copy.cpp – Simulates the cp command

```
// copy.cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cerr << "Usage: ./copy <source_file> <destination_file>\n";
        return 1;
    }

    string sourceFile = argv[1];
    string destFile = argv[2];

    ifstream src(sourceFile, ios::binary);
    if (!src) {
        cerr << "Error opening source file: " << sourceFile << endl;
        return 1;
    }

    ofstream dest(destFile, ios::binary);
    if (!dest) {
        cerr << "Error creating destination file: " << destFile << endl;
        return 1;
    }

    dest << src.rdbuf(); // Copy content
    src.close();
    dest.close();
```

```
    cout << "File copied from " << sourceFile << " to " << destFile << endl;
    return 0;
}
```

grep.cpp – Simulates the grep command

```
// grep.cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cerr << "Usage: ./grep <filename> <search_word>\n";
        return 1;
    }

    string filename = argv[1];
    string word = argv[2];
    ifstream file(filename);

    if (!file) {
        cerr << "Error opening file: " << filename << endl;
        return 1;
    }

    string line;
    bool found = false;
    while (getline(file, line)) {
        if (line.find(word) != string::npos) {
```

```
            cout << "Line: " << line << endl;
            found = true;
        }
    }

    if (!found) {
        cout << "Word \"" << word << "\" not found in " << filename << endl;
    }

    return 0;
}
```

main.cpp – Uses fork(), execlp(), waitpid(), getpid(), exit()

```
// main.cpp
#include <iostream>
#include <unistd.h>     // fork(), execlp(), getpid()
#include <sys/wait.h>   // waitpid()
#include <cstdlib>      // exit()

using namespace std;

int main(int argc, char* argv[]) {
    if (argc != 5) {
        cerr << "Usage: ./main <src_file> <dest_file> <grep_file> <word>\n";
        return 1;
    }

    cout << "Main Process ID: " << getpid() << endl;

    pid_t pid1 = fork();
    if (pid1 == 0) {
```

```
        // Child 1: run copy
        cout << "[Child 1 - PID: " << getpid() << "] Executing copy...\n";
        execlp("./copy", "copy", argv[1], argv[2], NULL);
        perror("execlp failed for copy");
        exit(1);
    } else if (pid1 < 0) {
        perror("Failed to fork for copy");
        exit(1);
    }

    pid_t pid2 = fork();
    if (pid2 == 0) {
        // Child 2: run grep
        cout << "[Child 2 - PID: " << getpid() << "] Executing grep...\n";
        execlp("./grep", "grep", argv[3], argv[4], NULL);
        perror("execlp failed for grep");
        exit(1);
    } else if (pid2 < 0) {
        perror("Failed to fork for grep");
        exit(1);
    }

    // Parent process waits
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
    cout << "[Parent - PID: " << getpid() << "] Both child processes completed.\n";

    return 0;
}
```

🧪 Compilation & Usage

Compile:

```
g++ copy.cpp -o copy

g++ grep.cpp -o grep

g++ main.cpp -o main
```

Run:

```
echo "This is a sample line with keyword." > input.txt

echo "Another line without." >> input.txt


./main input.txt output.txt output.txt searchword
```

Features Demonstrated:

fork() – To create new processes

execlp() – To run copy and grep commands

waitpid() – To wait for both children

getpid() – To show process IDs

exit() – Used in children if execlp() fails

ASSIGNMENT 2

Write a program to implement scheduling algorithms – FCFS, SJF, Round Robin and Priority.

```
#include <iostream>

#include <vector>

#include <algorithm>

#include <climits>

using namespace std;

struct Process {

int id, at, bt, ct, tat, wt, rt, priority;

};

// Function Prototypes
```

```
void fcfs_scheduling(int n, vector<Process>& proc);
void preemptive_sjf(vector<Process>& proc, int n);
void non_preemptive_sjf(vector<Process>& proc, int n);
void non_preemptive_priority_scheduling(vector<Process>& proc, int n);
void preemptive_priority_scheduling(vector<Process>& proc, int n);
void round_robin_scheduling(int n, vector<Process>& proc, int quant);
void printResults(vector<Process>& proc, int n);
int main() {
int n, choice, quantum;
cout << "Select Scheduling Algorithm:\n";
cout << "1. FCFS\n2. SJF (Non-Preemptive)\n3. SJF (Preemptive)\n";
cout << "4. Priority Scheduling (Non-Preemptive)\n5. Priority Scheduling (Preemptive)\n";
cout << "6. Round Robin\nEnter choice: ";
cin >> choice;
cout << "Enter the number of processes: ";
cin >> n;
vector<Process> proc(n);
for (int i = 0; i < n; i++) {
proc[i].id = i + 1;
cout << "Enter Arrival Time and Burst Time for Process " << i + 1 << ": ";
cin >> proc[i].at >> proc[i].bt;
proc[i].rt = proc[i].bt;
proc[i].priority = 0; // Default
}
if (choice == 4 || choice == 5) {
for (int i = 0; i < n; i++) {
cout << "Enter Priority for Process " << i + 1 << ": ";
cin >> proc[i].priority;
}
}
if (choice == 6) {
```

```
cout << "Enter Time Quantum: ";
cin >> quantum;
}
switch (choice) {
case 1: fcfs_scheduling(n, proc); break;
case 2: non_preemptive_sjf(proc, n); break;
case 3: preemptive_sjf(proc, n); break;
case 4: non_preemptive_priority_scheduling(proc, n); break;
case 5: preemptive_priority_scheduling(proc, n); break;
case 6: round_robin_scheduling(n, proc, quantum); break;
default: cout << "Invalid choice!\n"; return 0;
}
printResults(proc, n);
return 0;
}
// FCFS Scheduling
void fcfs_scheduling(int n, vector<Process>& proc) {
sort(proc.begin(), proc.end(), [](Process a, Process b) {
return a.at < b.at;
});
proc[0].ct = proc[0].at + proc[0].bt;
for (int i = 1; i < n; i++) {
proc[i].ct = max(proc[i].at, proc[i - 1].ct) + proc[i].bt;
}
for (int i = 0; i < n; i++) {
proc[i].tat = proc[i].ct - proc[i].at;
proc[i].wt = proc[i].tat - proc[i].bt;
}
}
// Preemptive SJF
void preemptive_sjf(vector<Process>& proc, int n) {
```

```
int completed = 0, time = 0;
while (completed < n) {
int idx = -1, min_bt = INT_MAX;
for (int i = 0; i < n; i++) {
if (proc[i].at <= time && proc[i].rt > 0 && proc[i].rt < min_bt) {
min_bt = proc[i].rt;
idx = i;
}
}
if (idx == -1) time++;
else {
proc[idx].rt--;
if (proc[idx].rt == 0) {
proc[idx].ct = time + 1;
proc[idx].tat = proc[idx].ct - proc[idx].at;
proc[idx].wt = proc[idx].tat - proc[idx].bt;
completed++;
}
time++;
}
}
}
// Non-Preemptive SJF
void non_preemptive_sjf(vector<Process>& proc, int n) {
sort(proc.begin(), proc.end(), [](Process a, Process b) {
return a.bt < b.bt;
});
fcfs_scheduling(n, proc);
}
// Non-Preemptive Priority Scheduling
void non_preemptive_priority_scheduling(vector<Process>& proc, int n) {
```

```
sort(proc.begin(), proc.end(), [](Process a, Process b) {
return a.priority < b.priority;
});
fcfs_scheduling(n, proc);
}
// Preemptive Priority Scheduling
void preemptive_priority_scheduling(vector<Process>& proc, int n) {
int completed = 0, time = 0;
while (completed < n) {
int idx = -1, highest_priority = INT_MAX;
for (int i = 0; i < n; i++) {
if (proc[i].at <= time && proc[i].rt > 0 && proc[i].priority < highest_priority) {
highest_priority = proc[i].priority;
idx = i;
}
}
if (idx == -1) time++;
else {
proc[idx].rt--;
if (proc[idx].rt == 0) {
proc[idx].ct = time + 1;
proc[idx].tat = proc[idx].ct - proc[idx].at;
proc[idx].wt = proc[idx].tat - proc[idx].bt;
completed++;
}
time++;
}
}
}
// Round Robin Scheduling
void round_robin_scheduling(int n, vector<Process>& proc, int quant) {
```

```
vector<int> temp(n);
for (int i = 0; i < n; i++) temp[i] = proc[i].bt;
int sum = 0, count = 0, i = 0, y = n;
while (y != 0) {
if (temp[i] <= quant && temp[i] > 0) {
sum += temp[i];
temp[i] = 0;
count = 1;
} else if (temp[i] > 0) {
temp[i] -= quant;
sum += quant;
}
if (temp[i] == 0 && count == 1) {
y--;
proc[i].ct = sum;
proc[i].tat = proc[i].ct - proc[i].at;
proc[i].wt = proc[i].tat - proc[i].bt;
count = 0;
}
i = (i == n - 1) ? 0 : (proc[i + 1].at <= sum ? i + 1 : 0);
}
}
// Print Final Results
void printResults(vector<Process>& proc, int n) {
int totalWT = 0, totalTAT = 0;
cout << "\nID\tAT\tBT\tCT\tTAT\tWT\n";
for (int i = 0; i < n; i++) {
totalWT += proc[i].wt;
totalTAT += proc[i].tat;
cout << proc[i].id << "\t" << proc[i].at << "\t" << proc[i].bt << "\t"
<< proc[i].ct << "\t" << proc[i].tat << "\t" << proc[i].wt << "\n";
```

```
}
cout << "\nTotal TAT: " << totalTAT << ", Total WT: " << totalWT << endl;
cout << "Average TAT: " << (float)totalTAT / n << ", Average WT: " << (float)totalWT / n << endl;
}
```

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

using namespace std;

struct Process {
   int id;
   int burst_time;
   int priority;
   int arrival_time;
   int waiting_time;
   int turn_around_time;
   int completion_time;
};

void findCompletionTime(vector<Process>& processes) {
   int n = processes.size();
   sort(processes.begin(), processes.end(), [](Process& a, Process& b) {
      return a.arrival_time < b.arrival_time;
   });

   processes[0].completion_time = processes[0].arrival_time + processes[0].burst_time;

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

```
        if (processes[i].arrival_time > processes[i - 1].completion_time) {
            processes[i].completion_time = processes[i].arrival_time + processes[i].burst_time;
        } else {
            processes[i].completion_time = processes[i - 1].completion_time + processes[i].burst_time;
        }
    }
}

void findWaitingTimeAndTurnAroundTime(vector<Process>& processes) {
    int n = processes.size();
    for (int i = 0; i < n; i++) {
        processes[i].turn_around_time = processes[i].completion_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turn_around_time - processes[i].burst_time;
    }
}

void printTable(const vector<Process>& processes) {
    cout << "\nProcess ID | Burst Time | Arrival Time | Waiting Time | Turnaround Time | Completion Time\n";
    for (const auto& process : processes) {
        cout << "    " << process.id
             << "      |     " << process.burst_time
             << "     |      " << process.arrival_time
             << "      |      " << process.waiting_time
             << "      |       " << process.turn_around_time
             << "       |       " << process.completion_time << "\n";
    }
}

void printGanttChart(const vector<Process>& processes) {
    cout << "\nGantt Chart:\n";
```

```
    for (const auto& process : processes) {
        cout << "| P" << process.id << " ";
    }
    cout << "|\n";
    for (int i = 0; i < processes.size(); i++) {
        if (i == 0) cout << "0 ";
        cout << processes[i].completion_time << " ";
    }
    cout << "\n";
}

void calculateAverages(const vector<Process>& processes) {
    int total_waiting_time = 0;
    int total_turnaround_time = 0;
    int n = processes.size();

    for (int i = 0; i < n; i++) {
        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turn_around_time;
    }

    double avg_waiting_time = (double)total_waiting_time / n;
    double avg_turnaround_time = (double)total_turnaround_time / n;

    cout << "\nAverage Waiting Time: " << avg_waiting_time << endl;
    cout << "Average Turnaround Time: " << avg_turnaround_time << endl;
}

void FCFS() {
    int n;
    cout << "Enter the number of processes: ";
```

```
    cin >> n;

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

    findCompletionTime(processes);
    findWaitingTimeAndTurnAroundTime(processes);
    printTable(processes);
    printGanttChart(processes);
    calculateAverages(processes);
}

void SJF() {
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

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

