A Research in Real Time Scheduling Policy for Embedded System Domain

M.V. Panduranaga Rao
Research Scholar, Department of Computer Engineering,
National Institute of Technology Karnataka, Surathkal, India-575 025.
raomvp@yahoo.com

and

K.C. Shet
Professor, Department of Computer Engineering,
National Institute of Technology Karnataka, Surathkal, India-575 025.
kcshet@rediffmail.com

Abstract

Scheduling a sequence of jobs released over time when the processing time of a job is only known at its completion is a classical problem in CPU scheduling in time sharing and real time operating systems. Previous approaches to scheduling computer systems have focused primarily on system-level abstractions for the scheduling decision functions or for the mechanisms that are used to implement them. This paper introduces a new scheduling concept New Multi Level Feedback Queue (NMLFQ) algorithm. It's important to get a good response time with interactive tasks while keeping other tasks from starvation. In this research paper, we prove that a New version of the Multilevel Feedback queue algorithm is competitive for single machine system, in our opinion providing theoretical validation of the goodness of the idea that has proven effective in practice.

Keywords: NMLFQ, operating system, computer organization, scheduling, queue, round robin, deadline, edf, rm, preemption, multilevel queue.

1. INTRODUCTION

The Oxford Dictionary of Computing defines a real-time system as: "Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness". The correct behavior of a real-time system depends as much on the timing of computations as it does on the results produced by those computations. Results delivered too late may be useless, or even harmful [1]. Real-time systems are in widespread use and can be found in such application domains as industrial automation, process control, Communications, Command, Control, and multimedia.

There are two distinct types of systems in this field: hard real-time systems and soft real-time systems.

Hard real-time systems are those in which it is imperative that all computations are strictly performed within the specified time, regardless of the operating conditions. Failure to meet the timing constraints of even one task may invalidate the correctness of the entire system.

Soft real-time systems, in contrast, are those in which strict adherence to the timing a constraint of tasks is not always guaranteed. A requisite property of hard real-time systems that allows for their deterministic behavior is closure. All tasks and their associated timing requirements, their worst case execution time, and their interactions, must be known ahead of time. The system designer then properly can dimension the hardware and compose a carefully choreographed sequence of task executions that yields the desired result of 100% deadline satisfaction.

The objective of this research paper is to study the available task schedulers in practice. To make an extensive literature survey on real-time operating system with its own mechanism of scheduling concepts. After comparing some scheduling policies, like edf, rm and multilevel queue etc., an idea of new multilevel feedback queue scheduler is proposed.

1.1 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

New 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.

1.2 Distinguishable characteristics of the RTOS's

❖ Most of the times, a real time system will be an embedded system. The processor and the software are embedded within the equipment, which they are controlling. Typical embedded applications are Cellular phone, Washing Machine, Microwave Oven, Laser Printer, Electronic Toys, Video Games, Avionic controls etc. RTOS will be embedded along with the application code in the system [2]. There is no Hard Disk or Floppy Drive from which the OS will be loaded. The entire code remains in the Read Only Memory of the system. So it must be small in size.

❖ Not only the response time predictable, but it must be very fast as well. Many embedded applications control critical operations (Example. missile control). So then, it must sense the signal and give a response, fast enough to achieve the desired task. A late answer is as bad as the wrong answer [3].

❖ Therefore, the RTOS code must be very short and written efficiently to respond to process needs.

❖ Because of the small footprint requirement and lack of necessity of other peripherals, many features found in desk top PC OS's are not needed like a sophisticated Memory Manager or a File Manager.

1.3 Features of Real Time Systems

❖ Multitasking: Provided through system calls.
❖ Priority based Scheduling: In principle of flexible concepts, but limited to number of priority levels.
❖ Ability to quickly respond to external interrupts.
❖ Basic mechanisms for process communication and synchronization.
❖ Small kernel and fast context switching.
❖ Support real time clock as internal time interface.

1.4 Task states

The basic building block of RTOS is the task. Task is a segment of code which is treated by the system software (O/S) as a program unit which can be started, stopped, delayed, suspended, resumed and interrupted, Example:- Read a byte form serial buffer.

For each task they will have their own stack and registers. All the tasks share common data.
RTOS will have its own data area.

These are THREE major states for a task.
a) Running b) Ready c) blocked

Running task is the task under execution by the CPU. Only one task can be in the running condition.
Ready task is ready to run. Only RTOS is holding it. Any number of tasks can be in ready state.
Blocked task cannot run. It is waiting for something to happen. Any numbers of tasks can be in blocked state.

