1_simulate.cpp

```
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include <cstdlib>
using namespace std;
void simulate_cp() {
    char source[100], destination[100];
    cout << "Enter source file:";
    cin >> source;
    cout << "Enter destination file:";
    cin >> destination;
    pid_t pid = fork();
    if (pid == 0) {
        cout << "Child (cp) PID: " << getpid() << endl;
        execl("/bin/cp", "cp", source, destination, NULL);
        perror("execl failed");
        exit(1);
    } else {
        wait(NULL);
        cout << "Parent (cp): Copy operation complete\n";
    }
}
void simulate_grep() {
    char word[100], file[100];
    cout << "Enter word to search: ";
    cin >> word;
    cout << "Enter file to search in: ";
    cin >> file;
    pid_t pid = fork();
    if (pid == 0) {
        cout << "Child (grep) PID: " << getpid() << endl;
        // FIXED: use execlp so it finds grep in PATH
        execlp("grep", "grep", word, file, NULL);
        // Only runs if exec fails
        perror("execlp failed");
        exit(1);
    } else {
        wait(NULL);
        cout << "Parent (grep): Grep operation complete\n";
    }
}
int main() {
    int choice;
    do {
        cout << "\nLinux Command Simulation Menu:\n";
        cout << "1. Simulate cp command\n";
        cout << "2. Simulate grep command\n";
        cout << "3. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1: simulate_cp(); break;
            case 2: simulate_grep(); break;
            case 3: cout << "Exiting...\n"; exit(0);
            default: cout << "Invalid choice!\n";
        }
    } while (choice != 3);
    return 0;
}
```

2_cpu_scheduling_algorithms.cpp

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
using namespace std;
struct Process {
    int pid, at, bt, priority, wt, tat, ct, rt, remaining_bt;
    int temp_priority;
};
void sortByArrival(vector<Process>& p, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].at > p[j + 1].at) {
                swap(p[j], p[j + 1]);
            }
        }
    }
}
void fcfs(vector<Process>& p, int n) {
    sortByArrival(p, n);
    int t = 0;
    for (int i = 0; i < n; i++) {
        if (t < p[i].at)
            t = p[i].at;
        p[i].ct = t + p[i].bt;
        t = p[i].ct;
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
    }
}
void sjf(vector<Process>& p, int n) {
    sortByArrival(p, n);
    vector<bool> completed(n, false);
    int t = 0, completedCount = 0;
    while (completedCount < n) {
        int minIndex = -1, minBT = 999;
        for (int i = 0; i < n; i++) {
            if (!completed[i] && p[i].at <= t && p[i].bt < minBT) {
                minBT = p[i].bt;
                minIndex = i;
            }
        }
        if (minIndex == -1) {
            t++;
        } else {
            t += p[minIndex].bt;
            p[minIndex].ct = t;
            p[minIndex].tat = p[minIndex].ct - p[minIndex].at;
            p[minIndex].wt = p[minIndex].tat - p[minIndex].bt;
            completed[minIndex] = true;
            completedCount++;
        }
    }
}
void roundRobin(vector<Process>& p, int n, int quantum) {
    int t = 0, completedCount = 0;
    vector<bool> completed(n, false);
    vector<bool> started(n, false);
    vector<bool> in_q(n, false);
    queue<int> q;
    for (int i = 0; i < n; i++) {
        p[i].remaining_bt = p[i].bt;
    }
    while (completedCount < n) {
        for (int i = 0; i < n; i++) {
            if (!in_q[i] && !completed[i] && p[i].at <= t) {
                q.push(i);
                in_q[i] = true;
            }
        }
        if (q.empty()) {
            t++;
            continue;