```
    sort(processes.begin(), processes.end(), [](Process& a, Process& b) {
        if (a.arrival_time == b.arrival_time)
            return a.burst_time < b.burst_time;
        return a.arrival_time < b.arrival_time;
    });

    findCompletionTime(processes);
    findWaitingTimeAndTurnAroundTime(processes);
    printTable(processes);
    printGanttChart(processes);
    calculateAverages(processes);
}

void RoundRobin() {
    int n, quantum;
    cout << "Enter the number of processes: ";
    cin >> n;
    cout << "Enter the quantum time: ";
    cin >> quantum;

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

    vector<int> remaining_burst_time(n);
```

```
for (int i = 0; i < n; i++) {
    remaining_burst_time[i] = processes[i].burst_time;
}

int time = 0;
bool done;

while (true) {
    done = true;

    for (int i = 0; i < n; i++) {
        if (remaining_burst_time[i] > 0 && processes[i].arrival_time <= time) {
            done = false;

            if (remaining_burst_time[i] > quantum) {
                time += quantum;
                remaining_burst_time[i] -= quantum;
            } else {
                time += remaining_burst_time[i];
                processes[i].completion_time = time;
                remaining_burst_time[i] = 0;
            }
        }
    }

    if (done) break;

    // Handle case where CPU is idle (no process has arrived)
    bool allFuture = true;
    for (int i = 0; i < n; i++) {
        if (remaining_burst_time[i] > 0 && processes[i].arrival_time <= time) {
```

```
            allFuture = false;
            break;
        }
    }
    if (allFuture) time++;
  }

  findWaitingTimeAndTurnAroundTime(processes);
  printTable(processes);
  printGanttChart(processes);
  calculateAverages(processes);
}


void PriorityScheduling() {
  int n;
  cout << "Enter the number of processes: ";
  cin >> n;

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

  sort(processes.begin(), processes.end(), [](Process& a, Process& b) {
```

```
        if (a.arrival_time == b.arrival_time)
            return a.priority < b.priority;
        return a.arrival_time < b.arrival_time;
    });

    findCompletionTime(processes);
    findWaitingTimeAndTurnAroundTime(processes);
    printTable(processes);
    printGanttChart(processes);
    calculateAverages(processes);
}

int main() {
    int choice;
    do {
        cout << "\nScheduling Algorithms Menu:\n";
        cout << "1. First Come First Serve (FCFS)\n";
        cout << "2. Shortest Job First (SJF)\n";
        cout << "3. Round Robin (RR)\n";
        cout << "4. Priority Scheduling\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1: FCFS(); break;
            case 2: SJF(); break;
            case 3: RoundRobin(); break;
            case 4: PriorityScheduling(); break;
            case 5: cout << "Exiting program..." << endl; break;
            default: cout << "Invalid choice! Please try again." << endl;
```

```
        }
    } while (choice != 5);

    return 0;
}
```

____

ASSIGMENT NO.3

Write a program to simulate inter process communication mechanism using pipes and redirection.

```
#include <iostream>
#include <unistd.h>
#include <cstring>
#include <sys/wait.h>
using namespace std;
// Function to convert string to uppercase
string to_uppercase(const string &input) {
string result = input;
for (char &ch : result) ch = toupper(ch);
return result;
}
int main(int argc, char* argv[]) {
if (argc != 2) {
cerr << "Usage: " << argv[0] << " <message_to_child>\n";
return 1;
}
string message = argv[1];
int pipe1[2]; // Parent -> Child
int pipe2[2]; // Child -> Parent
if (pipe(pipe1) == -1 || pipe(pipe2) == -1) {
perror("Pipe creation failed");
return 1;
```

```
}
pid_t pid = fork();
if (pid < 0) {
perror("Fork failed");
return 1;
}
else if (pid == 0) {
// Child process
close(pipe1[1]); // Close write end of pipe1
close(pipe2[0]); // Close read end of pipe2
char buffer[256];
read(pipe1[0], buffer, sizeof(buffer));
close(pipe1[0]);
string received(buffer);
string processed = to_uppercase(received);
// Redirect stdout to pipe2
dup2(pipe2[1], STDOUT_FILENO);
close(pipe2[1]);
cout << processed << endl;
}
else {
// Parent process
close(pipe1[0]); // Close read end of pipe1
close(pipe2[1]); // Close write end of pipe2
write(pipe1[1], message.c_str(), message.length() + 1);
close(pipe1[1]);
char result[256];
read(pipe2[0], result, sizeof(result));
close(pipe2[0]);
wait(NULL); // Wait for child to finish
cout << "Parent received from child: " << result;
```

```
}
return 0;
}
```

____

ASSIGNMENT NO 4

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>
using namespace std;

int balance = 1000;
pthread_mutex_t balance_mutex;

struct Params {
    int id, amount;
    bool deposit, sync;
};

void* reader(void* arg) {
    auto* p = (Params*)arg;
    if (p->sync) pthread_mutex_lock(&balance_mutex);

    cout << (p->sync ? "[SYNC] " : "[UNSYNC] ")
         << "Reader " << p->id << " balance: $" << balance << endl;

    if (p->sync) pthread_mutex_unlock(&balance_mutex);

    usleep(100000);
    return nullptr;
}
```

```
void* writer(void* arg) {
   auto* p = (Params*)arg;
   if (p->sync) pthread_mutex_lock(&balance_mutex);

   string action = p->deposit ? "Deposit" : "Withdraw";

   if (p->deposit) {
      balance += p->amount;
      cout << (p->sync ? "[SYNC] " : "[UNSYNC] ")
          << "Writer " << p->id << " " << action << " $" << p->amount
          << ", new balance: $" << balance << endl;
   } else if (balance >= p->amount) {
      balance -= p->amount;
      cout << (p->sync ? "[SYNC] " : "[UNSYNC] ")
          << "Writer " << p->id << " " << action << " $" << p->amount
          << ", new balance: $" << balance << endl;
   } else {
      cout << (p->sync ? "[SYNC] " : "[UNSYNC] ")
          << "Writer " << p->id << " failed to " << action
          << " $" << p->amount << " (Insufficient funds)" << endl;
   }

   if (p->sync) pthread_mutex_unlock(&balance_mutex);

   usleep(100000);
   return nullptr;
}

void runSimulation(bool sync) {
   pthread_t readers[3], writers[3];
```

```
    Params r_params[3] = {
        {1, 0, false, sync},
        {2, 0, false, sync},
        {3, 0, false, sync}
    };

    Params w_params[3] = {
        {4, 500, true, sync},
        {5, 300, false, sync},
        {6, 400, true, sync}
    };

    for (int i = 0; i < 3; ++i) {
        pthread_create(&readers[i], nullptr, reader, &r_params[i]);
        pthread_create(&writers[i], nullptr, writer, &w_params[i]);
    }

    for (int i = 0; i < 3; ++i) {
        pthread_join(readers[i], nullptr);
        pthread_join(writers[i], nullptr);
    }
}

int main() {
    int choice;
    cout << "1. Without Sync\n2. With Sync\nEnter choice: ";
    cin >> choice;

    if (choice == 2) pthread_mutex_init(&balance_mutex, nullptr);
```

```
    runSimulation(choice == 2);

    if (choice == 2) pthread_mutex_destroy(&balance_mutex);

    cout << "\nSimulation Completed!\n";
    return 0;
}
```

