);

    while ((n = recv(client_fd, buf, sizeof(buf) - 1, 0)) > 0) {
        buf[n] = '\0';
        // Simple sanitation: ensure newline
        // Broadcast received message to others
        broadcast_message(buf, client_fd);
    }

    // Client disconnected or error
    if (n == 0) {
        // Connection closed
        snprintf(buf, sizeof(buf), "User %d left the chat.\n", client_fd);
        broadcast_message(buf, client_fd);
    } else {
        perror("recv");
    }

    close(client_fd);
    remove_client(client_fd);
    return NULL;
}

void handle_sigint(int sig) {
    (void)sig;
    printf("\nShutting down server...\n");

    // close listening socket
    if (listen_fd >= 0) close(listen_fd);

    // close all client sockets
    pthread_mutex_lock(&clients_mutex);
    for (int i = 0; i < client_count; ++i) {
        close(clients[i]);
    }
    pthread_mutex_unlock(&clients_mutex);

    exit(0);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        return 1;
    }

    signal(SIGINT, handle_sigint);

    int port = atoi(argv[1]);

    listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd < 0) {
        perror("socket");
        return 1;
    }

    // Allow quick reuse
    int opt = 1;
    setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

```
struct sockaddr_in srvaddr;
memset(&srvaddr, 0, sizeof(srvaddr));
srvaddr.sin_family = AF_INET;
srvaddr.sin_addr.s_addr = INADDR_ANY;
srvaddr.sin_port = htons(port);

if (bind(listen_fd, (struct sockaddr *)&srvaddr, sizeof(srvaddr)) < 0) {
    perror("bind");
    close(listen_fd);
    return 1;
}

if (listen(listen_fd, 10) < 0) {
    perror("listen");
    close(listen_fd);
    return 1;
}

printf("Chat server listening on port %d\n", port);

while (1) {
    struct sockaddr_in cliaddr;
    socklen_t clilen = sizeof(cliaddr);
    int *connfd_p = malloc(sizeof(int));
    if (!connfd_p) {
        fprintf(stderr, "malloc failed\n");
        continue;
    }

    *connfd_p = accept(listen_fd, (struct sockaddr *)&cliaddr, &clilen);
    if (*connfd_p < 0) {
        perror("accept");
        free(connfd_p);
        continue;
    }

    pthread_mutex_lock(&clients_mutex);
    if (client_count >= MAX_CLIENTS) {
        pthread_mutex_unlock(&clients_mutex);
        const char *msg = "Server full. Try later.\n";
        send(*connfd_p, msg, strlen(msg), 0);
        close(*connfd_p);
        free(connfd_p);
        continue;
        }

        clients[client_count++] = *connfd_p;
        pthread_mutex_unlock(&clients_mutex);

        pthread_t tid;
        pthread_create(&tid, NULL, handle_client, connfd_p);
        pthread_detach(tid);

        char addrstr[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &cliaddr.sin_addr, addrstr, sizeof(addrstr));
        printf("New connection from %s:%d (fd=%d)\n", addrstr, ntohs(cliaddr.sin_port),
*connfd_p);
    }

    // unreachable
    close(listen_fd);
    return 0;
}
```