        }
        int idx = q.front();
        q.pop();
        if (!started[idx]) {
            p[idx].rt = t - p[idx].at;
            started[idx] = true;
        }
        int exec = min(quantum, p[idx].remaining_bt);
        p[idx].remaining_bt -= exec;
        t += exec;
        for (int i = 0; i < n; i++)
            if (p[i].at <= t && !in_q[i] && !completed[i]) {
                q.push(i);
                in_q[i] = true;
            }
        if (p[idx].remaining_bt > 0) q.push(idx);
        else {
            p[idx].ct = t;
            p[idx].tat = p[idx].ct - p[idx].at;
            p[idx].wt = p[idx].tat - p[idx].bt;
            completed[idx] = true;
            completedCount++;
        }
    }
}
// Non-Preemptive Priority Scheduling Algorithm
void priorityScheduling(vector<Process>& p, int n) {
    sortByArrival(p, n);
    vector<bool> completed(n, false);
    int t = 0, completedCount = 0;
    while (completedCount < n) {
        int minIndex = -1, minPriority = 999;
        for (int i = 0; i < n; i++) {
            if (!completed[i] && p[i].at <= t && p[i].priority < minPriority) {
                minPriority = p[i].priority;
                minIndex = i;
            }
        }
        if (minIndex == -1) {
            t++;
        } else {
            t += p[minIndex].bt;
            p[minIndex].ct = t;
            p[minIndex].tat = p[minIndex].ct - p[minIndex].at;
            p[minIndex].wt = p[minIndex].tat - p[minIndex].bt;
            completed[minIndex] = true;
            completedCount++;
        }
    }
}
// Preemptive Priority Scheduling Algorithm
void preemptivePriority(vector<Process>& p, int n) {
    sortByArrival(p, n);
    int time = 0, completed = 0;
    for (int i = 0; i < n; i++) {
        p[i].remaining_bt = p[i].bt;
        p[i].temp_priority = p[i].priority;
    }
    const int MIN = -9999;
    while (completed < n) {
        int maxPriority = MIN;
        int idx = -1;
        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].remaining_bt > 0 && p[i].temp_priority > maxPriority) {
                maxPriority = p[i].temp_priority;
                idx = i;
            }
        }
        if (idx != -1) {
            p[idx].remaining_bt--;
            time++;
            if (p[idx].remaining_bt == 0) {
                p[idx].ct = time;
                p[idx].tat = p[idx].ct - p[idx].at;
                p[idx].wt = p[idx].tat - p[idx].bt;
                p[idx].temp_priority = MIN;
                completed++;
            }
        } else {
            time++;
        }
    }
}
void displayResults(const vector<Process>& p, int n) {
    float totalWT = 0, totalTAT = 0;
    cout << "\nPID\tAT\tBT\tPriority\tWT\tTAT\tCT\n";
    for (int i = 0; i < n; i++) {
        totalWT += p[i].wt;
        totalTAT += p[i].tat;
        cout << p[i].pid << "\t" << p[i].at << "\t" << p[i].bt << "\t" <<
        p[i].priority << "\t\t" << p[i].wt << "\t" << p[i].tat << "\t" << p[i].ct << endl;
    }
    cout << fixed << setprecision(2);
    cout << "\nAverage Waiting Time: " << totalWT / n << endl;
    cout << "Average Turnaround Time: " << totalTAT / n << endl;
}
int main() {
    int n, choice, quantum;
    cout << "Enter number of processes: ";
    cin >> n;
    vector<Process> p(n);
    cout << "Enter process details (AT BT Priority):\n";
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        cout << "Process " << p[i].pid << ": ";
        cin >> p[i].at >> p[i].bt >> p[i].priority;
    }
    do {
        cout << "\nChoose Scheduling Algorithm:\n";
        cout << "1. FCFS\n2. SJF\n3. Round Robin\n4. Priority Scheduling (Non-Preemptive)\n5.
