5/22/2021

Project: Multi Level Feedback Queue Scheduling Algorithm

Subject: Operating System

Submitted to: Ma'am Sobia Khalid

By:
Mahrukh_037
Hala Ali Khan_007

Contents

1 Abstract:

While talking about CPU Scheduling, many different approaches are used including SJF (Shortest Job First), RR (Round Robin), FCFS (First Come First Serve), MLQ (Multilevel queue), MLFQ (Multilevel feedback queue) scheduling. Multilevel Feedback Queue (MLFQ) algorithm allows the processes to swap between the queues according to their burst time. A time quantum is assigned to each queue and when the burst time of a particular process exceeds the given time quantum it is shifted to the next queue. Multilevel feedback queue scheduling algorithm allows a process to move between queues as it enters the system. Here, the process is assigned to a priority queue and it stays in the queue as long as it is executed. After execution the process' feedback is given to the higher priority queue hence it moves to a higher priority queue. Different queues may have different policies or all the queues can have same policy. In this paper MLFQ is implemented by assigning time quantum to different queues and then turn around time is calculated.

Different algorithms can be used for Real time and Multilevel queue scheduling which improves different factors like waiting, turnaround time and starvation etc. but still a lot of improvement is required. Multilevel feedback queue scheduling removes the starvation issue of lower priority queues.

Keywords: *MLQS (Multilevel Queue Scheduling), MLFQS (Multilevel Feed Back Queue Scheduling), Average Waiting Time, Average Turnaround Time.*

2 Introduction:

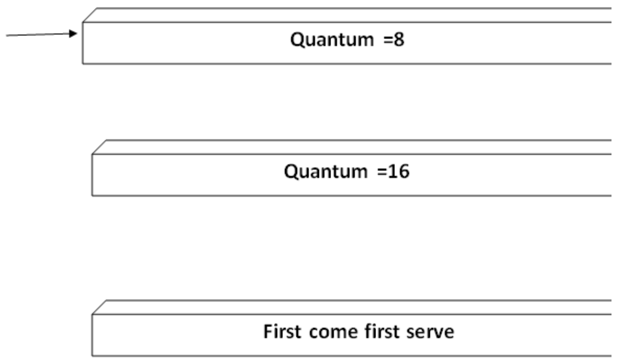
In general round-robin algorithm, different time quantum is used. The Number of ready queue algorithm between the queue, algorithm inside a queue also but the queues may change from system to system. There is a various class of scheduling algorithm which is created for situations like in which the processes are easily divided into the different groups. There is a common division between the foreground(interactive) process and background (batch process) (Chahar & Raheja, 2013). There are two types processes have the different response time, the different requirement of resources so these processes need different scheduling algorithms. The foreground processes have priority(externally defined) over background processes.

In the multilevel queue scheduling algorithm partition the ready queue has divided into seven separate queues. Based on some priority of the process; like memory size, process priority, or process type these processes are permanently assigned to one queue. Each queue has its own scheduling algorithm. For example, some queues are used for the foreground process and some for the background process.

2.1 Multilevel feedback queue scheduling :

MLQS algorithm processes are always put away in one queue in the framework and don't move between the queues. There is some different queue for onscreen or off-screen measures yet the cycles don't move starting with one queue then onto the next queue and these cycles don't change their onscreen or off-screen nature (Yadav & Upadhayay, 2012), these kind of course of action enjoys the benefit of low booking yet it is unbendable in nature.

MLFQS permits an interaction to move between the queues. The processes are discrete with various CPU burst time. Assuming an interaction utilizes a lot of CPU time, it will be moved to the least priority queue. This thought leaves I/O bound and intelligent cycles in the higher priority queue. Essentially, the cycle which stands by too long in a lower priority queue might be moved to a higher priority queue .This type of maturing forestalls starvation



The **multilevel feedback queue scheduler** has the following parameters:

- Number of queues in the system.
- Scheduling algorithm for each queue in the system.
- The strategy used to decide when the process is moved up to a higher priority queue.
- The method used to determine when to demote a queue to a lower priority queue.
- The method which determines the process that will enter in queue and when it is required to be executed.

