# 01: پروگرام نمبر

## Basics of UNIX commands

## Objective:

یونیکس کمانڈز وہ مختصر ہدایات ہیں جو صارف یونیکس شیل میں داخل کرتا ہے تاکہ آپریٹنگ سسٹم سے مخصوص کام انجام دیے جائیں، جیسے فائلیں دیکھنا، ڈائریکٹریز بنانا، یا فائلز کو منتقل کرنا۔ یہ کمانڈز سسٹم کے ساتھ براہ راست رابطے کا ذریعہ ہیں اور اس کی فعالیت کو کنٹرول کرتی ہیں۔

i. **mkdir :** used to create a directory

@anees10z ➜ /workspaces/Operating-System> mkdir demo


ii. **cd :** used to change directory

/workspaces/Operating-System> cd Shell-Programming

@anees10z ➜ /workspaces/Operating-System/Shell-Programming

iii. **cd .. :** used to go previous directory
@anees10z ➜ /workspaces/Operating-System (main) $

iv. **ls :** used to listing directories

@anees10z ➜ /workspaces/Operating-System> ls
CPU-Scheduling-Algorithm  Shell-Programming

v. **ls –a :** list all file including hidden

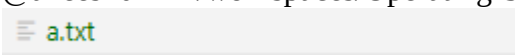
@anees10z ➜ /workspaces/Operating-System (main) $ ls -a
. .. .git CPU-Scheduling-Algorithm README.md Shell-Programming

vi. **rmdir :** used to remove an empty directory

@anees10z ➜ /workspaces/Operating-System> rmdir demo

vii. **touch :** used to create files

@anees10z ➜ /workspaces/Operating-System (main) $ touch a.txt




viii. **mv :** used for move files

@anees10z ➜ /workspaces/Operating-System (main) $ mv a.txt c.txt

c.txt
```
1  anees khan
2  OS LAB
```

ix. **cp :** used to copy file contents

@anees10z ➜ /workspaces/Operating-System (main) $ cp c.txt d.txt

d.txt

```
1    anees khan
2    OS LAB
```

x. **rm :** used to remove files

@anees10z ➜ /workspaces/Operating-System (main) $ rm b.txt

xi. **pwd :** print working directory

@anees10z ➜ /workspaces/Operating-System (main) $ pwd
/workspaces/Operating-System

xii. **cat :** used to display file contents

@anees10z ➜ /workspaces/Operating-System (main) $ cat c.txt
this is
demo for
display
file
contents

xiii. **head :** view specific line (first 10 lines)

@anees10z ➜ /workspaces/Operating-System (main) $ head c.txt
this is
demo for
display
file
contents

xiv. **tail :** view specific line (last 10 lines)

@anees10z ➜ /workspaces/Operating-System (main) $ tail c.txt
43
ed
ed
s
a
this is
demo for
display
file
contents

xv. **wc :** counting word, lines, and chars in a file

@anees10z ➜ /workspaces/Operating-System (main) $ wc d.txt
1 4 17 d.txt

xvi. **more :** Allows viewing of large files page by page

```
@anees10z ➜ /workspaces/Operating-System (main) $ more d.txt
anees khan
OS LAB
```

xvii. **uname –a :** Displays detailed information about the kernel

```
@anees10z ➜ /workspaces/Operating-System (main) $ uname –a
Linux codespaces-1fa8f6 6.8.0-1021-azure #25~22.04.1-Ubuntu SMP Thu Jan 16
21:37:09 UTC 2025 x86_64 x86_64 x86_64 GNU/Linux
```

xviii. **free –m :** Displays memory usage in megabytes

```
@anees10z ➜ /workspaces/Operating-System (main) $ free -m
               total        used        free      shared  buff/cache   available
Mem:            7938        1329         261          60        6347        6232
Swap:              0           0           0
```

xix. **uptime :** Displays how long the system has been running along with the average load

```
@anees10z ➜ /workspaces/Operating-System (main) $ uptime
 07:33:30 up  1:36,  0 users,  load average: 0.13, 0.19, 0.14
```

xx. **date :** display current date and time

```
@anees10z ➜ /workspaces/Operating-System (main) $ date
Sun Mar 16 07:35:00 UTC 2025
```

xxi. **whoami :** Display the current user name

```
@anees10z ➜ /workspaces/Operating-System (main) $ whoami
codespace
```

xxii. **ps :** Display the currently running processes

```
@anees10z ➜ /workspaces/Operating-System (main) $ ps
    PID TTY          TIME CMD
   3087 pts/1    00:00:00 bash
  43685 pts/1    00:00:00 ps
```

# 02: پروگرام نمبر

## SHELL PROGRAMMING

**Objective:**

شیل پروگرامنگ یونیکس شیل میں کمانڈز اور اسکرپٹس لکھنے کا عمل ہے تاکہ خودکار طریقے سے کام انجام دیے جائیں۔ یہ صارف کو کمانڈز کو جوڑنے، لوپس، کنڈیشنز اور متغیرات استعمال کرنے کی سہولت دیتی ہے تاکہ پیچیدہ ٹاسکس آسان ہوں۔

1. Hello World

```
#!/bin/bash
echo "Hello, World!"
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-
Programming (main) $ bash hello.sh
Hello, World!
```

2. Add Two Numbers

```
#!/bin/bash
n1=20
n2=30
sum=$((n1+n2))
echo "sum = $sum"
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-
Programming (main) $ bash AddTwoNumbers.sh
sum = 50
```

3. Subtract Two Numbers

```
#!/bin/bash
n1=20
n2=30
diff=$((n2-n1))
echo "Difference = $diff"
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-
Programming (main) $ bash AddTwoNumbers.sh
Difference = 10
```

4. Multiply Two Numbers

```
#!/bin/bash
n1=2
n2=3
prd=$((n1*n2))
echo "Product = $prd"
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-
Programming (main) $ bash MultiplyTwoNumbers.sh
Product = 6
```

5. Divide Two Numbers

```
#!/bin/bash
n1=22
n2=5
quo=$((n1/n2))
rem=$((n1%n2))
echo "Qoutient = $quo"
echo "Remainder = $rem"
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-Programming (main) $
bash DivideTwoNumbers.sh
Qoutient = 4
Remainder = 2
```

6. Check even or Odd

```
#!/bin/bash
read -p "Enter a number: " num
if ((num %2 ==0)); then
echo "$num is even."
else
echo "$num is odd."
Fi
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-
Programming (main) $ bash evenOdd.sh
Enter a number: 6
6  is even.
```

7. Get user input

```
#!/bin/bash
read -p "What is your name? " name
echo "Hello, $name!"
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-
Programming (main) $ bash getUserInput.sh
What is your name? anees
Hello, anees!
```

8. For loop example

```
#!/bin/bash
for i in {1..3}; do echo "Number $i"; done
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-Programming
(main) $ bash forLoop.sh
Number 1
Number 2
Number 3
```

9. While loop example

```
#!/bin/bash
n=3
```

```
while [ $n -gt 0 ]; do echo "$n"; n=$((n - 1)); done
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-Programming (main) $
bash whileLoop.sh
3
2
1
```

10. Factorial Calculation

```
#!/bin/bash
read -p "Enter a number: " num
factorial=1
for ((i=1; i<=num; i++)); do
 factorial=$((factorial * i))
done
echo "Factorial of $num is $factorial"
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-Programming
(main) $ bash factorial.sh
Enter a number: 5
Factorial of 5 is 120
```

11. Reverse String

```
#!/bin/bash
read -p "Enter a string: " str
echo "$str" | rev
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-Programming (main) $
bash reverseStr.sh
Enter a string: anees
seena
```

12. Simple Menu Selection

```
#!/bin/bash
PS3="Select an option (1-3): "
options=("Option 1" "Option 2" "Quit")
select opt in "${options[@]}"; do
case $opt in
"Option 1") echo "You chose Option 1";;
"Option 2") echo "You chose Option 2";;
"Quit") break;;
*) echo "Invalid option";;
esac
done
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-Programming
(main) $ bash menu.sh
1) Option 1
2) Option 2
3) Quit
Select an option (1-3): 3
```

13. Generate Fibonacci Sequence

```
#!/bin/bash
read -p "Enter number of terms: " n
a=0
b=1
for ((i=0; i<n; i++)); do
   echo "$a"
   fn=$((a + b))
   a=$b
   b=$fn
done
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-Programming (main) $
bash fibonacci.sh
Enter number of terms: 5
0
1
1
2
3
```

14. Display current date and time

```
#!/bin/bash
date +"%Y-%m-%d %H:%M:%S"
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-Programming (main) $
bash dateTime.sh
2025-03-08 07:16:48
```

15. Search for a specific word in a file

```
#!/bin/bash
read -p "Enter word to search for: " word
grep -o "$word" demo.txt | wc -l
```

```
@anees10z ➜ /workspaces/Operating-System/Shell-Programming (main) $
bash searchWordInFile.sh
Enter word to search for: hello
2
```

# 03: پروگرام نمبر

## Implementation of CPU scheduling Algorithms

### Objective: (a) Implement FCFS Algorithm

### Description:

First-Come, First-Served (FCFS) is a non-preemptive CPU scheduling algorithm where processes are executed in the order they arrive in the ready queue, using a simple FIFO approach.

