THE NATIONAL INSTITUTE OF ENGINEERING, MYSURU

(AN AUTONOMOUS INSTITUTE UNDER VTU, BELAGAVI)

In partial fulfillment of the requirements for the completion of tutorial in the course

Operating System

Semester 5

Computer Science and Engineering

Submitted by

VARUN M -- 4NI19CS120

SURAJ PRAKASH -- 4NI19CS108

To the course instructor

Dr JAYASRI B S

(Associate Professor)

DEPARTMENT OF COMPTER SCIENCE AND ENGINEERING

THE NATIONAL INSTITUTE OF ENGINEERING

Mysuru-570008

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 VARUN M (4NI19CS120),

SURAJ PRAKASH (4NI19CS108) in partial fulfillment 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 Course Instructor

_____

(Dr JAYASRI B S)

1 CONTENTS

2 SJF – CPU SCHEDULING ALGORITHM

2.1 CPU SCHEDULING ALGORITHM:

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. These algorithms are either non-preemptive or preemptive. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

2.2 SJF:

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution.

2.3 ALGORITHM:

1- Sort all the process according to the arrival time.
2- Then select the process which has maximum arrival time and minimum Burst time.
3- After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.

2.4 ADVANTAGES AND DISADVANTAGES:

Advantages:

- It reduces the average waiting time over FIFO (First in First Out) algorithm.
- SJF method gives the lowest average waiting time for a specific set of processes.

Disadvantages:

- It requires additional Data Structure to be implemented.
- Its execution is bit complicated.

- Its execution may need substantial hardware assistance.

2.5 Example:

Suppose we have the following 5 processes with process ID's P1, P2, P3, P4 and P5 and they arrive into the CPU in the following manner:

PID	Arrival Time	Burst Time
1	1	7
2	3	3
3	6	2
4	7	10
5	9	8

Explanation:

Since No Process arrives at time 0 hence; there will be an empty slot in the **Gantt chart** from time 0 to 1 (the time at which the first process arrives).

According to the algorithm, the OS schedules the process which is having the lowest burst time among the available processes in the ready queue.

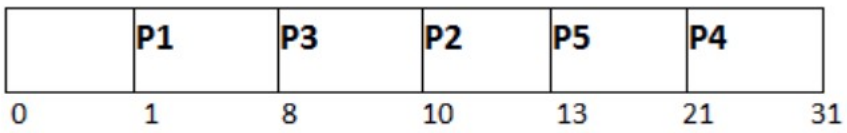
Till now, we have only one process in the ready queue hence the scheduler will schedule this to the processor no matter what is its burst time.

This will be executed till 8 units of time. Till then we have three more processes arrived in the ready queue hence the scheduler will choose the process with the lowest burst time.

Among the processes given in the table, P3 will be executed next since it is having the lowest burst time among all the available processes.

So that's how the procedure will go on in **shortest job first (SJF)** scheduling algorithm.

Gantt Chart:



PID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	1	7	8	7	0
2	3	3	13	10	7
3	6	2	10	4	2
4	7	10	31	24	14
5	9	8	21	12	4

