Linux Implementation of a New Model for Handling Task Dynamics in Proportional Share Based Scheduling Systems

A thesis presented to
the faculty of
the Russ College of Engineering and Technology of Ohio University

In partial fulfillment
of the requirements for the degree
Master of Science

Rami R. Al-Ouran
June 2010
© 2010 Rami R. Al-Ouran. All Rights Reserved.
This thesis titled
Linux Implementation of a New Model for Handling Task Dynamics in Proportional
Share Based Scheduling Systems

by
RAMI R. AL-OURAN

has been approved for
the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

______________________________
Frank Drews
Assistant Professor of Electrical Engineering and Computer Science

______________________________
Dennis Irwin
Dean, Russ College of Engineering and Technology
Abstract

AL-OURAN, RAMI R., M.S., June 2010, Electrical Engineering and Computer Science

Linux Implementation of a New Model for Handling Task Dynamics in Proportional Share Based Scheduling Systems (78 pp.)

Director of Thesis: Frank Drews

Proportional Share Scheduling (PSS) algorithms have two characteristics that make them attractive to be used in a scheduling design: flexibility and fairness. To support time sensitive tasks a new feature should be added to the PSS design which is reservation guarantee. A number of PSS algorithms have been introduced to add that feature. Earliest Eligible Virtual Deadline First (EEVDF) is a PSS based algorithm that succeeded in integrating flexibility and fairness along with reservation guarantee. The main feature of EEVDF is ensuring that the difference between the service time the task should receive in an ideal system and the service time it actually receives is bounded at all times which makes reservation guarantee possible. In this thesis a new model based on the EEVDF algorithm was implemented to dynamically support a mix of applications with different requirements in a seamless manner. These applications include time-sensitive applications, time sensitive applications with added benefit and regular applications. The new model also adds more flexibility to the original EEVDF model especially for handling system dynamics. This new model was implemented as a loadable kernel module which makes debugging and improving the model easier and more efficient.

Approved: __________________________

Frank Drews

Assistant Professor of Electrical Engineering and Computer Science
To Mom and Dad for their Love and Patience
Acknowledgements

First all thanks go to God for his many blessings and mercy. I would like to deeply thank my beloved family. My dear father thank you for believing in me and trusting me; through my whole life you always had faith in me and you were there to support me. To my dear mother who flourished me with her love and believed in me during my ups and downs. To my dear brother Nedal who I love and admire. To my beloved brother Hussam whom I think of and miss all the time. I would like to extend my greatest gratitude to my advisor Dr. Frank Drews who helped and supported me and is still helping me during this journey. Also I would like to deeply thank Dr. Lonnie Welch who supported me and whom I learn from everyday. I would also like to extend my thanks to my committee members Dr. Shawn Ostermann, Dr. Avinash Kodi and Dr. Khurrum Bhutta. Thanks to the Graduate College for supporting me financially. I learned so much there and will always be grateful. Special thanks to Michael Tedesco who was a great boss and a greater friend. Thanks to Dr. Usha Matta who was like family for me in Athens.
# Table of Contents

Abstract . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3  
Dedication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4  
Acknowledgements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5  
List of Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8  
List of Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9  

1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10  
1.1 Proportional Share Scheduling . . . . . . . . . . . . . . . . . . . . . 10  
1.2 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12  
1.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13  
1.4 Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14  

2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15  
2.1 Scheduling Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 15  
2.2 Proportional Share Scheduling . . . . . . . . . . . . . . . . . . . . . 16  
  2.2.1 The Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17  
  2.2.2 The Concept of Virtual Time . . . . . . . . . . . . . . . . . . . . 18  
  2.2.3 The EEVDF Algorithm . . . . . . . . . . . . . . . . . . . . . . . 20  
  2.2.4 Dynamic Operations in the Proportional Scheduler Algorithm 23  
  2.2.5 Extensions to PSS . . . . . . . . . . . . . . . . . . . . . . . . . . . 24  
    2.2.5.1 Extensions by Stoica et al. . . . . . . . . . . . . . . . . . 25  
    2.2.5.2 Extensions by Goddard et al. . . . . . . . . . . . . . . . . 26  
2.3 Proportional Share Scheduler Algorithms . . . . . . . . . . . . . . . . 27  
  2.3.1 Stride Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . 27  
  2.3.2 The BERT Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . 28  
  2.3.3 The SMART Scheduler . . . . . . . . . . . . . . . . . . . . . . . 28  
  2.3.4 Virtual-Time Round-Robin (VTRR) . . . . . . . . . . . . . . . . 29  
2.4 Linux Schedulers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29  
  2.4.1 The O(1) Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . 29  
  2.4.2 The Completely Fair Scheduler . . . . . . . . . . . . . . . . . . 31  

3 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34  
3.1 The Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35  
3.2 Handling Lag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38  
3.3 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39  
  3.3.1 Changes Made to the Linux Kernel . . . . . . . . . . . . . . . . 41  
  3.3.2 Main Data Structures . . . . . . . . . . . . . . . . . . . . . . . . 41
3.3.3 Function Pointers ........................................... 45
3.3.4 Task Creation ................................................. 48
3.3.5 Task Scheduling .............................................. 48

4 Experiments and Results ........................................... 56
  4.1 Experimental Setup ........................................... 56
  4.2 Types of Experiments .......................................... 56
  4.3 Results and Interpretation ................................... 57
    4.3.1 Regular tasks ........................................... 57
    4.3.2 Time Sensitive Tasks ................................... 61
    4.3.3 Time Sensitive Tasks with Benefit ...................... 63
    4.3.4 Mixed Tasks ............................................ 65
    4.3.5 Lag Charts ............................................. 67
  4.4 Handling System Dynamics .................................... 69
  4.5 Graceful Degradation ......................................... 70

5 Conclusions and Future Work .................................... 74
  5.1 Study Limitations ............................................ 74
  5.2 Future Work ................................................... 75
List of Tables

3.1 Effective Weight Calculation ........................................... 38
List of Figures

4.1 Three Regular Tasks with Initial Weights 3,2,1 ............... 58
4.2 Five Regular Tasks with Initial Weights 8,3,2,6,4 ............... 59
4.3 Three Time Sensitive Tasks with Initial Shares 2,6,1 ........... 61
4.4 Four Time Sensitive Tasks with Initial Shares 4,2,3,1 ........... 62
4.5 Three Time Sensitive Tasks with Initial Shares 1,2,3 and initial weights 8,4,1 ........................................... 63
4.6 Three Time Sensitive Tasks with Initial Shares 2,2,2 and initial weights 3,1,8 ........................................... 64
4.7 Three Tasks with Initial Shares 3,5,0 and initial weights 0,2,3 ....... 65
4.8 Three Tasks with Initial Shares 2,0,0 and initial weights 1,4,6 ....... 66
4.9 Lag for Three PSS Tasks ........................................... 67
4.10 Lag for Three PSS Tasks ........................................... 68
4.11 Three Tasks with Initial Shares 0,0,0 and initial weights 4,2,1. Demonstrating Task Blocking ........................................... 70
4.12 Three Tasks with Initial Shares 5,3,1 and initial weights 1,1,1. Demonstrating Task Blocking ........................................... 71
4.13 Three Tasks with Initial Shares 0,0,0 and initial weights 6,4,2. Demonstrating Task Blocking ........................................... 72
4.14 Three Tasks with Initial Shares 7,4,3 and initial weights 0,0,0. Demonstrating Graceful Degradation ........................................... 73
1 Introduction

1.1 Proportional Share Scheduling

Modern operating system designs are moving towards having one general operating system that supports different types of applications at the same time. With different types of applications comes the challenge of different requirements imposed by these applications. In non-real time applications, the main concern is fast response time and efficiency where some deadlines could be possibly missed without severe consequences. On the other hand, meeting deadlines is the main concern for real-time applications. Integrating these two different types of applications is not trivial in the system design, and different scheduling algorithms have been proposed to solve that. The requirements to achieve this integration are fairness and flexibility on one hand and meeting strict deadlines on the other hand[1]. Proportional share resource allocation algorithms achieve flexibility and fairness whereas reservation-based algorithms guarantee meeting strict deadlines [1].

In Proportional Share Scheduling (PSS) the tasks or clients are assigned weights, and based on the this weight value and the weights of other tasks that exist in the system a share is assigned to the task. The share represents a “fraction of the resource’s capacity that is allocated to a client”[2]. So if a task has weight $w$ and the sum of weights of all the active tasks in the system is $W$, “then the share (fraction) of the CPU allocated to the client is $f = w/W$”[2]. Fairness is ensured in Proportional Share Scheduling (PSS) since each task will be allocated a share of a resource in accordance to its weight and other weights in the system [1]. Flexibility is ensured because tasks can join and leave the competition for a resource without any restrictions [1], and if a new task joins or leaves competition for a resource then the shares of the tasks will change to reflect that. PSS algorithm with its flexibility
in supporting dynamic events such as tasks joining or leaving, and its way of changing task shares whenever such events occur results in allocation uncertainties [1], and that makes supporting real-time tasks; which require a fixed percentage of a resource, difficult to achieve[1]. Reservation-based algorithms on the other hand can guarantee such fixed shares, and tasks will receive their fixed allocations, but that leads to limiting the flexibility of the system. Examples of reservation-based algorithms include rate monotonic or earliest deadline first [3] which are characterized by their strict restrictive policies. To allow proportional share resource algorithms to support real-time tasks, it must guarantee fixed shares to real-time tasks. This means that when tasks leave or join competition the weights in the system must change to reflect that, and assure that real-time tasks still get guaranteed shares. This problem of assigning and reassigning weights to tasks is referred to by the “weight-assignment problem” [2].

