**Lab no: 1**

**Date: 2024/08/22**

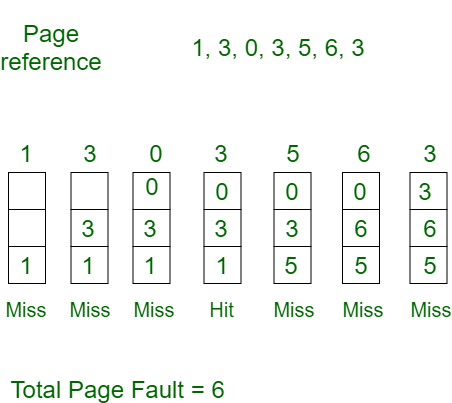


**Title: Write a program to calculate the no. of page fault for user input page reference and frame size using FIFO page replacement algorithm.**

**FIFO Page Replacement Algorithm:**

FIFO, also known as **First IN First OUT** Page Replacement Algorithm manages memory by replacing the oldest page in memory with a new page when a page fault occurs. It operates on a simple queue-based principle: the first page loaded into memory is the first to be replaced. While easy to implement, FIFO can lead to inefficiencies, such as Belady's Anomaly, where increasing the number of page frames paradoxically increases the number of page faults. Despite this, its simplicity makes it a viable option for certain memory management scenarios.

**For example**:



**Compiler: Dev C++**

**Language: C**

**Source code:**

#include <iostream>

#include <iomanip>

using namespace std;

bool check(const int frames[], int nFrame, int page)

{

for (int i = 0; i < nFrame; ++i)

{

if (frames[i] == page)

return true;

}

return false;

}

void display(const int frames[], int nFrame, const string& status)

{

for (int i = 0; i < nFrame; ++i)

{

if (frames[i] != -1)

cout << setw(5) << frames[i];

else

cout << setw(5) << "-";

}

cout << setw(5) <<"\t"<< status;

cout << endl;

}

int Replace(const int ref[], int nPage, int nFrame)

{

int frames[nFrame];

int front = 0;

int rear = 0;

int Miss = 0;

for (int i = 0; i < nFrame; ++i)

{

frames[i] = -1;

}

cout << setw(15) << "Reference";

for (int i = 0; i < nFrame; ++i)

{

cout << setw(5) << "F" << (i + 1);

}

cout << setw(10) << "Status" << endl;

cout << "---------------------------------------------" << endl;

for (int i = 0; i < nPage; ++i)

{

int currentPage = ref[i];

string status;

if (!check(frames, nFrame, currentPage))

{

Miss++;

status = "Page fault";

frames[rear] = currentPage;

rear = (rear + 1) % nFrame;

if (rear == front)

front = (front + 1) % nFrame;

}

else

status = "No fault";

cout << setw(15) << currentPage;

display(frames, nFrame, status);

}

return Miss;

}

int main()

{

int nFrame, nPage;

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* FIFO Page Replacement Algorithm \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" <<endl;

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

cin >> nFrame;

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

cin >> nPage;

int\* ref = new int[nPage];

cout << "Enter the reference string: ";

for (int i = 0; i < nPage; ++i)

{

cin >> ref[i];

}

int Miss = Replace(ref, nPage, nFrame);

cout << endl<<"Number of page faults: " << Miss << endl;

delete[] ref;

cout << endl<<"Compiled By: Prashan Shrestha." << endl;

return 0;

}

**Output:**

****

**Lab no: 2**

**Date: 2024/08/22**

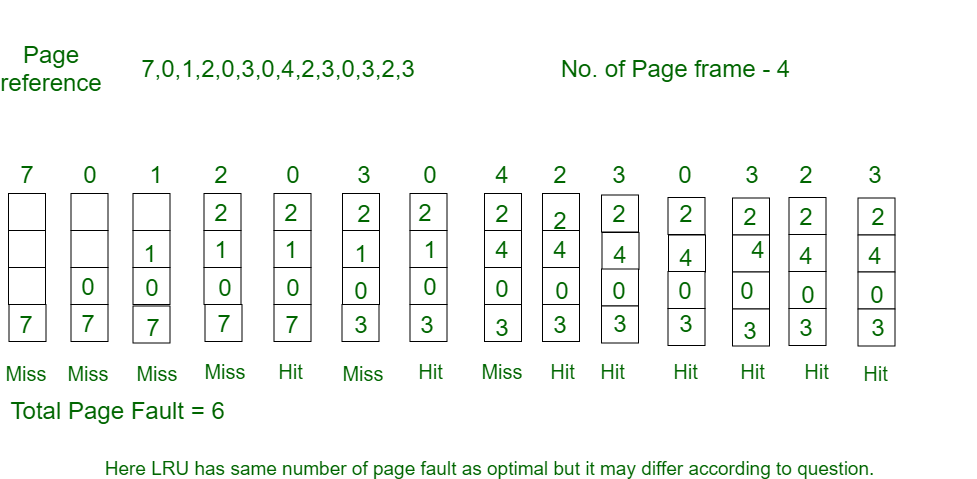


**Title: Write a program to calculate the no. of page fault for user input page reference and frame size using LRU page replacement algorithm.**

**LRU (Least Recently Used) Page Replacement Algorithm:**

The **Least Recently Used (LRU) Page Replacement Algorithm** is a page replacement strategy used in memory management, where the system replaces the page that has not been used for the longest period of time when a new page needs to be loaded into memory. This algorithm is based on the assumption that pages that have been heavily used in the past are more likely

**For example**:



**Compiler: Dev C++**

**Language: C**

**Source code:**

#include <iostream>

#include <iomanip>

using namespace std;

bool check(const int frames[], int nFrame, int page)

{

for (int i = 0; i < nFrame; ++i)

{

if (frames[i] == page)

return true;

}

return false;

}

void display(const int frames[], int nFrame, const string& status)

{

for (int i = 0; i < nFrame; ++i)

{

if (frames[i] != -1)

cout << setw(5) << frames[i];

else

cout << setw(5) << "-";

}

cout << setw(5) << "\t" << status;

cout << endl;

}

void updateFrames(int frames[], int nFrame, int page)

{

int pos = -1;

for (int i = 0; i < nFrame; ++i)

{

if (frames[i] == page)

{

pos = i;

break;

}

}

if (pos == -1)

{

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

{

frames[i] = frames[i + 1];

}

frames[nFrame - 1] = page;

}

else

{

for (int i = pos; i < nFrame - 1; ++i)

{

frames[i] = frames[i + 1];

}

frames[nFrame - 1] = page;

}

}

int ReplaceLRU(const int ref[], int nPage, int nFrame)

{

int frames[nFrame];

int Miss = 0;

for (int i = 0; i < nFrame; ++i)

{

frames[i] = -1;

}

cout << setw(15) << "Reference";

for (int i = 0; i < nFrame; ++i)

{

cout << setw(5) << "F" << (i + 1);

}

cout << setw(10) << "Status" << endl;

cout << "---------------------------------------------" << endl;

for (int i = 0; i < nPage; ++i)

{

int currentPage = ref[i];

string status;

if (!check(frames, nFrame, currentPage))

{

Miss++;

status = "Page fault";

} else

status = "No fault";

updateFrames(frames, nFrame, currentPage);

cout << setw(15) << currentPage;

display(frames, nFrame, status);

}

return Miss;

}

int main()

{

int nFrame, nPage;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*LRU Page Replacement Algorithm\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

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

cin >> nFrame;

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

cin >> nPage;

int\* ref = new int[nPage];

cout << "Enter the reference string: ";

for (int i = 0; i < nPage; ++i)

{

cin >> ref[i];

}

int Miss = ReplaceLRU(ref, nPage, nFrame);

cout << endl << "Number of page faults: " << Miss << endl;

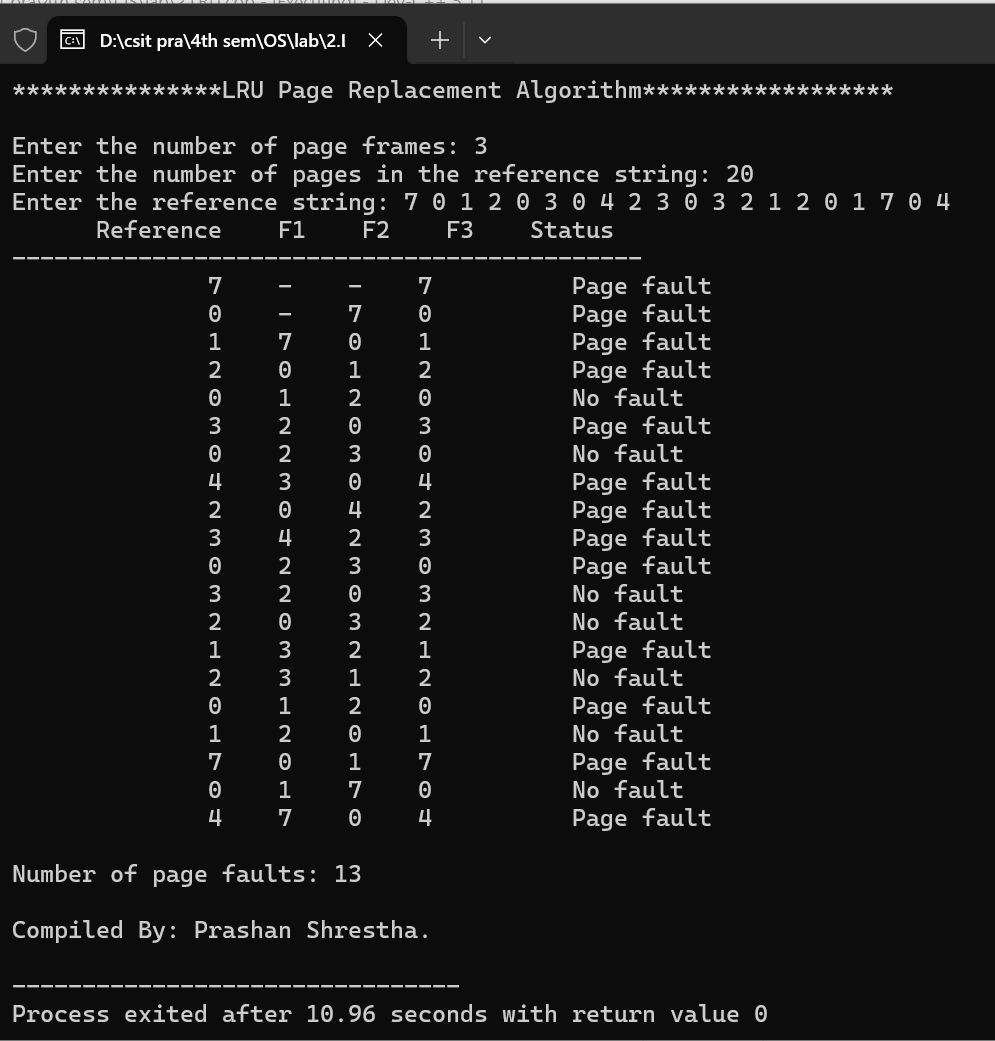
delete[] ref;

cout <<endl<< "Compiled By: Prashan Shrestha." << endl;

return 0;

}

**Output:**

****

**Lab no: 3**

**Date: 2024/08/22**

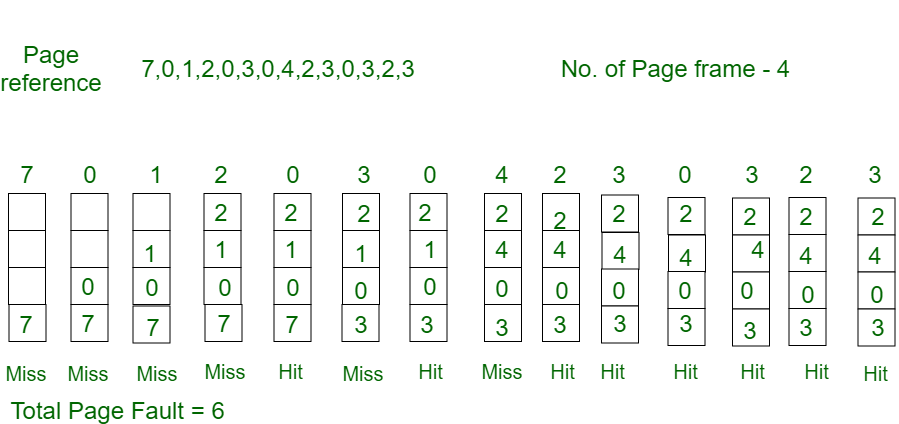


**Title: Write a program to calculate the no. of page fault for user input page reference and frame size using OPR page replacement algorithm.**

**OPR Page Replacement Algorithm:**

The **Optimal Page Replacement (OPR) Algorithm** is a theoretical page replacement strategy that provides the best possible performance by minimizing the number of page faults. The algorithm replaces the page that will not be used for the longest period in the future. Although it is impossible to implement in a real operating system since it requires future knowledge of the reference string, it serves as a benchmark to compare the performance of other page replacement algorithms.