Total Turn Around Time = 7+10+4+24+12

= 57 milliseconds

Average Turn Around Time= Total Turn Around Time / Total No. of Processes

= 57 / 5

= 11.4 milliseconds

Total Waiting Time = 0 + 7 + 2 + 14 +4

= 27 milliseconds

Average Waiting Time = Total Waiting Time / Total No. of Processes

= 27/5

= 5.4 milliseconds

2.6 CODE

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

```
using namespace std;

struct process {
    int pid;
    int arrival_time;
    int burst_time;
    int start_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
    int response_time;
};


int main(){

    int n;
    struct process p[100];
    float avg_turnaround_time;
    float avg_waiting_time;
    float avg_response_time;
    int total_turnaround_time = 0;
    int total_waiting_time = 0;
    int total_response_time = 0;
    int total_idle_time = 0;
    int is_completed[100];
    for(int i=0;i<100;i++)
    is_completed[i]=0;

    cout<<"Enter the number of processes: ";
    cin>>n;
    for(int i = 0; i < n; i++) {
```

```
    cout<<"Enter arrival time of process "<<i+1<<": ";
    cin>>p[i].arrival_time;
    cout<<"Enter burst time of process "<<i+1<<": ";
    cin>>p[i].burst_time;
    p[i].pid = i+1;
    cout<<endl;
}

int current_time = 0;
int completed = 0;
 while(completed != n) {
    int idx = -1;
    int mn = 10000000;
    for(int i = 0; i < n; i++) {
        if(p[i].arrival_time <= current_time && is_completed[i] == 0) {
            if(p[i].burst_time < mn) {
                mn = p[i].burst_time;
                idx = i;
            }
            if(p[i].burst_time == mn) {
                if(p[i].arrival_time < p[idx].arrival_time) {
                    mn = p[i].burst_time;
                    idx = i;
                }
            }
        }
    }
    if(idx != -1) {
        p[idx].start_time = current_time;
        p[idx].completion_time = p[idx].start_time + p[idx].burst_time;
        p[idx].turnaround_time = p[idx].completion_time - p[idx].arrival_time;
        p[idx].waiting_time = p[idx].turnaround_time - p[idx].burst_time;
        p[idx].response_time = p[idx].start_time - p[idx].arrival_time;
```

```
            total_turnaround_time += p[idx].turnaround_time;
            total_waiting_time += p[idx].waiting_time;
            total_response_time += p[idx].response_time;

            is_completed[idx] = 1;
            completed++;
            current_time = p[idx].completion_time;
        }
        else {
            current_time++;
        }

    }

    avg_turnaround_time = (float) total_turnaround_time / n;
    avg_waiting_time = (float) total_waiting_time / n;
    avg_response_time = (float) total_response_time / n;

    cout<<endl<<endl;

    cout<<"Process
ID\t"<<"AT\t"<<"BT\t"<<"ST\t"<<"CT\t"<<"TAT\t"<<"WT\t"<<"RT\t"<<"\n"<<endl
;

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

cout<<"P"<<p[i].pid<<"\t\t"<<p[i].arrival_time<<"\t"<<p[i].burst_time<<"\t"<<p[i].st
art_time<<"\t"<<p[i].completion_time<<"\t"<<p[i].turnaround_time<<"\t"<<p[i].waiti
ng_time<<"\t"<<p[i].response_time<<"\t"<<"\n"<<endl;
    }
    cout<<"Average Turnaround Time = "<<avg_turnaround_time<<endl;
    cout<<"Average Waiting Time = "<<avg_waiting_time<<endl;
```

```
    cout<<"Average Response Time = "<<avg_response_time<<endl;

    return 0;
}
```

2.7 OUTPUT

```
Enter the number of processes: 5
Enter arrival time of process 1: 1
Enter burst time of process 1: 7

Enter arrival time of process 2: 3
Enter burst time of process 2: 3

Enter arrival time of process 3: 6
Enter burst time of process 3: 2

Enter arrival time of process 4: 7
Enter burst time of process 4: 10

Enter arrival time of process 5: 9
Enter burst time of process 5: 8



Process ID      AT      BT      ST      CT      TAT     WT      RT

P1              1       7       1       8       7       0       0

P2              3       3       10      13      10      7       7

P3              6       2       8       10      4       2       2

P4              7       10      21      31      24      14      14

P5              9       8       13      21      12      4       4

Average Turnaround Time = 11.4
Average Waiting Time = 5.4
Average Response Time = 5.4
```

3 LRU - PAGE REPLACEMENT ALGORITHM

3.1 PAGE REPLACEMENT ALGORITHM:

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in. Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

3.2 LRU

LRU policy follows the concept of locality of reference as the base for its page replacement decisions. LRU policy says that pages that have not been used for the longest period of time will probably not be used for a long time. So, when a page fault occurs, it is better to replace the page that has been unused for a long time. This strategy is called LRU (Least Recently Used) paging. LRU policy is often used as a page replacement algorithm and considered to be good

3.3 ADVANTAGES AND DISADVANTAGES

Advantages :

It is open for full analysis. In this, we replace the page which is least recently used, thus free from Belady's Anomaly. Easy to choose page which has faulted and hasn't been used for a long time.

