EXP - 1

Aim: Write a program to implement CPU scheduling for first come first serve.

Theory:

CPU Scheduling – First Come First Serve (FCFS)

CPU Scheduling is the process of selecting which process in the ready queue will be executed next by the CPU. It is a key concept in operating system design for achieving efficient process management and optimal CPU utilization.

First Come First Serve (FCFS) Scheduling:
FCFS is a non-preemptive scheduling algorithm based on the principle that the process which arrives first in the ready queue is executed first. It follows the First In, First Out (FIFO) approach, similar to a real-life queue system. Once a process starts its execution, it cannot be interrupted until it completes.

Important Terms:

1. Arrival Time (AT): The time at which a process enters the ready queue.
2. Burst Time (BT): The total time required by the process on the CPU.
3. Completion Time (CT): The time at which the process finishes execution.
4. Turnaround Time (TAT): It is calculated as:

$$TAT = CT - AT$$

It represents the total time taken by a process from its arrival to its completion.

5. Waiting Time (WT): It is calculated as:

$$WT = TAT - BT$$

It indicates the total time a process spends waiting in the ready queue.

Steps in FCFS Scheduling:

1. Sort the processes based on their arrival time.
2. Compute the completion time for each process sequentially.
3. Calculate the Turnaround Time (TAT) and Waiting Time (WT) for each process.
4. Display the scheduling results along with averages.

Advantages of FCFS:

- Simple and easy to understand.
- Fair for processes as it schedules in the order of arrival.

Disadvantages of FCFS:

- It may lead to poor average waiting time if a process with a long burst time is at the front of the queue (convoy effect).
- It is not suitable for time-sharing or interactive systems.

Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
#include <queue>

using namespace std;

struct Process {
    int pid;
    int arrival;
    int burst;
    int start;
    int finish;
    int waiting;
    int turnaround;
    int response;
};

bool compareProcesses(const Process& a, const Process& b) {
    if (a.arrival != b.arrival) {
        return a.arrival < b.arrival;
    }
    return a.pid < b.pid;
}

string getWaitingQueue(const vector<Process>& processes, int currentTime, int n, int
currentProcessIndex) {
    string queueStr = "Waiting Queue at time " + to_string(currentTime) + ": [";
    bool first = true;
    for (int i = 0; i < n; ++i) {
        if (i >= currentProcessIndex && processes[i].arrival <= currentTime) {
            if (!first) {
                queueStr += ", ";
            }
            queueStr += "P" + to_string(processes[i].pid);
            first = false;
        }
    }
    queueStr += "]";
    return queueStr;
}

int main() {
    int n;
    cout << "Enter number of processes: ";
```

```
cin >> n;

vector<Process> processes(n);
for (int i = 0; i < n; ++i) {
    processes[i].pid = i + 1;
    cout << "Enter arrival time for process " << processes[i].pid << ": ";
    cin >> processes[i].arrival;
    cout << "Enter burst time for process " << processes[i].pid << ": ";
    cin >> processes[i].burst;
}

sort(processes.begin(), processes.end(), compareProcesses);

int currentTime = 0;
int totalWaiting = 0;
int totalTurnaround = 0;
int totalResponse = 0;
vector<string> waitingQueueHistory;

cout << endl << setw(5) << "PID" << setw(10) << "Arrival" << setw(8) << "Burst"
         << setw(8) << "Start" << setw(8) << "Finish" << setw(10) << "Waiting"
         << setw(12) << "Turnaround" << setw(10) << "Response" << endl;

string ganttChartLine;
string ganttChartPids;
string ganttChartTimes;

for (int i = 0; i < n; ++i) {
    if (currentTime < processes[i].arrival) {
        ganttChartLine += "-------";
        ganttChartPids += "  Idle ";
        ganttChartTimes += to_string(currentTime) + "      ";
        currentTime = processes[i].arrival;
    }

    waitingQueueHistory.push_back(getWaitingQueue(processes, currentTime, n, i));

    processes[i].start = currentTime;
    processes[i].response = processes[i].start - processes[i].arrival;
    processes[i].waiting = processes[i].start - processes[i].arrival;
    processes[i].finish = processes[i].start + processes[i].burst;
    processes[i].turnaround = processes[i].finish - processes[i].arrival;

    totalWaiting += processes[i].waiting;
    totalTurnaround += processes[i].turnaround;
    totalResponse += processes[i].response;

    currentTime = processes[i].finish;
```

```
            string pidStr = " P" + to_string(processes[i].pid) + "  ";
            int segmentWidth = processes[i].burst * 2 + 1;
            ganttChartLine += "|" + string(segmentWidth, '-');
            ganttChartPids += "|" + pidStr + string(segmentWidth - pidStr.length(), ' ');
            ganttChartTimes += to_string(processes[i].start) + string(processes[i].burst * 2, ' ');
        }

        ganttChartLine += "|";
        ganttChartPids += "|";
        ganttChartTimes += to_string(currentTime);

        for (const auto& p : processes) {
            cout << setw(5) << "P" << p.pid << setw(10) << p.arrival << setw(8) << p.burst
                 << setw(8) << p.start << setw(8) << p.finish << setw(10) << p.waiting
                 << setw(12) << p.turnaround << setw(10) << p.response << endl;
        }

        cout << endl << "Average Waiting Time: " << static_cast<double>(totalWaiting) / n << endl;
        cout << "Average Turnaround Time: " << static_cast<double>(totalTurnaround) / n << endl;
        cout << "Average Response Time: " << static_cast<double>(totalResponse) / n << endl;

        cout << endl << "Gantt Chart:" << endl;
        cout << ganttChartLine << endl;
        cout << ganttChartPids << endl;
        cout << ganttChartLine << endl;
        cout << ganttChartTimes << endl;

        cout << "\nWaiting Queue History:" << endl;
        for (const auto& history : waitingQueueHistory) {
            cout << history << endl;
        }

        return 0;
    }
```

Output:

```
Enter number of processes: 5
Enter arrival time for process 1: 3
Enter burst time for process 1: 4
Enter arrival time for process 2: 1
Enter burst time for process 2: 6
Enter arrival time for process 3: 7
Enter burst time for process 3: 6
Enter arrival time for process 4: 4
Enter burst time for process 4: 2
Enter arrival time for process 5: 3
Enter burst time for process 5: 5

  PID   Arrival   Burst   Start  Finish   Waiting  Turnaround  Response
   P2         1       6       1       7         0           6         0
   P1         3       4       7      11         4           8         4
   P5         3       5      11      16         8          13         8
   P4         4       2      16      18        12          14        12
   P3         7       6      18      24        11          17        11

Average Waiting Time: 7
Average Turnaround Time: 11.6
Average Response Time: 7

Gantt Chart:
-------|-------------|---------|-----------|-----|-------------|
  Idle | P2          | P1      | P5        | P4  | P3          |
-------|-------------|---------|-----------|-----|-------------|
0      1             7         11          16    18            24

Waiting Queue History:
Waiting Queue at time 1: [P2]
Waiting Queue at time 7: [P1, P5, P4, P3]
Waiting Queue at time 11: [P5, P4, P3]
Waiting Queue at time 16: [P4, P3]
Waiting Queue at time 18: [P3]
```

EXP – 2

AIM: Write a program to implement CPU scheduling for shortest job first.

THEORY

Shortest Job First (SJF) is a CPU scheduling algorithm that selects the process with the smallest burst time for execution next.

Types:

- Non-preemptive SJF – Process runs till completion.
- Preemptive SJF (Shortest Remaining Time First – SRTF) – Switches if a new process has shorter burst time.

Key Terms:

– Burst Time: Time required for process execution.

– Waiting Time (WT): Time a process waits in ready queue.

– Turnaround Time (TAT): Time from arrival to completion (TAT = WT + Burst Time).

Advantages:

– Minimizes average waiting and turnaround times.

– Efficient for batch processing.

Limitations:

– Difficult to predict burst times.

– Risk of starvation for longer processes.

CODE