Client.c

```
// client.c
// Compile: gcc -pthread -o client client.c
// Run: ./client <server-ip> <port>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>

#include <pthread.h>

#define BUF_SIZE 1024

int sockfd = -1;

void *recv_thread(void *arg) {
    (void)arg;
    char buf[BUF_SIZE];
    ssize_t n;
    while ((n = recv(sockfd, buf, sizeof(buf) - 1, 0)) > 0) {
        buf[n] = '\0';
        // Print message from server
        printf("%s", buf);
        fflush(stdout);
    }
    if (n == 0) {
        printf("Server closed connection.\n");
    } else {
        perror("recv");
    }
    exit(0);
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <server-ip> <port>\n", argv[0]);
        return 1;
    }

    const char *server_ip = argv[1];
    int port = atoi(argv[2]);

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("socket");
        return 1;
    }

    struct sockaddr_in srvaddr;
    memset(&srvaddr, 0, sizeof(srvaddr));
    srvaddr.sin_family = AF_INET;
    srvaddr.sin_port = htons(port);

    if (inet_pton(AF_INET, server_ip, &srvaddr.sin_addr) <= 0) {
        fprintf(stderr, "Invalid address: %s\n", server_ip);
        close(sockfd);
        return 1;
    }

    if (connect(sockfd, (struct sockaddr *)&srvaddr, sizeof(srvaddr)) < 0) {
        perror("connect");
        close(sockfd);
        return 1;
    }

    printf("Connected to %s:%d. Type messages and press Enter to send.\n", server_ip, port);

    pthread_t tid;
    pthread_create(&tid, NULL, recv_thread, NULL);
    pthread_detach(tid);

    char input[BUF_SIZE];
    while (fgets(input, sizeof(input), stdin) != NULL) {
        size_t len = strlen(input);
        if (len == 0) continue;
        // send input to server
        ssize_t sent = send(sockfd, input, len, 0);
        if (sent < 0) {
            perror("send");
            break;
        }
    }

    close(sockfd);
    return 0;
}
```

42. Create a program that spawns three threads: one prints even numbers, another prints odd numbers, and the third prints prime numbers.

```
BEGIN

    DECLARE thread IDs t_even, t_odd, t_prime
    DECLARE integer N = 50   // upper limit for numbers

    // 1. DEFINE thread functions
    FUNCTION print_even():
        FOR i FROM 1 TO N DO
            IF i MOD 2 == 0 THEN
                PRINT "Even:", i
                SLEEP for 0.1 seconds
            END IF
        END FOR
    END FUNCTION

    FUNCTION print_odd():
        FOR i FROM 1 TO N DO
            IF i MOD 2 != 0 THEN
                PRINT "Odd:", i
                SLEEP for 0.1 seconds
            END IF
        END FOR
    END FUNCTION

    FUNCTION print_prime():
        FOR i FROM 2 TO N DO
            DECLARE flag = 1
            FOR j FROM 2 TO sqrt(i) DO
                IF i MOD j == 0 THEN
                    flag = 0
                    BREAK
                END IF
            END FOR
            IF flag == 1 THEN
                PRINT "Prime:", i
                SLEEP for 0.1 seconds
            END IF
        END FOR
    END FUNCTION

    // 2. CREATE threads
    CREATE thread t_even to run print_even()
    CREATE thread t_odd to run print_odd()
    CREATE thread t_prime to run print_prime()

    // 3. WAIT for all threads to finish
    JOIN t_even
    JOIN t_odd
    JOIN t_prime

    PRINT "All threads completed."

END
```

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

```
#define N 50  // Upper limit for numbers

// Function to check if a number is prime
int is_prime(int n) {
    if (n <= 1) return 0;
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0)
            return 0;
    }
    return 1;
}

// Thread function for even numbers
void* print_even(void* arg) {
    for (int i = 1; i <= N; i++) {
        if (i % 2 == 0) {
            printf("Even: %d\n", i);
            usleep(100000); // sleep 0.1 seconds
        }
    }
    pthread_exit(NULL);
}

// Thread function for odd numbers
void* print_odd(void* arg) {
    for (int i = 1; i <= N; i++) {
        if (i % 2 != 0) {
            printf("Odd: %d\n", i);
            usleep(100000);
        }
    }
    pthread_exit(NULL);
}

// Thread function for prime numbers
void* print_prime(void* arg) {
    for (int i = 2; i <= N; i++) {
        if (is_prime(i)) {
            printf("Prime: %d\n", i);
            usleep(100000);
        }
    }
    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t t_even, t_odd, t_prime;

    // Create threads
    pthread_create(&t_even, NULL, print_even, NULL);
    pthread_create(&t_odd, NULL, print_odd, NULL);
    pthread_create(&t_prime, NULL, print_prime, NULL);

    // Wait for all threads to complete
    pthread_join(t_even, NULL);
    pthread_join(t_odd, NULL);
    pthread_join(t_prime, NULL);

    printf("All threads completed.\n");
    return 0;
}
```