Disadvantage :

The problem is to determine an order for the frames defined by the time of last use. So, implementing LRU will require extra overhead either software implementation with extra computational time or implementation by hardware. To implement LRU we can use a linked list of all pages in memory, with the most recent page at the font and the least recent page at

the rare. So, update deletion and insertion of linked lists is a time-consuming operation. Another way to implement LRU is special hardware either counter or stack.

3.4 ALGORITHM:

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

1. Start traversing the pages.

i) If set holds less pages than capacity.

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

b) Simultaneously maintain the recent occurred index of each page in a map called indexes.

c) Increment page fault

ii) Else If current page is present in set, do nothing. Else

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

b) Replace the found page with current page.

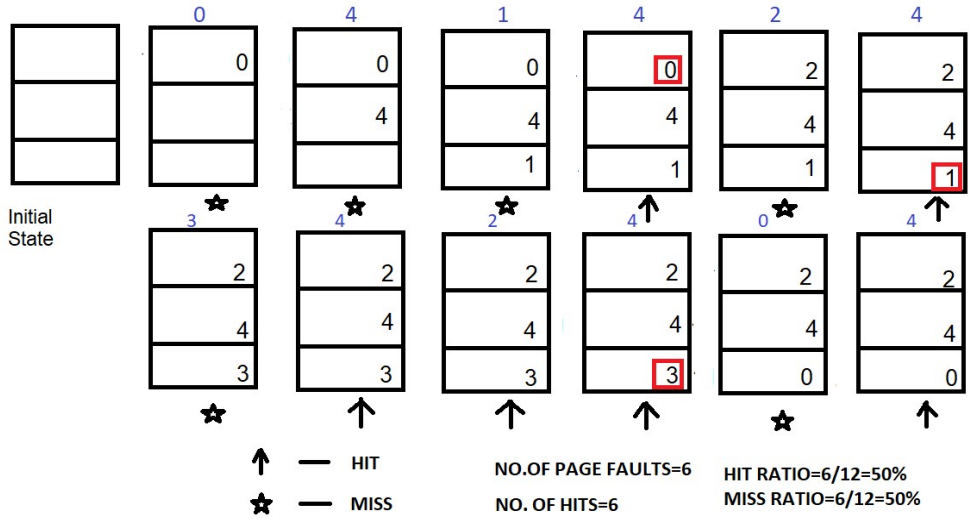
c) Increment page faults.

d) Update index of current page.

2. Return page faults.

3.5 EXAMPLE

Sample String : 0 4 1 4 2 4 3 4 2 4 0 4



3.6 CODE:

```
#include<stdio.h>

                    #include<stdio.h>
                    int findLRU(int time[], int n){
                    int i, minimum = time[0], pos = 0;

                    for(i = 1; i < n; ++i){
                    if(time[i] < minimum){
                    minimum = time[i];
                    pos = i;
                    }
                    }
                    return pos;
                    }

                    int main()
                    {
                       int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0,
                    time[10], flag1, flag2, i, j, pos, faults = 0;
                    printf("Enter number of frames: ");
                    scanf("%d", &no_of_frames);
                    printf("Enter number of pages: ");
                    scanf("%d", &no_of_pages);
                    printf("Enter reference string: ");
                       for(i = 0; i < no_of_pages; ++i){
                        scanf("%d", &pages[i]);
                       }
```

```
for(i = 0; i < no_of_frames; ++i){
    frames[i] = -1;
   }

   for(i = 0; i < no_of_pages; ++i){
    flag1 = flag2 = 0;

    for(j = 0; j < no_of_frames; ++j){
    if(frames[j] == pages[i]){
    counter++;
    time[j] = counter;
  flag1 = flag2 = 1;
  break;
  }
   }

    if(flag1 == 0){
for(j = 0; j < no_of_frames; ++j){
    if(frames[j] == -1){
    counter++;
    faults++;
    frames[j] = pages[i];
    time[j] = counter;
    flag2 = 1;
    break;
    }
    }
    }

    if(flag2 == 0){
    pos = findLRU(time, no_of_frames);
    counter++;
    faults++;
```

