2004

Statistical closed-loop process scheduling

James D. Remus

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Statistical Closed-Loop Process Scheduling

by

James D. Remus

August 2004

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Engineering

Rochester Institute of Technology

Approved by:

Greg P. Semeraro
Primary Advisor: Dr. Greg P. Semeraro, Assistant Professor

Roy S. Czernikowski
Advisor: Dr. Roy S. Czernikowski, Professor

Juan C. Cockburn
Advisor: Dr. Juan C. Cockburn, Associate Professor
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Statistical Closed-Loop Process Scheduling

I, James D. Remus, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

James D. Remus
James D. Remus

8/2/04
Date
Dedication

To Carrie. Your love, support, and encouragement have made this work possible. I could never have done this without you. I will love you always.
Acknowledgements

I would like to thank my advisor, Dr. Greg Semeraro. Your guidance throughout this work has been extremely helpful, and I greatly appreciate the countless hours you have spent in helping me work toward this goal. I would also like to thank the rest of my committee: Dr. Czernikowski and Dr. Cockburn. Thank you for all of your time and effort.
Abstract

Traditionally, scheduling algorithms have been implemented as open-loop control systems. This allows the operating system to make quick decisions on the order in which tasks should be scheduled without requiring complex calculations. It is very common for a task to be assigned a priority based on its anticipated performance, or based on general process characteristics (i.e., I/O bound versus CPU bound). The problem with this type of scheduling, and this type of control system in general, is that it is rigid and lacks the ability to adjust based on the actual performance of the system and its processes. This work is an examination of a simple closed-loop scheduling algorithm that dynamically adjusts the way tasks are scheduled based on the actual system and process performance. It is believed that by making this change to the scheduling algorithm, several important aspects of system performance will be affected. The system resources can be more efficiently utilized because scheduling parameters are dynamically adjusted to compensate for the current system load. The apparent responsiveness of the system, from the point of view of the applications, will increase because processes will be treated more fairly. Also, the overall system throughput will improve, because the closed-loop control system allows the scheduler to make better decisions on the order in which tasks should be run. The implementation of a closed-loop scheduler will result in an increase in the overhead of the scheduling algorithm; however, it is believed that this increase in overhead will be minimal. Extensive testing of the algorithm using a wide variety of applications will be used to demonstrate that the increase is indeed acceptable, given the other benefits of the algorithm.

Due to the fact that the proposed scheduling algorithm is statistical in nature, it does not apply to hard real-time operating systems, but could be used to improve soft real-time operating systems, which have less stringent deadline requirements, and in general purpose time-sharing operating systems. Although this algorithm could have been implemented in any operating system, Linux was chosen as the base platform for this work due to its open source nature. Linux has the additional benefit of providing a well-known environment, and utilities that facilitate the measurements necessary to evaluate the performance of the algorithm.
This work demonstrates that increased overhead required for a closed-loop system is reasonable, and that closed-loop scheduling can provide certain benefits over traditional open-loop schedulers. When compared to the original Linux kernel, the throughput performance degraded typically between 1.5% and 2% depending on the process mix; however, some of the changes to the base kernel can be used to explain this performance degradation. The system clock rate was increased from 100 Hz to 1000 Hz to obtain the timer granularity necessary for the closed-loop control system. Previous work measured a 3.1% increase in overhead when using a 1000 Hz system clock. Measurements were taken on a custom version of the original kernel that was built with a 1000 Hz system clock, which support that claim. When compared to the base kernel with a 1000 Hz system clock, the closed-loop scheduler produces better performance. This work also demonstrates the disadvantage of an open-loop scheduler. An application was developed with fixed length CPU bursts and periodic I/O requests to show that blindly giving the CPU to I/O bound processes and using epochs to age processes results in a significant number of unnecessary process switches that inevitably degrades the performance of the machine. The closed-loop scheduling algorithm balances the load across the processes more evenly, resulting in better performance under a high system load.
Glossary

**Epoch** – Term used by the original Linux 2.4 scheduler to describe a scheduling period. An epoch begins with all processes being assigned their specific time quantum and ends when all processes in the runqueue have exhausted their time quantum.

**Kernel** – Term used to refer to the basic set of functions provided by an operating system. The kernel includes all of the hardware specific code of the operating system, and includes device drivers, memory managers, scheduling algorithm, resource managers, etc. that are used to provide the basic framework for user programs.

**Preempt** – Used in the context of operating systems, preemption refers to the act of forcibly removing the CPU from a process, typically when its timeslice expires or when a higher priority process becomes eligible to run.

**Priority** – A numeric representation assigned to every process in the system that is used to allow the scheduling algorithm to compare the processes in the system when determining which process will be selected to run. High priority processes are favored over low priority processes. The manner in which priorities are assigned is generally unique to a particular scheduling algorithm.

**Priority Ceiling Protocol** – One solution to the priority inversion problem, whereby the process that holds the lock to a shared resource is given the highest priority of any task that uses that shared resource. By doing so, this protocol effectively limits the number of times a higher priority process can be delayed by a lower priority process to one.

**Priority Inversion** – A problem in real-time systems in which a high priority task is unable to run because a medium priority task has preempted a low priority task that holds the lock to a shared resource that is used by both the high and low priority tasks.
Quality of Service (QoS) – Term used to describe the movement in the networking industry to achieve more efficient use of resources by categorizing data into subsets. By doing so, preference may be given to data with strict deadlines, as in communications and real-time audio and video.

Resource Deficit Factor (RDF) – The feedback variable of the closed-loop scheduling algorithm, defined in Equation 5.1 as the percentage of time the processor has been running compared to the total time running or waiting in the runqueue.

Runqueue – The list of processes that are ready to run, but are waiting for the processor. This list also includes the currently running task. In classical Linux terminology, the runqueue is the list of processes that are in the TASK_RUNNING state.

Scheduling Algorithm – The algorithm responsible for determining the order in which tasks are run. The implementation of this algorithm resides within the operating system kernel and is called to select the process that will be next to run.

Time Quantum – Term used in the original Linux scheduling algorithm that referred to the amount of CPU time, in ticks of the system timer, a process has remaining in the current epoch.

Timeslice – The maximum amount of CPU time, in ticks of the system timer, that a process may be given before it is preemted.
# Table of Contents

Thesis Release Permission Form ................................................................................................. ii

Dedication ................................................................................................................................. iii

Acknowledgements ................................................................................................................... iv

Abstract ..................................................................................................................................... v

Glossary ...................................................................................................................................... vii

Table of Contents ..................................................................................................................... ix

List of Tables .......................................................................................................................... xi

List of Figures .......................................................................................................................... xii

Chapter 1 Introduction ............................................................................................................. 1

  1.1 Motivation .......................................................................................................................... 2

  1.2 Organization ...................................................................................................................... 3

Chapter 2 Scheduling Algorithm Background .................................................................... 4

  2.1 Traditional Scheduling Algorithms .................................................................................. 4

    2.1.1 First Come First Serve Scheduling ........................................................................ 7

    2.1.2 Shortest Job First Scheduling ............................................................................... 8

    2.1.3 Priority Scheduling ............................................................................................... 8

    2.1.4 Round-Robin Scheduling ...................................................................................... 9

    2.1.5 Multilevel Queue and Multilevel Feedback Queue Scheduling ............................. 9

  2.2 Real-Time Scheduling Algorithms .................................................................................. 10

    2.2.1 Rate Monotonic Assignment ................................................................................. 13

    2.2.2 Earliest Deadline First .......................................................................................... 14

  2.3 Scheduling Background Summary .................................................................................. 14

Chapter 3 Control Systems in Scheduling ......................................................................... 16

  3.1 Control Systems Background ......................................................................................... 16

  3.2 Control Systems in Scheduling Algorithms ................................................................... 18

Chapter 4: Linux 2.4 Background ......................................................................................... 23
List of Tables

5.1 Parameters Added to the Task Structure ..................................................39
5.2 Global Control System Constants ...............................................................41
6.1 Selected Benchmarks .................................................................................48
6.2 Compute Intensive Load Application Combinations .................................52
6.3 Memory Intensive Load Application Combinations ....................................53
6.4 Typical Load Application Combinations ....................................................53
6.5 Mixed Load Application Combinations ......................................................53
List of Figures

2.1 Memory Map for a Multiprogramming System ......................................................... 4
2.2 Priority Inversion Timeline ..................................................................................... 12
3.1 Open-Loop Control System Block Diagram ............................................................. 16
3.2 Closed-Loop Control System Block Diagram ........................................................... 16
3.3 Closed-Loop Control System with Disturbance ....................................................... 17
4.1 Example of a Simple Epoch Timeline ...................................................................... 30
7.1 Control System Parameter Adjustment ................................................................... 58
7.2 Control System Priority Adjustment ....................................................................... 59
7.3 Average Performance Degradation for Varying Process Mixes .............................. 61
7.4 Fairness Test Performance for All Kernels .............................................................. 64
7.5 Fairness Test Performance for 1000 Hz Kernels ..................................................... 65
7.6 Fairness Test Standard Deviation of Execution Time ............................................. 66
Chapter 1 Introduction

Increased system performance is desirable. This has been one of the primary motivating factors of the desire for the computer industry to produce faster and faster components. Moore’s Law predicted that the number of transistors on a chip would double almost every year [13]. Although not entirely accurate [14], it has characterized the growth of the microprocessor industry very well. This growth has led to rapid increases in processor speed as well as overall computer performance; however, these hardware improvements are not the only factors that impact system performance. Operating systems are just as important as the hardware on which they run in determining the overall system performance. In many cases, the operating system must find a balance between managing system resources and user processes in such a way as to maximize the overall performance of the system while still providing timely responses to interactive processes. Most modern operating systems attempt to provide some protection to prevent malicious users or processes from corrupting important operating system information as well as information that belongs to other users or processes. One solution to this problem, as chosen by the UNIX and Linux operating systems [9], is to support multiple execution modes. User mode, which is the mode in which all user processes are run, restricts a user process from accessing any kernel data structures or hardware devices directly. If these devices or data are needed, the process must use a system call, which is an interface that allows a user level process to ask for a particular kernel service. Once a system call has been issued, a kernel mode process can validate the user request, and fulfill it if the user process has the proper permissions to do so [9]. The kernel is a layer of abstraction that is used to enforce the rules of the operating system, and includes the most important set of system tasks, such as the memory manager, scheduling algorithm, interrupt and exception handlers, file system managers, device drivers, time keepers, etc. [16]. All of these tasks work in combination to perform the critical duties of the operating system. Improvements made in these areas can improve the overall performance of a system. Although much of the kernel contains hardware specific processes that interact with the hardware at the lowest level, there are a significant number of kernel processes that provide the environment in which user processes are run. The scheduling algorithm is one of the most important kernel functions from the point of view of the user applications. This algorithm is directly responsible for selecting which of the user processes will be serviced by the machine.
Although process scheduling is one of the core functions of the operating system, it is often overlooked when attempting to improve system performance. Many of the characteristics that were incorporated into the first scheduling algorithms are still present in those that exist today. The Linux scheduling algorithm still makes many of the same assumptions and contains many of the same characteristics as the UNIX scheduling algorithm from which it was born [9]. There have been improvements over the years, but one thing has remained constant: the scheduling algorithm is still an open-loop control system. Because there is no feedback as to the actual system or process performance, the current Linux scheduling algorithm is unable to adjust to the current conditions. As a result, the performance of the system is not as good as it could be. This work is an examination of whether the increased overhead of a simple closed-loop control system can indeed yield an increase in utilization and performance on a given machine. This algorithm was initially applied to the TASKING kernel used in [12], but it has been extended and adapted for use in a mainstream operating system.

1.1 Motivation

The motivation for this work stems from the fact that many of the scheduling algorithms found in major operating systems are implemented as open loop control systems [36]. A fundamental concept of control systems theory is that open-loop control systems, although simple and stable under some conditions, are rigid and may not be able to produce the desired system response under all circumstances. By closing the loop, the controller is made aware of the current system response and can take additional action to reduce the difference between the actual and desired responses. One of the most important problems faced by the designers of the first scheduling algorithms in the 1960s and 1970s, was how to improve the resource utilization of the machine, especially CPU utilization. Due to the extremely high cost of execution time, one of the primary requirements of these early algorithms was that they had to have minimal overhead in achieving their goals. It was unacceptable for the scheduling algorithm to consume anything but the smallest amount of the system resources. Unless a significant gain in overall system performance could be achieved, the use of additional resources by the scheduler could not be justified. Although control systems theory was not likely considered when designing the early scheduling algorithms, the resulting schedulers were nonetheless open-loop control systems. As
computers continued to evolve, so too did the scheduling algorithms that were used to govern the system. However, one thing has remained fairly constant: open-loop schedulers are still the predominant type of scheduling algorithm used in operating systems. This solution made sense in the past, when execution time was expensive, processors were slow, and memory was small and costly. The extra resources needed to implement a closed-loop solution could not be justified despite the potential advantages of a closed-loop solution. Now that processors have broken the 3 GHz barrier [66][67][68] and memory is cheap [69][70], why should open-loop scheduling remain the solution chosen by most operating systems? Why should processes continue to be scheduled in an unfair manner simply because scheduling has always been done that way? This work demonstrates that closed-loop scheduling algorithms are indeed feasible by showing that the increased overhead of a simple closed-loop scheduling algorithm is not significant, and examines the fairness of the resulting schedules. By closing the loop, the scheduler will be aware of the current system status and can adjust the scheduling parameters to produce a more appropriate response and result in a schedule that is fairer from the perspective of the applications.

1.2 Organization

The remainder of this document is organized as follows. Chapter 2 contains background information on traditional scheduling algorithms. Chapter 3 discusses the relationship between control systems and the computer industry and previous research in how control systems have been used in scheduling. Chapter 4 examines the Linux 2.4 kernel and details the native scheduling algorithm used. The closed-loop scheduling algorithm used in this work, and how it differs from the original Linux algorithm, is presented in Chapter 5. Chapter 6 examines the testing and verification that was performed on the algorithms. Chapter 7 presents the results obtained from the tests and compares the performance of the feedback scheduler to the original. Conclusions drawn from the data and future work are discussed in Chapter 8.
Chapter 2 Scheduling Algorithm Background

This chapter addresses basic scheduling algorithm background. Many of the general classes of scheduling algorithms are discussed as well as the ways in which algorithm performance can be measured.

