The National Institute of Engineering, Mysuru
Department of Computer Science and Engineering

Operating System tutorials (CS5C02) – 2021-22

To the Course Instructor

Dr. JAYASRI BS

(Associate Professor)

Operating System Report

Title: Page replacement and Scheduling Algorithms

Team Details

Sl. No.	USN	NAME
1.	4NI19CS101	SHALINI S G
2.	4NI19CS100	SAUMYA NIGAM

FIFO (First In First Out page replacement algorithm)

Introduction

• In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

• 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.

How it works !

- Operating system keeps track of all the pages in a queue

- If a page is already present in set, it's a hit and there will be no change in the queue. If the page in the string is not present, it's a page fault.

- If set holds less pages than capacity, insert page into the set one by one until all page requests are processed. Simultaneously maintain the pages in the queue to perform FIFO and increment page faults. If current page is already present, it does nothing.

- If the queue is full, remove the first page from the queue, replace that first page with current page in the string and store it back in the queue.

- Then page faults are incremented.

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:
```

Program

```cpp
#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;
```

```cpp
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]);
            }
        }
```

```cpp
        /* 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;
}
```
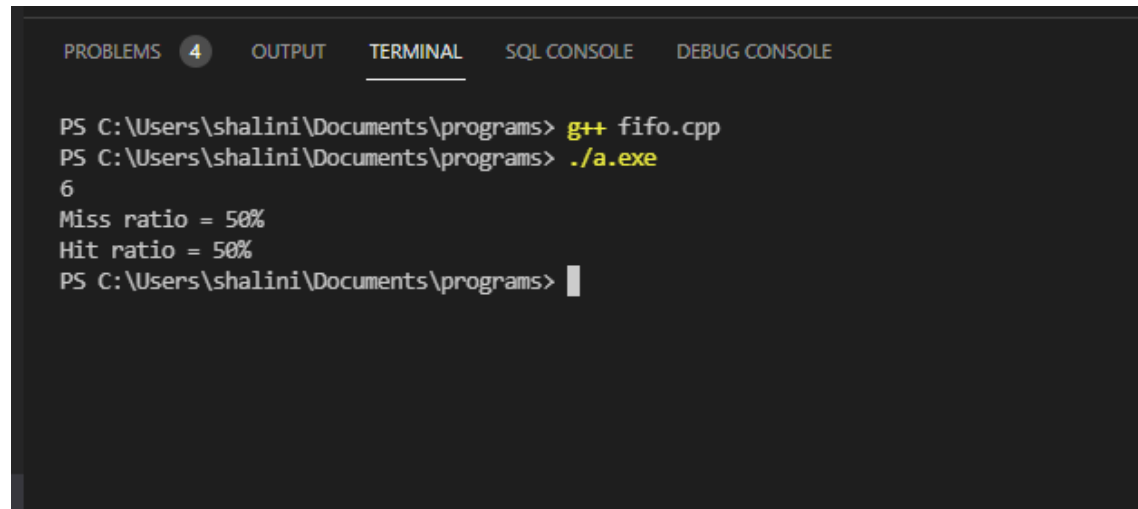
```cpp
int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,
                   2, 3, 0, 3, 2};
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    float miss_per, hit_per;
    cout << pageFaults(pages, n, capacity);
    miss_per = ((pageFaults(pages, n, capacity)*1.0)/n)*100;
    hit_per = 100 - miss_per;
    cout<<"\nMiss ratio = "<<miss_per<<"%";
    cout<<"\nHit ratio = "<<hit_per<<"%";
    return 0;
}
```

```
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>
```

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

- **Disadvantages:**
- It does not consider the priority or burst time of the processes.
- 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.

FCFS (First Come First Serve CPU scheduling algorithm)

Introduction:

- 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.

- 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.

How it works !

- Number of processes and process id's are given as input in the order of their arrival time

- Waiting Time and Turn Around Time is calculated with Burst Time known

- The following formulas are used for calculation

  1. Completion time of current process = Completion Time of previous process + Burst Time of previous process

2. Wait Time of current process = Completion Time of current process - Arrival Time of current process

3. Turn Around Time of current process = Burst Time of current process + Waiting Time of current process

- Total Waiting Time and Total Turn Around Time are calculated

- Finally, Average Waiting Time and Average Turn Around Time are calculated by using following formulas

  1. Average Waiting Time = Total Waiting Time / Number of processes

  2. Average Turn Around Time = Total Turn Around Time / Number of processes

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
```

Program

```cpp
#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 service_time[n];
    service_time[0] = at[0];
    wt[0] = 0;


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

        // Find waiting time for current process = sum - at[i]
```

```c
    wt[i] = service_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);
```

```cpp
    // 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;
}
```

```cpp
// 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;
}
```
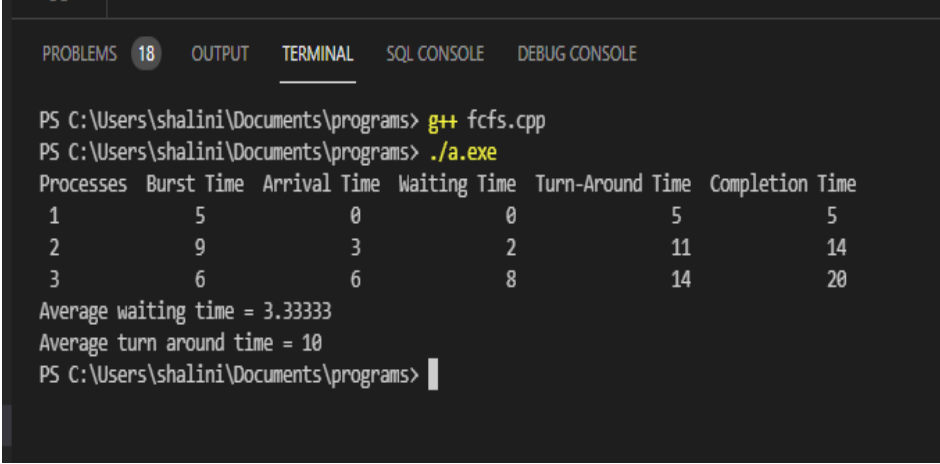
Advantages:

This scheduling algorithm is simple, easy and based on First In First Out principle

Disadvantages:

- The scheduling method is non preemptive, the process will run to the completion.

- Due to the non-preemptive 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.