A Simplified Study of Scheduler for Real Time and Embedded System Domain

M.V. Panduranga Rao[1], DR. K.C. Shet [2], K. Roopa[3]

[1] Research Scholar NITK, Surathkal Mangalore, India, raomvp@yahoo.com
[2]Professor NITK, Surathkal Mangalore, India, kcshet@yahoo.co.uk
[3] Software Engineer SYMBIAN, Bangalore, India, roopa.sindhe@gmail.com

Abstract

In this research paper, a new algorithm Parametric Multi Level Feedback Queue PMLFQ has been presented for solving the problems and minimizing the response time. In this algorithm, a parametric approach has been utilized for defining the optimized quantum of each queue and number of queues.

One of the most efficient algorithms for real time scheduling is Multi-Layer Queue (MLQ), algorithm that is based on use of several queues. The most serious problem of this method is the not being able to define the optimized number of queues and quantum of each queue for processing. Number of the queues and quantum of each queue affect the response time directly.

The simulations show that the PMLFQ algorithm, regarding response time and waiting time, act better by 10%, than all the other algorithms.

Keywords: *PMLFQ, Scheduling, Queue, round robin, parametric, sjn, deadline, edf, preemption, multilevel queue.*

1. Preamble

1.1 Real-Time Scheduling Algorithms

Scheduling algorithms are one of the most important algorithms in operating systems, which plays a key role. These algorithms have been designed for optimized use of processes from processors.

- Hard real-time systems – required to complete a critical task within a guaranteed amount of time.
    - Scheduler must know how long each task will take to perform *resource reservation*
- Soft real-time computing – requires that critical processes receive priority over less fortunate ones.
    - must have priority scheduling
    - "real-time" processes must have highest
    - "real-time" priorities must not degrade over time
    - dispatch latency must be low

For a Schedulable real time system given
- m periodic events
- event i occurs within period P_i and requires C_i seconds

then the load can only be handled if

$$\sum_{i=1}^{m} Ci / Pi \leq 1 \qquad [1]$$

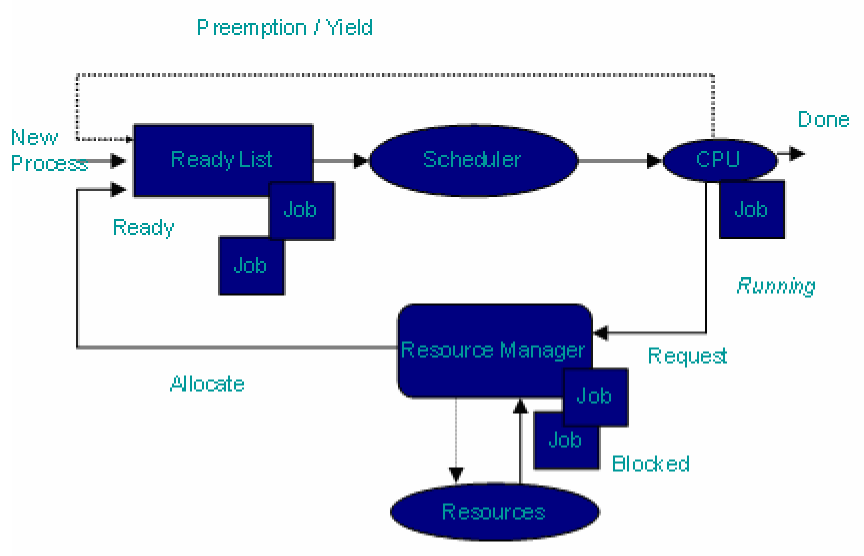

Fig. 1.1 Simpler Processor Scheduling Model

2. Algorithm Evaluation

Define Criteria. Examples:

- Maximize CPU utilization under the constraint that response time <= 1 second
- Maximize throughput so that turnaround time is (on average) linearly proportional to total execution time

Parametric scheduler using the concept of multiple queuing has been implemented. This scheduler also considers an important parameter viz. priority which is a factor to be considered while developing scheduling policies for soft and firm real time systems.

In this scheduler multiple waiting queues have been implemented where each of the ready processes wait for the CPU cycle. The processes go into each of the queues based on their priority levels viz.0-49-low priority, 50-99 medium priority and 100-149 high priority.

The high priority queues are given a greater cpu cycles than the lower priority ones hence avoiding starvation and allowing the higher priority processes more CPU cycles. The scheduling in the queues happens in a round robin fashion minimizing the possibilities of a very high priority process being ignored for long in the queue.

2.1 The Basic conceptual model of Parametric Multilevel Feedback Queue

- Give newly runnable process a high priority and a very short time slice. If process uses up the time slice without blocking then decrease priority by 1 and double time slice for next time.
- Go through the above example, where the initial values are 1ms and priority 100.
- Keep a history of recent CPU usage for each process: if it is getting less than its share, boost priority. If it is getting more than its share, reduce priority.
- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  • number of queues
  • scheduling algorithms for each queue
  • method used to determine when to upgrade a process
  • method used to determine when to demote a process
  • method used to determine which queue a process will enter when that process needs service

- Overhead: number of context swaps.
- Efficiency: utilization of CPU and devices.
- Response time: how long it takes to do something.

