P-3279

# A Modified O(1) Scheduling Algorithm for Real-Time Tasks

**By**

**M.BALAJI**

**Reg. No: 0820108004**

**of**

**KUMARAGURU COLLEGE OF TECHNOLOGY**
**(An Autonomous Institution Affiliated to Anna University, Coimbatore)**

**COIMBATORE – 641 006**

**A PROJECT REPORT**

**Submitted to the**

**FACULTY OF INFORMATION AND COMMUNICATION ENGINEERING**

***In partial fulfillment of the requirements for the award of the degree of***

**MASTER OF ENGINEERING**

**IN**

**COMPUTER SCIENCE AND ENGINEERING**

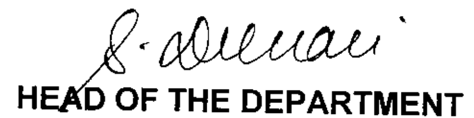
**MAY 2010**

## BONAFIDE CERTIFICATE

Certified that this project report titled "**A MODIFIED O(1) SCHEDULING ALGORITHM FOR REAL-TIME TASKS**" is the bonafide work of **M.BALAJI (0820108004)** who carried out the research under my supervision. Certified further, that to the best of my knowledge the work reported here in does not form part of any other project report of dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

**GUIDE**

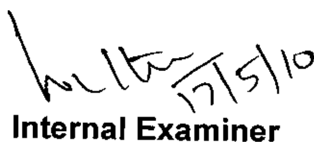
**(Ms. P.DEVAKI)**



**HEAD OF THE DEPARTMENT**

**(Ms. P.DEVAKI)**

The candidate with **University Register No. 0820108004** was examined by us in Project Viva-Voce examination held on 17/5/10



**Internal Examiner**

**External Examiner**

## ACKNOWLEDGEMENT

I express my profound gratitude to our Chairman **Padmabhusan Arutselver Dr. N. Mahalingam B.Sc, F.I.E** for giving this great opportunity to pursue this course.

I would like to begin by thanking to **Dr. S.Ramachandran.**, **Ph.D.,** *Principal* for providing the necessary facilities to complete my thesis.

I take this opportunity to thank **Dr. S.Thangasamy**, **Ph.D.,** *Dean*, *Research and Development*, for his precious suggestions.

I register my hearty appreciation to **Ms. P.Devaki M.E., (Ph.D),** *Head of the Department, Computer Science and Engineering,* my thesis advisor. I thank for her support, encouragement and ideas. I thank her for the countless hours she has spent with me, discussing everything from research to academic choices.

I thank all project committee members for their comments and advice during the reviews. Special thanks to **Ms.V.Vanitha M.E., (Ph.D),** *Assistant professor*, Department of Computer science and Engineering, for arranging the brain storming project review sessions.

I would like to convey my honest thanks to **Mr. T. Chandra Vel Kumar**, all **Teaching** staff members and **Non Teaching** staff members of the department for their support. I would like to thank all my classmates who gave me a proper light moments and study breaks apart from extending some technical support whenever I needed them most.

I dedicate this project work to my **parents** and my **friends** for no reasons but feeling from bottom of my heart, without their love this work wouldn't possible.

# VIVEKANANDHA

INSTITUTE OF ENGINEERING AND TECHNOLOGY FOR WOMEN

ELAYAMPALAYAM, TIRUCHENGODE.

**Department of Computer Applications**

## NATIONAL CONFERENCE ON " EMERGING TECHNOLOGIES IN ADVANCED COMPUTING AND COMMUNICATION "

13 - March, 2010

CERTIFICATE

This is to Certify that Mr. / ~~Ms.~~ / ~~Dr.~~ M. BALAJI of II ME (CSE), Kumaraguru College of Technology has participated in the " National Conference on ETACC '10 ", organized by " VIVACIOUS " the professional association of Computer Applications on 13th March 2010 and presented a paper on A Modified O(1) Scheduling Algorithm for Realtime Tasks.

A. Ahamed hakeel
HOD & Organizing Secretary

Principal

Chairman & Secretary

## ABSTRACT

Real-time task scheduling algorithm must schedule the task according to strict time constraints based on the priority of the tasks. Due to the advancement in computer technology, embedded microcontrollers of near future will have the processing capability of today's servers. So, the real time scheduling algorithm should satisfy real time requirements and also it must act fair to other type of tasks. Researchers in the real-time system community have designed and studied many advanced scheduling algorithms. However, most of these algorithms have not been implemented since it is very difficult to support new scheduling algorithms on most operating systems. To solve this problem, the scheduling mechanism in Linux is enhanced to provide a flexible scheduling framework .The reason for choosing the Linux kernel 2.6.9.34 edition to ameliorate is that, it is designed with modularization mind, so it can be easily transplanted to embedded system by simplifying its modules. It also uses O(1) scheduling algorithm which tries to decrease the overall execution time of all tasks.

The main goal of the proposed architecture is to provide a scheduling algorithm, which makes a perfect balance between fairness and quick response. The work involves reservation of I/O waiting queue to reduce the response time of tasks waiting for IO, by swapping the task from IO waiting queue to active queue as soon as it gets IO response .The expired queue is removed, so that the time taken for switching is reduced. Hence, the time taken for real-time tasks to complete its execution is also reduced.

## ஆய்வுச் சுருக்கம்

நிகழ்நேர வேலை அட்டவணை நெறிமுறை, வேலையின் முதன்மையை அடிப்படையாக கொண்டும், கடுமையான நேர தடைகளை கருத்தில் கொண்டும், அட்டவணைபடுத்தப்பட வேண்டும். கணினிகளின் தொழில்நுட்ப முன்னேற்றத்தினால் எதிர்கால நுண்கட்டுப்பாளர்கள், இன்றைய வழங்கிகளின் பொருத்தமான செயலாக்கம் கொண்டுள்ளதாக இருக்க வேண்டும். ஆகையினால் எதிர்கால நிகழ்நேர அட்டவணை நெறிமுறை, நிகழ்நேர தேவையை மட்டும் பூர்த்தி செய்வதாக இல்லாமல் மற்ற வகை வேலைகளும் நியாயமாக செயல்படுத்தப்பட ஏதுவாகவும் இருக்க வேண்டும். இந்த திருத்திய O(1) நிகழ்நேர வேலைக்கான அட்டவணை நெறிப்படுத்துதல் லினிக்ஸ் 2.6.9.34 ஐ அடிப்படையாக கொண்டுள்ளது. நிகழ்நேர முறை சமூக ஆராய்ச்சியாளர்கள் பலர், சிலபல முன்னேற்றமடைந்த அட்டவணை நெறிமுறைகளை ஆராய்ந்தும், வரைப்படுத்தியும் உள்ளார்கள். ஆனால் அந்த புதிய அட்டவணை நெறிமுறைகள் பல இயக்கு தளங்களில் ஆதரவு படுத்த மிகச்சிரமமாக இருந்ததனால், அந்த அட்டவணை நெறிமுறைகளை செயல்படுத்தப்பட முடியவில்லை. இந்த குறையை சரி செய்வதற்கு லினிக்ஸில் உள்ள அட்டவணை இயக்குமுறை, இணக்கமுள்ள அட்டவணை வரம்புறுகு முறையை வழங்குவதற்கு ஏதுவாக மேம்படுத்தப்பட்டுள்ளது.

லினிக்ஸ் 2.6.9.34 ஐ தேர்ந்தெடுக்க காரணம், இதில் உள்ள O(1) அட்டவணை நெறிமுறை நன்றாக இருப்பதால் இந்த நெறி முறை சீராக்குவதற்கு சாத்தியமாகவுள்ளது. வேகமான பிரதி நகல் நெறிமுறை அட்டவணை வழங்குவதுதான் இந்த முன்மொழியும், கட்டமைப்பின் முக்கிய குறிக்கோள். இந்த வேலையில் உள்ளீடு, வெளியீடு காத்திருக்கும் வரிசையை பதிவு செய்யும் நேரம் ஈடுபடுத்தப்பட்டுள்ளது. இதில் காலாலதியான வரிசையை நீக்கி, நிகழ் நேர வேலையின் நம்பகத்தன்மை மேம்படுத்தப்பட்டுள்ளது.

