Ex.No. 01 First-Come-First-Served (FCFS)

AIM: To implement First-come-First-served (FCFS)

INTRODUCTION: FCFS is a simplest scheduling algorithm. In this process the process that comes first will be executed first and next process starts only after the previous get fully executed.

TERMINOLOGIES USED:

- Arrival Time: The time at which the process arrives in the ready queue.
- Completion Time: The time at which the process complete its execution.
- Turn Around time = Completion time - Arrival time.
- Waiting time = (Turn Around time - Burst time) if all the process arrive in 0th time then waiting time of 1st process will be 0; later Waiting time[i] = Waitingtime[i-1] + Bursttime[i-1].

ALGORITHM:

1. Input the process along with their burst time (bt).
2. Find the waiting time (wt) for all processes. As first process that comes need not to wait so waiting time for the process 1 will be 0. i.e; wt[0] = 0; Waiting time for all other processes is. i.e; for process i -> wt[i] = bt[i-1] + wt[i-1]
3. Find turnaround time = waiting time + burst time for all processes.
4. Find average waiting time = total waiting time / no of processes.
5. Find average turnaround time = total turn around time / no of processes.

CODE:

```
#include<stdio.h>
void findWaitingTime(int processes[], int n, int bt[],
int wt[]){
    wt[0] = 0;
    for (int i = 1; i<n;i++)
        wt[i] = bt[i-1] + wt[i-1];
}
```

```
void findTurnAroundTime(int processes[], int n, int bt[],
int wt[], int tat[]){
    for(int i=0; i<n; i++)
        tat[i] = bt[i] + wt[i];
}
void findavgTime(int processes[], int n, int bt[]){
    int wt[n], tat[n], total_wt=0, total_tat=0;
    findWaitingTime(processes, n, bt, wt);
    findTurnAroundTime(processes, n, bt, wt, tat);
    printf("Processes  Burst time  Waiting time
Turn around time\n");
    for(int i=0; i<n; i++){
        total_wt += wt[i];
        total_tat += tat[i];
        printf("    %d", (i+1));
        printf("          %d", wt[i]);
        printf("          %d\n", tat[i]);
    }
    float avg_wt = (float) total_wt / (float)n;
    float avg_tat = (float) total_tat / (float) n;
    printf("Average waiting time = %f", avg_wt);
    printf("\n");
    printf("Average turn around time = %f \n", avg_tat);
}
int main(){
    int n;
    printf("Enter the number of processes: ");
    scanf("%d",&n);
    int processes[n];
    int burst_time[n];
    for(i=0; i<n; i++){
        processes[i] = i+1;
        printf("Enter burst time for processes %d:",
        i+1);
        scanf("%d", &burst_time[i]);
    }
    findavgTime(processes, n, burst_time);
    return 0;
}
```

Output:

Enter the number of processes : 3
Enter burst time for process 1 : 10
Enter burst time for process 2 : 3
Enter burst time for process 3 : 12

Processes	Burst time	Waiting time	Turn around time
1	10	0	10
2	3	10	13
3	12	13	25

Average waiting time = 7.666667
Average turn around time = 16.000000

Result: Thus the First come - First serve (FCFS) has implemented in C programming language and output has been verified.

o/p Verified
20/8/2014.

EX.NO:02 SHORTEST JOB FIRST (SJS)

AIM:

To implement Shortest-Job-First-Scheduling Algorithms.

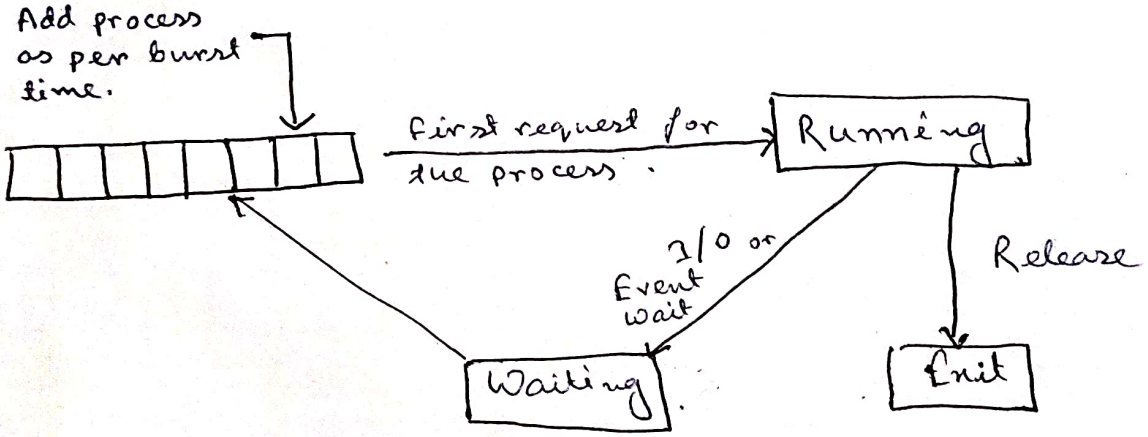
Introduction:

