THE NATIONAL INSTITUTE OF ENGINEERING, MYSURU

(An Autonomous Institute under VTU, Belagavi)

Bachelor of Engineering

in

Computer Science and Engineering

Operating Systems

Submitted by

CHANDAN KUMAR 4NI19CS036

RAKESH SHARMA 4NI19CS091

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, Belagavi)

ESTD : 1946

CERTIFICATE

This is to certify the work carried out by CHANDAN KUMAR (4NI19CS036), RAKESH SHARMA (4NI19CS091) 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

Dr. JAYASRI B S

Professor & dean (EAB)

Table of Contents

SJF:

LRU:

Shortest Job First Scheduling Algorithm

- **Description:**

Shortest Job First (SJF) is an algorithm in which the process having the smallest burst 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 terminates. If two processes have same burst time then the process which came earlier in the ready queue is preferred.

- **Algorithm:**
  - **Step1:** Take the set of processes as the input with the corresponding arrival time and burst time.
  - **Step2:** Search for the process with lowest burst time which is not yet executed and present in the ready queue.
  - **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<bits/stdc++.h>
using namespace std;
struct process{
int pid;
int AT;
int BT;
int TAT;
int CT;
int RT;
int WT;
int start_time;
};
int main()
{
  int n;
  struct process p[50];
  int iscompleted[50]={0};
  cout<<"Enter the number of processes : ";
  cin>>n;
  for(int i =0;i<n;i++)
  {
    cout<<"Enter the arrival time of process"<<i+1<<" : ";
    cin>>p[i].AT;
    cout<<"Enter the Burst time of process"<<i+1<<"   : ";
    cin>>p[i].BT;
     p[i].pid= i+1;
    cout<<endl;  }
   int current_time=0;
   int completed =0;
```

```
while(completed!=n)
{  int index=-1;
  int mi=INT_MAX;
            //find the process with lowest burst_time which is not yet completed
   for(int i=0;i<n;i++)
   {
      if(p[i].AT<=current_time&&iscompleted[i]==0)
      {
         if(p[i].BT < mi)  // if variable (mi) is greater the burst time of the ith process
         {
            mi=p[i].BT;
            index=i;
         }
         else if(p[i].BT == mi) // if two process have same burst_time, process which
                                 // arrived early in the ready queue is preferred.
         {
            if(p[i].AT<p[index].AT)
            {
               index=i;
            }
         }
      }
  }

   if(index!=-1)            //if the process is found
   {
     p[index].start_time=current_time;
     p[index].CT = p[index].start_time + p[index].BT;
     p[index].TAT = p[index].CT - p[index].AT;
     p[index].WT = p[index].TAT - p[index].BT;
```

```
        iscompleted[index]=1;
        completed++;
        current_time= p[index].CT;
      }
      else
      {
         current_time++;  //if process is not yet arrived in the ready queue.
      }
   }
   cout<<endl<<endl<<endl;
   cout<<"Process\tAT\tBT\tCT\tTAT\tWT\n\n";
   int a=0,twt=0,trt=0;
   for(int i=0;i<n;i++)
   {
cout<<p[i].pid<<"\t"<<p[i].AT<<"\t"<<p[i].BT<<"\t"<<p[i].CT<<"\t"<<p[i].TAT<<"\t"
<<p[i].WT<<"\n";
      a+=p[i].TAT;      //Total turn around Time
      twt+=p[i].WT;     //Total Waiting Time
      trt+=p[i].RT;     //Total Response Time
   }
   cout<<"\nAverage Turn around time = "<<(float)a/n;
   cout<<"\nAverage Waiting Time = " <<(float)twt/n;
  //N cout<<"\n\n\n\n\n\n\n";
}
/* AT=Arrival Time
   BT- Burst Tme
   CT = Completion Time
   TAT = Turn Around Time
   WT = Waiting Time
   RT = Response Time */
```

➢ Output:

```
"C:\Users\Chandan\Desktop\OS_tutorial\SJF.exe"
Enter the number of processes : 7
Enter the arrival time of process1 : 0
Enter the Burst time of process1   : 8

