COPS: A Framework for Consumer Oriented Proportional-share Scheduling

by

Abhijit Deodhar

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Godmar Back, Chair

Kirk Cameron

Eli Tilevich

May 14, 2007

Blacksburg, Virginia

Keywords: Scheduling, proportional share, VTRR, resource management

Copyright ©Abhijit Deodhar 2007

COPS: A Framework for Consumer Oriented Proportional-share Scheduling

Abhijit Deodhar

Abstract

Scheduling forms an important aspect of operating systems because it has a direct impact on system performance. Most existing general-purpose schedulers use a priority-based scheme to schedule processes. Such priority-based mechanisms cannot guarantee proportional fairness for every process. Proportional share schedulers maintain fairness among tasks based on given weight values. In both of these scheduler types, the scheduling decision is done per-process. However, system usage policies are typically set on a *per-consumer* basis, where a consumer represents a group of related processes that may belong to the same application or user.

The COPS framework uses the idea of consumer sets to group processes. Its design guarantees system usage per consumer, based on relative weights. We have added a share management layer on top of a proportional share scheduler to ease the administrative job of share assignment for these consumer sets. We have evaluated our system in real world scenarios and show that the CPU usage for consumer sets with CPU-bound processes complies with the administrator-defined policy goals.

Acknowledgments

I wish to express sincere thanks to my advisor, Dr. Godmar Back for guiding me throughout the thesis. I thank him for giving me the opportunity to work on a wonderful thesis topic. The knowledge I gained from him in these two years will immensely help me in my career.

I also want to thank my committee members Dr. Kirk Cameron and Dr. Eli Tilevich for giving valuable comments on my thesis.

I want to thank my brother, Akshay for proof reading my thesis and giving insights. My parents have supported me throughout this journey. I thank them for giving me the opportunity to study abroad and providing emotional support. Finally, I would like to thank God for all the strength He gave for successfully completing this work.

Contents

List of Figures vii

List of Tables ix

1 Introduction 1

1.1 Motivation 1

1.1.1 Problem Statement 4

1.2 Contributions 6

1.3 Outline 6

2 Background 7

2.1 Concept of CPU Scheduling 7

2.2 Scheduling in Linux 2.4 9

2.3 Scheduling in Linux 2.6 9

2.4 Policies used by the Linux Scheduler 11

2.5 Proportional Share Scheduling 12

2.5.1 Terminology 12
2.5.2 Comparison with Priority based scheduling 15
2.6 VTRR Algorithm 16

3 Design **20**
3.1 Motivating Applications 20
3.2 Consumer Sets and Active Sets 24
3.2.1 Active Set Invariant 27
3.3 Data Structures 31
3.4 Algorithms 33
3.4.1 Algorithm Complexity 37
3.5 Framework Portability 39

4 Framework Components **40**
4.1 Overview 40
4.2 VTRR Scheduler Port 42
4.2.1 Data Structures 44
4.2.2 /proc Interface 45
4.3 User Space Tools 45
4.3.1 COPS Daemon 46
4.3.2 COPS Client 47
4.3.3 Web Interface 48

4.4 Development Tools 49

5 Experimental Evaluation **52**

5.1 Evalution Goals and Test Scenarios 52

5.2 Experimental Setup 54

5.3 Experimental Results 55

5.3.1 Multiple Process Scenarios 55

5.3.2 Multiple Consumer Set Scenarios 56

5.4 Limitations 67

6 Related Work **68**

6.1 Proportional Share Schedulers 68

6.2 Resource Management Frameworks 71

6.3 Resource Management Policies 73

7 Future Work and Conclusions **75**

7.1 Contributions and Conclusions 75

7.2 Future Work 76

Bibliography **78**

List of Figures

2.1 Process states 8

2.2 Linux runqueue structure 10

2.3 VTRR flowchart 17

3.1 Process tree 21

3.2 Relation between consumer set and active set 25

3.3 Pipe scenario under equal shares policy 30

3.4 Pipe scenario under no-split policy 30

3.5 Active sets representation 32

4.1 System components 41

4.2 Web interface screenshot 49

5.1 Infinite loop scenario 55

5.2 Service error for infinite loop scenario 56

5.3 Fork-wait-exit scenario 57

5.4 Loop vs fork-exit-wait 58

5.5 Fork-wait-exit with loop 59
5.6 Kernel compilation 60
5.7 Multiple httpd instances 61
5.8 Service error for httpd scenario 62
5.9 Pipe scenario A 63
5.10 Pipe scenario B 64
5.11 Pipe scenario C 65
5.12 Multimedia player 66

List of Tables

3.1 Share distribution for loop and random | sort 29

4.1 Share table 46

Chapter 1

Introduction

1.1 Motivation

Most operating system schedulers are designed for a mix of workloads. Server machines may run multiple server applications such as web servers and build servers, whereas desktop users may perform multiple activities simultaneously, such as word processing, browsing the Internet, playing computer games, listening to music, etc. Therefore, to be suitable for both desktop and server applications, the scheduler needs to meet conflicting goals of minimizing response times for interactive applications while maximizing overall system throughput. Tasks (or processes) are generally classified into two types: tasks which spend most of their time performing computations are called CPU bound tasks, whereas tasks which spend most of their time waiting for input-output operations are called I/O bound tasks.

To satisfy the conflicting scheduling goals for CPU and I/O bound tasks, the Linux operating system uses a variant of the multilevel feedback queue scheme (MLFQS) [CK68]. This class of schedulers uses multiple queues of ready-to-run (or runnable) tasks, each queue representing a priority level. Tasks move among different priority levels depending on their behavior (CPU

bound or I/O bound). To reduce response time, I/O bound tasks are given higher priority than CPU bound tasks. Users can decrease a task's priority using the 'nice' command, but only superusers can increase a task's priority. The priority value set by the nice command does not directly correspond to resource rights, a limitation that applies to most general-purpose schedulers.

Proportional share schedulers assign tickets (also known as shares or weights) to every consumer of a shared resource. The number of tickets assigned to a consumer defines the relative proportion of the resource rights of that consumer with respect to other consumers. For example, if three consumers are assigned 3, 2 and 1 tickets, then a proportional share scheduler will distribute the resource using a ratio of $\frac{1}{2} : \frac{1}{3} : \frac{1}{6}$ or 50%:33%:17%. The advantage of this method is that consumers make progress at a rate proportional to their tickets.

A number of proportional share schedulers has been proposed [KL88, DKS89, WW94, NVZ01, CNC05, CNS06, DC99]. However, we do not find them integrated into the scheduler code in mainstream operating systems. One of the reasons for this lack of adoption is the lack of a share management layer. Proportional share schedulers require appropriate share values to be defined for each process. Since operating systems may have hundreds of processes running simultaneously, it is difficult to specify a share value for every process individually. A share management framework abstracts the shares of individual processes away from the system administrator or user. To demonstrate the need for a framework like COPS (Consumer Oriented Proportional-share Scheduler), we describe examples that show that existing schedulers fail to address some common practical scenarios.

Consider the way in which a computer system is used. Multiple servers may be hosted on the same machine. For instance, a web server and a build server for faster kernel compilation may be running on the same machine. Under normal load conditions, the requests for both servers are satisfied in reasonable time. Imagine that due to an upcoming event, the traffic on the Web server shoots up. A load balancing module of the Web server might fork

off more processes to handle these additional incoming connections, thinking that it is the only application running on the server. Because the OS assigns CPU use on a per-process basis, this increase in the number of processes could starve the build server, slowing down compilation.

Consider another scenario involving a university lab machine which is shared by students and professors in a department. Imagine that a student is working towards a conference paper deadline and hence running multiple, CPU-intensive experiments. The experiments run by the student monopolize the CPU. Other users logged on to the system would observe longer response times for their tasks. The problem of assigning a suitable share of CPU to each user could be solved if we had a mechanism to specify resource rights for each user and a scheduler that would respect these settings.

These examples have in common that if a consumer (any application or user) creates multiple CPU bound processes, longer task completion times for the remaining consumers may result. The root of the problem lies in the way in which CPU scheduling is performed - per-process and not per-consumer. Because per-process scheduling may lead to undesirable results, a consumer-oriented scheduling framework is necessary. To implement consumer-oriented scheduling, we need to either specify which processes belong to a consumer or provide a way that automatically identifies processes belonging to a consumer.

Previous approaches to identifying related processes include the approach used in SWAP [ZN04] and the resource container approach [BDM99]. SWAP tries to identify process dependencies using the processes' system call history. It uses a confidence interval based model to predict the relationship between processes. Because the feedback based model is based on confidence intervals, it may err.

In some cases, the binding between a process and an application may not be static. For example, clients may use a network socket to communicate with a server. Whenever a new

client request is received by the server, the server process handles the client request as a part of the client connection. In the resource container model, a process executes on behalf of a container, defined as an entity that contains all system resources being used by an application. Although resource containers provide a way to handle the client/server scenario by executing a process on behalf of different resource principals, this approach requires rewriting applications to make use of it.

1.1.1 Problem Statement

This thesis solves the following problem:

"How can we integrate proportional share scheduling into existing operating systems so that the resulting system is flexible, easy to use, and provides per-consumer fairness guarantees based on administrator-defined policy goals?"

We solve this problem using the concepts of *Consumer Sets* and *Active Sets*. A consumer set consists of all processes that share a proportion of the CPU. These processes could be processes created by an application, user, or group of users; in addition, administrators can directly assign processes to consumer sets. A consumer set's active set is the subset of runnable (also known as ready-to-run) processes in the consumer set. The administrator specifies the weights/proportions for different consumers rather than for individual processes. These weight values are mapped to a cumulative share value for each consumer set. An underlying proportional share scheduler handles the scheduling by assigning shares to each process within the consumer set based on a scheduling policy. At runtime, new processes may be added to a consumer set or processes may terminate. Processes could also change their state from running to blocked (waiting for I/O), blocked to ready-to-run and ready-to-run to running. In each case, our framework must maintain the proportionality of weight assignments for each consumer set. We address this problem using *Share Management Poli-*

cies. When a process in a consumer set blocks, we reassign its shares to other runnable processes in the consumer set, thereby maintaining the proportion of weights.

Our approach is not a new scheduler, but a framework on top of the VTRR [NVZ01] proportional share scheduler. We have integrated VTRR (Virtual Time Round Robin) as a new scheduling policy that can coexist with the existing Linux scheduler. This approach gives the user the freedom to choose which tasks are scheduled by the existing scheduler and which ones by the COPS framework. By default, all boot time system services and daemons are scheduled using the existing Linux scheduler. The VTRR scheduler is used to manage the remaining CPU time, which is then distributed to different active sets based on weights. COPS comprises user space tools such as a Web interface for configuration of consumer sets, a share daemon to convert consumer weights to shares and a client program to start a new application as a member of a consumer set. These tools ease the administrator's job of share assignment, thus adding practical value to our system.

To evaluate our system, we implemented the COPS framework in the Linux kernel and analyzed its behavior using real world scenarios. We found that COPS successfully isolates consumers from each other. It can be used in multi-threaded applications for isolating applications under different loads and in multi-user systems for isolating different users. For consumer sets with CPU-bound applications, we found the CPU usage for the consumer sets conforms to the weight assignment. The service error, defined as the difference between the actual CPU time consumed and the expected CPU time over a time interval, was small in all cases we observed. The COPS framework is easy to use and can be adopted by system administrators to manage shared systems.

1.2 Contributions

The contributions of this thesis are as follows:

1. The active set idea and its implementation in the Linux kernel.

2. Share management policies for managing active sets.

3. A share management framework consisting of user space tools including a Web interface for easy share management.

1.3 Outline

Chapter 2 discusses the terminology and concepts related to proportional share scheduling and the VTRR algorithm in detail. Chapter 3 discusses the design of active sets, the algorithms for operations on active sets and the share management framework, followed by the kernel implementation details in chapter 4. Chapter 5 talks about our experimental evaluation and the results obtained. We then present the related work in chapter 6 and our conclusions in chapter 7.

Chapter 2

Background

2.1 Concept of CPU Scheduling

Multitasking environments can have multiple runnable processes at any given time. The CPU scheduler makes the decision regarding which process to run next, choosing among the currently runnable processes. This decision has a direct impact on resource utilization and overall system performance. The scheduler is invoked during the following events: [SGG04]

- A process switches from the running to wait state.
- A process switches from the running to ready-to-run state.
- A process switches from the wait state to ready-to-run state.
- A process exits normally or is terminated by the user.

Figure 2.1 shows the different states in which a process may be during its lifetime [Lov04]. A process which is running on the CPU may sleep (block) for an I/O operation (for example, a file read operation to read a file from disk) and hence transition to the process interrupted

(blocked) state. When the I/O operation completes, the process is woken up (unblocks) and it enters the ready-to-run state. The scheduler maintains a runqueue of processes that are in the ready-to-run state. When a new process is created, for example using the 'fork' system call in Linux, the newly created process is added to the runqueue. Based on the specific scheduling algorithm, a process is selected to run on the CPU. A process running on the CPU may be preempted either because its timeslice expires or a higher priority process becomes ready-to-run, preempting the currently running process. When a process exits, it enters the 'zombie' state until the parent invokes the 'wait' system call to retrieve the child's exit status.

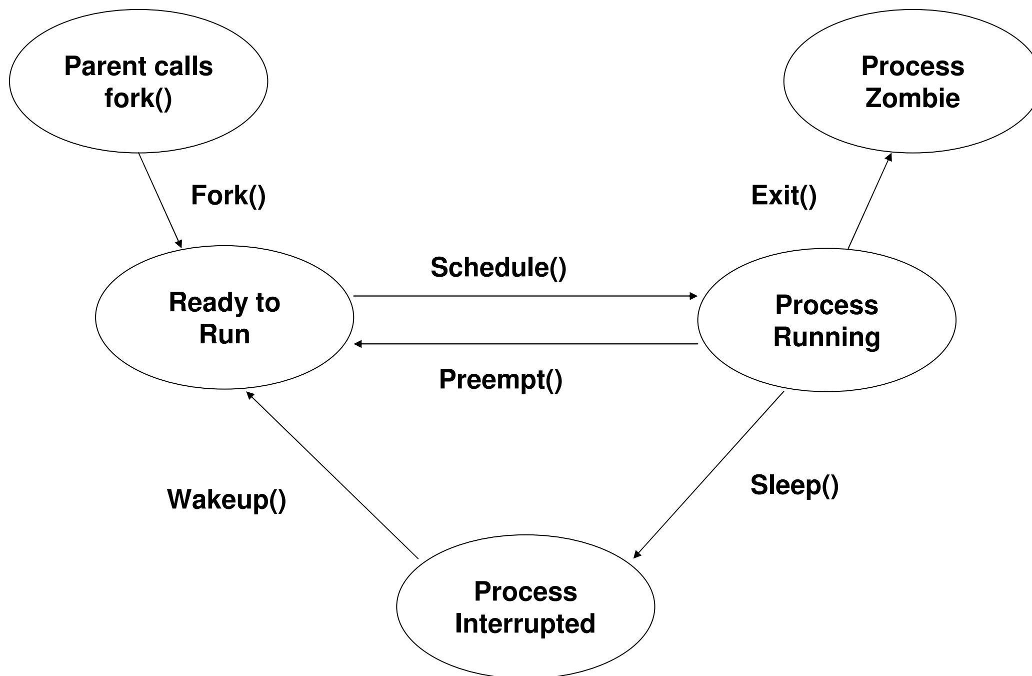

Figure 2.1: Process states

2.2 Scheduling in Linux 2.4

The Linux 2.4 kernel implements a simple dynamic priority based scheduler. Tasks are assigned different priority levels. Priority levels 1 - 99 are for real time tasks, whereas static priority levels 100 - 139 directly map to user process nice values -20 to +19. In Linux, the lower the number, the higher is the priority. Thus a value of 1 corresponds to the highest priority and 139 corresponds to the lowest priority. By default, every process has a nice value of zero which maps to a static priority level of 120. The scheduling decision is based on each task's goodness value, which is calculated based on the nice value (static priority) and interactivity (dynamic priority). Tasks which voluntarily give up their timeslice are given a higher goodness value when they become runnable.

The scheduler scans all runnable tasks and selects the task with maximum goodness value for execution. As the number of tasks on the runqueue increases, the time to select a process for execution increases linearly. Hence the scheduler is not scalable. Secondly, the scheduler maintains a single runqueue for all processors and does not use processor affinity when scheduling tasks, which adversely affects performance on a SMP system. Because of these disadvantages, the Linux scheduler was rewritten in the 2.6 kernel.

2.3 Scheduling in Linux 2.6

The goals of the redesigned scheduler were as follows: [Lov04]

- Scheduling decision in constant, O(1) time.
- Fairness in process selection to avoid starvation.
- Support for SMP systems.

- Scalable to a large number of runnable processes.