Shortest-Job-first Scheduling (SJFS) is a scheduling policy that select the waiting process with the smallest execution time to execute next.

Types of SJFS:

(i) Pre-emptive SJFS: It is a type of scheduling algoritum in which job are inserted into ready queue as soon as they arive. The process having the shortest burst time start to get execute first, even if the shortest burst time arrives the current burst is remove from the exicution process and the job having the shortest burst time gets allocated in the CPU cycle.

(ii) Non-Preemptive SJFS: If one process is allocated for the CPU, the process sticks to the CPU untill the process becomes in a waiting state or get executed.



TERMINOLOGIES USED:

- Completion Time: Time at which process completes its execution.
- Turn Around Time: The diffrence between completion time and arrival time.

  TAT = CT - AT
- Waiting Time (W.T.): The diffrence between turn around time and burst time.

  WT = TAT - BT

ALGORITHM:

1. Short all the process according to the arrival time.
2. Then select the process that has minimum arival time and minimum Burst time.
3. After Completion of the process make a pool of the process that arrives afterward till the completion of the previous process and select the process among the pool which is having minimum Burst time.
4. Calculate the waiting time: set the waiting time of the first process to 0.
   For each subsequent process, add the Burst time of ~~each~~ all previous processes to calculate the waiting time. Adjust for negative waiting time by setting them to 0.
5. Calculate turn around time = Burst Time + Waiting Time for each process.
6. Calculate Average waiting time and average turnaround time.
7. Output Result: Print the details of each process Process ID, Arival Time, Burst Time, Waiting Time, and Turnaround Time.
   Print the average waiting time and turnaround time.

CODE:

```
# include <stdio.h>
int main (){
    int A[100][5];
    int i, j, n, total=0, index, temp;
    float avg-wt, avg-tat;
    printf(" Enter the number of Processes: ");
    scanf("%d", &n);
    Printf("Enter arival time and Burst time: \n");
```

```
for(i=0; i<n; i++){
    printf("P%d\n", i+1);
    printf("Arival Time: ");
    scanf("%d", &A[i][1]);
    printf("Burst Time: ");
    scanf("%d", &A[i][2]);
    A[i][0] = i+1;
}
for(i=0; i<n; i++){
    index = i;
    for(j=i+1; j<n; j++){
        if(A[j][1] < A[index][1]){
            index = j;
        }
    }
    for(j=0; j<5; j++){
        temp = A[i][j];
        A[i][j] = A[index][j];
        A[index][j] = temp;
    }
}
A[0][3] = 0;
total = 0;
for(i=1; i<n; i++){
    A[i][3] = 0;
    for(j=0; j<i; j++){
        A[i][3] += A[j][2];
    }
    A[i][3] -= A[i][1];
    if(A[i][3] <0) A[i][3] = 0;
    total += A[i][3];
}
avg_wt = (float)total/n;
total = 0;
printf("P\tAT\tBT\tWT\tTAT\n");
for(i=0; i<n; i++){
    A[i][4] = A[i][2] + A[i][3];
    total += A[i][4];
    printf("P%d\t%d\t%d\t%d\t%d\n",
    A[i][0], A[i][1], A[i][2], A[i][3], A[i][4]);
}
avg_tat = (float)total/n;
```

```
    printf("Average Waiting Time = %.2f \n",avg_wt);
    printf("Average Turnaround Time = %.2f \n", avg_tat);
    return 0;
}
```

OUTPUT

```
Enter number of processes: 4
Enter arival time and burst time:
P1:
Arival time: 1
Burst time: 4
P2:
Arival time: 2
Burst time: 5
P3:
Arival time: 3
Burst time: 6
P4:
Arival time: 4
Burst time: 7
```

P	AT	BT	WT	TAT
P1	1	4	0	4
P2	2	5	2	7
P3	3	6	6	12
P4	4	7	11	18

Average Waiting Time = 4.75
Average Turnaround Time = 10.25

RESULT:

Thus the Shortest Job first scheduling algorithm is implemented succesfully and the output has been verified.

