

**Operating Systems (CSE 316) Project**

**Project on Implementation of Multilevel feedback queue in C**

**Submitted By**: Nishant Pandey

**Registration Number**: 11810605

**Course Code & Title**: CSE316

**Email Address**: [unexme21@gmail.com](mailto:unexme21@gmail.com)

**GitHub Link**: <https://github.com/unexh/Multilevel-Feedback-Queue-in-C.git>

**Submitted To**: Dr. Baljit Singh Saini

**Question Statement:**

" Design a scheduling program to implements a Queue with two levels:  
Level 1: Fixed priority pre-emptive Scheduling  
Level 2: Round Robin Scheduling  
For a Fixed priority pre-emptive 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 pre-empts 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. "

**Approach to solve the problem (Explanation of Problem):**

1. Fixed priority pre-emptive Scheduling (Queue 1)

\* Priority 0 is highest priority.

\* Pre-emptive:

If one process e.g. P1 is scheduled and running, now another process with higher priority comes e.g. P2. New process (high priority)

process P2 pre-empts currently running process P1 and process P1 will go to second level queue.

2. Round Robin Scheduling (Queue 2)

\* Quantum: 4-unit time

\* All the processes 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.

Suppose Queue 1 is empty and currently process from Queue 2 is being executed. Now, if at this time a new process arrives then new process will be part of Queue 1. So, new

process should be scheduled as Queue 1 has higher priority than Queue 2. Again, after Queue 1 becomes empty Queue 2 will resume execution.

**Code (Program):** (upto page 19)

#include<stdio.h>

#include<stdlib.h>//for malloc and calloc

#include<stdbool.h>

#include<limits.h>

///defining a structure of a process

struct ProcessData{

int Number;

int Pid;

int BurstTime;

int Priority;

int ArrivalTime;

int Finishtime;

int RemainTime;

int StartTime;

int WaitTime;

int ResTime;

};

typedef struct ProcessData PD;

PD CurrentProcess;

//Input func : that takes processes from user

void takeInputProcess(int count,PD\* process){

for(int i=0;i<count;i++){

printf("Enter Pid, ArrivalTime, BurstTime and PriorityNumber for Process[%d] : ",i+1);

scanf("%d %d %d %d",&process[i].Pid,&process[i].ArrivalTime,&process[i].BurstTime,&process[i].Priority);

//take input

process[i].Number=i+1;

process[i].RemainTime = process[i].BurstTime;

}

}

void printProcessArray(int count,PD\* process){

//printing for testing input just taken

printf("\n\nPrinting ProcessArray Entered:\n");

for(int i=0;i<count;i++){

printf("Process[%d] : Pid:%d | ArrivalTime:%d |BurstTime:%d | PriorityNumber:%d\n",process[i].Number,process[i].Pid,process[i].ArrivalTime,process[i].BurstTime,process[i].Priority);

}

}

int sortingArrival(const PD\* a,const PD\* b){

return (\*a).ArrivalTime-(\*b).ArrivalTime;

}

bool pidSort(const PD\* a , const PD\* b){

return (\*a).Pid > (\*b).Pid;

}

bool numberSort( const PD\* a ,const PD\* b){

return (\*a).Number > (\*b).Number;

}

///implementing Queue

// A structure to represent a queue

struct Queue

{

int front, rear, size;

unsigned capacity;

PD Data[10];

};

typedef struct Queue NormalQueue;

// function to create a queue of given capacity.

// It initializes size of queue as 0

struct Queue\* createQueue(unsigned capacity)

{

struct Queue\* queue = (struct Queue\*) malloc(sizeof(struct Queue));

queue->capacity = capacity;

queue->front = queue->size = 0;

queue->rear = capacity - 1; // This is important, see the enqueue

//queue->Data = (PD\*) malloc(queue->capacity \* sizeof(PD));

return queue;

}

// Queue is full when size becomes equal to the capacity

int isFull(struct Queue\* queue)

{ return (queue->size == queue->capacity); }

// Queue is empty when size is 0

int isEmpty(struct Queue\* queue)

{ return (queue->size == 0); }

// Function to add an item to the queue.

// It changes rear and size

void enqueue(struct Queue\* queue, PD prcoessDq )

{

if (isFull(queue))

return;

queue->rear = (queue->rear + 1)%queue->capacity;

queue->Data[queue->rear]=prcoessDq;

queue->size = queue->size + 1;

//printf("Pid [%d], enqueued to queue\n", queue->Data[queue->rear].Pid);

//printf("enqueued Successfully..\n");

}

//Function to remove an item from queue.

