**LABORATORY RECORD**

**OPERATING SYSTEM**

**Student Name- ARPIT SINGH**

**Student Regn. No.- 23BAI11069**

**Interim Semester 2024-25**

**Slot: D11+D12+A14**

**Year- 2024-25**

**Faculty Name- Dr. Nilesh Kunhare**

**School of Computing Science & Engineering**

**VIT Bhopal University**

A blue and white logo

Description automatically generated

Table of Contents:

|  |  |  |
| --- | --- | --- |
| S. No | Program’s Name | Page No |
| 1 | Simulate the following CPU Scheduling Algorithms: |  |
|  | a) FCFS | 3 |
|  | b) SJF | 5 |
|  | c) Priority | 7 |
|  | d) Round Robin | 9 |
| 2 | Simulate MVT and MFT | 13 |
| 3 | Simulate Banker’s Algorithm for Deadlock Avoidance | 15 |
| 4 | Simulate Banker’s Algorithm for Deadlock Prevention | 19 |
| 5 | Simulate all Page Replacement Algorithms: |  |
|  | a) FIFO | 22 |
|  | b) LRU | 23 |
|  | c) Optimal | 25 |
| 6 | Simulate Paging Technique of Memory Management | 27 |

**1.First Come First Serve (FCFS) Scheduling**

**#include <iostream>**

**using namespace std;**

**void FCFS(int processes[], int n, int bt[], int wt[], int tat[]) {**

**wt[0] = 0;**

**for (int i = 1; i < n; i++)**

**wt[i] = wt[i - 1] + bt[i - 1];**

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

**tat[i] = wt[i] + bt[i];**

**}**

**int main() {**

**int processes[] = {1, 2, 3, 4};**

**int n = 4;**

**int burst\_time[] = {5, 3, 8, 6};**

**int waiting\_time[n], turnaround\_time[n];**

**FCFS(processes, n, burst\_time, waiting\_time, turnaround\_time);**

**cout << "Processes Burst Time Waiting Time Turnaround Time\n";**

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

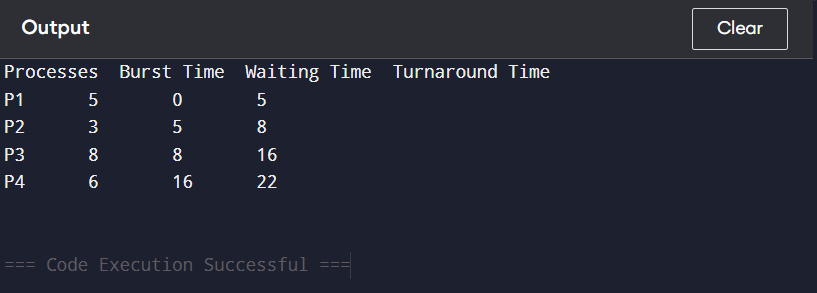
**cout << "P" << processes[i] << "\t\t" << burst\_time[i] << "\t\t"**

**<< waiting\_time[i] << "\t\t" << turnaround\_time[i] << endl;**

**}**

**return 0;**

**}**

****

**2.SJF (Shortest Job First)**

**#include <iostream>**

**#include <algorithm>**

**using namespace std;**

**struct Process {**

**int id, bt, wt, tat;**

**};**

**bool compare(Process a, Process b) {**

**return a.bt < b.bt;**

**}**

**int main() {**

**Process p[] = {{1, 6}, {2, 8}, {3, 7}, {4, 3}};**

**int n = 4;**

**sort(p, p + n, compare);**

**p[0].wt = 0;**

**for (int i = 1; i < n; i++)**

**p[i].wt = p[i - 1].wt + p[i - 1].bt;**

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

**p[i].tat = p[i].wt + p[i].bt;**

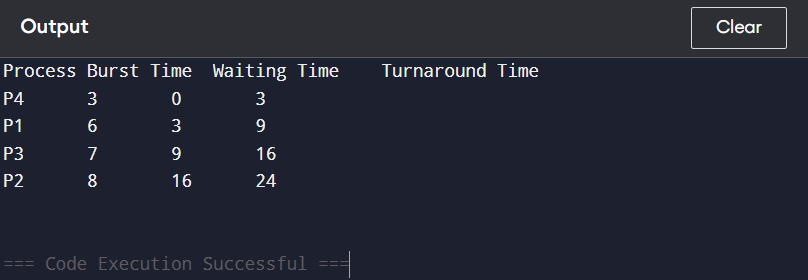
**cout << "Process\tBurst Time\tWaiting Time\tTurnaround Time\n";**