OV

8/11/24

EX.NO.: 03 PRIORITY SCHEDULING

AIM: To implement priority scheduling using C.

ALGORITHM:

1. Input:
   Number of Process
   For each process
   - Burst Time: The exicution time of each Process
   - Priority: A high priority value indicate that a process should be exicuted earlier.
2. Process ID: Each process is assigned a unique ID for ~~easy~~ easy identification and display.
3. Shorting algo: Process are short in decending order of priority, the highest-pririty process is placed in the list first.
4. Calculation:
   - Wait time: Time each process wait before starting.
   - Turnaround Time: Calculate as wait time + Burst Time for each process.
   - Average:
     Averag waiting time = Total waiting time/ Number of process
     Average turnaround time = Total turn around time / Number of process.
5. Output:
   - Display the scheduled orger of process exicution with start and end time.
   - Display each process burst time, wait time and turnaround time.
   - Display the average waiting time and average turnaround time.

CODE:

```
#include <stdio.h>
void swap (int *a, int *b)
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main(){
    int n;
    printf("Enter Number of Processes: ");
    scanf("%d %d", &b[i], &p[i]);
    index[i] = i+1;
}   scanf("%d", &n);
    int b[n], p[n], index[n];
    for(int i=0; i<n; i++){
        printf("Enter Burst Time and Priority value
        for process %d: ", i+1);
        scanf("%d %d", &b[i], &p[i]);
        index[i] = i+1;
    }
    for(int i=0; i<n; i++){
        int max_priority = p[i];
        int max_index = i;
        for(int j=i; j<n; j++){
            if(p[j] > max_priority){
                max_priority = p[j];
                max_index = j;
            }
        }
        swap(&p[i], &p[max_index]);
        swap(&b[i], &b[max_index]);
        swap(&index[i], &index[max_index]);
    }
    int t=0;
    printf("Order of process execution is: \n");
    for(int i=0; i<n; i++){
        printf("P%d is executed from %d to %d \n",
        index[i], t, t+b[i]);
        t += b[i];
    }
    printf("\n");
```

```
printf("Process ID  Burst Time  Wait Time  Turnarou
Time\n");
int wait_time = 0
int total_wait_time = 0
int total_turnaround_time = 0;
for (int i = 0 ; i<n; i++){
    int turnaround_time = wait_time + b[i] ;
    printf("P%d      %d      %d      %d\n", index[i],
    b[i], wait_time, turnaround_time);
    total_wait_time += wait_time;
    total_turnaround_time += turnaround_time;
    wait_time += b[i];
}
double avg_wait_time = (double) total_wait_time/n;
double avg_turnaround_time = (double) total_turnarou-
ndtime / n ;
printf("\n Average Waiting Time: %.2f \n",
avg_wait_time);
printf(" Average Turnaround Time: %.2f \n",
avg_turnaround_time);
return 0 ;
}
```

Output:

```
Enter the number of Processes : 3
Enter Burst Time and Priority value of Processes 1 : 10 2
Enter Burst Time and Priority Value of Processes 2 : 5 0
Enter Burst Time and Priority Value of Processes 3 : 8 1
Order of Process Execution is:
P1 is executed from 0 to 10:
P2 is executed from 10 to 18
P3 is executed from 18 to 23
```

Process ID	Burst Time	Wait Time	Turnaround Time
P1	10	0	10
P2	8	10	18
P3	5	18	23

Average Waiting Time : 9.33
Average Turnaround Time : 17.00

Result : Thus the Priority scheduling is implemented successfully and output has been verified

8/11/24

EX.NO.: 04 ROUND ROBIN SCHEDULING

AIM:

To implement Round Robin Scheduling Algorithm using C

ALGORITHM:

1. Set the burst time of each process
   Set total time to 0 and complete process to 0
2. Process loop: For each process in order
   - check if process can complete: if remaining burst time ≤ quantum, add remaining burst time to total time, mark the process as complete, and calculate waiting time and turnaround time
   - If process need more time: substract the quantum forom remaning burst time and add quantum to total time.
   - Repeat until all process to complete.
3. Calculate Average: Compute average waiting time and turnaround time.
4. Output: Print each process burst time, turnaround time, waiting time, and the average.

