Scheduling algorithms for Linux

Anders Peter Fugmann

IMM-THESIS-2002-65

IMM

Trykt af IMM, DTU

Foreword

This report is the result of a masters thesis entitled “Scheduling algorithms for Linux”. The thesis was completed during the period January through October 2002 under the supervision of Associate Professor Jørgen Steensgaard-Madsen at the section of Computer Science and Engineering (CSE), under the department of Informatics and Mathematical Modelling (IMM), at the Technical University of Denmark (DTU).

I would like to thank my supervisor, Jørgen Steensgaard-Madsen, for assistance and guidance. A special thanks to Gitte B. Rasmussen for moral backup and support.

October 11, 2002

Anders P. Fugmann

Abstract

In this report, general scheduling theory is presented, and the Linux scheduler is described in detail. A simulator has been implemented in order to evaluate scheduling algorithms for Linux. The simulator has been calibrated successfully, using some characteristic types of processes, and the behavior of a realistic process mix has been examined. Also, measurements for evaluating scheduler algorithms have been described, and new algorithms for Linux have been evaluated through simulation.

Keywords

Scheduling, process queues, kernel augmentation, Linux, process simulator, calibration.

Contents

1 Preface **1**

1.1 Executive summary 1

1.2 Prerequisites 1

1.3 Typographical conventions 1

1.4 Terminology 2

2 Introduction **3**

2.1 Purpose 3

2.2 Scope 4

2.3 Organization 4

3 Theory **7**

3.1 Terms 7

3.1.1 Fairness 7

3.1.2 Time-sharing. 7

3.1.3 Space-sharing 8

3.1.4 Dynamic priority. 8

3.1.5 Make-span 8

3.1.6 Memory locality 8

3.2 Global queue 8
3.2.1 Load sharing 9
3.2.2 Batch scheduling 9
3.3 Round-robin scheduling 9
3.3.1 Weighted round-robin 10
3.3.2 Virtual round-robin 10
3.3.3 Virtual-time round-robin 10
3.4 Local queues 11
3.4.1 Load-balancing 11
3.4.2 Two-level scheduling 12
3.4.3 Hierarchical queues 13
3.4.4 Gang scheduling 13

4 Linux 15
4.1 Overview 15
4.2 Scheduler 18
4.2.1 Processes 20
4.2.2 Measurements 21
4.3 Instrumentation in Linux 23
4.3.1 Implementation 23
4.4 Tests 25
4.4.1 Kernel compilation 25
4.4.2 Test setup 26
4.4.3 Process execution times and process characteristics . 27
4.4.4 PE utilization 29
4.4.5 Static tests 30
4.5 Hypothesis 33

5 Simulator **37**
5.1 Design . . . 37
5.1.1 Interface . . . 39
5.1.2 Process modeling . . . 42
5.1.3 Simulator flow . . . 43
5.2 Implementation . . . 44
5.2.1 Process description . . . 44
5.3 Calibration . . . 45
5.3.1 Process modeling . . . 45
5.3.2 Static tests . . . 48
5.4 Restrictions . . . 48

6 Modifications **53**
6.1 Round-robin scheduler . . . 54
6.1.1 Purpose . . . 54
6.1.2 Algorithm . . . 54
6.1.3 Implementation . . . 55
6.1.4 Tests . . . 56
6.2 Local-queue scheduler . . . 59
6.2.1 Purpose . . . 60
6.2.2 Algorithm . . . 60
6.2.3 Implementation . . . 62
6.2.4 Tests . . . 62
6.3 Two-level scheduler . . . 66
6.3.1 Purpose . . . 66
6.3.2 Algorithm . . . 67
6.3.3 Implementation . . . 69
6.3.4 Tests . . . 69
6.4 Summary . . . 71

7 Status 75

7.1 Evaluation of the simulator 75

7.2 Evaluation of the Modifications 76

7.3 Further work 76

7.3.1 Simulator 76

7.3.2 Linux modifications 77

8 Conclusion 79

8.1 The project 79

8.2 The process 80

Bibliography 81

Index 82

A Project description 91

B Linux augmentation 93

C Simulator 103

C.1 Task description 103

C.2 Source 105

List of Figures

4.1 Process states 17
4.2 Dependency graph for a compilation job 26
4.3 Compilation of the Linux kernel 28
4.4 PE utilization during compilation 30
4.5 Total compile times as a function of the number of concurrent processes 31
4.6 PE service time. 32
4.7 Deviance from optimal PE service time. 32
4.8 PE efficiency 33
4.9 Deviance from optimal PE efficiency. 34

5.1 Design overview of the simulator 38
5.2 State diagram and transitions for processes in the simulator. The number of possible states have been extended with pending, stopped and assigned. 39
5.3 Simulation of Linux kernel compilation. 47
5.4 Deviance from optimal PE service time. 49
5.5 Deviance from optimal PE efficiency. 49
5.6 Number of PE switches per process. 50

6.1 Round-robin: Deviation from optimal PE service time. . . . 56

6.2 Round-robin: Deviation from optimal PE efficiency. 57

6.3 Round-robin: Number of PE switches per process. 57

6.4 Simulation of Linux kernel compilation using the round-robin scheduler 59

6.5 Local-queue: Deviation from optimal PE service time. . . . 63

6.6 Local-queue: Deviation from optimal PE efficiency. 64

6.7 Local-queue: Number of PE switches per process. 64

6.8 Simulation of Linux kernel compilation using local queues . 65

6.9 Simulation of Linux kernel compilation using the two-level scheduler 70

List of Tables

5.1 Process mix used in tests. 48

6.1 Constants used in the two-level scheduler implementation . 69

Chapter 1

Preface

1.1 Executive summary

This report is the result of a master thesis entitled *Scheduling algorithms for Linux*. Through the study of general scheduling theory and the Linux scheduler in particular, modifications are made to the Linux scheduler. A simulator has been developed, in order to assess the characteristics of different scheduler algorithms. The implemented algorithms were evaluated by comparing results from the simulator with presented theoretical values.

1.2 Prerequisites

In this thesis, it is presumed that the reader has knowledge of the UNIX operating system and process handling. Terms such as virtual memory and input/output subsystem should be familiar to the reader. No prior knowledge of Linux is assumed.

1.3 Typographical conventions

In this report the following typographical conventions are used:

Italics is used for the introduction of new terms.
`Constant width` is used for files, function names, variable names, etc.

1.4 Terminology

Process is used to describe a single schedulable entity in an operating system.

Task is used as a synonym for process.

Job is used to describe all the activities involved in completing any work on a computer from start to finish. A job may involve several processes.

PE is an acronym for Processing Element. A processor on which processes can be executed.

Service. A process is said to be serviced by a PE, when it has exclusive access to a PE.

Chapter 2

Introduction

The focus on schedulers has increased in recent years after the introduction of multitasking environments in which multiple jobs share the same hardware.

The purpose of the scheduler is to decide which process should have access to the processor. In early Unix systems, the primary use was *batch jobs*, which would run until finished, and then the next batch job would be scheduled. This is called *non-preemptive* scheduling. Later, preemptive scheduling was introduced, allowing the system to execute processes seemingly in parallel, by switching between jobs. This has lead to the design of systems with multiple processors, allowing true parallelism between processes.

Today machines are widely used as multiuser systems, in which users run programs in parallel. Also, multi-processor systems are more common, and available to the desktop market.

2.1 Purpose

The goal is to provide a context to investigate scheduler theory. Experiments will be made in order to evaluate schedulers for multiprocessor systems. The main focus will be on multi processing, and non real-time scheduling.

2.2 Scope

Many systems exist today that have multiple processors, and it is therefore relevant to discuss which type of system this theses will focus on. Since the most supported platform by Linux is the Intel 386, a reasonable restraint is to focus on this, more specifically the Intel SMP platform. The Linux scheduler is, however, not strictly platform dependent and algorithms found may be generic enough to be used on other similar platforms.

As many applications already exist on the Linux platform, this thesis will not concentrate on how to change system-calls to the kernel, e.g. requiring their applications to have its own scheduler. This does not exclude the possibility of adding new system calls, though compatibility with existing applications are required. In short, this means that suggested algorithms will primarily focus on the scheduler in the kernel, supporting the existing kernel-API.

2.3 Organization

The above goal will be obtained through the following tasks:

- Theory
  In this chapter, a general overview of terms and algorithms will be presented.
- Linux
  The current scheduler in Linux is described, and modified for instrumentation purposes. Tests are made, in order to hypothesize pros and cons for the existing Linux scheduler.
- Simulator
  For evaluating different scheduler strategies and algorithms, a simulator will be constructed. This chapter describes the purpose, design, implementation and calibration of the simulator.
- Modifications
  Based on hypothesis made while examining the Linux scheduler, modifications to the existing Linux scheduler will be described and results for the simulator will be used in order to evaluate the modifications.
- Status
  A summary of the project, and suggestions for future work will be presented in this chapter.

- Conclusion
  The findings and work made throughout the project will be summarized, and a personal view of the project will be given.

Chapter 3

Theory

This section will provide an overview of some terms and algorithms used in scheduling theory. The algorithms mentioned in this chapter will form a base for later modifications to the Linux scheduler.

3.1 Terms

3.1.1 Fairness

Fairness describes the property of a scheduler, and describes the ability of a scheduler algorithm to share system resources between processes. A Fair share scheduler attempts to give equal service to all processes, a property which is generally perceived to be a critical requirement of a scheduler [KL88, NWZ01].

3.1.2 Time-sharing.

This policy allows multiple processes to share the same processing element, PE. This is done by allowing all processes, in turn, to be assigned to a PE for a small amount of time (a time quantum). Most schedulers, including the UNIX scheduler, use this form of policy [HS91, NWZ01, Fei97, BC01].

3.1.3 Space-sharing

Opposed to the time-sharing policy, space-sharing restricts processes to be scheduled only on a subset of PE's. PE's are partitioned, and each partition is allocated specifically for a job [DA99].

3.1.4 Dynamic priority.

Time-sharing scheduling is often based on a priority. This priority can either be static, or changed dynamically by the operating system to either allow sooner execution or delayed execution of processes in order to achieve some desired scheduling properties [BC01, HS91].

3.1.5 Make-span

Make-span of a schedule is the last process's finish time, i.e. make-span defines the amount of work done over time. This closely relates to how well a scheduler algorithm utilizes the hardware, e.g. keeping the PE busy 100% of the time[1].

3.1.6 Memory locality

On non-uniform memory systems, processes can access local memory more quickly than remote memory. Memory locality denotes this relationship. In shared memory systems, cache usually exists on a PE, and processes will be able to access cached memory quicker than non-cached memory and should therefore ideally always be executed on the same PE [And00]. Studies have shown that bringing data into the local memory results in an extra overhead ranging between 30%-60% of the total execution time. [ML92].

3.2 Global queue

One global queue is often used in shared memory systems. All processes in the ready state (for a description of process states, see 4.1), are placed on

[1]Also called the scheduler efficiency [TW97]

the local queue, and the scheduler selects processes from the global queue, executes them for some time, and then returns the processes to the global queue. This has the advantage of *load sharing* [Fei97, DA99], as load is distributed equally across all PE's in the system. The disadvantage is the lack of memory locality, since processes typically runs on different PE's, resulting in less use of memory locality (cache affinity) [ML92, ML93].

3.2.1 Load sharing

Load sharing describes a system that does not allow a PE to be idle, if there is work waiting to be done. This is easily achieved using global queues.

3.2.2 Batch scheduling

Batch scheduling is one of the earliest forms of scheduling, in which processes are collected in sets, and then scheduled for execution. Usually a scheduled process would be run until completion. Batch scheduling was usually used on, at the time, expensive systems where users would pay for execution time.

3.3 Round-robin scheduling

The round-robin scheduler [TW97] is a very simple preemptive scheduler. Opposed to batch scheduling, where processes are run until finished, the round-robin scheduler allows multiple processes to be run in parallel by executing each processes for a small period of time, a *time quantum*. The round-robin scheduler selects processes from the front of a global queue of runnable processes. When a process blocks or has expired its quantum it is placed at the back of the queue (blocked processes are first placed at the back of the queue after reentering the ready state). The round-robin scheduler has the advantage of very little administrative overhead, as scheduling is done in constant time.

3.3.1 Weighted round-robin

The standard round-robin does not deal with different priorities of processes. All processes are executed equally. In the weighted round-robin[2], quantum is based on the priority of the process. A high prioritized process receives a larger quantum, and by this receives execution time proportional with its priority. This is a very common extension to the round-robin scheduler and will be referred to simply as the round-robin scheduler.

3.3.2 Virtual round-robin

The virtual round-robin [Sta01, HS91] scheduler is an extension to the standard round-robin scheduler. As the round-robin scheduler treats I/O-bound processes and PE-bound processes equally, an I/O-bound process which often gives up its time-slice, will therefore be given an unfair amount of processor time compared to PE-bound processes. For a description of I/O-bound and PE-bond processes, see 4.2.1.

The virtual round-robin scheduler addresses the unfair treatment of I/O-bound processes by allowing processes to maintain their quantum when blocked, and placing the blocked process at the front of the global queue when it returns to the ready state. A process is only returned to the back of the queue when is has used its full quantum. Studies have shown that this algorithm is better than the standard round-robin scheduler in terms of fairness between I/O-bound processes and PE-bound processes.

3.3.3 Virtual-time round-robin

The round-robin and virtual round-robin schedulers both use a variable quantum for processes, as priorities are implemented by changing the quantum given to each process. The virtual-time round-robin [NWZ01] uses a fixed quantum, but changes the frequency by which a process is executed in order to implement priorities. This has the advantage that response times are generally improved for high prioritized processes, while the administrative overhead is still constant time.

[2]The term 'Weighted round-robin' is used in [NWZ01], while [TW97] does not differentiate the two algorithms

3.4 Local queues

Global queues are easy to implement, but cannot be used effectively on distributed memory systems because of the lack of memory locality. Also, a global queue can cause queue congestion on multiprocessor systems as the queue has to be locked whenever the queue is accessed.

Using a local queue for each PE can help resolve the above problems as queue congestion cannot occur since only one PE uses each queue [Fei97]. Binding processes on a local queue can also provide memory locality, and is therefore suitable for distributed memory systems. Shared memory multiprocessor systems can benefit from using local queues, as some memory locality is present as well on theses systems in the form of processor cache. [TW97, Fei97, KF01]

3.4.1 Load-balancing

By assigning processes to a local queue and only scheduling the process on the queue to the PE to which the queue is assigned, load balancing is required in order to avoid idle PE's (empty queues) while there are still processes in the ready state on other queues [Fei97, Mit98]. Load-balancing is done in order to distribute load equally between the local queues. Usually, load is defined as the number of processes on a local queue, and by moving processes between queues load can be distributed equally.

The load balancing algorithm must be stable, and not overreact to small changes in load, as this would result in processes being bounced between and thus degrade performance. This stability can be achieved by balancing only when the imbalance between queues is above a certain threshold.

Work-sharing

Load balancing through work-sharing means that processes are pushed from a local queue to queues with less load [Mit98, BL94]. Whenever a PE has a too high load, processes are distributed from the local queue to other queues. This however has the disadvantage that processors must gather load information and select what queue to migrate processes to, while the PE is heavily loaded. One scheme for selecting target for migration is the *gradient* [Fei97] model. In this model the system maintains a vector pointing to the least loaded PE, and processes are migrated to the PE pointed to by this vector.

Work-stealing

In contrast to work-sharing, work-stealing steals processes from other queues, when the load on the local queue is low [Mit98, BP, BL94]. This has the advantage that balancing is done by the least loaded PE's, and the penalty of load balancing has less effect on system throughput. One method to provide load-balancing using work stealing, is to have an *idle* process on each local queue[3]. Whenever this process is executed, it tries to steal work from other queues. This however has the disadvantage that work-stealing only occurs whenever a PE is idle, and big differences can exist between the local queues, as long a there is at least one runnable process on each local queue.

3.4.2 Two-level scheduling

Both global queue and local queue algorithms have some advantages over each other. To summarize, a global queue algorithm is only useful for shared memory systems, and provides automatic load sharing between PE's. Local queues provide memory affinity, but requires load balancing.

Instead of choosing between the two types of scheduling algorithms, both can be used in combination in order to obtain the advantages of both. This is called two-level scheduling. In two-level scheduling, a global queue is often used to hold all processes. Processes are then scheduled to the second-level scheduler for execution. The second level scheduling can either be controlled by the OS or by the application itself. This is called *self-scheduling*, as either the jobs themselves or the PE's schedule the processes themselves.

One scheme is *chunking* [Fei97], where a set of processes are scheduled in one go from the first-level scheduler. When processes expire their quantum, they are returned to the first level scheduler. This has the advantage that queue congestion is avoided, as only a set of processes are scheduled, instead of scheduling all processes at once.

[3]An idle process is executed, only when the PE is idle as a PE cannot stop execution. Usually the idle process is implemented as an infinite loop

3.4.3 Hierarchical queues

Using Hierarchical queues is another way of obtaining the advantages of both a global queue and local queues [DA99]. This can reduce congestion of the global queue, while maintaining a degree of load-sharing. New processes are placed in the global queue, and each PE has a local queue. In between, there exists a hierarchy of queues. Whenever a local queue is empty, it tries to get work from its parent queue. If there are more processes than PE's, extra processes are placed in the global queue.

3.4.4 Gang scheduling

The term gang scheduling is used when the scheduler selects a set of processes (a gang) to be scheduled in parallel on a given set of PE's. The processes selected often belong to the same job. The idea is to schedule cooperating threads together. Since the machine will become dedicated to the specific jobs, busy-wait is tolerable. Gang scheduling is only useful when multiple PE's are available.

Gang-scheduling is often used in combination with *partitioning*. Partitioning is a scheme where PE's are split up in groups. Processes are then allowed only to be serviced by PE's within the partition to which the processes are scheduled. Partitioning can be useful if, for example, the cost of moving a process from one PE to another in not equal for all PE's. An example of such a system is hyper-threading processors such as Pentium4, where each processor can execute two processes in parallel, using the same processor cache.

Chapter 4

Linux

This chapter will give an overview of the process management in Linux and describe the Linux 2.4 scheduler in detail, including tests and augmentations made to the Linux kernel in order to retrieve per process data.

4.1 Overview

Linux is a UNIX[1] clone licensed under the GNU public license. It was created by Linus Thorvald in 1992, with inspiration from Minix created by Andrew S. Tanenbaum. Today Linux has it biggest market share as a server operating system, but is to a lesser degree also used on workstations. Linux is written in the programming language C.

The Linux operating system supports multiple architectures. The primary architecture is the Intel i386 platform, including both the uniprocessor (UP) and the symmetric multi-processor (SMP) versions.

The basic idea of an operating system is to provide a base to control machine resources and allow processes to be executed on the hardware, usually also providing an abstraction layer for the hardware available. Linux is a multitasking operating system, which means that it is able to execute multiple process in parallel. While a processor at any instant of time can only

[1]UNIX was developed by AT&T Bell Labs in 1969

execute one process, Linux implements a process scheduler, which switches between all runnable processes very quickly to provide pseudo-parallelism. Having multiple processors in the system allows for true parallelism, but since processors usually is a limited resource, a scheduler is still required to guarantee pseudo-parallelism.

Originally, a process was defined as an executing program, a program counter, registers and variables and some memory space in which the program executes. A running process is protected from other processes, and can regard itself as running exclusively on the system[2].

When a process starts, it becomes *ready* to be served by a PE. When the scheduler selects the process for execution, the process enters the *running* state. In this state, the process can either be preempted by the scheduler or block while waiting for data. When a process is preempted, it reenters the ready state. If the process is *blocked* while waiting for data it is suspended and the PE allocated to the process is released. When data becomes ready, the process reenters the ready state, waiting to be serviced. To summarize, the possible states are:

- Ready.
  The process is ready to run, and is waiting to be serviced. From this state the process can only enter the running state.
- Running.
  The process is being served by a PE. In this state, the process can either be preempted and put in the ready state, or make a blocking call and enter the blocked state.
- Blocked.
  The process is waiting for input from another process or IO subsystem. When data is available, the process enters the ready state.

The states and transitions are shown on figure 4.1 on the facing page.

The state blocked covers all possible actions made by a process, when data is not available. This includes I/O[3] communication, *page-faults*, file system operations and waiting for some event to happen (e.g. time to elapse or processes to terminate).

Linux implements *Virtual Memory.* This allows the system to overcommit memory by writing non-active memory to disc. Non-active means that the

[2]Later, the definition has been relaxed to e.g. allow multiple processes to share the same code and/or memory segments

[3]Input/Output

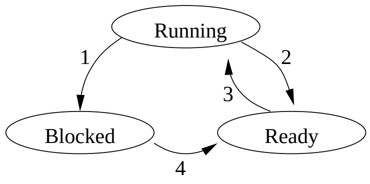

Figure 4.1: Process states

memory is not currently being accessed by a currently serviced process. The memory in Linux is split up in four kibibyte (KiB[4]) pages. In theory, the physical memory space can be regarded as a cache for the virtual memory, with the restriction that any page being accessed by an currently serviced process must be in the cache. If a process accesses a page which is not in the cache, the operating system receives a page-fault and reads the page from disc without notifying the process. If the cache is full, a non-active page in the cache is selected to be removed from the cache (*i.e. it is swapped out*). Since this technique would require all memory blocks in the virtual memory to be allocated on the disc, Linux waits until a block must be removed from the cache before writing it to the disc, in order to minimize disc access. This has the effect that the disc is not always synchronized with the virtual memory. Also this does not limit the virtual memory to be equal to the number of blocks reserved for swapping, but rather the number of allocated blocks possible to be equal to the physical memory plus the disk space allocated for swapping.

The Linux kernel also implement's a write-back caching system for block device access. (hard drives, floppy drives, etc.). This means that when a process reads from or writes to a file, the data is cached, and that a write request takes very little time, compared to the time it physically takes to write to the disc. For read request, the first read request will take normal time, and following identical read request may be shortened, if the data is still in the cache.

Because of the virtual memory and the block cache, it is non-trivial to predict process execution times, and would require a thorough analysis to assess average-case and worst-case cache times.

[4]1 KiB = 2^{10} bytes, as defined by the International Electromechanical Commission (IEC)

4.2 Scheduler

Linux 2.4 uses time-sharing scheduling [NWZ01, Fei97, BC01, HS91]. This is implemented by using a global queue [DA99, Fei97] and dynamic priorities [BC01]. This section will describe in detail how the current scheduler is implemented.

The global queue [Fei97, DA99] in Linux holds all processes which are in the ready state and running state. This queue is called the *run queue.*

To implement fairness [HS91, NWZ01] every task in the system is given a time quantum based on the priority level of the task. The quantum specifies how long the process may be serviced by a PE, within one *scheduling cycle.* A cycle starts when all processes in the system are given new quantum, and ends when all processes on the run queue have been serviced equal to their quantum. The cycle is repeated indefinitely. This ensures that high priority processes may be serviced longer than low prioritized processes, within a scheduling cycle in a linear fashion. This is accomplished by giving bigger quantum to higher prioritized processes, and lower prioritized processes will receive a smaller quantum. Priority levels range from -20 to 19. The quantum, Q_P, given to each process P is given in equation 4.1.

$$Q_P = 10 - \frac{priority}{2} \tag{4.1}$$

where Q_P is the time quantum given to process P, and *priority* is the priority level of the process.

To keep track of serviced time for each process within a scheduling cycle, a counter is set equal to the quantum at the start of each cycle. For each system tick[5], this counter is decreased by one for all processes in the running state. By this, the counter represents the quantum left for each process. To increase PE effectiveness[6] for blocked and waiting processes, some of a process's unused quantum can be transfered to the next cycle. The calculation of the counter C for process P, is given in equation 4.2,

$$C_{P,n} = Q_P + \frac{1}{2}C_{P,n-1} \tag{4.2}$$

[5]A tick is generated by a timer interrupt, which occurs every 1/100 seconds
[6]See section 4.2.2 on page 22

where $C_{P,n}$ is the value of the counter for process P at the start of cycle n, and $C_{P,n-1}$ is the value of the counter at the end of cycle $n-1$. Q_P is the quantum given to the process, as given in equation 4.1 on the facing page. Given these formulas, the counter can have the value $[1;40]$.

The algorithm implements time-sharing scheduling, where each active process receives service proportional with its priority within one cycle. Since the duration of a cycle is finite, no starvation can occur.

Within a cycle, scheduling occurs whenever:

- A process is inserted on the run queue.
- A process enters the blocking state.
- The quantum of a process expires.
- A process yields its service.

If multiple PE's exist in the system, the scheduler runs locally on the particular PE to which a process is to be scheduled.

The selection of a new process is based on a *goodness* value, which is calculated runtime for each process, and the process with the highest value is selected. Therefore, it is said that the scheduling algorithm uses *dynamic priorities*, since the goodness value of a process changes during execution. The value is based on several parameters of the process. Below is a list of these parameters and how these affect the goodness value.

- Remaining quantum.
  The goodness value is initially set to this. If the quantum is zero, no further examination is done.
- PE affinity.
  If the process was previously serviced by the current PE, the goodness value is incremented by 15. It is sought to increase the effectiveness of the PE-cache.
- Avoid page faults.
  If the memory space for the process is the same as the previously serviced process, the goodness value is increased by 1. This is done to somewhat avoid unnecessary page faults.
- Priority.
  Lastly, the value is incremented by $20 - priority$.

The complexity of the algorithm is $O(N)$, where N is the number of ready processes in the system, as a goodness value must be calculated for all running processes for each schedule.

On systems with multiple PE's, a locking mechanism is implemented to guarantee that the run queue remains consistent, since the scheduler runs locally to a PE, and schedule can be invoked on several PE's in parallel. It is also necessary to synchronize with queue insertions and deletions from other parts of the kernel, for example under process creation and deletion.

Because of this locking mechanism, queue congestion can occur where multiple PE's try to access the queue, and thus damaging overall performance in the system, as scheduling overhead is increased.

4.2.1 Processes

In this section different process classes will be defined, and the desired properties will be described.

PE-bound processes.
A process is said to be PE-bound if it requires relatively long periods of computation time (> 1 *sec*) and seldom enters the blocked state. For PE-bound processes, it is relevant to only examine the PE efficiency (i.e. how much PE time the process receives) [HS91, dSS00].

I/O-bound processes.
An I/O bound process uses most of its time waiting for I/O to complete, i.e. in the blocked state. The process will only be serviced in bursts, for quick processing of data and issuing new I/O commands. To keep the hardware busy, the delay between each I/O command should be as low as possible.

Interactive processes.
An interactive process primarily awaits for user interaction. Since user interaction compared to PE-cycles is very long, an interactive process should be optimized for response time. In this case PE service time is less important, since the user cannot issue commands at any rate comparable to processor cycles. An interactive processes is somewhat similar to an I/O-bound process, as the process only needs service in bursts.

Processes properties

In the above, some important properties have been identified with respect to optimizing different processes:

- Efficiency.
- Response time.
- I/O command delay.

The I/O command delay is closely related to the response time. If the response time for a process is low, the process will receive service shortly after entering the ready state. Naturally, a process can only issue I/O commands when in the running state, so the I/O command delay will improve when the response time for a process improves.

A process can seldom be classified into one of the above three classes of processes, but rather a mix of these. This means that it is not relevant only to optimize a process for one property alone. It is therefore desirable to be able to monitor the type of a running process, and optimize accordingly.

4.2.2 Measurements

This section will describe how to evaluate a process with respect to the properties as describe in section 4.2.1 on the preceding page.

Let the *service time* of a process define the time the process is in the running state (*run time*), and let *PE efficiency* for a process define the time a process is in the running state, divided by the time the process is in the running or ready state.

To evaluate make-span, the service time for a process is compared to the service time under perfect fairness. Process response times can be evaluated in the same manner, by comparing the response time with the response time under perfect fairness.

Next, formulas will be given to find the service time under perfect fairness, denoted as *optimal service time* and the PE efficiency under perfect fairness, denoted as *optimal PE efficiency.*

Let the share, Q_p, define the relation between time in the running state, and time in the blocked state for process p. If $Q_p = 1$, the process never enters the blocked state, and if $Q_p = 0$, the process never leaves the blocked state. Let the *proportional share*, S_a, for a process, a, be defined as:

$$S_a = \frac{Q_A}{\sum_{p \in \mathcal{P}} Q_p} \tag{4.3}$$

where $\mathcal{P}$ is the set of all processes in the system.

From this, *perfect fairness* [NWZ01] is defined as the ideal state in which each process is serviced equal to their proportional share.

Optimal PE efficiency

Let the PE efficiency, T, of a process, p, be defined as the proportion of the time the process is not in the blocked state, and the service time of the process, as show in equation 4.4.

$$T(p) = \frac{T_{running}(p)}{T_{running}(p) + T_{ready}(p)} \tag{4.4}$$

Where $T_{running}(p)$ is the time spent by process p in the running state, and $T_{ready}(p)$ is the time spent in the ready state.

Optimal PE efficiency is defined as the PE efficiency under perfect fairness. Since a process cannot be serviced by a PE while in the blocked state, the PE efficiency is not defined for processes in the blocked state.

