**FCFS WITHOUT AT:**

#include <iostream>

using namespace std;

struct Process {

int pid; // Process ID

int bt; // Burst Time

int ct; // Completion Time

int tat; // Turnaround Time

int wt; // Waiting Time

};

int main() {

int n;

cout << "Enter number of processes: ";

cin >> n;

Process proc[n];

// Input burst times

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

proc[i].pid = i + 1;

cout << "Enter burst time for process " << proc[i].pid << ": ";

cin >> proc[i].bt;

}

// FCFS Scheduling logic (no arrival time)

proc[0].ct = proc[0].bt;

proc[0].tat = proc[0].ct;

proc[0].wt = 0;

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

proc[i].ct = proc[i - 1].ct + proc[i].bt;

proc[i].tat = proc[i].ct;

proc[i].wt = proc[i].tat - proc[i].bt;

}

// Output

cout << "\nPID\tBT\tCT\tTAT\tWT\n";

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

cout << proc[i].pid << "\t" << proc[i].bt << "\t" << proc[i].ct << "\t"

<< proc[i].tat << "\t" << proc[i].wt << endl;

}

return 0;}

**FCFS WITH ARRIVAL TIME**

#include <iostream>

using namespace std;

struct Process {

int pid; // Process ID

int at; // Arrival Time

int bt; // Burst Time

int ct; // Completion Time

int tat; // Turnaround Time

int wt; // Waiting Time

};

int main() {

int n;

cout << "Enter number of processes: ";

cin >> n;

Process proc[n];

// Input process details

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

proc[i].pid = i + 1;

cout << "Enter arrival time and burst time for process " << proc[i].pid << ": ";

cin >> proc[i].at >> proc[i].bt;

}

// Sort processes by arrival time (FCFS basis)

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

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

if (proc[i].at > proc[j].at) {

swap(proc[i], proc[j]);

}

}

}

// Calculate Completion, Turnaround and Waiting times

proc[0].ct = proc[0].at + proc[0].bt;

proc[0].tat = proc[0].ct - proc[0].at;

proc[0].wt = proc[0].tat - proc[0].bt;

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

if (proc[i].at > proc[i - 1].ct)

proc[i].ct = proc[i].at + proc[i].bt; // CPU idle time

else

proc[i].ct = proc[i - 1].ct + proc[i].bt;

proc[i].tat = proc[i].ct - proc[i].at;

proc[i].wt = proc[i].tat - proc[i].bt;

}

// Output results

cout << "\nPID\tAT\tBT\tCT\tTAT\tWT\n";

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

cout << proc[i].pid << "\t" << proc[i].at << "\t" << proc[i].bt << "\t"

<< proc[i].ct << "\t" << proc[i].tat << "\t" << proc[i].wt << endl;

}

return 0;

}

**SJF WITHOUT AT**

#include <iostream>

using namespace std;

struct Process {

int pid;

int bt;

int ct;

int tat;

int wt;

};

int main() {

int n;

cout << "Enter number of processes: ";

cin >> n;

Process proc[n];

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

proc[i].pid = i + 1;

cout << "Enter Burst Time for Process " << proc[i].pid << ": ";

cin >> proc[i].bt;

}

// Sort by Burst Time (Shortest Job First)

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

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

if (proc[j].bt < proc[i].bt) {

swap(proc[i], proc[j]);

}

}

}

// Compute CT, TAT, WT

proc[0].ct = proc[0].bt;

proc[0].tat = proc[0].ct;

proc[0].wt = 0;

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

proc[i].ct = proc[i - 1].ct + proc[i].bt;

proc[i].tat = proc[i].ct;

proc[i].wt = proc[i].tat - proc[i].bt;

}

// Output

cout << "\nPID\tBT\tCT\tTAT\tWT\n";

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

cout << proc[i].pid << "\t" << proc[i].bt << "\t" << proc[i].ct

<< "\t" << proc[i].tat << "\t" << proc[i].wt << endl;

}

return 0;

}

**SJF WITH AT**

#include <iostream>

#include <vector>

using namespace std;

struct Process {

int pid;

int at;

int bt;

int ct;

int tat;

int wt;

bool done = false;

};

int main() {

int n;

cout << "Enter number of processes: ";

cin >> n;

vector<Process> proc(n);

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

proc[i].pid = i + 1;

cout << "Enter Arrival Time and Burst Time for Process " << proc[i].pid << ": ";

cin >> proc[i].at >> proc[i].bt;

}

int completed = 0, time = 0;

while (completed < n) {

int idx = -1;

int minBT = 1e9;

// Find process with shortest BT that has arrived and is not done

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

if (!proc[i].done && proc[i].at <= time && proc[i].bt < minBT) {

minBT = proc[i].bt;

idx = i;

}

}

if (idx != -1) {

proc[idx].ct = time + proc[idx].bt;

proc[idx].tat = proc[idx].ct - proc[idx].at;

proc[idx].wt = proc[idx].tat - proc[idx].bt;

time = proc[idx].ct;

proc[idx].done = true;

completed++;