2.1 Traditional Scheduling Algorithms

Much of the information in this section was derived from the excellent text on operating systems by Silberschatz, Galvin, and Gagne [16]. A scheduling algorithm is responsible for deciding the order in which tasks are executed. The first computers executed a single task to completion before selecting the next one in the list. They lacked an actual software scheduling algorithm, and required the operator to select the task that was to be run. This method was used simply because all of the input devices on early computers were serial in nature. Programs were typically read in using a card reader, and once completed, the machine would begin executing the program. The operator would typically order the programs based on which needed to be completed first, which program was the most important, or whoever was willing to pay a premium to have their program run before others. Although simple, this method was a very inefficient use of the system’s resources. The CPU sat idle while waiting for the slower components (i.e., I/O devices) to complete resource requests. The advent of multiprogramming, in the early 1960s, significantly improved the performance of the computer by allowing a different task to be selected when the computer was blocked on an I/O request. This corresponded to the development of disk technology and the ability to store multiple programs in memory simultaneously, which allowed the computer access to many jobs. One memory configuration for a multiprogramming machine is shown in the figure below.

<table>
<thead>
<tr>
<th>Scheduler/Operating System</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Process 2</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Process n</td>
</tr>
</tbody>
</table>

Figure 2.1: Memory Map for a Multiprogramming System
Multiprogramming is conceptually very simple. One process is run until it is forced to wait for the completion of a request, typically an I/O request. At that time, rather than have the CPU sit idle, another process is selected to run until it must wait for a request to be fulfilled. This pattern repeats for all time, or until all processes have completed. Multiprogramming takes advantage of machine level parallelism. Because I/O devices do not require the CPU to complete their requests, the CPU can be given to another process while the current one waits for its request to complete. Performing many different operations at the same time, in parallel, allows the machine to complete more work in the same amount of time than if each request was treated sequentially. This is the same principle under which modern machines run, only parallelism has been extended to the thread level, as is done in simultaneous multithreaded processors. With access to a pool of jobs that wanted to run, there was a need for a way to select the next process to run. It was at this point in time that the scheduling algorithm was born. The first schedulers were simply special programs that were run each time a process blocked for I/O and attempted to select the best task order. The best task order is a subjective measure of the performance of the scheduling algorithm, and depends primarily on the intended purpose of the machine. For example, two common goals of schedulers have been to maximize system performance or to maximize system utilization. It is worth noting that these two goals are not necessarily the same. Performance is a measure that depends largely on the intended purpose of the system, while utilization is a measure of the percentage of time that the system, or one of its parts, is being used. Performance may have different meanings depending on the situation. For example, in one system high performance may mean high throughput, which would likely correspond to high utilization, while another system may define high performance as an extremely responsive system, which may not perform as well under high utilization. Multiprogramming effectively increased system utilization, especially CPU utilization, and performance in terms of throughput, but it was not without its problems, as it did not allow for timely access to the system. Programs that had long CPU burst times forced all other programs in the system to wait for long periods of time. This type of behavior may be acceptable for a system that runs only background processes, but in a multi-user system that relies on interactive processes, the system will seem unresponsive. Out of this major drawback, multitasking was born. Multitasking allows a machine to quickly switch between tasks, forcibly halting the currently running program if necessary, to allow for timely access to the machine, while maintaining the advantages of multiprogramming. This is
typically done through the use of timer interrupts. In a multitasking system, a process is given the CPU until it makes a request that requires it to block, or until a set maximum amount of time has expired, whichever comes first. The maximum amount of time is usually defined in such a way that the system will remain responsive to interactive processes even if most processes run for the maximum amount of time before they are preempted by the timer interrupt. There is significantly more overhead associated with a multitasking system when compared to a multiprogramming system because the scheduler is run significantly more often. By preempting a process and selecting another, the multitasking scheduler is forcing the system to spend time on something other than executing user processes. Depending on the speed of the hardware and the frequency of the interrupts, the time spent switching processes may not be negligible and may impact the performance of the machine. This decrease in throughput has been viewed as a necessary price to pay in order to achieve a fair and responsive system.

Over the years, many different scheduling algorithms have been developed, each with its own set of advantages and disadvantages, and varying degrees of complexity, usually for a different purpose or to meet the needs of a specific application. There has also been investigation into how to measure the performance of a scheduling algorithm. Although there are many differing opinions on the characteristics that a scheduling algorithm should have, traditionally there are a common set of criteria on which scheduling algorithms are judged: CPU utilization, throughput, turnaround time, waiting time, and response time [16]. CPU utilization is a measure, in percent, of the amount of time the CPU spends executing user tasks. Throughput is a measure of the number of tasks that are completed in a given amount of time. Turnaround time refers to the time between when a process enters the system and when it has completed execution. Waiting time is the amount of time that a process is eligible to be run, but is not granted the CPU by the scheduler. Response time was developed for interactive systems, and is a measure of the amount of time between the submission of a request and the first response. There are other criteria for scheduling algorithms that are more subjective. For instance it is widely accepted that a scheduling algorithm should be fair, although there is no set definition for what "fair" is. In many cases, fairness depends on the intended purpose of the system. For example, in an interactive system a multiprogramming implementation would not be fair because interactive processes would be forced to wait for long periods of time while CPU intensive processes
completed their long CPU bursts. This is an extreme example, but the same case can be made for a multitasking implementation that has selected a large value for the minimum timeslice. This results in an operating system that is not responsive to its interactive processes. Similarly, in a system that is primarily focused on processing CPU intensive tasks, it is not fair for the scheduler to make the CPU intensive tasks wait so that it can provide the smallest response time to the interactive tasks. It is also important to look at the big picture. Even if a scheduling algorithm performs well from the point of view of the tasks, its performance must also be measured from the point of view of the operating system. If a scheduling algorithm requires a significant amount of system resources to make scheduling decisions, it may be less desirable than a simpler algorithm that produces a slightly inferior schedule. Oftentimes theoretical analysis of algorithms is done without regarding the time required to make the scheduling decisions, context switches, etc. This is done to produce findings that are not implementation or hardware specific. However, on actual implementations the amount of time that the operating system spends making scheduling decisions may not be negligible. This is especially true for the time required to perform context switches. If processes are changed too rapidly, the performance of the machine suffers because a significant amount of time is spent performing context switches rather than allowing user processes to complete actual work.

The scheduling problem can be shown to be NP-hard [55]. Since NP-hard and NP-complete problems are often solvable only by using a statistical and/or approximate solution, the use of a statistical closed-loop scheduling algorithm is justified, and is an appropriate solution to the problem of process scheduling. This provides the basis for the statistical closed-loop scheduling algorithm used in this work.

2.1.1 First Come First Serve Scheduling

The first scheduling algorithms were very simple. Possibly the most basic algorithm is First Come First Serve (FCFS) scheduling. This algorithm can be implemented with a simple queue. The process at the head of the queue is run to completion. It is then removed, and the next process is executed. This pattern continues until there are no more tasks to be run. Despite the advantage of simplicity, this algorithm has serious drawbacks. The average wait time for processes is relatively long when compared with other algorithms. Even when adapted to run on
A multiprogramming or multitasking machine, the CPU Utilization and resource utilization are significantly lower than with other algorithms [16].

2.1.2 Shortest Job First Scheduling

Another well-known scheduling algorithm is Shortest Job First (SJF). This algorithm sorts processes in order of increasing execution time, or in the case of multitasking systems, in order of increasing time until the next I/O request (i.e., length of time the CPU will be used before it must be stalled due to an I/O request). This algorithm provides the optimal waiting time for a given set of processes; however, it is more theoretical than practical. In a real system, the execution time or the duration of the next CPU burst are not known at scheduling time, and are very difficult to predict [16]. SJF also has the possibility of indefinite blocking or starvation. If a system is constantly given short tasks to execute, the existing longer tasks may never be given an opportunity to run. Because of this, the only real value SJF provides is a means to compare other algorithms to the optimal solution, in terms of waiting time, for a given set of processes.

2.1.3 Priority Scheduling

Priority scheduling algorithms maintain a value for every process in the system. Scheduling decisions are then based on that value (priority). There are two basic types of priority scheduling: static and dynamic. Static priority scheduling algorithms assign a priority to a task when it enters the system. Every task keeps its assigned priority until it has finished execution. Dynamic priority scheduling algorithms assign an initial priority to tasks when they enter the system, but then allow processes to change priority based on performance, process characteristics, etc. Priority algorithms can be preemptive, meaning that the processor can be taken away from the currently executing task if a task with a higher priority enters the system, or non-preemptive, meaning that the currently running task is allowed to finish executing. SJF is a static priority algorithm that assigns the highest priority to the shortest task. As mentioned earlier with SJF, one drawback to priority scheduling is the possibility of starvation. As long as there are higher priority tasks in the system, a task may never be executed. There are many dynamic priority algorithms that overcome this problem. They typically involve the use of an aging formula, in which the priority of a process increases by some amount after a specified period of time. Eventually the process will reach a priority that will allow it to be run.
2.1.4 Round-Robin Scheduling

Round-Robin scheduling was developed for the first multitasking computers. It is simply a preemptive extension of the FCFS algorithm. A fixed timeslice is established, and the process at the head of the queue is given the processor for at most that amount of time. After the process has been preempted, or has willingly yielded the CPU, it is placed at the end of the queue and the next task is serviced. The problems with round-robin scheduling are similar to that of FCFS. In general, the wait time is long, and the performance of the system heavily depends on the duration of the timeslice and the set of processes.

2.1.5 Multilevel Queue and Multilevel Feedback Queue Scheduling

Another class of scheduling algorithms is multilevel queue scheduling. This scheduling algorithm is an extension of the priority scheduling algorithm, and creates several ready queues for different types of processes at different priority levels. Queues are configured in a hierarchy by priority. If there are any tasks in the highest queue, they will be executed before any tasks in the lower queues. It is also possible to timeslice between the queues. This allows the system to group processes with different requirements. For example, interactive processes can be placed in one queue, which are serviced at one rate to ensure good responsiveness, while background processes can be placed in a different queue with a separate priority and timeslice. Multilevel feedback queue scheduling is an extension of multilevel queue scheduling that allows processes to change priority queues. The idea behind this is to group processes by actual process characteristics as opposed to preassigned process types. Although these algorithms are the most generic of the scheduling algorithms discussed, they require careful selection of the criteria used to define the boundaries between the priority queues, and, in the case of the multilevel feedback queue, the criteria for switching queues. They are also very complex and difficult to implement when compared to the simple priority scheduling algorithms. The UNIX operating system uses a multilevel feedback queue scheduling algorithm with four classes spanning 128 priority levels [16]. Priority scheduling is used across queues, and round-robin scheduling within queues. It is important to note that the "feedback" used in multilevel feedback queue scheduling, as used in several UNIX variants, does not imply the use of a control system, but rather an ad hoc implementation. Once a process has been selected from one of the queues, it is executed until its
timeslice expires or it blocks for I/O. At that point, a calculation is performed based on the previous CPU burst, and the process is inserted into a higher or lower priority queue accordingly. This is an important distinction to make, because the result calculation performed may not truly indicate the nature of the process, but rather its behavior for a single CPU burst. The chosen heuristic may be a good guideline, but it does not have theoretical support in control systems theory. It is this distinction that separates the algorithms discussed in this section from the algorithm presented in this work.

2.2 Real-Time Scheduling Algorithms

A significantly different class of scheduling algorithms is real-time (RT) scheduling algorithms, which are used in real-time systems. There are two major classifications of real-time systems: soft and hard. Scheduling algorithms used in soft real-time systems have much less strict requirements than those used in hard real-time systems. In a soft real-time system, all that is required is that the scheduler guarantees that high priority tasks are favored over those with lower priorities [16]. This allows a general purpose operating system to offer some support for real-time tasks, such as streaming audio and video. It is important to keep in mind that when dealing with soft real-time systems performance is not guaranteed. Streaming audio and video sound and perform well when all of the real-time deadlines are met, but when the deadlines are not met, either due to high system load or lack of throughput from the source, the loss of audio quality or dropped frames are clearly noticeable. This is a tradeoff that is made when using a soft real-time system. It is not necessarily undesirable, since servers with soft real-time operating systems may not want to compromise other services during a peak load. Another difference between soft and hard real-time operating systems is the type of kernel that can be used. Depending on the definition of "real" time, there may be other differences as well. In many cases a hard real-time kernel must be preemptable, if kernel code can execute longer than the definition of real time, whereas a soft real-time operating system does not necessarily have this requirement. A preemptable kernel refers to a kernel that allows interrupts and events to occur and be serviced even while kernel code is executing. In a non-preemptable kernel, interrupts will only be serviced once execution has left kernel mode and reentered user mode.
In reality there is often a mix of normal time sharing and real-time processes in the system at any given time. This is especially true in general-purpose operating systems that attempt to provide soft real-time support. Ideally, all real-time processes would meet their deadlines, but with the presence of other processes in the system, there are a number of possibilities that could prevent this from becoming reality. If a non-preemptable kernel is used, as is the case with the Linux operating system, a system request from one task may interfere with the performance of a real-time task, even though the real-time task has higher priority. The amount of time required to execute the kernel code could impact the performance of the real-time process. Soft real-time systems generally use the same types of priority scheduling algorithms that have already been discussed, with a few exceptions. Real-time tasks are typically assigned the highest priorities. A common solution is to assign a separate priority range to real-time tasks, with all values in that range being higher than normal user tasks. This is the approach used in the Linux kernels, even though, in the strict sense, Linux is not a real-time operating system. Linux allows tasks to be declared as real-time tasks, but the only assurance that is made is that the process will have a higher priority than the normal time-sharing processes. The same algorithm is used to select the next task to run, and the decision is based only on priority. The Linux scheduling algorithm will be discussed in detail in Chapter 4.