43. Write a multithreaded program on Linux that uses the pthread library

```
BEGIN

    DECLARE integer NUM_THREADS = 3
    DECLARE array threads[NUM_THREADS] of thread IDs

    // 1. DEFINE thread function
    FUNCTION worker(arg):
        DECLARE thread_id = (integer)arg
        PRINT "Thread", thread_id, "started."
        SLEEP for 1 second
        PRINT "Thread", thread_id, "finished."
        EXIT thread
    END FUNCTION

    // 2. CREATE threads
    FOR i FROM 0 TO NUM_THREADS - 1 DO
        PRINT "Creating thread", i
        CALL pthread_create(&threads[i], NULL, worker, (void*)i)
    END FOR

    // 3. WAIT for all threads to finish
    FOR i FROM 0 TO NUM_THREADS - 1 DO
        CALL pthread_join(threads[i], NULL)
    END FOR
    PRINT "All threads have completed."

END
```

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

#define NUM_THREADS 3

// Thread function
void* worker(void* arg) {
    int thread_id = (int)(size_t)arg;  // Cast argument
    printf("Thread %d: started.\n", thread_id);
    sleep(1); // simulate work
    printf("Thread %d: finished.\n", thread_id);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int i;

    printf("Main: Starting multithreaded program...\n");

    // 1. Create threads
    for (i = 0; i < NUM_THREADS; i++) {
        printf("Main: Creating thread %d\n", i);
        if (pthread_create(&threads[i], NULL, worker, (void*)(size_t)i) != 0) {
            perror("pthread_create failed");
            exit(EXIT_FAILURE);
        }
    }

    // 2. Wait for threads to complete
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
        printf("Main: Joined thread %d\n", i);
    }

    printf("Main: All threads have completed.\n");
    return 0;
}
```

44. Implement the producer–consumer problem using multithreading in Java

```
BEGIN

    DECLARE a shared buffer (queue) with MAX_SIZE = 5

    // 1. DEFINE Producer thread
    CLASS Producer IMPLEMENTS Runnable:
        METHOD run():
            LOOP forever
                LOCK shared buffer
                WHILE buffer is full DO
                    WAIT on buffer
                END WHILE
                PRODUCE an item
                ADD item to buffer
                PRINT "Produced: " + item
                NOTIFY all waiting threads
                UNLOCK buffer
                SLEEP for random short time
            END LOOP

    // 2. DEFINE Consumer thread
    CLASS Consumer IMPLEMENTS Runnable:
        METHOD run():
            LOOP forever
                LOCK shared buffer
                WHILE buffer is empty DO
                    WAIT on buffer
                END WHILE
                REMOVE item from buffer
                PRINT "Consumed: " + item
                NOTIFY all waiting threads
                UNLOCK buffer
                SLEEP for random short time
            END LOOP

    // 3. MAIN PROGRAM
    CREATE shared buffer (Queue)
    CREATE one Producer thread
    CREATE one Consumer thread
    START both threads

END
```

```java
// File: ProducerConsumer.java
import java.util.LinkedList;
import java.util.Queue;

class SharedBuffer {
    private final Queue<Integer> buffer = new LinkedList<>();
    private final int MAX_SIZE = 5;

    // Producer adds items to the buffer
    public synchronized void produce(int value) throws InterruptedException {
        while (buffer.size() == MAX_SIZE) {
            System.out.println("Buffer full! Producer waiting...");
            wait();
        }
        buffer.add(value);
        System.out.println("Produced: " + value);
        notifyAll();  // Notify consumers
    }

    // Consumer removes items from the buffer
    public synchronized int consume() throws InterruptedException {
        while (buffer.isEmpty()) {
            System.out.println("Buffer empty! Consumer waiting...");
            wait();
        }
        int value = buffer.remove();
        System.out.println("Consumed: " + value);
        notifyAll();  // Notify producer
        return value;
    }
}

// Producer thread
class Producer implements Runnable {
    private final SharedBuffer buffer;

    public Producer(SharedBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        int value = 0;
        try {
            while (true) {
                buffer.produce(value++);
                Thread.sleep(500); // simulate time to produce
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

