Lab Final

Lab M: Implementation in shell

```
#!/bin/bash

# Basic arithmetic using let


let a=5+4

echo $a   # 9


let "a = 5 + 4"

echo $a   # 9


let a++

echo $a   # 10


let "a = 4 * 5"

echo $a   # 20


let "a = $1 + 30"

echo $a   # depends on input (if $1 = 15 → 45)
```

Lab Number-03

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

```
#!/bin/bash

# Program to find area and circumference of a circle


echo "Enter radius of circle:"

read r
```

```
area=$(echo "3.1416 * $r * $r" | bc)
circumference=$(echo "2 * 3.1416 * $r" | bc)

echo "Area of Circle = $area"
echo "Circumference of Circle = $circumference"
```

2. Write a Shell program to find the roots of a quadratic equation.

```
#!/bin/bash
# Program to find roots of quadratic equation

echo "Enter values of a, b, c:"
read a b c

d=$(echo "$b * $b - 4 * $a * $c" | bc)  # discriminant

if [ $d -lt 0 ]; then
    echo "Roots are imaginary"
elif [ $d -eq 0 ]; then
    root=$(echo "scale=2; -$b / (2 * $a)" | bc)
    echo "Roots are real and equal: $root"
else
    sqrt_d=$(echo "scale=2; sqrt($d)" | bc -l)
    root1=$(echo "scale=2; (-$b + $sqrt_d) / (2 * $a)" | bc -l)
    root2=$(echo "scale=2; (-$b - $sqrt_d) / (2 * $a)" | bc -l)
    echo "Roots are real and different: $root1 , $root2"
fi
```

3. Write a Shell program to Find out the Area and Perimeter of Rectangle.

```
#!/bin/bash

# Program to find area and perimeter of a rectangle


echo "Enter length and breadth:"

read l b


area=$(echo "$l * $b" | bc)

perimeter=$(echo "2 * ($l + $b)" | bc)


echo "Area of Rectangle = $area"

echo "Perimeter of Rectangle = $perimeter"
```

4. Write a Shell program to Find the Perimeter of a Circle, Rectangle and Triangle

```
#!/bin/bash

# Program to find perimeter of Circle, Rectangle, and Triangle


echo "Choose shape: "

echo "1. Circle"

echo "2. Rectangle"

echo "3. Triangle"

read choice


case $choice in

  1)

    echo "Enter radius:"
```

```
    read r
    perimeter=$(echo "2 * 3.1416 * $r" | bc)
    echo "Perimeter (Circumference) of Circle = $perimeter"
    ;;
  2)
    echo "Enter length and breadth:"
    read l b
    perimeter=$(echo "2 * ($l + $b)" | bc)
    echo "Perimeter of Rectangle = $perimeter"
    ;;
  3)
    echo "Enter three sides a, b, c:"
    read a b c
    perimeter=$(echo "$a + $b + $c" | bc)
    echo "Perimeter of Triangle = $perimeter"
    ;;
  *)
    echo "Invalid choice!"
    ;;
Esac
```

Lab Exercise

Write a Shell program to find the sum of odd and even numbers from a set of numbers.

```
#!/bin/bash
# Program to find the sum of odd and even numbers from a set of numbers

echo "Enter numbers separated by space:"
```

```
read -a arr   # read input into an array

sum_even=0
sum_odd=0

for num in "${arr[@]}"
do
   if (( num % 2 == 0 )); then
      sum_even=$((sum_even + num))
   else
      sum_odd=$((sum_odd + num))
   fi
done

echo "Sum of Even Numbers = $sum_even"
echo "Sum of Odd Numbers  = $sum_odd"
```

Write a Shell program to Check Triangle is Valid or Not

```
#!/bin/bash
# Program to check whether a triangle is valid or not

echo "Enter three sides of triangle:"
read a b c

if (( a + b > c && a + c > b && b + c > a )); then
   echo "The triangle is VALID"
else
```

```
    echo "The triangle is NOT VALID"
fi
```

Lab Number -04

Lab Task

1. Write a Shell program to find the sum of odd and even numbers from a set of numbers.

```
#!/bin/bash
# Program to find sum of odd and even numbers

echo "Enter numbers separated by space:"
read -a arr

sum_even=0
sum_odd=0

for num in "${arr[@]}"
do
    if (( num % 2 == 0 )); then
        sum_even=$((sum_even + num))
    else
        sum_odd=$((sum_odd + num))
    fi
done

echo "Sum of Even Numbers = $sum_even"
echo "Sum of Odd Numbers  = $sum_odd"
```