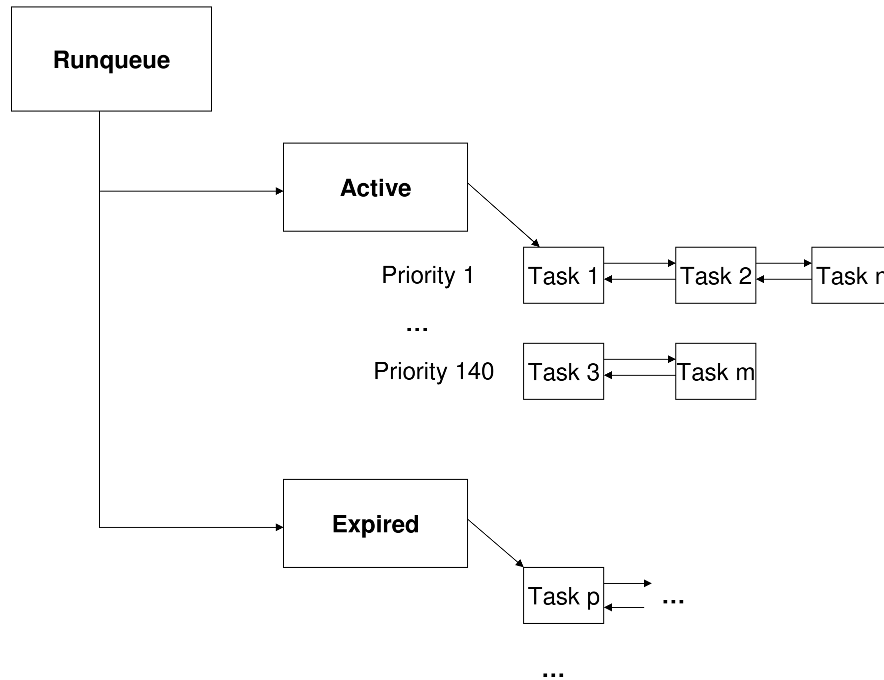

Figure 2.2: Linux runqueue structure

The scheduler for the 2.6 kernel series is commonly known as the "O(1) scheduler." It uses a simple mechanism to achieve constant time process selection. The scheduler maintains two priority arrays, the active array and the expired array as shown in figure 2.2. Tasks with equal priority are maintained in a queue, with one queue per priority level. Initially, the active array contains all tasks that have timeslices left. The kernel maintains a bitmap of the priority levels in the system. The scheduler then picks up the first task based on the priority bitmap and executes it for one timeslice. As tasks complete their time quantum, their priority is recalculated and they are added to the expired array. When all tasks from the active array have exhausted their timeslices and have been moved to the expired array, the scheduler swaps the pointers pointing to the active array and expired arrays. The pointer

manipulation takes constant time. The priority recalculation is done as each task uses its timeslice and hence there is no need to scan the array to find the task whose priority is being recalculated. Choosing a new task is based on finding the first 1-bit in a bitmap which takes a small, bounded amount of time. Hence the overall time complexity of this algorithm is O(1).

The O(1) scheduler supports SMP systems by using the concept of processor affinity, a mechanism used to bind a process to a particular CPU. Each processor is associated with its own scheduler runqueue. Tasks are migrated between processors only if there is an imbalance in the runqueue sizes. The scheduler implements a load balancer which monitors the runqueue status periodically. If the load balancer finds an empty runqueue, it tries to migrate higher priority tasks from the busiest runqueue to this runqueue. Higher priority tasks are chosen because it is more important to fairly distribute high-priority tasks than lower-priority ones [Lov04]. This mechanism ensures that some processors do not stay idle while others are overloaded.

2.4 Policies used by the Linux Scheduler

The Linux scheduler provides three scheduling classes or policies [IEE04]: SCHED_RR, SCHED_FIFO and SCHED_OTHER. SCHED_RR and SCHED_FIFO are real-time scheduling classes. Linux provides soft real-time behavior, which means that the Linux kernel tries to schedule applications within timing deadlines by giving scheduling preference to real-time tasks over tasks in any other scheduling class. However Linux cannot guarantee that real-time tasks will always be able to meet their timing constraints [Lov04].

- SCHED_FIFO

  SCHED_FIFO is a real-time, first-in first-out scheduling policy. It does not use times-

lices. A process is run to completion before selecting a new process for execution. Being a real-time scheduling policy, tasks in this class have higher priority than tasks in the SCHED_OTHER class.

- SCHED_RR

  This policy is similar to SCHED_FIFO : it is a real-time policy, however it uses timeslices. In this policy, a task is executed up to a predetermined timeslice, after which it is preempted and added to the end of the runqueue as per the round-robin algorithm.

- SCHED_OTHER [1] This policy is used for processes that are not in the SCHED_FIFO or SCHED_RR classes. Processes scheduled by the SCHED_OTHER policy are subject to dynamic priority adjustment.

2.5 Proportional Share Scheduling

Proportional share scheduling [SGG04] is a type of scheduling in which each process is given a weight or share value. A proportional share scheduler assigns CPU time to different tasks in proportion to their share value. Many proportional share schedulers have been developed for CPU scheduling [KL88, Ess90, Hen84, WW94] and packet scheduling in networks [BZ96, PG93].

2.5.1 Terminology

In this section, we define commonly used terms with respect to proportional share scheduling.

1. *Proportional Fairness*

[1] This policy is referred to as SCHED_NORMAL in the Linux source code

A proportional share algorithm for a resource is said to be fair if every task utilizes the resource exactly in proportion to its share. Let $Share_A$ denote the share of a task A and $Service_A$ denote the amount of time the task utilized the resource. A system is proportionally fair for a time interval (t_1,t_2) if for each task A, the following holds [NVZ01]:

$$Service_A = \frac{(t_2 - t_1) * Share_A}{\sum_i Share_i} \tag{2.1}$$

Proportional fairness can be achieved in two ways; by varying the time quantum for each task based on share value or by varying the frequency of execution of each task. An algorithm that is proportionally fair is said to have perfect *proportional sharing accuracy*.

2. *Service Error*

   In a real world system, not all tasks are able to consume a time-multiplexed resource such as the CPU simultaneously. Instead, each task is given a time interval for execution and the scheduling decision is made after every discrete time intervals. Therefore it is not possible to maintain fairness across all time intervals. The service error metric is used to measure the accuracy of proportional sharing. The service error is defined as the difference between the actual resource consumption and the expected (ideal) resource consumption for a task in the interval (t_1,t_2). Mathematically, it is expressed as [NVZ01]:

$$ServiceError_A = Service_A - \frac{(t_2 - t_1) * Share_A}{\sum_i Share_i} \tag{2.2}$$

   The service error is an indicator of the proportional sharing accuracy of an algorithm. A positive error indicates that the task used more than its intended allocation, whereas a negative error indicates that the task received less than its intended allocation during that time interval. The service error should be small, with zero being the ideal case.

3. *Virtual Time*

Some proportional share schedulers use virtual time (VT) to measure the degree to which a task received its proportional allocation relative to other tasks. Virtual time represents the *relative* rate of execution of tasks in order to make the next scheduling decision. To calculate VT, we need to scale the service time by the share value. Formally, it is expressed as [DKS89]:

$$VT_A(t) = \frac{Service_A(t)}{Share_A} \tag{2.3}$$

4. *Queue Virtual Time*

   Some proportional share schedulers use the Queue Virtual Time (QVT) as a basis to decide if a task is ahead or behind in its proportional allocation. QVT is calculated as:

$$QVT(t) = \frac{t}{\sum_i Share_i} \tag{2.4}$$

   If a task's VT == QVT, then that task has received its proportional allocation of resources. If a task's VT < QVT, then the task has received less than its proportional share. If a task's VT > QVT, then the task has received more than its proportional share.

5. *Work-conserving Scheduler*

   A scheduling algorithm is said to be work conserving if it never lets a processor idle as long as there are runnable threads in the system [CAGS00]. In proportional share scheduling, each task is assigned a share. If a task is I/O bound, it may not use its entire share of the CPU. For example, if a text editor is assigned 50% of CPU time, it may use only a fraction of CPU and sleep on user input for the remaining time. Therefore, there may be surplus resources that are not used by any task. A work-conserving proportional share scheduler allocates such surplus resources proportionally among the runnable tasks. Hence, a task may receive more than its configured share unless the system is fully loaded and all competing tasks are runnable [KKC05].

2.5.2 Comparison with Priority based scheduling

In contrast to priority scheduling, proportional share scheduling assigns shares to each task. The share value defines a proportion of the CPU for that task. This mechanism guarantees that the CPU allocation is proportionally fair in the long term and avoids starvation of processes. As the load on the system increases (there are more ready-to-run processes), the system degrades gracefully because every task receives proportionally less CPU than it requested. The disadvantages of proportional share schedulers are potentially long waiting times for I/O bound applications due to the inability to distinguish between interactive and CPU bound tasks. For example, in an interactive editor application, the editor must be scheduled whenever there is a keystroke or mouse input from the user. Such a situation is better handled by a priority scheduler by giving the editor a high priority.

Proportional share schedulers are not well-suited to multimedia applications like audio players. The reason behind this behavior is that audio-player applications have strict latency requirements. It is necessary to play audio frames at precise time intervals, which requires a minimum CPU time guarantee within a time interval. In a proportional sharing scheme, as the number of runnable tasks increases, the CPU time allocated to a task within a fixed time interval decreases proportionally. Therefore, proportional sharing schemes are not suitable for multimedia applications.

In the BVT algorithm [DC99], a latency sensitive application such as a multimedia player can *borrow* virtual time from their future CPU allocation to gain scheduling preference. This concept is known as *warping*. However, if a task crosses its warp time limit, defined as the time for which a task is allowed to run warped, then BVT eventually penalizes the task using the unwarp time requirement U_i, which prevents the task from running warped again until time U_i. The unwarp time requirement limits the latency a task can add to other tasks.

In a resource reservation based approach, a task A can reserve X units of time out of every

Y units. Such a mechanism guarantees that for every time interval of size Y, task A (if runnable) is scheduled for at least X time units. This stronger guarantee provided by a resource reservation based approach is better suited for multimedia applications than a proportional share approach [JRR97].

2.6 VTRR Algorithm

Virtual-Time Round-Robin[NVZ01] is a proportional share scheduler. It can provide good proportional sharing accuracy with O(1) scheduling overhead. The name VTRR is derived from combining the round robin concept from WRR (weighted round-robin) [SGG04] and the virtual time concept from WFQ [DKS89].

- WRR assigns a time quantum equal to the share value of the task, but runs all tasks with the same frequency. Although its time complexity is O(1), it may exhibit a high service error, especially for larger share values, because the round-robin algorithm gives all shares due to a task at once, while other tasks wait for their turn. The task currently scheduled is far ahead in its ideal allocation (positive service error). At any given time, the sum of the service errors for all tasks must be zero. Therefore, other tasks which are waiting accrue a negative service error [NVZ01].

- WFQ uses the concept of Virtual finishing time (VFT) to schedule tasks. VFT is the virtual time a task would have after scheduling it for one time quantum. All tasks are ordered on the runqueue from smallest to largest VFT. WFQ assigns the same time quantum to each task, but schedules tasks with different frequency based on the VFT. After each time quantum, the VFT is updated and the task is inserted into the runqueue based on the updated VFT value. Although the time complexity is O(n), the service error for each task is bounded by a single time quantum, resulting in better

scheduling accuracy [NVZ01] than WRR.

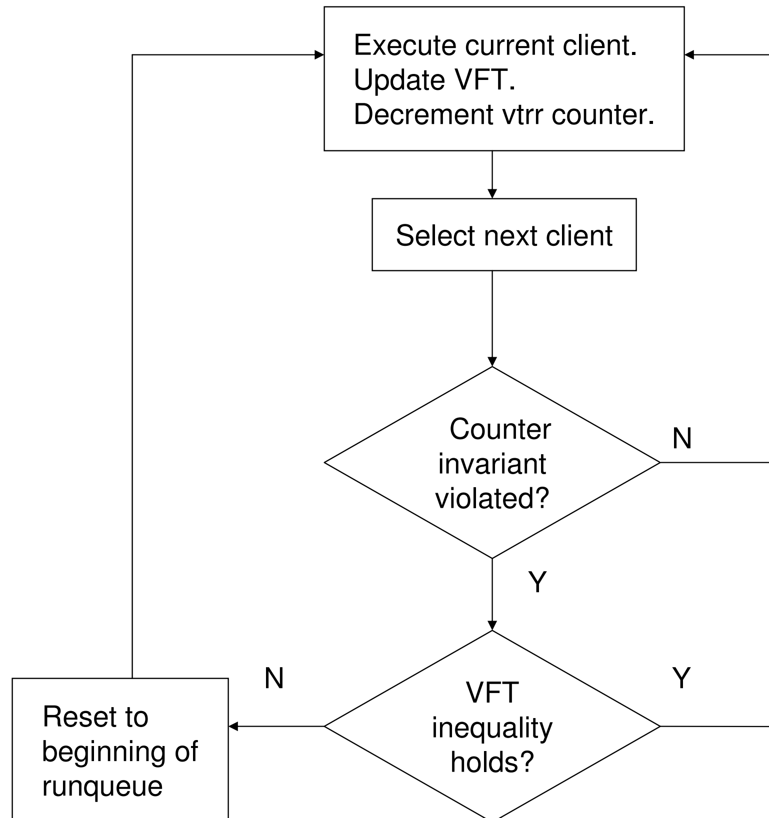

Figure 2.3: VTRR flowchart

The VTRR algorithm maintains a scheduling state consisting of the time quantum (which is equal for all tasks), the number of total shares, and QVT. VTRR defines a scheduling cycle as the sequence of allocations whose overall length is equal to the sum of shares of all runnable tasks in the system. Each task is associated with a share value, VFT, and a counter that tracks the number of remaining quanta the task should receive before the end of the cycle. Tasks on the runqueue are sorted based on their share values from highest to lowest. The time counter invariant and the VFT inequality are used to guide the algorithm

in maintaining proportional fairness at all times. At the beginning of the scheduling cycle, the counters for all runnable processes are initialized to their share value. When a process is scheduled for one time quantum, its counter is decremented. To ensure perfect fairness, all counter values should be zero at the end of the scheduling cycle. The *time counter invariant* enforces that if A $<$ B (task A sits ahead of task B on the queue), then the counter value of task B is always less than or equal to task A's counter value. Additionally, VTRR also tracks the relative progress of a task with respect to the other tasks using the *VFT inequality* (equation (2.5)) which compares the task's VFT with the QVT the system would have after one time quantum Q.

$$VFT_c(t) - QVT(t + Q) < \frac{Q}{Share_c} \tag{2.5}$$

The rationale behind VFT inequality is that the next task to be executed may be far behind its proportional allocation, thus having a large service error. Even after executing it for one time quantum, the task would still remain behind its proportional allocation. Therefore, to improve proportional fairness, the next task is chosen if the VFT inequality holds. However, if the VFT inequality is violated, it means that the next task would receive more than its proportional allocation. In this case, VTRR resets back to the first task on the runqueue. The process is repeated in a round-robin manner for all tasks on the runqueue as shown in figure 2.3.

When a newly forked task or blocked task becomes runnable, it is added to the runqueue based on its share value. For a new task, the VFT is set to the QVT of the system because it has not yet consumed any resources. For a blocked task which becomes runnable, the VFT is set by taking the maximum of its original VFT and the QVT. The rationale behind this decision is that a task cannot trick the algorithm and receive more CPU time by blocking and then immediately becoming runnable. At the same time, the QVT ensures that the task is treated similar to a newly forked task which became runnable. To preserve the time counter invariant, the counter for the runnable task is clamped by taking the minimum of

the counters of its neighbors. When a task blocks or is suspended, it is simply taken out of the runqueue. If it was the *current* task, a new task is chosen for execution.

In summary, VTRR uses the share value to determine the position of the task on the runqueue. It schedules in a round-robin manner using the time counter invariant and VFT inequality to maintain proportional fairness and to minimize the incurred service error.

VTRR is an O(1) algorithm due to the following properties:

- Scheduling decisions are made by maintaining the current position and choosing the next task on the sorted runqueue, which requires constant time.
- The VFT update and current position update requires O(1) time.
- The cost of resetting the time counters is amortized over N scheduling decisions and does not depend on the number of processes. Hence this step takes O(1) time.

COPS is based on VTRR as the underlying proportional share algorithm. However, the COPS framework could be easily modified to use other proportional share algorithms.

Chapter 3

Design

In this chapter, we describe the concept of active sets in detail. We present the algorithms that operate on active sets and their time complexity. To motivate this discussion, we first revisit typical use cases of our system.

3.1 Motivating Applications

- Consider a university lab machine which is used by graduate students in the department. The lab administrator wants to assign a higher share of CPU resources to students facing an assignment deadline than to other users on the system. At the same time, he wants to ensure that a user cannot trick the system by logging in multiple times to increase his share of CPU. The administrator's goal is to maintain proportional fairness among different users of the system.