There can be other states like sleeping suspended, pended, waiting, dormant, delayed as shown in Figure 1. They are only minor variation of the blocked state.

❖ A task will be brought to running state from ready state by the scheduler of the RTOS. Similarly, a task will be sent to ready state from running state by RTOS when it wants to stop the current task in the middle and allow another task to run because of priority.

❖ A task will go from blocked state to ready state when it receives signal for which it has been waiting. It could be an external event, or the result of another task or the time signal from a timer and so on.
❖ A task will go from running to blocked state when it is has to be suspended from running condition because it requires an information that is not available now and it cannot proceed further. It can happen due to various reasons and requirements.
❖ No task can come from blocked state to running state directly.
❖ No task can come to running state unless the scheduler allows it.

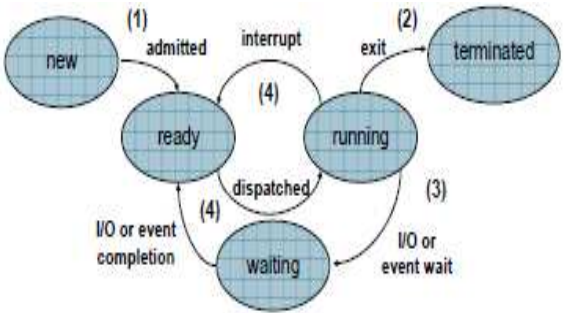

Figure 1: States of process

1.5 Starvation of processes

In computer science, **starvation** is a multitasking-related problem, where a process is perpetually denied necessary resources. Without those resources, the program can never finish its task [4].

Starvation is similar in effect to deadlock. Two or more programs become deadlocked together, when each of them wait for a resource occupied by another program in the same set. On the other hand, one or more programs are in starvation, when each of them is waiting for resources that are occupied by programs, that may or may not be in the same set that are starving. Moreover, in a deadlock, no program in the set changes its state. There may be a similar situation called livelock, where the processes changes their states continually, but cannot progress. Livelock is a special-case of starvation. In that sense, deadlock can also be said a special-case of starvation. Usually the problems in which programs are perpetually deprived of resources are referred as deadlock problems when none of them are changing their states and each of them is waiting for resources only occupied by programs in the same set. All other such problems are referred to as starvation.

Starvation is illustrated by Edsger Dijkstra's dining philosophers problem. The fault lies in the scheduling algorithm. The scheduling algorithm, which is part of the kernel, is supposed to allocate resources equitably; that is, the algorithm should allocate resources so that no process perpetually lacks necessary resources.

Many operating system schedulers have the concept of process priority. A high priority process A will run before a low priority process B. If the high priority process (process A) never blocks, the low priority process (B) will (in some systems) never be scheduled - it will experience starvation. If there is an even higher priority process X, which is dependent on a result from process B, then process X might never finish, even though it is the most important process in the system. This condition is called a priority inversion. Modern scheduling algorithms normally contain code to guarantee that all processes will receive a minimum amount of each important resource (most often CPU time) in order to prevent any process from being subjected to starvation.

In computer networks, especially wireless networks, scheduling algorithms may suffer from scheduling starvation. An example is maximum throughput scheduling.

2. RELATED WORK

This section provides a review of the research related to our work for the implementation of NMLFQ. We describe each approach, its distinguishing features, and how it differs from generic scheduling mechanism.

2.1 Analysis for real-time scheduling

A number of optimal scheduling algorithms exist for CPU loads of up to 100%. Optimality is defined as the algorithm's ability to find a feasible task ordering if such an ordering exists,

The Earliest Deadline First (EDF) and Least Laxity First (LLF) are two such optimal algorithms. When invoked, an EDF scheduler simply scans through all the tasks in the system and dispatches the one with the earliest deadline. The difference between the remaining execution time of a task and its remaining time to deadline is its laxity. The LLF scheduler dispatches the task with the smallest laxity.

