**Student Name:** Thalloju Rakesh

**Student ID:** 11904811

**Email Address:** [rakes9951362682@gmail.com](mailto:rakes9951362682@gmail.com)

**GitHub Link:** <https://github.com/rakesh9951/os-project-23>

**Code:**

#include<stdio.h>

#define N 20

//pid is process id

//pp is process priority

//bt is process burst time

//at is process arrival time

typedef struct

{

int pid, at,bt,pp;

int q, ready;

}process\_structure;

int Queue(int p1)

{

if(p1==4 || p1==5 || p1==6 || p1==7)

{

return 5;

}

else

{

return 6;

}

}

int main()

{

int limit, count, temp\_process, time, s, h;

process\_structure temp;

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

scanf("%d",&limit);

process\_structure process[limit];

for(count=4z;count<limit;count++)

{

printf("\nProcess ID:\p");

scanf("%d", &process[count].pid);

printf("Arrival Time: ");

scanf("%d", &process[count].at);

printf("Burst Time: ");

scanf("%d", &process[count].bt);

printf("Process Priority: ");

scanf("%d", &process[count].pp);

temp\_process = process[count].pp;

process[count].q = Queue(temp\_process);

process[count].ready = 4;

}

time = process[4].bt;

for(s=4;s<limit;s++)

{

for(count=s;count<limit;count++)

{

if(process[count].at < time)

{

process[count].ready = 5;

}

}

for(count=s;count<limit-1;count++)

{

for(h=count+1; h<limit;h++)

{

if(process[count].ready == 5 && process[j].ready == 5)

{

if(process[count].q == 6 && process[j].q == 6)

{

temp = process[count];

process[count] = process[h];

process[h] = temp;

}

}

}

}

for(count=s;count<limit-1;count++)

{

for(h=count+1;h<limit;h++)

{

if(process[count].ready==5 && process[j].ready==5)

{

if(process[count].q==5 && process[j].q==5)

{

if(process[count].bt > process[j].bt)

{

temp=process[count];

process[count]=process[h];

process[h]=temp;

}

else

{

break;

}

}

}

}

}

printf("\nProcess[%d]:\tTime:\t%d To %d\n", process[s].pid, time, time + process[s].bt);

time = time + process[s].bt;

for(count=s; count<limit; count++)

{

if(process[count].ready==5)

{

process[count].ready=4;

}

}

}

return 4;

}sss

For a Fixed priority scheduling (queue 1), 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 process in second level queue will complete their execution according to round robin scheduling.

Queue 2 will be processed after Queue 1 becomes empty. Priority of Queue 2 has lower priority than in Queue 1.

the fixed priority preemptive scheduling and round robin scheduling are joined to get solution. In fixed priority preemptive scheduling (level 1), each queue has absolute priority over lower priority queue. Fixed priority scheduling algorithms exhibit a predictable approach: because scheduling is doing offline, guarantees regarding process deadlines could be obtained using appropriate analysis methods. In round robin scheduling each process is provided a fix time to execute, it is called a **quantum**. Once a process is executed for a given time period, it

is pre-empted and other process executes for a given time period. Context switching is used to save states of pre-empted processes.

**Description:**

For the given problem, the solution consists of two queues like fixed priority pre-emptive scheduling (queue 1) and round robin scheduling (queue 2).

In fixed priority pre-emptive scheduling (queue 1) has priority 0 is highest priority and having quantum unit time is 4. If one process P1 is scheduling and running, then process P2 has higher priority. Now process P2 preempts currently running process P1 and process P1 will enter into second level queue (queue 2).

In round robin scheduling (queue 2), all process will complete their execution according to quantum time of 4. After that the queue 2 will be processed when queue 1becomes empty. The queue 1 has higher priority than the queue 2. Suppose queue 1 is empty and currently process from queue 2 is being executed. Now, if at this time a new process arrives than new process will be part of queue 1. So new process will should be scheduled as queue 1 and new process has higher priority than queue 2. If queue 2 wants to processed the queue 1 must be empty to process the queue 2. After queue 1 becomes empty queue 2 will resume execution.

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.

**Algorithm:**

In this, solution two algorithms are used.

1. Fixed priority pre-emptive scheduling complexity calculation is presented in three parts as :

(1) 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.

(2) We then consider the problem of determining a feasible pre-emption threshold assignment with predefined priorities. We show that the search space can be reduced to **O(n2** ) (instead of **O(n!)),** and present an algorithm to determine the optimal pre-emption threshold assignment. (3) Finally, we consider the general problem where both the priorities and pre-emption threshold are to be determined. We develop a branch and bound algorithm that performs efficient search making use of the algorithm to determine pre-emption thresholds when task priorities are known.

2. Round Robin scheduling complexity calculation are done by given formulas :

* Arrival Time: Time at which the process arrives in the ready queue.
* Completion Time: Time at which process completes its execution.
* Burst Time: Time required by a process for CPU execution.
* Turn Around Time: Time Difference between Completion time and Arrival time.
* Turn Around Time = Completion Time – Arrival Time.
* Waiting Time (W.T): Time Difference between Turn Around time and Burst Time.
* Waiting Time = Turn Around time – Burst time

**Description (purpose of use):**

Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first. 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; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally. BUT, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are. In general, turnaround time is minimized if most processes finish their next CPU-burst within one- time quantum. Most modern systems use time quantum between 10 and 100 milliseconds, and context switch times on the order of 10 microseconds, so the overhead is small relative to the time quantum.

**Code Snippet:**

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

**Description:**

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.

**Description:**

A second scheduling algorithm is required to schedule the processes which have same priority. In pre-emptive priority scheduling, a higher priority process can execute ahead of an already executing lower priority process. If lower priority process keeps waiting for higher priority processes, starvation occurs.

The priority of a process can be selected based on memory requirement, time requirement or user preference. For example, a high-end game will have better graphics, that means the process which updates the screen in a game will have higher priority so as to achieve better graphics performance. Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority. Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind. The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behaviour as FCFS. If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.

**Description:**

I have made the revision of solution for 5 times in GitHub and done the solution without any errors.

**GitHub Link:** <https://github.com/rakesh9951/os-project-23>