3 Literature Review:

Scheduling algorithm which is used by CPU plays very important role in the efficiency of multitasking operating systems. Response time (amount of time taken by process to get CPU), waiting time (amount of time a process waits for its complete execution) of the process gets affected by the selection of scheduling policy. This selection may also effect starvation (it occurred if a process waits for its complete execution) (Mulder et al., 2020). FCFS, SJF, Round Robin Scheduling, Priority Scheduling, MLQ and MLFQ are the currently existing CPU scheduling algorithms.

Among these algorithms MLFQ may be prove one of the most powerful algorithm for CPU scheduling. MLFQ is the extension of MLQ scheduling algorithm. While MLQ (Multilevel Queue) Scheduling results from the combination of FCFS and RR scheduling algorithms. Careful attention is required while allocating a process to CPU in order to prevent CPU starvation. Different objectives are being ordered by different CPU scheduling algorithms and may favor one type of process over another (Raheja et al., 2016). Different measure have been suggested to compare the performance of CPU scheduling algorithms. These performance include throughput, waiting time, response time, CPU utilization, turnaround time, scheduler efficiency and context switching. A powerful scheduling algorithm is always that which optimizes all the performance measures. Maximizing CPU usage, minimizing turnaround time, minimizing response time, minimizing the context switching, minimizing throughput, minimizing waiting response these all are optimizing performance measures. The difference MLQ and MLFQ is that in Multilevel Queue, processes are not allowed to switch among queues whereas, in Multilevel Feedback Queue allows the processes in ready queue to execute in several queues according to the scheduling policy applied to each queue and the process burst time. If the execution of a process is not completed in one queue then it is passed to subsequent queue for the execution purpose(Shukla et al., 2010). Selection of time quantum and the burst time of a process for different queues affects both time of each process and execution sequence.

The way to MLFQ scheduling in this way lies in how the scheduler sets priorities. Maybe than giving a fixed need to each work (Dwivedi & Gupta, 2014), MLFQ fluctuates the need of a task dependent on its noticed conduct. In the event that, for instance, a task over and again gives up the CPU while hanging tight for contribution from the console, MLFQ will keep its need high, as this is the means by which an intelligent interaction may act. On the off chance that, all things considered, a task utilizes the CPU seriously for significant stretches of time, MLFQ will lessen its need. Thusly, MLFQ will attempt to find out about measures as they run, and along these lines utilize the historical backdrop of the work to foresee its future conduct (Umadevi et al., 2017). In existing multilevel queue scheduling procedure CPU time will be cut statically between numerous lines where more elevated level lines consistently gets static, most extreme portion of CPU time and are constantly planned first for execution. Where measures in lower level queues are booked when the CPU time allotted to higher level queues or all cycles in higher level queues finishes their execution. This causes high reaction time and starvation issue for measures in lower level queues. Cycles inside various queues are planned utilizing distinctive scheduling procedures. Cycles in undeniable level lines are normally scheduled utilizing cooperative planning procedure, where cycles living in lower level queues is typically scheduled utilizing the conventional FCFS scheduling strategy.

Multiprogramming framework permits different jobs to be executed simultaneously, by exchanging the CPU time among numerous cycles existing in the framework. To plan numerous cycles inside the framework for execution scheduler is required. At whatever point the CPU gets inactive or finishes execution of a cycle, CPU scheduler chooses another interaction from the prepared line to run straightaway. Central processor scheduler requires the planning calculation to pick next interaction to run. There are many planning calculations, each with its own attributes. Multilevel Feedback Queue CPU scheduling algorithm is one of the calculations which we will examine in this paper

First come first served: In this scheduling strategy the positions in prepared line are planned for the request where they show up in the line. Occupation which starts things out gets CPU time first then next, etc. It is a customary planning strategy where every one of the positions have same need.

Round Robin Scheduling: In Round Robin Scheduling procedure numerous cycles dwelling in prepared queue are scheduled utilizing a time sharing configuration where each interaction gets equivalent measure of CPU time. The time or the quantum of time for which the cycles gets CPU can be a fixed time quantum determined by client or can be a variable which can be determined at variable time. After the time quantum distributed to an interaction terminates, CPU is apportioned to the following cycle in the queue. This Scheduling procedure has no starvation issue.

Shortest Job first: The Job in the queue which has the briefest burst time is scheduled first to run and afterward the following most brief occupation gets the CPU time. In this planning procedure if there are enormous number of little burst time measures exist in the framework at that point measures having bigger burst time need to trust that quite a while will get the CPU.