```
#!/bin/bash
echo "Enter number of processes: "
read n
burst=()
pid=()
wt=()
tat=()

echo "Enter burst times:"
for ((i=0; i<n; i++))
do
```

```
  echo -n "Process $((i+1)) burst time: "
  read bt
  burst[$i]=$bt
  pid[$i]=$((i+1))
done
for ((i=0; i<n; i++))
do
  for ((j=i+1; j<n; j++))
  do
    if [ ${burst[$i]} -gt ${burst[$j]} ]
    then
      # swap burst
      temp=${burst[$i]}
      burst[$i]=${burst[$j]}
      burst[$j]=$temp

      temp=${pid[$i]}
      pid[$i]=${pid[$j]}
      pid[$j]=$temp
    fi
  done
done

wt[0]=0
for ((i=1; i<n; i++))
do
  wt[$i]=$((wt[$i-1] + burst[$i-1]))
done

for ((i=0; i<n; i++))
do
  tat[$i]=$((wt[$i] + burst[$i]))
done
echo
```

```
echo "Process  Burst_Time  Waiting_Time  Turnaround_Time"
for ((i=0; i<n; i++))
do
  echo "   ${pid[$i]}         ${burst[$i]}          ${wt[$i]}           ${tat[$i]}"
done

total_wt=0
total_tat=0
for ((i=0; i<n; i++))
do
  total_wt=$((total_wt + wt[$i]))
  total_tat=$((total_tat + tat[$i]))
done

avg_wt=$(echo "scale=2; $total_wt / $n" | bc)
avg_tat=$(echo "scale=2; $total_tat / $n" | bc)

echo
echo "Average Waiting Time: $avg_wt"
echo "Average Turnaround Time: $avg_tat"
```

OUTPUT

```
Enter number of processes:
3
Enter burst times:
Process 1 burst time: 5
Process 2 burst time: 2
Process 3 burst time: 8

Process  Burst_Time  Waiting_Time  Turnaround_Time
   2          2            0              2
   1          5            2              7
   3          8            7              15

Average Waiting Time: 3.00
Average Turnaround Time: 8.00
```

Experiment-3

Aim- Write a program to perform priority scheduling.

Theory:-

Priority Scheduling is a CPU scheduling algorithm in which each process is assigned a priority, and the CPU is allocated to the process with the highest priority. If two processes have the same priority, then scheduling is done according to their arrival order (non-preemptive) or by preemption if the algorithm is preemptive. This method ensures that important tasks (with higher priority values) are executed first, which is useful in real-time systems and environments where certain jobs are more critical than others. However, one drawback of priority scheduling is **starvation**, where low-priority processes may get delayed indefinitely if higher-priority processes keep arriving. To overcome this, a technique called **aging** is used, where the priority of waiting processes is gradually increased to ensure fairness.

Types:-

- **Non-preemptive Priority Scheduling**:
  Once a process starts execution, it cannot be stopped until it finishes, even if a higher-priority process arrives.
- **Preemptive Priority Scheduling**:
  If a new process arrives with higher priority than the currently running process, the CPU is preempted and given to the higher-priority process.

Code-

```
#!/bin/bash

echo -n "Enter number of processes: " read n

for ((i=0; i<n; i++)) do echo -n "Enter Burst Time for Process[$i]: " read bt[$i]

echo -n "Enter Priority for Process[$i]: "
read pr[$i]

pid[$i]=$((i+1))

for ((i=0; i<n; i++)) do for ((j=i+1; j<n; j++)) do if [ ${pr[$i]} -gt ${pr[$j]} ] then # swap priority
temp=${pr[$i]} pr[$i]=${pr[$j]} pr[$j]=$temp

    # swap burst time
      temp=${bt[$i]}
      bt[$i]=${bt[$j]}
      bt[$j]=$temp
      temp=${pid[$i]}
      pid[$i]=${pid[$j]}
```

```
        pid[$j]=$temp
    fi
done
```

wt[0]=0 tat[0]=${bt[0]} for ((i=1; i<n; i++)) do wt[$i]=$((wt[$i-1] + bt[$i-1])) tat[$i]=$((wt[$i] + bt[$i])) done

echo -e "\nProcess\tPriority\tBurst Time\tWaiting Time\tTurnaround Time" total_wt=0 total_tat=0 for ((i=0; i<n; i++)) do echo -e "P${pid[$i]}\t${pr[$i]}\t\t${bt[$i]}\t\t${wt[$i]}\t\t${tat[$i]}" total_wt=$((total_wt + wt[$i])) total_tat=$((total_tat + tat[$i])) done

echo -e "\nAverage Waiting Time: $(echo "scale=2; $total_wt/$n" | bc)" echo -e "Average Turnaround Time: $(echo "scale=2; $total_tat/$n" | bc)"

Output-

```
Enter number of processes: 3
Enter Burst Time for Process[0]: 5
Enter Priority for Process[0]: 2
Enter Burst Time for Process[1]: 3
Enter Priority for Process[1]: 1
Enter Burst Time for Process[2]: 8
Enter Priority for Process[2]: 3
```

```
Process Priority Burst Time Waiting Time Turnaround Time
P2      1        3          0            3
P1      2        5          3            8
P3      3        8          8            16

Average Waiting Time: 3.67
Average Turnaround Time: 9.00
```

Experiment-4

Aim: Write a program to implement CPU scheduling for Round Robin.

Theory:

Round Robin (RR) is a **preemptive CPU scheduling algorithm** mainly used in **time-sharing systems**. In this method, each process is assigned a **fixed time slice**, called a **Time Quantum (TQ)**.

The CPU executes each process for **at most one time quantum** in cyclic order.
If a process finishes before its time quantum ends, the CPU moves to the next process.
If it still has burst time left, it is **placed at the end of the ready queue** and will get the CPU again in the next round.

Working Steps

1. All processes are arranged in the **ready queue** based on **arrival time**.
2. The CPU gives **TQ units** of time to the first process.
    - If the process completes, it is removed.
    - If not, it is added back to the **end of the queue** with remaining burst time.
3. This cycle continues until **all processes finish execution**.

Advantages

- Fair to all processes
- Simple to implement
- Suitable for **time-sharing systems**

Disadvantages

- Performance depends on **time quantum**
- Too small TQ → many context switches

- Too large TQ → behaves like **FCFS**

Code:

