FCFS

#include<iostream>

#include<iomanip>

#include<algorithm>

using namespace std;

struct processes

{

    int at,ct,pid,tat,bt,wt,rt,start\_time;

}ps[100];

bool compare(struct processes a,struct processes b)

{

    return a.at<b.bt;

}

int main()

{

    int n;

    cin>>n;

    for(int i=0;i<n;i++)

    {

        cout<<"Enter process id: "<<endl;

        cin>>ps[i].pid;

    }

     for(int i=0;i<n;i++)

    {

         cout<<"Enter Arrival Time: "<<endl;

        cin>>ps[i].at;

    }

     for(int i=0;i<n;i++)

    {

         cout<<"Enter Burst Time: "<<endl;

        cin>>ps[i].bt;

    }

   sort(ps,ps+n,compare);

   int total\_tat=0,total\_wt=0;

   for(int i=0;i<n;i++)

   {

       if(i==0){

           ps[0].start\_time=ps[0].at;

       }

       else{

           ps[i].start\_time=max(ps[i].at,ps[i-1].ct);

       }

       ps[i].ct=ps[i].start\_time+ps[i].bt;

       ps[i].tat=ps[i].ct-ps[i].at;

       ps[i].wt=ps[i].tat-ps[i].bt;

       total\_tat+=ps[i].tat;

       total\_wt+=ps[i].wt;

   }

   cout<<"Process id"<<"/t/t"<<"Burst Time"<<"/t/t"<<"Completion Time"<<"/t/t"<<"TAT"<<"/t/t"<<"Waiting Time"<<endl;

   for(int i=0;i<n;i++)

   {

       cout<<ps[i].pid<<"/t/t"<<ps[i].bt<<"/t/t"<<ps[i].ct<<"/t/t"<<ps[i].tat<<"/t/t"<<ps[i].wt<<endl;

   }

   double avgtat=(double)total\_tat/(double)n;

    double avgwt=(double)total\_wt/(double)n;

    cout<<"Average Turn Around Time: "<<avgtat<<endl;

     cout<<"Average Waiting Time: "<<avgwt<<endl;

  return 0;

}

SJF

#include<bits/stdc++.h>

using namespace std;

struct process

{

    int at,wt,bt,start\_time,ct,pid,tat;

}ps[100];

int main()

{

    int n;

    cin>>n;

     for(int i=0;i<n;i++)

    {

        cout<<"Enter process id: "<<endl;

        cin>>ps[i].pid;

    }

     for(int i=0;i<n;i++)

    {

         cout<<"Enter Arrival Time: "<<endl;

        cin>>ps[i].at;

    }

     for(int i=0;i<n;i++)

    {

         cout<<"Enter Burst Time: "<<endl;

        cin>>ps[i].bt;

    }

    int current\_time=0,completed=0;

    bool is\_completed[n]={false};

    int total\_tat=0,total\_wt=0;

    while(completed!=n)

    {

        int min\_ind=-1;

        int mn=INT\_MAX;

        for(int i=0;i<n;i++)

        {

            if(ps[i].at<=current\_time && is\_completed[i]==false){

                 if(ps[i].bt<mn)

            {

                mn=ps[i].bt;

                min\_ind=i;

            }

            else if(ps[i].bt==mn)

            {

                if(ps[i].at<ps[min\_ind].at){

                    mn=ps[i].bt;

                    min\_ind=i;

                }

            }

            }

        }

        if(min\_ind==-1)

        {

            current\_time++;

        }

        ps[min\_ind].start\_time=current\_time;

        ps[min\_ind].ct=ps[min\_ind].start\_time+ps[min\_ind].bt;

        ps[min\_ind].tat=ps[min\_ind].ct-ps[min\_ind].at;

        ps[min\_ind].wt =ps[min\_ind].tat-ps[min\_ind].bt;

        current\_time+=ps[min\_ind].ct;

        total\_tat+=ps[min\_ind].tat;

        total\_wt+=ps[min\_ind].wt;

        is\_completed[min\_ind]=true;

        completed++;

