INDEX

S.No.	Experiment Name	Page no.	Date
1	FCFS Scheduling		
2	SJF Scheduling		
3	SRTF Scheduling		
4	RR Scheduling		
5	Priority Scheduling		
6	MLFQ Scheduling		
7	Banker's Algorithm for Deadlock Avoidance		
8	Deadlock Detection Algorithm		
9	FIFO Page Replacement Algorithm		
10	LRU Page Replacement Algorithm		
11	LFU Page Replacement Algorithm		
12	Contiguous Allocation Algorithm		
13	Linked Allocation Algorithm		
14	Indexed Allocation Algorithm		

Scanned with OKEN Scanner

Experiment No. 01

FIRST COME FIRST SERVE CPU SCHEDULING ALGORITHM

AIM:

To write a program to simulate the CPU scheduling algorithm First Come First Serve (FCFS)

DESCRIPTION:

The First Come First Serve (FCFS) scheduling algorithm is a non-preemptive scheduling algorithm where processes are executed in the order they arrive in the ready queue. This program simulates the FCFS algorithm by taking user input for arrival times and burst times of processes. It then calculates the waiting time, completion time, and average waiting time for all processes.

ALGORITHM:

1. Input the number of processes, arrival time, and burst time for each process.
2. Sort the processes based on their arrival time.
3. For each process:
    - Waiting Time (WT) calculation:
        - Calculate waiting time for the current process as the difference between the completion time of the previous process (or the arrival time of the current process if it arrives later) and its arrival time.
    - Turnaround Time (TAT) calculation:
        - Calculate turnaround time for the current process as the sum of its burst time and waiting time.
4. Display the arrival time, burst time, completion time, waiting time, and turnaround time for each process.
5. Calculate the total waiting time and total turnaround time for all processes.
6. Display the average waiting time and average turnaround time by dividing the total waiting time and total turnaround time by the number of processes, respectively.

PROGRAM:

```
#include <iostream>
#include <algorithm>


using namespace std;


struct Process {
      int id;
      int arrival_time;
      int burst_time;
};

bool compareArrivalTime(const Process &a, const Process &b) {
      return a.arrival_time < b.arrival_time;
}

int main() {


      int n;
      cout << "Enter the number of processes: ";
```


Scanned with OKEN Scanner

```
cin >> n;


Process processes[n];
cout << "Enter arrival time and burst time for each process:\n";
for (int i = 0; i < n; ++i) {
    processes[i].id = i + 1;
    cout << "Process " << processes[i].id << ":\n";
    cout << "\tArrival time: ";
    cin >> processes[i].arrival_time;
    cout << "\tBurst time: ";
    cin >> processes[i].burst_time
 }

 sort(processes, processes + n, compareArrivalTime);

 int total_waiting_time = 0;
 int total_turnaround_time = 0;


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


int current_time = 0;


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

   int waiting_time = max(0, current_time -
                                    processes[i].arrival_time);

   int completion_time = current_time + processes[i].burst_time;
   int turnaround_time = waiting_time + processes[i].burst_time;

   total_waiting_time += waiting_time;
   total_turnaround_time += turnaround_time;

   cout << "P" << processes[i].id << "\t" << processes[i].arrival_time
        << "\t\t" <<  processes[i].burst_time << "\t\t"
        << completion_time   << "\t\t" << waiting_time << "\t\t"
        << turnaround_time << "\n";

   current_time = completion_time;
}
```


Scanned with OKEN Scanner

```
    cout << "Avg. Waiting Time: " << (double)total_waiting_time / n
        << endl;
    cout << "Avg. Turnaround Time: " << (double)total_turnaround_time / n
        << endl;
    return 0;
}
```

OUTPUT:

```
Enter the number of processes: 3
Enter arrival time and burst time for each process:

Process 1:
   Arrival time: 0
   Burst time: 3
Process 2:
   Arrival time: 1
   Burst time: 2
Process 3:
   Arrival time: 2
   Burst time: 4


Process   Arrival     Burst   Completion   Waiting   Turnaround
   ID       Time       Time       Time        Time        Time

 P1         0          3          3           0           3

 P2         1          2          5           2           4

 P3         2          4          9           5           9


Average Waiting Time: 2.33333
Average Turnaround Time: 5.33333
```


Scanned with OKEN Scanner

Experiment No. 02

SHORTEST JOB FIRST CPU SCHEDULING ALGORITHM

AIM:

To write a program to stimulate the CPU scheduling algorithm Shortest Job First (Non-Preemptive).

DESCRIPTION:

The Shortest Job First (SJF) scheduling algorithm is a non-preemptive scheduling algorithm that aims to minimize the average waiting time for processes. It selects the process with the shortest burst time for execution first. This program simulates the SJF algorithm by taking user input for arrival times and burst times of processes. It then sorts the processes based on burst time and calculates the waiting time, completion time, and average waiting time for all processes.

ALGORITHM:

1. Get the number of processes from the user.
2. Create arrays to store arrival time and burst time for each process.
3. Iterate through each process:
   - Get arrival time and burst time from the user.
4. Sort the processes based on their burst time (ascending order).
5. Calculate waiting time for each process (similar to FCFS but based on the order after sorting).
6. Calculate completion time for each process (similar to FCFS).
7. Calculate average waiting time by dividing the total waiting time by the number of processes.

PROGRAM:

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

struct Processes{
   int pid;
   int arrival_time;
   int burst_time;
   int completion_time;
   int turnAround_time;
   int waiting_time;
};

//to sort as per arrival time
bool compareArrivalTime(const Processes &a, const Processes &b) {
   return a.arrival_time < b.arrival_time;
}

//to sort back as per input so that output would be in the same order
bool comparePid(const Processes &a, const Processes &b) {
   return a.pid < b.pid;
}

void SJF_Implementation(vector<Processes>& P){
```


Scanned with OKEN Scanner

```
//sorted according to arrival time
sort(P.begin(), P.end(), compareArrivalTime);

int n = P.size();
int total_burst_time = 0;
//to calculate total burst time which is order independent
for (int i = 0; i < n; i++) total_burst_time += P[i].burst_time;

//scheduling starts with processes with lowest arrival time
int current_time = P[0].arrival_time;

 //to store the processes who have arrived in as per current time

// (arrival time < current time)
vector<vector<int>> v;

int index = 0; //to keep track of processes that have arrived

while(total_burst_time > 0){

    //store current index of processes and their burst time in vector v
    while(current_time >= P[index].arrival_time && index < n){

        v.push_back({index,P[index].burst_time});

        //sort in order that smallest burst time first
        if(v.size()>1){
        sort(v.begin(),
             v.end(),
             [](const vector<int> &a, const vector<int> &b){
                    return a[1]< b[1];
                }
            );
        }
        index++;
    }
    // calculation of completion time, turnaround time and waiting time
    if(!v.empty()) {
        P[v[0][0]].completion_time = current_time + v[0][1];

        P[v[0][0]].turnAround_time = P[v[0][0]].completion_time
                                    - P[v[0][0]].arrival_time;
        P[v[0][0]].waiting_time = P[v[0][0]].turnAround_time
                                    - P[v[0][0]].burst_time;

        //updating values for current_time and total_brust_time
        current_time = current_time + P[v[0][0]].burst_time;
        total_burst_time = total_burst_time - P[v[0][0]].burst_time;
```


Scanned with OKEN Scanner

```
            //pop the first element from vector as its execution is completed
            //and new first element is again one with lowest job burst time
            v.erase(v.begin());

        }
    }

    //sort back as of input order
    sort(P.begin(), P.end(), comparePid);

    //to get avg tat and avg wt
    int total_tat = 0; int total_wt = 0;

    //print the output table
    cout << "Process_ID \tArrival_Time \tBurst_Time \tWaiting_Time
            \tTurnaround_Time \tCompletion_Time \n";


    for (auto& p : P) {

        total_tat += p.turnAround_time; total_wt += p.waiting_time;

        cout << "P"<< p.pid << "\t\t\t" << p.arrival_time << "\t\t"
             << p.burst_time << "\t\t" << p.waiting_time << "\t\t"
             << p.turnAround_time << "\t\t" << p.completion_time
             << "\n";
    }
    cout << "Average turn_around_time = "
         << static_cast<float>(total_tat+0.0)/n << endl;
    cout << "Average waiting_time = "
         << static_cast<float>(total_wt)/n << endl;
}
int main(){
    int n;
    cout << "Enter the number of Processes: ";
    cin >> n;

    vector<Processes> P(n);
    for (int i = 0; i < n; ++i) {
        P[i].pid = i + 1;
        cout << "Arrival Time & Burst Time for P" << i + 1 << " : ";
        cin >> P[i].arrival_time >> P[i].burst_time;
    }
    SJF_Implementation(P);
}
```


Scanned with OKEN Scanner

OUTPUT:

```
Enter the number of Processes: 5

Arrival Time & Burst Time for P1 : 3 4
Arrival Time & Burst Time for P2 : 0 3
Arrival Time & Burst Time for P3 : 3 2
Arrival Time & Burst Time for P4 : 2 5
Arrival Time & Burst Time for P5 : 3 3
```

Process ID	Arrival Time	Burst Time	Waiting Time	Turnaround Time	Completion Time
P1	3	4	5	9	12
P2	0	3	0	3	3
P3	3	2	0	2	5
P4	2	5	10	15	17
P5	3	3	2	5	8

```
Average turn_around_time = 6.8
Average waiting_time = 3.4
```


Scanned with OKEN Scanner

Experiment No. 03

SHORTEST REMAINING TIME FIRST CPU SCHEDULING ALGORITHM

AIM:

To write a program to simulate the CPU scheduling algorithm Shortest Remaining Time First (SRTF).

DESCRIPTION:

The Shortest Remaining Time First (SRTF) scheduling algorithm is a preemptive version of the Shortest Job First (SJF) algorithm. It selects the process with the shortest remaining burst time for execution. If a new process arrives with a shorter remaining burst time, the currently executing process is preempted, and the new process is given the CPU. This program simulates the SRTF algorithm by taking user input for arrival times and total burst times of processes. It then calculates the execution time for each process based on preemption and tracks the waiting time, completion time, and average waiting time.

ALGORITHM:

1. Get the number of processes from the user.
2. Create arrays to store arrival time, total burst time, and remaining burst time for each process.
3. Iterate through a time quantum (a small time slice).
   - For each process:
     - Decrement the remaining burst time by the time quantum.
     - If the remaining burst time becomes zero, the process is complete.
       - Calculate its waiting time and completion time.
     - If a new process arrives with a shorter remaining burst time than the currently executing process:
       - Preempt the current process and add its remaining burst time back.
       - Set the new process as the running process.
4. Repeat step 3 until all processes are complete.

Calculate the average waiting time by dividing the total waiting time by the number of processes.

PROGRAM:

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

struct Processes{
 int pid;
 int arrival_time;
 int burst_time;
 int completion_time;
 int turnAround_time;
 int waiting_time;
};

//to sort as per arrival time
bool compareArrivalTime(const Processes &a, const Processes &b) {
 return a.arrival_time < b.arrival_time;
```


Scanned with OKEN Scanner

```
}

//to sort back as per input so that output would be in the same order
bool comparePid(const Processes &a, const Processes &b) {
 return a.pid < b.pid;
}

void SRTF_Implementation(vector<Processes>& P){

 //sorted according to arrival time
 sort(P.begin(), P.end(), compareArrivalTime);

 int n = P.size();
 int total_burst_time = 0;

 //to calculate total burst time which is order independent
 queue<int> arrivalTimeRecord; //queue to store arrival time (sorted)
 for (int i = 0; i < n; i++) {
          total_burst_time += P[i].burst_time;
          arrivalTimeRecord.push(P[i].arrival_time);
 }

 arrivalTimeRecord.pop(); //popping 1st arrival time as scheduling
                          //starts from this point of time

 //scheduling starts with processes with lowest arrival time
 int current_time = P[0].arrival_time;

 //to store the processes who have arrived in as per current time
    //(arrival time < current time)
 vector<vector<int>> v;

 int index = 0; //to keep track of processes that have arrived

 while(total_burst_time > 0){

          //store index of processes and their burst time in vector v
          while(current_time >= P[index].arrival_time && index < n) {
               v.push_back({index,P[index].burst_time});
               index++;
          }

          //sort in order that smallest burst time first
          if(v.size()>1){
              sort(v.begin(),
                   v.end(),
                   [](const vector<int> &a, const vector<int> &b){
                       return a[1] < b[1];
                       }
  );
          }
```


Scanned with OKEN Scanner

```
            if(!arrivalTimeRecord.empty()){

                while(current_time != arrivalTimeRecord.front() &&
                                    P[v[0][0]].burst_time != 0 ){

                 v[0][1] = v[0][1] - 1;
                 current_time++;
                 total_burst_time--;
                }
               if(P[v[0][0]].burst_time == 0 && !v.empty())
              {
                P[v[0][0]].completion_time = current_time;
                P[v[0][0]].turnAround_time = P[v[0][0]].completion_time
                                            - P[v[0][0]].arrival_time;
                P[v[0][0]].waiting_time = P[v[0][0]].turnAround_time
                                            - v[0][1];
                 v.erase(v.begin());

                } else {
                 arrivalTimeRecord.pop();
                }

            } else{

          // calculation of completion time, turnaround time and waiting
        // time after covering AT of all processes ie no new process now
        // it will work as SJF as no new process to preempt

                 if(!v.empty()) {
                P[v[0][0]].completion_time = current_time + v[0][1];
                P[v[0][0]].turnAround_time = P[v[0][0]].completion_time
                                            - P[v[0][0]].arrival_time;
                P[v[0][0]].waiting_time = P[v[0][0]].turnAround_time
                                            - P[v[0][0]].burst_time;

                //updating values for current_time and total_brust_tim
                      current_time = current_time + v[0][1];
                      total_burst_time = total_burst_time - v[0][1];

                //pop the first element from vector as its executed fully
                //& new first element is again one with lowest job burst time
                      v.erase(v.begin());
                }
            }
        }
            //sort back as of input order
        sort(P.begin(), P.end(), comparePid);

        //to get avg tat and avg wt
        int total_tat = 0; int total_wt = 0;
```


Scanned with OKEN Scanner

```
//print the output table
cout << "Process_ID \tArrival_Time \tBurst_Time \tWaiting_Time
            \tTurnaround_Time\tCompletion_Time\n";

for (auto& p : P) {

         total_tat += p.turnAround_time;
        total_wt += p.waiting_time;

        cout << "P"<< p.pid << "\t\t\t" << p.arrival_time << "\t\t"
             << p.burst_time << "\t\t" << p.waiting_time << "\t\t"
             << p.turnAround_time << "\t\t" << p.completion_time << "\n";
 }

      cout << "Average turn_around_time = " <<
                                static_cast<float>(total_tat+0.0)/n << endl;
 cout << "Average waiting_time = " <<
                                static_cast<float>(total_wt)/n << endl;

}

int main(){
 int n;
 cout << "Enter the number of Processes: ";
 cin >> n;

 vector<Processes> P(n);

 for (int i = 0; i < n; ++i) {
          processes[i].id = i + 1;
          cout << "Process " << processes[i].id << ":\n";
          cout << "\tArrival time: ";  cin >> processes[i].arrival_time;
         cout << "\tBurst time: ";    cin >> processes[i].burst_time;
      }

 SRTF_Implementation(P);
}
```


Scanned with OKEN Scanner

OUTPUT:

```
Enter the number of Processes: 5

Arrival Time & Burst Time for P1 : 2 7
Arrival Time & Burst Time for P2 : 3 12
Arrival Time & Burst Time for P3 : 4 5
Arrival Time & Burst Time for P4 : 6 9
Arrival Time & Burst Time for P5 : 5 3
```

Process ID	Arrival Time	Burst Time	Waiting Time	Turnaround Time	Completion Time
P1	2	7	3	10	12
P2	3	12	23	35	38
P3	4	5	8	13	17
P4	6	9	11	20	26
P5	5	3	0	3	8

```
Average turn_around_time = 16.2
Average waiting_time = 9
```


Scanned with OKEN Scanner

Experiment No. 04

ROUND ROBIN CPU SCHEDULING ALGORITHM

AIM:

To write a program to simulate the CPU scheduling Round Robin (RR) algorithm.

DESCRIPTION:

The Round Robin (RR) scheduling algorithm is a preemptive scheduling algorithm that provides a fair share of CPU time to each process. It employs a time quantum (a small time slice) and assigns the CPU to each process in a round-robin fashion for a specific time quantum. If a process doesn't finish within the time quantum, it's preempted and added back to the ready queue. This program simulates the RR algorithm by taking user input for arrival times, total burst times, and the time quantum. It then tracks the execution time for each process based on preemption and calculates the waiting time, completion time, and average waiting time.

ALGORITHM:

1. Get the number of processes, time quantum, and arrival time and total burst time for each process from the user.
2. Create arrays to store arrival time, total burst time, remaining burst time, and waiting time for each process.
3. Iterate through a time quantum.
    - For the currently running process:
        - Decrement the remaining burst time by the time quantum.
        - If the remaining burst time becomes zero, the process is complete.
            - Calculate its waiting time and completion time.
        - Otherwise, add the process back to the ready queue (preemption).
    - If the ready queue is not empty, select the next process for execution.
4. Repeat step 3 until all processes are complete.
5. Calculate the average waiting time by dividing the total waiting time by the number of processes.