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

**cout << "P" << p[i].id << "\t\t" << p[i].bt << "\t\t" << p[i].wt << "\t\t" << p[i].tat << endl;**

**return 0;**

**}**

****

**3. Priority Scheduling**

**#include <iostream>**

**#include <algorithm>**

**using namespace std;**

**struct Process {**

**int id, bt, pri, wt, tat;**

**};**

**bool compare(Process a, Process b) {**

**return a.pri > b.pri; // Higher priority first**

**}**

**int main() {**

**Process p[] = {{1, 10, 1}, {2, 1, 3}, {3, 2, 2}, {4, 1, 4}};**

**int n = 4;**

**sort(p, p + n, compare);**

**p[0].wt = 0;**

**for (int i = 1; i < n; i++)**

**p[i].wt = p[i - 1].wt + p[i - 1].bt;**

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

**p[i].tat = p[i].wt + p[i].bt;**

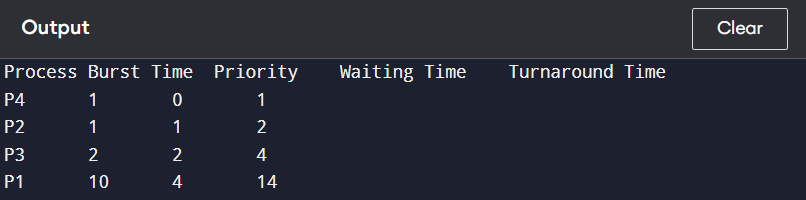
**cout << "Process\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n";**

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

**cout << "P" << p[i].id << "\t\t" << p[i].bt << "\t\t" << p[i].wt << "\t\t" << p[i].tat << endl;**

**return 0;**

**}**

****

**4. Round Robin**

**#include <iostream>**

**#include <queue>**

**using namespace std;**

**struct Process {**

**int id, bt, remaining\_bt;**

**};**

**void roundRobin(Process processes[], int n, int quantum) {**

**queue<int> q;**

**int waiting\_time[n] = {0}, turnaround\_time[n];**

**int time = 0;**

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

**q.push(i);**

**while (!q.empty()) {**

**int i = q.front();**

**q.pop();**

**if (processes[i].remaining\_bt <= quantum) {**

**time += processes[i].remaining\_bt;**

**turnaround\_time[i] = time;**

**processes[i].remaining\_bt = 0;**

**} else {**

**time += quantum;**

**processes[i].remaining\_bt -= quantum;**

**q.push(i);**

**}**

**}**

**cout << "Process\tBurst Time\tTurnaround Time\n";**

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

**cout << "P" << processes[i].id << "\t\t" << processes[i].bt << "\t\t"**

**<< turnaround\_time[i] << endl;**

**}**

**int main() {**

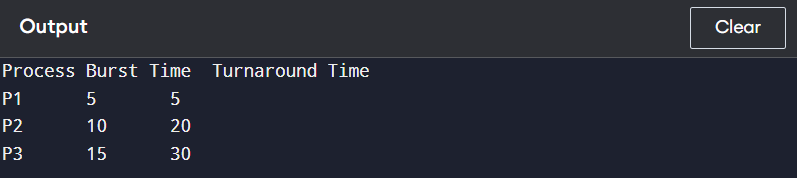
**Process processes[] = {{1, 5, 5}, {2, 10, 10}, {3, 15, 15}};**