- Consider another scenario in which a web server is hosted on a machine that is also used to perform kernel compilations. The web server creates processes to handle new connections. The kernel compilation generates a large process tree as shown in Figure

3.1. This tree comprises a number of processes such as the C compiler frontend cc1, the assembler program as, instances of the bash shell, and others. These processes may run for some time, then block on I/O, run again for some time and finally exit, passing through multiple states during their lifetime. We want to assign the CPU to these two applications such that if one of the servers is loaded with multiple requests, the other server should not be starved of CPU.

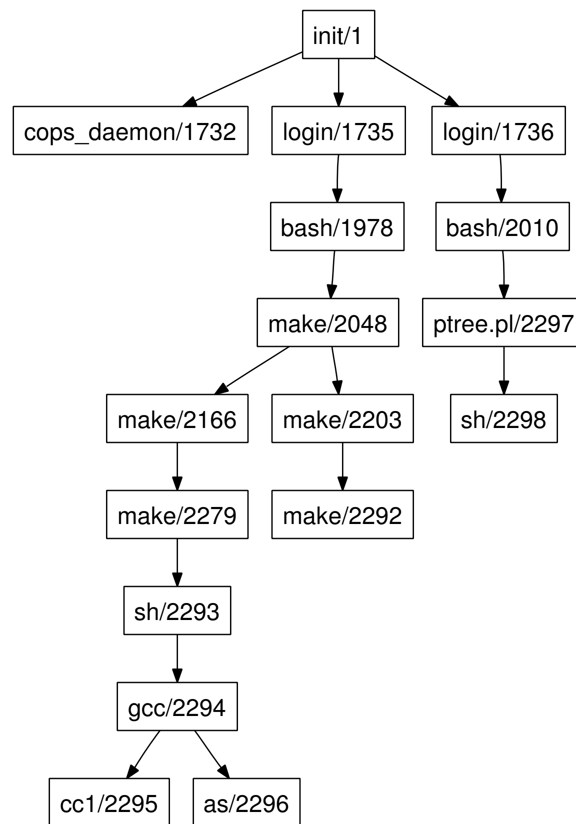

Figure 3.1: Process tree

From the above examples, we observe that:

1. Consumers may create a number of processes at runtime. These processes may switch between the ready-to-run and blocked states.

2. At a given point in time, only a subset of these processes might be in the ready-to-run state, in which they are eligible to be scheduled.

3. Usually processes belonging to one consumer share an ancestor process in the process hierarchy, although not always.

Our goal is to achieve proportional fairness among different consumers of the system, independent of the number and state of their comprising processes.

Let us examine what happens if we try to use existing techniques to solve these problems. Linux provides the *nice* command to adjust a process's priority. To provide equal CPU time for two processes A and B, we could simply assign the same nice value to both processes to achieve our goal. However, consider the case of two users using a system, user A and user B. Our goal is to assign *equal CPU time to these two users*. If user A ran three processes and user B ran two processes, the administrator would have to calculate the appropriate nice values for each process of user A and user B to achieve the goal, which would require modeling the effect of nice on the process's goodness value, then computing its inverse function.

1. We observe that it is cumbersome to handle complex scenarios with nice. The problem arises because nice values do not correspond directly to CPU share values. Even if an administrator does compute the correct nice to share relation, the effort would need to be repeated frequently. For instance, if user A created another process, this child process would inherit the nice value from the parent process, causing an increase in CPU usage for user A and thereby not respecting the previously computed proportionality of CPU share.

2. The processes owned by user A and B may change their state. When processes block,

they do not use their assigned CPU share. To maintain the proportional fairness between user A and B, we would need to recompute the nice values on each process state change such that the runnable processes for each user always consume their assigned shares. This scenario could potentially be handled by using tools such as strace to monitor the process states and trigger a recomputation of nice values from user space. Because of limitations in the implementation of the kernel's tracing interface, user space tools are not capable of tracking kernel events accurately [GP05]. There is a possibility that they may miss events. Therefore, we recognized the need for a kernel space implementation to address this problem.

A consumer may consist of multiple processes. To achieve our goal of proportional fairness among consumers, we could use a mechanism which assigned a cumulative share value to each consumer and a framework which distributed these cumulative shares among processes belonging to that consumer. However, processes consume CPU (utilize their share) only when they are in the ready-to-run state. If a process belonging to a consumer is blocked on an I/O operation, it will not consume any shares. Because processes do not utilize their share in the blocked state, static share assignment to processes does not maintain proportional fairness. Hence, we investigated a technique that transferred shares from the blocked process to the ready-to-run process.

Initially, we focused on selected process-related system calls where we knew that share transfer was necessary. We implemented the share allocation policies designed by [GP05] in the kernel space. These policies are as follows:

1. On process creation, i.e., fork(2), split the shares equally among the parent and the child process.

2. If the parent blocks to wait for the child process to complete, i.e., wait(2), transfer the shares of the parent process to the child process on which it blocks.

3. On process termination, i.e., exit(2), return the child's shares back to the parent process.

These policies have some limitations:

1. It is necessary to monitor all system calls in the kernel that trigger state-changing events such as process creation, termination and blocking. For example, on a wait, we have to track the pid of the process on which it is waiting in order to transfer shares, which is difficult if the parent does not specify which child it is waiting for (i.e., when wait(NULL) call is used.)

2. This scheme does not handle other blocking scenarios in which a share transfer is necessary. For example, if a process waits for data from a pipe or socket, the shares would not be reassigned.

Our design overcomes these limitations. We define a *consumer set* as a group of related processes. For the discussion below, the *consumerId* is the name associated with a consumer set. In our implementation, it could be the name of the application or the username of the user. The administrator assigns a *weight* value for this consumer set relative to other sets in the system. COPS maps the weight value to a share value per consumer set. Hence, from the kernel perspective, the entire consumer set is associated with a cumulative *share* value.

3.2 Consumer Sets and Active Sets

A consumer set is a group of related processes. An active set is the subset of ready-to-run processes from the consumer set. An active set is empty if there are no ready-to-run processes in the consumer set. If all processes in the consumer set are ready-to-run, then the active

set is equal to the consumer set. Figure 3.2 shows the relationship between consumer and active set. COPS manages set membership and share transfer within sets.

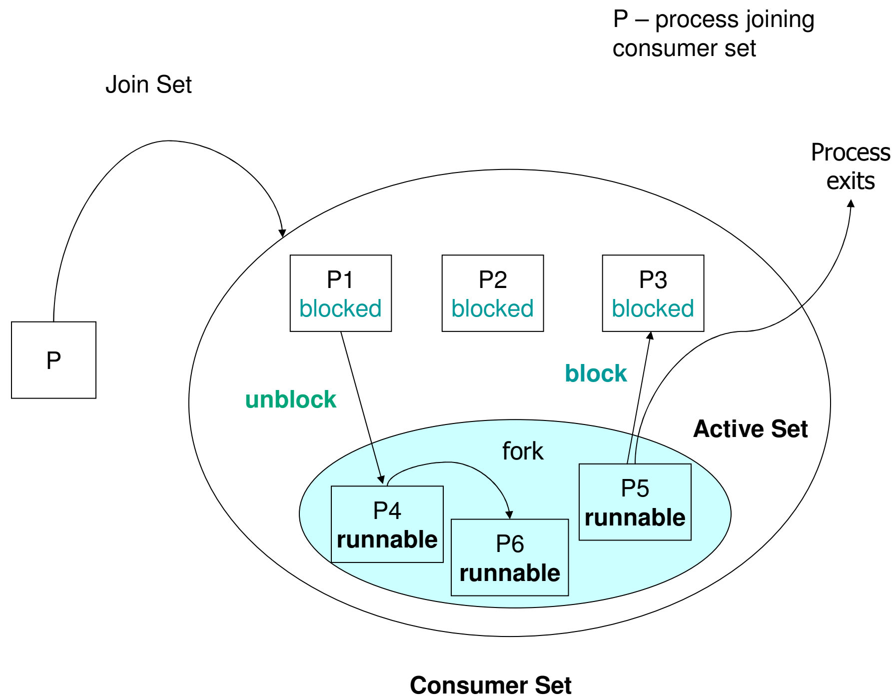

Figure 3.2: Relation between consumer set and active set

Managing Set Membership. It is necessary to identify which processes should belong to a consumer set. Nieh et al [ZN04] tried to solve the problem using an automatic approach, by detecting process dependencies using their system call history. This approach uses a feedback-based confidence evaluation model to predict the relationship between resource requesters, defined as a process that requests a resource and resource providers, defined as a process that may be holding the requested resource. Because the resource provider is not always known, this approach carries an inherent uncertainty, which may lead to errors.

Banga et al used resource containers [BDM99] to model system resources used by server

processes. Their approach executes a process on behalf of different resource principals, but requires rewriting of application programs to achieve good default behavior.

By contrast, our approach uses the concept of a 'seed' process. When an application is started, the first process created by it is called the seed process. This process is added to the consumer set. All descendants created by this seed process are automatically added to the consumer set. We also provide API's for adding additional processes to the consumer set. This mechanism can be used if the administrator believes that a process is related to the consumer set and should consume resources as per the policy defined for that consumer set. For example, the administrator may assign a cumulative share value for each user on the system. During login, a user's login shell is added to a consumer set. All programs run by this user will be added to his consumer set. Even if a user tried to trick the system by logging in multiple times, his overall CPU share would stay the same.

Share Management Policies. After identify the members of a consumer set, we need to balance the shares among them so as to respect administrator defined policy goals. For example, the Apache web server application uses a number of preforked httpd process instances which are in the blocked state when there are no connection requests. When a new connection request arrives, one of these processes is woken up and it becomes runnable to handle that connection. Conversely, if a process *blocks* on some event, it is removed from the runqueue. Thus the application owning this process would not use its shares completely, thereby disturbing the proportionality. To respect the administrator's weight assignments, we need to maintain the proportional share scheme at all times, which required enforcing the active set invariant, which we define next.

3.2.1 Active Set Invariant

We define the *Active Set Invariant Property* as follows:

> The sum of the shares of all processes in an active set always remains constant.

Let us denote the total shares assigned to a consumer set C as N and the share value of a process i in the active set A as S_i. Mathematically, we can express the invariant property as

$$\sum S_i = N, \text{for i} \in A \tag{3.1}$$

$$S_i = 0, \text{for i} \notin A \tag{3.2}$$

Equations (3.1) and (3.2) say that at any point, the runnable processes in the consumer set must use the cumulative shares assigned to it and the blocked processes should use none. If we are able to guarantee this property, then the following holds: If the active set is not empty, it uses its assigned cumulative share at all times, i.e., it never underutilizes the CPU. As explained in chapter 2, the VTRR algorithm is work-conserving. Therefore, an active set may use surplus resources if all other competing active sets are empty. For example, if the system has two consumer sets with equal share assignment, but one of the consumer sets has an empty active set, then the non-empty active set will use the entire CPU. It follows from the above discussion that using a proportional share scheduler, the resulting schedule will be fair to all applications based on their share assignments.

Although we assign a weight value to each consumer set, the proportional share scheduler still needs a share assignment for each set. Hence we need to convert the weight to a cumulative share value per set. The conversion is done by the COPS daemon program, which maps the relative weights assigned for consumers to cumulative shares for each active set. The cumulative shares assigned to a set are distributed to processes within the set based on a

policy. We defined a policy under VTRR (called vtrr_policy) which assigns equal shares to all processes in the active set. Whenever a runnable process in an active set blocks, we remove the process from the active set and distribute its shares equally among the remaining runnable processes. When a process wakes up and transitions in the ready-to-run state, we add it to the active set and rebalance the shares by dividing the total shares for that active set by n, where n is the number of runnable processes in the active set. In this way, we maintain the active set invariant at all times.

The Linux kernel does not support floating point arithmetic. Therefore, all operations in the kernel must be implemented using integer arithmetic. This issue led to some implementation challenges. If an active set contains more processes than the cumulative share value for that set, then some processes in that set would be assigned zero shares if integer division is used. However, a share value of zero implies that the process is not allowed to consume any CPU. Therefore, a process with zero shares would never be scheduled, which could lead to starvation. To prevent this scenario, we *scale* the shares for all active sets in the kernel by a factor of 10 if the number of processes in an active set exceeds its current cumulative share assignment. Scaling the shares requires removing all processes belonging to all active sets from the runqueue and enqueuing them back with their scaled share value. Scaling shares is an expensive operation with a time complexity of O(m*n), where m is the number of runnable processes in each active set and n is the total number of active sets in the system. However, our default weight to share assignments mapping is 1:100. A typical active set does not contain more than 50 processes. Hence the scaling of shares is not triggered often.

When the share value is not a multiple of the number of runnable processes, the remaining r shares are distributed among the first r processes on the list of active tasks for that set.

The 'no-split' policy. In scenarios where a process within a consumer set executes frequently for a fraction of its timeslice before blocking, the active set may fail to utilize CPU

in proportion to its share assignment. For example, imagine one consumer containing a CPU bound program such as an infinite loop, and a second, competing consumer set containing a pair of producer and consumer processes communicating through a bounded buffer such as a Unix pipe. For instance, in the shell command 'random | sort', a random number generator program ('random') act as a data producer and the Unix 'sort' utility which sorts the random numbers acts as a data consumer. Both of these consumer sets are assigned 100 shares each. Individually, the random number generator program and the sort program are CPU bound processes. The three possible share distributions, based on which process is running and which one is blocked, are shown in table 3.1.

Table 3.1: Share distribution for loop and random | sort

loop running, random running, sort blocked	100, 100, 0
loop running, random blocked, sort running	100, 0, 100
loop running, random running, sort running	100, 50, 50

We observed that the share distribution among the consumer sets was 50% each (50% for loop and 50% for random|sort), when *only one process* among random and sort was runnable. However, when both the processes (random and sort) in the consumer set became runnable, the CPU distribution would change to approximately 66:33 (66% for the loop and 33% for random|sort), instead of the expected 50:50.

This anomaly can be explained as follows: The random number generator blocks when the pipe is full. The sort program blocks when the pipe is empty. When all three processes are runnable, the VTRR runqueue has three processes loop, random, sort in that order. The random number program runs for a very small time, close to a jiffy [1], fills up the pipe with the generated random numbers and hence does not utilize its 40ms timeslice completely.

[1] A jiffy is the time period defined by the HZ variable in the Linux kernel and is equal to one tick of the computer's system clock

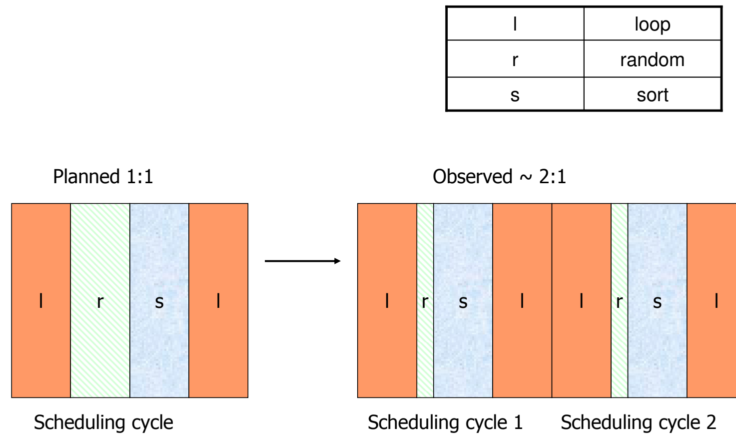

Figure 3.3: Pipe scenario under equal shares policy

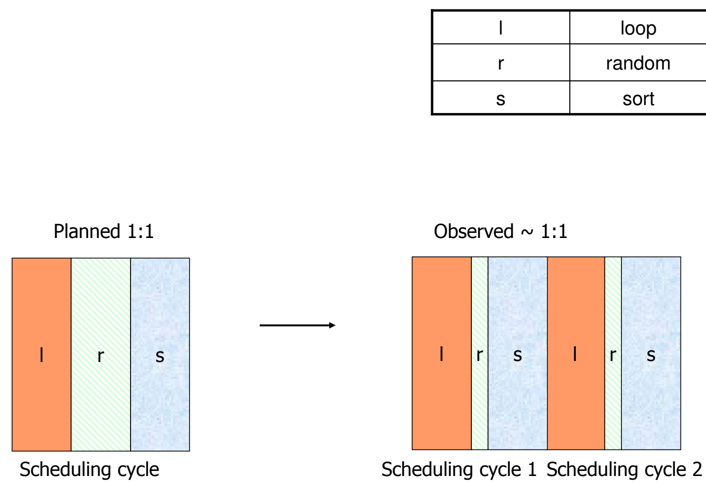

Figure 3.4: Pipe scenario under no-split policy

The sort program repeatedly reads random numbers from the pipe, sorts them, and writes the number read to a temporary file, which is later merge-sorted. These actions cause sort to use up its entire timeslice. In VTRR, the number of timeslices a process receives in one scheduling cycle is proportional to its share value. Hence, the loop having 100 shares executes twice in each scheduling cycle compared to sort having only 50 shares. The resulting VTRR schedule is shown in figure 3.3.