    }

    cout<<"Process id"<<"\t"<<"Arrival Time"<<"\t"<<"Burst Time"<<"\t"<<"Completion Time"<<"\t"<<"TAT"<<"\t"<<"Waiting Time"<<endl;

   for(int i=0;i<n;i++)

   {

       cout<<ps[i].pid<<"\t\t"<<ps[i].at<<"\t\t"<<ps[i].bt<<"\t\t"<<ps[i].ct<<"\t\t"<<ps[i].tat<<"\t\t"<<ps[i].wt<<endl;

   }

   double avgtat=(double)total\_tat/(double)n;

    double avgwt=(double)total\_wt/(double)n;

    cout<<"Average Turn Around Time: "<<avgtat<<endl;

     cout<<"Average Waiting Time: "<<avgwt<<endl;

  return 0;

}

LJF

#include <iostream>

#include <algorithm>

using namespace std;

struct process

{

    int pid, at, tat, bt, wt, start\_time, ct;

} ps[100];

int main()

{

    int n;

    cout << "Enter number of processes: ";

    cin >> n;

    for (int i = 0; i < n; i++)

    {

        cout << "Enter process id: " << endl;

        cin >> ps[i].pid;

    }

    for (int i = 0; i < n; i++)

    {

        cout << "Enter Arrival Time: " << endl;

        cin >> ps[i].at;

    }

    for (int i = 0; i < n; i++)

    {

        cout << "Enter Burst Time: " << endl;

        cin >> ps[i].bt;

    }

    int total\_tat=0,total\_wt=0;

    int current\_time=0,completed=0;

    bool is\_completed[n]={false};

    while(completed!=n)

    {

        int ind=-1;

        int mx=0;

        for(int i=0;i<n;i++)

        {

            if(ps[i].at<=current\_time && is\_completed[i]==false)

            {

                if(ps[i].bt>mx)

                {

                    mx=ps[i].bt;

                    ind=i;

                }

                else if(ps[i].bt==mx)

                {

                    if(ps[i].at<ps[ind].at)

                    {

                        mx=ps[i].bt;

                        ind=i;

                    }

                }

            }

        }

        if(ind==-1) current\_time++;

        else{

            ps[ind].start\_time=current\_time;

            ps[ind].ct = ps[ind].start\_time+ps[ind].bt;

            ps[ind].tat = ps[ind].ct-ps[ind].at;

            ps[ind].wt=ps[ind].tat-ps[ind].bt;

            is\_completed[ind]=true;

            completed++;

            current\_time+=ps[ind].bt;

            total\_tat+=ps[ind].tat;

            total\_wt+=ps[ind].wt;

        }

    }

   cout<<"Process id"<<"\t"<<"Arrival Time"<<"\t"<<"Burst Time"<<"\t"<<"Completion Time"<<"\t"<<"TAT"<<"\t"<<"Waiting Time"<<endl;

   for(int i=0;i<n;i++)

   {

       cout<<ps[i].pid<<"\t\t"<<ps[i].at<<"\t\t"<<ps[i].bt<<"\t\t"<<ps[i].ct<<"\t\t"<<ps[i].tat<<"\t\t"<<ps[i].wt<<endl;

   }

   double avgtat=(double)total\_tat/(double)n;

    double avgwt=(double)total\_wt/(double)n;

    cout<<"Average Turn Around Time: "<<avgtat<<endl;

     cout<<"Average Waiting Time: "<<avgwt<<endl;

  return 0;

}

PRIORITY(NON Premtive)

#include <iostream>

#include <algorithm>

using namespace std;

struct process

{

    int pid, at, tat, bt, wt, start\_time, ct;

} ps[100];

int main()

{

    int n;

    cout << "Enter number of processes: ";

    cin >> n;

    for (int i = 0; i < n; i++)

    {

        cout << "Enter process id: " << endl;

        cin >> ps[i].pid;