**int n = 3, quantum = 5;**

**roundRobin(processes, n, quantum);**

**return 0;**

**}**

****

**2.a) MVT (Multiple Variable Tasks)**

**#include <iostream>**

**using namespace std;**

**void allocateMVT(int memorySize, int processSize[], int n) {**

**int remainingMemory = memorySize;**

**cout << "Process\t\tProcess Size\tStatus\n";**

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

**if (processSize[i] <= remainingMemory) {**

**remainingMemory -= processSize[i];**

**cout << "P" << i + 1 << "\t\t" << processSize[i] << "\t\tAllocated\n";**

**} else {**

**cout << "P" << i + 1 << "\t\t" << processSize[i] << "\t\tNot Allocated\n";**

**}**

**}**

**cout << "Remaining Memory: " << remainingMemory << " units\n";**

**}**

**int main() {**

**int memorySize = 100; // Total available memory**

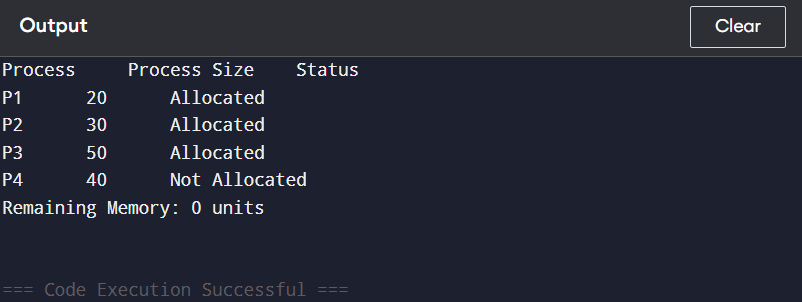
**int processSize[] = {20, 30, 50, 40}; // Sizes of processes**

**int n = sizeof(processSize) / sizeof(processSize[0]);**

**allocateMVT(memorySize, processSize, n);**

**return 0;**

**}**

****

**2.b) MFT (Multiple Fixed Tasks)**

**#include <iostream>**

**using namespace std;**

**void allocateMFT(int blockSize, int memorySize, int processSize[], int n) {**

**int numBlocks = memorySize / blockSize;**

**int remainingBlocks = numBlocks;**

**cout << "Process\t\tProcess Size\tStatus\t\tBlock Wasted\n";**

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

**if (processSize[i] <= blockSize && remainingBlocks > 0) {**

**remainingBlocks--;**

**cout << "P" << i + 1 << "\t\t" << processSize[i] << "\t\tAllocated\t" << blockSize - processSize[i] << endl;**

**} else {**

**cout << "P" << i + 1 << "\t\t" << processSize[i] << "\t\tNot Allocated\t--\n";**

**}**

**}**

**cout << "Remaining Blocks: " << remainingBlocks << endl;**

**}**

**int main() {**

**int memorySize = 100;**

**int blockSize = 25;**

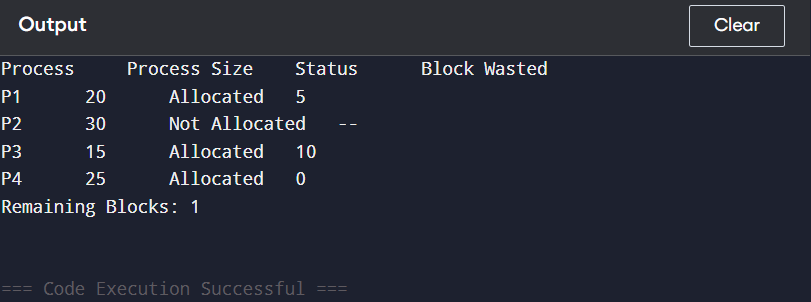
**int processSize[] = {20, 30, 15, 25};**

**int n = sizeof(processSize) / sizeof(processSize[0]);**

**allocateMFT(blockSize, memorySize, processSize, n);**

**return 0;**

**}**

****

**3.Simulate Bankers algorithm for Deadlock Avoidance**

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**class BankersAlgorithm {**