The discovery of the existence of priority inversion in some of the first hard real-time systems prompted a significant change to the underlying framework of hard real-time systems. Priority inversion refers to the condition where a medium priority task can be given preference over a higher priority task because the medium priority task has preempted a low priority task that has obtained the lock for a resource needed by the higher priority task. For example, consider the following scenario. A medium priority task preempts a lower priority task that was running, but which has a lock on a shared resource. A high priority task, which needs to use that shared resource, is unable to run because the low priority task holds the lock, so the medium priority task continues to run. This is known as a priority inversion [46]. Priority inversions can result in missed deadlines for the high priority tasks and can even result in deadlocks.
The priority ceiling protocol is one solution to this problem. Under this protocol, the task that holds the lock for a shared resource is given the highest priority of any task that uses that resource, while it holds the lock [38]. This protocol effectively eliminates the inversion by insuring that a high priority task can be delayed at most once by a lower priority task that accesses the same shared resource [39], and does not allow the medium priority task to run while the resource is locked.

In contrast to soft real-time systems, hard real-time systems have strict scheduling requirements. There are generally two different types of hard real-time systems: open and closed. Open systems allow new tasks to be submitted at any time. Closed systems are those in which no new tasks are submitted to the system (i.e., many embedded real-time systems), and have a fixed predefined process set for which all deadlines must be met. In hard real-time systems, timing is critical. This strict timing structure cannot be broken, even if doing so would produce positive results elsewhere in the system [11]. For example, it is unacceptable for a hard real-time system to miss a scheduling deadline, even if doing so would improve the overall performance of the system. Open hard real-time systems can be further subdivided into two classes. Strict systems must immediately reject a task if its deadlines cannot be guaranteed, whereas the less strict systems will accept a task, but fail if the task cannot be completed by its deadline. Generally failure is defined as the inability to meet deadlines because the scheduler is unable to produce a proper schedule or due to over-utilization. Hard real-time systems are typically used for a very
specific purpose, oftentimes in embedded applications. Preemption is extremely important to real-time systems, especially hard real-time systems, which often require specialized hardware and have additional operating system requirements. For example, all code, even operating system kernel code, is typically preemptable. This facilitates the scheduler in making sure that all tasks meet their deadlines. Non-preemptable code may be acceptable if the duration of the code is far less than the magnitude of time dealt with by the system (i.e., a system that deals with time in seconds can easily accommodate non-preemptable code on the order of milliseconds).

Because of the additional requirements of hard real-time systems, an entirely different class of scheduling algorithms has been developed. In 1973, Lu and Layland were among the first to examine the problem of scheduling periodic tasks with hard deadlines equal to the task periods [4]. Out of their research, they developed the optimal static and dynamic priority scheduling algorithms. Rate Monotonic Assignment (RMA) and Earliest Deadline First (EDF) were the algorithms that emerged from this research, and have dominated the real-time scheduling community for the last thirty years [2][3][5][6][7][8].

2.2.1 Rate Monotonic Assignment

Rate Monotonic Assignment is a static priority algorithm that assigns increasing priorities to tasks with decreasing periods. In other words, the task with the shortest period (highest frequency) receives the highest priority, while the task with the longest period (lowest frequency) receives the lowest priority. RMA is considered optimal, meaning that as long as a set of tasks can be scheduled by some fixed priority algorithm, it can also be scheduled using RMA [11]. It should be noted that because RMA is a static priority algorithm, it requires a significant amount of insight about the tasks that will be run on the system if it is to actually be implemented. Although this algorithm is effective in systems with periodic tasks and known schedules, in general it results in a low guaranteed processor utilization. According to the rate monotonic theory, the least upper bound for processor utilization is given by the following equation [4]:

\[ \text{lub}(U) = n \left( 2^\frac{1}{n} - 1 \right) \quad (\text{Eqn. 2.1}) \]

This expression converges to \( \ln(2) \), or approximately 69.3% utilization. This means that as the number of periodic tasks increases, the deadlines of all tasks can always be met as long as the total processor utilization does not exceed 69.3%. In reality, the actual processor utilization is
significantly below \( \ln(2) \) because worst case execution time must be used when determining the task periods. Another disadvantage of RMA is that it does not provide graceful degradation. There is no relationship between the degree of over-utilization and the number of deadlines missed. It is also important to note that RM theory applies only to periodic tasks with known periods and aperiodic tasks that can be modeled as periodic tasks.

2.2.2 Earliest Deadline First

Earliest Deadline First is a dynamic priority scheduling algorithm that also applies to hard real-time systems. In this scheduling algorithm, the highest priority is given to the task with the deadline closest to the current time. EDF is an optimal scheduling algorithm in a single processor system, in that it can schedule any set of task that can be scheduled by any other dynamic priority algorithm; however, this is not necessarily the case in a multiprocessor system [11]. When compared to RMA, EDF has the advantage of efficiency. While the least upper bound for processor utilization for Rate Monotonic Scheduling is approximately 69\%, EDF guarantees feasibility at 100\% utilization [4].

Despite the apparent advantages of EDF over RMA, EDF is not practical except in extremely simple systems. In most real-time systems, it is very difficult to determine the earliest deadline out of all of the tasks. Even if the earliest deadline could be determined, doing so is especially costly considering that this information would have to be determined dynamically every time the scheduler was invoked. This is one reason why, in general, dynamic priority algorithms are more difficult to analyze, and less predictable when compared with static priority algorithms. Because hard real-time systems must be extremely reliable and predictable, in practice the Rate Monotonic algorithm is generally preferred [2][3][5][6][7][8] despite its relatively low processor utilization.

2.3 Scheduling Background Summary

All of the algorithms discussed in this chapter are general classes of scheduling algorithms. Actual implementations may inherit characteristics from one or more of these classes, but there are often implementation specific characteristics as well. Although most operating systems do not have exactly the same scheduling algorithm, in many cases they share common features.
Most algorithms use task priorities to select the process order, but the way in which they determine the priority of a task and when, or if, a task is assigned a different priority, depend heavily upon the desired purpose of the system. Chapter 4 discusses the implementation of the original 2.4.7-10 Linux scheduling algorithm in detail, since this is the base algorithm against which this work is compared.
Chapter 3 Control Systems in Scheduling

This chapter examines how control systems have been used in scheduling algorithms, and discusses previous related work.

3.1 Control Systems Background

Control systems can be broken into two different types: open-loop control systems and closed-loop control systems. Figure 3.1 shows the basic block diagram of an open-loop control system.

![Open-Loop Control System Block Diagram](image)

As can be seen in the above figure, the control system is characterized by the controller and the plant. The controller is responsible for taking some action against the plant based on the current inputs. The plant encompasses the remainder of the system, which responds to the instructions of the controller to produce the actual system response, or outputs. In an open-loop control system, the system responds only to the current set of inputs. Figure 3.2 shows the basic block diagram of a closed-loop control system.

![Closed-Loop Control System Block Diagram](image)
As can be seen in the above diagram, the main difference between the open-loop and closed-loop systems is that the controller of the closed-loop system is made aware of the current system response through the use of feedback. The primary advantage of a closed-loop control system is that it is better able to produce the desired output because it is aware of the current system response. This is especially true when the system is faced with disturbances. Disturbances may manifest themselves anywhere in the control system. Figure 3.3 shows an example of a control system with a disturbance.

![Figure 3.3: Closed-Loop Control System with Disturbance](image)

In the above figure, the plant is separated to demonstrate that the disturbance has occurred somewhere within the plant itself. If an open-loop control system was used, the controller would have no knowledge of the disturbance, and the output would be affected for the duration of the disturbance. However, in the closed loop system, the controller is aware of the actual system response, and can compensate to produce the desired system response. To illustrate this point, consider the cruise control system of an automobile. If cruise control was implemented as an open-loop control system, for a given desired speed, the cruise control would apply the same pressure to the accelerator regardless of the current conditions. This type of control would have been acceptable for the normal case, when the car was traveling on level ground, but as soon as a hill was introduced, the car would travel below the desired speed while going uphill and faster than the desired speed while going downhill. This is an unacceptable response in this type of application. In order to correct this problem the loop must be closed. The actual speed of the car
must be fed back to the controller so it can adjust the accelerator pressure to maintain the desired speed. This same argument can be made for the scheduling algorithm. Because an open-loop scheduler reacts the same way for any set of processes, it may fail to produce the desired system performance level. By closing the loop and allowing the scheduler to be aware of the actual system performance and the types of processes that are currently running on the system, it can adjust the scheduling parameters to improve the performance of the machine. This will be discussed further in Chapters 4 and 5.

3.2 Control Systems in Scheduling Algorithms

A significant amount of research has been done on how computers can be used effectively in control systems. Today, embedded systems control everything from cruise control on automobiles [23][56] to automated assembly lines [57]. In fact it may be more difficult to find a type of control system that computers have not impacted than to list those they have. Control systems and advancements in control system theory have led to the improvement of many products, such as increased safety in the automotive, aircraft, shipping, and railroad industries, as well as the development of new applications that were not previously possible, including space travel and communications satellites [59]. Despite this fact, there has been far less research into how control systems can be used to improve the performance of computer systems themselves. Control systems have been used to make improvements in the areas of multiple clock domain microarchitecture to reduce the amount of energy required per instruction by controlling the voltage and clock frequency to different areas of a microprocessor [58]. Similarly, control systems have been used to improve branch prediction, and also in compilers. Some compilers have used statistics from profiling runs to provide input for compiling an optimized version of the application by eliminating the overhead associated with the most frequently executed portions of the application. This is done by learning the call patterns of the common subroutines. The scheduling algorithm is one of the most prominent areas within the operating system where control systems could be used to improve overall system performance, especially in general purpose machines. Although some research has been conducted in the area of applying control systems theory, the majority has attempted to provide a theoretical framework for feedback in real-time scheduling [10][27][29][30][36][60]. Lu’s Ph.D. thesis [30] developed feedback control scheduling as a framework for quality of service guarantees in unpredictable
environments, as well as deriving models of systems to be controlled. The work done in [10] extends this by applying control systems theory to scheduling design and establishing dynamic models for real time systems. It also provides a generic framework for which a number of different scheduling policies could be used to control the real-time system.

The problem of scheduling extends far beyond just process scheduling. Scheduling is also at the forefront of the networking industry, which presents many unique scenarios where control systems have proven useful. Over the past decade, the Internet has grown into one of the primary sources of information and commerce on the planet. Over this time, a vastly growing number of services have been provided. The fact that in networking, the same medium is used to support a vast array of applications, each with their own set of requirements, is very interesting and challenging from a scheduling point of view. For example, in normal Internet browsing, the user would like to have the ability to surf as quickly as possible, but there are no hard deadlines that must be met; however, when considering transaction based services or communications such as Voice over Internet Protocol (VOIP), there are deadlines that have to be met in order to achieve a reasonable quality of service (QoS). Lu and Abdelzaher [22] attempted to model web servers that offer services such as online trading, banking, and other business transactions, in which performance guarantees must be met within a specified time frame regardless of the current server load. Once they had established a model for the system, they implemented a feedback loop that allowed them to achieve their performance guarantees. They later went on to analyze high volume, timing critical web servers [21], and, through the use of a control loop, were able to maintain levels of utilization that would allow the server to meet all individual real-time deadlines. Similarly, Lu [30] and Lu et al., [37] applied feedback control theory to quality of service aware webservers to achieve relative delay guarantees under various workloads.

The majority of work that has been done on applying control systems theory to process scheduling has focused on hard real-time systems. This has been true for a number of reasons. Hard real-time systems are typically implemented as embedded systems and are designed in an ad hoc fashion [23]. Despite the movement in the embedded industry to use consumer off the shelf (COTS) products, many embedded systems are still designed using custom hardware or custom configurations of COTS products, depending on the intended purpose of the system.
Also, the real-time operating system is often tailored specifically for that system, regardless of whether a custom operating system is developed or a commercially available one, such as VxWorks [61], Integrity [62], or RTLinux [63], is used with custom modifications. Although custom operating systems require the most effort to develop, they are the easiest to experiment with, in the sense that the developer is not restricted by the existing operating system design. A control system can be incorporated into the scheduling algorithm more easily, since the developer has complete freedom to design the operating system. Applying scheduling feedback theory to a commercial hard real-time operating system may not require as much time to develop, but there is the issue of working within the existing framework. An alternative to actually implementing the control system within the scheduling algorithm itself is to have a separate kernel level control task that performs the necessary calculations for the feedback [23]. Regardless of the type, these operating systems tend to have a very small footprint in comparison to general purpose operating systems, because they are often designed to work on a small, known group of hardware components, and embedded systems often have memory constraints that must be considered. This is advantageous because the scale of the changes can be less than in a larger general purpose operating system. Also, since hard real-time systems are created for a singular purpose, the benefits of using feedback scheduling are easier to measure. If the addition of a control system allows the operating system to better meet its deadlines, then it can be labeled a success.

Although hard real-time systems have been used most often in feedback scheduling research, this does not mean that control systems would not be beneficial in general purpose operating systems. There are a number of general purpose operating systems that support soft real-time tasks, which could benefit from an improved scheduler. However, soft real-time tasks are not the only processes that would benefit from such an improvement. Traditional time sharing tasks would benefit as well. These processes have the additional benefits of having no deadline requirements. This gives even more freedom in the control system design, and also simplifies the definition of performance. Since there are no deadlines, for a given set of time sharing processes, one measure of performance can simply be the length of time required to complete all processes. As mentioned in Chapter 2, there are a number of ways to measure the performance of a scheduling algorithm, and feedback scheduling may be able to improve some of these
measures. This work intends to explore the impact of adding a closed-loop scheduling algorithm to Linux on the performance metrics of the operating system.