Let H define the number of PE's in the system. The optimal PE efficiency, T_{opt}, equals the proportion of total available PE's on the system to the PE requirement. The PE requirement is the sum of shares for all processes, $\mathcal{P}$, in the system. Since the process for which the optimal PE efficiency is to be calculated, is in the running or ready state, the share for this process is set to one. The optimal PE efficiency for a process, p, is given in equation 4.5.

$$T_{opt}(p) = \text{Min}\{\frac{H}{1 + \sum_{i \in \mathcal{P}/p} Q_i}, 1\} \tag{4.5}$$

Optimal service time

The service time obtained for a process, p, within a time interval, is the time process P has spent in the running state within the given interval. Let the optimal service time be defined as the service a process receives during an interval, under perfect fairness. Let the optimal service time received by a process, p, during the interval (t_1, t_2) be denoted as $W(p, t_1, t_2)$. The optimal service time can then be found by multiplying the proportional share by the number of PE's in the system. As a process can never be serviced by more than one PE at a time, the optimal service time can be found. The optimal service time is given in equation 4.6.

$$W(p, t_1, t_2) = Min\{(H \cdot (t_2 - t_1) \cdot S_p, (t_2 - t_1)\} \tag{4.6}$$

4.3 Instrumentation in Linux

Currently it is not possible to gather per thread statistics in the Linux kernel. This section will describe how the kernel is augmented to include per process statistics.

The idea of the augmentation is to be able to retrieve data from a running kernel. The data sought should be able to answer two questions: What are the characteristics of the process, and how has the process been handled by the scheduler.

The characteristics should explain how long the task is in the running state, and how long the process has been blocked. The scheduling data should explain how long the process has been in the ready-state, and how many times the process has been scheduled. These data can then be used to calculate PE efficiency, and service time to be compared to optimal values.

4.3.1 Implementation

The implementation is split into two parts: a bookkeeping system, and a data retrieval part.

Bookkeeping

The bookkeeping system updates the per-process data, and makes the data available to the data retrieval. The data retrieval system makes all per-process data available to user-space programs through a device node.

To avoid overhead in the kernel, the bookkeeping is only updated whenever a process changes its state. As explained in section 4.2 on page 18, when a process changes its state to blocking, the active process is removed from the run queue, and vice versa. The timing measurement is based on the number of system ticks, which only allows time measurements to have an accuracy of 1/100 second on the Intel i386 platform. The system also keeps track of the number of preemptions, and the number of times it was assigned to a processor.

The bookkeeping data is kept in the general process structure in Linux, `task_struct`. The specific fields can be seen in appendix B on page 93.

Data collection

The data collection part of the instrumentation must provide a method for user space programs to access the data.

Data can be retrieved in several ways from the kernel: through a file-system node, and through the system log. Writing data to the system log can be regarded as push-technology, by only writing data to the log when available. This can reduce the overhead of data gathering, since data can be restricted only to contain information about process changes. However it is not possible to use the system log, when the run queue lock is held, i.e. from within the scheduler, because the `printk` routine calls `wake_up` to activate `klogd`, the kernel logging daemon. Since wakeup tries to reacquire the run queue lock, deadlock occurs.

Using a node in the file system (device node) has the limitation that user space programs must acquire the data, and thus pull technology must be used. The data obtained through the device node is thus a snapshot of all processes in the system. This does not require the system to remember state changes, and does not cause problems if data is not read. The disadvantage is, that state changes can be lost if data is not read frequently enough.

Data collection is implemented as a module, which can be inserted into the Linux kernel. When inserted, data can be retrieved through a device node. The device node receives a dynamically assigned major number, and has minor number 1. See [Rub98]. The modifications made to the Linux kernel and the source code for the module to be inserted is listed in appendix B on page 93. The kernel source can be obtained from [ea02].

4.4 Tests

In this section, the scheduler in Linux will be tested by repeatedly performing a heavy multiprocess job. An obvious choice is repeated compilation of the the Linux kernel, varying the number of forked processes.

4.4.1 Kernel compilation

The compilation of a kernel is one of the most used tests when trying to determine how well a change in Linux affects overall performance. Usually the total runtime is examined, but the test can also be used as a stress test to show how the system handles multiple processes. The processes examined in this test make state changes numerous times, in order to read the files needed. This confirms to the usual process behavior, which is to be processor bound only for short periods of time.

A single compilation of a C file must read the file to be compiled from the disk, and then start parsing it. While parsing, all included files must be read from disk. When the parsing is done, the compiler starts to translate the written code into machine binary code, which is a processor intensive process. When the binary code has been generated, the process writes the result back to disk. To summarize, a compilation process will start by making several file system requests, after which it becomes PE-bound, and at last makes a single file system request.

To link two or more object files, the process must first read the object files from the disk, and then resolve all the symbols in the files. After this, the result is written to disk.

In general, both compilation and linking have the same pattern. Read some files from disc, process the data, and then write the result to disk. As described in section 4.1 on page 15, Linux implements a block cache. This means that block operations may take little or no time if the block is already in the cache. Since header files are read multiple times during the compilation of the Linux kernel, they are likely be cached, and read times are dramatically reduced, depending on the size of the block cache. Similarly, since linking only links previously created object files, the task of reading the object files is reduced.

When compiling a Linux kernel numerous files must be compiled and linked together. This is done recursively, until the kernel itself is built. An example of the dependencies is shown on figure 4.2, which shows that in order to process the top level linking, three other linker processes must be completed, and so forth.

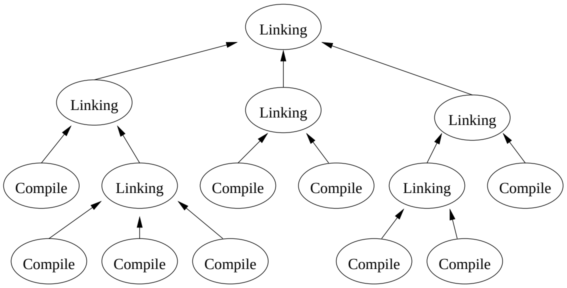

Figure 4.2: Dependency graph for a compilation job

The program `make`, is responsible for compiling and linking the Linux Kernel in the correct order. A `Makefile` is supplied with the Linux kernel source. In this file, all dependencies are defined, and this file is used by `make`. Make has the ability to spawn several child's simultaneously, if some rule's dependencies needs to be made.

Using make, it is very easy to test how Linux handles different load patterns, as the number of spawned processes can easily be controlled. The processes examined in the compilation tests exclude the make-processes themselves, since these do no real work other than waiting for spawned processes to complete before spawning new processes.

4.4.2 Test setup

To test kernel compilation, the augmented Linux kernel is booted in single-user mode, by instructing the kernel to use a shell as the `init`, thereby insuring that Linux does not start any other services, normally started through `init`. To avoid cache influences, all tests are preceded by a `make dep clean`. This scans all files to remake the dependency tree, and removes all object files. Before starting the compilation, data sampling through the augmented kernel is started. The actual tests are started by issuing the command:
`make -j` *N*

Where N defines the number of maximum concurrent processes. $N \in 1, 2, 4, .., 32$.

Data sampling is stated as a real-time process, in order to guarantee that the process is not starved by the compilation processes. Data sampling is done with a frequency of 40 Hz.

All tests are made on a Dual AMD Athlon MP 1500, running at 1.3 GHz. The hard-disc is a Maxtor DiamondMax IDE drive. The system has 256MiB DDR2100 system Ram.

4.4.3 Process execution times and process characteristics

In this test, process characteristics are plotted as a function of process execution time, to see if there is any correlation between I/O-bound processes, PE-bound processes and the execution time.

Doubling the number of processes will have several effects on the process execution time. If resources are available, a better utilization of the system resources will occur and execution time should improve. If the system resources are already exhausted, then adding extra processes to the system will result in further administrative overhead, and the total compile time should increase. Also when doubling the number of processes while system resources are exhausted, the completion time for a single process is doubled, disregarding the extra administrative overhead introduced. When adding more processes, it should also be possible to identify bottlenecks in the system: if raw processing power is the bottle-neck, adding more processes will cause processes to remain longer in the ready-state, and if disc-access is the bottle-neck, processes will become more I/O-bound.

Results

The results are presented in figure 4.3 on the next page. The data has been normalized by dividing the process execution times by C, as given below:

$$C = \begin{cases} 1 & \text{number of concurrent processes} \\ & \text{¡ number of PE's.} \\ \frac{\text{number of concurrent processes}}{\text{number of PE's}} & \text{otherwise.} \end{cases}$$

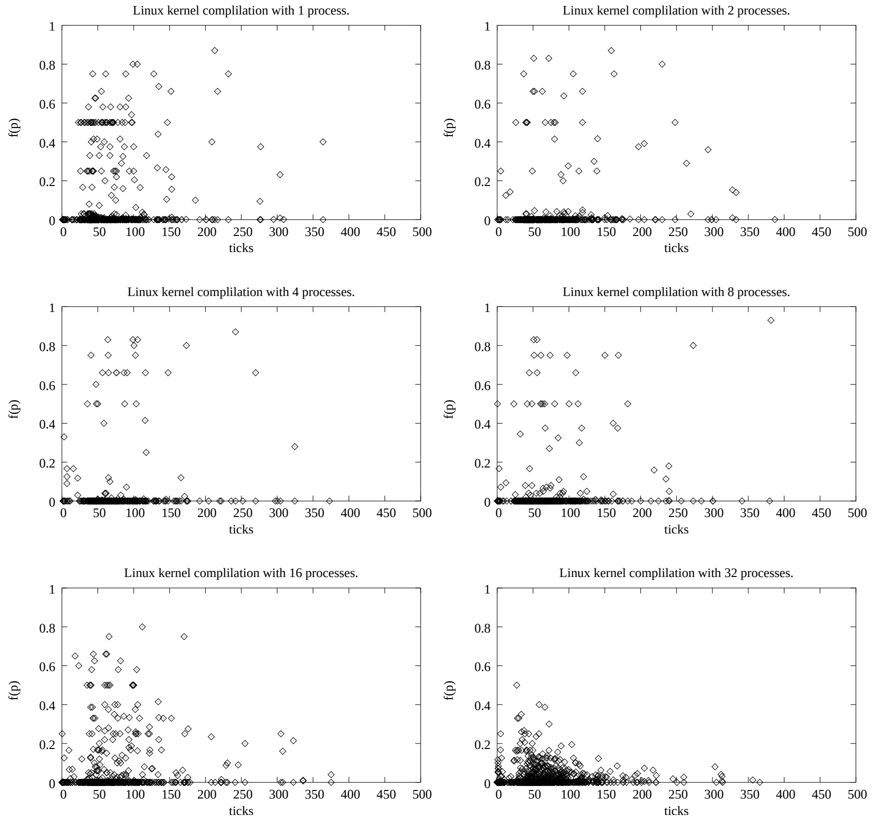

Figure 4.3: Compilation of the Linux kernel

Each point on figure 4.3 represents a process, where the position on the x-axis specifies the total execution time for the process in ticks, and the position on the y-axis specifies the behavior, $f(p)$, of the process found by: $f(p) = T_{running}(p)/(T_{running}(p) + T_{blocked}(p))$, where $T_{running}(p)$ is the time spent by process p in the running state, and $T_{blocked}(p)$ is the time spent in the blocked state.

It can be seen in figure 4.3, that the execution time for each process is proportional with the number of concurrent processes. It is also noticed, that the characteristics of the processes change when the number of concurrent processes is increased. As seen in the result for a compilation with 16 concurrent processes, processes tend to become more I/O-bound. In general, most processes are I/O bound, requiring almost no service on a PE, which indicates many small compilations.

Since the process times are proportional with the number of processes, this suggests that the I/O resources are a limited resource like the PE's. Increasing the number of concurrent processes clearly shows that processes become more I/O bound, which suggests that disc access is becoming the bottleneck.

4.4.4 PE utilization

based on the previous test, it is speculated that when having too many processes in the system, PE utilization will become lower because the compilation processes tends to become I/O bound. Adding extra processes should also increase compile times, as it would imply more administrative overhead due to the use of a global run queue. Also the lack of PE affinity may show as an increase in compile times, if processes are being bounced from one PE to another.

In this test PE utilization is measured as a function of the number of concurrent processes in order to validate the expectations above. In this test no data sampling has been done, as only the total PE usage and total runtime for the compilation process is needed. This is done by using the command:

`time make -j` *N*

where N defines the number of maximum concurrent processes. $N \in 1, 2, 4, .., 32$.

Results

As seen on figure 4.4 on the following page, the PE utilization falls as the number of concurrent processes is increased. This indicates that when processes begins to compete for I/O, the PE's are left idle for longer amounts of time. The compile times shown on figure 4.5 on page 31 also rise as the PE efficiency falls. The compile times do not display any visible added overhead in scheduling the processes. This would have resulted in a linear increase in compile-times. The effect of moving a process from one PE to another is not seen either, suggesting that this does not happen to a significant degree and that the Linux scheduler does honor PE affinity to some extent. It is hypothesized that the reason to why the compiles times increase when then number of concurrent processes is over 25, is also because the system starts swapping as it does not have memory enough to hold both processes memory, and files read in the block cache.

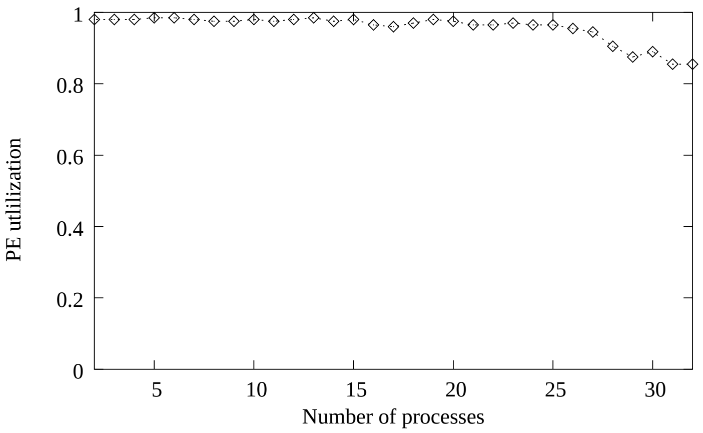

Figure 4.4: PE utilization during compilation

4.4.5 Static tests

As described in section 4.2 on page 18, the Linux scheduler should improve process response times for I/O-bound processes, as processes can receive quantum while in the blocked state, and are therefore likely to have a higher dynamic priority when reinserted on the run queue after having been blocked.

The tests in this section will examine how I/O-bound processes are treated by the Linux scheduler compared to PE-bound processes. To test this, five processes with different characteristics are started, and service time and processor efficiency is measured, using the kernel instrumentation. The process characteristics are the same as those used in the simulator tests, and a description can be found in table 5.1 on page 48.

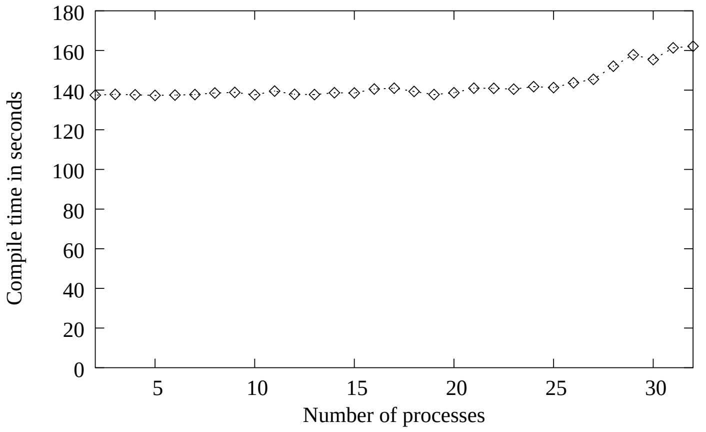

Figure 4.5: Total compile times as a function of the number of concurrent processes

In order to assure that these processes perform as described, the processes sleep instead of issuing I/O commands. By using sleep, the processes still enter the blocked state when necessary, but do not compete for I/O service. Also the time spent in the blocked state is easily controlled when using sleep. The tests are therefore called *static*, as process characteristics cannot change during execution.

To ease creation of the processes, the simulator includes functionality to spawn real processes according to a file containing process descriptions. See 5.2. The process description file is in appendix C.1 on page 103.

Results

In figure 4.6 on the next page, PE service times are graphed as a function of the total execution time. It is clearly seen that the more I/O-bound a process becomes, the less it is serviced, which is in accordance to the process descriptions.

When service times are compared with the optimal service time for each process, is is observed, that I/O-bound processes receive more service relative to their share than PE-bound processes. On figure 4.7 on the following page, the deviation from optimal service time is graphed as a func-

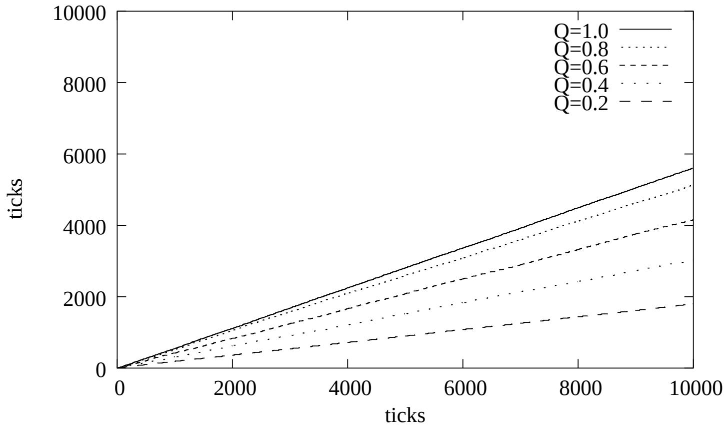

Figure 4.6: PE service time.

tion of execution time. The deviation is found by using the formula: *deviation = measurement − optimal/optimal*, in which case a positive deviation indicates that a process has received more service time than its proportional share, and vice versa.

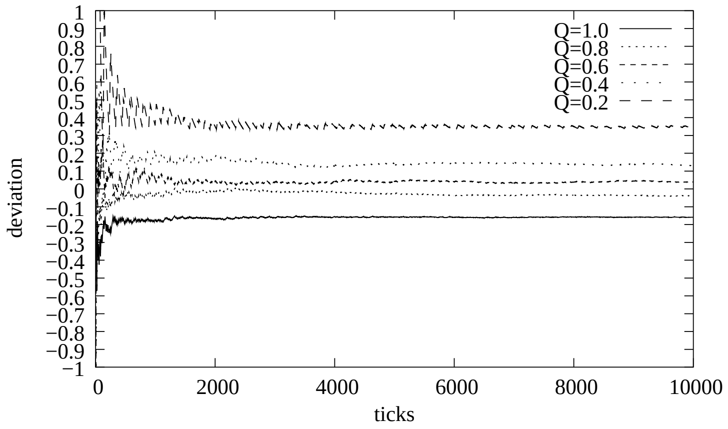

Figure 4.7: Deviance from optimal PE service time.

When examining response times, the result indicate the same trend as found above. This is shown on figure 4.8. The Linux scheduler does a good job in increasing the response times for I/O-bound processes. The more I/O bound the process is, the better response time, since I/O processes have a better PE efficiency than PE-bound processes.

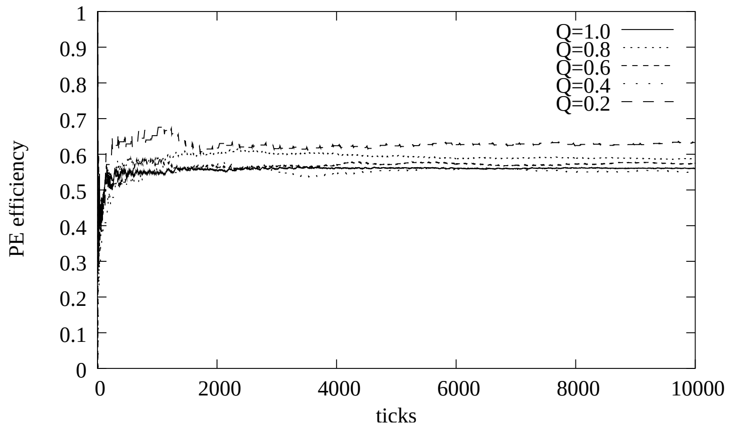

Figure 4.8: PE efficiency

On figure 4.9 on the following page, the measured PE efficiency for each process is compared with the optimal PE efficiency, by plotting the measured PE efficiency deviation from the optimal PE efficiency. It is seen that the PE efficiency for the most I/O-bound process is 19% over the optimal PE efficiency. While the PE-bound process is penalized, it is interesting to see that the PE-bound process is only penalized by 14%, and thus confirms the theory, that increasing PE efficiency for I/O-bound processes does penalize PE-bound processes to the same degree.

4.5 Hypothesis

Based on the analysis and tests of the scheduler implementation in Linux, this section will hypothesize on the strengths and weaknesses in the Linux scheduler.

The tests have shown that I/O-bound processes have very good response times, as the PE efficiency is considerably higher than the optimal PE

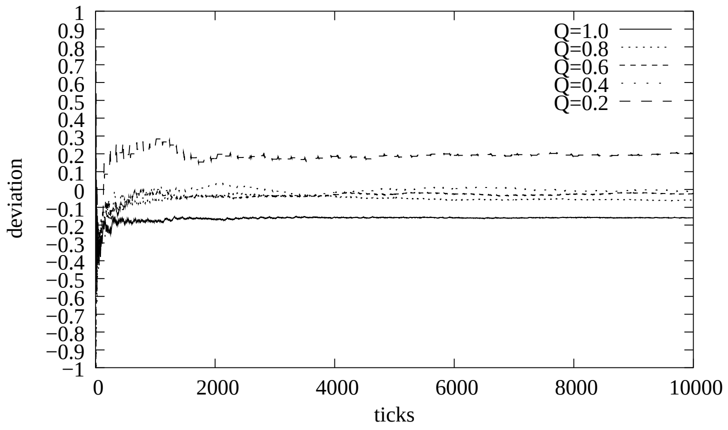

Figure 4.9: Deviance from optimal PE efficiency.

efficiency, while PE-bound processes are not overly penalized by this. This property is due to the use of dynamic priorities, which unfortunately has several disadvantages.

For each schedule all processes are examined, and scheduling overhead is proportional with the number of processes in the ready state. Traversing the list of all processes in the ready state for each schedule also has the disadvantage that cache may be invalidated, as rather large structures has to be read from memory.

The scheduler algorithm has proved to be predictable, and provides fair treatment to all processes, which by [KL88] is an important property of a UNIX scheduler.

As the scheduler uses a global queue, it is hypothesized that queue congestion can occur on systems with multiple PE's. It is, however, difficult to hypothesize on the performance impact this will have. Studies have shown that Linux does not scale well in systems with more than four processors. The advantage of using a global queue is load-sharing, while cache-affinity is sacrificed.

To summarize, the strengths of the Linux scheduler are:

- Good response times.
- Fair toward I/O-bound processes.

- Scheduling is predicable.
- Load-sharing.

It is however hypothesized that the implementation has the following weaknesses:

- Scheduling overhead is proportional with the number of processes.
- Queue congestion can occur.
- Cache affinity is not honored.
- Cache is somewhat invalidated for each schedule.

Chapter 5

Simulator

To evaluate different scheduling strategies, a simulator has been implemented. In this chapter the simulator is described and it is argued that results obtained from simulating a scheduling algorithm will be realistic compared to implementing and running processes in a live environment. The chapter will conclude that the properties of a scheduler implementation can be assessed and used to evaluate the algorithm.

5.1 Design

The goal of the simulator is to make it possible to evaluate scheduler algorithms and the results of changing parameters in these. This section will describe how the scheduler is designed in order to obtain this property.

The focus of the simulator is to test schedulers to be implemented in Linux. The result of a simulation should be able to indicate trends with respect to process scheduling, which are also present in the Linux kernel.

To ease the transition between a Linux implementation of a scheduler and the simulator, the simulator has been designed to allow a minimum of changes to the scheduler implementation. This design decision makes it necessary to closely model the Linux process model and environment within the simulator. Also an interface is defined between the simulator and scheduler implementation. This interface is sought to match the interface in the

Linux kernel to the scheduler. As no strict interface to the scheduler in the Linux kernel exists, it has been necessary to define one.

An overview of the simulator if given on figure 5.1. The simulator consists of the following elements:

- Global scheduler.
  This part handles the task of feeding new processes to the process scheduler. The global scheduler accepts process specifications from multiple sources, and maintains rules about how many and which processes are fed to the process scheduler.
- Process scheduler.
  The process scheduler which is to be examined.
- I/O scheduler.
  The I/O scheduler represents the I/O subsystem in the Linux kernel. It handles I/O resource allocations in order to simulate I/O load generated by multiple processes. The I/O scheduler does not make any distinction on I/O requests.
- Process specification.
  This part keeps track of each process' internal state, in order to allow different process mixes in the system.

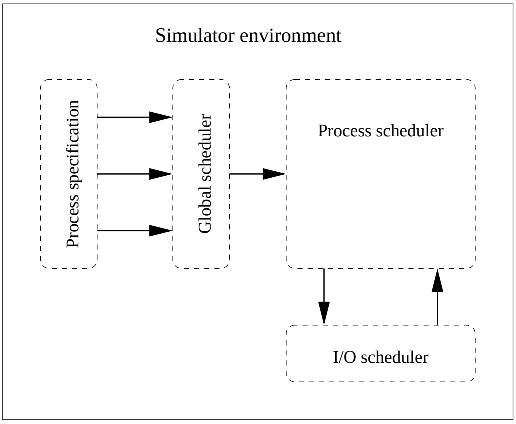

Figure 5.1: Design overview of the simulator

The design of the simulator leads to an extension of the process state diagram given in figure 4.1 on page 17, in order to keep track of which subsystem the process is in. The new state diagram is shown in figure 5.2. The new states introduced in the simulator are:

- Pending.
  A process in this state has been created, but has not yet been scheduled by the global scheduler. When the process is scheduled, the state enters the Running state, by transition 7.
- Stopped.
  When a process terminates, it enters this state. A process can only be stopped when not consuming any resources, and can therefore only enter the stopped state through transition 8 from the ready state, as the simulator does not allow processes to be killed.
- Assigned.
  Because of the introduction of an I/O scheduler, a new state is needed to describe when the process is actually being assigned to an I/O resource. A process can only enter this state while blocked through transition 6.

All processes start in the pending state, and are simulated until reaching the stopped state.

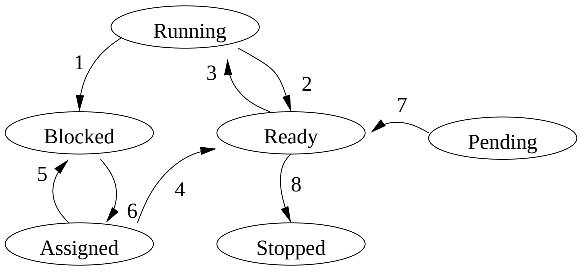

Figure 5.2: State diagram and transitions for processes in the simulator. The number of possible states have been extended with pending, stopped and assigned.

5.1.1 Interface

As it has been a design goal to minimize the number of changes necessary for moving a process scheduler between the simulator and the Linux kernel,

```
#ifndef __SCHED_INTERFACE_H__
#define __SCHED_INTERFACE_H__

#include "simulator_interface.h"

/* Get the id of the "current" processor. */
int smp_processor_id();

/* Initialize structures and specify idle tasks */
void sched_init(Task *idle_task);

void do_timer();

void schedule();

int wake_up_process(Task *p);

int task_has_cpu(Task *task);

Task* get_current();

#endif /* __SCHED_INTERFACE_H__ */
```

Listing 5.1: Interface required by the simulator

an interface is provided in the simulator, which must be implemented by the scheduler.

The interface is split up into two parts: one part of which is the functions and data structures provided by the simulator for the scheduler, and the second is the functions and data structures provided by the scheduler. Since the scheduler in Linux has never been developed with focus on changeability, Linux does not provide a clean interface between the environment and the scheduler, and it has been necessary to define a suitable one. In the list below, all items in the interface are described and deviations from the Linux function signatures, if any, are explained.

Scheduler interface

In listing 5.1 the interface required by the simulator is given. Below is description of all functions and data structures in the interface.

`list <*task_list> tasks`
: This list provides access for the scheduler to all tasks in the simulator. The list elements are of type `Task*`, which correspond to

```
#ifndef __SIMULATOR_INTERFACE_H__
#define __SIMULATOR_INTERFACE_H__

#include "task.h"
#include "globals.h"
#include <list>

void smp_send_reschedule(int cpu);

/* Access to all tasks in the system */
extern list<Task*> tasks;

extern unsigned int jiffies;

#endif /* __SIMULATOR_INTERFACE_H__ */
```

Listing 5.2: Interface provided by the simulator

the `task_struct` structure in Linux. The `task_struct` in Linux holds all process information, and structures used by the scheduling algorithm is contained in this.

`void smp_send_reschedule(int cpu)`
: This function allows the scheduler to invoke scheduling on another PE. The argument `cpu` denotes the PE on which schedule should be invoked.

`ungined long jiffies`
: This variable hold the number of ticks the simulator has been running.

Simulator interface

The interface provided by the simulator to the scheduling implementation is given in listing 5.2. Following is a complete description of the interface.

`int smp_processor_id()`
: The function returns the id of the current PE, on which code is currently being executed.

`void sched_init(Task *idle_task)`
: This function is called for each PE in the system before scheduling starts, in order to allow the scheduler to initialize data structures. In Linux, this function is only called once. The reason for this devia-

tion in the simulator, is to provide a mechanism for the simulator to provide idle tasks in an explicit manner to be used by the scheduler whenever a PE is idle.

`void do_timer()`
: This function is called for each timer tick. The function does not accept any arguments, which is a deviation from Linux, as the function in Linux requires a structure containing all processor registers values as argument, to be used for debugging (i.e. stack-traces.).

`void schedule()`
: This is the main scheduler function. This is called whenever a process leaves the running state, and when a process has the `need_resched` flag set. The schedule function is responsible for assigning processes to PE's.

`int wake_up_process(Task *p)`
: This function is called whenever a process enters the ready state. The function argument is the process to be inserted on the run queue.

`int task_has_cpu(Task *task)`
: The scheduler must provide a method to find out if a process is in the running state. The function should return a non-zero value if the process argument is currently serviced by a PE.

