International Journal of Enhanced Research in Science Technology & Engineering, ISSN: 2319-7463
Vol. 3 Issue 3, March-2014, pp: (535-538), Impact Factor: 1.252, Available online at: www.erpublications.com

Real Time Scheduling in Embedded System

Kalki Narayan Jindal[1], Sukhwinder Singh[2]
[1]Student, E&EC Department, PEC University of Technology, Chandigarh, INDIA
[2]Professor, E&EC Department, PEC University of Technology, Chandigarh, INDIA

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 [1]. 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.

I. 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" [2]. 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. 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 [3]. 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.

A. 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 embeddedwithin 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.

B. 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.

C. 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 [4], Example:- Read a byteform

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

D. Starvation of processes

In computer science, starvationis a multitasking-related problem, where a process is perpetually denied necessary resources. Without those resources, the program can never finish its task. 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 [5]. 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 [6]. All other such problems are referred to as starvation. Starvation is illustrated by Edsger Dijkstra's dining philosopher's problem.

II. RELATED WORK

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

A. Analysis for real-time scheduling

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 [6]. 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.

B. 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 ihas 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 inter process communication
- 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.

C. 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 [7]. 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.

III. 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.

A scheduler that provides good theoretical guarantees is not effective when the kernel is not responsive orits timers are not accurate [8]. Conversely, a responsive kernel with an accurate timing mechanism is unable to handle a large class of time-sensitive applications without an effective scheduler. 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.

A. 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 [9]. 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.In order to select a process to run, the NMLFQ scheduler must consider the priority of each process. Actually, thereare two kinds of priority:

B. Static priority

This kind is assigned by the users to real-time processes and ranges from 1 to 99 [9]. It is never changed by the scheduler.

C. 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 [10].

IV. EXPERIMENTAL RESULTS OF NMLFQ SCHEDULER

The design of conventional NMLFQ scheduler uses the time slices in the range from 10ms to 200ms. 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. 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.

V. 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.
- 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.

VI. 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.

ACKNOWLEDGMENT

I am very thankful to Prof. Sukhwinder Singh for providing me support and information on my topic. He gave me this opportunity which helped me learn new things.

REFERENCES

[1]. Chih-Lin Hu, "On-Demand Real-Time Information Dissemination: A General Approach with Fairness, Productivity and Urgency"
[2]. Gauthier L, Yoo S and Jerraya A, "Automatic generation and targeting of application-specific operating systems and embedded systems software".
[3]. Ghosh S., Mosse D. and Melhem R., "Fault-Tolerant Rate Monotonic Scheduling", Journal of Real-Time Systems,
[4]. Kenneth J. Duda and David R. Cheriton, "Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler".
[5]. Leung J. Y. T. and Whitehead J., "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks", Performance Evaluation, number
[6]. Lu, C., Stankovic, A., Tao, G. and Son, H.S. "Feedback Control Real-time Scheduling: Framework, Modeling and Algorithms",
[7]. Manimaran G. and Siva Ram Murthy C., "A fault-tolerant dynamic scheduling algorithm for multiprocessor realtime systems and its analysis", IEEE Trans on Parallel and Distributed Systems, Volume 9, Issue 11,
[8]. Sha L., Rajkumar R. and Lehoczky J. P., "Priority inheritance protocols: an approach to real-time synchronization", IEEE Transactions on Computers, Volume 39,
[9]. Wang J and Ravindran Binoy, "Time-utility function-driven switched Ethernet: packet scheduling algorithm, Implementation, and feasibility analysis".
[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.
[11]. CLEI Electronic Journal, Volume 12, Number 2, Paper 4, August 2009 A Research in Real Time Scheduling Policy for Embedded System Domain.