**private:**

**int numProcesses, numResources;**

**vector<vector<int>> allocation, max, need;**

**vector<int> available;**

**public:**

**BankersAlgorithm(int p, int r) : numProcesses(p), numResources(r) {**

**allocation.resize(p, vector<int>(r));**

**max.resize(p, vector<int>(r));**

**need.resize(p, vector<int>(r));**

**available.resize(r);**

**}**

**void inputData() {**

**allocation = {{0, 1, 0},**

**{2, 0, 0},**

**{3, 0, 2},**

**{2, 1, 1},**

**{0, 0, 2}};**

**max = {{7, 5, 3},**

**{3, 2, 2},**

**{9, 0, 2},**

**{2, 2, 2},**

**{4, 3, 3}};**

**available = {3, 3, 2};**

**for (int i = 0; i < numProcesses; i++) {**

**for (int j = 0; j < numResources; j++) {**

**need[i][j] = max[i][j] - allocation[i][j];**

**}**

**}**

**}**

**bool isSafe() {**

**vector<int> work = available;**

**vector<bool> finish(numProcesses, false);**

**vector<int> safeSeq;**

**while (safeSeq.size() < numProcesses) {**

**bool progressMade = 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] > work[j]) {**

**canAllocate = false;**

**break;**

**}**

**}**

**if (canAllocate) {**

**for (int j = 0; j < numResources; j++) {**

**work[j] += allocation[i][j];**

**}**

**finish[i] = true;**

**safeSeq.push\_back(i);**

**progressMade = true;**

**}**

**}**

**}**

**if (!progressMade) {**

**cout << "System is in an unsafe state.\n";**

**return false;**

**}**

**}**

**cout << "System is in a safe state. Safe Sequence: ";**

**for (int i = 0; i < safeSeq.size(); i++) {**

**cout << "P" << safeSeq[i] << " ";**

**}**

**cout << endl;**

**return true;**

**}**

**};**

**int main() {**

**int numProcesses = 5, numResources = 3;**

**BankersAlgorithm bankers(numProcesses, numResources);**

**bankers.inputData();**

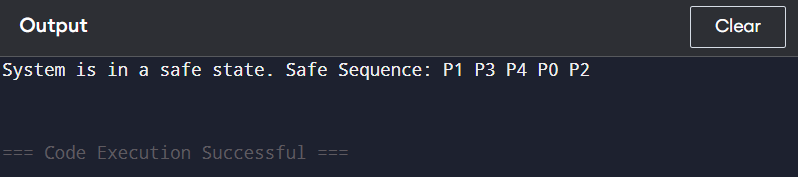
**if (!bankers.isSafe()) {**

**cout << "Deadlock detected. The system is unsafe.\n";**

**}**

**return 0;**

**}**

****

**4. Simulate Bankers Algorithm for deadlock Prevention**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

string bankers\_algorithm(vector<string>& processes, vector<int>& available, vector<vector<int>>& max\_claims, vector<vector<int>>& allocations) {

int num\_processes = processes.size();

int num\_resources = available.size();

vector<vector<int>> need(num\_processes, vector<int>(num\_resources));

vector<int> work = available;

vector<bool> finish(num\_processes, false);

vector<string> safe\_sequence;

for (int i = 0; i < num\_processes; ++i)

for (int j = 0; j < num\_resources; ++j)

need[i][j] = max\_claims[i][j] - allocations[i][j];

while (safe\_sequence.size() < num\_processes) {

bool allocated = false;

for (int i = 0; i < num\_processes; ++i) {

if (!finish[i]) {

bool can\_allocate = true;

for (int j = 0; j < num\_resources; ++j) {

if (need[i][j] > work[j]) {

can\_allocate = false;

break;

}

}

if (can\_allocate) {

for (int j = 0; j < num\_resources; ++j)

work[j] += allocations[i][j];

safe\_sequence.push\_back(processes[i]);

finish[i] = true;

allocated = true;

}

}

}

if (!allocated)

return "Deadlock detected. No safe sequence.";