**For example**:



**Compiler: Dev C++**

**Language: C**

**Source code:**

#include <iostream>

#include <iomanip>

using namespace std;

bool check(const int frames[], int nFrame, int page)

{

for (int i = 0; i < nFrame; ++i)

{

if (frames[i] == page)

return true;

}

return false;

}

void display(const int frames[], int nFrame, const string& status)

{

for (int i = 0; i < nFrame; ++i)

{

if (frames[i] != -1)

cout << setw(5) << frames[i];

else

cout << setw(5) << "-";

}

cout << setw(5) << "\t" << status;

cout << endl;

}

int predict(int ref[], int nPage, int frames[], int nFrame, int index)

{

int farthest = index;

int pos = -1;

for (int i = 0; i < nFrame; ++i)

{

int j;

for (j = index; j < nPage; ++j)

{

if (frames[i] == ref[j])

{

if (j > farthest)

{

farthest = j;

pos = i;

}

break;

}

}

if (j == nPage)

return i;

}

return (pos == -1) ? 0 : pos;

}

int ReplaceOPR(int ref[], int nPage, int nFrame)

{

int frames[nFrame];

int Miss = 0;

for (int i = 0; i < nFrame; ++i)

{

frames[i] = -1;

}

cout << setw(15) << "Reference";

for (int i = 0; i < nFrame; ++i)

{

cout << setw(5) << "F" << (i + 1);

}

cout << setw(10) << "Status" << endl;

cout << "---------------------------------------------" << endl;

for (int i = 0; i < nPage; ++i)

{

int currentPage = ref[i];

string status;

if (!check(frames, nFrame, currentPage))

{

Miss++;

if (i < nFrame)

frames[i] = currentPage;

else {

int pos = predict(ref, nPage, frames, nFrame, i + 1);

frames[pos] = currentPage;

}

status = "Page fault";

} else

status = "No fault";

cout << setw(15) << currentPage;

display(frames, nFrame, status);

}

return Miss;

}

int main()

{

int nFrame, nPage;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Optimal Page Replacement Algorithm\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

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

cin >> nFrame;

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

cin >> nPage;

int\* ref = new int[nPage];

cout << "Enter the reference string: ";

for (int i = 0; i < nPage; ++i)

{

cin >> ref[i];

}

int Miss = ReplaceOPR(ref, nPage, nFrame);

cout << endl << "Number of page faults: " << Miss << endl;

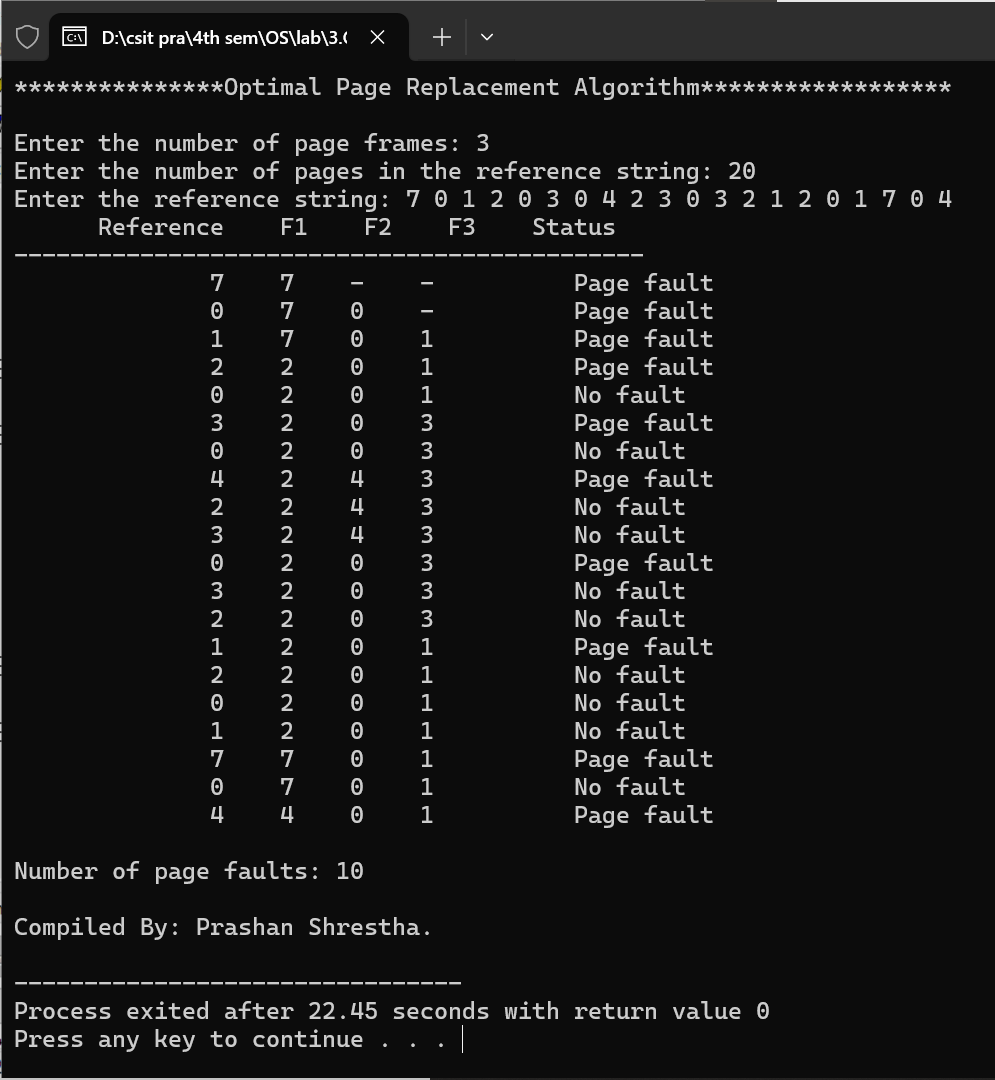
delete[] ref;

cout <<endl<< "Compiled By: Prashan Shrestha." << endl;

return 0;

}

**Output:**

****

**Lab no: 4**

**Date: 2024/08/22**

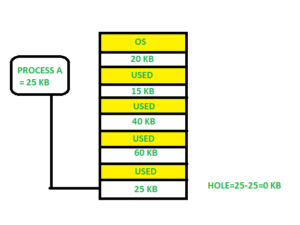


**Title: Write a program to simulate memory allocation strategy for user input free space list and incoming process request using Best Fit Allocation.**

**Best Fit Allocation:**

The Best Fit Allocation algorithm is a memory management technique used in operating systems to allocate the smallest available memory partition that is large enough to satisfy a process's requirements. This strategy aims to reduce wasted space in memory, known as fragmentation, by ensuring that each process fits into the smallest possible partition. However, it can sometimes lead to inefficiencies if smaller gaps in memory are left unallocated, making it harder to fit future processes.

**For example**:



**Compiler: Dev C++**

**Language: C**

**Source code:**

#include <iostream>

#include <iomanip>

using namespace std;

#define MAX 100

void printTable(int processSize[], int allocation[], int n)

{

cout << "\nProcess No.\tProcess Size\tPartition No." << endl;

cout << "-------------------------------------------------" << endl;

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

{

cout << setw(2) << (i + 1)

<< setw(22) << processSize[i];

if (allocation[i] != -1)

cout << setw(12) << (allocation[i] + 1) << endl;

else

cout << " - " << endl;

}

}

void bestFit(int partitionSize[], int m, int processSize[], int n)

{

int allocation[n];

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

{

allocation[i] = -1;

}

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

{

int bestIdx = -1;

for (int j = 0; j < m; ++j)

{

if (partitionSize[j] >= processSize[i])

{

if (bestIdx == -1 || partitionSize[bestIdx] > partitionSize[j])

bestIdx = j;

}

}

if (bestIdx != -1)

{

allocation[i] = bestIdx;

partitionSize[bestIdx] -= processSize[i];

}

}

printTable(processSize, allocation, n);

}

int main()

{

cout << "\t\t\*\*\*\*\*\*\*\*\*\*\*\* Best Fit Algorithm \*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

int partitionSize[MAX], processSize[MAX];

int m, n;

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

cin >> m;

cout << "\nEnter the size of each partition:\n";

for (int i = 0; i < m; ++i)

{

cout << "Partition " << (i + 1) << ": ";

cin >> partitionSize[i];

}

cout << "\nEnter the number of processes: ";

cin >> n;

cout << "\nEnter the size of each process:\n";

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

{

cout << "Process " << (i + 1) << ": ";

cin >> processSize[i];

}

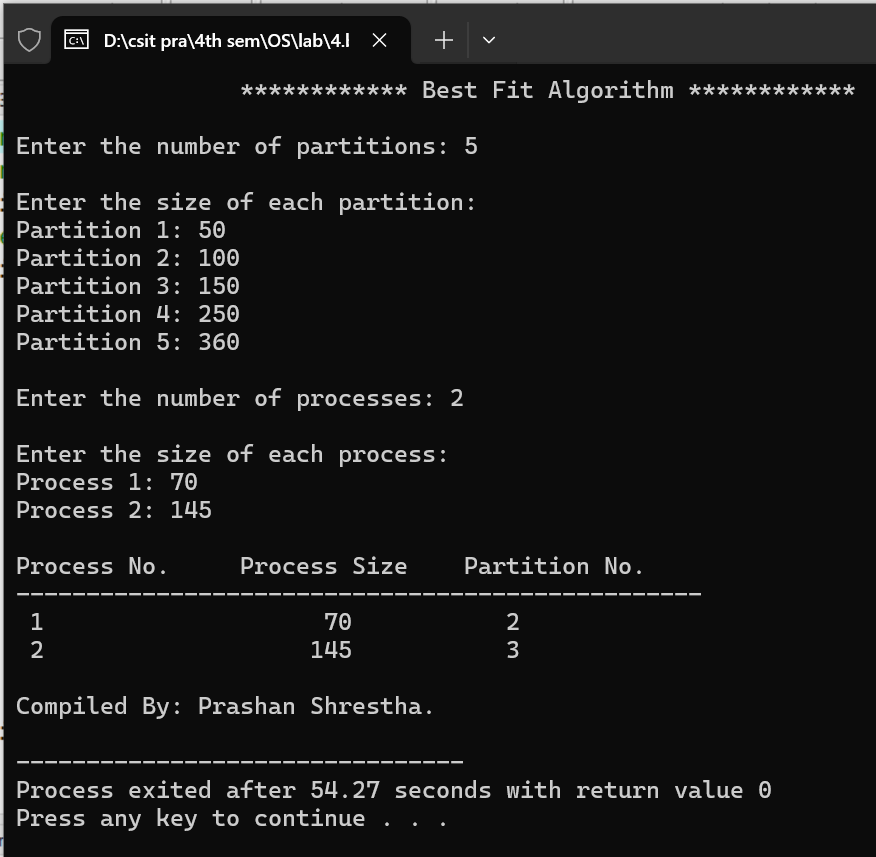
bestFit(partitionSize, m, processSize, n);

cout << endl<<"Compiled By: Prashan Shrestha." << endl;

return 0;

}

**Output:**

****

**Lab no: 5**

**Date: 2024/08/22**

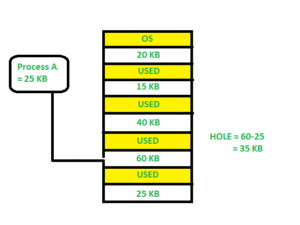


**Title: Write a program to simulate memory allocation strategy for user input free space list and incoming process request using Worst Fit Allocation.**

**Worst Fit Allocation:**

The Worst Fit Allocation algorithm is a memory management technique used in operating systems to allocate the largest available memory partition to a process. The idea behind this strategy is to leave the largest possible free space after the allocation, which might be more useful for future processes that require large blocks of memory. However, this can lead to inefficient memory utilization and increased fragmentation over time.