} else {

time++; // If no process has arrived yet, increment time

}

}

// Output

cout << "\nPID\tAT\tBT\tCT\tTAT\tWT\n";

for (const auto& p : proc) {

cout << p.pid << "\t" << p.at << "\t" << p.bt << "\t"

<< p.ct << "\t" << p.tat << "\t" << p.wt << endl;

}

return 0;

}

**PRIORITY WITHOUT :**

#include <iostream>

using namespace std;

struct Process {

int pid;

int bt; // Burst Time

int priority; // Lower value means higher priority

int ct; // Completion Time

int tat; // Turnaround Time

int wt; // Waiting Time

};

int main() {

int n;

cout << "Enter number of processes: ";

cin >> n;

Process proc[n];

// Input burst time and priority for each process

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

proc[i].pid = i + 1;

cout << "Enter Burst Time and Priority for Process " << proc[i].pid << ": ";

cin >> proc[i].bt >> proc[i].priority;

// Basic validation

if (proc[i].bt < 0 || proc[i].priority < 0) {

cout << "Invalid input! Burst Time and Priority must be non-negative.\n";

return 1;

}

}

// Sort by priority (ascending order — lower number = higher priority)

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

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

if (proc[j].priority < proc[i].priority) {

swap(proc[i], proc[j]);

}

}

}

// Calculate Completion, Turnaround, and Waiting times

proc[0].ct = proc[0].bt;

proc[0].tat = proc[0].ct;

proc[0].wt = 0;

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

proc[i].ct = proc[i - 1].ct + proc[i].bt;

proc[i].tat = proc[i].ct;

proc[i].wt = proc[i].tat - proc[i].bt;

}

// Display table

cout << "\nPID\tBT\tPriority\tCT\tTAT\tWT\n";

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

cout << proc[i].pid << "\t" << proc[i].bt << "\t" << proc[i].priority << "\t\t"

<< proc[i].ct << "\t" << proc[i].tat << "\t" << proc[i].wt << endl;

}

return 0;

}

**PRIORITY WITH AT**

#include <iostream>

#include <vector>

using namespace std;

struct Process {

int pid;

int at;

int bt;

int priority;

int ct;

int tat;

int wt;

bool done = false;

};

int main() {

int n;

cout << "Enter number of processes: ";

cin >> n;

vector<Process> proc(n);

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

proc[i].pid = i + 1;

cout << "Enter Arrival Time, Burst Time, and Priority for Process " << proc[i].pid << ": ";

cin >> proc[i].at >> proc[i].bt >> proc[i].priority;

}

int completed = 0, time = 0;

while (completed < n) {

int idx = -1;

int highestPriority = 1e9;

// Find highest priority process that has arrived and is not done

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

if (!proc[i].done && proc[i].at <= time) {

if (proc[i].priority < highestPriority) {

highestPriority = proc[i].priority;

idx = i;

}

}

}

if (idx != -1) {

time += proc[idx].bt;

proc[idx].ct = time;

proc[idx].tat = proc[idx].ct - proc[idx].at;

proc[idx].wt = proc[idx].tat - proc[idx].bt;

proc[idx].done = true;

completed++;

} else {

time++; // Idle time if no process has arrived

}

}

// Output

cout << "\nPID\tAT\tBT\tPriority\tCT\tTAT\tWT\n";

for (auto& p : proc) {

cout << p.pid << "\t" << p.at << "\t" << p.bt << "\t" << p.priority << "\t\t"

<< p.ct << "\t" << p.tat << "\t" << p.wt << endl;

}

return 0;

}

**ROUND ROBBIN WITHOUT AT**

#include <iostream>

using namespace std;

struct Process {

int pid;

int bt;

int rt; // Remaining Time

int ct;

int tat;

int wt;

};

int main() {

int n, tq;

cout << "Enter number of processes: ";

cin >> n;

cout << "Enter Time Quantum: ";

cin >> tq;

Process proc[n];

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

proc[i].pid = i + 1;

cout << "Enter Burst Time for Process " << proc[i].pid << ": ";

cin >> proc[i].bt;

proc[i].rt = proc[i].bt;

}

int time = 0;

bool done;

while (true) {

done = true;

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

if (proc[i].rt > 0) {

done = false;

if (proc[i].rt > tq) {

time += tq;

proc[i].rt -= tq;

} else {

time += proc[i].rt;

proc[i].ct = time;

proc[i].rt = 0;

}

}

}

if (done)

break;

}

// Calculate TAT and WT

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

proc[i].tat = proc[i].ct;

proc[i].wt = proc[i].tat - proc[i].bt;

}

// Output

cout << "\nPID\tBT\tCT\tTAT\tWT\n";

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

cout << proc[i].pid << "\t" << proc[i].bt << "\t" << proc[i].ct << "\t"

<< proc[i].tat << "\t" << proc[i].wt << endl;

}

return 0;

}

**ROUND ROBBIN WITH AT**

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

struct Process {

int pid;

int at;

int bt;

int rt;

int ct;

int tat;

int wt;

bool visited = false;

};

int main() {

int n, tq;

cout << "Enter number of processes: ";

cin >> n;

cout << "Enter Time Quantum: ";

cin >> tq;

vector<Process> proc(n);

