Understanding the Linux 2.6.8.1 CPU Scheduler

By Josh Aas

17th February 2005

Contents

1 Introduction

1.1 Paper Overview

Linux kernel development is relatively fast-paced given the size and complexity of the code base. This is because of its widespread adoption by hobbyists, home users, businesses (including many Fortune 500 companies), and educational institutions. The Linux kernel mailing list (LKML, a mailing list for kernel developers), as of summer 2004, averages about 300 messages per day from between 50 and 100 different developers. These numbers do not include most architecture-specific discussions, which happen on separate lists. In the year before August 1st, 2004, over 16,000 patches of widely varying sizes were committed to the official Linux kernel [7]. This pace of development has led to a situation where very few of the kernel's major components are adequately documented at the implementation level.

This lack of documentation makes it more difficult for new contributors, students, researchers, and even veteran contributors to understand the Linux kernel's implementation. For all of these people, implementation-level documentation of the Linux kernel provides many benefits. Obviously, those who wish to contribute to the Linux kernel must have a fairly good understanding of its actual implementation. But why is it valuable for students and researchers to understand the Linux kernel at the implementation level? Isn't the theory behind it or a general idea of what is going on enough? Since the Linux kernel is "developed with a strong practical emphasis more than a theoretical one" [6], many decisions are made in reaction to Linux's real-world performance. This means that it is quite common for Linux's implementation to diverge from theoretical foundations; when this happens, it is usually for a good reason. Understanding deployed algorithms, the reasoning behind divergences from theory, and the weaknesses in theories that real-world applications bring to light is essential for the development of future algorithms.

For the reasons listed above, Linux needs documentation specific to its implementation, not just the theory that may or may not have at one time been the basis for the design choices made by its developers. This paper on the Linux 2.6.8.1 scheduler was inspired by Mel Gorman's thesis on the Linux virtual memory (VM) system [6], which current Linux VM developers probably reference and value more than any other piece of documentation on the subject.

The goal of this paper is to provide in-depth documentation of the Linux 2.6.8.1 CPU scheduler. This documentation will hopefully be of use to kernel developers who must work with the code, as well as students and researchers who wish to understand the implementation of a real, working scheduler. Hopefully this paper will greatly reduce the amount of time required to gain a detailed understanding of how the Linux 2.6.8.1 scheduler works. In the same way that Mr. Gorman's documentation of the Linux 2.4.20 VM system is still very helpful in understanding the VM system in the Linux 2.6.x series of kernels, it is hoped that this paper will remain relevant for many versions of the Linux kernel beyond 2.6.8.1.

1.2 Linux Kernel Literature

While the Linux kernel lacks up-to-date code-level documentation, there is a reasonable amount of higher-level and introductory documentation available. Any of the following literature is highly recommended reading for those who wish to gain a basic knowledge of kernel internals.

Linux Kernel Development by Robert Love (a highly respected Linux kernel hacker) was released in 2004 [4] [1]. It covers the Linux 2.6.x kernel series, and as of fall 2004 it is perhaps the only book to do so (most others cover Linux 2.4.x and earlier). At 332 pages, it is quite manageable as a book to read page-by-page and to use as a reference. It gives a general overview of each of the Linux kernel's components, and helps to illustrate how they fit together. It contains a well-written overview of the Linux 2.6.x scheduler.

Robert Love's *Linux Kernel Development* may be the only book available that covers the Linux 2.6.x kernel, but there are several books available about the Linux 2.4.x kernel that may be helpful in understanding many components of the Linux 2.6.x kernels (some component have not changed drastically). Books providing such coverage include:

- *Understanding the Linux Kernel, 2nd Edition* by Daniel Bovet and Marco Cesati. O'Reilly, 2003.
- *Linux Device Drivers, 3rd Edition* by Jonathan Corbet, Alessandro Rubini and Greg Kroach-Hartman. O'Reilly, 2005.
- *IA-64 Linux Kernel: Design and Implementation* by David Mosberger and Stephane Eranian. Prentice Hall PTR, 2002.
- *Understanding the Linux Virtual Memory Manager* by Mel Gorman. Prentice Hall PTR, 2004. (*http://www.skynet.ie/˜mel/projects/vm/*)

The Linux Documentation Project (*http://www.tldp.org/*) is another good source of documentation. It contains documents covering many different aspects of Linux distributions and the Linux kernel.

Archives of all past conversation on the official Linux kernel development mailing list (LKML) are available on many web sites. Simply search for "LKML archive" using a search engine such as Google (*http://www.google.com/*). LKML should be read liberally and posted to conservatively.

Last but not least, the documentation distributed with the kernel source itself is quite helpful. It can be found in the `Documentation/` directory.

Unfortunately, Linux documentation covering kernels prior to the 2.6.x series will be of minimal use in understanding the scheduler described in this paper because the scheduler was heavily modified between the 2.4.x and 2.6.x kernel series.

1.3 Typographic Conventions

New concepts and URLs are *italicized*. Binaries, commands, and package names are in **bold**. Code, macros, and file paths are in a `constant-width font`. Paths to included files will be written with brackets around them (e.g. `<linux/sched.h>`); these files can be found in the `include/` directory of the Linux kernel source code. All paths are rooted in the Linux kernel source code unless otherwise noted. Fields in a structure are referred to with an arrow pointing from the structure to the field (e.g. `structure->field`).

1.4 About this Document

This document was written in LaTeX using the LyX editor on SuSE Linux 9.x and Mac OS X 10.3.x. It is made available in HTML, PDF, and LaTeX form. It can be downloaded from the author's web site (*http://josh.trancesoftware.com/linux/*).

1.5 Companion CD

The companion disc included with this document includes the full source code of the Linux 2.6.8.1 kernel, a patch to add in-depth comments to the scheduler code, and a digital copy of this document. The disc is an ISO-9660 formatted CD that should work in any modern operating system. To apply the scheduler comments patch, move it to the directory `kernel/` in your Linux source code, **cd** into that directory, and run the following command:

patch -p0 < sched_comments.patch

[1] The Second Edition of the book was published by Novell in 2005.

2 Linux Kernel Source Code

2.1 Getting the Source

The Linux kernel source code is an essential resource for learning about the kernel. In attempting to gain a detailed understanding of the kernel, no paper can entirely replace reading the code itself. This paper will refer to it heavily. The Linux kernel source code is available at The Linux Kernel Archives (*http://www.kernel.org*). The main page of the kernel archive lists the latest release from each kernel series, including complete source code, upgrade patches, and change logs. All released versions of the Linux kernel are available on the archive's FTP site (*ftp://ftp.kernel.org/*).

2.2 Kernel Versioning

Linux kernels have version numbers in the form W.X.Y.Z. The W position is rarely incremented - only when an extremely significant change has been made to the kernel, such that a considerable amount of software that works on one version won't work on another. This has only happened once in the history of Linux (thus the "2" at the beginning of the kernel version this paper focuses on, 2.6.8.1).

The X position denotes the kernel *series*. An even series indicates a stable release series, and an odd series denotes a development release series. Historically, the series number is incremented every couple of years. Development of older series' continues as long as there is interest. For example - though Linux 2.0 was originally released in June of 1996, version 2.0.40 was released in February of 2004 (largely by/for people who want to continue to support older hardware).

The Y position is the *version number*, which is normally incremented for every release. Often it is the last position in a kernel version (e.g. 2.6.7), but occasionally there is a need to fix something critical in a release. In such cases the Z position is incremented.[2] The first instance of this happening was the release of the 2.6.8.1 kernel. The 2.6.8 kernel contains a very serious bug in its Network File System (NFS) implementation. This was discovered very soon after its release, and thus 2.6.8.1 was released containing little more than a fix for that specific bug.

2.3 Code Organization

There are quite a few subdirectories within each Linux source code package. Subdirectories that it would be most helpful to know about while reading this paper are:

Documentation a directory containing lots of good documentation on kernel internals and the development process

arch a directory containing architecture-specific code; it contains one subdirectory for each supported architecture (e.g. i386, ia64, ppc64...)

include a directory containing header files

kernel a directory containing the main kernel code

mm a directory containing the kernel's memory management code

3 Overview of Processes and Threads

It is important to have a decent understanding of both processes and threads before learning about schedulers. Explaining processes and threads in depth is outside of the scope of this document, thus only a summary of the things that one must know about them is provided here. Readers of this document are strongly encouraged to gain an in-depth understanding of processes and threads from another source. Excellent sources are listed in the bibliography[2, 3, 4, 5].

[2]Starting from 2.8.11.1 in March 2005, there are some specific criteria for using the Z number, e.g. the patch must have less than 100 lines of code.

3.1 Programs and Processes

A *program* is a combination of instructions and data put together to perform a task when executed. A *process* is an instance of a program (what one might call a "running" program). An analogy is that programs are like classes in languages like C++ and Java, and processes are like objects (instantiated instances of classes). Processes are an abstraction created to embody the state of a program during its execution. This means keeping track of the data that is associated with a thread or threads of execution, which includes variables, hardware state (e.g. registers and the program counter, etc...), and the contents of an address space[3] [1].

3.2 Threads

A process can have multiple threads of execution that work together to accomplish its goals. These threads of execution are aptly named *threads*. A kernel must keep track of each thread's stack and hardware state, or whatever is necessary to track a single flow of execution within a process. Usually threads share address spaces, but they do not have to (often they merely overlap). It is important to remember that only one thread may be executing on a CPU at any given time, which is basically the reason kernels have CPU schedulers. An example of multiple threads within a process can be found in most web browsers. Usually at least one thread exists to handle user interface events (like stopping a page load), one thread exists to handle network transactions, and one thread exists to render web pages.

