CS3461 - Operating Systems Laboratory

Morning [AM] Batch

---

1. **a.** Using the appropriate UNIX Command check whether the remote host is responding well or not.

✅ **Shell Script:**

```
#!/bin/bash

HOST="google.com"

if ping -c 1 $HOST > /dev/null 2>&1; then
   echo "$HOST is responding."
else
   echo "$HOST is not responding."
fi
```

▶️ **Sample Output (when the host is reachable):**

```
google.com is responding.
```

❌ **Sample Output (when the host is not reachable or down):**

```
google.com is not responding.
```

To run it:

1. Save the script to a file, e.g. check_host.sh
2. Make it executable: chmod +x check_host.sh
3. Run it: ./check_host.sh

b. Write a Shell program to count the number of vowels in a line of text.

Here is a **simple shell script** to count the number of vowels in a line of text using basic UNIX tools:

✅ **Shell Script:**

```
#!/bin/bash

echo "Enter a line of text:"
read line

count=$(echo "$line" | grep -o -i "[aeiou]" | wc -l)

echo "Number of vowels: $count"
```

🔍 **Explanation:**

- grep -o -i "[aeiou]": Finds all vowels (case-insensitive, one per line).
- wc -l: Counts the number of lines (i.e., vowels).

▶️ **Sample Output:**

```
Enter a line of text:
Hello World
Number of vowels: 3
```

This is a short and effective way to count vowels in shell scripting.

c. Write a C program for implementing Inter-process communication using shared memory concept.

Here's a **simple and short C program** demonstrating **Inter-Process Communication (IPC)** using **shared memory** with basic logic.

✅ **Single File C Program for Shared Memory IPC**

This program forks a child process. The **parent writes** a message to shared memory, and the **child reads** it.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <string.h>

int main() {
    // Create shared memory segment
    int shmid = shmget(IPC_PRIVATE, 1024, IPC_CREAT | 0666);
    char *shm = (char*) shmat(shmid, NULL, 0);

    if (fork() == 0) {
        // Child process - Reader
        sleep(1); // Wait for parent to write
        printf("Child read: %s\n", shm);
        shmdt(shm);
    } else {
        // Parent process - Writer
        strcpy(shm, "Hello from parent!");
        shmdt(shm);
        wait(NULL); // Wait for child
        shmctl(shmid, IPC_RMID, NULL); // Remove shared memory
    }

    return 0;
}
```

🧠 **Short Logic Explanation:**

- shmget: Creates a shared memory segment.
- shmat: Attaches it to the process's address space.
- fork: Creates a child process.
- strcpy: Parent writes to shared memory.
- printf: Child reads from shared memory.
- shmdt: Detaches.
- shmctl: Deletes the shared memory.

▶️ Sample Output:

```
Child read: Hello from parent!
```

This compact example keeps the logic clear and is ideal for educational or demonstration purposes.

2. a. Using the appropriate UNIX command print the last 10 lines of user specified file to standard output.

Here's a **simple shell script** that prints the **last 10 lines** of a **user-specified file** using the tail command.

✅ **Shell Script:**

```
#!/bin/bash

echo "Enter filename:"
read filename

tail -n 10 "$filename"
```

🔍 **Explanation:**

- read filename: Reads the filename from the user.
- tail -n 10 "$filename": Prints the last 10 lines of the file.

▶️ **Sample Output:**

```
Enter filename:
example.txt
(Line 1)
(Line 2)
...
(Line 10 from the end of example.txt)
```

This script uses short logic and relies on the built-in tail command — efficient and widely supported.

b. Write a shell program to find the sum and average of four integers.

Here's a **simple shell script** to calculate the **sum and average** of four integers using short logic:

✅ **Shell Script:**

```
#!/bin/bash

echo "Enter four integers:"
read a b c d

sum=$((a + b + c + d))
avg=$((sum / 4))

echo "Sum: $sum"
echo "Average: $avg"
```

▶️ **Sample Output:**

```
Enter four integers:
10 20 30 40
Sum: 100
Average: 25
```

💡 **Notes:**

- read a b c d reads 4 space-separated integers.
- $((...)) is used for arithmetic in bash.

This is short, clear, and works on any standard UNIX/Linux shell.

c. Write a C Program for simulating a deadlock detection model.

Here's a **simpler and more straightforward** C program to simulate deadlock detection. It assumes:

- A fixed number of processes and resources.
- Input for allocation, request, and available resources is hardcoded for simplicity.
- It checks which processes are deadlocked.

Simple Deadlock Detection Program in C

```
#include <stdio.h>

#define P 3  // Number of processes
#define R 2  // Number of resource types

int main() {
    int allocation[P][R] = {
        {1, 0},
        {0, 1},
        {1, 1}
    };

    int request[P][R] = {
        {0, 1},
        {1, 0},
        {0, 0}
    };

    int available[R] = {0, 0};
    int finished[P] = {0};  // Track if process can finish
    int i, j, progress;