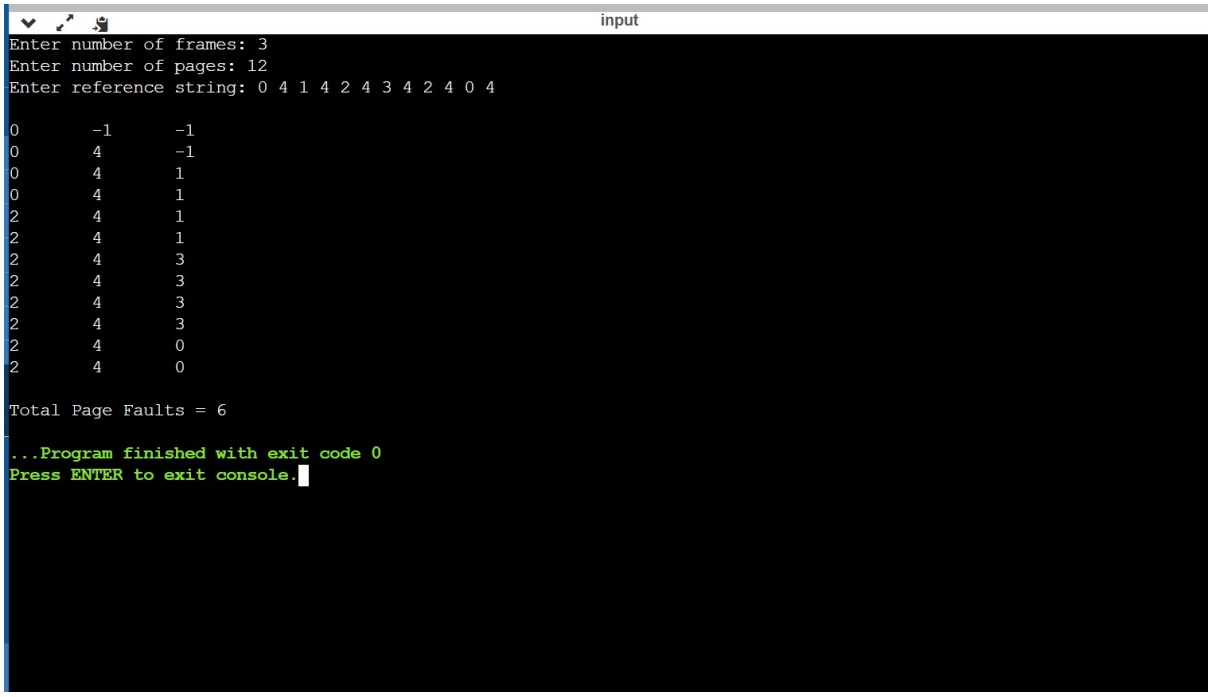
```
            frames[pos] = pages[i];
            time[pos] = counter;
            }

            printf("\n");

            for(j = 0; j < no_of_frames; ++j){
            printf("%d\t", frames[j]);
            }
        }
        printf("\n\nTotal Page Faults = %d", faults);

          return 0;
        }
```

3.7 OUTPUT:



4 GITHUB LINKS

VARUN M (4NI19CS120) -- https://github.com/varun-mgowda

SURAJ PRAKASH (4NI19CS108) -- https://github.com/SurajPrakash41