**CSE 316**

**OPERATING SYSTEMS**



Assignment Done

By

Name : - Ansh Yadav

Email : - anshrao1395@gmail.com

Section : - K18TG

Roll No : - 23

Reg. No. : - 11803306

**School of Computer Science and Engineering**

**Faculty of Technology And Sciences**

**Name of the faculty member: - Shivali Chopra**

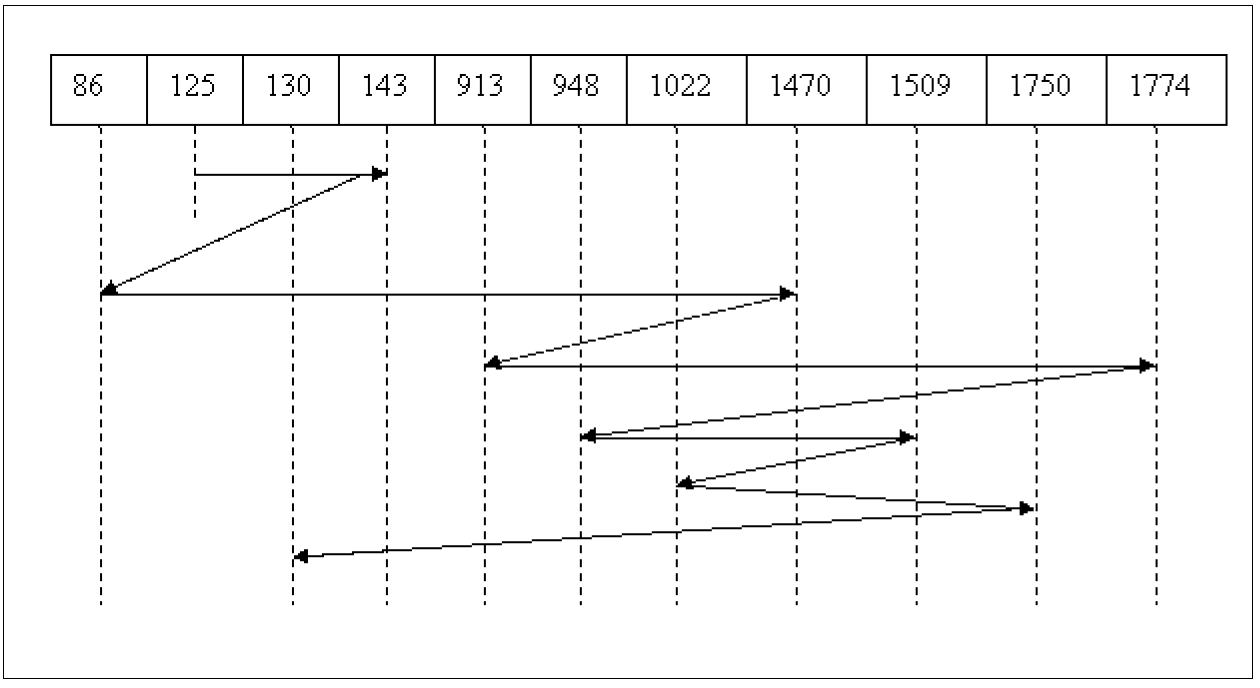
**Q1. Write a C program to solve the following problem: Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO**

**order,is:**

**86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130**

**Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the FCFS disk-scheduling algorithms?**

FCFS is the simplest [disk](https://www.geeksforgeeks.org/disk-scheduling-algorithms/) scheduling algorithm. As the name suggests, this algorithm entertains requests in the order they arrive in the disk queue. The algorithm looks very fair and there is no starvation (all requests are serviced sequentially) but generally, it does not provide the fastest service.



ALGORITHM

• **Step-1:** Create an Array named Queue to store processes in order.

• **Step-2:** Take input of Current header and Previous Requested.

• **Step-3:** For all values compare two values and subtract bigger one from smaller one to get distance between them.

• **Step-4:** Add all distances after traversing all positions.

**Problem**

From the given position which is 143 move to next positions according to FCFS(First come first serve) manner.