    // Work initially = available
    int work[R];
    for (i = 0; i < R; i++) {
        work[i] = available[i];
    }
```

```
do {
    progress = 0;
    for (i = 0; i < P; i++) {
        if (!finished[i]) {
            // Check if request[i] <= work
            int canFinish = 1;
            for (j = 0; j < R; j++) {
                if (request[i][j] > work[j]) {
                    canFinish = 0;
                    break;
                }
            }
            if (canFinish) {
                // Process can finish; release allocated resources
                for (j = 0; j < R; j++) {
                    work[j] += allocation[i][j];
                }
                finished[i] = 1;
                progress = 1;
            }
        }
    }
} while (progress);

// Print deadlocked processes
printf("Deadlocked Processes: ");
int deadlockFound = 0;
for (i = 0; i < P; i++) {
    if (!finished[i]) {
        printf("P%d ", i);
        deadlockFound = 1;
    }
}
if (!deadlockFound) {
    printf("None");
}
```

```
    printf("\n");

    return 0;
}
```

What this program does:

- It tries to find a process whose requests can be satisfied with the currently available resources (work).
- If such a process is found, it "finishes" and releases its allocated resources back to work.
- It repeats until no more progress can be made.
- Any process not finished is considered deadlocked.

Sample Output:

```
Deadlocked Processes: None
```

3. a. Give the correct UNIX commands that display a list of users who are currently logged in to a computer and which prints the calendar of the current month and year.

Sure! Here's a **simple shell script** that demonstrates both commands together with output:

```
#!/bin/bash

echo "Users currently logged in:"
who

echo -e "\nCurrent month's calendar:"
cal
```

How to run:

1. Save the above in a file, e.g. userinfo.sh
2. Make it executable: chmod +x userinfo.sh
3. Run it: ./userinfo.sh

Sample Output:

```
Users currently logged in:
john     tty1         May 27 09:00
alice    pts/0        May 27 09:10

Current month's calendar:
     May 2025
Su Mo Tu We Th Fr Sa
           1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

b. Write a shell program to compute simple interest and compound interest.

Here's a **simple shell script** to compute both **Simple Interest (SI)** and **Compound Interest (CI)**:

✅ **Shell Script:**

```
#!/bin/bash

echo "Enter Principal:"
read p
echo "Enter Rate of Interest (in %):"
read r
echo "Enter Time (in years):"
read t

# Simple Interest formula: SI = (P * R * T) / 100
si=$(echo "scale=2; ($p * $r * $t) / 100" | bc)

# Compound Interest formula: CI = P * (1 + R/100)^T - P
ci=$(echo "scale=2; $p * (1 + $r / 100)^$t - $p" | bc -l)

echo "Simple Interest: $si"
echo "Compound Interest: $ci"
```

🧠 **Explanation:**

- bc is used for floating-point arithmetic.
- scale=2 sets 2 decimal precision.
- bc -l loads math library to support power (^).

▶️ **Sample Output:**

```
Enter Principal:
1000
Enter Rate of Interest (in %):
5
Enter Time (in years):
2
```

```
Simple Interest: 100.00
Compound Interest: 102.50
```

c. Write a C program for implementing Round Robin Scheduling algorithm.

Here's the **simplest and shortest C program** to simulate **Round Robin Scheduling** using minimal logic — great for understanding the basic flow.

✅ **Minimal Round Robin C Program**

```
#include <stdio.h>

int main() {
    int bt[3] = {5, 15, 4};  // Burst times for 3 processes
    int rt[3] = {5, 15, 4};  // Remaining times
    int quantum = 4, time = 0, done;

    do {
        done = 1;
        for (int i = 0; i < 3; i++) {
            if (rt[i] > 0) {
                done = 0;
                int t = (rt[i] > quantum) ? quantum : rt[i];
                time += t;
                rt[i] -= t;
                printf("P%d executed for %d units, total time: %d\n", i + 1, t, time);
            }
        }
    } while (!done);

    return 0;
}
```

▶️ **Sample Output:**

```
P1 executed for 4 units, total time: 4
P2 executed for 4 units, total time: 8
P3 executed for 4 units, total time: 12
P1 executed for 1 units, total time: 13
P2 executed for 4 units, total time: 17
P2 executed for 4 units, total time: 21
```

```
P2 executed for 3 units, total time: 24
```

🧠 **What it does:**

- Three processes with fixed burst times
- Runs each in order for up to quantum time
- Stops when all are done

4. **a.** Using the desired UNIX command rename a file of the user to a new name.

To **rename a file in UNIX**, you use the mv (move) command.

✅ **Command Syntax:**

```
mv old_filename new_filename
```

▶️ **Example:**

```
mv report.txt summary.txt
```

This renames report.txt to summary.txt.

🧠 **Explanation:**

- mv can be used to move or rename files.
- If both filenames are in the same directory, it acts as a rename.