# TABLE OF CONTENTS

| CONTENTS | PAGE NO. |
|---|---|
| **ABSTRACT (ENGLISH)** | Iv |
| **ABSTRACT (TAMIL)** | v |
| **LIST OF FIGURES** | vii |
| **1. INTRODUCTION** | 01 |
| 1.1 Project Outline | 01 |
| 1.2 Problem Definition | 03 |
| **2. LITERATURE SURVEY** | 04 |
| 2.1 Overview of Real-time Scheduling Algorithm | 04 |
| 2.1.1 Introduction | 04 |
| 2.1.2 Rate monotonic | 05 |
| 2.1.3 Early deadline first | 08 |
| 2.1.4 Least slack time first | 11 |
| 2.2 O(1) Scheduling Algorithm | 12 |
| 2.2.1 Run queue | 14 |
| 2.2.2 Timeslice | 17 |
| **3. DETAILS OF MODIFIED O(1) SCHEDULING ALGORITHM** | 22 |
| 3.1 Improvement of queue management | 22 |
| 3.2 Improvement of process analysis | 25 |
| **4. SIMULATION AND RESULTS** | 29 |
| **5. CONCLUSION AND FUTURE ENHANCEMENTS** | 31 |
| **APPENDIX** | 32 |
| **REFERENCES** | 51 |

# LIST OF FIGURES


| FIGURE NO | CAPTION | PAGE NO |
|---|---|---|
| 2.1 | Runqueue of O(1) scheduling algorithm with active priority array | 15 |
| 2.2 | Runqueue with two set of runnable process | 16 |
| 3.3 | Selecting highest priority task in runqueue | 17 |
| 3.1 | Runqueue of modified O(1) Scheduling algorithm | 26 |
| 4.1 | Comparison of overall completion of all process | 30 |
| 4.2 | Comparison of response time of real time task | 30 |

# CHAPTER 1

# INTRODUCTION

## 1.1 PROJECT OUTLINE

Real-time computing is required in many application domains, such as avionics systems, traffic control system, and automated factory systems. Each application has peculiar characteristics in terms of timing constraints and computational requirements (such as periodicity, criticality of the deadlines, response time, etc). Some mission-critical real-time systems may suffer irreparable damages if a deadline is missed. It is the system builder's responsibility to choose an operating system that can support and schedule these jobs according to their timing specifications so that no deadline will be missed.

On the other hand, some soft real-time applications such as streaming audio/video and multiplayer games also have timing constraints and require performance guarantees from the underlying operating system. The application output provided to users is optimized by meeting the maximum number of real-time constraints (e.g., deadlines). But unlike hard real-time applications, occasional violations of these constraints may not result in a useless execution of the application or catastrophic consequences.

Advances in computer technology have also dramatically changed the design of many real-time controller devices that are being used on a daily basis. Many traditional mechanical controllers have been gradually replaced by digital chips that are much

cheaper and more powerful. In fact, computing power of future embedded digital controllers will be at the same level as that in today's big system servers. As a result, future embedded devices must be able to handle complex application requirements, real-time or otherwise. How we can design real-time operating systems (RTOSs) to support applications with mixed real-time and non real-time performance requirements will be an important issue.

These three types of timing requirements (hard real-time, soft real-time, and non real-time) are all important. It is the goal of the Modified O(1) scheduling algorithm to satisfy these different requirements for many real-time systems.

## 1.2 PROBLEM DEFINITION

To determine the assignment of real time tasks to the given processor such that

- The response time of the real time tasks to be reduced.
- The stability (task must complete it's execution within its deadline) of real time task to be enhanced.
- The overall completion time of all the tasks is to be minimized.
- All priority constraints are to be satisfied for real time task.

# CHAPTER 2

# LITERATURE SURVEY

## 2.1. OVERVIEW OF REAL TIME SCHEDULING ALGORITHMS

### 2. 1.1. INTRODUCTION

Many real-time scheduling algorithms have been proposed in literature to deal with timing constraints, starting from the classical Rate Monotonic (RM) or Earliest Deadline First (EDF) algorithms[2] to Least Slack First(LSF) .These algorithms priority of tasks are all based on some special characteristic variable such as (deadline , idle time or value), and these algorithms was performed under very restrictive assumptions (independent tasks, fixed execution times and periods, completely preemptive scheduling, and so on).However, it's far from enough that the priority only bases on some special variable. Some authors propose that modification in a conventional OS based on a monolithic kernel approach is a better choice. They [8] modify the kernel in order to introduce schedule the interrupt handlers, preemption model and save some limited form of device scheduling. In general, all these works introduce a new scheduling algorithm in the kernel. However, since conventional kernels provide a quantum based resource allocation and their purpose is to make tasks running fairly, so it's very difficult to modify them for real-time system, only a few algorithms can be implemented on them easily. Proportional Share algorithms [8], being based on a per-quantum CPU allocation, are expressly designed to be implemented on a conventional kernel. Another interesting technology named Resource Kernels (RK) is growing up recently. An RK is a resource centric kernel that complements the OS kernel providing support for QOS, and enabling the use of reservation techniques in traditional OS.

### 2.1.2 RATE MONOTONIC [2]:

The term rate monotonic derives from a method of assigning priorities to a set of processes as a monotonic function of their rates. While rate monotonic scheduling systems use rate monotonic theory for actually scheduling sets of tasks, rate monotonic algorithm can be used on tasks scheduled by many different systems to reason about schedulablility. We say that a task is schedulable if the sum of its preemption, execution, and blocking is less than its deadline. A system is schedulable if all tasks meet their deadlines. Rate monotonic algorithm provides a mathematical and scientific model for reasoning about schedulability.

The following assumptions are made for rate monotonic algorithm:

- Task switching is instantaneous.
- Task interactions are not allowed.
- Tasks become ready to execute precisely at the beginning of their periods and relinquish the CPU only when execution is complete.
- Task deadlines are always at the start of the next period.
- Tasks with shorter periods are assigned higher priorities; the criticality of tasks is not considered
- Task execution is always consistent with its rate monotonic priority: a lower priority task never executes when a higher priority task is ready to execute.

It is immediately obvious that some of these assumptions do not completely conform to actual systems. The importance of these assumptions is that they allow reasoning with certainty about whether or not a set of tasks can be scheduled.

Given certain information about a particular set of tasks, under rate monotonic conditions, one can evaluate certain tests to understand whether or not those tasks can all meet their deadlines in a real time system. Because these values are known at design time and are monotonic, any analysis and scheduling can be done statically. Static scheduling is one advantage that the industry has a strong preference for in hard real-time applications.

Liu & Layland (1973) proved that for a set of n periodic tasks with unique periods, a feasible schedule that will always meet deadlines exists if the CPU utilization is below a specific bound (depending on the number of tasks). The schedulability test for RMS is:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1)$$

where $C_i$ is the computation time, $T_i$ is the release period (with deadline one period later), and n is the number of processes to be scheduled. For example $U \leq 0.8284$ for n = 2. When the number of processes tends towards infinity this expression will tend towards:

$$\lim_{n \to \infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693147\ldots$$

So, a rough estimate is that RMS in the general case can meet all the deadlines if CPU utilization is 69.3%. The other 30.7% of the CPU can be dedicated to lower-priority non real-time tasks. It is known that a randomly generated periodic task system will meet all deadlines when the utilization is 85% or less, however this fact depends on knowing the exact task statistics (periods, deadlines) which cannot be guaranteed for all task sets.

The rate monotonic priority assignment is optimal meaning that if any static priority scheduling algorithm can meet all the deadlines, then the rate monotonic algorithm can too.

### 2.1.3 EARLY DEADLINE FIRST [2]

Earliest Deadline First (EDF) or Least Time to Go is a dynamic scheduling algorithm used in real-time operating systems. It places processes in a priority queue. Whenever a scheduling event occurs (task finishes, new task released, etc.) the queue will be searched for the process closest to its deadline. This process is the next to be scheduled for execution.

EDF is an optimal scheduling algorithm on preemptive uniprocessors, in the following sense: if a collection of independent jobs, each characterized by an arrival time, an execution requirement, and a deadline, can be scheduled (by any algorithm) such that all the jobs complete by their deadlines, the EDF will schedule this collection of jobs such that they all complete by their deadlines.

With scheduling periodic processes that have deadlines equal to their periods, **EDF** has a utilization bound of 100%. Thus, the schedulability test for **EDF** is:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1,$$

where the $\{C_i\}$ are the worst-case computation-times of the *n* processes and the $\{T_i\}$ are their respective inter-arrival periods (assumed to be equal to the relative deadlines).