    }

    for (int i = 0; i < n; i++)

    {

        cout << "Enter Arrival Time: " << endl;

        cin >> ps[i].at;

    }

    for (int i = 0; i < n; i++)

    {

        cout << "Enter Burst Time: " << endl;

        cin >> ps[i].bt;

    }

    int total\_tat=0,total\_wt=0;

    int current\_time=0,completed=0;

    bool is\_completed[n]={false};

    while(completed!=n)

    {

        int ind=-1;

        int mx=0;

        for(int i=0;i<n;i++)

        {

            if(ps[i].at<=current\_time && is\_completed[i]==false)

            {

                if(ps[i].bt>mx)

                {

                    mx=ps[i].bt;

                    ind=i;

                }

                else if(ps[i].bt==mx)

                {

                    if(ps[i].at<ps[ind].at)

                    {

                        mx=ps[i].bt;

                        ind=i;

                    }

                }

            }

        }

        if(ind==-1) current\_time++;

        else{

            ps[ind].start\_time=current\_time;

            ps[ind].ct = ps[ind].start\_time+ps[ind].bt;

            ps[ind].tat = ps[ind].ct-ps[ind].at;

            ps[ind].wt=ps[ind].tat-ps[ind].bt;

            is\_completed[ind]=true;

            completed++;

            current\_time+=ps[ind].bt;

            total\_tat+=ps[ind].tat;

            total\_wt+=ps[ind].wt;

        }

    }

   cout<<"Process id"<<"\t"<<"Arrival Time"<<"\t"<<"Burst Time"<<"\t"<<"Completion Time"<<"\t"<<"TAT"<<"\t"<<"Waiting Time"<<endl;

   for(int i=0;i<n;i++)

   {

       cout<<ps[i].pid<<"\t\t"<<ps[i].at<<"\t\t"<<ps[i].bt<<"\t\t"<<ps[i].ct<<"\t\t"<<ps[i].tat<<"\t\t"<<ps[i].wt<<endl;

   }

   double avgtat=(double)total\_tat/(double)n;

    double avgwt=(double)total\_wt/(double)n;

    cout<<"Average Turn Around Time: "<<avgtat<<endl;

     cout<<"Average Waiting Time: "<<avgwt<<endl;

  return 0;

}

FIFO

#include<bits/stdc++.h>

using namespace std;

int main()

{

    int n;

    cout<<"Enter number of pages: ";

    cin>>n;

    int str[n];

    cout<<"Enter your reference string: "<<endl;

    for(int i=0;i<n;i++)

    {

        cin>>str[i];

    }

    cout<<"Enter number of frames: ";

    int frames;

    cin>>frames;

    int hits=0,faults=0,ind=0;

    vector<int> s(frames);

    for(int i=0;i<n;i++)

    {

        if(s.size()<frames)

        {

            s.push\_back(str[i]);

            faults++;

        }

        else{

            if(find(s.begin(),s.end(),str[i])!=s.end())

            {

                hits++;

            }

            else{

                s[ind]=str[i];

                ind++;

                ind=ind%frames;

                faults++;

            }

        }

    }

  cout<<"Number of Faults are: "<<faults<<endl;

  cout<<"Number of Hits are: "<<hits<<endl;

  return 0;

}

ROUND ROBIN

// C++ program to implement Round Robin CPU Scheduling Algorithm

// Video Explanation: https://www.youtube.com/watch?v=8uGblP-ag3o&list=PLVKIC9j3jSYtCEAffM\_51qVWeQus70cqc&index=10

#include <iostream>

#include <algorithm>

#include <queue>

#include <iomanip>

#include <climits>

using namespace std;

struct process\_struct

{

    int pid;

    int at;

    int bt;

    int ct, wt, tat, rt, start\_time;

    int bt\_remaining;

} ps[100];

bool comparatorAT(struct process\_struct a, struct process\_struct b)

{

    int x = a.at;

    int y = b.at;

    return x < y;

    //    if(x > y)

