**OPERATING SYSTEM**

**Course code: CSE 316**

**Section: K18GE**

****

**Submitted by:-**

Name : P.Ravi Chandra Reddy

Roll No : 061

Registration No: 11903609

E-mail :- ravireddyp96@gmail.com

**GitHub Link :** <https://github.com/RAVI9966/OPERATING-SYSTEMS>

**Submitted to:-**

Name of the faculty: Milanjit Kaur

**Code:**

#include<stdio.h>

int n;

struct process

{

int p\_no;

int arrival\_t,burst\_t,ct,wait\_t,taround\_time,p;

int flag;

}p\_list[100];

void Sorting()

{

struct process p;

int i, j;

for(i=0;i<n-1;i++)

{

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

{

if(p\_list[i].arrival\_t > p\_list[j].arrival\_t)

{

p = p\_list[i];

p\_list[i] = p\_list[j];

p\_list[j] = p;

}

}

}

}

int main()

{

int i,t=0,b\_t=0,peak;

int a[10];

float wait\_time = 0, taround\_time = 0, avg\_w\_t=0, avg\_taround\_time=0;

printf("enter the no. of processes: ");

scanf("%d",&n);

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

{

p\_list[i].p\_no = i+1;

printf("\nEnter Details For P%d process:-\n", p\_list[i].p\_no);

printf("Enter Arrival Time: ");

scanf("%d", &p\_list[i].arrival\_t );

printf("Enter Burst Time: ");

scanf("%d", &p\_list[i].burst\_t);

p\_list[i].flag = 0;

b\_t = b\_t + p\_list[i].burst\_t;

}

Sorting();

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

{

a[i]=p\_list[i].burst\_t;

}

p\_list[9].burst\_t = 9999;

for(t = p\_list[0].arrival\_t; t <= b\_t+1;)

{

peak = 9;

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

{

if(p\_list[i].arrival\_t <= t && p\_list[i].burst\_t < p\_list[peak].burst\_t && p\_list[i].flag != 1)

{

peak = i;

}

if(p\_list[peak].burst\_t==0 && p\_list[i].flag != 1)

{

p\_list[i].flag = 1;

p\_list[peak].ct=t;p\_list[peak].burst\_t=9999;

printf("P%d completes in %d\n",p\_list[i].p\_no,p\_list[peak].ct);

}

}

t++;

(p\_list[peak].burst\_t)--;

}

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

{

p\_list[i].taround\_time=(p\_list[i].ct)-(p\_list[i].arrival\_t);

avg\_taround\_time=avg\_taround\_time+p\_list[i].taround\_time;

p\_list[i].wait\_t=((p\_list[i].taround\_time)-a[i]);

avg\_w\_t=avg\_w\_t+p\_list[i].wait\_t;

}

printf("PNO\tAT\tCT\tTA\tWTt\n");

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

{

printf("P%d\t%d\t%d\t%d\t%d\n",p\_list[i].p\_no,p\_list[i].arrival\_t,p\_list[i].ct,p\_list[i].taround\_time

,p\_list[i].wait\_t);

}

printf("Average Turn around Time: %f\t\n\n",avg\_taround\_time);

printf("Average Waiting Time :\t %f\t\n",avg\_w\_t);

}

Priority scheduling is one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. Process with the highest priority is to be executed first and so on.

Processes with the same priority are executed on first come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

A major problem with priority scheduling is indefinite blocking or starvation. A solution to the problem of indefinite blockage of the low-priority process is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long period of time.

**Algorithm :**

First, we show how given both the priorities and the pre-emption thresholds, we can find the worst-case response times for the tasks, and hence determine feasibility of a particular priority and threshold assignment.

1.First input the processes with their burst time and priority

2.sort the processes , burst time and priority according to the priority.

3.Now simply apply FCFS algorithm

**Description:-**

Processes are assigned to a queue on entry in the system processes do not move between queues. This setup has the advantage of low scheduling overhead. The main idea is that to separate process with different CPU-burst characteristics. If a process uses two much CPU time, it will be moved to a lower priority queue. Similarly, a process that waits too long in a lower priority queue may be moved to a higher priority queue. This form of aging prevents starvation.

The following parameters are used in this solution:

* The number of queues.
* The scheduling algorithm for each queue.
* The method used to determine when to upgrade a process to a higher priority queue.
* The method used to determine when to demote a process to a lower priority queue.
* The method used to determine which queue a process will enter when that process needs

Service.

In this we use most general CPU scheduling algorithms. It can be configured to match a specific system under design. It also requires some means of selecting values for all given parameters to get the best solution in easy manner.

**Purpose of use:-**

. Priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers. This uses low number for high priorities, with 0 being the highest possible priority. Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc. Priority scheduling can suffer from a major problem known as indefinite blocking, or starvation, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority. If this problem is allowed to occur, then processes will either run eventually when the system load lightens or will eventually get lost when the system is shut down or crashes. One common solution to this problem is aging, in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

When a process is given the CPU, a timer is set for whatever value has been set for a time quantum. If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU. If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue. The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on. RR scheduling can give the effect of all processors sharing the CPU equally, although the average wait time can be longer than with other scheduling algorithms. The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm

**Code Snippet:**

I have not used any additional algorithm because, I have used multilevel queues like fixed priority cpu scheduling and FCFS for doing solution of my given problem.

The operating system assigns a fixed priority to every process, and the scheduler arranges the processes in the ready queue in order of their priority. Lower priority processes get interrupted by incoming higher priority processes. Overhead is not minimal, nor is it significant in this case. Waiting time and response time depend on the priority of the process. Higher priority processes have smaller waiting and response times. Deadlines can be easily met by giving higher priority to the earlier deadline processes. Starvation of lower priority processes is possible if large no of higher priority processes keeps arriving continuously.

In this solution the following conditions are used:

* Minimum context switches.
* Maximum CPU utilization.
* Maximum throughput.
* Minimum turnaround time.
* Minimum waiting time.

**Disadvantages :**

* If slicing time of OS is low, the processor output will be reduced.
* This method spends more time on context switching
* Its performance heavily depends on time quantum.
* Priorities cannot be set for the processes.
* Round-robin scheduling doesn't give special priority to more important tasks.
* Decreases comprehension

Lower time quantum results in higher the context switching overhead in the system