ASSIGNMENT NO 5

Write a program to implement Banker's Algorithm for deadlock avoidance.

```
#include <iostream>
#include <vector>
using namespace std;
void calculateNeed(vector<vector<int>>& max, vector<vector<int>>& allocation,
vector<vector<int>>& need, int n, int m) {
for (int i = 0; i < n; i++)
for (int j = 0; j < m; j++)
need[i][j] = max[i][j] - allocation[i][j];
}
bool isSafe(vector<vector<int>>& allocation, vector<vector<int>>& need, vector<int>& available, int
n, int m) {
vector<bool> finished(n, false);
vector<int> work = available;
vector<int> safeSequence;
cout << "\nNeed Matrix:\n";
for (int i = 0; i < n; i++) {
for (int j = 0; j < m; j++)
cout << need[i][j] << " ";
cout << endl;
}
cout << "\nChecking for Safe Sequence...\n";
```

```
int count = 0;
while (count < n) {
bool found = false;
for (int i = 0; i < n; i++) {
if (!finished[i]) {
bool possible = true;
for (int j = 0; j < m; j++) {
if (need[i][j] > work[j]) {
possible = false;
break;
}
}
if (possible) {
cout << "Process P" << i << " is executing.\n";
for (int j = 0; j < m; j++)
work[j] += allocation[i][j];
safeSequence.push_back(i);
finished[i] = true;
found = true;
count++;
}
}
}
if (!found) {
cout << "\nSystem is in an UNSAFE state. No safe sequence exists.\n";
return false;
}
}
cout << "\nSystem is in a SAFE state.\nSafe Sequence: ";
for (int i = 0; i < safeSequence.size(); i++)
cout << "P" << safeSequence[i] << (i == safeSequence.size() - 1 ? "\n" : " -> ");
```

```
return true;
}
int main() {
int n, m;
cout << "Enter number of processes: ";
cin >> n;
cout << "Enter number of resource types: ";
cin >> m;
vector<vector<int>> allocation(n, vector<int>(m));
vector<vector<int>> max(n, vector<int>(m));
vector<vector<int>> need(n, vector<int>(m));
vector<int> available(m);
cout << "\nEnter Allocation Matrix:\n";
for (int i = 0; i < n; i++) {
cout << "P" << i << ": ";
for (int j = 0; j < m; j++)
cin >> allocation[i][j];
}
cout << "\nEnter Maximum Matrix (must be >= allocation):\n";
for (int i = 0; i < n; i++) {
while (true) {
cout << "P" << i << ": ";
bool valid = true;
for (int j = 0; j < m; j++) {
cin >> max[i][j];
if (max[i][j] < allocation[i][j]) {
valid = false;
}
}
if (!valid) {
cout << "Error: Maximum values must be >= Allocation values for P" << i << ". Re-enter row.\n";
```

```
} else {
break;
}
}
}
cout << "\nEnter Available Resources:\n";
for (int j = 0; j < m; j++)
cin >> available[j];
calculateNeed(max, allocation, need, n, m);
isSafe(allocation, need, available, n, m);
return 0;
}
```

```
#include <bits/stdc++.h>
using namespace std;

// Function to input system data: available resources, max need, allocation, and need matrices
void inputData(int numProcesses, int numResources, vector<int> &available, vector<vector<int>>
&maxNeed, vector<vector<int>> &allocated, vector<vector<int>> &need) {
  cout << "\nEnter the Available Resources:\n";
  for (int i = 0; i < numResources; i++) {
    cin >> available[i];
    if (available[i] < 0) throw invalid_argument("Available resources cannot be negative.");
  }

  cout << "\nEnter the Maximum Need matrix:\n";
  for (int i = 0; i < numProcesses; i++) {
    for (int j = 0; j < numResources; j++) {
      cin >> maxNeed[i][j];
      if (maxNeed[i][j] < 0) throw invalid_argument("Maximum need cannot be negative.");
    }
```

```
    }

    cout << "\nEnter the Allocation matrix:\n";
    for (int i = 0; i < numProcesses; i++) {
        for (int j = 0; j < numResources; j++) {
            cin >> allocated[i][j];
            need[i][j] = maxNeed[i][j] - allocated[i][j]; // Calculate need
        }
    }
}

// Function to display the current system state
void displayState(int numProcesses, int numResources, const vector<int> &available, const vector<vector<int>> &maxNeed, const vector<vector<int>> &allocated, const vector<vector<int>> &need) {
    cout << "\nCurrent System State:";

    // Display available resources
    cout << "\nAvailable Resources: ";
    for (int res : available) cout << res << " ";

    // Display maximum need matrix
    cout << "\nMaximum Need Matrix:\n";
    for (int i = 0; i < numProcesses; i++) {
        cout << "P" << i << ": ";
        for (int val : maxNeed[i]) cout << val << " ";
        cout << endl;
    }

    // Display allocation matrix
    cout << "\nAllocation Matrix:\n";
    for (int i = 0; i < numProcesses; i++) {
```

```
        cout << "P" << i << ": ";
        for (int val : allocated[i]) cout << val << " ";
        cout << endl;
    }

    // Display need matrix
    cout << "\nNeed Matrix:\n";
    for (int i = 0; i < numProcesses; i++) {
        cout << "P" << i << ": ";
        for (int val : need[i]) cout << val << " ";
        cout << endl;
    }
}

// Safety Algorithm to check if the system is in a safe state
bool isSafe(int numProcesses, int numResources, const vector<int> &available, const vector<vector<int>> &allocated, const vector<vector<int>> &need, vector<int> &safeSequence) {
    vector<int> work = available;
    vector<bool> finish(numProcesses, false);
    int count = 0;

    cout << "\nSafety Algorithm Execution:\n";
    while (count < numProcesses) {
        bool found = false;
        for (int i = 0; i < numProcesses; i++) {
            if (!finish[i]) {
                bool canAllocate = true;
                cout << "\n\nChecking Process P" << i << ":\n";
                cout << "Current Work Vector: ";
                for (int w : work) cout << w << " ";
                cout << "\nNeed Vector for P" << i << ": ";
```

```
        for (int n : need[i]) cout << n << " ";

        for (int j = 0; j < numResources; j++) {
          if (need[i][j] > work[j]) {
            canAllocate = false;
            cout << "\nCannot allocate to P" << i << " as Need > Work for resource " << j;
            break;
          }
        }

        if (canAllocate) {
          cout << "\nAllocating resources to P" << i;
          for (int j = 0; j < numResources; j++) {
            work[j] += allocated[i][j];
          }
          finish[i] = true;
          safeSequence.push_back(i);
          found = true;
          count++;

          cout << "\nUpdated Work Vector: ";
          for (int w : work) cout << w << " ";
          cout << "\nSafe Sequence so far: ";
          for (int k : safeSequence) cout << "P" << k << " ";
        }
      }
    }

    if (!found) {
      cout << "\n\nNO SAFE SEQUENCE FOUND - System is in UNSAFE state!";
      return false;
```

```
        }
    }

    cout << "\n\nSAFE SEQUENCE FOUND: ";
    for (int i : safeSequence) cout << "P" << i << " ";
    cout << "\nSystem is in SAFE state!";
    return true;
}

// Resource Request Algorithm to handle resource requests
bool requestResources(int numProcesses, int numResources, int process, vector<int> &available,
vector<vector<int>> &allocated, vector<vector<int>> &need, vector<int> &request) {
    if (process < 0 || process >= numProcesses) {
        throw out_of_range("Invalid process number.");
    }

    cout << "\nRequesting resources for Process P" << process << ":\n";