With scheduling periodic processes that have deadlines equal to their periods, EDF has a utilization bound of 100% i.e. EDF can guarantee that all deadlines are met provided that the total CPU utilization is not more than 100%. So, compared to fixed

priority scheduling techniques like rate-monotonic scheduling, EDF can guarantee all the deadlines in the system at higher loading.

However, when the system is overloaded, the set of processes that will miss deadlines is largely unpredictable (it will be a function of the exact deadlines and time at which the overload occurs). This is a considerable disadvantage to a real time systems designer. The algorithm is also difficult to implement in hardware and there is a tricky issue of representing deadlines in different ranges (deadlines must be rounded to finite amounts, typically a few bytes at most). If one uses modular arithmetic to calculate future deadlines relative to now, the field storing a future relative deadline must accommodate at least the value of the (("duration" {of the longest expected time to completion} * 2) + "now"). Therefore EDF is not commonly found in industrial real-time computer systems. Instead, most real-time computer systems use fixed priority scheduling (usually rate-monotonic scheduling). With fixed priorities, it is easy to predict that overload conditions will cause the low-priority processes to miss deadlines, while the highest-priority process will still meet its deadline.

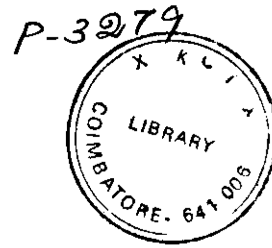
There is a significant body of research dealing with EDF scheduling in real-time computing, it is possible to calculate worst case response times of processes in EDF, to deal with other types of processes than periodic processes and to use servers to regulate overloads.

Undesirable deadline interchanges may occur with EDF scheduling. When a shared resource is accessed by processes using critical sections within a process (to prevent it from being pre-empted by another process with an earlier deadline waiting for access to the same shared resource), it becomes important for the scheduler to

temporarily assign the earliest deadline from amongst the other processes waiting for the resource, to the process while it is within its critical section to prevent the processes with earlier deadlines miss their respective deadline, especially if the process within its critical section has a much longer time to complete and its exit from its critical section and subsequent release of the shared resource may be delayed. Also it may be further delayed by other processes with earlier deadlines which do not share the same resource and thus can preempt it during its critical section. This hazard of deadline interchange within a critical section is analogous to priority inversion when using fixed priority pre-emptive scheduling.

To speed up the search within a linked list queue, the waiting processes within the list should be sorted according to their deadlines. When a cyclic or new process is given a new deadline, it is then inserted before the first process with a later deadline. This way, the processes with the earliest deadlines are always at the beginning of the list, reducing the time to find them.

P-3279





### 2.1.4 LEAST SLACK TIME FIRST [3]

The LSF (Least Slack First) algorithm assigns a priority to a task according to its executing urgency. The smaller the remaining slack time of a task is, the sooner it needs to be executed. However, LSF may frequently cause switching or serious thrashing among tasks, which augments the overhead of a system and restricts its application. Assigning a preemption threshold in the scheduling policy can decrease the switching among tasks, however, the existing assigning methods are limited to the fixed priority such that they are not applied to the LSF algorithm. In order to relieve the thrashing caused by LSF, some applicable assigning schemes are presented to the LSF algorithm based on the preemption threshold. Every task is dynamically assigned a preemption threshold that is dynamically changing with the executing urgency of the task and is not limited by the number of tasks. Simulations show that, by using the improved LSF policy, the switching among tasks decreases greatly while the missed deadline percentage decrease

More formally, the *slack time* for a process is defined as:

$$(d - t) - c'$$

Where $d$ is the process deadline, $t$ is the real time since the cycle start, and $c'$ is the remaining computation time. LST scheduling is most useful in systems comprising mainly aperiodic tasks, because no prior assumptions are made on the events rate of occurrence. The main weakness of LST is that it does not look ahead, and works only on the current system state. Thus, during a brief overload of system resources, LST can be sub-optimal. It will also be suboptimal when used with uninterruptible processes.

## 2.2 O(1) SCHEDULING ALGORITHM [10]

During the 2.5 kernel development series, the Linux kernel received a new scheduler, commonly called the O(1)scheduler because its algorithmic behavior(O(1) is an example of big-o notation. In short, it means the scheduler can do its thing in constant time, regardless of the size of the input), solved the short comings of the previous Linux scheduler and introduced powerful new features and performance characteristics. A common type of scheduling algorithm is priority-based scheduling. The idea is to rank processes based on their worth and need for processor time. Processes with a higher priority run before those with a lower priority, whereas processes with the same priority are scheduled round-robin (one after the next, repeating). The processes with a higher priority also receive a longer timeslice. The runnable process with timeslice remaining and the highest priority always runs. Both the user and the system may set a process's priority to influence the scheduling behavior of the system. Linux builds on this idea and provides dynamic priority-based scheduling for non-real time tasks. This concept begins with an initial base priority and then enables the scheduler to increase or decrease the priority dynamically to fulfill scheduling objectives. For example, a process that is spending more time waiting on I/O than running is clearly I/O bound. Under Linux, it receives an elevated dynamic priority. As a counterexample, a process that continually uses up its entire timeslice is processor bound it would receive a lowered dynamic priority.

There are sufficient reasons to choose the Linux 2.6.9.34 edition to modify,

- Linux is open source, it's easy to understand and rewrite
- It's almost supported by diversified hardware

- It's designed in a modularization mind, so it can easily transplant to an embedded system by simplify its module.
- Linux 2.6.9.34 edition's kernel use O(1) schedule algorithm, which try to decrease the overall execution time of all tasks. To sum up, Linux 2.6.9.34 edition match requirements perfectly.

The reason for choosing O(1) scheduling algorithm to ameliorate because it is designed with specific goals in mind, they are,

- Implement fully O(1) scheduling: Every algorithm in the new scheduler completes in constant-time, regardless of the number of running processes.
- Implement perfect SMP scalability: Each processor has its own locking and individual runqueue.
- Implement improved SMP affinity: Attempt to group tasks to a specific CPU and continue to run them there. Only migrate tasks from one CPU to another to resolve imbalances in runqueue sizes.
- Provide good interactive performance: Even during considerable system load, the system should react and schedule interactive tasks immediately.
- Provide fairness: No process should find itself starved of timeslice for any reasonable amount of time. Likewise, no process should receive an unfairly high amount of timeslice.

### 2.2.1 Runqueues

The basic data structure in the scheduler is the runqueue. The runqueue is defined in kernel/sched.c as struct runqueue. The runqueue is the list of runnable processes on a given processor, there is one runqueue per processor. Each runnable process is on exactly one runqueue. The runqueue additionally contains per-processor scheduling information. Consequently, the runqueue is the primary scheduling data structure for each processor. The structure of runqueue with active array is shown in figure 2.1

**Priority Array**

Each runqueue contains two priority arrays, the active and the expired array. Priority arrays are defined in kernel/sched.c as struct prio_array. Priority arrays are the data structures that provide O(1) scheduling. Each priority array contains one queue of runnable processes per priority level. The priority arrays also contain a priority bitmap used to efficiently discover the highest-priority runnable task in the system.

The members of the structure priority array are nr_active ,bitmap,queue.MAX_PRIO is the number of priority levels on the system. By default, this is 140. Thus, there is one struct list_head for each priority. BITMAP_SIZE is the size that an array of unsigned long typed variables would have to be to provide one bit for each valid priority level. With 140 priorities and 32-bit words, this is five. Thus, bitmap is an array with five elements and a total of 160 bits.