### Algorithm:

1. Read the number of processes (`n`). Initialize arrays for burst time (`bt[]`), waiting time (`wt[]`), and turnaround time (`tat[]`).
2. Take input for burst times of `n` processes and store them in `bt[]`.
3. Set the waiting time of the first process to `0` as `wt[0] = 0`.
4. Calculate the waiting time for each process using `wt[i] = wt[i-1] + bt[i-1]`, ensuring that each process starts execution immediately after the previous one.
5. Compute the turnaround time for each process using `tat[i] = bt[i] + wt[i]`, which represents the total time a process takes from entering the queue to completion.
6. Display a formatted table containing process ID, burst time, waiting time, and turnaround time for all processes.
7. Calculate the total waiting time and total turnaround time by summing up all values in `wt[]` and `tat[]`.
8. Compute the average waiting time (`avg_wt`) as `total_wt / n` and the average turnaround time (`avg_tat`) as `total_tat / n`.
9. Print the calculated average waiting time and average turnaround time.

## C-Program:

```c
#include <stdio.h>
// FCFS algorithm without Arival Time
void main()
{
    int n;
    printf("Enter number of Process: ");
    scanf("%d", &n);

    int bt[n];
    int wt[n];
    int tat[n];

    // Burst time of processes
    printf("\nEnter burst time for Processes :\n");
    for (int i = 0; i < n; ++i)
    {
        printf(" Burst time of Process: P%d = ", i);
        scanf("%d", &bt[i]);
    }

    // calculating waiting time
    wt[0] = 0;
    for (int i = 1; i < n; i++)
```

```
    {
        wt[i] = wt[i - 1] + bt[i - 1];
    }

    // calculating Turn Around time
    for (int i = 0; i < n; i++)
    {
        tat[i] = bt[i] + wt[i];
    }
    // Display results in well-formatted table
    printf("\n-------------------------------------------\n");
    printf("| %-5s | %-10s | %-12s | %-15s |\n", "P-ID",
          "Burst Time", "Waiting Time", "Turnaround Time");
    printf("-------------------------------------------\n");

    for (int i = 0; i < n; ++i)
    {
        printf("| %-5d | %-10d | %-12d | %-15d |\n", i,
               bt[i], wt[i], tat[i]);
    }
    printf("-------------------------------------------\n");
    // Calculate average waiting time and turnaround time
    float total_wt = 0, total_tat = 0;
    for (int i = 0; i < n; i++)
    {
        total_wt += wt[i];
        total_tat += tat[i];
    }
    printf("\nAverage Waiting Time: %.2f\n", total_wt / n);
    printf("Average Turnaround Time: %.2f\n", total_tat / n);
}
```

## Output

```
Enter number of Process: 3

Enter burst time for Processes :
 Burst time of Process: P0 = 1
 Burst time of Process: P1 = 3
 Burst time of Process: P2 = 4


-------------------------------------------
| P-ID  | Burst Time | Waiting Time | Turnaround Time
-------------------------------------------
| 0     | 1          | 0            | 1
| 1     | 3          | 1            | 4
| 2     | 4          | 4            | 8
-------------------------------------------

Average Waiting Time: 1.67
Average Turnaround Time: 4.33
```

## Objective: (b) Implement SRTF Algorithm

## Description:

Shortest Remaining Time First (SRTF) is a preemptive scheduling algorithm where the process with the shortest remaining burst time gets CPU priority. If a new process arrives with a smaller burst time, the CPU switches to it.

## Algorithm:

1. Read the number of processes (`n`). Initialize arrays for arrival time (AT) burst time (BT), remaining time (RT), completion time (CT), turnaround time (TAT), and waiting time (WT). Also, initialize `total_tat` and `total_wt` to 0.
2. Take input for arrival time (AT) and burst time (BT) for each process. Set remaining time (RT) equal to burst time (BT) initially.
3. Start execution from time = 0.
4. At each time unit, find the process with the shortest remaining time among those that have already arrived (AT ≤ time) and have remaining execution time (RT > 0).
5. If no process is available at the current time, increment time and check again.
6. If a process is found, execute it for 1 time unit by decrementing its remaining time (RT) and incrementing the current time (time).
7. If a process completes execution (RT == 0), update its completion time (CT = time) and increment the finished process count.
8. Repeat steps 4–7 until all processes are completed (`finished == n`).
9. Compute turnaround time (TAT) = CT - AT and waiting time (WT) = TAT - BT for each process.
10. Display a table with Process ID, AT, BT, CT, TAT, and WT.
11. Calculate average turnaround time (`avg_tat = total_tat / n`) and average waiting time (`avg_wt = total_wt / n`).
12. Print `avg_tat` and `avg_wt`.

# C-Program:

```c
#include <stdio.h>

int main() {
    int n, i, time = 0, smallest, finished = 0;
    printf("Enter number of processes: ");
    scanf("%d", &n);

    int at[n], bt[n], rt[n], ct[n], tat[n], wt[n];
    float total_tat = 0, total_wt = 0;

    for (i = 0; i < n; i++) {
        printf("Enter AT and BT for P%d: ", i + 1);
        scanf("%d %d", &at[i], &bt[i]);
        rt[i] = bt[i]; // Remaining time = Burst time initially
    }

    while (finished < n) {
        smallest = -1;
        for (i = 0; i < n; i++) {
            if (at[i] <= time && rt[i] > 0 && (smallest == -1 || rt[i]
               < rt[smallest]))
                smallest = i;
```

```
        }

        if (smallest == -1) {
            time++; // If no process is available, move time forward
            continue;
        }

        rt[smallest]--; // Execute process for 1 unit
        time++;

        if (rt[smallest] == 0) { // If process completes
            finished++;
            ct[smallest] = time;
        }
    }

    printf("\nP\tAT\tBT\tCT\tTAT\tWT\n");
    for (i = 0; i < n; i++) {
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt[i];
        total_tat += tat[i];
        total_wt += wt[i];
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], ct[i],
               tat[i], wt[i]);
    }

    printf("\nAvg TAT: %.2f", total_tat / n);
    printf("\nAvg WT: %.2f\n", total_wt / n);

    return 0;
}
```

## Output

```
Enter number of processes: 3
Enter AT and BT for P1: 0 4
Enter AT and BT for P2: 2 2
Enter AT and BT for P3: 1 1

P   AT  BT  CT  TAT WT
1   0   4   7   7   3
2   2   2   4   2   0
3   1   1   2   1   0

Avg TAT: 3.33
Avg WT: 1.00
```

## Objective: (c) Implement Round Robin Algorithm

## Description:

Round Robin (RR) Scheduling is a preemptive CPU scheduling algorithm where each process gets a fixed time slice (quantum) for execution in a cyclic order. If a process doesn't finish within its time quantum, it is moved to the end of the queue. This ensures fair CPU allocation and reduces starvation.

## Algorithm:

1. Read the number of processes (`n`) and the time quantum (`quantum`). Initialize arrays for arrival time (AT), burst time (BT), remaining time (RT), completion time (CT), turnaround time (TAT), and waiting time (WT). Also, set `total_tat = 0` and `total_wt = 0`.
2. Take input for arrival time (AT) and burst time (BT) of each process, and initialize remaining time (RT) as burst time (BT).
3. Set `time = 0` and `completed = 0` to track the execution time and completed processes.
4. Repeat until all processes are completed:
   Check all processes one by one.
   - If a process has arrived (`AT[i] ≤ time`) and has remaining execution time (`RT[i] > 0`):
   - If `RT[i] > quantum`, execute it for `quantum` time, then reduce `RT[i]`.
   - Else, execute it for its remaining time (`RT[i]`), set `RT[i] = 0`, update completion time (CT[i] = time), and increase `completed` count.
   - If no process is available at the current time, increment `time`.
5. Calculate turnaround time (TAT) using `TAT[i] = CT[i] - AT[i]`.
6. Calculate waiting time (WT) using `WT[i] = TAT[i] - BT[i]`.
7. Print a table displaying Process ID, AT, BT, CT, TAT, and WT.
8. Compute average turnaround time (`avg_tat = total_tat / n`) and average waiting time (`avg_wt = total_wt / n`).
9. Print the values of `avg_tat` and `avg_wt`.