3.3 Scheduling

Multitasking kernels (like Linux) allow more than one process to exist at any given time, and furthermore each process is allowed to run as if it were the only process on the system. Processes do not need to be aware of any other processes unless they are explicitly designed to be. This makes programs easier to develop, maintain, and port [1]. Though each CPU in a system can execute only one thread within a process at a time, many threads from many processes appear to be executing at the same time. This is because threads are scheduled to run for very short periods of time and then other threads are given a chance to run. A kernel's scheduler enforces a thread scheduling policy, including when, for how long, and in some cases where (on *Symmetric Multiprocessing (SMP)* systems) threads can execute. Normally the scheduler runs in its own thread, which is woken up by a timer interrupt. Otherwise it is invoked via a system call or another kernel thread that wishes to yield the CPU. A thread will be allowed to execute for a certain amount of time, then a context switch to the scheduler thread will occur, followed by another context switch to a thread of the scheduler's choice. This cycle continues, and in this way a certain policy for CPU usage is carried out.

3.4 CPU and I/O-bound Threads

Threads of execution tend to be either *CPU-bound* or *I/O-bound* (Input/Output bound). That is, some threads spend a lot of time using the CPU to do computations, and others spend a lot of time waiting for relatively slow I/O operations to complete. For example - a thread that is sequencing DNA will be CPU bound. A thread taking input for a word processing program will be I/O-bound as it spends most of its time waiting for a human to type. It is not always clear whether a thread should be considered CPU or I/O bound. The best a scheduler can do is guess, if it cares at all. Many schedulers do care about whether or not a thread should be considered CPU or I/O bound, and thus techniques for classifying threads as one or the other are important parts of schedulers.

Schedulers tend to give I/O-bound threads priority access to CPUs. Programs that accept human input tend to be I/O-bound - even the fastest typist has a considerable amount of time between each keystroke during which the program he or she is interacting with is simply waiting. It is important to give programs that interact with humans priority since a lack of speed and responsiveness is more likely to be perceived when a human is expecting an immediate response than when a human is waiting for some large job to finish.

It is also beneficial to the system as a whole to give priority to programs that are I/O-bound but not because of human input[4]. Because I/O operations usually take a long time it is good to get them started

[3] An address space is the set of memory addresses that a process is allowed to read and/or write to.

[4] It is fortunate that both human-interactive and non-human-interactive I/O activity should be awarded a higher priority since there is really no way to tell at the scheduler level what I/O was human-initiated and what was not. The scheduler does

as early as possible. For example, a program that needs a piece of data from a hard disk has a long wait ahead before it gets its data. Kicking off the data request as quickly as possible frees up the CPU to work on something else during the request and helps the program that submitted the data request to be able to move on as quickly as possible. Essentially, this comes down to parallelizing system resources as efficiently as possible. A hard drive can seek data while a CPU works on something else, so having both resources working as early and often as possible is beneficial. Many CPU operations can be performed in the time it takes to get data from a hard drive.

3.5 Context Switching

Context switching is the process of switching from one thread of execution to another. This involves saving the state of the CPU's registers and loading a new state, flushing caches, and changing the current virtual memory map. Context switches on most architectures are a relatively expensive operation and as such they are avoided as much as possible. Quite a bit of actual work can be done during the time it takes to perform a context switch. How context switching is handled is highly architecture-dependent and is not really part of a kernel's scheduler, though the way it is done can greatly influence a scheduler's design. Context switching code in the Linux kernel is defined in the files `include/asm-[arch]/mmu_context.h` (change current virtual memory mapping) and `include/asm-[arch]/system.h` (perform CPU context switch, e.g. PC and general registers).

3.6 Linux Processes/Threads

Linux takes a unique approach to implementing the process and thread abstractions. In Linux, all threads are simply processes that might share certain resources. Instead of being something different than a thread or a group of threads, a process in Linux is simply a group of threads that share something called a ***thread group ID (TGID)*** and whatever resources are necessary. In order to reconcile Linux's treatment of processes and threads with the terms themselves, the term "task" will be used from here on to mean a Linux thread - it does not mean thread in the POSIX sense. "Process" or "thread" will be used only when the difference really matters. In the Linux task structure `task_struct` (one of which exists for each thread), the TGID that is a process's POSIX PID is stored as `[task_struct]->tgid`. Linux assigns unique "PID"s to every thread (`[task_struct]->pid`), but the (POSIX) PID that most people think of is really a task's TGID. It is worth mentioning that this model, combined with certain tricks like a COW (Copy On Write) forking algorithm[5] causes process and thread spawning to be very fast and efficient in Linux, whereas spawning a process is much more expensive than spawning threads[6] on many other operating systems (e.g. BSD UNIX® and Microsoft® Windows®).

Unfortunately, further details about Linux's process and thread implementation would be out of the scope of this paper. It is only important to know that Linux considers processes to be merely groups of threads and does not differentiate between the two. Because of this, Linux schedules threads only, essentially ignoring what POSIX processes they belong to.

4 Linux Scheduling Goals

4.1 Linux's Target Market(s) and their Effects on its Scheduler

An operating system's scheduling algorithm is largely determined by its target market, and vice-versa. Understanding an operating system's target market helps to explain its scheduling goals, and thus its scheduling algorithm.

not know whether a program is blocked waiting for keyboard input or it is blocked waiting for data from a hard drive.

[5]Normally a call to `fork()` causes a copy of the caller's resources to be created and labeled as a child. Copy On Write means that the resources are not actually copied until the child's resources differ from the parent's (i.e. the child or parent tries to write to some shared data). Even then, only the differing resources are copied and thus no longer shared. This saves time in the usual case where `fork()`is immediately followed by a call to `exec()` because if `fork()` did not use COW, a copy of the parent's executable data (text section) would be created only to be overwritten by new data taken in during the `exec()` call.

[6]Operating systems that differentiate between process and thread spawning often referred to threads as lightweight processes (LWPs).

Linux was originally created by Linus Torvalds for use on his personal computer. However, despite its origins, Linux has become known as a server operating system. There are many reasons for this, not the least of which is the fact that most software designed to run on top of the Linux kernel is meant for users with a relatively high skill level or inherits design qualities targeting more skilled users. This led to Linux's notoriously complex and unrefined graphical user interface options (compared to Apple® and Microsoft® operating systems) and subsequent relegation to the server room. Linux's exposure in the server market guided its development along the lines of the one market that it initially succeeded in. Linux's prowess as a server operating system is nowadays perhaps matched only by a few operating systems such as Sun's Solaris and IBM's AIX. However, cost and legal advantages are causing many companies to replace both of those operating systems with Linux as well.

While Linux has made a name for itself in the server operating systems arena, many users and developers believe that it can also be a success on the desktop. In the last several years, there has been a push to optimize the Linux kernel for the desktop market. Perhaps the biggest step in that direction was the scheduler written by Ingo Molnar for the 2.6.x kernel series. Molnar designed his scheduler with the desktop and the server market in mind, and as a result desktop performance is much improved in Linux distributions based on 2.6.x kernels. Targeting both the server and the desktop market imposes particularly heavy demands on the kernel's scheduler, and thus the Linux kernel's scheduler is an interesting case study in how to please two very different markets at the same time.

4.2 Efficiency

An important goal for the Linux scheduler is efficiency. This means that it must try to allow as much real work as possible to be done while staying within the restraints of other requirements. For example - since context switching is expensive, allowing tasks to run for longer periods of time increases efficiency. Also, since the scheduler's code is run quite often, its own speed is an important factor in scheduling efficiency. The code making scheduling decisions should run as quickly and efficiently as possible. Efficiency suffers for the sake of other goals such as interactivity, because interactivity essentially means having more frequent context switches. However, once all other requirements have been met, overall efficiency is the most important goal for the scheduler.

4.3 Interactivity

Interactivity is an important goal for the Linux scheduler, especially given the growing effort to optimize Linux for desktop environments. Interactivity often flies in the face of efficiency, but it is very important nonetheless. An example of interactivity might be a keystroke or mouse click. Such events usually require a quick response (i.e. the thread handling them should be allowed to execute very soon) because users will probably notice and be annoyed if they do not see some result from their action almost immediately. Users don't expect a quick response when, for example, they are compiling programs or rendering high-resolution images. They are unlikely to notice if something like compiling the Linux kernel takes an extra twenty seconds. Schedulers used for interactive computing should be designed in such a way that they respond to user interaction within a certain time period. Ideally, this should be a time period that is imperceptible to users and thus gives the impression of an immediate response.

4.4 Fairness and Preventing Starvation

It is important for tasks to be treated with a certain degree of fairness, including the stipulation that no thread ever starves. Starvation happens when a thread is not allowed to run for an unacceptably long period of time due to the prioritization of other threads over it. Starvation must not be allowed to happen, though certain threads should be allowed to have a considerably higher priority level than others based on user-defined values and/or heuristic indicators. Somehow, threads that are approaching the starvation threshold (which is generally defined by a scheduler's implementors) must get a significant priority boost or one-time immediate preemption before they starve. Fairness does not mean that every thread should have the same degree of access to CPU time with the same priority, but it means that no thread should ever starve or be able to trick the scheduler into giving it a higher priority or more CPU time than it ought to have.

4.5 SMP Scheduling

Since the Linux kernel supports multiprocessing, its scheduler must work (and work well for that matter) when scheduling tasks across more than one CPU on the same motherboard. This means keeping track of which tasks are running on which CPUs, making sure any given task is not executing on more than one CPU at a time, and in general doing all of the accounting necessary to efficiently schedule tasks across multiple CPUs. Since all CPUs generally access the same memory and system resources, the scheduler is primarily concerned with making the best use of processor time. There is little reason to prefer one CPU over another in terms of choosing where to schedule a task. The most conspicuous consideration is caching - by scheduling a given task on the same CPU as often as possible, the likelihood of that CPU's cache being hot increases.