PROGRAM:

```
#include<bits/stdc++.h>
using namespace std;

struct Processes{
 int pid;
 int arrival_time;
 int burst_time;
 int completion_time;
 int turnAround_time;
 int waiting_time;
};

//to sort as per arrival time
bool compareArrivalTime(const Processes &a, const Processes &b) {
 return a.arrival_time < b.arrival_time;
}

//to sort back as per input so that output would be in the same order
```


Scanned with OKEN Scanner

```
bool comparePid(const Processes &a, const Processes &b) {
 return a.pid < b.pid;
}

void RR_Implementation(vector<Processes>& P, int TQ){

 //sorted according to arrival time
 sort(P.begin(), P.end(), compareArrivalTime);

 int n = P.size();
 int total_burst_time = 0;

 //queue to get track of turns
 queue<int> turnQueue;
 turnQueue.push(0);
 int brustTime[n];
 for (int i = 0; i < n; i++) {
          total_burst_time += P[i].burst_time;
          brustTime[i] = P[i].burst_time; //to store updated burst time
 }

 int index = 1;
 int current_time = P[0].arrival_time;


      while(!turnQueue.empty()){


          while(current_time + TQ >= P[index].arrival_time && index < n){
                turnQueue.push({index});
                index++;
          }


          int var = turnQueue.front();


          if(brustTime[var] - TQ > 0){
              brustTime[var] = brustTime[var] - TQ;
              turnQueue.push(var);
              turnQueue.pop();
              current_time = current_time + TQ;
         }else {
              current_time = current_time + brustTime[var];
              P[var].completion_time = current_time;
              P[var].turnAround_time = current_time - P[var].arrival_time;
              P[var].waiting_time = P[var].turnAround_time
                                                  - P[var].burst_time;
              brustTime[var] = 0;
              turnQueue.pop();
         }
```


Scanned with OKEN Scanner

```
        }

        //sort back as of input order
        sort(P.begin(), P.end(), comparePid);

        //to get avg tat and avg wt
        int total_tat = 0; int total_wt = 0;


        //print the output table
        cout << "Process_ID \tArrival_Time \tBurst_Time \tCompletion_Time
                \tTurnaround_Time \tWaiting_Time\n";

        for (auto& p : P) {

            total_tat += p.turnAround_time;
            total_wt += p.waiting_time;

            cout << "P"<< p.pid << "\t\t\t" << p.arrival_time << "\t\t"
                << p.burst_time << "\t\t" << p.completion_time << "\t\t"
                << p.turnAround_time << "\t\t" << p.waiting_time << "\n";

        }
        cout << "Average turn_around_time = "
                                << static_cast<float>(total_tat+0.0)/n << endl;
        cout << "Average waiting_time = "
                                << static_cast<float>(total_wt)/n << endl;
    }
    int main(){
        int n,t;
        cout << "Enter the number of Processes: ";
        cin >> n;

        vector<Processes> P(n);
        for (int i = 0; i < n; ++i) {
        P[i].pid = i + 1;
        cout << "Arrival Time & Burst Time for P" << i + 1 << " : ";
        cin >> P[i].arrival_time >> P[i].burst_time;
        }
        cout << "Enter Time-Quantum: ";
        cin >> t;
        RR_Implementation(P,t);


    }
```


Scanned with OKEN Scanner

OUTPUT:

```
Enter the number of Processes: 4


Arrival Time & Burst Time for P1 : 2 4
Arrival Time & Burst Time for P2 : 1 3
Arrival Time & Burst Time for P3 : 4 5
Arrival Time & Burst Time for P4 : 6 1


Enter Time-Quantum: 2
```

Process ID	Arrival Time	Burst Time	Waiting Time	Turnaround Time	Completion Time
P1	2	4	10	8	4
P2	1	3	6	5	2
P3	4	5	14	10	5
P4	6	1	11	5	4

```
Average turn_around_time = 7
Average waiting_time = 3.75
```


Scanned with OKEN Scanner

Experiment No. 05

PRIORITY CPU SCHEDULING ALGORITHM

AIM:

To write a program to simulate the CPU Scheduling Priority Algorithm.

DESCRIPTION:

The Priority CPU Scheduling Algorithm is a non-preemptive (process runs to completion without interruption) or preemptive (process can be interrupted by a higher priority process) technique for assigning CPU time to processes. Each process is assigned a priority level, with higher numbers indicating higher priority.

ALGORITHM:

1. Get the number of processes, and for each process:
   - Arrival time
   - Burst time
   - Static priority (higher number represents higher priority)
2. Sort the processes based on their arrival time.
   - Iterate through the sorted list of processes:
   - If a process has arrived and is not yet completed, select the process with the highest priority (lowest numerical value).
   - Execute the process for its entire burst time.
3. Update the completion time for the process.
4. Calculate the waiting time for each process (completion time - arrival time - burst time).
5. Calculate the average waiting time by dividing the total waiting time by the number of processes.

PROGRAM:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct Process {
      int pid;
      int arrival_time;
      int burst_time;
      int priority;
      int completion_time = 0;
      int turnaround_time = 0;
      int waiting_time = 0;
};

// functions to pass as parameters of sorting processes vector
//comparing priority arrival_time and Pid(for output in same order as of
input)

bool comparePriority(const Process &a, const Process &b) {
```


Scanned with OKEN Scanner

```
        if ( a.priority == b.priority) return a.arrival_time < b.arrival_time;
        return a.priority > b.priority;
    }

    bool compareAT(const Process &a, const Process &b) {
        return a.arrival_time < b.arrival_time;
    }

    bool comparePid(const Process &a, const Process &b) {
        return a.pid < b.pid;
    }

    void PriorityScheduling(vector<Process>& processes) {

        //sort according to arrival time
        sort(processes.begin(), processes.end(), compareAT);

        int n = processes.size(), current_time = processes[0].arrival_time;

        int ii = 1;
        //to store arrived process in Ready queue
        vector<Process> ReadyQueue;
        //to store executed process records for output
        vector<Process> executed;

        //as processes are sorted acc. to AT earliest to arrive is ready first
        ReadyQueue.push_back(processes[0]);
        //to keep track of process as process with higher priority in

        // ReadyQueue preempt current running process
        Process current_process;

        while(!ReadyQueue.empty()){

            current_process = ReadyQueue.front();

            //per unit time execution
            current_time++;
            Current_process.burst_time--;

            //if process is completed executed
            if(current_process.burst_time == 0){

                current_process.completion_time = current_time;
                ReadyQueue.erase(ReadyQueue.begin());
                executed.push_back(current_process); //store executed process

                if(!ReadyQueue.empty()) current_process = ReadyQueue.front();
                else break;

                continue;
```


Scanned with OKEN Scanner

```
            }

            //to get processes into ReadyQueue acc. to their A.T
            while(processes[ii].arrival_time <= current_time && ii < n){
                ReadyQueue.push_back(processes[ii]);
                ii++;
            }

            ReadyQueue.erase(ReadyQueue.begin());
            ReadyQueue.push_back(current_process);
              //with updated B.T(remaining)sort the Queue based on Priority
            sort(ReadyQueue.begin(), ReadyQueue.end(), comparePriority);
        }

        //sort input vector as executed vector as per Pid
         sort(processes.begin(), processes.end(), comparePid);
        sort(executed.begin(), executed.end(), comparePid);

         //get completion time from executed vector and calculate T.A.T and W.T
        for(int i = 0; i < n; i++){
            processes[i].completion_time = executed[i].completion_time;
            processes[i].turnaround_time = processes[i].completion_time
                                                - processes[i].arrival_time;
            processes[i].waiting_time = processes[i].turnaround_time
                                                - processes[i].burst_time;
        }

        //to get avg tat and avg wt
        int total_tat = 0; int total_wt = 0;

        // Output the result
        cout << "Process_ID \tArrival_Time \tBurst_Time \tPriority

                \tWaiting_Time \tTurnaround_Time \tCompletion_Time\n";

        for (auto& p : processes) {

             total_tat += p.turnaround_time;
             total_wt += p.waiting_time;

             cout << "P" << p.pid << "\t\t\t" << p.arrival_time << "\t\t"
                  << p.burst_time << "\t\t" << p.priority << "\t\t"

                  << p.waiting_time << "\t\t" << p.turnaround_time

                  << "\t\t" << p.completion_time << "\n";
        }
        cout << "Average turn_around_time = "
                                << static_cast<float>(total_tat+0.0)/n << endl;
        cout << "Average waiting_time = "
                                << static_cast<float>(total_wt)/n << endl;
    }
```


Scanned with OKEN Scanner

```
int main() {
    cout << "This program demonstrates priority scheduling algorithm \n"
         << "higher no indicates higher priority " < endl;
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;
    vector<Process> processes(n);
    for (int i = 0; i < n; ++i) {
        processes[i].pid = i + 1;
        cout << "Arrival Time, Burst Time, Priority for Process P"
             << i + 1 << ": ";
        cin >> processes[i].arrival_time >> processes[i].burst_time
            >> processes[i].priority;
    }
    PriorityScheduling(processes);
    return 0;
}
```

OUTPUT:

```
This program demonstrates priority scheduling algorithm

higher no indicates higher priority


Enter the number of processes: 5


Arrival Time, Burst Time, Priority for Process P1: 7 7 7

Arrival Time, Burst Time, Priority for Process P2: 8 8 8

Arrival Time, Burst Time, Priority for Process P3: 3 7 4

Arrival Time, Burst Time, Priority for Process P4: 6 2 7

Arrival Time, Burst Time, Priority for Process P5: 1 3 2
```

Process_ID	A.T	B.T	Priority	W.T	T.A.T	C.T
P1	7	7	7	9	16	23
P2	8	8	8	0	8	16
P3	3	7	4	17	24	27
P4	6	2	7	0	2	8
P5	1	3	2	24	27	28

```
Average turn_around_time = 15.4
Average waiting_time = 10
```


Scanned with OKEN Scanner

Experiment No. 06

MULTI LEVEL FEEDBACK QUEUE CPU SCHEDULING ALGORITHM

AIM:

To write a program to simulate the Multilevel Feedback Queue (MLFQ) CPU Scheduling Algorithm.

DESCRIPTION:

The MLFQ algorithm organizes processes into multiple queues, each with different priority levels. Processes are initially placed in a queue with a high priority. If a process uses too much CPU time, it is demoted to a lower priority queue. Conversely, if a process does not use much CPU time, it may be promoted to a higher priority queue. This dynamic adjustment of priorities allows the scheduler to respond effectively to different types of processes.

ALGORITHM:

1. Create multiple queues, each with a different priority level.
2. Initialize processes in the highest priority queue.
3. Execute processes in the highest priority queue until it is empty.
4. If a process uses the CPU for a certain time quantum and is not finished, demote it to the next lower priority queue.
5. Repeat steps 3-4 until all queues are empty or a termination condition is met.
6. Optionally, processes that wait for a long time in a lower priority queue may be promoted to a higher priority queue.

PROGRAM:

```
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>
#include <stdexcept> //for validating priority
#define maxTQ 10000 // for processes with priority level 0 - no preemption

using namespace std;
struct Process {
   int pid;
   int arrival_time;
   int burst_time;
   int priority;
   int completion_time;
   int turnAround_time;
   int waiting_time;
};

bool compareAT(const Process &a, const Process &b) {
   return a.arrival_time < b.arrival_time;
}

bool comparePid(const Process &a, const Process &b) {
   return a.pid < b.pid;
}
// Function to perform MLFQ scheduling
void MLFQ_Scheduling(vector<Process>& processes) {
```


Scanned with OKEN Scanner

```
    sort(processes.begin(), processes.end(), compareAT);
    int n = processes.size(), current_time = processes[0].arrival_time;
    int ct[n]; // To store completion time


    vector<queue<Process>> priorityQueues(3);
                                // 3 priority levels are considered here

    int completedProcesses = 0;
    int ii = 0;
    int count = 0;

  while (completedProcesses < n && count++ < 30) {

          // Add arriving processes to the appropriate priority queue
          // Add arriving processes to the appropriate priority queue
          while (ii < n && processes[ii].arrival_time <= current_time) {
              priorityQueues[processes[ii].priority].push(processes[ii]);
              ii++;
         }

         // Choose the highest priority non-empty queue
         int chosenQueue = -1;
         for (int i = 0; i < 3; ++i) {
             if (!priorityQueues[i].empty()) {
           chosenQueue = i;
           break;
            }
        }

        if (chosenQueue == -1) {
           current_time++;
           continue; // No processes in any queue, proceed to next time unit
        }

        int quantum = (chosenQueue != 0) ?

                                  ((chosenQueue == 1) ? 2 : 4) : maxTQ;

      // Execute processes from the chosen queue for a quantum or until
        // completion
        Process currentProcess = priorityQueues[chosenQueue].front();
        priorityQueues[chosenQueue].pop();
      int executionTime = min(quantum, currentProcess.burst_time);
        current_time += executionTime;
        currentProcess.burst_time = currentProcess.burst_time

                                                  - executionTime;
        // Check if the process has completed or needs to be

      // promoted or demoted
        if (currentProcess.burst_time == 0) {
```


Scanned with OKEN Scanner

```
            completedProcesses++;
            ct[currentProcess.pid - 1] = current_time;
        } else if (currentProcess.burst_time <= quantum/2) {
            // Promote the process if its remaining time is less than half
            // of its quantum time
            currentProcess.priority = max(currentProcess.priority - 1, 1);
            priorityQueues[currentProcess.priority].push(currentProcess);
        } else if (currentProcess.burst_time >= quantum*5){
            // Demote the process to a lower-priority queue
            currentProcess.priority = min(currentProcess.priority + 1, 2);
            priorityQueues[currentProcess.priority].push(currentProcess);
        } else {
            priorityQueues[currentProcess.priority].push(currentProcess);
        }
    }
    // sort as per the input (based on process no/Pid)
    sort(processes.begin(), processes.end(), comparePid);

    for(int i = 0; i < n; i++){
        processes[i].completion_time = ct[i];
        processes[i].turnAround_time = processes[i].completion_time
                                        - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnAround_time
                                        - processes[i].burst_time;
    }

    //to get avg tat and avg wt
    int total_tat = 0; int total_wt = 0;

    // Output the result
    cout << "ProcessID \tArrival_Time \tBurst_Time \tPriority \tWaiting_Time "
        << "\tturnaround_Time\tCompletion_Time\n";

    for (auto& p : processes) {
        total_tat += p.turnAround_time;
        total_wt  += p.waiting_time;
        cout << "P" << p.pid << "\t\t\t" << p.arrival_time << "\t\t"
             << p.burst_time << "\t\t" << p.priority << "\t\t"
             << p.waiting_time << "\t\t" << p.turnAround_time << "\t\t"
             << p.completion_time << "\n";
    }
    cout << "Average turn_around_time = "
        << static_cast<float>(total_tat+0.0)/n << endl;
    cout << "Average waiting_time = "
        << static_cast<float>(total_wt)/n << endl;

}
```


Scanned with OKEN Scanner

```
int main() {

   cout << "This program Multilevel Feedback Queue scheduling algorithm \n"
         << "For simple demonstration here are 3 queues with "

        <<  priority level 0,1,2"
        << "(0 being the highest priority)" << endl;

int n;
   cout << "Enter the number of processes: ";
   cin >> n;

   vector<Process> processes(n);
   for (int i = 0; i < n; ++i) {
       processes[i].pid = i + 1;
       cout << "Arrival Time, Burst Time, Priority(0,1,2) for Process P"

           << i + 1 << ": ";
       cin >> processes[i].arrival_time >> processes[i].burst_time

           >> processes[i].priority;
       if(processes[i].priority<0 || processes[i].priority > 2 ){
            throw invalid_argument("invalid priority value");
       }
   }

   MLFQ_Scheduling(processes);

   return 0;
}
```


Scanned with OKEN Scanner

OUTPUT:

```
This program Multilevel Feedback Queue scheduling algorithm

For simple demonstration here are 3 queues with priority level 0,1,2

(0 being the highest priority)


Enter the number of processes: 5

Arrival Time, Burst Time, Priority(0,1,2) for Process P1: 3 6 2

Arrival Time, Burst Time, Priority(0,1,2) for Process P2: 5 4 1

Arrival Time, Burst Time, Priority(0,1,2) for Process P3: 5 6 0

Arrival Time, Burst Time, Priority(0,1,2) for Process P4: 3 4 0

Arrival Time, Burst Time, Priority(0,1,2) for Process P5: 2 6 2
```

Process_ID	A.T	B.T	Priority	W.T	T.A.T	C.T
P1	3	6	2	19	25	28
P2	5	4	1	13	17	22
P3	5	6	0	5	11	16
P4	3	4	0	3	7	10
P5	2	6	2	10	16	18

```
Average turn_around_time = 15.2

Average waiting_time = 10
```


Scanned with OKEN Scanner

Experiment No. 07

BANKER'S ALGORITHM FOR DEAD LOCK AVOIDANCE

AIM:

To implement the Banker's Algorithm for deadlock avoidance in an operating system.

DESCRIPTION:

Banker's Algorithm is a deadlock avoidance algorithm used in operating systems to manage resources efficiently. It ensures that the system never enters an unsafe state, where deadlock might occur, by checking for safety before allocating resources to processes. Each process in the system must declare the maximum number of instances of each resource type it may need and the total number of resources available in the system must be declared. Based on this information, the Banker's Algorithm determines whether a given resource allocation state is safe or not before granting the resource request to a process.

ALGORITHM:

- **Safety Algorithm**
  1. Initialize work (vector) = available
  2. Finish [i] = false
  3. Find i such that
     Finish [i] == false
     && Need [i] <= Work [j] { for all j : 0 to m }
     If not goto 5
  4. Work = Work + allocation [i]
     Finish [i] = true
     goto 3
  5. If Finish [i] == True { For all i }
     then safe state
     else
     unsafe state
- **Resource Request Algorithm**
  1. Check Request i < Available
  2. Check Request i < Need i (safety check)
  3. Assume we satisfy the request
     Available = Available - Request
     Need [i] = Need [i] - Request
     Alloc [i] += Request
  4. Run Safety Algo
  5. If safe grant request [i] else Block P [i]

PROGRAM:

```
#include <iostream>
#include <vector>
using namespace std;

class BankersAlgorithm {
        private:
                vector<vector<int>> allocated;
                vector<vector<int>> max;
                vector<int> available;
```


Scanned with OKEN Scanner

```
        // Function to calculate the need matrix
        vector<vector<int>> calculateNeed() {
            int n = max.size();
            vector<vector<int>> need(n, vector<int>(max[0].size(), 0));
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < max[0].size(); j++) {
                    need[i][j] = max[i][j] - allocated[i][j];
                }
            }
            return need;
        }
    public:
        // Constructor to initialize the allocated, max, and available
    vectors
        BankersAlgorithm(const vector<vector<int>>& allocated,
                         const vector<vector<int>>& max,
                         const vector<int>& available)
            : allocated(allocated), max(max), available(available) {}

        // Function to check if the system is in a safe state
        bool SafetyAlgorithm(vector<int>& safeSeq) {
            int n = max.size();
            vector<bool> finished(n, false);
            vector<int> work = available;
            vector<vector<int>> need = calculateNeed();

            int count = 0;
            while (count < n) {
                bool found = false;
                for (int i = 0; i < n; i++) {
                    if (!finished[i]) {
                        int j;
                        for (j = 0; j < max[0].size(); j++) {
                            if (need[i][j] > work[j]) {
                                break;
                            }
                        }
                    if (j == max[0].size()) {
                        for (int k = 0; k < max[0].size(); k++) {
                            work[k] += allocated[i][k];
                        }
                        safeSeq.push_back(i);
                        finished[i] = true;
                        found = true;
                        count++;
                    }
                }
            }
            if (!found) {
                return false; // System is in unsafe state
            }
        }
```


Scanned with OKEN Scanner

```
        return true; // System is in safe state
    }

    // Function to handle resource request
    bool ResourceRequest(int processId, const vector<int>& request) {
        vector<vector<int>> need = calculateNeed();
        for (int i = 0; i < request.size(); i++) {
            if (request[i] > need[processId][i] ||
                                        request[i] > available[i])
            {
                return false; // Request cannot be granted
            }
    }

        // Try to allocate resources temporarily
        for (int i = 0; i < request.size(); i++) {
            available[i] -= request[i];
            allocated[processId][i] += request[i];
        }

        // Check if system is still in a safe state
        vector<int> safeSeq;
        if (!SafetyAlgorithm(safeSeq)) {
        // Restore the previous state if it leads to an unsafe state
          for (int i = 0; i < request.size(); i++) {
              available[i] += request[i];
              allocated[processId][i] -= request[i];
          }
          return false; // Request cannot be granted
        }
        return true; // Request granted successfully
    }

};


int main() {
 int n, m; // n processes and m resources types
 cout << "Enter the number of processes: ";
 cin >> n;
 cout << "Enter the number of resources types: ";
 cin >> m;

 // Maximum resources that can be allocated to each process
 vector<vector<int>> max(n, vector<int>(m, 0));
 cout << "\nEnter the maximum resources for each process:\n";
 for (int i = 0; i < n; i++) {
     cout << "\tFor process P" << i << ": ";
     for (int j = 0; j < m; j++) {
         cin >> max[i][j];
     }
```


Scanned with OKEN Scanner

```
    }


    // Resources currently allocated to each process
    vector<vector<int>> allocated(n, vector<int>(m, 0));
    cout << "\nEnter the resources allocated to each process:\n";
    for (int i = 0; i < n; i++) {
        cout << "\tFor process P" << i << ": ";
        for (int j = 0; j < m; j++) {
            cin >> allocated[i][j];
        }
    }


   // Available resources
    vector<int> available(m, 0);
    cout << "\nEnter the available resources: ";
    for (int i = 0; i < m; i++) {
        cin >> available[i];
    }



   // Create BankersAlgorithm object
    BankersAlgorithm ba(allocated, max, available);

        bool canAlloc = true;


   // Handle resource request
    while(canAlloc){
            int processId;
            vector<int> request(m, 0);
            cout << "\nEnter the process ID for resource request: ";
            cin >> processId;
            cout << "Enter the resource request for process P"

                << processId << ": ";
            for (int i = 0; i < m; i++) {
                cin >> request[i];
            }

            if (ba.ResourceRequest(processId, request)) {
                cout << "\n\tRequest granted successfully" << endl;
            } else {
                cout << "\n\tRequest cannot be granted" << endl;
                canAlloc = false;
        }
    }

    return 0;
   }
```


Scanned with OKEN Scanner

OUTPUT:

```
Enter the number of processes: 5
Enter the number of resources types: 4

Enter the maximum resources for each process:
      For process P0: 2 2 1 1
      For process P1: 1 2 1 2
      For process P2: 1 1 2 1
      For process P3: 3 1 2 0
      For process P4: 2 1 1 0

Enter the resources allocated to each process:
      For process P0: 1 1 0 0
      For process P1: 0 1 1 0
      For process P2: 1 0 1 1
      For process P3: 2 1 2 0
      For process P4: 1 0 1 0

Enter the available resources: 5 3 5 3


Enter the process ID for resource request: 1
Enter the resource request for process P1: 0 0 0 1


      Request granted successfully


Enter the process ID for resource request: 3
Enter the resource request for process P3: 1 0 0 2


      Request cannot be granted
```


Scanned with OKEN Scanner

Experiment No. 08

DEAD LOCK DETECTION ALGORITHM

AIM:

The aim of this program is to simulate and implement the Deadlock Detection algorithm. Deadlock detection is a technique used in operating systems to identify and resolve deadlocks, which occur when two or more processes are unable to proceed because each is waiting for the other to release a resource.

DESCRIPTION:

Deadlock detection is an essential mechanism in operating systems to ensure system stability and prevent resource contention issues. This algorithm systematically examines the resource allocation graph to identify deadlocks by looking for cycles. Once a deadlock is detected, appropriate actions can be taken to resolve it, such as aborting processes, releasing resources, or preempting resources from other processes.

ALGORITHM:

1. Read the number of processes, n.
2. Initialize a 2D vector wait_for_graph of size n x n to represent the wait-for-graph.
3. Read the wait-for-graph from the user.
4. Call the detectDeadlock function with the wait_for_graph and n as parameters.
5. Inside the detectDeadlock function:
    a. Calculate the in-degree for each process by counting incoming edges.
    b. Initialize a queue to store processes with an in-degree of 0.
    c. Perform topological sorting using BFS:
        i. Enqueue processes with an in-degree of 0.
        ii. Dequeue processes and update in-degrees of dependent processes.
    d. If all processes are visited, return false (no deadlock), else true (deadlock detected).
6. If deadlock_detected is true, print "Deadlock detected!", else print "No deadlock detected."

PROGRAM:

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>

using namespace std;

// Function to detect deadlocks using the wait-for-graph algorithm
bool detectDeadlock(const vector<vector<int>>& wait_for_graph, int n) {
 // Initialize in-degree array to track incoming edges for each process
 vector<int> in_degree(n, 0);

 // Calculate in-degree for each process
 for (int i = 0; i < n; ++i) {
         for (int j = 0; j < n; ++j) {
             if (wait_for_graph[i][j] == 1) {
            in_degree[j]++;
             }
        }
```


Scanned with OKEN Scanner

```
    }

    // Queue to store processes with in-degree of 0
    queue<int> q;

    // Add processes with in-degree of 0 to the queue
    for (int i = 0; i < n; ++i) {
            if (in_degree[i] == 0) {
                q.push(i);
            }
    }

    // Set to track visited processes
    unordered_set<int> visited;

    // Perform topological sorting using BFS
    while (!q.empty()) {
                int curr_process = q.front();
                q.pop();
                visited.insert(curr_process);
               // Update in-degree of processes waiting for curr_process
               for (int i = 0; i < n; ++i) {
                    if (wait_for_graph[curr_process][i] == 1) {
                   in_degree[i]--;
                   if (in_degree[i] == 0) {
                            q.push(i);
                   }
                    }
               }
    }
    // If all processes are visited, there is no deadlock
    return visited.size() != n;
}

int main() {
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

    // Initialize the wait-for-graph adjacency matrix
    vector<vector<int>> wait_for_graph(n, vector<int>(n, 0));

    cout << "Enter the wait-for-graph "
             << (1 if process i is waiting for process j, 0 otherwise):\n";
    for (int i = 0; i < n; ++i) {
               cout << "For process " << i << ":\n";
               for (int j = 0; j < n; ++j) {
                    cin >> wait_for_graph[i][j];
               }
    }
```


Scanned with OKEN Scanner

```
    // Detect deadlocks
    bool deadlock_detected = detectDeadlock(wait_for_graph, n);


        if (deadlock_detected) {
                cout << "Deadlock detected!" << endl;
        } else {
                cout << "No deadlock detected." << endl;
        }
        return 0;
    }
```

OUTPUT:

```
Enter the number of processes: 3

Enter the wait-for-graph
(1 if process i is waiting for process j, 0 otherwise):

For process 0:
0 1 0
For process 1:
0 0 1
For process 2:
1 0 0

    Deadlock detected!
```


Scanned with OKEN Scanner

Experiment No. 09

FIRST-IN-FIRST-OUT PAGE REPLACEMNET ALGORITHM

AIM:

To implement the First-In-First-Out (FIFO) page replacement algorithm in C++.