    cout << "Requested Vector: ";
    for (int r : request) cout << r << " ";

    // Check if request exceeds the process's maximum need
    for (int i = 0; i < numResources; i++) {
        if (request[i] > need[process][i]) {
            cout << "\nError: Request exceeds maximum need for resource " << i;
            return false;
        }
    }

    // Check if request exceeds the available resources
    for (int i = 0; i < numResources; i++) {
        if (request[i] > available[i]) {
```

```
            cout << "\nError: Request exceeds available resources for resource " << i;
            return false;
        }
    }

    // Save current state for potential rollback
    vector<int> savedAvailable = available;
    vector<vector<int>> savedAllocated = allocated;
    vector<vector<int>> savedNeed = need;

    // Allocate requested resources
    for (int i = 0; i < numResources; i++) {
        available[i] -= request[i];
        allocated[process][i] += request[i];
        need[process][i] -= request[i];
    }

    cout << "\n\nAfter Allocation:\n";
    displayState(numProcesses, numResources, available, allocated, allocated, need);

    // Check if the system remains in a safe state
    vector<int> safeSequence;
    if (isSafe(numProcesses, numResources, available, allocated, need, safeSequence)) {
        cout << "\nRequest can be granted immediately!";
        return true;
    } else {
        // Rollback if not safe
        available = savedAvailable;
        allocated = savedAllocated;
        need = savedNeed;
        cout << "\n\nRequest cannot be granted - restoring previous state.";
```

```
        return false;
    }
}

// Recursive function to find all safe sequences
void findAllSafeSequences(int numProcesses, int numResources, vector<int> &available, vector<vector<int>> &allocated, vector<vector<int>> &need, vector<bool> &finish, vector<int> &currentSequence, vector<vector<int>> &allSequences) {
    bool found = false;


    for (int i = 0; i < numProcesses; i++) {
        if (!finish[i]) {
            bool canAllocate = true;
            for (int j = 0; j < numResources; j++) {
                if (need[i][j] > available[j]) {
                    canAllocate = false;
                    break;
                }
            }


            if (canAllocate) {
                for (int j = 0; j < numResources; j++) {
                    available[j] += allocated[i][j];
                }


                finish[i] = true;
                currentSequence.push_back(i);


                // Recurse
                findAllSafeSequences(numProcesses, numResources, available, allocated, need, finish, currentSequence, allSequences);
```

```
            // Backtrack
            for (int j = 0; j < numResources; j++) {
                available[j] -= allocated[i][j];
            }

            finish[i] = false;
            currentSequence.pop_back();
            found = true;
        }
    }
}

if (!found && currentSequence.size() == numProcesses) {
    allSequences.push_back(currentSequence);
}
}

void displayAllSafeSequences(int numProcesses, int numResources, vector<int> available,
vector<vector<int>> allocated, vector<vector<int>> need) {
    vector<bool> finish(numProcesses, false);
    vector<int> currentSequence;
    vector<vector<int>> allSequences;

    findAllSafeSequences(numProcesses, numResources, available, allocated, need, finish,
currentSequence, allSequences);

    if (allSequences.empty()) {
        cout << "\nNo safe sequences found. System is in UNSAFE state.\n";
    } else {
        cout << "\nAll Possible Safe Sequences:\n";
        for (const auto &seq : allSequences) {
            for (int pid : seq) {
```

```
            cout << "P" << pid << " ";
          }
          cout << "\n";
        }
    }
}

// Menu function to interact with the user
void menu(int numProcesses, int numResources, vector<int> &available, vector<vector<int>>
&maxNeed, vector<vector<int>> &allocated, vector<vector<int>> &need) {
    int choice;
    do {
        cout << "\nMENU:\n1. Display Current System State\n2. Check System Safety\n3. Request
Resources\n4. Show All Safe Sequences\n5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
          case 1:
            displayState(numProcesses, numResources, available, maxNeed, allocated, need);
            break;
          case 2: {
            vector<int> safeSequence;
            if (isSafe(numProcesses, numResources, available, allocated, need, safeSequence)) {
              cout << "\nSystem is in SAFE state!";
            } else {
              cout << "\nSystem is in UNSAFE state!";
            }
            break;
          }
          case 3: {
            int process;
            cout << "Enter process number making the request: ";
```

```
            cin >> process;
            if (process < 0 || process >= numProcesses) {
                throw out_of_range("Invalid process number.");
            }

            vector<int> request(numResources);
            cout << "Enter resource request vector:\n";
            for (int i = 0; i < numResources; i++) {
                cin >> request[i];
                if (request[i] < 0) throw invalid_argument("Resource request cannot be negative.");
            }
            requestResources(numProcesses, numResources, process, available, allocated, need,
request);
            break;
        }
        case 4:
            displayAllSafeSequences(numProcesses, numResources, available, allocated, need);
            break;
        case 5:
            cout << "\nExiting the program.";
            break;
        default:
            cout << "\nInvalid choice. Please try again.";
        }
    } while (choice != 5);
}

// Main function
int main() {
    try {
        int numProcesses, numResources;
```

```
        cout << "Enter number of processes: ";
        cin >> numProcesses;
        if (numProcesses <= 0) {
            throw invalid_argument("Number of processes must be positive.");
        }

        cout << "Enter number of resources: ";
        cin >> numResources;
        if (numResources <= 0) {
            throw invalid_argument("Number of resources must be positive.");
        }

        vector<int> available(numResources);
        vector<vector<int>> maxNeed(numProcesses, vector<int>(numResources));
        vector<vector<int>> allocated(numProcesses, vector<int>(numResources));
        vector<vector<int>> need(numProcesses, vector<int>(numResources));

        inputData(numProcesses, numResources, available, maxNeed, allocated, need);

        for (int i = 0; i < numProcesses; i++) {
            for (int j = 0; j < numResources; j++) {
                if (need[i][j] < 0) {
                    throw runtime_error("Error: Need[" + to_string(i) + "][" + to_string(j) + "] is negative.
(Allocation > Max Need)");
                }
            }
        }

        menu(numProcesses, numResources, available, maxNeed, allocated, need);

    } catch (const exception &e) {
```

```
    cerr << "\nError: " << e.what();
  }

  return 0;
}
```