Priority Scheduling: In this scheduling method measures are allocated a priority value where the most highest priority process consistently gets CPU first for execution; need can be appointed relying on different factors like kind of cycle, conduct of interaction and so forth

Fair share scheduling: In Fair share scheduling algorithm every one of the cycles gets decent amount of CPU asset. No interaction has higher priority than others; all are of same level and gets same measure of CPU share.

Multilevel Queue Scheduling: In this scheduling procedure prepared queue is partitioned into various sub-lines and processes are allotted to theses queues relying upon their order which further rely on measure attributes. Each sub queue in Multilevel Queue Scheduling method has a need level allocated to them. Queue containing high priority processes are constantly scheduled first for execution. Cycles in lower priority queues get CPU time solely after the fruition of completion of processes in higher queues. Processes inside various queues are scheduled utilizing diverse scheduling procedures, for example, RR, FCFS and so on

Multilevel feedback queue scheduling: In multilevel feedback queue scheduling likewise ready queue is separated into numerous sub queues however in multilevel feedback queue scheduling strategy cycles can move in between different queues.

4 Methodology:

4.1 Existing Approach:

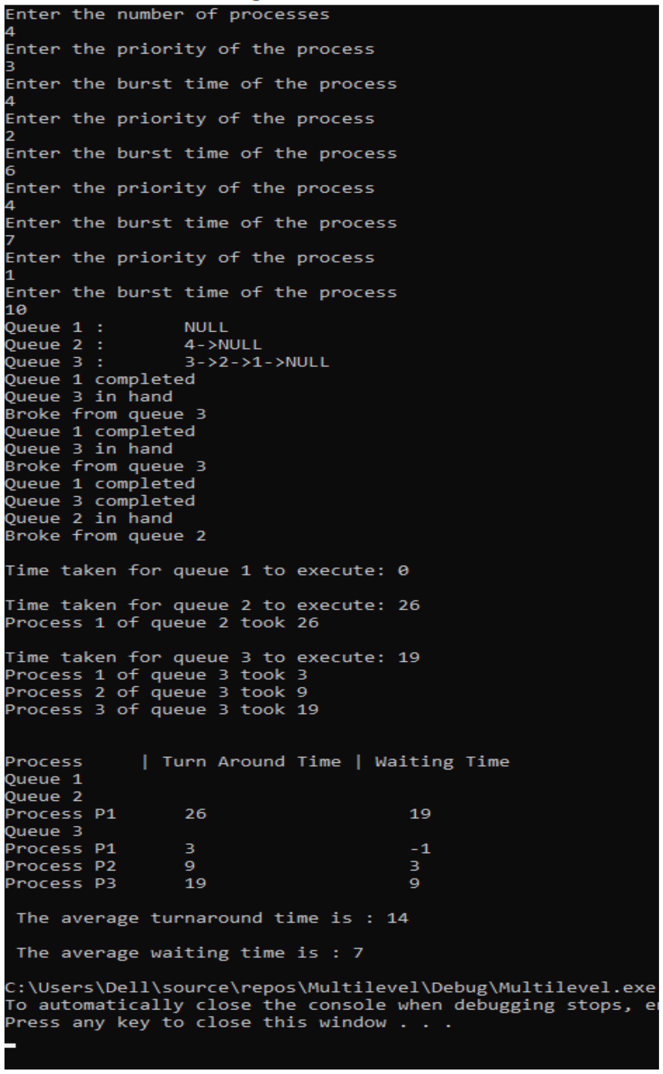
In order to improve the efficiency concept of applying SJF before RR from top queue onwards is used, while another proposed method of implementing dynamic quantum is on basis of burst time of process using some complex functions. Authors of different research articles focused on implementing both of the above approaches but this could not solve the problem entirely. As in MLQ no feedback was provided from the lowest priority level to the highest priority level. This means that the processes that are present in the lowest priority queue keep starving for the processor and the resources. Using the approach of MLFQ the problem of starvation is solved.