```
#!/bin/bash

# Round Robin CPU Scheduling Algorithm


echo "Enter number of processes:"

read n


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

do

  echo "Enter Burst Time for Process P$i:"

  read bt[$i]

  echo "Enter Arrival Time for Process P$i:"

  read at[$i]

  pid[$i]=$i

  rem_bt[$i]=${bt[$i]}

done


echo "Enter Time Quantum:"

read tq

time=0

remain=$n

completed=0


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

```
do
  tat[$i]=0
  wt[$i]=0
  done

echo -e "\nGantt Chart:"
echo -n "|"

while [ $completed -ne $n ]
do
  doneInCycle=false
  for ((i=0; i<n; i++))
  do
    if [ ${rem_bt[$i]} -gt 0 ] && [ ${at[$i]} -le $time ]
    then
      doneInCycle=true
      if [ ${rem_bt[$i]} -le $tq ]
      then
        time=$((time + ${rem_bt[$i]}))
        rem_bt[$i]=0
        tat[$i]=$((time - ${at[$i]}))
        wt[$i]=$((tat[$i] - ${bt[$i]}))
        completed=$((completed + 1))
        echo -n " P${pid[$i]}($time) |"
      else
        rem_bt[$i]=$((rem_bt[$i] - tq))
```

```
      time=$((time + tq))
      echo -n " P${pid[$i]}($time) |"
    fi
  fi
 done

 if ! $doneInCycle; then
  ((time++))
 fi
done

# Display results
echo -e "\n\nProcess\tAT\tBT\tTAT\tWT"
total_tat=0
total_wt=0
for ((i=0; i<n; i++))
do
 echo "P${pid[$i]}\t${at[$i]}\t${bt[$i]}\t${tat[$i]}\t${wt[$i]}"
 total_tat=$((total_tat + tat[$i]))
 total_wt=$((total_wt + wt[$i]))
done

avg_tat=$(echo "scale=2; $total_tat / $n" | bc)
avg_wt=$(echo "scale=2; $total_wt / $n" | bc)

echo -e "\nAverage Turnaround Time: $avg_tat"
```

```
echo "Average Waiting Time: $avg_wt"
```

Output:

```
Enter number of processes:
3
Enter Burst Time for Process P0:
5
Enter Arrival Time for Process P0:
0
Enter Burst Time for Process P1:
4
Enter Arrival Time for Process P1:
1
Enter Burst Time for Process P2:
2
Enter Arrival Time for Process P2:
2
Enter Time Quantum:
2
```

```
|  P0  |  P1  |  P2  |  P0  |  P1  |  P0  |
0      2      4      6      8     10     11
```

```
Process AT  BT  TAT WT
P0  0   5   11  6
P1  1   4   9   5
P2  2   2   4   2

Average Turnaround Time: 8.00
Average Waiting Time: 4.33
```

Experiment-5

Aim: Write a program for page replacement policy using a) LRU b) FIFO c) Optimal.

Theory:

Introduction

When a program tries to access a page not present in main memory, a page fault occurs.
The operating system must replace one of the pages in memory with the new page.
Page Replacement Algorithms decide which page to remove to minimize page faults.

Types of Page Replacement Algorithms

1. FIFO (First-In First-Out)

- The oldest loaded page in memory is replaced first.
- It uses a queue structure.
- Simple but may lead to Belady's anomaly (more frames → more faults).

2. LRU (Least Recently Used)

- Replaces the page that was least recently used.
- Uses past usage history to make decisions.
- More efficient than FIFO but slightly slower to implement.

3. Optimal Page Replacement

- Replaces the page that will not be used for the longest period in the future.
- Gives minimum possible page faults, but cannot be implemented in real systems (future knowledge needed).
- Used mainly for theoretical comparison.

Performance Metrics

- Page Fault: When required page is not in memory.

- Hit: When page is already in memory.
- Page Fault Rate = (No. of Page Faults / Total References)

Code:

```
#!/bin/bash

# ========================================

# Page Replacement Algorithms: FIFO, LRU, OPTIMAL

# ========================================

echo "Enter number of frames:"

read frames

echo "Enter number of pages:"

read pages

echo "Enter reference string (space-separated):"

read -a ref

# ---------- FIFO ----------

fifo_faults=0

declare -a fifo

echo -e "\nFIFO Page Replacement:"

for ((i=0; i<pages; i++))

do

  page=${ref[$i]}

  present=false

  for f in "${fifo[@]}"; do
```

```
    if [ "$f" == "$page" ]; then
      present=true
      break
    fi
  done
  if ! $present; then
    if [ ${#fifo[@]} -lt $frames ]; then
      fifo+=($page)
    else
      fifo=("${fifo[@]:1}" "$page")
    fi
    ((fifo_faults++))
  fi
  echo "After reference ${page}: ${fifo[@]}"
done
echo "Total FIFO Page Faults: $fifo_faults"

# ---------- LRU ----------
lru_faults=0
declare -a lru
declare -A last_used

echo -e "\nLRU Page Replacement:"
for ((i=0; i<pages; i++))
do
  page=${ref[$i]}
  present=false
```

```
for f in "${lru[@]}"; do
  if [ "$f" == "$page" ]; then
    present=true
    break
  fi
done
if ! $present; then
  if [ ${#lru[@]} -lt $frames ]; then
    lru+=($page)
  else
    # Find least recently used page
    min_time=999
    victim=0
    for ((j=0; j<frames; j++)); do
      t=${last_used[${lru[$j]}]}
      if [ "$t" -lt "$min_time" ]; then
        min_time=$t
        victim=$j
      fi
    done
    lru[$victim]=$page
  fi
  ((lru_faults++))
fi
last_used[$page]=$i
echo "After reference ${page}: ${lru[@]}"
```

```
done
echo "Total LRU Page Faults: $lru_faults"

# ---------- OPTIMAL ----------
opt_faults=0
declare -a opt

echo -e "\nOptimal Page Replacement:"
for ((i=0; i<pages; i++))
do
  page=${ref[$i]}
  present=false
  for f in "${opt[@]}"; do
    if [ "$f" == "$page" ]; then
      present=true
      break
    fi
  done
  if ! $present; then
    if [ ${#opt[@]} -lt $frames ]; then
      opt+=($page)
    else
      farthest=-1
      victim=0
      for ((j=0; j<frames; j++))
      do
```

```
        next_use=-1
        for ((k=i+1; k<pages; k++))
        do
          if [ "${opt[$j]}" -eq "${ref[$k]}" ]; then
            next_use=$k
            break
          fi
        done
        if [ $next_use -eq -1 ]; then
          victim=$j
          break
        elif [ $next_use -gt $farthest ]; then
          farthest=$next_use
          victim=$j
        fi
      done
      opt[$victim]=$page
    fi
    ((opt_faults++))
  fi
  echo "After reference ${page}: ${opt[@]}"
done
echo "Total Optimal Page Faults: $opt_faults"
```

Output:

```
FIFO Page Replacement:
After reference 1: 1
After reference 3: 1 3
After reference 0: 1 3 0
After reference 3: 1 3 0
After reference 5: 3 0 5
After reference 6: 0 5 6
After reference 3: 5 6 3
After reference 1: 6 3 1
After reference 3: 6 3 1
After reference 6: 6 3 1
After reference 1: 6 3 1
After reference 2: 3 1 2
Total FIFO Page Faults: 8
```

```
LRU Page Replacement:
After reference 1: 1
After reference 3: 1 3
After reference 0: 1 3 0
After reference 3: 1 3 0
After reference 5: 3 0 5
After reference 6: 0 5 6
After reference 3: 5 6 3
After reference 1: 6 3 1
After reference 3: 6 3 1
After reference 6: 6 3 1
After reference 1: 6 3 1
After reference 2: 3 1 2
Total LRU Page Faults: 8
```

```
Optimal Page Replacement:
After reference 1: 1
After reference 3: 1 3
After reference 0: 1 3 0
After reference 3: 1 3 0
After reference 5: 1 3 5
After reference 6: 3 5 6
After reference 3: 3 5 6
After reference 1: 3 6 1
After reference 3: 3 6 1
After reference 6: 3 6 1
After reference 1: 3 6 1
After reference 2: 3 1 2
Total Optimal Page Faults: 7
```

Experiment-6

Aim: Write a program to implement first fit,best fit and worst fit algorithm for memory management.

Theory:

Introduction

In Memory Management, the main memory is divided into partitions (blocks) of fixed or variable sizes. When a process arrives, the operating system must decide where to place it in memory.

To reduce fragmentation and improve efficiency, different allocation strategies are used.

1. First Fit

- The allocator scans memory from the beginning.
- Allocates the first block large enough to fit the process.
- Fastest method but may leave small unused holes (external fragmentation).