Linux was designed to be a general purpose operating system. As a general purpose operating system, it must be able to support a wide range of system loads; therefore, its scheduling algorithm must be able to support a wide range of system loads. The values of scheduling parameters must be chosen carefully to insure acceptable performance under all types of situations. The system must be responsive to processes that require user input, yet capable of efficiently executing CPU intensive applications. This means that timeslices must be small enough to allow interactive process to preempt background processes, but long enough so that a reasonable amount of work can be done between preemptions and that the overhead associated with the preemption is reasonable. It also implies that priority should be given to processes that interact with a user, such that the system appears responsive. In traditional scheduling algorithms, the values chosen for scheduling parameters, including priority and timeslice, must be a compromise to allow for acceptable performance under the vast majority of scenarios. This compromise, by definition, concedes the fact that it is not the optimal solution. The optimal solution is the one in which the maximum performance of the system is achieved for any given system load. Because typical scheduling algorithms are static open-loop control systems, they cannot achieve this goal. These algorithms react the same for every given set of processes. For example, the Linux scheduling algorithm is an open-loop control system. It characterizes processes by priority and time quantum, which only vary by small predictable amounts throughout the course of an epoch (more on this in Chapter 4). The scheduling algorithm does not adapt to the current system conditions. Linux, because of its open source nature, allows users to modify scheduling algorithm parameters and compile custom kernels. Linux forums contain many posts from people who have modified the scheduling algorithm to suit their needs. Some have increased the rate of the system clock and decreased the timeslices to make the system more responsive. Others have increased the allowable timeslices to improve the performance of CPU intensive processes. All of these users have identified a scenario under which the scheduling algorithm does not perform as well as it could [64]. There is usually no theoretical support for performance gains achieved by simply changing the values of the scheduling parameters [9] and although the changes may provide improvements under some
conditions, the values are still constants and it is possible to find a set of processes for which the performance achieved by the original parameter values is superior. However, by using a control system to dynamically modify these parameters, the scheduling algorithm may be able to come closer to the optimal solution. Since the control system dynamically adjusts the values of these parameters, it does not face the same shortcomings as a static system. This is the basis for the work described in Chapter 5.
Chapter 4: Linux 2.4 Background

This chapter examines the characteristics of the Linux 2.4 kernel, and the reasons why it was chosen as the base platform for this work. The features of the scheduling algorithm used in the standard Linux kernel are also discussed.

4.1 Linux Kernel Background

Linux has become an increasingly popular solution in the modern world. The Linux 2.4 kernel offered a long list of significant improvements over its 2.2 predecessor, primarily in security, I/O support, scalability, and multiprocessor support [9][65]. The first Linux 2.4 kernel was released in January 2001. It has since proven its reliability and performance, and as a result numerous vendors offered their own version of the Linux operating system, which used the 2.4 Linux kernel, including Red Hat, Mandrake, SuSE, GenToo, Debian, as well as many others. Its low price tag, free in some cases unless technical support was needed, made it a tempting solution for companies and end users looking to save money. The open source nature of Linux also made it an almost ideal platform for academic research. Its modular design and the ability to make customized kernels to suit a particular purpose are very appealing. Linux has the additional advantages of having a large online community, in which users freely exchange ideas in forums, where 3rd party documentation projects are available, and where freely available knowledge bases are maintained by many of the different Linux distribution companies. For these reasons the 2.4.7-10 kernel was chosen as the base platform for this work. Specifically, the Red Hat 7.2 operating system was used. At the start of this work, this operating system had been released for over a year [41]. This was done purposely to insure that the operating system was proven and well received by the Linux community. Although this work was implemented in the 2.4.7-10 kernel, it could be easily ported to other versions of the kernel. The scheduling algorithm and kernel files do tend to change between versions, but the algorithm used in this work does not rely on a specific kernel version.

Linux allows the user a large degree of freedom to customize the operating system to meet their particular needs. The kernel can be compiled with as much or as little as required. For instance,
a kernel that must be as fast as possible on a known set of hardware can be compiled with support for only the required devices built in, but a kernel that must be able to adapt to a changing set of hardware components can be compiled with a small set of known components while the rest can be compiled as modules. Modules are extensions of the kernel that can be compiled separately, and dynamically linked, on demand, into an already existing kernel [9]. Modules can be loaded automatically or manually when a particular device or feature is needed. Many of the high level operating system functions, including the file systems, device drivers, network layers, etc. can be compiled as modules [9]. Modules do have their limitations. They are forced to work within the existing framework of the kernel. For example, modules may not modify any of the structures or interfaces that have been compiled and statically linked into the kernel. In most cases, that is not an issue, and using modules allows the kernel to remain small and flexible. This way only the core features are included in the statically linked kernel while extended features are provided via modules. Linux offers a straightforward configuration utility that allows the user to select which features are compiled into the kernel, those that are to be compiled as modules, and which features are to be omitted completely. Many of the features offered by Linux could be removed from the kernel because they were unnecessary for this project. By removing as many features as possible, the number of sources for error in the project can be reduced. For example, by eliminating support for features such as printer support, sound card support, etc., the number of handlers and processes that are running on the system can be reduced, since interrupts from these devices no longer need to be serviced. Only the most essential components, such as the display adapter and processor type, were built into the kernel. Most others, including the network interface card device driver, were compiled as modules so that they could be loaded if needed. This will increase the reliability of the timing measurements taken during testing, because there are fewer processes that could potentially be scheduled during the tests. The reasons for this will be discussed in detail in Chapter 6.

There are a large number of structures, system defines, and variables that comprise the framework in which the scheduling algorithm operates. It is important to fully understand what resources are available and how they are organized so that they can be used by the scheduler. Every user level process that runs on a 2.4 Linux system has its own task structure. This
structure is used to store process specific information. The definition for this structure is as follows [17]:

```c
struct task_struct {
    /*
     * offsets of these are hardcoded elsewhere - touch with care
     */
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    unsigned long flags; /* per process flags, defined below */
    int sigpending;
    mm_segment_t addr_limit; /* thread address space:
                                0-0xBFFFFFFF for user-thread
                                0-0xFFFFFFFF for kernel-thread */
    struct exec_domain *exec_domain;
    volatile long need_resched;
    unsigned long ptrace;

    int lock_depth; /* Lock depth */

    /*
     * offset 32 begins here on 32-bit platforms. We keep
     * all fields in a single cacheline that are needed for
     * the goodness() loop in schedule().
     */
    long counter;
    long nice;
    unsigned long policy;

    struct mm_struct *mm;
    int has_cpu, processor;
    unsigned long cpus_allowed;
    /*
     * (only the 'next' pointer fits into the cacheline, but
     * that's just fine.)
     */
    struct list_head run_list;
    unsigned long sleep_time;

    struct task_struct *next_task, *prev_task;
    struct mm_struct *active_mm;

    /* task state */
    struct linux_binfmt *binfmt;
    int exit_code, exit_signal;
    int pdeath_signal; /* The signal sent when the parent dies */
};
```
unsigned long personality;
int did_exec:1;
pid_t pid;
pid_t pgrp;
pid_t tty_old_pgrp;
pid_t session;
pid_t tgid;
/* boolean value for session group leader */
int leader;
/
* pointers to (original) parent process, youngest child, younger sibling,
* older sibling, respectively. (p->father can be replaced with
* p->p_pptr->pid)
*/
struct list_head thread_group;

/* PID hash table linkage. */
struct task_struct *pidhash_next;
struct task_struct **pidhash_pprev;

wait_queue_head_t wait_chldexit;   /* for wait4() */
struct completion *vfork_done;      /* for vfork() */
unsigned long rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
struct tms times;
unsigned long start_time;
long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-
specific */
unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
int swappable:1;
/* process credentials */
uid_t uid, euid, suid, fsuid;
gid_t gid, egid, sgid, fsgid;
int ngroups;
gid_t groups[NGROUPS];
kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
int keep_capabilities:1;
struct user_struct *user;
/* limits */
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
/* file system info */
As can be seen from the above definition, there is a significant amount of process specific information that must be stored for every process. Although all the information is used somewhere in the kernel, only a few of these attributes are actually used by the scheduler. Through careful examination of the Linux scheduling algorithm [40], the attributes that are of
importance when considering scheduling include need_resched, counter, nice, policy, has_cpu, processor, cpus_allowed, run_list, next_task, prev_task, and active_mm. These will be discussed further in Section 4.2.

The next_task, prev_task, and run_list attributes are of particular importance when considering the organization of the processes within the operating system. Linux maintains several different mechanisms for organizing tasks. All processes currently in the system are maintained in a doubly linked list aptly named the process list. The prev_task and next_task attributes reference the previous and next processes in the process list. For efficiency in scheduling, all processes that are either running or ready to run are included in another doubly linked list known as the runqueue. The run_list attribute is a structure that contains another pair of process pointers that reference the previous and next processes in the runqueue. This allows the scheduler to quickly examine only the tasks that are able to be scheduled. Linux also maintains a hash table so that processes can be accessed via their process ID. This is the only manner in which stopped or zombie tasks can be accessed. In Linux processes will enter the stopped state when certain signals are received from the operating system. Processes in the stopped state, although technically still in the system, are not running and will never be selected by the scheduler while they remain in this state. The zombie state is a special state that is used to describe a child process (one that has been spawned by another process via the fork() system call) that has terminated, but has not yet joined with the parent. Since these tasks have no chance of being scheduled in their current state, there is no reason to maintain other structures with references to them. Linux also manages a number of wait queues. Typically wait queues correspond to a specific system event, timing event, or interrupt. They are used frequently in resource requests and in inter-process communication (IPC). Any process that requires notification when an event has occurred, or a request has completed, must be placed on the proper wait queue. At that time, another process is selected by the scheduler. Waiting processes may be one of two types, interruptible or uninterruptible, with the difference being that interruptible tasks can be woken up by signals even if the event on which they were waiting has not yet occurred.
There are a significant number of system defines that are used to define values in the operating system, but perhaps the most important define used in the Linux kernel is the value of HZ. HZ is used to define the frequency of the system timer. In the standard Linux kernel, this value is set to 100, which means that all operating system timers are in integral amounts of approximately 10 milliseconds (actually closer to 10.5ms [9]). This define provides the finest level of timing granularity that can be obtained by the operating system. Its importance to the closed-loop scheduling algorithm used in this work will be discussed in Chapter 5.

4.2 Linux 2.4 Scheduling Algorithm

The Linux 2.4 scheduling algorithm is a dynamic priority based scheduling algorithm that also supports real-time applications. Fundamental to the scheduling algorithm is the idea of epochs. Figure 4.1 shows a simple example of an epoch on a Linux machine with only two processes running on the system.
1. Start of Epoch 1. Process A is selected by scheduler
   Process A
   Goodness = 26
   Counter = 6
   Process B
   Goodness = 21
   Counter = 5

2. Process A blocks for I/O, Process B scheduled
   Process A
   Goodness = N/A (not in runqueue)
   Counter = 4
   Process B
   Goodness = 21
   Counter = 5

3. Process B is preempted when Process A’s I/O request completes
   Process A is scheduled
   Process A
   Goodness = 24
   Counter = 4
   Process B
   Goodness = 20
   Counter = 4

   Process A
   Goodness = 0
   Counter = 0
   Process B
   Goodness = 20
   Counter = 4

5. Process B blocks for I/O
   No runnable processes with time remaining. End of Epoch 1
   Process A
   Goodness = 0
   Counter = 0
   Process B
   Goodness = N/A (not in runqueue)
   Counter = 2

6. Start of Epoch 2
   Process A
   Goodness = 6 + 20 = 26
   Counter = 0 + 5 + 1 = 6
   Process B
   Goodness = 6 + 20 - 4 = 22
   Counter = 1 + 4 + 1 = 6

Figure 4.1: Example of a Simple Epoch Timeline

At the beginning of each epoch, every process in the system is assigned a calculated amount of time that is referred to as the process’ time quantum, or timeslice. This time quantum is the maximum amount of CPU time, measured in ticks of the system timer that can be dedicated to that process during the current epoch. The time quantum is stored in the counter attribute of task structure of each process, as is indicated in Figure 4.1. The time quantum is calculated using the following equation:

\[
counter = \frac{\text{counter}}{2} + \frac{20 - \text{nice}}{4} + 1 \quad (\text{Equation 4.1})
\]

Processes may be assigned different time quanta depending on the type of process and its priority. From the scheduler’s point of view, the priority of a process is determined by the goodness of the process. The goodness of a process is the value returned by performing the \texttt{goodness()} function on that process, and is determined by the following expression:

\[
goodness = counter + 20 - \text{nice} + \text{proc\_bonus} + \text{mem\_bonus} \quad (\text{Equation 4.2})
\]
where counter is the number of CPU ticks remaining in the current epoch and nice is the nice value of the process. The nice value of a process ranges from \(-20\) (highest priority) to 19 (lowest priority). The proc_bonus is a bonus used only in a symmetric multiprocessing (SMP) environment, and has no meaning in a uniprocessor system. The mem_bonus is a bonus of 1 given to a process if it owns the currently active memory descriptor. In a given epoch, the CPU may be given to a process any number of times, as long as the time quantum of the process has not expired. For example, the CPU may be taken away from a process when it blocks for an I/O request, but that process may be selected to run later on in the same epoch once the I/O request has been fulfilled. When all tasks that are currently in the runqueue have exhausted their given quanta, the current epoch ends and the scheduler recalculates the time quanta for all process for the new epoch. It is important to note that only processes in the runqueue (those that are not on a wait queue, but are ready to run) are considered in scheduling and in determining when the end of an epoch has been reached.

