



Lab-task-Report
On
CSE-3632
Operating Systems Lab

submitted to —

Mohammad Zainal Abedin

Assistant Professor ,Dept of CSE

submitted by—

Name : MD. Faisal Hoque Rifat
Metric No : C221076
Section : 6BM
Semester : 6th
Date of Submission: 17-01-2025

1. FCFS Cpu Scheduling Algorithm:

Algorithm:

The mechanics of FCFS are straightforward:

- **Arrival:** Processes enter the system and are placed in a queue in the order they arrive.
- **Execution:** The CPU takes the first process from the front of the queue, executes it until it is complete, and then removes it from the queue.
- **Repeat:** The CPU takes the next process in the queue and repeats the execution process.

This continues until there are no more processes left in the queue.

Code:

```
#include<bits/stdc++.h>

using namespace std;

int main()
{
    int n;

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

    cin >> n;

    vector<int> processIDs(n), burstTime(n), waitingTime(n), turnaroundTime(n);

    for (int i = 0; i < n; i++)
    {
        processIDs[i] = i + 1;

        cout << "Enter Burst Time for Process P" << processIDs[i] << ": ";

        cin >> burstTime[i];
    }

    waitingTime[0] = 0; // First process has no waiting time

    for (int i = 1; i < burstTime.size(); i++)
```

```

{
    waitingTime[i] = waitingTime[i - 1] + burstTime[i - 1];
}

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

{
    turnaroundTime[i] = waitingTime[i] + burstTime[i];
}

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

for (size_t i = 0; i < processIDs.size(); i++)

{
    cout << "P" << processIDs[i] << "\t\t" << burstTime[i] << "\t\t" << waitingTime[i] << "\t\t"
    << turnaroundTime[i] << endl;
}

double totalWaitingTime = 0, totalTurnaroundTime = 0;

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

{
    totalWaitingTime += waitingTime[i];
    totalTurnaroundTime += turnaroundTime[i];
}

cout << "Average Waiting Time: " << totalWaitingTime / n << endl;

cout << "Average Turnaround Time: " << totalTurnaroundTime / n << endl;
}

```

Link: <https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/FCFS.cpp>

Output:

```
C:\WINDOWS\system32\cmd.  X  +  v
Enter the number of processes: 3
Enter Burst Time for Process P1: 10
Enter Burst Time for Process P2: 5
Enter Burst Time for Process P3: 8
Process Burst Time      Waiting Time      Turnaround Time
P1          10           0                10
P2           5          10                15
P3           8          15                23
Average Waiting Time: 8.33333
Average Turnaround Time: 16
Press any key to continue . . . |
```

Discussion:

First-Come, First-Served (FCFS) Scheduling operates as a non-preemptive strategy, meaning that once a process begins executing on the CPU, it runs to completion without being interrupted. This approach ensures that processes are handled in the exact order of their arrival in the ready queue. FCFS does not differentiate between processes based on priority levels or other criteria, treating all processes equally. While simple and straightforward, the algorithm can lead to issues such as the **convoy effect**, where shorter processes may be delayed by longer ones ahead in the queue.

2. Shortest Job First (SJF) Cpu Scheduling Algorithm:

Algorithm:

- Sort all the processes according to the arrival time.
- Then select that process that has minimum arrival time and minimum Burst time.
- After completion of the process make a pool of processes that arrives afterward till the completion of the previous process and select that process among the pool which is having minimum Burst time.

How to compute below times in SJF using a program?

- Completion Time: Time at which process completes its execution.
- Turn Around Time: Time Difference between completion time and arrival time.
Turn Around Time = Completion Time – Arrival Time
- Waiting Time(W.T): Time Difference between turn around time and burst time.
Waiting Time = Turn Around Time – Burst Time

Code:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    cout << "Enter the number of processes: ";
    cin>>n;
    vector<int> processID(n), burstTime(n), waitingTime(n), turnaroundTime(n);
    for (int i = 0; i < n; i++)
    {
        processID[i] = i + 1;
        cout << "Enter Burst Time for Process P" << processID[i] << ": ";
        cin >> burstTime[i];
    }
    for (int i = 0; i < n - 1; i++)
    {
        int minIndex = i;
        for (int j = i + 1; j < n; j++)
        {
            if (burstTime[j] < burstTime[minIndex])
            {
                minIndex = j;
            }
        }
        swap(burstTime[i], burstTime[minIndex]);
        swap(processID[i], processID[minIndex]);
    }
}
```

```

waitingTime[0] = 0;
for (int i = 1; i < n; i++)
{
    waitingTime[i] = waitingTime[i - 1] + burstTime[i - 1];
}
for (int i = 0; i < n; i++)
{
    turnaroundTime[i] = waitingTime[i] + burstTime[i];
}
cout << "Process\tBurst Time\tWaiting Time\tTurnaround Time\n";
for (int i = 0; i < n; i++)
{
    cout << "P" << processID[i] << "\t\t" << burstTime[i] << "\t\t" <<
waitingTime[i] << "\t\t" << turnaroundTime[i] << endl;
}
double totalWaitingTime = 0, totalTurnaroundTime = 0;
for (int i = 0; i < n; i++)
{
    totalWaitingTime += waitingTime[i];
    totalTurnaroundTime += turnaroundTime[i];
}
cout << "Average Waiting Time: " << totalWaitingTime / n << endl;
cout << "Average Turnaround Time: " << totalTurnaroundTime / n << endl;
}

```

Link: <https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/SJF.cpp>

Output:

```
C:\WINDOWS\system32\cmd.  ×  +  v
Enter the number of processes: 5
Enter Burst Time for Process P1: 5
Enter Burst Time for Process P2: 2
Enter Burst Time for Process P3: 5
Enter Burst Time for Process P4: 7
Enter Burst Time for Process P5: 3
Process Burst Time      Waiting Time      Turnaround Time
P2          2           0             2
P5          3           2             5
P3          5           5            10
P1          5          10            15
P4          7          15            22
Average Waiting Time: 6.4
Average Turnaround Time: 10.8
Press any key to continue . . . |
```

Discussion:

- SJF is better than the [First come first serve](#)(FCFS) algorithm as it reduces the average waiting time.
- SJF is generally used for long term scheduling
- It is suitable for the jobs running in batches, where run times are already known.
- SJF is probably optimal in terms of average turnaround time.

3. Shortest Remaining Time First:

Algorithm:

In the **Shortest Remaining Time First (SRTF) scheduling algorithm**, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a **smaller amount of time**.

Code:

```
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int n;

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

    cin >> n;

    vector<int> arrivalTime(n), burstTime(n), remainingTime(n);
    vector<int> completionTime(n), waitingTime(n), turnaroundTime(n);

    for (int i = 0; i < n; i++)
    {
        cout << "Enter Arrival Time for Process P" << i + 1 << ": ";

        cin >> arrivalTime[i];

        cout << "Enter Burst Time for Process P" << i + 1 << ": ";

        cin >> burstTime[i];

        remainingTime[i] = burstTime[i];
    }

    int complete = 0, currentTime = 0, minIndex = -1;

    int totalWaitingTime = 0, totalTurnaroundTime = 0;

    vector<pair<int, int>> ganttChart; // {StartTime, ProcessID}

    vector<int> ganttTimes;

    while (complete < n)
```



```

{
    minIndex = -1;
    int minRemaining = INT_MAX;
    for (int i = 0; i < n; i++)
    {
        if (arrivalTime[i] <= currentTime && remainingTime[i] > 0 &&
remainingTime[i] < minRemaining)
        {
            minRemaining = remainingTime[i];
            minIndex = i;
        }
    }
    if (minIndex == -1)
    {
        currentTime++;
        continue;
    }
    if (ganttChart.empty() || ganttChart.back().second != minIndex)
    {
        ganttChart.push_back({currentTime, minIndex});
    }
    remainingTime[minIndex]--;
    currentTime++;
}

```

```

    if (remainingTime[minIndex] == 0)
    {
        complete++;

        completionTime[minIndex] = currentTime;

        turnaroundTime[minIndex] = completionTime[minIndex] -
arrivalTime[minIndex];

        waitingTime[minIndex] = turnaroundTime[minIndex] -
burstTime[minIndex];

        totalWaitingTime += waitingTime[minIndex];

        totalTurnaroundTime += turnaroundTime[minIndex];
    }
}

for (auto entry : ganttChart)
{
    ganttTimes.push_back(entry.first);
}

ganttTimes.push_back(currentTime);

cout << "\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround
Time\n";

for (int i = 0; i < n; i++)
{
    cout << "P" << i + 1 << "\t\t" << arrivalTime[i] << "\t\t" << burstTime[i] <<
"\t\t" << waitingTime[i] << "\t\t" << turnaroundTime[i] << endl;
}

```

```

cout << "Average Waiting Time: " << (double)totalWaitingTime / n << endl;

cout << "Average Turnaround Time: " << (double)totalTurnaroundTime / n <<
endl;

cout << "\nGantt Chart: \n";

// Print the top row (process blocks)

cout << "|";

for (int i = 0; i < ganttChart.size(); i++)
{
    int processId = ganttChart[i].second + 1;

    int width = ganttTimes[i + 1] - ganttTimes[i]; // Width of the block

    cout << " P" << processId << " ";

    for (int j = 1; j < width; j++)
        cout << " "; // Add extra spaces for wider blocks

    cout << "|";
}

cout << endl;

// Print the bottom row (time intervals)

for (size_t i = 0; i < ganttTimes.size(); i++)
{
    cout << ganttTimes[i];

    if (i < ganttTimes.size() - 1)
    {
        int width = ganttTimes[i + 1] - ganttTimes[i]; // Width of the block

```

```

        for (int j = 0; j < width + 2; j++)

            cout << " "; // Match width of the process blocks

        }

    }

    cout << endl;

    return 0;

}

```

Link: <https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/SRTF.cpp>

Output:

```

C:\WINDOWS\system32\cmd.  X  +  v
Enter the number of processes: 3
Enter Arrival Time for Process P1: 0
Enter Burst Time for Process P1: 5
Enter Arrival Time for Process P2: 1
Enter Burst Time for Process P2: 3
Enter Arrival Time for Process P3: 2
Enter Burst Time for Process P3: 8

Process Arrival Time    Burst Time    Waiting Time    Turnaround Time
P1                0            5            3                8
P2                1            3            0                3
P3                2            8            6               14
Average Waiting Time: 3
Average Turnaround Time: 8.33333

Gantt Chart:
| P1 | P2 | P1 | P3 |
0  1  4  8  16
Press any key to continue . . . |

```

Discussion:

- SRTF has a higher complexity than other scheduling algorithms like FCFS (First Come First Serve) and RR (Round Robin), because it requires frequent context switches and preemptions.

4. Round Robin Scheduling Algorithm

Algorithm:

- Create an array **rem_bt[]** to keep track of remaining burst time of processes. This array is initially a copy of **bt[]** (burst times array)
- Create another array **wt[]** to store waiting times of processes. Initialize this array as 0.
- Initialize time : $t = 0$
- Keep traversing all the processes while they are not done. Do following for **i'th** process if it is not done yet.
 - If $\text{rem_bt}[i] > \text{quantum}$
 - $t = t + \text{quantum}$
 - $\text{rem_bt}[i] -= \text{quantum};$
 - Else // Last cycle for this process
 - $t = t + \text{rem_bt}[i];$
 - $\text{wt}[i] = t - \text{bt}[i]$
 - $\text{rem_bt}[i] = 0;$ // This process is over

Once we have waiting times, we can compute turn around time $\text{tat}[i]$ of a process as sum of waiting and burst times, i.e., $\text{wt}[i] + \text{bt}[i]$.

Code:

```
#include<bits/stdc++.h>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int n, timeQuantum;
```

```

// Input number of processes and time quantum

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

cin >> n;

cout << "Enter the time quantum: ";

cin >> timeQuantum;


vector<int> burstTime(n), remainingTime(n), waitingTime(n, 0),
turnaroundTime(n, 0);

vector<pair<int, int>> ganttChart; // To store process execution and its time


// Input burst times
for (int i = 0; i < n; i++) {

    cout << "Enter Burst Time for Process P" << i + 1 << ": ";

    cin >> burstTime[i];

    remainingTime[i] = burstTime[i];

}


queue<int> readyQueue; // Queue for processes
for (int i = 0; i < n; i++) {

    readyQueue.push(i); // Add all processes to the queue initially

}

```

```

int currentTime = 0;

while (!readyQueue.empty()) {

    int currentProcess = readyQueue.front();

    readyQueue.pop();

    // Execute the process

    ganttChart.push_back({currentProcess + 1, currentTime}); // Record process
and start time

    if (remainingTime[currentProcess] <= timeQuantum) {

        // Process completes execution

        currentTime += remainingTime[currentProcess];

        turnaroundTime[currentProcess] = currentTime;

        waitingTime[currentProcess] = turnaroundTime[currentProcess] -
burstTime[currentProcess];

        remainingTime[currentProcess] = 0;