DESCRIPTION:

The FIFO page replacement algorithm works on the principle of evicting the oldest page in the memory whenever a page fault occurs. It maintains a queue of pages in memory, and when a new page needs to be brought in, the oldest page present in the memory is replaced. This algorithm is straightforward and easy to implement, making it suitable for systems where simplicity is prioritized over optimization.

ALGORITHM:

1. Initialize an empty queue to store pages.
2. For each page reference in the input sequence:
    - Check if the page is already in memory:
        - If yes, continue to the next page reference.
        - If no, and there is space in memory, add the page to memory and continue.
        - If memory is full, evict the oldest page (the one at the front of the queue) and add the new page to memory.
3. Continue this process until all page references are processed.

PROGRAM:

```
#include <iostream>
#include <queue>
#include <unordered_set>

using namespace std;

void fifoPageReplacement(vector<int>& pages, int capacity) {
 unordered_set<int> memory; //fast lookups and no duplicates
 queue<int> fifoQueue; //queue for fifo tracking

 int pageFaults = 0;
 for (int i = 0; i < pages.size(); ++i) {
      if (memory.size() < capacity) {
           if (memory.find(pages[i]) == memory.end()) {
           memory.insert(pages[i]);
           fifoQueue.push(pages[i]);
           ++pageFaults;
           }
      } else {
           if (memory.find(pages[i]) == memory.end()) {
           int oldestPage = fifoQueue.front();
           fifoQueue.pop();
           memory.erase(oldestPage);
           memory.insert(pages[i]);
           fifoQueue.push(pages[i]);
```


Scanned with OKEN Scanner

```
                ++pageFaults;
                }
            }
        }

    cout << "Total Page Faults: " << pageFaults << endl;
}
int main() {
    int n;
    cout << "Enter total no of page reference : ";
    cin >> n;
    cout << "Enter all indexes : ";
    vector<int> pages(n);
    for (int i = 0; i < n; i++) {
            cin >> pages[i];
    }

    cout << "Enter no. of frames available : ";
    int capacity; cin >> capacity;

    cout << "FIFO Page Replacement Algorithm\n";
    fifoPageReplacement(pages, n, capacity);

    return 0;
}
```

OUTPUT:

Enter total no of page reference : 15

Enter all indexes : 7 3 0 7 2 3 0 6 5 7 2 4 1 4 1

Enter no. of frames available : 4

FIFO Page Replacement Algorithm
Total Page Faults: 9

Scanned with OKEN Scanner

Experiment No. 10

LEAST RECENT USED PAGE REPLACEMNET ALGORITHM

AIM:

The aim of this program is to simulate the Least Recently Used (LRU) page replacement algorithm.

DESCRIPTION:

The LRU algorithm replaces the least recently used page when a page fault occurs. It maintains a record of the order in which pages are accessed and selects the page that has not been accessed for the longest time for replacement. This program demonstrates how the LRU algorithm works in a simplified virtual memory system.

ALGORITHM:

1. Initialize an empty deque frames to store pages in memory
2. Initialize an empty unordered_map pageMap to map page ID to its position in the queue
3. Initialize pageFaults to 0
4. For each pageRequest in the pageRequests vector:
    a. Check if the pageRequest is already in memory by checking pageMap
    b. If the page is in memory:
        i. Update its access time to the current time
        ii. Move the page to the front of the deque (most recently used)
    c. If the page is not in memory:
        i. Increment pageFaults
        ii. If the number of frames in memory is equal to capacity:
            - Evict the least recently used page from the back of the deque
            - Remove the corresponding entry from pageMap
        iii. Create a new Page object for the pageRequest with accessTime set to the current time
        iv. Add the new page to the front of the deque
        v. Add an entry in pageMap mapping the page ID to its position in the deque
5. Print the total number of page faults

PROGRAM:

```
#include <iostream>
#include <deque>
#include <unordered_map>
#include <vector>

using namespace std;

struct Page {
    int id;
    int accessTime; // Time of the last access
};
```


Scanned with OKEN Scanner

```
// Function to simulate LRU page replacement
void LRU(vector<int>& pageRequests, int frameSize) {

    deque<Page> frames;  // Queue to store pages in memory

    // Map page ID to its position in the queue
    unordered_map<int, deque<Page>::iterator> pageMap;
    int pageFaults = 0;

    for (int pageRequest : pageRequests) {

        // Check if page is already in memory
        if (pageMap.count(pageRequest)) {
            // Update access time and move page to front of queue MRU
            auto it = pageMap[pageRequest];
            it->accessTime = frames.size();

                // Current time is the number of elements in the queue
            frames.erase(it);
            frames.push_front(*it);

        } else {
            // Page fault: load page into memory
            pageFaults++;
            if (frames.size() == frameSize) {
                // Evict least recently used page (back of the queue)
                pageMap.erase(frames.back().id);

                // Remove the evicted page from the pageMap
                frames.pop_back();
            }

            Page newPage;
            newPage.id = pageRequest;
            newPage.accessTime = frames.size();
            frames.push_front(newPage);
            pageMap[pageRequest] = frames.begin();

            // Update pageMap with the new page
        }
    }
    cout << "Page Faults: " << pageFaults << endl;
}
```


Scanned with OKEN Scanner

```
int main() {

    int n;
    cout << "Enter total number of page references: ";
    cin >> n;
    cout << "Enter all indexes: ";

    vector<int> pages(n);

    for (int i = 0; i < n; i++) {
    cin >> pages[i];
    }
    cout << "Enter number of frames available: ";
    int capacity;
    cin >> capacity;
    cout << "Least Recently Used (LRU) Page Replacement Algorithm\n";
    LRU(pages, capacity);

    return 0;
}
```

OUTPUT:

Enter total number of page references: 15
Enter all indexes: 7 3 0 7 2 3 0 6 5 7 2 4 1 4 1
Enter number of frames available: 4

Least Recently Used (LRU) Page Replacement Algorithm
Page Faults: 10

Scanned with OKEN Scanner

Experiment No. 11

LEAST FREQUENTLY USED PAGE REPLACEMNET ALGORITHM

AIM:

The aim of this program is to simulate the Least Frequently Used (LFU) page replacement algorithm. LFU aims to minimize the number of page faults by evicting the page that has been least frequently accessed.

DESCRIPTION:

The LFU page replacement algorithm works by keeping track of the frequency of each page's access. When a page needs to be replaced and the available frames are full, LFU selects the page with the lowest frequency count for eviction.

ALGORITHM:

1. Initialize a data structure to store pages along with their frequencies.
2. For each page request:
    - If the page is present in memory, update its frequency count.
    - If the page is not present in memory:
        - If the available frames are not full, load the page into memory.
        - If the available frames are full, evict the page with the lowest frequency count and load the new page.
3. Repeat step 2 for all page requests.
4. Count the total number of page faults.

PROGRAM:

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <algorithm>

using namespace std;

// Page structure to hold page id and its frequency
struct Page {
     int id;
     int frequency;
};

// Comparator for sorting pages by frequency in ascending order
bool compareFrequency(const Page& a, const Page& b) {
     return a.frequency < b.frequency;
}

// Function to simulate LFU page replacement
void LFU(vector<int>& pageRequests, int frameSize) {
     unordered_map<int, int> frequencyMap; // Map to store page frequency
     unordered_map<int, bool> presentMap;
```


Scanned with OKEN Scanner