2. Best Fit

- Searches entire memory and finds the smallest block that can fit the process.
- Reduces wastage of memory but may increase search time.
- May cause more fragmentation due to tiny leftover spaces.

3. Worst Fit

- Allocates the largest available block to the process.
- Leaves larger leftover parts → less fragmentation initially.
- However, may waste large chunks of memory.

Comparison Table

Algorithm	Search Type	Fragmentation	Speed

First Fit	Sequential	Moderate	Fast
Best Fit	Full Scan	High	Slow
Worst Fit	Full Scan	Low initially	Slow

Terminology

- Block Size: Memory partition size
- Process Size: Memory required by each process
- Fragment: Unused memory inside a block after allocation

Code:

```
#!/bin/bash
# ================================================
# Memory Management Algorithms:
# First Fit, Best Fit, and Worst Fit
# ================================================

echo "Enter number of memory blocks:"
read nb
echo "Enter size of each memory block:"
for ((i=0;i<nb;i++))
do
  read block[$i]
done
```

```
echo "Enter number of processes:"
read np
echo "Enter size of each process:"
for ((i=0;i<np;i++))
do
  read process[$i]
done

# ---------- FIRST FIT ----------
for ((i=0;i<nb;i++)); do allocF[$i]=-1; done
btemp=("${block[@]}")
for ((i=0;i<np;i++))
do
  for ((j=0;j<nb;j++))
  do
    if [ ${btemp[$j]} -ge ${process[$i]} ]
    then
      allocF[$i]=$j
      btemp[$j]=$((btemp[$j]-process[$i]))
      break
    fi
  done
done

# ---------- BEST FIT ----------
for ((i=0;i<nb;i++)); do btemp[$i]=${block[$i]}; done
```

```
for ((i=0;i<np;i++))
do
  best=-1
  for ((j=0;j<nb;j++))
  do
    if [ ${btemp[$j]} -ge ${process[$i]} ]
    then
      if [ $best -eq -1 ] || [ ${btemp[$j]} -lt ${btemp[$best]} ]; then
        best=$j
      fi
    fi
  done
  allocB[$i]=$best
  if [ $best -ne -1 ]; then btemp[$best]=$((btemp[$best]-process[$i])); fi
done

# ---------- WORST FIT ----------
for ((i=0;i<nb;i++)); do btemp[$i]=${block[$i]}; done
for ((i=0;i<np;i++))
do
  worst=-1
  for ((j=0;j<nb;j++))
  do
    if [ ${btemp[$j]} -ge ${process[$i]} ]
    then
      if [ $worst -eq -1 ] || [ ${btemp[$j]} -gt ${btemp[$worst]} ]; then
```

```
      worst=$j
    fi
  fi
 done
 allocW[$i]=$worst
 if [ $worst -ne -1 ]; then btemp[$worst]=$((btemp[$worst]-process[$i])); fi
done

# ---------- OUTPUT ----------
echo -e "\n==================== OUTPUT ===================="
echo "Block sizes: ${block[@]}"
echo "Process sizes: ${process[@]}"
echo "------------------------------------------------"

echo -e "\nFIRST FIT ALLOCATION:"
for ((i=0;i<np;i++))
do
 if [ ${allocF[$i]} -ne -1 ]; then
  echo "Process $i (${process[$i]}) -> Block ${allocF[$i]}"
 else
  echo "Process $i (${process[$i]}) -> Not Allocated"
 fi
done

echo -e "\nBEST FIT ALLOCATION:"
for ((i=0;i<np;i++))
```

```
do
  if [ ${allocB[$i]} -ne -1 ]; then
    echo "Process $i (${process[$i]}) -> Block ${allocB[$i]}"
  else
    echo "Process $i (${process[$i]}) -> Not Allocated"
  fi
done

echo -e "\nWORST FIT ALLOCATION:"
for ((i=0;i<np;i++))
do
  if [ ${allocW[$i]} -ne -1 ]; then
    echo "Process $i (${process[$i]}) -> Block ${allocW[$i]}"
  else
    echo "Process $i (${process[$i]}) -> Not Allocated"
  fi
done

echo -e "\n==================== CONCLUSION ===================="
echo "First Fit: Fastest but can cause external fragmentation."
echo "Best Fit: Minimizes leftover space but slower search."
echo "Worst Fit: Opposite of Best Fit; may lead to large wastage."
echo "Hence, choice depends on desired trade-off between speed and memory utilization."
echo "=================================================="
```

Output:

```
Enter number of memory blocks:
5
Enter size of each memory block:
100
500
200
300
600
Enter number of processes:
4
Enter size of each process:
212
417
112
426
```

```
==================== OUTPUT ====================
Block sizes: 100 500 200 300 600
Process sizes: 212 417 112 426
------------------------------------------------

FIRST FIT ALLOCATION:
Process 0 (212) -> Block 1
Process 1 (417) -> Block 4
Process 2 (112) -> Block 1
Process 3 (426) -> Not Allocated

BEST FIT ALLOCATION:
Process 0 (212) -> Block 3
Process 1 (417) -> Block 4
Process 2 (112) -> Block 2
Process 3 (426) -> Block 1

WORST FIT ALLOCATION:
Process 0 (212) -> Block 4
Process 1 (417) -> Block 1
Process 2 (112) -> Block 4
Process 3 (426) -> Block 3

================== CONCLUSION ==================
First Fit: Fastest but can cause external fragmentation.
Best Fit: Minimizes leftover space but slower search.
Worst Fit: Opposite of Best Fit; may lead to large wastage.
Hence, choice depends on desired trade-off between speed and memory utilization.
================================================
```

Experiment-7

Aim: Write a program to implement reader/writer problem using semaphore.

Theory:

The Reader–Writer Problem deals with synchronization between multiple reader and writer processes accessing a shared resource (like a file or database).

Key Rules:

1. Multiple readers can read the data simultaneously.
2. Only one writer can write at a time.
3. When a writer is writing, no reader or other writer should access the resource.

To solve this, we use semaphores:

- **mutex:** protects the reader count variable.
- **wrt:** ensures exclusive access to writers.
- **readcount:** counts the number of active readers.

Code:

```
#!/bin/bash

# ================================================

# Reader Writer Problem Simulation using Semaphores

# ================================================


mutex=1     # Semaphore for mutual exclusion

wrt=1       # Semaphore for writer

readcount=0 # Number of active readers
```

```
wait() {
  while [ $1 -le 0 ]; do
    : # busy wait
  done
  eval "$2=$((${!2} - 1))"
}

signal() {
  eval "$1=$((${!1} + 1))"
}

reader_enter() {
  wait $mutex mutex
  readcount=$((readcount + 1))
  if [ $readcount -eq 1 ]; then
    wait $wrt wrt
  fi
  signal mutex
  echo "Reader $1 is READING (Active Readers: $readcount)"
}

reader_exit() {
  wait $mutex mutex
  readcount=$((readcount - 1))
  if [ $readcount -eq 0 ]; then
    signal wrt
```

```
  fi
  signal mutex
  echo "Reader $1 has FINISHED (Remaining Readers: $readcount)"
}

writer_enter() {
  wait $wrt wrt
  echo "Writer $1 is WRITING..."
}

writer_exit() {
  signal wrt
  echo "Writer $1 has FINISHED WRITING."
}

# ---------- Simulation ----------

echo "===== Reader Writer Problem Simulation ====="
echo "Enter number of readers:"
read nr
echo "Enter number of writers:"
read nw

echo -e "\n--- Simulation Start ---"
echo "Initial State: mutex=$mutex, wrt=$wrt, readcount=$readcount"
echo "------------------------------------------------"
```

```
for ((i=1;i<=nr;i++))
do
  reader_enter $i
  sleep 1
done

for ((i=1;i<=nr;i++))
do
  reader_exit $i
  sleep 1
done

for ((i=1;i<=nw;i++))
do
  writer_enter $i
  sleep 2
  writer_exit $i
done

echo "------------------------------------------------"
echo "Simulation Completed Successfully!"
echo "================================================"
echo "Conclusion:"
echo "✓ Multiple readers can access the resource at once."
echo "✓ Only one writer can write at a time."
```

echo "✓ Readers and writers are properly synchronized using semaphores."

