A Simplistic Study of Scheduler for Real-Time and Embedded System Domain

M.V. Panduranga Rao
Research Scholar, NITK, Surathkal Mangalore, India
+919844002757
raomvp@yahoo.com

K.C. Shet
Professor, NITK, Surathkal Mangalore, India
+919845237101
kcshet@rediffmail.com

ABSTRACT

This paper presents a new algorithm *Parametric Multi Level Feedback Queue [PMLFQ]*. The algorithm PMLFQ has been presented for solving the problems and minimizing the response time. In this algorithm, a *parametric approach* has been used for defining the optimized quantum of each queue and number of queues.

Performance evaluation and summary of Scheduling Algorithms are done. The criteria for comparing scheduling algorithms, its background significance and features of parametric scheduler are detailed. Defining optimized quantum for the queue by PMLFQ function and designing of the parametric multilevel queue scheduler is expressed. Simulation and experimental results of parametric queue scheduler are discussed briefly.

Categories and Subject Descriptors

C.3 [Special-purpose and application-based systems]: Real-time and embedded systems – Scheduler.

D.4.1 [operating systems]: Process Management – Scheduling.

D.4.7 [operating systems]: Organization and Design -- Real-time systems and embedded systems.

General Terms

Algorithms, computer science education, operating system, theory, documentation, performance, design and experimentation.

Keywords

PMLFQ, CPU scheduling, queue, round robin, FCFS, parametric, sjf, deadline, feedback, preemption, multilevel queue and process.

1 INTRODUCTION

Scheduling is the problem of assigning a set of processes (tasks) to a set of resources subject to a set of constraints. Examples of scheduling constraints include deadlines (i.e., job i must be completed by time T), resource capacities (i.e., limited memory space), precedence constraints on the order of tasks (i.e., sequencing of cooperating tasks according to their activities), and priorities on tasks (i.e., finish job P as soon as possible while meeting the other deadlines).

Scheduling algorithm is 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 number of times.
  - Scheduler must know how long each task will take to perform *resource reservation*
- Soft real-time systems – requires the critical processes receive priority over less fortunate ones.
  - must have priority scheduling
  - "real-time" priorities must not degrade over time
  - dispatch latency must be low
- Hard Real-Time Systems

The hard real-time system guarantees that a critical task will be completed within a specified number of times. Each process and it's allowed running time are presented to the system, and it determines about whether or not it is possible to complete the process in the showed number of times. To do this, the system looks at the sum of the time it takes to run each of the various operating system functions that the process uses and compares it to the allowed run time of the process. If it is possible to complete the task, then the system accepts it. Otherwise, it is deemed impossible and rejected by the system.

The method described above is intended for the hard real-time system consisting of special-purpose software running on hardware that is dedicated to a critical process. These specialized systems lack the full functionality offered by modern computers and operating systems.

Hard real-time guarantees cannot be made on systems with secondary storage or virtual memory. There is an unpredictable variance in the number of times it takes to execute a particular process on one of these systems. It may take significantly longer to run a process on a machine with virtual memory if the effective access time is high.

Whenever a page fault occurs, the access time increases. This is shown by

Effective access time = $(1 - p) * ma + p *$ (page fault time) → [1]

Where p is the probability of a page fault and ma is the memory access time [9].

- Soft Real-Time Systems

The soft real-time system attempts to complete its tasks by their respective deadlines, but does not guarantee this will happen.

Critical processes receive higher priority over those that are not critical. In a time-sharing system with soft real-time capabilities, those processes that are real-time must be given the highest priority and this priority must not degrade over time. However, the priority of the processes that are not real-time in the system can decrease over time. This could cause longed, delays for these processes, sometimes even starvation. The dispatch latency in the soft real-time system must be as small as possible. A small switching time will provide more time for each process to run.

Dispatch latency times can become very long in an operating system that requires a system call to complete or an I/O block to occur before performing a context switch. This is the case in the UNIX operating system [2]. One way to solve this issue is to make the system calls preemptible, by inserting preemption points in the system calls. A preemption point is a location in the code that is not critical; interrupting the process at this point will not harm its data. Whenever a preemption point is reached, the system checks to see if a higher-priority process needs to be run. If it does, control of the CPU is given to the process. Once it is finished, the CPU goes back to the initial preemption point and continues execution.