**For example**:



**Compiler: Dev C++**

**Language: C**

**Source code:**

#include <iostream>

#include <iomanip>

using namespace std;

#define MAX 100

void printTable(int processSize[], int allocation[], int n)

{

cout << "\nProcess No.\tProcess Size\tPartition No." << endl;

cout << "-------------------------------------------------" << endl;

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

cout << setw(2) << (i + 1)

<< setw(22) << processSize[i];

if (allocation[i] != -1)

cout << setw(12) << (allocation[i] + 1) << endl;

else

cout << " - " << endl;

}

}

void worstFit(int partitionSize[], int m, int processSize[], int n)

{

int allocation[n];

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

{

allocation[i] = -1;

}

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

{

int worstIdx = -1;

for (int j = 0; j < m; ++j)

{

if (partitionSize[j] >= processSize[i])

{

if (worstIdx == -1 || partitionSize[worstIdx] < partitionSize[j])

worstIdx = j;

}

}

if (worstIdx != -1)

{

allocation[i] = worstIdx;

partitionSize[worstIdx] -= processSize[i];

}

}

printTable(processSize, allocation, n);

}

int main()

{

cout << "\t\t\*\*\*\*\*\*\*\*\*\* Worst Fit Algorithm \*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

int partitionSize[MAX], processSize[MAX];

int m, n;

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

cin >> m;

cout << "\nEnter the size of each partition:\n";

for (int i = 0; i < m; ++i)

{

cout << "Partition " << (i + 1) << ": ";

cin >> partitionSize[i];

}

cout << "\nEnter the number of processes: ";

cin >> n;

cout << "\nEnter the size of each process:\n";

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

{

cout << "Process " << (i + 1) << ": ";

cin >> processSize[i];

}

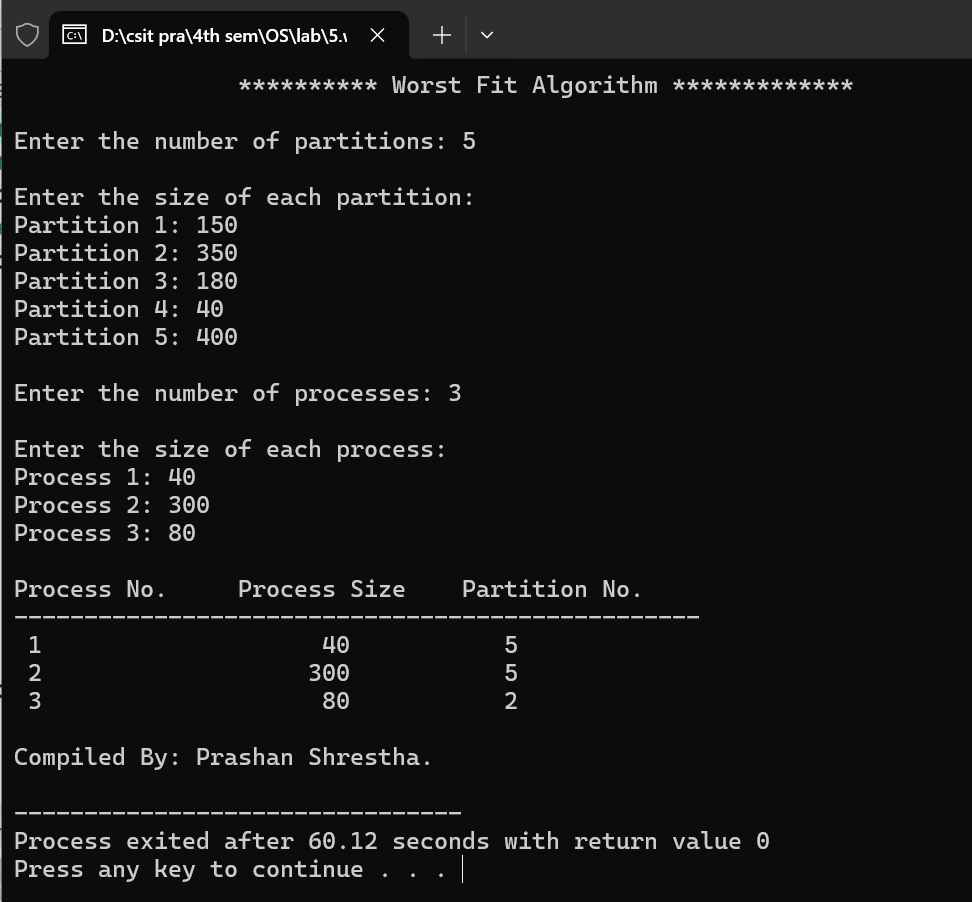
worstFit(partitionSize, m, processSize, n);

cout << endl<<"Compiled By: Prashan Shrestha." << endl;

return 0;

}

**Output:**



**Lab no: 6**

**Date: 2024/08/22**

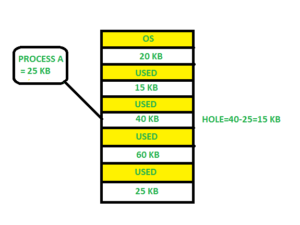


**Title: Write a program to simulate memory allocation strategy for user input free space list and incoming process request using First Fit Allocation.**

**First Fit Allocation:**

**The First Fit Allocation algorithm is a memory management technique used in operating systems where the first available memory partition that is large enough to accommodate a process is selected for allocation. The process scans the memory from the beginning and allocates the process to the first suitable partition it encounters. This approach is straightforward and fast, but it can lead to fragmentation over time as memory is allocated in the order of arrival rather than in a more optimized manner.**

**For example**:



**Compiler: Dev C++**

**Language: C**

**Source code:**

#include <iostream>

#include <iomanip>

using namespace std;

#define MAX 100

void printTable(int processSize[], int allocation[], int n)

{

cout << "\nProcess No.\tProcess Size\tPartition No." << endl;

cout << "-------------------------------------------------" << endl;

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

cout << setw(2) << (i + 1)

<< setw(22) << processSize[i];

if (allocation[i] != -1)

cout << setw(12) << (allocation[i] + 1) << endl;

else

cout << " - " << endl;

}

}

void firstFit(int partitionSize[], int m, int processSize[], int n)

{

int allocation[n];

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

{

allocation[i] = -1;

}

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

{

for (int j = 0; j < m; ++j)

{

if (partitionSize[j] >= processSize[i])

{

allocation[i] = j;

partitionSize[j] -= processSize[i];

break;

}

}

}

printTable(processSize, allocation, n);

}

int main()

{

cout << "\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*First Fit Algorithm\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

int partitionSize[MAX], processSize[MAX];

int m, n;

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

cin >> m;

cout << "\nEnter the size of each partition:\n";

for (int i = 0; i < m; ++i)

{

cout << "Partition " << (i + 1) << ": ";

cin >> partitionSize[i];

}

cout << "\nEnter the number of processes: ";

cin >> n;

cout << "\nEnter the size of each process:\n";

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

{

cout << "Process " << (i + 1) << ": ";

cin >> processSize[i];

}

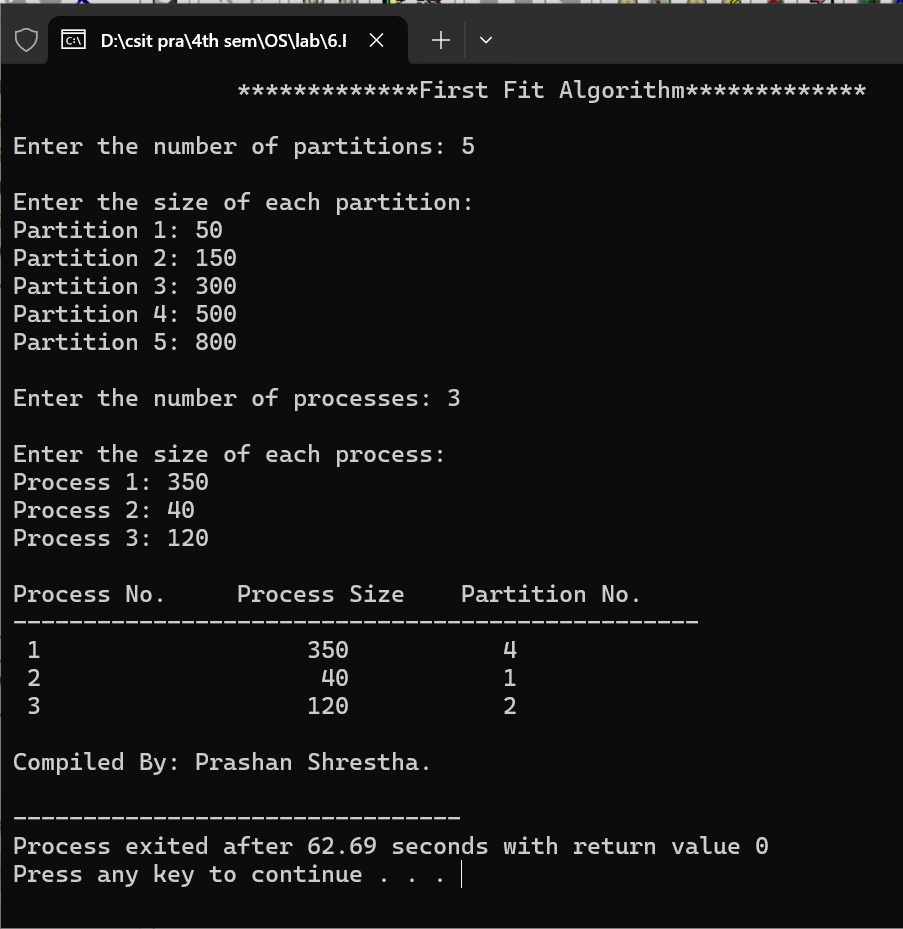
firstFit(partitionSize, m, processSize, n);

cout << endl<<"Compiled By: Prashan Shrestha." << endl;

return 0;

}

**Output:**

****

**Lab no: 7**

**Date: 2024/08/22**



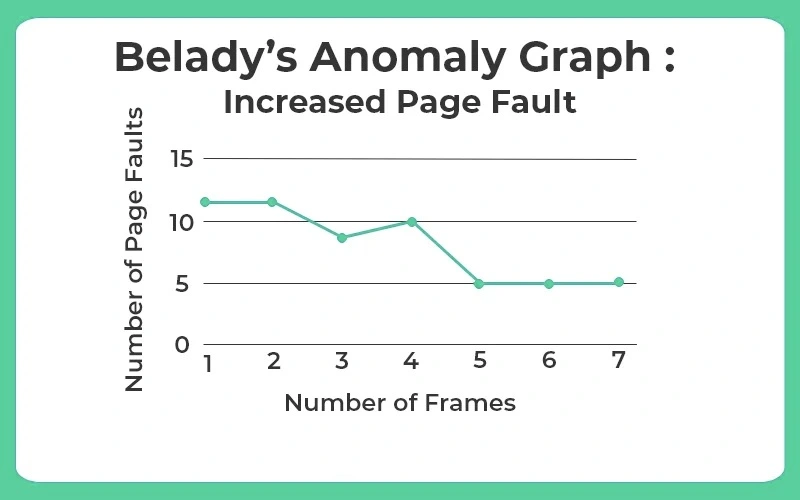
**Title: Write a program to demonstrate Belady’s Anomaly for user input page reference string.**

**Belady’s Anomaly:**

Belady’s Anomaly is a counterintuitive behavior observed in some page replacement algorithms, notably the First-In-First-Out (FIFO) algorithm, where increasing the number of page frames can paradoxically lead to an increase in the number of page faults. This anomaly defies the intuitive expectation that more page frames should reduce the number of page faults, as it would seemingly provide more space to hold pages and reduce the likelihood of replacements.

Belady's Anomaly highlights the limitations of FIFO and underscores the complexity of optimal page replacement strategies. It was first identified by Laszlo Belady in 1969 and serves as a critical concept in the study of operating systems and memory management.

**For example:**



**Compiler: Dev C++**

**Language: C**

**Source code:**

#include <iostream>

#include <iomanip>

using namespace std;

bool isPresent(int frames[], int nFrame, int page)

{

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

if (frames[i] == page) {

return true;

}

}

return false;

}

void display(const int frames[], int nFrame, const string& status)

{

for (int i = 0; i < nFrame; ++i)

{

if (frames[i] != -1)

cout << setw(5) << frames[i];

else

cout << setw(5) << "-";

}

cout << setw(15) << status;

cout << endl;

}

int FIFOPageReplacement(int ref[], int nPage, int nFrame)

{

int frames[nFrame];

int pageFaults = 0;

int index = 0;

for (int i = 0; i < nFrame; ++i)

{

frames[i] = -1;

}

cout << setw(15) << "Reference";

for (int i = 0; i < nFrame; ++i)

{

cout << setw(5) << "F" << (i + 1);

}

cout << setw(10) << "Status" << endl;

cout << "---------------------------------------------" << endl;

for (int i = 0; i < nPage; ++i)

{

int currentPage = ref[i];

string status;

if (!isPresent(frames, nFrame, currentPage))

{

status = "Page fault";

frames[index] = currentPage;

index = (index + 1) % nFrame;

pageFaults++;

} else

status = "No fault";

cout << setw(15) << currentPage;

display(frames, nFrame, status);

}

return pageFaults;

}

int main()

{

int nPage;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Belady's Anomaly \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

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

cin >> nPage;

int ref[nPage];

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

for (int i = 0; i < nPage; ++i)

{

cin >> ref[i];

}

int startFrame, endFrame;

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

cin >> startFrame;

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

cin >> endFrame;

for (int nFrame = startFrame; nFrame <= endFrame; ++nFrame)

{

cout << "\nNumber of frames: " << nFrame << endl;

int pageFaults = FIFOPageReplacement(ref, nPage, nFrame);

cout << "Total Page Faults with " << nFrame << " frames: " << pageFaults << endl;