4.2 Implementation:

```
// Multilevel.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include <iostream>
using namespace std;

struct process {
    int priority;
    int burst_time;
    int tt_time;
    int total_time = 0;
};

struct queues {
    int priority_start;
    int priority_end;
    int total_time = 0;
    int length = 0;
    process* p;
    bool executed = false;
};

bool notComplete(queues q[]) {
    bool a = false;
    int countInc = 0;
    for (int i = 0; i < 3; i++) {
        countInc = 0;
        for (int j = 0; j < q[i].length; j++) {
            if (q[i].p[j].burst_time != 0) {
                a = true;
            }
            else {
                countInc += 1;
            }
        }
        if (countInc == q[i].length) {

            q[i].executed = true;
        }
    }
    return a;
}



void sort_ps(queues q) {
    //Queue q has to be sorted according to
priority of processes
    for (int i = 1; i < q.length; i++) {
        for (int j = 0; j < q.length - 1; j++) {
            if (q.p[j].priority < q.p[j + 1].priority)
{
                process temp = q.p[j + 1];
                q.p[j + 1] = q.p[j];
                q.p[j] = temp;
            }
        }
    }
}


void checkCompleteTimer(queues q[]) {
    bool a = notComplete(q);
    for (int i = 0; i < 3; i++) {
        if (q[i].executed == false) {
            for (int j = 0; j < q[i].length; j++) {
                if (q[i].p[j].burst_time != 0) {
                    q[i].p[j].total_time += 1;
                }
            }
            q[i].total_time += 1;
        }
    }
}

int main()
{

    //Initializing 3 queues
    queues q[3];
    q[0].priority_start = 7;
    q[0].priority_end = 9;
    q[1].priority_start = 4;
    q[1].priority_end = 6;
    q[2].priority_start = 1;
    q[2].priority_end = 3;

    int no_of_processes, priority_of_process,
burst_time_of_process;
    //Prompt User for entering Processes and
assigning it to respective queues.
    cout << "Enter the number of processes\n";
    cin >> no_of_processes;
    process p1[sizeof(no_of_processes)];

    for (int i = 0; i < no_of_processes; i++) {
        cout << "Enter the priority of the
process\n";
        cin >> priority_of_process;
        cout << "Enter the burst time of the
process\n";
        cin >> burst_time_of_process;
        p1[i].priority = priority_of_process;
        p1[i].burst_time                       =
burst_time_of_process;
        p1[i].tt_time = burst_time_of_process;
        for (int j = 0; j < 3; j++) {
            if (q[j].priority_start <= priority_of_process && priority_of_process <= q[j].priority_end) {
                q[j].length++;
            }
        }
    }

    for (int i = 0; i < 3; i++) {
        int len = q[i].length;
        q[i].p = new process[len];
    }



    int a = 0;
    int b = 0;
    int c = 0;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < no_of_processes; j++) {
            if ((q[i].priority_start <= p1[j].priority) && (p1[j].priority <= q[i].priority_end)) {
                if (i == 0) {
                    q[i].p[a++] = p1[j];

                }
                else if (i == 1) {
                    q[i].p[b++] = p1[j];
                }
                else {
                    q[i].p[c++] = p1[j];
                }
            }
        }
    }

    a--; b--; c--;
    for (int i = 0; i < 3; i++) {
        cout << "Queue " << i + 1 << " : \t";
        for (int j = 0; j < q[i].length; j++) {
            cout << q[i].p[j].priority << "->";
        }
        cout << "NULL\n";
    }


    //While RR on multiple queues is not complete, keep on repeating
    int timer = 0;
    int l = -1;
    int rr_timer = 4;
    int counter = 0;
    int counterps = 0;
    int counterfcfs = 0;
    while (notComplete(q)) {
        if (timer == 10) {
            timer = 0;
        }
        l += 1;
        if (l >= 3) {
            l = l % 3;
        }

        //Process lth queue if its already not executed
        //If its executed change the value of l
        if (q[l].executed == true) {
            cout << "Queue " << l + 1 << " completed\n";
            l += 1;
            if (l >= 3) {
                l = l % 3;
            }
            continue;
        }

        //Finally you now have a queue which is not completely executed
        //Process the incomplete processes over it

        if (l == 0) {
            cout << "Queue " << l + 1 << " in hand\n";
            //Round Robin Algorithm for q=4
            if (rr_timer == 0) {
                rr_timer = 4;
