THE NATIONAL INSTITUTE OF ENGINEERING, MYSURU

(An Autonomous Institute under VTU, Belagavi)

Bachelor of Engineering

in

Computer Science and Engineering

Operating Systems

Submitted by

Nishant Sharma 4NI19CS076

Harsh Babal 4NI19CS048

Under the Guidance of

Dr. Jayasri B S

Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

2021-2022

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

THE NATIONAL INSTITUTE OF ENGINEERING

(An Autonomous Institute under VTU, Belgavi)

CERTIFICATE

This is to certify the work carried out by Nishant Sharma (4NI19CS076), Harsh babal (4NI19CS048) in partial fulfilment of the requirements for the completion of tutorial in the course Operating System in the V semester, Department of Computer Science and Engineering as per the academic regulations of The National Institute of Engineering, Mysuru, during the academic year 2021-2022.

Signature of the Couse Instructor

Dr. JAYASRI B S -- Professor & Dean (EAB)

Table of Contents

SJF:

Sl no.	Contents	Page no.
1.	Description	1
2.	Algorithm	1
3.	Implementation and Output	4
4.	Advantages	7
5.	Disadvantages	7

FIFO:

Sl no.	Contents	Page no.
1.	Description	8
2.	Algorithm	8
3.	Implementation and Output	9
4.	Advantages&Disadvantages	11
5.	Github Links	11

Shortest Job First Scheduling Algorithm

Description:

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method is non-preemptive, once the CPU cycle is allocated to process, the process holds it till it reaches a waiting state or terminated. If two processes have same burst time then FCFS (First Come First Serve) is used to break the tie. It significantly reduces the average waiting time for other processes awaiting execution.

Algorithm:

Step1: Take the set of processes as the input with the corresponding arrival time and burst time.

Step2: Sort these processes based on their arrival time and burst time in ascending order**.**

Step3: The process with least arrival time is executed completely and corresponding completion time, turn-around time and waiting time are updated.

Step4: The process that has arrived before the completion of current process execution and has the minimum burst time will be executed next.

Step5: Repeat Step4 until all the processes are executed completely.

Implementation:

```
#include<iostream>
using namespace std;
int SJF_tab[10][6];

//Function to sort in ascending of arrival time
void sortAT(int SJF_tab[][6],int n)
{
  for (int i=0;i<n;i++)
  {
        for (int j=0;j<n-i-1;j++)
        {
            if (SJF_tab[j][1]>SJF_tab[j+1][1])
            {
                for (int k=0;k<3;k++)
                {
                    swap(SJF_tab[j][k],SJF_tab[j+1][k]);
                }
            }
        }
    }

}
```

```
//Funtion to calculate completion time,turn around time and burst time
//4th column has completion time
//5th column has Turn around time
//6th column has Waiting Time
void calculate(int SJF_tab[][6],int n)
{
    SJF_tab[0][3]=SJF_tab[0][1]+SJF_tab[0][2];            //completion time of process arriving first
    SJF_tab[0][4]=SJF_tab[0][3]-SJF_tab[0][1];            //Turn around time of process arriving first
    SJF_tab[0][5]=SJF_tab[0][4]-SJF_tab[0][2];            //Waiting time of process arriving first
    int prevCT,val;
    for(int i=1;i<n;i++)
    {
      prevCT=SJF_tab[i-1][3];                 //Completion time of previous process
      int lowBT=SJF_tab[i][2];                //Burst Time of current process

       /*Check if any of the next processes have arrived before
       completion of the current process and has lowest burst time*/
      for(int j=i;j<n;j++)
      {
          if(prevCT>=SJF_tab[j][1] && lowBT>=SJF_tab[j][2])
        {
            lowBT=SJF_tab[j][2];
            val=j;
        }
      }
      //update the process details with next lowest burst time
        SJF_tab[val][3] = prevCT+ SJF_tab[val][2];
        SJF_tab[val][4] = SJF_tab[val][3] - SJF_tab[val][1];
        SJF_tab[val][5] = SJF_tab[val][4] - SJF_tab[val][2];
        for (int k = 0; k < 6; k++) {
            swap(SJF_tab[val][k], SJF_tab[i][k]);   //reordering based on execution of process
        }
    }
}
```

Implementation:

```
int main()
{
  int n,sumTAT=0,sumWT=0;
  cout<<"Enter the number of processes:";     //input for number of processes
  cin>>n;
  for(int i=0;i<n;i++)
  {
     cout<<"Enter the arrival time and burst time for P"<<i+1<<":"; //input of arrival and burst time
     SJF_tab[i][0]=i+1;
     cin>>SJF_tab[i][1];
     cin>>SJF_tab[i][2];
  }
   cout<<"ProcessID\tArrival Time\tBurst Time\n"; //display the input
  for(int i=0;i<n;i++)
  {
   cout<<SJF_tab[i][0]<<"\t\t"<<SJF_tab[i][1]<<"\t\t"<<SJF_tab[i][2]<<endl;
  }
  cout<<endl;
   sortAT(SJF_tab,n);    //sorting input based on ascending order of arrival time
   calculate(SJF_tab,n); //calculate the completion time,turn around time and waiting time
   cout<<"ProcessID\tArrival Time\tBurst Time\tCompletion Time\tTurnAround Time\tWaiting Time\n";
   //display the table after all the calculations
  for(int i=0;i<n;i++)
  {
   cout<<SJF_tab[i][0]<<"\t\t"<<SJF_tab[i][1]<<"\t\t"<<SJF_tab[i][2]<<"\t\t"<<SJF_tab[i][3]<<"\t\t"
   <<SJF_tab[i][4]<<"\t\t"<<SJF_tab[i][5]<<endl;
  }
  for(int i=0;i<n;i++)
    {
        sumTAT+=SJF_tab[i][4];         //summation of turn around time
        sumWT+=SJF_tab[i][5];          //summation of waiting time
    }

  cout<<"Average Turn Around Time="<<sumTAT/(float)n<<"ms"<<endl; //display average turn around time
  cout<<"Average Waiting Time="<<sumWT/(float)n<<"ms"<<endl;      //display average waiting time
  return 0;
}
```