//It changes front and size

void dequeue(struct Queue\* queue)

{

if (isEmpty(queue))

return INT\_MIN;

PD dequeuedProcessD = queue->Data[queue->front];

queue->front = (queue->front + 1)%queue->capacity;

queue->size = queue->size - 1;

//printf("Item Dequeued Successfully\n");

}

//Function to get front of queue

PD front(struct Queue\* queue)

{

if (isEmpty(queue)){

printf("Empty Queue..\n");

return;

}

return queue->Data[queue->front];

}

//Function to get rear of queue

PD rear(struct Queue\* queue)

{

if (isEmpty(queue))

return;

return queue->Data[queue->rear];

}

//////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//////

//Priority Queue implementation

typedef struct node {

PD data;

// Lower values indicate higher priority

int priority;

struct node\* next;

} Node;

// Function to Create A New Node

Node\* newNode(PD d, int p)

{

Node\* temp = (Node\*)malloc(sizeof(Node));

temp->data = d;

temp->priority = p;

temp->next = NULL;

//printf("First Node added\n");

return temp;

}

// Return the value at head

PD peek(Node\*\* head)

{

return (\*head)->data;

}

// Removes the element with the

// highest priority form the list

void pop(Node\*\* head)

{

Node\* temp = \*head;

(\*head) = (\*head)->next;

free(temp);

}

// Function to push according to priority

void push(Node\*\* head, PD d, int p)

{

//printf("Pushing Next Node\n");

Node\* start = (\*head);

// Create new Node

Node\* temp = newNode(d, p);

// Special Case: The head of list has lesser

// priority than new node. So insert new

// node before head node and change head node.

if ((\*head)->priority > p) {

// Insert New Node before head

temp->next = \*head;

(\*head) = temp;

}

else {

// Traverse the list and find a

// position to insert new node

while (start->next != NULL &&

start->next->priority < p) {

start = start->next;

}

// Either at the ends of the list

// or at required position

temp->next = start->next;

start->next = temp;

}

}

// Function to check is list is empty

int isEmptyPQ(Node\*\* head)

{

return (\*head) == NULL;

}

int main(){

int processCount=0;

printf("Enter the total number of process : ");

scanf("%d",&processCount);

//test

printf("Total number Entered :[%d]",processCount);

printf("\n\n");

PD \*process,\*processCopy;

process = (PD\*) calloc(processCount,sizeof(PD));

if(process == NULL){

printf("Sorry No allocation is possible Right Now!\n");

}

//taking input

takeInputProcess(processCount,process);

//printing input

printProcessArray(processCount,process);

processCopy=process;

//sorting on the basis of arrival time

qsort(process,processCount,sizeof(process[0]),sortingArrival);

//printf("After Qsort (on arrival time) processArray is : \n");

//printProcessArray(processCount,process);

int totalExecutionTime = 0;

totalExecutionTime = totalExecutionTime + process[0].ArrivalTime;

for(int i=0;i<processCount;i++){

if(totalExecutionTime>=process[i].ArrivalTime){

totalExecutionTime = totalExecutionTime + process[i].BurstTime;

}

else{

int difference = (process[i].ArrivalTime - totalExecutionTime);

totalExecutionTime = totalExecutionTime + difference + process[i].BurstTime;//change

}

}

int Ghant[totalExecutionTime]; //Ghant Chart

for(int i =0;i<totalExecutionTime;i++){

Ghant[i]=-1;

}

printf("\nTotal Execution Time : [%d]",totalExecutionTime);

NormalQueue\* rQ = createQueue(processCount);//Queue

Node\* pQ = NULL;//PriorityQueue

int cpuState = 0; //idle if 0 then Idle if 1 the Busy

int quantum = 4 ; //Time Quantum

CurrentProcess.Pid = -2;

CurrentProcess.Priority = 999999;

int Clock,pQProcesss=0,rQProcesss=0;

for(Clock=0;Clock<totalExecutionTime;Clock++){

/\*\*Insert the process with same Arrival time in Priority Queue\*\*/

for(int i=0;i<processCount;i++){

if(Clock == process[i].ArrivalTime){

if(pQ==NULL){

pQ = newNode(process[i],process[i].Priority);

}

else{

push(&pQ,process[i],process[i].Priority);

}

}

}

if(cpuState==0){//CPU Idle

if(!isEmptyPQ(&pQ)){

CurrentProcess = peek(&pQ);

cpuState=pQProcesss= 1;

pop(&pQ);//test POP function

quantum = 4;

}

else if(!isEmpty(rQ)){

CurrentProcess=front(rQ);

cpuState=rQProcesss= 1;

dequeue(rQ);

quantum=4;

}

}

else if(cpuState==1){ //If CPU has any process

if(pQProcesss==1 && (!isEmptyPQ(&pQ))){

if(peek(&pQ).Priority < CurrentProcess.Priority){ //If new process has high priority

enqueue(rQ,CurrentProcess); //push CurrentProcess to rQ

CurrentProcess = peek(&pQ);

pop(&pQ);

quantum=4;