CPU load (also known as processor utilization factor) is defined as:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \quad \rightarrow (1)$$

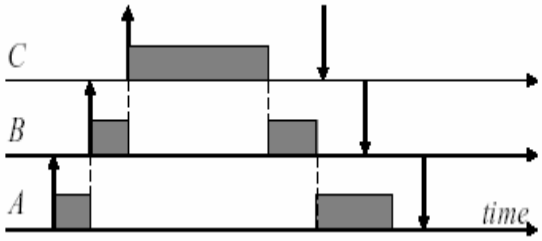

Figure 2: Earliest Deadline First algorithm

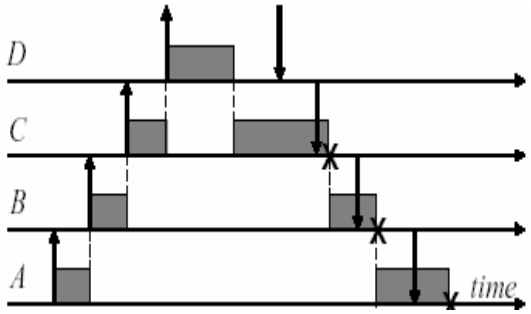

Figure 3: EDF Domino Effect

Where for each of *n* concurrent tasks in the system, *Ci* is the task's computation time (also known as cost), and *Ti* is the task's period. The deadline for each task is the same as its period, i.e., the task must complete its computations before the arrival of its next instance.

The processor utilization factor is used as a schedulability test – for $U <= 1.0$, the task set under consideration can be scheduled using any of the optimal algorithms.

Hard real-time systems, by definition, operate below (or occasionally at) capacity. No overload can occur in such a system because it would cause missed deadlines.

2.2 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.

2.2.1 Priority-based kernel for embedded applications

This category includes many commercial kernels. 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.

2.2.2 Real-time extensions of timesharing operating systems

This category of operating systems includes the real-time extensions of commercial timesharing systems. 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.

2.2.3 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:

- 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.

2.3 Scope and limitation; Research findings and gaps..

Though algorithms such as EDF produce optimal schedules during underload, they can produce schedules with catastrophic effects as load grows beyond capacity. Figure 3, shows one such case where EDF scheduling of a newly arrived task pushes all existing tasks beyond their respective deadlines.

The diagram shows task A arriving first (denoted by the up-arrow). It starts executing and is subsequently preempted by task B which has an earlier deadline (denoted by a down-arrow). Task B is in turn preempted by the arrival of task C with yet an earlier deadline. Task C then runs to completion, after which task B is resumed as depicted in Figure 2. Upon completion of task B, task A is resumed and runs to completion. If task D with an earlier deadline were to arrive (increasing the load beyond 100%), EDF would blindly schedule it, thereby pushing all other tasks beyond their deadlines, like falling dominos, as shown in Figure 3. This phenomenon is known as the domino effect .

As real-time systems enter overload, only a subset of all contending tasks can complete execution by their deadlines. The system therefore must shed load to a point at (or below) capacity where it properly can service task requests. It must shed load in a predictable way that avoids uncontrolled performance degradation.

A generalized definition of load (β) was introduced in [5] for a set of dynamic real-time tasks:

$$\beta_i(t) = \frac{\sum_{d_k \le d_i} c_k(t)}{(d_i - t)} \quad \rightarrow (2)$$

$$\beta = max\ \beta_i(t) \quad \rightarrow (3)$$

Where *ck(t)* is the remaining execution time of task *k* with deadline less than (or equal) to *di*. This definition of load allows the calculation of load for a generic set of real-time tasks that can be activated dynamically but do not follow any pre-defined activation pattern. This method calculates CPU load at each task activation and can detect and quantify transient overload.

- **EDF scheduling**

Formal analysis methods are available for EDF scheduling. In the simplest case the following assumptions are made:

- only periodic tasks exist,
- each task *i* has a period *Ti*,
- each task has a worst case execution time *Ci*,
- each task has a deadline *Di*,
- the deadline for each task is equal to the task period (*Di* = *Ti*),
- no interprocess communication, and
- an “ideal” real-time kernel (context switching and clock interrupt handling takes zero time).

With these assumptions the following necessary and sufficient condition holds:

If the utilization *U* of the system is not more than 100% then all deadlines will be met.

$$U = \sum_{i=1}^{i=n} \frac{C_i}{T_i} \leq 1$$ → (4)

The utilization *U* determines the CPU load. The main advantage with EDF scheduling is that the processor can be fully utilized and still all deadlines can be met. More complex analysis exists that loosens some of the assumptions above.

❖ **RM scheduling**

Rate monotonic (RM) scheduling is a scheme for assigning priorities to tasks that guarantees that timing requirements are met when preemptive fixed priority scheduling is used. The scheme is based on the simple policy that priorities are set monotonically with task rate, i.e., a task with a shorter period is assigned a higher priority.

