The National Institute Of Engineering, Mysuru

Logo, company name

Description automatically generated

**Operating Systems Tutorial Report**

Program Implementation of

Round Robin process scheduling and

Least Recently Used page replacement

algorithms using C++ programming

Submitted to

**Dr. Jayasri B S**

**Professor & Dean (EAB)**

Department Of Computer Science and Engineering

Submitted by

**Shreesh Kulkarni (4NI19CS103)**

**Santhrupth S (4NI19CS098)**

**Process Scheduling**

* **What is Process scheduling?**

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

* **Round Robin Scheduling:**

Round Robin scheduling algorithm is one of the most popular scheduling algorithm which can actually be implemented in most of the operating systems. This is the pre-emptive version of first come first serve scheduling.

The Algorithm focuses on Time Sharing. In this algorithm, every process gets executed in a cyclic way. A certain time slice is defined in the system which is called time quantum.

Each process present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will terminate else the process will go back to the ready queue and waits for the next turn to complete the execution.

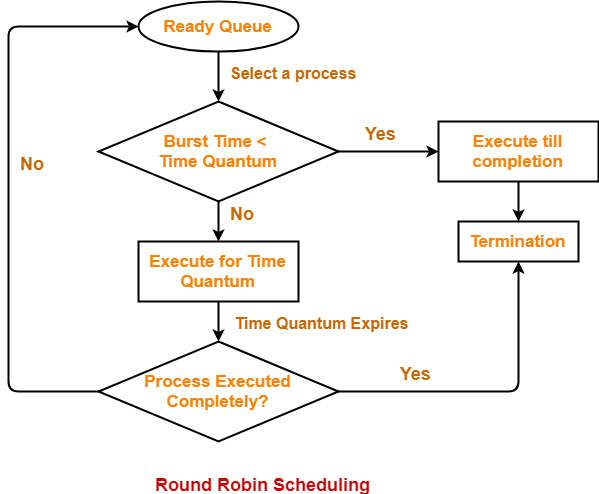
* **Some important formulae:**

Turn around time (TAT) = Completion Time (CT) – Arrival Time (AT)

Wait Time (WT) = Turn around time (TAT) – Burst Time (BT)

Response Time (RT) = Start Time – Arrival time (AT)

* **Advantages of Round robin algorithm:**
* There is fairness since every process gets equal share of CPU.
* The newly created process is added to end of ready queue.
* A round-robin scheduler generally employs time-sharing, giving each job a time slot or quantum.
* While performing a round-robin scheduling, a particular time quantum is allotted to different jobs.
* Each process get a chance to reschedule after a particular quantum time in this scheduling.
* No convoy effect is seen here.
* **Disadvantages of Round robin algorithm:**
* There is Larger waiting time and Response time.
* There is Low throughput.
* There is Context Switches.
* Gantt chart seems to come too big (if quantum time is less for scheduling. For Example: 1 ms for big scheduling.)
* Time consuming scheduling for small quantum.
* **Flow Chart:**



* **Algorithm:**

**Step 1:** Input Number of processes n and time quantum q.

**Step 2:** Input arrival time and burst time of processes P1 to PN.

**Step 3:** Initialize remaining burst time rt as the burst time, total time tottime as the sum of all burst times, current time ctime as zero

**Step 4:** Push at the back of ready queue, all processes whose arrival time is less than the current time and it is not already present in the ready queue

**Step 5:** Start executing task from the front of the queue, assigning CPU access for a maximum of time quantum.

- If remaining burst time is less than or equal to time quantum add the remaining time to current time and set completion time ct as current time.

- If remaining burst time is greater than time quantum, add q to current time and push the process to the back of the queue after the already arrived processes.

Repeat 4 and 5 until current time is equal to total time.

**Step 6:** Calculate TAT, WT and RT using above mentioned formulae

**Step 7:** Output the results

* **Code:**

#include<bits/stdc++.h>

#include<vector>

using namespace std;

struct process

{

int at,bt,ct,st,rt,tat,wt;

};

int main()

{

vector<int>rq;

int i,n,ctime=0,tottime=0,q;

float avgtat=0,avgwt=0,avgrt=0;

cout<<"Enter number of processes: ";

cin>>n;

cout<<"Enter quantum time: ";

cin>>q;

process arr[n];

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

{

cout<<"Enter arrival time and burst time of P"<<i+1<<": ";

cin>>arr[i].at>>arr[i].bt;

tottime+=arr[i].bt;

arr[i].rt=arr[i].bt;

arr[i].st=-1;

}

int j=0;

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

{

if(arr[i].at<=ctime&&arr[i].rt>0&&arr[i].st==-1)