✅ **Shell Script Example:**

```
#!/bin/bash

echo "Enter the current filename:"
read old

echo "Enter the new filename:"
read new

mv "$old" "$new"
echo "File renamed from $old to $new"
```

Here's how the **shell script** and its **output** would look when run:

✅ **Shell Script:**

```
#!/bin/bash

echo "Enter the current filename:"
read old

echo "Enter the new filename:"
read new
```

```
mv "$old" "$new"
echo "File renamed from $old to $new"
```

▶️ **Sample Output (Terminal Run):**

```
Enter the current filename:
report.txt
Enter the new filename:
summary.txt
File renamed from report.txt to summary.txt
```

💡 **Before and After:**

- **Before:** File report.txt exists.
- **After:** File report.txt is renamed to summary.txt.

b. Write a Shell program to find the area and circumference of a circle.

Here's a **simple shell script** to calculate the **area** and **circumference** of a circle given the radius:

✅ **Shell Script:**

```
#!/bin/bash

echo "Enter radius of the circle:"
read r

pi=3.1416

area=$(echo "scale=4; $pi * $r * $r" | bc)
circumference=$(echo "scale=4; 2 * $pi * $r" | bc)

echo "Area of circle: $area"
echo "Circumference of circle: $circumference"
```

🔍 **Explanation:**

- Uses bc for floating-point calculations.
- scale=4 sets precision to 4 decimal places.
- Area formula: $\pi * r^2$
- Circumference formula: $2 * \pi * r$

▶️ **Sample Output:**

```
Enter radius of the circle:
5
Area of circle: 78.5400
Circumference of circle: 31.4160
```

c. Write a program for implementing Priority Scheduling algorithm.

Here's the **absolute minimal** Priority Scheduling example in C — no arrays for waiting/turnaround, just print order based on priority:

```
#include <stdio.h>

int main() {
   int bt[] = {10, 5, 8}, pr[] = {2, 1, 3}, i, j, n=3, temp;
   int p[] = {1,2,3};

   for(i=0;i<n-1;i++)
      for(j=0;j<n-1-i;j++)
         if(pr[j]>pr[j+1]){
            temp=pr[j]; pr[j]=pr[j+1]; pr[j+1]=temp;
            temp=bt[j]; bt[j]=bt[j+1]; bt[j+1]=temp;
            temp=p[j]; p[j]=p[j+1]; p[j+1]=temp;
         }

   int wt=0;
   printf("P\tBT\tWT\tTAT\n");
   for(i=0;i<n;i++){
      printf("P%d\t%d\t%d\t%d\n",p[i],bt[i],wt,wt+bt[i]);
      wt+=bt[i];
   }

   return 0;
}
```

Output:

```
P   BT      WT      TAT
P2  5       0       5
P1  10      5       15
P3  8       15      23
```

This is the shortest **non-preemptive priority scheduling** code that sorts by priority and calculates waiting & turnaround times on the fly.

5. a. Write a Shell program to display student grades.

Here's a **simple shell script** that reads a student's marks and displays their grade based on a basic grading scale:

✅ **Shell Program to Display Student Grades**

```
#!/bin/bash

echo "Enter student's marks (0-100):"
read marks

if [ $marks -ge 90 ]; then
   grade="A"
elif [ $marks -ge 80 ]; then
   grade="B"
elif [ $marks -ge 70 ]; then
   grade="C"
elif [ $marks -ge 60 ]; then
   grade="D"
else
   grade="F"
fi

echo "Grade: $grade"
```

▶️ **Sample Run:**

```
Enter student's marks (0-100):
85
Grade: B
```

b. Write a C program for implementing the concept of synchronization in threads.

Here's a **super simple and very short** C program showing basic thread synchronization with minimal code:

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

int counter = 0;
pthread_mutex_t lock;

void* increment(void* arg) {
    pthread_mutex_lock(&lock);
    counter++;  // critical section
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);

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

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

    printf("Counter = %d\n", counter);

    pthread_mutex_destroy(&lock);
    return 0;
}
```

What it does:

- Two threads increment a shared counter **once** each.

- Mutex lock ensures no conflict.

Output:

```
Counter = 2
```

This is the **smallest example** showing synchronization using a mutex lock.

6. **a.** Use the appropriate UNIX command for printing the manual page of any given specific command.

Here's a **simple shell script** that takes a command name from the user and displays its manual page using the man command:

```
#!/bin/bash

echo "Enter the command to view its manual page:"
read cmd

man $cmd
```

How to run:

1. Save as showman.sh
2. Make executable: chmod +x showman.sh
3. Run: ./showman.sh

Sample Output:

```
Enter the command to view its manual page:
ls
```

This will open the manual page for ls.

b. Write a Shell program to generate Fibonacci series.

Here's a simple shell script to generate a Fibonacci series up to n terms:

```
#!/bin/bash

echo "Enter number of terms:"
read n

a=0
b=1

echo "Fibonacci Series:"
for (( i=0; i<n; i++ ))
do
  echo -n "$a "
  fn=$((a + b))
  a=$b
  b=$fn
done
echo
```

How it works:

- Reads number of terms (n) from user.
- Starts with 0 and 1.
- Prints the series using a loop.

Sample output:

```
Enter number of terms:
7
Fibonacci Series:
0 1 1 2 3 5 8
```

c. Write a C program for implementing the concept of paging in memory management.

Here's a **shorter and simpler** version of a paging simulation in C:

```
#include <stdio.h>

#define PAGE_SIZE 256

int main() {
   int page_table[4] = {2, 0, 3, 1};
   int logical, page, offset, physical;

   printf("Enter logical address (0-1023): ");
   scanf("%d", &logical);

   page = logical / PAGE_SIZE;
   offset = logical % PAGE_SIZE;
   physical = page_table[page] * PAGE_SIZE + offset;

   printf("Physical address: %d\n", physical);
   return 0;
}
```

How it works:

- Uses fixed page table for 4 pages.
- Calculates physical address by simple formula.
- No error checks to keep it minimal.

Sample Input/Output:

```
Enter logical address (0-1023): 300
Physical address: 44
```

7. a. Write the appropriate UNIX commands for printing the output to a terminal and to print the processes running in a system.

Here's a **short shell script** that:

1. Prints output to the terminal
2. Shows running processes

✅ **Shell Script:**

```
#!/bin/bash

echo "Displaying message to terminal:"
echo "Hello, Terminal!"

echo -e "\nRunning processes:"
ps
```

▶️ **Sample Output:**

```
Displaying message to terminal:
Hello, Terminal!

Running processes:
  PID TTY          TIME CMD
 1001 pts/0    00:00:00 bash
 1020 pts/0    00:00:00 ps
```

You can run this by saving it as process_view.sh, then:

```
chmod +x process_view.sh
./process_view.sh
```

b. Write a shell program to check whether the given number is positive or negative.

Here's a simple shell script to check whether a given number is positive, negative, or zero:

✅ **Shell Script: Check Positive or Negative**

```
#!/bin/bash

echo "Enter a number:"
read num

if [ $num -gt 0 ]; then
   echo "The number is positive."
elif [ $num -lt 0 ]; then
   echo "The number is negative."
else
   echo "The number is zero."
fi
```

▶️ **Sample Output:**

```
Enter a number:
-3
The number is negative.
```

This script uses if, elif, and else to compare the number.

c. Write a C program for implementing LRU page replacement algorithm.

Sure! Here's a **small and simple C program** to simulate the **LRU (Least Recently Used) Page Replacement** algorithm. It's minimal, easy to understand, and suitable for learning.

✅ **Simple LRU Page Replacement – C Code**

```
#include <stdio.h>

int main() {
    int pages[] = {1, 2, 3, 2, 4, 1, 5}, n = 7;
    int frames[3] = {-1, -1, -1}, time[3] = {0}, count = 0, faults = 0;
    int i, j, pos, used;

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

        for (j = 0; j < 3; j++) {
            if (frames[j] == pages[i]) {
                time[j] = ++count;
                used = 1;
                break;
            }
        }

        if (!used) {
            pos = 0;
            for (j = 1; j < 3; j++)
                if (time[j] < time[pos]) pos = j;

            frames[pos] = pages[i];
            time[pos] = ++count;
            faults++;
        }

        // Print current frame content
        printf("Frames: ");
        for (j = 0; j < 3; j++)
```

```
            printf("%d ", frames[j]);
        printf("\n");
    }

    printf("Total Page Faults = %d\n", faults);
    return 0;
}
```

▶️ **Sample Output:**

```
Frames: 1 -1 -1
Frames: 1 2 -1
Frames: 1 2 3
Frames: 1 2 3
Frames: 4 2 3
Frames: 4 1 3
Frames: 4 1 5
Total Page Faults = 5
```

✨ **Features:**

- Fixed page reference string and 3 frames.
- Simple time-tracking logic for LRU.
- Easy to expand with user input if needed.

8. a. Write a shell script to display the digits which are in odd position in a given number.

Here's a **simple shell script** that displays the **digits in odd positions** of a given number (from left to right):

✅ **Shell Script: Digits in Odd Positions**

```
#!/bin/bash

echo "Enter a number:"
read num

len=${#num}
echo "Digits in odd positions:"

for (( i=0; i<len; i++ ))
do
  if (( i % 2 == 0 )); then   # 0-based index: 0, 2, 4 are odd positions (1st, 3rd, 5th...)
    echo -n "${num:$i:1} "
  fi
done

echo
```

▶️ **Sample Run:**

```
Enter a number:
493628
Digits in odd positions:
4 3 2
(1st = 4, 3rd = 3, 5th = 2)
```

b. Write a C program for implementing Optimal page replacement algorithm.

Here's a **short and easy-to-understand C program** that implements the **Optimal Page Replacement Algorithm** with minimal code and logic, suitable for quick learning.

✅ **Minimal Optimal Page Replacement C Program**