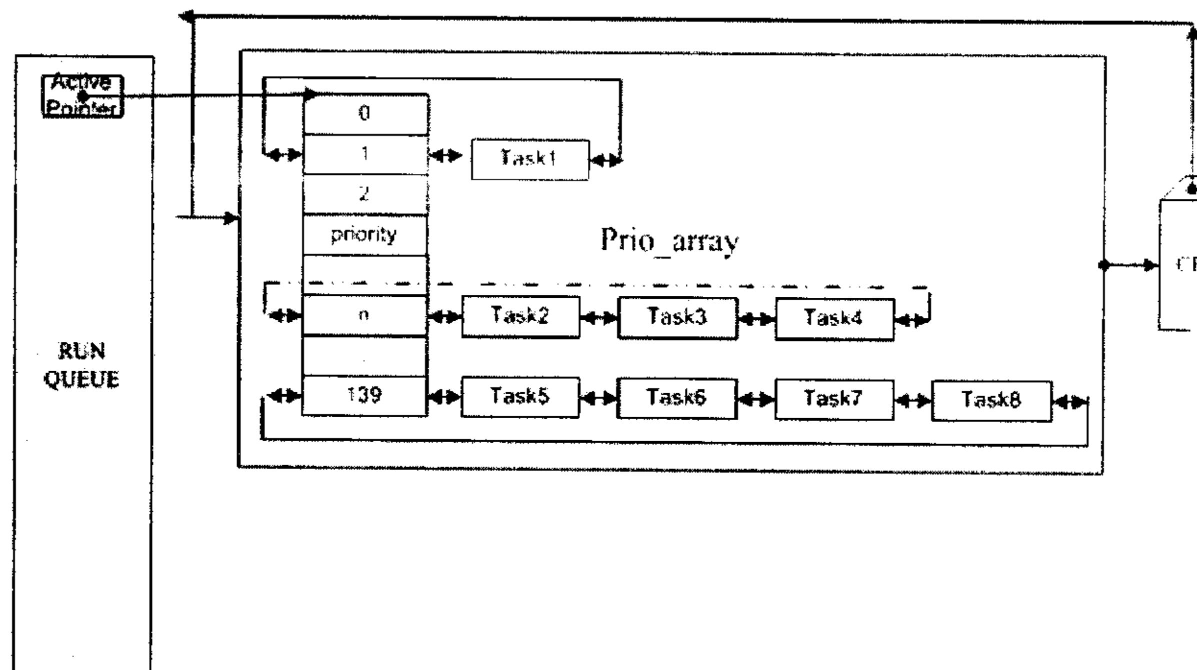



**Figure 2.1: Runqueue of O(1) scheduling algorithm with active priority array**

Each priority array contains a bitmap field that has at least one bit for every priority on the system. Initially, all the bits are zero. When a task of a given priority becomes runnable (that is, its state is set to TASK_RUNNING), the corresponding bit in the bitmap is set to one. For example, if a task with priority seven is runnable, then bit seven is set. Finding the highest priority task on the system is therefore only a matter of finding the first set bit in the bitmap. Because the number of priorities is static, the time to complete this search is constant and unaffected by the number of running processes on the system. Furthermore, each supported architecture in Linux implements a fast find first set algorithm to quickly search the bitmap. This method is called sched_find_first_bit(). Much architecture provides a find-first-set instruction that

operates on a given word. On these systems, finding the first set bit is as trivial as executing this instruction at most a couple of times.

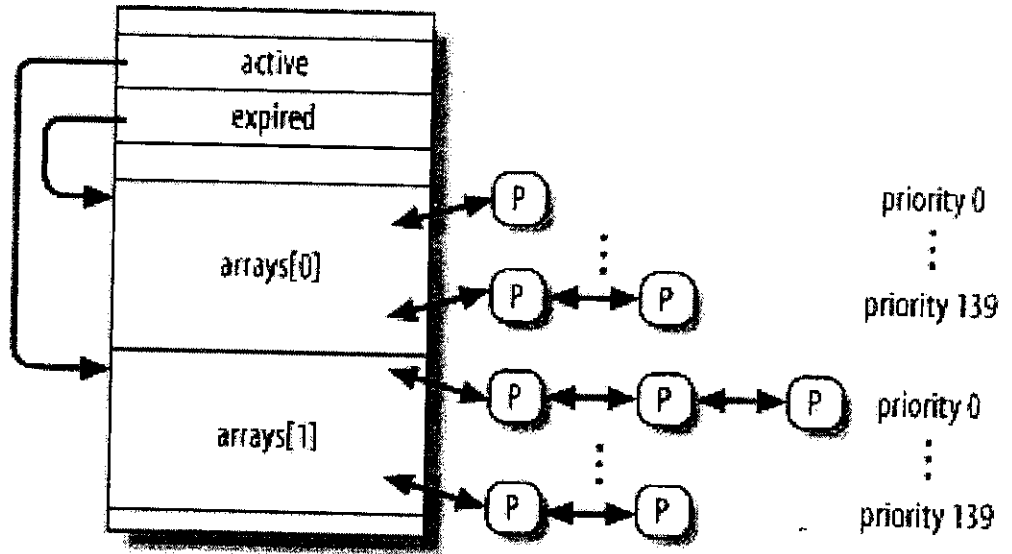



**Figure 2.2: Runqueue with two set of runnable process**

Each priority array also contains an array named queue of struct list_head queues, one queue for each priority. Each list corresponds to a given priority and in fact contains all the runnable processes of that priority that are on this processor's runqueue. Finding the next task to run is as simple as selecting the next element in the list. Within a given priority, tasks are scheduled round robin.The priority array also contains a counter, nr_active. This is the number of runnable tasks in this priority array.

The scheduler maintains two priority arrays for each processor: both an active array and an expired array. The active array contains all the tasks in the associated runqueue that have timeslice left. The expired array contains all the tasks in the associated

runqueue that have exhausted their timeslice. When each task's timeslice reaches zero, its timeslice is recalculated before it is moved to the expired array. Recalculating all the timeslices is then as simple as just switching the active and expired arrays. Because the arrays are accessed only via pointer, switching them is as fast as swapping two pointers. Figure 2.4 shows the runqueue with two sets of runnable tasks.

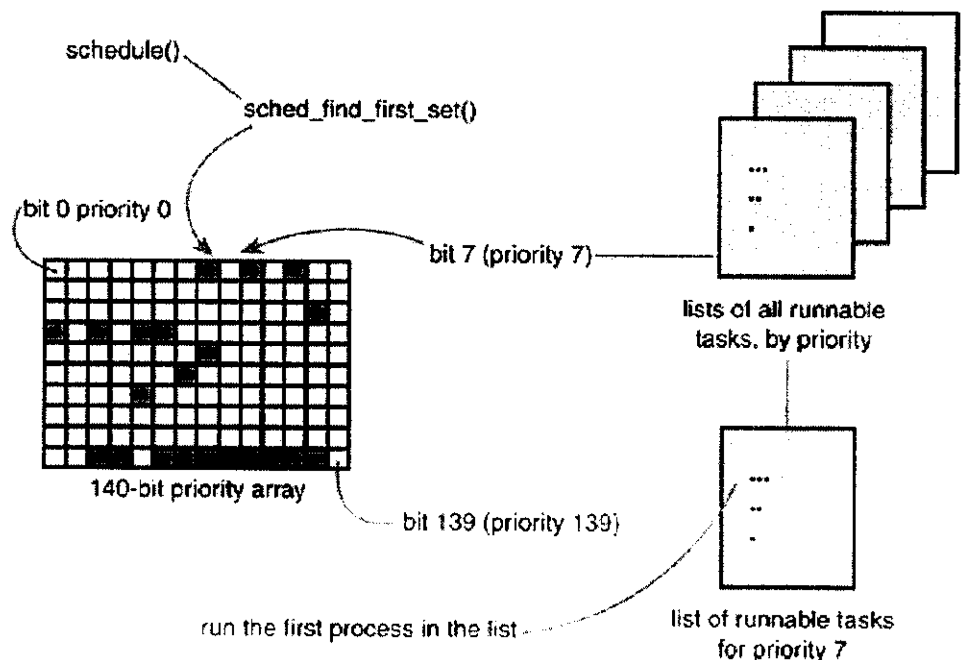



**Figure 2.3: Selecting the highest priority task in runqueue**

### 2.2.2 Timeslice

The timeslice is the numeric value that represents how long a task can run until it is preempted. The scheduler policy must dictate a default timeslice, which is not a trivial

exercise. Too long a timeslice causes the system to have poor interactive performance; the system will no longer feel as if applications are concurrently executed. Too short a timeslice causes significant amounts of processor time to be wasted on the overhead of switching processes because a significant percentage of the system's time is spent switching from one process with a short timeslice to the next. Furthermore, the conflicting goals of I/O-bound versus processor-bound processes again arise: I/O-bound processes do not need longer timeslices (although they do like to run often), whereas processor-bound processes crave long timeslices

With this argument, it would seem that any long timeslice would result in poor interactive performance. In many operating systems, this observation is taken to heart, and the default timeslice is rather low for example, 20ms. Linux, however, takes advantage of the fact that the highest priority process always runs. The Linux scheduler bumps the priority of interactive tasks, enabling them to run more frequently. Consequently, the Linux scheduler offers a relatively high default timeslice. Furthermore, the Linux scheduler dynamically determines the timeslice of a process based on priority. This enables higher-priority (allegedly more important) processes to run longer and more often. Implementing dynamic timeslices and priorities provides robust scheduling performance.