{

arr[i].st=-2;

rq.push\_back(i);

}

}

while(ctime<tottime)

{

cout<<ctime<<" P"<<rq[j]+1<<" ";

if(arr[rq[j]].st==-2)

{

arr[rq[j]].st=ctime;

}

if(arr[rq[j]].rt<=q)

{

ctime+=arr[rq[j]].rt;

arr[rq[j]].rt=0;

arr[rq[j]].ct=ctime;

}

else

{

ctime+=q;

arr[rq[j]].rt-=q;

}

cout<<ctime<<endl;

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

{

if(arr[i].at<=ctime&&arr[i].rt>0&&arr[i].st==-1)

{

arr[i].st=-2;

rq.push\_back(i);

}

}

if(arr[rq[j]].rt>0)

{

rq.push\_back(rq[j]);

}

j++;

}

cout<<endl<<endl<<"Process AT\tBT\tCT\tTAT\tWT\tRT"<<endl;

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

{

arr[i].tat=arr[i].ct-arr[i].at;

arr[i].wt=arr[i].tat-arr[i].bt;

arr[i].rt=arr[i].st-arr[i].at;

cout<<"P"<<i+1<<"\t"<<arr[i].at<<"\t"<<arr[i].bt<<"\t"<<arr[i].ct<<"\t"<<arr[i].tat<<"\t"<<arr[i].wt<<"\t"<<arr[i].rt<<endl;

avgtat+=arr[i].tat;

avgwt+=arr[i].wt;

avgrt+=arr[i].rt;

}

cout<<endl<<"Average Turn Around time= "<<(float)avgtat/n;

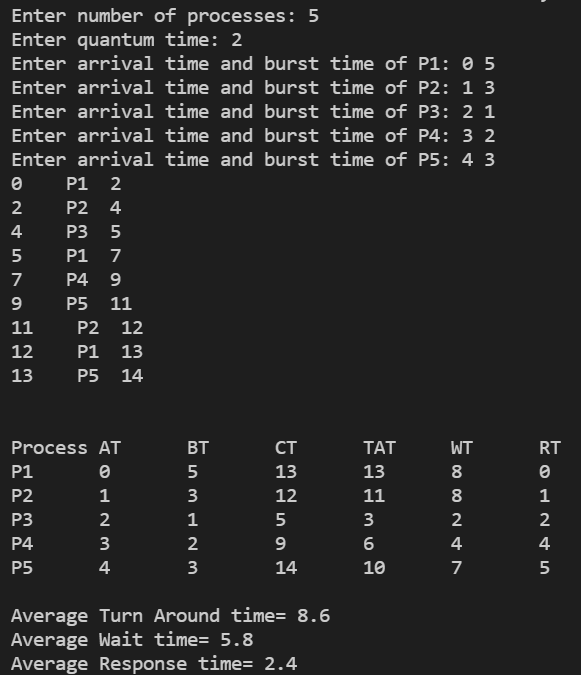
cout<<endl<<"Average Wait time= "<<(float)avgwt/n;

cout<<endl<<"Average Response time= "<<(float)avgrt/n;

return 0;

}

* **Input and output:**



**Page Replacement**

* **What is Page replacement Algorithm?**

Page Replacement Algorithm decides which page to remove, also called swap out when a new page needs to be loaded into the main memory. Page Replacement happens when a requested page is not present in the main memory and the available space is not sufficient for allocation to the requested page.

When the requested page is already stored in the main memory it is called a hit. On the other hand, when the page has to be loaded from disk, it is a miss or a page fault.

* **Least Recently Used (LRU)**

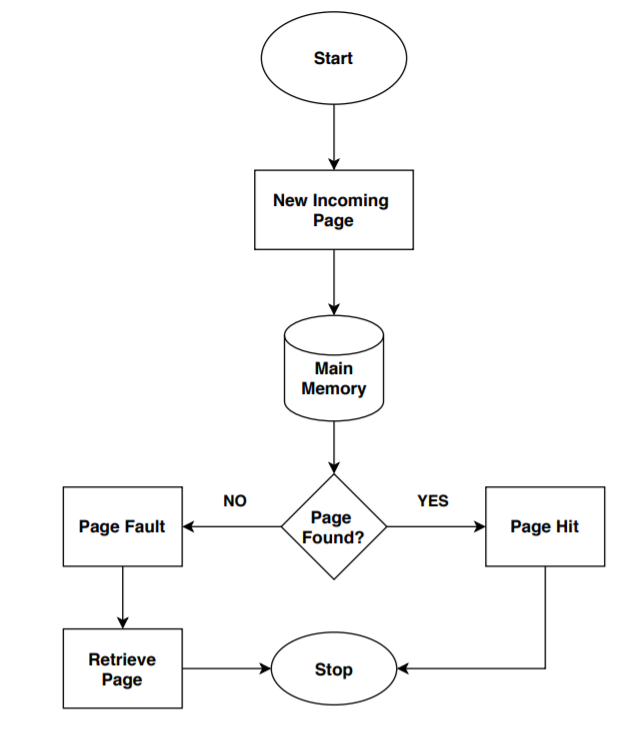
Least Recently Used page replacement algorithm keeps track of page usage over a short period of time. It works on the idea that the pages that have been most heavily used in the past are most likely to be used heavily in the future too.