Enter the arrival time of process2 : 1
Enter the Burst time of process2   : 2

Enter the arrival time of process3 : 3
Enter the Burst time of process3   : 4

Enter the arrival time of process4 : 4
Enter the Burst time of process4   : 1

Enter the arrival time of process5 : 5
Enter the Burst time of process5   : 6

Enter the arrival time of process6 : 6
Enter the Burst time of process6   : 5

Enter the arrival time of process7 : 10
Enter the Burst time of process7   : 1



Process AT      BT      CT      TAT     WT

1       0       8       8       8       0
2       1       2       11      10      8
3       3       4       16      13      9
4       4       1       9       5       4
5       5       6       27      22      16
6       6       5       21      15      10
7       10      1       12      2       1

Average Turn around time = 10.7143
Average Waiting Time = 6.85714
Process returned 0 (0x0)   execution time : 23.295 s
Press any key to continue.
```

➢ Advantages:

- SJF is frequently used for long term scheduling.
- It reduces the average waiting time over FIFO (First in First Out) algorithm.
- It is appropriate for the jobs running in batch, where run times are known in advance.
- SJF method gives the lowest average waiting time for a specific set of processes.

➢ Disadvantages:

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

LEAST RECENTLY USED (LRU) PAGE REPLACEMENT ALGORITHM

- **Description:**

Least Recently Used (LRU) page replacement algorithm works on the concept that the pages that are heavily used in previous instructions are likely to be used heavily in next instructions. And the page that are used very less are likely to be used less in future. Whenever a page fault occurs, the page that is least recently used is removed from the memory frames. Page fault occurs when a referenced page in not found in the memory frames.

- **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 recent occurred index of each page in a map called indexes.

c) Increment page fault

ii) Else

If current page is present in set, do nothing.

Else

a) Find the page in the set that was least recently used. We find it using index array. We basically need to replace the page with minimum index.

b) Replace the found page with current page.

c) Increment page faults.

d) Update index of current page.

2. Return page faults.

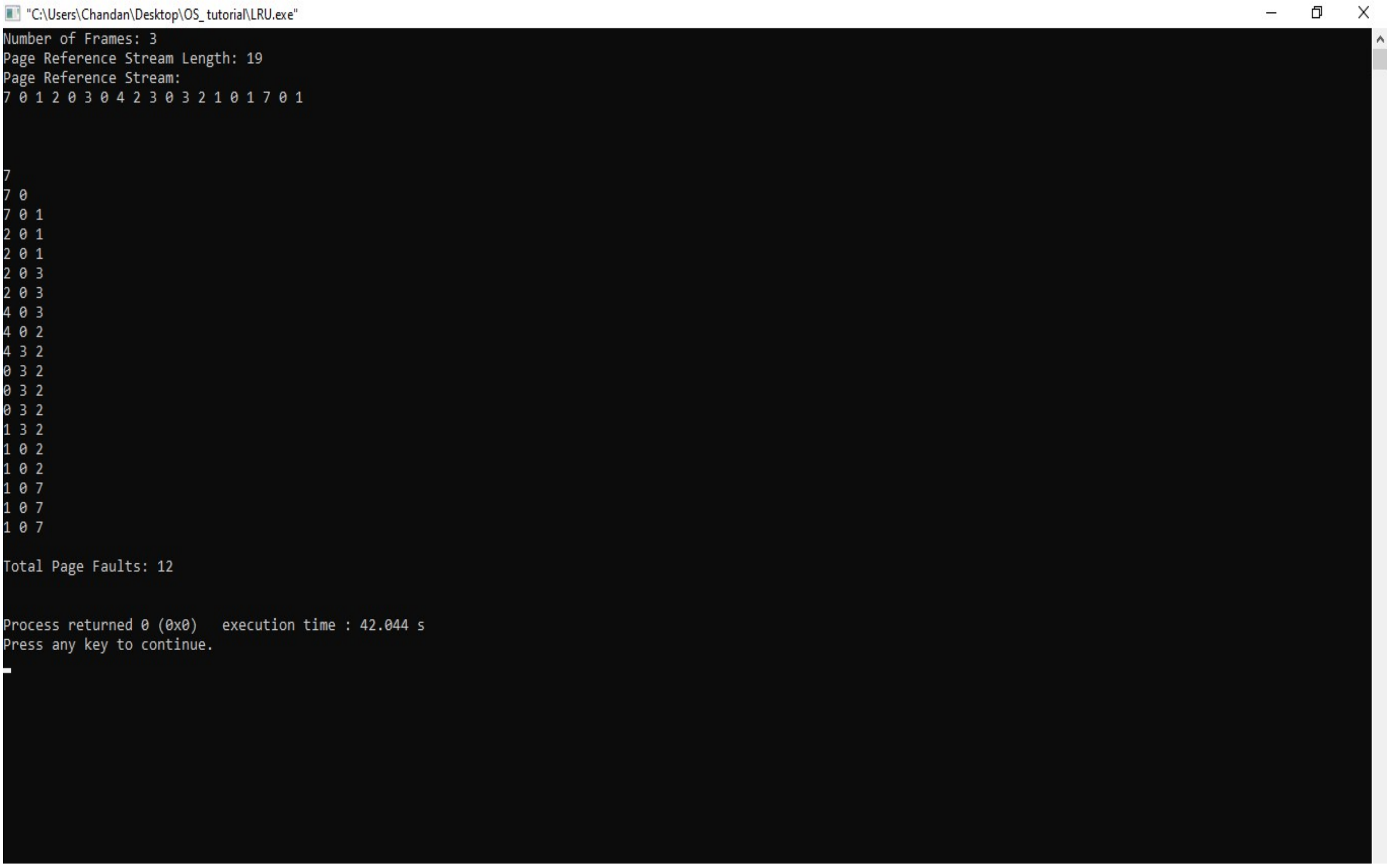
➢ Implementation:

```
#include<bits/stdc++.h>
using namespace std;
const int N=100005;
int n;
int frame_size;
int pages[N];
bool check(vector<int> v,int n)    //function to check if page is available in any of the frame
{
    vector<int>::iterator i;
    i= find(v.begin(),v.end(),n);
    if(i!=v.end())
    return 0;
    else
    return 1;
}
void lru_page_replacement()
{
   vector<int> s;
   map<int, int> indexes;
   int page_faults = 0;

 for (int i=0; i<n; i++)
   {
     if(check(s,pages[i]))  //check if page[i] is present in any of the frame, if not found then,
       {
          if (s.size() < frame_size) //check if all frames are not occupied.
          {
             s.push_back(pages[i]);
             page_faults++;
          }
```

```
            else
            {
                int lru = INT_MAX, val;
                for (int j=0;j<s.size();j++)
                {
                    if (indexes[s[j]] < lru)  //search for page which was least recently used.
                    {
                        lru = indexes[s[j]];
                        val = j;
                    }
                }
               s[val]=pages[i];
                page_faults++;
            }
        }
        indexes[pages[i]] = i;   //to maintain the recent occurred index of each page.
        for(int i=0;i<s.size();i++)  //print all frames.
        {
            cout<<s[i]<<" ";
        }
        cout<<endl;
    }
    cout<<"\nTotal Page Faults: "<<page_faults<<"\n\n";
}
int main()
{
    cout<<"Number of Frames: ";
    cin>>frame_size;

    cout<<"Page Reference Stream Length: ";
    cin>>n;
```

```
    cout<<"Page Reference Stream:\n";
    for(int i=0; i<n; i++)
        cin>>pages[i];

        cout<<endl<<endl<<endl;
    lru_page_replacement();

}
```

➢ Output:



Advantages:

- Easy to choose page which has faulted and hasn't been used for a long time.
- We replace the page which is least recently used, thus free from Belady's Anomaly.

Disadvantages:

- Complex Implementation.
- Expensive.
- Requires hardware support.