```
#include <stdio.h>

int find_optimal(int pages[], int frames[], int n, int index, int f) {
    int i, j, farthest = index, pos = -1, used;

    for (i = 0; i < f; i++) {
        used = 0;
        for (j = index; j < n; j++) {
            if (frames[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    pos = i;
                }
                used = 1;
                break;
            }
        }
        if (!used) return i; // Not used again
    }
    return (pos == -1) ? 0 : pos;
}

int main() {
    int pages[] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};
    int n = 12, f = 3, frames[3] = {-1, -1, -1}, faults = 0, i, j, hit;

    for (i = 0; i < n; i++) {
        hit = 0;
        for (j = 0; j < f; j++)
            if (frames[j] == pages[i]) hit = 1;
```

```
        if (!hit) {
            int pos = -1;
            for (j = 0; j < f; j++)
                if (frames[j] == -1) pos = j;

            if (pos == -1) pos = find_optimal(pages, frames, n, i + 1, f);
            frames[pos] = pages[i];
            faults++;
        }

        printf("Frames: ");
        for (j = 0; j < f; j++) printf("%d ", frames[j]);
        printf("\n");
    }

    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

🔹 **Sample Output:**

```
Frames: 1 -1 -1
Frames: 1 2 -1
Frames: 1 2 3
Frames: 4 2 3
Frames: 4 2 3
Frames: 4 1 3
Frames: 5 1 3
Frames: 5 1 3
Frames: 5 2 3
Frames: 5 2 1
Frames: 4 2 1
Frames: 4 2 5
Total Page Faults: 9
```

💡 **Key Points:**

- Replaces the page **not used for the longest time** in the future.

- Best-case algorithm for page replacement but needs future knowledge.
- Uses fixed 3-page frames and 12-page references for simplicity.

9. a. Write the procedure and show the installation of any Linux flavor as a guest Operating system on top of Windows operating system.

Here is a **simple and clear procedure** to install a Linux distribution (like **Ubuntu**) as a **guest OS** on a **Windows system** using **VirtualBox**.

✅ **Installing Linux (Ubuntu) on Windows using VirtualBox**

🔧 **Requirements:**

- A Windows computer
- VirtualBox (free)
- Ubuntu ISO file (Download Ubuntu)

📝 **Step-by-Step Procedure**

Step 1: Install VirtualBox on Windows

1. Go to https://www.virtualbox.org/
2. Download the latest version for Windows.
3. Run the installer and follow the prompts to install.

Step 2: Download Ubuntu ISO

1. Visit https://ubuntu.com/download/desktop
2. Choose Ubuntu LTS version (e.g., 22.04 LTS)
3. Download the ISO file (around 2–3 GB)

Step 3: Create a New Virtual Machine

1. Open **VirtualBox** and click **"New"**
2. Enter name: e.g., Ubuntu
3. Type: Linux, Version: Ubuntu (64-bit)
4. Click **Next**

Step 4: Allocate RAM

- Choose memory (minimum **2048 MB**, recommended **4096 MB** or more)
- Click **Next**

Step 5: Create a Virtual Hard Disk

1. Choose **Create a virtual hard disk now**
2. Click **Create**
3. Choose **VDI** (VirtualBox Disk Image)
4. Storage: **Dynamically allocated**

5. Size: Set at least **20 GB**
6. Click **Create**

Step 6: Mount Ubuntu ISO

1. Select your new VM, click **Settings**
2. Go to **Storage**
3. Click on **Empty** under Controller: IDE
4. Click the **disc icon** → Choose a disk file
5. Select your downloaded **Ubuntu ISO**

Step 7: Start the VM and Install Ubuntu

1. Click **Start** to boot the VM
2. Ubuntu installer will load
3. Follow on-screen steps:
    - Select language, keyboard
    - Choose "Install Ubuntu"
    - Choose **Erase disk and install Ubuntu** (safe inside VM)
    - Set your username and password
4. Click **Install Now**

Step 8: Complete Installation

1. After install, Ubuntu will prompt you to restart
2. Before restarting, **remove the ISO**:
    - Go to **Devices > Optical Drives > Remove disk from virtual drive**
3. Press **Enter** to reboot

✅ **Done!**

You now have **Ubuntu Linux running as a guest OS on top of Windows** using VirtualBox.

b. Write the UNIX command needed for creating a directory.

Here's the complete **small C program** to create a directory using mkdir(), along with a **sample output**.

✅ **C Program: Create a Directory**

```
#include <stdio.h>
#include <sys/stat.h>
int main() {
    if (mkdir("myfolder", 0777) == 0)
        printf("Directory created successfully.\n");
    else
        perror("mkdir");  // Shows error if creation fails
    return 0;
}
```

▶️ **How to Compile and Run:**

1. Save the code as create_dir.c
2. Open your terminal and run:

```
gcc create_dir.c -o create_dir
./create_dir
```

📤 **Sample Output:**

Case 1: Directory doesn't exist

```
Directory created successfully.
```

Case 2: Directory already exists

```
mkdir: File exists
```

This is the shortest and cleanest way to programmatically create a directory in a UNIX/Linux environment using C.