`Task* get_current()`
: This function must return a pointer to the structure representing the process currently being serviced by the PE, pointed to by `smp_processor_id`.

5.1.2 Process modeling

The simulator must be able to model processes with different properties, and in order to make this flexible and extensible, any process in the simulator is associated with a set of properties.

A property is a state machine which can change the state of the process and/or create a new processes to be scheduled. Each property is notified for each system tick, and if the state of the process changes.

The following properties are available:

- Processor bound task.
  Specifies that a process is PE-bound, and should never enter the blocked state.
- I/O bound process.
  This property specifies that the process will enter the blocked state, after having received a specific amount of service time. The process then leaves the blocked state after having received a specified amount of I/O service.
- Periodic process.
  A periodic task emulates a real-time process, that requires to be serviced for a specified amount of time on a PE within a specified period. When the process have obtained its required service time, the process blocks until the period ends.
- Process forking.
  This property describes a process that can fork a number of children. When a forked process enters the stopped state, this property is not notified, but the property itself must keeps track of all processes forked. For each tick the property examines how may forked processes are still active[1] and forks new processes if necessary, keeping the number of forked processes constant.
- Limited runtime.
  All properties above describe an endless cycle. This property allows a process to terminate when it has received a specified amount of service.

By combining several properties to a process, it is possible to create complex processes without having to introduce new properties.

Each process property does its own internal bookkeeping, partly to be used to determine process behavior, and partly to enable monitoring of different properties, such as deadlines, PE-efficiency, etc.

5.1.3 Simulator flow

When the simulation is started, *ticks* are sent to every subsystem[2] and all active processes in order to simulate that time passes. When a process

[1]processes not in the pending state
[2]I/O-scheduler, global scheduler, etc.

receives a tick, it is allowed to change state. If a process serviced by a PE changes its state, all subsystems are notified. If the process changes state from running to blocked or vice versa, both the process scheduler and the I/O scheduler reschedule a new process to be serviced. If a process enters the stopped state, the global process scheduler selects a process in the pending state to move to the running state. If any subsystem causes a process to change state, the process is notified in order to update statistics.

After each tick has been sent, the state of all processes is printed to standard output, similar to the kernel argumentation as described in section 4.3.1 on page 24. It should be noted, that the simulator never discards a process, even if it enters the stopped state. This means that for every tick, data is printed for all processes ever created.

If no processes can be scheduled on a PE, a special idle process is selected for execution.

5.2 Implementation

The simulator has been written in C++ in order to allow inheritance and virtual functions for the process properties. The process structure has been implemented as a C++ class, requiring the scheduler code to be compiled in C++ mode in order to access the process list and process descriptions. Furthermore, the simulator uses C++ Standard Template Library, STL, for list operations. This means that the list API in Linux has not been implemented, and must be changed using STL. The source code for the simulator is listed in appendix C.2 on page 105.

5.2.1 Process description

When the simulator is started, a file containing the global system descriptions and process descriptions is read. This file is a text file, which allows lines to start with an arbitrary number of spaces and tabulations, and any line starting with a hash mark is ignored. A sample of this file is given in appendix appendix C.1 on page 103

The global system settings are: system ticks per second, number of PE's and number of I/O resources. Also the maximal number of concurrent processes and the length of the simulation in ticks are specified.

5.3 Calibration

To calibrate the simulator, it is examined how well simulation results concur to real tests done in Linux 2.4. Two types of tests are made. First the simulator is calibrated to match kernel compilation tests. The second test maps the processes defined in the simulator to corresponding Linux processes, and compares the results.

5.3.1 Process modeling

In section 4.4.3 on page 27, it was seen that when increasing the number of concurrent processes during a kernel compilation, the system became I/O-bound, and each process would spend more time in the blocked state. In order to examine if this phenomenon can be reproduced by the simulator, a suitable process mix must be defined, calibrated and then simulated.

In the compilation tests, two classes of processes are identified:

- Strongly I/O-bound - $f(p) \to 0$
- Lesser I/O-bound - $f(p) \in [0.1; 0.9]$

Both classes of processes have a total execution time of up to 400 system ticks, while most of the processes are terminated before 200 system ticks has elapsed.

To describe this load, a process with the *forktask* property is used to spawn processes. The processes spawned have the *IOTask* property, in order to specify that spawned processes are I/O-bound.

To describe the two classes of processes, two processes of the above type were initially defined, each with different parameters for I/O and PE time required. However, two process descriptions did not provide a sufficiently nuanced description of the processes, and the process description has therefore been extended to three processes, all with the forktask property. The parameters for process characteristics of the spawned processes and the environment parameters have then been calibrated to match the results found in section 4.4.1 on page 25. The final process descriptions are shown in listing 5.3 on the next page.

While calibrating the process mix, it was found that the results in section 4.4.1 on page 25, were best matched when the number of I/O resources was set to three, by specifying `io_resources=3`.

```
Globals: hz=100, cpus=2, io_resources=3, tasks=J, \
ticks=14000

# I/O-bound
process {
      pid:1, grp:1, nice:0, policy: 0, run: 0
      CPU Task:
      Fork Task: forkprob=(J,0) durration=(10,15) \
      type='IO Task: cpu=(1,3), io=(50,100)'
}

process {
      pid:2, grp:1, nice:0, policy: 0, run: 0
      CPU Task:
      Fork Task: forkprob=(J,0) durration=(40,80) \
      type='IO Task: cpu=(1,2), io=(5,10)'
}

process {
      pid:3, grp:3, nice:0, policy: 0, run: 0
      CPU Task:
      Fork Task: forkprob=(J,0) durration=(120,240) \
      type='IO Task: cpu=(10,10), io=(5,0)'
}
```

Listing 5.3: Simulator control file used to simulate a kernel compilation. `J` defines the number of concurrent processes

The process mix found above is then simulated repeatably while increasing the number of concurrent processes. This is done by changing the `tasks` value in the simulator control file, and for each value run a simulation for 14000 system ticks. This equals a simulation of 140 seconds, as is was found in section 4.4.4 on page 29.

Figure 5.3 on the facing page shows the results of simulation of compiling a kernel after calibration. On the figure, each point represents one process, where the x-axis is the total execution time of the process, and the y-axis is the process characteristics found by:

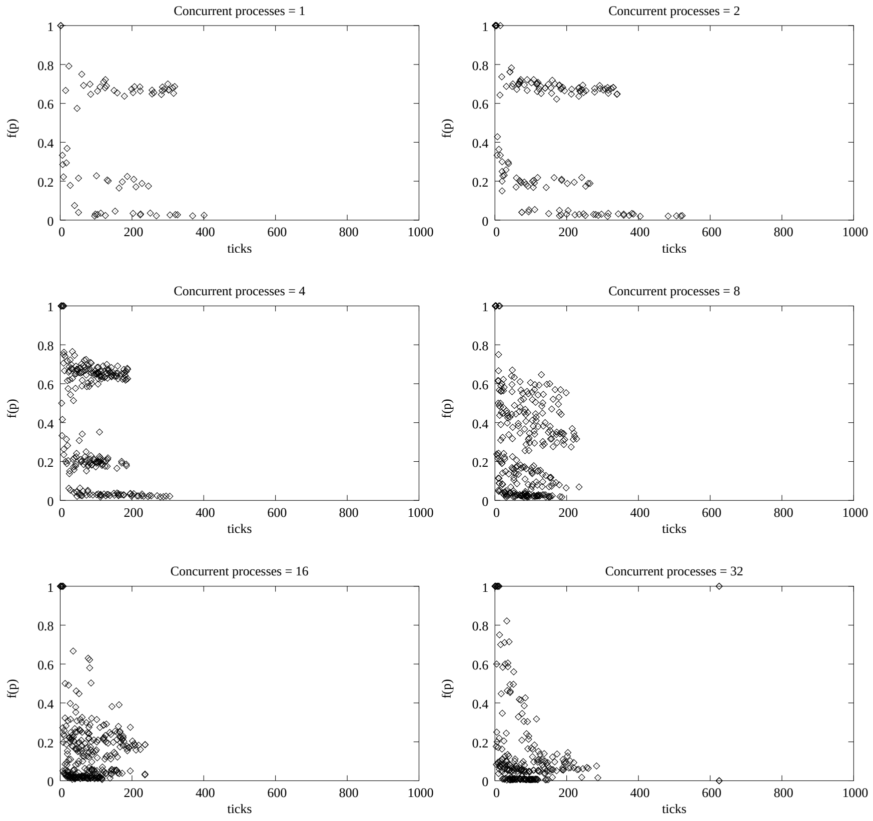

Figure 5.3: Simulation of Linux kernel compilation.

$$f(p) = \frac{T_{running}(p)}{T_{running}(p) + T_{blocked,assigned}(p)}$$

where $T_{running}(p)$ is the time spent by process p in the running state, and $T_{blocked,assigned}(p)$ is the time spent in the blocked and assigned state.

It is observed that the diversity of process characteristics does not match those found in the tests made in Linux. It is also observed that the tendency of processes to become more I/O bound is in fact present.

The lack of diversity in process characteristics leads to the conclusion, that a more complex process description would be preferable. Some studies exist where process behavior is defined using statistical models, e.g. *Sevik's model* and *Dowdy's model* [Wu93]. Implementing these may lead to a closer match, and this could therefore be a suggestion of future work.

5.3.2 Static tests

The static tests are made to examine how well the simulator can predict process scheduling, compared to the result found testing the Linux scheduler. The process mix tested is given in table 5.1.

Process	Type	period	Share (Q)	Prop. share (S)
1	PE-bound	∞	1.0	0.333
2	IO-bound	100	0.8	0.266
3	IO-bound	100	0.6	0.200
4	IO-bound	100	0.4	0.133
5	IO-bound	100	0.2	0.066
			3.0	1.000

Table 5.1: Process mix used in tests.

The number of I/O resources are set to five to avoid processes competing for I/O, as I/O activity in the Linux tests were implemented using a sleep call, which can be regarded is as an infinite I/O resource. The control file for the simulator is given in appendix C.1 on page 103.

On figure 5.4, 5.5 and 5.6, it is observed that results are identical to those found while testing the Linux scheduler, and that the static process-mix needs not to be calibrated further.

5.4 Restrictions

As the simulator only sets up a simplified environment, data gathered from the simulator cannot be expected to excatly match how processes are scheduled in the Linux kernel. Following is a list of restrictions in the simulator:

- Time is only measured in ticks. For each tick statistics is collected based on the state of each process. This means, that the resolution of

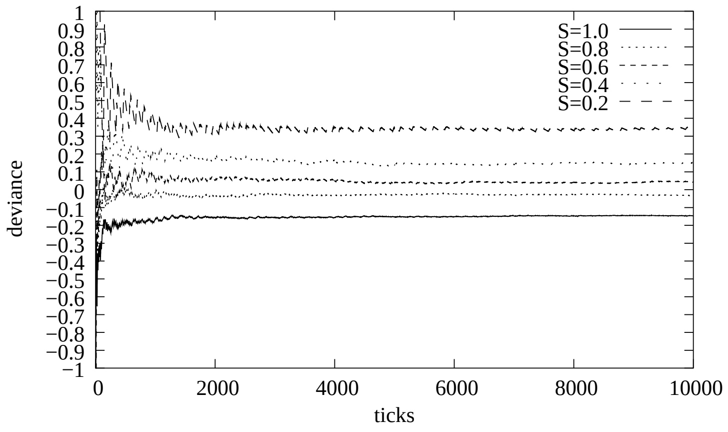

Figure 5.4: Deviance from optimal PE service time.

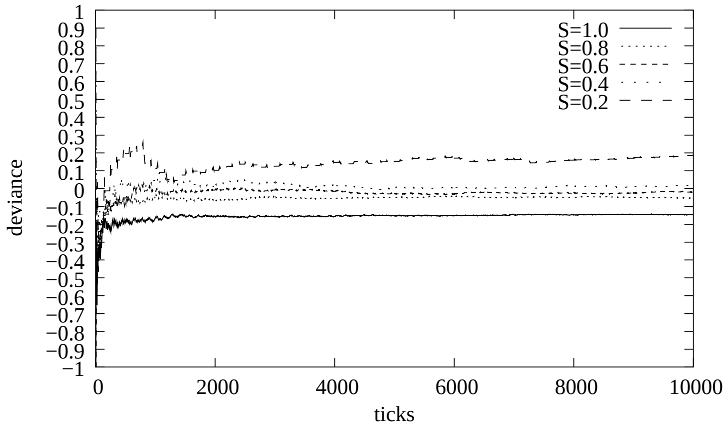

Figure 5.5: Deviance from optimal PE efficiency.

the statistics gathered by the simulator are in ticks, and it is assumed in the simulator that a process will only change state whenever a tick occurs. This is not the case in Linux, as a process can change its state in between two timer ticks, i.e. if an interrupt occurs, or if a process enters the blocked state.

- Scheduler latency is not measured. The simulator assumes that all

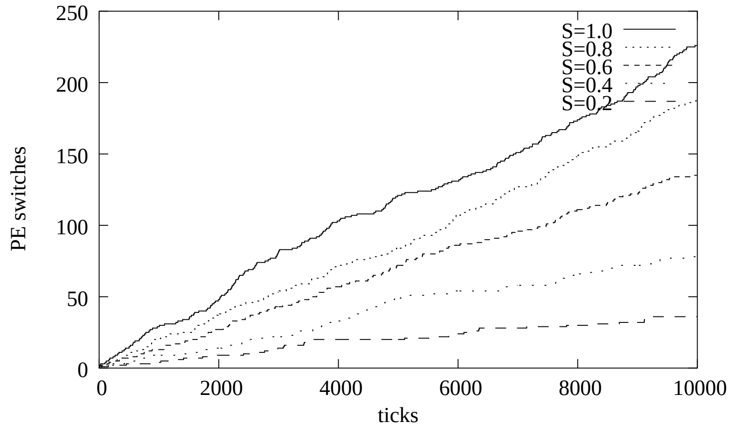

Figure 5.6: Number of PE switches per process.

processing time is assigned to processes in the system, or to idle processes. One way to simulate the overhead of scheduling is to add a process to the environment which has the same processing requirements as the scheduler. The problem here is to find out how much processing time the scheduler algorithm uses and how to model this. A further complication is that the simulator can only represent time in ticks. It is hypothesized, that the time it takes to make a scheduling decision is smaller than one system tick, and adding a process to model scheduling overhead would require a higher resolution of time.

- No penalty for PE change. On the Intel SMP 386 platform cache exists on each PE, and processes can gain processing efficiency running on a *warm* cache compared to running on a PE with a *cold* cache. A warm cache denotes that memory structures used by a process are in the cache. The term cold cache defines that no memory structures for the process are cached. The simulator does not take this cache-process relationship into account, and can therefore not penalize process execution times if the scheduler moves a process from one PE to another.
- No model of memory usage for processes. Processes with high memory requirements may cause page faults, resulting in memory has to be read from disk, and thus would have a negative performance impact. This situation is not modeled in the simulator.

- The simulator is implemented as a single-threaded application. This means that schedule cannot happen in parallel for two PE's, and that locking of data structures is not necessary for scheduler implementations in the simulator. As the simulator does not have locking the effects of queue congestion cannot be simulated. No studies have here been made on the effect of queue congestion, and could be a subject for further investigation.

Chapter 6

Modifications

In this chapter the theory of scheduler will be used to make modifications on the existing scheduler in Linux, partly in order to study the implications of different algorithms, and partly to address some of the imperfections in the current scheduler.

The following algorithms have been implemented:

- Round-robin scheduler.
- Local-queue scheduler.
- Two-level scheduler.

The current scheduler has a complexity of $O(n)$, where n is the number of processes in either ready and running state. Every time `schedule` is called, a goodness value for each process is calculated, and scheduling decisions are based on these values.

6.1 Round-robin scheduler

6.1.1 Purpose

The complexity of the Linux scheduler implies that the administrative overhead will rise with the number of non-blocked processes. This is not a problem when PE's are not fully utilized, but when the number of processes in the running state increases, any administrative overhead will only make the situation worse with respect to make-span. To address this problem, a round-robin scheduler is implemented and simulated.

6.1.2 Algorithm

The round-robin scheduling algorithm reduces administrative overhead. This is because only a constant number of elements is examined when a scheduling decision has to be made, thus giving a complexity of $O(1)$. The scheduling algorithm is shown below. A scheduling decision is made when calling $roundrobin(p, Q)$, where p is process serviced by the PE on which the scheduler is invoked, and Q is the list of all processes in the ready state.

```
$roundrobin(p, q :: Q) =$

    if
        $quantum(p) = 0 : (q, Q + [p])$

        $state(p) \neq \text{running} : (q, Q)$

        $state(p) = \text{running} \wedge quantum(p) > 0 : (p, q :: Q)$
    fi
$roundrobin(p, []) = (p, [])$
```

where the function $quantum(p)$, returns the remainder of the quantum for process p, and the function $state(p)$, returns the state for process p.

Whenever a process reenters the ready state having been in the blocked state, it is inserted to the back of the ready queue.

The round-robin scheduler differs from the Linux scheduler in not using dynamic priorities. This has a number of drawbacks:

- Lack of cache affinity.
  The round-robin scheduler only based scheduling decisions on the state and quantum of a process. Since there is no relation between quantum left and on which PE the process has previously been served, the round-robin algorithm may cause processes to be bounced between PE's.

- Queue congestion.
  The round-robin uses a global queue to hold all processes in the ready state. Whenever scheduling is called on a PE, modifications may occur on this list. While removing an element from the front of a list may prove to be inexpensive, a lock is still required to avoid problems if scheduling is invoked on another PE. Also, if a process spawns a child, or a process leaves the running state (terminates), modifications to the list will occur, and access to the list must be synchronized.

- Unfair treatment of I/O bound processes.
  As mentioned before, I/O bound processes only requires service in bursts. When an I/O bound process reenters the ready state from the blocked state, it must wait until all other processes in the ready state have been serviced before receiving service. Because of these long delays before an I/O bound process receives service after reentering the ready state, the PE efficiency is expected to decrease for I/O bound processes.

6.1.3 Implementation

In the implementation of the round-robin scheduler, it has been a priority to make as few changes to the existing scheduler as possible.

The Linux scheduler keeps all processes in the running and ready state on a global queue (the run queue). This structure is reused in the round robin implementation. This is done by leaving processes in the running state in the front of the run queue. This means that when a new process is to be scheduled, the run queue must be traversed until a process not in the active state is found. The result of this is that the complexity of the algorithm is $O(p)$, where p is the number of PE's in the system, but remains constant with respect to the number of processes in the ready state.

The implementation for the round-robin scheduler can be seen in appendix C.2 on page 156. The implementation does not contain any locking primitives, as the simulator does not offer any functions for this.

6.1.4 Tests

To compare the round-robin scheduler to the Linux scheduler, two tests have been made. The process mix, as described in section 4.4 on page 25, has been simulated, followed by the kernel compilation tests, using the process mix as described in section section 5.3 on page 45.

Static process mix

Using the process mix as shown in table 5.1 on page 48, the round-robin simulator is simulated, and results are compared with those found while testing the Linux scheduler. As the Linux scheduling algorithm has only been tested on a dual PE machine, these tests are only made with `cpus=2`. In these tests, only the deviation from optimal values and the number of PE switches will be examined.

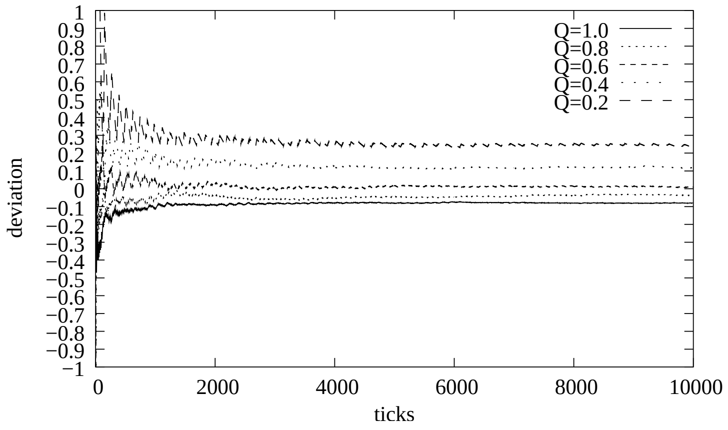

Figure 6.1: Round-robin: Deviation from optimal PE service time.

On figure 6.1, the PE service time deviation from the optimal PE service time is plotted as a function of time. It is seen that the service time deviation is below zero for PE-bound processes, while the deviation is higher

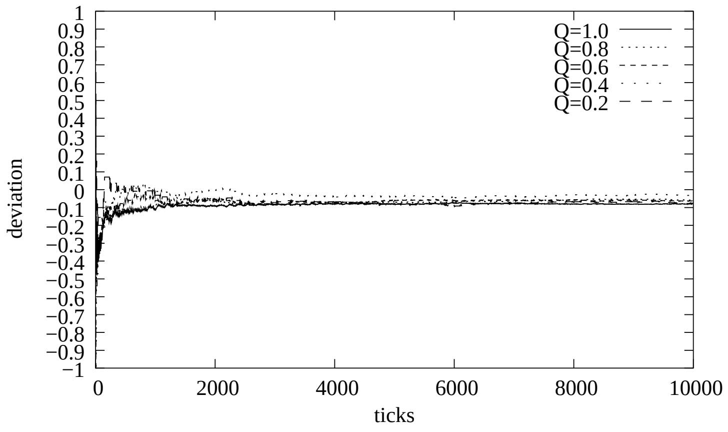

Figure 6.2: Round-robin: Deviation from optimal PE efficiency.

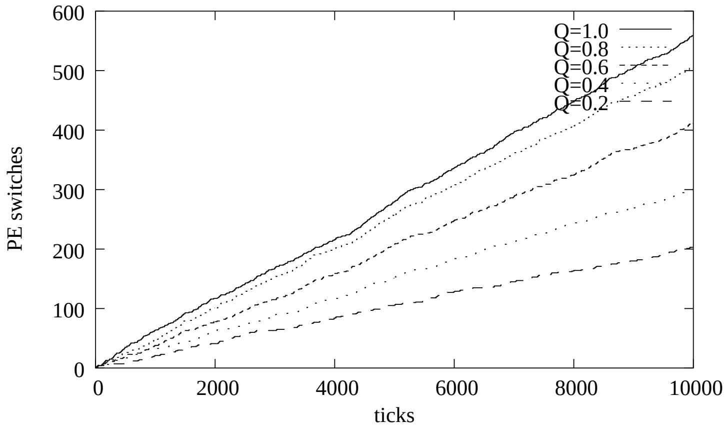

Figure 6.3: Round-robin: Number of PE switches per process.

for I/O-bound processes. Compared to the Linux scheduler, I/O-bound processes receive less service time, and PE-bound processes receives more service time. This is in accordance with the statement made before, about expecting I/O-bound processes to be penalized, although it does not happen to the degree assumed.

Figure 6.2 on the preceding page shows that PE efficiency for all processes is close to the optimal PE efficiency, and that the PE-efficiency is almost equal for all the processes. This indicates predictability, and that the round-robin scheduler gives fair service to all processes. Compared to the Linux scheduler, it is observed that I/O-bound processes have less PE efficiency using the round-robin scheduler, while PE-bound processes benefit from this.

Since the round-robin does not honor cache affinity for processes, the number of times a process is switched from one PE to another should increase. On figure 6.3 on the page before, this pattern is easily seen, as processes are moved between PE's significantly more often than in the Linux scheduler. I/O-bound processes are moved more often than PE-bound processes, as the factor for I/O-bound processes is approximately four compared to the Linux scheduler, while it is approximately 2.5 for the PE-bound process. This is in accordance with the theory, where the number of PE switches should be proportional to the number of process preemptions. This however does not mean that I/O processes suffer more because of lack of PE affinity, because I/O bound processes requires service less frequently than PE-bound processes, and would properly not benefit much from PE-affinity.

Compilation tests

In this test, the process-mix found during calibration of the simulator is used. See listing 5.3 on page 46. For each simulation, the number of concurrent processes is doubled, from one process up to 32 concurrent processes.

On figure 6.4 on the next page, each point represents one process, where the x-axis is the total execution time of the process, and the y-axis is the process characteristics, $f(x) = \frac{T_{running}(p)}{T_{running}(p)+T_{blocked,assigned}(p)}$. It is seen, that the results do not vary significantly from those found using the standard Linux scheduler. When the number of concurrent processes is increased, processes become more I/O-bound. The goal of the round-robin scheduler is to reduce the administrative overhead. However, as the simulator does not take administrative scheduler overhead into account, the advantage of less administrative overhead for the round-robin scheduler will not show improvements in the kernel compilation tests.

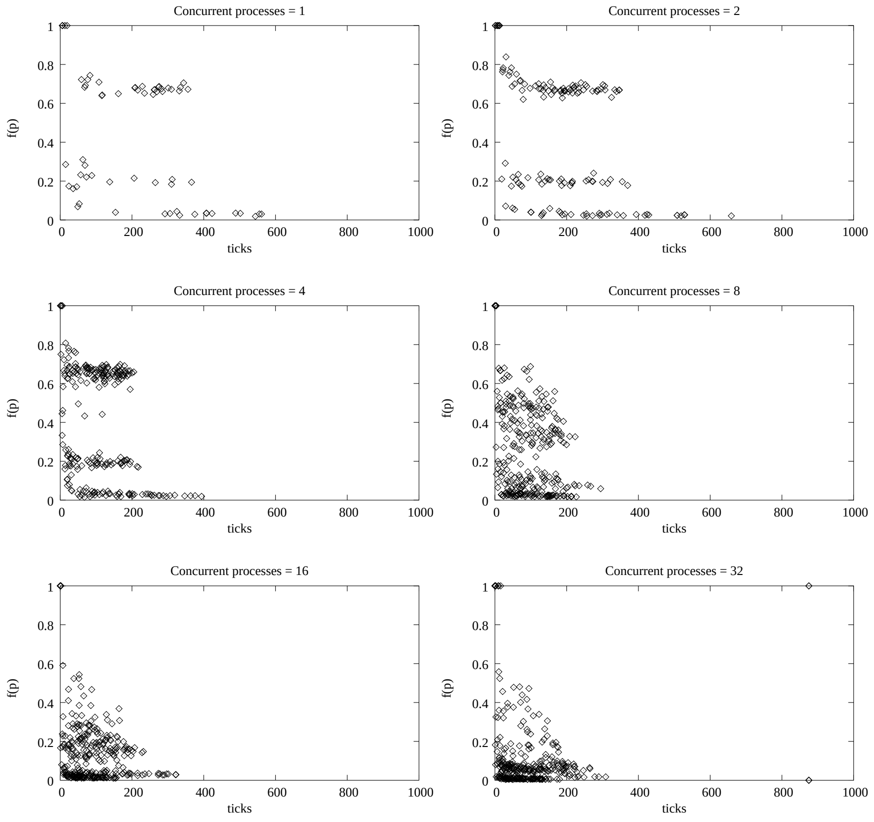

Figure 6.4: Simulation of Linux kernel compilation using the round-robin scheduler

6.2 Local-queue scheduler

In this section a local-queue scheduler will be described and simulated. The results will show, that the algorithm performs very good in terms of honoring cache affinity. It can however be seen, that PE's are not utilized 100%, and it is concluded, that this algorithm best suites systems with many processes, to guarantee that PE's fully utilized.

6.2.1 Purpose

The results from the round robin scheduler tests show that I/O processes are not significantly prioritized over PE-bound processes, as is the case in the Linux scheduler. The round robin algorithm has the advantage of constant execution time, which means that queue congestion is less likely to occur with the round-robin scheduler than with the Linux scheduler.

Queue congestion can also be avoided by creating multiple queues for processes in the ready state, one for each PE - a local-queue scheduler. When a scheduling decision is to be made, the queue assigned for the PE on which scheduling is invoked (local queue) is examined, thus allowing scheduling to occur on multiple PE's in parallel.

Local queues have the advantage of honoring PE affinity for processes. As a process can only be on one local queue, it will always be scheduled on the same PE, unless the process is moved to another local queue. If a process renters the running state from the blocked state, it is placed in the same local queue as it was before it entered the blocked state.

6.2.2 Algorithm

To each PE in the system, a local queue is assigned. The selection of a process to be selected for service is done by examining the local queue for the PE on which schedule is called. The selection of processes from each local queue, is the same as in the Linux scheduler.

Load-balancing

Having multiple queues requires load balancing between the queues in order to assure fairness and to optimize make-span. If no load balancing exists, a situation where one queue holds multiple processes in the ready state, while other queues are empty, can arise.

The purpose of load balancing is to distribute processes across the local queues, trying to keep the number of process in the ready state on each queue equal.

A change in the total number of processes in either the ready or running state implies that the number of processes on a local queue has changed.

Re-balancing the queue is therefore only necessary when a process is inserted on or removed from a local queue. Balancing is then called on the PE for which the number of processes on the local queue has changed.

The algorithm chosen for load-balancing runs local to a PE and tries to keep the load on the queue equal to the general system load. In this context, load is defined as the number of processes in the ready or running state on a queue. From this the system load, is defined as the total number of processes in either running or ready state divided by the number of PE's in the system. This is done by work stealing and work sharing.

Load-balancing must not be done too aggressively, since this can result in processes being bounced between PE's, and thus losing the PE affinity. To avoid processes being bounced between PE's, a measurement for imbalance is defined. Whenever the imbalance is greater than a defined threshold, the upper threshold, processes are balanced in order to bring the imbalance below a specified threshold, the lower threshold.

As the length of a local queue can only change whenever a process enters or leaves the blocked state, it is only necessary to balance the queue in these cases.