Priority inversion occurs in the soft real-time system when one or more processes are using resources needed by a high-priority process. In this situation, the lower-priority processes are given a priority equal to the high-priority process until they are done using the desired resource or resources. Once the resource is released, each process is given its previous priority level. This technique is known as priority-inheritance protocol [6], and it prevents high-priority processes from spending a lot of time waiting for resources to become available.

1.1 Scheduling Policies

CPU scheduling strategically allocates the CPU to a process based on a specified criterion. There are many different methods of selecting which process will be given control of the CPU. Each of these methods follows a different scheduling algorithm and has advantages and disadvantages.

Simplifying Assumptions

- One process per user
- One thread per process
- Processes are independent

Researchers developed these algorithms in the 70's when these assumptions were more realistic, and it is still an open problem, how to relax these assumptions

1.2 Overview of Scheduling Algorithms

Most operating systems today support multiple processes running at the same time. Since only one task could be performed at a time, by each CPU in the system, this force the operating system to make decisions about which process to run and when. The methods of scheduling implemented depend on the application at hand. In real-time embedded systems, the performance and predictability of the scheduler is of most importance. High performance makes the use of cheaper components possible and decreases the system response time.

To maximize system performance it's in the best interest of the operating system to make the CPU idle-time as low as possible. However, in the real-time environment, this problem becomes more complex, because it has to react very fast on external events, as well as uphold deadlines to prevent damage or personal injuries. The common list of scheduling algorithms is as follows.

- **FCFS:** First Come First Served- Not fair, and average waiting time is poor.
- **Round Robin**: Use a time slice and preemption to alternate jobs. Fair, but average waiting time is poor.
- **SJF**: Shortest Job First- Not fair, but average waiting time is minimized presuming we can accurately predict the length of the next CPU burst. Starvation is possible.
- **Multilevel Queuing**: Round robin on priority queue. An implementation (approximation) of SJF.
- **Lottery Scheduling**: Jobs get, tickets and scheduler randomly picks winning ticket. Fairer with a low average waiting time, but less predictable.

1.3 Criteria for Comparing Scheduling Algorithms

- **CPU Utilization** The percentage of time that the CPU is busy.
- **Throughput** The number of processes completing in a unit of time.
- **Turnaround time** The length of time it takes to run a process from initialization to termination, including all the waiting time.
- **Waiting time** The total number of times that a process is waiting in the ready queue.
- **Response time** number of times it takes from when a request was submitted until the first response is produced, not output.

1.4 Optimization Criteria in Real Cases…

- Minimize the variance in the response time
- Minimize the average waiting time
- Minimize turnaround time
- Maximize CPU utilization
- Maximize throughput

In 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

$$\sum_{i=1}^{m} Ci / Pi \leq 1$$ → [2]

The purpose of CPU scheduling is to maximize the utilization of the CPU. To do this, a process should be running always. For example, if a process is waiting for some event to occur before it is able to continue execution, then it should not have control of the CPU. If it does, then the CPU is being wasted.

1.5 Simple Process Model

- The application is assumed to consist of a fixed set of processes
- All processes are periodic, with known periods
- The processes are completely independent of each other
- All system's overheads, context-switching times and so on are ignored (i.e, assumed to have zero cost)
- All processes have a deadline equal to their period (that is, each process must complete before it is next released)
- All processes have a fixed worst-case execution time

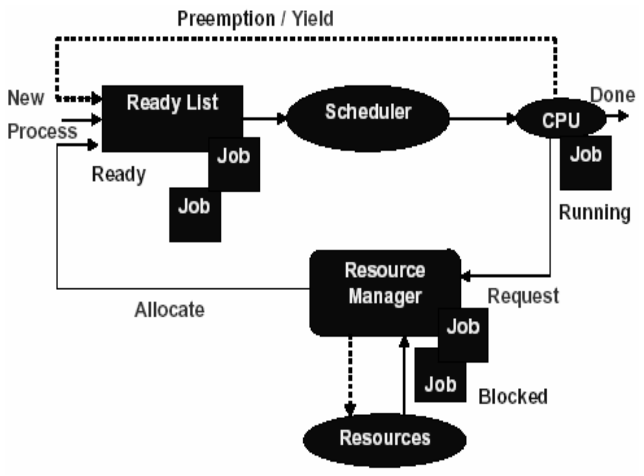


Figure 1. Simpler Processor Scheduling Model.