4.6 SMT Scheduling

The Linux kernel supports scheduling multiple threads on a single *Symmetric Multi-Threading (SMT)* chip. While the concept of SMT has been around for some time, Intel's Hyper-Threading (HT) technology made SMT technology mainstream. Essentially, each physical SMT chip can have more than one virtual processor with the caveat that the virtual processors share certain resources (e.g. some types of cache). Because certain resources are shared, virtual processors should not be treated in the same way that regular processors are.

4.7 NUMA Scheduling

The Linux kernel supports *Non-Uniform Memory Access (NUMA)*, which means it can run a single system image across more than one node if such hardware is present (essentially a node is defined as a motherboard). At a hardware level, a node is something like a traditional uniprocessor or multiprocessor machine in that it has its own CPU(s) and memory. However, NUMA systems treat multiple nodes as parts of a single system running a single system image (i.e. one instance of the Linux kernel). This is usually accomplished through some sort of high-speed interconnect (such as SGI's NUMAlink technology), which connects nodes at a more of a motherboard level than at a networking level. This means that all CPUs are capable of executing any thread, and all of the memory across nodes is accessible via the same address space (i.e. any CPU can allocate memory on any node on the system). NUMA support involves being aware of cache issues similar to those in SMP scheduling, but can also include issues of memory locality (i.e. if a CPU is executing a thread which is allocating memory from a local memory bank, it would be inefficient to move the thread across the system as memory requests would take longer to fulfill). Perhaps the biggest issue that a scheduler supporting NUMA needs to tackle is the possibility of having far more CPUs to schedule on than most SMP systems. Common SMP systems might have anywhere from 2-8 processors, but NUMA systems might have hundreds of processors. At the time of this writing, SGI is shipping NUMA systems containing 512 processors. This is the largest number of processors any company has been able to run under a single Linux system image, and the limit to which the Linux 2.6.8.1 scheduler has been stretched.

4.8 Soft Real-Time Scheduling

The Linux scheduler supports soft *real-time (RT)* scheduling. This means that it can effectively schedule tasks that have strict timing requirements. However, while the Linux 2.6.x kernel is usually capable of meeting very strict RT scheduling deadlines, it does not guarantee that deadlines will be met. RT tasks are assigned special scheduling modes and the scheduler gives them priority over any other task on the system. RT scheduling modes include a first-in-first-out (FIFO) mode which allows RT tasks to run to completion on a first-come-first-serve basis, and a round-robin scheduling mode that schedules RT tasks in a round-robin fashion while essentially ignoring non-RT tasks on the system.

4.9 Scheduling Performance Perspectives

In terms of schedulers, there is no single definition of performance that fits everyone's needs; that is, there is not a single performance goal for the Linux scheduler to strive for. The many definitions of good scheduling performance often lead to a give-and-take situation, such that improving performance in one sense hurts performance in another. Some improvements to the Linux scheduler help performance all-around, but such improvements are getting more and more hard to come by. A good example of a give-and-take performance issue is desktop vs. server vs. *high performance computing (HPC)* performance.

The most important performance metric for desktop users is perceived performance - that is, how fast does a machine seem to respond to requests such as mouse clicks and key presses. If a user is compiling a kernel in the background and typing in a word processor in the foreground, he or she is unlikely to notice if the kernel compile takes an extra minute because it is constantly interrupted by the word processor responding to keystrokes. What matters most to the users is that when he or she presses a key, the word processor inserts and displays the desired character as quickly as possible. This entails a CPU making a context switch to the word processor's thread as soon as possible after the user presses a key. In order for this to happen, the currently running thread must either give up the processor before its timeslice is up, or its timeslice must be short enough that the delay between the time the keystroke happens and the timeslice ends is imperceptible to the user. Since context switching is expensive, context switches must be minimized while happening frequently enough to provide good perceived performance to interactive users (e.g. word processors). Fewer context switches means better real efficiency, since more time is spent doing actual work and less is spent switching tasks. More context switches means the system is more responsive to user input. On interactive desktop systems, the desired behavior is to have context switching happen often enough that user input seems to get an immediate response without happening so often that the machine becomes very inefficient.

Server systems generally focus less on perceived performance than desktop systems. They are relatively more concerned with actual performance; that is, reducing the overall amount of time it takes to complete a set of tasks. Since users are normally willing to put up with a longer response delay (e.g. they are willing to wait longer for a web page to be transmitted over the network than they are for a keystroke to cause a character to appear in a word processing document), more of an emphasis is placed on overall efficiency via fewer context switches. If three complex database queries on a database loaded into memory happen at the same time, it is most likely better to get them done faster overall than it is to do them inefficiently for the sake of returning results at the same time and thus lowering the average response time. People and applications submitting complex database queries generally have much lower response time expectations than people who are typing characters into a word processor. However, if, for example, two massive files are requested from an FTP server, it would be unacceptable for the server to completely finish sending one file before beginning to send the other (the most extreme but perhaps overall most efficient case, potential I/O concerns aside). Thus server systems, while having lower response time requirements than desktop systems, are still expected to operate within some responsiveness expectations.

HPC systems generally require the least immediate response times as they tackle very large problems that can take days to solve. Given a set of tasks, overall efficiency is the imperative and this means that context switches for the sake of responsiveness must be minimized (or perhaps all but done away with?). Response time expectations are generally the lowest for HPC applications, and thus they represent the true opposite of desktop computing performance ideals. Servers tend to be somewhere in the middle.

This comparison illustrates the point that there is no universal ideal for scheduler performance. A scheduler that seems superb to a desktop user might be a nightmare for someone running HPC applications. The Linux scheduler strives to perform as well as possible in all types of situations, though it is impossible for it to perform ideally for everyone. Desktop users are constantly crying out for more tuning for their needs while at the same time HPC users are pushing for optimization towards their performance ideal.

5 The Linux 2.6.8.1 Scheduler

5.1 Origins and the Significance of an O(1) Scheduling Algorithm

5.1.1 Origins of the Linux 2.6.8.1 Scheduler

During the Linux 2.5.x development period, a new scheduling algorithm was one of the most significant changes to the kernel. The Linux 2.4.x scheduler, while widely used, reliable, and in general pretty good, had several very undesirable characteristics (see section 6). The undesirable characteristics were quite embedded in its design, and thus when Ingo Molnar rose to the challenge of fixing it he produced an entirely new scheduler instead of making modifications to the old one. The fact that the Linux 2.4.x scheduling algorithm contained O(n) algorithms was perhaps its greatest flaw, and subsequently the new scheduler's use of only

O(1) algorithms was its most welcome improvement.

5.1.2 What is an O(1) Algorithm

An algorithm operates on input, and the size of that input usually determines its running time. Big-O notation is used to denote the growth rate of an algorithm's execution time based on the amount of input. For example - the running time of an O(n) algorithm increases linearly as the input size n grows. The running time of an O(nˆ2) grows quadratically. If it is possible to establish a constant upper bound on the running time of an algorithm, it is considered to be O(1) (one might say it runs in "constant time"). That is, an O(1) algorithm is guaranteed to complete in a certain amount of time regardless of the size of the input.

5.1.3 What Makes the Linux 2.6.8.1 Scheduler Perform in O(1) Time

The Linux 2.6.8.1 scheduler does not contain any algorithms that run in worse than O(1) time. That is, every part of the scheduler is guaranteed to execute within a certain constant amount of time regardless of how many tasks are on the system. This allows the Linux kernel to efficiently handle massive numbers of tasks without increasing overhead costs as the number of tasks grows. There are two key data structures in the Linux 2.6.8.1 scheduler that allow for it to perform its duties in O(1) time, and its design revolves around them - runqueues and priority arrays.

5.2 Runqueues

5.2.1 Overview

The runqueue data structure is the most basic structure in the Linux 2.6.8.1 scheduler; it is the foundation upon which the whole algorithm is built. Essentially, a runqueue keeps track of all runnable tasks assigned to a particular CPU. As such, one runqueue is created and maintained for each CPU in a system. Each runqueue contains two priority arrays, discussed in section 5.3. All tasks on a CPU begin in one priority array, the active one, and as they run out of their timeslices they are moved to the expired priority array. During the move, a new timeslice is calculated. When there are no more runnable tasks in the active priority arrays, it is simply swapped with the expired priority array (which entails simply updating two pointers). The job of the runqueue is to keep track of a CPU's special thread information (idle thread, migration thread) and to handle its two priority arrays.

5.2.2 Data Structure

The runqueue data structure is defined as a struct in `kernel/sched.c`. It is not defined in `kernel/sched.h` because abstracting the scheduler's inner workings from its public interface is an important architectural goal. The runqueue struct contains the following variables:

- `spinlock_t lock`

This is the lock that protects the runqueue. Only one task can modify a particular runqueue at any given time.

- `unsigned long nr_running`

The number of runnable tasks on the runqueue.

- `unsigned long cpu_load`

The load of the CPU that the runqueue represents. The load is recalculated whenever `rebalance_tick()` is called, and is the average of the old load and the current `(nr_running * SCHED_LOAD_SCALE)`. The latter macro simply increases the resolution of the load average.

- `unsigned long long nr_switches`

The number of context switches that have occurred on a runqueue since its creation. This value isn't actually used for anything useful in the kernel itself - it is simply exposed in the proc filesystem as a statistic.