Note that a process does not have to use all its timeslice at once. For example, a process with a 100-millisecond timeslice does not have to run for 100 milliseconds in one go or risk losing the remaining timeslice. Instead, the process can run on five different reschedules for 20 milliseconds each. Thus, a large timeslice also benefits

interactive tasks: Although they do not need such a large timeslice all at once, it ensures they remain runnable for as long as possible.

**Recalculating timeslice for non real time tasks**

When a process's timeslice runs out, the process is considered expired. A process with no timeslice is not eligible to run until all other processes have exhausted their timeslices (that is, they all have zero timeslice remaining). At that point, the timeslices for all processes are recalculated.

Processes have an initial priority that is called the nice value. This value ranges from 20 to +19 with a default of zero. Nineteen is the lowest and 20 is the highest priority. This value is stored in the static_prio member of the process's task_struct. The variable is called the static priority because it does not change from what the user specifies. The scheduler, in turn, bases its decisions on the dynamic priority that is stored in prio. The dynamic priority is calculated as a function of the static priority and the task's interactivity.

The method effective_prio() returns a task's dynamic priority. The method begins with the task's nice value and computes a bonus or penalty in the range 5 to +5 based on the interactivity of the task. For example, a highly interactive task with a nice value of ten can have a dynamic priority of five. Conversely, a mild processor hog with a nice value of ten can have a dynamic priority of 12. Tasks that are only mildly interactive at some theoretical equilibrium of I/O versus processor usage receive no bonus or penalty and their dynamic priority is equal to their nice value.

Of course, the scheduler does not magically know whether a process is interactive. It must use some heuristic that is capable of accurately reflecting whether a task is I/O bound or processor bound. The most indicative metric is how long the task sleeps. If a task spends most of its time asleep, then it is I/O bound. If a task spends more time runnable than sleeping, it is certainly not interactive. This extends to the extreme: A task that spends nearly all the time sleeping is completely I/O bound, whereas a task that spends nearly all its time runnable is completely processor bound.

To implement this heuristic, Linux keeps a running tab on how much time a process is spent sleeping versus how much time the process spends in a runnable state. This value is stored in the sleep_avg member of the task_struct. It ranges from zero to MAX_SLEEP_AVG, which defaults to 10 milliseconds. When a task becomes runnable after sleeping, sleep_avg is incremented by how long it slept, until the value reaches MAX_SLEEP_AVG. For every timer tick the task runs, sleep_avg is decremented until it reaches zero.

This metric is surprisingly accurate. It is computed based not only on how long the task sleeps but also on how little it runs. Therefore, a task that spends a great deal of time sleeping, but also continually exhausts its timeslice, will not be awarded a huge bonusthe metric works not just to award interactive tasks but also to punish processor-bound tasks. It is also not vulnerable to abuse. A task that receives a boosted priority and timeslice quickly loses the bonus if it turns around and hogs the processor. Finally, the metric provides quick response. A newly created interactive process quickly receives a large sleep_avg. Despite this, because the bonus or penalty is applied

against the initial nice value, the user can still influence the system's scheduling decisions by changing the process's nice value.

Timeslice, on the other hand, is a much simpler calculation. It is based on the static priority. When a process is first created, the new child and the parent split the parent's remaining timeslice. This provides fairness and prevents users from forking new children to get unlimited timeslice. After a task's timeslice is exhausted, however, it is recalculated based on the task's static priority. The function task_timeslice() returns a new timeslice for the given task. The calculation is a simple scaling of the static priority into a range of timeslices. The higher a task's priority, the more timeslice it receives per round of execution.

# CHAPTER 3

# DETAILS OF MODIFIED O(1) SCHEDULING ALGORITHM

## 3.1 Improvement of queue management

The basic data structure in the scheduler is the runqueue. The runqueue is defined in kernel/sched.c as struct runqueue. The runqueue is the list of runnable processes on a given processor; there is one runqueue per processor. Each runnable process is on exactly one runqueue. The runqueue additionally contains per-processor scheduling information. Consequently, the runqueue is the primary scheduling data structure for each processor

### Priority Array

Each runqueue contains one priority array, the active array. Priority arrays are defined in kernel/sched.c as struct prio_array. Priority arrays are the data structures that provide O(1) scheduling. Priority array contains one queue of runnable processes per priority level. The priority arrays also contain a priority bitmap used to efficiently discover the highest-priority runnable task in the system.

The members of the structure priority array are nr_active ,bitmap,queue.MAX_PRIO is the number of priority levels on the system. By default, this is 140. Thus, there is one struct list_head for each priority. BITMAP_SIZE is the size that an array of unsigned long typed variables would have to be to provide one bit for each valid priority level. With 140 priorities and 32-bit words, this is five. Thus, bitmap is an array with five elements and a total of 160 bits.

The priority array contains a bitmap field that has at least one bit for every priority on the system. Initially, all the bits are zero. When a task of a given priority becomes runnable (that is, its state is set to TASK_RUNNING), the corresponding bit in the bitmap is set to one. For example, if a task with priority seven is runnable, then bit seven is set. Finding the highest priority task on the system is therefore only a matter of finding the first set bit in the bitmap. Because the number of priorities is static, the time to complete this search is constant and unaffected by the number of running processes on the system. Furthermore, each supported architecture in Linux implements a fast find first set algorithm to quickly search the bitmap. This method is called sched_find_first_bit(). Much architecture provides a find-first-set instruction that operates on a given word. On these systems, finding the first set bit is as trivial as executing this instruction at most a couple of times.

Each priority array also contains an array named queue of struct list_head queues, one queue for each priority. Each list corresponds to a given priority and in fact contains all the runnable processes of that priority that are on this processor's runqueue. Finding the next task to run is as simple as selecting the next element in the list. Within a given priority, tasks are scheduled round robin. The priority array also contains a counter, nr_active. This is the number of runnable tasks in this priority array. The scheduler maintains one priority array for the processor: an active array. The active array contains all the tasks in the associated runqueue that have timeslice left. When each task's timeslice reaches zero, its timeslice is recalculated before it is moved to the tail of the queue. By using one priority array, we save time caused by context switching and arrays swapping. When a process's time reach zero, it will be moved to the tail of its

priority runqueue, and wait for next timeslice. This method will improve the efficiency of real-time tasks obviously.

The act of picking the next task to run and switching to it is implemented via the schedule() function. This function is called explicitly by kernel code that wants to sleep and it is invoked whenever a task is to be preempted. The schedule() function is run by processor, which makes its own decisions on what process to run next. First, the active priority array is searched to find the first set bit. This bit corresponds to the highest priority task that is runnable. Next, the scheduler selects the first task in the list at that priority. This is the highest priority runnable task on the system and is the task the scheduler will run. Two important points should be noted from the previous code. First, it is very simple and consequently quite fast. Second, the number of processes on the system has no effect on how long this code takes to execute. There is no loop over any list to find the most suitable process. In fact, nothing affects how long the schedule() code takes to find a new task. It is constant in execution time.

## 3.2 Improvement of Process analysis

The key feature of this algorithm in process analysis is that the algorithm distributes timeslice dynamically. Processes can be classified as either I/O-bound or processor-bound. I/O-bound is characterized as a process that spends much of its time waiting on I/O requests. Consequently, such a process is often runnable. Conversely, processor-bound processes spend much of their time executing code. The scheduling policy in a system must attempt to satisfy two conflicting goals: fast process response time (low latency) and maximal system utilization (high throughput). The project aims to provide good interactive response, optimizes for process response (low latency), thus favoring I/O-bound processes over processor-bound processors. An I/O queue array is made. Each array contains one queue of I/O-bound processes. Each queue corresponds to one I/O response. When the I/O finish, the waiting process is swapped quickly from I/O queue runqueue in a short time.

### IO Waiting Queue

Tasks that are sleeping (blocked) are in a special non-runnable state. This is important because without this special state, the scheduler would select tasks that did not want to run or, worse, sleeping would have to be implemented as busy looping. A task sleeps for a number of reasons, but always while it is waiting for some event. The event can be a specified amount of time, more data from a file I/O, or another hardware event. A task can also involuntarily go to sleep when it tries to obtain a contended semaphore in the kernel .A common reason to sleep is file I/O for example, the task issued a read() request on a file, which needs to be read in from disk. As another example, the task could be waiting for keyboard input. Whatever the case, the kernel

behavior is the same:The task marks itself as sleeping, puts itself on a wait queue, removes itself from the runqueue, and calls schedule() to select a new process to execute. Waking back up is the inverse: the task is set as runnable, removed from the wait queue, and added back to the runqueue.