}

}

else if(rQProcesss==1 &&(!isEmptyPQ(&pQ))){//If process is from RQ and new process come in PQ

enqueue(rQ,CurrentProcess);

CurrentProcess=peek(&pQ);

pop(&pQ);

rQProcesss=0;

pQProcesss=1;

quantum=4;

}

}

if(CurrentProcess.Pid != -2){// Process Execution

CurrentProcess.RemainTime--;

quantum--;

Ghant[Clock]=CurrentProcess.Pid;

if(CurrentProcess.RemainTime == 0){//if process Finishes

cpuState=0;

quantum=4;

CurrentProcess.Pid=-2;

CurrentProcess.Priority = 999999;

rQProcesss=pQProcesss=0;

}

else if(quantum==0){//If time Quantum of a current running process Finish

enqueue(rQ,CurrentProcess);

CurrentProcess.Pid=-2;

CurrentProcess.Priority=999999;

rQProcesss=pQProcesss=cpuState=0;

}

}

}

qsort(process,processCount,sizeof(process[0]),pidSort);

//print ProcessArray

/\*

printf("After Qsorting again (pidSort) data :\n");

printProcessArray(processCount,process);

\*/

for(int i=0;i<processCount;i++){

for(int j=totalExecutionTime;j>=0;j--){

if(Ghant[j]==i+1){

process[i].Finishtime=j+1;

break;

}

}

}

for(int i=0;i<processCount;i++){

for(int j=0;j<totalExecutionTime;j++){

if(Ghant[j]==i+1){

process[i].StartTime=j;

break;

}

}

}

qsort(process,processCount,sizeof(process[0]),numberSort);

/\*

//check sorting

printf("Result after numberSort\n");

printProcessArray(processCount,process);

\*/

for(int i=0;i<processCount;i++){

process[i].ResTime=process[i].StartTime-process[i].ArrivalTime;

process[i].WaitTime=(process[i].Finishtime - process[i].ArrivalTime) - process[i].BurstTime;

}

printf("\n\nResult : \n");

for(int i=0;i<processCount;i++){

printf("Process[%d] | Pid:[%d] | ResponseTime:[%d]| FinishTime:[%d] | WaitTime:[%d]\n",process[i].Number,process[i].Pid,process[i].ResTime,process[i].Finishtime,process[i].WaitTime);

}

return 0;

}

**Sone Explanation of Code:**

First an object of structure type is created which is used to store process data.

Then various input functions are called to update those default values to its question specific values.

After which, I have Implemented 2 queues with the help of structures and supporting function of these ADTs (Abstract Data Types), these queues are:

1. rQ: a simple queue (Array implementation).
2. pQ: a priority queue (linked List implementation).

The above mentioned queues are used to calculate various response time and other expected outputs with the logic loops defined in the main().

**Complexity of Program:**

Time Complexity:

O(n^2): n = processCount

1. in program blocks (for inside for)

O(nlogn): n = processCount

1. qsort (void\* base, size\_t num, size\_t size, int (\*comparator)(const void\*, const void\*))

O(n): n = processCount

1. takeInputProcess(int count,PD\* process )
2. printProcessArray(int count,PD\* process)
3. push(Node\*\* head, PD d, int p)

O(1): n = processCount

1. sortingArrival(const PD\* a,const PD\* b)
2. pidSort(const PD\* a , const PD\* b)
3. numberSort( const PD\* a ,const PD\* b)
4. createQueue(unsigned capacity)
5. isFull(struct Queue\* queue)
6. isEmpty(struct Queue\* queue)
7. enqueue(struct Queue\* queue, PD prcoessDq )’
8. dequeue(struct Queue\* queue)
9. front(struct Queue\* queue)
10. rear(struct Queue\* queue)
11. newNode(PD d, int p)
12. peek(Node\*\* head)
13. pop(Node\*\* head)
14. isEmptyPQ(Node\*\* head)

**Overall Time Complexity = O( n^2 ) + O( nlog(n) ) + 3\*O( n) ) + 14\*O( 1 )**

**= O( n^2 )** n = processCount

**Assumptions (Constraints):**

1. Problem is written in C language.
2. processCount, as well as all the inputs provided must be an integer.
3. Input format: <Pid> <Arrival Time> <Burst Time> <Priority>.
4. Output format: <Pid> <Response Time> <Finish Time> <Waiting Time>.
5. Next query will arrive before the termination of first query (After sorting all queries according to their Arrival Time).
6. All time calculations are in minutes.
7. Time Quantum is fixed to 4.

**Additional Algorithm used in Program:**

Quick Sort algorithm is used in program for sorting all queries according to their Arrival Time. The reason for that is the user can enter queries in any order and to solve the problem using Round Robin, I needed all queries in sorted order. Thus, this algorithm is used. From c std library, see : qsort.

