THE NATIONAL INSTITUTE OF ENGINEERING, MYSURU

(An Autonomous Institute under VTU, Belagavi)

Bachelor of Engineering

in

Computer Science and Engineering

Operating Systems (CS5C02)

Semester: 5

Submitted by

Tanmay Kumar 4NI19CS114

Yash Chauhan 4NI19CS123

Course Instructor:

Dr. Jayasri B S

(Professor and Dean – EAB)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

2021-2022

Table of Contents

SJF:

LRU:

Round Robin Scheduling Algorithm

Description:

A **Round-Robin scheduling algorithm** is used to schedule the process fairly for each job a time slot or quantum and the interrupting the job if it is not completed by then the job come after the other job which is arrived in the quantum time that makes these scheduling fairly.

Algorithm:

Step 1: Organize all processes according to their arrival time in the ready queue. The queue structure of the ready queue is based on the FIFO structure to execute all CPU processes.

Step 2: Now, we push the first process from the ready queue to execute its task for a fixed time, allocated by each process that arrives in the queue.

Step 3: If the process cannot complete their task within defined time interval or slots because it is stopped by another process that pushes from the ready queue to execute their task due to arrival time of the next process is reached. Therefore, CPU saved the previous state of the process, which helps to resume from the point where it is interrupted. (If the burst time of the process is left, push the process end of the ready queue).

Step 4: Similarly, the scheduler selects another process from the ready queue to execute its tasks. When a process finishes its task within time slots, the process will not go for further execution because the process's burst time is finished.

Step 5: Similarly, we repeat all the steps to execute the process until the work has finished.

Implementation:

```cpp
#include <iostream>
using namespace std;

void queueUpdation(int queue[], int timer, int arrival[], int n, int maxProccessIndex)
{
    int zeroIndex;
    for (int i = 0; i < n; i++)
    {
        if (queue[i] == 0)
        {
            zeroIndex = i;
            break;
        }
    }
    queue[zeroIndex] = maxProccessIndex + 1;
}

void queueMaintainence(int queue[], int n)
{
    for (int i = 0; (i < n - 1) && (queue[i + 1] != 0); i++)
    {
        int temp = queue[i];
        queue[i] = queue[i + 1];
        queue[i + 1] = temp;
    }
}

void checkNewArrival(int timer, int arrival[], int n, int maxProccessIndex, int queue[])
{
    if (timer <= arrival[n - 1])
    {
        bool newArrival = false;
        for (int j = (maxProccessIndex + 1); j < n; j++)
        {
            if (arrival[j] <= timer)
            {
                if (maxProccessIndex < j)
                {
                    maxProccessIndex = j;
                    newArrival = true;
                }
            }
        }
        // adding the incoming process to the ready queue
        // When new Process Arrives
        if (newArrival)
            queueUpdation(queue, timer, arrival, n, maxProccessIndex);
    }
}

int main()
{
    int n, tq, timer = 0, maxProccessIndex = 0;
    float avgWait = 0, avgTT = 0;
    cout << "\nEnter the Time Quanta:";
    cin >> tq;
    cout << "\nEnter the No. of Processess:";
    cin >> n;
    int arrival[n], burst[n], wait[n], turn[n], queue[n], temp_burst[n];
    bool complete[n];

    cout << "\nEnter Arrival Time of Processess : ";
    for (int i = 0; i < n; i++)
        cin >> arrival[i];

    cout << "\nEnter the Burst Time of Processess : ";
    for (int i = 0; i < n; i++)
    {
        cin >> burst[i];
        temp_burst[i] = burst[i];
    }