Sleeping is handled via wait queues. A wait queue is a simple list of processes waiting for an event to occur. Processes put themselves on a wait queue and mark themselves not runnable. When the event associated with the wait queue occurs, the processes on the queue are awakened. It is important to implement sleeping and waking correctly, to avoid race conditions.

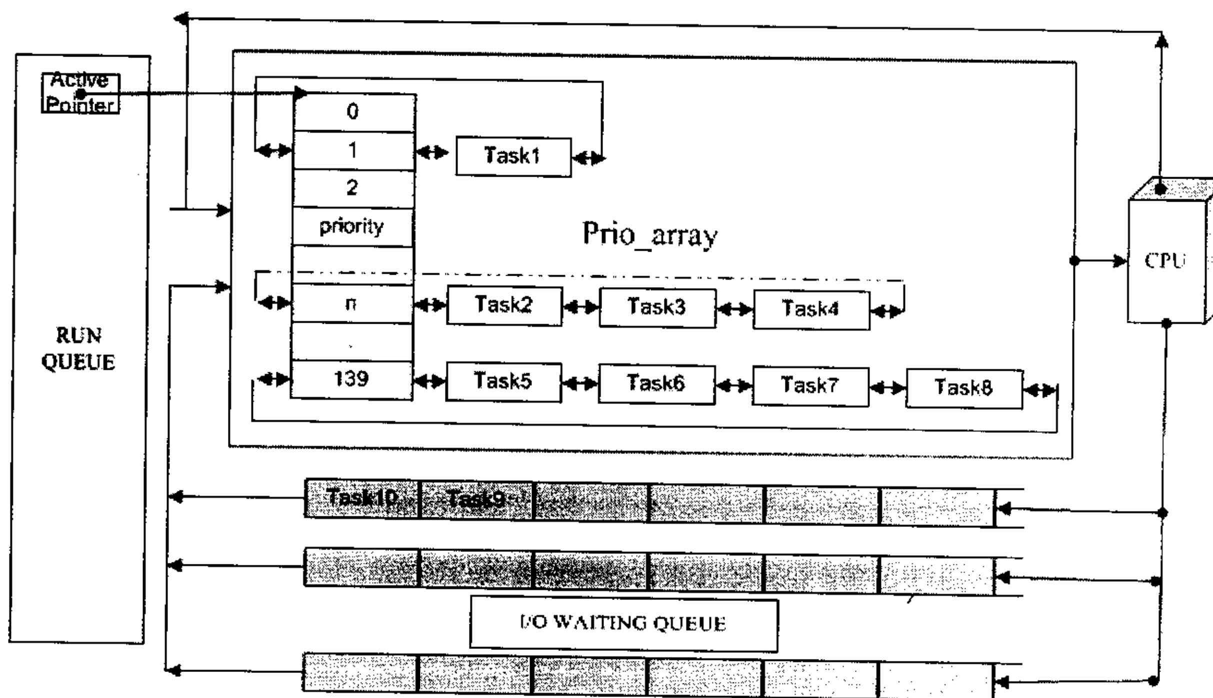



**Figure 3.1:Runqueue of Modified O(1) Scheduling algorithm**

The task performs the following steps to add itself to a wait queue:

1. Creates a wait queue entry.
2. Adds itself to a wait queue via add_wait_queue(). This wait queue awakens the process when the condition for which it is waiting occurs. Of course, there needs to be code elsewhere that calls wake_up() on the queue when the event actually does occur.
3. Changes the process state to TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE.
4. If the state is set to TASK_INTERRUPTIBLE, a signal wakes the process up. This is called a spurious wake up (a wake-up not caused by the occurrence of the event). So check and handle signals.
5. Tests whether the condition is true. If it is, there is no need to sleep. If it is not true, the task calls schedule().
6. When the task awakens, it again checks whether the condition is true. If it is, it exits the loop. Otherwise, it again calls schedule() and repeats.
7. Now that the condition is true, the task can set itself to TASK_RUNNING and remove itself from the wait queue via remove_wait_queue().

If race condition occurs before the task goes to sleep, the loop terminates, and the task does not erroneously go to sleep. Note that kernel code often has to perform various other tasks in the body of the loop. For example, it might need to release locks before calling schedule() and reacquire them after or react to other events.

Waking is handled via wake_up(), which wakes up all the tasks waiting on the given wait queue. It calls try_to_wake_up(), which sets the task's state to

TASK_RUNNING, calls activate_task() to add the task to a runqueue, and sets need_resched if the awakened task's priority is higher than the priority of the current task. The code that causes the event to occur typically calls wake_up() afterward. For example, when data arrives from the hard disk, the VFS calls wake_up() on the wait queue that holds the processes waiting for the data.

An important note about sleeping is that there are spurious wake-ups. Just because a task is awakened does not mean that the event for which the task is waiting has occurred; sleeping should always be handled in a loop that ensures that the condition for which the task is waiting has indeed occurred.

# CHAPTER 4

# SIMULATION AND ANALYSIS

Experiment environment is based on CPU: 1.73 GHz, Memory: 256MB, Hard Disk: 8GB. The test program to test Modified algorithm and Linux 2.6.9.34, creates a group of tasks. Each client/server pair listens on a socket, the writer (client) send 10000 messages to each socket and the receiver (server) listens on the socket. These are typical real-time tasks. We use a shell script to run 20, 40, 60 tasks in Linux 2.6.9.34 and Modified algorithm. The Results are shown in Figure 4.1 and Figure 4.2. The results show that the overall completion time of all the tasks and the response time of each task in Modified algorithm are less compared to that of Linux 2.6.9-34.