CODE:

```
#include<stdio.h>

void main()
{
    int i, num_processes, sum=0, completed=0, quantum,
    waiting_time=0, turnaround_time=0;
    int arival_time[10], burst_time[10], remaining_bt[10];
    float avg_waiting_time, avg_turnaround_time;
    printf("Total number of system in the system:");
    scanf("%d",&num_processes);
    complete = num_processes;
    for(i=0; i< num_processes; i++)
    {
        printf("\n Enter the Arival and Burst time of Process
        [%d]\n", i+1);
        printf("Arrival time: ");
        scanf("%d", &arrival_time[i]);
```

```
    printf ("Burst Time: ");
    scanf ("%d", &burst_time[i]);
    remaining_bt[i] = burst_time[i];
}
printf ("Enter time quantum: ");
scanf ("%d", &quantum);
printf ("\n Processes No \t Burst Time \t Turnaround Time
 \t Waiting Time \n");
for (sum = 0, i = 0 ; completed != 0;)
{
    if (remaining_bt[i] <= quantum && remaining_bt[i
     > 0)
    {
        sum += remain_bt[i];
        remaining_bt[i] = 0;
        complete --;
        printf ("Process[%d] \t %d \t \t %d \t \t
        %d \n", i+1, burst_time[i], sum - arival_time[
        sum - arrival_time[i] - burst_time[i]);
        waiting_time += sum - arrival_time[i] -
        burst_time[i];
        turnaround_time += sum - arrival_time[i];
    }
    else if (remaing_bt[i] > 0)
    {
        remaining_bt[i] -= quantum;
        sum += quantum;
    }
    if (i == num_processes - 1)
    {
        i = 0
    }
    else if (arrival_time[i+1] <= sum)
    {
        i+1;
    }
    else
    {
        i = 0;
    }
}
avg_waiting_time = (float) waiting_time /
num_processes;
```

```
    avg_turnaround_time = (float) turnaround_time /
    num_processes
    printf("\nAverage Turnaround Time: %f",
    avg_turnaround_time);
    printf("\nAverage Waiting Time: %f", avg_
    waiting_time);
}
```

OUTPUT :

```
Total number of processes in the system: 4
Enter the Arival and Burst time of the Process[1]
Arival time: 0
Burst time: 8
Enter the Arival and Burst time of the Process[2]
Arival time: 1
Burst time: 5
Enter the Arival and Burst time of the process[3]
Arival time: 2
Burst time: 10
Enter the Arival and Burst time of the Process[4]
Arival time: 3
Burst time: 11
Enter the time Quantum: 6
```

Process No	Burst Time	Turnaround Time	Waiting Time
Process[2]	5	10	5
Proc[1]	8	25	17
Process[3]	10	27	17
Process[4]	11	31	20

```
Average Turnaround Time: 23.25
Average Waiting Time: 14.75
```

RESULT:
Thus the Round Robin Scheduling algorithm has been implemented successfully and output has been verified.

8/11/24

Output:

Enter the number of available memory block: 5
Enter the size of each memory block:
size of block1 : 100
size of block2 : 500
size of block3 : 200
size of block 4 : 300
size of block 5 : 600
Enter the number of process : 4
Enter the size of each process :
size of process 1 : 212
size of process2 : 417
size of process 3 : 112
size of process 4 : 426
Process P of (size) is allocated to block
Process 1 of 212 --> Block 4
Process 2 of 417 --> Block 2
Process 3 of 112 --> Block 3
Process 4 of 426 --> block 5

OJ

EX.NO:05 MEMORY MANAGEMENT USING BEST FIT

AIM:

To implement paging technique for memory management using Best-fit

ALGORITHM

1. Input the number and size of memory block.
2. Input the number and size of the processes.
3. Initialize allocation array (alloc) to -1 (meaning not allocating initially).
4. For each process:
   - Calculate available space in each block after placing the procces (if the block is large enough).
   - Find the block with the smallest available space that fit the procces.
   - Allocate the procces to that block, if possible, and mark the block as used.
5. Print the allocation result.