# C-Program:

```c
#include <stdio.h>

int main() {
    int n, quantum, i, time = 0, completed = 0;

    printf("Enter number of processes and time quantum: ");
    scanf("%d %d", &n, &quantum);

    int at[n], bt[n], rt[n], ct[n], tat[n], wt[n];
    float total_tat = 0, total_wt = 0;

    for (i = 0; i < n; i++) {
        printf("Enter AT and BT for P%d: ", i + 1);
        scanf("%d %d", &at[i], &bt[i]);
        rt[i] = bt[i];
    }
```

```
    while (completed < n) {
        int done = 1;
        for (i = 0; i < n; i++) {
            if (rt[i] > 0 && at[i] <= time) {
                done = 0;
                if (rt[i] > quantum) time += quantum, rt[i] -= quantum;
                else time += rt[i], rt[i] = 0, ct[i] = time,
                completed++;
            }
        }
        if (done) time++;
    }

    printf("\nP\tAT\tBT\tCT\tTAT\tWT\n");
    for (i = 0; i < n; i++) {
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt[i];
        total_tat += tat[i], total_wt += wt[i];
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], ct[i],
               tat[i], wt[i]);
    }

    printf("\nAvg TAT: %.2f", total_tat / n);
    printf("\nAvg WT: %.2f\n", total_wt / n);

    return 0;
}
```

## Output

```
Enter number of processes and time quantum: 3 4
Enter AT and BT for P1: 2 4
Enter AT and BT for P2: 1 7
Enter AT and BT for P3: 0 3

P   AT  BT  CT  TAT WT
1   2   4   7   5   1
2   1   7   14  13  6
3   0   3   3   3   0

Avg TAT: 7.00
Avg WT: 2.33
```

## Objective: (d) Implement Priority Scheduling Algorithm

## Description:

Priority Scheduling executes processes based on their priority value. The highest priority process runs first, and it can be preemptive or non-preemptive.

## Algorithm:

1. Read input: Take the number of processes (`n`) and store arrival time (AT), burst time (BT), and priority (PR) for each process. Also, store process IDs (`P`).
2. Sort processes by priority: Lower priority values indicate higher priority. If two processes have the same priority, their order remains unchanged.
3. Initialize time tracking: Start execution from `time = 0`.
4. Process execution:
    - If `time < AT[i]`, set `time = AT[i]` to handle CPU idle time.
    - Calculate completion time (CT) using `CT[i] = time + BT[i]`.
    - Update `time = CT[i]`.
5. Compute turnaround time (TAT) and waiting time (WT):
    - `TAT[i] = CT[i] - AT[i]`
    - ‘WT[i] = TAT[i] - BT[i]`
6. Print process details: Display Process ID, AT, BT, PR, CT, TAT, WT in tabular form.
7. Calculate averages: Compute average turnaround time (avg_TAT) and average waiting time (avg_WT).
8. Print average times: Display `avg_TAT` and `avg_WT`.

# C-Program:

```c
#include <stdio.h>

int main() {
    int n, i, j;

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

    int at[n], bt[n], pr[n], ct[n], tat[n], wt[n], p[n];
    float total_tat = 0, total_wt = 0;

    for (i = 0; i < n; i++) {
        printf("Enter Arrival Time, Burst Time, and Priority for
               Process %d: ", i + 1);
        scanf("%d %d %d", &at[i], &bt[i], &pr[i]);
        p[i] = i + 1; // Store process number
    }

    // Sorting by priority (lower number = higher priority)
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (pr[i] > pr[j]) {
```

```
                int temp;

                temp = pr[i]; pr[i] = pr[j]; pr[j] = temp;
                temp = at[i]; at[i] = at[j]; at[j] = temp;
                temp = bt[i]; bt[i] = bt[j]; bt[j] = temp;
                temp = p[i]; p[i] = p[j]; p[j] = temp;
            }
        }
    }

    int time = 0;

    for (i = 0; i < n; i++) {
        if (time < at[i])
            time = at[i]; // Handle CPU idle time

        ct[i] = time + bt[i]; // Completion Time
        time = ct[i]; // Update current time
    }

    printf("\nP\tAT\tBT\tPR\tCT\tTAT\tWT\n");

    for (i = 0; i < n; i++) {
        tat[i] = ct[i] - at[i]; // Turnaround Time
        wt[i] = tat[i] - bt[i]; // Waiting Time

        total_tat += tat[i];
        total_wt += wt[i];

        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i], at[i], bt[i],
                pr[i], ct[i], tat[i], wt[i]);
    }

    printf("\nAverage TAT: %.2f", total_tat / n);
    printf("\nAverage WT: %.2f\n", total_wt / n);

    return 0;
}
```

## Output

```
Enter number of processes: 3
Enter Arrival Time, Burst Time, and Priority for Process 1: 1 3 1
Enter Arrival Time, Burst Time, and Priority for Process 2: 3 5 4
Enter Arrival Time, Burst Time, and Priority for Process 3: 5 4 2

P   AT  BT  PR  CT  TAT WT
1   1   3   1   4   3   0
3   5   4   2   9   4   0
2   3   5   4   14  11  6

Average TAT: 6.00
Average WT: 2.00
```

# 04: پروگرام نمبر

## Implementation of all file allocation strategies

### Objective: (a) Implement Contiguous File Allocation

### Description:

Contiguous File Allocation is a disk allocation method where a file is stored in continuous memory blocks on the disk. It allows fast access but may cause fragmentation and difficulty in resizing files.

### Algorithm:

1. Initialize disk:
   - Take input for `total_blocks` (total disk space).
   - Create an array `disk[]` and initialize all blocks as free (`0`).
2. Take input:
   - Ask the user for the `starting block` and `number of blocks required`.
3. Check space availability:
   - If `start + length > total_blocks`, print an error and exit.
4. Check for already allocated blocks:
   - If any block in the required range is already allocated (`1`), print an error and exit.
5. Allocate blocks:
   - Mark the required blocks as allocated (`1`).
   - Print the allocated block numbers and confirm successful allocation.

## C-Program:

```c
#include <stdio.h>

int main() {
    int total_blocks, start, length, i, flag = 0;

    printf("Enter total number of disk blocks: ");
    scanf("%d", &total_blocks);

    int disk[total_blocks]; // Array to track allocated blocks

    // Initialize disk (0 = free, 1 = allocated)
    for (i = 0; i < total_blocks; i++)
        disk[i] = 0;

    printf("Enter starting block: ");
    scanf("%d", &start);

    printf("Enter number of blocks required: ");
    scanf("%d", &length);
```

```
    // Check if allocation is possible
    if (start + length > total_blocks) {
        printf("Error: Not enough space available!\n");
        return 0;
    }

    for (i = start; i < start + length; i++) {
        if (disk[i] == 1) { // Block already allocated
            flag = 1;
            break;
        }
    }

    if (flag == 1) {
        printf("Error: Some blocks are already allocated!\n");
    } else {
        for (i = start; i < start + length; i++)
            disk[i] = 1; // Mark blocks as allocated

        printf("Allocated Blocks: ");
        for (i = start; i < start + length; i++)
            printf("%d ", i);

        printf("\nFile allocated successfully!\n");
    }

    return 0;
}
```

## Output

```
Enter total number of disk blocks: 10
Enter starting block: 3
Enter number of blocks required: 5
Allocated Blocks: 3 4 5 6 7
File allocated successfully!
```

## Objective: (b) Implement Linked File Allocation

## Description:

Linked File Allocation stores file data in scattered disk blocks, with each block pointing to the next. It avoids fragmentation but has slower access due to sequential traversal.

## Algorithm:

1. Initialize disk storage:

   - Read the total number of blocks (`n`).
   - Create an array `disk[n]` to track allocation, initializing all blocks as `-1` (free).

2. Read starting block:

   - Take input for the starting block.
   - If the starting block is invalid (greater than `n`), display an error and exit.

3. Allocate blocks using linked structure:

   - Read next block numbers in sequence until the user enters `-1` or the disk is full.
   - Validate each block to ensure it is within range and not already allocated.
   - Store the next block reference in the `disk` array to create a linked chain.

4. Display allocated blocks:

   - Start from the initial block and follow links until `-1` is reached.
   - Print the block sequence representing the file storage.

# C-Program:

```c
#include <stdio.h>

int main() {
    int n, start, next, count = 0;

    printf("Enter total number of blocks: ");
    scanf("%d", &n);

    int disk[n]; // Array to track allocation
    for (int i = 0; i < n; i++)
        disk[i] = -1; // Initialize as free

    printf("Enter starting block: ");
    scanf("%d", &start);

    if (start >= n) {
        printf("Error: Invalid starting block!\n");
        return 0;
    }

    int current = start;
    printf("Enter up to %d block links (-1 to stop):\n", n - 1);

    while (count < n - 1) { // Ensure it doesn't exceed disk size
        scanf("%d", &next);
        if (next == -1) break;
```

```
            if (next >= n || disk[next] != -1) { // Check validity
                printf("Error: Invalid or already allocated block!\n");
                continue;
            }
            disk[current] = next;
            current = next;
            count++;
        }

        printf("\nAllocated Blocks:\n");
        current = start;
        while (current != -1) {
            printf("%d -> ", current);
            current = disk[current];
        }
        printf("NULL\n");

        return 0;
}
```

## Output

```
Enter total number of blocks: 5
Enter starting block: 1
Enter up to 4 block links (-1 to stop):
3 2 4 -1

Allocated Blocks:
1 -> 3 -> 2 -> 4 -> NULL
```

## Objective: (c) Implement Indexed File Allocation

## Description:

Indexed File Allocation stores all disk block addresses of a file in a single index block. This allows direct access to any block, avoiding fragmentation, but requires extra space for the index table.