We built the no-split vtrr_policy to handle this case. Consumer sets using this policy do not split shares on a fork. Rather, we clone the parent process shares for the child process. Likewise, shares are not rebalanced on a sleep or a wakeup event. Hence, for the scenario with loop in one consumer set and random and sort in another consumer set, loop, random and sort are assigned 100 shares each. We show the resulting VTRR schedule with this policy in figure 3.4. As shown in the figure, loop being CPU-bound always utilizes its timeslice. Random does not utilize its timeslice and executes for a jiffy. However, sort now has 100 shares at par with loop. Hence, in one scheduling cycle, sort will receive exactly same number of timeslices as loop and execute as many times as loop resulting in a 1:1 split between the two consumer sets. The no-split policy uses the fact that a 100:1:100 share distribution is closer to a 1:1 CPU split than the 100:1:50 distribution which maps to a 2:1 CPU split under the equal shares policy. The no-split policy should only be applied if the CPU time consumed by a process such as random before blocking is small. If applied in this way, this policy does not compromise proportional fairness. Other consumer sets would only see a minimal change in their CPU consumption.

3.3 Data Structures

Figure 3.5 shows active sets implemented as a circular doubly linked list in the kernel. We used the Linux kernel's linked list libraries to handle insertion and deletion in the list. Each

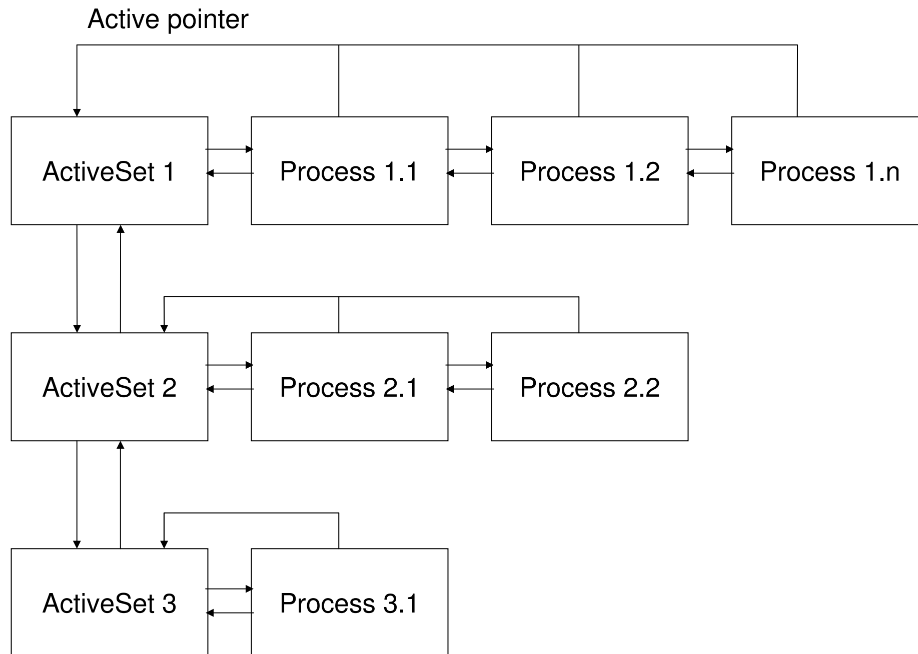

Figure 3.5: Active sets representation

active set contains a linked list of processes. Each process on the active set list has a pointer (called active pointer) to the active set to which it belongs. Whenever a new set is created, it is added to this list. The active set data structure contains the following fields:

- setId - the kernel assigns a unique setId to each active set when it is created.
- active_shares - this field stores the share value for this active set.
- set_count - maintains a count of the active (runnable) processes in this set, which is used during share distribution.
- activelisthead - is a list_head pointer to the list of processes belonging to this set.

3.4 Algorithms

In this section, we discuss the functionality related to active sets. The operations for a set are:

1. Create a set.
2. Join a set.
3. Change set weight.
4. Remove set.
5. Display set.
6. Rebalance shares within a set.

1. **Create a Set**

   A new set is created when the administrator wants to setup the system for a consumer. For instance, if the administrator modifies the login program for users on the system, the set is created when a user logs in for the first time. The steps for set creation are as follows:

   1. Get the pid of the current process.

   2. The administrator specifies default consumer weight information in a config file. Read the weight config file, if it exists.

   3. If yes, create a set using the weight information.

   4. If no, create a set using default weight values.

   5. Assign a unique setId to the set.

   6. Add the current process as a member of this set.

   7. Return the setId.

   Algorithm: CreateSet

2. **Join a set**

   This algorithm can be called from two different code paths:

   - When a process in a consumer set creates a new process, the child process is added to the same set using the following steps:
     1. Get the process control block (PCB) of the parent process.
     2. Clone the active pointer (shown in figure 3.5) of the parent for the child.
     3. Add the child process to the active set linked list.
   - An administrator explicitly adds a process to an existing consumer set, for example, the framework adds all login shells belonging to the same user to the consumer set that was created on the first login by that user. The algorithm is as follows:
     1. Find the setId corresponding to this consumerId.
     2. If setId not found, return error.
     3. Find the setId data structure from the linked list of consumer sets.
     4. Add this process to the linked list of processes in this consumer set.
     5. Rebalance shares within the consumer set.
     6. Return success.

   Algorithm: JoinSet

3. **Change active set weight**

   To accommodate changing system loads, the administrator may be required to change a consumer set's weight, either by increasing it or decreasing it to suit current demands. The algorithm described below handles the weight change as it affects the consumer set's active set. The algorithm is passed the consumerId of the set and the new weight to be assigned. The algorithm is as follows:

   1. Find the setId corresponding to this consumerId.
   2. If setId not found, return error.

3. Map the weight value assigned by the administrator to a share value for the active set in the kernel.

4. Distribute the new shares equally among all the runnable processes in the set.

5. Return success.

Algorithm: SetWeight

4. **Remove a Set**

The administrator can remove a set from the system using the remove set option. This option could be implemented either by terminating all processes associated with the set or by changing the scheduling policy of these processes to something other than VTRR. We chose to terminate the processes associated with this consumer set. We maintain a reference count for the active set. The count is incremented when a process is added to the set and decremented each time a process in the set exits. The algorithm takes the consumerId as a parameter.

1. Find the setId corresponding to this consumerId.

2. If setId not found, return error.

3. Send the KILL signal to all processes in the active set or change the processes scheduling policy.

4. Free the active set data structure.

5. Return success.

Algorithm: RemoveSet

5. **Display Set**

The administrator can view the current status of an active set in the system using the display set option. The algorithm needs the consumerId to display information for a particular set.

1. Find the setId corresponding to this consumerId.

2. If setId not found, return error.

3. Find the setId from the linked list of active sets.

4. Display share value, pid, process count, user CPU time (utime) and system CPU time (stime) for this set.

5. Return success.

Algorithm: DisplaySet

Share Adjustment. The module adjust_active_shares implements the share distribution. It is invoked whenever a process executes a blocking system call, thereby triggering a share adjustment. In the Linux scheduler code, the task wakeup of the blocked process and subsequent enqueue operation to the runqueue is handled by the function activate_task. The deactivate_task implements the reverse functionality, i.e., putting a task to sleep by removing it from the runqueue.

A newly forked process needs to be added to the parent's active set. The process addition requires a change of shares for all processes in that set. The sched_fork() function sets the process state for a newly forked process. We placed a hook in this function to trigger our share adjustment code.

To port our design to another platform, a developer would need to identify the routines that implement the functionality equivalent to activate_task, deactivate_task and sched_fork in Linux. The share adjustment algorithm takes the current task's process control block and the state of that process. The share adjustment algorithm contains the following steps:

1. Find the setId to which this process belongs.
2. If the process is newly forked, increment reference count for this set.
3. If the reference count equals the share value of this set, trigger scaling of shares.
4. If process state is ready-to-run, add this process to the active list.
5. If state is any of the blocking states, delete the process from the active list.
6. If process has exited, decrement reference count.

7. If reference count drops to zero, free the active set's data structure.

8. Count the active processes in this set.

9. Distribute the shares equally among all the active processes. Distribute any left-over shares to the first r processes, where r is the number of shares remaining after division.

10. Since the shares for all processes in this active set may have changed, remove each process belonging to this active set from the runqueue, set its new share value and add it back to the runqueue.

Algorithm: Adjust Shares

Scaling Shares. Scaling of shares is triggered in the kernel when the active set count becomes greater than the shares assigned for the set. The algorithm for scaling shares is as follows:

1. Multiply the shares of the set by a factor of 10, shares = shares * 10 for each process in the active list.

2. Dequeue each process from the runqueue and enqueue it based on its new, scaled share value.

3. Repeat steps 1 and 2 for all the active sets in the system.

Algorithm: Scale Shares

3.4.1 Algorithm Complexity

In this section, we discuss the time complexity analysis of our algorithms. We use 'm' to denote the number of processes in each active set, 'n' to denote the total number of active sets in the system, and 'k' to denote the number of processes on the VTRR runqueue.

1. Create Set - The algorithm creates a new set and adds it to the linked list. It takes constant time O(1).

2. Join Set - This algorithm can be invoked in two ways:

   (a) The parent process forks a child - in this case we already have a reference to the parent process's active set. We need to simply clone it which takes constant time O(1).

   (b) If the administrator explicitly adds a process to an existing active set - then we need to search for the correct setId. Although the active sets could be arranged in a heap or stored in a sorted list, in which case the complexity of search would be O(log n), our implementation uses Linux's linked list. We require linear search, which takes O(n) time.

3. Display Set - We need to search for the appropriate active set to display its information. We could use a heap or sorted list to achieve search in O(log n) time. Our implementation needs to linearly search over all active sets. Hence it takes O(n) time.

4. Change Weight - The algorithm searches for the correct set based on its setId and reassigns the shares among its processes. Hence it takes O(log n) + O(m) time. Our implementation needs O(n) + O(m) due to its linear search.

5. Adjust Shares - This procedure is invoked when a process within the active set blocks or becomes runnable. The algorithm examines the tasks belonging to this set. It dequeues each runnable task from the runqueue, sets its shares and enqueues it back on the runqueue. Finding the active set takes constant time as it merely involves looking at the active pointer in the PCB. The dequeue operation and set shares is constant time. The enqueue operation takes O(k) time. Hence the total time taken is O(m * k). The enqueue operation could be optimized to be O(log k) by using binary search in the sorted runqueue and hence leading to a total time complexity of O(m * log k).

3.5 Framework Portability

Our framework design is easily portable to other platforms as follows:

1. Proportional share scheduler: We have used VTRR as a proportional share algorithm for our work. We have not changed the VTRR algorithm in any way. The COPS framework does not require any changes to consumers, hence any other proportional share scheme could be used to achieve the same goals.

2. Platform: One could implement the design of active sets and add hooks to the process creation, process blocking and wakeup calls on any platform other than Linux to enjoy the benefits of consumer oriented proportional sharing.

Chapter 4

Framework Components

4.1 Overview

The components of our framework include the share management layer, the kernel components such as the active set and VTRR scheduler implementations and the administrative tools to configure consumer sets. We describe the development tools used for the implementation in section 4.4. Figure 4.1 shows how the different components of our system fit together.

The share management layer consists of the following:

1. *COPS daemon*: This program is the heart of the share management layer. It is the central point from which the administrator can interact with the kernel. It translates the user requests for creating sets or changing weights into system calls for the kernel. It maintains information on the consumer sets in the system.

2. *COPS client*: This program is a command line utility. Administrators can also use this program instead of the Web interface to configure consumers and weights. It is

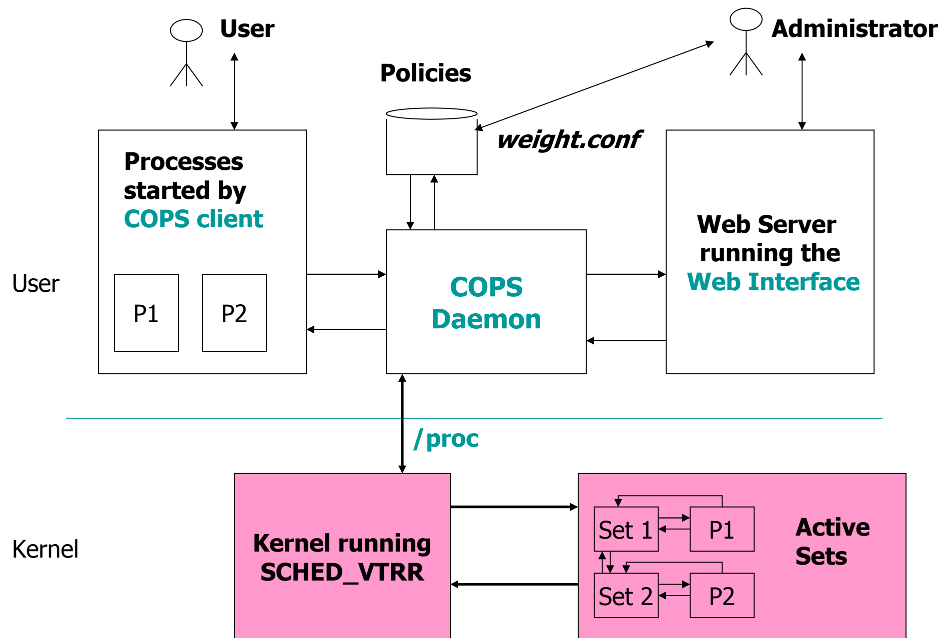

Figure 4.1: System components

also used as a driver program to start new applications.

3. *Web interface*: A simple web interface helps the administrator in configuring the system and managing the system at runtime.

4. *Share allocation policies*: The COPS daemon uses the administrator defined share allocation policies from the weight config file for setting default share values.

The components in the kernel consists of:

1. VTRR Scheduler: We ported the VTRR scheduler implementation from [NVZ01] from the 2.4 Linux kernel to the 2.6 Linux kernel.

2. Active Sets Implementation: The active sets implementation maintains the active set invariant.

3. */proc* interface: This interface displays information about active sets.

The implementation of these components is described in subsequent sections.

4.2 VTRR Scheduler Port

[NVZ01] implemented VTRR for the Linux 2.4 kernel. There have been major revisions to the kernel since then, especially related to scheduling. The 2.6 kernel series introduced the O(1) scheduler, as discussed in Section 2.3. To test our design in a state-of-the-art operating system, it was necessary to port VTRR to the current 2.6 kernel series. We ported the 2.4 VTRR implementation to Linux version 2.6.14.1 and integrated it with the 2.6 scheduler.

Integration with O(1) scheduler

The majority of our kernel implementation (about 500 lines) is restricted to the kernel file kernel/sched.c. We added a small amount of code to the header file include/linux/sched.h. We made the following changes to integrate our code with the existing scheduler:

1. Removed mapping of share values to nice

   In the VTRR implementation for the 2.4 kernel, the share values directly mapped to the nice value range using a macro:

   $$\text{SHARE} = (20 \text{ - } (\text{nice}))$$

   This scheme allowed the use of the existing nice(2) system call, but limited the granularity with which shares can be set. Every runnable process would have to be given

a minimum share value of 1 in order to be scheduled by VTRR. The maximum share value would be 40. Thus, the relative share proportion of two processes could be no higher than 1:40, which would prevent scaling of shares as described in Section 3.4. Instead of reusing the nice field, we added a separate share field to the task_struct. This method overcomes the disadvantages of nice described in Chapter 3 and supports finer granularity of share assignments.

2. A new scheduling policy class SCHED_VTRR

   The current scheduler has three policies called SCHED_OTHER, SCHED_FIFO and SCHED_RR as described in Chapter 2. We added a new policy called SCHED_VTRR for tasks scheduled by the VTRR algorithm.

3. Separate VTRR runqueue

   We want the administrator to control which processes are scheduled by the VTRR policy, and which processes should be subjected to the existing other policies. We added the functions vtrr_enqueue and vtrr_dequeue to support enqueue and dequeue operations on the VTRR queue.

4. Coexistence between SCHED_VTRR and other scheduling policies

   One of the design considerations was when to schedule VTRR tasks. We initially tried to assign a default share value to all processes and schedule all processes using the VTRR policy. But the resulting system took much longer to start up than when booted using the existing Linux scheduler. This behavior was observed because with an equal share assignment, kernel threads that would otherwise run at a lower priority in the normal scheduler, would now progress at a uniform rate with respect to other threads in the VTRR system. We did not want the COPS framework to affect the essential system processes, daemons, kernel threads started at bootup. Therefore, we decided to schedule boot time system services using the SCHED_OTHER policy and

use the VTRR scheduling policy only if there were no other runnable processes in the SCHED_OTHER policy. The schedule() function was modified such that VTRR tasks are scheduled only if there are no processes in the SCHED_OTHER runqueue. If there are no VTRR tasks as well, then the idle process is scheduled.

5. Creating Active Sets

   We modified the login shell for all users in the system, except the superuser. When a user logs on with his credentials, the system contacts the COPS daemon, which then creates a new consumer set for this user with a default weight value. If a consumer set for this user already exists due to a previous login, then the login shell process is added to that set. Therefore, a user cannot exceed his share assignment simply by logging in using multiple terminals. Each process forked by the shell becomes a member of the active set for this user.