- `unsigned long expired_timestamp`

Time since last priority array swap (active <–> expired).

- `unsigned long nr_uninterruptible`

Number of uninterruptible tasks on the runqueue.

- `unsigned long long timestamp_last_tick`

Timestamp of last scheduler tick. Primarily used in the task_hot macro, which decides whether a task should be considered cache hot or not (i.e. is some of the task's data likely to still be in CPU caches).

- `task_t *curr`

Pointer to the currently running task.

- `task_t *idle`

Pointer to a CPU's idle task (i.e. the task that runs when nothing else is running).

- `struct mm_struct *prev_mm`

Pointer to the virtual memory mapping of the previously running task. This is used in efficiently handling virtual memory mappings in terms of cache hotness.

- `prio_array_t *active`

The active priority array. This priority array contains tasks that have time remaining from their timeslices.

- `prio_array_t *expired`

The expired priority array. This priority array contains tasks that have used up their timeslices.

- `prio_array_t arrays[2]`

The actual two priority arrays. Active and expired array pointers switch between these.

- `int best_expired_prio`

The highest priority of any expired task. Used in the EXPIRED_STARVING macro to determine whether or not a task with a higher priority than the currently running task has expired.

- `atomic_t nr_iowait`

The number of tasks on a runqueue waiting on I/O. Used for kernel stats (i.e. is a CPU waiting on I/O or is it just idle?).

- `struct sched_domain *sd`

The scheduler domain that a runqueue belongs to. Essentially this is a group of CPUs that can share tasks between them. See section 5.8.2 for more information.

- `int active_balance`

Flag used by the migration thread to determine whether or not a runqueue needs to be balanced (i.e. whether or not it is considerably busier than others).

- `int push_cpu`

The CPU that a runqueue should be pushing tasks to when being balanced.

- `task_t *migration_thread`

A CPU's migration thread. The migration thread is the thread that looks after task migration concerns (i.e. does this CPU need to be balanced).

- `struct list_head migration_queue`

List of tasks that need to be migrated to other CPUs.

5.2.3 Locking

Only one task may modify a CPU's runqueue at any given time, and as such any task that wishes to modify a runqueue must obtain its lock first. Obtaining multiple runqueue locks must be done by order of ascending runqueue address in order to avoid deadlocks. A convenient function for obtaining two runqueue locks is `double_rq_lock(rq1, rq2)`, which handles lock ordering itself. Its opposite, `double_rq_unlock(rq1, rq2)`, does the same but unlocks instead of locks. Locking a runqueue that a certain task is in can be done with `task_rq_lock(task, &flags)`.

5.3 Priority Arrays

5.3.1 Overview

This data structure is the basis for most of the Linux 2.6.8.1 scheduler's advantageous behavior, in particular its O(1) (constant) time performance. The Linux 2.6.8.1 scheduler always schedules the highest priority task on a system, and if multiple tasks exist at the same priority level, they are scheduled round-robin with each other. Priority arrays make finding the highest priority task in a system a constant-time operation, and also makes round-robin behavior within priority levels possible in constant-time. Furthermore, using two priority arrays in unison (the active and expired priority arrays) makes transitions between timeslice epochs a constant-time operation. An epoch is the time between when all runnable tasks begin with a fresh timeslice and when all runnable tasks have used up their timeslices.

5.3.2 Data Structure

- `unsigned int nr_active`

The number of active tasks in the priority array.

- `unsigned long bitmap[BITMAP_SIZE]`

The bitmap representing the priorities for which active tasks exist in the priority array. For example - if there are three active tasks, two at priority 0 and one at priority 5, then bits 0 and 5 should be set in this bitmap. This makes searching for the highest priority level in the priority array with a runnable task as simple as a constant-time call to `__ffs()`, a highly optimized function for finding the highest order bit in a word (`sched_find_first_bit()` is essentially a wrapper for `__ffs()`).

- `struct list_head queue[MAX_PRIO]`

An array of linked lists. There is one list in the array for each priority level (`MAX_PRIO`). The lists contain tasks, and whenever a list's size becomes > 0, the bit for that priority level in the priority array bitmap is set. When a task is added to a priority array, it is added to the list within the array for its priority level. The highest priority task in a priority array is always scheduled first, and tasks within a certain priority level are scheduled round-robin.

5.3.3 How Priority Arrays are Used

Among tasks with timeslice remaining, the Linux 2.6.8.1 scheduler always schedules the task with the highest priority (timeslice is essentially the period of time a task is allowed to execute before other tasks are given a chance - see section 5.4). Priority arrays allow the scheduler's algorithm to find the task with the highest priority in constant time.

Priority arrays are an array of linked lists, one for each priority level (in Linux 2.6.8.1 there are 140 priority levels). When a task is added to a priority array, it is added to the list for its priority level. A bitmap of size `MAX_PRIO + 1` (actually it might be a bit larger since it must be implemented in word-sized chunks) has bits set for each priority level that contains active tasks. In order to find the highest priority task in a priority array, one only has to find the first bit set in the bitmap. Multiple tasks of the same priority are scheduled round-robin; after running, tasks are put at the bottom of their priority level's list. Because finding the first bit in a finite-length bitmap and finding the first element in a list are both operations with a finite upper bound on how long the operation can take, this part of the scheduling algorithm performs in constant, O(1) time.

When a task runs out of timeslice, it is removed from the active priority array and put into the expired priority array. During this move, a new timeslice is calculated. When there are no more runnable tasks in the active priority array, the pointers to the active and expired priority arrays are simply swapped. Because timeslices are recalculated when they run out, there is no point at which all tasks need new timeslices calculated for them; that is, many small constant-time operations are performed instead of iterating over however many tasks there happens to be and calculating timeslices for them (which would be an undesirable O(n) time algorithm). Swapping the active and expired priority array pointers is a constant time operation, which avoids the O(n) time operation of moving n tasks from one array or queue to another.

Since all operations involved in the maintenance of a system of active and expired priority arrays occur constant O(1) time, the Linux 2.6.8.1 scheduler performs quite well. The Linux 2.6.8.1 scheduler will perform its duties in the same small amount of time no matter how many tasks are on a system.

5.4 Calculating Priority and Timeslice

5.4.1 Static Task Prioritization and the `nice()` System Call

All tasks have a static priority, often called a *nice* value. On Linux, nice values range from -20 to 19, with higher values being lower priority (tasks with high nice values are nicer to other tasks). By default, tasks start with a static priority of 0, but that priority can be changed via the `nice()` system call. Apart from its initial value and modifications via the `nice()` system call, the scheduler never changes a task's static priority. Static priority is the mechanism through which users can modify task's priority, and the scheduler will respect the user's input (in an albeit relative way).

A task's static priority is stored in its `static_prio` variable. Where `p` is a task, `p->static_prio` is its static priority.

5.4.2 Dynamic Task Prioritization

The Linux 2.6.8.1 scheduler rewards I/O-bound tasks and punishes CPU-bound tasks by adding or subtracting from a task's static priority. The adjusted priority is called a task's dynamic priority, and is accessible via the task's prio variable (e.g. `p->prio` where `p` is a task). If a task is interactive (the scheduler's term for I/O bound), its priority is boosted. If it is a CPU hog, it will get a penalty. In the Linux 2.6.8.1 scheduler, the maximum priority bonus is 5 and the maximum priority penalty is 5. Since the scheduler uses bonuses and penalties, adjustments to a task's static priority are respected. A mild CPU hog with a nice value of -2 might have a dynamic priority of 0, the same as a task that is neither a CPU nor an I/O hog. If a user changes either's static priority via the `nice()` system call, a relative adjustment will be made between the two tasks.

5.4.3 I/O-bound vs. CPU-bound Heuristics

Dynamic priority bonuses and penalties are based on interactivity heuristics. This heuristic is implemented by keeping track of how much time tasks spend sleeping (presumably blocked on I/O) as opposed to running. Tasks that are I/O-bound tend to sleep quite a bit as they block on I/O, whereas CPU-bound task rarely sleep as they rarely block on I/O. Quite often, tasks are in the middle, and are not entirely CPU-bound or I/O-bound so the heuristic produces some sort of scale instead of a simple binary label (I/O-bound or CPU-bound). In the Linux 2.6.8.1 scheduler, when a task is woken up from sleep, its total sleep time is added to its `sleep_avg` variable (though a task's `sleep_avg` is not allowed to exceed `MAX_SLEEP_AVG` for the sake of mapping sleep_avg onto possible bonus values). When a task gives up the CPU, voluntarily or involuntarily, the time the current task spent running is subtracted from its `sleep_avg`. The higher a task's `sleep_avg` is, the higher its dynamic priority will be. This heuristic is quite accurate since it keeps track of both how much time is spent sleeping as well as how much time is spent running. Since it is possible for a task to sleep quite a while and still use up its timeslice, tasks that sleep for a long time and then hog a CPU must be prevented from getting a huge interactivity bonus. The Linux 2.6.8.1 scheduler's interactivity heuristics prevent this because a long running time will offset the long sleep time.

5.4.4 The `effective_prio()` Function

The `effective_prio()` function calculates a task's dynamic priority. It is called by `recalc_task_prio()`, the thread and process wakeup calls, and `scheduler_tick()`. In all cases, it is called after a task's `sleep_avg` has been modified, since `sleep_avg` is the primary heuristic for a task's dynamic priority.