____

ASSIGNMENT NO 6.Problem Statement : Write a program to simulate memory allocation techniques: First Fit, Best Fit, Next Fit and Worst Fit.

```
#include<stdio.h>
#include<stdlib.h>
int M;
int N;
int Holes[10];
int Process[10];
void FirstFit() {
int CopyHoles[10];
int CopyProcess[10];
for (int i = 0; i < M; i++) {
CopyHoles[i] = Holes[i];
}
for (int i = 0; i < N; i++) {
CopyProcess[i] = Process[i];
}
int index = 0;
for(int i = 0; i < N; i++) {
int found = 0;
for(int j = 0; j < M; j++) {
if(CopyHoles[j] >= CopyProcess[i]) {
found = 1;
index = j;
break;
```

```
}
}
if(found == 0) {
printf("Process %d cannot be allocated\n", i+1);
break;
}
else {
printf("Process %d allocated to hole %d || Process Size = %d || Hole Size = %d || Updated Hole Size = %d\n", i+1, index+1, CopyProcess[i], CopyHoles[index], CopyHoles[index] - CopyProcess[i]);
CopyHoles[index] -= CopyProcess[i];
}
}
}
void BestFit() {
int CopyHoles[10];
int CopyProcess[10];
for (int i = 0; i < M; i++) {
CopyHoles[i] = Holes[i];
}
for (int i = 0; i < N; i++) {
CopyProcess[i] = Process[i];
}
for(int i = 0; i < N; i++) {
int index = -1;
int small = 999;
for(int j = 0; j < M; j++) {
if(CopyHoles[j] >= CopyProcess[i] && CopyHoles[j] < small) {
small = CopyHoles[j];
index = j;
}
}
```

```
if(index == -1) {

printf("Process %d cannot be allocated\n", i+1);

break;

}

else {

printf("Process %d allocated to hole %d || Process Size = %d || Hole Size = %d || Updated Hole Size = %d\n", i+1, index+1, CopyProcess[i], CopyHoles[index], CopyHoles[index] - CopyProcess[i]);

CopyHoles[index] -= CopyProcess[i];

}

}

}

void WorstFit() {

int CopyHoles[10];

int CopyProcess[10];

for (int i = 0; i < M; i++) {

CopyHoles[i] = Holes[i];

}

for (int i = 0; i < N; i++) {

CopyProcess[i] = Process[i];

}

for(int i = 0; i < N; i++) {

int index = -1;

int large = -999;

for(int j = 0; j < M; j++) {

if(CopyHoles[j] >= CopyProcess[i] && CopyHoles[j] > large) {

large = CopyHoles[j];

index = j;

}

}

if(index == -1) {

printf("Process %d cannot be allocated\n", i+1);
```

```
break;
}
else {
printf("Process %d allocated to hole %d || Process Size = %d || Hole Size = %d || Updated Hole Size = %d\n", i+1, index+1, CopyProcess[i], CopyHoles[index], CopyHoles[index] - CopyProcess[i]);
CopyHoles[index] -= CopyProcess[i];
}
}
}
void NextFit() {
int CopyHoles[10];
int CopyProcess[10];
for (int i = 0; i < M; i++) {
CopyHoles[i] = Holes[i];
}
for (int i = 0; i < N; i++) {
CopyProcess[i] = Process[i];
}
int index = 0;
for(int i = 0; i < N; i++) {
int found = 0;
int count = 0;
for(int j = index; count < M; j = (j + 1) % M) {
if(CopyHoles[j] >= CopyProcess[i]) {
index = j;
found = 1;
break;
}
count++;
}
if(found == 0) {
```

```
printf("Process %d cannot be allocated\n", i+1);
break;
}
else {
printf("Process %d allocated to hole %d || Process Size = %d || Hole Size = %d || Updated Hole Size = %d\n", i+1, index+1, CopyProcess[i], CopyHoles[index], CopyHoles[index] - CopyProcess[i]);
CopyHoles[index] -= CopyProcess[i];
}
}
}
int main() {
int choice;
printf("Enter the No. of Holes(Max = 10): ");
scanf("%d", &M);
printf("Enter the No. of Processes(Max = 10): ");
scanf("%d", &N);
printf("Enter the Hole Size one by one:\n");
for(int i = 0; i < M; i++) {
scanf("%d", &Holes[i]);
}
printf("Enter the Process Size one by one:\n");
for(int i = 0; i < N; i++) {
scanf("%d", &Process[i]);
}
do {
printf("\n**********Menu**********\n");
printf("1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Next Fit\n5. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch(choice) {
case 1: {
```

```c
printf("\nFirst Fit Allocation\n");
FirstFit();
printf("--------------------------------------------------------------\n");
break; }
case 2: {
printf("\nBest Fit Allocation\n");
BestFit();
printf("--------------------------------------------------------------\n");
break; }
case 3: {
printf("\nWorst Fit Allocation\n");
WorstFit();
printf("--------------------------------------------------------------\n");
break; }
case 4: {
printf("\nNext Fit Allocation\n");
NextFit();
printf("--------------------------------------------------------------\n");
break; }
case 5: {
printf("Exiting...\n");
break; }
default: {
printf("Invalid choice. Please try again.\n");
break; }
}
} while(choice != 5);
return 0;
}
```