**Explanation**

To do so we have to add all time taken by head to move from one to another position in order of allotting next position *pg. 8*

Distance=125 to 143=18; 143 to 86=57; 86 to 1470=1384; 1470 to 913 =557; 913 to 1774=861; 1774 to 948=826; 948 to 1509=561; 1509 to 1022=487; 1022 to 1750=728; 1750 to 130=1620;

**Code Snippet**

#include<stdio.h>

int main()

{

int n;

int previous,cur;

printf("Enter number of processes: \n");

scanf("%d",&n);

int queue[n];

printf("Enter the Previous Requested position\n");

scanf("%d",&previous);

printf("Enter the current header position\n");

scanf("%d",&cur);

printf("Enter Processes in sequence: \n");

for(int j=0;j<n;j++)

{

scanf("%d",&queue[j]);

}

int total =0 ;

if (previous>cur)

{

total = previous+cur;

}

else

{

total = cur-previous;

}

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

{

if (queue[i]>cur)

{

total += (queue[i]-cur);

cur = queue[i];

}

else

{

total += (cur-queue[i]);

cur = queue[i];

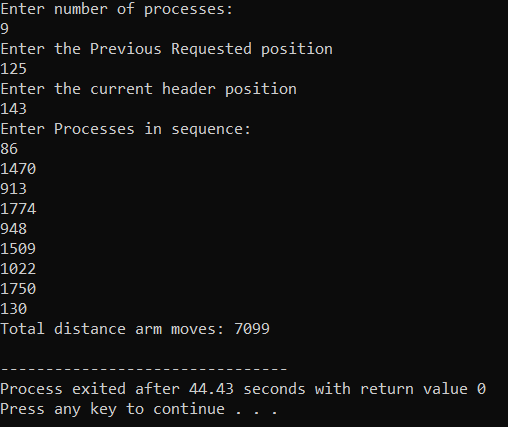
}

}

printf("Total distance arm moves: %d\n", total);

}

**Output**

****

**Q17. Design a scheduling program to implements a Queue with two levels:**

**Level 1 : Fixed priority preemptive Scheduling**

**Level 2: Round Robin Scheduling**

**For a Fixed priority preemptive Scheduling (Queue1), the Priority 0 is highest priority. If one process P1 is scheduled and running, another process P2 with higher priority comes. The New process (high priority) process P2 preempts currently running process P1 and process P1 will go to second level queue. Time for which process will strictly execute must be considered in the multiples of 2. All the processes in second level queue will complete their execution according to round robin scheduling.**

**Consider: 1. Queue 2 will be processed after Queue 1 becomes empty.**

**2. Priority of Queue 2 has lower priority than in Queue 1.**

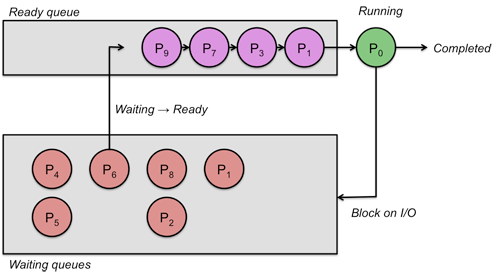
*Ans: -*

Fixed-priority preemptive scheduling is a scheduling system commonly used in real-time systems. With fixed priority preemptive scheduling, the scheduler ensures that at any given time, the processor executes the highest priority task of all those tasks that are currently ready to execute.

The preemptive scheduler has a clock interrupt task that can provide the scheduler with options to switch after the task has had a given period to execute—the time slice. This scheduling system has the advantage of making sure no task hogs the processor for any time longer than the time slice. However, this scheduling scheme is vulnerable to process or thread lockout: since priority is given to higher-priority tasks, the lower-priority tasks could wait an indefinite amount of time. One common method of arbitrating this situation is aging, which gradually increments the priority of waiting processes and threads, ensuring that they will all eventually execute. Most real-time operating systems (RTOSs) have preemptive schedulers. Also turning off time slicing effectively gives you the non-preemptive RTOS.

**First-Come, First-Served Scheduling**

Possibly the most straightforward approach to scheduling processes is to maintain a FIFO (first-in, first-out) run queue. New processes go to the end of the queue. When the scheduler needs to run a process, it picks the process that is at the head of the queue. This scheduler is non-preemptive. If the process has to block on I/O, it enters the *waiting* state and the scheduler picks the process from the head of the queue. When I/O is complete and that waiting (blocked) process is ready to run again, it gets put at the end of the queue.