    //      return false;  // Apply sorting

    //    return true;   // no sorting

}

bool comparatorPID(struct process\_struct a, struct process\_struct b)

{

    int x = a.pid;

    int y = b.pid;

    return x < y;

}

int main()

{

    int n, index;

    int cpu\_utilization;

    queue<int> q;

    bool visited[100] = {false}, is\_first\_process = true;

    int current\_time = 0, max\_completion\_time;

    int completed = 0, tq, total\_idle\_time = 0, length\_cycle;

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

    cin >> n;

    float sum\_tat = 0, sum\_wt = 0, sum\_rt = 0;

    cout << fixed << setprecision(2);

    for (int i = 0; i < n; i++)

    {

        cout << "\nEnter Process " << i << " Arrival Time: ";

        cin >> ps[i].at;

        ps[i].pid = i;

    }

    for (int i = 0; i < n; i++)

    {

        cout << "\nEnter Process " << i << " Burst Time: ";

        cin >> ps[i].bt;

        ps[i].bt\_remaining = ps[i].bt;

    }

    cout << "\nEnter time quanta: ";

    cin >> tq;

    // sort structure on the basis of Arrival time in increasing order

    sort(ps, ps + n, comparatorAT);

    q.push(0);

    visited[0] = true;

    while (completed != n)

    {

        index = q.front();

        q.pop();

        if (ps[index].bt\_remaining == ps[index].bt)

        {

            ps[index].start\_time = max(current\_time, ps[index].at);

            total\_idle\_time += (is\_first\_process == true) ? 0 : ps[index].start\_time - current\_time;

            current\_time = ps[index].start\_time;

            is\_first\_process = false;

        }

        if (ps[index].bt\_remaining - tq > 0)

        {

            ps[index].bt\_remaining -= tq;

            current\_time += tq;

        }

        else

        {

            current\_time += ps[index].bt\_remaining;

            ps[index].bt\_remaining = 0;

            completed++;

            ps[index].ct = current\_time;

            ps[index].tat = ps[index].ct - ps[index].at;

            ps[index].wt = ps[index].tat - ps[index].bt;

            ps[index].rt = ps[index].start\_time - ps[index].at;

            sum\_tat += ps[index].tat;

            sum\_wt += ps[index].wt;

            sum\_rt += ps[index].rt;

        }

        // check which new Processes needs to be pushed to Ready Queue from Input list

        for (int i = 1; i < n; i++)

        {

            if (ps[i].bt\_remaining > 0 && ps[i].at <= current\_time && visited[i] == false)

            {

                q.push(i);

                visited[i] = true;

            }

        }

        // check if Process on CPU needs to be pushed to Ready Queue

        if (ps[index].bt\_remaining > 0)

            q.push(index);

        // if queue is empty, just add one process from list, whose remaining burst time > 0

        if (q.empty())

        {

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

            {

                if (ps[i].bt\_remaining > 0)

                {

                    q.push(i);

                    visited[i] = true;

                    break;

                }

            }

        }

    } // end of while

    // Calculate Length of Process completion cycle

    max\_completion\_time = INT\_MIN;

    for (int i = 0; i < n; i++)

        max\_completion\_time = max(max\_completion\_time, ps[i].ct);

    length\_cycle = max\_completion\_time - ps[0].at; // ps[0].start\_time;

    cpu\_utilization = (float)(length\_cycle - total\_idle\_time) / length\_cycle;

    // sort process ID so that in output comes in Original order (just for interactivity- Not needed otherwise)

    sort(ps, ps + n, comparatorPID);