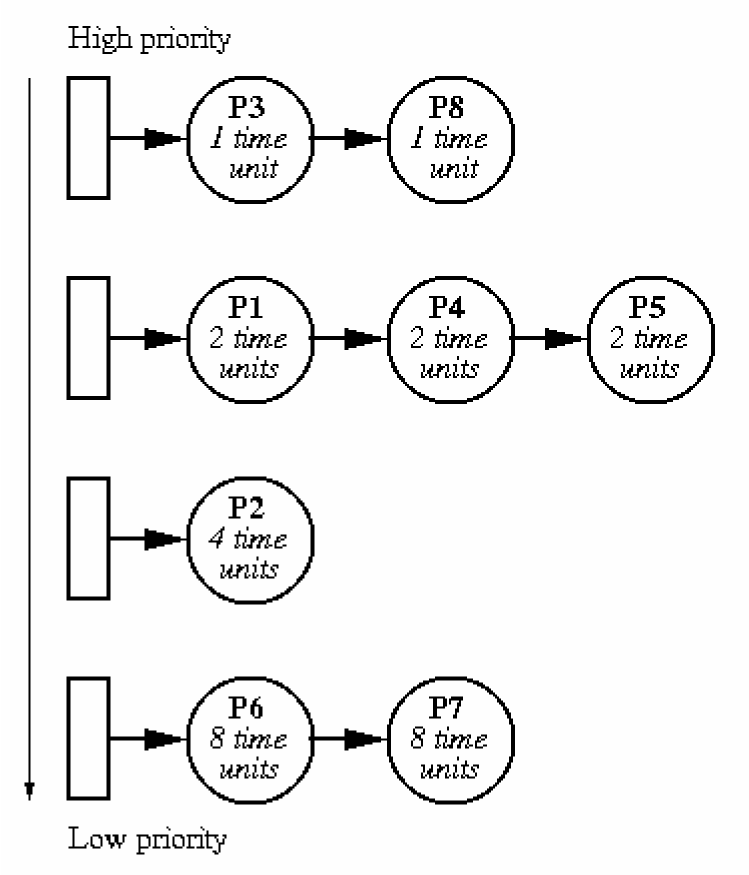

Fig. 2.1 Parametric Multilevel-feedback-queue scheduler

Example:

Three queues:

- $Q0$– RR with time quantum 8 milliseconds
- $Q1$– RR time quantum 16 milliseconds
- $Q2$– FCFS

- Scheduling
  - A new job enters queue *Q0*which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue $Q1$.
  - At $Q1$job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue $Q2$.

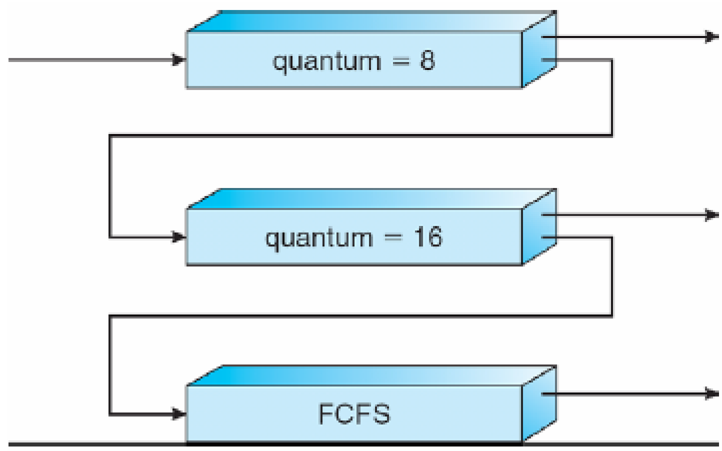

Fig. 2.2 Example of a Parametric Multilevel-feedback-queue scheduler

3. Different available Scheduling Algorithms and their characteristics..

There are several scheduling algorithms which assigns processor to execute processes. There is no scheduling algorithm that works perfectly in all cases, so for a specific application we should consider several parameters such as waiting time, total response time and utilization, in algorithm selection. For example non-preemptive algorithms like FCFS and SJF are suitable when a high throughput system is needed as in batch-processing systems, and preemptive scheduling like MLQ and Round Robin (RR) are used to provide response time and fair dispatching of CPU time as in interactive systems. The simplest scheduling algorithm that is used in most of operating systems is FCFS, which is non-preemptive minimum overhead algorithm. On the other hand, response time is not favored and no emphasis is put on throughput, damaging short and IO processes. The main advantage of this method is that no process starved. This algorithm is used in several operating systems because of its simple implementation and low overhead. FCFS is an unfair algorithm and results in weak average waiting time, while SRT and HRRN provide good response time and high overhead. RR is a fair algorithm with weak average waiting time. Moreover, SJF is an unfair algorithm with the minimum average waiting time and needs prediction. The SRT algorithm damages long processes and is liable to starvation, but because of its prediction, it has better response time in comparison with other algorithms. It is not always possible to predict the execution time of processes and there is a possibility of failure in prediction, so SRT is used theoretically.