4.2.1 Data Structures

The per process task_struct data structure was modified to incorporate the VTRR scheduler specific details and the active set information, as described below.

1. VTRR information

   - vft - maintains the process's virtual finishing time.
   - stride - the stride is the process share / sum_of_shares.
   - vtrr_counter - records the current counter of the process. The counter is used by VTRR in the time counter invariant.
   - share - stores the process's share value.

2. Active Set information

- active_ptr - it points to the active set this task belongs to.
- active_tasks - it is a pointer to the list_head of active tasks to which this process belongs.
- vtrr_policy - the value in this field determines if the no-split policy is in effect for this active set or whether it is subject to the equal shares policy.

4.2.2 /proc Interface

Statistical information about active sets is exported to user tools through a /proc interface. We modified the fs/proc/array.c and fs/proc/base.c to provide the active set display information. We added a directory called *activesets* under /proc. The file entries in this directory correspond to the active set identifiers in the kernel. For example, /proc/activesets/995 indicates the active set 995 created by the kernel. Entries are created when a new active set is created using the login shell or explicitly using the web interface. Each active set shows the pid of the currently running processes in that set, the share allotment for that set and the total count of processes in the set. The entry disappears when the active set is removed explicitly by a command or when the last process in that set exits.

In most scenarios, the administrator sets the relative weights using the Web interface and the COPS daemon and the kernel handles the final mapping from weights to shares. If needed, an administrator can directly control share assignment to different sets using the /proc interface. This setting overrides the values set by the COPS daemon.

4.3 User Space Tools

The command-line tool cops_client and a service cops_daemon interact with active sets in the kernel. In addition, a web-based interface simplifies the assignment of weights to

consumer sets.

4.3.1 COPS Daemon

The COPS daemon is a user space process started during system bootup. Each consumer set has a consumerId, which is a administrator defined name for that consumer set. The daemon reads from the file /etc/weight.conf which contains all currently configured consumerId to weights mappings. It creates the share table (figure 4.1) during initialization. The daemon can be contacted using a socket-based interface from the COPS client or from the web server application providing the web interface to COPS. The daemon maintains the status of all consumer sets and handles the mapping from consumer weights to consumer shares. The administrator specifies relative weight values for consumerIds. For example, if user A needs twice the CPU time of user B, the administrator would set the weights as 2:1 (user A : user B).

An initial scaling factor of 1:100 is used to map the weight value to a share value. This initial scaling factor is chosen to avoid triggering the scale shares algorithm as described in Chapter 3, unless the number of processes in an active set exceeds 100. New consumer sets can be created at runtime. Table 4.1 shows sample entries of records maintained by the COPS daemon. A setId of -1 indicates that the set is not yet created, whereas a positive setId indicates the actual setId.

Table 4.1: Share table

ConsumerId	Weight	SetId	Shares
adeodhar	1	995	100
Apache	2	996	200
guest	1	-1	100

We developed a socket-based protocol to pass commands such as 'join set', 'create set', etc. and their parameters to the daemon. Depending on the command, the daemon invokes one of the following system calls, which we added to the kernel to support COPS:

1. *joinset* - to add a process to a consumer set.
2. *changesetweight* - to change the weight value for a consumer set.
3. *displayset* - to display information for a consumer set.

The kernel executes that system call and reports the status of the system call back to the daemon. The kernel updates the consumer set information that is visible in the /proc interface. The daemon then reads the information from the /proc interface and updates its share table to maintain the up-to-date status of all consumer sets. The daemon has the ability to commit the share table changes to disk using the 'commit' command, which writes the columns consumerId and weight to the file /etc/weight.conf.

4.3.2 COPS Client

The COPS client is the driver program used to start new consumers. It communicates with the COPS daemon using a socket interface.

1. Its purpose is to run a given application under a specific consumerId. It can also add applications to an existing consumer set. We used a common Unix technique called prefixing, similar to the Unix 'time' command. For example, the following command would add a new Apache web server to the consumer set using the current user's consumerId:

   ```
   cops_client -e /usr/sbin/httpd
   ```

The COPS client program uses the environment variable USER to find the username, which is used as the default consumerId. We can also explicitly specify a consumerId as follows:

```
cops_client -c Apache -e /usr/sbin/httpd
```

This command starts Apache under the consumerId 'Apache'. The newly created active set is visible in the Web interface as well. The administrator can also add a specific process to an active set using the "join set" command from the Web interface.

2. The COPS client also acts as a wrapper program for the login shell. We modified the file /etc/passwd to replace the default user shell by our wrapper shell script, which invokes the cops_client program. The cops_client program changes the scheduling policy for this process to SCHED_VTRR, adds the shell to a consumer set, sets its shares and executes a login shell, which inherits the settings created by cops_client. This activity is completely transparent to the user logging on.

4.3.3 Web Interface

We have developed a simple Web interface to interact with the daemon. Figure 4.2 shows a screenshot of the web interface. It is written in PHP and communicates with the COPS daemon using sockets. The administrator can perform the following tasks using the interface:

1. Create consumerIds
2. Display active sets
3. Change weight assignments of sets
4. Remove an active set

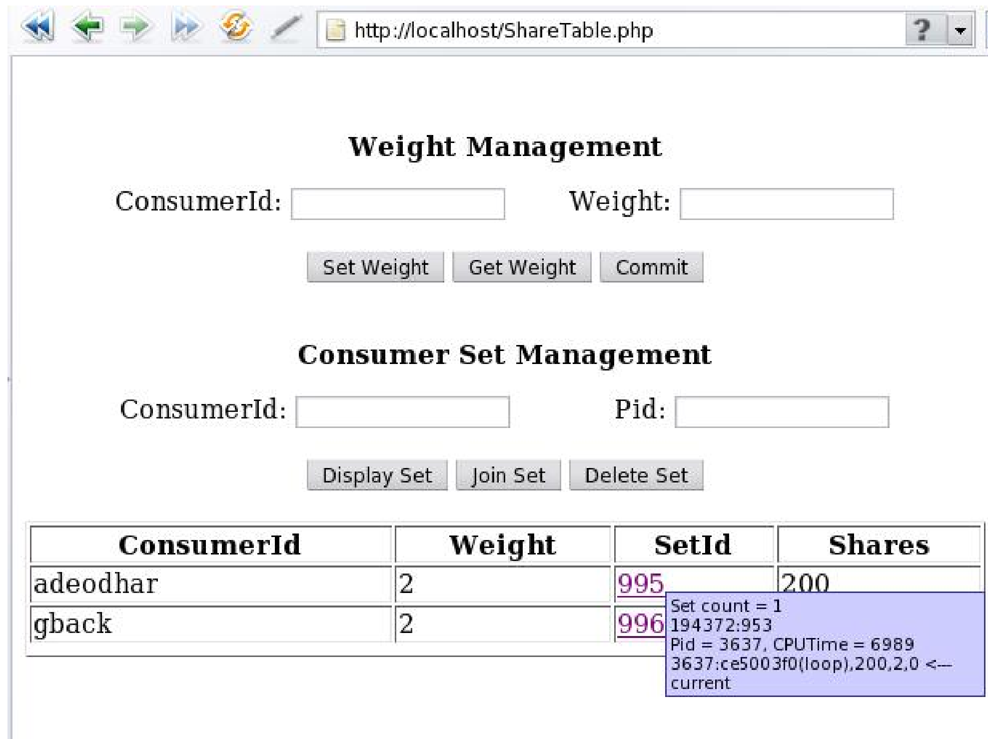

Figure 4.2: Web interface screenshot

5. Commit the share table to disk

We used the overLib library [Bos05] to display the details of an active set as a tooltip, including:

- Pids of processes in the active set
- Count of total processes in this set
- Cumulative share value of the set

4.4 Development Tools

We used a number of tools during our implementation.

Use of the VMware virtual machine emulator. We used the VMware Workstation virtual machine emulator [VMw98] for development purposes. Most of our development was related to the kernel scheduler code. We encountered frequent crashes, kernel panics, boot failures, etc. during code development. VMware enabled us to run different kernel versions, all residing on the same physical machine. It also simplified taking snapshots of a working kernel and incrementally adding features to the code.

Separate build machine. Because compiling the kernel code under VMware turned out to be time-consuming, we set up a fast build machine for building kernel images. The virtual machine's BIOS uses the network boot option to download the built kernel image from the build machine. This setup immensely helped in making quick changes to kernel, generating an image and testing the new kernel.

Command language interpreter. To test our implementation, we wanted to create test scenarios that mirror user activities. To create such test scenarios quickly, we implemented a command language interpreter. It is a perl script that parses scenario files. A scenario file contains a sequence of commands such as:

1. infiniteloop - to create a infinite loop
2. fork - to create a new process
3. sleep - to sleep for a specified number of seconds
4. exit - to terminate a process
5. pipe - to create a producer-consumer scenario connected by a Unix pipe
6. socket - to create a socket

The script internally calls the Unix specific system calls. For example, the short script shown below describes a fork-wait scenario that models a shell interaction. The parent forks a child process and waits for it to complete. The child process identified by *pid1* executes the scenario contained in the file *child-scenario.txt*.

```
fork $pid1 child-scenario.txt
wait $pid1
```

Chapter 5

Experimental Evaluation

5.1 Evalution Goals and Test Scenarios

Our primary goal for the experiments was to show that the service error is small for representative use scenarios. As defined in Section 2.5.1, the service error is the difference between the actual CPU time obtained by a process and the expected CPU time for that process. We extended the service error definition to consumer sets. We measure the degree to which the actual CPU utilization for a consumer set complies with the administrator defined weight.

We designed our tests to reflect common activities performed by a user. Our experiments included the following scenarios:

- Infinite loop.

  This scenario was used to model a CPU-intensive scientific computation.

- Fork - exit - wait/sleep.

  The Unix shell typically uses the fork-exit-wait model. The shell forks off a child process for executing a command and waits for the child to complete. This test models

user interaction with the shell.

- Kernel compilation.

  The compilation of any program is a typical activity on many systems. The compilation process is mostly CPU bound, except when the compiler reads and writes source and assembly files on disk or when the assembler and linker write and read object files during the compilation process. These phases will be I/O-bound. The kernel compilation was used to model a mix of CPU and I/O bound tasks.

- Pipes.

  Pipes are a common means for inter-process communication. Pipes can be used to model producer-consumer scenarios with a bounded buffer that causes frequent state changes. For example, a typical command would be ls *.txt | wc. We model the inter-process communication behavior on our system using this test.

- Apache Web server.

  The Apache web server uses a pre-forked thread model to handle HTTP connections. An Apache server could share the CPU with a scientific computational task on the same machine. We wanted to observe the CPU time obtained for each connection of the Web server in the presence of a competing CPU intensive consumer set, modeled by an infinite loop.

- Multimedia player.

  As mentioned in Chapter2, multimedia applications have strict latency requirements, which cannot be guaranteed by proportional share schedulers [SAWJ97]. Nevertheless, we wanted to observe the behavior of our share management implementation with respect to such applications. Hence, we included the XMMS audio player as a test scenario.

The above experimental scenarios demonstrate the result of share adjustments on CPU

bound tasks, a mix of CPU and I/O bound tasks, purely I/O bound and real world server workloads. The experiments also cover most of the blocking system calls in Linux that exercise COPS's share rebalancing algorithm.

5.2 Experimental Setup

The experimental setup consisted of a machine running Linux kernel version 2.6.14.1, modified to include our VTRR port and the COPS implementation. To obtain accurate scheduling measurements, we recompiled the kernel to use a system timer with 1000Hz frequency, giving a resolution of 1ms. All experiments were performed on a Intel Pentium D processor running at 3.00 Ghz with 2GB RAM. Since our algorithm is intended for uniprocessors, we disabled one of its cores. To minimize the CPU usage by processes not subject to COPS, we turned off system services which were not required. Nevertheless, these processes still use a fraction of the CPU. This fraction is marked as "others" in our experimental results.

Plotting Tools. To observe the CPU split between the consumer sets in real-time, we used a real time plot program called Kst [N+06]. We wrote a program based on the Linux top command to read the /proc interface and find the CPU usage of the top 10 processes in the system. This program writes its output to a file, which is used as the input to Kst. The plot shows the relative CPU usage of the processes in the system. This plot can be used to visually verify the actual CPU usage against the share assignment. If an administrator changes the share or weight values of the sets, the changes in the resulting CPU usage can be observed immediately. In addition to real-time plotting tools, we used the gnuplot program [WKL+96] to generate plots that show the CPU time used by a process (from /proc/<pid>/schedstat) over an interval of one second.

5.3 Experimental Results

5.3.1 Multiple Process Scenarios

We ported the VTRR patch from the 2.4 kernel to the 2.6 kernel. We wanted to test whether our port worked correctly on the 2.6 kernel, even when no consumer sets are involved. We wanted to check that per process share assignments are respected by VTRR. We ran an experiment with two infinite loop processes. Each loop was assigned 100 shares. The graph in figure 5.1 shows the two infinite loops. Since both the processes have equal shares, we

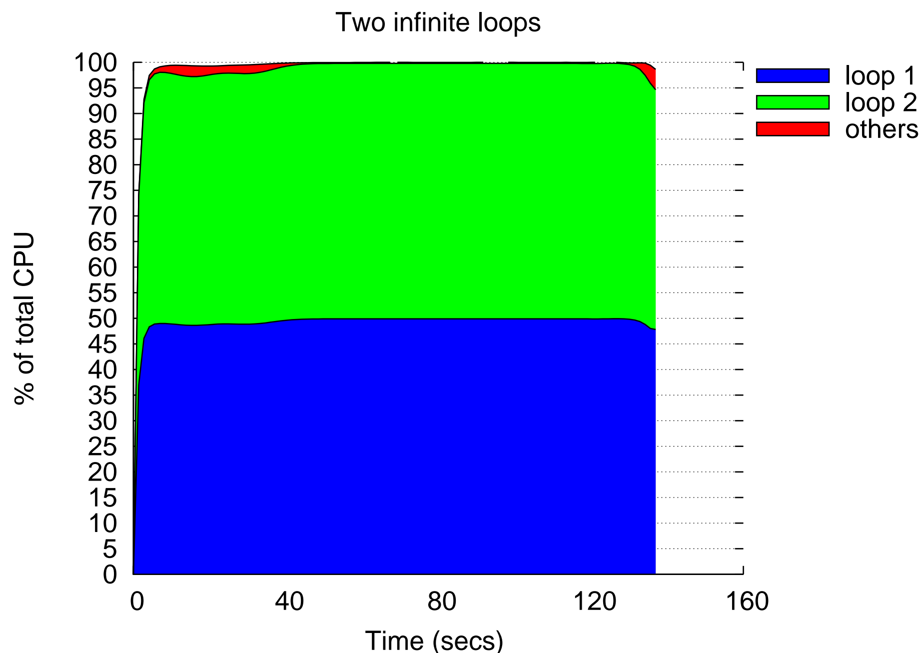

Figure 5.1: Infinite loop scenario: In this scenario, each loop receives exactly its proportional allocation. The experiment indicates that our VTRR port is working.

expect to see an equal CPU distribution for both processes. We obtained a CPU usage of 50:50, as expected. The CPU utilization plotter program, the service error plotter and other kernel daemons use some CPU capacity, which is marked as 'others' in the graph. The

service error was plotted for every second for the above experiment (figure 5.2). The service

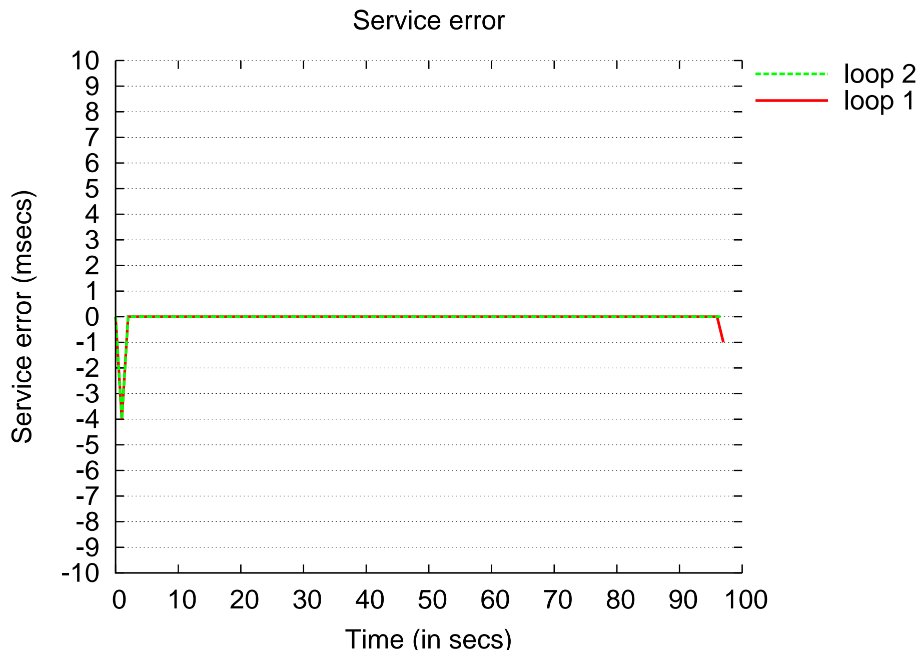

Figure 5.2: Service error for infinite loop scenario: Service Error is close to zero

error for both loops is zero for most of the considered time interval, indicating an exact correspondence of CPU time to assigned shares.