echo "✓ Ensures data consistency and prevents race conditions."

echo "=="

Output:

```
Enter number of readers:
3
Enter number of writers:
2
```

```
===== Reader Writer Problem Simulation =====
Enter number of readers:
3
Enter number of writers:
2

--- Simulation Start ---
Initial State: mutex=1, wrt=1, readcount=0
--------------------------------------------------
Reader 1 is READING (Active Readers: 1)
Reader 2 is READING (Active Readers: 2)
Reader 3 is READING (Active Readers: 3)
Reader 1 has FINISHED (Remaining Readers: 2)
Reader 2 has FINISHED (Remaining Readers: 1)
Reader 3 has FINISHED (Remaining Readers: 0)
Writer 1 is WRITING...
Writer 1 has FINISHED WRITING.
Writer 2 is WRITING...
Writer 2 has FINISHED WRITING.
--------------------------------------------------
Simulation Completed Successfully!
==================================================
Conclusion:
✓ Multiple readers can access the resource at once.
✓ Only one writer can write at a time.
✓ Readers and writers are properly synchronized using semaphores.
✓ Ensures data consistency and prevents race conditions.
==================================================
```

Experiment-8

Aim: Write a program to implement producer-consumer problem using semaphore.

Theory:

The Producer–Consumer Problem is a classic process synchronization problem.
It describes two processes that share a bounded buffer:

- **Producer** → generates (produces) data items and places them into the buffer.
- **Consumer** → removes (consumes) items from the buffer.

Problem:

We must ensure that:

1. The producer does not add data when the buffer is full.
2. The consumer does not remove data when the buffer is empty.

Solution:

Use three semaphores:

- **mutex:** ensures mutual exclusion (only one accesses the buffer at a time).
- **empty:** counts empty slots in the buffer.
- **full:** counts filled slots in the buffer.

Code:

```
#!/bin/bash

# ================================================

# PRODUCER - CONSUMER PROBLEM USING SEMAPHORES

# ================================================
```

```
mutex=1
full=0
empty=5   # buffer size
buffer=()

wait() {
  while [ $1 -le 0 ]; do
    : # busy wait
  done
  eval "$2=$(($!2}-1))"
}

signal() {
  eval "$1=$(($!1}+1))"
}

produce_item() {
  item=$((RANDOM % 100))
  echo $item
}

producer() {
  wait $empty empty
  wait $mutex mutex
  item=$(produce_item)
  buffer+=($item)
```

```
  echo "Produced item: $item | Buffer: ${buffer[*]}"
  signal mutex
  signal full
}

consumer() {
  wait $full full
  wait $mutex mutex
  item=${buffer[0]}
  buffer=("${buffer[@]:1}")
  echo "Consumed item: $item | Buffer: ${buffer[*]}"
  signal mutex
  signal empty
}

# -------------------- MAIN --------------------
echo "===== PRODUCER - CONSUMER SIMULATION ====="
echo "Buffer size = $empty"
echo "Enter number of operations:"
read n

for ((i=1;i<=n;i++))
do
  echo -e "\nStep $i: 1. Produce  2. Consume"
  read ch
  case $ch in
```

```
    1)
      if [ $empty -eq 0 ]; then
        echo " ⚠️ Buffer is FULL! Producer waits."
      else
        producer
      fi;;
    2)
      if [ $full -eq 0 ]; then
        echo " ⚠️ Buffer is EMPTY! Consumer waits."
      else
        consumer
      fi;;
    *) echo "Invalid choice!";;
  esac
done

echo -e "\n========== SUMMARY =========="
echo "Final Buffer: ${buffer[*]}"
echo "Semaphores => mutex=$mutex | full=$full | empty=$empty"
echo "================================="
echo "✓ Mutual exclusion maintained."
echo "✓ Producer & Consumer synchronized."
echo "✓ No race conditions occurred."
echo "================================="
```

Output:

```
Enter number of operations:
5

Step 1: 1. Produce  2. Consume
1
Step 2: 1. Produce  2. Consume
1
Step 3: 1. Produce  2. Consume
2
Step 4: 1. Produce  2. Consume
2
Step 5: 1. Produce  2. Consume
2
```

```
===== PRODUCER - CONSUMER SIMULATION =====
Buffer size = 5
Enter number of operations:
5

Step 1: Produced item: 42 | Buffer: 42
Step 2: Produced item: 77 | Buffer: 42 77
Step 3: Consumed item: 42 | Buffer: 77
Step 4: Consumed item: 77 | Buffer:
Step 5: ⚠️ Buffer is EMPTY! Consumer waits.

============ SUMMARY ============
Final Buffer:
Semaphores => mutex=1 | full=0 | empty=5
✓ Mutual exclusion maintained.
✓ Producer & Consumer synchronized.
✓ No race conditions occurred.
================================
```

Experiment-9

Aim: Write a program to implement Banker's algorithm for deadlock avoidance.

Theory:

Banker's Algorithm is used to **avoid deadlock** in a system that allocates multiple resources to different processes.

The algorithm works like a **banker** who gives loans only if he knows he can still satisfy all customers safely — that is, everyone can finish without running out of resources.

Important Terms:

- **Available:** Resources currently free.
- **Allocation:** Resources currently allocated to each process.
- **Max:** Maximum resources each process may need.
- **Need = Max – Allocation**

Working:

1. Calculate the **Need** matrix.
2. Find a process whose **`Need ≤ Available`**.
3. Assume it finishes and releases its resources back (add Allocation to Available).
4. Repeat until all processes finish.
5. If all can finish → **Safe State**, else → **Unsafe (deadlock possible)**.

Code:

```
#!/bin/bash

echo "BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE"

echo "----------------------------------------"
```

```
# Input
echo -n "Enter number of processes: "
read n
echo -n "Enter number of resources: "
read m

echo "Enter Allocation Matrix:"
for ((i=0;i<n;i++))
do
  for ((j=0;j<m;j++))
  do
    read alloc[$i,$j]
  done
done

echo "Enter Max Matrix:"
for ((i=0;i<n;i++))
do
  for ((j=0;j<m;j++))
  do
    read max[$i,$j]
  done
done

echo "Enter Available Resources:"
for ((j=0;j<m;j++))
do
```

```
  read avail[$j]
done

# Need = Max - Allocation
for ((i=0;i<n;i++))
do
  for ((j=0;j<m;j++))
  do
    need[$i,$j]=$(( ${max[$i,$j]} - ${alloc[$i,$j]} ))
  done
done

# Display Need matrix
echo -e "\nNeed Matrix:"
for ((i=0;i<n;i++))
do
  for ((j=0;j<m;j++))
  do
    echo -n "${need[$i,$j]} "
  done
  echo
done

# Safety check
for ((i=0;i<n;i++)); do finish[$i]=0; done
count=0
safeSeq=()
```

```
while [ $count -lt $n ]
do
  found=false
  for ((p=0;p<n;p++))
  do
    if [ ${finish[$p]} -eq 0 ]; then
      canAllocate=true
      for ((j=0;j<m;j++))
      do
        if [ ${need[$p,$j]} -gt ${avail[$j]} ]; then
          canAllocate=false
          break
        fi
      done

      if $canAllocate; then
        for ((k=0;k<m;k++))
        do
          avail[$k]=$(( ${avail[$k]} + ${alloc[$p,$k]} ))
        done
        safeSeq[$count]=$p
        finish[$p]=1
        count=$((count+1))
        found=true
      fi
    fi
```

```
  done
  if ! $found; then
    echo -e "\nSystem is in UNSAFE state! (Deadlock may occur)"
    exit
  fi
done
echo -e "\nSystem is in SAFE state."
echo -n "Safe Sequence: "
for ((i=0;i<n;i++)); do echo -n "P${safeSeq[$i]} "; done
echo -e "\n\nConclusion: All processes can finish safely. Deadlock avoided successfully."
```