    } else {

        // Process partially executes and goes back to the queue

        currentTime += timeQuantum;

        remainingTime[currentProcess] -= timeQuantum;

        readyQueue.push(currentProcess); // Add process back to the queue

    }

}

```

```

// Add the final time to the Gantt Chart

ganttChart.push_back({-1, currentTime}); // -1 indicates end of execution


// Display Gantt Chart with Time

cout << "\nGantt Chart:\n";

for (size_t i = 0; i < ganttChart.size() - 1; i++) {

    cout << "| P" << ganttChart[i].first << " ";

}

cout << "| \n";

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

{

    cout << ganttChart[i].second << " ";

}

cout << "\n";


// Display process details

cout << "\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n";

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

{

    cout << "P" << i + 1 << "\t\t" << burstTime[i] << "\t\t" << waitingTime[i] <<
"\t\t" << turnaroundTime[i] << "\n";

}

```



```
// Calculate averages

double totalWaitingTime = 0, totalTurnaroundTime = 0;

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

    totalWaitingTime += waitingTime[i];

    totalTurnaroundTime += turnaroundTime[i];

}

cout << "Average Waiting Time: " << totalWaitingTime / n << "\n";

cout << "Average Turnaround Time: " << totalTurnaroundTime / n << "\n";


return 0;

}
```

Link: <https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/RR.cpp>

Output:

```
C:\WINDOWS\system32\cmd.  X  +  v

Enter the number of processes: 4
Enter the time quantum: 3
Enter Burst Time for Process P1: 8
Enter Burst Time for Process P2: 4
Enter Burst Time for Process P3: 3
Enter Burst Time for Process P4: 9

Gantt Chart:
| P1 | P2 | P3 | P4 | P1 | P2 | P4 | P1 | P4 |
0 3 6 9 12 15 16 19 21 24

Process Burst Time      Waiting Time      Turnaround Time
P1          8           13           21
P2          4           12           16
P3          3            6            9
P4          9           15           24
Average Waiting Time: 11.5
Average Turnaround Time: 17.5

Press any key to continue . . . |
```

Discussion:

Round Robin assigns each process with a fixed time quantum during which it can execute on the CPU. Once the time quantum expires, the next process in line is allocated the CPU for its turn. No process can run for more than one quantum while others are waiting in the ready queue. If a process needs more CPU time to complete after exhausting one quantum, it goes to the end of ready queue to await the next allocation. This cyclical distribution ensures that each process gets an equal opportunity to utilize the CPU.

5. Banker's Algorithm:

Algorithm:

Banker's algorithm is a **resource allocation** and **deadlock avoidance algorithm** developed by **Edsger Dijkstra** that tests for safety by simulating the allocation of predetermined maximum possible amounts of all **resources**, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

Let n be the number of processes in the system and m be the number of resource types. Then we need the following data structures:

- **Available:** A vector of length m indicates the number of available resources of each type. If $\text{Available}[j] = k$, there are k instances of resource type R_j available.
- **Max:** An $n \times m$ matrix defines the maximum demand of each process. If $\text{Max}[i,j] = k$, then P_i may request at most k instances of resource type R_j .
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $\text{Allocation}[i,j] = k$, then process P_i is currently allocated k instances of resource type R_j .
- **Need:** An $n \times m$ matrix indicates the remaining resource need of each process. If $\text{Need}[i,j] = k$, then P_i may need k more instances of resource type R_j to complete the task.

Note: $\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$. $n=m-a$.

Code:

```
#include<bits/stdc++.h>

using namespace std;

int main()
```

```

{
    int n, m;

    n = 5;

    m = 3;

    int alloc[n][m] = {{0, 1, 0},{2, 0, 0},{3, 0, 2},{2, 1, 1},{0, 0, 2}};

    int max[n][m] = {{7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3}};

    int need[n][m];

    for(int i = 0; i<n; i++)
    {
        for(int j = 0; j<m; j++)
        {
            need[i][j] = max[i][j]-alloc[i][j];
        }
    }

    int avail[m]={3, 3, 2};

    int ans[n],f[n], index = 0;

    for(int i = 0; i<n; i++)
    {
        f[i] = 0;
    }

    for(int k = 0; k<n; k++)
    {
        for(int i = 0; i<n; i++)
        {
            if(f[i] == 0)

```

```

{
    int flag = 0;
    for(int j = 0; j<m; j++)
    {
        if(need[i][j]>avail[j])
        {
            flag = 1;
            break;
        }
    }
    if(flag == 0)
    {
        ans[index] = i;
        index++;
        for(int j = 0; j<m; j++)
        {
            avail[j]+=alloc[i][j];
        }
        f[i] = 1;
    }
}

}

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

```

```

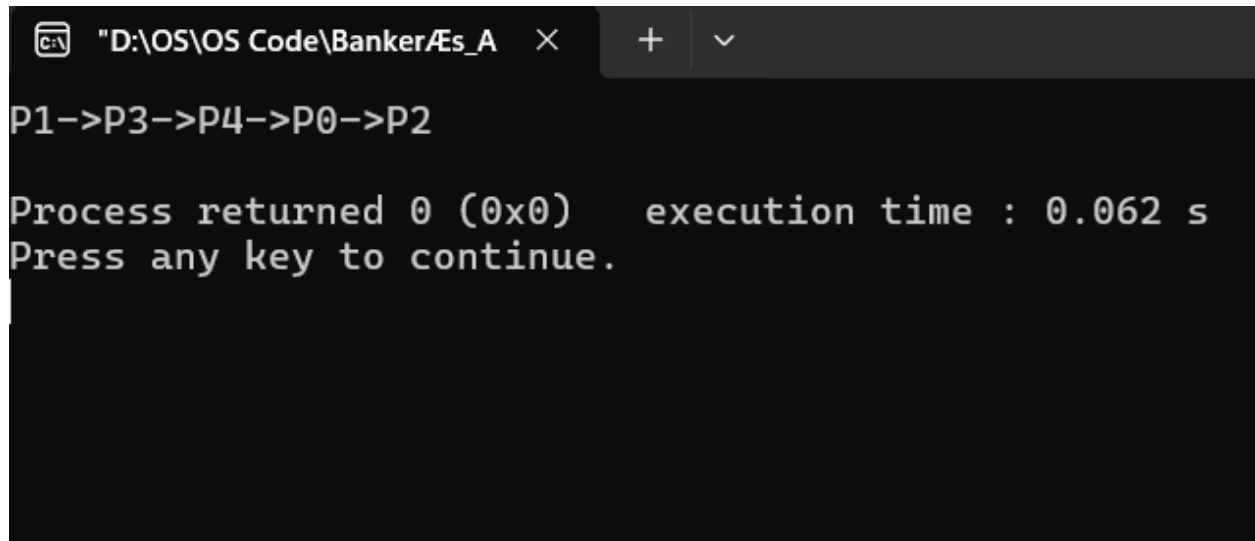
{
    if(f[i] == 0)
    {
        flag = 0;
        break;
    }
}
if(flag == 0)
{
    cout<<"NOT in safe state"<<endl;
}
else
{
    for(int i = 0; i<n-1; i++)
    {
        cout<<"P"<<ans[i]<<"->";
    }
    cout<<"P"<<ans[n-1]<<endl;
}
return 0;
}

```

Link:

https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/Banker%E2%80%99s_Algorithm.cpp

Output:

A screenshot of a Windows command prompt window. The title bar shows the file path "D:\OS\OS Code\Banker\Es_A" and standard window controls. The command prompt displays the output of a program: "P1->P3->P4->P0->P2", "Process returned 0 (0x0) execution time : 0.062 s", and "Press any key to continue.".