Preemptive Priority Scheduling\n6. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                fcfs(p, n);
                displayResults(p, n);
                break;
            case 2:
                sjf(p, n);
                displayResults(p, n);
                break;
            case 3:
                cout << "Enter Time Quantum: ";
                cin >> quantum;
                roundRobin(p, n, quantum);
                displayResults(p, n);
                break;
            case 4:
                priorityScheduling(p, n);
                displayResults(p, n);
                break;
            case 5:
                preemptivePriority(p, n);
                displayResults(p, n);
                break;
            case 6:
                cout << "Exiting the program...\n";
                break;
            default:
                cout << "Invalid choice! Please enter a valid option.\n";
        }
    } while (choice != 6);
    return 0;
}
// Sample Input:
// PID    AT     BT     Priority     WT     TAT    CT
// 1      0      5      10           7      12     12
// 2      1      4      20           3      7      8
// 3      2      2      30           0      2      4
// 4      4      1      40           0      1      5
// remaing fcfs example solved ok jay
// 1. FCFS
// Average Waiting Time: 9.60
// Average Turnaround Time: 13.40
// 2. SJF
// Average Waiting Time: 3.20
// Average Turnaround Time: 7.00
// 3. Round Robin
//tq=1
// Average Waiting Time: 5.40
// Average Turnaround Time: 9.20
// 4. Priority Scheduling (Non-Preemptive)
// Average Waiting Time: 8.20
// Average Turnaround Time: 12.00
// 5. Preemptive Priority Scheduling
// Average Waiting Time: 8.60
// Average Turnaround Time: 12.40
```

3_pipe.cpp

```
#include <iostream>
#include <unistd.h>
#include <cstring>
using namespace std;
int main() {
    int pipefd[2];
    char buffer[100];
    pipe(pipefd); // create pipe
    pid_t pid = fork(); // create child process
    if (pid == 0) {
        // Child process
        close(pipefd[0]); // Close reading end
        const char* msg = "Hello from Child!";
        write(pipefd[1], msg, strlen(msg));
        close(pipefd[1]); // Done writing
    } else {
        // Parent process
        close(pipefd[1]); // Close writing end
        read(pipefd[0], buffer, sizeof(buffer));
        cout << "Parent received: " << buffer << endl;
        close(pipefd[0]); // Done reading
    }
    return 0;
}
```

3_pipe_two_way_communication.cpp

```
#include <iostream>
#include <unistd.h>
#include <cstring>
#include <sys/types.h>
using namespace std;
int main() {
    int pipefds1[2], pipefds2[2];
    pid_t pid;
    char pipe1writemessage[] = "Hi";
    char pipe2writemessage[] = "Hello";
    char readmessage[100];
    // Create first pipe
    if (pipe(pipefds1) == -1) {
        cerr << "Unable to create pipe 1" << endl;
        return 1;
    }
    // Create second pipe
    if (pipe(pipefds2) == -1) {
        cerr << "Unable to create pipe 2" << endl;
        return 1;
    }
    pid = fork();
    if (pid < 0) {
        cerr << "Fork failed" << endl;
        return 1;
    }
    if (pid > 0) {
        // ----- Parent Process -----
        close(pipefds1[0]); // Close read end of pipe1
        close(pipefds2[1]); // Close write end of pipe2
        cout << "In Parent: Writing to pipe 1 – Message is '" << pipe1writemessage << "'" << endl;
        write(pipefds1[1], pipe1writemessage, strlen(pipe1writemessage) + 1);
        read(pipefds2[0], readmessage, sizeof(readmessage));
        cout << "In Parent: Reading from pipe 2 – Message is '" << readmessage << "'" << endl;
        close(pipefds1[1]);
        close(pipefds2[0]);
    } else {
        // ----- Child Process -----
        close(pipefds1[1]); // Close write end of pipe1
        close(pipefds2[0]); // Close read end of pipe2
        read(pipefds1[0], readmessage, sizeof(readmessage));
        cout << "In Child: Reading from pipe 1 – Message is '" << readmessage << "'" << endl;
        cout << "In Child: Writing to pipe 2 – Message is '" << pipe2writemessage << "'" << endl;