}

string result;

for (const string& p : safe\_sequence)

result += p + " ";

return result;

}

int main() {

vector<string> processes = {"P1", "P2", "P3", "P4"};

vector<int> available = {3, 3, 2};

vector<vector<int>> max\_claims = {{7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}};

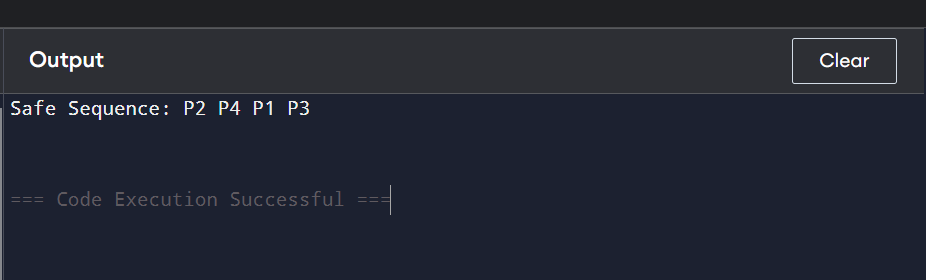
vector<vector<int>> allocations = {{0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1}};

string result = bankers\_algorithm(processes, available, max\_claims, allocations);

cout << "Safe Sequence: " << result << endl;

return 0;

}



**5. Simulate all Page Replacement Algorithms**

a) FIFO

b) LRU

c) Optimal  
  
A. FIFO  
  
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int fifo(vector<int>& pages, int capacity) {

vector<int> memory;

int faults = 0;

for (int page : pages) {

if (find(memory.begin(), memory.end(), page) == memory.end()) {

if (memory.size() < capacity)

memory.push\_back(page);

else {

memory.erase(memory.begin());

memory.push\_back(page);

}

faults++;

}

}

return faults;

}

int main() {

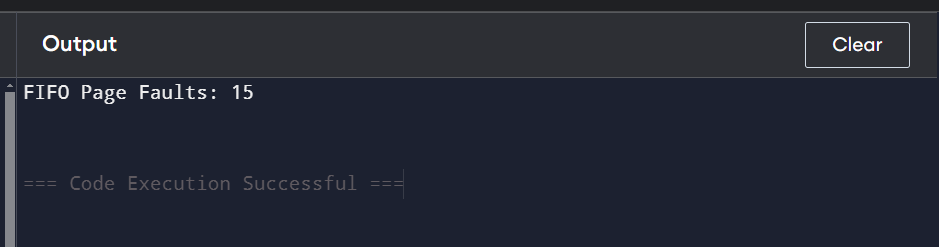
vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int capacity = 3;

cout << "FIFO Page Faults: " << fifo(pages, capacity) << endl;

return 0;

}



B. LRU  
#include <iostream>

#include <vector>

#include <unordered\_map>

#include <algorithm>

using namespace std;

int lru(vector<int>& pages, int capacity) {

vector<int> memory;

unordered\_map<int, int> recent;

int faults = 0;

for (int i = 0; i < pages.size(); i++) {

int page = pages[i];

if (find(memory.begin(), memory.end(), page) == memory.end()) {

if (memory.size() < capacity)

memory.push\_back(page);

else {

int lru\_page = memory[0];

for (int p : memory) {

if (recent[p] < recent[lru\_page])

lru\_page = p;

}

memory.erase(find(memory.begin(), memory.end(), lru\_page));

memory.push\_back(page);

}

faults++;

}

recent[page] = i;

}

return faults;

}

int main() {

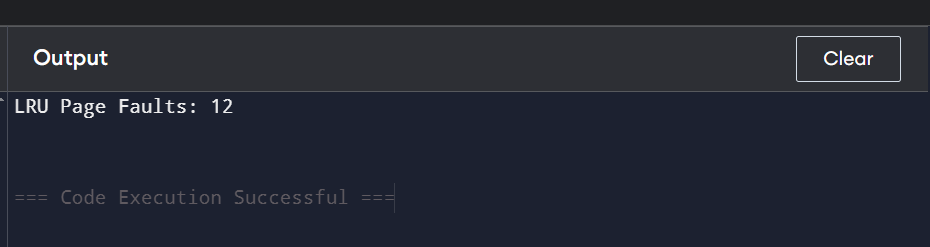
vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int capacity = 3;

cout << "LRU Page Faults: " << lru(pages, capacity) << endl;

return 0;