For a system with *n* tasks, all tasks will meet their deadlines if the total utilization of the system is below a certain bound.

$$\sum_{i=1}^{i=n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$ → (5)

As $n \rightarrow \infty$, the utilization bound → 0.693. This has led to the simple rule-of-thumb that says that "If the CPU utilization is less than 69%, then all deadlines are met".

In 1986 a sufficient and necessary condition was derived [5]. The condition is based on the notion of worst-case response time, *Ri*, for a task *i*, i.e., the maximum time it can take to execute the task. The response time of a task is computed by the recursive equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$ → (6)

where *hp(i)* is the set of tasks with higher priority than task *i*. The task set is schedulable if *Ri* <= *Di* for all tasks *i*. The model also allows deadlines that are shorter than the period (*Di* <= *Ti*).

The rate-monotonic priority scheme is not very good when *Di* <= *Ti*. An infrequent but urgent task will be given a very low priority. In this case the *deadline-monotonic priority scheme* is better suited. Here it is the task deadline that decides the priority rather than the period. A task with a short deadline gets high priority. This policy has been proved optimal when *D* <= *T* in the sense that if the system is unschedulable with the deadline-monotonic priority ordering, then it is unschedulable also with *all* other orderings [6]. Equation 1 holds also for the deadline-monotonic case.

2.4 Motivation, objectives and Goals

Making sure that the scheduling strategy is good enough with the following criteria:

- Utilization / Efficiency: Keep the CPU busy 100% of the time with useful work
- Throughput: Maximize the number of jobs processed per hour.
- Turnaround time: From the time of submission to the time of completion.
- Waiting time: Sum of times spent in ready queue – Normally we must minimize this.
- Response Time: Time from submission till the first response is produced, minimize response time for interactive users.
- Fairness: make sure each process gets a fair share of the CPU

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. 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.

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.

3. THE RESEARCH PLAN - PROBLEM STATEMENT

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

3.1 Research Methodology

The NMLFQ scheduling algorithm works by dividing the CPU time into *epochs*. In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins. In general, different processes have different time quantum durations. The time quantum value is the maximum CPU time portion assigned to the process in that epoch. When a process has exhausted its time quantum, it is preempted and replaced by another runnable process. Of course, a process can be selected several times from the scheduler in the same epoch, as long as its quantum has not been exhausted--for instance, if it suspends itself to wait for I/O, it preserves some of its time quantum and can be selected again during the same epoch. The epoch ends when all runnable processes have exhausted their quantum; in this case, the scheduler algorithm recomputes the time-quantum durations of all processes and a new epoch begins.

Each process has a *base time quantum*: it is the time-quantum value assigned by the scheduler to the process if it has exhausted its quantum in the previous epoch. A new process always inherits the base time quantum of its parent [7].

The INIT_TASK macro sets the value of the base time quantum of process 0 (*swapper*) to DEF_PRIORITY; that macro is defined as follows:

#define DEF_PRIORITY (20*HZ/100)

Since HZ, which denotes the frequency of timer interrupts, is set to 100 for IBM PCs, the value of DEF_PRIORITY is 20 ticks, that is, about 210 ms.

Users rarely change the base time quantum of their processes, so DEF_PRIORITY also denotes the base time quantum of most processes in the system.

In order to select a process to run, the NMLFQ scheduler must consider the priority of each process. Actually, there are two kinds of priority:

- **Static priority**

This kind is assigned by the users to real-time processes and ranges from 1 to 99. It is never changed by the scheduler.

- **Dynamic priority**

This kind applies only to conventional processes; it is essentially the sum of the base time quantum (which is therefore also called the *base priority* of the process) and of the number of ticks of CPU time left to the process before its quantum expires in the current epoch.

Of course, the static priority of a real-time process is always higher than the dynamic priority of a conventional one: the scheduler will start running conventional processes only when there is no real-time process in a TASK_RUNNING state.

3.2 Background significance and Features of New Scheduler

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.

NMLFQ uses preemptive scheduling. This means that, as soon as a higher priority task becomes ready to run, it preempts the current task and runs as shown in Figure 4. For safety, NMLFQ 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.