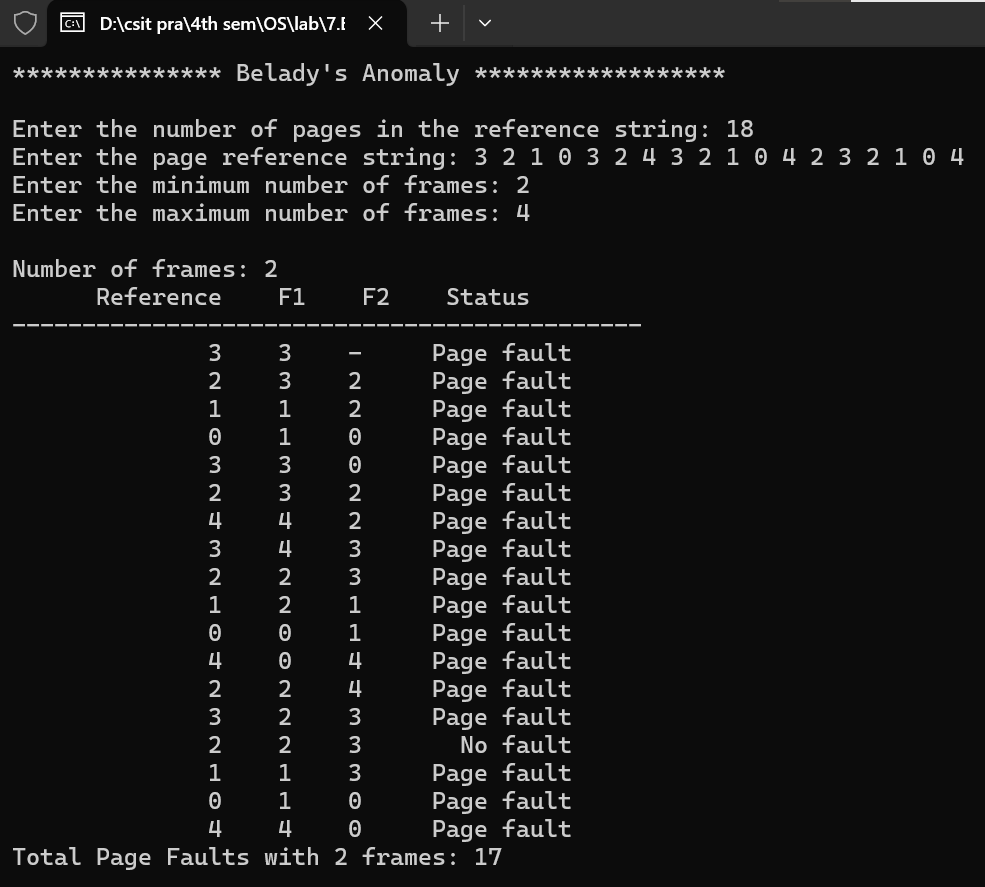
}

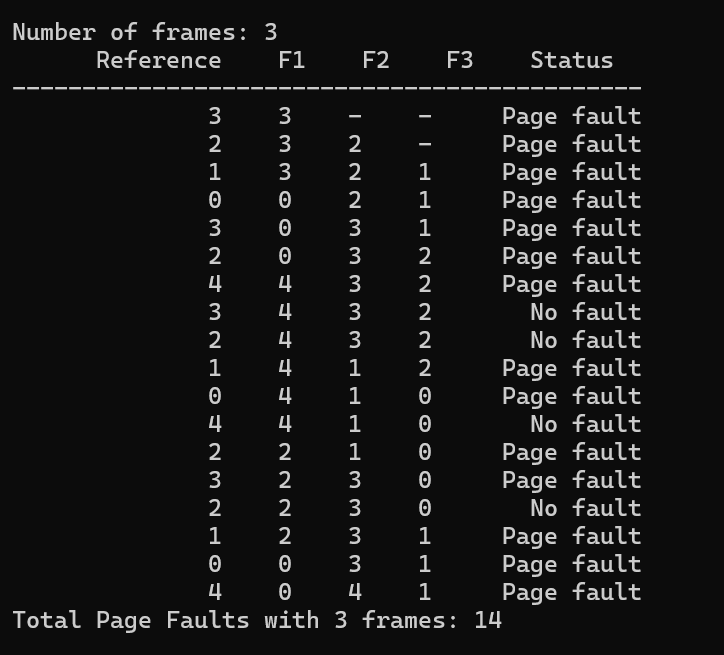
cout << endl<<"Compiled By: Prashan Shrestha." << endl;

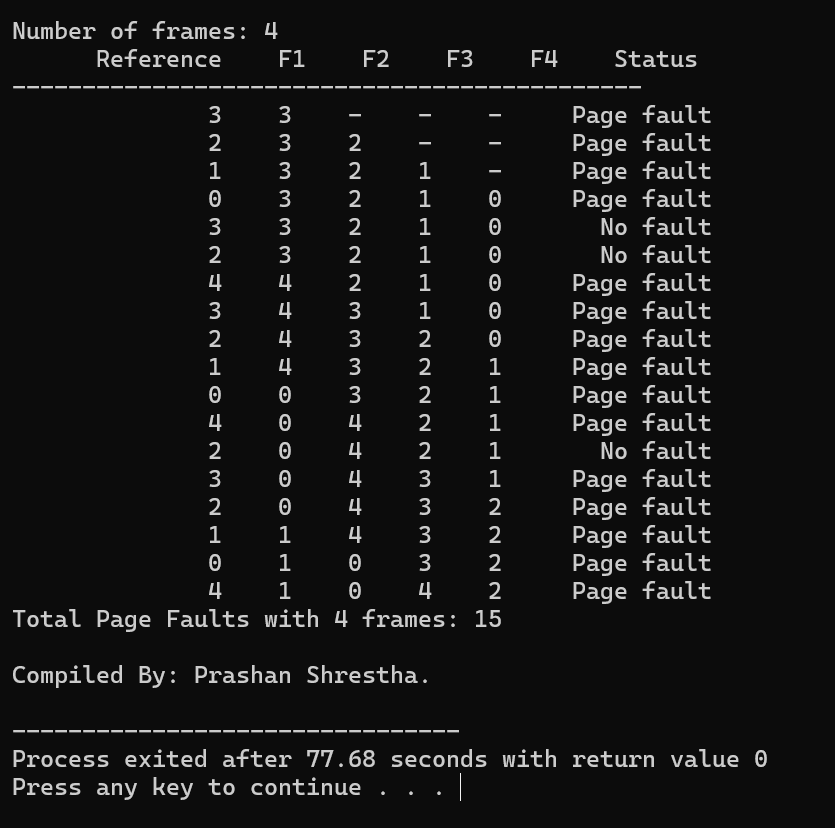
return 0;

}

**Output:**







From the above output,

|  |  |
| --- | --- |
| Frame size | Page Fault |
| 2 | 17 |
| 3 | 14 |
| 4 | 15 |

We can see that the number of page fault is decreasing in frame size 3, while the number of page fault is increased in frame size 4 even if the size is more. This is **Belady’s anomaly.**

**Lab no: 8**

**Date: 2024/9/20**



**Title: Write a program to calculate average Turn Around Time, and Waiting Time for user input processes using FCFS and their parameters. Also draw it’s Gantt chart**

**First-Come, First-Served (FCFS) Scheduling:**

**First-Come, First-Served (FCFS)** is one of the simplest scheduling algorithms used in operating systems to manage processes. In FCFS scheduling, the process that arrives first is executed first. It operates in a non-preemptive manner, meaning once a process starts executing, it runs to completion before the next process begins. FCFS follows the FIFO (First-In-First-Out) principle, and subsequent processes wait in the queue until the CPU becomes available.

**Key Concepts:**

1. **Arrival Time:**
   * The time at which a process arrives in the ready queue.
2. **Burst Time:**
   * The total time required by a process for execution on the CPU.
3. **Turnaround Time:**
   * The total time taken from the arrival of the process to its completion.
   * Formula:

Turnaround Time=Completion Time−Arrival Time

1. **Waiting Time:**
   * The total time a process spends waiting in the ready queue before it gets executed.
   * Formula:

Waiting Time=Turnaround Time−Burst Time

**Compiler: Dev C++**

**Language: C++**

**Source code:**

#include <iostream>

#include <iomanip>

using namespace std;

int main() {

    cout << "\t\t\*\*\*\*\*\*\*\*\* FIRST COME, FIRST SERVE ALGORITHM \*\*\*\*\*\*\*\*\*\*\*\n\n";

    int n;

    cout << "Enter number of processes: ";

    cin >> n;

    int bt[20], wt[20], tat[20], p[20];

    float totalWT = 0, totalTAT = 0;

    cout << "Enter Burst Time for each process:\n";

    for (int i = 0; i < n; i++) {

        cout << "P" << i + 1 << " Burst Time: ";

        cin >> bt[i];

        p[i] = i + 1;

    }

    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];

    }

    for (int i = 0; i < n; i++) {

        totalWT += wt[i];

        totalTAT += tat[i];

    }

    float avgWT = totalWT / n;

    float avgTAT = totalTAT / n;

    cout << "\nProcess No.\tBurst Time\tTurnaround Time\tWaiting Time" << endl;

    cout << "-------------------------------------------------" << endl;

    for (int i = 0; i < n; i++) {

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

    }

    cout << "-------------------------------------------------" << endl;

    cout << "\nAverage Waiting Time: " << fixed << setprecision(2) << avgWT << endl;

    cout << "Average Turnaround Time: " << fixed << setprecision(2) << avgTAT << endl;

    cout << "\nGantt Chart:\n\n";

    cout << "+";

    for (int i = 0; i < n; i++) {

        cout << "-----+";

    }

    cout << "\n|";

    for (int i = 0; i < n; i++) {

        cout << " P" << p[i] << "  |";

    }

    cout << "\n+";

    for (int i = 0; i < n; i++) {

        cout << "-----+";

    }

    cout << "\n0";

    int totalTime = 0;

    for (int i = 0; i < n; i++) {

        totalTime += bt[i];

        cout << setw(6) << totalTime;

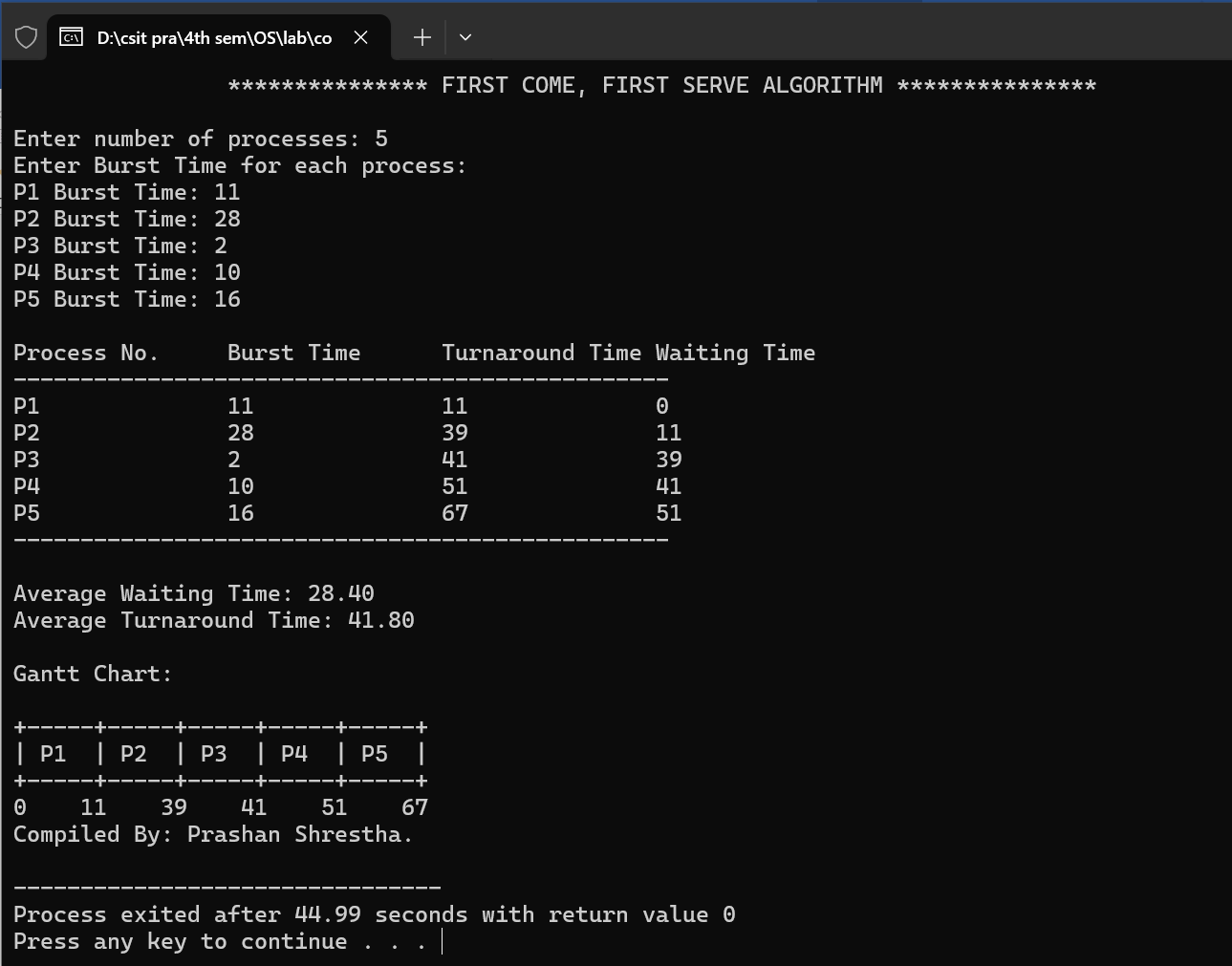
    }

    cout << endl << "Compiled By: Prashan Shrestha." << endl;

    return 0;