## Algorithm:

1. Initialize disk storage:
- Define an array `f[MAX]` to track allocated blocks (set all to free).
- Define `index[MAX]` to store indexed block references.
2. Accept file allocation requests:
- Input an index block(`ind`). If already allocated, ask for another.
- Input the number of blocks (`n`) required and their block numbers.
3. Validate and allocate blocks:
- Check if all requested blocks are free.
- If valid, allocate blocks by updating `f[]` and store references in `index[]`.
- If any block is already allocated, display an error and retry.
4. Display allocation:
   - Print the index block and its allocated blocks.
5. Repeat or exit:
   - Ask the user if they want to allocate more files (`1` for Yes, `0` for No).

# C-Program:

```c
#include <stdio.h>
#define MAX 50 // Maximum number of blocks

int f[MAX], index[MAX]; // Disk block status and index table

int main() {
    int n, ind, i, count, c;

    // Initialize all blocks as free
    for (i = 0; i < MAX; i++)
        f[i] = 0;

    while (1) {
        printf("Enter index block: ");
        scanf("%d", &ind);

        if (ind < 0 || ind >= MAX) {
            printf("Error: Invalid index block! Try again.\n");
            continue;
        }
        if (f[ind]) {
            printf("Index block already allocated! Try another.\n");
            continue;
        }
        printf("Enter number of blocks needed: ");
        scanf("%d", &n);
```

```
        if (n <= 0 || n > MAX) {
            printf("Error: Invalid number of blocks! Try again.\n");
            continue;
        }
        printf("Enter block numbers: ");
        count = 0;

        for (i = 0; i < n; i++) {
            scanf("%d", &index[i]);

            // Check if block is within range and not allocated
            if (index[i] >= 0 && index[i] < MAX && !f[index[i]])
                count++;
        }
        if (count == n) { // All blocks are free
            f[ind] = 1;
            for (i = 0; i < n; i++)
                f[index[i]] = 1;

            printf("Allocated\nFile Indexed\n");
            for (i = 0; i < n; i++)
                printf("%d -> %d\n", ind, index[i]);
        } else {
            printf("Error: Some blocks are already allocated or
                    invalid! Try again.\n");
        }
        printf("Enter more files? (1-Yes / 0-No): ");
        scanf("%d", &c);

        if (c != 1) {
            printf("Exiting program...\n");
            break;
        }
    }
    return 0;
}
```

## Output

```
Enter index block: 1
Enter number of blocks needed: 3
Enter block numbers: 2 4 5
Allocated
File Indexed
1 -> 2
1 -> 4
1 -> 5
Enter more files? (1-Yes / 0-No): 0
Exiting program...
```

## Objective: (d) Implement File Allocation Table (FAT)

## Description:

File Allocation Table (FAT) is a simple and efficient file system used in storage devices. It maintains a table that maps file blocks, tracking their sequence and location. Each file's blocks are linked using entries in the FAT, enabling easy access and management. It is widely used in USB drives, memory cards, and older operating systems.

## Algorithm:

1. **Initialize Variables:** Define arrays to store filenames, sizes, start blocks, and a File Allocation Table (FAT) to track file storage.
2. **Create File Function**:
   - Check if the maximum file limit is reached.
   - Take user input for filename, size, and start block.
   - Validate the start block to ensure it is within limits.
   - Store file details and mark the start block in the FAT.
3. List Files Function: Display stored files along with their size and start block.
4. Main Execution:

- Take input for the number of files.
- Call the file creation function for each file.
- Display the stored files using the list function.
- Exit the program successfully.

# C-Program:

```c
#include <stdio.h>
#include <string.h>
#define MAX 100

char filenames[MAX][20];
int sizes[MAX], starts[MAX], fat[MAX], count = 0;

void createFile() {
    if (count >= MAX) {
        printf("File limit reached!\n");
        return;
    }
    char name[20];
    int size, start;
    printf("Enter filename, size (in bytes), and start block: ");
    scanf("%s %d %d", name, &size, &start);

    if (start >= MAX) {
        printf("Invalid start block!\n");
        return;
    }

    strcpy(filenames[count], name);
    sizes[count] = size;
    starts[count] = start;
    fat[start] = -1;
    count++;
}
```

```c
void listFiles() {
    printf("Files:\n");
    for (int i = 0; i < count; i++)
        printf("%s, %dB, Start: %d\n", filenames[i], sizes[i], starts[i]);
}

int main() {
    int n;
    printf("Enter number of files: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        createFile();
    }
    listFiles();
    return 0;
}
```

## Output

```
Enter number of files: 3
Enter filename, size (in bytes), and start block: a.txt 102 2
Enter filename, size (in bytes), and start block: b.txt 839 1
Enter filename, size (in bytes), and start block: c.txt 238 4
Files:
a.txt, 102B, Start: 2
b.txt, 839B, Start: 1
c.txt, 238B, Start: 4
```

## Objective: (e) Implement Inode Allocation

## Description:

Inode Allocation is a file system technique where each file is represented by an inode (index node). The inode stores metadata like file size, permissions, ownership, and pointers to data blocks, enabling efficient file management and access.

## Algorithm:

1. Initialize an array to store filenames and their corresponding inode numbers.
2. Define a function createFile() to add new files:
- If the maximum file limit is reached, display an error.
- Accept the filename as input and assign a unique inode number.
- Increment the file count.
3. Define a function listFiles() to display all stored files along with their inode numbers.
4. In the main() function:
- Accept the number of files to be created.
- Call createFile() for each file.
- Display the stored files using listFiles().

# C-Program:

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

#define MAX_FILES 10

char filenames[MAX_FILES][20];
int inodes[MAX_FILES], count = 0;

void createFile() {
    if (count >= MAX_FILES) {
        printf("File limit reached!\n");
        return;
    }
    printf("Enter filename: ");
    scanf("%s", filenames[count]);
    inodes[count] = count + 1; // Assign inode number
    count++;
}
void listFiles() {
    printf("Files in System:\n");
    for (int i = 0; i < count; i++)
        printf("Filename: %s, Inode: %d\n", filenames[i], inodes[i]);
}
int main() {
    int n;
    printf("Enter number of files: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        createFile();
    listFiles();
    return 0;
}
```

## Output

```
Enter number of files: 2
Enter filename: a.txt
Enter filename: b.txt
Files in System:
Filename: a.txt, Inode: 1
Filename: b.txt, Inode: 2
```

# 05: پروگرام نمبر

## Objective: Implementation of Semaphores

## Description:

Semaphores are synchronization tools used in operating systems to manage concurrent processes and prevent race conditions. They help in controlling access to shared resources using signaling mechanisms.

## Algorithm:

1. Initialize Semaphore
   - A semaphore (`semaphore`) is initialized with a value of 3, meaning a maximum of 3 threads can access the resource simultaneously.
2. Create Threads
   - An array of 5 threads (`Num_threads = 5`) is created.
   - Each thread is assigned a unique ID and starts executing `accessResource()`.
3. Thread Execution (`accessResource`)

- Each thread prints that it is waiting for the resource.
- It then calls `sem_wait(&semaphore)`, which decrements the semaphore count. If the count is 0, the thread waits.
- Once allowed, it prints that it is accessing the resource and simulates a 2-second usage with `sleep(2)`.
- After finishing, it prints a message and calls `sem_post(&semaphore)`, incrementing the semaphore value to allow other threads access.
- The allocated memory for thread ID is freed.

4. Thread Completion
   - The `pthread_join()` function ensures all threads complete execution before the program ends.
5. Destroy Semaphore
   - `sem_destroy(&semaphore)` is called to clean up the semaphore before exiting.

# C-Program:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>
// to run this program in terminal add -lpthread like shown below
// gcc semaphore.c -lpthread
// ./a.out
#define Num_threads 5

sem_t semaphore;

void *accessResource(void *arg)
{
    printf("Thread %d is waiting to access the resource \n", *(int *)arg);
    sem_wait(&semaphore);
    printf("Thread %d is accesing resource \n", *(int *)arg);
    sleep(2);
```

```
    printf("Thread %d has finished accessing the resource\n", *(int *)arg);
    sem_post(&semaphore);
    free(arg);
    return NULL;
}

int main()
{
    sem_init(&semaphore, 0, 3);
    pthread_t threads[Num_threads];

    for (int i = 0; i < Num_threads; i++)
    {
        int *threadId = malloc(sizeof(int));
        *threadId = i;
        pthread_create(&threads[i], NULL, accessResource, threadId);
    }
    for (int i = 0; i < Num_threads; i++)
    {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&semaphore);

    return 0;
}
```

## Output

```
Thread 0 is waiting to access the resource
Thread 0 is accesing resource
Thread 1 is waiting to access the resource
Thread 1 is accesing resource
Thread 2 is waiting to access the resource
Thread 2 is accesing resource
Thread 4 is waiting to access the resource
Thread 3 is waiting to access the resource
Thread 0 has finished accessing the resource
Thread 2 has finished accessing the resource
Thread 4 is accesing resource
Thread 3 is accesing resource
Thread 1 has finished accessing the resource
Thread 4 has finished accessing the resource
Thread 3 has finished accessing the resource
```

# 06: پروگرام نمبر

## Implementation of all File Organization Techniques

### Objective: (a) Implement Sequential File Organization

### Description:

Sequential File Organization is a method of storing and accessing records in a file sequentially, one after another. Data is stored in a fixed order, making it efficient for reading large amounts of data but slower for random access. It is commonly used in applications like payroll processing and log files, where data is processed in order.