Every process has a base quantum associated with it, which is inherited from its parent and is a function of that process' priority and nice value. The nice value is a fixed amount that a process may raise or lower its priority. Every process is given a time quantum of at least 1, and half of any time remaining from the previous epoch is added to the base quantum for the current epoch. In the example in Figure 4.1, Process B was blocked for an I/O request when epoch 1 ended. The portion of Process B's time quantum that was not used during the first epoch was not completely lost, but rather half of it was delayed to the next epoch. This is indicated by the additional 1 tick that was added to the counter at the beginning of epoch 2. The quantum represents a number of ticks of the system timer, which is governed by the \(\texttt{HZ} \) define. The base time quantum can range from 1 to 11 ticks, depending on the nice value of the process. Equation 4.1 is a geometric series with an upper bound of 22 ticks (or about 230 ms) \([9]\). Since the scheduling algorithm uses only integer arithmetic, the maximum value that counter can achieve is 21 ticks. This occurs only when the process has the maximum nice value (-20) and has received the full bonus from many previous epochs.

The Linux scheduling algorithm makes its decision based on priority. It examines the entire runqueue, which contains all of the tasks that are able to be run, and selects the task with the
highest priority. It is important to note that Linux separates conventional (time sharing) and real-
time tasks. Conventional tasks have a dynamic priority, which is the sum of the base quantum 
and the number of ticks left in the quantum in the current epoch. Real-time processes are 
assigned static priorities ranging from 1 to 99. All real-time tasks have a higher priority than 
conventional tasks, and are dealt with in a first-in first-out (FIFO) or round-robin (RR) fashion 
depending on the policy of the task. It is important to note that the Linux definition of priority 
differs from the definition of priority used in real-time systems. This is the primary reason why 
Linux cannot technically be considered a real-time operating system.

The Linux scheduling algorithm is implemented in the `schedule()` function. This function 
may be invoked directly or in a deferred (lazy) manner from a number of different kernel 
factors [9]. The scheduler is invoked directly whenever the currently executing process must 
block due to a required resource being unavailable. The lazy invocation is performed by setting 
the `need_resched` flag in the task structure. This is done typically out of convenience when 
it is known that `schedule()` will be invoked directly in the near future. Lazy invocation is 
used when the current process has exhausted its quantum, when a higher priority task is woken 
up, or if either the `sched_setscheduler()` or `sched_yield()` system calls are issued 
[9]. The following paragraph describes how the Linux scheduler works. It is important to note 
that this description is a summary of the important features of the scheduler. There are a number 
of actions that are taken by the scheduler that are omitted from this description because they are 
unrelated to the scheduling algorithm, but must be included in the actual implementation. These 
features include locking mechanisms to protect data and other such operating system 
housekeeping chores.

Regardless of how it is invoked, the scheduling algorithm proceeds in the same manner. The 
purpose of the `schedule()` function is to determine the next process that will be run and, 
ultimately, to begin execution of that process. This may or may not involve a process switch, 
depending on whether a different task is chosen to replace the one that is currently executing. 
The first operation performed by the scheduling algorithm is to check if the current process is a 
real-time process that has exhausted its time quantum. If so, a new quantum is assigned and the 
process is moved to the end of the `runqueue`. The new quantum can range from 1 to 11 ticks,
with a default value of 6 ticks. The default value is commonly used, so the process typically results in a new quantum of about 60 milliseconds. Next, if the current process (the one that was running when the schedule function was invoked) was waiting for a request that has since been fulfilled, it is added to the runqueue to give it an opportunity to be selected again by the scheduler; however, if its requests have not yet been fulfilled, it is removed from the runqueue since it cannot run even if given the CPU. Next the need_resched flag for the current process is set to 0 to indicate that the scheduler has been run if it was invoked using the lazy method. At this point, all of the housekeeping has been done, and the scheduler may now select the next task to run. There is always at least one process that can run, even if there are no processes currently in the runqueue. This process is known as the idle task. The idle task has a fixed priority, which is a value that is less than any other process. It is selected to run if, and only if, there are no other processes in the system that are ready to execute. As soon as a new task enters the system or an existing one wakes up, it will be given the CPU over the idle task. The idle task is selected as the default process in the case that examination of the runqueue yields no runnable processes. For each process in the runqueue, the scheduling algorithm calls the goodness() function on that process. The goodness() function examines the task structure of the specified process to determine the priority of the process. For conventional processes, the goodness() function computes and returns the priority of the process. As long as the process has CPU time remaining in the current epoch, the priority is calculated using Equation 4.2. A bonus of 15 is given to processes that last ran on this processor. This is done to minimize the number of times that processes switch processors, and to better take advantage of processor cache; however, on a uniprocessor system, this bonus is not given to any process, and thus has no effect on scheduling whatsoever. If the process owns the currently active memory descriptor, a bonus of 1 is added. In this (normal) case, a process will result in a goodness value that ranges from 2 to 77 inclusive on a multiprocessor system and between 2 and 62 on a single processor system. If the process has no CPU time remaining in the current epoch, then a goodness value of 0 is returned. A special scenario occurs when a process has voluntarily yielded the processor. In this case, a goodness value of -1 is returned, since there is no reason to select a process that asks the operating system to schedule a different task. For real-time processes, the goodness function returns 1000 plus the real-time priority of the process. This guarantees that the priority of the real-time process is higher than any of the conventional
processes. The scheduling algorithm keeps track of the process with the highest priority (goodness value) and once it has gone through the entire runqueue, the process with the highest priority is the one selected to run next. A special case occurs when all tasks in the runqueue return 0 because all quanta have expired. This indicates the end of an epoch. At this point, every task in the process list is assigned a new quantum, which is defined by Equation 4.1. This insures that each process is given at least one tick of CPU time, but less than 22 ticks. The scheduling algorithm is then run again to select the best process. At this point, the best process has been selected, and all that remains to be done is the actual process switch. If the next process to run was the process that was just running, no switch is required.

4.3 Performance of the Linux 2.4 Scheduling Algorithm

The Linux scheduling algorithm is both simple and straightforward. There are no complex calculations required, and as a result it is very efficient under a relatively light system load; however, the algorithm does not scale well [9]. Every time the scheduler is invoked, the goodness calculations are made for every task in the runqueue. As a result, this algorithm is of order O(n) in the number of runnable processes. Even though the computations are simple, they are made every time regardless of whether or not there has been a change in the base priority of the process. With a large number of processes, this can be quite costly. Also, as described in the previous section, the scheduling algorithm recalculates the priority and time quantum of all processes at the beginning of each epoch. With a large number of processes, this may not be an insignificant amount of time. For example, in the VolanoMark application, which is a benchmark tool that is used to measure the performance of VolanoChat, found that between 37 and 55 percent of kernel time was spent inside the scheduler [42]. VolanoChat is a Java implementation of a chat room server. The VolanoMark test establishes a Java socket connection to a chat room server for all of the simulated users. Each connection is handled by its own thread, and since the number of servers in the benchmark ranges from 5 to 25, the number of threads can range from 400 to 2000 threads [42]. This makes the VolanoMark an excellent test for scheduling efficiency.

Through careful examination of the algorithm, it can be seen that Linux gives preference to I/O bound processes as opposed to CPU bound processes. Because I/O bound processes will often
yield the CPU because of an I/O request, it will typically receive a higher goodness value than a
CPU bound process that exhausts more of its time quantum before being preempted. The
reasoning behind this is to provide good response time for interactive processes, of which the
majority are I/O bound. This preference for I/O processes is not necessarily the optimal solution.
Database applications may be I/O bound, but do not require short response times. Conversely,
some interactive applications may be CPU bound, in which case they may seem unresponsive.
Although these types of applications are generally the exception to the rule, blindly giving
preference to I/O bound processes may result in less than desirable system performance [9].

One problem with this type of scheduling is that the actual performance of the system has no
effect on scheduling. Tasks are scheduled in the same manner regardless of the current system
load. For example, if there are not currently any I/O bound processes running on the system,
why not allow the timeslices of the CPU bound processes to increase? By giving these processes
longer periods of CPU time, they are preempted less, and thus the overhead of scheduling is
reduced. This would also extend the epoch time, further reducing the overhead because the
quanta can be recalculated less frequently. However, without some sort of feedback mechanism,
there is no way for the scheduler to be made aware of these types of situations. As a result, the
system will not be used as efficiently as it could have been. By incorporating the feedback
mechanism into the scheduler, the system will be able to take advantage of situations like the one
described above, while maintaining the flexibility to handle loads on the other end of the
spectrum.
Chapter 5 Closed-Loop Scheduling Algorithm

This chapter contains a description of the closed-loop scheduling algorithm used in this project. The algorithm [43], which was adapted from an algorithm that was meant to be implemented into the TASKING kernel that was used in [12][15], was designed for simplicity, to allow for easy integration into the existing Linux kernel architecture, and also so that the increase in overhead would be minimal.

5.1 Algorithm Specification

Ideally during the design phase of any software system, implementation details should not influence the design, but rather a list of requirements should be established. However, this is not an ideal situation. There is an existing framework that must be considered, and the design of this closed-loop algorithm will take into account the implementation of the Linux 2.4 operating system, including the organization of task structures, process queues, etc. The goal of this algorithm is not only to improve upon the original scheduling algorithm, but also to change as little of the existing infrastructure as possible. This is done in an attempt to compare the two different types of scheduling as fairly as possible. By changing as little other operating system functionality as possible, the number of other factors that could affect the performance measurements is reduced.

The algorithm used in this work attempts to dynamically alter its scheduling behavior based on the current system performance. The closed-loop control system adjusts two key existing scheduling parameters, namely the priority and time quantum (timeslice) of each task, as well as several new parameters that are used to hold control system information. The priority and timeslice are the two most important parameters in the scheduling algorithm because they determine which process is selected to run, and for how long it will be given the CPU (assuming that it does not block for a resource request before its quantum expires). The use of these parameters is the same as the original Linux scheduling algorithm. The difference lies in that their values will be adjusted based on the feedback control variable, which will be referred to as the resource deficit factor (RDF). The resource deficit factor is a measure of the amount of CPU
time a process has been given compared with the total amount of time the process has been either running or waiting to run. It is calculated using the following equation:

\[
\text{rdf} = \frac{100 \times \text{time}\_\text{running}}{\text{time}\_\text{running} + \text{time}\_\text{ready}}
\]  

(Equation 5.1)

This value can range from 0 to 100. The goal of the scheduler is to balance the resource deficit factor across the entire system. This is accomplished by maintaining an average resource deficit factor for all processes, and adjusting the scheduling parameters in a manner that forces all tasks to approach the average RDF. The average resource deficit factor is calculated as follows:

\[
\text{avg\_rdf} = \frac{(\text{num\_tasks} - 1) \times \text{avg\_rdf} + \text{current\_rdf}}{\text{num\_tasks}}
\]  

(Equation 5.2)

where \text{num\_tasks} is the current number of processes in the system and \text{current\_rdf} is the resource deficit factor of the currently scheduled process. In this system, fairness is defined as all tasks having the same resource deficit factor. This means that over time, all tasks should be given the same amount CPU time in proportion to the amount of time they have been running or waiting for the CPU. It is important not to confuse the resource deficit factor with priority or timeslice, because they are not the same. The resource deficit factor is used to adjust these parameters.

Process priority for this algorithm is determined using a ratio similar to that of the RDF. It is defined using the following relation:

\[
\text{priority} = \frac{10000 \times \text{time}\_\text{ready}}{\text{time}\_\text{running} + \text{time}\_\text{ready}}
\]  

(Equation 5.3)

The priority of a conventional process can range from 0 to 10,000. Real-time tasks are given a bonus of 10,000, and can range from 10,000 to 20,000. This maintains the soft real-time policy of Linux, which is that all real-time tasks have a priority higher than conventional tasks, but allows the control system to schedule real-time tasks as well, rather than simply using a FIFO or round-robin approach. This ratio assigns a process a priority that is essentially the opposite of the RDF, in that it considers the amount of time that a process has been waiting for the CPU. This also allows for process aging, as long as the following assumptions are true:

1. The scheduling algorithm recalculates the priority of a process in the ready queue every time the scheduling algorithm is run.
2. Once a process has entered the ready queue, it may not leave unless it is selected by 
the scheduling algorithm as the next task to run. 

Because none of these features were changed from the original Linux 2.4 kernel, these 
asumptions may be taken as truth. 

In contrast to the original Linux scheduling algorithm, the closed-loop control system allows the 
process timeslice to be much more flexible. The original algorithm assigns the time quantum 
based on the base priority of a process, which made this essentially a fixed value, unless the 
process had CPU time remaining at the end of the previous epoch. The introduction of the 
resource deficit factor allows the timeslice of a process to be adjusted based on its performance 
relative to the system performance. One of the goals of the control system is to allow CPU 
bound processes to obtain longer timeslices since they will not often have to block due to an I/O 
request. This reduces the number of unnecessary preemptions due to expired time quanta even if 
there are no other tasks that are ready to run. The following relation describes how the timeslice 
is adjusted for the currently running process: 

\[
\text{if} \ ( (\text{timeslice} < \text{MAX\_TIMESLICE}) \ \text{AND} \ (\text{rdf} > \text{avg\_rdf}) )
\]
\[
\quad \text{timeslice} = \text{timeslice} + 1
\]
\[
\text{otherwise if} \ ( (\text{timeslice} > \text{MIN\_TIMESLICE}) \ \text{AND} \ (\text{rdf} < \text{avg\_rdf}) )
\]
\[
\quad \text{timeslice} = \text{timeslice} - 1
\]

where \text{timeslice} and \text{rdf} are the current timeslice and resource deficit factor of the 
currently running process, \text{avg\_rdf} is the average resource deficit factor of the system, and 
\text{MAX\_TIMESLICE} and \text{MIN\_TIMESLICE} are the maximum and minimum timeslice values 
chosen for the system. This conditional attempts to identify the current system conditions and 
adjust the timeslice of the process accordingly. On a system with a large number of CPU bound 
processes, the current process is likely to exhaust its timeslice before being preempted by a 
higher priority process (i.e., a higher priority task has been woken up because its resource request 
has been fulfilled). This will result in the resource deficit factor of that process exceeding the 
average, assuming the system was at steady state prior to when the process ran. In this case, the 
timeslice of the process should be increased to reduce the number of preemptions. In another 
case, if the process ran but was preempted before it exceeded the average resource deficit factor,
then it is likely that the system is in an I/O bound state. In this case the timeslice should be reduced so that the system is more responsive.