The algorithm for load-balancing is:

```
$Balance(P, N)$:
    if
        $abs(load(P) - S) \; > \; T_{upper}$ :
        do
            $abs(load(P) - S) \; > \; T_{lower} \wedge load(P) \; > \; S$ :
                let $(p :: P, Q :: N) = (P, Q)$
                in $(P, p :: Q :: N)$
                end

            $abs(load(P) - S) \; > \; T_{lower} \wedge load(P) \; < \; S$ :
                let $(p :: P, Q :: N) = (P, rev(Q))$
                in $(P, rev(p :: Q :: N))$
                end
        od
    fi
```

Where S is the average system load (number of processes in ready or running state divided by the total number of PE's), P is the list of all processes on the local queue for the PE on which balance is called, and N is the sorted list of local queues, where the queue with the smallest load is the leftmost element. The function $abs(x)$ returns the absolute value of x, and the function $load(P)$ returns the number of processes on the queue P. T_{upper} and T_{lower} are the upper and lower thresholds for the load balancing.

Thresholds

Some constraints exist on the upper and lower thresholds in order to guarantee that the inner loop of the load-balancing algorithm will terminate: $T_{lower} \geq 1 - 1/P$, where P is the total number of PE's in the system. Obviously the upper threshold must be greater than the lower threshold in order for the algorithm to work as expected.

Locking

The local-queue implementation requires a lock on each queue to synchronize list-operations between the *move* and scheduling, and a global lock to avoid load-balancing being activated in parallel on two different PE's.

6.2.3 Implementation

The local-queue algorithm as described above has been implemented in the simulator, see appendix C.2 on page 135. In order to keep track of the number of processes in either running or ready state, the functions `add_to_runqueue`, and `delete_from_runqueue` has been modified to update a global variable upon insertions or deletions in any queue.

6.2.4 Tests

The same tests are performed, as described in section 6.1.4 on page 56. Next the results will be presented, and commented.

Static process mix

Figure 6.5 graphs the deviation between PE service time and the optimal PE service time as a function of time. An improvement over the round-robin scheduler is seen, where I/O-bound processes are serviced more than their share. This suggests that I/O-bound processes are treated more fairly. Compared to the standard Linux scheduler, PE-bound processes receives more service, and I/O-bound processes receives less service when using the local queue implementation. The local queue implementation is still very comparable to the standard Linux scheduler in terms of how I/O-bound processes are prioritized in relation to PE-bound processes. This is expected, as the multi-queue still uses dynamic priorities.

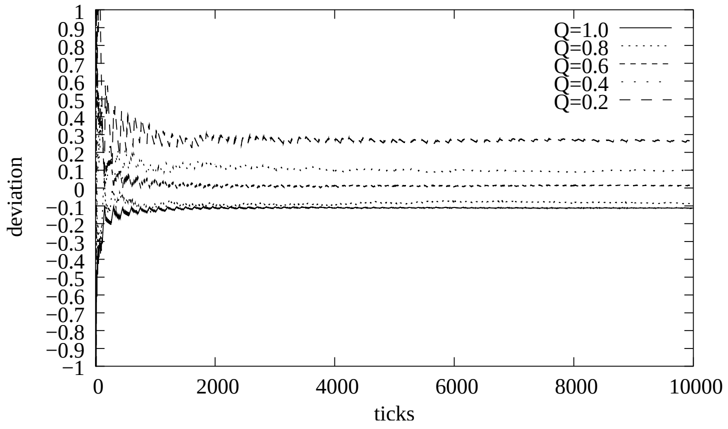

Figure 6.5: Local-queue: Deviation from optimal PE service time.

On figure 6.6 on the next page is seen that all processes have equal PE-efficiency, and that all process have a PE efficiency lower than the optimal PE-efficiency. This is quite different from the behavior of the Linux scheduler, and indicates that the PE efficiency is not 100% (PE's are not utilized as effectivly as in the Linux scheduler). This difference is easily explained, as the Linux scheduler has load-sharing, while the local-queue implementation uses a load balancer, which can result in two processes being on one queue, while the other queue is empty.

Figure 6.7 on the following page shows the number of times a process has been moved from one PE to another. The result indicates a major

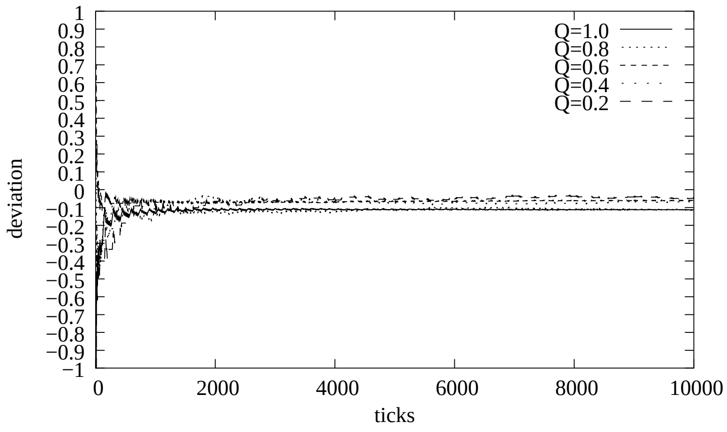

Figure 6.6: Local-queue: Deviation from optimal PE efficiency.

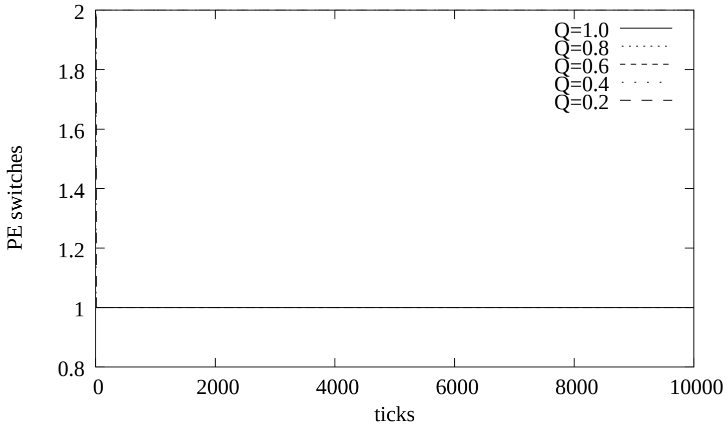

Figure 6.7: Local-queue: Number of PE switches per process.

improvement, as no process is moved more than twice over the period of 10000 ticks. This shows that cache affinity is indeed honored using local queues, as theorized.

Compilation tests

The local-queue scheduler implementation has been simulated using the process mix as defined in listing 5.3 on page 46. As the results for the static tests show that the PE service time is very similar to the values found using the Linux scheduler, no improvements are expected for simulation of kernel compilation. This is confirmed on figure 6.9 on page 70.

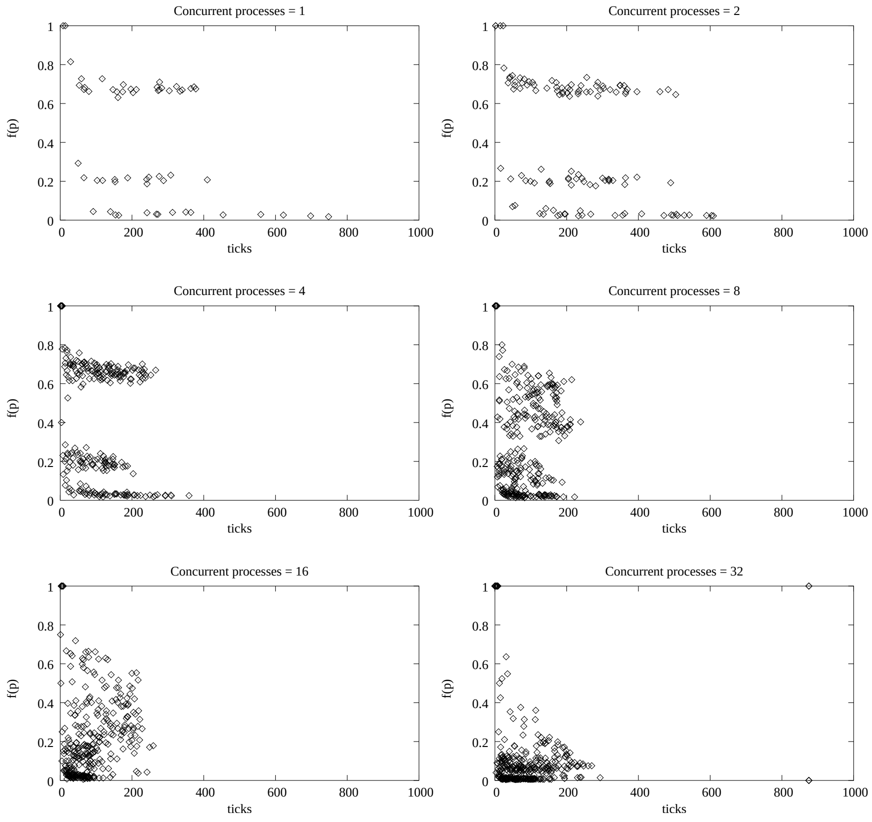

Figure 6.8: Simulation of Linux kernel compilation using local queues

6.3 Two-level scheduler

The compilation tests made for the previous scheduler modifications show that processes tend to become I/O-bound as the number of processes rise. To counter this, a two-level scheduler has been implemented.

6.3.1 Purpose

The purpose of the two level scheduler is to reduce the load on the system by scheduling processes in batches, and thus delaying execution of a set of processes for longer periods.

Scheduling only a part of the total processes on the system at a time has the following benefits:

- Better I/O throughput.
  As the number of processes in the running, ready and blocked state decreases, processes are less likely be waiting for I/O to complete.
- Less swapping.
  Suspended processes are more likely to be swapped out than running processes. This means that running processes are less likely to be preempted due to page-faults, and trashing can be avoided.
- Better cache usage.
  As only a smaller set of processes are in the running state over a period of time, PE-cache and memory cache are less likely to be invalidated.

Because the two-level processes are scheduled in batches, the scheduler has some drawbacks:

- Increased response times.
  As a process can be suspended for long periods of time, response times can increased dramatically, and interactive processes would suffer greatly from this.
- Less utilization of system resources.
  If for example all processes in the current batch are in the blocked state, the PE's would be idle, even if a process suspended by the first-level scheduler is in the ready state.

6.3.2 Algorithm

The first level scheduler selects a set of processes, a batch, to be scheduled for a specified amount of time. Rather than selecting a constant number of processes for each batch, the processes selected are based on the system load to avoid any subsystem (PE or I/O) to be idle. Statistics are kept for each process, and PE and I/O load is calculated based on scheduled processes[1].

The first level scheduler keeps processes in two lists: a ready queue and an expired queue. These queues are used to guarantee fairness. All new processes are placed on the ready queue and processes to be scheduled are selected from this queue. When a process has been scheduled for a defined period of time, C_{rq}, the process is removed from the run queue, in the second level scheduler, and placed on the expired queue. When the ready queue becomes empty, all processes from the expired queue are moved to the ready queue. This is repeated indefinitely. While processes are executed, the system keeps track of time spent in the running state and blocked state for each process. The first-level scheduler also keeps track of overall system load, U_{PE} and U_{IO}, based on process statistics. Whenever a process, p is scheduled, the utilization values are updated as follows: $U_{PE} \mathrel{+}= T_{running}(p)/T_{blocked}(p)$ and $U_{IO} \mathrel{+}= 1-(T_{running}(p)/T_{blocked}(p))$. Processes are only scheduled to the second level scheduler while these values are under specified thresholds (T_{PE} and T_{IO}).

```
$tick(p :: P, Q) =$
    if
        $T_{rq}(p) \geq C_{rq} : tick(P, p :: Q)$
        $T_{rq}(p) < C_{rq} :$
            let $(P', Q') = tick(P, Q)$
            in $(p :: P', Q')$
            end
    fi
$| \; tick([], Q) = ([], Q)$

$bestfit(p :: []) = p$
$| \; bestfit(p :: P) =$
    let
```

[1]The term *scheduled processes*, in this context denotes processes that have been scheduled to the second level scheduler.

```
        $p' = bestfit(P)$
        $u = U_{PE}/U_{IO}$
    in
        if
            $abs(T_{running}(p)/T_{blocked}(p) - u) \ < \ abs(T_{running}(p')/T_{blocked}(p') - u$
            $abs(T_{running}(p)/U_{blocked}(p) - u) \ \geq \ abs(T_{running}(p')/T_{blocked}(p') - u$
        fi
    end

$fill(P, Q, R) =$
    if
        $U_{IO} < T_{IO} \wedge nonempty(P)$ :
            let $p = bestfit(P)$
            in $fill(P - [p], Q, p :: R)$
            end

        $U_{PE} < T_{PE} \wedge nonempty(P)$ :
            let $p = bestfit(P)$
            in $fill(P - [p], Q, p :: R)$
            end

        $empty(P) \wedge nonempty(Q) : fill(Q, P, R)$

        $U_{IO} \geq T_{IO} \wedge U_{PE} \geq T_{PE} : (P, Q, R)$
    fi
```

where the function $nonempty(P)$ tests if the given list argument is a nonempty list, and $empty(P)$ tests if the given list argument is the empty list. U_{IO} is a measure for the I/O load on the system, and U_{PE} is a measure for the PE load on the system. These values are updated, whenever a process is removed from the ready queue or added to the expired queue. The constants T_{IO} and T_{PE} define load thresholds for I/O load and PE load respectively. The function $T_{rq}(p)$ returns the time process p has been scheduled to the second level scheduler.

The main function is $fill(P, Q, R)$, where P is the list of ready processes, Q is the list of expired processes, and R is the list of processes scheduled to the second-level scheduler. $fill$ is called whenever the system load (U_{IO} or U_{PE}) changes. The function $tick(P, Q)$ is called for every $C_{rq}/2$ system ticks, where P is the list of processes scheduled to the second level scheduler, and Q is the list of expired processes. Fill is called whenever the system load (U_{IO} or U_{PE}) changes.

The function *tick* is called for every $C_{rq}/2$ system ticks, and removes processes which have been scheduled for more than C_{rq} system ticks.

6.3.3 Implementation

The implementation of the first level scheduler can be found in appendix C.2 on page 129. The second level scheduler is almost identical to Linux scheduler, with the only change that `add_to_runqueue` and `del_from_qunqueue` has been renamed, as the functions in the first-level scheduler must be called. The implementation of the second-level scheduler is found in appendix C.2 on page 166.

The constants used in the implementation are listed in table 6.1.

Constant	Value
T_{PE}	4
T_{IO}	4
C_{rq}	400

Table 6.1: Constants used in the two-level scheduler implementation

6.3.4 Tests

Figure 6.9 on the following page shows the results of repeatedly compiling the kernel while increasing the number of concurrent compilation processes. It is observed that processes do not become I/O-bound as seen in the previous tests. This indicates that the two-level scheduling is not saturating the I/O system, and processes seem to behave better. It is also observed that the total execution time for the processes is not extended even though the processes are suspended for longer periods of time. It is therefore hypothesized that that the total compilation time is not significantly reduced, though no tests are made to confirm this.

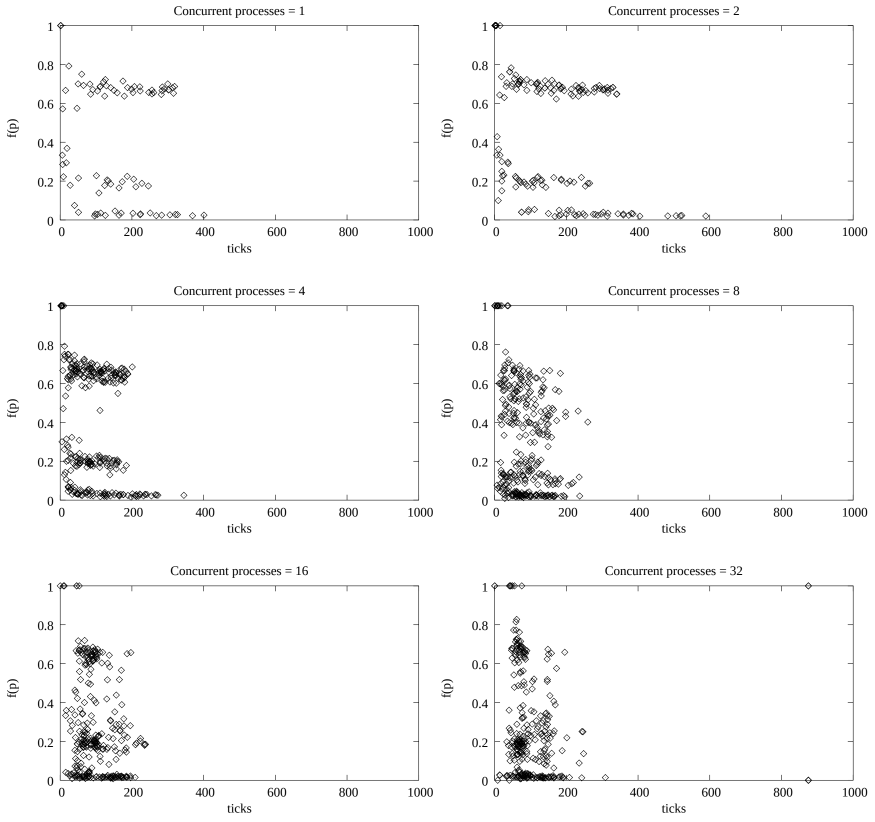

Figure 6.9: Simulation of Linux kernel compilation using the two-level scheduler

Static process mix

The results from tests using a static process mix, are equal to those found with the standard Linux scheduler. This is to be expected, as the load is low, and all processes are scheduled to the second level processor. The results are not graphed, as these are equal to those for the Linux scheduler.

6.4 Summary

This section will summarize the findings in this chapter. The section will conclude that while some of the experiments did lead to better scheduling properties, the current Linux scheduler is still competitive, even in multi-processor environments.

Round-robin scheduler

The round-robin has a very low administrative overhead, as scheduling is done in constant time. The PE's are utilized 100%, when work is available, as the algorithm has automatic load-sharing. The algorithm is very predictable which by [KL88] is a desirable property of a scheduler. The algorithm, however, has several disadvantages: I/O-bound processes are treated unfair compared to PE-bound processes, and process cache affinity is not honored.

Local-queue scheduler

The algorithm has the advantage that queue congestion is less likely to occur, and scheduling can be done on multiple PE's in parallel. Load-balancing is implemented, and the algorithm therefore results in slightly more overhead than the Linux scheduler. As the algorithm implemented uses dynamic priorities when scheduling from local queues, the scheduler behaves very closely to the Linux scheduler. I/O bound processes are still prioritized higher than PE-bound processes, though not as much as in the Linux scheduler. Process cache affinity is honored and processes are not bounced between PE's. The algorithm has the disadvantage that PE's are not always utilized 100%, even when processes in the ready state exists.

Two-level scheduler

Two-level scheduling, is implemented by suspending a set of processes for longer periods of time. While the load is low, this algorithm performs exactly as the Linux scheduler though a slightly administrative overhead is introduced in the first-level scheduling. In the compilation tests it has been shown that processes do not tend to become I/O bound, and it is hypothesized that *trashing*[2] is less likely to occur. The disadvantage of this algorithm, is that, when load increases response times becomes longer as processes are suspended for longer periods of time. Interactive processes would suffer greatly under this algorithm when there is a high load, why the algorithm cannot be used on system where interactive jobs are executed.

It has not been possible to improve the Linux scheduler through modifications to this, while maintaining all of the advantages in the existing Linux scheduler.

It is hypothesized that if knowledge of the type of jobs which would be executed on the system exists, this could be used to compile-time select the scheduler, which is the most efficient for the specific job-mix and usage. The list below suggests which job load for which the implemented algorithms performs the best:

Linux scheduler:
: Suitable for standard workstation use where few processes is in the running or ready state at a time, as this proves very good response times.

Round-robin scheduler:
: Suitable for systems with few PE's on which PE-bound processes are executed, as the scheduler has very low administrative overhead. An example is systems used to do large calculations.

Local-queue:
: Suitable for large systems with several PE's, where the number of processes in either ready or running state is significantly higher than the number of PE's. The process mix can be both PE-bound and I/O-bound processes. Examples of is login servers.

Two-level:
: Suitable for systems in where a very high load can exist, and resources are scarce compared to the load of the system. The system should not run any interactive processes. A Example of this is web-servers, which periodically receives huge amounts of hits, saturating the systems resources.

An alternative to compile-time selection of the scheduler algorithm, is to select the scheduler algorithm used by the kernel run-time. The run-time

[2]The term *tashing*, is used if all system resources are used for administrative tasks, such as swapping.

selection could be done by a super-user, who would change the scheduling algorithm if the usage pattern/job-mix on the system changes. Another possibility would be to let the kernel itself detect changes in the job-mix, and select the appropriate scheduler algorithm itself. It is to be noted, that this would require qualitative measurements in order to classify the job-mix present in the system, and that the scheduler algorithm must not be changed too often, as some administrative overhead would be expected when changing the scheduler.

Chapter 7

Status

This chapter contains an evaluation of the simulator and the modifications made, and an overview of possible future work on the simulator, including ideas for related projects.

7.1 Evaluation of the simulator

Much work has gone into designing and implementing the simulator. During the course of the project, the simulator have been extended in various ways, proving that the simulator has been modular enough to allow extensions to be added easily.

Portability

The simulator has been designed to allow portability of code between the simulator and the Linux kernel in terms of function signatures. This proved not to be an easy task as the scheduler implementation in Linux is not designed for changeability and no exact interface between scheduler and kernel exist as such. The chosen interface between scheduler implementation and the simulator does reflect actual functions in the Linux kernel, and as such does allow portability between the Linux kernel and simulator.

Calibration

The calibrations of the simulator have shown that the simulator can predict trends present in the current Linux scheduler. Static tests have proved that the simulator does produce results that are an exact match to those obtained from the Linux kernel. However, several limitations exist in the description of processes in the simulator. This meant that a lot of time went into calibrating the simulator. Also, it proved hard to find suitable tests as little material exists on this.

7.2 Evaluation of the Modifications

Three modifications have been designed and implemented, and it is hypothesized that real Linux implementation would show results equal to those found by the simulator. Preliminary tests of a round-robin scheduler in Linux have been made which have confirmed the above hypothesis. Due to time limitation and in order to focus on the scope, these tests have unfortunately not been included in this thesis. This could therefore be a recommendation for future work in order to prove the above hypothesis.

7.3 Further work

7.3.1 Simulator

Calibration of the simulator has shown that the process description is not sufficiently advanced to complex processes. A suggestion for future work would be to implement more complex process descriptions and behaviors. Below is a list of suggested possible extensions to the process modeling:

- Statistical models
  Process behaviors could be based on statistical models, which would more closely match real process behaviors.
- Interprocess communication
  Interprocess is often used to synchronize processes, and the possibility of modeling interprocess communication would be necessary if jobs using this were to be simulated.

The simulator has not been developed for examining real-time applications. It should however not require much work to introduce the notion of dead-lines in the simulator, and allow the simulator to test if dead-lines are reached. Future projects on real-time scheduling on Linux could benefit from such an extension.

7.3.2 Linux modifications

Results have shown that all of the implemented algorithms have both strength and weaknesses, and it is hypothesized that if it was possible to change the scheduler algorithm runtime, improvements could result.

Linux development kernel

A new scheduler algorithm by Ingo Molner have recently been implemented into the Linux development branch 2.5. The scheduler is a local-queue implementation with an complexity of O(1). For further studies on the subject of this theses, it is recommended that this scheduler algorithm is examined.

Chapter 8

Conclusion

8.1 The project

General scheduling theory has been presented, and the Linux scheduler implementation has been dissected.

Measurements for evaluating scheduler algorithms, in terms of PE service time and PE efficiency have been devised. The Linux scheduler has been evaluated and statistics obtained through a modified kernel were compared to theoretical values.

A simulator has been developed in order to ease evaluation of scheduler algorithms. The resulting simulator is highly modularized, allowing new process descriptions / behaviors to be added easily.

A interface between the simulator and scheduler implementation has been defined, matching a defined interface in the Linux kernel. This allows almost direct portability between simulator and kernel.

The simulator has been calibrated using both a static and a realistic process mix. It has been shown that the simulator can be used to assess the properties of different scheduler algorithms, though it has been suggested, that process descriptions could be improved.

Based on properties of the Linux kernel alternative algorithms were selected and subsequently implemented. Tests have shown, that while improvements to specific areas of the Linux scheduler are possible, it has not as hoped been possible to find a scheduler algorithm which is generally better than the Linux scheduler. It has been hypothesized that selecting algorithms based on machine work-load, can lead to improvements, though no further studies have been made on this subject.

8.2 The process

For me, the project has been both educational and interesting. Not only educational in terms of the subject of this thesis, but also in terms of project planning. The course of the project has been a mixed journey, where much time has been spend dissecting the Linux kernel, to understand exactly how the kernel and the scheduler in particular are implemented. Much time has been spent developing and calibrating the simulator, which for awhile removed focus from the main subject. This has however resulted in better understanding of process behaviors.

If any recommendation were to be given to fellow students, I would advice them to set up well-defined goals as early in the project as possible.

Bibliography

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distrubuted Programming.* Addison-Westley, 2000.

[BC01] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel.* O'Reilly & Associates, Inc., 2001.

[BL94] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, November 1994.

[BP] Robert D. Blimofe and Dionosios Papadopoulos. The performance of work stealing in multiprogrammed environments.

[DA99] Sivarama P. Dandamudi and Samir Ayachi. Performance of hierarchical processor scheduling in shared-memory multiprocessor systems. *IEEE Transactions on Computers*, 48(11):1202–1213, 1999.

[dSS00] Fabricio Alves Barbosa da Silva and Isaac D. Scherson. Improving parallel job scheduling using runtime measurements. In *JSSPP*, pages 18–38, 2000.

[ea02] Linux Thorvalds et. al. Linux v. 2.4.18 kernel source, 2002. http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.18.tar.gz.

[Fei97] D. Feitelson. Job scheduling in multiprogrammed parallel systems ibm research report rc, 1997.

[HS91] S. Haldar and D. K. Subramanian. Fairness in processor scheduling in time sharing systems. *Operating Systems Review*, Vol 25. Issue 1.:4–18, 1991.

[KF01] Mike Kravetz and Hubertus Franke. Multi-queue scheduler for linux. IBM Linux Technology Center, 2001.

http://lse.sourceforge.net/scheduling/mq1.html.

[KL88] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[Mit98] Michael Mitzenmacher. Analyses of load stealing models based on differential equations. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 212–221, 1998.

[ML92] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. Technical Report TR410, 1992.

[ML93] Evangelos P. Markatos and Thomas J. LeBlanc. Locality-based scheduling in shared-memory multiprocessors. Technical Report 94, FORTH-ICS / TR-094, 1993.

[NWZ01] Jason Nieh, Chris Waill, and Hua Zhong. Virtual-time round-robin: An O(1) proportional share scheduler. In *Proceedings of the 2001 USENIX Annual Technical Conferencen*, June 2001.

[Rub98] Alessandro Rubini. *Linux device drivers.* O'Reilly & Associates, Inc., 1998.

[Sta01] William Stallings. *Operating systems.* Prentice Hall, fourth edition, 2001.

[TW97] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems: Design and implementation.* Prentice Hall, second edition, 1997.

[Wu93] Chee-Shong Wu. Processor scheduling in multiprogrammed shared memory numa multiprocessors. Master's thesis, Department of Computer Science, University of Toronto, 1993. Describes some models for process execution time.

Index

Batch scheduling, 9

dynamic priority, 8

fairness, 7, 18

gang scheduling, 13
global queue, 8, 18

Hierarchical queues, 13
hyper-threading, 13

load-balancing, 11, 60
    lower threshold, 61
    upper threshold, 61
local queue, 11, 60

make-span, 8
Memory locality, 8

partition, 8
partitioning, 13
PE efficiency, 21
Perfect fairness, 22
process states, 16
proportional share, 21

quantum, 7, 9

Round-robin, 9, 54
    Virtual, 10
    Virtual-time, 10
    Weighted, 10
run time, 21

self-scheduling, 12
service time, 21
SMP, 15
space-sharing, 8

time-sharing, 7
two-level scheduling, 12, 66
    chunking, 12

UNIX, 15

work-sharing, 11, 61
    gradient, 11
work-stealing, 12, 61

Scheduling algorithms for Linux
Supplement

Anders Peter Fugmann

IMM-THESIS-2002-65

IMM

Trykt af IMM, DTU

Appendix A

Project description

English title: Linux scheduling algorithems
Advisor: Jørgen Steensgaard-Madsen
Period: 1/1 - 11/10-2002
Credit: 45 ECTS points
Participant: Anders Fugmann

English project description

The goal will be pursued partly by studying the relevant literature on the subject of scheduling algorithms for multiprocessor systems generally, partly by studying the actual implementation in the Linux kernel. Based on this studies, evaluation-criteria will be defined, and by relating the actual implementation to the theoretical work some suggestions for improvements might result. In any case, it is the goal to find quantitative qualities that can be tested and measured to assess variations in algorithms.