            }

            for (int i = 0; i < q[l].length; i++) {
                if (q[l].p[i].burst_time == 0) {
                    counter++;
                    continue;
                }
                if (counter == q[l].length) {
                    break;
                }
                while (rr_timer > 0 && q[l].p[i].burst_time != 0 && timer != 10) {
                    cout << "Executing queue 1 and " << i + 1 << " process for a unit time. Process has priority of " << q[l].p[i].priority << "\n";
                    q[l].p[i].burst_time--;
                    checkCompleteTimer(q);
                    rr_timer--;
                    timer++;

                }
                if (timer == 10) {
                    break;
                }
                if (q[l].p[i].burst_time == 0 && rr_timer == 0) {
                    rr_timer = 4;
                    if (i == (q[i].length - 1)) {
                        i = -1;
                    }
                    continue;
                }
                if (q[l].p[i].burst_time == 0 && rr_timer > 0) {
                    if (i == (q[i].length - 1)) {
                        i = -1;
                    }
                    continue;
                }
                if (rr_timer <= 0) {
                    rr_timer = 4;
                    if (i == (q[i].length - 1)) {
                        i = -1;
                    }
                    continue;
                }

            }
        }
        else if (l == 1) {
            cout << "Queue " << l + 1 << " in hand\n";
            sort_ps(q[l]);
            //Priority Scheduling
            for (int i = 0; i < q[l].length; i++) {
                if (q[l].p[i].burst_time == 0) {
                    counterps++;
                    continue;
                }
                if (counterps == q[l].length) {
                    break;
                }
                while (q[l].p[i].burst_time != 0 && timer != 10) {
                    cout << "Executing queue 2 and " << i + 1 << " process for a unit time. Process has priority of " << q[l].p[i].priority << "\n";
                    q[l].p[i].burst_time--;
                    checkCompleteTimer(q);
                    timer++;

                }
                if (timer == 10) {
                    break;
                }
                if (q[l].p[i].burst_time == 0) {
                    continue;
                }

            }
        }
        else {
            cout << "Queue " << l + 1 << " in hand\n";
            //FCFS
            for (int i = 0; i < q[l].length; i++) {
                if (q[l].p[i].burst_time == 0) {
                    counterfcfs++;
                    continue;
                }
                if (counterfcfs == q[l].length) {
                    break;
                }
                while (q[l].p[i].burst_time != 0 &&
    timer != 10) {
                    cout << "Executing queue 3 and
    " << i + 1 << " process for a unit time. Process
    has priority of " << q[l].p[i].priority << "\n";
                    q[l].p[i].burst_time--;
                    checkCompleteTimer(q);

                    timer++;
                }
                if (timer == 10) {
                    break;
                }
                if (q[l].p[i].burst_time == 0) {
                    continue;
                }

            }

        }
        cout << "Broke from queue " << l + 1
    << "\n";
      }

      for (int i = 0; i < 3; i++) {
        cout << "\nTime taken for queue " << i
    + 1 << " to execute: " << q[i].total_time <<
    "\n";
        for (int j = 0; j < q[i].length; j++) {
            cout << "Process " << j + 1 << " of
    queue " << i + 1 << " took " <<
    q[i].p[j].total_time << "\n";
        }
      }

      int sum_tt = 0;
      int sum_wt = 0;

      cout << "\n\nProcess       | Turn Around
    Time | Waiting Time\n";
      for (int i = 0; i < 3; i++) {
        cout << "Queue " << i + 1 << "\n";
        for (int j = 0; j < q[i].length; j++) {
            cout << "Process P" << j + 1 << "\t"
    << q[i].p[j].total_time << "\t\t      " <<
    q[i].p[j].total_time - q[i].p[j].tt_time << "\n";
            sum_tt += q[i].p[j].total_time;
            sum_wt += q[i].p[j].total_time -
    q[i].p[j].tt_time;
        }
      }

      cout << "\n The average turnaround time is
    : " << sum_tt / no_of_processes << endl;
      cout << "\n The average waiting time is : "
    << sum_wt / no_of_processes << endl;

    }
```