Although each of the individual parameters can be examined separately, it is important to analyze how the control system modifies the values of the parameters to create a more responsive system. The resource deficit factor is used to control both the priority and timeslice of a process, which are related in an inverse way. Processes with high resource deficit factors, relative to the average, will receive lower priorities, but longer timeslices, while processes with low resource deficit factors will receive high priorities, but short timeslices.

### 5.2 Algorithm Implementation

The closed-loop scheduling algorithm examined in this work is significantly different than the original Linux scheduling algorithm, which, not surprisingly, required significant modification to implement the desired solution. Despite the significant changes to the algorithm, the original kernel was analyzed extensively to determine how to implement the closed-loop scheduling algorithm, but also to minimize the impact of the additions on the rest of the kernel. The resulting implementation required the modification of only five kernel source files. A number of new scheduling related parameters were added to the task structure that is associated with every process in the system. These are identified in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>The priority of the process</td>
</tr>
<tr>
<td>timeslice</td>
<td>The time quantum given to the process</td>
</tr>
<tr>
<td>resource_deficit_factor</td>
<td>The resource deficit factor of the process</td>
</tr>
<tr>
<td>transition_ts</td>
<td>The timestamp of the last state transition (i.e., from ready to running)</td>
</tr>
<tr>
<td>total_running_time</td>
<td>The total amount of CPU time (measured in ticks of the system timer) that the process has been given</td>
</tr>
<tr>
<td>total_ready_time</td>
<td>The total amount of time (measured in ticks of the system timer) that the process has spent in the runqueue waiting to be</td>
</tr>
</tbody>
</table>
These parameters were added to the end of the task structure because there are numerous instances where offsets into the structure are hard coded in the kernel. This makes inserting the parameters into the middle of the structure difficult, since all hard coded offsets would have to be identified and modified. In the original task structure, the parameters used in scheduling were stored near the beginning of the structure, and were stored in such a way as to allow all of the parameters to fit within a single cache line to improve the performance of the scheduling algorithm. Because the new scheduling parameters had to be appended to the structure, the scheduling algorithm may not perform as well as it could have if the new parameters fit within the cache line.

As their names imply, the priority and timeslice fields were added to maintain the current priority and timeslice of the process. Their values, as described in the previous section, are a function of the resource deficit factor. The priority of a task is treated the same as in the original Linux kernel, with the exception of having a larger range of legal priorities. The timeslice, however, is treated very differently. Because process priorities and timeslices are adjusted by the control system, there is no longer a need for epochs. Epochs were necessary in the original kernel to insure that all processes were given an opportunity to run within a known amount of time. Under the closed loop scheduling algorithm, a process is given its full timeslice (i.e. the counter field in the task structure is set to the current value of timeslice) every time it is selected by the scheduling algorithm. It is the responsibility of the control system to make certain that a process is assigned an appropriate priority and timeslice for the current system load. A resource_deficit_factor is maintained for every process in the system, and its value is updated whenever a process makes a state transition (i.e., from running to waiting, from waiting to ready, from ready to running, from running to ready, etc.). Additional updates are performed on each task in the ready queue every time the scheduler is invoked. This insures that processes in the runqueue will age to prevent them from being starved. The additional calculations required to compute the priority and timeslice could have an impact on the overhead of the scheduler. This is analyzed by the tests performed, and the
findings are presented in Chapter 7. The *transition_ts* parameter was added to maintain the timestamp of the last state transition that was made by the process. Conveniently, the Linux kernel maintains a count of the number of ticks of the system timer since the last reboot. This count is stored in a global variable called *jiffies*. By storing this value as the timestamp, it can easily be compared to the current time when the process makes its next state transition. The transition timestamp is updated whenever the process changes state. The *total_running_time* and *total_ready_time* fields are used to account for the total amount of time a process spends in that particular state. These values are used in the calculation for the *resource_deficit_factor* of the process, and measured in ticks of the system timer. The *total_running_time* is updated after the process has had the CPU taken away from it, and is calculated using the following equation:

\[
\text{total\_running\_time} = \text{total\_running\_time} + \text{timeslice} - \text{counter}
\]  \hspace{1cm} (Equation 5.4)

The *total_ready_time* is updated for every process in the runqueue that is not running, whenever the scheduler is invoked.

Accompanying the additions to the task structure are several system-wide parameters that are used by the control system. The two global variables used are the *avg_resource_deficit_factor* and *present_num_tasks*. *avg_resource_deficit_factor* is used to store the average of all resource deficit factors currently in the system, while *present_num_tasks* is used to track the number of processes in the system. The number of tasks is used in the calculation for *avg_resource_deficit_factor*, as given above in Equation 5.2. It is incremented and decremented in the existing *fork()* and *do_exit()* kernel methods, which are used by all Linux system calls that create or terminate processes. The global constants that were used in the control system are described in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HZ</td>
<td>1000</td>
<td>The number of system timer interrupts per second.</td>
</tr>
<tr>
<td>MIN_TIMESLICE</td>
<td>1</td>
<td>Minimum timeslice, in system timer ticks, that can be assigned to a process. (Approximately 1ms)</td>
</tr>
</tbody>
</table>
Aside from the changes made to the scheduling algorithm, the single most important change made to the existing kernel was to change the value of the HZ define. This value was increased from 100 to 1000, causing the system timer to interrupt at a rate of 1000 interrupts per second. This allows the minimum timeslice to be approximately 1ms instead of 10ms. As previous work has shown [12][18], smaller timeslices cause increases in overhead due to the increased time spent in interrupt handlers and also due to the proportionally longer amount of time required for task switching. Koster [18] measured the increase in overhead to be approximately 3.1% for a 1ms clock tick, and over 10% if the value of HZ was increased to 10,000. Because the time required to switch tasks is relatively constant, decreasing the timeslice increases the percentage of time spent switching when compared to running. Another significant disadvantage of the increased frequency is the fact that the value of jiffies rolls over ten times faster than in the original kernel. In the original kernel, this 32-bit value was able to keep track of over 497 days worth of system timer interrupts. By increasing the frequency ten times, jiffies rolls over in less than 50 days. Because of this, additional checking is required to insure that jiffies has not rolled over since the state transition timestamp was recorded. Despite these disadvantages, the increased frequency of the system timer provides a distinct advantage from the point of view of the control system by giving it the potential to react more quickly. All closed-loop control systems require a certain amount of time to settle for given a set of inputs, or after a disturbance has been introduced. By decreasing the period of the system timer, the sampling rate of the system is increased, which will allow the control system to respond more quickly and appropriately to the current situation. The MIN_TIMESLICE and MAX_TIMESLICE values were defined to establish hard boundaries for that portion of the control system. This is done to reduce the settling time of the system when the current conditions change. For example, if there was only one process in the system, and that process happened to be a CPU bound process that rarely blocked for a resource request, its timeslice would continue to grow if there was no maximum limit. When a new process enters the system, it may have to wait a significant amount of time before it would be given the CPU simply because of the enormous value of the existing

| MAX_TIMESLICE | 600 | Maximum timelice, system timer ticks, that can be assigned to a process. (Approximately 0.6 seconds) |
---|---|---|

Table 5.2: Global Control System Constants
process’ timeslice. The settling time for this scenario could be quite lengthy, which is not a desirable characteristic of a control system. For this reason, MAX_TIMESLICE was defined to be 600 ticks.

The majority of the changes were implemented in two functions: goodness() and schedule(). As described in section 4.2, the goodness() function is responsible for calculating the priority of the process. As in the original kernel the closed-loop scheduling algorithm calculates the priority of the specified process, but it does so using the new definition of priority. Additionally, checks are made to insure that the value of jiffies has not rolled over since the time of the last state transition and also to be certain that the value of total_time is not greater than what can be contained in a 32-bit unsigned integer. If all is well, the priority of the process is calculated using the formula in Equation 5.3. If the total time is greater than 32 bits, the time_ready and time_running are decreased by a factor of 2 to correct the problem before calculating the priority. This method was chosen because it effectively decreases the times without having a significant impact on the resource deficit factor calculation. The schedule() function required significant modification from the original algorithm. Immediately after obtaining and releasing the necessary locks, the scheduler was modified to update the total_running_time and transition_point_timestamp of the process that was just running. As in the original kernel, the state of the process is then updated, and it is removed from the runqueue if it is waiting on a resource request. The avg_resource_deficit_factor is then calculated using Equation 5.2 based on the value of the resource_deficit_factor of the process that was just running. Its timeslice is then adjusted as described in Section 5.1. The scheduler then loops through the list of processes in the runqueue to determine the one with the highest priority (goodness). Once the best process has been selected, its counter is reset to the current value of its timeslice, its resource_deficit_factor is calculated, and the process switch, if necessary, begins. Modifications were also made to the try_to_wake_up() function. This function is used to wake up a process on any one of the wait queues. Once a process has been woken up, its transition_point_timestamp is updated and the resource_deficit_factor is recalculated. The only other kernel modifications were made to the main kernel function to
initialize the global parameters used by the closed-loop scheduling algorithm, and to the
fork() and do_exit() functions to monitor the current number of processes in the system.

All of the code that was modified was done so using conditional compiles. This allowed
multiple versions of the kernel to be built using the same source files, changing only a few global
defines found in the top-level kernel makefile.

5.3 Kernel Logging
A number of other features were added to the kernel for the sole purpose of providing the ability
to monitor and analyze the control system. In order to insure that the control system was
functioning properly and that the values of the constants and parameters were appropriate, a
logging mechanism was built into the scheduling algorithm. The original Linux kernel provides
a printk() function, which has traditionally been used for kernel debugging and exception
logging. The printk() function is very similar the printf() C function, but the output of
the print statement, combined with an identifier flag, is picked up by the syslog daemon process
and deposited in one of the files in /var/log. The identifier flag is used by syslog to determine in
which file to place the message. By default, the majority of messages are logged to the
/var/log/messages file. For the purposes of the control system, the KERN_NOTICE identifier
was used, and the messages of this type were deposited in a separate log file. This established a
nice container for any scheduling data that would be useful to log. It also provided a means of
debugging the new scheduling algorithm during its initial stages.

Many of the parameters used in the scheduling algorithm, including the priority, resource deficit
factor, average RDF, and timeslice, as well as the number of processes and the process ID of the
currently executing process were logged using this mechanism. This allowed insight to be
gained into the control system. Because the logging of this data could severely impact the
performance of the operating system, it was added as another conditional compile to allow it to
be turned on or off using a global define in the top-level makefile. By creating two kernels,
benchmarks could be run on the kernel without logging to measure performance, while the
kernel with logging would allow for verification that the control system was properly managing
the scheduler. The printk() function was designed specifically for the purpose of logging
important operating system and kernel information and as such should have minimal impact on the performance of the operating system, and thus the scheduling algorithm. Although the resulting schedules between the logging enabled and logging disabled kernels would not likely be identical for a fixed set of processes, the difference should be minimal. The major difference between the two would be the need for the syslog daemon process to be scheduled to process the logs produced by the use of printk() when logging was enabled.
Chapter 6 Testing and Verification

This chapter contains a complete description of the testing strategy used to compare the performance of the scheduling algorithms. It also presents the different benchmarking tools that were used in the testing.

6.1 Testing Introduction

Testing the performance of a scheduling algorithm can prove to be a difficult task. Since there is not necessarily one correct schedule, a number of characteristics must be considered when comparing the two scheduling algorithms. Of the performance metrics described in Chapter 2, CPU utilization, throughput, and fairness were the considered the most important in this work. A two-point approach to testing was taken in order to compare the performance of the two scheduling algorithms. While CPU utilization and throughput are objective metrics that can be easily observed, fairness is subjective and more difficult to quantify. To prove that implementing a closed loop scheduler did not significantly degrade the throughput and CPU utilization of the kernel, a large number of test scenarios using many different mixes of processes were run on both of the kernels. The durations of the tests were measured, as was the CPU utilization achieved by each of the algorithms. The specifics of these scenarios are discussed in detail later in this chapter. Comparing the fairness of two different algorithms has historically proven to be a difficult task. Similar difficulties have been encountered in the field of simultaneous multithreading processors, such as the Pentium 4, where multiple threads of execution contend for the processor simultaneously. Many people have struggled to find an answer to the question: “Is it fair for one thread to delay the execution of another?” [71][72]. Because the algorithm for thread selection is implemented in hardware, it has a fixed response for a given set of inputs. Research has shown that it is possible for a process to exploit this algorithm and actually perform a denial of service (DoS) attack on the machine by preventing other processes from being given the CPU [45]. Similar questions can be raised for normal process scheduling. Is it fair to delay one process to improve the performance of another? In order to examine the fairness of the scheduling algorithms, the system was exposed to a large process load. While the system was being taxed with a large number of applications, a separate
application was run, and the responses of the two algorithms were observed to determine how fairly the new process was treated. This test makes the basic assumption that almost any scheduling algorithm can be fair when the system is lightly loaded. The true test of fairness is when the algorithm must efficiently schedule scarce system resources (i.e., the CPU under heavy system load).

6.2 Test Applications

One advantage of using the Linux operating system was having a wide range of available benchmarking tools, many of which were free. For this work, select applications from the MediaBench, Olden, and SPEC2000 suites were used. The MediaBench applications were initially developed for use in embedded systems [49], but have been used extensively in workstations that are commonly used for multimedia and communications. All of the applications in the MediaBench suite have been studied previously [47]. One interesting characteristic of the MediaBench applications is that they are rate-based applications [15] that contain a small portion of code that must be executed periodically at fixed intervals. The execution time of these applications varies from frame to frame as a function of the input data rather than as a characteristic of the actual algorithm [48]. The Olden benchmark suite was originally designed to evaluate the Olden single-program, multiple-data (SPMD) machine and compilation system [15][51]. Since its inception, the suite has grown with the addition of five more applications. Generally, the applications in the Olden suite are memory intensive, pointer based applications with complex memory access patterns [50], and perform simple operations on data structures, including linked-lists and binary trees [15]. These applications focus more on how the data structures are traversed and manipulated rather than actual data processing algorithms. The SPEC2000 applications [52] were designed to exercise the CPU and memory of a workstation. While most of the benchmarks fall under the category of typical workstation applications, some are closer to applications that would run on a server [15].