}

**Output:**



**Lab no: 9**

**Date: 2024/9/20**

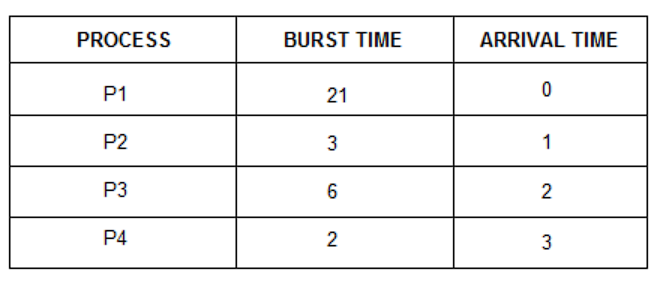


**Title: Write a program to calculate average Turn Around Time, and Waiting Time for user input processes using SJF and their parameters. Also draw it’s Gantt chart.**

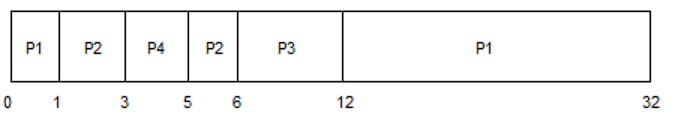
**Shortest Job First (SJF) Scheduling:**

**Shortest Job First (SJF)** is a CPU scheduling algorithm that selects the process with the shortest burst time for execution. It minimizes average waiting and turnaround times, making it more efficient than FCFS. SJF can be non**-**preemptive, where processes run to completion, or preemptive, allowing shorter processes to interrupt ongoing ones. While it optimizes performance, SJF may cause starvation for longer processes if shorter ones keep arriving. It is ideal for systems where burst times are predictable.

**Example:**



Gantt chart following SJF:



**Compiler: Dev C++**

**Language: C++**

**Source code:**

#include <iostream>

#include <iomanip>

using namespace std;

int main() {

    cout << "\t\t\*\*\*\*\*\*\*\* SHORTEST JOB FIRST ALGORITHM \*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

    int n;

    cout << "Enter number of processes: ";

    cin >> n;

    int bt[20], wt[20], tat[20], p[20];

    float totalWT = 0, totalTAT = 0;

    cout << "Enter Burst Time for each process:\n";

    for (int i = 0; i < n; i++) {

        cout << "P" << i + 1 << " Burst Time: ";

        cin >> bt[i];

        p[i] = i + 1;

    }

    for (int i = 0; i < n - 1; i++) {

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

            if (bt[j] < bt[i]) {

                swap(bt[i], bt[j]);

                swap(p[i], p[j]);

            }

        }

    }

    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];

    }

    for (int i = 0; i < n; i++) {

        totalWT += wt[i];

        totalTAT += tat[i];

    }

    float avgWT = totalWT / n;

    float avgTAT = totalTAT / n;

    cout << "\nProcess No.\tBurst Time\tTurnaround Time\tWaiting Time" << endl;

    cout << "-------------------------------------------------" << endl;

    for (int i = 0; i < n; i++) {

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

    }

    cout << "-------------------------------------------------" << endl;

    cout << "\nAverage Waiting Time: " << fixed << setprecision(2) << avgWT << endl;

    cout << "Average Turnaround Time: " << fixed << setprecision(2) << avgTAT << endl;

    cout << "\nGantt Chart:\n\n";

    cout << "+";

    for (int i = 0; i < n; i++) {

        cout << "-----+";

    }

    cout << "\n|";

    for (int i = 0; i < n; i++) {

        cout << " P" << p[i] << "  |";

    }

    cout << "\n+";

    for (int i = 0; i < n; i++) {

        cout << "-----+";

    }

    cout << "\n0";

    int totalTime = 0;

    for (int i = 0; i < n; i++) {

        totalTime += bt[i];

        cout << setw(6) << totalTime;

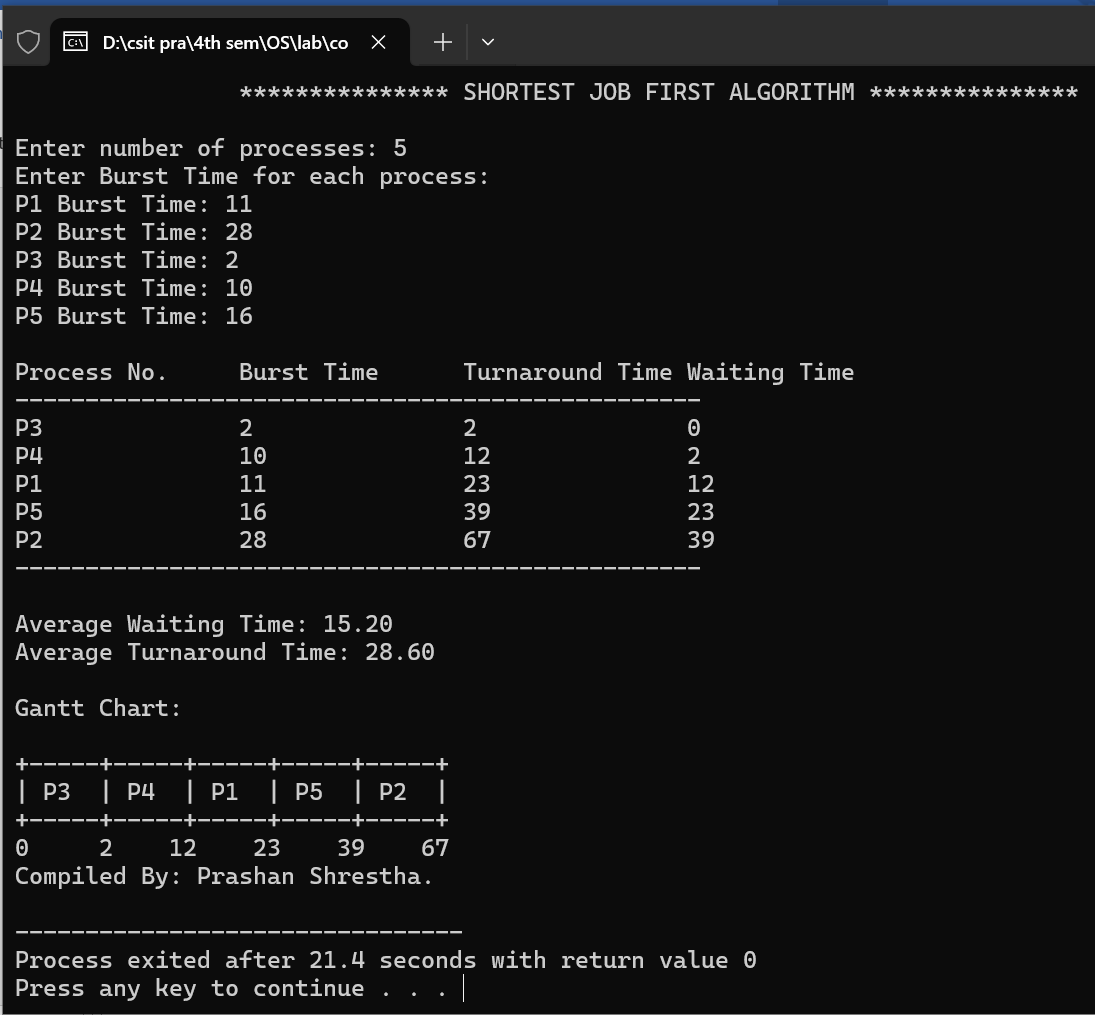
    }

    cout << endl<<"Compiled By: Prashan Shrestha." << endl;

    return 0;

}

**Output:**



**Lab no: 10**

**Date: 2024/9/20**



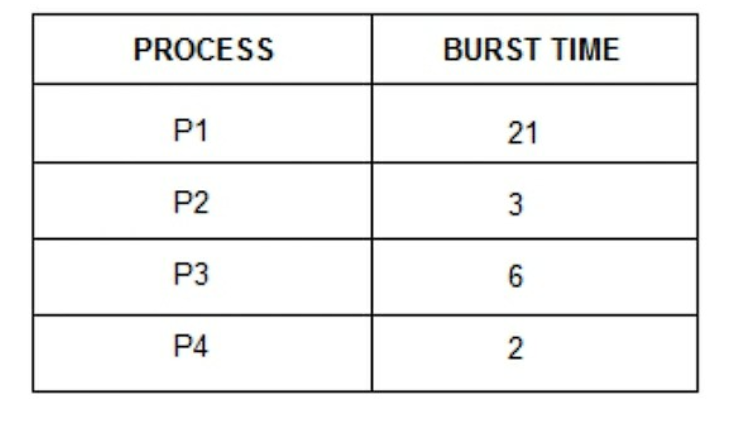
**Title: Write a program to calculate average Turn Around Time, and Waiting Time for user input processes using RR and their parameters. Also draw it’s Gantt chart**

**Round Robin (RR) Scheduling:**

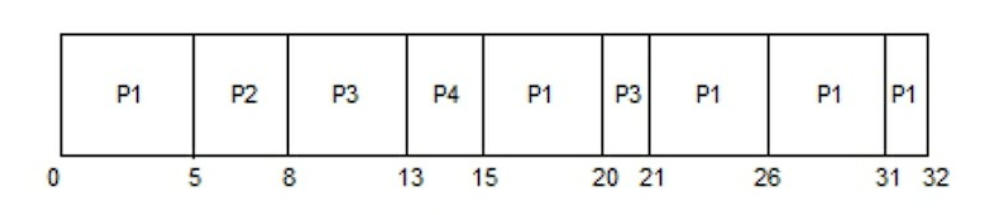
**Round Robin (RR)** is a preemptive CPU scheduling algorithm designed to provide fair and time-shared process execution. Each process in the ready queue is assigned a fixed time slice or quantum, during which it can execute. If a process does not finish within its quantum, it is preempted and moved to the back of the queue, allowing other processes to execute.

**Characteristics of Round Robin:**

1. **Preemptive:** Ensures no single process monopolizes the CPU.
2. **Time Quantum:** A fixed time slice determines how long each process can execute in one turn.
3. **Fairness:** All processes get an equal opportunity to run, avoiding starvation.