        write(pipefds2[1], pipe2writemessage, strlen(pipe2writemessage) + 1);
        close(pipefds1[0]);
        close(pipefds2[1]);
    }
    return 0;
}
```

4_reader_writer.cpp

```
#include <iostream>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
using namespace std;
int data = 0;
int rc = 0; // Reader count
// Binary semaphores
sem_t db;     // Controls access to the database (writer or first reader)
sem_t mutex;  // Protects rc (reader count)
// Semaphore wrappers
void down(sem_t* s) {
    sem_wait(s);
}
void up(sem_t* s) {
    sem_post(s);
}
// Reader function (synchronized)
void* reader(void*) {
    down(&mutex);
    rc++;
    if (rc == 1)
        down(&db); // First reader locks DB
    up(&mutex);
    cout << "[Reader] Reading data: " << data << endl;
    sleep(1);
    down(&mutex);
    rc--;
    if (rc == 0)
        up(&db); // Last reader unlocks DB
    up(&mutex);
    return NULL;
}
// Writer function (synchronized)
void* writer(void*) {
    down(&db);
    data++;
    cout << "[Writer] Writing data: " << data << endl;
    sleep(1);
    up(&db);
    return NULL;
}
int main() {
    pthread_t r1, r2, w1, w2;
    int choice;
    // Initialize semaphores as binary semaphores
    sem_init(&db, 0, 1);    // Binary semaphore for DB
    sem_init(&mutex, 0, 1); // Binary semaphore for rc
    cout << "1. Without Synchronization (for reference)\n2. With Synchronization (binary
semaphores)\nEnter choice: ";
    cin >> choice;
    if (choice == 1) {
        // UNSYNCHRONIZED (just for demonstration)
        pthread_create(&r1, NULL, [](void*) -> void* {
            cout << "[Reader] Reading data (no sync): " << data << endl;
            sleep(1);
            return NULL;
        }, NULL);
        pthread_create(&w1, NULL, [](void*) -> void* {
            data++;
            cout << "[Writer] Writing data (no sync): " << data << endl;
            sleep(1);
            return NULL;
        }, NULL);
        pthread_create(&r2, NULL, [](void*) -> void* {
            cout << "[Reader] Reading data (no sync): " << data << endl;
            sleep(1);
            return NULL;
        }, NULL);
        pthread_create(&w2, NULL, [](void*) -> void* {
            data++;
            cout << "[Writer] Writing data (no sync): " << data << endl;
            sleep(1);
            return NULL;
        }, NULL);
    } else {
        // SYNCHRONIZED using binary semaphores
        pthread_create(&r1, NULL, reader, NULL);
        pthread_create(&w1, NULL, writer, NULL);
        pthread_create(&r2, NULL, reader, NULL);
        pthread_create(&w2, NULL, writer, NULL);
    }
    // Wait for threads to finish
    pthread_join(r1, NULL);
    pthread_join(w1, NULL);
    pthread_join(r2, NULL);
    pthread_join(w2, NULL);
    // Cleanup
    sem_destroy(&db);
    sem_destroy(&mutex);
    return 0;
}
// Reader Code
// void Reader() {
//     while (true) {
```

```
//        down(mutex);         // Lock to modify rc
//        rc = rc + 1;
//        if (rc == 1)
//            down(db);   // First reader locks DB
//        up(mutex);           // Allow others
//        // Reading section
//        DB -critical Section
//        down(mutex);
//        rc = rc - 1;
//        if (rc == 0)
//            up(db);          // Last reader unlocks DB
//        up(mutex);
//    }
// }
// Writer Code
// void Writer() {
//    while (true) {
//        down(db);         // Lock DB for writing
//        Update_data();       // Writing section
//        up(db);              // Release DB
//    }
// }
```

5_bankers_algorithm.cpp

```
#include <iostream>
using namespace std;
void inputData(int processes, int resources, int allocation[][10], int max[][10],
               int available[]) {
    cout << "Enter allocation matrix:\n";
    for (int i = 0; i < processes; i++)
        for (int j = 0; j < resources; j++)
            cin >> allocation[i][j];
    cout << "Enter max matrix:\n";
    for (int i = 0; i < processes; i++)
        for (int j = 0; j < resources; j++)
            cin >> max[i][j];
    cout << "Enter available resources:\n";
    for (int i = 0; i < resources; i++)
        cin >> available[i];
}