The Earliest Eligible Virtual Deadline First (EEVDF) algorithm is a proportional share algorithm first introduced by Stoica et.al [4] which successfully achieved integration between a wide range of real-time applications and non-real-time applications. EEVDF has the advantages of flexibility and fairness inherited from the characteristics of proportional share algorithm, but additionally supports a wide range of real-time applications because of its ability to allocate resources accurately and guarantees that the difference between the service time requested and service time received is bounded at all times [4]. EEVDF makes its scheduling decisions based on virtual time. Each task is associated with a virtual start time and a virtual deadline where the task with earliest virtual deadline is the most eligible task [4]. Virtual time rate changes based on the number of tasks in the system and EEVDF with the use of virtual time tries to emulate an ideal system. EEVDF and virtual time will be explored thoroughly in later chapters. Stoica et.al. [4] then extended
the EEVDF model and introduced the notion of duality between weights and shares to be used with the EEVDF algorithm in order to add reservation properties to the algorithm and solve the weight assignment problem. In this dual model the weights and shares are duals of each other and the model is complex to implement. Goddard et al. [2] proposed another model to solve the weight assignment problem where each task also has a weight and share but they are not duals. If a task is a real-time task then its share is fixed while its weight is variable to change whenever system dynamics occur. For non real-time tasks the share can vary but their weights are fixed. This model divides the tasks into two strict classes real-time tasks and regular tasks and the model treats the tasks differently. According to Goddard et al. [2] even though EEVDF supports real-time applications; in some instances when system dynamics occur EEVDF produces wrong results unless the virtual deadlines are recalculated. Therefore Goddard et al. [2] also introduced another method for calculating virtual deadlines when system dynamics occur.

1.2 Motivation

EEVDF presented a unified framework that supported real and non-real time applications. Additionally the dualism between weights and shares added to EEVDF by Stoica et al.[1] added flexibility to the system but with the expense of complexity in calculating task weights when system dynamics occur in which Goddard et.al. refer to as the weight assignment problem. Goddard et.al extended the EEVDF model by having two distinct classes of tasks where they have either fixed shares or fixed weights. This extension limits the types of applications that could exist in the system. In our model we propose a solution to the weight assignment problem by treating all the tasks the same in the system. In our model each task is assigned an initial share and initial weight and from that pair an effective weight is used for
making scheduling decisions. Only this effective weight value is used when making any scheduling decisions. Moreover, changes have been made to the way the virtual deadlines are calculated when system dynamics occur to ensure the system is still fair, balanced and still supports real-time tasks with strict deadlines. Regular tasks will have a zero value for its initial share and some value for its initial weight, thus each regular task will compete in accordance to its weight. An example will be a compiler task. With initial share values and zero initial weights, that represents a time sensitive task that requires a minimum guaranteed share. The zero value of the initial weight indicates that the task does not use excess idle resources. A guest operating system is an example. If there are values for both the initial shares and initial weights then that represents a task that requires a guaranteed share but can benefit from additional resources. A multimedia player is an example.

1.3 Contributions

This thesis work presents an implementation of a new model to support mixed mode applications in dynamic, open systems where regular tasks, time sensitive tasks and time sensitive tasks with added benefit can co-exist. Additionally proper handling of weight adjustment in the presence of system dynamics was supported. Another contribution was handling of client blocking where the strategy proposed by Stoica et al.

[4] was followed, but with applying the correct adjustment for virtual time and eligible time for a task that arrives with a lag not equal to zero. This new model was implemented as a loadable module in the Linux kernel where the implementation as a module has a number of advantages such as ease of debugging and ease of migrating the implementation to other Linux versions. In our model each task is associated with a pair of an initial share and initial weight in the form of \((\bar{f}_i, \bar{w}_i)\) where \(\bar{f}_i\) is the initial share and \(\bar{w}_i\) is the initial weight. From this pair an
effective weight is calculated. This effective weight is used in the system to calculate the resource share.

1.4 Organization

Chapter 2 will talk about the background related literature. Chapter 3 will describe the new model proposed and the actual implementation in the Linux kernel. Chapter 4 will evaluate the implementation and display and explain the results. Chapter 5 will summarize the results and talk about future work.
2 Background

2.1 Scheduling Overview

Scheduling is considered a crucial key component in the operating system design. Scheduling could be briefly defined as dividing the resources between processes and deciding which process is eligible to use the available resource. In any scheduler design three main goals should be considered: fairness, efficiency and response time [5, 6]. A process could be simply defined as any program in execution; in other words it is using a resource to do any kind of logical or mathematical operation. Along with the CPU the process requires other resources such as the memory or input/output devices. According to that, processes could be classified as batch, interactive or time-sensitive processes. Batch processes are those processes that only need to use the CPU to perform its operations and do not require any services from input/output devices. An example of a batch process is a program that has a long mathematical equation to be calculated. Interactive processes are those processes that require services from input/output devices. This implies that an interactive process will not be using the CPU all the time since it will be waiting for a service from the input/output device. As a result the process will be blocked until the service is ready which allows other processes to use the CPU during that time. A time-sensitive process requires a uniform CPU allocation where its operations should start and complete at specific times. Failure in meeting these deadlines will result in errors and undesired results. An example is executing multimedia applications, or on a larger scale chemical plant operations where very critical operations are included and deadlines must be strictly met [4, 6, 7].
2.2 Proportional Share Scheduling

Tijdeman was one of the first to study the proportional share allocation algorithm\cite{8}. Tijdeman used proportional share allocation to solve the problem of selecting state chairmen for each state based on the state’s weight\cite{8}. In the operating system world, the same main concept is applied. Each process is associated with a weight that determines its share or its percentage of a shared resource. In Proportional Share Scheduling (PSS) all the processes are treated the same and the scheduler assigns shared resources to the processes in discrete time quantum’s \cite{4}. A time quantum is defined as “the maximum time interval the client is allowed to run before another scheduling decision is made”\cite{9}. Furthermore, “In a proportional share system, processes make progress at a precise, uniform rate according to the share of system resources they are to receive” and each process appears to be running on a dedicated processor “whose capacity is a fraction of that of the actual processor” \cite{5}. According to Stoica et al. \cite{4} PSS aims to achieve the following goals:

1. Integrate both real-time and non-real-time processes in one general operating system.

2. Ensure fairness between all existing processes, since all the processes are treated equally. There is no difference between a real-time process and a non-real-time process.

3. Adding more flexibility to the operating system and providing better graceful degradation.
2.2.1 The Model

In this section the Proportional Share Scheduling (PSS) model used by Stoica et al. will be introduced [4][7]. In this model a number of processes, both real-time and non-real-time, run in a system where they are competing for a shared resource such as a CPU. Each process is called a client, and the client can be in one of two states: active which means the client is competing for the resource, or passive when it is not. The resource is always allocated in discrete time called a quantum and the client joins the competition by issuing a request that is associated with the service time for that request. Each client that enters the system is associated with a weight that determines the share of the resource the process is entitled to use. “If a process’s share of the processor is $s$ then in any interval of length $t$, the process is guaranteed to receive $(s \times t) \pm \epsilon$ units of processor time where $0 \leq \epsilon \leq \delta$, for some constant $\delta$”[4]. At the beginning of each quantum a process is chosen based on a scheduling policy to use the resource. The chosen process can use the resource for the whole length of the quantum or can release the resource before the end of the quantum. Let $w_i$ be the weight associated to process $i$ and let $C_a(t)$ be the set of all the active processes at time $t$ then the share $f_i(t)$ of a process $i$ at time $t$ is [4]:

$$f_i(t) = \frac{w_i}{\sum_{j \in C_a(t)} w_j} \quad (2.1)$$

The client is entitled to use the resource for $f_i(t)(t_2 - t_1)$ time units if the client’s share remains constant during that time interval. When the client’s share is not constant during a time interval $(t_1, t_2)$ then the service time it receives is [4]:

$$S_i(t_1, t_2) = \int_{t_1}^{t_2} f_i(\tau) \, d\tau \quad (2.2)$$

Since the client cannot always receive the exact amount of time it is entitled for due to quantization; that leads to what is called as a service time lag. The lag is the
difference between the service time the client should receive in an ideal system and the time it actually receives\[2, 4, 5\]. Let $t_i^0$ be the time when client $i$ becomes active, and let $s_i(t_i^0, t)$ be the service time the client receives during the interval $[t_i^0, t]$ and $S_i(t_i^0, t)$ be the ideal service time the client is supposed to receive during that interval, then the lag of the client $i$ at time $t$ is \[4\]:

$$\text{lag}_i(t) = S_i(t_i^0, t) - s_i(t_i^0, t) \quad (2.3)$$

When talking about the ideal fluid flow model the lag will refer to “how far “behind” or “ahead” the client is in its execution from where it should be”\[2\]. If the client’s lag is positive then the client is behind schedule in accordance to the ideal fluid-flow system. If the lag is negative then the client is ahead of schedule and has received more service time than it is supposed to in accordance to the ideal fluid-flow system\[5\]. Additionally, this lag value “quantifies the allocation accuracy” and will be used “as the main parameter in characterizing our proportional share algorithm”\[4\]. To make real-time execution possible in using proportional share algorithms this lag value should be bounded for all the active clients, and that is what the EEVDF algorithm (which is a proportional share algorithm) achieves. EEVDF will be explained in section 2.2.3.