In LRU, whenever page replacement happens, the page which has not been used for the longest amount of time is replaced.

Hit ratio = Number of Hits / length of reference string

Miss ratio= Number of page faults / length of reference string

* **Advantages of LRU**
* 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.
* **Disadvantages of LRU**
* It requires additional Data Structure to be implemented.
* Hardware assistance is high.
* **Flow Chart**



* **Algorithm**

**Step 1:** Input the length of reference string, the reference string and the number of frames.

**Step 2:** Create array to keep track of the frames and the pages in them.

**Step 3:** Load next page request from the reference string and check the frames for the requested page

* If page is found, increase hit count
* If page is not found, empty frame is available, load the requested page in empty frame, increase page fault count.
* If No empty frames available and page not found in the frames, check the reference string preceding the current request for n unique requests, the oldest request is the current least recently used page. Find and replace the page with the required page, increase page fault count.

Repeat the process by traversing through the entire reference string.

**Step 4:** Calculate hit and miss ratio and Output appropriate results

* **Code**

#include <stdlib.h>

#include <stdio.h>

int main(int argc, char const \*argv[])

{

int length;

int num\_frames;

printf("Enter the length of the reference string : ");

scanf("%d", &length);

printf("Enter the Reference string : ");

int reference\_str[length];

for (int i = 0; i < length; i++)

{

scanf("%d", &reference\_str[i]);

}

printf("Enter the Number of Frames : ");

scanf("%d", &num\_frames);

int memory[num\_frames];

for (int i = 0; i < num\_frames; i++)

{

memory[i] = -1;

}

// LRU

int fault\_count = 0;

printf("\n\n The Page Replacement Process Is : \n");

for (int i = 0; i < length; i++)

{

// Get Current Requested Page

int cur\_page = reference\_str[i];

int page\_found = 0;

// Look for page in memory

for(int j = 0; j < num\_frames; j++)

{

if(memory[j] == cur\_page)

{

page\_found = 1;

break;

}

}

// If page is not found

if(!page\_found)

{

// Increase Fault Count

fault\_count++;

// Cached status : to check for any empty slot

int is\_cached = 0;

// <-----------Cache the page --------------->

for(int j = 0; j < num\_frames; j++)

{

if(memory[j] == -1)

{

memory[j] = cur\_page;

break;

}

}

// If no empty slot is found then remove the LRU page

if(!is\_cached)

{

int check\_count = 0;

int check\_status[num\_frames];

for(int m = 0; m < num\_frames; m++)

{

check\_status[m] = 0;

}

for(int cur = i-1; cur >= 0; cur--)

{

int val = reference\_str[cur];

for(int n= 0; n < num\_frames; n++)

{

if(val == memory[n])

{

check\_status[n] = 1;

check\_count++;

break;

}

}

if(check\_count == num\_frames-1)

break;

}

// Put the page where check status = 0

for(int cur = 0; cur < num\_frames; cur++)

{

if(check\_status[cur] == 0)

{

memory[cur] = cur\_page;

break;

}

}

}

// Print the current Memory snap

for(int j = 0; j < num\_frames; j++)

{

printf("%d\t", memory[j]);

}

printf("\tPage Fault Num : %d\n", fault\_count);

}

// When page is present in Cache

else

{

// Print the current Memory snap

for(int j = 0; j < num\_frames; j++)

{

printf("%d\t", memory[j]);

}

printf("\n");

}

}

printf("\n \* Total Page Faults : %d\n", fault\_count);

printf(" \* Miss Ration : %d/%d = %.3f\n", fault\_count, length, (float(fault\_count)/float(length)));

printf(" \* Hit Ration : %d/%d = %.3f\n", (length - fault\_count), length, (float(length-fault\_count)/float(length)));

/\* printf("\n \* Hit Ration : %d/%d = %d\n", fault\_count); \*/

return 0;

}

* **Input and output:**