In RR the overhead is low and there is no starvation, and this leads to a proper the response time. In this algorithm, the time slice should be selected carefully in such a way that algorithm presents an objective behavior to have suitable overhead. Feedback scheduling algorithm works better than feedback queues in decision making and preemption in a time period schedules the processes, and consequently MLFQ is an approximation of SJF. This algorithm makes the I/O bound processes better without emphasizing on throughput, response time and possibility of starvation. In this approach, the number of queues and the time quantum are chosen by default value. MLFQ is used in interactive and I/O bound systems, the time slice between the queues is generally %80 for foreground and %20 for background. The general scheduler in Unixbased systems is based on MLFQ and some modern operating systems use MLFQ as well [9]. By taking a small quantum for layers, the response time of interactive processes is optimized; on the other hand by taking a larger quantum the throughput of the system is increased.

Generally in PMLFQ scheduling different queues with different priority are used. Each queue has its own scheduling algorithm. All processes are selected from the high priority queues to execute. This method may cause starvation, and generally the low priority queues should have a higher quantum. Because of using queues, this algorithm can be easily implemented to perform the operating systems scheduling. Since this algorithm is used in many cases, its response time should be optimized in comparison with other algorithms.

4. An extensive literature survey for examples of real-time operating systems

Current real-time operating systems can be divided into three main categories.

1. Priority-based kernel for embedded applications,
2. Real-time extensions of timesharing operating systems, and
3. Hard real-time operating systems.

4.1 Priority-based kernel for embedded applications

This category includes many commercial kernels such as VRTX [5,15], pSOSystem [6], OS9 [7], VxWorks [8], Chorus [1], Sumo [2], Harmony [4] and so on that, for many aspects, are optimized versions of timesharing operating systems. In general, the objective of such kernels is to achieve high performance in terms of average response time to external events. As a consequence, the main features that distinguish these kernels are a fast context switch, a small size, efficient interrupt handling, the ability to keep process code and data in main memory, the use of pre-emptable primitives, and the presence of fast communication mechanisms to send signals and events.

4.2 Real-time extensions of timesharing operating systems

This category of operating systems includes the real-time extensions of commercial timesharing systems. For instance, RT-LINUX [9], RT-UNIX [6] and RT-MACH [6] represent the real-time extensions of LINUX, UNIX and MACH, respectively.

The advantage of this approach mainly consists in the use of standard peripheral devices and interfaces that allow to speed up the development of real-time applications and to simplify portability on different hardware platforms. On the other hand, the main disadvantage of such extensions is that their basic kernel mechanisms are not appropriate for handling computations with real-time constraints. For example, the use of fixed priorities can be a serious limitation in applications that require a dynamic creation of tasks; moreover, a single priority can be reducible to represent a task with different attributes, such as importance, deadline, period, periodicity, and so on.

4.3 Hard Real-time operating systems

The lack of commercial operating systems capable of efficiently handling task sets with hard timing constraints, induced researchers to investigate new computational paradigms and new scheduling strategies aimed at guaranteeing a highly predictable timing behavior. The operating systems conceived with such a novel software technology are called *hard real-time operating systems* and form the third category of systems outlined above.

The main characteristics that distinguish this new generation of operating systems include [10]:

- The ability to treat tasks with explicit timing constraints, such as periods and deadlines.
- The presence of guarantee mechanisms that allow to verify in advance whether the application constraints can be met during execution.
- The possibility to characterize tasks with additional parameters, which are used to analyze the dynamic performance of the system.
- The use of specific resource access protocols that avoid priority inversion and limit the blocking time on mutually exclusive resources.

Expressive examples of operating systems that have been developed according to these principles are CHAOS [32], MARS [33,34], Spring [35,36,37], ARTS [38,47], RK [39], TIMIX [40], MARUTI [41,49], HARTOS [42,43], YARTOS [44], HARTIK [14,45,46] and CHIMERA [48]. Most of these kernels do not represent a commercial product but are the result of considerable efforts carried out in universities and research centers.

4.4 Research findings and gaps..

Some of the problems with MLFQ are the number of priority levels of queues, finding a suitable scheduling algorithm for each queue, finding a suitable scheduling mechanism for each queue, assigning time quantum for each queue, assigning initial static priorities, adjusting dynamic priorities, favoring I/O bound processes, differentiating foreground processes and background processes, and considering client against server environment. The MLFQ approach is used in PMLFQ scheduling system in such a way that the response time is decreased and the functionality of the system is improved. The optimum number of queue and the quantum for each queue are found using a fault tolerant mechanism to achieve these goals. As the proposed mechanism considers these objectives simultaneously, they do not have any negative impacts on each others. In PMLFQ scheduling, the operating system can modify the number of queues and the quantum of each queue according to the existing processes.

4.5 Motivation, objectives and Goals

In hard real-time systems, tasks have to be performed not only correctly, but also in a timely fashion. Otherwise, there might be severe consequences. In a hard real-time system, task scheduling algorithms ensure that tasks meet their deadlines. Scheduling involves allocating resources and time to tasks so that the system meets certain performance requirements [4].

Task scheduling in hard real-time systems can be *static or dynamic*. A static approach calculates schedules for tasks off-line and it requires the complete *prior* knowledge of tasks' characteristics. A dynamic approach determines schedules for tasks on the fly and allows tasks to be dynamically invoked. Although, the static approach has a low run-time cost, they are inflexible and cannot adapt to a changing environment or to an environment whose behavior is not completely predictable. When new tasks are added to a static system, the schedule for the entire system must be recalculated which is expensive in terms of time and money. In contrast, dynamic approaches involve higher run-time costs, but, because of the way they are designed, they are flexible and can easily adapt to changes in the environment.