### 2.2.2 The Concept of Virtual Time

Algorithms that implement fair share scheduling such as Proportional Share Scheduling (PSS) are based on an ideal fluid-flow system that runs in the virtual time domain, where each client receives at least its requested service time( share of the resource)\[4, 7, 10\]. Bavier et al. state that the “fluid model describes the ideal real-time behaviour of a fair sharing system” and virtual time plays the role of tying this theoretical model with the algorithm that implements it\[10\]. Furthermore,
virtual time is used to simplify the complex ideal fluid-flow model and act as “a bridge between theory and code”[10]. The virtual clock algorithm was first introduced by Zhang [11] as a traffic control algorithm for high-speed packet switching networks. The virtual clock principal is similar to Time Division multiplexing (TDM) where each flow has its reserved transmission rate. In virtual time, during each virtual time tick (unit) each active client with weight $w$ is serviced $w$ real time units. Virtual time is computed based on the number of active clients in the system and might be slower, faster or equal to real time. The rate of virtual time is dependent on the number of active clients in the system, such that the rate speeds up when the number of active clients decreases and slows down when the number of active clients in the system increases [4, 7]. Virtual time could be defined as a function of real time[4]:

$$V(t) = \int_{0}^{t} \frac{1}{\sum_{j \in C_{a}(\tau)} w_{j}} d\tau$$

(2.4)

The ideal service time that a client with share $f$ should receive during the interval $[t_1, t_2]$ in an ideal fluid-flow system is [4]:

$$S_i(t_1, t_2) = \int_{t_1}^{t_2} f_i(\tau) d\tau$$

(2.5)

The above equation in virtual time notation is [4]:

$$S_i(t_1, t_2) = (V(t_2) - V(t_1))w_i$$

(2.6)

Since all the scheduling decisions are made in the virtual time domain, each client has a virtual eligible time $V(e)$ (or $ve$) which represents the starting time for the client’s request and a virtual deadline $V(d)$ (or $vd$) which specifies the time the client should finish its request. Let $t_0^i$ be the time when client $i$ becomes active, time $t$ when the client issues a request, $w_i$ the client’s weight, $s_i(t_0^i, t)$ the service
time the client received during the last service, then the virtual eligible time for client \( i \) is:

\[
V(e) = V(t_0^i) + \frac{s_i(t_0^i, t)}{w_i}
\]  
(2.7)

The virtual deadline for request \( i \) is defined as:

\[
V(d) = V(t_0^i) + \frac{r}{w_i}
\]  
(2.8)

where \( r \) is the length of the request issued by the client.

### 2.2.3 The EEVDF Algorithm

Proportional share scheduling aims at making the lags of all the active clients in the system as minimum as possible to ensure fairness. To accomplish that, at the end of each time quantum and before choosing the next eligible client to use the next quantum, the lags of all the active clients are calculated. Those clients that have negative lags (received more service time than ideal) are considered not eligible for competition for the use of the next quantum. On the other hand, client’s with positive lags are considered eligible since they have not received their ideal service time and are added to the pool of eligible clients that can compete for the use of the next quantum. If multiple eligible clients exist, then the client with earliest virtual deadline is chosen to use the next quantum, and this is the basis for the Earliest Eligible Virtual Deadline First (EEVDF) algorithm.

The Earliest Eligible Virtual Deadline First (EEVDF) makes scheduling decisions based on virtual time, where each client has an associated virtual eligible time \( ve \) and a virtual deadline \( vd \). Based on that, EEVDF uses the following rule to make scheduling decisions: “A new quantum is allocated to the client which has the eligible request with the earliest virtual deadline” [4]. All the clients that have a virtual eligible time less than the current virtual time \( ve \leq V(t) \) at the time of making the scheduling decision are grouped into the eligible list of clients. From
among those eligible clients the client with the minimal virtual deadline is chosen by
the scheduler as the next process for execution. Let \( r^k \) be the length of the \( k^{th} \)
request issued by client \( i \) and let \( ve^k_i \) and \( vd^k_i \) represent the virtual eligible time and
the virtual deadline for client \( i \), then the virtual eligible time for the first request
\( ve^{(1)}_i \) issued by client \( i \) which becomes active at time \( t \) is equal to the current virtual
time[4]:

\[
ve^{(1)} = V(t)
\] (2.9)

The virtual deadline for the first request \( vd^{(1)}_i \) issued by client \( i \) which becomes
active at time \( t \) is:

\[
vd^{(1)} = V(t) + \frac{r^0_i}{w_i}
\] (2.10)

For each request that follows the first request the virtual eligible time and the
virtual deadline issued by client \( i \) would be computed as [4]:

\[
ve^{(k+1)} = ve^{(k)} + \frac{u^{(k)}}{w_i}
\] (2.11)

\[
vd^{(k)} = ve^{(k)} + \frac{u^{(k)}}{w_i}
\] (2.12)

where \( u^{(k)} \) is the service time that client \( i \) actually receives during its \( k^{th} \) request.

Below is a step by step detailed description of the EEVDF algorithm and the
operations the scheduler makes before switching to the next eligible process [12] :

1. Let \( C_a(t) \) be the set of clients active at the current scheduling time \( t \).

2. Let \( j \in C_a(t) \) be the client that had its \( k_j \)th request \( r^{k_j}_j \) serviced during the
current scheduling cycle and \( a^k_j \) be the service time that client \( j \) actually
received during its \( k^{th} \) request.
3. Update the virtual time \( V(t) \) by:

\[
V(t) \leftarrow V(t) + \frac{a_j^k}{\sum_{i \in C_a(t)} w_i}
\]  

(2.13)

4. Determine the virtual eligibility time of client \( j \)'s next request by:

\[
ve_j^{k_j+1} = vd_j^{k_j}
\]  

(2.14)

5. Compute the virtual deadline of client \( j \)'s next request by:

\[
vd_j^{k_j+1} = ve_j^{k_j+1} + \frac{a_j^k}{w_j}
\]  

(2.15)

6. Update \( k_j \) by \( k_j \leftarrow k_j + 1 \)

7. Compute the subset of clients eligible at \( V(t) \) that is clients with

\[
ve_i^{k_i} \leq V(t)
\]  

(2.16)

8. Pick and execute the client with the earliest virtual deadline from the subset of eligible clients from step (7).

Three theoretical results pertain to the EEVDF algorithm and which proves how EEVDF can handle real-time tasks:

1. Lemma[7]: At any moment of time \( t \), the sum of the lags of all the active clients is zero, i.e,

\[
\sum_{i \in A(t)} lag_i(t) = 0
\]  

(2.17)

2. Theorem[4]: Let \( r \) be the size of the current request issued by client \( k \) in an PS-system with quantum \( q \). Then the lag of the client \( k \) at any time \( t \) while the request is pending is bounded as follows

\[
- r < lag_k(t) < max(R_{max}, q),
\]  

(2.18)
where $R_{max}$ represents the maximum duration of any request issued by any client in the system. Moreover, these bounds are asymptotically tight.

3. Corollary[4]: If no request of client $k$ is larger than a time quantum, then at any time $t$ its lag is bounded as follows:

$$-q < lag_k(t) < q.$$ (2.19)

2.2.4 Dynamic Operations in the Proportional Scheduler Algorithm

Any scheduler design always considers all the states a process can be in and must be able to make decisions when processes change states such as becoming active, blocking, sleeping, terminating. Dynamic operations refer to clients that join the competition for a shared resource or clients that leave competition. Accounting for these operations is considered important in the proportional scheduler design, since these dynamic operations affect the fairness of the proportional system and at any time $t$ the sum of the lag of all the active clients should be zero. The lag was defined before as the difference between the service time the client should receive in an ideal fluid-flow system and the service time it actually receives. The EEVDF algorithm introduced by Stoica et al. [4] aims to be able to handle all these different dynamic operations and do the required operations to make sure that each client still receives its share of the resource. For instance, if a client leaves competition with a negative lag; meaning it received more service time than its share, then the extra time that the client received is time lost for other active clients. That violates fairness between clients. Additionally, if a client leaves early before using its whole quantum; meaning having a positive lag, then the extra time should be distributed between the remaining clients fairly according to their weights. To assure fairness, the virtual time must be updated accordingly whenever a client leaves or joins the
competition. The following operations and changes should be applied to the calculations of EEVDF scheduling whenever a client joins or leaves the competition for a shared resource [4]:

1. If a client joins or leaves with a lag of zero, then no changes have to be done.

2. If a client leaves competition with a negative lag (received more service time than it was allotted) then the client is removed from the eligible list, but the client’s departure is delayed until its lag is zero.

3. If a client leaves competition with a positive lag, then this means the client received less service time than it was supposed to. In this case the current virtual time is updated as:

$$V(t) \leftarrow V(t) + \frac{\text{lag}_j(t)}{\sum_{i \in C(t) \setminus \{j\}} w_i}$$  \hspace{1cm} (2.20)

where $\sum_{i \in C(t) \setminus \{j\}} w_i$ is the sum of all the active clients weights except for the client that leaves the competition.

4. If a client joins the competition with a negative lag, then the client is not eligible and must wait.

5. If a client joins the competition with a positive lag, then this means the client did not receive enough service time during its last run. Thus, the current virtual time is updated as:

$$V(t) \leftarrow V(t) - \frac{\text{lag}_j(t)}{\sum_{i \in C(t)} w_i}$$  \hspace{1cm} (2.21)

### 2.2.5 Extensions to PSS

Extensions have been proposed by Stoica et al. [13] and Goddard et al. [2] to the original proportional share scheduler algorithm to integrate real-time and
non-real-time processes in one general operating system that achieves fairness, flexibility and guarantees meeting strict deadlines. Extensions also have been made to the algorithm through this thesis work.