The first thing effective_prio does is return a task's current priority if it is a RT task. The function does not give bonuses or penalties to RT tasks. The next two lines are key:

```
bonus = CURRENT_BONUS(p) - MAX_BONUS / 2;
prio = p->static_prio - bonus;
```

`CURRENT_BONUS` is defined as follows:

```
#define CURRENT_BONUS(p) \
NS_TO_JIFFIES((p)->sleep_avg) * MAX_BONUS / MAX_SLEEP_AVG)
```

Essentially, `CURRENT_BONUS` maps a task's sleep average onto the range 0-`MAX_BONUS`, which is 0-10. If a task has a high `sleep_avg`, the value returned by `CURRENT_BONUS` will be high, and vice-versa. Since `MAX_BONUS` is twice as large as a task's priority is allowed to rise or fall (`MAX_BONUS` of 10 means that the priority adjustment can be from +5 to -5), it is divided by two and that value is subtracted from `CURRENT_BONUS(p)`. If a task has a high `sleep_avg` and `CURRENT_BONUS(p)` returns 10, then the bonus variable would be set to 5. Subsequently, the task's static priority would get 5 subtracted from it, which is the maximum bonus that a task can get. If a task had a `sleep_avg` of 0, its CURRENT_BONUS(p) value might be 0. In that case, the bonus value would get set to -5 and the task's static priority would get -5 subtracted from it, which is the same as adding 5. Adding five is the maximum penalty a task's priority can get, which is the desired behavior for a CPU hog that never sleeps.

Once a new dynamic priority has been calculated, the last thing that `effective_prio()` does is within the non-RT priority range. For example - if a highly interactive task has a static priority of -20, it cannot be given a 5 point bonus since it already has the maximum non-RT priority.

5.4.5 Calculating Timeslice

Timeslice is calculated by simply scaling a task's static priority onto the possible timeslice range and making sure a certain minimum and maximum timeslice is enforced[7]. The higher the task's static priority (the lower the task's `static_prio` value) the larger the timeslice it gets. The `task_timeslice()` function is simply a call to the `BASE_TIMESLICE` macro which is defined as:

```
#define BASE_TIMESLICE(p) (MIN_TIMESLICE + \
((MAX_TIMESLICE - MIN_TIMESLICE) * \
(MAX_PRIO-1 - (p)->static_prio) / (MAX_USER_PRIO-1)))
```

Essentially, this is the minimum timeslice plus the the task's static priority scaled onto the possible timeslice range, `(MAX_TIMESLICE - MIN_TIMESLICE).`

It is important to remember that an interactive task's timeslice may be broken up into chunks, based on the `TIMESLICE_GRANULARITY` value in the scheduler. The function `scheduler_tick()` checks to see if the currently running task has been taking the CPU from other tasks of the same dynamic priority for too long (`TIMESLICE_GRANULARITY`). If a task has been running for `TIMESLICE_GRANULARITY` and task of the same dynamic priority exists a round-robin switch between other tasks of the same dynamic priority is made.

5.4.6 Fairness when Forking New Tasks

When new tasks (threads or processes) are forked, the functions `wake_up_forked_thread()` and `wake_up_forked_process()` decrease the sleep_avg of both parents and children. This prevents highly interactive tasks from spawning other highly interactive tasks. Without this check, highly interactive tasks could keep spawning new tasks in order to hog the CPU. With this check, `sleep_avg` and subsequently priority are decreased, increasing the

[7] In Robert Love's "Linux Kernel Development," he incorrectly states that timeslice is calculated based on dynamic priority. While his statement is fundamentally incorrect, Con Kolivas pointed out in an IRC conversation with the author that a loose enough interpretation (too loose, however) can argue that dynamic priority does affect timeslices. This is because if dynamic priority is high enough (a task is interactive enough), timeslices maybe be broken into chunks based on `TIMESLICE_GRANULARITY` so that a tasks cannot hog the CPU from other tasks of the same dynamic priority (see the `scheduler_tick()` function). However, the total timeslice is calculated only with a task's static priority, and breaking the timeslice up during an epoch is another issue.

likelihood that both parent and child will be preempted by a higher priority task. Note that timeslice does not decrease for parent or child since timeslice is based only on static priority and not the dynamic priority that is influenced by `sleep_avg`.

5.4.7 Interactive Task Reinsertion

Every 1ms, a timer interrupt calls `scheduler_tick()`. If a task has run out of timeslice, it is normally given a new timeslice and put on the expired priority array for its runqueue. However, `scheduler_tick()` will reinsert interactive tasks into the active priority array with their new timeslice so long as nothing is starving in the expired priority array. This helps interactive performance by not letting interactive tasks sit in the expired array while non-interactive tasks use up their timeslices (which might be a long time since non-interactive tasks tend to be CPU hogs).

5.4.8 Interactivity Credits

Interactive credits help to control the rise and fall rate of the interactive status of tasks. Essentially, tasks get an interactive credit when they sleep for a long time, and lose an interactive credit when they run for a long time. A task's interactive credit value is stored in its `interactive_credit` variable. If a task has more than 100 interactivity credits it is considered to have high interactivity credit. If a task has less then -100 interactivity credits it is considered to have low interactivity credit. Interactive credits matter in the following situations:

1. Low interactivity credit tasks waking from uninterruptible sleep are limited in their sleep_avg rise since they are probably CPU hogs waiting on I/O. A CPU hog that only occasionally waits on I/O should not gain an interactive sleep_avg level just because it waits for a long time once.

2. High interactivity credit tasks get less run time subtracted from their sleep_avg in order to prevent them from losing interactive status too quickly. If a task got high credit, it must have slept quite a bit at least 100 times recently and thus it should not lose interactive status just because it used up a lot of CPU time once.

3. Low interactivity credit tasks can only get one timeslice worth of sleep_avg bonus during dynamic priority recalculation (`recalc_task_prio()`). They must not have been sleeping too much recently in order to have low interactivity credit and thus they should not get too much of a bonus as they will probably hog the CPU.

5.5 Sleeping and Waking Tasks

5.5.1 Why Sleep?

Tasks do not always want to run, and when this is the case they go to sleep (or "block"). Tasks sleep for many reasons; in all cases they are waiting for some event to occur. Sometimes tasks sleep while they wait for data from a device (e.g. a keyboard, a hard drive, an ethernet card), sometimes they sleep while waiting for a signal from another piece of software, and sometimes they sleep for a certain amount of time (e.g. waiting while trying to obtain a lock).

Sleeping is a special state in which tasks cannot be scheduled or run, which is important since if they could get scheduled or run execution would proceed when it shouldn't and sleeping would have to be implemented as a busy loop. For example - if a task could be run after requesting data from a hard drive but before it was sure the data had arrived, it would have to constantly check (via a loop) to see whether or not the data had arrived.

5.5.2 Interruptible and Uninterruptible States

When a task goes to sleep, it is usually in one of two states - interruptible or uninterruptible. A task in the interruptible state can wake up prematurely to respond to signals while tasks in the uninterruptible state cannot. For example - if a user uses the **kill** command on a task, the **kill** command will attempt to do its job by sending a SIGTERM signal to the task. If the task is in the uninterruptible state, it will ignore the signal until the event it was originally waiting for occurs. Tasks in the interruptible state would respond to the signal immediately (though their response won't necessarily be to die as the user probably wants).

5.5.3 Waitqueues

A waitqueue is essentially a list of tasks waiting for some event or condition to occur. When that event occurs, code controlling that event will tell the waitqueue to wake up all of the tasks in it. It is a centralized place for event notifications to be "posted." Sleeping tasks are added to waitqueues before going to sleep in order to be woken up when the event they are waiting for occurs.

5.5.4 Going to Sleep

Tasks put themselves to sleep by making system calls, and those system calls usually take something like the following steps to ensure a safe and successful sleep period[4]:

1. Create a wait queue via `DECLARE_WAITQUEUE()`.
2. Add task to the wait queue via `add_wait_queue()`. The wait queue will wake up any added tasks when the condition they are waiting for happens. Whatever code is making that condition true will need to call `wake_up()` on the waitqueue when appropriate.
3. Mark task as sleeping, either `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`.
4. Begin a loop that calls `schedule()` with a test to see if the condition is true or not. If it is true initially then schedule() will not be called because sleeping is unnecessary. Otherwise, call `schedule()` to give up the CPU. Since the task has been marked as sleeping, it will not be scheduled again (until it wakes up).
5. When the task wakes up, the loop's condition check will be executed again. This will prevent spurious wakeups, which can happen. If the condition has occurred, the loop will exit. Otherwise it will repeat and call `schedule()` again.
6. Once the condition is true, mark task as `TASK_RUNNING` and remove it from the wait queue via `remove_wait_queue()`.

5.5.5 Waking Up

The `try_to_wake_up()` function is responsible for trying to wake up tasks. When a waitqueue is told to wake up, `try_to_wake_up()` is called on each task in the waitqueue, and then tasks are removed from the waitqueue. The task is marked `TASK_RUNNING`, and then it is added back to the appropriate runqueue to be scheduled again.

5.6 The Main Scheduling Function

5.6.1 Overview

The `schedule()` function is the main scheduler function. Its job is to pick a new task to run and switch to it. It is called whenever a task wishes to give up the CPU voluntarily (often through the `sys_sched_yield()` system call), and if `scheduler_tick()` sets the `TIF_NEED_RESCHED` flag on a task because it has run out of timeslice, then `schedule()` will get called when preempts are re-enabled[8]. `scheduler_tick()` is a function called during every system time tick, via a clock interrupt. It checks the state of the currently running task and other tasks in a CPU's runqueue to see if scheduling and load balancing is necessary (and will invoke them if so).