void calculateNeedMatrix(int processes, int resources, int max[][10],
                         int allocation[][10], int need[][10]) {
    for (int i = 0; i < processes; i++)
        for (int j = 0; j < resources; j++)
            need[i][j] = max[i][j] - allocation[i][j];
}
bool isSafeState(int processes, int resources, int allocation[][10], int need[][10],
                 int available[]) {
    bool finish[10] = {false};
    int ava[10], safeSequence[10], count = 0;
    for (int i = 0; i < resources; i++)
        ava[i] = available[i];
    while (count < processes) {
        bool found = false;
        for (int i = 0; i < processes; i++) {
            if (!finish[i]) {
                int j;
                for (j = 0; j < resources; j++)
                    if (need[i][j] > ava[j])
                        break;
                if (j == resources) {
                    for (j = 0; j < resources; j++)
                        ava[j] += allocation[i][j];
                    safeSequence[count++] = i;
                    finish[i] = true;
                    found = true;
                }
            }
        }
        if (!found) {
            cout << "System is not in a safe state.\n";
            return false;
        }
    }
    cout << "System is in a safe state.\nSafe sequence: ";
    for (int i = 0; i < processes; i++)
        cout << safeSequence[i] << " ";
    cout << "\n";
    return true;
}
int main() {
    int processes, resources;
    cout << "Enter number of processes: ";
    cin >> processes;
    cout << "Enter number of resources: ";
    cin >> resources;
    int allocation[10][10], max[10][10], need[10][10], available[10];
    inputData(processes, resources, allocation, max, available);
    calculateNeedMatrix(processes, resources, max, allocation, need);
    isSafeState(processes, resources, allocation, need, available);
    return 0;
}
// Enter number of processes: 5
// Enter number of resources: 3
// Enter allocation matrix:
// 0 1 0
// 2 0 0
// 3 0 2
// 2 1 1
// 0 0 2
// Enter max matrix:
// 7 5 3
// 3 2 2
// 9 0 2
// 4 2 2
// 5 3 3
// Enter available resources:
// 3 3 2
// System is in a safe state.
// Safe sequence: 1 3 4 0 2
```

6_memory_allocation_strategies.cpp

```
#include<iostream>
#include <climits>
using namespace std;
void first_fit(int blocks, int blocksSize[], int process, int processSize[]) {
    for(int i=0;i<process;i++) {
        for(int j=0;j<blocks;j++) {
            if(processSize[i]<=blocksSize[j]) {
                blocksSize[j]-=processSize[i];
                cout<<"Process "<<i+1<<" fitted in block "<<j+1<<endl;
                break;
            }
        }
    }
}
void next_fit(int blocks, int blocksSize[], int process, int processSize[]) {
    int lastOccupied=0;
    for(int i=0;i<process;i++) {
        for(int j=lastOccupied;j<blocks;j++) {
            if(processSize[i]<=blocksSize[j]) {
                blocksSize[j]-=processSize[i];
                lastOccupied=j;
                cout<<"Process "<<i+1<<" fitted in block "<<j+1<<endl;
                break;
            }
        }
    }
}
void best_fit(int blocks, int blocksSize[], int process, int processSize[]) {
    for(int i=0;i<process;i++) {
        int diff = INT_MAX;
        int index = -1;
        for(int j=0;j<blocks;j++) {
            if(blocksSize[j] >= processSize[i] && (blocksSize[j] - processSize[i]) < diff) {
                diff = blocksSize[j] - processSize[i];
                index = j;
            }
        }
        if(index != -1) {
            blocksSize[index] -= processSize[i];
            cout<<"Process "<<i+1<<" fitted in block "<<index+1<<endl;
        } else {
            cout<<"Process "<<i+1<<" could not be allocated\n";
        }
    }
}
void worst_fit(int blocks, int blocksSize[], int process, int processSize[]) {
    for(int i=0;i<process;i++) {
        int diff = INT_MIN;
        int index = -1;
        for(int j=0;j<blocks;j++) {
            if(blocksSize[j] >= processSize[i] && (blocksSize[j] - processSize[i]) > diff) {
                diff = blocksSize[j] - processSize[i];
                index = j;
            }
        }