2. Write a Shell program to find the smallest number from a set of numbers.

```
#!/bin/bash
# Program to find the smallest number

echo "Enter numbers separated by space:"
read -a arr

smallest=${arr[0]}

for num in "${arr[@]}"
do
   if (( num < smallest )); then
      smallest=$num
   fi
done

echo "Smallest Number = $smallest"
```

3. Write a Shell program to find the sum of all numbers between 50 and 100, which are divisible by 3 and not divisible by 5.

```
#!/bin/bash
# Program to find sum of numbers between 50 and 100 divisible by 3 but not by 5
sum=0
for (( i=50; i<=100; i++ ))
do
   if (( i % 3 == 0 && i % 5 != 0 )); then
      sum=$((sum + i))
```

```
    fi
done
echo "Sum = $sum"
```

4.Write a Shell program to find the second highest number from a set of numbers.

```
#!/bin/bash
# Program to find second highest number

echo "Enter numbers separated by space:"
read -a arr

# Sort array in ascending order
sorted=($(for i in "${arr[@]}"; do echo $i; done | sort -n))

len=${#sorted[@]}
second_highest=${sorted[$((len-2))]}

echo "Second Highest Number = $second_highest"
```

5. Write a Shell program to find the factorial of a number using for loop.

```
#!/bin/bash
# Program to find factorial of a number

echo "Enter a number:"
read n

fact=1
```

```
for (( i=1; i<=n; i++ ))
do
    fact=$((fact * i))
done

echo "Factorial of $n = $fact"
```

6. Write a Shell program to generate Fibonacci series.

```
#!/bin/bash
# Program to generate Fibonacci series

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
```

2.1.4 Lab Task

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

```
#!/bin/bash

echo -n "Enter a number: "
read num

smallest=9

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

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

2. Write a Shell program to find the second largest digit from a number.

```
#!/bin/bash

echo -n "Enter a number: "
read num
```

```
largest=-1
second=-1

while [ $num -gt 0 ]
do
    digit=$((num % 10))

    if [ $digit -gt $largest ]
    then
        second=$largest
        largest=$digit
    elif [ $digit -gt $second ] && [ $digit -ne $largest ]
    then
        second=$digit
    fi

    num=$((num / 10))
done

if [ $second -eq -1 ]
then
    echo "No second largest digit found (all digits same)"
else
    echo "Second largest digit is: $second"
fi
```

3. Write a Shell program to find the sum of digits of a number.

```
#!/bin/bash

echo -n "Enter a number: "
read num

sum=0

while [ $num -gt 0 ]
do
   digit=$((num % 10))
   sum=$((sum + digit))
   num=$((num / 10))
done

echo "Sum of digits = $sum"
```

4. Write a Shell program to check the given integer is Armstrong number or not.

```
#!/bin/bash

echo -n "Enter a number: "
read num

original=$num
n=${#num}   # number of digits
sum=0

while [ $num -gt 0 ]
do
   digit=$((num % 10))
   power=1
   for ((i=1; i<=n; i++))
   do
      power=$((power * digit))
   done
   sum=$((sum + power))
   num=$((num / 10))
done

if [ $sum -eq $original ]
then
   echo "$original is an Armstrong number"
else
```

```
    echo "$original is NOT an Armstrong number"
fi
```

2.1.5 Until Loop

Lab Task

1. Write a Shell program to find the largest number between two numbers using function.

```
#!/bin/bash

# Function to find largest
largest() {
   if [ $1 -gt $2 ]
   then
      echo "$1 is larger"
   else
      echo "$2 is larger"
   fi
}

echo -n "Enter first number: "
read a
echo -n "Enter second number: "
read b

largest $a $b
```

2. Write a Shell program to find the sum of the numbers passed as parameters.

```
#!/bin/bash


# Function to calculate sum

sum_numbers() {

   sum=0

   for num in "$@"

   do

      sum=$((sum + num))

   done

   echo "Sum = $sum"

}
```

```
# Calling function with all command line arguments

sum_numbers "$@"
```

2.3 Arrays

Lab Task

1. Write a Shell program to find the sum of odd and even numbers.