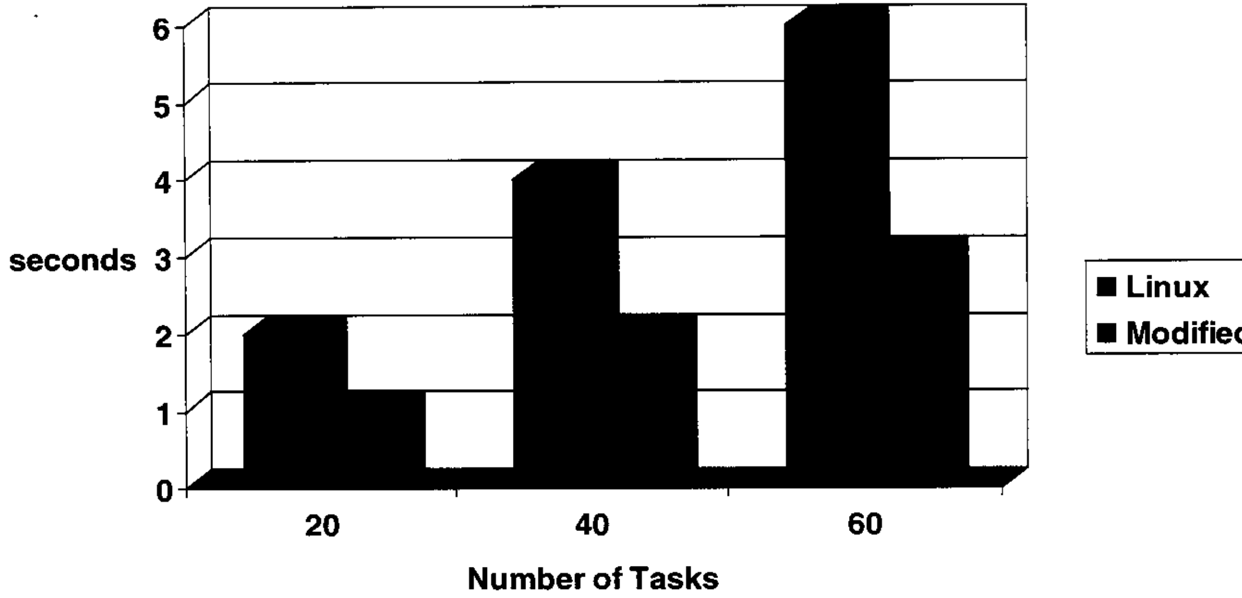



**Figure 4.1 Comparison of overall completion of all processes**

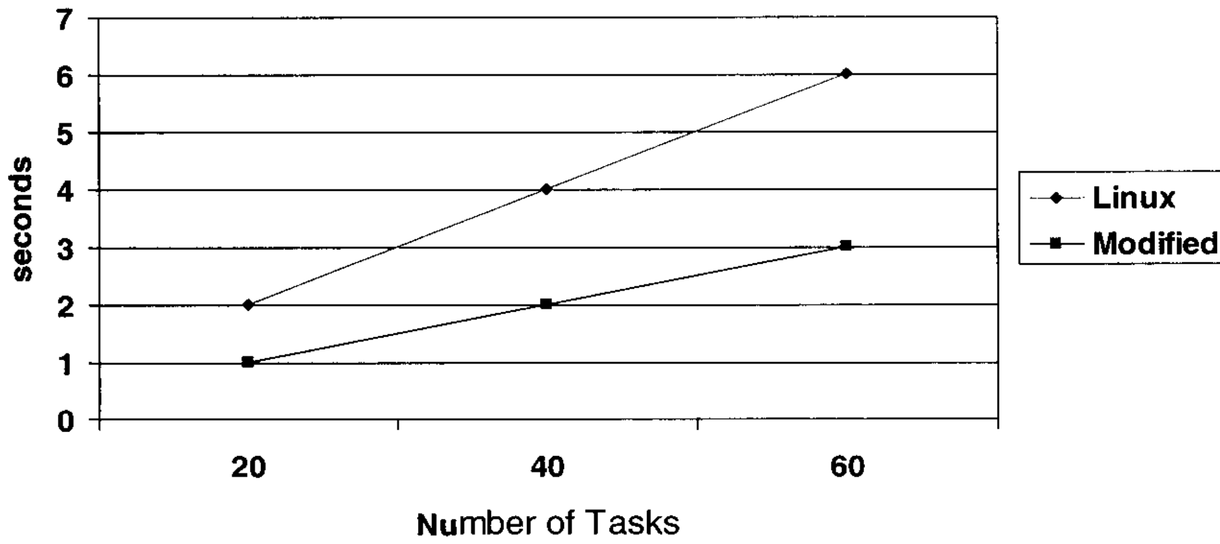



**Figure 4.2: Comparison of response of real time tasks**

# CHAPTER 5

# CONCLUSION AND FUTURE ENHANCEMENTS

In this project, **A Modified O(1) Scheduling Algorithm for Real-time Tasks** based on Linux 2.6.9.34 kernel, a scheduling algorithm which is a mixture of normal operating system scheduling and real-time operating system scheduling is presented. Due to the advancement in computer technology, embedded microcontrollers of near future will have the processing capability of today's servers, thus the need for real-time operating system to support applications with mixed real-time and non real-time performance requirements increases. Linux is modified to satisfy this demand. This modified algorithm deals with the real-time tasks rapidly and accurately, it also considers other kinds of tasks.

For further work, the work needed to be improved, to make real time tasks execute within their deadline and test more complex real-time processes. Also the modified algorithms compatibility with other Real time systems must be improved.

## APPENDIX

### Code for Runqueue

```
        struct runqueue
{
        spinlock_t lock;

        /*
         * nr_running and cpu_load should be in the same cacheline because
         * remote CPUs use both these fields when doing load calculation.
         */
        unsigned long nr_running;
        #ifdef CONFIG_SMP
                unsigned long cpu_load;
        #endif
        unsigned long long nr_switches;

        /*
         * This is part of a global counter where only the total sum
         * over all CPUs matters. A task can increase this counter on
         * one CPU and if it got migrated afterwards it may decrease
         * it on another CPU. Always updated under the runqueue lock:
         */
        unsigned long nr_uninterruptible;
```

```
unsigned long long timestamp_last_tick;
task_t *curr, *idle;
struct mm_struct *prev_mm;
prio_array_t *active,  arrays;

atomic_t nr_iowait;

#ifdef CONFIG_SMP
        struct sched_domain *sd;

        /* For active balancing */
        int active_balance;
        int push_cpu;

        task_t *migration_thread;
        struct list_head migration_queue;
#endif

#ifdef CONFIG_SCHEDSTATS
        /* latency stats */
        struct sched_info rq_sched_info;

        /* sys_sched_yield() stats */
        unsigned long yld_exp_empty;
```

```
unsigned long yld_act_empty;
unsigned long yld_both_empty;
unsigned long yld_cnt;

/* schedule() stats */
unsigned long sched_noswitch;
unsigned long sched_switch;
unsigned long sched_cnt;
unsigned long sched_goidle;

/* pull_task() stats */
unsigned long pt_gained[MAX_IDLE_TYPES];
unsigned long pt_lost[MAX_IDLE_TYPES];

/* active_load_balance() stats */
unsigned long alb_cnt;
unsigned long alb_lost;
unsigned long alb_gained;
unsigned long alb_failed;

/* try_to_wake_up() stats */
unsigned long ttwu_cnt;
unsigned long ttwu_attempts;
unsigned long ttwu_moved;
```

```
            /* wake_up_new_task() stats */
            unsigned long wunt_cnt;
            unsigned long wunt_moved;

            /* sched_migrate_task() stats */
            unsigned long smt_cnt;

            /* sched_balance_exec() stats */
            unsigned long sbe_cnt;
        #endif
};
```

**Code for Priority Array**

```
struct prio_array
{
        unsigned int nr_active;
        unsigned long bitmap[BITMAP_SIZE];
        struct list_head queue[MAX_PRIO];
};
```

**Code for IO Waiting Queue**

```
add_wait_queue(q, &wait);
while (!condition) {     /* condition is the event that we are waiting for */
      set_current_state(TASK_INTERRUPTIBLE); /* or TASK_UNINTERRUPTIBLE */
```

```
        if (signal_pending(current))
            /* handle signal */
        schedule();
}
set_current_state(TASK_RUNNING);
remove_wait_queue(q, &wait);
```

**Code for recalculating timeslice**

```
static void recalc_task_prio(task_t *p, unsigned long long now)
{
        unsigned long long __sleep_time = now - p->timestamp;
        unsigned long sleep_time;

        if (__sleep_time > NS_MAX_SLEEP_AVG)
                sleep_time = NS_MAX_SLEEP_AVG;
        else
                sleep_time = (unsigned long)__sleep_time;

        if (likely(sleep_time > 0)) {
                /*
                 * User tasks that sleep a long time are categorised as
                 * idle and will get just interactive status to stay active &
                 * prevent them suddenly becoming cpu hogs and starving
                 * other processes.
                 */
```

```
if (p->mm && p->activated != -1 &&
        sleep_time > INTERACTIVE_SLEEP(p)) {
            p->sleep_avg = JIFFIES_TO_NS(MAX_SLEEP_AVG -
                        DEF_TIMESLICE);
} else {
        /*
         * The lower the sleep avg a task has the more
         * rapidly it will rise with sleep time.
         */
        sleep_time *= (MAX_BONUS - CURRENT_BONUS(p)) ? : 1;

        /*
         * Tasks waking from uninterruptible sleep are
         * limited in their sleep_avg rise as they
         * are likely to be waiting on I/O
         */
        if (p->activated == -1 && p->mm) {
            if (p->sleep_avg >= INTERACTIVE_SLEEP(p))
                sleep_time = 0;
            else if (p->sleep_avg + sleep_time >=
                        INTERACTIVE_SLEEP(p)) {
                p->sleep_avg = INTERACTIVE_SLEEP(p);
                sleep_time = 0;
            }
```

```
                }
                /*
                 * This code gives a bonus to interactive tasks.
                 *
                 * The boost works by updating the 'average sleep time'
                 * value here, based on ->timestamp. The more time a
                 * task spends sleeping, the higher the average gets -
                 * and the higher the priority boost gets as well.
                 */
                p->sleep_avg += sleep_time;

                if (p->sleep_avg > NS_MAX_SLEEP_AVG)
                        p->sleep_avg = NS_MAX_SLEEP_AVG;
        }
    }

    p->prio = effective_prio(p);
}
```