### Algorithm:

1. Open `sequential.txt` file in append mode to store file information.

2. If the file fails to open, display an error message and return.

3. Create File Function:

- Take user input for filename and size.
- Write the filename and size into `sequential.txt`.
- Close the file and display a success message.

4. List Files Function:

- Open `sequential.txt` in read mode.
- If the file does not exist, display "No files found" and return.
- Read stored file records and print them sequentially.
- Close the file.

5. Main Function:

- Display menu options:
  - Add File
  - List Files
  - Exit
- Based on user input:
  - Call `createFile()` to add a new file entry.
  - Call `listFiles()` to display stored files.
  - Exit on choosing option 3.

## C-Program:

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

void createFile() {
    FILE *fp = fopen("sequential.txt", "a");
    if (!fp) {
        printf("Error opening file!\n");
        return;
    }

    char filename[20];
    int size;
```

```
    printf("Enter filename and size (KB): ");
    scanf("%s %d", filename, &size);
    fprintf(fp, "%s %d\n", filename, size);
    fclose(fp);

    printf("File stored sequentially!\n");
}

void listFiles() {
    FILE *fp = fopen("sequential.txt", "r");
    if (!fp) {
        printf("No files found!\n");
        return;
    }

    char filename[20];
    int size;
    printf("Stored Files:\n");
    while (fscanf(fp, "%s %d", filename, &size) != EOF) {
        printf("%s - %d KB\n", filename, size);
    }
    fclose(fp);
}

int main() {
    int choice;
    do {
        printf("\n1. Add File\n2. List Files\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);

        if (choice == 1) createFile();
        else if (choice == 2) listFiles();
    } while (choice != 3);

    return 0;
}
```

## Output

```
1. Add File
2. List Files
3. Exit
Enter choice: 1
Enter filename and size (KB): file1 100
File stored sequentially!
1. Add File
2. List Files
3. Exit
Enter choice: 1
Enter filename and size (KB): file2 200
File stored sequentially!
```

```
1. Add File
2. List Files
3. Exit
Enter choice: 2
Stored Files:
file1 - 100 KB
file2 - 200 KB
1. Add File
2. List Files
3. Exit
Enter choice: 3
```

## Objective: (b) Implement Random or Direct File Organization

### Description:

Random (or Direct) File Organization is a file storage method where records are accessed directly using a key or an address, rather than sequentially. This allows for fast retrieval of data without scanning the entire file. It is commonly used in databases, hashing, and indexed file systems to improve performance.

### Algorithm:

1. Define a File structure with filename and size attributes.

2. Create a file (random.dat) for random access storage.
3. Function createFile():

- Open random.dat in append binary mode (ab).
- Take filename and size as input from the user.
- Write the file details to random.dat using fwrite().
- Close the file.

4. Function readFile(position):

    - Open random.dat in read binary mode (rb).
    - Seek to the given position using fseek().
    - Read file details using fread().
    - Display file details if the position is valid.

5. In main():

    - Display a menu with options:
        - Add a file
        - Read file at a specific position
        - Exit
    - Perform operations based on the user's choice using a loop.

## C-Program:

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

typedef struct {
    char filename[20];
    int size;
} File;

void createFile() {
    FILE *fp = fopen("random.dat", "ab");
    if (!fp) {
        printf("Error opening file!\n");
```

```
        return;
    }

    File f;
    printf("Enter filename and size (KB): ");
    scanf("%s %d", f.filename, &f.size);
    fwrite(&f, sizeof(File), 1, fp);
    fclose(fp);

    printf("File stored!\n");
}

void readFile(int position) {
    FILE *fp = fopen("random.dat", "rb");
    if (!fp) {
        printf("No files found!\n");
        return;
    }

    File f;
    fseek(fp, position * sizeof(File), SEEK_SET);
    if (fread(&f, sizeof(File), 1, fp))
        printf("File at position %d: %s - %d KB\n", position, f.filename,
               f.size);
    else
        printf("Invalid position!\n");

    fclose(fp);
}

int main() {
    int choice, pos;
    do {
        printf("\n1. Add File\n2. Read File at Position\n3. Exit\nEnter
               choice: ");
        scanf("%d", &choice);

        if (choice == 1) createFile();
        else if (choice == 2) {
            printf("Enter position: ");
            scanf("%d", &pos);
            readFile(pos);
        }
    } while (choice != 3);

    return 0;
}
```

## Output

```
1. Add File
2. Read File at Position
3. Exit
Enter choice: 1
Enter filename and size (KB): file1 50
File stored!
1. Add File
2. Read File at Position
3. Exit
Enter choice: 1
Enter filename and size (KB): file2 100
File stored!
1. Add File
2. Read File at Position
3. Exit
Enter choice: 2
Enter position: 1
File at position 1: file2 - 100 KB
1. Add File
2. Read File at Position
3. Exit
Enter choice: 3
```

## Objective: (c) Implementation Serial File Organization

## Description:

Serial file organization is a method where data is stored sequentially, one record after another, in a file. Records are added at the end of the file, and retrieval typically requires searching through the entire file to locate a specific record. It is simple and efficient for small data sets but becomes slow for larger datasets.

## Algorithm:

1. Create File:
    - Open the file in append mode (`"a"`).
    - Take input for `filename` and `size`.
    - Write the input data (filename and size) to the file.
    - Close the file.

2. List Files:
    - Open the file in read mode (`"r"`).
    - Read the file contents (filename and size) line by line until the end of the file.
    - Display the filename and its size.
    - Close the file.

3. Main Program Logic:
    - Continuously prompt the user with options: Add a file, List files, or Exit.
    - Perform the corresponding action based on the user's choice:
    - If choice is 1, call `createFile()` to add a file.
    - If choice is 2, call `listFiles()` to display stored files.
    - Exit when the user chooses option 3.

# C-Program:

```c
#include <stdio.h>

void createFile() {
    FILE *fp = fopen("serial.txt", "a");
    if (!fp) {
        printf("Error opening file!\n");
        return;
    }

    char filename[20];
    int size;
    printf("Enter filename and size (KB): ");
    scanf("%s %d", filename, &size);
    fprintf(fp, "%s %d\n", filename, size);
    fclose(fp);

    printf("File stored in serial order!\n");
}

void listFiles() {
    FILE *fp = fopen("serial.txt", "r");
    if (!fp) {
```

```
        printf("No files found!\n");
        return;
    }

    char filename[20];
    int size;
    printf("Stored Files:\n");
    while (fscanf(fp, "%s %d", filename, &size) != EOF)
        printf("%s - %d KB\n", filename, size);
    fclose(fp);
}

int main() {
    int choice;
    do {
        printf("\n1. Add File\n2. List Files\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);

        if (choice == 1) createFile();
        else if (choice == 2) listFiles();
    } while (choice != 3);

    return 0;
}
```

## Output

```
1. Add File
2. List Files
3. Exit
Enter choice: 1
Enter filename and size (KB): file1.txt 10
File stored in serial order!
1. Add File
2. List Files
3. Exit
Enter choice: 2
Stored Files:
file1.txt - 10 KB
1. Add File
2. List Files
3. Exit
Enter choice: 3
```

## Objective: (d) Implement Indexed Sequential File Organization

## Description:

Indexed Sequential File Organization (ISAM) stores data sequentially with an index for faster access. The index allows quick retrieval of records, combining efficient searching with ordered data storage.

## Algorithm:

1. Create File:

- Open the file "indexed-seq.txt" in append mode.
- If file opening fails, display an error message and exit.
- Prompt the user to enter the filename, file size, and index.
- Store the input in the file as a line with the format: `index filename size`.
- Close the file and display a success message.

2. List Files:

- Open the file "indexed-seq.txt" in read mode.
- If file opening fails, display an error message and exit.
- Read the contents of the file line by line.
- For each line, extract the index, filename, and size.
- Display the extracted information in a readable format.
- Close the file.

3. Main Loop:

- Prompt the user to select an option:
  - Option 1: Call `createFile()` to add a new file.
  - Option 2: Call `listFiles()` to display stored files.
  - Option 3: Exit the program.