With first-come, first-served scheduling, a process with a long CPU burst will hold up other processes, increasing their turnaround time. Moreover, it can hurt overall throughput since I/O on processes in the *waiting* state may complete while the CPU bound process is still running. Now devices are not being used effectively. To increase throughput, it would have been great if the scheduler instead could have briefly run some I/O bound process so that could run briefly, request some I/O and then wait for that I/O to complete. Because CPU bound processes don’t get preempted, they hurt interactive performance because the interactive process won’t get scheduled until the CPU bound one has completed.

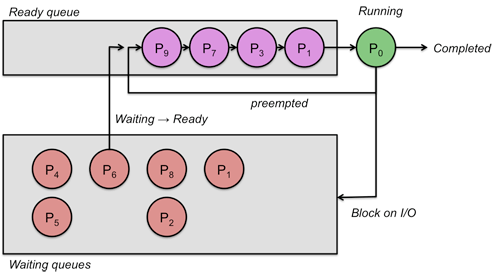
**Advantage**: FIFO scheduling is simple to implement. It is also intuitively fair (the first one in line gets to run first).

**Disadvantage**: The greatest drawback of first-come, first-served scheduling is that it is not preemptive. Because of this, it is not suitable for interactive jobs. Another drawback is that a long-running process will delay all jobs behind it.

**Round robin scheduling**

Round robin scheduling is a preemptive version of first-come, first-served scheduling. Processes are dispatched in a first-in-first-out sequence but each process is allowed to run for only a limited amount of time. This time interval is known as a **time-slice** or **quantum**. If a process does not complete or get blocked because of an I/O operation within the time slice, the time slice expires and the process is preempted. This preempted process is placed at the back of the run queue where it must wait for all the processes that were already in the queue to cycle through the CPU.

If a process gets blocked due to an I/O operation before its time slice expires, it is, of course, enters a blocked because of that I/O operation. Once that operation completes, it is placed on the end of the run queue and waits its turn.



A big advantage of round robin scheduling over non-preemptive schedulers is that it dramatically improves average response times. By limiting each task to a certain amount of time, the operating system can ensure that it can cycle through all ready tasks, giving each one a chance to run.

With round robin scheduling, interactive performance depends on the length of the quantum and the number of processes in the run queue. A very long quantum makes the algorithm behave very much like first come, first served scheduling since it’s very likely that a process with block or complete before the time slice is up. A small quantum lets the system cycle through processes quickly. This is wonderful for interactive processes. Unfortunately, there is an overhead to context switching and having to do so frequently increases the percentage of system time that is used on context switching rather than real work.

Advantage: Round robin scheduling is fair in that every process gets an equal share of the CPU. It is easy to implement and, if we know the number of processes on the run queue, we can know the worst-case response time for a process.

Disadvantage: Giving every process an equal share of the CPU is not always a good idea. For instance, highly interactive processes will get scheduled no more frequently than CPU-bound processes.

**Code Snippet : -**

#include<stdio.h>

#include<stdio.h>

#include<stdlib.h>

struct process

{

int p\_name;

int arvlTym, wytngTym, trnArndTym, prty, BTCopy,BT;

}q1[10],q2[10];

void main()

{

struct process s1;

int i,j,time=0,qt1,qt2,bu\_t=0,largest,tp,tp2,count=0,k,pf2=0,n,pos,flag=0,y,t1,t2;

float WT=0,TA= 0,avgWT,avgTA;

printf("\n Enter Total Number of Processes:\n");

scanf("%d", &tp);

n=tp;

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

{

printf("\nEnter Process name\n");

//fflush(stdin);//to flush the buffer

scanf("%d",&q1[i].p\_name);

printf("Enter Arrival Time\n");

scanf("%d",&q1[i].arvlTym);

printf("Enter Burst Time\n");

scanf("%d",&q1[i].BT);

q1[i].BTCopy = q1[i].BT;

printf("Enter Priority\n");

scanf("%d",&q1[i].prty);

}

printf("\nEnter Time Quantum for Fixed Priority queue:-");

scanf("%d",&qt1);

printf("\nEnter Time Quantum for Round Robin queue:-");

scanf("%d",&qt2);

printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");