5.6.2 The `schedule()` Function

The first thing that schedule does is check to make sure it's not being called when it shouldn't be (during an atomic period). After that, it disables preemption and determines the length of time that the task to be scheduled out has been running. That time is then reduced if a task has high interactivity credit since it would be undesirable for a task that usually waits on I/O to lose interactivity status due to a single long period of CPU usage. Next, if the function is entered off of a kernel preemption interruptible tasks with a signal pending get a state of `TASK_RUNNING` and uninterruptible tasks get removed from the runqueue. This is because if a task can be interrupted and it has a signal pending, it needs to handle that signal. Tasks that are not interruptible should not be on the runqueue.

At this point, it is time to look for the next task to run. If there are no runnable tasks in the runqueue, an attempt at load balancing is made. If balancing does not bring any runnable tasks, then a switch to the idle task is made. If there are runnable tasks in the runqueue but not in the active priority array, then the active and retired priority arrays are swapped.

At this point there is a runnable task in the active priority array. Next, the active priority array's bitmap is checked to find the highest priority level with a runnable task. After that, dependent sleeping tasks on virtual SMT CPU's are given a chance to run. If there is a dependent sleeper (which might only happen on an SMT system), the current CPU (which is a virtual CPU sharing physical CPU resources with the virtual CPU that has a dependent sleeper) switches to idle so the dependent sleeper can wake up and do what it needs to do.

If there has not been a switch to the idle task for one reason or another at this point, a check is performed to see if the task chosen to run next is not RT and has been woken up. If it is not an RT task and was woken up, it is given a slightly higher `sleep_avg` and its dynamic priority is recalculated. This is a way to give another small bonus to sleeping tasks. Once this check has been performed and a bonus possible awarded, the wakeup flag is cleared.

Now `schedule()` is ready to make an actual task switch. This point in the algorithm is a `goto` target, and whatever task is pointed to by the `next` variable is switched to. Earlier decisions to schedule the idle task had simply set `next` to the idle task and skipped to this point. Here, the previous task has its `TIF_NEED_RESCHED` flag cleared, context switch statistical variables are updated, and the previous task gets its run time deducted from its sleep_avg. Also, an interactive credit is deducted from the previous task if its `sleep_avg` dropped below 0 and its credit is neither too high nor too low. This is because if its `sleep_avg` is less than 0 it must not have been sleeping very much. With this setup complete, the actual context switch is made so long as the previous task and the new task are not the same task. After the context switch, preemption is reenabled since it was disabled during the scheduling algorithm. The final part of the `schedule()` function checks to see if preemption was requested during the time in which preemption was disabled, and reschedules if it was.

5.7 Load Balancing

5.7.1 Why do Load Balancing?

Tasks stay on particular CPUs for the most part. This is for cache hotness and memory bank proximity reasons. However, sometimes a CPU has more tasks on it than other CPUs in a system. For instance, on a dual processor system, it is entirely possible that all tasks could be assigned to one CPU and the other CPU would sit idle. Obviously, this is a less-than-optimal situation. The solution is to move some tasks from one CPU to another CPU in order to balance the system. Load balancing is a very important part of any kernel in charge of more than one CPU.

5.7.2 Scheduler Domains

Each node in a system has a scheduler domain that points to its parent scheduler domain. A node might be a uniprocessor system, an SMP system, or a node within a NUMA system. In the case of a NUMA system, the parent scheduler domain of a node's domain would contain all CPUs in the system.

Each scheduler domain divides its CPUs into groups. On a uniprocessor or SMP system, each physical CPU would be a group. The top level scheduler domain containing all CPUs in a NUMA system would have one group for each node, and the groups would contain all CPUs in the node. Groups are maintained as a circular linked list, and the union of all groups is equivalent to the domain. No CPU can be in multiple groups.

A domain's load is balanced only within that domain. Tasks are moved between groups in a domain only when groups within a domain become unbalanced. The load of a group is the sum of the loads of its CPUs.

5.7.3 CPU Load

Since there is one runqueue per active CPU in a system, it makes sense for that data structure to keep track of each CPU's load. Each runqueue maintains a variable called `cpu_load`, which stores a value representing the CPU's load. When runqueues are initialized, their `cpu_load` is set to zero, and the variable is updated every time `rebalance_tick()` is called. `rebalance_tick()` is called at the end of `scheduler_tick()` and also earlier in `scheduler_tick()` if the current CPU is idle (if the current CPU is idle then load balancing is probably desirable before trying to schedule). In `rebalance_tick()`, the current runqueue's cpu_load variable is set to the average of the current load and the old load. The current load is determined by multiplying the runqueue's current number of active tasks by `SCHED_LOAD_SCALE`. The latter macro is a large number (it's actually 128) and is simply used to increase the resolution of load calculations.

5.7.4 Balancing Logic

Load balancing is invoked via the `rebalance_tick()` function, which is called by `scheduler_tick()`. `rebalance_tick()` first updates the current CPU's load variable, then goes up the CPU's scheduler domain hierarchy attempting to rebalance. It only attempts to balance a scheduler domain for a CPU if the scheduler domain has not been balanced for longer than its balance interval. This is very important - since all CPUs share a top level scheduler domain, it would be undesirable to balance that domain every time a CPU has a timer tick. Imagine how often the top level domain would get balanced on a 512 processor NUMA system if that were the case.

If `rebalance_tick()` determines that a scheduler domain needs to be balanced, it calls `load_balance()` on that domain. `load_balance()` looks for the busiest group in the domain, and if there is no busiest group it exits. If there is a busiest group, it checks to see if the busiest group contains the current CPU - if so, it exits. `load_balance()` pulls tasks to less loaded groups instead of pushing them from overloaded groups. Once the busiest group has been identified, `load_balance()` attempts to move tasks from the busiest group's busiest runqueue to the current CPU's runqueue via `move_tasks()`. The rest of `load_balance()` is largely devoted to updating heuristics according to whether or not load balancing succeeded and cleaning up locks.

`move_tasks()` attempts to move up to a certain number of tasks from the busiest group to the current group. `move_tasks()` attempts to take tasks from the target runqueue's expired priority array first, and within that array it takes the lowest priority tasks first. Tasks are moved by calling `pull_task()`. `pull_task()` moves tasks by dequeuing them from their current runqueue, and enqueuing them on their destination runqueue. The operation is quite short and simple, a testament to the scheduler's clean design.

5.7.5 Migration Threads

Every CPU has a migration thread, which is a kernel thread that runs at a high priority and makes sure that runqueues are balanced. The thread executes the loop in the function `migration_thread()` until it is told to stop for some reason (i.e. the CPU goes down for one reason or another). If task migration has been requested (e.g. via `migrate_task()` for CPU assignment or active balancing reasons), the migration thread will see the request and carry it out.

5.8 Soft RT Scheduling

The Linux 2.6.8.1 scheduler provides soft RT scheduling support. The "soft" adjective comes from the fact that while it does a good job of meeting scheduling deadlines, it does not guarantee that deadlines will be met.

5.8.1 Prioritizing Real-Time Tasks

RT tasks have priorities from 0 to 99 while non-RT task priorities map onto the internal priority range 100-140. Because RT tasks have lower priorities than non-RT tasks, they will always preempt non-RT tasks. As long as RT tasks are runnable, no other tasks can run because RT tasks operate with different scheduling schemes, namely `SCHED_FIFO` and `SCHED_RR`. Non-RT tasks are marked `SCHED_NORMAL`, which is the default scheduling behavior.

5.8.2 `SCHED_FIFO` Scheduling

`SCHED_FIFO` tasks schedule in a first-in-first-out manner. If there is a `SCHED_FIFO` task on a system it will preempt any other tasks and run for as long as it wants to. `SCHED_FIFO` tasks do not have timeslices. Multiple `SCHED_FIFO` tasks are scheduled by priority - higher priority `SCHED_FIFO` tasks will preemt lower priority `SCHED_FIFO` tasks.

5.8.3 `SCHED_RR` Scheduling

`SCHED_RR` tasks are very similar to `SCHED_FIFO` tasks, except that they have timeslices and are always preempted by `SCHED_FIFO` tasks. `SCHED_RR` tasks are scheduled by priority, and within a certain priority they are scheduled in a round-robin fashion. Each `SCHED_RR` task within a certain priority runs for its allotted timeslice, and then returns to the bottom of the list in its priority array queue.

5.9 NUMA Scheduling

5.9.1 Scheduler Domain/Group Organization

The scheduler domain system is a critical component of Linux 2.6.8.1's NUMA support. NUMA architectures differ from uniprocessor and SMP systems in that a NUMA system can contain multiple nodes. It is typical for each node to have a local memory bank and certain other resources that are best used by CPU's that are physically nearby. For example - while a CPU in a NUMA system can usually use memory on any node in the system, it is faster to access memory on local banks than it is to access memory that may be physically 20 feet and several NUMA links away. In short, since NUMA systems can be physically very large with less than optimal connections between nodes, resource proximity becomes an issue. The issue of proximity makes organizing resources into groups important, and that is exactly what the scheduler domain system does.

On a NUMA system, the top level scheduler domain contains all CPUs in the system. The top level domain has one group for each node; that group's CPU mask contains all CPUs on the node. The top level domain has one child scheduler domain for each node, and each child has one group per physical CPU (the group could have multiple virtual CPUs in the case of SMT processors). This scheduler domain structure is set up with special domain initialization functions in the scheduler which are only compiled if `CONFIG_NUMA` is true.