A scheduling problem in a hard real-time system is defined by the model of the system, by the nature of the tasks to be scheduled and by the objectives of a scheduling algorithm. The systems can be uni-processor or multiprocessor, centralized or distributed. The system model is the arrangement of one or more nodes connected by a communication network. A hard real-time system task is characterized by its timing constraints, its precedence constraints and resource requirements. They can be periodic or non-periodic, pre-emptable or non-pre-emptable. The scheduling algorithm may be an optimal algorithm or an approximate or heuristic algorithm. A scheduling algorithm is said to be optimal if, for any set of tasks, it always produces a schedule, which satisfies the constraints of the tasks, whenever any other algorithm can do so. An approximate or heuristic algorithm is necessary whenever an optimal solution is difficult and computationally intractable.

The PMLFQ algorithm model discussed in this paper assumes that each task:

- Repeatedly executes at a known fixed rate (its "period").
- Must end before the beginning of its next period (its "deadline").
- Does not need to synchronize with others in order to execute.
- Can be interrupted at any point in time and replaced by another task in the CPU.
- Does not suspend voluntarily.
- Has zero preemption cost (task-switch times and scheduling-algorithm execution load are neglected).
- Is ready while its assigned processing time is not exhausted. After running out of execution units, the task blocks until its next period.

The scheduling problem for providing precise allocations has been extensively studied in the literature but most of the work relies on some strict assumptions such as full preemptibility of tasks. A responsive kernel with an accurate timing mechanism enables implementation of such CPU scheduling strategies because it makes the assumptions more realistic and improves the accuracy of scheduling analysis.

Similarly, a scheduler that provides good theoretical guarantees is not effective when the kernel is not responsive or its timers are not accurate. Conversely, a responsive kernel with an accurate timing mechanism is unable to handle a large class of time-sensitive applications without an effective scheduler. Unfortunately, these solutions have generally not been integrated: on one hand, real-time research has developed good schedulers and analyzed them from a mathematical point of view, and on the other hand, there are real systems that provide a responsive kernel but provide simplistic schedulers that are only designed to be fast and efficient [1]. Real-time operating systems integrate these solutions for time-sensitive tasks but tend to ignore the performance overhead of their solutions on throughput-oriented applications [7].

In a real-time computer system, correctness depends on the time at which the results are given. In practice, this means that for real-time systems to behave properly, some critical subset of a system's tasks should complete their processing before their deadlines. Failure to do so can lead to human, environmental, and / or economic damages. Compounding the problem is that emerging systems that are distributed, dynamic, and adaptive will put demands that are even more stringent on real-time systems. The success of teams of robots working in hazardous environments, on-board space-shuttle systems, and underwater or outer space autonomous vehicles, for instance, will all be strongly dependent on the timeliness of their computational results.

5. Research Plan

5.1 Problem statement

In a multi-task system, several processes are kept in the main memory and processor is kept active to run a process while the others are waiting. The key to Multi-Programming is scheduling.

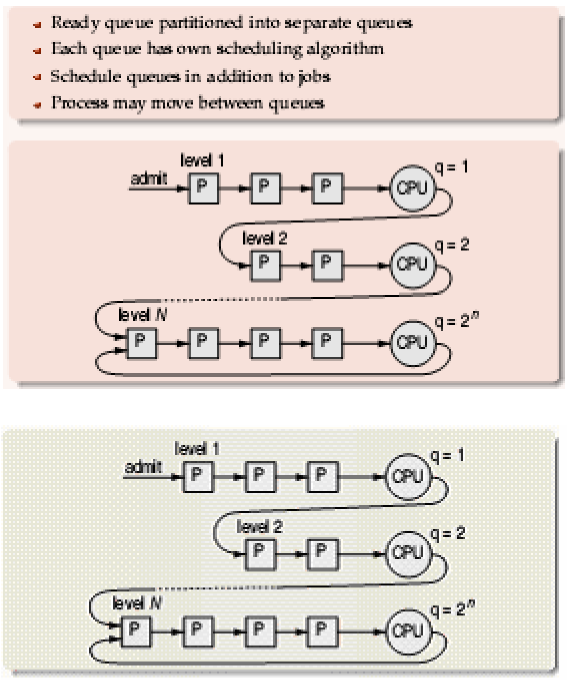

Fig. 5.1 Priority Levels of Parametric Multilevel-feedback-queue scheduler

Here we are concerned with the work on short time scheduling which is the analysis of processes existing at the main memory to be executed by the processor. The goal of this work is allocating time in a way that one or some systematic behavior is optimized. Many criterions have been mentioned for evaluating scheduling in different research papers, that among them we can refer to two important criteria:

1]- from the viewpoint of user,

2]- from the viewpoint of system.

Each of these two categories has many criteria to be discussed. In time-sharing, we try to reduce the response time variation because the goal of some operating systems is providing all users' services are in a suitable way and minimizing the response time for users. As it is known, Multi layer Queue (MLQ) scheduling is designed from some prepared queues and the respective processor of each queue, MLFQ scheduling acts the same as MLQ and process can move dynamically in different queues. So processes that need a large amount of CPU time are sent to the lower queues and process requiring I/O bound or related to interactive process are sent to queues with higher priority of response. PMLFQ scheduling algorithm is focused on total time, response time and application of priority, but it is tried not to apply the negative influences over the mentioned criteria. Our main job in this research progress is optimizing response time of MLFQ by using a parametric approach [9]. The MLFQ scheduling organizes the queues to minimize the queuing delay and optimize the queuing environment efficiency.