```
"D:\OS\OS Code\Banker\Es_A"
P1->P3->P4->P0->P2
Process returned 0 (0x0) execution time : 0.062 s
Press any key to continue.
```

Discussion:

The Banker's Algorithm is a method used by computers to manage resources like memory or processing power. It helps avoid a problem called deadlock, where programs get stuck and can't finish their work. The algorithm works like a banker giving out loans—it only "lends" resources if it's sure there will still be enough for every program to finish. By doing this, it keeps everything running smoothly, even when multiple programs are using resources at the same time.

6. Deadlock Detection Algorithm:

Algorithm :

A deadlock occurs when a group is stuck in a process because everyone is hanging onto resources while waiting for other ways to get them.

1. Steps of Algorithm
2. Step 1
3. 1. Let Work(vector) length = m
4. 2. Finish(vector) length = n
5. 3. Initialize Work= Available.
6. 1. **if** Allocation = 0 $\forall i \in [0, N-1]$, then Finish[i] = **true**;
7. otherwise, Finish[i]= **false**.
8. Step 2
9. 1. Find the index i with the conditions
- 10.1. Finish[i] == **false**
- 11.2. Work \geq Request i
12. If exists no i, go to step 4.
13. Step 3
- 14.1. Work += Allocation i
- 15.2. Finish[i] = **true**
16. Go to Step 2.
17. Step 4
- 18.1. For some i in [0, N), **if** Finish[i]==**false**, the deadlock occurred. Finish [i]==**false**, the process P_i is deadlocked.

Code:

```
#include <bits/stdc++.h>

using namespace std;

int main() {

    int allocation[10][10];
```



```

int request[10][10];

int available[10];

int resources[10];

int work[10];

int marked[10];

int num_processes, num_resources;


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

cin >> num_processes;


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

cin >> num_resources;


// Input total resources
for (int i = 0; i < num_resources; i++) {

    cout << "Enter the total amount of Resource R" << i + 1 << ": ";

    cin >> resources[i];

}

// Input request matrix
cout << "Enter the request matrix:\n";

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

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

        cin >> request[i][j];
    }
}

```

```

    }
}

// Input allocation matrix
cout << "Enter the allocation matrix:\n";
for (int i = 0; i < num_processes; i++) {
    for (int j = 0; j < num_resources; j++) {
        cin >> allocation[i][j];
    }
}

// Initialize available resources
for (int j = 0; j < num_resources; j++) {
    available[j] = resources[j];
    for (int i = 0; i < num_processes; i++) {
        available[j] -= allocation[i][j];
    }
}

// Mark processes with zero allocation
for (int i = 0; i < num_processes; i++) {
    int count = 0;
    for (int j = 0; j < num_resources; j++) {

```

```

        if (allocation[i][j] == 0) {
            count++;
        } else {
            break;
        }
    }
    if (count == num_resources) {
        marked[i] = 1;
    }
}

// Initialize work with available resources
for (int j = 0; j < num_resources; j++) {
    work[j] = available[j];
}

// Mark processes with requests <= work
for (int i = 0; i < num_processes; i++) {
    int can_be_processed = 1;
    if (marked[i] != 1) {
        for (int j = 0; j < num_resources; j++) {
            if (request[i][j] > work[j]) {
                can_be_processed = 0;
                break;
            }
        }
    }
}

```

```

    }
    if (can_be_processed) {
        marked[i] = 1;
        for (int j = 0; j < num_resources; j++) {
            work[j] += allocation[i][j];
        }
    }
}

// Check for unmarked processes (deadlock)
int deadlock = 0;
for (int i = 0; i < num_processes; i++) {
    if (marked[i] != 1) {
        deadlock = 1;
        break;
    }
}

if (deadlock) {
    cout << "Deadlock detected\n";
} else {
    cout << "No deadlock possible\n";
}

return 0;

```

}

Link:

https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/Banker%E2%80%99s_Algorithm.cpp

Output:

```
Output Clear
^ Enter the number of processes: 3
Enter the number of resources: 3
Enter the total amount of Resource R1: 4
Enter the total amount of Resource R2: 5
Enter the total amount of Resource R3: 7
Enter the request matrix:
1 2 3
4 5 6
7 8 9
Enter the allocation matrix:
5 6 7
1 2 3
7 8 9
Deadlock detected

=== Code Execution Successful ===
```

Discussion:

Deadlock detection algorithms help keep computer systems stable and reliable. There are different ways to implement these algorithms, each with its own pros and cons. The right choice depends on what the system needs. In general, these algorithms are essential for making sure operating systems work smoothly and efficiently, especially in complex setups.

7. Least Recently Used (LRU)

Algorithm :

Let **capacity** be the number of pages that memory can hold. Let **set** be the current set of pages in memory.

1- Start traversing the pages.

i) **If set holds less pages than capacity.**

- a) Insert page into the set one by one until the size of **set** reaches **capacity** or all page requests are processed.
- b) Simultaneously maintain the recent occurred index of each page in a map called **indexes**.
- c) Increment page fault

ii) **Else**

If current page is present in **set**, do nothing.

Else

- a) Find the page in the set that was least recently used. We find it using index array. We basically need to replace the page with minimum index.
- b) Replace the found page with current page.
- c) Increment page faults.
- d) Update index of current page.

2. Return page faults.

Code:

```
#include<bits/stdc++.h>
```

```
using namespace std;
```

```
int main()
```

```

{
    vector<int>ref_string({7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1});

    int frames = 3;

    list<int>memory;

    set<int>st;

    int page_fault = 0;

    for(int i = 0; i<ref_string.size(); i++)
    {
        if(st.count(ref_string[i]) == 0)
        {
            page_fault++;

            if(memory.size() == frames)
            {
                int remove = memory.front();

                memory.pop_front();

                st.erase(remove);
            }

            memory.push_back(ref_string[i]);

            st.insert(ref_string[i]);
        }
        else
        {
            memory.remove(ref_string[i]);
        }
    }
}

```

```

        memory.push_back(ref_string[i]);
    }
}

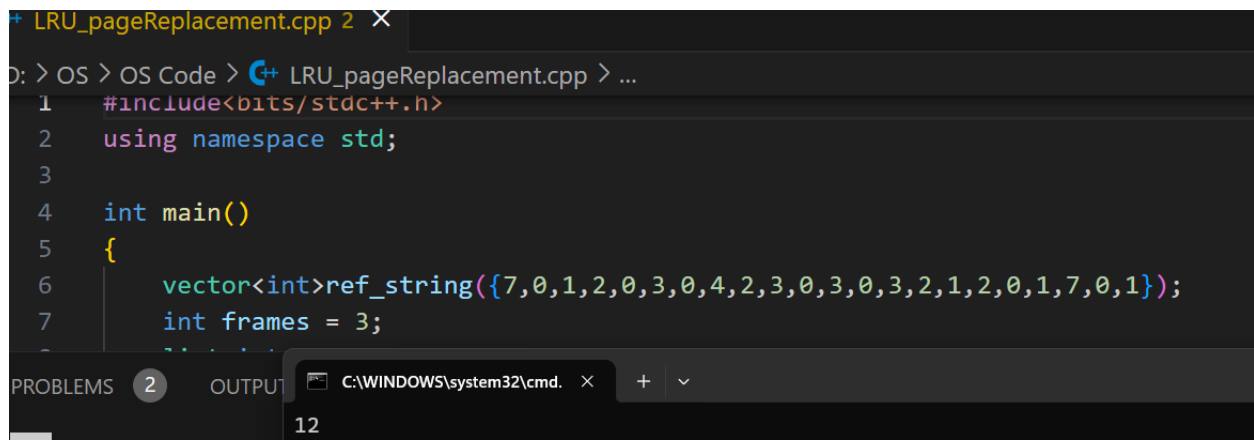
cout<<page_fault<<endl;
}

```

Link:

https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/LRU_pageReplacement.cpp

Output:



```

+ LRU_pageReplacement.cpp 2 X
D:\> OS > OS Code > C++ LRU_pageReplacement.cpp > ...
1  #include<bits/stdc++.h>
2  using namespace std;
3
4  int main()
5  {
6      vector<int>ref_string({7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1});
7      int frames = 3;
8      ...

```

PROBLEMS 2 OUTPUT C:\WINDOWS\system32\cmd. X + v

12

Discussion:

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

8. FIFO (First-In-First-Out)

Algorithm :

1- Start traversing the pages.

i) If set holds less pages than capacity.

a) Insert page into the set one by one until
the size of set reaches capacity or all
page requests are processed.

b) Simultaneously maintain the pages in the
queue to perform FIFO.

c) Increment page fault

ii) Else

If current page is present in set, do nothing.

Else

a) Remove the first page from the queue
as it was the first to be entered in
the memory

b) Replace the first page in the queue with
the current page in the string.

c) Store current page in the queue.

d) Increment page faults.

2. Return page faults.

Code:

```
#include<bits/stdc++.h>
```

```
using namespace std;
```

```
int main()
```

```

{
    vector<int>ref_string({7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1});

    int frames = 3;

    queue<int>memory;

    set<int>st;

    int page_fault = 0;

    cout<<ref_string.size()<<endl;

    for(int i = 0; i<ref_string.size(); i++)
    {
        if(st.count(ref_string[i]) == 0)
        {
            page_fault++;

            if(memory.size() == frames)
            {
                int remove = memory.front();

                memory.pop();

                st.erase(remove);
            }

            memory.push(ref_string[i]);

            st.insert(ref_string[i]);
        }
    }

    cout<<page_fault<<endl;

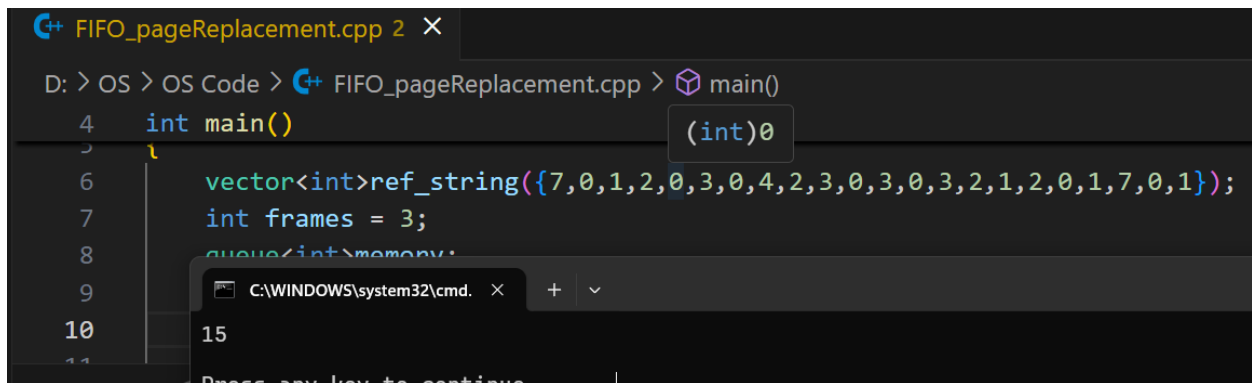
```

```
}
```

Link:

https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/FIFO_pageReplacement.cpp

Output:



```
C++ FIFO_pageReplacement.cpp 2 X
D: > OS > OS Code > C++ FIFO_pageReplacement.cpp > main()
4 int main() (int)0
5 {
6     vector<int>ref_string({7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1});
7     int frames = 3;
8     queue<int>memory;
9
10    15
11    Press any key to continue
```

Discussion:

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

9.Second-Chance Page Replacement Algorithm:

Algorithm :

Create an array **frames** to track the pages currently in memory and another Boolean array **second_chance** to track whether that page has been accessed since it's last

replacement (that is if it deserves a second chance or not) and a variable **pointer** to track the target for replacement.

1. Start traversing the array **arr**. If the page already exists, simply set its corresponding element in **second_chance** to true and return.
2. If the page doesn't exist, check whether the space pointed to by **pointer** is empty (indicating cache isn't full yet) – if so, we will put the element there and return, else we'll traverse the array **arr** one by one (cyclically using the value of **pointer**), marking all corresponding **second_chance** elements as false, till we find a one that's already false. That is the most suitable page for replacement, so we do so and return.
3. Finally, we report the page fault count.

Code:

```
#include<bits/stdc++.h>

using namespace std;

int main()
{
    vector<int>ref_string({7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1});
    int frames = 3;
    queue<int>memory;
    set<int>st;
    set<int>second_chance;
    int page_fault = 0;
    for(int i = 0; i<ref_string.size(); i++)
    {
        if(st.count(ref_string[i]) == 0)
        {
```

```
page_fault++;
if(memory.size() == frames)
{
    while (true)
    {
        int remove = memory.front();
        memory.pop();
        if(second_chance.count(remove) == 1)
        {
            second_chance.erase(remove);
            memory.push(remove);
        }
        else
        {
            st.erase(remove);
            break;
        }
    }
}
memory.push(ref_string[i]);
st.insert(ref_string[i]);
}
else
{
    second_chance.insert(ref_string[i]);
```

```

    }
}

cout<<page_fault<<endl;

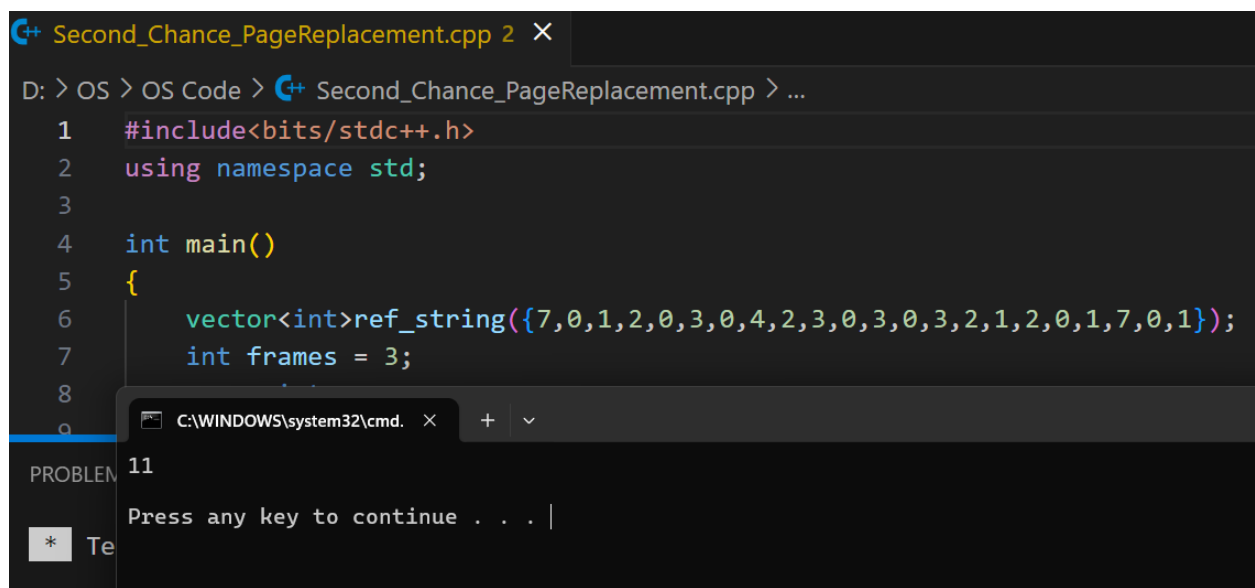
return 0;
}

```

Link:

https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/Second_Chance_PageReplacement.cpp

Output:



The screenshot shows a C++ IDE with a file named `Second_Chance_PageReplacement.cpp` open. The code is as follows:

```

1  #include<bits/stdc++.h>
2  using namespace std;
3
4  int main()
5  {
6      vector<int>ref_string({7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1});
7      int frames = 3;
8
9
10
11

```

The output window shows the text: `Press any key to continue . . . |`. The terminal window shows the command prompt path: `C:\WINDOWS\system32\cmd.`

Discussion:

The second chance algorithm is a page replacement policy that uses a FIFO algorithm and a hardware-provided reference bit. The page table is traversed in a FIFO (circular queue) manner. If a page table is found with its reference bit not set, then that page is selected as the next victim.

10.First-Best-Worst Fit Algorithm:

Introduction:

Memory allocation is a critical task in modern operating systems, and one of the most commonly used techniques is contiguous memory allocation. Contiguous memory allocation involves allocating memory to processes in contiguous blocks, where the starting address of each block is adjacent to the previous one. In this blog post, we'll explore three cases of contiguous memory allocation: First Fit, Best Fit, and Worst Fit, with humor, emojis, and solved examples.

Code:

//First Fit

```
#include<bits/stdc++.h>

using namespace std;

int main()

{

    vector<int>block_size = {100, 500, 200, 300, 600};
```

```

vector<int>process_size = {212, 417, 112, 426};

int n = block_size.size();

int m = process_size.size();

vector<int>alloc(m, -1);

for(int i = 0; i<m; i++)
{
    for(int j = 0; j<n; j++)
    {
        if(block_size[j]>=process_size[i])
        {
            alloc[i] = j+1;
            block_size[j]-=process_size[i];
            break;
        }
    }
}

cout<<"Process\tSize\tAllocation"<<endl;

for(int i = 0; i<m; i++)
{
    cout<<"P"<<i+1<<"\t";

    cout<<process_size[i]<<"\t";

    if(alloc[i] == -1) cout<<"NOT Allocated"<<endl;