```

```
    }

    for (int i = 0; i < n; i++)
    { // Initializing the queue and complete array
        complete[i] = false;
        queue[i] = 0;
    }
    while (timer < arrival[0]) // Incrementing Timer until the first process arrives
        timer++;
    queue[0] = 1;

    while (true)
    {
        bool flag = true;
        for (int i = 0; i < n; i++)
        {
            if (temp_burst[i] != 0)
            {
                flag = false;
                break;
            }
        }
        if (flag)
            break;

        for (int i = 0; (i < n) && (queue[i] != 0); i++)
        {
            int ctr = 0;
            while ((ctr < tq) && (temp_burst[queue[0] - 1] > 0))
            {
                temp_burst[queue[0] - 1] -= 1;
                timer += 1;
                ctr++;

                // Checking and Updating the ready queue until all the processes arrive
                checkNewArrival(timer, arrival, n, maxProccessIndex, queue);
            }
            // If a process is completed then store its exit time
            // and mark it as completed
            if ((temp_burst[queue[0] - 1] == 0) && (complete[queue[0] - 1] == false))
            {
                // turn array currently stores the completion time
                turn[queue[0] - 1] = timer;
                complete[queue[0] - 1] = true;
            }

            // checks whether or not CPU is idle
            bool idle = true;
            if (queue[n - 1] == 0)
            {
                for (int i = 0; i < n && queue[i] != 0; i++)
                {
                    if (complete[queue[i] - 1] == false)
                    {
                        idle = false;
                    }
                }
            }
            else
                idle = false;

            if (idle)
            {
                timer++;
                checkNewArrival(timer, arrival, n, maxProccessIndex, queue);
            }

            // Maintaining the entries of processes
            // after each premption in the ready Queue
            queueMaintainence(queue, n);
        }
    }

    for (int i = 0; i < n; i++)
    {
        turn[i] = turn[i] - arrival[i];
        wait[i] = turn[i] - burst[i];
    }

    cout << "\nProgram No.\tArrival Time\tBurst Time\tWait Time\tTurnAround Time"
         << endl;
    for (int i = 0; i < n; i++)
    {
        cout << i + 1 << "\t\t" << arrival[i] << "\t\t"
             << burst[i] << "\t\t" << wait[i] << "\t\t" << turn[i] << endl;
    }
    for (int i = 0; i < n; i++)
    {
        avgWait += wait[i];
        avgTT += turn[i];
    }
    cout << "\nAverage Wait Time : " << (avgWait / n)
         << "\nAverage Turn Around Time : " << (avgTT / n);

    return 0;
}
```

Output:

```
PS C:\Users\Tanmay> cd "c:\Users\Tanmay\Desktop\5sem\OS\" ; if ($?) { g++ RoundRobinAlgo.cp

Enter the Time Quanta:2

Enter the No. of Processess:4

Enter Arrival Time of Processess : 0 1 2 3

Enter the Burst Time of Processess : 5 4 2 1

Program No.     Arrival Time    Burst Time      Wait Time       TurnAround Time
1               0               5               7               12
2               1               4               6               10
3               2               2               2               4
4               3               1               5               6

Average Wait Time : 5
Average Turn Around Time : 8
PS C:\Users\Tanmay\Desktop\5sem\OS>
```

Advantages:

1. It does not face any starvation issues or convoy effect.
2. Each process gets equal priority to the fair allocation of CPU.
3. It is easy to implement the CPU Scheduling algorithm.
4. Each new process is added to the end of the ready queue as the next process's arrival time is reached.
5. Each process is executed in circular order that shares a fixed time slot or quantum.
6. Every process gets an opportunity in the round-robin scheduling algorithm to reschedule after a given quantum period.

Disadvantages:

1. If the time quantum is lower, it takes more time on context switching between the processes.
2. It does not provide any special priority to execute the most important process.
3. The waiting time of a large process is higher due to the short time slot.
4. The performance of the algorithm depends on the time quantum.
5. The response time of the process is higher due to large slices to time quantum.

Least Recently Used Page Replacement Algorithm

Description:

The Least Recently Used (LRU) page replacement policy replaces the page that has not been used for the longest period of time. It is one of the algorithms that were made to approximate, if not better the efficiency of the optimal page replacement algorithm. The optimal algorithm assumes the entire reference string to be present at the time of allocation and replaces the page that will not be used for the longest period of time. LRU page replacement policy is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time.

Algorithm:

Let pages_num be the number of pages that memory can hold. Let frame array be the current set of pages in memory.

1.Start traversing the pages.

i) If frame array holds less pages than pages_num.

1) Insert page into the frame array one by one until the size of frame array reaches capacity or all page requests are processed.

2) Simultaneously maintain the recent occurred index of each page in a array called fcount.

3) Increment count.