5.2 Background significance and Features of Parametric Scheduler

The scheduler is the most important part of any kernel. It determines whether to run a new task. If so, it suspends or stops the current task and resumes or starts the new task. The parametric scheduler is a preemptive scheduler. It runs the longest-waiting task at the highest occupied priority level.

Preemptive Scheduling is the best algorithm for embedded systems. Despite this, many well-known RTOS's (e.g. WinCE, embedded NT, Linux, and PharLap ETS) utilize priority time slicing. Under this algorithm, when a higher priority task becomes ready to run, it must wait until the end of the current time slice to be dispatched. Hence response time is governed by the granularity of the time slice. However, if the granularity is set too fine, the processor spends too much time thrashing - i.e. interrupting the current task to find out if a higher priority task is waiting. This pretty much precludes hard real time response without using an over-kill processor.

PMLFQ uses preemptive scheduling. This means that, as soon as a higher priority task becomes ready to run, it preempts the current task and runs. For safety, PMLFQ does permit the current task to lock the scheduler if necessary (i.e. if it is in a critical section of code.) The programmer controls locking.

Priority Levels: Some kernels (e.g. uC/OS) require each task to have a unique priority. This is limiting, because, within a group of equally important tasks, it is usually better for the task that has waited the longest, to run first. One task per priority level does not permit this. Allowing multiple tasks at the same priority level also permits round robin scheduling among those tasks. This is a good way to share resources equally among the lowest priority tasks in the system. PMLFQ also permits time slicing among the lowest priority tasks - this is even more equal.

Scheduler Locking: PMLFQ allows the current task to lock the scheduler. Many kernels do not provide this feature. Why is it important? With the addition of locking, there are three ways to protect access to a resource: (1) disabling interrupts, (2) semaphores, and (3) locking.

The first method is the only way to protect a resource shared between foreground and background. However, it causes interrupt latency and should be used as little as possible.

Semaphores are the traditional method to protect resources shared between tasks. Semaphores are resource-specific and do not add interrupt latency. However, they do cost a fair bit of processor overhead - typically on the order of 100 instructions to signal and test a semaphore. Hence using semaphores is inefficient for short, critical sections of code.

This is where locking has an advantage. Locking + unlocking require only about 10 instructions, if no preemption results, and about 50, if preemption does result. Obviously, processor overhead is less. Also, task latency is less. Lock and unlock are very short operations, so if the critical section is short, the sum of these plus the critical section of code will be less than the code in the signal and test functions.

Layered Ready Queue

PMLFQ is unique in having a layered ready queue. The ready queue is where ready-to-run tasks are enqueued. Nearly all kernels, other than PMLFQ, utilize a single, ordered ready queue. To enqueue a new task requires searching from the beginning of the queue until the last enqueued task, of equal priority, is found. Then the new task is enqueued at that point. Obviously, if there are many ready tasks, this could take a long time.

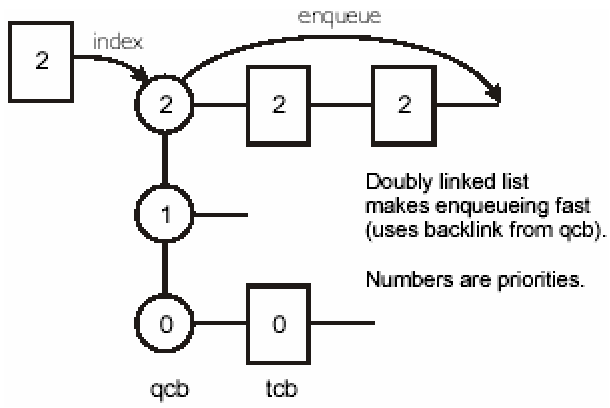

Fig. 5.2 Layered ready queue

For PMLFQ, we observed that since PMLFQ tasks are permitted to share priority levels (which is not true for some kernels) the typical embedded system needs no more than 5-10 priority levels [3]. (How finely can you slice and dice the relative importance of each task?) Hence, why not have a separate queue for each priority level? This is how the PMLFQ ready queue is implemented. Each level is headed by a queue control block (qcb). The qcb's are contiguous in memory and in order by priority. Hence enqueueing a task is but a two-step process: (1) index to the correct qcb, based upon the priority of the new task, and (2) follow the backward link of the qcb to the end of the queue and link in the task. This fast, two-step process takes the same amount of time regardless of how many tasks are in the ready queue. There could be 10 or there could be 100,000. It would not make any difference!

Finding the next task to dispatch is also very fast because PMLFQ maintains an rq_top pointer. rq_top points at the tcb of the next task to run.

5.3 Designing the parametric multilevel queue scheduler

As it was mentioned before, in MLFQ the operating system builds several separate queues and specifies the quantum for each queue. Generally in this method, all processes end in the mentioned queue and move out of system. In these methods, the number of queue and the quantum size are specified while the process is running, so the operating system has no role in controlling the number of queues and amount of each layer’s quantum.