10. Write a C program for implementing FCFS disk scheduling strategy.

Here's a **simple C program** implementing the **FCFS (First-Come, First-Served)** disk scheduling algorithm.

✅ **FCFS Disk Scheduling C Program**

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

int main() {
    int n, i, head, seek = 0;

    printf("Enter number of disk requests: ");
    scanf("%d", &n);

    int requests[n];
    printf("Enter disk requests sequence:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &requests[i]);

    printf("Enter initial head position: ");
    scanf("%d", &head);

    for (i = 0; i < n; i++) {
        seek += abs(requests[i] - head);
        head = requests[i];
    }

    printf("Total seek time (distance) = %d\n", seek);
    return 0;
}
```

▶️ **How It Works:**

- Takes input of request queue and initial head position.
- Calculates total head movement by serving requests in order.
- Prints the total seek distance (head movement).

Sample Run:

```
Enter number of disk requests: 4
Enter disk requests sequence:
98 183 37 122
Enter initial head position: 53
Total seek time (distance) = 236
```

11. Write a C program to implement SSTF disk scheduling strategy.

Here's a simple and clear **C program to implement the SSTF (Shortest Seek Time First)** disk scheduling algorithm:

✅ **SSTF Disk Scheduling in C**

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

int main() {
    int n, head, i, j, closest, distance, total_seek = 0;

    printf("Enter number of disk requests: ");
    scanf("%d", &n);

    int requests[n], visited[n];
    printf("Enter disk requests sequence:\n");
    for(i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
        visited[i] = 0; // mark all as unvisited
    }

    printf("Enter initial head position: ");
    scanf("%d", &head);

    for(i = 0; i < n; i++) {
        closest = -1;
        distance = 1000000; // large number

        for(j = 0; j < n; j++) {
            if (!visited[j]) {
                int diff = abs(head - requests[j]);
                if(diff < distance) {
                    distance = diff;
                    closest = j;
                }
            }
```

```
        }

        visited[closest] = 1;
        total_seek += distance;
        head = requests[closest];

        printf("Serviced request: %d\n", requests[closest]);
    }

    printf("Total seek time = %d\n", total_seek);
    return 0;
}
```

▶️ **How it works:**

- Keeps track of unvisited requests.
- Each time picks the request closest to current head position.
- Moves the head and adds to total seek time.

📩 **Sample Output:**

```
Enter number of disk requests: 4
Enter disk requests sequence:
98 183 37 122
Enter initial head position: 53
Serviced request: 37
Serviced request: 98
Serviced request: 122
Serviced request: 183
Total seek time = 186
```

12.a. Give the UNIX command that for getting information from the DNS server.

Here's a **small C program** that uses the system() function to run the dig command and get DNS information, along with a **sample output**.

✅ **C Program to Query DNS Using dig Command**

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

int main() {
    char domain[100];

    printf("Enter a domain name: ");
    scanf("%s", domain);

    char command[150];
    snprintf(command, sizeof(command), "dig %s", domain);

    printf("\nGetting DNS information for: %s\n\n", domain);
    system(command);  // Executes the dig command

    return 0;
}
```

💡 **How It Works:**

- Takes a domain name as input
- Constructs a dig command
- Executes the command using system() to display DNS records

▶️ **Sample Run:**

```
Enter a domain name: openai.com
Getting DNS information for: openai.com
; <<>> DiG 9.16.1-Ubuntu <<>> openai.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 12345
```

```
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 3
;; QUESTION SECTION:
;openai.com.                    IN      A

;; ANSWER SECTION:
openai.com.             299     IN      A       104.22.59.161
...
```

✅ This is a great way to call UNIX utilities from within a C program.

b. Write a Shell script for calculating the area of a triangle.

Here's a **simple shell script** to calculate the **area of a triangle** using the formula:

$$Area = \frac{1}{2} \times \text{base} \times \text{height}$$

✅ **Shell Script: Area of a Triangle**

```
#!/bin/bash

echo "Enter base of the triangle:"
read base

echo "Enter height of the triangle:"
read height

area=$(echo "0.5 * $base * $height" | bc)

echo "Area of the triangle is: $area"
```

▶️ **Sample Run:**

```
Enter base of the triangle:
10
Enter height of the triangle:
5
Area of the triangle is: 25.0
```

🛠 **Notes:**

- This script uses bc (basic calculator) for floating-point arithmetic.
- Make sure bc is installed (most Linux systems have it by default).

Let me know if you want a version that takes all input in one line!

c. Write a C program for implementing Best Fit allocation strategy.

Here's a **very simple and minimal C program** for implementing the **Best Fit memory allocation** strategy — easy to read and suitable for beginners.

✅ **Simplified C Program: Best Fit Allocation**