```
// Map to check if page is present in frame
    vector<Page> frames; // Vector to store frames
    int pageFaults = 0;
    for (int pageRequest : pageRequests) {
        if (presentMap.count(pageRequest)) {
            // Increment frequency if page is already present
            frequencyMap[pageRequest]++;
        } else {
            // Page fault: load page into memory
            pageFaults++;
            if (frames.size() < frameSize) {
                // If frames are not full, add page to frames
                presentMap[pageRequest] = true;
                frequencyMap[pageRequest] = 1;
                frames.push_back({pageRequest, 1});
            } else {
                // Evict page with lowest frequency
                auto minFreqPage = min_element(frames.begin(),
                                               frames.end(),
                                               compareFrequency);
                presentMap.erase(minFreqPage->id);
                frequencyMap.erase(minFreqPage->id);
                presentMap[pageRequest] = true;
                frequencyMap[pageRequest] = 1;
                *minFreqPage = {pageRequest, 1};
            }
        }
    }
    cout << "Page Faults: " << pageFaults << endl;
}
int main() {
    int n;
    cout << "Enter total number of page references: ";
    cin >> n;
    cout << "Enter all indexes: ";

    vector<int> pages(n);
    for (int i = 0; i < n; i++) {
    cin >> pages[i];
    }

    cout << "Enter number of frames available: ";
    int capacity;
    cin >> capacity;
    cout << "Least Frequently Used (LFU) Page Replacement Algorithm\n";
    LFU(pages, capacity);

    return 0;
}
```


Scanned with OKEN Scanner

OUTPUT:

Enter total number of page references: 15
Enter all indexes: 7 3 0 7 2 3 0 6 5 7 2 4 1 4 1
Enter number of frames available: 4

Least Recently Used (LRU) Page Replacement Algorithm
Page Faults: 10

Scanned with OKEN Scanner

Experiment No. 12

CONTIGUOUS ALLOCATION ALGORITHM

AIM:

To implement the Contiguous Allocation algorithm in C++, which is used for memory allocation in operating systems.

DESCRIPTION:

Contiguous allocation is a memory allocation technique where each process is allocated contiguous memory blocks. In this simulation, we'll implement a simple version of contiguous allocation using a memory map to keep track of allocated and free memory blocks.

ALGORITHM:

1. Initialize a directory to store file metadata (start block and length) using an unordered_map.
2. Initialize a vector to represent disk blocks, where each index represents a block and each element represents whether the block is allocated or free.
3. Implement a constructor for the ContiguousAllocation class that takes the total disk size as input and initializes the diskBlocks vector accordingly.
4. Implement a method createFile() to create a new file. This method takes the file name and size as input parameters.
    - Search for contiguous free blocks in the disk using findContiguousBlocks() method.
    - If contiguous blocks are found, update the directory with the file metadata and mark the blocks as allocated using markBlocksAsAllocated() method.
    - If contiguous space is not available, display an error message indicating insufficient space.
5. Implement a method deleteFile() to delete an existing file. This method takes the file name as an input parameter.
    - Check if the file exists in the directory.
    - If found, release the allocated blocks using releaseBlocks() method and remove the file entry from the directory.
    - If not found, display an error message indicating file not found.
6. Implement a method displayDirectory() to display the contents of the directory, showing the file names, start blocks, and lengths.
7. Implement helper methods:
    - findContiguousBlocks(): Search for contiguous free blocks of the given size in the diskBlocks vector.
    - markBlocksAsAllocated(): Mark the contiguous blocks as allocated.
    - releaseBlocks(): Release the allocated blocks.

PROGRAM:

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;
struct File {
 int startBlock;
 int length;
};

class ContiguousAllocation {
```


Scanned with OKEN Scanner

```
private:
 unordered_map<string, File> directory;
 vector<bool> diskBlocks;
public:
 ContiguousAllocation(int diskSize) : diskBlocks(diskSize, false) {}
 void createFile(string fileName, int fileSize) {
            int start = findContiguousBlocks(fileSize);
            if (start != -1) {
                directory[fileName] = {start, fileSize};
                 markBlocksAsAllocated(start, fileSize);
                 cout << "File '" << fileName << "' created successfully."
                      << endl;
            } else {
                 cout << "Insufficient contiguous space to create file '"
                      << fileName << "'." << endl;
            }
 }
 void deleteFile(string fileName) {
            if (directory.find(fileName) != directory.end()) {
                File file = directory[fileName];
                directory.erase(fileName);
                releaseBlocks(file.startBlock, file.length);
                cout << "File '" << fileName << "' deleted successfully."
                     << endl;
            } else {
                cout << "File '" << fileName << "' not found." << endl;
        }
 }
 void displayDirectory() {
            cout << "Directory Listing:" << endl;
            for (auto& entry : directory) {
                 cout << "File: " << entry.first << ", Start Block: "
                      << entry.second.startBlock << ", Length: "
                      << entry.second.length << endl;
            }
 }
private:
 int findContiguousBlocks(int fileSize) {
           int consecutiveFreeBlocks = 0;
           for (int i = 0; i < diskBlocks.size(); ++i) {
                if (diskBlocks[i] == false) {
               consecutiveFreeBlocks++;
               if (consecutiveFreeBlocks == fileSize) {
                        return i - fileSize + 1; // Starting block index
               }
                } else {
               consecutiveFreeBlocks = 0;
                }
           }
```


Scanned with OKEN Scanner

```
            return -1; // Insufficient contiguous space
    }
    void markBlocksAsAllocated(int startBlock, int length) {
            for (int i = startBlock; i < startBlock + length; ++i) {
                diskBlocks[i] = true;
            }
    }
    void releaseBlocks(int startBlock, int length) {
            for (int i = startBlock; i < startBlock + length; ++i) {
                 diskBlocks[i] = false;
        }
    }

  };
  int main() {
        int diskSize, option;
        cout << "Enter the size of the disk: ";
        cin >> diskSize;
        ContiguousAllocation fs(diskSize);
        do {
          cout << "\nContiguous Allocation Menu:\n";
          cout << "1. Create File\t\t 2. Delete File\n”;
          cout << "3. Display Directory\t 4. Exit\n";
          cout << "Enter your choice: ";
          cin >> option;
          switch (option) {
                case 1: {
                string fileName; int fileSize;
                cout << "Enter file name & size :  ";
                cin >> fileName >> fileSize;
                cout << "Enter file size: ";
                fs.createFile(fileName, fileSize);
                break;
                } case 2: {
                string fileName;
                cout << "Enter file name to delete: ";
                cin >> fileName;
                fs.deleteFile(fileName);
                break;
                } case 3:
                fs.displayDirectory();
                break;
                case 4:
                cout << "Exiting...";
                break;
                default:
                cout << "Invalid option! Please try again.";
           }
          } while (option != 4);

  cout << "\n Exited";
  return 0; }
```


Scanned with OKEN Scanner

OUTPUT:

```
Enter the size of the disk: 32

Contiguous Allocation Menu:
1. Create File          2. Delete File
3. Display Directory    4. Exit
Enter your choice: 1
Enter file name & size :  BootFile 2
Enter file size: File 'BootFile' created successfully.

Contiguous Allocation Menu:
1. Create File          2. Delete File
3. Display Directory    4. Exit
Enter your choice: 1
Enter file name & size :  StudyMaterial-HD_videos 13
Enter file size: File 'StudyMaterial-HD_videos' created successfully.

Contiguous Allocation Menu:
1. Create File          2. Delete File
3. Display Directory    4. Exit
Enter your choice: 1
Enter file name & size :  GTA_V 32
Enter file size: Insufficient contiguous space to create file 'GTA_V'.

Contiguous Allocation Menu:
1. Create File          2. Delete File
3. Display Directory    4. Exit
Enter your choice: 3
Directory Listing:
File: StudyMaterial-HD_videos, Start Block: 2, Length: 13
File: BootFile, Start Block: 0, Length: 2

Contiguous Allocation Menu:
1. Create File          2. Delete File
3. Display Directory    4. Exit
Enter your choice: 4
Exited!
```


Scanned with OKEN Scanner

Experiment No. 13

LINKED ALLOCATION ALGORITHM

AIM:

To write a C++ program to simulate the linked allocation algorithm, which is used for file allocation in operating systems.

DESCRIPTION:

In the linked allocation strategy, each file is stored as a linked list of disk blocks. Each block contains a pointer to the next block in the file. This program will simulate how files are allocated and managed using linked allocation.

ALGORITHM:

1. Initialize a data structure to represent the disk blocks. Each block should contain data and a pointer to the next block.
2. Define a data structure to represent a file. Include attributes such as file name, starting block index, and file size.
3. Implement a class to represent the disk, which includes methods for file allocation, deletion, and displaying disk status.
4. In the file allocation method:
    a. Find contiguous free blocks to allocate the file.
    b. Update the file information with the starting block index and size.
    c. Allocate file blocks by creating a linked list of disk blocks.
5. In the file deletion method:
    a. Free the blocks allocated to the file by resetting their pointers.
    b. Remove the file entry from the file table.
6. Display the current disk status, including information about allocated files and free blocks.