2.2.5.1 Extensions by Stoica et al.

To integrate real-time and non-real-time clients in one general operating system Stoica et al. [13] characterizes each client with a weight and a share that are duals of each other, and if one of them is fixed then the other one is fixed too. The weight \( w \) determines how much service time a client is asking for during a time interval, and the share \( f \) determines the fraction of the resource the client asks for. Furthermore, this pair \((w, f)\) defines the type of allocation the client asks for. To achieve proportional share allocation the weight \( w \) is fixed and to achieve resource reservation the share \( f \) is fixed. The share of a client is defined as [13]:

\[
f = \frac{w}{W} \tag{2.22}
\]

where \( W \) is the total weight of all the active clients. Also the client’s weight could be calculated as:

\[
w = \frac{W'f}{(1 - f)} \tag{2.23}
\]

where \( W' = W - w \). It is clear that the share of a client depends on the number of clients competing in the system. To maintain the fixed fraction of a resource in which the real-time client needs to meet its deadline its share \( f \) has to be fixed at all times. That could be possible in such a dynamic system where clients enter and leave the system by changing the weights of other processes that are not real-time to keep the shares of real-time processes fixed[13].
2.2.5.2 Extensions by Goddard et al.

The authors [2] proposed two extensions: First, assigning weights to the real-time clients using a different approach that assures that a real-time client always receives its fixed share of a resource. Second, re-computing the virtual deadlines for the existing clients whenever a client joins or leaves the competition to make sure the EEVDF algorithm is correct and no client will miss its deadline by more than one quantum size ($q$). In their approach, Goddard et al. [2] are using the same EEVDF algorithm used by Stoica et al.[13], but instead of using the dualism between weights and shares where for n real-time clients n equations need to be resolved, calculating $w$ is much simpler. To resolve the problem of assigning weights to clients Goddard et al. fixes the weight that a real-time client receives and makes it equal to its share $w_i = f_i$. After each real-time client receives its share, the share remaining of the resource is divided between the non-real-time clients in proportion to their weights. Let $F$ be the total utilization of the resource by the each real-time client $i$ with share $f$ then:

$$F = \sum_{i=1}^{n} f_i$$

(2.24)

The weights of the non-real-time clients would be computed as :

$$w_i = \frac{\bar{w}_i}{\sum_{k=n+1}^{n+m} \bar{w}_k} (1 - F)$$

(2.25)

where $\bar{w}_i$ is the original weight of the non-real-time client $i$. This equation results in allocating the unused share $(1 - F)$ to the non-real-time clients according to their weights. Whenever a client joins or leaves the competition, the weights of the real-time clients remain unchanged and only the weights of the non-real-time clients need to be recomputed. Regarding changing the virtual deadlines, whenever a client joins or leaves competition the weights of the non-real-time clients have to be
recomputed and then the virtual deadlines for the clients would be calculated as:

\[ v_{d_{i,k}} = V(t) + \frac{\text{lag}_i(t) + \bar{S}_i(t, d_{i,k})}{w_i} \]  

(2.26)

where \( \bar{S}_i(t_1, t_2) \) represents the service time client \( i \) would have received in a fluid-flow system if none of the weights have changed.

### 2.3 Proportional Share Scheduler Algorithms

Besides the EEVDF algorithm proposed by Stoica et al.\[4\] other proportional share allocation algorithms exist that follow different approaches to integrate real and non-real time applications in one system. Some of these algorithms will be explored in this section.

#### 2.3.1 Stride Scheduling

A proportional share resource algorithm introduced by Waldspurger et al.\[14\] that is based on “rate-based flow-control algorithms for networks”. Stride scheduling is characterized by low response time and supporting dynamic operations such as clients joining and leaving competition for a resource, and clients changing their shares. In this algorithm each client is associated with three parameters: tickets, stride and a pass in which a resource is granted in discrete time units where each unit is called a quantum. Making a scheduling decision is based on the value of the pass, where the client with the shortest pass is selected to use the resource.

Tickets are “abstract, first-class objects that can be issued in different amounts and passed between clients”\[14\]. Tickets serve two purposes: defining how much resource allocation a client is supposed to receive and defining the clients response time. The resource allocation is proportional to the number of tickets the client owns, whereas the response time for the client is inversely proportional to the number of tickets the client owns. In a system that has a total of \( T \) tickets and a client that owns \( t \)
tickets, then for \( n \) consecutive allocations the client’s allocation would be \( n \times \frac{t}{T} \).

A stride represents the time interval between two resource allocations for a client and equals the reciprocal of the client’s tickets. Passes are virtual time representation for strides [14].

### 2.3.2 The BERT Scheduler

BERT (Best Effort and RealTime) scheduler [15] is based on the virtual time clock to implement proportional share allocation between tasks. BERT integrates best effort tasks such as compilers and real-time tasks such as multimedia applications, and is characterized by a mechanism called “stealing” that is used to handle overload situations. Moreover, stealing is used to guarantee that real-time tasks meet their deadlines, so if a real-time task is expected to miss its deadline based on the number of cycles it is granted, then the scheduler will steal cycles from another client which is considered an unimportant task and grant those cycles to the real-time task to meet its deadline. The difference between the real-time deadline and the virtual deadline for the task determines how many cycles to steal. BERT has the ability to handle overload situations gracefully because it distinguishes between important and unimportant tasks and uses the stealing mechanism to make sure important tasks meet their deadlines [15].

### 2.3.3 The SMART Scheduler

The SMART(Scheduler for Multimedia And Real-Time applications) scheduling algorithm was introduced by Nieh et al.[16] to integrate real-time and conventional processes in one scheduler design. The scheduling decisions are based on importance and urgency. Urgency determines when to give the resource allocation, whereas importance specifies the allocation that a process is supposed to receive. All the processes that are considered important are grouped together and from that group a
process is chosen based on its urgency to execute. The importance of a process is specified by a value-tuple consisting of a priority and a Biased Virtual Finishing Time (BVFT) that measures how much did the process receive of its proportional share. Urgency is based on earliest deadline first and is associated with real-time processes only. This design allows real-time tasks to meet their deadlines in a system that has a mix of real-time and conventional processes.[16]

2.3.4 Virtual-Time Round-Robin (VTRR)

This scheduler algorithm proposed by Nieh et al. [9] is a proportional share scheduler that is characterized by its low scheduling overhead due to its ability of making scheduling decisions in O(1) time instead of O(N). Moreover, the implementation is very simple where according to Nieh et al. [9] the implementation on Linux required only 100 lines of code. VTRR uses Round Robin for scheduling the tasks where each task runs for one time quantum, and uses the concept of virtual time and virtual finish time for implementing the fair queuing algorithm. An advantage of VTRR over the original fair queuing algorithm is that the position of a client in the run queue does not change at each scheduling decision. Once a client is given a share, it is inserted in its proper position in the run queue (from largest to smallest share) and the positions only change when a client’s share changes.[9]

2.4 Linux Schedulers

2.4.1 The O(1) Scheduler

As mentioned before, the scheduler plays a vital role in any operating system design; in this section the Linux scheduler (the O(1) scheduler) will be explained. The O(1) scheduler was the scheduler implemented in Linux version 2.6 up to version 2.6.23. Since Linux is a multi tasking operating system then it must give the
impression to the user that all the processes are running at the same time even though only one process can run at a time. The scheduler has to choose between many processes competing for the processor to run, and makes this decision based on the priority of the process. Each process is assigned a priority and these priorities in Linux are dynamic which means they change based on the interactivity of the process. Interactive processes which have to yield most of the time waiting on events or input/output operations have a higher priority than batch processes which spend most of their time executing code on the CPU and don’t yield. On the other hand, real-time processes always have the highest priority and can always preempt any interactive or batch processes and use the CPU immediately. The scheduling algorithm implemented in the 2.6 version of Linux was a major improvement in the Linux design. In this version the scheduler is called the O(1) scheduler in which the time to find an eligible process (the process with highest priority) to run is always constant regardless of the number of processes being searched [17][18][6]. This speeded up the search for the eligible process and reduced the overhead caused by the scheduler where it had to go through all the existing processes linearly and find the eligible process to run. This O(1) scheduler is based on a number of new data structures such as the runqueue data structure which is a per-CPU data structure that stores all the processes that are in the running state. The runqueue has two arrays: the active array and the passive array. Whenever a new process is created it is inserted into the active array and waits its turn to be executed based on the priority assigned to it; once the process is chosen for execution it runs for a length of 1 time slice (unless it is preempted by another higher priority process) and then is inserted into the passive queue. One note about this scheduler is that interactive processes are inserted back at the end of the active array instead of the passive array because it has higher priority than normal processes and they are more likely
to run later. After all the processes in the active queue run, the active and passive queues are exchanged and the expired processes are provided a chance to run again. Having the interactive processes inserted at the end of the active queue instead of the passive queue does not mean that the active queue will run infinitely or to allow other processes in the passive queue to starve since there is a limit on the maximum time that processes in the passive array will wait. Another important data structure is the process descriptor which contains all the information relevant to the process created. This structure has a number of fields which are important for the scheduler to operate such as priority, static priority, average sleep time, and time slice. Most of the code related to the functionality of the scheduler is located in two files sched.c and sched.h; here the most important functions that are used by the scheduler most frequently will be mentioned. The scheduler_tick() function is the function that is called at each time tick. It checks the status of the currently running process to examine if it needs rescheduling or not. The try_to_wake_up() function awakes a sleeping process that was waiting on an event or an input/output device. This function reinserts the process into the active array and makes it available and ready to run again. The schedule() function which is the most important function is called whenever a new process is to be chosen for execution; it selects the next process to be executed and performs the complete operation of switching between the old running process and the newly chosen process. This includes switching the memory regions for the two processes and the pointer to the newly chosen process.