5 Outcome:

5.1 Results:



```
Enter the number of processes
4
Enter the priority of the process
3
Enter the burst time of the process
4
Enter the priority of the process
2
Enter the burst time of the process
6
Enter the priority of the process
4
Enter the burst time of the process
7
Enter the priority of the process
1
Enter the burst time of the process
10
Queue 1 :       NULL
Queue 2 :       4->NULL
Queue 3 :       3->2->1->NULL
Queue 1 completed
Queue 3 in hand
Broke from queue 3
Queue 1 completed
Queue 3 in hand
Broke from queue 3
Queue 1 completed
Queue 3 completed
Queue 2 in hand
Broke from queue 2

Time taken for queue 1 to execute: 0

Time taken for queue 2 to execute: 26
Process 1 of queue 2 took 26

Time taken for queue 3 to execute: 19
Process 1 of queue 3 took 3
Process 2 of queue 3 took 9
Process 3 of queue 3 took 19


Process     | Turn Around Time | Waiting Time
Queue 1
Queue 2
Process P1      26                 19
Queue 3
Process P1      3                  -1
Process P2      9                  3
Process P3      19                 9

 The average turnaround time is : 14

 The average waiting time is : 7

C:\Users\Dell\source\repos\Multilevel\Debug\Multilevel.exe
To automatically close the console when debugging stops, e
Press any key to close this window . . .
```

5.2 Analysis

In the above source code and output, the user enters:

- Number of processes
- Enter priority of the process
- Burst time of the process

After entering the above mentioned data the program assigns the process to different queues on the basis of priority levels. After using RR and FCFS algorithm it calculates the average turnaround time and average waiting time.

6 Conclusion:

MLFQ is fascinating for the accompanying explanation: rather than requesting previously known information on the idea of a task, it notices the execution of a task and focuses on. Thusly, it figures out how to accomplish the smartest possible solution: it can convey brilliant overall execution (like SJF/STCF) for short-running jobs, and is reasonable and gains ground for long-running CPU-escalated responsibilities (Behera et al., 2021). Consequently, numerous frameworks, including BSD UNIX subordinates, Solaris, and Windows NT and subsequent Windows working frameworks utilize a type of MLFQ as their base scheduler.

Various calculations exists for Real time and Multilevel queue scheduling which improves various components like waiting, turnaround time and starvation and so forth however there is a still extent of progress. Multilevel queue scheduling eliminates the starvation issue of various processes however this procedure isn't reasonable for ongoing processes. To conquer this issue, authors have proposed Multilevel Queue with Priority and Time Sharing for Real time planning which has the properties of MLQS and furthermore real time processes (Hoganson, 2009). In the near future an attempt will be made to improve and carry on the thought exhaustively to build the exhibition of the current framework and to have greatest CPU usage. Processors will able to carry out the plan on actual equipment or in some genuine climate and attempt to eliminate the restrictions that will come during this process.

7 References:

Behera, D. H., Naik, R., & Parida, S. (2021). *Improved Multilevel Feedback Queue Scheduling Using Dynamic Time Quantum and Its Performance Analysis*.

Chahar, V., & Raheja, S. (2013). Fuzzy based multilevel queue scheduling algorithm. *2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 115–120.

Dwivedi, S. K., & Gupta, R. (2014). A simulator based performance analysis of multilevel feedback queue scheduling. *2014 International Conference on Computer and Communication Technology (ICCCT)*, 341–346.

Hoganson, K. (2009). Reducing MLFQ scheduling starvation with feedback and exponential averaging. *Journal of Computing Sciences in Colleges*, *25*(2), 196–202.

Mulder, D., Ssempala, J., Walton, T., Parker, B., Brough, S., Bush, S., Boroojerdi, S., & Tang, J. (2020). Impact of Quantum Values on Multilevel Feedback Queue for CPU Scheduling. *2020 Intermountain Engineering, Technology and Computing (IETC)*, 1–4.

Raheja, S., Dadhich, R., & Rajpal, S. (2016). Designing of vague logic based multilevel feedback queue scheduler. *Egyptian Informatics Journal*, *17*(1), 125–137.

Shukla, D., Jain, S., & Ojha, S. (2010). Effect of data model approach in state probability analysis of multilevel queue scheduling. *Journal of Advanced Networking & Applications (IJANA)*, *2*(1), 419–427.

Umadevi, K. S., Pranay, M. S. S., & Rachana, K. (2017). Multilevel queue scheduling in software defined networks. *2017 Innovations in Power and Advanced Computing Technologies (i-PACT)*, 1–4.

Yadav, R. K., & Upadhayay, A. (2012). A fresh loom for multilevel feedback queue scheduling algorithm. *International Journal of Advances in Engineering Sciences*, *2*(3), 21–23.