1.6 Process Terminology and States

A process is the execution of a program. It is often thought of as the unit of work in a computer system, because the operating system is constantly running processes.

1.6.1 PROCESS CREATION

Processes can be created by the kernel of the operating system or by the user. An example of a process created by the user is Microsoft Word. This process is only created once the user decides to launch Word. Processes that are created by the kernel take care of various system tasks. An existing process could also create a process, which is then referred to as the child process. The creating process is labeled the parent process.

1.6.2 PROCESS STATE

There are five possible states that a process may be in: new, running, waiting, ready, or terminated. These states are described below.

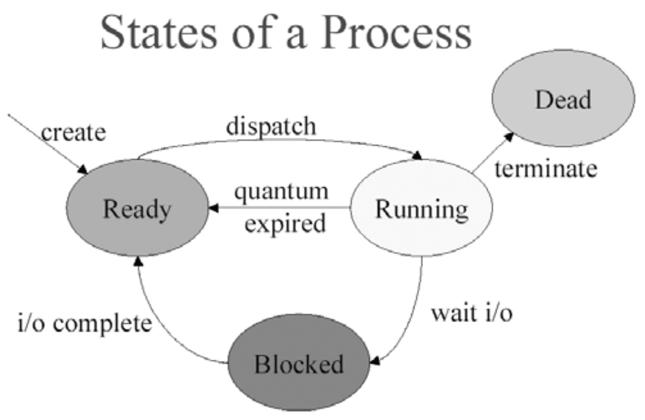


Figure 2. Process terminology and states.

- New – The process is being formed.
- Running – Instructions from the process are being executed.
- Waiting – The process is waiting for an event to happen before it can resume its execution.
- Ready – The process is ready to be run, but is waiting for the processor.
- Terminated – The process has completed its execution.

The CPU scheduler, which is part of the operating system of a computer, manages the allocation of the CPU among processes. A process is said to be running in the running state if it is currently using the CPU. A process is said to be ready in the ready state if it could use the CPU if it is available. A process is said to be blocked in the waiting state if it is waiting for some event to happen, such as an I/O completion event, before it can proceed. Various events can cause a process to change state. For example, when the currently running process makes an I/O request, it will change from running state to waiting state. When its I/O request completes, an I/O interrupt is generated and then that process will change from waiting state to ready state. For a single CPU system, only one process may be running at a time, but several processes may be ready and several may be blocked. All ready processes are kept on a ready queue. All blocked processes are placed on an I/O queue for the requested I/O device.

The scheduler uses a scheduling algorithm to decide which process from the ready queue to run when and for how long. Long (100ms) time slices make TLB [translation look-aside buffer] and cache state flushing infrequent, imposing minimal overhead on CPU-bound processes. The scheduler favors interactive processes by lowering the priority of processes as they consume CPU time and by preempting processes before their quanta expire if a higher priority sleeping process wakes up.

2 THE BASIC CONCEPTUAL MODEL OF PARAMETRIC MULTILEVEL FEEDBACK QUEUE

This algorithm works like the multilevel queue, but it is able to move a process from one queue to another. When a process uses too much CPU time, then it may be moved to a queue with a lower priority.

- 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.