# C-Program:

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

void createFile() {
    FILE *fp = fopen("indexed-seq.txt", "a");
    if (!fp) {
        printf("Error opening file!\n");
        return;
    }

    char filename[20];
    int size, index;
    printf("Enter filename, size (KB), and index: ");
    scanf("%s %d %d", filename, &size, &index);
    fprintf(fp, "%d %s %d\n", index, filename, size);
    fclose(fp);

    printf("File stored with index!\n");
}
```

```c
void listFiles() {
    FILE *fp = fopen("indexed-seq.txt", "r");
    if (!fp) {
        printf("No files found!\n");
        return;
    }

    char filename[20];
    int size, index;
    printf("Stored Files:\n");
    while (fscanf(fp, "%d %s %d", &index, filename, &size) != EOF)
        printf("Index: %d | %s - %d KB\n", index, filename, size);
    fclose(fp);
}

int main() {
    int choice;
    do {
        printf("\n1. Add File\n2. List Files\n3. Exit\nEnter choice:");
        scanf("%d", &choice);

        if (choice == 1) createFile();
        else if (choice == 2) listFiles();
    } while (choice != 3);

    return 0;
}
```

## Output

```
1. Add File
2. List Files
3. Exit
Enter choice: 1
Enter filename, size (KB), and index: file1.txt 100 1
File stored with index!

1. Add File
2. List Files
3. Exit
Enter choice: 1
Enter filename, size (KB), and index: file2.txt 200 2
File stored with index!

1. Add File
2. List Files
3. Exit
Enter choice: 2
Stored Files:
Index: 1 | file1.txt - 100 KB
Index: 2 | file2.txt - 200 KB
```

## Objective: (e) Implement Heap (Unordered) File Organization

## Description:

Heap (Unordered) File Organization stores records in random order without any indexing. New records are appended at the end, and searching requires scanning the entire file, making it simple but inefficient for large datasets.

## Algorithm:

1. Create File:

- Open the file `heap.txt` in append mode.
- If the file cannot be opened, display an error message.
- Generate a random position using `rand()` to simulate unordered storage.
- Accept the filename and its size (in KB) as input from the user.
- Write the random position, filename, and size to the file.
- Close the file after writing.

2. List Files:

- Open the file `heap.txt` in read mode.
- If the file cannot be opened, display a message indicating no files are found.
- Read each record (position, filename, and size) from the file.
- Display the stored files with their respective positions and sizes.
- Close the file after reading.

3. Main Program:

- Present a menu with three options: Add File, List Files, and Exit.
- Continue prompting the user for input until the Exit option is selected.

# C-Program:

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

void createFile() {
    FILE *fp = fopen("heap.txt", "a");
    if (!fp) {
        printf("Error opening file!\n");
        return;
    }
    char filename[20];
    int size, randomPos;
    // Random Position Generataion (Unordered Storage Concept)
    srand(time(0));  // Random seed set here
    randomPos = rand() % 100;  // Random Position (Example: 0-99)
    printf("Enter filename and size (KB): ");
    scanf("%s %d", filename, &size);
    fprintf(fp, "%d %s %d\n", randomPos, filename, size);
    fclose(fp);

    printf("File stored at random position %d (unordered)!\n",randomPos);
}
```

```c
void listFiles() {
    FILE *fp = fopen("heap.txt", "r");
    if (!fp) {
        printf("No files found!\n");
        return;
    }
    int pos, size;
    char filename[20];
    printf("Stored Files (Unordered):\n");
    while (fscanf(fp, "%d %s %d", &pos, filename, &size) != EOF)
        printf("Position: %d | %s - %d KB\n", pos, filename, size);
    fclose(fp);
}
int main() {
    int choice;
    do {
        printf("\n1. Add File\n2. List Files\n3. Exit\nEnter choice:
               ");
        scanf("%d", &choice);

        if (choice == 1) createFile();
        else if (choice == 2) listFiles();
    } while (choice != 3);
    return 0;
}
```

## Output

```
1. Add File
2. List Files
3. Exit
Enter choice: 1
Enter filename and size (KB): file1.txt 20
File stored at random position 47 (unordered)!
1. Add File
2. List Files
3. Exit
Enter choice: 1
Enter filename and size (KB): file2.txt 30
File stored at random position 12 (unordered)!
1. Add File
2. List Files
3. Exit
Enter choice: 2
Stored Files (Unordered):
Position: 47 | file1.txt - 20 KB
Position: 12 | file2.txt - 30 KB
```

# 07: پروگرام نمبر

## Objective: Implement Bankers algorithm for Dead Lock Avoidance

## Description:

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems. It ensures that resources are allocated in a way that avoids unsafe states, preventing deadlocks. The algorithm checks if granting a resource request leads to a safe state, where all processes can eventually finish. It uses matrices like Available, Max, Allocation, and Need to determine safety.

## Algorithm:

1. Input Data:

- Read the Allocation matrix, Max matrix, and Available resources from the user.
- Calculate the Need matrix using the formula: `Need[i][j] = Max[i][j] - Allocation[i][j]`.

2. Initialize:

- Initialize a `finish` array to track which processes have finished.
- Initialize a `work` array to hold available resources.
- Initialize a `safeSeq` array to store the safe sequence of processes

3. Check Safety:

- Set `work` as the available resources.
- For each process, check if it can proceed (i.e., if its needs can be met with the current `work` resources).
- If a process can proceed, allocate resources by adding the allocated resources back to `work`, mark the process as finished, and add it to the safe sequence.
- Repeat until all processes are finished or no more processes can proceed.

4. Result:

- If all processes can be finished, print the safe sequence.
- If no safe sequence can be found, print that the system is in an unsafe state.

## C-Program:

```c
#include <stdio.h>

#define P 5  // Number of processes
#define R 3  // Number of resources

int alloc[P][R], max[P][R], need[P][R], avail[R];

void inputData() {
    printf("Enter Allocation Matrix:\n");
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            scanf("%d", &alloc[i][j]);

    printf("Enter Max Matrix:\n");
```

```
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++) {
            scanf("%d", &max[i][j]);
            need[i][j] = max[i][j] - alloc[i][j]; // Need = Max - Alloc
        }

    printf("Enter Available Resources:\n");
    for (int i = 0; i < R; i++)
        scanf("%d", &avail[i]);
}

int isSafe() {
    int finish[P] = {0}, safeSeq[P], work[R], count = 0;

    for (int i = 0; i < R; i++) work[i] = avail[i];

    while (count < P) {
        int found = 0;
        for (int i = 0; i < P; i++) {
            if (!finish[i]) {
                int canAllocate = 1;
                for (int j = 0; j < R; j++)
                    if (need[i][j] > work[j])
                        canAllocate = 0;

                if (canAllocate) {
                    for (int k = 0; k < R; k++)
                        work[k] += alloc[i][k];

                    safeSeq[count++] = i;
                    finish[i] = 1;
                    found = 1;
                }
            }
        }
        if (!found) {
            printf("System is in an Unsafe State!\n");
            return 0;
        }
    }

    printf("Safe Sequence: ");
    for (int i = 0; i < P; i++)
        printf("P%d ", safeSeq[i]);
    printf("\n");
    return 1;
}

int main() {
    inputData();
    isSafe();
    return 0;
}
```

## Output

```
Enter Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter Max Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter Available Resources:
3 3 2
Safe Sequence: P1 P3 P4 P0 P2
```

# 08: پروگرام نمبر

## Objective: Implement an Algorithm for Dead Lock Detection

## Description:

Deadlock detection identifies if a system is in a deadlock state by checking for circular waits or blocked processes. It involves monitoring processes and resources, and detecting cycles in the resource allocation graph. If a cycle is found, a deadlock is detected, and corrective actions can be taken.

## Algorithm:

1. Initialize: Set up the finish array to keep track of processes that can be completed. Set `finish[i] = 0` for all processes. Initialize `work` as the available resources.
2. Check Processes: Loop through each process and check if its request can be fulfilled by the available resources (`request[i][j] <= work[j]`).
3. Allocate Resources: If a process can be executed (its request can be fulfilled), add its allocated resources to `work`, mark it as finished (`finish[i] = 1`), and update the available resources.
4. Repeat: Continue the process until either all processes are finished or no process can be executed (deadlock detected).
5. Output: If any process remains unfinished and cannot be executed, it is in a deadlock. Otherwise, the system is safe.