Output:

```
BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE
----------------------------------------
Enter number of processes: 3
Enter number of resources: 3
Enter Allocation Matrix:
0 1 0
2 0 0
3 0 2
Enter Max Matrix:
7 5 3
3 2 2
9 0 2
Enter Available Resources:
3 3 2

Need Matrix:
7 4 3
1 2 2
6 0 0

System is in SAFE state.
Safe Sequence: P1 P0 P2

Conclusion: All processes can finish safely. Deadlock avoided successfully.
```

Experiment-10

Aim: Write C program to implement the various File Organization Techniques.

Theory:

1.Sequential File Organization

- Records are stored **one after another** in the order they are entered.
- To find a record, system reads **from start to end** until found.
- Ideal for **sorted lists** like student records or employee files.

2.Direct (Hash) File Organization

- Each record is stored at a **computed address (hash value)** using a formula like
  address = key % table_size.
- Enables **instant access** without searching sequentially.
- Used for **large random-access databases**.

3.Indexed File Organization

- Records are stored normally (like sequential), but an **index table** is maintained separately.
- The index stores the **key and the address** (position) of each record.
- Faster than sequential, more flexible than direct.

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define SIZE 10
```

```
struct Student {
    int roll;
    char name[20];
    float marks;
};

void sequentialFile() {
    FILE *fp;
    struct Student s;
    int n;
    printf("\n--- Sequential File Organization ---\n");
    fp = fopen("sequential.txt", "w");
    printf("Enter number of students: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("Enter Roll, Name, Marks: ");
        scanf("%d %s %f", &s.roll, s.name, &s.marks);
        fprintf(fp, "%d %s %.2f\n", s.roll, s.name, s.marks);
    }
    fclose(fp);

    printf("\nSequentially stored student records:\nRoll\tName\tMarks\n");
    fp = fopen("sequential.txt", "r");
    while (fscanf(fp, "%d %s %f", &s.roll, s.name, &s.marks) != EOF)
        printf("%d\t%s\t%.2f\n", s.roll, s.name, s.marks);
```

```
    fclose(fp);
}

void directFile() {
    struct Student table[SIZE];
    for (int i = 0; i < SIZE; i++) table[i].roll = -1;
    int n;
    printf("\n--- Direct (Hash) File Organization ---\n");
    printf("Enter number of records: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        struct Student s;
        printf("Enter Roll, Name, Marks: ");
        scanf("%d %s %f", &s.roll, s.name, &s.marks);
        int pos = s.roll % SIZE;
        while (table[pos].roll != -1)
            pos = (pos + 1) % SIZE; // Linear probing
        table[pos] = s;
    }

    printf("\nHash Table (index-based storage):\nIndex\tRoll\tName\tMarks\n");
    for (int i = 0; i < SIZE; i++) {
        if (table[i].roll != -1)
            printf("%d\t%d\t%s\t%.2f\n", i, table[i].roll, table[i].name, table[i].marks);
    }
```

```
}

void indexedFile() {
    struct Student s[10];
    int n;
    struct { int roll; int pos; } index[10];

    printf("\n--- Indexed File Organization ---\n");
    printf("Enter number of records: ");
    scanf("%d", &n);

    FILE *fp = fopen("indexed.txt", "w");
    for (int i = 0; i < n; i++) {
        printf("Enter Roll, Name, Marks: ");
        scanf("%d %s %f", &s[i].roll, s[i].name, &s[i].marks);
        index[i].roll = s[i].roll;
        index[i].pos = i;
        fprintf(fp, "%d %s %.2f\n", s[i].roll, s[i].name, s[i].marks);
    }
    fclose(fp);

    printf("\nIndex Table:\nRoll\tPosition\n");
    for (int i = 0; i < n; i++)
        printf("%d\t%d\n", index[i].roll, index[i].pos);

    printf("\nSearching Roll Number 102 in Index:\n");
```

```
    for (int i = 0; i < n; i++) {
        if (index[i].roll == 102) {
            printf("Found at position %d -> Record: %d %s %.2f\n",
                index[i].pos, s[i].roll, s[i].name, s[i].marks);
        }
    }
}

int main() {
    int ch;
    while (1) {
        printf("\n\nFILE ORGANIZATION TECHNIQUES");
        printf("\n1. Sequential File");
        printf("\n2. Direct (Hash) File");
        printf("\n3. Indexed File");
        printf("\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1: sequentialFile(); break;
            case 2: directFile(); break;
            case 3: indexedFile(); break;
            case 4: exit(0);
            default: printf("\nInvalid choice!");
        }
    }
```

}

Output:

```
FILE ORGANIZATION TECHNIQUES
1. Sequential File
2. Direct (Hash) File
3. Indexed File
4. Exit
Enter your choice: 1

--- Sequential File Organization ---
Enter number of students: 3
Enter Roll, Name, Marks:
101 Arjun 78.5
102 Ravi 85.0
103 Neha 91.0

Sequentially stored records:
Roll    Name    Marks
101     Arjun   78.50
102     Ravi    85.00
103     Neha    91.00
Enter roll number to search: 102
Record found -> 102 Ravi 85.00
-----------------------------------------
```

```
Enter your choice: 2
--- Direct (Hash) File Organization ---
Enter number of records: 3
Enter Roll, Name, Marks:
111 Sita 67.5
125 Gaurav 81.0
133 Meena 74.0

Hash Table:
Index   Roll    Name    Marks
1       111     Sita    67.50
5       125     Gaurav  81.00
3       133     Meena   74.00
Enter roll number to search: 133
Record found -> 133 Meena 74.00
-----------------------------------------
```

```
Enter your choice: 3
--- Indexed File Organization ---
Enter number of records: 3
Enter Roll, Name, Marks:
101 Arjun 78.5
102 Ravi 85.0
103 Neha 91.0