**<u>Code for Real Time Task Scheduling</u>**

```
void scheduler_tick(void)
{
        int cpu = smp_processor_id();
        runqueue_t *rq = this_rq();
        task_t *p = current;
```

```
unsigned long long now = sched_clock();

update_cpu_clock(p, rq, now);
rq->timestamp_last_tick = now;

if (p == rq->idle) {
        if (wake_priority_sleeper(rq))
                goto out;
        rebalance_tick(cpu, rq, SCHED_IDLE);
        return;
}

/* Task might have expired already, but not scheduled off yet */
if (p->array != rq->active) {
        set_tsk_need_resched(p);
        goto out;
}
spin_lock(&rq->lock);
/*
 * The task was running during this tick - update the
 * time slice counter. Note: we do not update a thread's
 * priority until it either goes to sleep or uses up its
 * timeslice. This makes it possible for interactive tasks
 * to use up their timeslices at their highest priority levels.
 */
```

```
if (rt_task(p)) {
        /*
         * RR tasks need a special form of timeslice management.
         * FIFO tasks have no timeslices.
         */
        if ((p->policy == SCHED_RR) && !--p->time_slice) {
                p->time_slice = task_timeslice(p);
                p->first_time_slice = 0;
                set_tsk_need_resched(p);

                /* put it at the end of the queue: */
                requeue_task(p, rq->active);
        }
        goto out_unlock;
}
if (!--p->time_slice) {
        dequeue_task(p, rq->active);
        set_tsk_need_resched(p);
        p->prio = effective_prio(p);
        p->time_slice = task_timeslice(p);
        p->first_time_slice = 0;

        if (!rq->expired_timestamp)
                rq->expired_timestamp = jiffies;
```

```
        if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
                enqueue_task(p, rq->expired);
                if (p->static_prio < rq->best_expired_prio)
                        rq->best_expired_prio = p->static_prio;
        } else
                enqueue_task(p, rq->active);
} else {
        /*
         * Prevent a too long timeslice allowing a task to monopolize
         * the CPU. We do this by splitting up the timeslice into
         * smaller pieces.
         *
         * Note: this does not mean the task's timeslices expire or
         * get lost in any way, they just might be preempted by
         * another task of equal priority. (one with higher
         * priority would have preempted this task already.) We
         * requeue this task to the end of the list on this priority
         * level, which is in essence a round-robin of tasks with
         * equal priority.
         *
         * This only applies to tasks in the interactive
         * delta range with at least TIMESLICE_GRANULARITY to requeue.
         */
        if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
```

```
                    p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
                    (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
                    (p->array == rq->active)) {

                    requeue_task(p, rq->active);
                    set_tsk_need_resched(p);
                }
        }
out_unlock:
        spin_unlock(&rq->lock);
out:
        rebalance_tick(cpu, rq, NOT_IDLE);
}
```

**Shell script to run many real time tasks**

```
echo -n "Enter the directory Path where Server/Client program resides:"
read i
s_dir=$i

dir=/tmp/temp
        if [ -d $dir ]; then

                echo "TEMP Directory  present"
        else
                mkdir -p $dir/object
                echo "Directory created"
        fi

ls -l $s_dir | awk '{print $9}' | grep -v '^$' > $dir/tmp.txt

YTotal=$(cat $dir/tmp.txt|wc -l)
YLINES=1

if [ ${YTotal} -gt 0 ] ;then

        while [ $YLINES -le ${YTotal} ] ;do
```

```
        line=$(head -$YLINES $dir/tmp.txt| tail -1)

        echo "$line" >> $dir/summary.log

        cd $s_dir

        start_time=`date +%S` >> $dir/summary.log

        gcc -o $line.o $line 2>> $dir/error.log >> $dir/summary.log

        ./$line.o 2>> $dir/error.log >> $dir/output.log

        mv *.o $dir/object

        end_time=`date +%S` >> $dir/summary.log

        program_rt=$(($end_time-$start_time))

        echo "Program run time in seconds: $program_rt" >> $dir/summary.log
        YLINES=$(expr $YLINES + 1)
done

fi
```

**TEST PROGRAM:**

**SERVER:**

```
#include <time.h>

#include <sched.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

void error(char *msg)
{
   perror(msg);
   exit(1);
}

int main()
{
        int priority = 10;
        FILE *fp,*tp;
        time_t now;
        time( &now );
        if( fp = fopen("/home/sst.txt","a"))
        fprintf(fp,"\npri=%d start time= %.24s",priority,ctime(&now));
```

```
struct sched_param sp;
int ret;
sp.sched_priority = priority;
ret = sched_setscheduler(0, SCHED_RR, &sp);


int sockfd, newsockfd, portno, clilen,i;
char buffer[256];
struct sockaddr_in serv_addr, cli_addr;
int n;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
        error("ERROR opening socket");
bzero((char *) &serv_addr, sizeof(serv_addr));
portno = 2010;
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);
if (bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
   error("ERROR on binding");
listen(sockfd,5);
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd,(struct sockaddr *) &cli_addr,&clilen);
if (newsockfd < 0)
```

```
    error("ERROR on accept");
   bzero(buffer,256);
   n = read(newsockfd,buffer,255);
   if (n < 0)
          error("ERROR reading from socket");
   time( &now );
   if( tp = fopen("/home/set.txt","a"))
   fprintf(tp,"\nmain : priority = %d  end time=%.24s",priority,ctime(&now));
   return 0;
}
```

**CLIENT:**

```
#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include<sched.h>

void error(char *msg)
{
   perror(msg);
   exit(0);
}

int main()
{
        int priority = 10;
        FILE *fp,*tp;
        time_t now;
        time( &now );
        if( fp = fopen("/home/cst.txt","a"))
        fprintf(fp,"\npri=%d start time= %.24s",priority,ctime(&now));
```

```
struct sched_param sp;
int ret;
sp.sched_priority = priority;
ret = sched_setscheduler(0, SCHED_RR, &sp);


int sockfd, portno, n, i;
struct sockaddr_in serv_addr;
struct hostent *server;

char buffer[256];
portno = 2010;

sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
error("ERROR opening socket");
server = gethostbyname("localhost");
if (server == NULL)
{
fprintf(stderr,"ERROR, no such host\n");
exit(0);
}
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
```

```
bcopy((char *)server->h_addr,( char *)&serv_addr.sin_addr.s_addr, server-
>h_length);
serv_addr.sin_port = htons(portno);
if(connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr))<0)
error("ERROR connecting");
for(i=0;i<10000;i++)
{
bzero(buffer,256);
strcpy(buffer,"a");
n = write(sockfd,buffer,strlen(buffer));
}
if (n < 0)
error("ERROR writing to socket");
bzero(buffer,256);
time( &now );
if( tp = fopen("/home/cet.txt","a"))
fprintf(tp,"\npriority = %d, end time=%.24s",priority,ctime(&now));

return 0;
}
```

## REFERENCES

[1] Wang Chi, Zhou Huaibei, "A Modified O(1) Scheduling Algorithm for Real-time tasks". IEEE, 2006.1~4.

[2] Liu CL, Layland JW, "Scheduling algorithms for multiprogramming in a hard real-time environment". Journal of the ACM, 1973, 20(1):46~61.

[3] Jensen ED, Locke CD, Toduda H, "A time-driven scheduling model for real-time operating systems". In: Proc. of the 6th IEEE Real-Time Systems Symp. San Diego: IEEE Computer Press, 1985. 112~122.

[4] Buttazzo G, Spuri M, Sensini F, "Value vs. deadline in overload conditions." In: Proc. of the 19th IEEE Real-TimeData Mining Systems Symp. Pisa: IEEE Computer Society Press, 1995. 90~99.

[5] Biyabani SR, Stankovic JA, Ramamritham K, "The integration of deadline and criticalness in hard real-time scheduling". In: Proc. the 9th IEEE Real-Time Systems Symp. Huntsville: IEEE. Computer Society Press, 1988. 152~160.

[6] Tseng S-M, Chin YH, Yang W-P," Scheduling value-based transactions in real-time main-memory databases". In: Lin KJ, ed. Proc.of the 1st Int'l Workshop on Real-Time Databases: Issues and Applications. Newport Beach: Kluwer Academic Publishers, 1996.111~117.

[7] Burns A, Prasad D, Bondavalli A, Giandomenico FD, Ramamritham K, Stankovic J, Strigini L, "The meaning and role of value inscheduling flexible real-time systems". Journal of Systems Architecture, 2000,46(4):305~325.

[8] Y. Wang and **K.** Lin, “Implementing a general real-time scheduling framework in the red-linux real-time kernel”. In *Proceedings of IEEE Real-Time* Systems *Symposium*, Phoenix, December 1999.

[9] C. A. Waldspurger and W. E. Weihl, “Stride scheduling: Deterministic proportional-share resource management”. Technical Report MITILCSKM-528, Massachusetts Institute of Technology, June 1995.

[10] Robert Love,” Linux Kernel Development” Second Edition Jan 2006.

[11] Daniel P. Bovet, Marco Cesati, “Understanding the Linux Kernel”, Third Edition Nov 2005.