Appendix B

Linux augmentation

This section lists the changes made to the Linux kernel in order to implement instrumentation.

arch/i386/config.in

At line 425 is inserted:

```
425  tristate ' Scheduler statistics' CONFIG_DEBUG_STAT
```

Makefile

In the top-level Makefile for Linux, the versionnumber is changed by replacing line 4, with:

```
  4  EXTRAVERSION = -stat
```

include/linux/sched.h

The structure `task_struct` has been extended, by inserting the folloing at line 413:

```
413
414      /* time on run queue */
415      unsigned long run_queue_time;
416
417      /* Number of times a thread has been removed from the run queue */
418      unsigned long preemptions;
419
420      /* Total PE service time */
421      unsigned long cpu_time;
422
423      /* Number of times the process has been scheduled */
424      unsigned long sched_count;
425
426      /* Temporary variable holding the time of the last update of
427        time values (run_queue_time and cpu_time) */
428      unsigned long last_update;
429
430      /* flag to see if the timestamps has been updated */
431      unsigned long pe_add;
432
433      /* Variables used to calculate process charictaristics */
434      unsigned long intervals;
435      unsigned long interval;
436      unsigned long interval2;
437
438      unsigned long added;
439      unsigned long removed;
440      unsigned long cpu_interval;
```

In function `del_from_runqueue`, at line 898 is inserted:

```
898        p->preemptions++;
899
900        p->run_queue_time += jiffies - p->last_update;
901        p->last_update = -1;
902
903        /* Update the CPU/(CPU+IO) stats */
904        p->removed=jiffies;
```

linux/sched.c

At line 36 is inserted:

```
 36   #include "../drivers/stat/stat.h"
```

The function `add_to_runqueue` at line 325 is replaced with:

```
325   static inline void add_to_runqueue(struct task_struct * p)
326   {
     unsigned long io,cpu,frac;
     list_add(&p->run_list, &runqueue_head);
     nr_running++;

     p->last_update = jiffies;

     /* Updated the stats, but not the firs runqueue add */
     if (p->added < p->removed)
     {
        io=jiffies - p->removed;
        cpu=p->times.tms_stime - p->cpu_interval;
        frac=cpu*100/(cpu+io);
        p->interval+=frac;
        p->interval2+=frac*frac;
        p->intervals++;
        p->removed=jiffies;
        p->added=jiffies;
        p->cpu_interval=p->times.tms_stime;
     }
}
```

In function `schedule`, at line 697, is inserted:

```
697        if (prev->pe_add != -1)
698        {
699           prev->cpu_time += jiffies - prev-> pe_add;
700           prev->pe_add = -1;
701           prev->sched_count++;
702        }
703        next->pe_add = jiffies;
```

drivers/stat/Makefile

```
#
# Makefile for the kernel stat device driver.
#
# Gather statistics information at each schedule,
# And at each block or nonblock
#
# Anders Fugmann (afu@fugmann.dhs.org)

# CONFIG_DEBUG_STAT can either be ’y’ or undefined

O_TARGET := statistics.o
export-objs := stat.o
obj-$(CONFIG_DEBUG_STAT) += stat.o

include $(TOPDIR)/Rules.make

fastdep:
```

drivers/stat/stat.h

```
#ifndef __STAT_H__

#define __STAT_H__


#ifdef CONFIG_DEBUG_STAT

extern int stat_string(char* buffer, struct task_struct * p);
extern int stat_on(void);

#else

#define stat_string(buffer, p) 0
#define stat_on(void)   0

#endif /* CONFIG_DEBUG_STAT */


#endif /* __STAT_H__ */
```

drivers/stat/stat.c

```
/*******************************************************************
 * Stat character driver
 * The driver creates a character device, to which the current
 * stat of all tasks is written
 *
 * Written by: Anders Fugmann
 *
 *******************************************************************/

#include <linux/kernel.h>
#include <linux/module.h>

#include <linux/proc_fs.h>
#include <linux/wait.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/kdev_t.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/mm.h>
#include <linux/errno.h>
#include <linux/devfs_fs_kernel.h>
#include <linux/smp_lock.h>
#include <linux/spinlock.h>
#include <linux/ctype.h>
#include <asm/uaccess.h>
#include <linux/sched.h>

/* #include "stat.h" */

#define STAT_MINOR   1
```

```
/*=================================================================*
 * Forward declarations                                       *
 *=================================================================*/

static int __init stat_init(void);
static void __exit stat_exit(void);

static int stat_open(struct inode *inode, struct file *file);
static int stat_release(struct inode *inode, struct file *file);

/* File operations functions */
static loff_t stat_llseek(struct file *file, loff_t offset, int origin);
static int stat_open(struct inode *inode, struct file *file);
static int stat_release(struct inode *inode, struct file *file);

static ssize_t stat_read(struct file *file, char* buffer, size_t size,
                   loff_t *offset);

/* Entry end exit routines */
module_init(stat_init);
module_exit(stat_exit);

/*=================================================================*
 * Variables                                                  *
 *=================================================================*/
static devfs_handle_t devfs_handle;
static struct proc_dir_entry* proc_handle;

/* Major number */
static int major;

static volatile int stat_enabled;
static volatile int stat_open_lock = 0;
static volatile int stat_read_lock = 0;

/* File operations structure */
static struct file_operations stat_fops =
{
  THIS_MODULE,
  stat_llseek,
  stat_read,
  NULL, /* write nor allowed */
  NULL, /* readdir */
  NULL, /* poll */
  NULL, /* ioctl */
  NULL, /* mmap */
  stat_open,
  NULL, /* flush */
  stat_release,
  /* nothing more, fill with NULL's */
};


/*======================================================================*
 * Functions begin                                                 *
 *======================================================================*/

int stat_string(char* buffer, struct task_struct * p)
```

```
{
   unsigned long queue_time = p->run_queue_time;
   unsigned long cpu_time = p->cpu_time;
   if (p->last_update != -1)
      queue_time += jiffies - p->last_update;

   if (p->pe_add != -1)
      cpu_time += jiffies - p->pe_add;
   /* No divisions */
   /* unsigned long avg_wait_time = (100*wait_time)/p->times.tms_utime; */

   return
   sprintf(buffer,
         "tick:%lu cpu:%d pid:%d grp:%d policy:%lu "
         "nice:%lu rt_priority:%lu counter:%ld "
         "preemptions:%lu "
         "run_time:%lu "
         "run_queue_time:%lu "
         "cpu_time:%lu sched_count:%lu "
         "parent:%d "
         "time:%ld "
         "interval:%lu interval2:%lu intervals:%lu "
         "\n",
         jiffies, p->processor, p->pid, (int) p->pgrp, p->policy,
         p->nice, p->rt_priority, p->counter,
         p->preemptions,
         p->times.tms_utime + p->times.tms_stime,
         queue_time,
         cpu_time, p->sched_count,
         p->p_pptr->pid,
         (jiffies - p->start_time),
         p->interval, p->interval2, p->intervals);
}

int stat_on(void) {
   //return test_bit(0,&stat_open_lock);
   return stat_enabled;
}

/* procfs read/write operations */
int stat_proc_read(char *page, char **start, off_t offset, int count, int *eof, vo
{
   if (count > 1 && offset == 0)
   {
      page[0] = '0' + stat_enabled;
      page[1] = 0;
   }

   *eof = 1;
   return 2;
}

int stat_proc_write(struct file *file, const char *buffer, unsigned long count, vo
{
   char c;
   if (count > 0)
   {
      /* Read only one char */
      copy_from_user(&c,buffer,1);

      if (c == '0' || c == '1')
      {
         stat_enabled = c - '0';
         if (stat_enabled)
            printk(KERN_INFO "Stat: Statistics enabled.\n");
         else
            printk(KERN_INFO "Stat: Statistics disabled.\n");
      }
   }
   return 1;

}

static int __init stat_init(void)
{
   stat_enabled = 0;
   clear_bit(0, &stat_open_lock);
   clear_bit(0, &stat_read_lock);
   /* Create proc entry */
   proc_handle = create_proc_entry("sched_stat",
                            S_IFREG | S_IRUGO | S_IWUGO,
                            proc_root_driver);

    if (proc_handle) {
       proc_handle->read_proc=stat_proc_read;
       proc_handle->write_proc=stat_proc_write;
       printk(KERN_INFO "Stat: Registered proc device.\n");
    }


   major = devfs_register_chrdev(0, "stat", &stat_fops);
   printk(KERN_INFO "Stat: Registered with major number: %d\n", major);
   devfs_handle = devfs_mk_dir (NULL, "stat", NULL);
   devfs_register(devfs_handle, "sched",
              DEVFS_FL_DEFAULT,
              major, STAT_MINOR,
              S_IFCHR | S_IRUGO | S_IWUGO, &stat_fops, NULL);
   /* Create the buffer */
   return 0;
}

static void __exit stat_exit(void)
{
   devfs_unregister(devfs_handle);
   remove_proc_entry("sched_stat", proc_root_driver);
   printk(KERN_INFO "Stat: Modules deregistered\n");
}

static int stat_open(struct inode *inode, struct file *file)
{
   /* We can only handle one minor number */
   if (MINOR(inode->i_rdev) != STAT_MINOR)
      return -ENXIO;

   /* Only one reader at a time */
   if (test_and_set_bit(0,&stat_open_lock)) return -EBUSY;

   /* All ok, return success */
   return 0;

}
static int stat_release(struct inode *inode, struct file *file)
{
   clear_bit(0, &stat_open_lock);
   return 0;
}

/*
 * Seeking is not meaningfull for this type of device.
 */
static loff_t stat_llseek(struct file *file, loff_t offset, int origin)
{
 return -ESPIPE;
}

static ssize_t stat_read(struct file *file, char* buffer, size_t size,
                  loff_t *offset)
{
   int data_copied;
   char* local_buffer;
   int data_size;
   struct task_struct *p;

   if(test_and_set_bit(0, &stat_read_lock)) {
      //return -EBUSY;
   }
   local_buffer = kmalloc(size + 1024, GFP_KERNEL);
   if (!local_buffer) return -EAGAIN;

   /* Grab the run_queue_lock, disabling IRQ on the local
     CPU to avoid beeing interrupted in this process */
   /* IRQ's must be anabled at this time, so this is not dangerous */
   local_irq_disable();
   read_lock(&tasklist_lock);

   data_copied = 0;
   /* Dont go over the threshold */
   data_size = 0;
   for_each_task(p) {
      if (data_copied >= size)
         break;
      data_size = stat_string(&local_buffer[data_copied],p);
      data_copied += data_size;
      if (data_copied > size) {
         data_copied -= data_size;
         break;
      }
   }
   read_unlock(&tasklist_lock);
   local_irq_enable();
   /* Subtract offset, and add one for ending '0' */
   //data_copied++;

   /* Only copy data if there is data */
   if (data_copied > 0)
   {
      if(copy_to_user(buffer, local_buffer, data_copied) > 0)
         data_copied = -EFAULT;
   }
   else
      data_copied = 0;
   /* Clear the bit. */
   kfree(local_buffer);

   clear_bit(0, &stat_read_lock);
   return data_copied;

}

MODULE_LICENSE("GPL");
```

Appendix C

Simulator

C.1 Task description

Simulator control file for kernel compilation tests.

```
#define SCHED_OTHER     0
#define SCHED_FIFO      1
#define SCHED_RR        2

Globals: hz=100, cpus=2, io_resources=3, tasks=J, ticks=14000

# I/O-bound
process {
      pid:1, grp:1, nice:0, policy: 0, run: 0
      CPU Task:
      Fork Task: forkprob=(J,0) durration=(10,15) type='IO Task: cpu=(1,3), io=(50,100)'
}

process {
      pid:2, grp:1, nice:0, policy: 0, run: 0
      CPU Task:
      Fork Task: forkprob=(J,0) durration=(40,80) type='IO Task: cpu=(1,2), io=(5,10)'
}

process {
      pid:3, grp:3, nice:0, policy: 0, run: 0
      CPU Task:
      Fork Task: forkprob=(J,0) durration=(120,240) type='IO Task: cpu=(10,10), io=(5,0)
}
```

Simulator control file for static tests.

```
#define SCHED_OTHER     0
```

```
#define SCHED_FIFO      1
#define SCHED_RR        2

Globals: hz=100, cpus=2, io_resources=5, tasks=5, ticks=10000

process {
      pid:1, grp:1, nice:0, policy: 0, run: 0
      CPU Task:
}

process {
      pid:2, grp:1, nice:0, policy: 0, run: 1
      CPU Task:
      IO Task: cpu=(80,0), io=(20,0)
}

process {
      pid:3, grp:1, nice:0, policy: 0, run: 1
      CPU Task:
      IO Task: cpu=(60,0), io=(40,0)
}

process {
      pid:4, grp:1, nice:0, policy: 0, run: 1
      CPU Task:
      IO Task: cpu=(40,0), io=(60,0)
}

process {
      pid:5, grp:1, nice:0, policy: 0, run: 1
      CPU Task:
      IO Task: cpu=(20,0), io=(80,0)
}
```

C.2 Source

Makefile 106
cputask.cc 107
cputask.h 108
forktask.cc 108
forktask.h 111
global_sched.cc 112
global_sched.h 113
globals.cc 114
globals.h 114
iosched.cc 114
iosched.h 116
iotask.cc 116
iotask.h 119
killtask.cc 120
killtask.h 121
main.cc 121
main.h 124
periodictask.cc 124
periodictask.h 126
processes.cc 127
processes.h 129
sched-first_level.cc 129
sched-first_level.h 134
sched-local_queue.cc 135
sched-multi_queue.cc 147
sched-round_robin.cc 156
sched-second_level.cc 166
sched-test.cc 176
sched.cc 184
sched.h 194
sched_interface.h 196
simulator.h 196
simulator_interface.h 197
task.cc 197
task.h 201

Makefile

```
########################################################################
# Final Thesis
#
# Makefile for simulator
# Source : $RCSfile: Makefile,v $
#
# To change the scheduling algorithem used,
# modify the SCHEDULER variable below.
#
# Anders Fugmann.
#
#######################################################################

CFLAGS = -O2 -Wall -g -march=athlon -mcpu=athlon

TARGET = simulate

SCHED_LINUX = sched
SCHED_TWO_LEVEL = sched-first_level sched-second_level
SCHED_MULTI_QUEUE = sched-multi_queue
SCHED_LOCAL_QUEUE = sched-local_queue
SCHED_ROUND_ROBIN = sched-round_robin

# This variable holds the names of the C++ files to be linked.
# Use this to change the scheduler used.
SCHEDULER = $(SCHED_LINUX)

FILES = main processes globals task \
      iotask cputask periodictask killtask forktask iosched \
      global_sched $(SCHEDULER)

SRCS = $(addsuffix .cc, $(FILES)) $(addsuffix .cc, $(SCHEDULER))

OBJS = $(addsuffix .o, $(FILES))

TASKS=tasks

.PHONY: all dep clean

all: $(TARGET)

$(TARGET): $(OBJS) Makefile
      g++ $(CFLAGS) -o $(TARGET) $(OBJS)

.cc.o:
      g++ $(CFLAGS) -c -o $@ $<
.c.o:
      gcc $(CFLAGS) -c -o $@ $<

dep:
      gccmakedep $(SRCS)
clean:
      rm -f *o *~ $(TARGET) data* *.ps core Makefile.bak \
      *ps *fig
```

cputask.cc

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/times.h>
#include <time.h>

#include "cputask.h"
#include "task.h"
#include "sched_interface.h"

CpuTask::CpuTask()
{
   //initialisation
   prev_cpu = -1;
   cpu_switches = 0;
}

TaskProp* CpuTask::read(char* line)
{
   if (strcmp(line, "CPU Task:") == 0)
   {
      return new CpuTask();
   }
   else
      return NULL;
}


void CpuTask::statechange(enum task_state state, Task *task)
{

   /* Remember what cpu we was running on. This will have some
     influence on the run_time (cold cache etc.) */

   /* We do not react on this. */
   /* A cpu task just wants to know how muck time it gets. */

   /* This should be called whenever state changes from running to active */
}

int CpuTask::tick(Task *task)
{
   /* Check if the task wants to wait on some IO, or other stuff */
   if ((task->state == TASK_RUNNING) && (task->processor != prev_cpu))
   {
      cpu_switches++;
      prev_cpu = task->processor;
   }
   return 0;
}

void CpuTask::print(int tick)
{
   printf("cpu_switches:%d ", cpu_switches);
}

/* start a process */
```

```
void CpuTask::run()
{
   int i = 3;
   printf ("Pid: %d\t(CpuTask)\n", getpid());
   fflush(stdout);
   /* Just do some CPU work repeatably */
   while (1)
      i = i * i;
}
```

cputask.h

```
#ifndef __CPUTASK_H__
#define __CPUTASK_H__

#include "task.h"

class CpuTask : public TaskProp
{
 public:
   int run_time;
   int cpu_switches;
   int prev_cpu;
   int optim_time;
   int wait_time;
   int run_queue_time;

   void statechange(enum task_state state, Task *task);
   int tick(Task *task);
   void print(int tick);
   void run();

   static TaskProp* read(char* line);
   CpuTask();
};


#endif /* __CPUTASK_H__ */
```

forktask.cc

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/times.h>
#include <time.h>
#include <math.h>

#include "forktask.h"
#include "killtask.h"
#include "task.h"
#include "processes.h"
#include "global_sched.h"
```

```
#include "globals.h"

ForkTask::ForkTask(struct prop_struct *interval,
              struct prop_struct *durration,
              char *type)
{
   this->interval=interval;
   this->durration=durration;
   this->type=type;
   this->next_change=calc_prop(interval);

   forked_tasks = new list<Task*>;
   /* Ignore this pid as a live one */
   nr_tasks++;
}

TaskProp* ForkTask::read(char* line)
{
   struct prop_struct *interval = new prop_struct();
   struct prop_struct *durration = new prop_struct();


   char* type = new char[256];
   if (sscanf(line, "Fork Task: forkprob=(%d,%d) durration=(%d,%d) type=",
           &interval->mean, &interval->variance,
           &durration->mean, &durration->variance) == 4)
   {
      /* Read the type. */
      int start = strcspn(line, "'") + 1;
      int length = strcspn(&line[start], "'");
      strncpy(type, &line[start], length);
      return new ForkTask(interval, durration, type);
   }
   delete interval;
   delete durration;
   return NULL;
}


void ForkTask::statechange(enum task_state state, Task *task)
{
}

int ForkTask::tick(Task *task)
{
   int tasks = 0;
   /* Make sure that we have the expected number of tasks */
   list<Task*>::iterator iter;
   for (iter=forked_tasks->begin();iter != forked_tasks->end(); iter++)
      if ((*iter)->state != TASK_STOPPED)
         tasks++;
   /*else
    forked_tasks->remove(*iter); */

   while (next_change-tasks++ > 0)
   {
      /* Fork the process. Remember to add a terminating task */
      CpuTask *ct = new CpuTask();
      KillTask *kt = new KillTask(calc_prop(durration));
      Task* t = new Task(task->pid*1000+1+forks, task->grp,
                    task->nice, task->policy, 0);
      if (strlen(type) > 0)
      {
         TaskProp *tp = get_task_prop(type);
         if (tp == NULL)
            fprintf(stderr, "Error: Could not create Forked Task: ’%s’\n",
                 type);
         else
            t->add_task_prop(tp);

      }
      t->add_task_prop(ct);
      t->add_task_prop(kt);
      prop_list.push_back(ct);
      /* Let the simulator and system know of this process */
      global_add_task(t);
      forked_tasks->push_front(t);
      forks++;
   }
   next_change=calc_prop(this->interval);
   /* Suspend if running */
   if (task->state == TASK_RUNNING)
   {
      task->state = TASK_SUSPENDED;
      return 1 << task->processor;
   }
   return 0;
}

void ForkTask::print(int tick)
{
   float run_time=0;
   float efficiency=0;
   /* Go through all Cpu properties */
   CpuTask* ct;
   list<CpuTask*>::iterator iter;
   for (iter = prop_list.begin(); iter != prop_list.end(); iter++)
   {
      ct = (*iter);
      run_time += ct->run_time;
      efficiency += ct->run_time * 1.0 / ct->run_queue_time;
   }
   printf("fork_forked:%d fork_runtime:%f, fork_efficiency:%f ",
        forks, run_time/forks, efficiency/forks);
}

/* start a process */
void ForkTask::run()
{
   /* This is gonna be hard */
}

int ForkTask::calc_prop(struct prop_struct *prop)
{
   if (prop->mean == -1) return INT_MAX; /* Never */
   float length = prop->mean;
   length += (2.0*random()/RAND_MAX-1.0) * prop->variance/2.0;
   return (int) round(length);
}

ForkTask::~ForkTask()
{
   /* Delete all tasks */

}
```

forktask.h

```
#ifndef __FORKTASK_H__
#define __FORKTASK_H__

#include "task.h"
#include "cputask.h"
struct prop_struct
{
   int mean;
   int variance;
};


class ForkTask : public TaskProp
{
 public:
   /* State attributes */
   struct prop_struct *interval;
   struct prop_struct *durration;
   char *type;
   int next_change;

   /* Statictics attributes */
   int forks;
   int terminations;
   list<CpuTask*> prop_list;
   list<Task*> *forked_tasks;

   /* Functions */
   ForkTask(struct prop_struct *interval,
         struct prop_struct *durration,
         char *type);
   virtual ~ForkTask();

   void statechange(enum task_state state, Task *task);
   int tick(Task *task);
   void print(int tick);
   void run();

   static TaskProp* read(char* line);
   int calc_prop(struct prop_struct *prop);
};

#endif /* __FORKTASK_H__ */
```

global_sched.cc

```
#include "global_sched.h"
#include "sched_interface.h"
#include "task.h"
#include "globals.h"

/* List of task sources */
list<struct task_source *> sources;

void global_add_task(Task* p)
{
   int id = p->grp;
   struct task_source *ts = NULL;
   /* Find the task source */
   list<struct task_source *>::iterator i;

   for (i=sources.begin();i != sources.end(); i++)
   {
      if ((*i)->id == id)
      {
         ts = *i;
         break;
      }
   }
   /* If none was found, create a new entry */
   if (i == sources.end())
   {
      ts = new struct task_source;
      ts->task_list = new list<Task*>;
      ts->id = id;
      sources.push_back(ts);
   }
   ts->task_list->push_back(p);
}

/* Make sure that we have excatly <tasks> running tasks */
/* If there is less, only one task is added */
void global_sched(int nr_tasks)
{
   Task* p;
   list<Task*>::iterator it;
   /* First find out if any new tasks are needed */
   for (it = tasks.begin();it != tasks.end(); it++)
      if ((*it)->state != TASK_STOPPED)
         nr_tasks--;

   //printf("### Need to add %d tasks\n", nr_tasks);

   /* Still tasks to add? */
   static list<struct task_source *>::iterator iter;
   if (iter == NULL)
      iter = sources.begin();

   list<Task*> *selected = NULL;
   while (nr_tasks-- > 0)
   {
      /* This implements a RR.
```

```
        To store where we are, a static variable is used */

      for(unsigned int i=0; i < sources.size(); i++)
      {
         iter++;

         if (iter == sources.end())
         {
            iter = sources.begin();
         }

         if ((*iter)->task_list->size() > 0)
         {
            selected = (*iter)->task_list;
            break;
         }
      }

      /* If no lists was found, stop processing */
      if (selected == NULL)
         return;

      p = selected->front();
      p->state = TASK_RUNNING;
      tasks.push_front(p);
      wake_up_process(p);

      selected->remove(p);
      selected = NULL;
   }
}
```

global_sched.h

```
#ifndef __GLOBAL_SCHED_H__
#define __GLOBAL_SCHED_H__


#include "task.h"
#include "main.h"

extern list<struct task_source *> sources;

struct task_source {
   int id;
   list<Task*> *task_list;
};

void global_add_task(Task* p);
void global_remove_task(Task* p, int id);
void global_sched(int tasks);


#endif /* __GLOBAL_SCHED_H__ */
```

globals.cc

```
#include "globals.h"
#include "task.h"

/* Number of PE’s in the system */
int smp_num_cpus;

/* Number of ticks to simulate */
int ticks;

/* Number of IO resources */
int io_resources;

/* System freq */
int HZ;

/* Total number of tasks in the system */
int nr_tasks;

/* Every task that has ever exsisted in the simulation */
list<Task*> tasks;

/* Jiffies counter */
cycles_t jiffies;
```

globals.h

```
/* Global defines for the program */
#ifndef __GLOBALS_H__
#define __GLOBALS_H__

#include "task.h"
#define cycles_t unsigned int

extern int ticks;
extern int smp_num_cpus;
extern int io_resources;

extern int HZ;
extern int nr_tasks;

extern list<Task*> tasks;

extern cycles_t jiffies;

#endif /* __GLOBALS_H__ */
```

iosched.cc

```
#include <list>
#include <vector>
```

```
#include <algorithm>
#include "task.h"
#include "globals.h"
#include "iosched.h"


/* The io queue */
static list<Task*> io_queue;
/* List of tasks receiving IO */
static list<Task*> active;

/* Return true, if the thread is not already assigned to an IO resource */
int can_ioschedule(Task* task)
{
   return (!task_has_io(task));
}

/* Return true, if the process is retriving IO */
int task_has_io(Task* task)
{
   return (find(active.begin(), active.end(), task) != active.end());
}

/* Return true, if the task is waiting for IO */
int task_on_ioqueue(Task* task)
{
   return (find(io_queue.begin(), io_queue.end(), task) != io_queue.end());
}

/* Add a task to the IO-queue */
void add_task_ioqueue(Task* task)
{
   /* Add to tail */
   io_queue.push_back(task);
}

/* Remove a task to the IO-queue */
void remove_task_ioqueue(Task* task)
{
   io_queue.remove(task);
}

/* Schedule tasks waiting for IO */
/* Should be called for each tick */

void iosched(void)
{
   /* Remove all tasks from the active queue, that no longer
    * wants IO.
    * Also count the number of tasks receiving IO */

   int active_tasks = 0;
   list<Task*>::iterator iter;
   for (iter = active.begin(); iter != active.end(); iter++)
   {
      if ((*iter)->state == TASK_SUSPENDED)
         active_tasks++;
      else
      {
         //printf("### Removing pid from IO: %d\n",(*iter)->pid);
         active.remove(*iter);
         /* Remove also from the IO wait queue */
         io_queue.remove(*iter);
         iter--;
      }
   }

   /* Add upto io_resources to the list */
   while (active_tasks < io_resources)
   {
      for (iter = io_queue.begin(); iter != io_queue.end(); iter++)
      {
         /* Only schedule for IO, if it does not retrieve IO */
         if (can_ioschedule(*iter))
         {
            active.push_back(*iter);
            active_tasks++;
            break;
         }
      }
      if (iter == io_queue.end())
         break;
   }
}
```

iosched.h

```
#ifndef __IOSCHED_H__
#define __IOSCHED_H__

#include "task.h"

/* Return true, if the process is retriving IO */
int task_has_io(Task* task);

/* Return true, if the task is waiting for IO */
int task_on_ioqueue(Task* task);

/* Add a task to the IO-queue */
void add_task_ioqueue(Task* task);

/* Remove a task to the IO-queue */
void remove_task_ioqueue(Task* task);

/* Schedule tasks waiting for IO */
void iosched(void);

#endif /* __IOSCHED_H__ */
```

iotask.cc

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/times.h>
#include <time.h>
#include <stdio.h>
#include <math.h>

#include "iotask.h"
#include "task.h"
#include "sched_interface.h"
#include "globals.h"
#include "iosched.h"

IoTask::IoTask(struct prob_struct *cpu, struct prob_struct *io)
{
   prob_cpu = cpu;
   prob_io = io;

   cpu_length = prob_calc_length(prob_cpu);
   io_length = prob_calc_length(prob_io);

   last_cpu = 0;
   last_io = 0;
}


IoTask::~IoTask()
{
   free(prob_cpu);
   free(prob_io);
}

/* Construct a ioprob from a string */
TaskProp* IoTask::read(char* line)
{
   prob_struct *cpu, *io;
   cpu = (prob_struct*) malloc(sizeof(prob_struct));
   io = (prob_struct*) malloc(sizeof(prob_struct));

   if (sscanf(line, "IO Task: cpu=(%d,%d), io=(%d,%d)",
           &cpu->mean, &cpu->variance, &io->mean, &io->variance) == 4)
      return new IoTask(cpu, io);
   else
   {
      free(cpu);
      free(io);
      return NULL;
   }

}

void IoTask::statechange(enum task_state state, Task *task)
{
}

int IoTask::tick(Task *task)
{
   /* Check if the task wants to wait on some IO, or other stuff */
   /* we use the statistics counter from task */
   switch (task->state)
   {
      case TASK_RUNNING:
         if (task->cpu_time - last_cpu >= cpu_length)
         {
            io_length = prob_calc_length(prob_io);
            task->state = TASK_SUSPENDED;
            /* Add to io_scheduler */
            add_task_ioqueue(task);
            last_io = task->io_time;
            return 1 << task->processor;
         }
         break;

      case TASK_SUSPENDED:
         if (task->io_time - last_io >= io_length)
         {
            cpu_length = prob_calc_length(prob_cpu);
            task->state = TASK_RUNNING;
            /* Remove from the io_scheduler */
            remove_task_ioqueue(task);
            last_cpu = task->cpu_time;
            wake_up_process(task);
            return 1 << task->processor;
         }
         break;

      default:
         break;
   }
   return 0;
}

void IoTask::print(int tick)
{
}

/* Calcualte the length of a event */
int IoTask::prob_calc_length(struct prob_struct *prob)
{
   if (prob->mean == -1) return INT_MAX;
   float length = prob->mean;
   length += (2.0*random()/RAND_MAX-1.0) * prob->variance/2.0;
   return lroundf(length);
}

/* start a process */
void IoTask::run()
{
   struct tms buf, nbuf;
   clock_t time, ntime;
   struct timespec req, rem;
   int cpu_length, io_length;

   printf ("Pid: %d\t(IoTask) (%d, %d)\n", getpid(), prob_cpu->mean, prob_io
        ->mean);
   fflush(stdout);
   while (1)
   {
      cpu_length = prob_calc_length(prob_cpu);
      /* First get the req cpu_time */
      time = times(&buf);
      ntime = times(&nbuf);

      while (nbuf.tms_utime - buf.tms_utime < cpu_length)
      {
         cputime((cpu_length - (nbuf.tms_utime - buf.tms_utime)) * 10000000);
         ntime = times(&nbuf);
      }
      ntime = times(&nbuf);
      /* Ok now sleep the io time */
      io_length = prob_calc_length(prob_io);

      req.tv_sec = io_length/HZ;
      req.tv_nsec = (io_length%HZ) * 1000*1000*(1000/HZ);
      nanosleep(&req, &rem);
   }