5.9.2 NUMA Task Migration

When `scheduler_tick()` runs, it checks to see if groups in the base domain for the current CPU are balanced. If not, it balances groups within that domain. Once that domain is balanced, its parent domain is balanced (and then its parent and so on). This means that on a NUMA system per-node base scheduler domains allow for keeping tasks within a node, which is the desired behavior for resource proximity reasons. Since the scheduler balances between a scheduler domain's groups and not necessarily individual CPUs, when the top level domain is balanced tasks are moved between nodes only if any node is overburdened. Individual CPUs in a NUMA system are not considered during top level scheduler domain balancing (unless of course each node has only one CPU). Once a task becomes part of a new node, it stays within that node until its new node is an overburdened one. These levels of balancing discourage the unnecessary movement of tasks between nodes.

5.10 Scheduler Tuning

5.10.1 Reasons for Tuning

Linux users with some basic development skills might want to optimize the CPU scheduler for a particular type of use. Such people might include desktop users that want to sacrifice efficiency for response time, or sysadmins who want to sacrifice a server's response time for the sake of efficiency.

5.10.2 Scheduler Tuning Possibilities

Near the top of the file `kernel/sched.c`, there is a series of macros beginning with `MIN_TIMESLICE`, the definitions of which can be tuned in an attempt to achieve certain goals. These values can be tweaked within reason and the scheduler will function in a stable manner. After changing desired macro definitions, users should simply compile the kernel as they normally would. There is no sane way to change these values in an already-compiled kernel, and they are not modifiable on a running system. Some of the more interesting tuneable values are discussed in sections 5.10.3 - 5.10.6.

It is important to note that there are so many variables in the scheduler code and workloads that the scheduler can handle that almost nothing is guaranteed by any tweaking. The best way to approach tuning the scheduler is by trial and error, using the actual workload the tuned scheduler will work with.

5.10.3 `MIN_TIMESLICE` and `MAX_TIMESLICE`

`MIN_TIMESLICE` is the bare minimum timeslice that a task can receive. `MAX_TIMESLICE` is the maximum timeslice that a task can receive. The average timeslice is determined by averaging `MIN_TIMESLICE` and `MAX_TIMESLICE`, so increasing the value of either extreme will increase timeslice lengths in general. Increasing timeslice lengths will increase overall efficiency because it will lead to fewer context switches, but it will decrease response times. However, since I/O-bound tasks tend to have higher dynamic priorities than CPU-bound tasks, interactive tasks are likely to preempt other tasks no matter how long their timeslices are; this means that ineractivity suffers a bit less from long timeslices. If there are many tasks on a system, for example on a high-end server, higher timeslices will cause lower priority tasks to have to wait much longer to run. If most tasks are at the same dynamic priority, response time will suffer even more since none of the tasks will be preempting others in an attempt to give better response times.

5.10.4 `PRIO_BONUS_RATIO`

This is the middle percentage of the total priority range that tasks can receive as a bonus or a punishment in dynamic priority calculations. By default the value is 25, so tasks can move up 25% or down 25% from the middle value of 0. Since there are 20 priority levels above and below 0, by default tasks can receive bonuses and penalties of 5 priority levels.

Essentially this value controls the degree to which static, user-defined, priorities are effective. When this value is high, setting a task to a high static priority using the `nice()` system call has less of an effect since dynamic priority rewards and punishments will allow for more flexibility in dynamic priority calculatins. When this value is low static priorities are more effective.

5.10.5 `MAX_SLEEP_AVG`

The larger `MAX_SLEEP_AVG` gets, the longer tasks will need to sleep in order to be considered active. Increasing the value is likely to hurt interactivity, but for a non-interactive workload equality among all tasks may be desirable. Overall efficiency may increase since fewer increases in dynamic priority means fewer preemptions and context switches.

5.10.6 `STARVATION_LIMIT`

Interactive tasks are reinserted into the active priority array when they run out of timeslice, but this may starve other tasks. If another task has not run for longer than `STARVATION_LIMIT` specifies, then interactive tasks stop running in order for the starving tasks to get CPU time. Decreasing this value hurts interactivity since interactive tasks will more often be forced to give up the CPU for the sake of starving tasks, but fairness will increase. Increasing this value will increase interactive performance, but at the expense of non-interactive tasks.

6 The Linux 2.4.x Scheduler

6.1 The Algorithm

A basic understanding of the Linux 2.4.x scheduler is instructive in that it points out much of the logic behind the improvements made in the 2.6.x kernels.

The Linux 2.4.x scheduling algorithm divides time into "epochs," which are periods of time during which every task is allowed to use up its timeslice. Timeslices are computed for all tasks when epochs begin, which means that the scheduler's algorithm for timeslice calculation runs in O(n) time since it must iterate over every task.

Every task has a base timeslice, which is determined by its default or user-assigned nice value. The nice value is scaled to a certain number of scheduler ticks, with the nice value 0 resolving to a timeslice of about 200ms. When calculating a task's actual timeslice, this base timeslice is modified based on how I/O-bound a task is. Each task has a `counter` value, which contains the number of scheduler ticks remaining in its allotted timeslice at any given time. At the end of an epoch, a task might not have used up all of its timeslice (i.e. `p->counter > 0`) because it was not runnable (sleeping), presumably waiting on I/O. A task's new timeslice is calculated at the end of an epoch with the following code:

```
p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
```

The remaining scheduler tick count is shifted to the right one position (divided by two), and added to the base timeslice. In this way, tasks that do not use up their timeslices due to being I/O-bound get longer a longer timeslice in the next epoch. If a task suddenly becomes CPU-bound and uses up its whole timeslice, it quickly drops back to a base timeslice in the next epoch. However, this becomes more and more difficult to do as successive epochs of low timeslice utilization build up a task's timeslice (which is a desired effect).

When a task forks a new task, the parent's timeslice is split between itself and its child. This prevents tasks from hogging the CPU by spawning children.

The `schedule()` function selects the task that will run next by iterating over all runnable tasks and calling the `goodness()` function[8]. The task that evokes the highest return value from the goodness() function is run next. Goodness is generally determined by adding the process's `counter` value to its `nice` value, but in the case of RT tasks, 1000 is added to the result (RT tasks: `p->policy != SCHED_NORMAL`) so that they are always selected over non-RT tasks. An interesting optimization in the `goodness()` function is that if a task shares the same address space as the previous task (i.e. `p->mm == prev->mm`), it will get a slight boost to its goodness for the sake of taking advantage of cached pages. The goodness algorithm essentially boils down to the following[5]:

```
if (p->policy != SCHED_NORMAL)
return 1000 + p->rt_priority;
if (p->counter == 0)
return 0;
if (p->mm == prev->mm)
return p->counter + p->priority + 1;
return p->counter + p->priority;
```

6.2 Strengths

The Linux 2.4.x scheduling algorithm performs quite well but is fairly unremarkable, and as such, its strengths lie in the realm of the mundane.

[8]Iterating over tasks to find the one with the best goodness() value is another example of the Linux 2.4.x scheduler using an O(n) algorithm.

6.2.1 It Works

Although it is technically vague, the fact that the Linux 2.4.x scheduler "works" should not be discounted in terms of the credit it deserves. The demands on the Linux scheduler are high; Linux 2.4.x runs on many different types of very important systems, from Fortune 500 servers to NASA supercomputers, and it runs quite well. The Linux 2.4.x scheduler is robust and efficient enough to make Linux a major player in the computing world at the 2.4.x stage, which is more than can be said for many schedulers in the past.

6.2.2 (Relatively) Simple Logic

The Linux 2.4.x file `kernel/sched.c` is about 1/3 the size of `kernel/sched.c` in Linux 2.6.x. The algorithm is fairly straightforward, even if its potential behavior and effects are somewhat unpredictable. Tweaking scheduler behavior for specific situations is fairly easy in Linux 2.4.x, while improving it without a major overhaul is quite difficult.

6.3 Weaknesses

In "Understanding the Linux Kernel," Daniel Bovet and Marco Cesati expound on four weaknesses in the Linux 2.4.x scheduler: scalability, large average timeslices, a less-than-optimal I/O-bound task priority boosting strategy, and weak RT-application support[9][5].

6.3.1 Scalability

The Linux 2.4.x scheduler executes in O(n) time, which means that the scheduling overhead on a system with many tasks can be dismal. During each call to schedule(), every active task must be iterated over at least once in order for the scheduler to do its job. The obvious implication is that there are potentially frequent long periods of time when no "real" work is being done. Interactivity performance perception may suffer greatly from this. This problem has been solved in the Linux 2.6.x series by using algorithms that perform in O(1) time. Specifically, the Linux 2.6.x scheduler recalculates timeslices as each task uses up its timeslice. The Linux 2.4.x scheduler recalculates timeslices for all tasks at once, when all tasks have run out of their timeslices. Also, priority arrays in the Linux 2.6.x scheduler make finding the highest priority process (the one that should run next) as simple as finding the first set bit in a bitmap. The Linux 2.4.x scheduler iterates over processes to find the one with the highest priority.

6.3.2 Large Average Timeslices

The average timeslice assigned by the Linux 2.4.x scheduler is about 210ms [5]. This is quite high (recall that the average timeslice in the Linux 2.6.x scheduler is 100ms), and according to Bovet and Cesati, "appears to be too large for high-end machines having a very high expected system load." This is because such large timeslices can cause the time between executions of low-priority tasks (or simply unlucky ones if all priorities are equal) to grow quite large. For example - with 100 threads using all of their 210ms timeslices without pause, the lowest priority thread in the group might have to wait more than 20 seconds before it executes (an unlikely situation, but it illustrates the point). This problem does not appear to be mitigated by starvation checks or taking system load into account when calculating timeslices, which might not help anyway. Only process data fields are used in timeslice recalculation:

```
p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
```

The problem is lessened by the Linux 2.6.8 scheduler's lower average timeslices, but it is not entirely done away with. Essentially the system load just needs to be twice as much to create the same problem. It is important to remember that even though higher priority tasks can preempt tasks with long timeslices and thus maintain acceptable interactivity, that doesn't help tasks that are non-interactive and at the end of the line, but cannot wait for extremely long periods of time to execute. An example might be a web server that has retrieved data from an I/O source and is waiting to formulate an HTTP reply - a long wait to formulate the reply could cause timeouts on the client side of the connection.