```
#!/bin/bash

echo -n "Enter numbers (space separated): "
read -a arr

sum_even=0
sum_odd=0

for num in "${arr[@]}"
do
   if (( num % 2 == 0 ))
   then
      sum_even=$((sum_even + num))
   else
      sum_odd=$((sum_odd + num))
   fi
done

echo "Sum of even numbers = $sum_even"
echo "Sum of odd numbers  = $sum_odd"
```

2. Write a Shell program to find the average of n numbers.

```
#!/bin/bash

echo -n "Enter numbers (space separated): "
read -a arr

sum=0
count=${#arr[@]}

for num in "${arr[@]}"
do
   sum=$((sum + num))
done

average=$(echo "scale=2; $sum / $count" | bc)
```

```
echo "Average = $average"
```

3. Write a Shell Program to find the largest element of an array.

```
#!/bin/bash

echo -n "Enter numbers (space separated): "
read -a arr

smallest=${arr[0]}

for num in "${arr[@]}"
do
   if [ $num -lt $smallest ]
   then
      smallest=$num
   fi
done

echo "Smallest element = $smallest"
```

4. Write a Shell Program to find the smallest element of an array.

```
#!/bin/bash


echo -n "Enter numbers (space separated): "

read -a arr


smallest=${arr[0]}


for num in "${arr[@]}"

do

   if [ $num -lt $smallest ]

   then

      smallest=$num
```

```
    fi
done

echo "Smallest element = $smallest"
```

4 Lab Exercise

• Write a shell program to display odd position numbers (using For loop).

Sample Input:

Enter 7-digit number: 5867458

Output:

5

6

4

8

```
#!/bin/bash

echo -n "Enter a number: "
read num

len=${#num}

echo "Odd position digits:"
for ((i=0; i<$len; i+=2))   # 0,2,4... (odd positions because index starts from 0)
do
    echo "${num:$i:1}"
done
```

• Write a Shell program using while loop:

Sample Input:

Enter the number: 148541547854

Output:

1 = 2 times

4 = 4 times

8 = 2 times

5 = 3 times

7 = 1 times

```
#!/bin/bash


echo -n "Enter the number: "
read num


for d in {0..9}
do
   count=0
   temp=$num
   while [ $temp -gt 0 ]
   do
      digit=$((temp % 10))
      if [ $digit -eq $d ]
      then
         count=$((count + 1))
      fi
      temp=$((temp / 10))
```

```
    done
    if [ $count -gt 0 ]
    then
        echo "$d = $count times"
    fi
done
```

• Write a Shell program to find the 2nd highest and 3rd highest numbers from a set of numbers and sum of them using array.

Sample Input:

Enter the number of elements: 5

Enter the number: 10

Enter the number: 21

Enter the number: 30

Enter the number: 17

Enter the number: 5

Output:

The sum of first and last element is: (21+17) = 38

```
#!/bin/bash

echo -n "Enter the number of elements: "
read n

arr=()
for ((i=0; i<n; i++))
do
    echo -n "Enter number: "
```

```
    read val
    arr+=($val)
done

# Sort array in descending order
sorted=($(printf "%s\n" "${arr[@]}" | sort -nr))

second=${sorted[1]}
third=${sorted[2]}
sum=$((second + third))

echo "2nd highest = $second"
echo "3rd highest = $third"
echo "Sum = $sum"
```

• Write a Shell program to find the factorial of two different numbers and sum of the numbers using function.

Sample Input:

Factorial of 5 is 120

Factorial of 6 is 720

Output:

120 + 720 = 840

```
#!/bin/bash

factorial() {
    num=$1
    fact=1
    for ((i=1; i<=num; i++))
```

```
    do
        fact=$((fact * i))
    done
    echo $fact
}

echo -n "Enter first number: "
read a
echo -n "Enter second number: "
read b

fact1=$(factorial $a)
fact2=$(factorial $b)

echo "Factorial of $a is $fact1"
echo "Factorial of $b is $fact2"
echo "Sum = $((fact1 + fact2))"
```

• Write a Shell program to find total number of alphabets, digits or special characters in a string.

Sample Input:

Today is 12 November.

Output:

Alphabets = 15

Digits = 2

Special characters = 4

```
#!/bin/bash
```

```
echo -n "Enter a string: "
read -r str   # -r keeps special chars

alphabets=0
digits=0
special=0

for ((i=0; i<${#str}; i++))
do
    ch="${str:$i:1}"
    if [[ $ch =~ [a-zA-Z] ]]
    then
        alphabets=$((alphabets + 1))
    elif [[ $ch =~ [0-9] ]]
    then
        digits=$((digits + 1))
    else
        special=$((special + 1))
    fi
done

echo "Alphabets = $alphabets"
echo "Digits = $digits"
echo "Special characters = $special"
```

Lab number -05

Title: CPU Scheduling Algorithms to find

Turnaround Time and Waiting Time.

Lab Task

1. Implement Priority CPU Scheduling Algorithm.