    else cout<<alloc[i]<<endl;
}

return 0;

```



```
}
```

Link: <https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/firstFit.cpp>

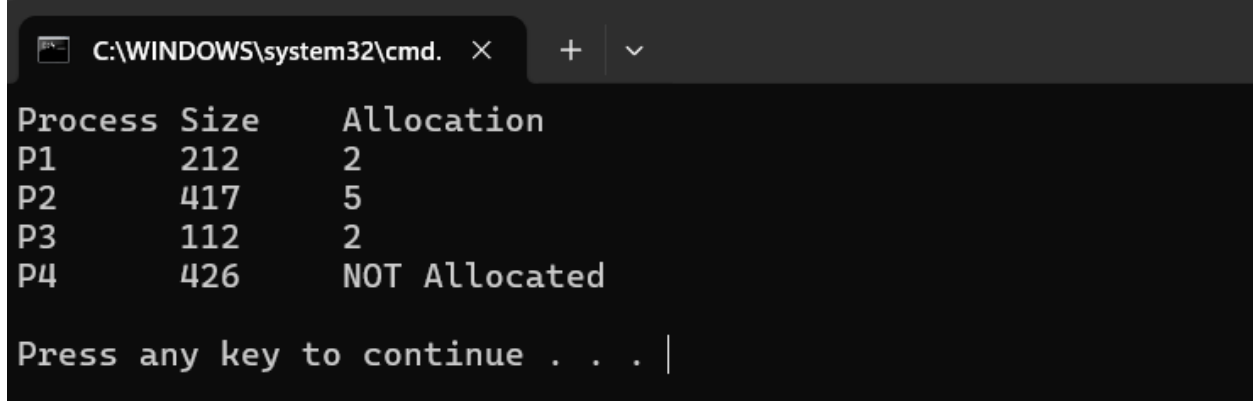
Input:

Block Sizes: {100, 500, 200, 300, 600}

Process Sizes: {212, 417, 112, 426}

Output:

```
vector<int>block_size = {100, 500, 200, 300, 600};
vector<int>process_size = {212, 417, 112, 426};
// ... (rest of the code) ...
```



Process	Size	Allocation
P1	212	2
P2	417	5
P3	112	2
P4	426	NOT Allocated

Press any key to continue . . . |

Code:

//Worst Fit

```
#include<bits/stdc++.h>
```

```
using namespace std;
```

```

int main()
{
    vector<int>block_size = {100, 500, 200, 300, 600};
    vector<int>process_size = {212, 417, 112, 426};
    int n = process_size.size();
    int m = block_size.size();
    vector<int>alloc(n, -1);
    for(int i = 0; i<n; i++)
    {
        int worstIndex = -1;
        for(int j = 0; j<m; j++)
        {
            if(block_size[j]>process_size[i])
            {
                if(worstIndex == -1 || block_size[j]>block_size[worstIndex])
                {
                    worstIndex = j;
                }
            }
        }
        if(worstIndex != -1)
        {
            alloc[i] = worstIndex+1;
            block_size[worstIndex]-=process_size[i];
        }
    }
}

```

```

    }

    cout<<"Process\tSize\tBlock"<<endl;

    for(int i = 0; i<n; i++)
    {
        cout<<"P"<<i+1<<"\t";

        cout<<process_size[i]<<"\t";

        if(alloc[i] == -1)
        {
            cout<<"NOT Allocated"<<endl;
        }

        else cout<<alloc[i]<<endl;
    }
}

```

Link:

<https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/worstFit.cpp>

Input:

Block Sizes: {100, 500, 200, 300, 600}

Process Sizes: {212, 417, 112, 426}

Output:

```
C++ worstFit.cpp X
D: > OS > OS Code > C++ worstFit.cpp > ...
1  #include<bits/stdc++.h>
2  using namespace std;
3  int main()
4  {
5      vector<int>block_size = {100, 500, 200, 300, 600};
6      vector<int>process_size = {212, 417, 112, 426};
7
8
9
PROBLEMS
* Executi
Press any key to continue . . . |
```

Process	Size	Block
P1	212	5
P2	417	2
P3	112	5
P4	426	NOT Allocated

Code:

//Best Fit

```
#include<bits/stdc++.h>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    vector<int>block_size = {100, 500, 200, 300, 600};
```

```
    vector<int>process_size = {212, 417, 112, 426};
```

```
    int n = block_size.size();
```

```
    int m = process_size.size();
```

```
    vector<int>alloc(m, -1);
```

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

```

{
    int bestIdx = -1;
    for(int j = 0; j<n; j++)
    {
        if(block_size[j]>=process_size[i])
        {
            if(bestIdx == -1 || block_size[j]<block_size[bestIdx])
            {
                bestIdx = j;
            }
        }
    }
    if(bestIdx!=-1)
    {
        alloc[i] = bestIdx+1;
        block_size[bestIdx]-=process_size[i];
    }
}

cout<<"Process\tSize\tAllocation"<<endl;
for(int i = 0; i<m; i++)
{
    cout<<"P"<<i+1<<"\t";
    cout<<process_size[i]<<"\t";
    if(alloc[i]==-1) cout<<"NOT Allocated"<<endl;
    else cout<<alloc[i]<<endl;
}

```

```
}  
  
return 0;  
  
}
```

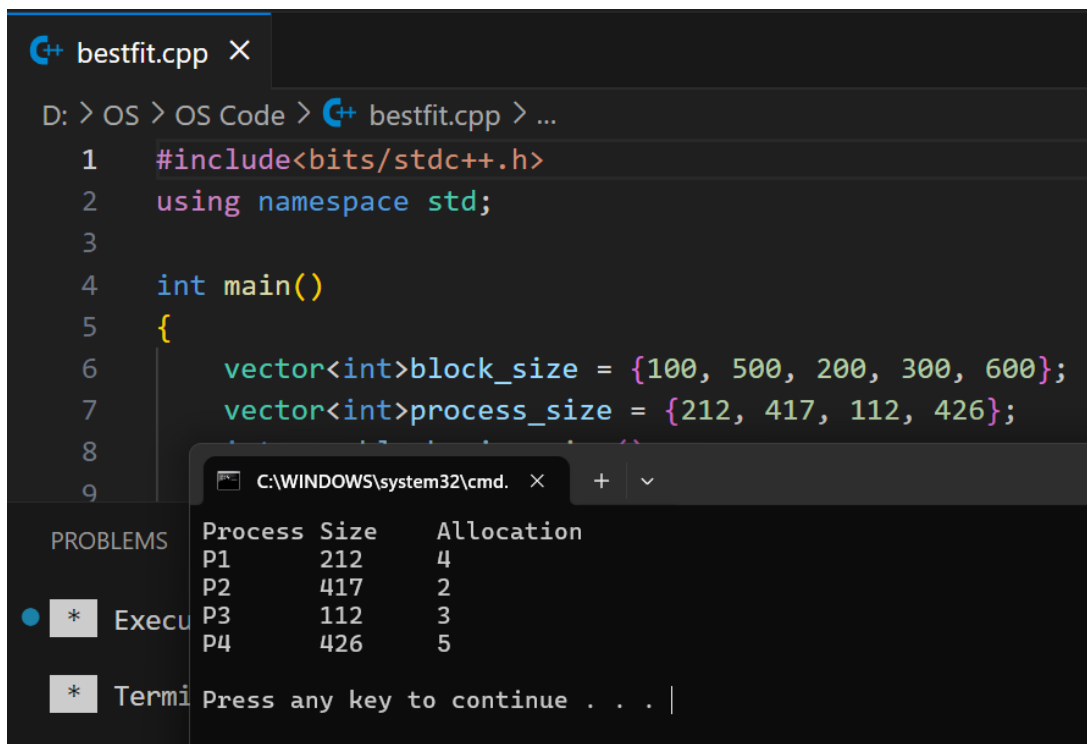
Link: <https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/bestfit.cpp>

Input:

Block Sizes: {100, 500, 200, 300, 600}

Process Sizes: {212, 417, 112, 426}

Output:



The screenshot shows a C++ IDE with a file named `bestfit.cpp`. The code defines two vectors: `block_size` with values {100, 500, 200, 300, 600} and `process_size` with values {212, 417, 112, 426}. Below the code, a terminal window displays the output of the program. The output is a table with three columns: 'Process', 'Size', and 'Allocation'. It lists four processes: P1 (Size 212, Allocation 4), P2 (Size 417, Allocation 2), P3 (Size 112, Allocation 3), and P4 (Size 426, Allocation 5). The terminal also shows a prompt 'Press any key to continue . . . |'.