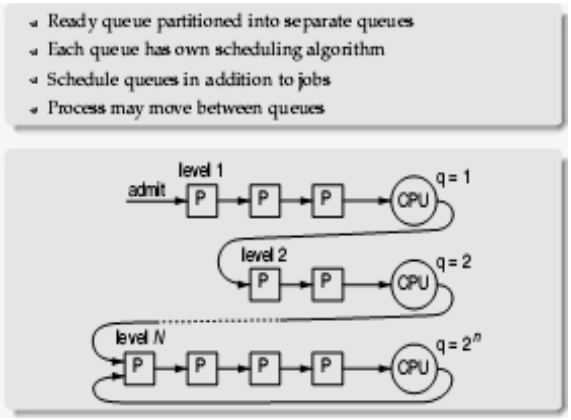

Figure 4: Priority Levels of Newc Multilevel-feedback-queue scheduler

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 [8]. 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. NMLFQ also permits time slicing among the lowest priority tasks - this is even more equal.

Scheduler Locking: NMLFQ 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 [9]. 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.

3.3 Designing the new multilevel queue scheduler

As it was mentioned before, in MLFQ the operating system builds several separate queues as in Figure 5 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.

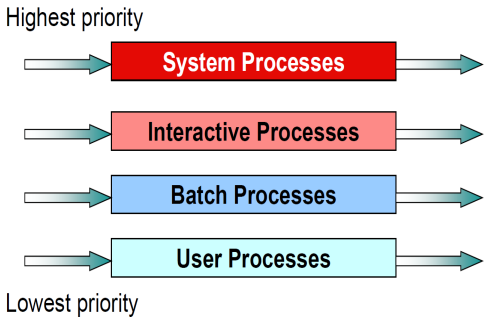

Figure 5: Distinguishing processes in queues

The scheduler keeps a list of process queues. Every queue gets a priority assigned. Processes start in a given priority queue. Different OSs use other numbers corresponding with priorities. In Unix, for example, the highest priority (which is chosen first) is 0. Lower priorities have a larger number. In other OSs this is just inverted. Different OSs also use other number of priorities, depending on for which situations or purposes they are designed.

Processes in queues with a higher priority get less CPU time, so a smaller 'time quantum' than processes in lower priority queues. In this way, interactive processes get less CPU time and computing processes more. This is depicted in Figure 8. But how do we know which processes are interactive and which are not? There is an easy solution: let processes move from queue to queue. When a process blocks before it's time quantum is spent, the scheduler will increase its priority. So interactive processes, which normally just read some input and then quit, will automatically promote to higher priorities. Processes which use their quantum get a lower priority, so computing processes, which take all the cpu time they can get, will automatically move to the lower priorities. Then, if there are multiple processes in a queue that are ready to be executed, they are scheduled round-robin. This is useful, because all processes in one queue are as important. This system is the best approximization of SJF, and so it is the best we can use. It has very little overhead and gives high response time to interactive processes. Real-time processes can get a fixed and high priority; in this way, they are always chosen when they need to be.

3.4 Class table of NMLFQ queue scheduler

Table 1: New multilevel feedback queue scheduler

SCHEDULER_NMLFQ_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()