# C-Program:

```c
#include <stdio.h>

#define P 3  // Number of processes
#define R 3  // Number of resources

int alloc[P][R], request[P][R], avail[R];

void inputData() {
    printf("Enter Allocation Matrix:\n");
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            scanf("%d", &alloc[i][j]);

    printf("Enter Request Matrix:\n");
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            scanf("%d", &request[i][j]);

    printf("Enter Available Resources:\n");
    for (int i = 0; i < R; i++)
        scanf("%d", &avail[i]);
}
int detectDeadlock() {
    int finish[P] = {0}, work[R], deadlock = 0;
```

```
    for (int i = 0; i < R; i++) work[i] = avail[i];

    while (1) {
        int found = 0;
        for (int i = 0; i < P; i++) {
            if (!finish[i]) {
                int canExecute = 1;
                for (int j = 0; j < R; j++)
                    if (request[i][j] > work[j])
                        canExecute = 0;
                if (canExecute) {
                    for (int k = 0; k < R; k++)
                        work[k] += alloc[i][k];

                    finish[i] = 1;
                    found = 1;
                }
            }
        }
        if (!found) break;
    }
    for (int i = 0; i < P; i++)
        if (!finish[i]) {
            deadlock = 1;
            printf("Deadlock detected! Process P%d is in deadlock.\n",i);
        }
    if (!deadlock) printf("No Deadlock detected! System is safe.\n");
    return deadlock;
}
int main() {
    inputData();
    detectDeadlock();
    return 0;
}
```

## Output

```
Enter Allocation Matrix:
1 2 0
1 1 1
0 0 1
Enter Request Matrix:
1 2 3
0 0 2
2 1 0
Enter Available Resources:
2 3 0
Deadlock detected! Process P0 is in deadlock.
Deadlock detected! Process P1 is in deadlock.
```

# 09: پروگرام نمبر

## Implementation of all-page replacement algorithms

### Objective: (a) Implement FIFO Page Replacement

### Description:

FIFO (First In First Out) Page Replacement is a simple page replacement algorithm where the oldest page in memory is replaced when a new page needs to be loaded. The pages are kept in a queue, and when a page fault occurs, the page at the front of the queue is removed and replaced by the new page. This method ensures that pages are replaced in the order they were brought into memory.

### Algorithm:

1. Initialize an empty set of frames.
2. For each page reference:
- Check if the page is already in one of the frames (Page Hit).
- If not, replace the oldest page in the frames (Page Fault), and move the FIFO pointer to the next position.
3. Repeat for all page references.
4. Count the total number of page faults.

## C-Program:

```c
#include <stdio.h>

#define MAX_FRAMES 3  // Number of frames
#define MAX_PAGES 7   // Number of page references

int main() {
    int pages[MAX_PAGES] = {1, 3, 0, 3, 5, 6, 3}; // Reference string
    int frames[MAX_FRAMES] = {-1, -1, -1}; // Initialize frames
    int front = 0, pageFaults = 0;

    printf("Page Reference | Frames       | Status\n");
    for (int i = 0; i < MAX_PAGES; i++) {
        int found = 0;
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == pages[i]) {
                found = 1;
                break;
            }
        }

        printf("       %d       | ", pages[i]);
        if (found) {
            printf("%d %d %d | Page Hit \n", frames[0], frames[1],frames[2]);
        } else {
            frames[front] = pages[i];  // Replace oldest page
            front = (front + 1) % MAX_FRAMES; // Move FIFO pointer
            pageFaults++;
            printf("%d %d %d | Page Fault \n", frames[0], frames[1],
```

```
                    frames[2]);
            }
        }

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

## Output

```
Page Reference | Frames       | Status
       1       | 1 -1 -1 | Page Fault
       3       | 1 3 -1 | Page Fault
       0       | 1 3 0 | Page Fault
       3       | 1 3 0 | Page Hit
       5       | 5 3 0 | Page Fault
       6       | 5 6 0 | Page Fault
       3       | 5 6 3 | Page Fault
Total Page Faults: 6
```

## Objective: (b) Implement Optimal Page Replacement

## Description:

Optimal Page Replacement (OPT) is a page replacement algorithm that replaces the page that will not be used for the longest period of time in the future. It provides the minimum number of page faults compared to other algorithms, but it is not implementable in practice because it requires future knowledge of page references. It is often used as a benchmark for comparing the efficiency of other page replacement algorithms.

## Algorithm:

1. Initialization:
- Define the total number of frames (`MAX_FRAMES`) and the number of page references (`MAX_PAGES`).
- Initialize the frames as empty (with -1 indicating that the frame is empty).
- Define the page reference string to be processed.
2. Page Reference Check:
- For each page reference, check if it is already in one of the frames.
- If the page is found, it's a Page Hit. No action is taken except printing the current frames.
3. Page Fault Handling:
- If the page is not found in the frames, it's a Page Fault.
- Find the page that should be replaced:
  - For each frame, predict which one should be replaced by checking which will be used farthest in the future (or not at all).
  - Replace the selected frame with the current page.
  - Increment the page fault counter.
4. Output:
   - After processing all pages, print the total number of page faults.

# C-Program:

```c
#include <stdio.h>

#define MAX_FRAMES 3  // Number of frames
#define MAX_PAGES 12  // Number of page references

int predict(int pages[], int frames[], int index) {
    int farthest = index, replace = -1;
    for (int i = 0; i < MAX_FRAMES; i++) {
        int j;
        for (j = index; j < MAX_PAGES; j++) {
            if (frames[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    replace = i;
                }
                break;
            }
        }
        if (j == MAX_PAGES) return i;
    }
    return (replace == -1) ? 0 : replace;
}
```

```c
int main() {
    // New Reference String
    int pages[MAX_PAGES] = {7, 0, 1, 2, 0, 3, 4, 2, 3, 0, 3, 2};
    int frames[MAX_FRAMES] = {-1, -1, -1}; // Initialize frames
    int pageFaults = 0;

    printf("Page Reference | Frames       | Status\n");
    for (int i = 0; i < MAX_PAGES; i++) {
        int found = -1;
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == pages[i]) {
                found = j;
                break;
            }
        }
        printf("       %d       | ", pages[i]);
        if (found != -1) { // Page Hit
            printf("%d %d %d | Page Hit \n", frames[0], frames[1],frames[2]);
        } else { // Page Fault
            int replaceIndex = (i < MAX_FRAMES) ? i : predict(pages,frames,i);
            frames[replaceIndex] = pages[i];
            pageFaults++;
            printf("%d %d %d | Page Fault \n", frames[0], frames[1],
                   frames[2]);
        }
    }
    printf("Total Page Faults: %d\n", pageFaults);
    return 0;
}
```

## Output

```
Page Reference | Frames       | Status
       7       | 7 -1 -1 | Page Fault
       0       | 7 0 -1 | Page Fault
       1       | 7 0 1 | Page Fault
       2       | 2 0 1 | Page Fault
       0       | 2 0 1 | Page Hit
       3       | 2 0 3 | Page Fault
       4       | 2 4 3 | Page Fault
       2       | 2 4 3 | Page Hit
       3       | 2 4 3 | Page Hit
       0       | 2 0 3 | Page Fault
       3       | 2 0 3 | Page Hit
       2       | 2 0 3 | Page Hit
Total Page Faults: 7
```

## Objective: (c) Implement LRU Page Replacement

## Description:

Least Recently Used (LRU) Page Replacement is a page replacement algorithm that replaces the page that has not been used for the longest period of time. It keeps track of the order in which pages are accessed and selects the least recently used page for replacement when a new page needs to be loaded into memory. LRU aims to minimize page faults by ensuring that the most recently accessed pages remain in memory.

## Algorithm:

1. Initialize:
   - Create an array `frames[]` to store the pages in memory, initialized to -1 (empty).
   - Create an array `recent[]` to track the usage order of pages, initialized to 0.
   - Set `time` to track the usage time of pages.
   - Set `pageFaults` to count the number of page faults.
2. For each page in the reference string:
   - Check if the page is already in the memory (`frames[]`):
   - If yes, it's a Page Hit, update the `recent[]` array with the current `time`.
   - If no, it's a Page Fault:
   - Find the Least Recently Used (LRU) page by checking which page has the smallest value in `recent[]`.
   - Replace the LRU page with the new page and update the `recent[]` array.
   - Increment the `pageFaults` counter.
3. Print the status for each page reference:
   - Display the current page reference, the frames, and whether it was a "Page Hit" or "Page Fault."
4. Finally, display the total number of page faults.

# C-Program:

```c
#include <stdio.h>

#define MAX_FRAMES 3  // Number of frames
#define MAX_PAGES 10  // Number of page references

int main() {
    // New Reference String
    int pages[MAX_PAGES] = {2, 3, 1, 5, 3, 4, 1, 2, 3, 5};
    int frames[MAX_FRAMES] = {-1, -1, -1}; // Initialize frames
    int recent[MAX_FRAMES] = {0, 0, 0}; // Track usage order
    int pageFaults = 0, time = 0;

    printf("Page Reference | Frames       | Status\n");
    for (int i = 0; i < MAX_PAGES; i++) {
        int found = -1;
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == pages[i]) {
                found = j;
                break;
            }
        }

        printf("       %d       | ", pages[i]);
```

```
        if (found != -1) { // Page Hit
            recent[found] = time++; // Update recent usage time
            printf("%d %d %d | Page Hit \n", frames[0], frames[1],frames[2]);
        } else { // Page Fault
            int lru = 0; // Find LRU page index
            for (int j = 1; j < MAX_FRAMES; j++)
                if (recent[j] < recent[lru])
                    lru = j;

            frames[lru] = pages[i];  // Replace LRU page
            recent[lru] = time++; // Update recent usage time
            pageFaults++;
            printf("%d %d %d | Page Fault \n", frames[0], frames[1],
                   frames[2]);
        }
    }

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

## Output

```
Page Reference | Frames       | Status
       2       | 2 -1 -1 | Page Fault
       3       | 3 -1 -1 | Page Fault
       1       | 3 1 -1 | Page Fault
       5       | 3 1 5 | Page Fault
       3       | 3 1 5 | Page Hit
       4       | 3 4 5 | Page Fault
       1       | 3 4 1 | Page Fault
       2       | 2 4 1 | Page Fault
       3       | 2 3 1 | Page Fault
       5       | 2 3 5 | Page Fault
Total Page Faults: 9
```

# 10: پروگرام نمبر

## Objective: Implement Shared memory and IPC

## Description:

Shared memory is an IPC method where multiple processes share a common memory space for fast data exchange. It is efficient but needs synchronization (e.g., semaphores) to avoid conflicts. IPC encompasses all techniques allowing processes to communicate, including shared memory, message passing, and sockets.