// Consumer thread
class Consumer implements Runnable {
    private final SharedBuffer buffer;

    public Consumer(SharedBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        try {
            while (true) {
                buffer.consume();
                Thread.sleep(800); // simulate time to consume
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

// Main class
public class ProducerConsumer {
    public static void main(String[] args) {
        SharedBuffer buffer = new SharedBuffer();

        Thread producerThread = new Thread(new Producer(buffer), "Producer");
        Thread consumerThread = new Thread(new Consumer(buffer), "Consumer");

        producerThread.start();
        consumerThread.start();
    }
}
```

45. Write a shell script that implements a simple calculator.

```
BEGIN

   DISPLAY "Simple Calculator"
   DISPLAY "------------------"

   // 1. ASK user for two numbers
   PRINT "Enter first number: "
   READ num1
   PRINT "Enter second number: "
   READ num2

   // 2. SHOW operation menu
   PRINT "Select operation:"
   PRINT "1. Addition"
   PRINT "2. Subtraction"
   PRINT "3. Multiplication"
   PRINT "4. Division"

   // 3. READ user choice
   READ choice

   // 4. PERFORM corresponding operation using case statement
   CASE choice OF
      1) result = num1 + num2
        PRINT "Result =", result
      2) result = num1 - num2
        PRINT "Result =", result
      3) result = num1 * num2
        PRINT "Result =", result
      4) IF num2 == 0 THEN
           PRINT "Error: Division by zero not allowed."
        ELSE
           result = num1 / num2
           PRINT "Result =", result
        END IF
      DEFAULT:
        PRINT "Invalid choice."
   END CASE

END
```

```
#!/bin/bash
# Simple Calculator Script
```

```
echo "Simple Calculator"
echo "------------------"

# 1. Read two numbers
read -p "Enter first number: " num1
read -p "Enter second number: " num2

# 2. Display menu
echo "Select an operation:"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"

# 3. Read user choice
read -p "Enter your choice [1-4]: " choice

# 4. Perform operation
case $choice in
   1)
      result=$(echo "$num1 + $num2" | bc)
      echo "Result = $result"
      ;;
   2)
      result=$(echo "$num1 - $num2" | bc)
      echo "Result = $result"
      ;;
   3)
      result=$(echo "$num1 * $num2" | bc)
      echo "Result = $result"
      ;;
   4)
      if [ "$num2" == "0" ]; then
         echo "Error: Division by zero not allowed."
      else
         result=$(echo "scale=2; $num1 / $num2" | bc)
         echo "Result = $result"
      fi
      ;;
   *)
      echo "Invalid choice."
      ;;
esac
```

46. Implement a digital clock using a shell script.

```
BEGIN

   PRINT "Digital Clock Started (Press Ctrl + C to Stop)"

   // 1. LOOP infinitely
   LOOP forever
      CLEAR screen
      GET current_time = output of date command in format HH:MM:SS
      PRINT "Current Time: " + current_time
      SLEEP for 1 second
   END LOOP

END
```

```
#!/bin/bash
# Digital Clock Script

echo "Digital Clock Started (Press Ctrl + C to Stop)"
sleep 1

while true
do
   clear
   # Display current time in HH:MM:SS format
   echo "================================"
   date +"        %H : %M : %S"
   echo "================================"
   sleep 1
done
```

47. Write a shell script that checks whether the system is connected to a network by using the ping command.

```
BEGIN

   DECLARE host as string = "8.8.8.8"      // Google's public DNS server

   PRINT "Checking network connectivity..."

   // 1. PING the host once and check the result
   EXECUTE command: ping -c 1 -W 2 host
```

```
    // 2. CHECK the exit status of ping command
    IF exit status == 0 THEN
        PRINT "Network is connected."
    ELSE
        PRINT "Network is not connected."
    END IF

END
```

```bash
#!/bin/bash
# Script to check network connectivity using ping

HOST="8.8.8.8"   # Google DNS (reliable for connectivity check)
echo "Checking network connectivity..."