5.3.2 Multiple Consumer Set Scenarios

We present the results of our active set implementation for the scenarios discussed in Section 5.1. Each experiment was run with and without an external load. The external load consisted of an infinite loop process which attempts to continuously hog the CPU. For the experiments that included an external load, the loop formed one consumer set and the test scenario such as fork-wait-exit or the web server formed a competing consumer set. The weight assignment for both competing consumer sets was equal.

1. **Fork-exit-wait**

In this test scenario, a parent process executes a loop for 7 seconds, then forks off a child process. The parent and child process each execute a second loop for 14 seconds concurrently. The parent process then waits for the child to exit. After the child exits at t=45, the parent continues executing in an infinite loop for 20 seconds. We expect to see an equal split of CPU share between the parent and child after fork. The parent waiting for the child process should give up its share which will then be redistributed to the child process in the set. On exit, the shares should be returned back to the parent. Figure 5.3 shows the graph obtained when we run this test scenario. As seen

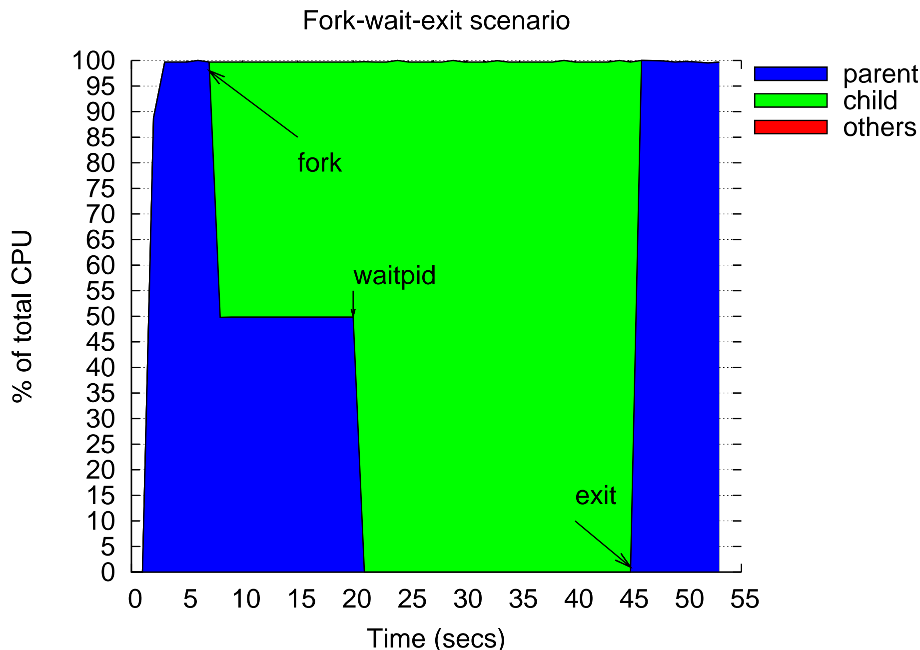

Figure 5.3: Fork-wait-exit scenario: In this scenario, we show how the shares are distributed equally on the fork call, transferred to child on the wait call and back to parent on the exit call. At t=7, the parent forks the child, both execute loop for 13 seconds; at t=20, the parent calls waitpid; at t=45, the child exits.

in the graph the parent process initially uses 100% of the CPU. After the fork, the CPU time for the parent drops to 50% and the child process uses the remaining 50%.

When the parent process waits, the child receives the parent's shares and hence uses 100% of the CPU. When the child exits, the parent reclaims its shares which increases its CPU utilization, back to 100%.

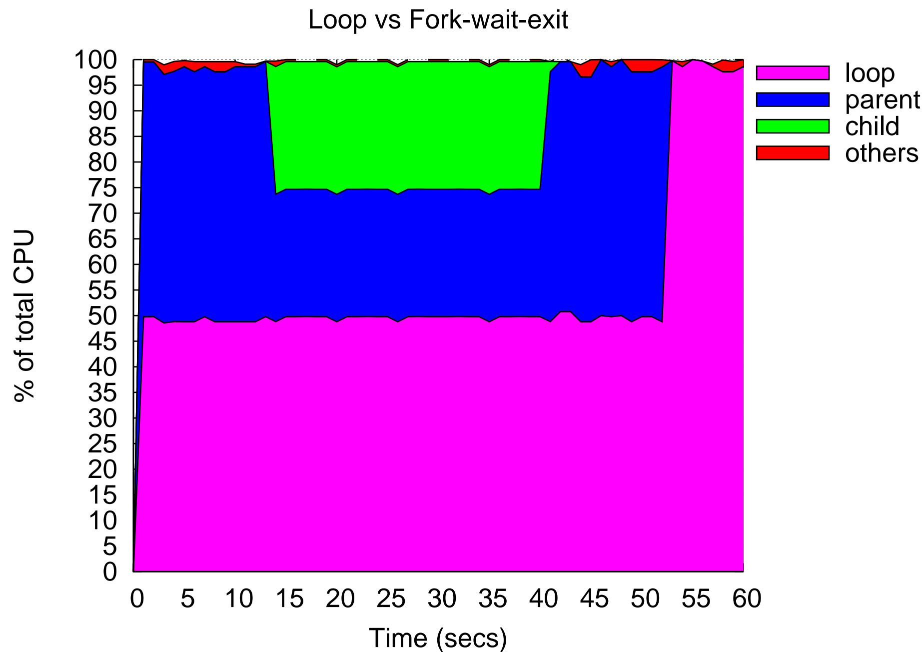

Figure 5.4: Loop vs fork-exit-wait: Both active sets receive shares in proportion to their weight.

2. **Fork-exit-wait with external load**

   We executed the same test as above, but with a competing external infinite loop in a separate consumer set. As seen from the graph (Figure 5.4), the loop starts running, taking up 50% of the CPU. The parent process from the fork-exit-wait test takes up the remaining 50%. As the child process is forked, the parent process's CPU usage drops. Thereafter, both consumers utilize 50% of the CPU, in accordance with their weight assignments as seen from the straight line at 50%, 75% and 100% in the graph. When the child process exits, the parent ramps up from 25% to 50%. Finally, the parent process from fork-exit-wait test exits, leaving no runnable process in that consumer

set. Since VTRR is work-conserving, the infinite loop process takes up the entire CPU.

The service error for this test is shown in figure 5.5. We see that the error is bounded in the range [-40,+40] ms. Each consumer set is expected to receive 500ms per second because of the equal weight assignment. The 40ms error value corresponds to a maximum error of 8% per second, which is relatively small.

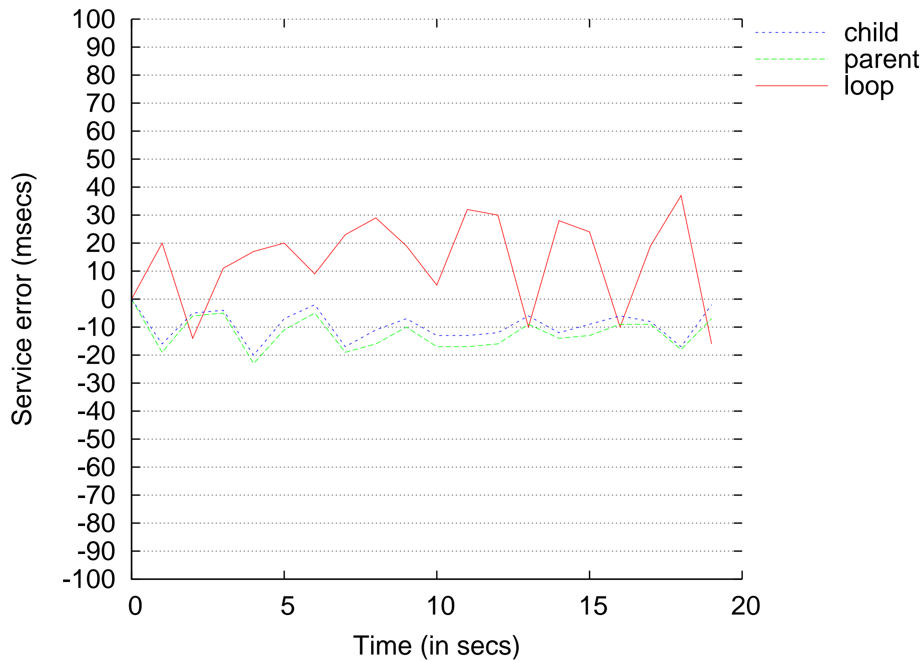

Figure 5.5: Fork-wait-exit with loop: The service error is about 30ms for loop and -20ms for fork-exit-wait, which is within reasonable limits.

3. **Kernel Compilation**

   In this experiment, we ran an external load in one consumer set and the kernel compilation job (make -j2) in other set. The graph obtained is shown in figure 5.6. The loop initially receives more CPU time. When the kernel compilation is started, the CPU usage drops for the loop. Both consumers receive approximately 50% of the CPU which is in accordance with the expected behavior.

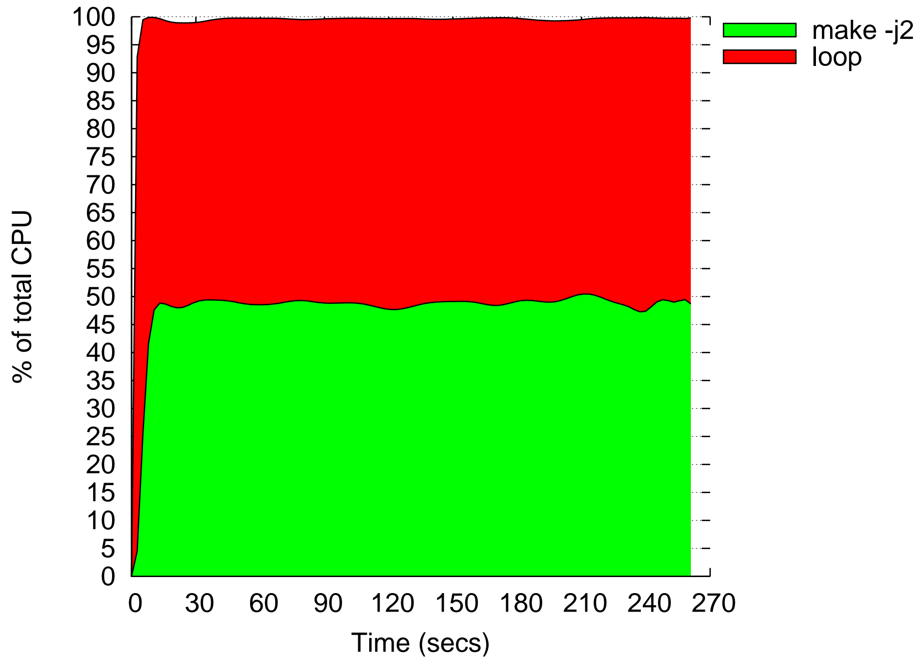

Figure 5.6: Kernel compilation: The scenario shows the loop in one set and two kernel compilation threads in another. Both sets use CPU based on their share.

4. **Apache Web Server and Jmeter**

   In this experiment, we placed the Apache web server in one consumer set and an external load in another set. Jmeter [Pro99] is a program used to test functional behavior and measure the performance of web applications under load. We used Jmeter (running on another machine) configured with 8 threads, representing 8 simultaneous users, to send requests to our web server. The HTTP request invoked a finite loop contained in a CGI script which hogged the CPU for more than 100 seconds, making it a CPU bound task. We expected to see equal CPU distribution among the two active sets. Also, each process was expected to receive equal share with respect to other processes in the same set. The results are shown in figure 5.7.

   The graph shows a 50:50 split in CPU usage between the two sets. Each httpd process receives an approximate CPU time of 6%, which can be inferred from the width of

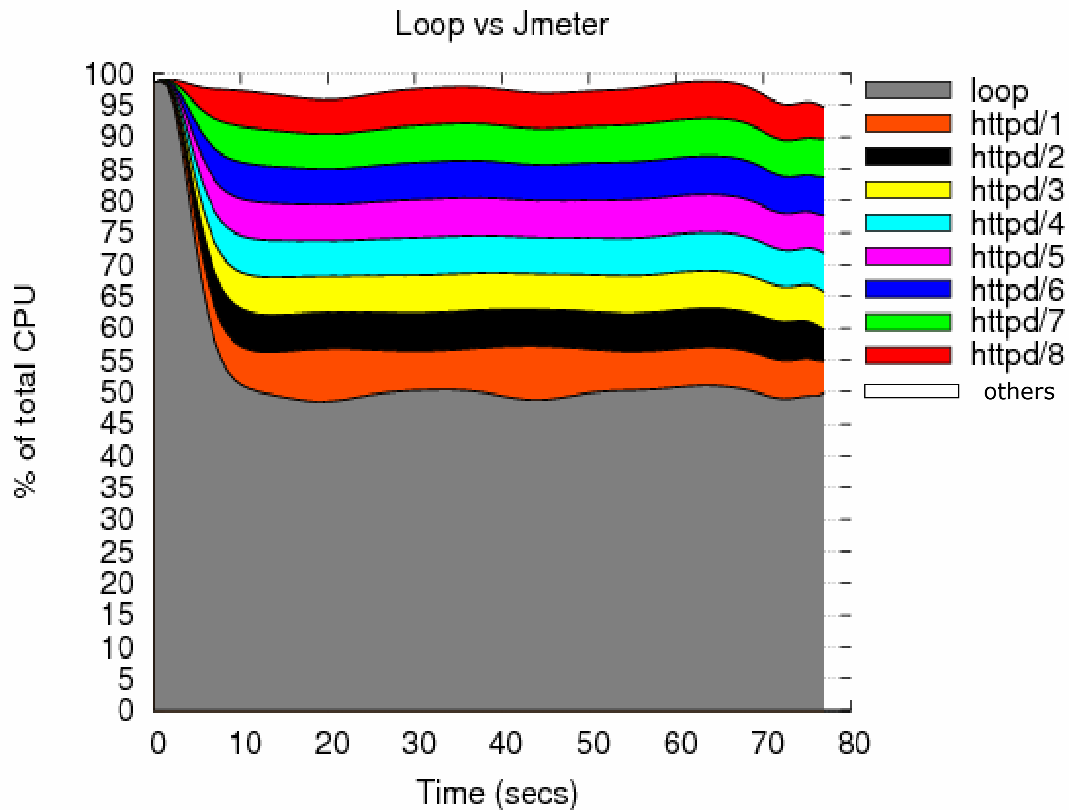

Figure 5.7: Multiple httpd instances: In this scenario, we show that the consumer set formed by the 8 Jmeter processes receives CPU in accordance with its share. The other consumer set (infinite loop) is not affected, despite of the larger number of processes in the competing set.

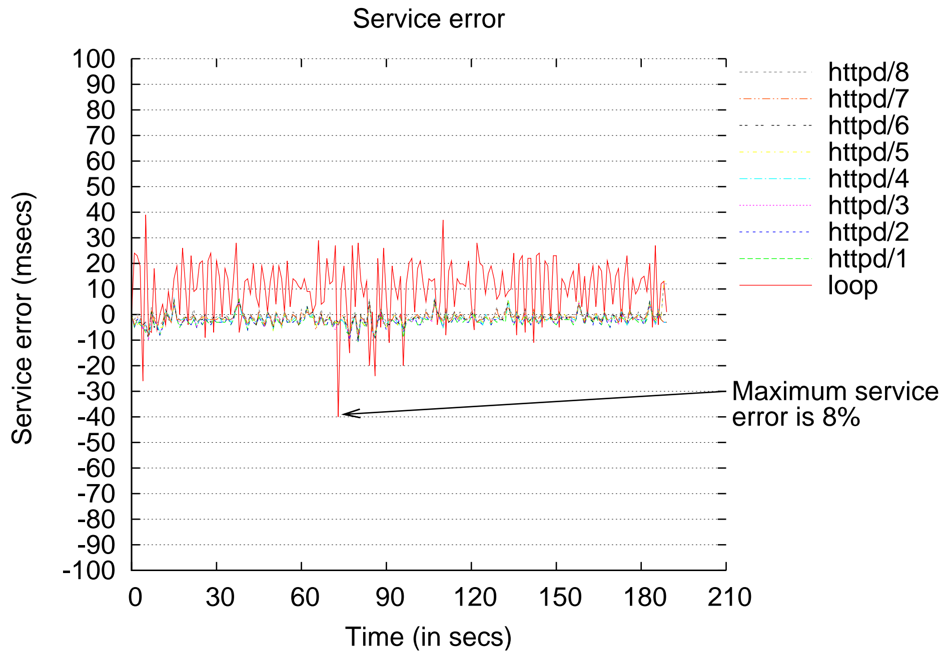

Figure 5.8: Service error for httpd scenario: The service error for loop is small. All httpd threads have almost zero error.

each colored band of httpd processes. The service error plot in figure 5.8 shows that the error for the httpd processes is close to zero at all times. For the loop, the error oscillates in the [-40, 40] ms range. The plot shows that the service error for the different active sets is within acceptable limits.