## Algorithm:

1. Create Shared Memory:
   - Use `shmget()` to create a shared memory segment with a specified size.
   - Return a shared memory ID (`shmid`).
2. Fork a Child Process:
   - Call `fork()` to create a child process.
   - Parent and child will communicate via the shared memory.
3. Child Process:
   - Attach the shared memory using `shmat()`.
   - Read the message written by the parent process.
   - Write a response back to the parent using `strcpy()` to write into shared memory.
   - Detach the shared memory using `shmdt()`.
4. Parent Process:
   - Attach the shared memory using `shmat()`.
   - Write a message into the shared memory for the child process.
   - Wait for the child to finish (using `wait()`).
   - Read the response from the child process.
   - Write another message to the shared memory.
   - Detach the shared memory using `shmdt()`.
5. Clean up:
   - After communication is done, remove the shared memory using `shmctl()` with the `IPC_RMID` flag.

## C-Program:

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

#define SHM_SIZE 1024  // Shared memory size

int main() {
    int shmid;
    char *shm;

    // Step 1: Create shared memory
    shmid = shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
```

```
        printf("Failed to create shared memory!\n");
        return 1;
    }

    // Step 2: Fork a child process
    pid_t pid = fork();

    if (pid == 0) {  // Child Process
        sleep(1);  // Ensure parent writes first
        shm = (char *)shmat(shmid, NULL, 0);
        printf("Child reads: %s\n", shm);

        // Child sends response to parent
        strcpy(shm, "Hello Parent! How are you?");
        printf("Child writes: %s\n", shm);
        sleep(1);

        // Child reads second message from parent
        printf("Child reads again: %s\n", shm);

        shmdt(shm);
    }
    else {  // Parent Process
        shm = (char *)shmat(shmid, NULL, 0);

        // Parent sends first message to child
        strcpy(shm, "Hello Child! How are you?");
        printf("Parent writes: %s\n", shm);

        wait(NULL); // Wait for child to respond

        // Parent reads child's response
        printf("Parent reads: %s\n", shm);

        // Parent sends another message
        strcpy(shm, "I'm doing great! Take care.");
        printf("Parent writes: %s\n", shm);

        shmdt(shm);
        shmctl(shmid, IPC_RMID, NULL); // Remove shared memory
    }
    return 0;
}
```

## Output

```
Parent writes: Hello Child! How are you?
Child reads: Hello Child! How are you?
Child writes: Hello Parent! How are you?
Child reads again: Hello Parent! How are you?
Parent reads: Hello Parent! How are you?
Parent writes: I'm doing great! Take care.
```

# 11: پروگرام نمبر

## Objective: Implement Paging Technique memory management

## Description:

Paging is a memory management technique that eliminates fragmentation by dividing physical memory into fixed-size blocks called frames and logical memory into pages of the same size. The operating system maintains a page table to map logical addresses to physical addresses. This allows efficient memory allocation and retrieval without requiring contiguous memory space.

## Algorithm:

1. Initialize Page Table
   - Create a page table that maps logical pages to physical frames.
   - Assign each logical page a corresponding frame.
2. User Input for Logical Address
   - Prompt the user to enter a logical address.
   - If the input is `-1`, terminate the program.
3. Calculate Page Number and Offset
   - Compute Page Number = `Logical Address / Page Size`
   - Compute Offset = `Logical Address % Page Size`
4. Check for Valid Page Number
   - If the page number is within the allocated range, proceed.
   - Otherwise, display an error message.
5. Compute Physical Address
   - Use the page table to get the frame number.
   - Compute Physical Address = `Frame Number * Page Size + Offset`
6. Display Output
   - Print the Page Number, Offset, and Physical Address.
7. Repeat Until Exit Condition
   - Continue accepting logical addresses until the user chooses to exit.

## C-Program:

```c
#include <stdio.h>

#define PAGE_SIZE 4   // Page size (4KB)
#define TOTAL_PAGES 5 // Total pages in memory

int main() {
    int pageTable[TOTAL_PAGES]; // Page Table
    int logicalAddress, pageNumber, offset, physicalAddress;
    // Initialize Page Table (Mapping Logical to Physical Pages)
    printf("Initializing Page Table...\n");
    for (int i = 0; i < TOTAL_PAGES; i++) {
        pageTable[i] = i; // Mapping Logical Page -> Physical Page
        printf("Page %d -> Frame %d\n", i, pageTable[i]);
    }
    while (1) {
        // Input logical address
        printf("\nEnter Logical Address (0 - %d) or -1 to Exit: ",
               PAGE_SIZE * TOTAL_PAGES - 1);
```

```
            scanf("%d", &logicalAddress);
            // Exit Condition
            if (logicalAddress == -1) {
                printf("Exiting program...\n");
                break;
            }
            // Calculate Page Number and Offset
            pageNumber = logicalAddress / PAGE_SIZE;
            offset = logicalAddress % PAGE_SIZE;
            // Check if Page Number is valid
            if (pageNumber < TOTAL_PAGES) {
                physicalAddress = pageTable[pageNumber] * PAGE_SIZE + offset;
                printf("Page Number: %d\nOffset: %d\n", pageNumber, offset);
                printf("Logical Address: %d -> Physical Address: %d\n",
                       logicalAddress, physicalAddress);
            } else {
                printf("Invalid Logical Address! Out of Bound.\n");
            }
        }
        return 0;
}
```

## Output

```
Initializing Page Table...
Page 0 -> Frame 0
Page 1 -> Frame 1
Page 2 -> Frame 2
Page 3 -> Frame 3
Page 4 -> Frame 4

Enter Logical Address (0 - 19) or -1 to Exit: 2
Page Number: 0
Offset: 2
Logical Address: 2 -> Physical Address: 2

Enter Logical Address (0 - 19) or -1 to Exit: 5
Page Number: 1
Offset: 1
Logical Address: 5 -> Physical Address: 5

Enter Logical Address (0 - 19) or -1 to Exit: -1
Exiting program...
```

# 12: پروگرام نمبر

## Objective: Implement Threading & Synchronization Applications

## Description:

Threading and synchronization are essential in multi-threaded applications to ensure efficient execution and prevent race conditions. Threading allows parallel execution of tasks, while synchronization ensures proper coordination between threads using mechanisms like mutexes, semaphores, and condition variables. These techniques improve performance and maintain data consistency in applications like operating systems, databases, and real-time systems.

## Algorithm:

1. Initialize a shared resource (`counter`) and a mutex lock.
2. Create a thread function (`increment`):
   - Lock the mutex before modifying `counter`.
   - Increment the `counter` and print the thread ID with the updated value.
   - Unlock the mutex after execution.
3. In `main()` function:
   - Initialize the mutex.
   - Create two threads that run the `increment` function.
   - Wait for both threads to finish execution using `pthread_join()`.
   - Destroy the mutex to free resources.
4. Print the final counter value and exit.

## C-Program:

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

int counter = 0;  // Shared resource
pthread_mutex_t lock;  // Mutex lock

void* increment(void* arg) {
    pthread_mutex_lock(&lock);  // Lock critical section
    counter++;
    printf("Thread %d: Counter = %d\n", *(int*)arg, counter);
    pthread_mutex_unlock(&lock);  // Unlock after execution
    return NULL;
}

int main() {
    pthread_t t1, t2;
    int id1 = 1, id2 = 2;

    pthread_mutex_init(&lock, NULL); // Initialize mutex

    pthread_create(&t1, NULL, increment, &id1);
    pthread_create(&t2, NULL, increment, &id2);
```

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

    pthread_mutex_destroy(&lock); // Destroy mutex

    printf("Final Counter: %d\n", counter);
    return 0;
}
```

## Output

```
Thread 1: Counter = 1
Thread 2: Counter = 2
Final Counter: 2
```