}
```

iotask.h

```
#ifndef __IOTASK_H__
#define __IOTASK_H__

#include "task.h"

/* Struct to define io proberbility */
struct prob_struct {
   int mean; /* Avarage length in ticks. */
   int variance; /* mean_time + [-1;1]*variance/2.0 */
};

class IoTask : public TaskProp
{
 public:
   /* Prob for io operations. */
   struct prob_struct *prob_cpu;
   struct prob_struct *prob_io;

   /* Internal variables */
   int cpu_length;
   int io_length;
   int last_cpu;
   int last_io;

   void statechange(enum task_state state, Task *task);
   int tick(Task *task);
   void print(int tick);
   int prob_calc_length(struct prob_struct *prob);
   void run();
   static TaskProp* read(char *line);
   IoTask(struct prob_struct *cpu, struct prob_struct *io);
   virtual ˜IoTask();
};

#endif /* __IOTASK_H__ */
```

killtask.cc

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/times.h>
#include <time.h>

#include "killtask.h"
#include "task.h"
#include "sched_interface.h"

KillTask::KillTask(int lifetime)
{
   this->lifetime = lifetime;
}

TaskProp* KillTask::read(char* line)
{
   int lifetime;

   if (sscanf(line, "Kill Task: lifetime=%d", &lifetime) == 1)
   {
      return new KillTask(lifetime);
   }
   return NULL;
}


void KillTask::statechange(enum task_state state, Task *task)
{
   /* Nothing to do */
}

int KillTask::tick(Task *task)
{
   /* Do only if we are running */
   if (task_has_cpu(task)) lifetime--;
   if (lifetime <= 0)
   {
      /* printf("### Stopping task: %d\n", task->pid); */
      task->state = TASK_STOPPED;
      return 1;
   }
   return 0;
}
void KillTask::print(int tick)
{
   /* No extra output */
}

/* start a process */
void KillTask::run()
{
   /* Not implemented */
}
```

killtask.h

```
#ifndef __KILLTASK_H__
#define __KILLTASK_H__

#include "task.h"

class KillTask : public TaskProp
{
 public:
   /* State attributes */
   int lifetime;

   KillTask(int lifetime);

   void statechange(enum task_state state, Task *task);
   int tick(Task *task);
   void print(int tick);
   void run();

   static TaskProp* read(char* line);
};

#endif /* __KILLTASK_H__ */
```

main.cc

```
#include <list>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include "task.h"
#include "sched_interface.h"
#include "simulator_interface.h"
#include "iosched.h"
#include "globals.h"
#include "processes.h"
#include "global_sched.h"
#include "main.h"

using namespace std;

int current_processor;
int smp_processor_id()
{
   return current_processor;
}

cycles_t get_cucles()
{
   return 0;
}

void smp_send_reschedule(int cpu)
{
   /* Normally we need to send a request to the other CPU */
   int old_cpu = current_processor;
   current_processor=cpu;
   schedule();
   current_processor=old_cpu;
}


void timer_tick()
{
   jiffies++;
   for (int cpu=0;cpu<smp_num_cpus;cpu++)
   {
      current_processor=cpu;
      do_timer();
      if (get_current()->need_resched)
         schedule();
   }
   iosched();
}

/* Print out all statistics */
void statistics(list<Task*> *task_list, int tick)
{
   list<Task*>::iterator iter;
   for (iter = task_list->begin();iter != task_list->end(); iter++)
   {
      (*iter)->print(tick);
   }
}

void tick(int ticks, list<Task*> *task_list, int print)
{

   /* Send this many ticks */
   for (int ttick=1;ttick<=ticks;ttick++)
   {
      /* Make sure that the tasks are there */
      global_sched(nr_tasks);
      timer_tick();

      /* Send a tick to all tasks */
      list<Task*>::iterator iter;
      for (iter = task_list->begin();iter != task_list->end(); iter++)
      {
         /* Call schedule on the current cpu? */
         /* Dont touch stopped_tasks */
         if ((*iter)->state == TASK_STOPPED) continue;

         unsigned int sched_cpus = (*iter)->tick();

         if (sched_cpus)
         {
            /* Schedule on selected cpu's */
            for (int i=0;i<smp_num_cpus;i++)
               if (sched_cpus | (1 << i))
               {
                  current_processor = i;
                  schedule();
                  iosched();
               }
         }
      }
      /* Glocal scheduler */
      //global_sched(nr_tasks);
      if (print)
         statistics(task_list, ttick);
   }
}

void run_tasks(list<Task*> l)
{
   list<Task*>::iterator iter;
   printf("Starting jobs.\n");
   for (iter = l.begin();iter != l.end(); iter++)
   {
      (*iter)->run();
   }
   fflush(stdout);
   wait(NULL);
}

int main(int argc, char *argv[])
{
   int cpu=0;
   int last = 0;
   list<Task*> *task_list;
   if (argc == 1)
   {
      printf("Usage: %s <task_file> [run] [last]\n",argv[0]);
      printf(" task_file: A file containing the definition of "
           "tasks to simulate.\n");
      printf(" run:   Given this keyword, the tasks will "
           "be executed\n");
      printf(" last:  Only print out final stats\n");
      return 1;
   }

   task_list = read_tasks(argv[1]);

   /* Should the tasks be started on the OS? */

   if (argc >= 3 && !strcmp("run", argv[2]))
   {
      run_tasks(*task_list);
      return 0;
   }

   if (argc >= 3 && !strcmp("last", argv[2]))
      last = 1;

   /* Begin schedule all tasks */
   list<Task*>::iterator iter;

   for (iter = task_list->begin();iter != task_list->end(); iter++)
   {
      if ((*iter)->pid >= 0)
         global_add_task(*iter);
      else /* Idle_task */
      {
         /* No stats can be retrieved for those */
         current_processor = cpu++;
         sched_init(*iter);
         tasks.push_front(*iter);
         nr_tasks++;

      }
   }
   /* Lets start the clock */
   tick(ticks, &tasks, !last);

   if (last)
   {
      statistics(&tasks, ticks);
   }
}
```

main.h

```
#ifndef __MAIN_H_
#define __MAIN_H_

#endif /* __MAIN_H_ */
```

periodictask.cc

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/times.h>
#include <time.h>

#include "periodictask.h"
#include "task.h"
#include "sched_interface.h"
#include "globals.h"
#include <math.h>
```

```
PeriodicTask::PeriodicTask(int period, int cpu_time)
{
   this->period = period;
   this->cpu_time = cpu_time;

   /* Internal variables */
   period_left=0, cpu_left=cpu_time, cpu_overdue=0, max_cpu_overdue=0;
}

/* Construct from a string */
TaskProp* PeriodicTask::read(char* line)
{
   int period, cpu;

   if (sscanf(line, "Periodic Task: period=%d, cpu=%d", &period, &cpu) == 2)
      return new PeriodicTask(period, cpu);
   else
      return NULL;
}

void PeriodicTask::statechange(enum task_state state, Task *task)
{
}

int PeriodicTask::tick(Task *task)
{
   if (task->state == TASK_STOPPED) return 0;
   --period_left;

   if (task_has_cpu(task) && --cpu_left <= 0) /* Implies task->state !=
        TASK_STOPPED */
   {
      /* Now sleep */
      task->state = TASK_SUSPENDED;
      return 1 << task->processor;
   }

   /* New period */
   if (period_left <= 0)
   {
      /* Remember cpu overdue */
      cpu_overdue += cpu_left;
      if (cpu_left > max_cpu_overdue)
         max_cpu_overdue = cpu_left;

      cpu_left = cpu_time;
      period_left = period;

      task->state = TASK_RUNNING;
      wake_up_process(task);
   }

   return 0;
}

void PeriodicTask::print(int tick)
```

```
{
   printf("period_left:%d per_cpu_left:%d per_cpu_overdue:%d
        per_max_cpu_overdue:%d ",
         period_left, cpu_left, cpu_overdue, max_cpu_overdue);
}

void PeriodicTask::run()
{
   struct tms buf, nbuf;
   clock_t time, ntime;
   struct timespec req, rem;
   long sleep_time;

   printf ("Pid: %d\t(PeriodicTask)\n", getpid());
   fflush(stdout);

   while (1)
   {
      /* First get the req cpu_time */
      time = times(&buf);
      ntime = times(&nbuf);

      while (nbuf.tms_utime - buf.tms_utime < cpu_time )
      {
         cputime((cpu_time - (nbuf.tms_utime - buf.tms_utime)) * 10000000);
         ntime = times(&nbuf);
      }
      ntime = times(&nbuf);
      /* Ok now sleep untill the period is over */
      sleep_time = period - (ntime - time);
      req.tv_sec = sleep_time/HZ;
      req.tv_nsec = (sleep_time%HZ)*1000000000/HZ;
      nanosleep(&req, &rem);
   }

}
```

periodictask.h

```
#ifndef __PERIODICTASK_H__
#define ___PERIODICTASK_H__

#include "task.h"

class PeriodicTask : public TaskProp
{
 public:
   int cpu_time;
   int period;

   /* Internal variables */
   int period_left;
   int cpu_left;

   /* Statistics variables */
   int max_cpu_overdue;
   int cpu_overdue;

   void statechange(enum task_state state, Task *task);
   int tick(Task *task);
   void print(int tick);
   void run();


   static TaskProp* read(char *line);
   PeriodicTask(int period, int cpu_time);
};

#endif /* ___PERIODICTASK_H__ */
```

processes.cc

```
#include <list>
#include <stdio.h>
#include "globals.h"
#include "task.h"
#include "iotask.h"
#include "cputask.h"
#include "periodictask.h"
#include "processes.h"
#include "killtask.h"
#include "forktask.h"

using namespace std;

/* Read all processes from a file
 * The structure if the file is defined by each subclass
 * A line is scanned, and given to all known modules, and if one
 * recognices it, it returns a class to be added to the property list
 */

char* read_line(char* buf, int len, FILE* f)
{
   do {
      if (!fgets(buf, len, f))
         return NULL;
   } while (buf[0] == '#');

   /* Remove all tabs and spaces */
   int i=0;

   while (buf[i]==' ' || buf[i]=='\t') i++;
   if (buf[strlen(buf)-1] == '\n')
      buf[strlen(buf)-1] = 0;
   return &buf[i];
}

list<Task*> *read_tasks(char* file_name)
{
   int ln = 0;
   list<Task*> *l = new list<Task*>;
   Task *t;
```

```
char buf[255];
char* line;


/* Read in the globals */
FILE* file;
if (!(file = fopen(file_name, "r")))
{
   fprintf(stderr,"Fatal: Unable to open file: %s\n", file_name);
   exit(-1);
}

while ((line = read_line(buf, 255, file)))
{
   if (line[0] != 0) break;
}

if (sscanf(line, "Globals: hz=%d, cpus=%d, io_resources=%d, "
        "tasks=%d, ticks=%d",
        &HZ, &smp_num_cpus, &io_resources, &nr_tasks, &ticks) != 5)
{
   fprintf(stderr, "Fatal: Globals must be defined\n");
   exit(-1);
}

/* Create idle_tasks */
for (int i=1; i <= smp_num_cpus;i++)
{
   t = new Task(-i,0,0,0,-1);
   t->add_task_prop(new CpuTask());
   (*l).push_front(t);
}


/* For each line */
while ((line = read_line(buf, 255, file)))
{
   ln++;

   /* Read "process {" */
   if (strcmp("process {", line) == 0)
   {
      /* Found a process entry
       * Start processing */
      if (!(line = read_line(buf, 255, file)))
         printf("### Missing process parameters on line: %d\n", ln);
      ln++;

      if ((t = Task::read(line)))
      {
         for (;;)
         {
            TaskProp* tp;
            if (!(line = read_line(buf, 255, file)))
            {
               printf("### Missing ’}’ on line: %d\n", ln);
               break;
            }
```

```
               ln++;
               if (line[0] == '}') break;

               /* Let all properties read the line */
               tp = get_task_prop(line);
               if (!tp) {
                  printf("### Unknow property '%s' on line: %d\n", line, ln);
                  continue;
               }
               t->add_task_prop(tp);
            }
         }
         else
            printf("### Could not read task definitions\n");
         (*l).push_front(t);
      } else
         if (line[0] != 0) printf("### Unknown line(%d): '%s'\n", ln, line);
   }
   fclose(file);

   printf("### Found and scanned %d tasks\n", (*l).size()-smp_num_cpus);

   return l;
}

TaskProp* get_task_prop(char* line) {
   TaskProp* tp;
   if ((tp = IoTask::read(line))) return tp;
   if ((tp = CpuTask::read(line))) return tp;
   if ((tp = PeriodicTask::read(line))) return tp;
   if ((tp = KillTask::read(line))) return tp;
   if ((tp = ForkTask::read(line))) return tp;
   fprintf(stderr, "Error: Cannot create property: %s\n", line);
   return NULL;
}
```

processes.h

```
#ifndef __PROCESSES_H__
#define __PROCESSES_H__

#include <list>
using namespace std;

list<Task*> *read_tasks(char* file_name);
TaskProp* get_task_prop(char *line);
#endif /* __PROCESSES_H__ */
```

sched-first_level.cc

```
#include <list>
#include <vector>
```

```
#include <math.h>
#include <algorithm>
#include "sched-first_level.h"
#include "task.h"
#include "globals.h"
#include "sched_interface.h"
#include "simulator_interface.h"
#include "sched.h"

/* List of task in the 2. level scheduler */
static list<Task*> *expired;
static list<Task*> *ready;


/* Utilization counters */
/* We have: IO Length, CPU Length. CPU = CPU/CPU+IO IO=IO/IO+CPU */

static int u_cpu;
static int u_io;
static int last_swap;

void add_utilization(Task *t)
{
   if (t->util_add==1) return;
   if (t->cpu_length > 0 || t->io_length >= 0)
   {
      u_cpu += 100*t->cpu_length/(t->cpu_length+t->io_length);
      u_io += 100*t->io_length/(t->cpu_length+t->io_length);
   }
   t->util_add = 1;
}

void sub_utilization(Task *t)
{
   if (t->util_add==0) return;
   if (t->cpu_length > 0 || t->io_length >= 0)
   {
      u_cpu -= 100*t->cpu_length/(t->cpu_length+t->io_length);
      u_io -= 100*t->io_length/(t->cpu_length+t->io_length);
   }
   t->util_add = 0;
}


void update_task_io(Task* t)
{
   if (t->io_start > -1)
      t->io_length += jiffies - t->io_start;

   /* And CPU starts. */
   t->cpu_start = jiffies;
}

void update_task_cpu(Task* t)
{
   if (t->cpu_start > -1)
      t->cpu_length += jiffies - t->cpu_start;
   /* And IO starts */
   t->io_start = jiffies;
}


Task* get_task()
{
   list<Task*>::iterator iter;
   int u_sys, u_best;
   Task* next = NULL;
   /* Choose a new task */
   u_sys=100*u_cpu/(u_cpu+u_io+1); /* Range [0..100] */
   u_best=101;
   int u_task;

   /* Find a task in the ready queue,
     with stats as close as possible to uSYS.
     By tests above the queue cannot be empty. */

   for (iter = ready->begin();iter != ready->end(); iter++)
   {
      if ((*iter)->cpu_length > 0)
         u_task = 100*(*iter)->cpu_length/
            ((*iter)->cpu_length+(*iter)->io_length);
      else
         u_task=START_UTIL;

      if (abs(u_task-u_sys) < u_best)
      {
         next = *iter;
         u_best = abs(u_task-u_sys);
      }
   }
   return next;
}

void swap_queues()
{
   list<Task*> *tmp = ready;
   ready = expired;
   expired = tmp;
   last_swap=jiffies;
}

void high_level_stats()
{
   printf("### Tasks on ready queue: ");
   list<Task*>::iterator iter;
   for (iter = ready->begin(); iter != ready->end(); iter++)
      printf("%d ", (*iter)->pid);
   printf("\n");

   printf("### Tasks on expired queue: ");
   for (iter = expired->begin(); iter != expired->end(); iter++)
      printf("%d ", (*iter)->pid);
   printf("\n");

   printf("### Utilization. cpu:%d, io:%d\n",
        u_cpu, u_io);

}

/* Schedule processes to the low level scheduler,
  until the threasholds are up */
void fill()
{
   Task* next;
   while ((u_cpu < IO_THRESHOLD) || (u_cpu < CPU_THRESHOLD) ||
        (run_queue.size()==0) )
   {
      if (ready->size() == 0)
      {
         if (expired->size() > 0)
            swap_queues();
         else
         {
            /* Break, since there are no process to schedule */
            break;
         }
      }
      /* Select a new task */
      next = get_task();
      if (next == NULL) break;
      next->sched_stamp=jiffies;
      next->cpu_start = jiffies;
      ready->remove(next);
      add_utilization(next);
      sc_add_to_runqueue(next);
   }
   //high_level_stats();
}

/* This function is called whenever a process enters the runqueue,
  and is the base for all desicions. This is where processes
  are trapped by the first level scheduler */

void add_to_runqueue(Task* t)
{
   if (task_on_runqueue(t)) return;

   if (t->io_length < 1)
   {
      /* New process */
      t->io_start=jiffies;
      t->io_length=1;
      t->cpu_length=1;
      expired->push_back(t);
      fill();
      return;
   }

   t->cpu_start=jiffies;

   /* Has the task expired? */
   if (jiffies - t->sched_stamp > TASK_THRESHOLD)
   {
      sub_utilization(t);
      expired->push_back(t);
      fill();
      return;
   }

   /* Update global utilization counters */
   sub_utilization(t);
   update_task_io(t);
   add_utilization(t);
   sc_add_to_runqueue(t);
}


/* Remove a tasks from the rq.
  The function updates the task stats,
  and if the task has been terminated, fill is called */

void del_from_runqueue(Task* t)
{
   sc_del_from_runqueue(t);

   if (t->state == TASK_STOPPED)
   {
      //printf("### Task is stopped: %d\n", t->pid);
      sub_utilization(t);
      fill();
      return;
   }
   /* Recalcualte task info */
   sub_utilization(t);
   update_task_cpu(t);
   add_utilization(t);
   t->io_start=jiffies;
}

/* For each tick, this functions is called.
  It periodically tests for expired tasks, and moves them to the
  expired queue. It then add new tasks to the queue from the ready task
  list */

void high_level_tick()
{
   Task* t;
   list<Task*>::iterator iter;

   /* If the period has not yet expired */
   if (jiffies - last_swap < HIGH_LEVEL_PERIOD)
      return;

   last_swap = jiffies;

   for (iter = run_queue.begin();iter != run_queue.end(); iter++)
   {
      t = *iter;
      if ((jiffies - t->sched_stamp) > TASK_THRESHOLD)
      {
         /* Remove the process from the rq */
         iter=run_queue.erase(iter);

         /* Preempt the process, if needed */
         if (task_has_cpu(t))
            smp_send_reschedule(t->processor);

         sub_utilization(t);

         update_task_cpu(t);
         expired->push_back(t);
      }
   }
   fill();
}

void high_level_init()
{
   expired = new list<Task*>;
   ready = new list<Task*>;
   u_cpu=0;
   u_io=0;
   last_swap = 0;
}
```

sched-first_level.h

```
#ifndef __TWO_LEVEL_H__
#define __TWO_LEVEL_H__

#include "task.h"

#define IO_THRESHOLD 200*smp_num_cpus
#define CPU_THRESHOLD 200*smp_num_cpus
#define HIGH_LEVEL_PERIOD 200
#define TASK_THRESHOLD 400
#define START_UTIL 100

/* From low_level_sched */
extern list<Task*> run_queue;

void update_task_io(Task* t);
void update_task_cpu(Task* t);
Task* get_task();
void swap_queues();
void fill();
void add_to_runqueue(Task* t);
void del_from_runqueue(Task* t);
void high_level_tick();
void high_level_init();

void sc_add_to_runqueue(Task* t);
void sc_del_from_runqueue(Task* t);

#endif /* __TWO_LEVEL_H__ */
```

sched-local_queue.cc

```
#include <list>
#include <vector>
#include <math.h>
#include <algorithm>
#include "simulator_interface.h"
#include "sched_interface.h"
#include "sched.h"
#include "globals.h"

using namespace std;

/* A list of running programs on the cpu's */
Task ** current;
Task ** idle_tasks;
/* Per CPU data */
struct schedule_data *cpu_schedule_data;

/* The run queues */
list<Task*> *run_queues;
/* Number of running processes */
int nr_running = 0;

#define PROC_CHANGE_PENALTY 0
#define UPPER_THRESHOLD 100+100/smp_num_cpus
#define LOWER_THRESHOLD (100-100/smp_num_cpus)

#define unlikely(x) x
/* Actually this is the original define */
#define prepare_to_switch() do { } while(0)

/* Assembler macro - This sets up the new task.
 * This is not needed in this simulator */
#define switch_to(prev,next,last) do { } while(0)

/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x) ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x) ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x) (x)
#elif HZ < 1600
#define TICK_SCALE(x) ((x) << 1)
#else
#define TICK_SCALE(x) ((x) << 2)
#endif
```

```
#define NICE_TO_TICKS(nice) (TICK_SCALE(20-(nice))+1)

/* Move a task from one run_queue to another.
  Make sure that the task is not in the running state */
void move_task(int from_rq, int to_rq)
{
   /* Select a task to be moved */
   list<Task*>::iterator iter;
   Task *p = NULL;
   for (iter = run_queues[from_rq].begin(); iter != run_queues[from_rq].end()
        ; iter++)
   {
      if (can_schedule((*iter), to_rq))
      {
         p = *iter;
         break;
      }
   }
   if (p == NULL)
      return;


   /* Move the selected task */
   run_queues[from_rq].remove(p);
   p->run_queue = to_rq;
   run_queues[to_rq].push_back(p);
}


/* This function is used to load balance the run_queues */
/* The function only balance the current run_queue against others. */

void balance()
{

   /* printf("### Balance\n"); */
   int this_rq = smp_processor_id(), rq;
   int imbalance = 0;
   int i;

   int local = run_queues[this_rq].size()*100;
   int avg = nr_running*100/smp_num_cpus;

   /* Test is an imbalance may exist */
   /* if (abs(avg-local) <= 1)
      return;
   */
   /* imbalance = (local - avg) * 100 / (local + avg); */
   imbalance = local-avg;

   if (abs(imbalance) < UPPER_THRESHOLD)
      return;

   /* Need to balance */
   /* Do this while above the LOWER_THRESHOLD */
   while (abs(imbalance) > LOWER_THRESHOLD)
   {
      /* Need to steal processes from other RQ's? */
      if (imbalance < 0)
      {
         rq = this_rq;
         for (i=0;i<smp_num_cpus;i++)
         {
            if (run_queues[i].size() > run_queues[rq].size())
               rq = i;
         }

         /* rq points to the longest queue */

         /* Steal the task in the back of the queue */
         move_task(rq, this_rq);

      }

      if (imbalance > 0)
      {
         rq = this_rq;
         for (i=0;i<smp_num_cpus;i++)
         {
            /* printf("### Queue: %d, Length: %d\n",
              i,run_queues[i].size()); */
            if (run_queues[i].size() < run_queues[rq].size())
               rq = i;
         }

         /* rq points to the shortest queue */

         /* Push a task to the shortest rq */
         move_task(this_rq, rq);
      }

      /* Recalculate imbalance */
      local = run_queues[this_rq].size()*100;

      /* if (abs(avg-local) <= 1)
         return;
         imbalance = (local - avg) * 100 / (local + avg); */
      imbalance = local-avg;
   }
}


int preemption_goodness(Task *prev, Task *p, int cpu)
{
   return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->
        active_mm);
}

void reschedule_idle(Task *p)
{
      int this_cpu = smp_processor_id();
      Task *tsk, *target_tsk;
      int cpu, best_cpu, i, max_prio;
      cycles_t oldest_idle;
      /*
       * shortcut if the woken up task's last CPU is
       * idle now.
       */
      best_cpu = p->processor;
      if (can_schedule(p, best_cpu)) {
            tsk = idle_task(best_cpu);
            if (cpu_curr(best_cpu) == tsk) {
                  int need_resched;
send_now_idle:
                  /*
                   * If need_resched == -1 then we can skip sending
                   * the IPI altogether, tsk->need_resched is
                   * actively watched by the idle thread.
                   */
                  need_resched = tsk->need_resched;
                  tsk->need_resched = 1;
                  if ((best_cpu != this_cpu) && !need_resched)
                        smp_send_reschedule(best_cpu);
                  return;
            }
      }

      /*
       * We know that the preferred CPU has a cache-affine current
       * process, lets try to find a new idle CPU for the woken-up
       * process. Select the least recently active idle CPU. (that
       * one will have the least active cache context.) Also find
       * the executing process which has the least priority.
       */
      oldest_idle = (cycles_t) -1;
      target_tsk = NULL;
      max_prio = 0;

      for (i = 0; i < smp_num_cpus; i++) {
            cpu = cpu_logical_map(i);
            if (!can_schedule(p, cpu))
                  continue;
            tsk = cpu_curr(cpu);
            /*
             * We use the first available idle CPU. This creates
             * a priority list between idle CPUs, but this is not
             * a problem.
             */
            if (tsk == idle_task(cpu)) {
/* Added in 2.4.17
#if defined(__i386__) && defined(CONFIG_SMP)
                 *//*
                   * Check if two siblings are idle in the same
                   * physical package. Use them if found.
                   *//*
                   if (smp_num_siblings == 2) {
                        if (cpu_curr(cpu_sibling_map[cpu]) ==
                           idle_task(cpu_sibling_map[cpu])) {
                              oldest_idle = last_schedule(cpu);
                              target_tsk = tsk;
                              break;
                        }
                  }
#endif
*/
                  if (last_schedule(cpu) < oldest_idle) {
                        oldest_idle = last_schedule(cpu);
                        target_tsk = tsk;
                  }
            } else {
                  if (oldest_idle == -1ULL) {
                        int prio = preemption_goodness(tsk, p, cpu);

                        if (prio > max_prio) {
                              max_prio = prio;
                              target_tsk = tsk;
                        }
                  }
            }
      }
      tsk = target_tsk;
      if (tsk) {
            if (oldest_idle != -1ULL) {
                  best_cpu = tsk->processor;
                  goto send_now_idle;
            }
            tsk->need_resched = 1;
            if (tsk->processor != this_cpu)
                  smp_send_reschedule(tsk->processor);
      }
      return;
}

/*
 * Wake up a process. Put it on the run-queue if it’s not
 * already there. The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you’re allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
static inline int try_to_wake_up(Task *p, int synchronous)
{
   //unsigned long flags;
      int success = 0;

      /*
       * We want the common case fall through straight, thus the goto.
       */
      /* spin_lock_irqsave(&runqueue_lock, flags); */
      p->state = TASK_RUNNING;
      if (task_on_runqueue(p))
            goto out;
      add_to_runqueue(p);
      if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
            reschedule_idle(p);
      success = 1;
out:
      /* spin_unlock_irqrestore(&runqueue_lock, flags); */
      return success;
}

int wake_up_process(Task * p)
{
      return try_to_wake_up(p, 0);
}


/* Calculate the goodness of the process */
int goodness(Task* p, int this_cpu, int this_mm)
{
   /* This is soly based on quantum and nice-level */
   int weight = -1;
   if (p->policy & SCHED_YIELD)
      return weight;

   if (p->policy == SCHED_OTHER) {
      /*
       * Give the process a first-approximation goodness value
       * according to the number of clock-ticks it has left.
       *
       * Don't do any other calculations if the time slice is
       * over..
       */
      weight = p->counter;
      if (!weight)
         return weight;

      if (p->processor == this_cpu)
         weight += PROC_CHANGE_PENALTY;

      /* .. and a slight advantage to the current MM */


      if (p->mm == this_mm || !p->mm)
         weight += 1;


      weight += 20 - p->nice;

      return weight;
   }

   /*
    * Realtime process, select the first one on the
    * runqueue (taking priorities within processes
    * into account).
    */
   weight = 1000 + p->rt_priority;
   return weight;
}

void __schedule_tail(Task* prev)
{
   task_release_cpu(prev);
}
```

```
void schedule_tail(Task* prev)
{
   __schedule_tail(prev);
}



void schedule()
{
   int this_cpu = smp_processor_id();
   list<Task*> run_queue = run_queues[this_cpu];
   struct schedule_data * sched_data;
   Task *prev, *next;
   int c;

   /*printf("### Sched - Queue %d, Size. %d Total: %d\n",
     this_cpu, run_queue.size(), nr_running);*/

need_resched_back:

   prev = get_current();
   next = idle_task(this_cpu);


   /* This is not modeled in the simulator
   if (unlikely(in_interrupt())) {
      printk("Scheduling in interrupt\n");
      BUG();
   }
   */

   /* release_kernel_lock(prev, this_cpu); */

   /*
    * 'sched_data' is protected by the fact that we can run
    * only one process per CPU.
    */
   sched_data = &cpu_schedule_data[this_cpu];

   /* spin_lock_irq(&runqueue_lock); */

   /* move an exhausted RR process to be last.. */
   if (unlikely(prev->policy == SCHED_RR))
      if (!prev->counter) {
         prev->counter = NICE_TO_TICKS(prev->nice);
         move_last_runqueue(prev);
      }
   /*
   switch (prev->state) {
      case TASK_INTERRUPTIBLE:
         if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
         }
      default:
          del_from_runqueue(prev);
      case TASK_RUNNING:;
   }
   */
   /* Since tasks themself set the state running, no need to check for
    * a pending signal. All in all this boild down to the following: */
   if (prev->state != TASK_RUNNING)
      del_from_runqueue(prev);

   prev->need_resched = 0;