Output:

```
PS D:\3rd Year\OS tut final> g++ SJF.cpp
PS D:\3rd Year\OS tut final> ./a
Enter the number of processes:7
Enter the arrival time and burst time for P1:0 8
Enter the arrival time and burst time for P2:1 2
Enter the arrival time and burst time for P3:3 4
Enter the arrival time and burst time for P4:4 1
Enter the arrival time and burst time for P5:5 6
Enter the arrival time and burst time for P6:6 5
Enter the arrival time and burst time for P7:10 1
ProcessID       Arrival Time    Burst Time
1               0               8
2               1               2
3               3               4
4               4               1
5               5               6
6               6               5
7               10              1

ProcessID       Arrival Time    Burst Time      Completion Time TurnAround Time Waiting Time
1               0               8               8               8               0
4               4               1               9               5               4
2               1               2               11              10              8
7               10              1               12              2               1
3               3               4               16              13              9
6               6               5               21              15              10
5               5               6               27              22              16
Average Turn Around Time=10.7143ms
Average Waiting Time=6.85714ms
PS D:\3rd Year\OS tut final>
```

Advantages:

1. SJF is frequently used for long term scheduling.
2. It reduces the average waiting time over FIFO (First in First Out) algorithm.
3. It is appropriate for the jobs running in batch, where run times are known in advance.

Disadvantages:

1. Job completion time must be known earlier, but it is hard to predict
2. May suffer with the problem of starvation.
3. SJF can't be implemented for CPU scheduling for the short term.

FIRST IN FIRST OUT PAGE REPLACEMENT ALGORITHM

Description:

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

Algorithm:

1- Start traversing the pages.

i) If set holds less pages than capacity.

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

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

c) Increment page fault

ii) Else

If current page is present in set, do nothing.

Else

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

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

c) Store current page in the queue.

d) Increment page faults.

2. Return page faults.

Implementation:

fifo_page_replacement.cpp > main()

```cpp
int main()
{
    cout<<"Enter the length of the refernce string and number of frames: ";

    int n,framesCount;

    cin>>n>>framesCount;

    vector<int>pages(n);

   // taking refernce string input

   cout<<"Enter the reference string: ";

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

    cout <<"Total number of page faults are: " <<pageFault(pages,n,framesCount) << "\n";
    return 0;

}
```

```cpp
int pageFault(vector<int>pages,int n,int framesCount)
{
    queue<int> q;
    set<int> frame;
    int pageFault_count = 0;

    for(int i=0;i<n;i++)
    {
        int pageNo = pages[i];
        auto it = frame.find(pageNo);
        if(it==frame.end())   // checking if that pageNo is already in the frame or not
        {
            if(frame.size()>=framesCount)  // if frame is full then we remove the first page from the queue
            {
                int replace = q.front();
                q.pop();                    // replaced page is removed from the queue
                frame.erase(replace);       // replaced page is removed from the set
            }

            frame.insert(pageNo); // new page is inserted in the set
            q.push(pageNo);       // new page is inserted in the queue
            pageFault_count++;   //increment page fault count
            flag=0;
        }
        else{flag=1;}

        display(q);
    }
    return pageFault_count;

}
```

IMPLEMENTATION:

fifo_page_replacement.cpp > ...

```cpp
int main()
{
    cout<<"Enter the length of the refernce string and number of frames: ";

    int n;

    cin>>n>>frameCount;

    vector<int>pages(n);

   // taking refernce string input

   cout<<"Enter the reference string: ";

    for(int i=0;i<n;i++)
    {
        cin>>pages[i];
    }
   cout<<endl<<"Frame Table"<<endl;
   int t=pageFault(pages,n,frameCount);
    cout <<"Total number of page faults are: " <<t<< "\n";
    return 0;

}
```

INPUT & OUTPUT:

```
raghavgoenka@pop-os:~/Documents/DataStructurexClass$ g++ fifo_page_replacement
.cpp
raghavgoenka@pop-os:~/Documents/DataStructurexClass$ ./a.out
Enter the length of the refernce string and number of frames: 19 4
Enter the reference string: 3 2 1 3 4 1 6 2 4 3 4 2 1  4 5 2 1 3 4

Frame Table
3       -1      -1      -1      MISS
3       2       -1      -1      MISS
3       2       1       -1      MISS
3       2       1       -1      HIT
3       2       1       4       MISS
3       2       1       4       HIT
```

Advantages:

1. It is simple and easy to understand and implement.
2. Easy to choose the page which needs to be replaced.

Disadvantages:

1. The process effectiveness is low.
2. When we increase the number of frames while using FIFO, we are giving more memory to processes. So, page fault should decrease, but here the page faults are increasing. This problem is called as Belady's Anamaly.

Github Links:

Nishant Sharma -https://github.com/nishantsk?tab=repositories

Harsh Babal