        if(index != -1) {
            blocksSize[index] -= processSize[i];
            cout<<"Process "<<i+1<<" fitted in block "<<index+1<<endl;
        } else {
            cout<<"Process "<<i+1<<" could not be allocated\n";
        }
    }
}
int main() {
    int blocks;
    cout<<"Enter number of blocks : ";
    cin>>blocks;
    int blocksSize[blocks];
    int copyofBlocksSize[blocks];
    cout<<"Enter sizes of blocks : ";
    for(int i=0;i<blocks;i++) {
        cin>>blocksSize[i];
    }
    int process;
    cout<<"Enter number of processes : ";
    cin>>process;
    int processSize[process];
    int copyofProcessSize[process];
    cout<<"Enter sizes of processes : ";
    for(int i=0;i<process;i++) {
        cin>>processSize[i];
    }
    int ch=0;
    while(ch!=5) {
        cout<<"Menu: \n";
        cout<<"1. First Fit\n";
        cout<<"2. Next Fit\n";
        cout<<"3. Best Fit\n";
        cout<<"4. Worst Fit\n";
        cout<<"5. Exit\n\n";
        cout<<"Enter a choice : ";
        cin>>ch;
        switch (ch)
        {
        case 1:
            copy(blocksSize,blocksSize+blocks,copyofBlocksSize);
            copy(processSize,processSize+process,copyofProcessSize);
            first_fit(blocks,copyofBlocksSize,process,copyofProcessSize);
            break;
        case 2:
            copy(blocksSize,blocksSize+blocks,copyofBlocksSize);
            copy(processSize,processSize+process,copyofProcessSize);
            next_fit(blocks,copyofBlocksSize,process,copyofProcessSize);
            break;
        case 3:
            copy(blocksSize,blocksSize+blocks,copyofBlocksSize);
            copy(processSize,processSize+process,copyofProcessSize);
            best_fit(blocks,copyofBlocksSize,process,copyofProcessSize);
            break;
        case 4:
            copy(blocksSize,blocksSize+blocks,copyofBlocksSize);
            copy(processSize,processSize+process,copyofProcessSize);
            worst_fit(blocks,copyofBlocksSize,process,copyofProcessSize);
            break;
        case 5:
            cout<<"Exiting...\n";
            break;
        default:
            cout<<"Invalid choice, please try again.";
            break;
        }
    }
    return 0;
}
// Enter sizes of blocks : 40 50 60 70
// Enter number of processes : 4
// Enter sizes of processes : 20 10 30 40
// Menu:
// 1. First Fit
// 2. Next Fit
// 3. Best Fit
// 4. Worst Fit
// 5. Exit
// Enter a choice : 1
// Process 1 fitted in block 1
// Process 2 fitted in block 1
// Process 3 fitted in block 2
// Process 4 fitted in block 3
// Menu:
// 1. First Fit
// 2. Next Fit
// 3. Best Fit
// 4. Worst Fit
// 5. Exit
// Enter a choice : 2
// Process 1 fitted in block 1
// Process 2 fitted in block 1
// Process 3 fitted in block 2
// Process 4 fitted in block 3
// Menu:
// 1. First Fit
// 2. Next Fit
// 3. Best Fit
// 4. Worst Fit
// 5. Exit
// Enter a choice : 3
// Process 1 fitted in block 1
// Process 2 fitted in block 1
// Process 3 fitted in block 2
// Process 4 fitted in block 3
// Menu:
// 1. First Fit
// 2. Next Fit
// 3. Best Fit
// 4. Worst Fit
// 5. Exit
// Enter a choice :
```

7_page_replacement_algorithms.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int fifo(vector<int> pages, int capacity, int &hits) {
    vector<int> frame(capacity, -1);
    int front = 0, faults = 0;
    hits = 0;
    for (int page : pages) {
        if (find(frame.begin(), frame.end(), page) != frame.end()) {
            hits++;
        } else {
            frame[front] = page;
            front = (front + 1) % capacity;
            faults++;
        }
    }
    return faults;
}
int lru(vector<int> pages, int capacity, int &hits) {
    vector<int> frame(capacity, -1);
    vector<int> lastUsed(capacity, -1);
    int faults = 0;
    hits = 0;
    for (int i = 0; i < pages.size(); i++) {
        int page = pages[i];
        bool found = false;
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == page) {
                hits++;
                found = true;
                lastUsed[j] = i;
                break;
            }
        }
        if (!found) {
            int lru_index = 0;
