

THE NATIONAL INSTITUTE OF ENGINEERING, MYSURU
(AN AUTONOMOUS INSTITUTE UNDER VTU, BELAGAVI)



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

Operating System
Semester 5
Computer Science and Engineering

Submitted by
SHALINI SG -- 4NI19CS101
SAUMYA NIGAM – 4NI19CS100

To the course instructor
Dr JAYASRI B S
(Associate Professor)

DEPARTMENT OF COMPUTER 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, Belagavi)



CERTIFICATE

This is to certify the work carried out by SHALINI SG (4NI19CS101),
SAUMYA NIGAM (4NI19CS100) 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)

Associate Professor

1 CONTENTS

2	FCFS – CPU SCHEDULING ALGORITHM	4
2.1	CPU Scheduling Algorithm:	4
2.2	FCFS:	4
2.3	Pseudocode:	4
2.4	Advantages and Disadvantages:.....	4
2.5	Example:	5
2.6	Code	6
2.7	Output	8
3	FIFO - PAGE REPLACEMENT ALGORITHM	9
3.1	Page Replacement Algorithm:	9
3.2	FIFO.....	9
3.3	Advantages and Disadvantages.....	9
3.4	Pseudocode:	10
3.5	Example	10
3.6	Code:	11
3.7	Output:	14
4	GITHUB Links	14

2 FCFS – 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 FCFS:

First come first serve (FCFS) scheduling algorithm simply schedules the jobs according to their arrival time. The job which comes first in the ready queue will get the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU.

2.3 PSEUDOCODE:

Data: Processes p[], No. of Processes n,

Burst Time bt[], Wait Time wt[n], Turn Around Time tat[n],

findWaitingTime(processes,n,bt,wt), findTurnAroundTime(processes,n,bt,wt,tat)

findAvgTime(processes,n,bt)

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

total_wt = total_wt + wt[i]

total_tat = total_tat + tat[i]

avgWaitingTime = total_wt / n

avgTurnAroundTime = total_tat / n

end

end

2.4 ADVANTAGES AND DISADVANTAGES:

Advantages: FCFS scheduling algorithm is simple, easy and based on First In First Out principle.

Disadvantages: It suffers from Convoy Effect. It does not consider the priority or burst time of the processes. The scheduling method is non pre-emptive, the process will run to the

completion. Due to the non-pre-emptive nature of the algorithm, the problem of starvation may occur. Although it is easy to implement, but it is poor in performance since the average waiting time is higher as compare to other scheduling algorithms.

2.5 EXAMPLE:

Suppose we have the following 3 processes with process ID's P1, P2, and P3 and they arrive into the CPU in the following manner:

PROCESSES	ARRIVAL TIME	BURST TIME
P1	0	5
P2	3	9
P3	6	6

Gant Chart:

P1	P2	P3
0	5	14
		20

Table:

PROCESSES	ARRIVAL TIME	BURST TIME	COMPLETION TIME	TURN AROUND TIME	WAIT TIME
P1	0	5	5	5	0
P2	3	9	14	11	2
P3	6	6	20	14	8

$$\begin{aligned}\text{Total Turn Around Time} &= 5 + 11 + 14 \\ &= 30 \text{ milliseconds}\end{aligned}$$

$$\begin{aligned}\text{Average Turn Around Time} &= \text{Total Turn Around Time} / \text{Total No. of Processes} = 30 / 3 \\ &= 10 \text{ milliseconds}\end{aligned}$$

$$\begin{aligned}\text{Total Waiting Time} &= 0 + 2 + 8 \\ &= 10 \text{ milliseconds}\end{aligned}$$

$$\begin{aligned}\text{Average Waiting Time} &= \text{Total Waiting Time} / \text{Total No. of Processes} = 10 / 3 \\ &= 3.33 \text{ milliseconds}\end{aligned}$$