In PMLFQ, we start with indefinite numbers of queues initially. An initial value of quantum is used for each queue. When a queue is being analyzed, its quantum value is defined by ***I*q***, where ***q*** is the initial value of quantum and ***I*** is the number of queues being considered [5]. For defining the numbers of layer and quantum of each layer is described in [2]. When the number of required queues and the average response time are specified based on the initial quantum of each layer, the quantum of queues should be modified in such way that the average response time of the processes are minimized [7].

According to the changes in the quantum of each queue, the movement of the processes to the lower queues is changed. So the processing time of the processes in lower layers is changed and as a result the quantum of lower layers affects the average response time. The optimized quantum has not been defined for lower queues and the average response time is related to the functionality of the whole system. Consequently the relation of the average response time and the quantum of a specified queue are not easily formulated [6].

To find the effect of the quantum changes, a queue should be selected and its quantum has been changed in such a way that the minimum number of processes has been moved to the lower queues [1].

Now, suppose that we have n queues in the default mode. We begin to change the quantum of layer n, since when the last queue is selected and its quantum is increased, there is no other queue to be eliminated. If the quantum of this queue is reduced a queue will be added. In this case we repeat this procedure for the newly added layer. After updating the quantum of the last queue, we continue with the previous queue, i.e. *n-1*, and change the quantum of it. The optimized average response time is specified by changing the queue *n-1*.

In this step, since there is a queue that is lower than the queue is being studied, and due to the changes made on the processes of the last queue after updating *n-1*, the optimized quantum of the last queue should be redefine [8]. Generally, when the optimized quantum of each layer is found, the quantum of lower levels should be updated. Finally, the best average response time can be calculated using the optimized quantum of each layer.

5.4 How to Optimize Average Response Time

In this step, for the beginning of the change of quantum, we must choose a queue, by the change of quantum of which, the least change in the transfer of processes to the lower queues takes place, because if we increase the quantum of a queue so much that no processes is given to the last queue, that queue is eliminated from studying in that period. By the reduction of quantum of that queue, there would be no possibility of moving processes to that queue, and we would face an addition of one queue to the number of queues being studied. So, we must begin to change the quantum of queue, from a queue that has the least of these changes. Now, supposing that we have ”n” queues in the default mode, we begin the job. For the start of changing quantums we begin from the layer n, because by choosing the last queue and by increasing its quantum, there is no other queue to be eliminated. It is only probable that by reducing the quantum of this queue, a queue will be added to the number of queues being studied, in this case we repeat the same job for the newly added layer.

If we want to devise a general rule or obtaining the optimized quantum, we must say; after getting the optimized quantum of each layer, the quantum of lower levels must be updated. So that, we reach at best average response time coming from the optimized changes on quantum of each layer [2].

The PMLFQ outputs are used to calculate the average response time. To equalize the entrance arrival time of these inputs with entrance time of average response time a delay function is used.

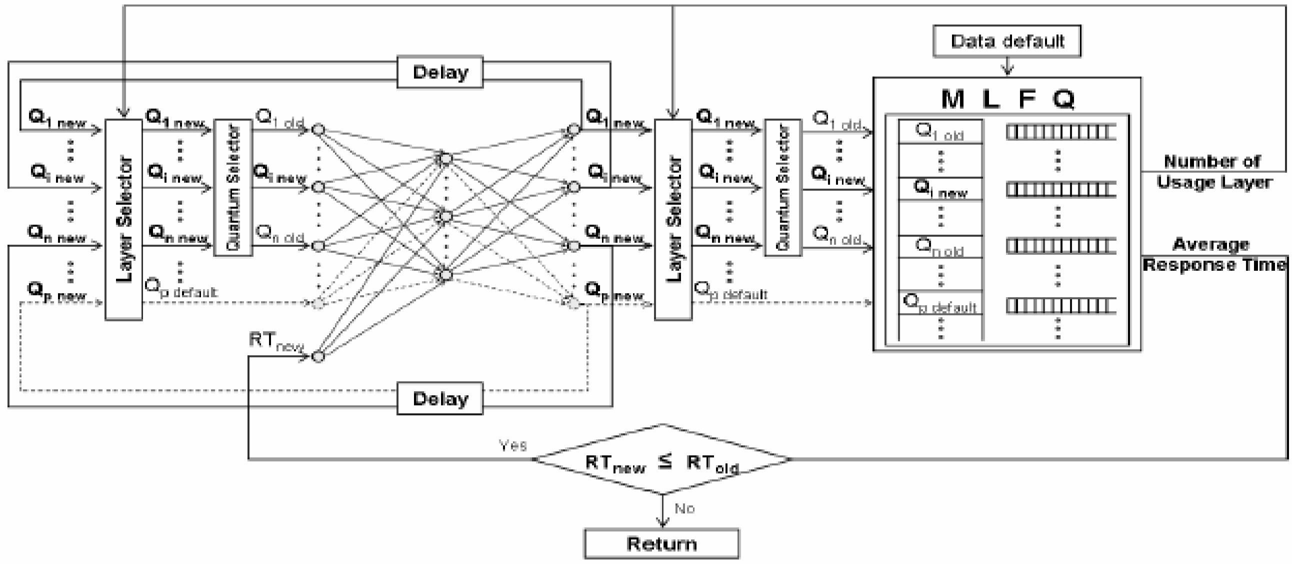

Fig. 5.4 Defining optimized quantum for the queue by PMLFQ function

Fig. 5.4 shows a schematic view of the function to find the optimized quantum of the queue ***I***, and the way in which the quantum is fed and also how to limit the number of queues. When the new average response time is found, it is compared with the former one. If it is less than the previous one, the new value is selected as the input of next stage to optimize the average response time. If the new value of the average response time is grater than the previous one, it means that the optimized average response time has been found.