PROGRAM:

```
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

struct DiskBlock {
 int data;
 int nextBlockIndex;
};

struct File {
 string fileName;
 int startBlockIndex;
 int fileSize;
};

class Disk {
private:
 vector<DiskBlock> blocks;
 unordered_map<string, File> files;
```


Scanned with OKEN Scanner

```
public:
 Disk(int size) : blocks(size) {
          // Initialize disk blocks
          for (int i = 0; i < size; ++i) {
               blocks[i].nextBlockIndex = -1;

               // -1 indicates end of the linked list
          }
 }

 void allocateFile(const string& fileName, int fileSize) {
          // Find contiguous free blocks to allocate the file
          int startIndex = findContiguousFreeBlocks(fileSize);
          if (startIndex == -1) {
               cout << "Error: Not enough contiguous free space"

                    << "to allocate file." << endl;
               return;
          }


          // Allocate file blocks and update file information
          files[fileName] = {fileName, startIndex, fileSize};
          int currentIndex = startIndex;
          for (int i = 1; i < fileSize; ++i) {
           blocks[currentIndex].nextBlockIndex = startIndex + i;
           currentIndex = startIndex + i;
           }
     }

     void deleteFile(const string& fileName) {
           if (files.find(fileName) == files.end()) {
                cout << "Error: File '" << fileName << "' not found." << endl;
                return;
            }

           // Free blocks allocated to the file
           int currentIndex = files[fileName].startBlockIndex;


           while (currentIndex != -1) {
                int nextIndex = blocks[currentIndex].nextBlockIndex;
                blocks[currentIndex].nextBlockIndex = -1;
                currentIndex = nextIndex;
           }

            // Remove file from the file table
            files.erase(fileName);
 }
```


Scanned with OKEN Scanner

```
void displayDiskStatus() {
        cout << "Disk Status:" << endl;
        for (const auto& file : files) {
            cout << "File: " << file.second.fileName << ", Size: "
                << file.second.fileSize << endl;
        }
        cout << "Free Blocks:";
        for (int i = 0; i < blocks.size(); ++i) {
        if (blocks[i].nextBlockIndex == -1) {
            cout << " " << i;
            }
        }
        cout << endl;
}

private:
 int findContiguousFreeBlocks(int fileSize) {

        int count = 0;
        int startIndex = -1;

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

            if (blocks[i].nextBlockIndex == -1) {
            if (startIndex == -1) {
                startIndex = i;
            }
            count++;
            } else {
            count = 0;
            startIndex = -1;
            }

            if (count == fileSize) {
            return startIndex;
            }
        }
        return -1;
 }
};
```


Scanned with OKEN Scanner

```
int main() {
 int diskSize;
 cout << "Enter disk size: ";
 cin >> diskSize;

 Disk disk(diskSize);

 int choice;


 do {
          cout << "\n1. Allocate File\n2. Delete File\"
             << n3. Display Disk Status\n4. Exit\nEnter choice: ";


          cin >> choice;
          switch (choice) {
               case 1: {
                string fileName;
                int fileSize;
                cout << "Enter file name and size: ";
                cin >> fileName >> fileSize;
                disk.allocateFile(fileName, fileSize);
                break;
               } case 2: {
                string fileName;
                cout << "Enter file name to delete: ";
                cin >> fileName;
                disk.deleteFile(fileName);
                break;
               } case 3:
                disk.displayDiskStatus();
                break;
               case 4:
                cout << "Exiting...";
                break;
               default:
                cout << "Invalid choice!";
          }


   } while (choice != 4);

   return 0;
}
```


Scanned with OKEN Scanner

OUTPUT:

```
Enter disk size: 10

1. Allocate File        2. Delete File
3. Display Disk Status   4. Exit
Enter choice: 1
Enter file name and size: file1 3

1. Allocate File        2. Delete File
3. Display Disk Status   4. Exit
Enter choice: 1
Enter file name and size: file2 2

1. Allocate File        2. Delete File
3. Display Disk Status   4. Exit
Enter choice: 3
Disk Status:
File: file1, Size: 3
File: file2, Size: 2
Free Blocks: 5 6 7 8 9

1. Allocate File        2. Delete File
3. Display Disk Status   4. Exit
Enter choice: 2
Enter file name to delete: file1

1. Allocate File        2. Delete File
3. Display Disk Status   4. Exit
Enter choice: 3
Disk Status:
File: file2, Size: 2
Free Blocks: 0 1 2 3 4 5 6 7 8 9

1. Allocate File        2. Delete File
3. Display Disk Status   4. Exit
Enter choice: 4
Exiting...
```


Scanned with OKEN Scanner

Experiment No. 14

INDEXING ALLOCATION ALGORITHM

AIM:

To implement the Indexed Allocation strategy for file management in a simulated environment.

DESCRIPTION:

Indexed Allocation is a file allocation strategy where each file has its own index block. This index block contains pointers to the actual blocks of data that make up the file. This approach is useful for managing files efficiently, especially when dealing with large files or when files are frequently accessed randomly.

ALGORITHM:

1. Initialize Disk Blocks:
   - Allocate memory structures to represent disk blocks. These blocks will store the actual data of the files.
2. Initialize Index Blocks:
   - Create index blocks to maintain pointers for each file. Each index block will contain pointers to the data blocks storing the file's content.
3. For Each File:
   a. Allocate Index Block: Assign an index block to the file, which will serve as an entry point to access its data blocks.
   b. Allocate Data Blocks: Reserve data blocks to store the file's content. The number of data blocks allocated depends on the file size.
   c. Update Index Block: Update the index block to include pointers to the allocated data blocks, establishing the link between the file and its data.
4. File Operations:
   a. Create File: Allocate necessary blocks and update index blocks to represent the new file.
   b. Read File: Retrieve data blocks associated with the file's index block to access its content.
   c. Write to File: Modify data blocks to reflect changes made to the file's content.
   d. Delete File: Free allocated blocks and remove references to the file from index blocks.
5. Error Handling: Handle scenarios such as insufficient disk space, file not found, or invalid operations gracefully to ensure robustness.

PROGRAM:

```
#include <iostream>
#include <vector>
#include <unordered_map>

using namespace std;

// Structure to represent a file block
struct Block {
 int blockId;
 string data;
};

// Structure to represent a file
struct File {
 int fileId;
```


Scanned with OKEN Scanner

```
 vector<int> indexBlock;
        // Index block containing pointers to data blocks
};

// Function to create a new file
void createFile(vector<File>& files, unordered_map<int,
                vector<Block>>& disk, int fileId, const vector<int>& blocks){
 File newFile;
 newFile.fileId = fileId;
 newFile.indexBlock = blocks;
 files.push_back(newFile);
 cout << "File created with ID: " << fileId << endl;
}

// Function to read a file
void readFile(const vector<File>& files, const unordered_map<int,
                                        vector<Block>>& disk, int fileId)
{
 for (const File& file : files) {
       if (file.fileId == fileId) {
             cout << "File data:" << endl;
             for (int blockId : file.indexBlock) {
             const vector<Block>& dataBlocks = disk.at(blockId);
             for (const Block& block : dataBlocks) {
                   cout << "Block ID: " << block.blockId << ", Data: "
                         << block.data << endl;
             }
             }
             return;
       }
 }
 cout << "File not found!" << endl;
}
// Function to delete a file
void deleteFile(vector<File>& files, unordered_map<int,
                                        vector<Block>>& disk, int fileId) {
 auto it = find_if(files.begin(),
                         files.end(),
                         [fileId](const File& file)
                                        { return file.fileId == fileId; }
                         );
 if (it != files.end()) {
          files.erase(it);
          cout << "File deleted successfully." << endl;
 } else {
          cout << "File not found!" << endl;
 }
```


Scanned with OKEN Scanner

```
}
int main() {
 vector<File> files;  // Vector to store file information
 unordered_map<int, vector<Block>> disk;  // Disk with blocks of data
 int choice;
 do {
      cout << "Options:" << endl;
      cout << "1. Create File" << endl;
      cout << "2. Read File" << endl;
      cout << "3. Delete File" << endl;
      cout << "4. Exit" << endl;
      cout << "Enter your choice: ";
      cin >> choice;
      switch (choice) {
           case 1: {
           int fileId;
           cout << "Enter file ID: ";
           cin >> fileId;
           int numBlocks;
           cout << "Enter number of index blocks: ";
           cin >> numBlocks;

           vector<int> indexBlocks(numBlocks);
           cout << "Enter index block numbers: ";
           for (int i = 0; i < numBlocks; ++i) {
                cin >> indexBlocks[i];
           }
           createFile(files, disk, fileId, indexBlocks);
           break;
           } case 2: {
           int fileId;
           cout << "Enter file ID to read: ";
           cin >> fileId;
           readFile(files, disk, fileId);
           break;
           } case 3: {
           int fileId;
           cout << "Enter file ID to delete: ";
           cin >> fileId;
           deleteFile(files, disk, fileId);
           break;
           } case 4:
           cout << "Exiting program." << endl;
           break;
           default:
           cout << "Invalid choice. Please try again." << endl;
      }
 } while (choice != 4);
 return 0;

   }
```


Scanned with OKEN Scanner

OUTPUT:

```
Options:
1. Create File
2. Read File
3. Delete File
4. Exit
Enter your choice: 1
Enter file ID: 1
Enter number of index blocks: 2
Enter index block numbers: 10 20
File created with ID: 1

Options:
1. Create File
2. Read File
3. Delete File
4. Exit
Enter your choice: 2
Enter file ID to read: 1
File data:
Block ID: 10, Data: Block 10 data
Block ID: 20, Data: Block 20 data

Options:
1. Create File
2. Read File
3. Delete File
4. Exit
Enter your choice: 3
Enter file ID to delete: 1
File deleted successfully.

Options:
1. Create File
2. Read File
3. Delete File
4. Exit
Enter your choice: 4
Exiting program.
```


Scanned with OKEN Scanner