```
1  #include<bits/stdc++.h>  
2  using namespace std;  
3  
4  int main()  
5  {  
6      vector<int>block_size = {100, 500, 200, 300, 600};  
7      vector<int>process_size = {212, 417, 112, 426};  
8  
9  
PROBLEMS  
* Execu  
* Termi  
Process Size    Allocation  
P1      212      4  
P2      417      2  
P3      112      3  
P4      426      5  
Press any key to continue . . . |
```

Discussion:

Contiguous memory allocation is an essential technique used in modern operating systems to allocate memory to processes. First Fit, Best Fit, and Worst Fit are popular algorithms used for contiguous memory allocation. Each algorithm has its advantages and disadvantages, but all are designed to optimize memory allocation and reduce fragmentation. With these techniques, operating systems can efficiently manage memory, making them a critical component of computer science and software engineering.

11.Pagging:

Introduction:

Paging is a mechanism used to manage memory by dividing both physical memory and processes' logical memory into fixed-sized blocks. These blocks are called **frames** (in physical memory) and **pages** (in logical memory). The system maintains a **page table** for each process, which maps logical pages to physical frames.

Code:

```
#include<bits/stdc++.h>

using namespace std;

int main()
{
    cout<<"Enter the Logical Address : "<<endl;

    int Logical_Address;

    cin>>Logical_Address;

    vector<int>Page_table({5, 6, 1, 2});
```

```

int page_size;

cout<<"Enter the Page Size : "<<endl;

cin>>page_size;

int page_no = Logical_Address/page_size;

int frame_no = Page_table[page_no];

int offset = Logical_Address%page_size;

int physical_address = frame_no*page_size + offset;

cout<<"Physical Address is : "<<endl;

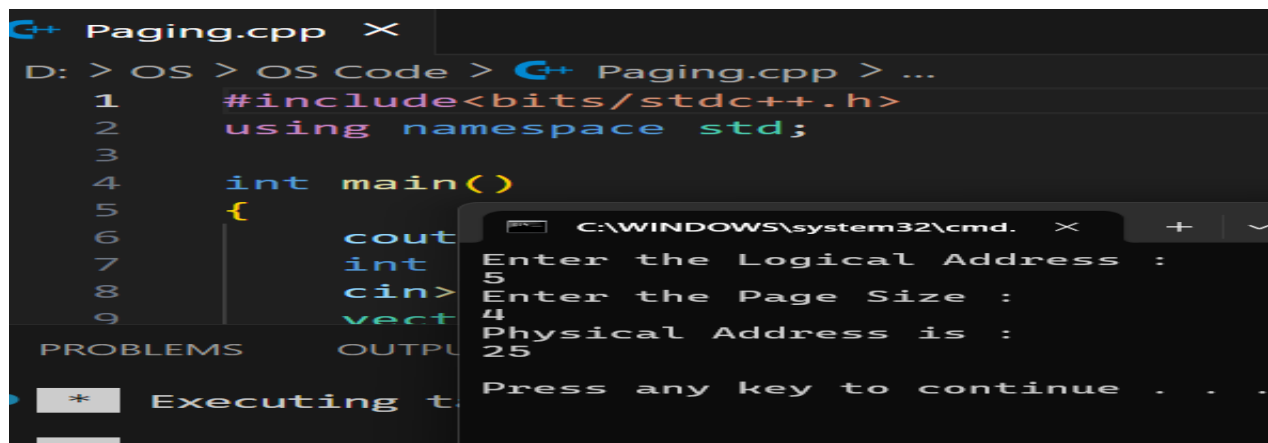
cout<<physical_address<<endl;

}

```

Link: <https://github.com/FaisalHoqueRifat/OS/blob/main/OS%20Code/Paging.cpp>

Output:



The screenshot shows a C++ IDE with a file named 'Paging.cpp'. The code is as follows:

```

1  #include<bits/stdc++.h>
2  using namespace std;
3
4  int main()
5  {
6      cout<<"Enter the Logical Address : ";
7      int Logical_Address;
8      cin>>Logical_Address;
9      cout<<"Enter the Page Size : ";
10     int Page_Size;
11     cin>>Page_Size;
12     int page_no = Logical_Address/Page_Size;
13     int frame_no = Page_table[page_no];
14     int offset = Logical_Address%Page_Size;
15     int physical_address = frame_no*Page_Size + offset;
16     cout<<"Physical Address is : ";
17     cout<<physical_address<<endl;
18 }

```

The output window shows the following execution:

```

Enter the Logical Address : 5
Enter the Page Size : 4
Physical Address is : 25
Press any key to continue . . .

```

Discussion:

Paging is fundamental in modern operating systems, balancing the need for efficient memory use and process isolation with the challenges of maintaining translation and performance overheads. Optimizations like multi-level paging, inverted page tables, and TLBs are critical to addressing these challenges.