```
#include <stdio.h>


struct Process {
    int pid;
    int bt;     // Burst Time
    int priority;
    int wt;     // Waiting Time
    int tat;    // Turn Around Time
};


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


    struct Process p[n];
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter Burst Time and Priority for P%d: ", i + 1);
        scanf("%d %d", &p[i].bt, &p[i].priority);
    }
```

```
// Sort processes based on priority (lower value = higher priority)
for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
        if (p[i].priority > p[j].priority) {
            struct Process temp = p[i];
            p[i] = p[j];
            p[j] = temp;
        }
    }
}

// Calculate Waiting Time and Turnaround Time
p[0].wt = 0;
p[0].tat = p[0].bt;
for (int i = 1; i < n; i++) {
    p[i].wt = p[i-1].wt + p[i-1].bt;
    p[i].tat = p[i].wt + p[i].bt;
}

// Display results
printf("\nPID\tBT\tPriority\tWT\tTAT\n");
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t\t%d\t%d\n", p[i].pid, p[i].bt, p[i].priority, p[i].wt, p[i].tat);
}
```

```
    // Average WT and TAT
    float total_wt = 0, total_tat = 0;
    for (int i = 0; i < n; i++) {
        total_wt += p[i].wt;
        total_tat += p[i].tat;
    }
    printf("\nAverage Waiting Time = %.2f\n", total_wt/n);
    printf("Average Turnaround Time = %.2f\n", total_tat/n);

    return 0;
}
```

2. Implement Round Robin CPU Scheduling Algorithm.

```
#include <stdio.h>

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

    int bt[n], rt[n], wt[n], tat[n];
    for (int i = 0; i < n; i++) {
        printf("Enter Burst Time for P%d: ", i + 1);
        scanf("%d", &bt[i]);
        rt[i] = bt[i]; // Remaining time
    }

    printf("Enter Time Quantum: ");
    scanf("%d", &tq);

    int time = 0, done;
    do {
        done = 1;
        for (int i = 0; i < n; i++) {
            if (rt[i] > 0) {
                done = 0;
                if (rt[i] > tq) {
```

```
                time += tq;
                rt[i] -= tq;
            } else {
                time += rt[i];
                wt[i] = time - bt[i];
                rt[i] = 0;
            }
        }
    }
} while (!done);

// Calculate Turnaround time
for (int i = 0; i < n; i++)
    tat[i] = bt[i] + wt[i];

// Display results
printf("\nPID\tBT\tWT\tTAT\n");
for (int i = 0; i < n; i++)
    printf("%d\t%d\t%d\t%d\n", i+1, bt[i], wt[i], tat[i]);

// Average WT and TAT
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);

return 0;
}
```

Lab Exercise

Implement the following problem using Scheduling Algorithms (Priority > SJF > FCFS).

Table 1: Sample Input

Process	Burst Time	Priority
P1	10	4
P2	13	1
P3	7	3
P4	15	2
P5	6	3
P6	10	4

Table 2: Sample Output

Process	Burst Time	Priority	Waiting

Time

Turnaround
Time
P2 13 1 0 13
P4 15 2 13 28
P5 6 3 28 34
P3 7 3 34 41
P1 10 4 41 51
P6 10 4 51 61

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

struct Process {
    char pid[5];
    int bt;
    int priority;
    int wt;
    int tat;
    int scheduled;  // 0 = not scheduled, 1 = scheduled
};

int main() {
    int n = 6;
    struct Process p[6] = {
        {"P1", 10, 4, 0, 0, 0},
        {"P2", 13, 1, 0, 0, 0},
        {"P3", 7, 3, 0, 0, 0},
        {"P4", 15, 2, 0, 0, 0},
        {"P5", 6, 3, 0, 0, 0},
        {"P6", 10, 4, 0, 0, 0}
    };

    int time = 0;

    for (int i = 0; i < n; i++) {
        int idx = -1;
        int highestPriority = 999;

        // Step 1: Find highest priority among unscheduled processes
        for (int j = 0; j < n; j++) {
            if (!p[j].scheduled && p[j].priority < highestPriority) {
                highestPriority = p[j].priority;
                idx = j;
```

```
            }
        }

        // Step 2: If multiple processes have same priority, pick shortest BT (SJF)
        int minBT = 999;
        for (int j = 0; j < n; j++) {
            if (!p[j].scheduled && p[j].priority == highestPriority) {
                if (p[j].bt < minBT) {
                    minBT = p[j].bt;
                    idx = j;
                }
            }
        }

        // Step 3: Schedule the selected process
        p[idx].wt = time;
        p[idx].tat = p[idx].wt + p[idx].bt;
        time += p[idx].bt;
        p[idx].scheduled = 1;
    }

    // Display output
    printf("Process\tBT\tPriority\tWT\tTAT\n");
    for (int i = 0; i < n; i++) {
        printf("%s\t%d\t%d\t\t%d\t%d\n",
               p[i].pid, p[i].bt, p[i].priority, p[i].wt, p[i].tat);
    }

    return 0;
}
```