All of the applications from these suites were run individually on the original Linux kernel. The average time for ten iterations was calculated. The benchmarks that ran too quickly were eliminated. This was necessary to insure that the execution time of the application was large in relation to the timeslice and also so that error in timing could be reduced to a negligible amount.
The granularity of the timer is limited to the rate of the system clock, which also determines the minimum timeslice. Applications that required only a few timeslices to complete could not be accurately timed and would make comparisons between the two kernels less valid. It was also necessary to allow the control system of the modified kernel time to settle. Closed-loop control systems cannot respond immediately to all conditions, so some settling time must be allowed. Processes that had extremely long execution times were also eliminated. This was done to establish a reasonable length of time for each test iteration. By eliminating applications from the suites on these criteria, the list of applications was reduced to the benchmarks that ran long enough to insure that the majority of the time was spent executing the benchmark, rather than in command interpreting, startup, or cleanup, but short enough to allow the tests to complete in a reasonable amount of time. The standard deviations of the execution times were calculated, and algorithms whose execution time varied too widely were also eliminated to minimize the effect on the results. Ideally, all of the applications used would have a fixed execution time from iteration to iteration. This would insure that any difference in execution time between the two kernels could be attributed to the different scheduling algorithms used; however, that proved not to be the case. Because the applications did indeed vary from iteration to iteration, the applications that had the smallest variance were chosen. All others were eliminated. The resulting set of benchmarks is shown in the following table.

<table>
<thead>
<tr>
<th>Compute Intensive</th>
<th>Memory Intensive</th>
<th>Typical</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>gcc</td>
<td>bisort</td>
</tr>
<tr>
<td>equake</td>
<td>health</td>
<td>swim</td>
</tr>
<tr>
<td>gzip</td>
<td>mcf</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsp</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Selected Benchmarks

Previous work [15] had classified the applications in these suites into varying categories of compute intensive, memory intensive, and typical processes. Several benchmarks from each category that met the above criteria were selected to serve as representatives of that particular group. The following is a brief description of the selected benchmarks [15].

**bzip2** – The bzip2 compression utility designed by Julian Seward, in the SPEC2000 suite.
bzip2 compresses files using the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding. Compression is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors, and approaches the performance of the PPM family of statistical compressors [53].

**equake** – This benchmark simulates the propagation of seismic waves in large basins, contained in the SPEC2000 suite.

The equake application “is a floating-point benchmark which uses finite element analysis over an unstructured mesh to determine ground motion everywhere within the simulated landscape which is a result of a seismic event.” [15]

**gzip** – The Lempel-Ziv coding (LZ77) compression utility found in the SPEC2000 suite.

This algorithm is a variable-to-fixed length code. The algorithm parses the input sequence into non-overlapping blocks of varying lengths while maintaining a dictionary of blocks that have already been processed. The general algorithm flow is as follows [15]:

1) Initialize the dictionary so that all blocks are of length one.
2) Search for the longest block that has appeared in the dictionary.
3) Encode the block using the index in the dictionary. This is the compression step, since the dictionary index is smaller than the block.
4) Add that block and the first symbol of the next block to the dictionary.
5) Go back to 2) as long as there is more data to compress.

**gcc** – A C language optimizing compiler for the Motorola 88100 processor from the SPEC2000 suite.

The gcc application is a classic compiler in the sense that it is generally a table-driven implementation of a push-down automata (PDA) and a finite state automata (FSA).

**health** – A Columbian health care problem solver from the Olden suite.

The health application simulates a health care system in which patients check-in and check-out of hospitals using arrays of doubly-linked lists. The present load on each
hospital determines the amount of processing that must be done for each operation. Some individuals [53] have argued that health does not simulate a "real" application because it often performs many unnecessary traversals of the doubly-linked lists. Since the Olden benchmarks were designed as kernel benchmarks rather than to simulate a "real" application [15], it is very well suited for this work, as it is the performance of the kernel that we are interested in rather than the performance of the applications.

**mcf** — A single-depot vehicle scheduling algorithm in public mass transportation system found in the SPEC2000 suite.

The mcf benchmark uses the network simplex algorithm, which is a specialized version of the simplex algorithm commonly used in network flow problems. The linear algebra of the general algorithm is replaced by network operations, including modifying spanning trees or finding cycles, all of which can be performed very quickly. This algorithm heavily relies on pointer and integer arithmetic, and is extremely memory intensive.

**tsp** — The traveling salesman problem solver found in the Olden benchmark suite.

The tsp algorithm builds a balanced binary tree recursively in postfix order. The computation proceeds by traversing the tree in the same manner in which it was built until the subtree contains 150 nodes. At that point an iterative algorithm performs floating point computations using a partitioning algorithm and a closest point heuristic.

**bisort** — A forward and backward integer sorting algorithm from the Olden suite.

The bisort application sorts a binary tree of integers, which was recursively initialized with random data using a constant random seed. Sorting is done by recursively traversing the tree forwards, then backwards. The disjoint bitonic sequences that result from this process are then recursively merged to obtain the final sorted result.

**swim** — A weather prediction program from the SPEC2000 suite based on a shallow water model equations solver.
The *swim* benchmark is a Fortran77 floating-point application that solves the system of shallow water equations using finite difference approximations on a $N_1 \times N_2$ grid. The implementation in the SPEC2000 suite uses a $1335 \times 1335$ matrix over a period of 512 timesteps.

The small nature of the MediaBench applications, due to their roots in embedded systems, resulted in the elimination of all benchmarks from this suite. There were enough applications from the SPEC2000 and Olden suites that fit the required criteria, so it was not necessary to search for other test data that could be used with the MediaBench applications. All tests described in this chapter were run on a 1 GHz AMD Athlon processor with 512 MB of RAM and a 2 GB swap partition.

### 6.3 Performance Testing

By categorizing the benchmarks based on process characteristics and selecting several from each of the categories, a wide range of test scenarios could be designed and a wide range of system loads could be tested. In this way, the scheduling algorithms could be tested under a completely compute intensive load, a memory intensive load, a typical load, and a wide range of mixed loads. The vast majority of combinations of benchmarks were used to insure that a fair comparison between the two scheduling algorithms could be made under a large range of process loads. Also, by testing many of the different combinations, conclusions drawn from the performance could be more generalized because the possibility that a particular benchmark strongly favored one scheduling algorithm over the other would be reduced. Since each of the different benchmarks had different execution times, multiple instances of the shorter benchmarks were run sequentially to simulate a longer benchmark. Although not exact, this method worked well in equalizing the execution time of the benchmarks to roughly the same duration. Using this method introduced some error into the timing measurements due to the additional command interpreting, startup, and cleanup overhead, but this time was insignificant in comparison to the duration of the tests. There was never an instance where a benchmark was run more than five consecutive times during one test. Since that particular test required more than an hour and fifteen minutes to complete, the additional overhead could be ignored. The GNU Make utility was also extensively used in this work. The `-j` option allowed each benchmark to be started in
its own thread at as close to the same time as the system will allow. This also created a single parent process that allowed the total execution time of each test load to be easily measured. Using the Make utility also contributed some additional overhead, but when compared to the total execution time of the test, the overhead is negligible, and since the same overhead should be experienced by both algorithms, the comparison is still valid.

The first phase of testing was conducted after the integration of the scheduling algorithm was completed. Since the addition of a closed loop control system to the scheduling algorithm required the kernel to be significantly altered, the modified kernel was tested for correctness to insure that the system performed all of the same functions as the original and that there were no major side effects of adding the control system. Once the kernel had been verified to be functional, the real testing required for this work could begin. As mentioned earlier, the testing was divided into two major categories: performance and fairness.

As described in Chapter 5, the addition of the closed-loop scheduling algorithm required a number of additional parameters to be kept for each process in the system, and a number of additional calculations had to be performed by the scheduling algorithm to provide the appropriate feedback. In order for closed-loop scheduling to be a viable scheduling solution, this additional overhead cannot degrade the performance of the system significantly. If the system performance achieved under closed-loop scheduling is significantly worse than the original, then the potential improvement to fairness is meaningless. However, if the performance degradation is small, then an improvement in fairness can be extremely helpful to certain types of applications. In order to compare the performance of the two scheduling algorithms, a large number of test scenarios using wide ranges of process types were run to insure that results were obtained for a wide variety of system loads. The following tables show the tests that were run on each of the kernels.

<table>
<thead>
<tr>
<th>Test #</th>
<th>bzip2</th>
<th>equake</th>
<th>gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 6.2: Compute Intensive Load Application Combinations (X = application included)
### Table 6.3: Memory Intensive Load Application Combinations (X = application included)

<table>
<thead>
<tr>
<th>Test #</th>
<th>gcc</th>
<th>health</th>
<th>mcf</th>
<th>tsp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### Table 6.4: Typical Load Application Combinations (X = application included)

<table>
<thead>
<tr>
<th>Test #</th>
<th>bisort</th>
<th>swim</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test #</th>
<th>bzip2</th>
<th>equake</th>
<th>gzip</th>
<th>gcc</th>
<th>health</th>
<th>mcf</th>
<th>tsp</th>
<th>bisort</th>
<th>swim</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: Mixed Load Application Combinations (X = application included)

Although not all combinations of the tests were run due to the extremely large number of possibilities and amount of time required, a large number of tests were run to get a representative sample for each of the different types of combinations. All of the above combinations were executed for ten iterations on both the original and modified kernels. The throughput and CPU utilization were measured for all iterations through the use of standard Linux `time` utility. This utility program measures the amount of real time a process has been running, the amount of user time the process has used, the amount of system time used, the CPU utilization percentage, and the number of page faults caused by the process. As mentioned previously, as many processes as possible were terminated on the machine prior to running the benchmarking tests to reduce as many external factors that could influence the results as possible. Only the processes required for normal system operation were allowed to continue running during the testing, and these processes were common to both kernels.
By using such a large number of combinations for multiple iterations, the possibility of a single test or combination skewing the results was reduced significantly. The shorter benchmarks in the tests were made longer by running them multiple times in succession until all benchmarks in that particular test finished at approximately the same time. This was done to insure that it was the scheduling that was being measured rather than the ability of the system to process individual applications. This is a subtle distinction, but a very important one. It was necessary for the all of the applications in the test to run for the vast majority of the overall execution time. This forced the scheduler to select between all of the processes in the test for most of the duration. If the shorter benchmarks had simply been allowed to terminate, only the long benchmarks would be running for a significant amount of the test’s execution time, resulting more in a measurement of the increased overhead of the modified kernel than a comparison of the schedulers. The comparison between the two kernels is still fair, since both were doing essentially the same amount of processing, differing only in how the processes in the test were scheduled. This method is not perfect, but it was simple and proved sufficient to measure the performance of the modified kernel in relation to the original.

6.4 Fairness Testing

The second phase of testing addressed the issue of fairness as seen from the point of view of the applications. The idea of fairness is difficult to define. Everyone has a notion of what fairness means, but it is often difficult to describe. In this case fairness was defined as the consistency to which the system was able to service a task for a given system load. This type of fairness was measured by establishing a fixed system load of the benchmarks used in the performance testing while measuring the performance of a test application. All nine applications from Table 6.1 were set to run continuously, meaning that once the application terminated, it was immediately restarted. A tenth process was then added to the system and the response was observed. This test process was developed to demonstrate the inherent flaws in open-loop process scheduling. The test application was designed to simulate a “database-type” process, in which periodic accesses are made to memory while performing some amount of processing on the data. The test process was created with a fixed-length CPU burst. The CPU burst simulated actual processing by performing a number of calculations on integers read from the memory database. Once the
process had completed its current CPU burst, it slept for a given amount of time before continuing with its next CPU burst. The sleep time was varied from 1000 ms between bursts to 10 ms between bursts to simulate different loads on the database. Because the CPU burst was constant across all request rates, the only variable was the rate of the requested CPU bursts. The total sleep time between CPU bursts was set to 5 minutes (300 seconds) for all requested rates. The actual execution time was defined by the following relation:

\[ exeTime = sleep\_time + cpu\_time + other\_time \]  \hspace{1cm} (Equation 6.1)

The sleep\_time was equal to five minutes. The cpu\_time was the total amount CPU time spent processing the CPU bursts of the test application. The number of bursts increased with the rate of the request rate. The other\_time consisted of the time spent executing other processes or time spent process switching. The tests with higher request rates performed more CPU bursts and process switches, and thus had longer execution times and higher CPU utilization. The time required to complete all of the different request rates on both the original and closed-loop kernels was measured while the nine other benchmarks were running. This allowed the fairness of the scheduling algorithms to be compared by examining how the test program was treated under a heavy system load. Times for ten iterations of the test process were recorded for each of the different sleep times, and the average execution time was calculated to reduce the impact of a single iteration. In order to eliminate the difference in the frequency of the system timer between the original and modified kernels, a third kernel was built that was identical to the original Linux kernel, but the system timer was increased to 1000 Hz to match the closed-loop kernel. The test application was run, and the data was collected as in the other two kernels. By analyzing a kernel with the original scheduling algorithm and a 1000 Hz system timer, all differences other than the scheduling algorithm were eliminated, producing a more valid comparison between the two scheduling algorithms.
Chapter 7 Performance Comparison

This chapter presents the results obtained from the tests described in the previous chapter and compares the performance of the closed-loop scheduler to that of the original.