2.6 CODE

```
#include<iostream>

using namespace std;

// Function to find the waiting time for all processes
void findWaitingTime(int processes[], int n, int bt[],
                    int wt[], int at[])
{
    int completion_time[n];
    completion_time[0] = at[0];
    wt[0] = 0;

    // calculating waiting time
    for (int i = 1; i < n ; i++)
    {
        // Add burst time of previous processes
        completion_time[i] = completion_time[i-1] + bt[i-1];

        // Find waiting time for current process = sum - at[i]
        wt[i] = completion_time[i] - at[i];

        // If waiting time for a process is in negative
        // that means it is already in the ready queue
        // before CPU becomes idle so its waiting time is 0
        if (wt[i] < 0)
            wt[i] = 0;
    }
}

// Function to calculate turn around time
void findTurnAroundTime(int processes[], int n, int bt[],
```

```

        int wt[], int tat[])
    {
        // Calculating turnaround time by adding bt[i] + wt[i]
        for (int i = 0; i < n ; i++)
            tat[i] = bt[i] + wt[i];
    }

// Function to calculate average waiting and turn-around times.
void findavgTime(int processes[], int n, int bt[], int at[])
{
    int wt[n], tat[n];

    // Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt, at);

    // Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);

    // Display processes along with all details
    cout << "Processes " << " Burst Time " << " Arrival Time "
        << " Waiting Time " << " Turn-Around Time "
        << " Completion Time \n";
    int total_wt = 0, total_tat = 0;
    for (int i = 0 ; i < n ; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        int compl_time = tat[i] + at[i];
        cout << " " << i+1 << "\t\t" << bt[i] << "\t\t"
            << at[i] << "\t\t" << wt[i] << "\t\t "
            << tat[i] << "\t\t" << compl_time << endl;
    }
}

```

```

    cout << "Average waiting time = "
        << (float)total_wt / (float)n;

    cout << "\nAverage turn around time = "
        << (float)total_tat / (float)n;
}

// Driver code
int main()
{
    // Process id's
    int processes[] = {1, 2, 3};

    int n = sizeof processes / sizeof processes[0];

    // Burst time of all processes
    int burst_time[] = {5, 9, 6};

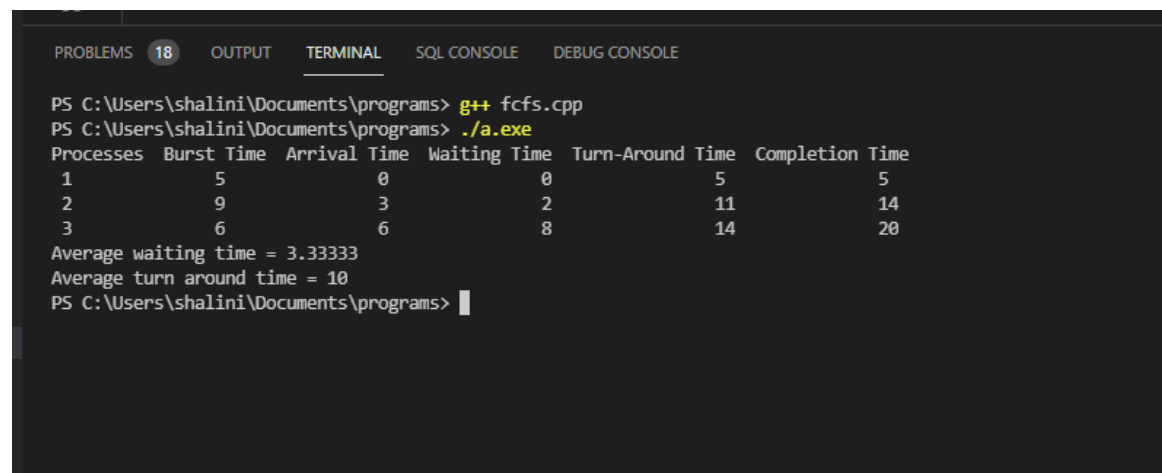
    // Arrival time of all processes
    int arrival_time[] = {0, 3, 6};

    findavgTime(processes, n, burst_time, arrival_time)

    return 0;
}

```

2.7 OUTPUT



```

PROBLEMS 18 OUTPUT TERMINAL SQL CONSOLE DEBUG CONSOLE
PS C:\Users\shalini\Documents\programs> g++ fcfs.cpp
PS C:\Users\shalini\Documents\programs> ./a.exe
Processes Burst Time Arrival Time Waiting Time Turn-Around Time Completion Time
1          5          0          0          5          5
2          9          3          2          11         14
3          6          6          8          14         20
Average waiting time = 3.3333
Average turn around time = 10
PS C:\Users\shalini\Documents\programs>

```


3 FIFO - 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 FIFO

First In First Out (FIFO) page replacement algorithm is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, 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.