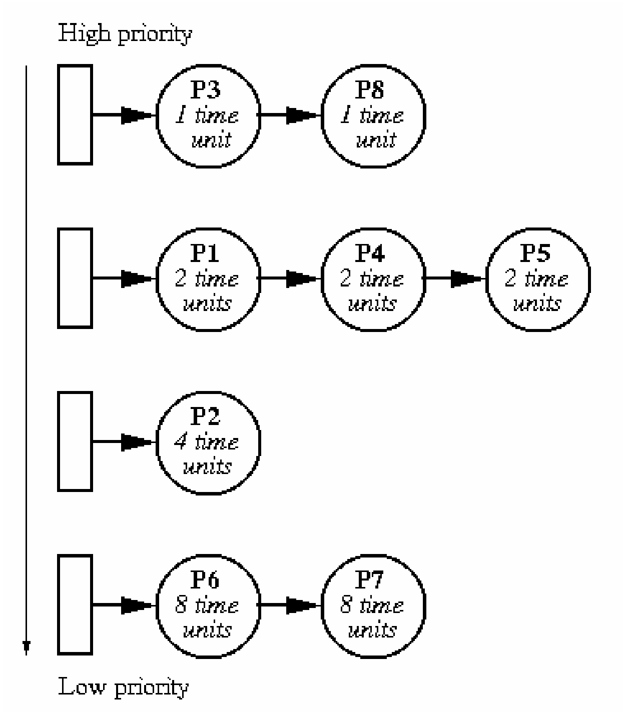


Figure 3. Parametric Multilevel-feedback-queue scheduler.

- A process could move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following guidelines;
    - 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 need service
- Overhead: number of contexts swaps.
- Efficiency: utilization of CPU and devices.
- Response time: how long it takes to do something.

Example:-

Three queues:

- *Q*0– RR with time quantum 8 milliseconds
- *Q*1– RR time quantum 16 milliseconds
- *Q*2– FCFS

Scheduling

- A new job enters, queue Q_0 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 Q_1.
- At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue *Q*2.

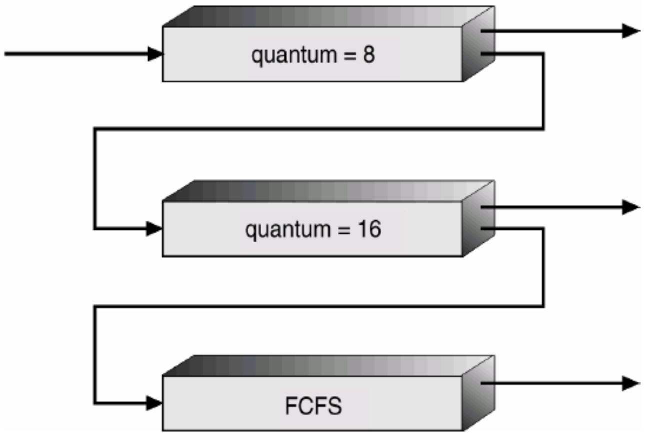


Figure 4. Example of a Parametric Multilevel-feedback-queue scheduler.

In a multilevel queue-scheduling algorithm, jobs are permanently assigned to a queue on entry to the system. Jobs do not move between queues and this can create starvation if the jobs running are long duration jobs. We have employed multilevel feedback queue scheduling as shown in figure 4, since it allows a job to move between queues. The idea is to separate processes with different requirements and priorities. If a job use too much CPU time or is very data intensive, it will be moved to a higher-priority queue. Similarly, a job that wait too long in a lower-priority queue may be moved to a higher priority queue.

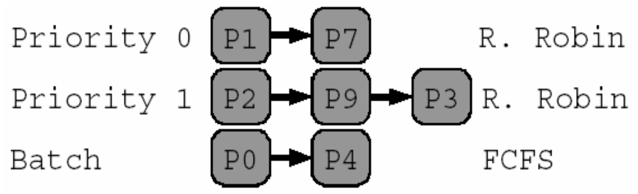


Figure 5. N Independent process queues.

P1,P2,P3,P4… Jobs, One queue per priority and one algorithm per queue.

3 LITERATURE REVIEWS

3.1 Different Available Scheduling Algorithms and Their Characteristics..

There are several scheduling algorithms which assign processor to execute processes. There is no scheduling algorithm that work 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 [Shortest Remaining Time] and HRRN [highest response ratio next] 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 lead to the proper response time. In this algorithm, the time slice should be selected carefully in such a way that algorithm present 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 Unix based 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 SCOPE AND LIMITATIONS

4.1 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
- Assigning time quantum for each queue
- Assigning initial static priorities
- Adjusting dynamic priorities
- Favoring I/O bound processes
- Differentiating foreground processes and background processes.