```
#include <stdio.h>

int main() {
    int blocks[5] = {100, 500, 200, 300, 600};
    int processes[4] = {212, 417, 112, 426};
    int allocation[4];
    int i, j;

    for (i = 0; i < 4; i++) allocation[i] = -1; // initialize allocation

    for (i = 0; i < 4; i++) {
        int best = -1;
        for (j = 0; j < 5; j++) {
            if (blocks[j] >= processes[i]) {
                if (best == -1 || blocks[j] < blocks[best])
                    best = j;
            }
        }

        if (best != -1) {
            allocation[i] = best;
            blocks[best] -= processes[i];
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (i = 0; i < 4; i++) {
        printf("%d\t\t%d\t\t", i + 1, processes[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
```

```
            printf("Not Allocated\n");
        }

        return 0;
}
```

Output:

Process No.	Process Size	Block No.
1	212	4
2	417	5
3	112	1
4	426	Not Allocated

Key Notes:

- Hardcoded block and process sizes to simplify input handling.
- Allocates the best-fitting block for each process.
- Easy to modify if you want user input later.

13. Write a Shell program to find the smallest digit from a number.

Here's the **simplest logic** possible for finding the **smallest digit in a number** using pure shell and minimal lines — **no loops if avoidable**, and **no external tools** like fold.

✅ **Simplest Logic Shell Script**

```
#!/bin/bash

read -p "Enter a number: " num
min=9

while [ $num -gt 0 ]; do
  digit=$(( num % 10 ))
  if [ $digit -lt $min ]; then min=$digit; fi
  num=$(( num / 10 ))
done

echo "Smallest digit is: $min"
```

💡 **Why it's simple:**

- Uses basic math only: % and / to extract digits.
- Only uses if and a while loop.
- No strings, arrays, or extra commands.

▶️ **Sample Output:**

```
Enter a number: 53897
Smallest digit is: 3
```

14. Write a C program for implementing Worst Fit allocation strategy.

Here's an even **simpler and shortest version** of the **Worst Fit memory allocation** in C, using **hardcoded inputs** and **minimal logic** — perfect for quick understanding and learning:

✅ **Simplest C Program: Worst Fit Allocation**

```
#include <stdio.h>

int main() {
    int blocks[3] = {300, 200, 100};
    int processes[3] = {120, 210, 50};
    int i, j, worst, allocation[3] = {-1, -1, -1};

    for (i = 0; i < 3; i++) {
        worst = -1;
        for (j = 0; j < 3; j++) {
            if (blocks[j] >= processes[i]) {
                if (worst == -1 || blocks[j] > blocks[worst])
                    worst = j;
            }
        }
        if (worst != -1) {
            allocation[i] = worst;
            blocks[worst] -= processes[i];
        }
    }

    for (i = 0; i < 3; i++) {
        printf("Process %d (%d KB) -> ", i+1, processes[i]);
        if (allocation[i] != -1)
            printf("Block %d\n", allocation[i]+1);
        else
            printf("Not Allocated\n");
    }

    return 0;
}
```

▶️ Sample Output:

```
Process 1 (120 KB) -> Block 1
Process 2 (210 KB) -> Block 2
Process 3 (50 KB) -> Block 1
```

🧠 Key Logic (in 1 line inside loop):

`if (worst == -1 || blocks[j] > blocks[worst])`

This selects the **largest fitting block**, which is the essence of **Worst Fit**.

15. Write a C program to implement C-LOOK disk scheduling strategy.

Here's a **shorter and simpler C program** implementing the **C-LOOK disk scheduling** algorithm with minimal complexity:

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

int main() {
    int req[] = {95, 180, 34, 119, 11, 123};
    int n = 6, head = 50, i, total = 0;

    // Sort requests
    for(i = 0; i < n-1; i++)
        for(int j = i+1; j < n; j++)
            if(req[i] > req[j]) {
                int t = req[i]; req[i] = req[j]; req[j] = t;
            }

    // Find start index
    int start = 0;
    while(start < n && req[start] < head) start++;

    // Service requests upwards
    int cur = head;
    for(i = start; i < n; i++) {
        total += abs(req[i] - cur);
        cur = req[i];
        printf("%d ", req[i]);
    }

    // Wrap to smallest requests below head
    if(start > 0) {
        total += abs(cur - req[0]);
        cur = req[0];
        for(i = 0; i < start; i++) {
            total += abs(req[i] - cur);
            cur = req[i];
```

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

    printf("\nTotal seek time = %d\n", total);
    return 0;
}
```

Output:

95 119 123 11 34

Total seek time = 209

Explanation:

- Hardcoded requests and head to keep code short.
- Simple bubble sort.
- Process requests upward from head.
- Jump to smallest and process remaining requests.

16. Write a C program for implementing LOOK disk scheduling strategy.

Here's a **shorter and simpler** version of the LOOK disk scheduling program, with **hardcoded inputs** for quick testing:

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

int main() {
    int req[] = {95, 180, 34, 119, 11, 123};
    int n = 6, head = 50, i, total = 0;

    // Sort requests
    for(i = 0; i < n-1; i++)
        for(int j = i+1; j < n; j++)
            if(req[i] > req[j]) {
                int t = req[i]; req[i] = req[j]; req[j] = t;
            }

    // Find start index
    int start = 0;
    while(start < n && req[start] < head) start++;

    int cur = head;

    // Move up servicing requests
    for(i = start; i < n; i++) {
        total += abs(req[i] - cur);
        cur = req[i];
        printf("%d ", req[i]);
    }
    // Move down servicing requests
    for(i = start - 1; i >= 0; i--) {
        total += abs(req[i] - cur);
        cur = req[i];
        printf("%d ", req[i]);
    }
```

```
    printf("\nTotal seek time = %d\n", total);
    return 0;
}
```