RESULT: Thus the paging technique for memory management using Best fit has been implemented successfully.

Output

```
Enter the number of available memory block: 5
Enter the size of each memory block:
Size of block 1 : 100
Size of block 2 : 500
Size of block 3 : 200
Size of block 4 : 300
Size of block 5 : 600
Enter the number of process : 4
Enter the size of each process:
Size of process 1 : 212
Size of process 2 : 417
Size of process 3 : 112
Size of process 4 : 426
Process allocation :
Process 1 of size 212 : --> Block 2
Process 2 of size 417 --> Block 5
Process 3 of size 112 --> Block 2
Process 4 of size 426 --> is not allocated
```

EX. NO.: 06 MEMORY MANAGEMENT USING FIRST FIT

AIM:

To implement paging technique for Memory Management using first-fit.

ALGORITHM:

1. Input the number and size of memory block
2. Input the number and size of processes.
3. Initialize the allocation array (alloc) to -1 for each process.
4. For each process:
   - Check each memory block in sequence
   - If a block can fit the process, ~~showing which process~~ allocate it by making the block as unavailable.
5. Print the allocating result, showing which process was assigned to which block, or if a process could not be allocated.

RESULT:

Thus the implementation for paging technique for memory management using first-fit has been implemented using C and the output has been implemented.

Output:

Enter the number of available memory block: 5
Enter the size of each memory block:
Size of block1 : 100
Size of block2 : 500
Size of block 3 : 200
Size of block 4 : 300
Size of block 5 : 600

Enter the number of procesn : 4
Enter the size of each procesn.
Size of ~~each~~ procesn 1 : 212
Size of procesn 2 : 417
Size of Procesn 3 : 112
Size of Procesn 4 : 426

Procesn 0 of {size} is allocated to block
Procesn 1 of 212 --> Block 5
Procesn 2 of 417 --> Block 2
Procesn 3 of 112 --> Block 5
Procesn 4 of 426 --> it is not allocated.

Ov

EX.NO.:07 Memory Management using Worst-Fit

AIM:

To implement paging technique for memory management using Worst-Fit

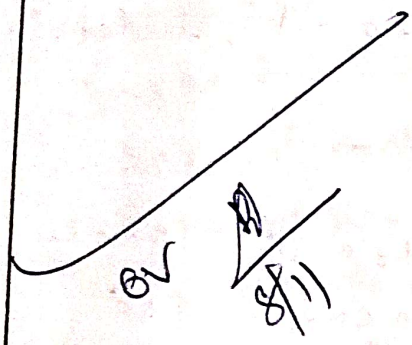
ALGORITHM:

1. Input the number of memory block and their sizes.
2. Input the number and size of the processes.
3. Initialize the alloc array to -1, indicating no process has been allocated.
4. For each process:
   - Calculate the remaining size of each block that can fit the process.
   - Find the block with maximum available space (worst fit) for the current process.
   - Allocate the process to this block if the remaining size is greater than 50.
5. Print which processes were allocated to which block, or indicate if a process couldn't be allocated.

Output:

Enter the value of n (number of item): 3
Enter an item : 2
Enter an item : 5
Enter an item : 9

Consumed item = 2
Consumed item = 5
Consumed item = 9



EX.NO:.08 PRODUCER - CONSUMER PROBLEM

AIM:

Implement Producer - Consumer problem using semaphore.

ALGORITHM:

1. The semaphore mutex, full and empty are initialized
2. In the case of producer process
   (i) Produce an item is to temporary variable
   (ii) If there is empty space in the buffer check the mutex value for enter into the critical section.
   (iii) If the mutex value is 0, allow the producer to add value in temporary variable to the buffer.
3. In the case of consumber processes.
   (i) It should wait if the buffer is empty.
   (ii) If there ~~buffer~~ is any item in the buffer check for mutex value, if the mutex == 0, remove item from buffer
   (iii) Signal the mutex value and reduce the empty value by 1.
   (iv) Consume the item.
4. Print the result.

RESULT:

Thus the implementation of producer-consumer problem using semaphore has been implemented successfully and the output has been verified.