    // Output

    cout << "\nProcess No.\tAT\tCPU Burst Time\tStart Time\tCT\tTAT\tWT\tRT\n";

    for (int i = 0; i < n; i++)

        cout << i << "\t\t" << ps[i].at << "\t" << ps[i].bt << "\t\t" << ps[i].start\_time << "\t\t" << ps[i].ct << "\t" << ps[i].tat << "\t" << ps[i].wt << "\t" << ps[i].rt << endl;

    cout << endl;

    cout << "\nAverage Turn Around time= " << (float)sum\_tat / n;

    cout << "\nAverage Waiting Time= " << (float)sum\_wt / n;

    cout << "\nAverage Response Time= " << (float)sum\_rt / n;

    cout << "\nThroughput= " << n / (float)length\_cycle;

    cout << "\nCPU Utilization(Percentage)= " << cpu\_utilization \* 100;

    return 0;

}

PRORITY(premtive)

#include<iostream>

**using** **namespace** std;

**int** main()

{

**int** bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg\_wt,avg\_tat;

**cout**<<"Enter Total Number of Process:";

    cin>>n;

**cout**<<"\nEnter Burst Time and Priority\n";

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

    {

**cout**<<"\nP["<<i+1<<"]\n";

**cout**<<"Burst Time:";

        cin>>bt[i];

**cout**<<"Priority:";

        cin>>pr[i];

        p[i]=i+1;           *//contains process number*

    }

*//sorting burst time, priority and process number in ascending order using selection sort*

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

    {

        pos=i;

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

        {

**if**(pr[j]<pr[pos])

                pos=j;

        }

        temp=pr[i];

        pr[i]=pr[pos];

        pr[pos]=temp;

        temp=bt[i];

        bt[i]=bt[pos];

        bt[pos]=temp;

        temp=p[i];

        p[i]=p[pos];

        p[pos]=temp;

    }

    wt[0]=0;            *//waiting time for first process is zero*

*//calculate waiting time*

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

    {

        wt[i]=0;

**for**(j=0;j<i;j++)

            wt[i]+=bt[j];

        total+=wt[i];

    }

    avg\_wt=total/n;      *//average waiting time*

    total=0;

**cout**<<"\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time";

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

    {

        tat[i]=bt[i]+wt[i];     *//calculate turnaround time*

        total+=tat[i];

**cout**<<"\nP["<<p[i]<<"]\t\t  "<<bt[i]<<"\t\t    "<<wt[i]<<"\t\t\t"<<tat[i];

    }

    avg\_tat=total/n;     *//average turnaround time*

**cout**<<"\n\nAverage Waiting Time="<<avg\_wt;

**cout**<<"\nAverage Turnaround Time="<<avg\_tat;

**return** 0;

}

LRTF

#include <bits/stdc++.h>

using namespace std;

// creating a structure of a process

struct process

{

    int processno;

    int AT;

    int BT;

    // for backup purpose to print in last

    int BTbackup;

    int WT;

    int TAT;

    int CT;

};

// creating a structure of 4 processes

struct process p[4];

// variable to find the total time

int totaltime = 0;

int prefinaltotal = 0;

// comparator function for sort()

bool compare(process p1, process p2)

{

    // compare the Arrival time of two processes

    return p1.AT < p2.AT;

}

// finding the largest Arrival Time among all the available

// process at that time

int findlargest(int at)

{

    int max = 0, i;

    for (i = 0; i < 4; i++)

    {

        if (p[i].AT <= at)

        {

            if (p[i].BT > p[max].BT)

                max = i;

        }

    }

    // returning the index of the process having the largest BT

    return max;

}

// function to find the completion time of each process

int findCT()

{

    int index;

    int flag = 0;

    int i = p[0].AT;

    while (1)

    {

        if (i <= 4)

        {

            index = findlargest(i);

        }

        else

            index = findlargest(4);

        cout << "Process executing at time " << totaltime

             << " is: P" << index + 1 << "\t";

        p[index].BT -= 1;

        totaltime += 1;

        i++;

        if (p[index].BT == 0)

        {

            p[index].CT = totaltime;

            cout << " Process P" << p[index].processno

                 << " is completed at " << totaltime;

        }

        cout << endl;

        // loop termination condition

        if (totaltime == prefinaltotal)

            break;

    }

}

int main()

{

    int i;

    // initializing the process number

    for (i = 0; i < 4; i++)