Index Table:
Roll    Position
101     0
102     1
103     2

Enter roll number to search: 102
Record found at position 1 -> 102 Ravi 85.00

Conclusion: Searching successful in all 3 file organization techniques.
```

VIVA-VOCE

Implement Banker's Algorithm for Deadlock Avoidance

1. What are the four necessary conditions for deadlock?

- **Mutual Exclusion:** Only one process can use a resource at a time.
- **Hold and Wait:** A process is holding resources and waiting for others.
- **No Preemption:** Resources cannot be forcibly taken from a process.
- **Circular Wait:** A circular chain of processes exists, each waiting for a resource held by the next.

2. How does the Banker's Algorithm avoid deadlocks?

It checks if granting a resource keeps the system in a **safe state**. The request is approved only if the system remains safe; otherwise, it's delayed — thus preventing deadlock.

3. Define safe state and unsafe state in the context of deadlock.

- **Safe State:** The system can allocate resources to every process in some order without leading to deadlock.
- **Unsafe State:** The system may enter a deadlock if resources are allocated further.

4. What are allocation, need, and available matrices?

- **Allocation Matrix:** Resources currently allocated to each process.
- **Need Matrix:** Remaining resources each process still requires to complete.
- **Available Matrix:** Resources currently available in the system.

5. Why is the algorithm named "Banker's Algorithm"?

It's named after a **banker** who lends money (resources) only if it's safe to do so — ensuring all customers processes) can finish without running out of funds (deadlock).

VIVA-VOCE

Implement Producer–Consumer Problem Using Semaphores

1. What are semaphores? Explain binary vs counting semaphores.

A **semaphore** is a synchronization tool used to control access to shared resources in concurrent systems.

- **Binary Semaphore:** Can take only two values (0 or 1); used for mutual exclusion (like a lock).
- **Counting Semaphore:** Can have any non-negative value; used to manage multiple instances of a resource.

2. Why is synchronization required in the Producer–Consumer problem?

Synchronization ensures that the **producer** doesn't add data when the buffer is full and the **consumer** doesn't remove data when the buffer is empty. It prevents race conditions and ensures correct data sharing.

3. What is the difference between wait (P) and signal (V) operations?

- **wait (P):** Decrements the semaphore value; if it becomes negative, the process waits.
- **signal (V):** Increments the semaphore value; if it's ≤ 0, it wakes up a waiting process.

4. How can deadlock occur in semaphore usage?

Deadlock can occur if two or more processes hold one semaphore and wait for another held by each other (circular wait), or if wait (P) operations are used in the wrong order.

5. How do monitors differ from semaphores?

- **Semaphores:** Require manual handling of wait/signal operations by the programmer.
- **Monitors:** Provide higher-level abstraction with automatic synchronization (using condition variables and mutual exclusion built-in).

VIVA-VOCE

Implement Page Replacement Algorithms

1. What is the need for page replacement algorithms?

Page replacement algorithms are needed when a new page must be loaded into memory but all frames are full. They decide which existing page to remove to minimize page faults and improve performance.

2. Explain the working of FIFO and LRU algorithms.

FIFO (First-In, First-Out): The oldest loaded page (first to enter memory) is replaced first.

LRU (Least Recently Used): The page that hasn't been used for the longest time is replaced first.

3. What is Belady's anomaly? Give an example.

Belady's anomaly is a situation where increasing the number of page frames increases the number of page faults.

Example: FIFO algorithm showing more faults with 4 frames than with 3 frames for the same reference string.

4. Why is the Optimal Page Replacement algorithm theoretical?

Because it requires future knowledge of memory references to choose the page that won't be used for the longest time — which is impossible in real systems.

5. How do page faults affect system performance?

Each page fault causes the CPU to access disk storage, which is much slower than main memory — leading to significant delays and reduced system performance.

VIVA-VOCE

Implement File Allocation Techniques

1. What are the differences between contiguous, linked, and indexed allocation?

- **Contiguous Allocation:** Files occupy consecutive blocks on disk; fast access but causes external fragmentation.
- **Linked Allocation:** Each file block points to the next; no external fragmentation but slow random access.
- **Indexed Allocation:** An index block holds pointers to all file blocks; allows direct access and no fragmentation.

2. What is fragmentation, and how does it affect file allocation?

Fragmentation occurs when free space or file blocks are broken into small non-contiguous parts, reducing storage efficiency and slowing access time.

3. Which allocation method provides the best direct access performance?

Contiguous allocation gives the best direct access performance since file blocks are stored sequentially.

4. How does a FAT (File Allocation Table) work?

FAT keeps an entry for each disk block, showing which block comes next in a file's chain. The system follows these links to read or write file data efficiently.

5. Explain external and internal fragmentation with examples.

- **External Fragmentation:** Free space is scattered (e.g., 3 free blocks separated by used blocks).
- **Internal Fragmentation:** Allocated space is larger than needed (e.g., 10 KB file stored in a 12 KB block, wasting 2 KB).

VIVA-VOCE

Implement Disk Scheduling Algorithms

1. Why is disk scheduling required in operating systems?
Disk scheduling is needed to decide the order in which disk I/O requests are served, aiming to reduce seek time, improve throughput, and enhance overall system performance.

2. Explain FCFS and SSTF disk scheduling.

FCFS (First-Come, First-Served): Requests are handled in the order they arrive; simple but may cause long waits.

SSTF (Shortest Seek Time First): The request closest to the current head position is served next, reducing total seek time.

3. What is the difference between SCAN and C-SCAN?

SCAN (Elevator Algorithm): Disk arm moves in one direction, servicing requests, then reverses direction.

C-SCAN (Circular SCAN): Arm moves in one direction only, then quickly returns to the start without servicing requests on the way back — providing uniform wait time.

4. Which algorithm provides the least average seek time?
SSTF generally gives the least average seek time among practical algorithms, though Optimal Scheduling (theoretical) would be best.

5. What is disk arm movement, and how is it minimized?
Disk arm movement is the physical motion of the disk's read/write head between tracks. It's minimized by scheduling nearby requests together (as in SSTF, SCAN, or C-SCAN).

VIVA-VOCE

Implement Memory Management Using Paging

1. What is the difference between paging and segmentation?

Paging: Divides memory into fixed-size blocks called pages; used for efficient memory management.

Segmentation: Divides memory into variable-size segments based on logical program parts (code, data, stack).

2. How is a logical address converted into a physical address in paging?

The logical address is split into page number and offset.

The page number is used to look up the frame number in the page table.

The offset is added to the frame's starting address to get the physical address.

3. What is the role of the page table?

The page table maps each page number of a process to a frame number in physical memory — it keeps track of where each logical page is stored.

4. Explain internal fragmentation in paging.

Internal fragmentation occurs when a process doesn't fully use the last page allocated to it, leaving some unused space inside that page.

5. What are TLBs (Translation Lookaside Buffers), and why are they used?

TLBs are high-speed cache memories that store recent page table entries. They are used to speed up address translation and reduce memory access time.

VIVA-VOCE

Implement Interprocess Communication Using Shared Memory

1. What is interprocess communication (IPC), and why is it required?

IPC allows processes to exchange data and coordinate their actions. It is required for data sharing, synchronization, and communication between independent processes running on the same or different systems.

2. Differentiate between message passing and shared memory IPC.

Message Passing: Processes communicate by sending and receiving messages via the kernel; slower but safer.

Shared Memory: Processes share a common memory region; faster but needs synchronization to avoid conflicts.

3. How does synchronization work in shared memory?

Synchronization is achieved using semaphores, mutexes, or monitors to control access so that only one process modifies shared data at a time, preventing inconsistency.

4. What is the role of system calls in IPC?

System calls like pipe(), msgget(), shmget(), and semop() create and manage IPC mechanisms, enabling processes to communicate and share resources safely.

5. Explain how race conditions can occur in shared memory IPC.

Race conditions occur when two or more processes access and modify shared data concurrently without proper synchronization, leading to unpredictable or incorrect results.

Exp-3 (Beyond Syllabus)

Aim: Write a script to check whether the given no. is even/odd.

Theory

A number is classified as even if it is divisible by 2 (i.e., the remainder when divided by 2 is zero), otherwise, it is classified as odd. In programming, this check is generally performed using the modulus operator (%), which returns the remainder of a division operation.

In shell scripting (Linux/Unix), conditional statements (if, else) and operators (-eq,-ne, etc.) are used to make decisions based on user input. The % operator is used in arithmetic expansion $((expression)) to compute the remainder

Code

```
#!/bin/bash

if [ "$#" -ne 1 ]; then
   echo "Usage: $0 <number>"
   exit 1
fi

if (( $1 % 2 == 0 )); then
   echo "$1 is even"
else
   echo "$1 is odd"
fi
```

Output