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

{

pos=i;

for(j=i+1;j<tp;j++)

{

if(q1[j].arvlTym < q1[pos].arvlTym)

pos=j;

}

s1 = q1[i];

q1[i] = q1[pos];

q1[pos] = s1;

}

time=q1[0].arvlTym;

for(i=0;tp!=0;i++)

{

while(count!=t1)

{

count++;

if(q1[i].arvlTym<=time)

{

for(j=i+1;j<tp;j++)

{

if(q1[j].arvlTym==time && q1[j].prty<q1[i].prty)

{

q2[pf2]=q1[i];

pf2++;

for(k=i; k<tp-1;k++)

q1[k]=q1[k+1];

tp--;

count=0;

i=j-1;

j--;

}

}

}

time++;

q1[i].BT--;

if(q1[i].BT==0)

{

q1[i].trnArndTym = time-q1[i].arvlTym;

q1[i].wytngTym=q1[i].trnArndTym - q1[i].BTCopy;

printf("%d\t|\t%d\t|\t%d\n",q1[i].p\_name,q1[i].trnArndTym,q1[i].wytngTym);

WT += time - q1[i].wytngTym;

TA += time - q1[i].trnArndTym;

for(k=i;k<tp-1;k++)

q1[k] = q1[k+1];

i--;

tp--;

count = t1;

break;

}

}

count=0;

if(q1[i].BT!=0)

{

q2[pf2]=q1[i];

pf2++;

for(k=i;k<tp-1;k++)

q1[k] = q1[k+1];

tp--;

}

if(i==tp-1)

i=-1;

}

tp2=pf2;

for(count=0;tp2!=0;)

{

if(q2[count].BT<=t2&&q2[count].BT>0)

{

time += q2[count].BT;

q2[count].BT=0;

flag=1;

}

else if(q2[count].BT>0)

{

q2[count].BT -= t2;

time+=t2;

}

if(q2[count].BT==0&&flag==1)

{

tp2--;

q2[count].trnArndTym = time - q2[count].arvlTym;

q2[count].wytngTym = q2[count].trnArndTym - q2[count].BTCopy;

printf("%d\t|\t%d\t|\t%d\n",q2[count].p\_name,q2[count].trnArndTym,q2[count].wytngTym);

TA += time-q2[count].arvlTym;

WT += time-q2[count].arvlTym - q2[count].BTCopy;

for(k=count; k<tp2;k++)

q2[k]=q2[k+1];

count--;

flag=0;

}

if(count==tp2-1)

count=0;

else

count++;

}

avgWT = WT/tp;

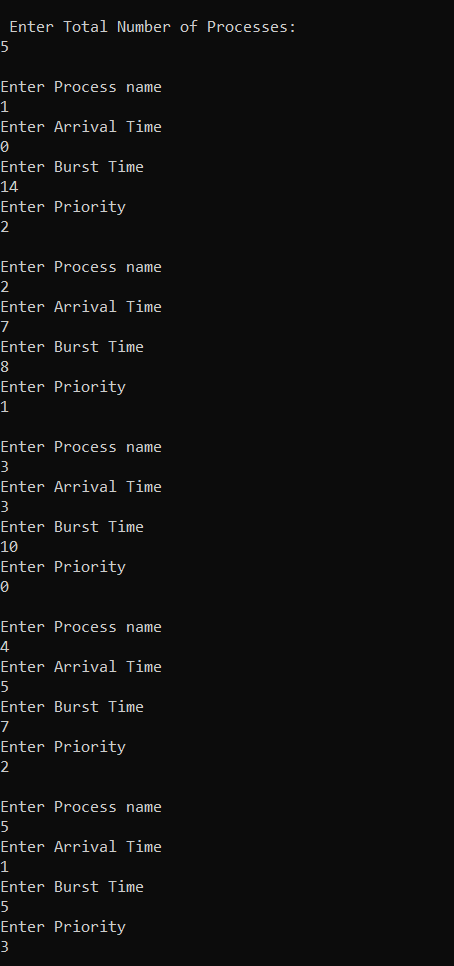
avgTA = TA/tp;

printf("\n Average Waiting Time= %f\n", avgWT);

printf("Avg Turnaround Time = %f" , avgTA);

}

**Output**

****