### 2.4.2 The Completely Fair Scheduler

The Completely Fair Scheduler (CFS) was introduced by Ingo Molnar in the Linux Kernel version 2.6.23. Molnar describes the CFS as "CFS basically models an
'ideal, precise multitasking CPU’ on real hardware” [19]. The idea is to maintain balance (fairness) in providing processor time to tasks[20]. The ideal scenario is to have the ability where a processor can run a number of tasks in parallel where the processor divides its power according to task priorities [20]. So the scheduler tries to emulate ideal multitasking hardware as much as possible. Of course a processor can only execute a single task at a time, which led to the introduction of virtual time in CFS. Virtual time is used to determine the eligibility time for a task to run on the processor assuming an ideal situation where there are multiple processors running in parallel and serving all the processes at the same time [20]. To assure fairness the scheduler when making a scheduling decision, chooses the process which has waited the most or the process with the ”gravest need” [21]. The CFS generally consists of the following components:

1. sched_entity objects which are used to represent and store the tasks in a red-black tree structure.

2. A system-wide runqueue fair_clock variable[19]. The clock’s rate is dependent on the number of tasks existing in the system, and increases at rate equal to \((1/\text{Number of tasks})\) of the actual real time.

3. A per-process variable wait_runtime [19] which reports how long the process was waiting while the CPU was executing another process. This variable gets incremented by a value that depends on the number of active tasks in the system and is virtual time.

4. The red-black tree structure which is used to store the tasks. The red-black tree is a ”type of self-balancing binary search tree”[19] and has \(O(\log N)\) time complexity for retrieving, inserting and deleting elements from it.
Based on the wait_runtime variable the tasks are inserted and ordered in the red black tree. The task with the highest wait_runtime (which means it has waited the most) would be inserted to the left most of the tree, and the scheduler will choose that task as the eligible task. Compared to the O(1) scheduler, CFS is much simpler and does not require the complex heuristics [19] used to classify the tasks into interactive and non-interactive by measuring how long each task sleeps. Also CFS does not use runqueues that store the tasks, instead maintains a time-ordered red-black tree” [20].
3 Methodology

Our model aims to support the following desired requirements:

1. Support for mixed-mode applications: this means that the model should support serving different types of applications and should insure that the applications requirements are met. The different classes of applications include:
   
   (a) Regular tasks which are not time sensitive.
   
   (b) Time-sensitive applications that require exact guaranteed resource shares.
   
   (c) Time sensitive applications that require guaranteed resource shares, but also can benefit form idle resources if available.
   
   (d) A mix of all the applications mentioned above.

2. Guaranteeing the theoretical results from the original PSS model proposed by Stoica et al. such as theoretical lag bounds.

3. Provide support for handling system dynamics such as clients arriving and joining the competition, and clients leaving the competition.

4. Provide support for handling client blocking such as clients sleeping and clients blocking on I/O operations.

5. The ability to handle overload and underload situations in the system efficiently. The system should provide graceful degradation in overload situations. On the other hand, the system should utilize the resources efficiently in underload situations and be able to distribute the idle resources effectively between the clients.
The remaining of this chapter will describe how the new model will handle and support the above mentioned requirements. In chapter 4 the results will demonstrate how these requirements have been met.

3.1 The Model

In this model each client $i \in C$ has an initial guaranteed share $\bar{f}_i \in [0, 1]$, and an initial weight $\bar{w}_i[12]$. Based on the initial share and initial weight values the different clients are classified into different categories. Below are the different categories:[12]

1. $\bar{f}_i = 0$ and $\bar{w}_i > 0$: The client is a regular client and does not ask for any guaranteed share. This client will compete with other clients based on its weight. An example is a compiler task.

2. $\bar{f}_i > 0$ and $\bar{w}_i = 0$: The client is a time sensitive client because it is asking for a guaranteed share. Since the weight is zero, then that means the client is not asking to make use of any idle resources in the system if they exist. An example would be a guest operating system running as a user-level task.

3. $\bar{f}_i > 0$ and $\bar{w}_i > 0$: The client is a time sensitive application that requires a guaranteed share, but also can make use of any idle resources. The client competes for these idle resources based on its initial weight. An example would be a multimedia player task.

Based on the initial share and initial weight values an effective share and an effective weight is calculated for each client. In our model, the EEVDF algorithm uses the effective weight for making all the subsequent operations and making scheduling decisions. The effective share represents the share of the client in addition of its share of idle resources. In this model the system can be in two states. An overload
state when the sum of the initial guaranteed shares of all the clients exceeds 100% \((\sum_{j \in C_a(t)} \bar{f}_j \geq 1)\), and an underload state when the sum of the initial guaranteed shares is less than 100% \((\sum_{j \in C_a(t)} \bar{f}_j < 1)\). In an overload state, only the clients that requested an initial guaranteed share \((f_i \neq 0)\) will be serviced. In an underload situation, all the clients that requested an initial guaranteed share \((f_i \neq 0)\) will be serviced first, and then the available ideal resources would be distributed to other clients based on their weights[12]. Below is a detailed description of both states:[12]

1. Overload State: This state occurs when \(\sum_{j \in C_a(t)} \bar{f}_j \geq 1\), then the effective weight is \(w_i = \bar{f}_i\). The priority is for clients with initial guaranteed shares which require fixed shares to carry out their operations.

2. Underload State: This state occurs when \(\sum_{j \in C_a(t)} \bar{f}_j < 1\). Each client receives its initial share in addition to its share from the idle resources based on its initial weight \(w_i\). The effective share would be:[12]

\[
f_i = \bar{f}_i + \left( \frac{\bar{w}_i}{\sum_{j \in C_a(t)} \bar{w}_j} \right) \cdot (1 - \sum_{j \in C_a(t)} \bar{f}_j) \quad (3.1)
\]

Since

\[
f_i = \frac{w_i}{\sum_{j \in C_a(t)} w_j} \quad (3.2)
\]

We obtain:

\[
w_i = \sum_{j \in C_a(t)} w_j \cdot \left( \frac{\bar{w}_i}{\sum_{j \in C_a(t)} \bar{w}_j} \right) \cdot (1 - \sum_{j \in C_a(t)} \bar{f}_j) \quad (3.3)
\]

Finally since:

\[
\frac{\sum_{j \in C_a(t)} \bar{w}_j}{\sum_{j \in C_a(t)} w_j} = 1 - \sum_{j \in C_a(t)} \bar{f}_j \quad (3.4)
\]

We conclude:

\[
\sum_{j \in C_a(t)} w_j = \frac{\sum_{j \in C_a(t)} \bar{w}_j}{1 - \sum_{j \in C_a(t)} \bar{f}_j} \quad (3.5)
\]
Overall:

$$w_i = \frac{\sum_{j \in C_a(t)} \bar{w}_j}{1 - \sum_{j \in C_a(t)} \bar{f}_j} \cdot \left( \bar{f}_i + \frac{\bar{w}_i}{\sum_{j \in C_a(t)} \bar{w}_j} \cdot \left( 1 - \sum_{j \in C_a(t)} \bar{f}_j \right) \right) \quad (3.6)$$

The effective weight would finally be simplified to :[12]

$$w_i = \frac{\bar{f}_i \cdot \sum_{j \in C_a(t)} \bar{w}_j}{1 - \sum_{j \in C_a(t)} \bar{f}_j} + \bar{w}_i \quad (3.7)$$

Our model extends the model presented by Goddard et al. [2] and specifically targets the weight assignment problem. In our model each task is assigned an initial share and initial weight and based on those two values the tasks created fall into one of three classes above. Dynamic operations like clients leaving, joining competition and clients blocking require recalculating the weights for the clients. From equation 3.1 it is noticed that when the initial share is zero (not a time-sensitive task) then there is no need to recalculate the equations when dynamic events occur since the equation simplifies to $w_i = \bar{w}_i$. For time sensitive tasks that have initial share values the effective weight has to be recalculated as in equation 3.7. As a result this proposed model is optimal for systems when the number of regular clients is larger than time-sensitive clients. To demonstrate the idea of initial shares and initial weights table 3.1 shows a set of 5 active clients competing for a resource. $\bar{f}_i$ and $\bar{w}_i$ represent the initial share and initial weight respectively. $f_i$ and $w_i$ represent the calculated effective share and effective weight respectively. These effective values are calculated using equations 3.1 and 3.7. In this example the total amount of requested guaranteed share is $\sum_{i=1}^{5} f_i = 0.3$. The percentage of idle resource available is $1 - \sum_{i=1}^{5} f_i = 0.7$ which represents an underload situation. Clients 3, 4, 5 represent regular tasks that do not ask for guaranteed shares. Clients 1, 2 are time sensitive tasks that require a guaranteed share(10% and 20% respectively) but also can benefit from available idle resources.
Table 3.1: Effective Weight Calculation

<table>
<thead>
<tr>
<th>Client</th>
<th>( \bar{f}_i )</th>
<th>( \bar{w}_i )</th>
<th>( f_i )</th>
<th>( w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>3</td>
<td>0.205</td>
<td>5.857</td>
</tr>
<tr>
<td>2</td>
<td>0.2</td>
<td>1</td>
<td>0.235</td>
<td>6.714</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4</td>
<td>0.14</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>5</td>
<td>0.175</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>7</td>
<td>0.245</td>
<td>7</td>
</tr>
</tbody>
</table>

3.2 Handling Lag

The initial share and initial weight model provides flexibility and allows easy support and integration of regular and time sensitive tasks, and makes sure the resources is fully and efficiently utilized. To make the model handle dynamic events such as clients blocking, sleeping and leaving and joining the competition for a resource; the lag of the client plays an important role in balancing the system and assuring fairness between the processes. Lag was defined as the difference between the service time the client actually receives and the service time it is supposed to receive in an ideal fluid-flow system. Calculating the lag is necessary whenever a scheduling decision is made and is used to balance the system when dynamic events occur in the system. In chapter two we showed how the PSS deals with clients that leave or join the competition with lag values other than zero.[4] In this model a simplified definition of the lag was introduced where the lag is[12]:

\[
lag_i(t) = (V(t) - ve_i).w_i - s_i(e_i, t)
\] (3.8)
If the lag value is calculated at times when the scheduler is called then the previous definition of the lag could be simplified more to [12]:

\[ \text{lag}_i(t) = (V(t) - ve^k_i)w_i \]  \hspace{1cm} (3.9)

This formula is much simpler than the one proposed by Stoica et al. [4] and also makes the implementation much simpler.