3.3 ADVANTAGES AND DISADVANTAGES

Advantages :

It is easy to understand and implement. The number of operations is limited in a queue makes the implementation simple.

Disadvantage :

When the number of incoming pages is large, it might not provide excellent performance.

When we increase the number of frames or capacity to store pages in the queue, it should give us less number of page faults.

Sometimes FIFO may behave abnormally, and it may increase the number of page faults. This behavior of FIFO is called Belady's anomaly.

In FIFO, the system should keep track of all the frames. Sometimes it results in slow process execution.

3.4 PSEUDOCODE:

Data: Pages P, Number of Pages N, Capacity C

Result: Number of page fault PF

Function FindPage Fault(P, N, C)

S = set();

QPage Queue();

PF = 0;

for i = 0 to length(N) do

 if length(S) < C then

 if P[i] not in S then

 S.add(P[i]);

 PF = PF +1;

 QPage.put(P[i]);

 end

 else

 if P[i] not in S then

 val = QPage.queue[0];

 QPage.get();

 S.remove(val);

 S.add(P[i]);

 QPage.put(P[i]);

 PF = PF +1;

 end

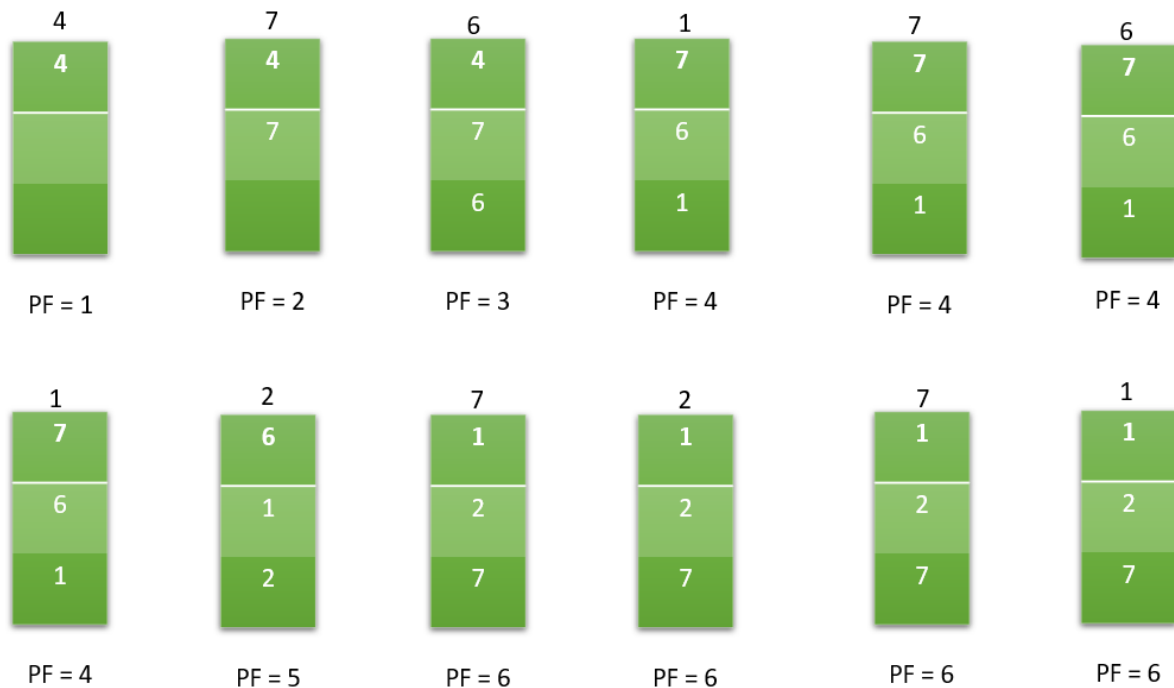
 end

end

return PF:

3.5 EXAMPLE

Sample String : 4 7 6 1 7 6 1 2 7 2 7 1



Number of page faults = 6

Miss – ratio = $(6/12) \times 100 = 50\%$

Hit – ratio = $100 - 50 = 50\%$

3.6 CODE:

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

using namespace std;

// Function to find page faults using FIFO
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 the pages in FIFO manner
    queue<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())
        {
            // Insert the current page into the set
            s.insert(pages[i]);

            // increment page fault
            page_faults++;

            // Push the current page into the queue
            indexes.push(pages[i]);
        }
    }
    /* If the set is full then need to perform FIFO i.e. remove the first page of the queue from set
    and queue both and insert the current page */
    else
    {
        // Check if current page is not already present in the set
        if (s.find(pages[i]) == s.end())
        {
            // Store the first page in the queue to be used to find and erase the page from the set
            int val = indexes.front();

```

```

        // Pop the first page from the queue
        indexes.pop();

        // Remove the indexes page from the set
        s.erase(val);

        // insert the current page in the set
        s.insert(pages[i]);

        // push the current page into the queue
        indexes.push(pages[i]);

        // Increment page faults
        page_faults++;
    }
}

return page_faults;
}

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

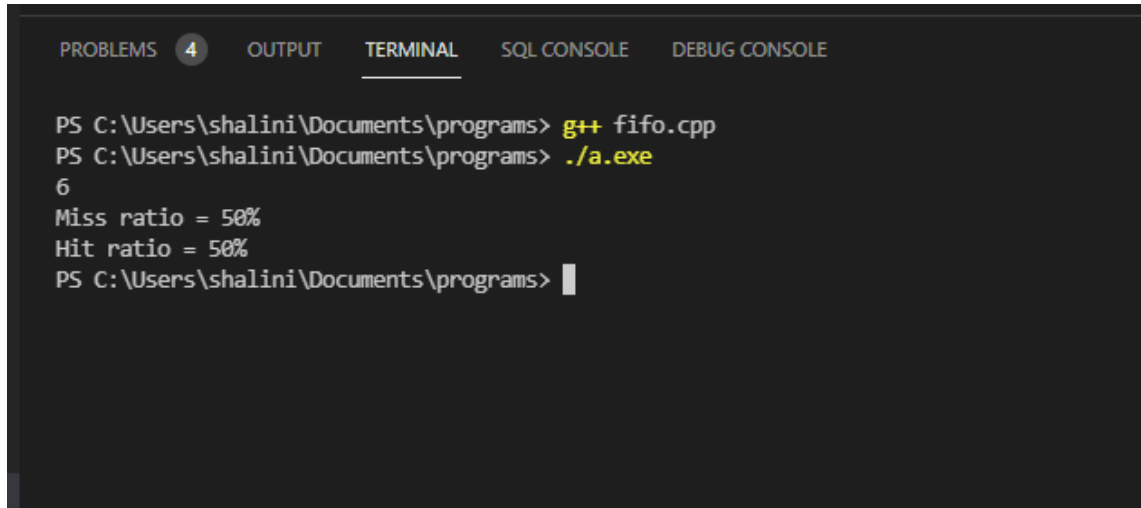
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    float miss_per, hit_per;

    cout << pageFaults(pages, n, capacity);
    miss_per = ((pageFaults(pages, n, capacity)*1.0)/n)*100;
    hit_per = 100 - miss_per;
    cout<<"\nMiss ratio = "<<miss_per<<"%";
    cout<<"\nHit ratio = "<<hit_per<<"%";
}

```

```
    return 0;
}
```

3.7 OUTPUT:



```
PROBLEMS 4 OUTPUT TERMINAL SQL CONSOLE DEBUG CONSOLE

PS C:\Users\shalini\Documents\programs> g++ fifo.cpp
PS C:\Users\shalini\Documents\programs> ./a.exe
6
Miss ratio = 50%
Hit ratio = 50%
PS C:\Users\shalini\Documents\programs> |
```

4 GITHUB LINKS

SHALINI SG (4NI19CS101) -- <https://github.com/shalinishygan>

SAUMYA NIGAM (4NI19CS100) -- <https://github.com/SaumyaNigam1404>