}



C. OPTIMAL  
  
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int optimal(vector<int>& pages, int capacity) {

vector<int> memory;

int faults = 0;

for (int i = 0; i < pages.size(); i++) {

int page = pages[i];

if (find(memory.begin(), memory.end(), page) == memory.end()) {

if (memory.size() < capacity)

memory.push\_back(page);

else {

int farthest = -1, to\_remove = -1;

for (int j = 0; j < memory.size(); j++) {

int next\_use = find(pages.begin() + i + 1, pages.end(), memory[j]) - pages.begin();

if (next\_use > farthest) {

farthest = next\_use;

to\_remove = j;

}

}

memory[to\_remove] = page;

}

faults++;

}

}

return faults;

}

int main() {

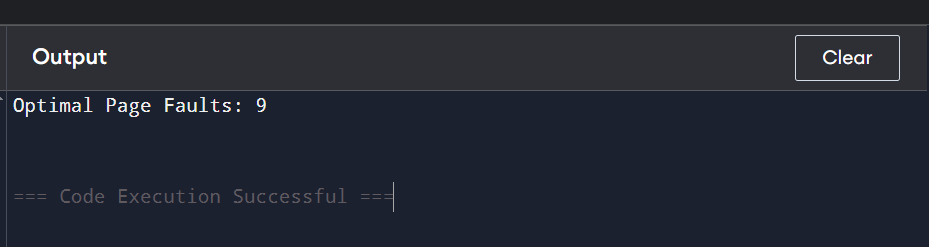
vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int capacity = 3;

cout << "Optimal Page Faults: " << optimal(pages, capacity) << endl;

return 0;

}



6. Simulate Paging Technique of Memory Management  
#include <iostream>

#include <vector>

#include <unordered\_map>

#include <algorithm>

using namespace std;

void pagingTechnique(int memorySize, int pageSize, unordered\_map<int, vector<int>>& processes) {

int numFrames = memorySize / pageSize;

vector<int> memory(numFrames, -1); // Initialize all frames to -1 (empty)

unordered\_map<int, unordered\_map<int, int>> pageTable; cout << "Total Memory Frames: " << numFrames << endl;

for (auto& process : processes) {

int pid = process.first;

vector<int>& pages = process.second;

cout << "Allocating pages for Process " << pid << ":\n";

for (int page : pages) {

auto emptyFrame = find(memory.begin(), memory.end(), -1);

if (emptyFrame != memory.end()) {

int frameIndex = emptyFrame - memory.begin();

memory[frameIndex] = pid;

pageTable[pid][page] = frameIndex;

cout << "Page " << page << " of Process " << pid << " added to Frame " << frameIndex << endl;

} else {

cout << "No free frames available for Page " << page << " of Process " << pid << endl;

}

}

}

cout << "\nFinal Memory Allocation:\n";

for (int i = 0; i < memory.size(); i++) {

if (memory[i] == -1) {

cout << "Frame " << i << ": Empty\n";

} else {

cout << "Frame " << i << ": Process " << memory[i] << endl;

}

}

cout << "\nPage Tables:\n";

for (auto& process : pageTable) {

int pid = process.first;

cout << "Process " << pid << " Page Table:\n";

for (auto& entry : process.second) {

cout << " Page " << entry.first << " -> Frame " << entry.second << endl;

}

}

}

int main() {

int memorySize = 16; //

int pageSize = 4;

unordered\_map<int, vector<int>> processes = {

{1, {0, 1}},

{2, {0, 2}},

{3, {1, 3}}

};

pagingTechnique(memorySize, pageSize, processes);

return 0;

}