            for (int j = 1; j < capacity; j++) {
                if (lastUsed[j] < lastUsed[lru_index]) {
                    lru_index = j;
                }
            }
            frame[lru_index] = page;
            lastUsed[lru_index] = i;
            faults++;
        }
    }
    return faults;
}
int optimal(vector<int> pages, int capacity, int &hits) {
    vector<int> frame(capacity, -1);
    int faults = 0;
    hits = 0;
    for (int i = 0; i < pages.size(); i++) {
        int page = pages[i];
        bool found = false;
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == page) {
                hits++;
                found = true;
                break;
            }
        }
        if (!found) {
            int replace_index = -1, farthest = i + 1;
            for (int j = 0; j < capacity; j++) {
                int k;
                for (k = i + 1; k < pages.size(); k++) {
                    if (frame[j] == pages[k]) break;
                }
                if (k == pages.size()) {
                    replace_index = j;
                    break;
                }
                if (k > farthest) {
                    farthest = k;
                    replace_index = j;
                }
            }
            if (replace_index == -1)
                replace_index = 0;
            frame[replace_index] = page;
            faults++;
        }
    }
    return faults;
}
int main() {
    int n, capacity;
    cout << "Enter number of pages: ";
    cin >> n;
    vector<int> pages(n);
    cout << "Enter the page sequence: ";
    for (int i = 0; i < n; i++) cin >> pages[i];
    cout << "Enter number of frames: ";
    cin >> capacity;
    int choice;
    do {
        cout << "\nMenu:\n";
        cout << "1. FIFO\n2. LRU\n3. Optimal\n4. Exit\nChoose: ";
        cin >> choice;
        int hits = 0, faults = 0;
        switch (choice) {
            case 1:
                faults = fifo(pages, capacity, hits);
                cout << "FIFO Page Faults: " << faults << ", Page Hits: " << hits << endl;
                break;
            case 2:
                faults = lru(pages, capacity, hits);
                cout << "LRU Page Faults: " << faults << ", Page Hits: " << hits << endl;
                break;
            case 3:
                faults = optimal(pages, capacity, hits);
                cout << "Optimal Page Faults: " << faults << ", Page Hits: " << hits << endl;
                break;
            case 4:
                cout << "Exiting...\n";
                break;
            default:
                cout << "Invalid option.\n";
        }
    } while (choice != 4);
    return 0;
}
// Enter number of pages: 7
// Enter the page sequence: 1 2 3 1 3 2 0
// Enter number of frames: 3
// Menu:
// 1. FIFO
// 2. LRU
// 3. Optimal
// 4. Exit
// Choose: 1
// FIFO Page Faults: 4, Page Hits: 3
// Menu:
// 1. FIFO
// 2. LRU
// 3. Optimal
// 4. Exit
// Choose: 2
// LRU Page Faults: 4, Page Hits: 3
// Menu:
// 1. FIFO
// 2. LRU
// 3. Optimal
// 4. Exit
// Choose: 3
// Optimal Page Faults: 4, Page Hits: 3
// Menu:
// 1. FIFO
// 2. LRU
// 3. Optimal
// 4. Exit
// Choose:
```

8_disk_scheduling.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <climits>
using namespace std;
// FCFS
void fcfs(vector<int> requests, int head) {
    int seek = 0;
    cout << "Sequence: " << head;
    for (int r : requests) {
        seek += abs(head - r);
        head = r;
        cout << " -> " << head;
    }
    cout << "\nTotal Seek Time (FCFS): " << seek << endl;
}
// SSTF
void sstf(vector<int> requests, int head) {
    vector<bool> visited(requests.size(), false);
    int seek = 0, count = 0;
    cout << "Sequence: " << head;
    while (count < requests.size()) {
        int min_dist = INT_MAX;
        int index = -1;
        for (int i = 0; i < requests.size(); i++) {
            if (!visited[i] && abs(head - requests[i]) < min_dist) {
                min_dist = abs(head - requests[i]);
                index = i;
            }
        }
        visited[index] = true;
        seek += abs(head - requests[index]);
        head = requests[index];
        cout << " -> " << head;