ASSIGNMENT NO. 7

Problem Statement : Write a to implement paging replacement algorithms :

a) FCFS

b) Least Recently Used (LRU)

c) Optimal algorithm

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

#define MAX_FRAMES 10
#define MAX_PAGES 50

int n, Size;

int isHit(int Frame[], int page) {
   for (int i = 0; i < Size; i++) {
      if (Frame[i] == page)
         return 1;
   }
   return 0;
}

void FCFS(int PageSeq[]) {
   printf("\n--- FCFS Page Replacement ---\n");
   int Frame[MAX_FRAMES];
   int front = 0, faults = 0;

   for (int i = 0; i < Size; i++) {
      Frame[i] = -1;
   }

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

```c
        if (!isHit(Frame, PageSeq[i])) {
            faults++;
            Frame[front] = PageSeq[i];
            front = (front + 1) % Size;
        }

        printf("Page %d: ", PageSeq[i]);
        for (int j = 0; j < Size; j++) {
            if (Frame[j] == -1)
                printf("- ");
            else
                printf("%d ", Frame[j]);
        }
        printf("\n");
    }

    printf("Total Page Faults (FCFS): %d\n", faults);
    printf("Total Page Hits (FCFS): %d\n", n - faults);
    printf("Hit Ratio: %.2f%%\n", ((float)(n - faults) / n) * 100);
}

void LRU(int PageSeq[]) {
    printf("\n--- LRU Page Replacement ---\n");
    int Frame[MAX_FRAMES];
    int count[MAX_FRAMES] = {0};
    int Time = 0, faults = 0;

    for (int i = 0; i < Size; i++) {
        Frame[i] = -1;
    }
```

```
for (int i = 0; i < n; i++) {
    Time++;
    int hit = 0;

    for (int j = 0; j < Size; j++) {
        if (Frame[j] == PageSeq[i]) {
            hit = 1;
            count[j] = Time;
            break;
        }
    }

    if (!hit) {
        faults++;
        int min = INT_MAX, replace_index = -1;

        for (int j = 0; j < Size; j++) {
            if (Frame[j] == -1) {
                replace_index = j;
                break;
            } else if (count[j] < min) {
                min = count[j];
                replace_index = j;
            }
        }

        Frame[replace_index] = PageSeq[i];
        count[replace_index] = Time;
    }

    printf("Page %d: ", PageSeq[i]);
```

```
        for (int j = 0; j < Size; j++) {
            if (Frame[j] == -1)
                printf("- ");
            else
                printf("%d ", Frame[j]);
        }
        printf("\n");
    }

    printf("Total Page Faults (LRU): %d\n", faults);
    printf("Total Page Hits (LRU): %d\n", n - faults);
    printf("Hit Ratio: %.2f%%\n", ((float)(n - faults) / n) * 100);
}

int predict(int PageSeq[], int Frame[], int index) {
    int Far = -1, Found = -1;

    for (int i = 0; i < Size; i++) {
        int j;
        for (j = index; j < n; j++) {
            if (Frame[i] == PageSeq[j]) {
                if (j > Far) {
                    Far = j;
                    Found = i;
                }
                break;
            }
        }
        if (j == n)
            return i;
    }
```

```
    return (Found == -1) ? 0 : Found;
}

void Optimal(int PageSeq[]) {
    printf("\n--- Optimal Page Replacement ---\n");
    int Frame[MAX_FRAMES];
    int faults = 0;

    for (int i = 0; i < Size; i++) {
        Frame[i] = -1;
    }

    for (int i = 0; i < n; i++) {
        if (!isHit(Frame, PageSeq[i])) {
            faults++;
            int j;
            for (j = 0; j < Size; j++) {
                if (Frame[j] == -1) {
                    Frame[j] = PageSeq[i];
                    break;
                }
            }

            if (j == Size) {
                int idx = predict(PageSeq, Frame, i + 1);
                Frame[idx] = PageSeq[i];
            }
        }

        printf("Page %d: ", PageSeq[i]);
        for (int j = 0; j < Size; j++) {
```

```
            if (Frame[j] == -1)
                printf("- ");
            else
                printf("%d ", Frame[j]);
        }
        printf("\n");
    }

    printf("Total Page Faults (Optimal): %d\n", faults);
    printf("Total Page Hits (Optimal): %d\n", n - faults);
    printf("Hit Ratio: %.2f%%\n", ((float)(n - faults) / n) * 100);
}

int main() {
    int PageSeq[MAX_PAGES], choice;

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

    printf("Enter The Page Reference String:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &PageSeq[i]);

    printf("Enter The Number of Frames: ");
    scanf("%d", &Size);

    do {
        printf("\nChoose Paging Algorithm:\n");
        printf("1. FCFS\n");
        printf("2. LRU\n");
        printf("3. Optimal\n");
```

```
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
          case 1:
            FCFS(PageSeq);
            break;
          case 2:
            LRU(PageSeq);
            break;
          case 3:
            Optimal(PageSeq);
            break;
          case 4:
            printf("Exiting program.\n");
            break;
          default:
            printf("Invalid choice! Try again.\n");
        }
    } while (choice != 4);

    return 0;
}
```

ASSIGNMENT NO.8

Problem Statement : Write a program to implement disk scheduling algorithms FIFO, SSTF, SCAN, C-SCAN.