   /*
    * this is the scheduler proper:
    */

repeat_schedule:
   /*
    * Default process to select..
    */
   c = -1000;
   list<Task*>::iterator iter;

   for (iter = run_queue.begin(); iter != run_queue.end(); iter++)
   {
      if (can_schedule((*iter), this_cpu)) {
         int weight = goodness((*iter), this_cpu, prev->active_mm);
         if (weight > c)
            c = weight, next = (*iter);
      }
   }

   /* Do we need to re-calculate counters? */
   if (!c) {
      /* Recalculate all counter on all tasks. */
      list<Task*>::iterator p;
      /* spin_unlock_irq(&runqueue_lock);
       * read_lock(&tasklist_lock);
       */
      for_all_tasks(p)
         if ((*p)->run_queue == this_cpu)
            (*p)->counter = ((*p)->counter >> 1) +
               NICE_TO_TICKS((*p)->nice);

      /*
       * read_unlock(&tasklist_lock);
       * spin_lock_irq(&runqueue_lock);
       */
      goto repeat_schedule;
   }

   /*
    * from this point on nothing can prevent us from
    * switching to the next task, save this fact in
    * sched_data.
    */

   /* Add the task to the running list */
   /* Mark which cpu the task is running on */
   /* Sched data is locale for the cpu itself */

   //sched_data->curr = next;
   schedule_tail(prev);
   task_set_cpu(next, this_cpu);
   /* spin_unlock_irq(&runqueue_lock); */

   if (unlikely(prev == next)) {
      /* We won't go through the normal tail, so do this by hand */
      prev->policy &= ~SCHED_YIELD;
      goto same_process;
   }

/*#ifdef CONFIG_SMP*/
   /*
    * maintain the per-process 'last schedule' value.
    * (this has to be recalculated even if we reschedule to
    * the same process) Currently this is only used on SMP,
    * and it's approximate, so we do not have to maintain
    * it while holding the runqueue spinlock.
    */

//   sched_data->last_schedule = get_cycles();

   /*
    * We drop the scheduler lock early (it's a global spinlock),
    * thus we have to lock the previous process from getting
    * rescheduled during switch_to().
    */

/*#endif*/ /* CONFIG_SMP */

   /* kstat.context_swtch++; */
   /*
    * there are 3 processes which are affected by a context switch:
    *
    * prev == .... ==> (last => next)
    *
    * It's the 'much more previous' 'prev' that is on next's stack,
    * but prev is set to (the just run) 'last' process by switch_to().
    * This might sound slightly confusing but makes tons of sense.
    */
   prepare_to_switch();
   /*
   {
      struct mm_struct *mm = next->mm;
      struct mm_struct *oldmm = prev->active_mm;
      if (!mm) {
         if (next->active_mm) BUG();
         next->active_mm = oldmm;
         atomic_inc(&oldmm->mm_count);
         enter_lazy_tlb(oldmm, next, this_cpu);
      } else {
         if (next->active_mm != mm) BUG();
         switch_mm(oldmm, mm, next, this_cpu);
      }

      if (!prev->mm) {
         prev->active_mm = NULL;
         mmdrop(oldmm);
      }
   }
   */

   /*
    * This just switches the register state and the
    * stack.
    */
   switch_to(prev, next, prev);
   __schedule_tail(prev);

 same_process:

   /* reacquire_kernel_lock(current); */
   if (get_current()->need_resched)
      goto need_resched_back;
   return;
}


/* Called for each timer interrupt on each processor. */
void do_timer()
{
   int cpu=smp_processor_id();
   /* A tick actually shouldent be cpu specific. */
   /* For now we work out the single cpu problem. */
   Task *p = get_current();
   if (p == idle_task(cpu))
   {
      p->need_resched = 1;
   }

   if (--p->counter <= 0) {
      p->counter = 0;
      p->need_resched = 1;
   }
}


/* Return the idle task for the spec. cpu */
Task* idle_task(int cpu)
{
   if (!idle_tasks[cpu])
      printf("## No idle task for PE:%d\n", cpu);
   return idle_tasks[cpu];
}

/* Return the current task on this cpu */
Task* get_current()
{
   return current[smp_processor_id()];
}


/* Remove a task from the run_queue */
void del_from_runqueue(Task *task)
```

```
{
   /* printf("### Delete from rq: %d\n", task->pid); */
   int i = run_queues[task->run_queue].size();
   run_queues[task->run_queue].remove(task);
   nr_running-= i-run_queues[task->run_queue].size();

   /* Balance if nessesary */
   balance();
}

/* Set the mask for the task */
void task_set_cpu(Task *task, int cpu)
{
   task->set_state(TASK_RUNNING);
   current[cpu] = task;
   task->processor = cpu;
   task->cpus_runnable = 1UL << cpu;
}

/* Change to mask to have no cpu */
void task_release_cpu(Task *task)
{
   task->cpus_runnable = ˜0UL;
}

/* Examine if the task is running on a cpu */
int task_has_cpu(Task *task)
{
   return (task->cpus_runnable != ˜0UL);
}

/* Move a task to the last of the runqueue */
void move_last_runqueue(Task *task)
{
   int i = run_queues[task->run_queue].size();
   run_queues[task->run_queue].remove(task);
   run_queues[task->run_queue].push_back(task);
   nr_running += run_queues[task->run_queue].size()-i;
}

/* Move a task to the front of a runqueue */
void move_first_runqueue(Task *task)
{
   int i = run_queues[task->run_queue].size();
   run_queues[task->run_queue].remove(task);
   run_queues[task->run_queue].push_front(task);
   nr_running += run_queues[task->run_queue].size()-i;
}

/* Add a task to the runqueue */
void add_to_runqueue(Task *task)
{
   /* Add to the previous runqueue.
     If this is negative, then add to the runqueue
     for this processor */
   if (task->run_queue < 0)
      task->run_queue = smp_processor_id();
   /* printf("### Adding task %d to rq:%d\n", task->pid, task->run_queue); */

   run_queues[task->run_queue].push_front(task);
   nr_running++;

   /* Balance if nessesary */
   balance();
}

/* Examine if a task can run on the spec. CPU */
int can_schedule(Task *p, int cpu)
{
   return (p->cpus_runnable & p->cpus_allowed & (1 << cpu));
}

Task* cpu_curr(int cpu)
{
   return current[cpu];
}

/*
 * On x86 all CPUs are mapped 1:1 to the APIC space.
 * This simplifies scheduling and IPI sending and
 * compresses data structures.
 */
int cpu_logical_map(int cpu)
{
   return cpu;
}

int cpu_number_map(int cpu)
{
   return cpu;
}

/* Check if the task is on any runqueue */
int task_on_runqueue(Task *p)
{
   /* Go through the runqueue */
   if (p->run_queue < 0) return 0;

   /* In linux, the thread would know itself through p->run_list.next */
   return (find(run_queues[p->run_queue].begin(),
             run_queues[p->run_queue].end(), p) != run_queues[p->run_queue].
                 end());
}

/* Retrive number of ticks since last schedule in this cpu */
/* This is updated for every tick */
cycles_t last_schedule(int cpu)
{
   return cpu_schedule_data[cpu].last_schedule;
}

/* Get System Ticks */
cycles_t get_cycles()
{
   return jiffies;
```

```
}

void sched_init(Task *idle_task)
{
   int cpu = smp_processor_id();
   if (!cpu)
   {
      current = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      idle_tasks = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      cpu_schedule_data = (struct schedule_data*)
         malloc(sizeof(struct schedule_data)*smp_num_cpus);
      run_queues = new list<Task*>[smp_num_cpus];

   }

   idle_tasks[cpu] = idle_task;
   /* set the idle task as the active task the CPU */
   task_set_cpu(idle_task, cpu);
}
```

sched-multi_queue.cc

```
#include <list>
#include <vector>
#include <math.h>
#include <algorithm>
#include "sched.h"
#include "simulator_interface.h"
#include "sched_interface.h"
#include "globals.h"

using namespace std;

/* A list of running programs on the cpu's */
Task ** current;
Task ** idle_tasks;
/* Per CPU data */
struct schedule_data *cpu_schedule_data;

#define PROC_CHANGE_PENALTY 15

/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x) ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x) ((x) >> 1)
```

```
#elif HZ < 800
#define TICK_SCALE(x) (x)
#elif HZ < 1600
#define TICK_SCALE(x) ((x) << 1)
#else
#define TICK_SCALE(x) ((x) << 2)
#endif

#define NICE_TO_TICKS(nice) (TICK_SCALE(20-(nice))+1)

struct queue {
   int counter;    /* The total quantum left for the queue */
   int number;
   list<Task*> active; /* List of active threads in this cycle */
};

/* Number of queues in the system */
#define NR_QUEUES 4

/* Queue 0, CPU-bound
  Queue 3, Interactive.
  Fairness are still in place.
  Interactive tasks are run for shorter periods of time, more
  frequently */

/* Allocate the queues */
static struct queue run_queues[NR_QUEUES];


#define unlikely(x) x
/* Actually this is the original define */
#define prepare_to_switch() do { } while(0)

/* Assembler macro - This sets up the new task.
 * This is not needed in this simulator */
#define switch_to(prev,next,last) do { } while(0)
void scheduling_functions_begin() {}

void reschedule_idle(Task *p)
{
      int this_cpu = smp_processor_id();
      Task *tsk, *target_tsk;
      int cpu, best_cpu, i, max_prio;
      cycles_t oldest_idle;

      /*
       * shortcut if the woken up task's last CPU is
       * idle now.
       */
      best_cpu = p->processor;
      if (can_schedule(p, best_cpu)) {
            tsk = idle_task(best_cpu);
            if (cpu_curr(best_cpu) == tsk) {
                  int need_resched;
send_now_idle:
                  /*
                   * If need_resched == -1 then we can skip sending
                   * the IPI altogether, tsk->need_resched is
                   * actively watched by the idle thread.
                   */
                  need_resched = tsk->need_resched;
                  tsk->need_resched = 1;
                  if ((best_cpu != this_cpu) && !need_resched)
                        smp_send_reschedule(best_cpu);
                  return;
            }
      }

      /*
       * We know that the preferred CPU has a cache-affine current
       * process, lets try to find a new idle CPU for the woken-up
       * process. Select the least recently active idle CPU. (that
       * one will have the least active cache context.) Also find
       * the executing process which has the least priority.
       */
      oldest_idle = (cycles_t) -1;
      target_tsk = NULL;
      max_prio = 0;

      for (i = 0; i < smp_num_cpus; i++) {
            cpu = cpu_logical_map(i);
            if (!can_schedule(p, cpu))
                  continue;
            tsk = cpu_curr(cpu);
            /*
             * We use the first available idle CPU. This creates
             * a priority list between idle CPUs, but this is not
             * a problem.
             */
            if (tsk == idle_task(cpu)) {
/* Added in 2.4.17
#if defined(__i386__) && defined(CONFIG_SMP)
                 *//*
                   * Check if two siblings are idle in the same
                   * physical package. Use them if found.
                   *//*
                   if (smp_num_siblings == 2) {
                        if (cpu_curr(cpu_sibling_map[cpu]) ==
                           idle_task(cpu_sibling_map[cpu])) {
                              oldest_idle = last_schedule(cpu);
                              target_tsk = tsk;
                              break;
                        }

                  }
#endif
*/
                  if (last_schedule(cpu) < oldest_idle) {
                        oldest_idle = last_schedule(cpu);
                        target_tsk = tsk;
                  }
            }
      }
      tsk = target_tsk;
      if (tsk) {
            if (oldest_idle != -1ULL) {
                  best_cpu = tsk->processor;
                  goto send_now_idle;
            }
            tsk->need_resched = 1;
            if (tsk->processor != this_cpu)
                  smp_send_reschedule(tsk->processor);
      }
      return;
}

/*
 * Wake up a process. Put it on the run-queue if it's not
 * already there. The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you're allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
static inline int try_to_wake_up(Task *p, int synchronous)
{
   //unsigned long flags;
      int success = 0;

      /*
       * We want the common case fall through straight, thus the goto.
       */
      /* spin_lock_irqsave(&runqueue_lock, flags); */
      p->state = TASK_RUNNING;
      if (task_on_runqueue(p))
            goto out;
      add_to_runqueue(p);
      if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
            reschedule_idle(p);
      success = 1;
out:
      /* spin_unlock_irqrestore(&runqueue_lock, flags); */
      return success;
}

int wake_up_process(Task * p)
{
      return try_to_wake_up(p, 0);
}


void __schedule_tail(Task* prev)
{
   task_release_cpu(prev);
}



void schedule_tail(Task* prev)
{
   __schedule_tail(prev);
}

/* We dont know anything about SCHED_RR or SCHED_FIFO.
```

```
  We could just implement these in a single queue, that
  must be empty before scheduling SCHED_OTHER */
void schedule()
{
   int this_cpu = smp_processor_id();
   list<Task*>::iterator p;

   struct schedule_data * sched_data;
   Task *prev, *next;
   int c;

   struct queue* active;
   int weight, new_weight;

   prev = get_current();
   next = idle_task(this_cpu);

   printf("### Schedule on PE: %d\n", this_cpu);

   /* Write the state of all tasks on one line */
   printf("### States: ");
   for_all_tasks(p) {
      printf("(%d,%d) ", (*p)->run_queue, (*p)->state);
   }
   printf("\n");

   if (prev != idle_task(this_cpu)) {
      //task_release_cpu(prev);
      if (prev->state != TASK_RUNNING)
         del_from_runqueue(prev);
      else {
         if (prev->counter==0) {
            /* Has the thread used all of its timeslice? */
            if ((!prev->moved) && (prev->run_queue > 0)) {
               printf("### Less interactive: %d\n", prev->pid);
               del_from_runqueue(prev);
               prev->moved++;
               add_to_runqueue_tail(prev);
            }
         }
         /*
         if (prev->quantum > 0)
         {
            move_first_runqueue(prev);
         }
         */
      }
   }

need_resched_back:

   weight=-100000;
   active=NULL;
   int t=-100000;
   /* Find the best task to run, if two queues have the same weight,
    * Select the late */
   for (c=0;c<NR_QUEUES;c++)
   {
      printf("### Queue %d: (%d,%d)\n",
           c,run_queues[c].number,run_queues[c].counter);
      if (run_queues[c].number && can_schedule(*run_queues[c].active.begin(),
           this_cpu)) {
         new_weight=run_queues[c].counter/run_queues[c].number;
         if (new_weight >= weight)
         {
            weight = new_weight;
            active = &run_queues[c];
            t=c;
         }
      }
   }

   /* Recalculate counters?
    * Counter are recalculated, if there was a runqueue, but counter <= 0 */
   if (active && active->counter <= 0)
   {

      printf("### Recalculating\n");
      for (c=1;c<NR_QUEUES;c++)
         run_queues[c].counter=0;

      for_all_tasks(p) {
         (*p)->counter = ((*p)->counter >> 2) + NICE_TO_TICKS((*p)->nice);
         if ((*p)->state == TASK_RUNNING)
         {
            run_queues[(*p)->run_queue].counter += (*p)->counter;
            (*p)->moved=0;
         }

      }
      goto need_resched_back;
   }


   /* Ok we got the queue, from which we should schedule */
   /* Schedule in a round robin fashion */

   /* A process assigned to another PE cannot be scheduled */
   if (active) {
      next = *active->active.begin();
      move_last_runqueue(next);
      next->quantum=(NICE_TO_TICKS(next->nice)>>next->run_queue);
   } else
      printf("### Choosing idle process\n");

   /* All process know their quantum.
     A extra quantum is added. A thread is only run for this quantum.
     The temporary quantum depends on the queue the thread is in. */

   /* release_kernel_lock(prev, this_cpu); */

   sched_data = &cpu_schedule_data[this_cpu];

   /* spin_lock_irq(&runqueue_lock); */

   prev->need_resched = 0;
   /*
    * this is the scheduler proper:
    */

   schedule_tail(prev);
   task_set_cpu(next, this_cpu);
   /* spin_unlock_irq(&runqueue_lock); */

   prepare_to_switch();

   /*
    * This just switches the register state and the
    * stack.
    */
   printf("### Prev: %d, Next: %d\n", prev->pid, next->pid);
   switch_to(prev, next, prev);

}

/* Called for each timer interrupt on each processor. */
void do_timer()
{
   /* A tick actually shouldent be cpu specific. */
   /* For now we work out the single cpu problem. */
   int cpu = smp_processor_id();
   Task *p = get_current();
   if (p == idle_task(cpu))
   {
      p->need_resched = 1;
   }

   run_queues[p->run_queue].counter--;
   --p->counter;
   --p->quantum;
   if (p->quantum <= 0) {
      p->need_resched = 1;
      p->quantum = 0;
   }

   if (p->counter <= 0) {
      p->need_resched = 1;
      p->counter = 0;
      p->quantum = 0;
   }
}

void scheduling_functions_end() {}

/* Return the idle task for the spec. cpu */
Task* idle_task(int cpu)
{
   return idle_tasks[cpu];
}

/* Return the current task on this cpu */
Task* get_current()
{
   return current[smp_processor_id()];
}


/* Remove a task from the run_queue */
void del_from_runqueue(Task *p)
{
   if (!task_on_runqueue(p)) return;
   printf("### Del from runqueue: %d,%d\n", p->pid,p->run_queue);
   if (!run_queues[p->run_queue].number)
   {
      printf("### Bug in del. No more processes on queue\n");
      //run_queues[p->run_queue].number++;
   }

   run_queues[p->run_queue].active.remove(p);
   run_queues[p->run_queue].number--;
   run_queues[p->run_queue].counter-=p->counter;

   /* Find out, if the task should be moved to another queue */
   /* If the task has used less than half of the temporary quanta */
   /* Do not move, if it has been moved before within this scheduling cycle
         */
   /* thread suspended. */
   if ((!p->moved) &&
      (p->run_queue < NR_QUEUES-1) &&
      (p->quantum < (NICE_TO_TICKS(p->nice) >> (p->run_queue+2)))) {
      printf("### More interactive: %d\n", p->pid);
      p->run_queue++;
   }
}

/* Set the mask for the task */
void task_set_cpu(Task *task, int cpu)
{
   task->set_state(TASK_RUNNING);
   current[cpu] = task;
   task->processor = cpu;
   task->cpus_runnable = 1UL << cpu;
}

/* Change to mask to have no cpu */
void task_release_cpu(Task *task)
{
   task->cpus_runnable = ~0UL;
}

/* Examine if the task is running on a cpu */
int task_has_cpu(Task *task)
{
   return (task->cpus_runnable != ~0UL);
}

/* Move a task to the last of the runqueue */
void move_last_runqueue(Task *p)
{
   run_queues[p->run_queue].active.remove(p);
   run_queues[p->run_queue].active.push_back(p);
```

```
}

/* Move a task to the front of a runqueue */
void move_first_runqueue(Task *p)
{
   run_queues[p->run_queue].active.remove(p);
   run_queues[p->run_queue].active.push_front(p);
}

/* Add a task to the runqueue */
void add_to_runqueue(Task *p)
{
   if (task_on_runqueue(p)) return;
   printf("### Add to runqueue: %d\n", p->pid);

   run_queues[p->run_queue].active.push_front(p);
   /* Add the rest of the quantum to the run_queue */
   run_queues[p->run_queue].counter += p->counter;
   run_queues[p->run_queue].number++;
}
void add_to_runqueue_tail(Task *p)
{
   if (task_on_runqueue(p)) return;
   printf("### Add to runqueue: %d\n", p->pid);

   run_queues[p->run_queue].active.push_back(p);
   /* Add the rest of the quantum to the run_queue */
   run_queues[p->run_queue].counter += p->counter;
   run_queues[p->run_queue].number++;
}

/* Examine if a task can run on the spec. CPU */
int can_schedule(Task *p, int cpu)
{
   return (p->cpus_runnable & p->cpus_allowed & (1 << cpu));
}

Task* cpu_curr(int cpu)
{
   return current[cpu];
}

/*
 * On x86 all CPUs are mapped 1:1 to the APIC space.
 * This simplifies scheduling and IPI sending and
 * compresses data structures.
 */
int cpu_logical_map(int cpu)
{
   return cpu;
}

int cpu_number_map(int cpu)
{
   return cpu;
}

/* Check if the task is on any runqueue */
```

```
int task_on_runqueue(Task *p)
{
   /* Go through the runqueue */
   /* In linux, the thread would know itself through p->run_list.next */
   if (p->run_queue == -1)
      p->run_queue = 1;

   int rq = p->run_queue;
   return ((find(run_queues[rq].active.begin(), run_queues[rq].active.end(),
        p) !=
         run_queues[rq].active.end()) /*||
      (find(run_queues[rq].expired.begin(), run_queues[rq].expired.end(), p)
           !=
      run_queues[rq].expired.end())*/);
}

/* Retrive number of ticks since last schedule in this cpu */
/* This is updated for every tick */
cycles_t last_schedule(int cpu)
{
   return cpu_schedule_data[cpu].last_schedule;
}

/* Get System Ticks */
cycles_t get_cycles()
{
   return jiffies;
}

void sched_init(Task *idle_task)
{
   int cpu = smp_processor_id();
   if (!cpu)
   {
      current = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      idle_tasks = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      cpu_schedule_data = (struct schedule_data*)
         malloc(sizeof(struct schedule_data)*smp_num_cpus);
   }

   idle_tasks[cpu] = idle_task;
   /* set the idle task as the active task the CPU */
   task_set_cpu(idle_task, cpu);
}
```

sched-round_robin.cc

```
#include <list>
#include <vector>
#include <algorithm>
#include <math.h>
#include "simulator_interface.h"
#include "sched_interface.h"
#include "sched.h"
#include "globals.h"
```

```
using namespace std;

/* A list of running programs on the cpu’s */
Task ** current;
Task ** idle_tasks;
/* Per CPU data */
struct schedule_data *cpu_schedule_data;

#define unlikely(x) x
/* Actually this is the original define */
#define prepare_to_switch() do { } while(0)

/* Assembler macro - This sets up the new task.
 * This is not needed in this simulator */
#define switch_to(prev,next,last) do { } while(0)

/* The run queue */
list<Task*> run_queue;

#define HZ 100
#define PROC_CHANGE_PENALTY 15


/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x) ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x) ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x) (x)
#elif HZ < 1600
#define TICK_SCALE(x) ((x) << 1)
#else
#define TICK_SCALE(x) ((x) << 2)
#endif

#define NICE_TO_TICKS(nice) (TICK_SCALE(20-(nice))+1)


int preemption_goodness(Task *prev, Task *p, int cpu)
{
   return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->
        active_mm);
}



void reschedule_idle(Task *p)
```

```
{
      int this_cpu = smp_processor_id();
      Task *tsk, *target_tsk;
      int cpu, best_cpu, i, max_prio;
      cycles_t oldest_idle;

      /*
       * shortcut if the woken up task's last CPU is
       * idle now.
       */
      best_cpu = p->processor;
      if (can_schedule(p, best_cpu)) {
            tsk = idle_task(best_cpu);
            if (cpu_curr(best_cpu) == tsk) {
                  int need_resched;
send_now_idle:
                  /*
                   * If need_resched == -1 then we can skip sending
                   * the IPI altogether, tsk->need_resched is
                   * actively watched by the idle thread.
                   */
                  need_resched = tsk->need_resched;
                  tsk->need_resched = 1;
                  if ((best_cpu != this_cpu) && !need_resched)
                        smp_send_reschedule(best_cpu);
                  return;
            }
      }

      /*
       * We know that the preferred CPU has a cache-affine current
       * process, lets try to find a new idle CPU for the woken-up
       * process. Select the least recently active idle CPU. (that
       * one will have the least active cache context.) Also find
       * the executing process which has the least priority.
       */
      oldest_idle = (cycles_t) -1;
      target_tsk = NULL;
      max_prio = 0;

      for (i = 0; i < smp_num_cpus; i++) {
            cpu = cpu_logical_map(i);
            if (!can_schedule(p, cpu))
                  continue;
            tsk = cpu_curr(cpu);
            /*
             * We use the first available idle CPU. This creates
             * a priority list between idle CPUs, but this is not
             * a problem.
             */
            if (tsk == idle_task(cpu)) {
/* Added in 2.4.17
#if defined(__i386__) && defined(CONFIG_SMP)
                 *//*
                   * Check if two siblings are idle in the same
                   * physical package. Use them if found.
                   *//*
                   if (smp_num_siblings == 2) {
                        if (cpu_curr(cpu_sibling_map[cpu]) ==
                           idle_task(cpu_sibling_map[cpu])) {
                              oldest_idle = last_schedule(cpu);
                              target_tsk = tsk;
                              break;
                        }

                  }
#endif
*/
                  if (last_schedule(cpu) < oldest_idle) {
                        oldest_idle = last_schedule(cpu);
                        target_tsk = tsk;
                  }
            } /*else {
                  if (oldest_idle == -1ULL) {
                        int prio = preemption_goodness(tsk, p, cpu);

                        if (prio > max_prio) {
                              max_prio = prio;
                              target_tsk = tsk;
                        }
                  }
                  }*/
      }
      tsk = target_tsk;
      if (tsk) {
            if (oldest_idle != -1ULL) {
                  best_cpu = tsk->processor;
                  goto send_now_idle;
            }
            tsk->need_resched = 1;
            if (tsk->processor != this_cpu)
                  smp_send_reschedule(tsk->processor);
      }
      return;
}

/*
 * Wake up a process. Put it on the run-queue if it's not
 * already there. The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you're allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
static inline int try_to_wake_up(Task *p, int synchronous)
{
   //unsigned long flags;
      int success = 0;

      /*
       * We want the common case fall through straight, thus the goto.
       */
      /* spin_lock_irqsave(&runqueue_lock, flags); */
      p->state = TASK_RUNNING;
      if (task_on_runqueue(p))
            goto out;
      add_to_runqueue(p);
      if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
            reschedule_idle(p);
      success = 1;
out:
      /* spin_unlock_irqrestore(&runqueue_lock, flags); */
      return success;
}

int wake_up_process(Task * p)
{
      return try_to_wake_up(p, 0);
}


/* Calculate the goodness of the process */
int goodness(Task* p, int this_cpu, int this_mm)
{
   /* This is soly based on quantum and nice-level */
   int weight = -1;
   if (p->policy & SCHED_YIELD)
      return weight;

   if (p->policy == SCHED_OTHER) {
      /*
       * Give the process a first-approximation goodness value
       * according to the number of clock-ticks it has left.
       *
       * Don't do any other calculations if the time slice is
       * over..
       */
      weight = p->counter;
      if (!weight)
         return weight;

      if (p->processor == this_cpu)
         weight += PROC_CHANGE_PENALTY;

      /* .. and a slight advantage to the current MM */


      if (p->mm == this_mm || !p->mm)
         weight += 1;


      weight += 20 - p->nice;

      return weight;
   }

   /*
    * Realtime process, select the first one on the
    * runqueue (taking priorities within processes
    * into account).
    */
   weight = 1000 + p->rt_priority;
   return weight;
}
```

```
void __schedule_tail(Task* prev)
{
   task_release_cpu(prev);
}



void schedule_tail(Task* prev)
{
   __schedule_tail(prev);
}



void schedule()
{
   int this_cpu = smp_processor_id();
   struct schedule_data * sched_data;
   Task *prev, *next;
   int c;

need_resched_back:

   prev = get_current();
   next = idle_task(this_cpu);


   /* This is not modeled in the simulator
   if (unlikely(in_interrupt())) {
      printk("Scheduling in interrupt\n");
      BUG();
   }
   */

   /* release_kernel_lock(prev, this_cpu); */

   /*
    * 'sched_data' is protected by the fact that we can run
    * only one process per CPU.
    */
   sched_data = &cpu_schedule_data[this_cpu];

   /* spin_lock_irq(&runqueue_lock); */

   /* move an exhausted RR process to be last.. */

   /*
     if (unlikely(prev->policy == SCHED_RR))
     if (!prev->counter) {
     prev->counter = NICE_TO_TICKS(prev->nice);
     move_last_runqueue(prev);
     }*/
   /*
   switch (prev->state) {
      case TASK_INTERRUPTIBLE:
         if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
         }
      default:
          del_from_runqueue(prev);
      case TASK_RUNNING:;
   }
   */
   /* Since tasks themself set the state running, no need to check for
    * a pending signal. All in all this boild down to the following: */
   if (prev->state != TASK_RUNNING)
      del_from_runqueue(prev);
   else
      move_last_runqueue(prev);


   prev->need_resched = 0;

   /*
    * this is the scheduler proper:
    */

// repeat_schedule:

   /*
    * Default process to select..
    */
   c = -1000;
   list<Task*>::iterator iter;
   for (iter = run_queue.begin(); iter != run_queue.end(); iter++)
   {
      if (can_schedule((*iter), this_cpu)) {
         next=(*iter);
         if (next->pid > 0)
            break;
      }
   }
   next->counter=NICE_TO_TICKS(next->nice);

   /*
    * from this point on nothing can prevent us from
    * switching to the next task, save this fact in
    * sched_data.
    */

   /* Add the task to the running list */
   /* Mark which cpu the task is running on */

   /* Sched data is locale for the cpu itself */

   //sched_data->curr = next;
   schedule_tail(prev);
   task_set_cpu(next, this_cpu);
   /* spin_unlock_irq(&runqueue_lock); */

   if (unlikely(prev == next)) {
      /* We won't go through the normal tail, so do this by hand */
      prev->policy &= ~SCHED_YIELD;
      goto same_process;
   }

/*#ifdef CONFIG_SMP*/
   /*
    * maintain the per-process 'last schedule' value.
    * (this has to be recalculated even if we reschedule to
    * the same process) Currently this is only used on SMP,
    * and it's approximate, so we do not have to maintain
    * it while holding the runqueue spinlock.
    */