**Gantt Chart:**



**Compiler: Dev C++**

**Language: C++**

**Source code:**

#include <iostream>

#include <iomanip>

using namespace std;

int main() {

    cout << "\t\t\*\*\*\*\*\*\*\*\*\* ROUND ROBIN SCHEDULING ALGORITHM \*\*\*\*\*\*\*\*\*\*n\n";

    int n, quantum;

    cout << "Enter number of processes: ";

    cin >> n;

    int bt[20], rem\_bt[20], wt[20], tat[20], p[20];

    float totalWT = 0, totalTAT = 0;

    cout << "Enter Burst Time for each process:\n";

    for (int i = 0; i < n; i++) {

        cout << "P" << i + 1 << " Burst Time: ";

        cin >> bt[i];

        rem\_bt[i] = bt[i];

        p[i] = i + 1;

    }

    cout << "Enter Time Quantum: ";

    cin >> quantum;

    int t = 0;

    bool done;

    int gantt[100], gantt\_time[100], g\_idx = 0;

    gantt\_time[g\_idx] = 0;

    do {

        done = true;

        for (int i = 0; i < n; i++) {

            if (rem\_bt[i] > 0) {

                done = false;

                gantt[g\_idx++] = p[i];

                if (rem\_bt[i] > quantum) {

                    t += quantum;

                    rem\_bt[i] -= quantum;

                } else {

                    t += rem\_bt[i];

                    rem\_bt[i] = 0;

                    tat[i] = t;

                }

                gantt\_time[g\_idx] = t;

            }

        }

    } while (!done);

    for (int i = 0; i < n; i++) {

        wt[i] = tat[i] - bt[i];

        totalWT += wt[i];

        totalTAT += tat[i];

    }

    float avgWT = totalWT / n;

    float avgTAT = totalTAT / n;

    cout << "\nGantt Chart:\n\n";

    cout << "+";

    for (int i = 0; i < g\_idx; i++) {

        cout << "-----+";

    }

    cout << "\n|";

    for (int i = 0; i < g\_idx; i++) {

        cout << " P" << gantt[i] << "  |";

    }

    cout << "\n+";

    for (int i = 0; i < g\_idx; i++) {

        cout << "-----+";

    }

    cout << "\n" << gantt\_time[0];

    for (int i = 1; i <= g\_idx; i++) {

        cout << setw(6) << gantt\_time[i];

    }

    cout << endl;

    cout << "\nProcess No.\tBurst Time\tTurnaround Time\tWaiting Time" << endl;

    cout << "-------------------------------------------------" << endl;

    for (int i = 0; i < n; i++) {

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

    }

    cout << "-------------------------------------------------" << endl;

    cout << "\nAverage Waiting Time: " << fixed << setprecision(2) << avgWT << endl;

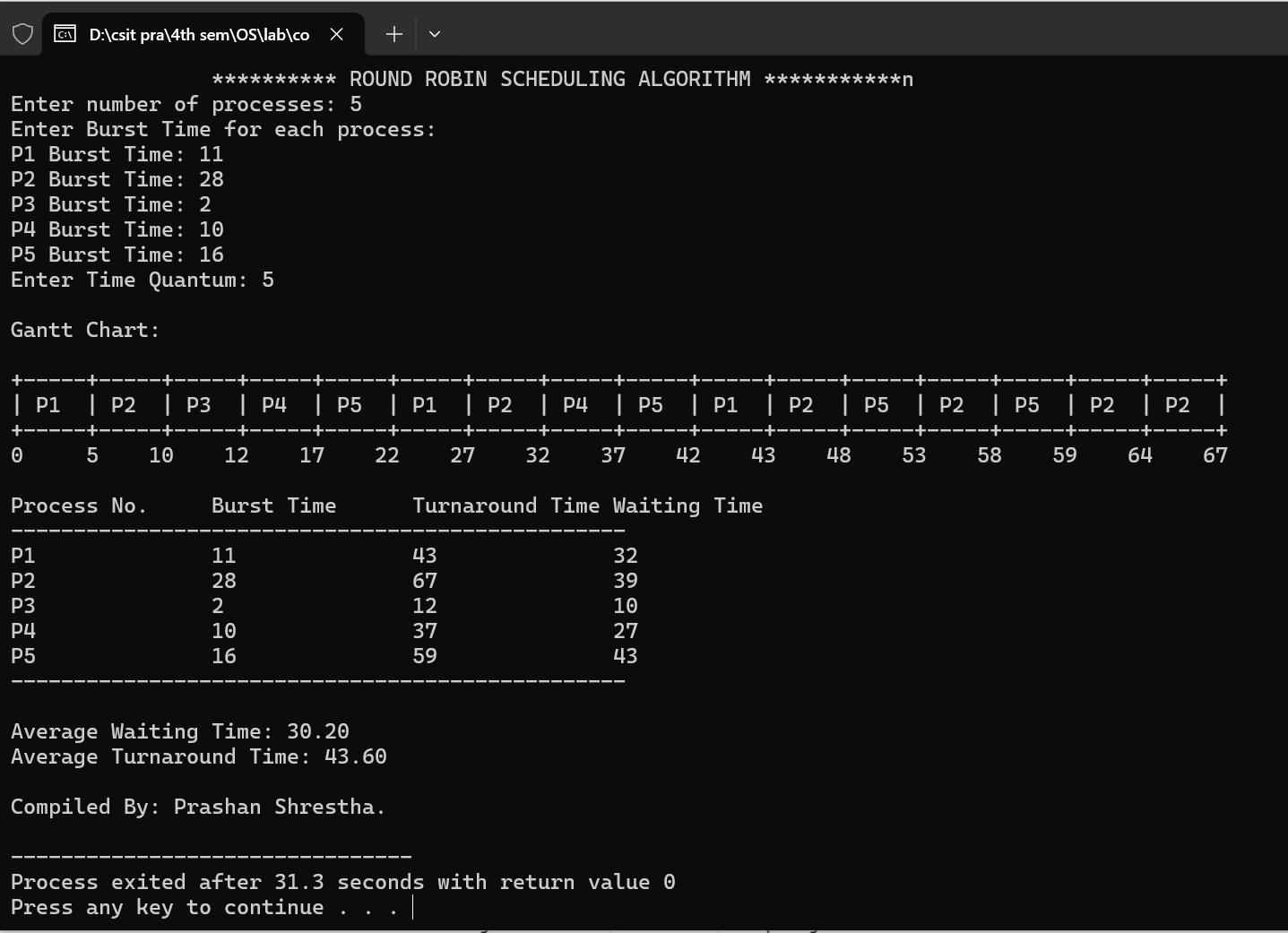
    cout << "Average Turnaround Time: " << fixed << setprecision(2) << avgTAT << endl;

    cout << "\nCompiled By: Prashan Shrestha." << endl;

    return 0;

}

**Output:**



**Lab no: 11**

**Date: 2024/9/20**



**Title: Write a program to calculate average Turn Around Time, and Waiting Time for user input processes using Priority Scheduling and their parameters. Also draw it’s Gantt chart**

**Priority Scheduling:**

**Priority Scheduling** is a CPU scheduling algorithm where each process is assigned a priority, and the CPU is allocated to the process with the highest priority. If two processes have the same priority, scheduling can follow First-Come, First-Served (FCFS) as a tie-breaking rule. This algorithm can be either preemptive or non-preemptive, depending on whether a higher-priority process can interrupt an ongoing one.

**Types of Priority Scheduling:**

1. **Preemptive Priority Scheduling:**If a new process with a higher priority arrives, it preempts the currently executing process.
2. **Non-Preemptive Priority Scheduling:**The CPU continues executing the current process until it finishes, even if a higher-priority process arrives.

**Advantages:**

* Allows critical processes to be executed quickly.
* Improves system responsiveness for high-priority tasks.

**Disadvantages:**

* **Starvation**: Low-priority processes may be delayed indefinitely if high-priority processes keep arriving.
* Requires careful priority assignment to balance performance.

**Compiler: Dev C++**

**Language: C++**

**Source code:**

#include <iostream>

#include <iomanip>

using namespace std;

int main() {

    cout << "\t\t\*\*\*\*\*\*\*\*\* PRIORITY SCHEDULING ALGORITHM \*\*\*\*\*\*\*\*\*\*\n\n";

    int n;

    cout << "Enter number of processes: ";

    cin >> n;

    int bt[20], priority[20], wt[20], tat[20], p[20];

    float totalWT = 0, totalTAT = 0;

    cout << "Enter Burst Time and Priority for each process:\n";

    for (int i = 0; i < n; i++) {

        cout << "P" << i + 1 << " Burst Time: ";

        cin >> bt[i];

        cout << "P" << i + 1 << " Priority: ";

        cin >> priority[i];

        p[i] = i + 1;

    }

    for (int i = 0; i < n - 1; i++) {

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

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

                swap(bt[i], bt[j]);

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

                swap(p[i], p[j]);

            }

        }

    }

    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];

    }

    for (int i = 0; i < n; i++) {

        totalWT += wt[i];

        totalTAT += tat[i];

    }

    float avgWT = totalWT / n;

    float avgTAT = totalTAT / n;

    cout << "\nProcess No.\tBurst Time\tPriority\tTurnaround Time\tWaiting Time" << endl;

    cout << "---------------------------------------------------------------------" << endl;

    for (int i = 0; i < n; i++) {

        cout << "P" << p[i] << "\t\t" << bt[i] << "\t\t" << priority[i] << "\t\t" << tat[i] << "\t\t" << wt[i] << endl;

    }

    cout << "---------------------------------------------------------------------" << endl;

    cout << "\nAverage Waiting Time: " << fixed << setprecision(2) << avgWT << endl;

    cout << "Average Turnaround Time: " << fixed << setprecision(2) << avgTAT << endl;

    cout << "\nGantt Chart:\n\n";

    cout << "+";

    for (int i = 0; i < n; i++) {

        cout << "-----+";

    }

    cout << "\n|";

    for (int i = 0; i < n; i++) {

        cout << " P" << p[i] << "  |";

    }

    cout << "\n+";

    for (int i = 0; i < n; i++) {

        cout << "-----+";

    }

    cout << "\n0";

    int totalTime = 0;

    for (int i = 0; i < n; i++) {

        totalTime += bt[i];

        cout << setw(6) << totalTime;

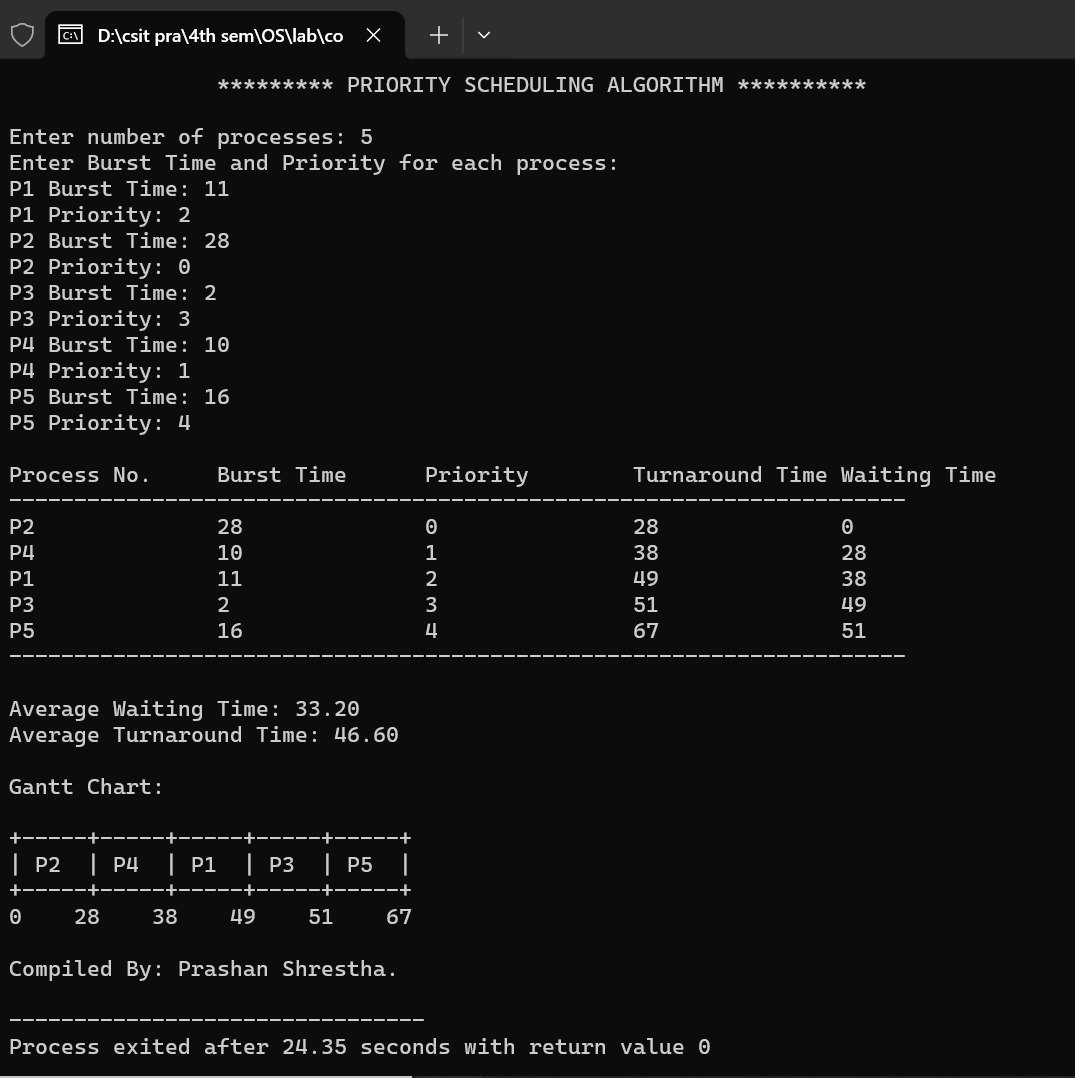
    }

    cout << endl;

    cout << "\nCompiled By: Prashan Shrestha." << endl;

    return 0; }

**Output:**



**Lab no: 12**

**Date: 2024/11/18**

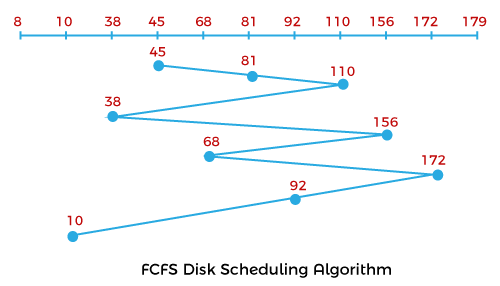


**Title: Write a program to calculate the seek time for user input request and total no of tracks in disk scheduling algorithm using FCFS algorithm.**

**First-Come, First-Served (FCFS) Disk Scheduling:**

The **First-Come, First-Served (FCFS)** algorithm is the simplest disk scheduling technique. In FCFS, requests for disk access are handled in the exact order they arrive in the queue. It does not optimize seek time, as it does not consider the current position of the disk head or the distance to the requested tracks.