Output Example:

```
95 119 123 180 34 11
Total seek time = 208
```

17. Write a shell program to find the smallest number between two numbers using function.

Here's a simple shell script that uses a function to find the smallest of two numbers:

```
#!/bin/bash

# Function to find smallest number
smallest() {
  if [ $1 -lt $2 ]; then
    echo "$1"
  else
    echo "$2"
  fi
}

# Read two numbers from user
read -p "Enter first number: " num1
read -p "Enter second number: " num2

# Call function and display result
result=$(smallest $num1 $num2)
echo "The smallest number is: $result"
```

Sample Run:

```
Enter first number: 45
Enter second number: 32
The smallest number is: 32
```

18. Write a C program for implementing contiguous file allocation strategy.

Here's a **shorter and simpler** C program for **Contiguous File Allocation** with minimal checks and logic:

```
#include <stdio.h>

int main() {
    int disk[20] = {0}; // disk blocks
    int start, length, i, n;

    printf("Number of files: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("File %d start block and length: ", i+1);
        scanf("%d%d", &start, &length);

        // Check if space free and allocate
        int free = 1;
        for (int j = start; j < start + length; j++)
            if (j >= 20 || disk[j]) free = 0;

        if (free)
            for (int j = start; j < start + length; j++)
                disk[j] = 1;
        else
            printf("Cannot allocate file %d\n", i+1);
    }

    printf("Disk allocation:\n");
    for (i = 0; i < 20; i++) printf("%d ", disk[i]);
    printf("\n");

    return 0;
}
```

Sample Run:

```
Number of files: 2
File 1 start block and length: 2 5
File 2 start block and length: 6 4
Disk allocation:
0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
```

This code:

- Uses a smaller disk array (20 blocks)
- Reads start block & length for each file
- Checks if blocks are free, then allocates
- Prints allocation map

19. Write a C program for implementing SCAN disk scheduling strategy.

Here's a **shorter and simpler** version of the SCAN disk scheduling program with hardcoded requests and head position:

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

int main() {
   int req[] = {95,180,34,119,11,123};
   int n = 6, head = 50, disk_size = 200;
   int i, total = 0;

   // Sort requests
   for(i=0; i<n-1; i++)
      for(int j=i+1; j<n; j++)
         if(req[i]>req[j]){
            int t=req[i]; req[i]=req[j]; req[j]=t;
         }

   int start=0;
   while(start<n && req[start]<head) start++;

   int cur = head;

   // Move up servicing
   for(i=start; i<n; i++){
      total += abs(req[i]-cur);
      cur = req[i];
      printf("%d ", req[i]);
   }

   // Move to disk end if not there
   if(cur != disk_size-1){
      total += abs((disk_size-1)-cur);
      cur = disk_size-1;
   }
```

```
    // Move down servicing remaining
    for(i=start-1; i>=0; i--){
        total += abs(req[i]-cur);
        cur = req[i];
        printf("%d ", req[i]);
    }

    printf("\nTotal seek time = %d\n", total);
    return 0;
}
```

Output:

```
95 119 123 180 34 11
Total seek time = 389
```

20. Write a C program for implementing C-SCAN disk scheduling strategy.

Here's a **shorter, simpler** C program for C-SCAN disk scheduling with hardcoded values:

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

int main() {
    int req[] = {95,180,34,119,11,123};
    int n = 6, head = 50, disk_size = 200;
    int i, total = 0;

    // Sort requests
    for(i=0; i<n-1; i++)
        for(int j=i+1; j<n; j++)
            if(req[i]>req[j]){
                int t=req[i]; req[i]=req[j]; req[j]=t;
            }

    int start=0;
    while(start<n && req[start]<head) start++;

    int cur = head;

    // Move up servicing requests
    for(i=start; i<n; i++){
        total += abs(req[i]-cur);
        cur = req[i];
        printf("%d ", req[i]);
    }

    // Move to end of disk if not there
    if(cur != disk_size-1){
        total += abs((disk_size-1)-cur);
        cur = disk_size-1;
    }

    // Jump to beginning
```

```
    total += abs(cur - 0);
    cur = 0;

    // Service requests from beginning to start-1
    for(i=0; i<start; i++){
        total += abs(req[i]-cur);
        cur = req[i];
        printf("%d ", req[i]);
    }

    printf("\nTotal seek time = %d\n", total);
    return 0;
}
```

Output:

```
95 119 123 180 11 34
Total seek time = 391
```