Lab number=06

Lab Exercise

Implement a code to solve the Memory Management technique problem.

Enter the number of Blocks– 4
Block 1 size: 280
Block 2 size: 350
Block 3 size: 300
Block 4 size: 320
Enter the number of processes – 4
Enter memory required for process 1 – 275
Enter memory required for process 2 – 400
Enter memory required for process 3 – 290
Enter memory required for process 4 – 293
Output:

Table 1: **Sample Output**

Processes	Processes size	Blocks	Blocks size	Allocated	Int. Frag.
1	275	1	280	YES	5
2	400	2	350	NO	—
3	290	3	350	YES	60
4	293	4	300	YES	7

```
#include <stdio.h>

int main() {
    int nBlocks, nProcesses;

    printf("Enter the number of Blocks: ");
    scanf("%d", &nBlocks);

    int blockSize[nBlocks], blockUsed[nBlocks];
    for(int i = 0; i < nBlocks; i++) {
        printf("Block %d size: ", i + 1);
        scanf("%d", &blockSize[i]);
        blockUsed[i] = 0; // 0 means free
    }

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

    int processSize[nProcesses], allocation[nProcesses];
    for(int i = 0; i < nProcesses; i++) {
        printf("Enter memory required for process %d: ", i + 1);
        scanf("%d", &processSize[i]);
        allocation[i] = -1; // -1 means not allocated
    }
```

```
    // First Fit Allocation
    for(int i = 0; i < nProcesses; i++) {
        for(int j = 0; j < nBlocks; j++) {
            if(!blockUsed[j] && blockSize[j] >= processSize[i]) {
                allocation[i] = j;  // allocate process i to block j
                blockUsed[j] = 1;   // mark block as used
                break;
            }
        }
    }

    // Display results
    printf("\nProcess\tMemory Required\tBlock Allocated\n");
    for(int i = 0; i < nProcesses; i++) {
        if(allocation[i] != -1)
            printf("P%d\t%d\t\tB%d\n", i+1, processSize[i], allocation[i]+1);
        else
            printf("P%d\t%d\t\tNot Allocated\n", i+1, processSize[i]);
    }

    return 0;
}
```

Lab number=07

Lab Task

1. Implement best fit contiguous memory allocation algorithm.

```
Enter the number of blocks: 4
Enter the number of files: 3

Enter the size of the blocks:-
Block 1: 5
Block 2: 8
Block 3: 4
Block 4: 10
Enter the size of the files:-
File 1: 1
File 2: 4
File 3: 7
```

3 Output

tput of the program is given below.