    {

        p[i].processno = i + 1;

    }

    // cout<<"arrival time of 4 processes : ";

    for (i = 0; i < 4; i++) // taking AT

    {

        p[i].AT = i + 1;

    }

    // cout<<" Burst time of 4 processes : ";

    for (i = 0; i < 4; i++)

    {

        // assigning {2, 4, 6, 8} as Burst Time to the processes

        // backup for displaying the output in last

        // calculating total required time for terminating

        // the function().

        p[i].BT = 2 \* (i + 1);

        p[i].BTbackup = p[i].BT;

        prefinaltotal += p[i].BT;

    }

    // displaying the process before executing

    cout << "PNo\tAT\tBT\n";

    for (i = 0; i < 4; i++)

    {

        cout << p[i].processno << "\t";

        cout << p[i].AT << "\t";

        cout << p[i].BT << "\t";

        cout << endl;

    }

    cout << endl;

    // sorting process according to Arrival Time

    sort(p, p + 4, compare);

    // calculating initial time when execution starts

    totaltime += p[0].AT;

    // calculating to terminate loop

    prefinaltotal += p[0].AT;

    findCT();

    int totalWT = 0;

    int totalTAT = 0;

    for (i = 0; i < 4; i++)

    {

        // since, TAT = CT - AT

        p[i].TAT = p[i].CT - p[i].AT;

        p[i].WT = p[i].TAT - p[i].BTbackup;

        // finding total waiting time

        totalWT += p[i].WT;

        // finding total turn around time

        totalTAT += p[i].TAT;

    }

    cout << "After execution of all processes ... \n";

    // after all process executes

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

    for (i = 0; i < 4; i++)

    {

        cout << p[i].processno << "\t";

        cout << p[i].AT << "\t";

        cout << p[i].BTbackup << "\t";

        cout << p[i].CT << "\t";

        cout << p[i].TAT << "\t";

        cout << p[i].WT << "\t";

        cout << endl;

    }

    cout << endl;

    cout << "Total TAT = " << totalTAT << endl;

    cout << "Average TAT = " << totalTAT / 4.0 << endl;

    cout << "Total WT = " << totalWT << endl;

    cout << "Average WT = " << totalWT / 4.0 << endl;

    return 0;

}

SRTF

#include<iostream>

**using** **namespace** std;

**int** **main**()

{

**int** a[**10**],b[**10**],x[**10**];

**int** waiting[**10**],turnaround[**10**],completion[**10**];

**int** i,j,smallest,count=**0**,time,n;

**double** avg=**0**,tt=**0**,end;

cout<<"**\n**Enter the number of Processes: "; //input

cin>>n;

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

{

cout<<"**\n**Enter arrival time of process: "; //input

cin>>a[i];

}

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

{

cout<<"**\n**Enter burst time of process: "; //input

cin>>b[i];

}

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

x[i]=b[i];

b[**9**]=**9999**;

**for**(time=**0**; count!=n; time++)

{

smallest=**9**;

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

{

**if**(a[i]<=time && b[i]<b[smallest] && b[i]>**0** )

smallest=i;

}

b[smallest]--;

**if**(b[smallest]==**0**)

{

count++;

end=time+**1**;

completion[smallest] = end;

waiting[smallest] = end - a[smallest] - x[smallest];

turnaround[smallest] = end - a[smallest];

}

}

cout<<"Process"<<"**\t**"<< "burst-time"<<"**\t**"<<"arrival-time" <<"**\t**"<<"waiting-time" <<"**\t**"<<"turnaround-time"<< "**\t**"<<"completion-time"<<endl;

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

{

cout<<"p"<<i+**1**<<"**\t\t**"<<x[i]<<"**\t\t**"<<a[i]<<"**\t\t**"<<waiting[i]<<"**\t\t**"<<turnaround[i]<<"**\t\t**"<<completion[i]<<endl;

avg = avg + waiting[i];

tt = tt + turnaround[i];

}

cout<<"**\n\n**Average waiting time ="<<avg/n;

cout<<" Average Turnaround time ="<<tt/n<<endl;

}

OPTIMAL PAGE REPLACEMENT

// CPP program to demonstrate optimal page

// replacement algorithm.