To handle system dynamics special attention should be paid to the lag value of the client joining or leaving the competition. If a client joins the competition with a positive or negative lag (meaning the client received less or more service time than supposed to in an ideal fluid flow system) then system virtual time is updated as follows:

\[ \bar{V}(t) \leftarrow V(t) - \frac{\text{lag}_i(t)}{\sum_{j \in C_a(t)} w_j} \]  \hspace{1cm} (3.10)

Based on that the virtual eligible time for first request for the returning client \( ve^0_i \) is:

\[ ve^0_i = \bar{V}(t) - \frac{\text{lag}_i(t)}{w_i} \]  \hspace{1cm} (3.11)

Using this new virtual eligible time after updating the virtual time guarantees that the sum of lags in the system at any time is still zero, which is in accordance with the theoretical assumptions.

### 3.3 Implementation

The extended EEVDF Proportional Share Scheduler (PSS) was implemented as a loadable module in Linux kernel 2.6.20. Minimal changes has been made to the kernel and all the scheduling events where handled through function pointers (also called function hooks or handlers) in the module. Implementing the PSS as a module rather than having several modifications to the kernel source code will allow the PSS scheduler to be migrated and implemented on other Linux kernels in a
faster easier way. Moreover, updating and maintaining the scheduler would be easier. If the PSS module is loaded, then every time the regular Linux scheduler makes a scheduling decision it will call the PSS scheduler through the function pointer pss_getnext() which will check if there are any existing PSS tasks, and from among those PSS tasks chooses the most eligible task.

Once the PSS module is inserted into the kernel a task can run as a PSS task by invoking the function sched_setscheduler() which sets the scheduling policy and other scheduling parameters for a process. In Linux kernel 2.6.20 four scheduling policies exist which are SCHED_FIFO, SCHED_RR, SCHED_NORMAL, and SCHED_BATCH. SCHED_PSS was added as another scheduling policy to the kernel by defining it in the sched.h file, where it has higher priority than SCHED_NORMAL and SCHED_BATCH and lower priority than SCHED_FIFO and SCHED_RR. By calling sched_setscheduler() with scheduling policy SCHED_PSS, and using the sched_param parameter value as the initial share and initial weight of the process a PSS task is created, and a virtual eligible time and a virtual deadline will be calculated for its request. The PSS task will then be added to the pool of tasks competing for CPU time. A PSS task can also be created if a PSS task forks other tasks.

As mentioned before, a number of function pointers were inserted into the kernel that are called when specific scheduling events occur assuming the PSS module is loaded. Initially all the function pointers are set to NULL and once the PSS module is loaded it assigns those pointers to their handlers which reside in the PSS module. In the following sections the implementation would be thoroughly explored.
3.3.1 Changes Made to the Linux Kernel

The changes that occurred in the kernel were only in two files in the Linux 2.6.20 kernel; these two files are: sched.h and sched.c. In sched.h the new PSS scheduling policy (SCHED_PSS) was defined and two new data structures were added to define the PSS task and the PSS request. In sched.c a number of function pointers were added to handle the different scheduling events. These function pointers call their respective functions that are implemented in the PSS module.

3.3.2 Main Data Structures

The two data structures implemented in sched.h are:

1. struct pss_Task: This structure is created for each PSS task. Each PSS task has two pointers one to the main task structure that encapsulates the PSS task and the second pointer is to the PSS request that is created by this PSS task. Each PSS task can request one PSS request at a time.
Listing 3.1: PSS Task Structure

```c
struct pss_Task{
    /* the process ID */
    int pid;

    /* the initial weight as requested by the process */
    int init_weight;

    /* the effective weight as calculated by the module */
    int eff_weight;

    /* the initial share as requested by the process */
    int init_share;

    /* the effective share as calculated by the module */
    int eff_share;

    /* the task lag calculated at each scheduling time */
    int lag;

    /* request length */
    int ideal_servtime;

    /* how much service time already received */
    int servtime;

    /* the task structure owner of this pss task */
    struct task_struct *owner;

    /* the structure pointing to the pss request issued by the pss task */
    struct pss_Request *pss_request;

    /* list head */
    struct list_head pss_task_list;
};
```
2. **struct pss_Request:** The second structure is the PSS request structure. This structure is created for each request issued by a PSS task. It includes a pointer to the PSS task that issued the request. The virtual eligible time and the virtual end time for each request is calculated when the PSS request is created.
3. struct requestlist: This structure exists in the PSS module and is used to create two doubly linked lists. One that holds all the requests from all the PSS tasks that are competing for a resource. The other list holds all the requests that are considered eligible, where a request is considered eligible if its virtual eligible time is less than the current virtual time.
3.3.3 Function Pointers

In this section the function pointers (handlers) implemented in the PSS module will be explored in addition to indicating where in the sched.c file each pointer is called. Following the function pointers other important subroutines implemented in the PSS module would be explored. These pointers are initially set to null until the PSS module is loaded, and are located in the sched.c file. Each time the handler is called it will call its respective handler in the module.

1. PSS_isloaded: This handler checks whether the PSS module is loaded into the kernel or not, and is the first one to be checked before proceeding into any other function handlers. If a task has changed its scheduling policy to SCHED_PSS but the PSS module is not loaded then a message will be returned to the user indicating that the module is not loaded and sets the process to SCHED_NORM policy.

2. PSS_setscheduler: This handler is located in the sched_setscheduler(..) function in sched.c. It is called when the user wishes to change the scheduling policy of the process to SCHED_PSS. The user also inputs the initial share and the initial weight for the PSS task using the priority field of the sched_param structure. This handler starts by changing the policy to SCHED_PSS and assigning the initial share and weight for the task. Next it calls create_psstask(..) to initialize and create the PSS task. After creation, the add_to_competition(..) will be called to add the task to competition. These two functions will explained in more detail in section 3.3.4.

3. PSS_getscheduler: Returns the current scheduling policy of the calling process.
4. PSS_fork: This handler is called when a new process is forked in which the parent’s scheduling policy was SCHED_PSS. PSS_fork is similar to the regular `sched_fork(..)` in sched.c where its main purpose is to initialize the task and make it ready to be added to competition. It initially calls `create_psstask(..)` then calls PSS_activate to add the task to competition.

5. PSS_terminate: Terminates the task and removes the task from competition. It also frees the memory allocated for the PSS task.

6. PSS_deactivate: Removes the task from competition. This occurs when the task sleeps or blocks waiting on another event to occur. This handler calls `leave_competition(..)` which updates the virtual time and updates the total shares and weights in the system then removes the task from competition. Once the process wakes up it will issue a new request and will be added to competition again.

7. PSS_activate: Adds the PSS task to the competition by calling the function `add_tocompetition(..)`. This handler is usually called from `wake_up_new_task(..)` in sched.c after the process has been initialized.

8. PSS_getnext: This handler by far is the most important handler. It is called from the function `schedule(..)` in sched.c at every tick if the PSS module is loaded. It is responsible for selecting the next eligible PSS task based on earliest virtual deadline, and updates the virtual time by calling `update_vtime(..)`. This handler will be explained in more detail in section 3.3.5.

Other subroutines implemented in the PSS module are:
1. `update_vtime(..)`: Updates the virtual time at every tick when the module is loaded.

2. `effective_weight(..)`: Calculates the effective share and effective weight for the PSS task. Based on whether the system is overloaded (total share exceeds 1) or not (total share less than 1) the effective share and effective weight for the task will be calculated. It also updates the total share and total weight in the system.

3. `update_virtualtime_joining(..)`: When a new task joins the competition for an available resource then the virtual time has to be updated to reflect that, and to keep the system consistent. The virtual time would be updated according to the value of the lag. So if the lag is zero (the task is not ahead or behind real-time) then the virtual time updates normally, otherwise the virtual time has to be updated as explained chapter 4.

4. `find_ve(..) and find_vd(..)`: Called to calculate the virtual eligible time and virtual deadline for the request.

5. `init_pss(..)`: Used to initialize all the objects for the PSS scheduler such as the PSS request list and the PSS eligible list. It also initializes the virtual time.

6. `update_virtualtime_leave(..)`: Updates the virtual time whenever a PSS task leaves competition. The way the virtual time is updated is dependent on the leaving task’s lag.

7. `leave_competition(..)`: Updates the virtual time by calling `update_virtualtime_leave(..)`, and updates the total weights and shares in the system. The task is then removed and memory freed.
8. `create_pssRequest(..)`: Called to create and initialize a PSS request for the PSS task. One request is created per task. The virtual eligible time and virtual deadline is calculated then the request is added to the list of PSS requests.

### 3.3.4 Task Creation

The functions responsible for creating a new PSS task and inserting any requests for competition are `create_psstask(..)` and `add_tocompetition(..)`. `create_psstask(..)` starts by allocating memory for the PSS task, then performs a number of initialization steps for the task and its request to prepare it for insertion into competition. It assigns the initial share and initial weight as requested and the virtual eligible time and the virtual end times would be initialized to zero. The next function called is `add_tocompetition(..)` which will perform the following steps:

1. Updates the total share and total weight values in the system.
2. Calls the function `effective_weight(..)` to calculate the effective share and effective weight as explained in chapter 4. The effective weight value is the weight that will be used for making any scheduling decisions.
3. Creates a request for the task and calculates the virtual eligible time and virtual deadline for the request.
4. Inserts the task’s request to the list of requests competing for the available resource.