Output :

Enter the number of processes and resources (m and n): 3 3

Enter the Claim (Maximum Need Matrix:

7 5 3
3 2 2
9 0 2

Enter the Allocated Matrix:

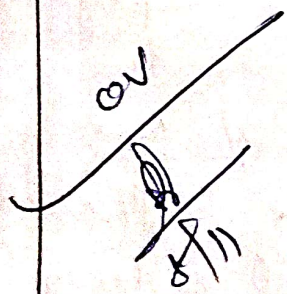
2 1 3
2 1 1
3 2 2

Enter the available resources:

3 3 2

The need matrix is in a Safe state

Safe sequence : P1 P0 P2



EX.NO.:09 BANKERS ALGORITHM FOR DEADLOCK AVOID

AIM:

To implement Bankers algorithm for Deadlock Avoidance

ALGORITHM:

1. Input:
   - Claim matrix (maximum resource need for each process).
   - Allocation matrix (resource currently allocated to each process).
   - Available resource (resource available in the system).
2. Need matrix calculation:
   Need[i][j] = Claim[i][j] - Allocation[i][j]
3. Seafty check (Bankers algorithm):
   - Start with available resources.
   - For each process, check if its resource need can be meet with available resources.
   - If a process can proceed:
     - Release its allocated resources.
     - Add it to the safe sequence.
     - Mark it as finished.
   - Repeat until either all process are finished or no process can proceed.
4. Output:
   - If all process finish, print the safe sequence.
   - If any process cannot proceed, the system is in unsafe state.

RESULT:

Thus the implementation of the Bankers algorithm for deadlock avoidance has been implemented successfully using C and the output has been verified.

Output :

Enter the number of page : 3
Enter the input string : 1 2 3 1 2 3 4
The number of page fault are : 4
The number of page hits are : 3

EX NO.: 11 PAGE REPLACEMENT ALGO : FIFO

AIM:

To implement page replacement algorithm FIFO (First in First Out)

ALGORITHM :

1. Input :
   - n = number of page (frame size).
   - rent = a sequence of page references (as a string)
2. Processing:
   - For each page reference
     - Check if the page is already in memory (page hit).
     - If not, it's a page fault. Add the page to the memory frame.
     - If the frame is full, use the critical method (count % n) to replace the oldest page.
3. Output:
   - Total number of page faults.
   - Total number of page hits.

RESULT: Thus the implementation of page replacement algorithm FIFO (First in First Out) has been implemented succesfully and the output has been verified.

Output:

Enter the input string : 7012030423032120170

Enter the number of page : 3

The number of page fault are : 10

The number of page hit are : 10

OV 8/11

EX. NO.:12 PAGE REPLACEMENT ALGORITHM: LRU

AIM:

To implement ~~least~~ page replacement algorithm least recently used (LRU).

ALGORITHM:

1. Initialize page frames with the first n pages and mark this as faults.
2. For each subsequent page request:
   - Check if it is already in memory (hit).
     if yes increase hit.
   - if it is missed:
     - Calculate the most recent occurrence of each page in memory.
     - Replace the last recently used page with the new page.
     - Increment the fault count (count).
3. Output: the total page fault and hit.

Result: Thus the implementation of page replacement algorithm LRU has been implemented and the output has been verified.

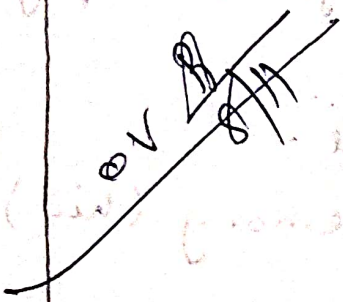
Output:

Enter the no. of page : 7

Enter the reference string : 1 2 3 1 2 3 4

Total page hits : 3

Total page Fault : 4



EX.NO.:13 PAGE REPLACEMENT ALGORITHM: LFU

AIM:

To implement page replacement algorithm LFU (least Frequently Used)

Algorithm:

1. Input:
   The program first ask for the number of pages and the reference string. and number of Frame available in the memory.
2. Initial Setup:
   The first page from the reference string is loaded into memory, causing a page fault.
3. Page Access:
   - For each subsequent page in the reference string, the program check if the page is already in memory.
   - If the page is found in memory, it is a page hit
   - If the page is not found, it is a page fault.
     - If there is space in memory, the page is added to memory.
     - If there is no space, the program chose the optimal page to replace (the page that will be used farthest in the future) and replace it.
4. Final Output: After processing all page in the reference string, the program output the total number of page hit and page fault.

Result: Thus the implementation of page replacement has been implemented successfully and the output has been verified.