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 numbers 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.

4.2 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 the hard real-time system, task scheduling algorithms ensure these tasks meet their deadlines. Scheduling involves allocating resources and time to task 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 change in the environment.

A scheduling problem in the 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. The 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-emptible or non-pre-emptible. 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.

5 OPEN PROBLEMS: THE RESEARCH PLAN

5.1 Problem Statement

The aim of this research is to study the policy mechanisms of different real-time schedulers in embedded systems domain and evaluation of performance of these mechanisms. In addition, to arrive at a common solution to simulate a parametric scheduling policy.

5.2 Research Methodology

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.

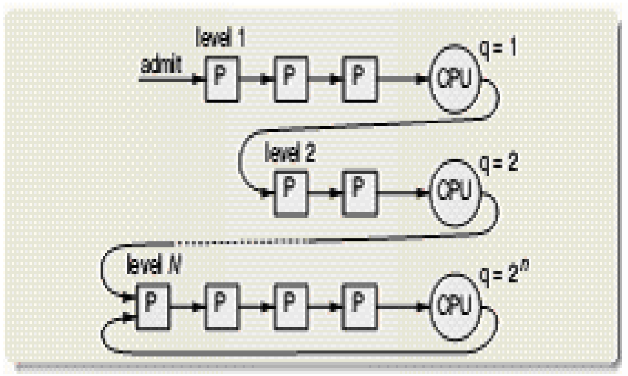


Figure 6. Priority Levels of Parametric Multilevel-feedback-queue scheduler.

The basic idea of PMLFQ is to organize jobs into a set of queues. Each job is processed for 2^i time units if in queue Q_i, before being promoted to queue Q_{i+1} if not completed. At any time, the machines process jobs in the lowest queues, in each queue giving priority to jobs at the front. While this algorithm turn out to be very effective in practice, it behaves very poorly with respect to a worst-case analysis, as explained below.

A good rule of thumb for flow time minimization is given by the Shortest Remaining Processing Time [SRPT] first rule. SRPT prescribes the preemption of a job on execution when a job with shorter remaining processing time is released. SRPT is indeed an optimal algorithm for a single machine [4] and provides the best known approximation for parallel machines [7]. However, a nonclairvoyant scheduling algorithm cannot stick to the SRPT rule since it has no knowledge of the processing time of the jobs before they are completed. As to MLF, it behaves on some instances very differently from the SRPT rule in that it may preempt jobs in a high queue that are nearly completed to process newly released jobs with large processing time in lower queues. This may force jobs with small remaining processing time to spend a long time in the system while other long jobs are processed. It has actually been shown that no deterministic nonclairvoyant algorithm can be competitive at all against a worst case adversary [9]. In order to circumvent these difficulties, a randomized version of MLF, called PMLFQ, was proposed for a single machine.

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 number of CPU times are sent to the lower queues and process requiring I/O bound or related to interactive process are sent to queue 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 jobs in this research paper are 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.

5.3 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.

- The PMLFQ scheduler employs a multi-level feedback queue in which processes with equal priority reside on the same run-queue. The scheduler runs processes round-robin from the highest priority non-empty run-queue.
- The scheduler prevents starvation by periodically raising the priority of processes that have not recently run.
- PMLFQ scheduler also employs higher priorities for processes holding kernel resources. These kernel priorities cause processes to release high-demand kernel

resources quickly, reducing the contention for these resources.

5.3.1 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.

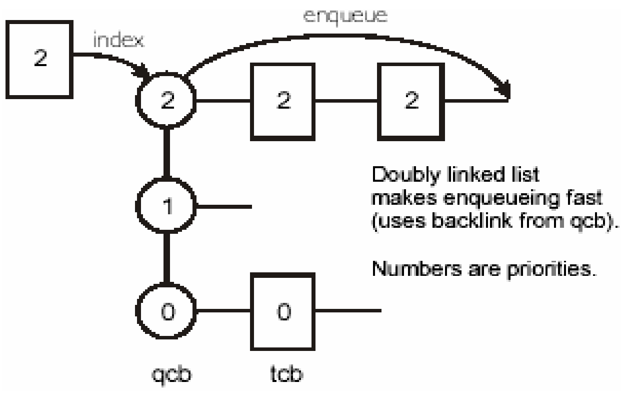

index
enqueue
2
2
2
2
1
0
0
Doubly linked list makes enqueueing fast (uses backlink from qcb).
Numbers are priorities.
qcb
tcb

Figure 7. 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.