It should be guaranteed that the calculated quantum is selected as the quantum of the specified queue. If the quantum of the other queues is changed, we should find their optimized quantum again. The pseudo code of the algorithm has been shown below.

5.5 PMLFQ algorithm:

1- Produce arrival time and service time for *n* process randomly using distribution function.

2- Get average response time, waiting time and maximum required layer in first stage and set the power quantum for each layer.

3- For each layer (*i*=*n* down to 1) update the value of queue quantum according to the maximum number of layers and average response time.

3.1- Find the optimum value of queue according to other queue quantum and the average response time that is found in the previous stages.

3.2- For each layer (*j*=*i+1* to *n*) repeat the step 3.2, consider the changes in other queue and update the quantum.

5.6 Class table of parametric queue scheduler

SCHEDULER_PARAMETRIC_QUEUE
Pid ,value Pri_first,pri_temp

Get_process_d() Get_arrival_time() Get_burst_time() Get_turn_around_time() Get_waiting_time() Set_process_id() Set_arrival_time() Set_burst_time() Set_turn_around_time() Set_waiting_time() Process() Scheduler()

5.7 Code snippet for scheduler

```
#include"scheduler_Parametric_queue.cpp"

struct priority
{
        int pid;
        int value;
        struct priority *next;
};

class parametric_triple_queue : public scheduler
{


      priority *pri_first,*pri_temp;
      priority *pri_second,*pri_temp_second;
      priority *pri_third,*pri_temp_third;

      int high_quantum,medium_quantum,low_quantum;
      public:
            int set_values (int,int,int,int);
            void set_quantum (int);
            int compute ();
            void destroy ();
};
```

Fig. 5.7 Parametric queue scheduler.

5.8 Simulation and experimental results of parametric queue scheduler

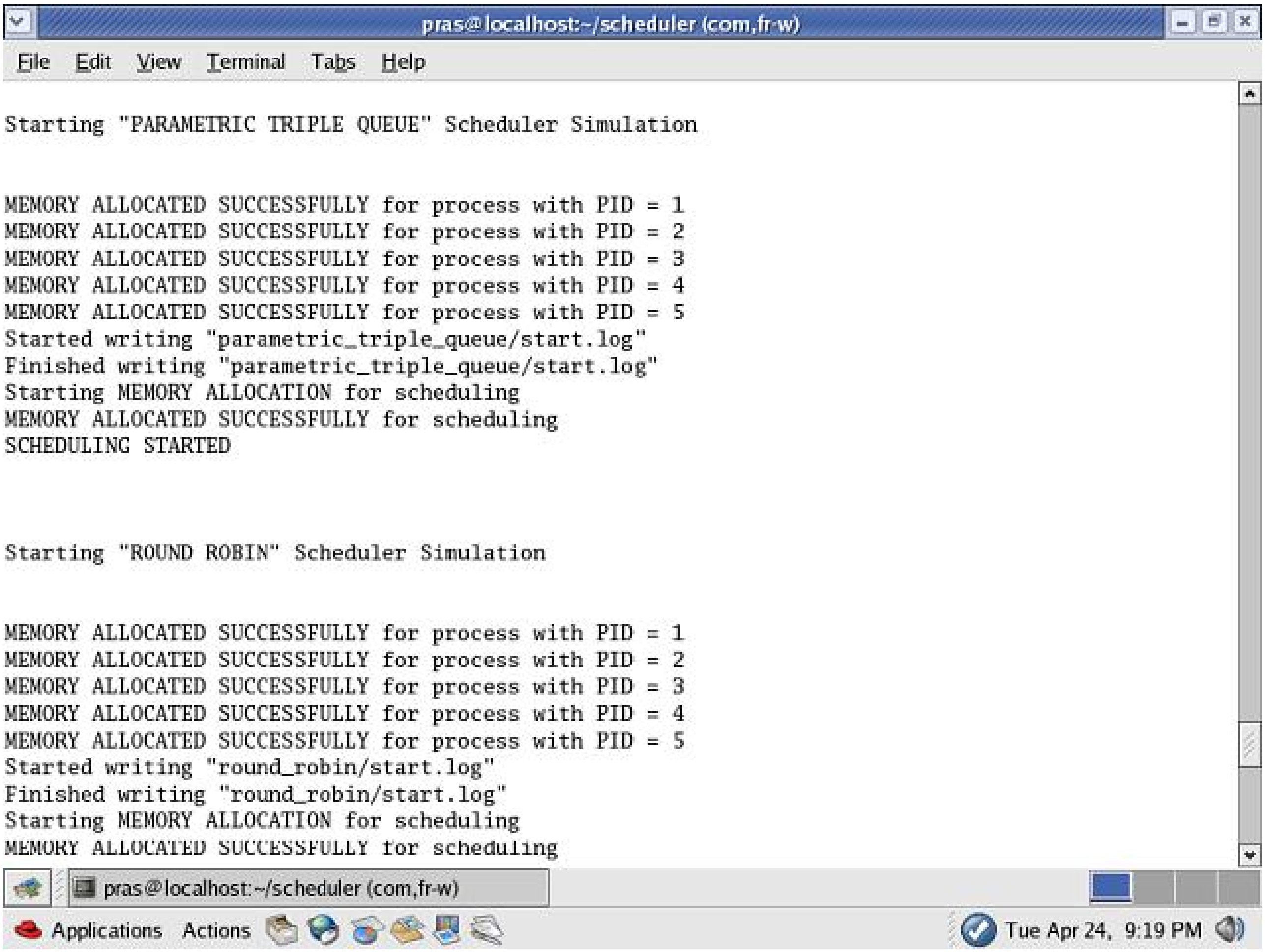

Fig. 5.8 Simulation of parametric queue scheduler.