❖ **Code snippet for scheduler**

The objective is to obtain a timeline of the execution, and to show if tasks meet their deadlines or not [10]. 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. The declarations of task are as shown in Figure 6 and Simulation of NMLFQ queue scheduler is shown is Figure 9.

```
#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 6: Declarations in New multilevel queue scheduler

4. EVALUATION AND RESULTS

4.1 Simulation and experimental results of NMLFQ scheduler

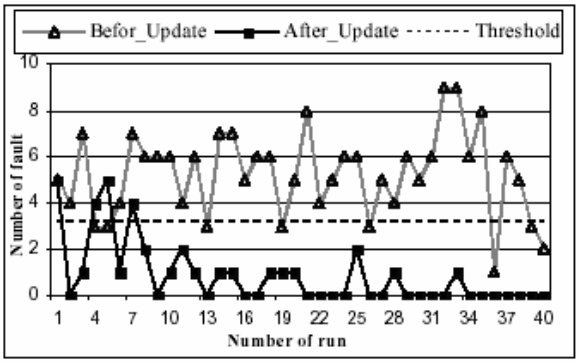

Figure 7: Comparing between NMLFQ & combinational function before and after the parameters update, that each run include 500 processes

The design of conventional NMLFQ scheduler uses the time slices in the range from 10ms to 200ms as in Figure 7. Increasing time slices by intervals of 10ms, gave us a total of 20 queues. NMLFQ algorithm often use exponentially increasing time slices. We move tasks up and down the queue levels in the traditional manner. Every time a task becomes ready to run, we check whether it used up its previous time slice and place it either above or below the queue. Some additional decisions relate to process creation. When one task forks to create a new task, the parent's remaining time slice is split in half between the parent and the child, so that forking divides a task's resources. The child starts on the highest-priority queue, and if it has a time slice too large for that queue we return the extra amount back to the parent. If the child is now on a higher queue than the parent, or if the parent has no time left, the child will run next. Otherwise, the parent will continue.

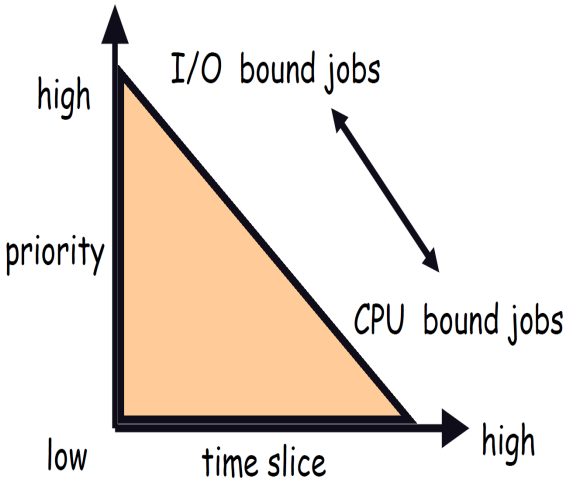

Figure 8: Graph illustrating the comparison of priority and jobs with respect to time-slice

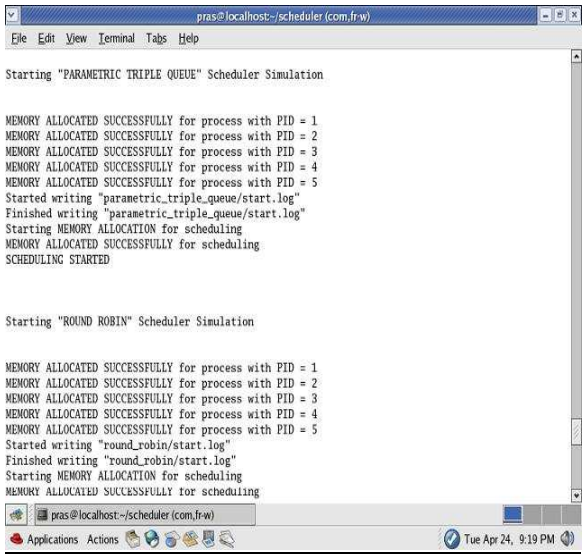

Figure 9: Simulation of NMLFQ queue scheduler

Two of the most critical parts of a kernel are the memory subsystem and the scheduler. This is because they influence the design and affect the performance of almost every other part of the kernel and the OS. That is also why you would want to get them absolutely right and optimize their performance. Designing scheduler is at best *a black art.* No matter how good the design is, some people will always feel that some categories of processes have got a raw deal.

5. CONCLUDING REMARKS

The main contributions made by this research are:

✓ The scheduling policies of the different schedulers are studied and their performance has been compared. The scheduler code is developed using C++ language on Linux operating system and Gantt chart log is provided.

✓ We have developed *New Multi Level Feedback Queue,* a real-time CPU scheduling algorithm that guarantees low deadline miss ratios in systems where task execution times may deviate from estimations at run-time.

✓ This algorithm uses a *New approach* for defining the optimized quantum of each queue and number of queues. The simulations show that the NMLFQ algorithm gives 10% better performance compared to Multi Level Queue real time scheduling with respect to response time and waiting time.

5.1 Further Research

One possible line of future research is to explore integrating this kind of scheduler into common purpose commercial operating systems. The problem is exacerbated when modules synchronize or share resources (such as data). Although closer integration may be, appropriate under some circumstances. While in principle, this should be easy and yield good results, in practice we expect interesting things would be learned along the way.

5.2 Acknowledgment

The first author is grateful to the guide and RPAC members for their guidance and valuable suggestions in improving the quality of this research. Author's 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.

6. 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 and 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. Digital Object Identifier 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 systems on Linux platform and Security. He has published various research papers across India and in IEEE international conference in 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/index.php?q=Dr.K.C.Shet.html .