File_no	File_size	Block_no	Block_size	Fragment
1	1	3	4	3
2	4	1	5	1
3	7	2	8	1

```
#include <stdio.h>

int main() {
    int nBlocks, nFiles;

    printf("Enter the number of blocks: ");
    scanf("%d", &nBlocks);
    printf("Enter the number of files: ");
    scanf("%d", &nFiles);
```

```
    int blockSize[nBlocks], blockUsed[nBlocks];
    for(int i = 0; i < nBlocks; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSize[i]);
        blockUsed[i] = 0; // 0 = free, 1 = used
    }

    int fileSize[nFiles];
    for(int i = 0; i < nFiles; i++) {
        printf("File %d: ", i + 1);
        scanf("%d", &fileSize[i]);
    }

    int allocation[nFiles], fragment[nFiles];

    for(int i = 0; i < nFiles; i++) {
        allocation[i] = -1;
        fragment[i] = 0;
        for(int j = 0; j < nBlocks; j++) {
            if(!blockUsed[j] && blockSize[j] >= fileSize[i]) {
                allocation[i] = j;              // allocate file i to block j
                fragment[i] = blockSize[j] - fileSize[i]; // calculate fragment
                blockUsed[j] = 1;               // mark block as used
                break;
            }
        }
    }

    // Display output
    printf("\nFile_no\tFile_size\tBlock_no\tBlock_size\tFragment\n");
    for(int i = 0; i < nFiles; i++) {
        if(allocation[i] != -1)
            printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
                  i+1, fileSize[i], allocation[i]+1, blockSize[allocation[i]], fragment[i]);
        else
            printf("%d\t%d\t\tNot Allocated\t-\t-\n", i+1, fileSize[i]);
    }

    return 0;
}
```

Lab Exercise

Implement first fit contiguous memory allocation algorithm.

Input of the program is given below.

```
Enter the number of blocks: 4
Enter the number of files: 3

Enter the size of the blocks:-
Block 1: 5
Block 2: 8
Block 3: 4
Block 4: 10
Enter the size of the files:-
File 1: 1
File 2: 4
File 3: 7
```

7.2 Output

Output of the program is given below.

File_no:	File_size :	Block_no:	Block_size:	Fragment
1	1	1	5	4
2	4	2	8	4
3	7	4	10	3

```
#include <stdio.h>

int main() {
    int nBlocks, nFiles;

    printf("Enter the number of blocks: ");
    scanf("%d", &nBlocks);
    printf("Enter the number of files: ");
    scanf("%d", &nFiles);

    int blockSize[nBlocks], blockUsed[nBlocks];
    for (int i = 0; i < nBlocks; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSize[i]);
        blockUsed[i] = 0; // 0 = free, 1 = used
    }

    int fileSize[nFiles];
    for (int i = 0; i < nFiles; i++) {
        printf("File %d: ", i + 1);
        scanf("%d", &fileSize[i]);
    }

    int allocation[nFiles], fragment[nFiles];

    for (int i = 0; i < nFiles; i++) {
        int bestIdx = -1;
        int minFragment = 9999;

        for (int j = 0; j < nBlocks; j++) {
```

```
            if (!blockUsed[j] && blockSize[j] >= fileSize[i]) {
                int tempFragment = blockSize[j] - fileSize[i];
                if (tempFragment < minFragment) {
                    minFragment = tempFragment;
                    bestIdx = j;
                }
            }
        }

        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            fragment[i] = blockSize[bestIdx] - fileSize[i];
            blockUsed[bestIdx] = 1;
        } else {
            allocation[i] = -1; // Not allocated
            fragment[i] = 0;
        }
    }

    // Display output
    printf("\nFile_no: File_size: Block_no: Block_size: Fragment\n");
    for (int i = 0; i < nFiles; i++) {
        if (allocation[i] != -1)
            printf("%d %d %d %d %d\n",
                  i+1, fileSize[i], allocation[i]+1, blockSize[allocation[i]], fragment[i]);
        else
            printf("%d %d Not Allocated - -\n", i+1, fileSize[i]);
    }

    return 0;
}
```

Lab number=08

Page Replacement Algorithms

Implems----FIFO

```
#include <stdio.h>


int main() {
   int pageFaultCount = 0;
   int pages[50], memory[20];
   int memoryIndex = 0;
   int numberOfPages, numberOfFrames;
   int i, j, k;

   printf("Enter number of pages: ");
   scanf("%d", &numberOfPages);

   printf("Enter the pages: ");
   for(i = 0; i < numberOfPages; i++) {
      scanf("%d", &pages[i]);
   }

   printf("Enter number of frames: ");
   scanf("%d", &numberOfFrames);

   // Initialize memory frames
   for(i = 0; i < numberOfFrames; i++) {
```

```
    memory[i] = -1;
}

printf("The Page Replacement Process is -->\n");

for(i = 0; i < numberOfPages; i++) {
    // Check if page already exists in memory
    for(j = 0; j < numberOfFrames; j++) {
        if(memory[j] == pages[i]) {
            break;
        }
    }
    if(j == numberOfFrames) {
        memory[memoryIndex] = pages[i];
        memoryIndex = (memoryIndex + 1) % numberOfFrames; // FIFO circular replacement
        pageFaultCount++;
    }
    printf("After accessing page %d: ", pages[i]);
    for(k = 0; k < numberOfFrames; k++) {
        if(memory[k] != -1)
            printf("%d\t", memory[k]);
        else
            printf("-\t");
    }
    if(j == numberOfFrames)
```

```
            printf("\tPage Fault No: %d", pageFaultCount);
        printf("\n");
    }
    printf("\nThe number of Page Faults using FIFO is: %d\n", pageFaultCount);
    return 0;
}
```