**Example:** Let's take a disk with 180 tracks (0-179) and the disk queue having input/output requests in the following order: 81, 110, 38, 156, 68, 172, 92, 10. The initial head position of the Read/Write head is 45.



Seek Time:

= (81-45) + (110-81) + (110-38) + (156-110) + (156-68) + (172-156) + (172-92) + (92-10)

= 36 + 29 + 72 + 46 + 88 + 16 + 80 + 82

= 449

**Compiler: Dev C++**

**Language: C++**

**Source code:**

#include <iostream>

#include <cstdlib>

using namespace std;

int main() {

    cout << "\t\t\*\*\*\*\*\* FCFS DISK SCHEDULING ALGORITHM \*\*\*\*\*\*\n\n";

    int n, i, head, totalSeekTime = 0;

    int \*requests;

    cout << "Enter the number of disk requests: ";

    cin >> n;

    requests = new int[n];

    cout << "\nEnter the disk requests:\n";

    for (i = 0; i < n; i++) {

        cin >> requests[i];

    }

    cout << "\nEnter the initial position of the disk head: ";

    cin >> head;

    cout << "\nSeek Sequence: " << head;

    for (i = 0; i < n; i++) {

        totalSeekTime += abs(requests[i] - head);

        head = requests[i];

        cout << " -> " << head;

    }

    cout << "\nTotal seek time: " << totalSeekTime << endl;

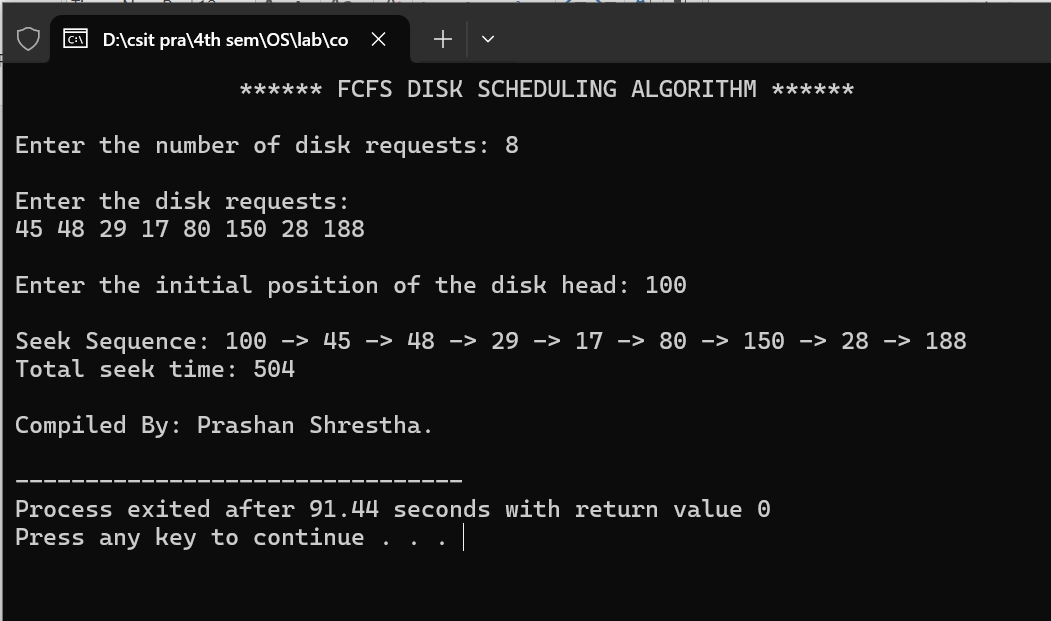
    delete[] requests;

    cout << "\nCompiled By: Prashan Shrestha." << endl;

    return 0;

}

**Output:**



**Lab no: 13**

**Date: 2024/11/18**

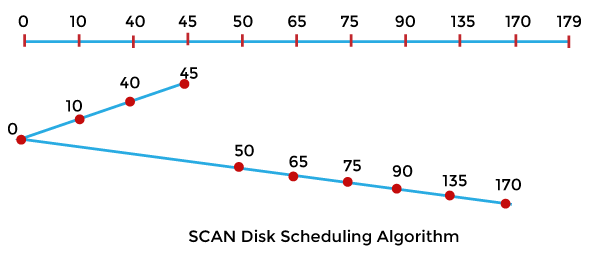


**Title: Write a program to calculate the seek time for user input request and total no of tracks in disk scheduling algorithm using SCAN algorithm.**

**SCAN Disk Scheduling:**

The **SCAN** disk scheduling algorithm, also known as the **Elevator Algorithm**, is designed to optimize disk arm movement and reduce seek time. The algorithm works by moving the disk arm in one direction (either inward or outward) servicing all requests in its path until it reaches the end of the disk. Once it reaches the end, the direction is reversed, and the disk arm services requests on the way back. This process repeats, much like an elevator moving up and down, hence the name.

**Example:** Let's take a disk with 180 tracks (0-179) and the disk queue having input/output requests in the following order: 75, 90, 40, 135, 50, 170, 65, 10. The initial head position of the Read/Write head is 45 and will move on the left-hand side.



Seek Time:

= (45-40) + (40-10) + (10-0) + (50-0) + (65-50) + (75-65) + (90-75) + (135-90) + (170-135)

= 5 + 30 +10 +50 +15 + 10 +15 + 45 + 35

= 215

**Compiler: Dev C++**

**Language: C++**

**Source code:**

#include <iostream>

#include <cstdlib>

using namespace std;

void sortRequests(int requests[], int n) {

    int i, j, temp;

    for (i = 0; i < n - 1; i++) {

        for (j = i + 1; j < n; j++) {

            if (requests[i] > requests[j]) {

                temp = requests[i];

                requests[i] = requests[j];

                requests[j] = temp;

            }

        }

    }

}

int main() {

    cout << "\t\t\*\*\*\*\*\* SCAN DISK SCHEDULING ALGORITHM \*\*\*\*\*\*\n\n";

    int n, i, head, totalSeekTime = 0, direction, end;

    int \*requests, \*seekSequence;

    cout << "Enter the number of disk requests: ";

    cin >> n;

    requests = new int[n];

    seekSequence = new int[n + 2];

    cout << "\nEnter the disk requests:\n";

    for (i = 0; i < n; i++) {

        cin >> requests[i];

    }

    cout << "\nEnter the initial position of the disk head: ";

    cin >> head;

    cout << "\nEnter the disk end point: ";

    cin >> end;

    cout << "\nEnter the direction (0 for left, 1 for right): ";

    cin >> direction;

    sortRequests(requests, n);

    int size = 0;

    if (direction == 1) {

        for (i = 0; i < n; i++) {

            if (requests[i] >= head) {

                seekSequence[size++] = requests[i];

            }

        }

        seekSequence[size++] = end;

        for (i = n - 1; i >= 0; i--) {

            if (requests[i] < head) {

                seekSequence[size++] = requests[i];

            }

        }

    } else {  // Left direction

        for (i = n - 1; i >= 0; i--) {

            if (requests[i] <= head) {

                seekSequence[size++] = requests[i];

            }

        }

        seekSequence[size++] = 0;

        for (i = 0; i < n; i++) {

            if (requests[i] > head) {

                seekSequence[size++] = requests[i];

            }

        }

    }

    cout << "\nSeek Sequence: " << head;

    for (i = 0; i < size; i++) {

        totalSeekTime += abs(seekSequence[i] - head);

        head = seekSequence[i];

        cout << " -> " << head;

    }

    cout << "\nTotal seek time: " << totalSeekTime << endl;

    delete[] requests;

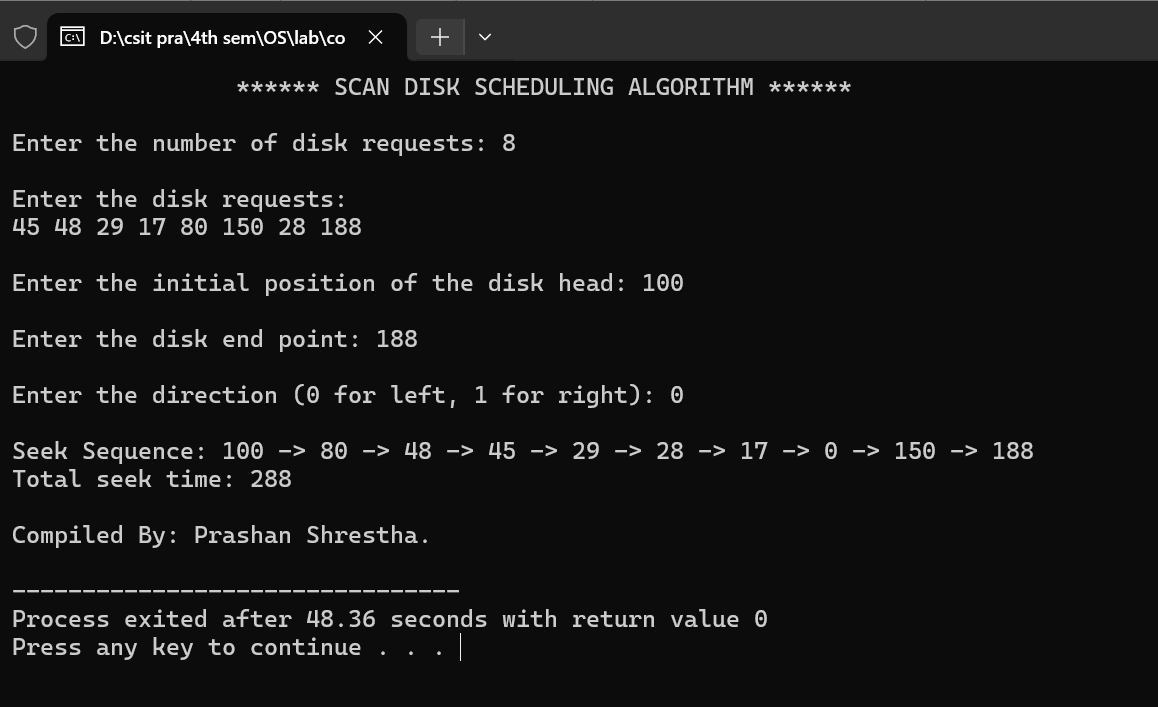
    delete[] seekSequence;

    cout << "\nCompiled By: Prashan Shrestha." << endl;

    return 0;

}

**Output:**



**Lab no: 14**

**Date: 2024/11/18**

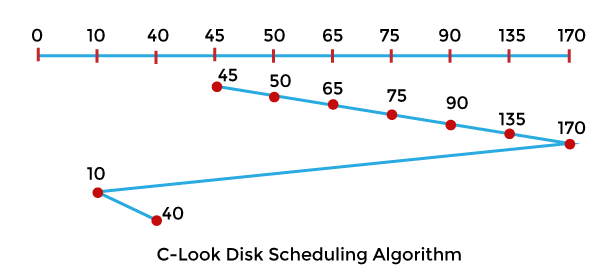


**Title: Write a program to calculate the seek time for user input request and total no of tracks in disk scheduling algorithm using C-LOOK algorithm.**

**C-LOOK Disk Scheduling:**

**C-LOOK (Circular LOOK)** is a disk scheduling algorithm that is a variation of the **LOOK** algorithm. The main difference between C-LOOK and LOOK is that in C-LOOK, when the disk arm reaches the end of the disk (either highest or lowest track), it doesn't reverse direction immediately. Instead, it jumps directly to the opposite end of the disk and starts servicing requests from there. This approach effectively minimizes the time spent on unnecessary movements, providing more efficient scheduling.

**Example:** Let's take a disk with 180 tracks (0-179) and the disk queue having input/output requests in the following order: 75, 90, 40, 135, 50, 170, 65, 10. The initial head position of the Read/Write head is 45 and will move on the left-hand side.