### 3.3.5 Task Scheduling

`PSS_getnext(..)` is the function responsible for fetching the next eligible task according to the algorithm, then passing the chosen task back to the `schedule()` function in sched.c where the task switch occurs and the newly chosen task will be
granted the use of the CPU. Below are the steps that are performed by PSS _getnext(..):

1. Update the virtual time for the system by calling `update_vtime(..)`.

2. Finding the next eligible PSS request by looking into the list of the PSS requests issued by all the PSS active tasks and choosing those requests which have a virtual eligible time less than the current virtual time. From that list of eligible requests choose the request with the minimum virtual deadline and return the task that owns that request. This would be the next task to use the resource.

3. Update the number of time quantums used by the chosen task. If the task has reached its request length (its request length) then the task will issue a new request where a new virtual eligible time and a new virtual deadline will be calculated. The new request will be added to the list of competing PSS requests.

4. Calculate the lags for all the active PSS tasks in the system. The lag for any task should always be bounded and not exceed the task’s request length.

5. Return the task to the schedule() function in sched.c where the actual task switch occurs.

As explained through this chapter all the scheduling events were handled through event handlers located in the module. This will help very much migrate the PSS scheduler to newer Linux kernel versions. Moreover, this helped very much in the development and debugging phases during the implementation. Below are a set of pseudo codes which demonstrate the underlying implementation.
Listing 3.4: Function Pointers

/* The function pointers(hooks) which are used to call functions from the PSS module */

int (*pss_isloaded)(void) = NULL;
EXPORT_SYMBOL(pss_isloaded);

int (*pss_setscheduler)(struct task_struct *, int, int) = NULL;
EXPORT_SYMBOL(pss_setscheduler);

int (*pss_getscheduler)(pid_t pid) = NULL;
EXPORT_SYMBOL(pss_getscheduler);

int (*pss_fork)(struct task_struct *) = NULL;
EXPORT_SYMBOL(pss_fork);

int (*pss_terminate)(struct task_struct *) = NULL;
EXPORT_SYMBOL(pss_terminate);

int (*pss_deactivate)(struct task_struct *) = NULL;
EXPORT_SYMBOL(pss_deactivate);

int (*pss_activate)(struct task_struct *) = NULL;
EXPORT_SYMBOL(pss_activate);

struct task_struct *(*pss_getnext)(void) = NULL;
EXPORT_SYMBOL(pss_getnext);
Listing 3.5: An example of a function Pointer Implementation in sched.c

1 /* ___activate_task — move a task to the runqueue */
2 static void ___activate_task(struct task_struct *p, struct rq *rq)
3 {
4     /* check if the task’s policy is PSS */
5     if (p->policy == SCHED_PSS){
6         /* check if PSS module is loaded */
7         if (pss_isloaded() == 1){
8             /* call the function pointer */
9             pss_activate(p);
10         }else{
11             /* pss module not loaded */
12             return failure;
13         }
14     }
15     return success;
16 }else{
17     /* continue as usual */
18     ...
19     ...
20 }
Listing 3.6: An example of a function handler implemented in the PSS module

```c
/* unblock a task/activate new task */

int PSS_activate(struct task_struct *p) {
    /* check if memory allocated for task */
    if (p->pss_task == NULL) {
        return failure;
    }

    /* finding total initial weight and initial share in PSS system */
    find_total_weight_init();
    find_total_share_init();

    /* finding effective weight for task p */
    effective_weight(p->pss_task);

    /* update virtual time */
    update_virtualtime_joining(p);

    /* inserting the new pss task */
    struct list_head *task_entry = &p->pss_task->pss_task_list;
    list_add(task_entry, &task_list->list);
    task_list->nr_tasks++;

    /* create a new request for the PSS task */
    create_request(p);

    /* find virtual eligible time and deadline for new request */
    find_virtual_eligible_time();
    find_virtual_deadline();

    /* insert the request */
    list_add(&p->pss_task->pss_request->pss_request_entry, &pss_Request_list->list);
    pss_Request_list->nr_requests++;

    return success;
}
```
Listing 3.7: Calling the PSS Module Scheduler from schedule() in sched.c

```c
struct task_struct *next;
struct task_struct *pss_next;

/* check if it is a real time task. Real time tasks have higher priority than PSS tasks */
if (!rt_task(next)){
    /* check if the PSS module is loaded or not */
    if (pss_isloaded != NULL){
        /* call the PSS scheduler to choose eligible task */
        if( pss_getnext != NULL){
            pss_next=pss_getnext();
            if(pss_next !=NULL){
                next=pss_next;
                /* perform task switching to new chosen task and grant the newly chosen task */
                goto switch_tasks;
            }
        }
    }else{
        /* continue normal schedule() function */
    
        ...
    ...
    }
}
```

Listing 3.8: The EVDV Algorithm

```c
struct task_struct *PSS_getnext(void) {
    /* check if pss is initialized */
    if (pss_init==FALSE) init_pss();
    /* if no requests available return to regular schedule() and let it choose another task */
    if (pss_Request_list->nr_requests == 1 || pss_Request_list->nr_requests == 0) {
        return NULL;
    }
    /* Timestamp */
    get_time();
    /* update system virtual time */
    void update_virtual_time();
    /* Choose next eligible request */
    struct pss_Request *task=NULL;
    list_for_each(ptr, &pss_Request_list->list) {
        task = list_entry(ptr, struct pss_Request, pss_request_entry);
        /* Is task eligible */
        if (task->find_virtual_eligible_time <= pss_time.system_virtual_time) {
            /* store task virtual deadline */
            min_virtual_deadline = task->vd;
        }
    }
    /* locate task with lowest virtual deadline */
    find_minimum_virtual_deadline();
};
```
Listing 3.9: EEVDF Algorithm Continued

1 /* check if task used all its time slices or not */
2 if(next_task->pss_task->servtime < next_task->pss_task->owner->time_slice){
3    /* update the request count */
4       update_request_count();
5    /* issue a new request immediately if reached max service time */
6    if(next_task->pss_task->servtime == next_task->pss_task->owner->time_slice){
7       /* remove previous request */
8       delete_request(next_req);
9       find_virtual_eligible_time();
10      find_virtual_deadline();
11      create_new_request();
12      /* insert the new request to list again */
13      insert_request();
14      /* compute lag */
15      compute_lag();
16      /* return task to schedule() for task switching */
17      return next_task;
18   }
19  }

4 Experiments and Results

In this chapter the results will be presented and explained. As mentioned in earlier chapters clients are classified into three types: regular, time-sensitive, time-sensitive with added benefit. The following figures will demonstrate how the proportional share scheduling algorithm will guarantee that each client receives its share.

4.1 Experimental Setup

An Intel Pentium 4 CPU with 2.80GHz and system memory of 1GB was used for implementing the PSS algorithm and for testing. The kernel version was Linux-2.6.20. The PSS algorithm was implemented as a loadable module and minimum modifications were applied to the kernel. The quantum size (tick size) was not modified from the original Linux-2.6.20 kernel on the i386 architecture which had a frequency of 1000HZ which makes the quantum size equal to 1 millisecond. This means the scheduler was called every 1 millisecond to check the available active tasks and make a scheduling decision to choose the next eligible task to run.

4.2 Types of Experiments

Two types of experiments were performed. One to measure the number of iterations (dummy while loop iterations) performed per second to check how much share does each task receive. This test program launches a real-time task that has higher priority than a PSS task. A number of PSS tasks will be launched and the real-time task sleeps and waits for the PSS tasks to finish execution. Each PSS task will run and the number of iterations executed per second would be reported [22]. The second type of experiments was to check the lag values for each PSS task. That was performed by calculating the lag value for each task whenever the scheduler is
called. The lag values were input to a log file that was read later to check the lag values. Charts were then used to display the lag values.

4.3 Results and Interpretation

In this section the results will be displayed and an explanation will follow each figure.

4.3.1 Regular tasks

Regular tasks are tasks with initial weights only and do not require a guaranteed share. Each client (task) will be competing for the available resource according to its weight. The initial weight and initial share for a task is converted to an effective weight that is used when making scheduling decisions.
In figure 4.1 three regular tasks with initial weights 3,2,1 respectively are competing. According to the algorithm task 12365 with weight 3 will receive a share of 3/6=50% and task 12366 with weight 2 will receive a share of 2/6=33% and task 12367 with weight 1 will receive a share of 1/6=16%. At time = 10.5 seconds when the three tasks are active, the total number of iterations performed by the three tasks is 10711, where task 12365 performed 5366 iterations which maps to 5366/10711=50% share. Task 12366 performed 3561 iterations which maps to 3561/10711=33.2% share. Task 12367 performed 1784 iterations which maps to 1784/10711=16.6% share.
In figure 4.2 five clients are competing with initial weights 8,3,2,6,4 respectively which corresponds to shares of 34%, 13%, 8%, 26%, 17%. According to the algorithm task 12968 with initial weight 8 will receive a share of $8/23=34\%$ and task 12969 with weight 3 will receive a share of $3/23=13\%$ and task 12970 with weight 2 will receive a share of $2/23=8.7\%$. Task 12971 will receive a share of $6/23=26\%$ and task 12972 will receive a share of $4/23=17\%$. At time $= 20.3$ seconds when the five tasks are active, the total number of iterations performed by the five tasks is 20648, where task 12968 performed 7166 iterations which maps to $7166/20648=34.7\%$ share. Task 12969 performed 2702 iterations which maps to $2702/20648=13\%$ share. Task 12970 performed 1806 iterations which maps to $1806/20648=8.7\%$ share. Task 12971 performed 5385 iterations which maps to
5385/20648=26% share. Task 12972 performed 3589 iterations which maps to 3589/20648=17.3% share. At time=40 seconds when three tasks are still competing, the weights of the three tasks 12972,12969,12970 will have shares of 33.3%, 22.2% and 44.4% respectively. When there are three clients in the system less competition leads the three remaining clients to receive more share.
4.3.2 Time Sensitive Tasks

Time sensitive tasks are tasks that require a fixed guaranteed share and does not make use of any idle resources after receiving its share. Figure 4.3 displays the behaviour of three tasks with initial shares only. In this case each task receives its guaranteed share but does not make use of any idle resources in the system. According to the algorithm the tasks will receive shares equal to 20%, 60% and 10% respectively, and they will be guaranteed these shares even in the existence of other regular tasks with initial weight values only. At time = 10 seconds the total number of iterations is 10711. Task 13982 with initial share 2 performed 2383 iterations which maps to 2383/10711 = 22% share. Task 13983 with initial share 6
performed 7159 iterations which maps to $7159/10711=66\%$ share. Task 13984 with initial share 1 performed 1169 iterations which maps to $1169/10711=10.9\%$ share.