7.1 Correctness Verification

The first phase of testing had very little to do with measurements of performance or comparisons between the original and modified scheduling algorithms, but rather was done merely to insure that the closed-loop scheduling algorithm was operating according to its design. All of the processes from the three benchmarking suites were run individually to prove that the closed-loop scheduler could accommodate individual processes well. This not only provided a means of selecting which processes to use for the real performance testing, but also served as a way to initially validate the performance of the scheduling algorithm after it had been implemented. The logging features that were built into the modified kernel were used extensively to further validate the scheduling algorithm. By carefully examining the values that were logged, it was possible to observe how the closed-loop scheduler adjusts the scheduling parameters to adapt to the current situation. When completely compute intensive loads were present, the scheduling algorithm increased the timeslices of the compute intensive processes and decreased their priorities. When completely memory intensive loads were present, the scheduling algorithm responded by assigning relatively short timeslices and high priorities when compared to the compute intensive programs. Several small test programs with CPU bursts and fixed I/O request periods were used to demonstrate that the scheduler assigned priorities and timeslices appropriately to the processes. Once the scheduler had proven to be reliable and operating according to its design, the actual performance and fairness testing was conducted. The following two diagrams demonstrate an example of how the control system adjusted the scheduling parameters to adapt to the current situation.
Figure 7.1: Control System Parameter Adjustment
Figures 7.1 and 7.2 represent the modifications that the closed-loop scheduler made to the scheduling parameters while running the bzip2 and gcc benchmarks while allowing several processes to remain in the background. In all there were between 40 and 50 processes running throughout the test, most of which were operating system processes. A sample was taken approximately every 100ms using the kernel logging feature described in Chapter 6. It is important to note that at a sampling rate of 10 Hz, the system was undersampled. This was done because the mechanism for sampling the system had to work within the framework of the system. Sampling at rates faster than 10 Hz significantly impacted the performance of the system, and did not accurately reflect its performance. It is possible that there were fluctuations in the parameters between the samples, but it is believed that these fluctuations were minimal. The information for the currently running process was logged. For this reason, two processes that would demand a large portion of the CPU were chosen to insure a significant number of data points. For the test shown, gcc terminated after approximately 5 minutes, while bzip2 terminated
after approximately 7 minutes. Both figures are shown in logarithmic scale to better demonstrate the settling time for the algorithm. Because both applications were the only two processes that demanded a significant amount of the CPU, their resource deficit factors ranged in the mid 40’s, which was expected since the two process combined consumed approximately 90 percent of the CPU. Both were larger than the average resource deficit factor, as is illustrated in Figure 7.1, and as a result their timeslice values quickly closed in on the maximum value allowed (600), and their priorities approached 6000. Once the gcc benchmark terminated, the priority of the bzip2 process decreased and its resource deficit factor increased until it terminated. This was expected since the process utilized approximately 90 percent of the CPU itself during that time. As can be seen by the above figures, the algorithm quickly settled into its steady-state response. It is important to note that because of the logarithmic scale of the figures, the settling time may appear longer than it actually was. The system typically settled within 20 to 30 seconds of the start of the test. This is an acceptable result considering several processes were started simultaneously. One must also consider that oftentimes processes exhibit different behavior at startup than they do once the actual application is running. This could delay the settling time of the control system. Many tests were conducted, all of which produced similar results to this example.

7.2 Performance Degradation

In order to implement a closed-loop control system into the Linux scheduling algorithm, significant modifications of several kernel functions as well as the addition of several parameters to the structures used to keep information about each and every process in the system were required. Despite the potential scheduling improvement these modifications and additions could provide, there was the distinct possibility that the overhead of the scheduler would be increased. When the modifications were combined with the increase in overhead caused by using a 1000 Hz system timer (rather than the 100 Hz timer used in the original Linux kernel), an increase in scheduling overhead and some overall performance degradation were expected. Provided the level of degradation was reasonable, and the resulting schedule was fairer to the applications, the closed-loop scheduler would provide some benefit. To insure that there was indeed a reasonable level of degradation between the original and closed-loop schedulers, the tests described in Chapter 6 were run on both kernels. The tests described in Tables 6.2, 6.3, 6.4, and 6.5 were run
to completion for 10 iterations on both the original and modified kernels. The average performance degradation for each of the different categories of tests is shown in the following figure.

![Average Real Performance Degradation for Different Process Mixes](image)

**Figure 7.3:** Average Performance Degradation for Varying Process Mixes
The information contained in Figure 7.3 demonstrates several important facts about the closed-loop scheduling algorithm. The most important fact is that the observed performance degradation caused by the addition of a closed-loop control system into the scheduling algorithm is indeed reasonable. For almost all process mixes, the average performance degradation falls between 1.5 and 2%. Another important observation is that the performance degradation of the kernel with the closed-loop scheduling algorithm is less than the increased overhead of the standard Linux kernel with a 1000 Hz system clock used in [18], which measured approximately 3.1%, and also the original Linux kernel used in this work that was modified to use a 1000 Hz system clock frequency (This kernel will be referred to as the 1000 Hz base kernel). The average of a random sample of the tests from Table 6.5 were used to determine the average performance degradation of the 1000 Hz version of the original kernel, and the average performance degradation was calculated to be 3.12%. This observation has several important implications regarding the performance of the closed-loop scheduler. When comparing the two kernels that use a 1000 Hz system clock frequency, the fact that the kernel with the closed-loop scheduler has a smaller average performance degradation suggests that the closed loop scheduler is able to compensate in some way for its increased overhead. Since the feedback scheduling kernel faces the same increased overhead for using a 1000 Hz system clock and also has the increased overhead caused by the additional calculations required for the control system, the fact that it is able to produce a lower performance degradation than the 1000 Hz base kernel is quite remarkable. Without having any knowledge of the internals of the closed-loop scheduling algorithm, one would expect that kernel to have higher performance degradation than the 1000 Hz base kernel. However, this result implies that the closed-loop scheduling algorithm must be doing something to reduce the overhead of the scheduler, and indeed it is. The closed-loop scheduling algorithm adjusts the scheduling parameters based on the current system load. The compute intensive processes are given longer timeslices so the scheduling algorithm is run less often. This does not significantly impact the response time of the system since interrupts still occur when a higher priority process wakes up, at which point the scheduling algorithm is run, resulting in the higher priority process being selected. Since the control system controls the aging of the processes, the time spent in the original kernel at the beginning of every epoch to calculate the timeslices for every process in the system is no longer necessary. This benefit
becomes even more important as the number of processes in the system grows, especially if many of them spend the majority of time sleeping. The scheduling algorithm also adjusts the timeslices for the memory intensive processes to a value more suitable to their CPU burst patterns. By replacing the concept of epochs with a closed-loop control system, there is no longer the unfortunate situation where a process switch occurs for a selected process that only has one tick of CPU time remaining, that is unless the process has earned that timeslice through past behavior. The closed-loop scheduler grants every process its full timeslice every time it is selected. This effectively reduces the process switch penalty that must be paid for the vast majority of situations.

Although the tests used in this work were not designed specifically to compare the CPU utilization between the two algorithms, it was measured for all tests to insure that the introduction of the closed-loop scheduling algorithm did not have an adverse affect on the utilization. The measured CPU utilization for all tests run on both algorithms was very comparable. The granularity provided by the Linux time application is limited to 1%. In all tests the average CPU utilization measured the same to the nearest percent. This effectively demonstrated the utilization under the closed-loop scheduler was not less than that of the original algorithm.

7.3 Fairness Comparison

The purpose of testing fairness under a heavy load was to force the scheduling algorithms to make scheduling decisions while the system resources were in high demand. Almost any scheduling algorithm can be fair under a light system load because there is little contention for resources. The true test of fairness is when the system is under stress. All of the test applications used in the performance testing were capable of demanding a high percentage of the CPU, and many of the processes were very demanding of memory and cache. By measuring the execution time of the test program, the responsiveness and fairness of the scheduling algorithms could be determined. For a scheduling algorithm to be fair, one would expect the algorithm to service a process with a similar response regardless of the rate at which the process requested to be serviced. In the case of the test application, the request rate was increased while maintaining a constant CPU burst. Ideally the scheduling algorithm would be able to service the test program
at the requested rate, resulting in an execution time that scaled according to the amount of work done by the process. In reality, one would expect the execution time of the test application to increase slightly as the request rate was increased, especially as it approached the rate of the system timer, at which point a significant amount of time would be spent process switching. The average execution time of the test program for the three kernels is shown in the following diagram.

![Diagram](image)

**Figure 7.4: Fairness Test Performance for All Kernels**

One of the most striking features of Figure 7.4 is the inability of the 100 Hz base kernel to service the test application at a rate of 100 Hz. This is due to the fact that the request rate is the same frequency as the system timer, resulting in the majority of time being wasted on process
switches. This result was not unexpected, and was the primary reason behind building the 1000 Hz base kernel. The data points for the 100 Hz base kernel were omitted from Figure 7.5 to allow for a better comparison between the two kernels with the 1000 Hz system timer.

![Fairness Test](image)

**Figure 7.5: Fairness Test Performance for 1000 Hz Kernels**

There are several key points of interest in Figures 7.4 and 7.5. First, the closed-loop scheduling algorithm consistently outperforms the two base kernels for requested burst rates of at least 10 Hz, while providing comparable performance at the lower requested burst rates. This indicates that the closed-loop scheduling algorithm is able to service the test application at a rate that is
closer to the requested rate. Second, the execution time of the test application is much more stable under the closed-loop kernel than in either of the two base kernels. This demonstrates that the closed-loop scheduling algorithm stabilized the system better than the algorithm used in the two base kernels. This point is further supported by analyzing the standard deviation of the execution times from the ten iterations of the test application, as Figure 7.6 demonstrates.

![Fairness Test Application Standard Deviation](image)

**Figure 7.6: Fairness Test Standard Deviation of Execution Time**

As can be seen in the above diagram, the standard deviation of the execution time of the test application is comparable to or less than that of the two other kernels for the vast majority of request rates. This is validation that the closed-loop scheduling algorithm has adjusted the scheduling parameters to better suit the current system load. It also demonstrates that the closed-
loop algorithm is able to provide a more consistent execution time for the test program. Another important observation is that the 100 Hz request rate begins to hurt the performance of all three kernels. At that rate, the time spent process switching becomes a significant part of the execution time. Together, these results prove that the closed-loop algorithm is not only fairer to the test application on average, but it is also more consistent in its servicing of the test process.
Chapter 8 Conclusions and Future Work

This chapter outlines the contributions made by this work and discusses additional work that could be done to further enhance the use of closed-loop schedulers.

8.1 Conclusions

This work has made two important contributions. First, it has shown that the increased overhead caused by the implementation of a closed-loop scheduling algorithm is reasonable. For the majority of the process loads run during testing, the performance degradation was between 1.5% and 2% of the original kernel, but as demonstrated by the measurements taken on the 1000 Hz base kernel, the increase in the system clock rate could explain this degradation. In fact, the performance of the kernel with the closed-loop scheduling algorithm often exceeded that of the base kernel with the 1000 Hz system clock. This performance improvement when compared to the 1000 Hz base kernel, demonstrates that the closed-loop scheduler can indeed compensate in many instances for its increased scheduling overhead. Although this was not one of the primary goals of this work, it is a welcome result nonetheless.

This work also demonstrated the flaws inherent in an open-loop scheduler. Open-loop control systems are rigid. They respond the same way to the current set of inputs without regard for the current system output. The test application showed that the open-loop scheduler could not continue to service the application at the requested rate as well as the closed-loop scheduler. In the open-loop scheduling algorithm, the epoch heuristic was used in an attempt to provide a level of fairness to the processes by insuring that all processes were given an opportunity to run for a limited amount of time in any given epoch. Depending on the time quantum assigned to the test application, the CPU burst may not have completed by the time its quantum expired. This would force the application to wait until the next epoch to complete its burst. Because the closed-loop scheduling algorithm could dynamically adjust its scheduling parameters, it was able to adjust the timeslice of the test application to better match the CPU burst. Since the control system was responsible for maintaining the level of fairness across the processes, the epoch system and its associated overhead were eliminated. Furthermore, the control system reduced the number of unnecessary process switches by granting each process its full, earned timeslice every time the
process was selected by the scheduler. By determining the amount of time spent waiting for the processor compared to the amount of time the process was running or waiting, the closed-loop scheduler was able to be fairer to the test process. Although the test application was designed for this work specifically, the results should hold for any application that has similar characteristics (i.e., database applications, transaction-based web applications, web servers).

8.2 Future Work

There are a number of areas in which this work could be extended. Although the choice to use Linux suited this work very well, there were some limiting factors that potentially hurt the performance of the closed-loop scheduling algorithm. The most significant problem was the way the task structure had been used throughout the Linux kernel. The additional attributes had to be appended to the structure because, as the comments that accompany the task structure indicate, there are numerous places where offsets into the task structure are hard coded. The original parameters used in scheduling are included at the beginning of the structure, and are organized to fit within a cache line to improve the performance of the scheduler. Attempting to insert the new parameters following the location of the existing parameters produced kernels that would not function. It was not within the scope of this work to identify all of the instances where hard coded offsets were used, but it would be interesting to see the effect on performance of allowing the new scheduling parameters to fall within the length of the cache line.

Another area that would be worth investigating is analyzing the effect of the system clock frequency on the performance of the closed-loop scheduler. The 1000 Hz frequency was selected for this project somewhat arbitrarily. It was chosen because it was a reasonable value, but there was no in-depth analysis done, and perhaps a lesser or greater frequency would provide better performance. Similarly, it would be interesting to modify the parameters that affect the settling time of the algorithm to see what effect that would have on the performance of the kernel.

Possibly the most important area of this work that would benefit from future exploration would be a continuation of the fairness testing. The application that was developed in this work to demonstrate the benefit of a closed-loop scheduler was simply an example of type of application
for which a closed-loop scheduling algorithm provide better performance and fairness. Investigation into real applications that would benefit from such a change in scheduling would be a worthwhile effort.
References


[40] Red Hat Linux 7.2, 2.4.7-10 kernel sched.c source file.


[53] From the bzip2 man page. Red Hat Linux 7.2.