Seek Time:

= (50-45) + (65-50) + (75-65) + (90-75) + (135-90) + (170-135) + (170-10) + (40-10)

= 5 + 15 + 10 + 15 + 45 + 35 + 160 + 30

= 315

**Compiler: Dev C++**

**Language: C++**

**Source code:**

#include <iostream>

#include <cstdlib>

using namespace std;

void sortRequests(int requests[], int n) {

    int i, j, temp;

    for (i = 0; i < n - 1; i++) {

        for (j = i + 1; j < n; j++) {

            if (requests[i] > requests[j]) {

                temp = requests[i];

                requests[i] = requests[j];

                requests[j] = temp;

            }

        }

    }

}

int main() {

    cout << "\t\t\*\*\*\*\*\*\*\* C-LOOK DISK SCHEDULING ALGORITHM \*\*\*\*\*\*\*\*\n\n";

    int n, i, head, totalSeekTime = 0, direction, end;

    int \*requests, \*seekSequence;

    cout << "Enter the number of disk requests: ";

    cin >> n;

    requests = new int[n];

    seekSequence = new int[n + 1];

    cout << "\nEnter the disk requests:\n";

    for (i = 0; i < n; i++) {

        cin >> requests[i];

    }

    cout << "\nEnter the initial position of the disk head: ";

    cin >> head;

    cout << "\nEnter the disk end point: ";

    cin >> end;

    cout << "\nEnter the direction (0 for left, 1 for right): ";

    cin >> direction;

    sortRequests(requests, n);

    int size = 0;

    if (direction == 1) {

        for (i = 0; i < n; i++) {

            if (requests[i] >= head) {

                seekSequence[size++] = requests[i];

            }

        }

        for (i = 0; i < n; i++) {

            if (requests[i] < head) {

                seekSequence[size++] = requests[i];

            }

        }

    } else {

        for (i = n - 1; i >= 0; i--) {

            if (requests[i] <= head) {

                seekSequence[size++] = requests[i];

            }

        }

        for (i = n - 1; i >= 0; i--) {

            if (requests[i] > head) {

                seekSequence[size++] = requests[i];

            }

        }

    }

    cout << "\nSeek Sequence: " << head;

    for (i = 0; i < size; i++) {

        totalSeekTime += abs(seekSequence[i] - head);

        head = seekSequence[i];

        cout << " -> " << head;

    }

    cout << "\nTotal seek time: " << totalSeekTime << endl;

    delete[] requests;

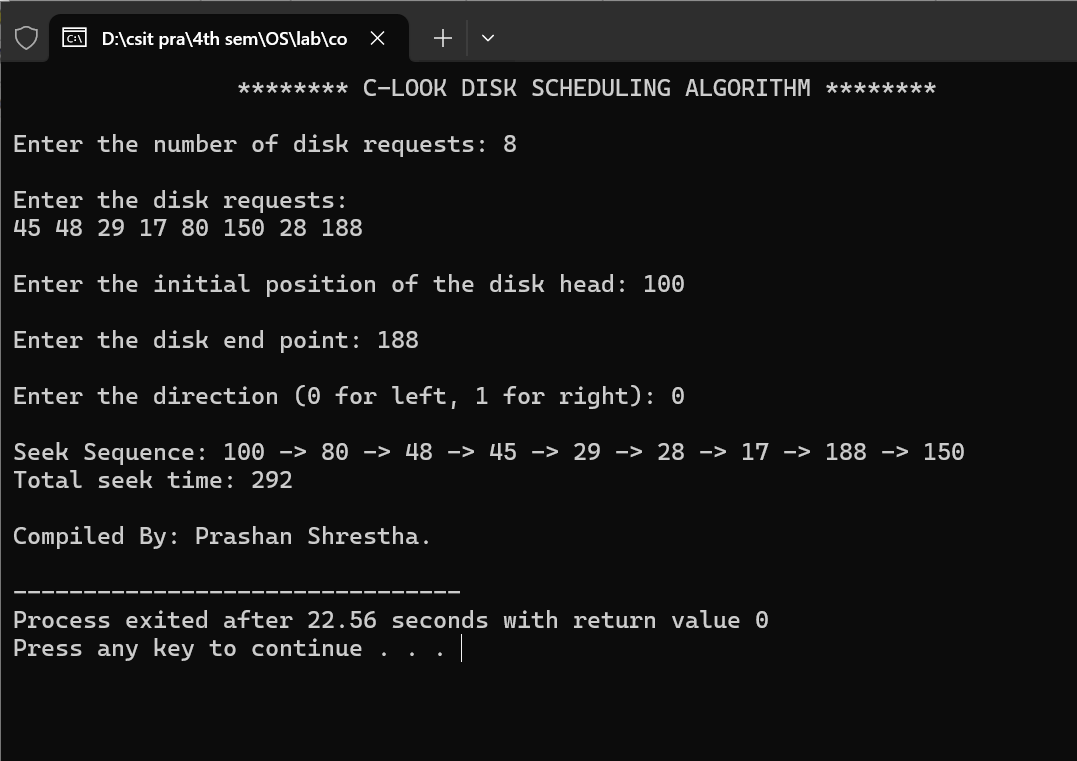
    delete[] seekSequence;

    cout << "\nCompiled By: Prashan Shrestha." << endl;

    return 0;

}

**Output:**



**Lab no: 15**

**Date: 2024/11/18**



**Title: Write a program that inputs segment no, limit and base and display the segment table and calculate physical address for user input logical address and segment no.**

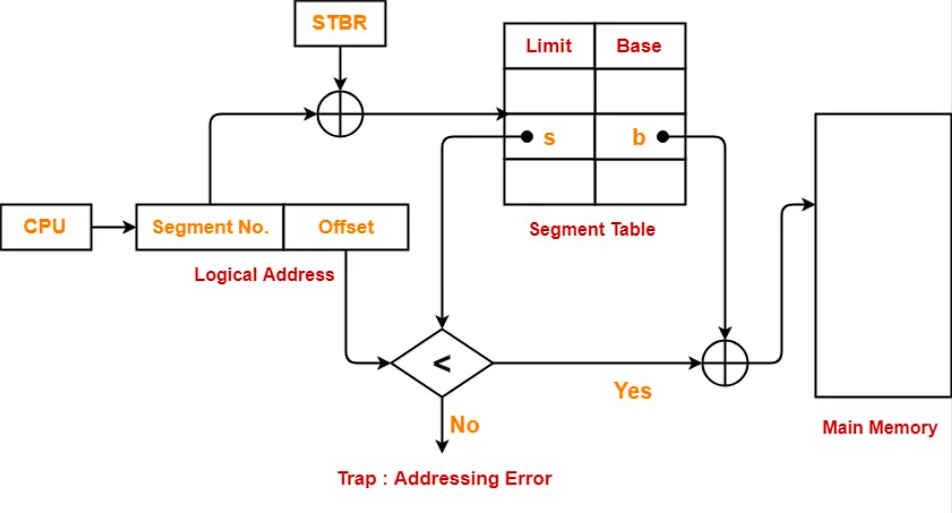
**Segmented memory** **system**:

In this memory is divided into segments of variable size. Each segment is a logical unit such as code, data, stack, or heap.A **segment table** is used in a segmented memory system to map logical addresses to physical memory addresses. It stores the **base address** and **limit** for each segment. Each entry in the table provides the starting point (base) and size (limit) of the segment in physical memory.

**Base and Limit:** The **base** is the starting physical memory address of a segment, while the **limit** is the length of the segment. The base and limit values allow the system to ensure that memory accesses are within the allocated bounds for a segment, preventing out-of-bound access and protecting memory from being overwritten.

**Logical Address:** A **logical address** is generated by the CPU and consists of a **segment number** and an **offset** within that segment. The logical address is used by the program to refer to a location in memory, but it needs to be translated into a physical address before accessing the memory.

**Physical Address:** A **physical address** is the actual location in the physical memory (RAM) where data is stored. It is calculated by adding the **base address** of the segment (from the segment table) to the **offset** of the logical address, resulting in the physical location where the data resides in memory.



**Compiler: Dev C++**

**Language: C++**

**Source code:**

#include <iostream>

using namespace std;

struct Segment {

    int segNo;

    int base;

    int limit;

};

void displayS(Segment segments[], int n) {

    cout << "\nSegment Table:\n\n";

    cout << "Segment No\tBase\tLimit\n";

    cout << "-------------------------------------" << endl;

    for (int i = 0; i < n; i++) {

        cout << segments[i].segNo << "\t\t" << segments[i].base << "\t" << segments[i].limit << endl;

    }

}

int calculatePA(Segment segments[], int n, int segNo, int LA) {

    for (int i = 0; i < n; i++) {

        if (segments[i].segNo == segNo) {

            if (LA < segments[i].limit) {

                return segments[i].base + LA;

            } else {

                cout << "Error: Logical address exceeds segment limit." << endl;

                return -1;

            }

        }

    }

    cout << "Error: Segment not found." << endl;

    return -1;

}

int main() {

    cout << "\t\t\*\*\*\*\*\*\*\*\* SEGMENTATION TABLE \*\*\*\*\*\*\*\*\*\n\n";

    int n, segNo, LA;

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

    cin >> n;

    Segment segments[n];

    for (int i = 0; i < n; i++) {

        cout << "\nEnter info of segment " << i << endl;

        cout << "Segment No: ";

        cin >> segments[i].segNo;

        cout << "Base: ";

        cin >> segments[i].base;

        cout << "Limit: ";

        cin >> segments[i].limit;

    }

    displayS(segments, n);

    cout << "\nEnter segment number and logical address:\n";

    cout << "Segment No: ";

    cin >> segNo;

    cout << "Logical Address: ";

    cin >> LA;

    int PA = calculatePA(segments, n, segNo, LA);

    if (PA != -1) {

        cout << "\nPhysical Address: " << PA << endl;

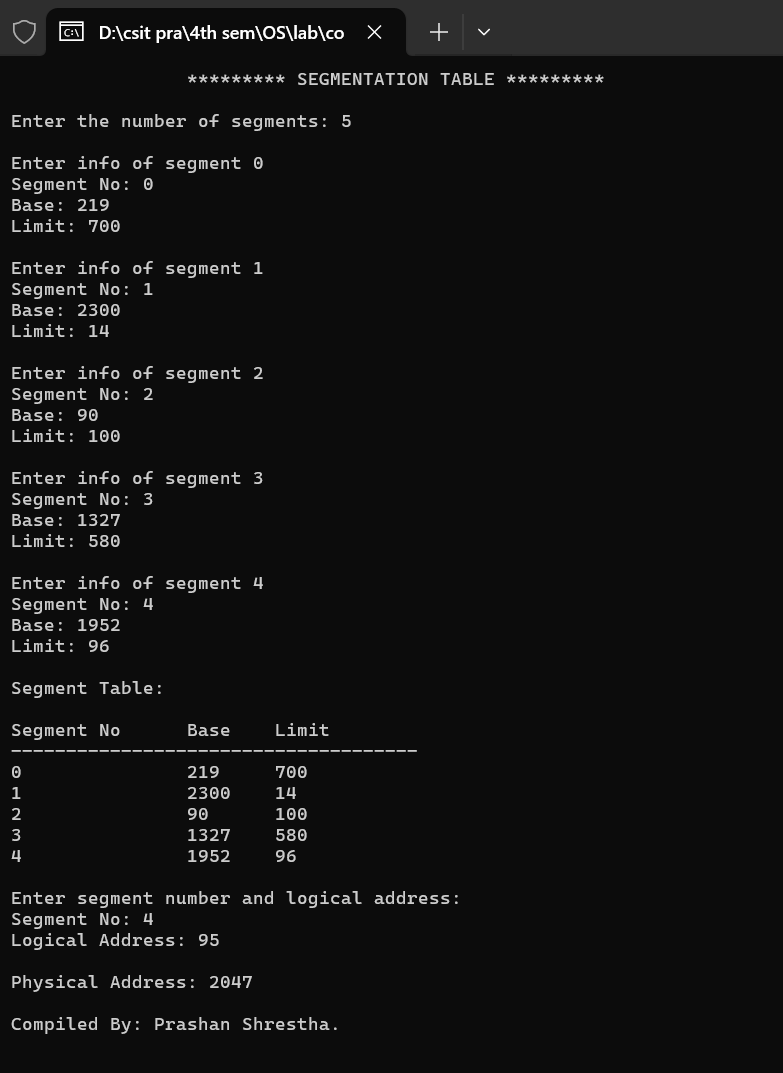
    }

    cout << "\nCompiled By: Prashan Shrestha." << endl;

    return 0;

}

**Output:**



**Index Page**

**Name: Prashan Shrestha Sub Code: CSC 264**

**Faculty:** BSc.CSIT

**Sem:** 4th Semester, 2nd year

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **LN** | **TITLE** | **DOS** | **GRADE** | **REMARKS** |
| 12 | Write a program to calculate the seek time for user input request and total no of tracks in disk scheduling algorithm using FCFS algorithm. |  |  |  |
| 13 | Write a program to calculate the seek time for user input request and total no of tracks in disk scheduling algorithm using SCAN algorithm. |  |  |  |
| 14 | Write a program to calculate the seek time for user input request and total no of tracks in disk scheduling algorithm using C-LOOK algorithm. |  |  |  |
| 15 | Write a program that inputs segment no, limit and base and display the segment table and calculates the physical address for user input logical address and segment number. |  |  |  |

*\*Faculty must grade as A, B, C or D*