#include <bits/stdc++.h>

using namespace std;

// Function to check whether a page exists

// in a frame or not

bool search(int key, vector<int>& fr)

{

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

        if (fr[i] == key)

            return true;

    return false;

}

// Function to find the frame that will not be used

// recently in future after given index in pg[0..pn-1]

int predict(int pg[], vector<int>& fr, int pn, int index)

{

    // Store the index of pages which are going

    // to be used recently in future

    int res = -1, farthest = index;

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

        int j;

        for (j = index; j < pn; j++) {

            if (fr[i] == pg[j]) {

                if (j > farthest) {

                    farthest = j;

                    res = i;

                }

                break;

            }

        }

        // If a page is never referenced in future,

        // return it.

        if (j == pn)

            return i;

    }

    // If all of the frames were not in future,

    // return any of them, we return 0. Otherwise

    // we return res.

    return (res == -1) ? 0 : res;

}

void optimalPage(int pg[], int pn, int fn)

{

    // Create an array for given number of

    // frames and initialize it as empty.

    vector<int> fr;

    // Traverse through page reference array

    // and check for miss and hit.

    int hit = 0;

    for (int i = 0; i < pn; i++) {

        // Page found in a frame : HIT

        if (search(pg[i], fr)) {

            hit++;

            continue;

        }

        // Page not found in a frame : MISS

        // If there is space available in frames.

        if (fr.size() < fn)

            fr.push\_back(pg[i]);

        // Find the page to be replaced.

        else {

            int j = predict(pg, fr, pn, i + 1);

            fr[j] = pg[i];

        }

    }

    cout << "No. of hits = " << hit << endl;

    cout << "No. of misses = " << pn - hit << endl;

}

// Driver Function

int main()

{

    int pg[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };

    int pn = sizeof(pg) / sizeof(pg[0]);

    int fn = 4;

    optimalPage(pg, pn, fn);

    return 0;

}

LRU

//C++ implementation of above algorithm

#include<bits/stdc++.h>

using namespace std;

// Function to find page faults using indexes

int pageFaults(int pages[], int n, int capacity)

{

    // To represent set of current pages. We use

    // an unordered\_set so that we quickly check

    // if a page is present in set or not

    unordered\_set<int> s;

    // To store least recently used indexes

    // of pages.

    unordered\_map<int, int> indexes;

    // Start from initial page

    int page\_faults = 0;

    for (int i=0; i<n; i++)

    {

        // Check if the set can hold more pages

        if (s.size() < capacity)

        {

            // Insert it into set if not present

            // already which represents page fault

            if (s.find(pages[i])==s.end())

            {

                s.insert(pages[i]);

                // increment page fault

                page\_faults++;

            }

            // Store the recently used index of

            // each page

            indexes[pages[i]] = i;

        }

        // If the set is full then need to perform lru

        // i.e. remove the least recently used page

        // and insert the current page

        else

        {

            // Check if current page is not already

            // present in the set

            if (s.find(pages[i]) == s.end())

            {

                // Find the least recently used pages

                // that is present in the set

                int lru = INT\_MAX, val;

                for (auto it=s.begin(); it!=s.end(); it++)

                {

                    if (indexes[\*it] < lru)

                    {

                        lru = indexes[\*it];

                        val = \*it;

                    }

                }

                // Remove the indexes page

                s.erase(val);

                // insert the current page

                s.insert(pages[i]);

                // Increment page faults

                page\_faults++;

            }

            // Update the current page index

            indexes[pages[i]] = i;

        }

    }

    return page\_faults;

}

// Driver code

int main()

{

    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

    int n = sizeof(pages)/sizeof(pages[0]);

    int capacity = 4;

    cout << pageFaults(pages, n, capacity);

    return 0;

}