**Boundary Conditions of Code:**

1. All the inputs entered should be of type int.
2. Inputs must be less than 150 in number and must be of type integer.
3. Next query will arrive before the termination of first query (After sorting all queries according to their Arrival Time).
4. Sum of burst time of all queries should be less than or equal to 120.
5. Time Quantum is fixed to 4.
6. No two Pid should be same.

**Test Cases:**

CASE 1:

In this Case we will try to test the programs on different arrival time and priorities given to it as an input,

**INPUTS**

processCount = 5

ProcessID Arrival Time Burst Time(minutes) Priority

1 0 14 2

2 7 8 1

3 3 10 0

4 5 7 2

5 1 5 3

Constants in program => Time Quantum = 4 mins

**Expected Output:**

ProcessID Response Time Finish Time Waiting Time

1 0 44 30

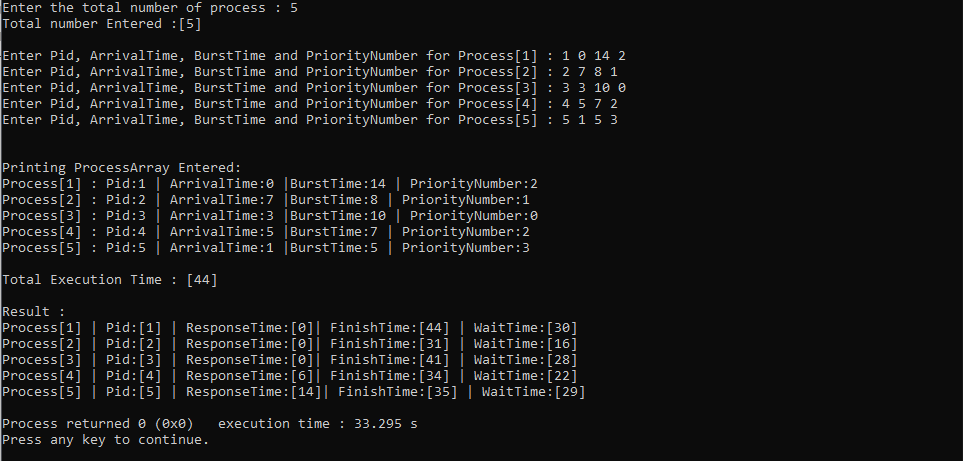
2 0 31 16

3 0 41 28

4 6 34 22

5 14 35 29

**Actual Output:**



As it is clear from the above snapshot, the program was able to satisfy the expected output successfully.

**CASE 2:**

**INPUTS**

Here we will try to input process having same arrival time and priorities,

processCount = 4

ProcessID Arrival Time Burst Time (minutes) Priority

1 0 4 1

2 0 3 1

3 0 8 2

4 10 5 1

Constants in program => Time Quantum = 4

**Expected Output:**

ProcessID Response Time Finish Time Waiting Time

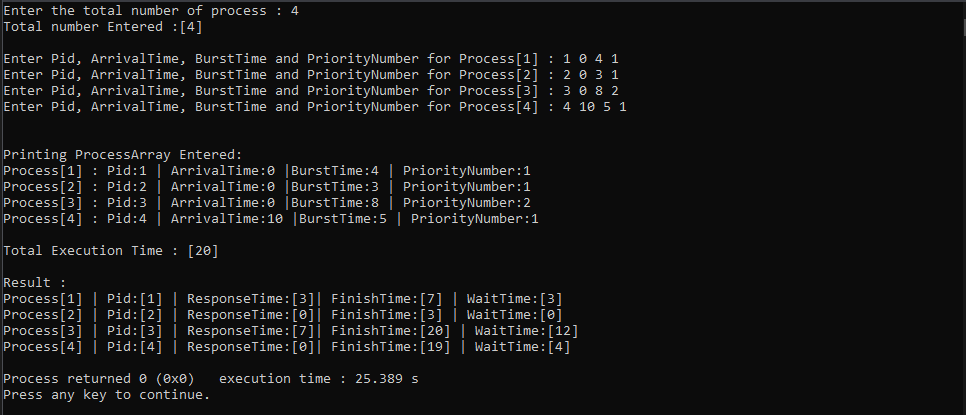
1 3 7 3

2 0 3 0

3 7 20 12

4 0 19 4

**Actual Output:**



As it is clear from the above snapshot, the program was able to satisfy the expected output successfully.

**I have done a total of 10 commits on this project on GitHub, the following link will verify this.**

**GitHub Link:** [**https://github.com/unexh/Multilevel-Feedback-Queue-in-C.git**](https://github.com/unexh/Multilevel-Feedback-Queue-in-C.git)