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

```c
#include <math.h>

int n, head, DiskSize;

void FIFO(int Arr[], int Head) {
    int TotalHM = 0;
    printf("\nFIFO Order: %d", Head);
    for (int i = 0; i < n; i++) {
        printf(" -> %d", Arr[i]);
        TotalHM += abs(Arr[i] - Head);
        Head = Arr[i];
    }
    printf("\nTotal Head Movement (FIFO): %d\n", TotalHM);
}

void SSTF(int Arr[], int Head) {
    int TotalHM = 0;
    int Finish[n];
    for (int i = 0; i < n; i++)
        Finish[i] = 0;

    printf("\nSSTF Order: %d", Head);
    for (int i = 0; i < n; i++) {
        int min = 1e9, index = -1;
        for (int j = 0; j < n; j++) {
            if (!Finish[j] && abs(Arr[j] - Head) < min) {
                min = abs(Arr[j] - Head);
                index = j;
            }
        }
        Finish[index] = 1;
```

```
        printf(" -> %d", Arr[index]);
        TotalHM += abs(Arr[index] - Head);
        Head = Arr[index];
    }
    printf("\nTotal Head Movement (SSTF): %d\n", TotalHM);
}

void SCAN(int Arr[], int Head) {
    int TotalHM = 0, Dir;
    printf("Enter Direction (Right = 1 / Left = 0): ");
    scanf("%d", &Dir);

    int Temp[n + 1];
    for (int i = 0; i < n; i++)
        Temp[i] = Arr[i];
    Temp[n] = Head;

    // Bubble sort
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j < n - i; j++) {
            if (Temp[j] > Temp[j + 1]) {
                int t = Temp[j];
                Temp[j] = Temp[j + 1];
                Temp[j + 1] = t;
            }
        }
    }

    int pos = 0;
    for (int i = 0; i <= n; i++) {
        if (Temp[i] == Head) {
```

```
            pos = i;
            break;
        }
    }

    printf("\nSCAN Order: %d", Head);
    if (Dir == 1) { // Right
        for (int i = pos + 1; i <= n; i++) {
            printf(" -> %d", Temp[i]);
            TotalHM += abs(Temp[i] - Head);
            Head = Temp[i];
        }
        if (Head != DiskSize - 1) {
            printf(" -> %d", DiskSize - 1);
            TotalHM += abs((DiskSize - 1) - Head);
            Head = DiskSize - 1;
        }
        for (int i = pos - 1; i >= 0; i--) {
            printf(" -> %d", Temp[i]);
            TotalHM += abs(Temp[i] - Head);
            Head = Temp[i];
        }
    } else { // Left
        for (int i = pos - 1; i >= 0; i--) {
            printf(" -> %d", Temp[i]);
            TotalHM += abs(Temp[i] - Head);
            Head = Temp[i];
        }
        if (Head != 0) {
            printf(" -> 0");
            TotalHM += abs(Head - 0);
```

```c
            Head = 0;
        }
        for (int i = pos + 1; i <= n; i++) {
            printf(" -> %d", Temp[i]);
            TotalHM += abs(Temp[i] - Head);
            Head = Temp[i];
        }
    }

    printf("\nTotal Head Movement (SCAN): %d\n", TotalHM);
}

void CSCAN(int Arr[], int Head) {
    int TotalHM = 0, Dir;
    printf("Enter Direction (Right = 1 / Left = 0): ");
    scanf("%d", &Dir);

    int Temp[n + 1];
    for (int i = 0; i < n; i++)
        Temp[i] = Arr[i];
    Temp[n] = Head;

    // Bubble sort
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j < n - i; j++) {
            if (Temp[j] > Temp[j + 1]) {
                int t = Temp[j];
                Temp[j] = Temp[j + 1];
                Temp[j + 1] = t;
            }
        }
```

```
}

int pos = 0;
for (int i = 0; i <= n; i++) {
    if (Temp[i] == Head) {
        pos = i;
        break;
    }
}

printf("\nC-SCAN Order: %d", Head);
if (Dir == 1) { // Right
    for (int i = pos + 1; i <= n; i++) {
        printf(" -> %d", Temp[i]);
        TotalHM += abs(Temp[i] - Head);
        Head = Temp[i];
    }
    if (Head != DiskSize - 1) {
        printf(" -> %d", DiskSize - 1);
        TotalHM += abs(DiskSize - 1 - Head);
        Head = DiskSize - 1;
    }
    printf(" -> 0");
    TotalHM += DiskSize - 1;
    Head = 0;
    for (int i = 0; i < pos; i++) {
        printf(" -> %d", Temp[i]);
        TotalHM += abs(Temp[i] - Head);
        Head = Temp[i];
    }
} else { // Left
```

```
        for (int i = pos - 1; i >= 0; i--) {
            printf(" -> %d", Temp[i]);
            TotalHM += abs(Temp[i] - Head);
            Head = Temp[i];
        }
        if (Head != 0) {
            printf(" -> 0");
            TotalHM += Head;
            Head = 0;
        }
        printf(" -> %d", DiskSize - 1);
        TotalHM += DiskSize - 1;
        Head = DiskSize - 1;
        for (int i = n; i > pos; i--) {
            printf(" -> %d", Temp[i]);
            TotalHM += abs(Temp[i] - Head);
            Head = Temp[i];
        }
    }

    printf("\nTotal Head Movement (C-SCAN): %d\n", TotalHM);
}

int main() {
    int ch;
    printf("Enter Number of Requests: ");
    scanf("%d", &n);

    int Arr[n];
    printf("Enter the Request Sequence:\n");
    for (int i = 0; i < n; i++)
```

```
    scanf("%d", &Arr[i]);

printf("Enter Initial Head Position: ");
scanf("%d", &head);

printf("Enter Total Disk Size: ");
scanf("%d", &DiskSize);

do {
    printf("\n******** MENU ********\n");
    printf("1. FIFO\n");
    printf("2. SSTF\n");
    printf("3. SCAN\n");
    printf("4. C-SCAN\n");
    printf("5. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &ch);

    switch (ch) {
        case 1:
            FIFO(Arr, head);
            break;
        case 2:
            SSTF(Arr, head);
            break;
        case 3:
            SCAN(Arr, head);
            break;
        case 4:
            CSCAN(Arr, head);
            break;
```

```
        case 5:
            printf("Exiting program.\n");
            break;
        default:
            printf("Invalid choice! Try again.\n");
        }
    } while (ch != 5);

    return 0;
}
```