# Try to ping once (-c 1) with a 2-second timeout (-W 2)
if ping -c 1 -W 2 $HOST > /dev/null 2>&1
then
    echo "✅ Network is connected."
else
    echo "❌ Network is not connected."
fi
```

48. Write a shell script to sort ten given numbers in ascending order.

```
BEGIN

    DECLARE array numbers[10]

    // 1. READ 10 numbers from user
    PRINT "Enter 10 numbers:"
    FOR i FROM 1 TO 10 DO
        READ numbers[i]
    END FOR

    // 2. SORT numbers in ascending order
    FOR i FROM 1 TO 9 DO
        FOR j FROM i+1 TO 10 DO
            IF numbers[i] > numbers[j] THEN
                SWAP numbers[i] and numbers[j]
            END IF
        END FOR
    END FOR

    // 3. DISPLAY sorted numbers
    PRINT "Numbers in ascending order:"
    FOR i FROM 1 TO 10 DO
        PRINT numbers[i]
    END FOR

END
```

```bash
#!/bin/bash
# Script to sort 10 numbers in ascending order

echo "Enter 10 numbers:"

# 1. Read 10 numbers into an array
for ((i=0; i<10; i++))
do
    read num
    numbers[i]=$num
done

# 2. Sort using bubble sort
for ((i=0; i<10; i++))
do
    for ((j=i+1; j<10; j++))
    do
        if [ ${numbers[i]} -gt ${numbers[j]} ]
        then
            # Swap numbers
            temp=${numbers[i]}
            numbers[i]=${numbers[j]}
            numbers[j]=$temp
        fi
    done
done

# 3. Display sorted numbers
echo "Numbers in ascending order:"
for ((i=0; i<10; i++))
do
    echo -n "${numbers[i]} "
done
echo
```

49. Create a program (or script) that prints —Hello World‖ with bold, blinking, and colored (red, blue, etc.) text effects.

```
BEGIN

   // ANSI Escape Codes:
   // \033[ - starts escape sequence
   // 1 - Bold
   // 5 - Blink
   // 31 - Red, 34 - Blue, etc.
   // 0 - Reset formatting

   // 1. PRINT “Hello World” in bold red color
   PRINT "\033[1;31mHello World (Bold Red)\033[0m"

   // 2. PRINT “Hello World” in bold blue color
   PRINT "\033[1;34mHello World (Bold Blue)\033[0m"

   // 3. PRINT “Hello World” with blinking green text
   PRINT "\033[5;32mHello World (Blinking Green)\033[0m"

   // 4. PRINT “Hello World” with combined bold + blinking + yellow
   PRINT "\033[1;5;33mHello World (Bold + Blinking Yellow)\033[0m"

END
```

```
#!/bin/bash
# Script to display "Hello World" with text effects

echo -e "\033[1;31mHello World (Bold Red)\033[0m"
sleep 1
echo -e "\033[1;34mHello World (Bold Blue)\033[0m"
sleep 1
echo -e "\033[5;32mHello World (Blinking Green)\033[0m"
sleep 1
echo -e "\033[1;5;33mHello World (Bold + Blinking Yellow)\033[0m"
sleep 1

# Reset colors
echo -e "\033[0m"
```

50. Write a shell script that checks whether a specified file exists in a given folder or drive.

```
BEGIN

   // 1. PROMPT user for folder path and file name
```

```
    PRINT "Enter the folder (directory) path:"
    READ folder
    PRINT "Enter the file name:"
    READ filename

    // 2. COMBINE folder and filename into a full path
    fullpath = folder + "/" + filename

    // 3. CHECK if the file exists
    IF file exists at fullpath THEN
        PRINT "✅ File exists at given location."
    ELSE
        PRINT "❌ File does not exist in the specified folder."
    END IF

END
```

```
#!/bin/bash
# Script to check whether a specified file exists in a given folder

echo "Enter the folder (directory) path:"
read folder

echo "Enter the file name:"
read filename

fullpath="$folder/$filename"

# Check if the file exists
if [ -f "$fullpath" ]; then
    echo "✅ File '$filename' exists in folder '$folder'."
else
    echo "❌ File '$filename' does NOT exist in folder '$folder'."
fi
```