[9]Actually, Bovet and Cesati are talking about the Linux 2.2.x scheduler in their book, but except for some major SMP handling changes, the scheduler did not change much between the 2.2.x and 2.4.x kernel series. The similarity between the Linux 2.2.x and 2.4.x schedulers is noted by them.

6.3.3 I/O-Bound Task Priority Boosting

The Linux 2.4.x scheduler's preference for I/O-bound tasks has some notable flaws. First, non-interactive tasks that are I/O-bound get a boost even though they do not need one. The example from Bovet and Tosati is a database application that must retrieve data from a disk or a network. Also, tasks that are interactive but also CPU-bound may appear to be unresponsive since the boost for interactivity and the penalty for high CPU usage can cancel each other out. Since the Linux 2.4.x scheduler assigns timeslices based on the time remaining from the last epoch plus a value based on the user nice value, the former value will be low for a CPU-bound task and subsequently if it also happens to be interactive, it will get a very small bonus.

Both problems cannot actually be solved until a better metric than sleep time is found for measuring interactivity. Since the basic logic is that sleepy tasks are interactive and non-sleepy tasks are not, pairing up antithetical characteristics is always going to be a problem. As for the former problem, with a non-interactive yet I/O-bound task, the Linux 2.6.x scheduler does categorize tasks that sleep too much as idle, and assigns them an average sleep time of:

```
if (...sleep_time > INTERACTIVE_SLEEP(p)) {
p->sleep_avg = JIFFIES_TO_NS(MAX_SLEEP_AVG - AVG_TIMESLICE);
...
}
```

This avoids giving excessively sleepy tasks huge bonuses. It is not a solution to the problem, but perhaps limits the extent to which it can manifest itself.

6.3.4 RT Application Support

The Linux 2.4.x kernel is not preemptable, and thus the support for RT applications is weak. Interrupts and exceptions result in short periods of kernel mode execution during which runnable RT tasks cannot resume execution immediately. This is unacceptable for RT tasks, which need to meet very strict deadlines reliably. Kernel preemptability adds a great degree of complication to kernel code, particularly concerning locking, and thus has been resisted so far. Linux 2.6.x is a preemptable kernel, and thus RT application support is considerably better. There are, however, certain points at which the Linux 2.6.x kernel cannot be preempted, so RT support is not perfect yet. There are other RT application support issues, such a the prioritization of access to system I/O resources, but they are beyond the scope of this paper.

7 The Future of the Linux Scheduler

7.1 Implementation Tuning vs. Algorithmic Changes

The Linux 2.6.8.1 scheduler is quite solid. It is unlikely that any major changes will be made in the near future because of the fact that further solid performance gains are difficult to measure. The massive number of different workload conditions under which the scheduler is expected to perform well is daunting, and it means that a tweak that helps under one workload is likely to hurt other workloads.

While the basic algorithms and data structures in the Linux 2.6.8.1 scheduler are unlikely to change much, the way things are implemented will continue to be improved (e.g. more efficient coding practices). This will not effect performance at an algorithmic level, but it will improve performance all-around (though to a relatively small extent). Features will be added, but the overall structure of the scheduler will probably not be modified to a very significant degree.

7.1.1 Scheduler Modes and Swappable Schedulers

Two interesting possibilities for future scheduler development are scheduler modes or swappable schedulers (the latter being much more likely).

Scheduler modes means breaking scheduler workloads into categories, and allowing root users to pick the scheduling behavior of a system dynamically. For example, there might be two modes, server and desktop, that a system administrator could put a machine into via a system call from a command. The desktop mode would favor interactivity performance, the server mode efficiency. Actual scheduler mode breakups are unlikely to be this simple, but even this simple setup might be beneficial to many people. In fact, the simplicity might actually be a boon for ease of use and development reasons. Scheduler modes could be implemented fairly easily by making the tuning macros into variables changeable during runtime and maintaining two sets of values with an interface for switching between the sets. While this is an interesting idea, it is unlikely that it will actually happen.

Swappable schedulers would allow users to specify the scheduler that should be used for their own tasks. A basic kernel scheduler would round-robin between users (perhaps favoring root with a longer timeslice?) and allow a user's chosen scheduler to pick tasks for a certain period of time. This way, interactive users could use a scheduler that favors interactive tasks, while non-interactive users could use a scheduler favoring their type of workload. This is a very simplistic description of swappable schedulers, but it gets the main idea across. There are a few different examples of kernels with swappable schedulers in one form or another, perhaps the most conspicuous being the GNU HURD kernel *(http://www.gnu.org/software/hurd/)*.

7.1.2 Shared Runqueues

The recent addition of SMT support to the Linux scheduler is not perfect. In an Ars Technica interview *(http://www.arstechnica.com)*[9], Robert Love put forward the example of a dual Pentium 4 HT workstation which would have four virtual processors. If three virtual processors are idle, and the fourth has two tasks on it, the scheduler should try to move one of the tasks to a different physical CPU instead of the other virtual CPU on the same chip. Right now this does not happen. The solution put forward by Love is shared runqueues for SMT architectures. If runqueues were shared, load balancing would balance among physical CPUs before virtual ones. Such a feature will likely be added to the Linux scheduler.

8 Final Notes

8.1 Acknowledgments

The author would like to acknowledge the assistance, influence, and inspiration he received from the following people and organizations.

8.1.1 Professors Libby Shoop and Richard K. Molnar, Macalester College, St. Paul, Minnesota, USA

Aside from acting as my advisor (Professor Shoop) and reader (Professor Molnar) for this paper, Professors Shoop and Molnar have taught me quite a lot and been wonderful supporters during my time at Macalester College. Professor Shoop has advised me on many independent projects concerning my own particular interests and was always willing to teach and learn about the chosen subjects with me. Having a professor willing to follow me beyond the subjects taught in classes has been wonderful. Professor Molnar taught the class that made the subject of algorithms a favorite of mine. Because of him, I still make very frequent use of "Introduction to Algorithms" for work and fun. Writing the final paper for his algorithms class was one of the best academic experiences I have had at Macalester.

-Josh Aas

8.1.2 Jeff Carr and the Free Software/OSS Community

In late 2000 and early 2001, Jeff Carr allowed me to spend time with his company, LinuxPPC, near Milwaukee WI, USA. Jeff was kind enough to allow me to stay at his house, put up with an incredible amount of my eager high school kid questions, and even took me out to LinuxWorld NY 2001 with the rest of the LinuxPPC crew. During the time I spent with Jeff, he taught me more than just technical things - he taught me about Open Source and Free Software philosophies with a passion I'll never forget. The Free Software/OSS community I've since become a part of has been an incredible extension of the kindness, support, and passion that I originally encountered in Jeff (in particular, Mike Pinkerton and the rest of the Mozilla.org folks!). Many of the things I am most happy to have achieved in my life so far were heavily inspired and supported by members of the Free Software/OSS community and for that I am very grateful.

-Josh Aas

8.1.3 Silicon Graphics, Inc. (SGI)

At the beginning of the summer of 2004, I began work as an intern at SGI, in their Linux System Software group headed by John Hesterberg. As of this writing in late December of 2004 I am still at SGI, going on my 8th month of what was originally a 3-month internship. I work on their Altix line of Linux-based high performance servers, doing various things in kernel and core operating system software (as well as honing my foosball skills). SGI has been an wonderful employer and I have learned an incredible amount since I started working there. My group members and manager have been amazing coworkers; I couldn't have asked for a better environment in which to grow. SGI has contributed to this paper significantly by teaching me about the Linux kernel, allowing me to use their hardware to play with the Linux scheduler on large machines, and supporting my belief in Free Software/OSS philosophies. As a company, they have truly embraced and learned to work with the community software development process.

-Josh Aas

8.2 About the Author

Josh Aas is a student at Macalester College in St. Paul, MN, USA. He will be receiving a double major in Computer Science and English Literature, two subjects which he never has and never will try to associate. At the time of this writing, he is employed by Silicon Graphics, Inc. (SGI) in their Linux System Software group, working on their Altix line of Linux-based high performance servers. When he isn't doing schoolwork or working at SGI, he spends much of his time working on open source software projects, particularly with Mozilla.org. Other interests include traveling, reading, running, sailing, and environmental action.

8.3 Legal (GNU FDL)

References

[1] Curt Schimmel, UNIX® Systems for Modern Architectures - Symmetric Multiprocessing and Caching for Kernel Programmers. Addison-Wesley, 1994.

[2] Andrew S. Tanenbaum, Albert S. Woodhull. Operating Systems Design and Implementation, 2nd Edition. Prentice Hall, 1997.

[3] Andrew S. Tanenbaum. Modern Operating Systems, 2nd Edition. Prentice Hall, 2001.

[4] Robert Love. Linux Kernel Development. Sams, 2004.

[5] Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel, 2nd Edition. O'Reilly, 2003.

[6] Mel Gorman. Understanding the Linux Virtual Memory Manager. Prentice Hall, 2004.

[7] Bitkeeper source management system, Linux project <linux.bkbits.net>

[8] Con Kolivas, Linux Kernel CPU Scheduler Contributor, IRC conversations, no transcript. December 2004.

[9] Jorge Castro, Ars Technica interviews Robert Love, http://arstechnica.com/columns/linux/linux-01-23-2004.ars