5.4 Designing the Parametric Multilevel Queue Scheduler

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**, Later 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 redefined [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.

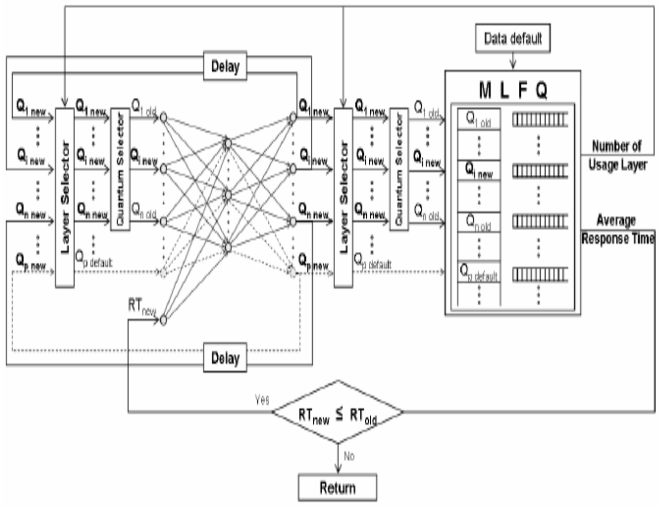


Figure 8. Defining optimized quantum for the queue by PMLFQ function.

Figure 8, 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 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 ();
};
```

Figure. 9. Basic functions of PMLFQ scheduler.

The objective is to obtain a timeline of the execution and to show if tasks meet their deadlines or not. Tasks are assumed to be hard real-time, preemptive, periodic, with deadline equal to the next instance's arrival time and independent (they do not need to synchronize with others in order to execute). They also do not suspend its execution voluntarily. All tasks start execution at the same time in the simulation.

```
void main( int argc, char *argv[])
{
  node n;
   task_t *task, *new;
  init( argc, argv);
  printf( "\nSelected Scheduling Algorithm: %s,\n", labels[ alg]);
  (sched_alg_init)();
  /* select which task to run next */
  for( sys_time=0;
    (merit_list->header->forward[0]!=NIL || request_list->
      header->forward[0]!=NIL) &&  sys_time <= max_time;
    sys_time++){
        /* and if current task emptied its allocated time... */
        if( current!= idle_task  &&  -- current->remaining == 0){
         current->state=DEAD;
         current->cycles++;
         delete_task( deadline_list, current->deadline, current);
         current= idle_task;
      }
      /* Look out for deadline failures */
     while(    key_of(  n=first_node_of(  deadline_list))  <=
sys_time){
         if( (task= n->v)->state != DEAD){
            printf( "At %d: task %c (\"%s\"),
                         instance %d, Deadline Failure%s\n",
              sys_time, task->sys_id, task->name,
                            task->instance, bell);
         }
         delete( deadline_list, n->key);
      }
      /* if it is time to launch a task... */
     while(    key_of(  n=first_node_of(  request_list))  <=
sys_time){
         task_init( (task= n->v) );
         delete( request_list, n->key);
         insert_task( deadline_list, task->deadline,task);
         insert_task(  request_list, task->deadline, task);
      }
      new = (sched_alg)();
      /* swap and register who's using the processor */
      if( current!=new){
         context_switches++;
         current->state=READY;
```