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

proc[i].pid = i + 1;

cout << "Enter Arrival Time and Burst Time for Process " << proc[i].pid << ": ";

cin >> proc[i].at >> proc[i].bt;

proc[i].rt = proc[i].bt;

}

queue<int> q;

int time = 0, completed = 0;

q.push(0);

proc[0].visited = true;

while (completed < n) {

if (q.empty()) {

time++;

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

if (!proc[i].visited && proc[i].at <= time) {

q.push(i);

proc[i].visited = true;

} }

continue;

}

int idx = q.front();

q.pop();

if (proc[idx].rt > tq) {

time += tq;

proc[idx].rt -= tq;

} else {

time += proc[idx].rt;

proc[idx].rt = 0;

proc[idx].ct = time;

proc[idx].tat = proc[idx].ct - proc[idx].at;

proc[idx].wt = proc[idx].tat - proc[idx].bt;

completed++;

} // Add newly arrived processes to queue

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

if (!proc[i].visited && proc[i].at <= time) {

q.push(i);

proc[i].visited = true;

}

} // Re-add current process if it's not finished

if (proc[idx].rt > 0) {

q.push(idx);

}

} // Output

cout << "\nPID\tAT\tBT\tCT\tTAT\tWT\n";

for (auto& p : proc) {

cout << p.pid << "\t" << p.at << "\t" << p.bt << "\t" << p.ct

<< "\t" << p.tat << "\t" << p.wt << endl;

}

return 0;}

**FIFO PAGE REPLACEMENT**

#include <iostream>

#include <queue>

#include <unordered\_set>

using namespace std;

int main() {

int n, frameSize;

cout << "Enter the number of pages in the reference string: ";

cin >> n;

int pages[n];

cout << "Enter the page reference string:\n";

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

cin >> pages[i];

}

cout << "Enter the number of frames: ";

cin >> frameSize;

unordered\_set<int> frames;

queue<int> order;

int pageFaults = 0;

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

int currentPage = pages[i];

// If page is not already in frames

if (frames.find(currentPage) == frames.end()) {

if (frames.size() == frameSize) {

// Remove the oldest page

int pageToRemove = order.front();

order.pop();

frames.erase(pageToRemove);

}

// Insert the new page

frames.insert(currentPage);

order.push(currentPage);

pageFaults++;

cout << "Page " << currentPage << " caused a page fault.\n";

} else {

cout << "Page " << currentPage << " hit (no fault).\n";

}

}

cout << "\nTotal Page Faults = " << pageFaults << endl;

return 0;

}

**LRU**

#include <iostream>

#include <list>

#include <unordered\_map>

using namespace std;

int main() {

int n, frameSize;

cout << "Enter number of pages: ";

cin >> n;

int pages[n];

cout << "Enter the page reference string:\n";

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

cin >> pages[i];

}

cout << "Enter number of frames: ";

cin >> frameSize;

list<int> frameList; // To maintain order (least recently used at front)

unordered\_map<int, list<int>::iterator> pageMap;

int pageFaults = 0;

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

int page = pages[i];

// Page not in frame -> page fault

if (pageMap.find(page) == pageMap.end()) {

pageFaults++;

// If frame full, remove LRU page

if (frameList.size() == frameSize) {

int lru = frameList.back();

frameList.pop\_back();

pageMap.erase(lru);

}

} else {

// If hit, move page to front

frameList.erase(pageMap[page]);

}

// Insert (or reinsert) page to front

frameList.push\_front(page);

pageMap[page] = frameList.begin();

cout << "Page " << page << (pageMap.size() > frameSize ? " caused" : " hit (no)") << " a page fault.\n";

}

cout << "\nTotal Page Faults (LRU) = " << pageFaults << endl;

return 0;

}

**OPTIMAL REPLACEMENT**

#include <iostream>

#include <vector>

#include <unordered\_set>

using namespace std;

int predict(const vector<int>& pages, const unordered\_set<int>& frames, int index) {

int farthest = index, result = -1;

for (int page : frames) {

int j;

for (j = index; j < pages.size(); j++) {

if (pages[j] == page) {

if (j > farthest) {

farthest = j;

result = page;

}

break;

}

}

// If not found in future, return this page

if (j == pages.size())

return page;

}

// If all pages are used again, remove the one farthest in future

return (result == -1) ? \*frames.begin() : result;

}

int main() {

int n, frameSize;

cout << "Enter number of pages: ";

cin >> n;

vector<int> pages(n);

cout << "Enter the page reference string:\n";

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

cin >> pages[i];

}

cout << "Enter number of frames: ";

cin >> frameSize;

unordered\_set<int> frames;

int pageFaults = 0;

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

int current = pages[i];

if (frames.find(current) != frames.end()) {

cout << "Page " << current << " hit (no fault).\n";

continue;

}

if (frames.size() < frameSize) {

frames.insert(current);

} else {

int pageToRemove = predict(pages, frames, i + 1);

frames.erase(pageToRemove);

frames.insert(current);

}

pageFaults++;

cout << "Page " << current << " caused a page fault.\n";

}

cout << "\nTotal Page Faults (Optimal) = " << pageFaults << endl;

return 0;

}