        count++;
    }
    cout << "\nTotal Seek Time (SSTF): " << seek << endl;
}
// SCAN
void scan(vector<int> requests, int head, int disk_size, string direction) {
    vector<int> left, right;
    int seek = 0;
    if (direction == "left") left.push_back(0);
    else right.push_back(disk_size - 1);
    for (int r : requests) {
        if (r < head) left.push_back(r);
        else right.push_back(r);
    }
    sort(left.begin(), left.end());
    sort(right.begin(), right.end());
    cout << "Sequence: " << head;
    if (direction == "left") {
        for (int i = left.size() - 1; i >= 0; i--) {
            seek += abs(head - left[i]);
            head = left[i];
            cout << " -> " << head;
        }
        for (int i = 0; i < right.size(); i++) {
            seek += abs(head - right[i]);
            head = right[i];
            cout << " -> " << head;
        }
    } else {
        for (int i = 0; i < right.size(); i++) {
            seek += abs(head - right[i]);
            head = right[i];
            cout << " -> " << head;
        }
        for (int i = left.size() - 1; i >= 0; i--) {
            seek += abs(head - left[i]);
            head = left[i];
            cout << " -> " << head;
        }
    }
    cout << "\nTotal Seek Time (SCAN): " << seek << endl;
}
// C-SCAN
void cscan(vector<int> requests, int head, int disk_size) {
    vector<int> left, right;
    int seek = 0;
    right.push_back(disk_size - 1);
    left.push_back(0);
    for (int r : requests) {
        if (r >= head) right.push_back(r);
        else left.push_back(r);
    }
    sort(left.begin(), left.end());
    sort(right.begin(), right.end());
    cout << "Sequence: " << head;
    for (int i = 0; i < right.size(); i++) {
        seek += abs(head - right[i]);
        head = right[i];
        cout << " -> " << head;
    }
    // Move to 0
    seek += (disk_size - 1);
    head = 0;
    cout << " -> " << head;
    for (int i = 0; i < left.size(); i++) {
        seek += abs(head - left[i]);
        head = left[i];
        cout << " -> " << head;
    }
    cout << "\nTotal Seek Time (C-SCAN): " << seek << endl;
}
int main() {
    int n, head, disk_size, choice;
    cout << "Enter number of disk requests: ";
    cin >> n;
    vector<int> requests(n);
    cout << "Enter request queue: ";
    for (int i = 0; i < n; i++) {
        cin >> requests[i];
    }
    cout << "Enter initial head position: ";
    cin >> head;
    cout << "Enter disk size: ";
    cin >> disk_size;
    do {
        cout << "\n--- Disk Scheduling Menu ---\n";
        cout << "1. FCFS\n";
        cout << "2. SSTF\n";
        cout << "3. SCAN\n";
        cout << "4. C-SCAN\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                fcfs(requests, head);
                break;
            case 2:
                sstf(requests, head);
                break;
            case 3: {
                string direction;
                cout << "Enter direction (left/right): ";
                cin >> direction;
                scan(requests, head, disk_size, direction);
                break;
            }
            case 4:
                cscan(requests, head, disk_size);
                break;
            case 5:
                cout << "Exiting program.\n";
                break;
            default:
                cout << "Invalid choice! Try again.\n";
        }
    } while (choice != 5);
    return 0;
}
// Enter number of disk requests: 7
// Enter request queue: 82 170 43 140 24 16 190
// Enter initial head position: 50
// Enter disk size: 200
//FCFS
// Enter your choice: 1
// Sequence: 50 -> 82 -> 170 -> 43 -> 140 -> 24 -> 16 -> 190
// Total Seek Time (FCFS): 642
//SSTF
// Enter your choice: 2
// Sequence: 50 -> 43 -> 24 -> 16 -> 82 -> 140 -> 170 -> 190
// Total Seek Time (SSTF): 208
//SCAN
// Enter your choice: 3
// Enter direction (left/right): right
// Sequence: 50 -> 82 -> 140 -> 170 -> 190 -> 199 -> 43 -> 24 -> 16
// Total Seek Time (SCAN): 332
// C-SCAN
// Enter your choice: 4
// Sequence: 50 -> 82 -> 140 -> 170 -> 190 -> 199 -> 0 -> 0 -> 16 -> 24 -> 43
// Total Seek Time (C-SCAN): 391
```