Figure 4.4: Four Time Sensitive Tasks with Initial Shares 4,2,3,1

Figure 4.4 displays four time sensitive tasks with initial shares of 4,2,3,1 that correspond to shares of 40%, 20%, 30%, 10% respectively. In the two cases above since there is no initial weight requested then each task will not benefit form any idle resources available in the system.
4.3.3 Time Sensitive Tasks with Benefit

Figure 4.5: Three Time Sensitive Tasks with Initial Shares 1, 2, 3 and initial weights 8, 4, 1

In figure 4.5 each task will receive its guaranteed requested share of 10%, 20%, and 30%. In addition to its share it will receive extra share from the idle resource based on its weight. In this case the total requested share is 60%, so there is an extra 40% available. This 40% will be divided according to the initial weight of each task. Task 1 with share of 10% will receive an extra 24% share based on its initial weight. Task 2 with share 20% will receive an extra 12% share. Task three with share 30% will receive an extra 3% share. This makes the total share for the three
tasks 34%, 32% and 33% respectively and this explains why the figure above shows the tasks are having very close shares.

Figure 4.6: Three Time Sensitive Tasks with Initial Shares 2,2,2 and initial weights 3,1,8

Figure 4.6 shows three tasks with equal initial shares of 20% but have different initial weights. This means all the tasks will receive equal share in addition to extra share from the idle resource. The total shares requested is 60% so this leaves an extra 40% share to be divided between the three tasks according to their initial weights. Task 7554 will receive a 20% share with an extra 26% share from the 40% idle resource which makes its total share 46%. Task 7552 will receive a total share of 30% and task 7553 will receive a total share of 23%. 
Mixed Tasks

Figure 4.7 shows three tasks with initial shares of 3, 5, 0 and initial weights of 0, 2, 3. Task 7821 will receive a guaranteed share of 50% and an extra 8% of the idle resource which makes its total share 58%. Task 7820 will receive a guaranteed share of 30% only since it is not asking for any benefits from the idle resource (initial weight is zero). Task 7822 which is a regular task will receive a share of 12%. After 16 seconds when task 7821 leaves the competition task 7820 will continue to receive its guaranteed share of 30% only and task 7822 share will increase to 70%.
Figure 4.8 shows three tasks with initial shares of 2,0,0 and initial weights of 1,4,6. Task 8344 has an initial weight of 20% which will be guaranteed to receive, and an initial weight of 1. The total weight that task 8344 will receive will be 27%. Task 8345 and task 8346 do not have any initial weights so they are considered regular tasks and compete based on their initial weights for the resource after task 8344 which has an initial share (considered a time sensitive task) will receive its share. Task 8346 will receive a share of 54% and task 8345 will receive a share of 36%. After task 8346 leaves the competition after 23 seconds, the shares of tasks 8344 and 8345 will change. Task 8344 will still receive its share of 20%, but will now receive an extra 16% instead of 7% since more idle resource is available now after task 8346 left, so its total share would be 36%. Task 8346 will now receive a share of 64%. 

Figure 4.8: Three Tasks with Initial Shares 2,0,0 and initial weights 1,4,6
4.3.5 Lag Charts

Lag as defined before is the difference between the ideal service time the task is supposed to receive and the actual service time the task actually receives. One of the goals of the PSS algorithm is to assure that the lag is bounded at all times. According to the algorithm the lag is bounded as \((-r < lag_k(t) < \max(R_{max}, q))\) where \(q\) is the quantum size and \(r\) is the length of the request issued by the task, and \(R_{max}\) represents the maximum duration of any request issued by any task in the system. This guarantees that any task will not be delayed more than \(\max(R_{max}, q)\).

For the three clients in figure 4.9 the lag for any client is clearly bounded. For task

![Figure 4.9: Lag for Three PSS Tasks](image)

1 which asks for a request of length 6 the lag is bounded at all times\((-6 < lag < 6)\). Task 2 with request length 3 has its lag bounded as \((-3 < lag < 3)\). Task 3 has its lag bounded as \((-2 < lag < 2)\).
Figure 4.10: Lag for Three PSS Tasks

For the three clients in figure 4.10, task 1 which asks for a request of length 4 the lag is bounded at all times ($-4 < lag < 4$). Task 2 with request length 2 has its lag bounded as ($-2 < lag < 2$). Task 3 with request length 1 has its lag bounded as ($-1 < lag < 1$).
4.4 Handling System Dynamics

To demonstrate task blocking and how clients behave in such cases number of tests have been performed. The figures that will follow will demonstrate how the weights of the tasks in the system are affected when a task sleeps and when it wakes up. The effective weights of the tasks will be recalculated to assure fairness between the remaining tasks. When the task wakes up and would like to join the competition again it will be assigned a new virtual eligible time and a new virtual deadline which will be affected by its lag (whether it left competition with a positive or negative lag). Also the virtual time of the system will be updated and recalculated with the value of the lag of the awaken task considered.

Figure 4.11 shows the behaviour of three PSS tasks where one of the tasks sleeps for an amount of time. Before task 9675 goes to sleep the three tasks receive their shares according to the PSS algorithm where task 9675 receives a share 57%, task receives 9676 28% and task 9677 14%. At time t=10 seconds task 9675 leaves the competition by going to sleep. That is when the shares of tasks 9676 and 9677 increase to 66% and 33% respectively. This happens because the system updates the virtual time when task 9675 leaves the competition and divides the idle resource fairly between the remaining tasks in the system, and that is why the remaining two tasks get increased share. The scheduler updates the virtual time based on the value of the lag that task 9675 had when going to sleep and recalculates the effective weights for the remaining tasks. At time=14 seconds task 9675 rejoins the competition; again the virtual time is updated and the effective weights are recalculated and the shares go back to their original values until task 9675 leaves competition. The two figures 4.12 and 4.13 also display the affect of sleeping tasks.
4.5 Graceful Degradation

Figure 4.14 displays the effect of graceful degradation. In this case the total initial share is larger than one, which results in giving each task a weight equal to its share and then calculating the effective weight based on that.

Figure 4.11: Three Tasks with Initial Shares 0,0,0 and initial weights 4,2,1. Demonstrating Task Blocking
Figure 4.12: Three Tasks with Initial Shares 5,3,1 and initial weights 1,1,1. Demonstrating Task Blocking
Figure 4.13: Three Tasks with Initial Shares 0,0,0 and initial weights 6,4,2. Demonstrating Task Blocking

A Plot of Operations performed by Tasks against time

- Task 10140 Initial Share=0 Initial Weight=6
- Task 10149 Initial Share=0 Initial Weight=4
- Task 10150 Initial Share=0 Initial Weight=2

Number of dummy operations performed vs Time in seconds
Figure 4.14: Three Tasks with Initial Shares 7,4,3 and initial weights 0,0,0. Demonstrating Graceful Degradation
5 Conclusions and Future Work

Having one general operating system design with a scheduler that supports both real and non-real-time applications has become a necessity these days with all the multimedia rich applications that exist. Real-time applications require guaranteed shares and non-real-time applications seek fairness. The ability to integrate these two types of applications is possible using proportional share resource allocation algorithms such as the Earliest Eligible Virtual Deadline First (EEVDF) algorithm. An implementation of a new model based on the Earliest Eligible Virtual Deadline First (EEVDF) model was implemented where each task was characterized by an initial share and initial weight that classifies tasks into one of three classes: time-sensitive tasks, time-sensitive tasks with benefit and regular tasks. This new model was implemented as a loadable module into the Linux kernel. A mixed mode of applications can co-exist together and compete for the same resource, and the model was able to ensure fairness and guarantee that time-sensitive tasks receive their guaranteed share. Moreover, graceful degradation in the system was possible during overload situations. Evaluation of this model has shown that the tasks requirements were met. Tasks with requested guaranteed shares received their shares whereas regular tasks received shares based on their weights. Additionally when overload situations occurred the system was able to handle the situation gracefully. The lag charts have also proven that there will be a bound on the lags that clients have.

5.1 Study Limitations

In the original EEVDF model proposed by Stoica et al. [4] the requests issued by the different tasks were stored in a binary search tree structure where it has a time complexity of $O(\log N)$ for retrieving, inserting and deleting elements. In our
model task requests are stored in a linked list which has higher time complexity compared to the binary search tree. Implementing the binary search tree would speed up retrieving eligible tasks. Another area would be better handling of graceful degradation when system overload situation occurs. In such cases time-sensitive applications get their shares guaranteed on the expenses of degrading the shares of regular tasks.

5.2 Future Work

Implementation of the model where the task’s requests are stored in a binary search tree. This would reduce the scheduling overhead and make the search for the eligible task much faster. Additionally providing protection for regular tasks from starvation in overload situations. Another important measurement would be measuring the scheduling overhead and comparing that to the original Linux scheduling overhead.
References