```
            current=new;
            current->state=RUNNING;
        }
        timeline.history[ sys_time]= current->sys_id;
        #ifdef DEBUG
        printf( "%d: %s\n", sys_time, timeline.history);
        #endif
  }
}
```

Figure 10. Simulation of the execution of a task set in a multitasking environment by scheduler.

6 EVALUATION AND RESULTS

6.1 Simulation and Experimental Results of Parametric Queue Scheduler

We define the processor *utilization* factor to be the fraction of processor time spent in the execution of the task set. In other words, the utilization factor is equal to one minus the fraction of idle processor time. Since ***Ci / Ti*** is the fraction of processor time spent in executing task ***Ti***, for m tasks, the utilization factor ***U*** is:

$$U = \sum_{i=1}^{m} (C_i / T_i)$$

→ [3]

In PMLFQ scheduling there is a time threshold and a job threshold. If the number of jobs submitted from the particular user increases beyond the job threshold then the priority of the jobs submitted above the threshold number is decreased and jobs are migrated to a lower priority queue. In other words, with an increasing number of jobs, the priority of jobs from a particular user start to decrease. Moreover, a time threshold is included to reduce the aging affect. With the passage of time, the priority of jobs in the lower priority queues is increased so that it can also have a chance of being executed after a certain wait time. In other words, the more time a job has to wait the more its priority continues to increase. This is illustrated in figure 11.

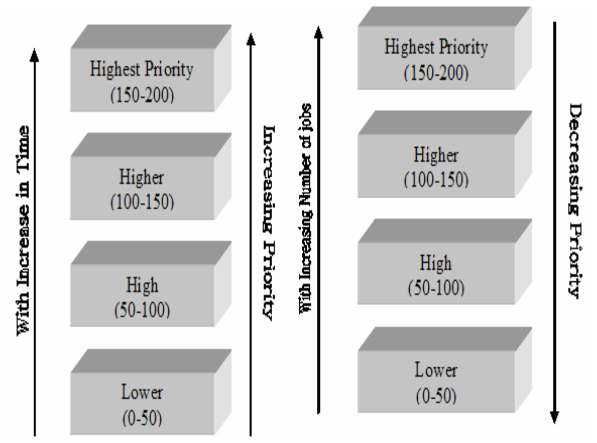
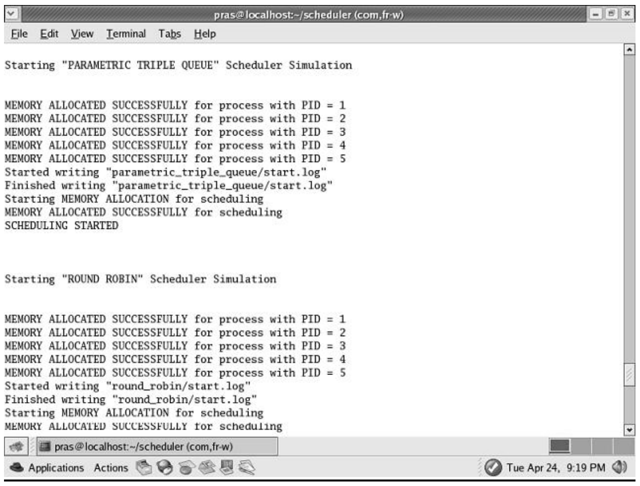


Figure 11. Priority with Time and Job Frequency.


pras@localhost:~/scheduler (com,fr-w)
File Edit View Terminal Tabs Help
pras@localhost:~/scheduler (com,fr-w)
Applications Actions
Tue Apr 24, 9:19 PM

```
Starting "PARAMETRIC TRIPLE QUEUE" Scheduler Simulation


MEMORY ALLOCATED SUCCESSFULLY for process with PID = 1
MEMORY ALLOCATED SUCCESSFULLY for process with PID = 2
MEMORY ALLOCATED SUCCESSFULLY for process with PID = 3
MEMORY ALLOCATED SUCCESSFULLY for process with PID = 4
MEMORY ALLOCATED SUCCESSFULLY for process with PID = 5
Started writing "parametric_triple_queue/start.log"
Finished writing "parametric_triple_queue/start.log"
Starting MEMORY ALLOCATION for scheduling
MEMORY ALLOCATED SUCCESSFULLY for scheduling
SCHEDULING STARTED