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

iii) Else

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

2) Replace the found page with current page.

3) Increment count.

4) Update index of current page in fcount.

2.Calculate number of hits, hit ratio, miss ratio as follows,

Number of hits=pages_num-count.

Hit ratio= (Number of hits/pages_num) *100

Miss ratio= (Count/pages_num) *100

Implementation:

```
#include<iostream>
#include<iomanip>
using namespace std;
int main( )
{
    int pages_num, frame_num, page[50],i ,count=0;
    cout<<"\nEnter the number of pages:";
    cin>> pages_num;  //it will store the number of Pages
    cout<<"\nEnter the Reference String:";
    for(i=0;i< pages_num;i++)
    {
        cin>>page[i];
    }
    cout<<"\nEnter the number of frames:";
    cin>> frame_num;
    cout<<endl;
    int frame[frame_num],fcount[frame_num];
    for(i=0;i< frame_num;i++)
    {
        frame[i]=-1;
        fcount[i]=0; //it will keep the track of when the page was last used
    }
    i=0;
```

Implementation:

```
    while(i< pages_num)
    {
        int j=0,flag=0;
        while(j< frame_num)
        {
            if(page[i]==frame[j]) //it will check whether the page already exist in frames or not
            {
                flag=1;
                fcount[j]=i+1;
            }
            j++;
        }
        j=0;
        if(flag == 1)
        {
            while(j< frame_num)
            {
                cout<<"\t|"<<frame[j]<<"|";    //display the frame contents for current hit iteration
                j++;
            }
            cout<<endl;
        }
        j=0;
        if(flag==0)                //if page is not present
        {
            int min=0,k=0;
            while(k<frame_num-1)
            {
                if(fcount[min]>fcount[k+1]) //It will calculate the page which is least recently used
                    min=k+1;
                k++;
            }
            frame[min]=page[i];
            fcount[min]=i+1;  //Increasing the time
            count++;           //it will count the total Page Fault
            while(j< frame_num)
            {
                cout<<"\t|"<<frame[j]<<"|";    //display the frame contents for current miss iteration
                j++;
            }
            cout<<endl;
        }
        i++;
    }
```

```
    int hit;
    float r_miss, r_hit;
    hit = pages_num-count;
    r_miss = ((count/(float)pages_num)*100);
    r_hit = ((hit/(float)pages_num)*100);
    cout<<"\nPage Fault: "<<count<<endl;
    cout<<"Number of Hits: "<<hit<<endl;
    cout<<"Miss Ratio: "<<fixed<<setprecision(3)<<r_miss<<"%"<<endl;
    cout<<"Hit Ratio: "<<fixed<<setprecision(3)<<r_hit<<"%"<<endl;
    return 0;
}
```

Output:

```
Enter the number of pages:16

Enter the Reference String:3 3 5 4 7 1 5 5 1 4 3 7 6 3 4 1

Enter the number of frames:4

        |3|     |-1|    |-1|    |-1|
        |3|     |-1|    |-1|    |-1|
        |3|     |5|     |-1|    |-1|
        |3|     |5|     |4|     |-1|
        |3|     |5|     |4|     |7|
        |1|     |5|     |4|     |7|
        |1|     |5|     |4|     |7|
        |1|     |5|     |4|     |7|
        |1|     |5|     |4|     |7|
        |1|     |5|     |4|     |7|
        |1|     |5|     |4|     |3|
        |1|     |7|     |4|     |3|
        |6|     |7|     |4|     |3|
        |6|     |7|     |4|     |3|
        |6|     |7|     |4|     |3|
        |6|     |1|     |4|     |3|

Page Fault: 9
Number of Hits: 7
Miss Ratio: 56.250%
Hit Ratio: 43.750%
PS D:\3rd Year\OS tut final>
```

Advantages:

1) LRU doesn't suffer from Belady's Anomaly.

2) The page in the main memory that hasn't been used in the longest will be chosen for replacement.

3) It generally gives fewer page faults than any other page replacement algorithm. So, LRU is the most utilized method.

4) It helps in the full analysis.

Disadvantages:

1) It is not easy to implement LRU because it requires hardware assistance.

2) It is expensive and more complex.

3) It needs an additional data structure.

THANK YOU !