1. Implement LRU page replacement algorithm.

```
#include <stdio.h>

int main() {
    int frames, pages;

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

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

    int reference[pages];
    printf("Enter reference string: ");
    for(int i = 0; i < pages; i++) {
        scanf("%d", &reference[i]);
    }

    int memory[frames];
```

```
int lastUsed[frames]; // Stores last usage index
int pageFaults = 0;

for(int i = 0; i < frames; i++) {
   memory[i] = -1;
   lastUsed[i] = -1;
}

printf("The Page Replacement Process is -->\n");

for(int i = 0; i < pages; i++) {
   int page = reference[i];
   int hit = 0;

   // Check if page already exists in memory
   for(int j = 0; j < frames; j++) {
      if(memory[j] == page) {
         hit = 1;
         lastUsed[j] = i; // update last used index
         break;
      }
   }

   if(!hit) {
      // Page fault occurs
```

```
pageFaults++;

// Find empty frame
int replaced = -1;
for(int j = 0; j < frames; j++) {
    if(memory[j] == -1) {
        memory[j] = page;
        lastUsed[j] = i;
        replaced = j;
        break;
    }
}

// If no empty frame, replace least recently used
if(replaced == -1) {
    int lruIndex = 0;
    int minUsed = lastUsed[0];
    for(int j = 1; j < frames; j++) {
        if(lastUsed[j] < minUsed) {
            minUsed = lastUsed[j];
            lruIndex = j;
        }
    }
    memory[lruIndex] = page;
    lastUsed[lruIndex] = i;
```

```
            }
        }

        // Print memory state after this page
        printf("For %d : ", page);
        if(hit) {
            printf("No page fault!");
        } else {
            for(int j = 0; j < frames; j++) {
                if(memory[j] != -1)
                    printf("%d ", memory[j]);
            }
        }
        printf("\n");
    }

    printf("Total no of page faults using LRU is: %d\n", pageFaults);
    return 0;
}
```

• Implement LFU page replacement algorithm.

```
#include <stdio.h>

int main() {
    int n, pages[50], frames[10], freq[10], time[10], i, j, k;
    int pageFaults = 0, m, min, minIndex;
```

```
printf("Enter number of frames: ");
scanf("%d", &n);
printf("Enter number of pages: ");
scanf("%d", &m);
printf("Enter reference string: ");
for(i = 0; i < m; i++)
   scanf("%d", &pages[i]);

// Initialize frames and frequency
for(i = 0; i < n; i++) {
   frames[i] = -1;
   freq[i] = 0;
   time[i] = 0;
}

printf("The Page Replacement Process is ->\n");
for(i = 0; i < m; i++) {
   int found = 0;

   // Check if page is already in frame
   for(j = 0; j < n; j++) {
      if(frames[j] == pages[i]) {
         found = 1;
         freq[j]++; // Increment frequency
```

```
            break;
        }
    }

    if(found) {
        printf("For %d : No page fault!\n", pages[i]);
    } else {
        pageFaults++;

        // Check if empty frame exists
        int placed = 0;
        for(j = 0; j < n; j++) {
            if(frames[j] == -1) {
                frames[j] = pages[i];
                freq[j] = 1;
                time[j] = i;
                placed = 1;
                break;
            }
        }

        // If no empty frame, replace LFU
        if(!placed) {
            min = freq[0];
            minIndex = 0;
```

```
            for(j = 1; j < n; j++) {
                if(freq[j] < min || (freq[j] == min && time[j] < time[minIndex])) {
                    min = freq[j];
                    minIndex = j;
                }
            }
            frames[minIndex] = pages[i];
            freq[minIndex] = 1;
            time[minIndex] = i;
        }

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

   printf("Total no of page faults using LFU is: %d\n", pageFaults);

   return 0;
}
```