5. **Pipe case**

We experimented using a pipe scenario as follows.

a) Without external load. A random number generator script was used to write to the pipe. The Unix sort utility was used at the reading end of the pipe. These two processes formed a consumer set. When this experiment was run, we observed that the active set always fully utilized the CPU. Figure 5.9 shows this behavior.

b) With external load We observed an interesting phenomenon in this case. The graph we obtained is shown in figure 5.10. We observe that periodically, the CPU utilization

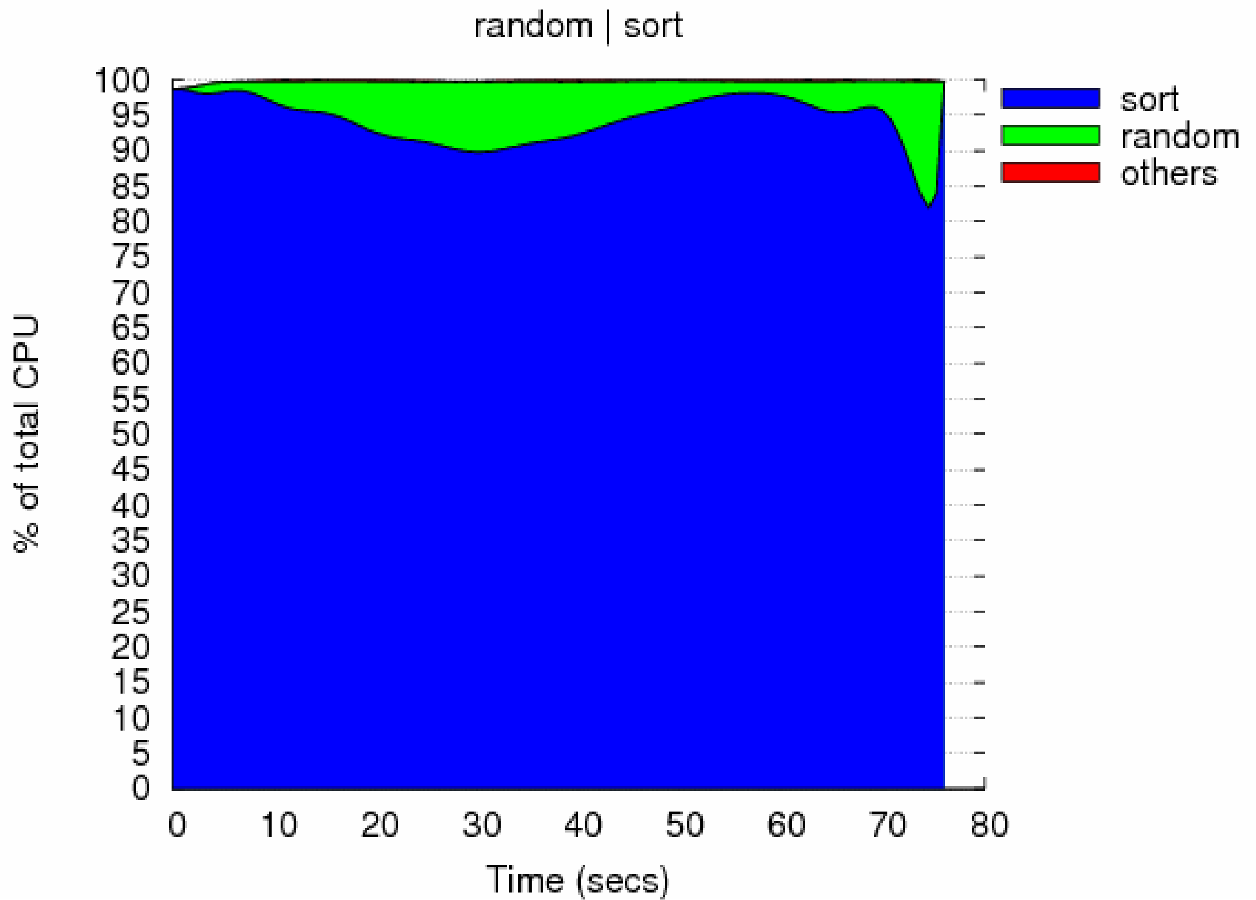

Figure 5.9: Pipe scenario A: In this scenario, two CPU bound tasks are connected by a pipe, forming one set. The active set fully utilizes the CPU at all times.

was more than 50% for the loop instead of the expected 50:50 split. This scenario motivated the no-split policy discussed in Chapter 3. We repeated the experiment using the no-split policy with share assignments of 100 each for loop, random and sort and obtained the graph shown in figure 5.11. Using this modified approach, we found a CPU distribution of 50:50 for loop and (random|sort).

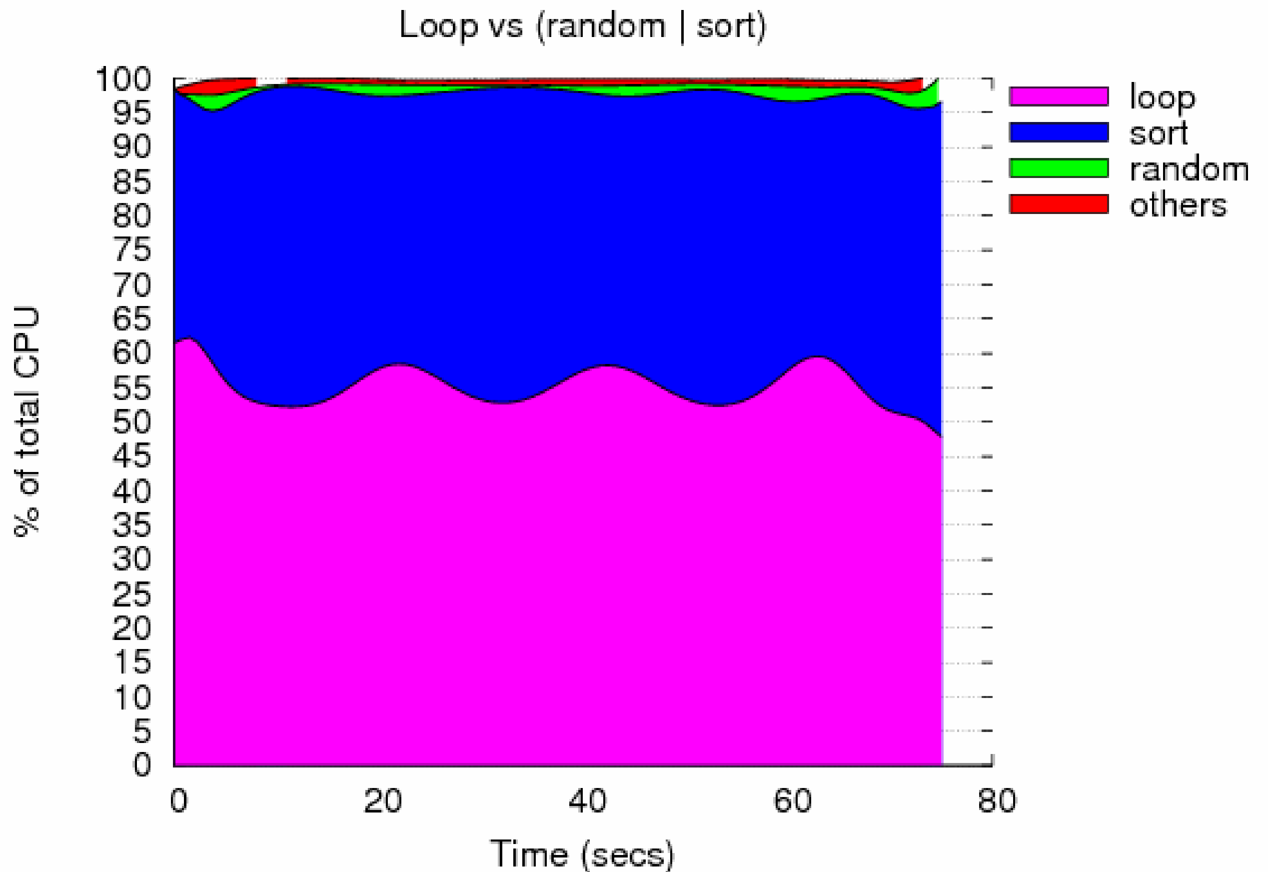

Figure 5.10: Pipe scenario B:In this scenario, two CPU bound tasks are connected by a pipe, forming one set. A loop runs as another consumer set. With the equal share distribution, loop receives more CPU than expected in every VTRR scheduling cycle compared to random and sort, because random frequently blocks before using up its timeslice.

6. **XMMS Multimedia Player**

   We ran the XMMS multimedia player against the loop in another set (figure 5.12). Because the loop is a heavily CPU bound task, it receives almost 100% of the CPU

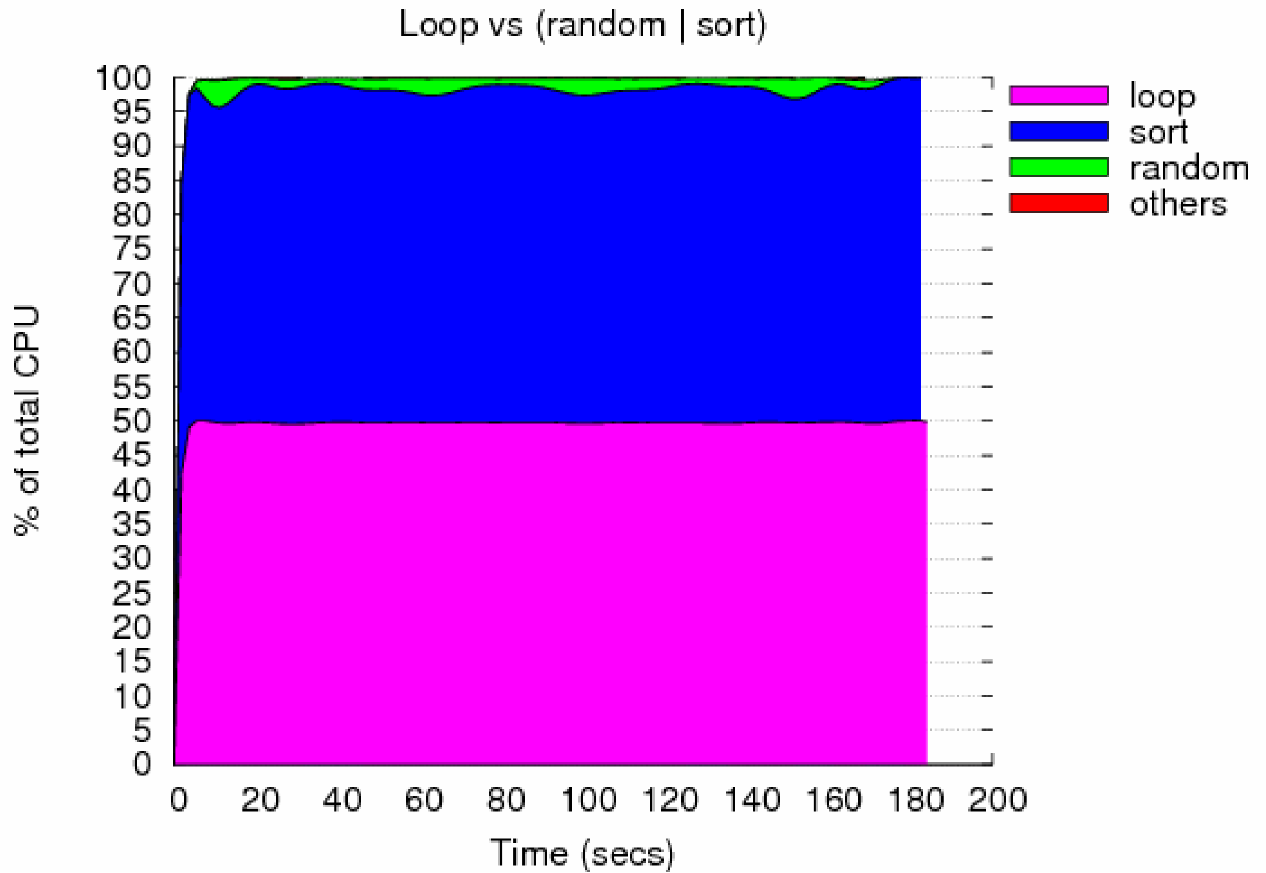

Figure 5.11: Pipe scenario C: The no-split policy works well for producer-consumer scenarios which unequal CPU consumption.

and the XMMS player receives almost none, resulting in cracked audio. If we assign significantly higher shares (twice as many shares as loop) to the player, we hear a continuous audio stream without cracks. When we assign higher shares to the XMMS player, it is inserted at the front of the VTRR runqueue whenever it becomes runnable, placing it ahead of all other tasks. It is immediately scheduled by VTRR, resulting in

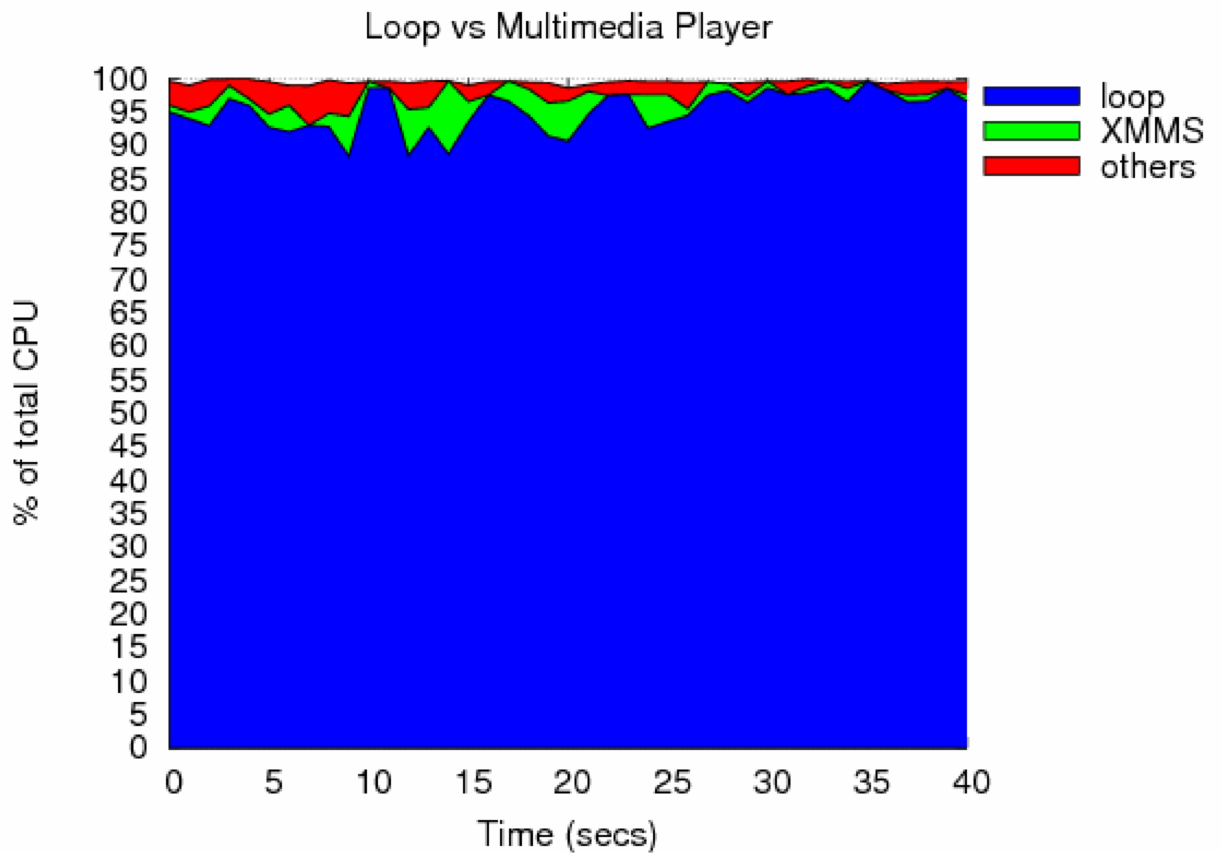

Figure 5.12: Multimedia player: The CPU bound loop takes up all the CPU. The I/O bound XMMS player does not use up its assigned share, but does not obtain control of the CPU within the required latency bounds, resulting in occasional cracked audio.

good audio quality without cracking noises.

5.4 Limitations

The COPS framework contains a number of limitations.

1. Our framework does not provide a way to limit CPU consumption of a consumer set. Resource limits may be useful in server farms to protect hosts from denial-of-service attacks. They could also be used to control resource consumption of poorly written or buggy applications. This drawback arises because of using the work-conserving VTRR algorithm, which could be overcome by using any other non-work conserving proportional share algorithm.