Since the process arrival time is randomly distributed, we used discrete event technique simulation. So the system state has been changed when an event occurred during the simulation time. At first, we sort the processes by their arrival time and then find the first process to handle and provide its service. The PMLFQ average response time is better by 10% than the other scheduling algorithms [10].

6. Conclusion

- ✓ As tasks interact, integrated resource scheduling is also necessary. Algorithms exist that support special cases, in which decisions deal with imprecise results, task-completion value, and so on. However, no algorithm is good for all cases.
- ✓ Real-time scheduling may seem unnecessary, but as the project's complexity and size increase, it's the only way to guarantee proper system behavior. It is certainly more predictable than ad hoc techniques.
- ✓ Since Number of the queues and quantum of each queue affect the response time directly. I continuously review the PMLFQ algorithm for solving these problems and minimizing the response time and waiting time.
- ✓ The PMLFQ is aimed to present an intelligent algorithm to optimize both the *average response time* and the *waiting time*. When the response and waiting time optimization is aimed, the PMLFQ shows a good performance.
- ✓ We tried to decrease the overhead of the system, however we have a little overhead to be calculated and compared with the response time. With more researches it can avoid starvation in

PMLFQ. This algorithm can also be used on distributed system, in an effective way that the research in this field is still being continued.

7. Acknowledgment

I am grateful for sincere advice and care by the guide and other superiors in achieving this research paper. My gratitude goes to the people, who previously succeeded in implementation of general scheduler mechanisms, process communication, event analysis, interrupt handling etc., and making it available for further development.

References

[1] Jensen et al. 02a, *Guest Editors*, “Introduction to Special Section on Asynchronous Real-Time Distributed System”, E. Douglas Jensen and Binoy Ravindran, *IEEE Transactions on Computers*, August 2005.

[2] Clark et al. 04, “Software Organization to Facilitate Dynamic Processor Scheduling”, Raymond K. Clark, E. Douglas Jensen, and Nicolas F. Rouquette, *Proc. of the IEEE Workshop on Parallel and Distributed Real-Time Systems*, Jan 2007.

[3] W.T. Chan, T.W. Lam and K.S. Mak, “Online Deadline Scheduling with Bounded Energy Efficiency”, *Proceedings of the 4th Annual Conference on Theory and Applications of Models of Computation (TAMC),* 416-427, 2007.

[4] A. Streit. “Evaluation of an Unfair Decider Mechanism for the Self-Tuning dynP Job Scheduler”, In Proceedings of the *13th international Heterogeneous Computing Workshop (HCW) at IPDPS*, pages 108 (book of abstracts, paper only on CD). IEEE Computer Society Press, 2004.

[5] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son and M. Marley, “Performance Specifications and Metrics for Adaptive Real-Time Systems,” *IEEE Real-Time Systems Symposium*, Orlando, FL, Dec 2006.

[6] Jensen 04, *Timeliness in Mesosynchronous Real-Time Distributed Systems,* E. Douglas Jensen Proc. of the 7th IEEE International Symposium on Object-Oriented Real-Time Computing, May 12-14, 2006.

[7] J. Wang and B. Ravindran, “Time-utility function-driven switched ethernet: Packet scheduling algorithm, implementation, and feasibility analysis," *IEEE Trans. on Parallel and Distributed Systems,* vol. 15, no. 2, pp. 119-133, February 2007.

[8] M.V. Panduranga Rao, Dr. K.C. Shet, and K. Roopa, “Efficient and Predictable Process Scheduling”, *International Conference on Contemporary Computing – MACMILLAN JOURNAL, 2008.* IC3-2008, Organized by University of FLORIDA and Jaypee Institute of Information Technology University. Page(s):131–140, 7-9 August 2008, *Noida, New Delhi, INDIA.*

[9] M.V. Panduranga Rao, Dr. K.C. Shet, R. Balakrishna and K. Roopa, “Development of Scheduler for Real Time and Embedded System Domain”, *22nd IEEE International Conference on Advanced Information Networking and Applications - Workshops, WAINA '08, 2008.* FINA 2008, Fourth International Symposium on Frontiers in Networking with Applications. Page(s):1 – 6, 25-28 March 2008, *Gino-wan, Okinawa, JAPAN.*

[10] M.V. Panduranga Rao, Dr. K.C. Shet, K. Roopa and K.J. Sri Prajna, “Implementation of a simple co-routine based scheduler“, In *Knowledge based computing systems & Frontier Technologies NCKBFT, MIT Manipal, Karnataka,INDIA.* 19th & 20th Feb 2007.

Article received: 2009-01-21