Starting "ROUND ROBIN" Scheduler Simulation


MEMORY ALLOCATED SUCCESSFULLY for process with PID = 1
MEMORY ALLOCATED SUCCESSFULLY for process with PID = 2
MEMORY ALLOCATED SUCCESSFULLY for process with PID = 3
MEMORY ALLOCATED SUCCESSFULLY for process with PID = 4
MEMORY ALLOCATED SUCCESSFULLY for process with PID = 5
Started writing "round_robin/start.log"
Finished writing "round_robin/start.log"
Starting MEMORY ALLOCATION for scheduling
MEMORY ALLOCATED SUCCESSFULLY for scheduling
```

Figure 12. 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].

7 CONCLUDING REMARKS

- 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. We continually 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 could also be used on distributed system, in an effective way that the research in this field is still being continued.

8 ACKNOWLEDGMENTS

I am grateful to the guide and RPAC members for their guidance and valuable suggestions in improving the quality of this research. My gratitude goes to the people, who previously succeeded in implementation of general scheduler mechanisms, process

communication, interrupt handling, different operating system concepts etc., and making it available for further development.

9 REFERENCES

[1] Chih-Lin Hu, “On-Demand Real-Time Information Dissemination: A General Approach with Fairness, Productivity and Urgency”, 21st International Conference on Advanced Information Networking and Applications, AINA '07, 2007. Page(s):362 – 369, 21-23 May 2007.

[2] Gauthier L, Yoo S and Jerraya A, “Automatic generation and targeting of application-specific operating systems and embedded systems software,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 20(11), pp.1293-1301, November 2005.

[3] Ghosh S., Mosse D. and Melhem R., “Fault-Tolerant Rate Monotonic Scheduling”, Journal of Real-Time Systems, pp. 149-181, 1998.

[4] Kenneth J. Duda , David R. Cheriton, “Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler”, Proceedings of the seventeenth ACM symposium on Operating systems principles, p.261-276, December 12-15, 1999, Charleston, South Carolina, United States.

[5] Leung J. Y. T. and Whitehead J., “On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks”, Performance Evaluation, number 2, pp. 237-250, 1982.

[6] Lu, C., Stankovic, A., Tao, G. and Son, H.S. “Feedback Control Real-time Scheduling: Framework, Modeling and Algorithms”, special issue of Real-Time Systems Journal on Control-Theoretic Approaches to Real-Time Computing, Vol. 23, No. 1/2 July / September, pp. 85-126, 2002.

[7] Manimaran G. and Siva Ram Murthy C., “A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis”, IEEE Trans on Parallel and Distributed Systems, Volume 9, Issue 11, Page(s):1137 - 1152 , Nov 1998.

[8] Sha L., Rajkumar R. and Lehoczky J. P., “Priority inheritance protocols: an approach to real-time synchronization”, IEEE Transactions on Computers, Volume 39, Issue 9, : Page(s):1175 - 1185 , Sept 1990.

[9] Wang J and Ravindran Binoy, “Time-utility function-driven switched Ethernet: packet scheduling algorithm, implementation, and feasibility analysis", IEEE Trans on Parallel and Distributed Systems, Volume 15, Issue 2, Page(s):119 - 133 , Feb 2004.

[10] Yamada S and Kusakabe S, “Effect of context aware scheduler on TLB”, IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008. Volume , Issue , Page(s):1 – 8, 14-18 April 2008. DOI= 10.1109/IPDPS.2008.4536361.

Author Biographies

Prof. M.V. Panduranga Rao is a research scholar at National Institute of Technology Karnataka, Mangalore, India. His research interests are in the field of real-time and embedded system domain on Linux platform. He has published various research papers across India and in 22nd IEEE international conference AINA-2008 at Okinawa, Japan**.** He has also authored two reference books on Linux Internals. He is the Life member of Indian Society for Technical Education and IAENG. His webpage can be found via,

http://www.pandurangarao.i8.com/

Dr. K.C. Shet obtained his PhD degree from Indian Institute of Technology, Bombay, Mumbai, India, in 1989. He has been working as a Professor in the Department of Computer Engineering, National Institute of Technology, Surathkal, Karnataka, India, since 1980. He has published over 200 papers in the area of Electronics, Communication & Computers. He is a member of Computer Society of India, Mumbai, India, and Indian Society for Technical Education, New Delhi, India. His webpage can be found via,

http://www.nitk.ac.in/~kcshet/index.html