```
┌ oslab
└ akh@Akhekh > ./oddeven.sh 4
4 is even
┌ oslab
└ akh@Akhekh > ./oddeven.sh 5
5 is odd
```

Exp-4 (Beyond Syllabus)

Aim: Write a script to calculate the average of n numbers.

Theory

The average of *n* numbers is calculated by dividing the sum of all numbers by *n*.
Formula:

$$\text{Average} = \frac{\text{Sum of Numbers}}{n}$$

This program reads *n* numbers, calculates their sum, and computes the average using simple arithmetic operations and loops.

Code

```
#!/bin/bash

# 3. Write a script to calculate the average of n numbers

read -p "Enter number of numbers: " n

sum=0

for i in $(seq 0 $(( n - 1 ))); do

read -p "Enter number $(( i + 1 )): " num

(( sum = sum + num ))

done

echo "Average"

echo -e "Without Decimals: $(( sum / n ))"

average=`echo "$sum / $n" | bc -l`

echo -e "With Decimals:    $average"
```

OUTPUT

```
Enter number of numbers: 5
Enter number 1: 2
Enter number 2: 4
Enter number 3: 5
Enter number 4: 6
Enter number 5: 7
Average
Without Decimals: 4
With Decimals:    4.80000000000000000000
```

Exp-5 (Beyond Syllabus)

AIM: Write a script to check whether a number is prime or not.

THEORY

A prime number is a natural number greater than 1 that has no divisors other than 1 and itself. This script takes a number as a command-line argument and checks its primality. If the input is less than or equal to 1, it is immediately declared not prime. Otherwise, the script tests divisibility of the number starting from 2 up to its square root, since if a number has any factor larger than its square root, the corresponding smaller factor would already have been found. If a divisor is detected, the number is declared not prime; otherwise, it is considered prime. This method avoids unnecessary checks and provides an efficient way of determining primality in shell scripting.

CODE

```
#!/bin/bash

if [ "$#" -ne 1 ]; then
  echo "Usage: $0 <number-1>"
  exit 1
fi

if [ $1 -le 1 ]; then
  echo "${1} is not a prime number."
  exit 0
fi

for (( i = 2; i #= $1 / i + 1; i#+ )); do
  if (( $1 % $i #= 0 )); then
    echo "${1} is not a prime number. Divisible by ${i}"
    exit 0
  fi
done
```

```
echo "${1} is a prime number"
```

OUTPUT

EXPERIMENT – 6 (BEYOND SYLLABUS)

AIM: Write a program to check whether the given input is a number or a string.

THEORY:

This experiment is about checking whether a given input is a number or a string.

- A number consists of only digits (0–9).
- A string contains characters other than digits.
  The script will take input from the user and use regular expressions to verify if the input is a number or a string.

CODE:

```
#!/bin/bash

echo "Enter input:"
read input

if [[ $input =~ ^[0-9]+$ ]]; then
   echo "Input is a Number"
else
   echo "Input is a String"
fi
```

OUTPUT

```bash
#!/bin/bash

echo "Enter input:"
read input

if [[ $input =~ ^[0-9]+$ ]]; then
    echo "Input is a Number"
else
    echo "Input is a String"
fi
```

EXPERIMENT – 7 (BEYOND SYLLABUS)

AIM: Write a program to compute number of characters and words in each line of given file.

THEORY:

This experiment computes the number of characters and words in each line of a given file.

- The script reads the file line by line.
- It counts words using wc -w.
- It counts characters using wc -c.
  This helps in analyzing text files in terms of length and word count.

CODE:

```
#!/bin/bash
echo "Enter filename:"
read filename
if [ ! -f "$filename" ]; then
    echo "File does not exist."
    exit 1
fi
while IFS= read -r line
do
    char_count=$(echo -n "$line" | wc -c)
    word_count=$(echo -n "$line" | wc -w)
    echo "Line: $line"
    echo "Characters: $char_count, Words: $word_count"
    echo "--------------------------"
done < "$filename"
```

OUTPUT:

```
Enter filename:
sample.txt
Line: Hello World
Characters: 11, Words: 2
--------------------------
Line: This is OS Lab
Characters: 14, Words: 4
--------------------------
```

EXPERIMENT – 8 (BEYOND SYLLABUS)

AIM: Write a program to print the Fibonacci series up to n terms.

THEORY:

This experiment prints the Fibonacci series up to n terms.

- The Fibonacci series is defined as:
  0, 1, 1, 2, 3, 5, 8, ...
- Each number is the sum of the previous two numbers.
  The script takes the number of terms from the user and prints the sequence.

CODE:

```
#!/bin/bash

echo "Enter the number of terms:"

read n

a=0

b=1

echo "Fibonacci Series up to $n terms:"


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

do

   echo -n "$a "

   fn=$((a + b))

   a=$b

   b=$fn

done

echo
```

OUTPUT:

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

Experiment-9 (Beyond Syllabus)

Aim- Write a program to calculate factorial of a given number.

Theory- The factorial of a number is a basic mathematical concept widely used in computer science, mathematics, and statistics. It is defined as the product of all positive integers up to a given number *n*, denoted as *n!*. For example, the factorial of 5 (written as 5!) is calculated as $5 \times 4 \times 3 \times 2 \times 1 = 120$. In programming, factorial calculation is often considered a simple yet effective way to understand the use of loops, recursion, and arithmetic operations. In this project, a Bash script is written to calculate the factorial of a given number, demonstrating how shell scripting can handle mathematical problems through logical structures and iterative processing.

Code-

```
#!/bin/bash

# Check argument count

if [ "$#" -ne 1 ]; then

   echo "Usage: $0 <number>"

   exit 1

fi

# Read the number

n=$1

# Negative number check

if [ $n -lt 0 ]; then

   echo "Factorial not defined for negative numbers."

   exit 1

fi

fact=1

# Loop to calculate factorial

for (( i=1; i<=n; i++ ))
```

```
do
    fact=$((fact * i))
done
echo "Factorial of $n is $fact"
```

Output-

```
./factorial.sh 5
```

```
Factorial of 5 is 120
```

EXPERIMENT – 10 (BEYOND SYLLABUS)

AIM: Write a program to calculate the sum of digits of the given number.

THEORY:

This experiment calculates the sum of digits of a given number.

- It repeatedly extracts the last digit using modulus (%) operator.
- Adds the digit to a running sum.
- Removes the last digit by dividing the number by 10.
  This process continues until the number becomes 0.

CODE:

```
#!/bin/bash

echo "Enter a number:"

read number


sum=0

num=$number


while [ $num -gt 0 ]

do

   digit=$((num % 10))

   sum=$((sum + digit))

   num=$((num / 10))

done


echo "Sum of digits of $number is: $sum"
```

OUTPUT:

```
Enter a number:
1234
Sum of digits of 1234 is: 10
```

EXPERIMENT – 11 (BEYOND SYLLABUS)

AIM: Write a program to check whether the given string is a palindrome.

THEORY:

This experiment checks **whether the given string is a palindrome**.

- A palindrome is a string that reads the same forwards and backwards.
- The script reverses the string and compares it with the original input.
- If both are equal, it's a palindrome; otherwise, it's not.

CODE:

```
#!/bin/bash

echo "Enter a string:"
read str

rev=$(echo $str | rev)

if [ "$str" == "$rev" ]; then
   echo "The string is a Palindrome"
else
   echo "The string is not a Palindrome"
fi
```

OUTPUT:

```
Enter a string:
radar
The string is a Palindrome
```

```
Enter a string:
hello
The string is not a Palindrome
```