2. Our consumer set framework does not provide latency guarantees to multimedia applications. This limitation is due to the VTRR algorithm and could be solved by using an algorithm such as BVT [DC99] which gives dispatch preference to latency-sensitive applications.

3. The COPS framework groups processes either using the seed process approach or by administrator-defined groups. Thus COPS cannot automatically infer process dependency like SWAP [ZN04]. Our future work would explore this option.

4. Our framework does not support hierarchies of active sets.

Chapter 6

Related Work

Our work is related to three distinct areas:

1. Proportional share scheduling.
2. Resource management frameworks.
3. Resource management policies.

6.1 Proportional Share Schedulers

Lottery scheduling [WW94] is a randomized resource allocation algorithm proposed by Waldspurger et al. In lottery scheduling, resource rights are represented by a proportional number of tickets. The lottery scheduling algorithm randomly chooses a ticket and the process holding that ticket is scheduled. In this type of probabilistic scheduling, tasks may not exactly receive their assigned proportional allocation over all time intervals. The error, defined as the difference between the actual allocation and the expected allocation based on probability, decreases as the number of allocations increases. When a task blocks because of a depen-

dency on another task, ticket transfers are used to transfer tickets from the blocked task to the dependent task, which is similar to our share transfer mechanism. A group of mutually trusting tasks is allocated a currency which is funded by tickets. Currencies are used to prevent tasks from utilizing resources of other tasks not belonging to the trust group. Currencies and tickets are related by an exchange rate, similar to the mapping between weights and shares in COPS. Ticket inflation, a mechanism to create more tickets, is used to allow flexibility to assign tickets within mutually trusting tasks. If the number of tickets within a trust group increases due to ticket inflation, then the exchange rate between the currency and tickets changes and each ticket value decreases. Compensation tickets are allocated to tasks that use only a fraction of their resource allocation. This mechanism can be used for achieving dispatch preference for I/O bound tasks. The lottery scheduler has an algorithmic complexity of O(n), while our approach uses VTRR which achieves scheduling in constant time, O(1).

Borrowed Virtual Time [DC99] is an approach to building a process scheduler that supports widely varying applications. It uses the concept of effective virtual time (EVT) for scheduling processes. The name 'borrowed virtual time' comes from the concept of warping, where a process can borrow from future CPU allocations up to a warp limit to gain temporary dispatch preference. EVT is calculated by taking the difference between the actual virtual time of a process and its virtual time warp. Warping is used when scheduling latency sensitive applications such as multimedia players. Our active set approach could use the BVT algorithm for scheduling multimedia applications. The BVT scheduler has been used by the virtual machine monitor Xen [BDF+03].

Nieh et al introduced the Group-ratio round robin (GR3) scheduler [CNC05] to overcome limitations of VTRR. It is an O(1) proportional share scheduler that reduces the service error for skewed share assignment compared to VTRR. The GR3 algorithm uses a client/task grouping strategy based on similarity of weights. In our approach, we group tasks created

by an application into a consumer set. The scheduling decision is based on two levels: an intergroup scheduler is used to select a particular group of tasks and an intragroup scheduler selects a particular task within the group to run on the CPU.

1. The group weight is defined as the sum of shares of all tasks in the group. The groups are sorted from largest to smallest share values. In intergroup scheduling, a group is chosen based on the ratios of the group weights. Thus groups with higher shares are ahead on the runqueue.

2. The intragroup scheduling is done to select a task among all the tasks in that group. This decision is based on share value and execution history.

Our implementation could be modified to use GR3 instead of VTRR for reducing the service error for scenarios with highly skewed weights.

Recently, Caprita et al presented Grouped Distributed Queues (GDQ) [CNS06], an O(1) proportional share algorithm for multiprocessor scheduling. In GDQ, processes with similar share values are combined to form a group which is similar to the GR3 algorithm discussed above. The GDQ algorithm uses a different technique than the existing Linux scheduler to achieve load balancing among multiprocessors. Unlike the existing Linux scheduler where each processor is associated with a runqueue and processes migrate from one runqueue to another during load balancing, GDQ keeps the fixed groups of processes and distributes processors among these groups. The idea of Multiprocessor Fair Queuing (MFQ) is used to reassign processors from one group to another at regular time intervals. This method ensures that groups progress at a rate proportional to their share value. The COPS framework could be implemented on multiprocessor platforms using GDQ as the underlying proportional share algorithm.

6.2 Resource Management Frameworks

Resource management frameworks provide resource management control to applications and the flexibility to set management policies, based on application usage and behavior.

Sullivan [SS00, SHS99] describes a resource management framework based on ticket exchanges. The objective is to achieve performance benefits in spite of resource interdependencies among different applications. This approach is used for managing different types of resources including CPU, memory and disk. Ticket exchanges are similar to bartering of goods, wherein a resource principal such as a user or application may request resources from other principals. In return, it can give up some resources which it may not be using to full capacity. For example, an application that needs more memory than allocated to it may exchange its disk tickets for memory tickets from another application that requires a larger allocation of disk space. The ticket exchange is symbiotic; it can be canceled at any time. By comparison, our approach deals only with CPU scheduling. In ticket exchanges, there exists a central entity called the dealer to coordinate exchanges. We do not have such a broker because we manage CPU shares within a consumer set. Thus, if a process within an active set blocks, we initiate a share adjustment to donate its shares among the other threads belonging to the same consumer. In Sullivan's resource management framework, the application may have to be modified to include a negotiator on behalf of the application. The negotiator would monitor the statistics of its resource usage and contact the dealer to initiate ticket exchanges.

A new abstraction called resource containers[BDM99] was proposed by Banga et al to separate the notions of protection domains and resource principals. A resource container contains all system resources used by an application, including CPU allotments. Our approach of active sets achieves similar goals as the resource container abstraction. Although operating systems regard a process as a schedulable entity, we charge resource usage per-application

and not per-process. The resource container approach supports sharing of a process among different containers, for example, a process communicating by a socket may be multiplexed among multiple connections and hence executes using different resource containers at different times. The container approach requires modification to application programs.

A fairly recent approach called SWAP [ZN04] is to monitor system call history to track process dependencies. The goal is to develop a general purpose scheduler that improves system performance by monitoring process dependencies. Process dependencies occur when processes interact via interprocess communication (IPC) mechanisms such as pipes, sockets, or named pipes (FIFOs). These IPC mechanisms are accessed by system calls. The SWAP model defines resource providers as processes holding a resource and resource requesters as processes requesting a resource. In some cases, the system call history may not provide complete and accurate dependency information. To handle these cases, SWAP associates a confidence interval with the dependency information. A feedback based confidence evaluation is used to predict which process in the list of resource providers can provide the resource quickly to the resource requester. The confidence interval values are updated by observing the process blocking behavior. This dependency information is used to schedule virtually runnable processes, defined as processes that are blocked, but remain on the runqueue because they have at least one runnable resource provider. When a virtually runnable process is scheduled, instead of scheduling it, the resource provider with the highest confidence value associated with that virtually runnable process is selected to run. This dependency tracking approach results in shorter duration of blocking and hence improved system performance. In our approach, instead of using system wide process tracking, we use consumer sets to group the dependent processes.

The idea of tickets and currencies from lottery scheduling is extended by Waldspurger for building an object-oriented resource management framework [WW96]. The framework uses currencies for funding processes and ticket transfers for redistribution of resource rights. Our

idea of share transfer among consumer set processes is similar to this concept. We use the idea of consumer sets to group related processes. Waldspurger's ticket transfer approach on a process state change is similar to our share transfer within a consumer set.

6.3 Resource Management Policies

The Class-based Kernel Resource Management (CKRM) [FNS+04] is a framework to achieve differentiated service for resources such as CPU, memory, network. A class is defined as a dynamic grouping of OS objects of a particular type that is associated with workload management policies. In CKRM, the resource manager allocates the resources to different classes and subjects the system to a policy after system boot up. The policy consists of class definitions and classification rules. A newly created process is classified to a class based on the policy in effect. The service requirements for a class are specified in terms of higher-level policy goals, applied to the specific workload characteristics. The framework monitors the system usage of resources and verifies if it meets the higher level workload goals. It automatically adjusts the resource share of a class to fall in line with the goals. CKRM implements techniques such as weighted fair sharing (WFS) and a two-level scheduler which can be used with incremental modifications to the existing O(1) scheduler. The idea of classes in CKRM is similar to our active set concept. Both approaches try to move the scheduling from being per-process to per-application. Our share management framework uses share adjustments on blocking system calls within the active set to respect administrator defined weights. CKRM-managed resource managers such as CPU scheduler are class-aware. The existing runqueue structure has been changed in CKRM to use a separate runqueue per class. In our case, we have implemented the VTRR scheduler algorithm as a scheduling policy that coexists with the O(1) scheduler.

Petrou et al extended Waldspurger's lottery scheduling [WW94] to improve the responsive-

ness for interactive applications by using the concept of dynamic ticket adjustments [PMG99]. This method preferentially schedules processes holding kernel resources. They do not transfer tickets in blocking and unblocking functions because it is not known whom to transfer tickets to. Our approach, on the other hand, encapsulates a group of processes created by a consumer into a consumer set. Using the consumer set idea sidesteps the problem of monitoring the dependencies among processes. Thus, we can use the blocking and unblocking routines to trigger share adjustments.

Chapter 7

Future Work and Conclusions

7.1 Contributions and Conclusions

We successfully built the COPS framework consisting of a proportional share scheduler, VTRR, coexisting with the O(1) Linux scheduler. The COPS framework guarantees CPU usage to consumers based on administrator defined weights. We built user space tools such as the web interface for configuring consumers, the COPS client driver program to start applications under a consumer and the COPS daemon that maps weights to cumulative shares for each consumer. The COPS framework eases the administrator's job of share management. Our framework is portable to other platforms with minimal changes to kernel code. The contributions of this research are:

1. We show how consumer-oriented scheduling can be implemented by using the active set idea and share management policies combined with a proportional share scheduler. The COPS framework is easy to use, transparent to the user and portable to other platforms.

2. With our work, we have shown an approach by which proportional share schedulers could coexist with the traditional Linux schedulers. The research has the potential to move proportional share scheduling from the research domain into the domain of practical applications.

3. The experimental results show that for the scenarios we tested, the COPS framework can guarantee proportional fairness based on weight assignments.

4. We implemented a working VTRR version for the 2.6 Linux kernel, which is available as a kernel patch.

7.2 Future Work

Our research could be expanded in the following directions:

1. COPS framework for multiprocessors - Our COPS framework is limited to uniprocessor systems. COPS could be implemented on multiprocessor/dual-core systems. A multiprocessor proportional share algorithm such as GDQ could be used for this purpose. The multiprocessor implementation would need to handle issue such as load-balancing among processors and setting processor affinity for consumer sets.

2. Security in share management layer - We have implemented a prototype share management layer. If deployed as a product, it will be necessary to add security in terms of user privileges for different commands. For example, the Web interface to manage consumer set weights should be accessible only by the system administrator.

3. Consumer set membership - In our framework, a process remains in the current consumer set for its lifetime. We plan to explore approaches similar to Resource containers wherein a process's binding to a consumer set may change dynamically depending on

whose behalf it executes. We could also explore approaches similar to SWAP for automatic process dependency detection to group related processes in a consumer set.

4. Measuring share adjustment overhead - In our current implementation, shares are adjusted within a consumer set whenever a process blocks, unblocks, exits or a new process is added to the set. We plan to evaluate the overhead associated with share adjustment for active sets with a large number of runnable processes or consumer sets in which processes block and unblock frequently.

5. Additional policies - We currently support two share allocation policies within consumer sets - equal distribution to all processes and the no-split policy. We plan to investigate additional policies for consumer sets based on application behavior.

6. Code release - We plan to submit our implementation as a patch to the open-source community, gather feedback and see it become a part of the Linux kernel.

Bibliography

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, February 1999.

[Bos05] Erik Bosrup. overlib package, 2005.

[BZ96] Jon C. R. Bennett and Hui Zhang. Wf2q: Worst-case fair weighted fair queueing. In *INFOCOM (1)*, pages 120–128, 1996.

[CAGS00] Abhishek Chandra, Micah Adler, Pawan Goyal, and Prashant Shenoy. Surplus fair scheduling: A Proportional-Share CPU scheduling algorithm for symmetric multiprocessors. In *OSDI*, pages 45–58. Usenix, 2000.

[CK68] Edward G. Coffman and Leonard Kleinrock. Feedback queueing models for time-shared systems. *Journal of the ACM*, 15(4):549–576, October 1968.

[CNC05] Bogdan Caprita, Jason Nieh, and Wong Chun Chan. Group round robin: improving the fairness and complexity of packet scheduling. In *ANCS '05: Proceedings of*

the 2005 symposium on Architecture for networking and communications systems, pages 29–40, New York, NY, USA, 2005. ACM Press.

[CNS06] Bogdan Caprita, Jason Nieh, and Clifford Stein. Grouped distributed queues: distributed queue, proportional share multiprocessor scheduling. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 72–81, New York, NY, USA, July 2006. ACM Press.

[DC99] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Symposium on Operating Systems Principles*, pages 261–276, New York, Dec 1999. ACM Press.

[DKS89] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Proceedings of ACM SIGCOMM*, pages 1–12, Austin TX USA, Sept 1989.

[Ess90] R. Essick. An event-based fair share scheduler. In *Proceedings of the Winter 1990 Usenix Conference*, pages 147–162, Berkeley CA USA, Jan 1990. Usenix.

[FNS+04] Hubertus Franke, Shailabh Nagar, Chandra Seetharaman, Vivek Kashyap, Haoqiang Zheng, and Jiantao Kong. Enabling autonomic workload management in Linux. *ICAC*, 100:314–315, 2004.

[GP05] Prasad Gopal and Kashmira Phalak. Share allocation policies. Technical report, Virginia Tech, Blacksburg, VA, Fall 2005.

[Hen84] G. Henry. The fair share scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, October 1984.

[IEE04] IEEE. Information technology - portable operating system interface (POSIX). In *IEEE Std 1003.1*. IEEE, December 2004.

[JRR97] Michael Jones, Daniela Rosu, and Marcel-Catalin Rosu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *SOSP*, pages 198–211. ACM, 1997.

[KKC05] Magnus Karlsson, Christos Karamanolis, and Jeff Chase. Controllable fair queuing for meeting performance goals. *Performance Evaluation*, 62(1-4):278–294, October 2005.

[KL88] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31:44–55, 1988.

[Lov04] Robert Love. *Linux Kernel Development*. Sams Publishing, 2004.

[N+06] Barth Netterfield et al. Kst real-time plotter, 2006.

[NVZ01] Jason Nieh, Christopher Vaill, and Hua Zhong. Virtual-time round-robin: An O(1) proportional share scheduler. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 245–259, Berkeley, CA, USA, 2001. USENIX Association.

[PG93] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[PMG99] David Petrou, John W. Milford, and Garth A. Gibson. Implementing lottery scheduling: Matching the specializations in traditional schedulers. In *Annual Technical Conference*, pages 1–14. Usenix, June 1999.

[Pro99] The Apache Jakarta Project. Jakarta Jmeter, 1999.

[SAWJ97] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. Technical report, Multimedia Computing and Networking, Norfolk, VA USA, February 1997.

[SGG04] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 7th edition, December 2004.

[SHS99] D.G. Sullivan, R. Haas, and M.I. Seltzer. Tickets and currencies revisited: Extensions to multi-resource lottery scheduling. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*. IEEE Computer Society, March 1999.

[SS00] David G. Sullivan and Margo I. Seltzer. Isolation with flexibility: A resource management framework for central servers. In *2000 USENIX Technical Conference*, pages 337–350, San Diego, California, June 2000. Usenix Association.

[VMw98] VMware Inc. Vmware workstation, 1998.

[WKL+96] Thomas Williams, Colin Kelley, Russell Lang, Dave Kotz, John Campbell, Gershon Elber, and Alexander Woo. Gnuplot, 1996.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, Nov 1994.

[WW96] C. A. Waldspurger and W. E. Weihl. An object-oriented framework for modular resource management. In *IWOOOS '96: Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*, page 138, Washington, DC, USA, 1996. IEEE Computer Society.

[ZN04] Haoqiang Zheng and Jason Nieh. SWAP: A scheduler with automatic process dependency detection. In *Proceedings of the First USENIX/ACM Symposium on*

Networked Systems Design and Implementation (NSDI-2004), pages 145–158, San Francisco, CA, March 2004. Usenix Association.

Vita

Abhijit Deodhar was born in the city of Pune, located in central India and famous for its educational institutions. He received his Bachelors degree in Computer Engineering from the Pune University in May 2004. Before attending graduate school at Virginia Tech, he worked at a storage software consulting company for a year. His responsibilities included writing device driver software and testing the SAN product for a storage startup, 3PARdata. His areas of interest include operating systems, especially Linux kernel development, file systems and networking. He likes to listen to Indian classical music and travel in his leisure.