//   sched_data->last_schedule = get_cycles();

   /*
    * We drop the scheduler lock early (it's a global spinlock),
    * thus we have to lock the previous process from getting
    * rescheduled during switch_to().
    */

/*#endif*/ /* CONFIG_SMP */

   /* kstat.context_swtch++; */
   /*
    * there are 3 processes which are affected by a context switch:
    *
    * prev == .... ==> (last => next)
    *
    * It's the 'much more previous' 'prev' that is on next's stack,
    * but prev is set to (the just run) 'last' process by switch_to().
    * This might sound slightly confusing but makes tons of sense.
    */
   prepare_to_switch();
   /*
   {
      struct mm_struct *mm = next->mm;
      struct mm_struct *oldmm = prev->active_mm;
      if (!mm) {
         if (next->active_mm) BUG();
         next->active_mm = oldmm;
         atomic_inc(&oldmm->mm_count);
         enter_lazy_tlb(oldmm, next, this_cpu);
      } else {
         if (next->active_mm != mm) BUG();
         switch_mm(oldmm, mm, next, this_cpu);
      }

      if (!prev->mm) {
         prev->active_mm = NULL;
         mmdrop(oldmm);
      }
   }
   */

   /*
    * This just switches the register state and the
    * stack.
    */
   switch_to(prev, next, prev);
   __schedule_tail(prev);

 same_process:

   /* reacquire_kernel_lock(current); */
   if (get_current()->need_resched)
      goto need_resched_back;
   return;
}

void sched_init(Task *idle_task)
{
   int cpu = smp_processor_id();
   if (!cpu)
   {
      current = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      idle_tasks = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      cpu_schedule_data = (struct schedule_data*)
         malloc(sizeof(struct schedule_data)*smp_num_cpus);
   }

   idle_tasks[cpu] = idle_task;
   /* set the idle task as the active task the CPU */
   task_set_cpu(idle_task, cpu);
}

/* Called for each timer interrupt on each processor. */
void do_timer()
{
   int cpu = smp_processor_id();
   /* A tick actually shouldent be cpu specific. */
   /* For now we work out the single cpu problem. */
   Task *p = get_current();
   if (p == idle_task(cpu))
   {
      p->need_resched = 1;
   }

   if (--p->counter <= 0) {
      p->counter = 0;
      p->need_resched = 1;
   }
}

/* Return the idle task for the spec. cpu */
Task* idle_task(int cpu)
{
   return idle_tasks[cpu];
}

/* Return the current task on this cpu */
Task* get_current()
{
   return current[smp_processor_id()];
}


/* Remove a task from the run_queue */
```

```
void del_from_runqueue(Task *task)
{
   run_queue.remove(task);
}

/* Set the mask for the task */
void task_set_cpu(Task *task, int cpu)
{
   task->set_state(TASK_RUNNING);
   current[cpu] = task;
   task->processor = cpu;
   task->cpus_runnable = 1UL << cpu;
}

/* Change to mask to have no cpu */
void task_release_cpu(Task *task)
{
   task->cpus_runnable = ~0UL;
}

/* Examine if the task is running on a cpu */
int task_has_cpu(Task *task)
{
   return (task->cpus_runnable != ~0UL);
}

/* Move a task to the last of the runqueue */
void move_last_runqueue(Task *task)
{
   del_from_runqueue(task);
   run_queue.push_back(task);
}

/* Move a task to the front of a runqueue */
void move_first_runqueue(Task *task)
{
   del_from_runqueue(task);
   run_queue.push_front(task);
}

/* Add a task to the runqueue */
void add_to_runqueue(Task *task)
{
   run_queue.push_front(task);
   /* nr_running++; */
}

/* Examine if a task can run on the spec. CPU */
int can_schedule(Task *p, int cpu)
{
   return (p->cpus_runnable & p->cpus_allowed & (1 << cpu));
}

Task* cpu_curr(int cpu)
{
   return current[cpu];
}
```

```
/*
 * On x86 all CPUs are mapped 1:1 to the APIC space.
 * This simplifies scheduling and IPI sending and
 * compresses data structures.
 */
int cpu_logical_map(int cpu)
{
   return cpu;
}

int cpu_number_map(int cpu)
{
   return cpu;
}

/* Check if the task is on any runqueue */
int task_on_runqueue(Task *p)
{
   /* Go through the runqueue */
   /* In linux, the thread would know itself through p->run_list.next */
   return (find(run_queue.begin(), run_queue.end(), p) != run_queue.end());
}

/* Retrive number of ticks since last schedule in this cpu */
/* This is updated for every tick */
cycles_t last_schedule(int cpu)
{
   return cpu_schedule_data[cpu].last_schedule;
}

/* Get System Ticks */
cycles_t get_cycles()
{
   return jiffies;
}
```

sched-second_level.cc

```
#include <vector>
#include <math.h>
#include <algorithm>
#include "simulator_interface.h"
#include "sched_interface.h"
#include "sched.h"
#include "sched-first_level.h"
#include "globals.h"


using namespace std;

/* A list of running programs on the cpu's */
Task ** current;
Task ** idle_tasks;
/* Per CPU data */
struct schedule_data *cpu_schedule_data;
```

```
/* The run queue */
list<Task*> run_queue;

#define PROC_CHANGE_PENALTY 15

#define unlikely(x) x
/* Actually this is the original define */

#define prepare_to_switch() do { } while(0)
/* Assembler macro - This sets up the new task.
 * This is not needed in this simulator */
#define switch_to(prev,next,last) do { } while(0)


/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x) ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x) ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x) (x)
#elif HZ < 1600
#define TICK_SCALE(x) ((x) << 1)
#else
#define TICK_SCALE(x) ((x) << 2)
#endif

#define NICE_TO_TICKS(nice) (TICK_SCALE(20-(nice))+1)


/* Setup idle tasks.
  These will only be run when there is no items in
  the run_queue */
void init_cpu(int cpu, Task *idle_task)
{
   idle_tasks[cpu] = idle_task;
   /* set the idle task as the active task the CPU */
   task_set_cpu(idle_task, cpu);
}

int preemption_goodness(Task *prev, Task *p, int cpu)
{
   return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->
        active_mm);
}
```

```
void reschedule_idle(Task *p)
{
      int this_cpu = smp_processor_id();
      Task *tsk, *target_tsk;
      int cpu, best_cpu, i, max_prio;
      cycles_t oldest_idle;

      /*
       * shortcut if the woken up task's last CPU is
       * idle now.
       */
      best_cpu = p->processor;
      if (can_schedule(p, best_cpu)) {
            tsk = idle_task(best_cpu);
            if (cpu_curr(best_cpu) == tsk) {
                  int need_resched;
send_now_idle:
                  /*
                   * If need_resched == -1 then we can skip sending
                   * the IPI altogether, tsk->need_resched is
                   * actively watched by the idle thread.
                   */
                  need_resched = tsk->need_resched;
                  tsk->need_resched = 1;
                  if ((best_cpu != this_cpu) && !need_resched)
                        smp_send_reschedule(best_cpu);
                  return;
            }
      }

      /*
       * We know that the preferred CPU has a cache-affine current
       * process, lets try to find a new idle CPU for the woken-up
       * process. Select the least recently active idle CPU. (that
       * one will have the least active cache context.) Also find
       * the executing process which has the least priority.
       */
      oldest_idle = (cycles_t) -1;
      target_tsk = NULL;
      max_prio = 0;

      for (i = 0; i < smp_num_cpus; i++) {
            cpu = cpu_logical_map(i);
            if (!can_schedule(p, cpu))
                  continue;
            tsk = cpu_curr(cpu);
            /*
             * We use the first available idle CPU. This creates
             * a priority list between idle CPUs, but this is not
             * a problem.
             */
            if (tsk == idle_task(cpu)) {
/* Added in 2.4.17
#if defined(__i386__) && defined(CONFIG_SMP)
                 *//*
                   * Check if two siblings are idle in the same
                   * physical package. Use them if found.
                   *//*
                   if (smp_num_siblings == 2) {
                         if (cpu_curr(cpu_sibling_map[cpu]) ==
                            idle_task(cpu_sibling_map[cpu])) {
                               oldest_idle = last_schedule(cpu);
                               target_tsk = tsk;
                               break;
                         }

                  }
#endif
*/
                  if (last_schedule(cpu) < oldest_idle) {
                        oldest_idle = last_schedule(cpu);
                        target_tsk = tsk;
                  }
            } else {
                  if (oldest_idle == -1ULL) {
                        int prio = preemption_goodness(tsk, p, cpu);

                        if (prio > max_prio) {
                              max_prio = prio;
                              target_tsk = tsk;
                        }
                  }
            }
      }
      tsk = target_tsk;
      if (tsk) {
            if (oldest_idle != -1ULL) {
                  best_cpu = tsk->processor;
                  goto send_now_idle;
            }
            tsk->need_resched = 1;
            if (tsk->processor != this_cpu)
                  smp_send_reschedule(tsk->processor);
      }
      return;
}

/*
 * Wake up a process. Put it on the run-queue if it's not
 * already there. The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you're allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
static inline int try_to_wake_up(Task *p, int synchronous)
{
   //unsigned long flags;
      int success = 0;

      /*
       * We want the common case fall through straight, thus the goto.
       */
      /* spin_lock_irqsave(&runqueue_lock, flags); */
      p->state = TASK_RUNNING;
      if (task_on_runqueue(p))
```

```
            goto out;
      add_to_runqueue(p);
      if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
            reschedule_idle(p);
      success = 1;
out:
      /* spin_unlock_irqrestore(&runqueue_lock, flags); */
      return success;
}

int wake_up_process(Task * p)
{
      return try_to_wake_up(p, 0);
}


/* Calculate the goodness of the process */
int goodness(Task* p, int this_cpu, int this_mm)
{
   /* This is soly based on quantum and nice-level */
   int weight = -1;
   if (p->policy & SCHED_YIELD)
      return weight;

   if (p->policy == SCHED_OTHER) {
      /*
       * Give the process a first-approximation goodness value
       * according to the number of clock-ticks it has left.
       *
       * Don't do any other calculations if the time slice is
       * over..
       */
      weight = p->counter;
      if (!weight)
         return weight;

      if (p->processor == this_cpu)
         weight += PROC_CHANGE_PENALTY;

      /* .. and a slight advantage to the current MM */


      if (p->mm == this_mm || !p->mm)
         weight += 1;


      weight += 20 - p->nice;

      return weight;
   }

   /*
    * Realtime process, select the first one on the
    * runqueue (taking priorities within processes
    * into account).
    */
   weight = 1000 + p->rt_priority;
   return weight;
```

```
}

void __schedule_tail(Task* prev)
{
   task_release_cpu(prev);
}



void schedule_tail(Task* prev)
{
   __schedule_tail(prev);
}



void schedule()
{
   int this_cpu = smp_processor_id();
   struct schedule_data * sched_data;
   Task *prev, *next;
   int c;

need_resched_back:

   prev = get_current();
   next = idle_task(this_cpu);


   /* This is not modeled in the simulator
   if (unlikely(in_interrupt())) {
      printk("Scheduling in interrupt\n");
      BUG();
   }
   */

   /* release_kernel_lock(prev, this_cpu); */

   /*
    * 'sched_data' is protected by the fact that we can run
    * only one process per CPU.
    */
   sched_data = &cpu_schedule_data[this_cpu];

   /* spin_lock_irq(&runqueue_lock); */

   /* move an exhausted RR process to be last.. */
   if (unlikely(prev->policy == SCHED_RR))
      if (!prev->counter) {
         prev->counter = NICE_TO_TICKS(prev->nice);
         move_last_runqueue(prev);
      }
   /*
   switch (prev->state) {
      case TASK_INTERRUPTIBLE:
         if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
         }
      default:
          del_from_runqueue(prev);
      case TASK_RUNNING:;
   }
   */
   /* Since tasks themself set the state running, no need to check for
    * a pending signal. All in all this boild down to the following: */
   if (prev->state != TASK_RUNNING)
      del_from_runqueue(prev);

   prev->need_resched = 0;

   /*
    * this is the scheduler proper:
    */

repeat_schedule:

   /*
    * Default process to select..
    */
   c = -1000;
   list<Task*>::iterator iter;
   for (iter = run_queue.begin(); iter != run_queue.end(); iter++)
   {
      if (can_schedule((*iter), this_cpu)) {
         int weight = goodness((*iter), this_cpu, prev->active_mm);
         if (weight > c)
            c = weight, next = (*iter);
      }
   }

   /* Do we need to re-calculate counters? */
   if (!c) {
      /* Recalculate all counter on all tasks. */
      list<Task*>::iterator p;
      /* spin_unlock_irq(&runqueue_lock);
       * read_lock(&tasklist_lock);
       */
      for_all_tasks(p)
         (*p)->counter = ((*p)->counter >> 1) +
         NICE_TO_TICKS((*p)->nice);

      /*
       * read_unlock(&tasklist_lock);
       * spin_lock_irq(&runqueue_lock);
       */
      goto repeat_schedule;
   }

   /*
    * from this point on nothing can prevent us from
    * switching to the next task, save this fact in
    * sched_data.
    */

   /* Add the task to the running list */
   /* Mark which cpu the task is running on */

   /* Sched data is locale for the cpu itself */

   //sched_data->curr = next;
   schedule_tail(prev);
   task_set_cpu(next, this_cpu);
   /* spin_unlock_irq(&runqueue_lock); */

   if (unlikely(prev == next)) {
      /* We won't go through the normal tail, so do this by hand */
      prev->policy &= ~SCHED_YIELD;
      goto same_process;
   }

/*#ifdef CONFIG_SMP*/
   /*
    * maintain the per-process 'last schedule' value.
    * (this has to be recalculated even if we reschedule to
    * the same process) Currently this is only used on SMP,
    * and it's approximate, so we do not have to maintain
    * it while holding the runqueue spinlock.
    */

//   sched_data->last_schedule = get_cycles();

   /*
    * We drop the scheduler lock early (it's a global spinlock),
    * thus we have to lock the previous process from getting
    * rescheduled during switch_to().
    */

/*#endif*/ /* CONFIG_SMP */

   /* kstat.context_swtch++; */
   /*
    * there are 3 processes which are affected by a context switch:
    *
    * prev == .... ==> (last => next)
    *
    * It's the 'much more previous' 'prev' that is on next's stack,
    * but prev is set to (the just run) 'last' process by switch_to().
    * This might sound slightly confusing but makes tons of sense.
    */
   prepare_to_switch();
   /*
   {
      struct mm_struct *mm = next->mm;
      struct mm_struct *oldmm = prev->active_mm;
      if (!mm) {
         if (next->active_mm) BUG();
         next->active_mm = oldmm;
         atomic_inc(&oldmm->mm_count);
         enter_lazy_tlb(oldmm, next, this_cpu);
      } else {
         if (next->active_mm != mm) BUG();
         switch_mm(oldmm, mm, next, this_cpu);
      }
      if (!prev->mm) {
         prev->active_mm = NULL;
         mmdrop(oldmm);
      }
   }
   */

   /*
    * This just switches the register state and the
    * stack.
    */
   switch_to(prev, next, prev);
   __schedule_tail(prev);

 same_process:

   /* reacquire_kernel_lock(current); */
   if (get_current()->need_resched)
      goto need_resched_back;
   return;
}

/* Called for each timer interrupt on each processor. */
void do_timer()
{
   int cpu = smp_processor_id();
   /* Send a tick to the high level scheduler */
   if (!cpu)
      high_level_tick();

   /* A tick actually shouldent be cpu specific. */
   /* For now we work out the single cpu problem. */
   Task *p = get_current();
   if (p == idle_task(cpu))
   {
      p->need_resched = 1;
   }

   if (--p->counter <= 0) {
      p->counter = 0;
      p->need_resched = 1;
   }
}

/* Return the idle task for the spec. cpu */
Task* idle_task(int cpu)
{
   return idle_tasks[cpu];
}

/* Return the current task on this cpu */
Task* get_current()
{
   return current[smp_processor_id()];
}
```

```
/* Remove a task from the run_queue */
void sc_del_from_runqueue(Task *task)
{
   run_queue.remove(task);
}

/* Set the mask for the task */
void task_set_cpu(Task *task, int cpu)
{
   task->set_state(TASK_RUNNING);
   current[cpu] = task;
   task->processor = cpu;
   task->cpus_runnable = 1UL << cpu;
}

/* Change to mask to have no cpu */
void task_release_cpu(Task *task)
{
   task->cpus_runnable = ~0UL;
}

/* Examine if the task is running on a cpu */
int task_has_cpu(Task *task)
{
   return (task->cpus_runnable != ~0UL);
}

/* Move a task to the last of the runqueue */
void move_last_runqueue(Task *task)
{
   del_from_runqueue(task);
   run_queue.push_back(task);
}

/* Move a task to the front of a runqueue */
void move_first_runqueue(Task *task)
{
   del_from_runqueue(task);
   run_queue.push_front(task);
}

/* Add a task to the runqueue */
void sc_add_to_runqueue(Task *task)
{
   run_queue.push_front(task);
   /* nr_running++; */
}

/* Examine if a task can run on the spec. CPU */
int can_schedule(Task *p, int cpu)
{
   return (p->cpus_runnable & p->cpus_allowed & (1 << cpu));
}

Task* cpu_curr(int cpu)
{
   return current[cpu];
}
```

```
/*
 * On x86 all CPUs are mapped 1:1 to the APIC space.
 * This simplifies scheduling and IPI sending and
 * compresses data structures.
 */
int cpu_logical_map(int cpu)
{
   return cpu;
}

int cpu_number_map(int cpu)
{
   return cpu;
}

/* Check if the task is on any runqueue */
int task_on_runqueue(Task *p)
{
   /* Go through the runqueue */
   /* In linux, the thread would know itself through p->run_list.next */
   return (find(run_queue.begin(), run_queue.end(), p) != run_queue.end());
}

/* Retrive number of ticks since last schedule in this cpu */
/* This is updated for every tick */
cycles_t last_schedule(int cpu)
{
   return cpu_schedule_data[cpu].last_schedule;
}

/* Get System Ticks */
cycles_t get_cycles()
{
   return jiffies;
}

void sched_init(Task *idle_task)
{
   int cpu = smp_processor_id();
   if (!cpu)
   {
      current = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      idle_tasks = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      cpu_schedule_data = (struct schedule_data*)
         malloc(sizeof(struct schedule_data)*smp_num_cpus);

      high_level_init();
   }

   idle_tasks[cpu] = idle_task;
   /* set the idle task as the active task the CPU */
   task_set_cpu(idle_task, cpu);
}
```

sched-test.cc

```
#include <list>
#include "globals.h"
#include "sched.h"
#include "sched_interface.h"
#include "task.h"

using namespace std;

/* A list of running programs on the cpu's */
Task ** current;
Task ** idle_tasks;
/* Per CPU data */
struct schedule_data *cpu_schedule_data;

cycles_t jiffies;


/* The run queue */
list<Task*> run_queue;

#define HZ 100
#define PROC_CHANGE_PENALTY 15


/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x) ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x) ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x) (x)
#elif HZ < 1600
#define TICK_SCALE(x) ((x) << 1)
#else
#define TICK_SCALE(x) ((x) << 2)
#endif

#define NICE_TO_TICKS(nice) (TICK_SCALE(20-(nice))+1)


/* Setup idle tasks.
  These will only be run when there is no items in
  the run_queue */

int preemption_goodness(Task *prev, Task *p, int cpu)
{
   return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->
        active_mm);
```

```
}

void reschedule_idle(Task *p)
{
      int this_cpu = smp_processor_id();
      Task *tsk, *target_tsk;
      int cpu, best_cpu, i, max_prio;
      cycles_t oldest_idle;

      /*
       * shortcut if the woken up task's last CPU is
       * idle now.
       */
      best_cpu = p->processor;
      if (can_schedule(p, best_cpu)) {
            tsk = idle_task(best_cpu);
            if (cpu_curr(best_cpu) == tsk) {
                  int need_resched;
send_now_idle:
                  /*
                   * If need_resched == -1 then we can skip sending
                   * the IPI altogether, tsk->need_resched is
                   * actively watched by the idle thread.
                   */
                  need_resched = tsk->need_resched;
                  tsk->need_resched = 1;
                  if ((best_cpu != this_cpu) && !need_resched)
                        smp_send_reschedule(best_cpu);
                  return;
            }
      }

      /*
       * We know that the preferred CPU has a cache-affine current
       * process, lets try to find a new idle CPU for the woken-up
       * process. Select the least recently active idle CPU. (that
       * one will have the least active cache context.) Also find
       * the executing process which has the least priority.
       */
      oldest_idle = (cycles_t) -1;
      target_tsk = NULL;
      max_prio = 0;

      for (i = 0; i < smp_num_cpus; i++) {
            cpu = cpu_logical_map(i);
            if (!can_schedule(p, cpu))
                  continue;
            tsk = cpu_curr(cpu);
            /*
             * We use the first available idle CPU. This creates
             * a priority list between idle CPUs, but this is not
             * a problem.
             */
            if (tsk == idle_task(cpu)) {
/* Added in 2.4.17
#if defined(__i386__) && defined(CONFIG_SMP)
                 *//*
                   * Check if two siblings are idle in the same
                   * physical package. Use them if found.
                   *//*
                   if (smp_num_siblings == 2) {
                        if (cpu_curr(cpu_sibling_map[cpu]) ==
                           idle_task(cpu_sibling_map[cpu])) {
                              oldest_idle = last_schedule(cpu);
                              target_tsk = tsk;
                              break;
                        }

                  }
#endif
*/
                  if (last_schedule(cpu) < oldest_idle) {
                        oldest_idle = last_schedule(cpu);
                        target_tsk = tsk;
                  }
            } else {
                  if (oldest_idle == -1ULL) {
                        int prio = preemption_goodness(tsk, p, cpu);

                        if (prio > max_prio) {
                              max_prio = prio;
                              target_tsk = tsk;
                        }
                  }
            }
      }
      tsk = target_tsk;
      if (tsk) {
            if (oldest_idle != -1ULL) {
                  best_cpu = tsk->processor;
                  goto send_now_idle;
            }
            tsk->need_resched = 1;
            if (tsk->processor != this_cpu)
                  smp_send_reschedule(tsk->processor);
      }
      return;
}

/*
 * Wake up a process. Put it on the run-queue if it's not
 * already there. The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you're allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
static inline int try_to_wake_up(Task *p, int synchronous)
{
   //unsigned long flags;
      int success = 0;

      /*
       * We want the common case fall through straight, thus the goto.
       */
      /* spin_lock_irqsave(&runqueue_lock, flags); */
      p->state = TASK_RUNNING;
      if (task_on_runqueue(p))
            goto out;
      add_to_runqueue(p);
      if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
            reschedule_idle(p);
      success = 1;
out:
      /* spin_unlock_irqrestore(&runqueue_lock, flags); */
      return success;
}

int wake_up_process(Task * p)
{
      return try_to_wake_up(p, 0);
}


/* Calculate the goodness of the process */
int goodness(Task* p, int this_cpu, int this_mm)
{
   /* This is soly based on quantum and nice-level */
   int weight = -1;
   if (p->policy & SCHED_YIELD)
      return weight;

   if (p->policy == SCHED_OTHER) {
      /*
       * Give the process a first-approximation goodness value
       * according to the number of clock-ticks it has left.
       *
       * Don't do any other calculations if the time slice is
       * over..
       */
      weight = p->counter;
      if (!weight)
         return weight;

      if (p->processor == this_cpu)
         weight += PROC_CHANGE_PENALTY;

      /* .. and a slight advantage to the current MM */


      if (p->mm == this_mm || !p->mm)
         weight += 1;


      weight += 20 - p->nice;

      return weight;
   }

   /*
    * Realtime process, select the first one on the
    * runqueue (taking priorities within processes
    * into account).
    */
   weight = 1000 + p->rt_priority;
   return weight;
}

void __schedule_tail(Task* prev)
{
   task_release_cpu(prev);
}



void schedule_tail(Task* prev)
{
   __schedule_tail(prev);
}



void schedule(int this_cpu)
{
   struct schedule_data * sched_data;
   Task *prev, *next;
   int c;

need_resched_back:

   prev = get_current(this_cpu);
   next = idle_task(this_cpu);


   /* This is not modeled in the simulator
   if (unlikely(in_interrupt())) {
      printk("Scheduling in interrupt\n");
      BUG();
   }
   */

   /* release_kernel_lock(prev, this_cpu); */

   /*
    * 'sched_data' is protected by the fact that we can run
    * only one process per CPU.
    */
   sched_data = &cpu_schedule_data[this_cpu];

   /* spin_lock_irq(&runqueue_lock); */

   /* move an exhausted RR process to be last.. */
   if (unlikely(prev->policy == SCHED_RR))
      if (!prev->counter) {
         prev->counter = NICE_TO_TICKS(prev->nice);
         move_last_runqueue(prev);
      }
   /*
   switch (prev->state) {
      case TASK_INTERRUPTIBLE:
         if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
         }
      default:
          del_from_runqueue(prev);
      case TASK_RUNNING:;
   }
   */
   /* Since tasks themself set the state running, no need to check for
    * a pending signal. All in all this boild down to the following: */
   if (prev->state != TASK_RUNNING)
      del_from_runqueue(prev);

   prev->need_resched = 0;

   /*
    * this is the scheduler proper:
    */

repeat_schedule:

   /*
    * Default process to select..
    */
   c = -1000;
   list<Task*>::iterator iter;
   for (iter = run_queue.begin(); iter != run_queue.end(); iter++)
   {
      if (can_schedule((*iter), this_cpu)) {
         int weight = goodness((*iter), this_cpu, prev->active_mm);
         if (weight > c)
            c = weight, next = (*iter);
      }
   }

   /* Do we need to re-calculate counters? */
   if (!c) {
      /* Recalculate all counter on all tasks. */
      list<Task*>::iterator p;
      /* spin_unlock_irq(&runqueue_lock);
       * read_lock(&tasklist_lock);
       */
      for_all_tasks(p)
         (*p)->counter = NICE_TO_TICKS((*p)->nice);

      /*
       * read_unlock(&tasklist_lock);
       * spin_lock_irq(&runqueue_lock);
       */
      goto repeat_schedule;
   }

   /*
    * from this point on nothing can prevent us from
    * switching to the next task, save this fact in
    * sched_data.
    */

   /* Add the task to the running list */
   /* Mark which cpu the task is running on */

   /* Sched data is locale for the cpu itself */

   //sched_data->curr = next;
   schedule_tail(prev);
   task_set_cpu(next, this_cpu);
   /* spin_unlock_irq(&runqueue_lock); */

   if (unlikely(prev == next)) {
      /* We won’t go through the normal tail, so do this by hand */
      prev->policy &= ˜SCHED_YIELD;
      goto same_process;
   }

/*#ifdef CONFIG_SMP*/
   /*
    * maintain the per-process ’last schedule’ value.
    * (this has to be recalculated even if we reschedule to
    * the same process) Currently this is only used on SMP,
    * and it’s approximate, so we do not have to maintain
    * it while holding the runqueue spinlock.
    */

//   sched_data->last_schedule = get_cycles();

   /*
    * We drop the scheduler lock early (it’s a global spinlock),
    * thus we have to lock the previous process from getting
    * rescheduled during switch_to().
    */

/*#endif*/ /* CONFIG_SMP */

   /* kstat.context_swtch++; */
   /*
    * there are 3 processes which are affected by a context switch:
    *
    * prev == .... ==> (last => next)
    *
    * It’s the ’much more previous’ ’prev’ that is on next’s stack,
    * but prev is set to (the just run) ’last’ process by switch_to().
    * This might sound slightly confusing but makes tons of sense.
    */
   prepare_to_switch();
   /*
   {
      struct mm_struct *mm = next->mm;
      struct mm_struct *oldmm = prev->active_mm;
      if (!mm) {
         if (next->active_mm) BUG();
         next->active_mm = oldmm;
         atomic_inc(&oldmm->mm_count);
         enter_lazy_tlb(oldmm, next, this_cpu);
      } else {
         if (next->active_mm != mm) BUG();
         switch_mm(oldmm, mm, next, this_cpu);
      }
      if (!prev->mm) {
         prev->active_mm = NULL;
         mmdrop(oldmm);
      }
   }
   */

   /*
    * This just switches the register state and the
    * stack.
    */
   switch_to(prev, next, prev);
   __schedule_tail(prev);

 same_process:

   /* reacquire_kernel_lock(current); */
   if (get_current(this_cpu)->need_resched)
      goto need_resched_back;
   return;
}

/* Called for each timer interrupt on each processor. */
void do_timer(int cpu)
{
   /* A tick actually shouldent be cpu specific. */
   /* For now we work out the single cpu problem. */
   Task *p = get_current(cpu);
   if (p == idle_task(cpu))
   {
      p->need_resched = 1;
   }

   if (--p->counter <= 0) {
      p->counter = 0;
      p->need_resched = 1;
   }
}

void sched_init(int cpu, Task *idle_task)
{
   if (!cpu)
   {
      current = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      idle_tasks = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      cpu_schedule_data = (struct schedule_data*)
         malloc(sizeof(struct schedule_data)*smp_num_cpus);
   }

   idle_tasks[cpu] = idle_task;
   /* set the idle task as the active task the CPU */
   task_set_cpu(idle_task, cpu);
}
```

sched.cc

```
#include <list>
#include <vector>
#include <math.h>
#include <algorithm>
#include "simulator_interface.h"
#include "sched_interface.h"
#include "sched.h"
#include "globals.h"

using namespace std;

/* A list of running programs on the cpu’s */
Task ** current;
Task ** idle_tasks;
/* Per CPU data */
struct schedule_data *cpu_schedule_data;

/* The run queue */
list<Task*> run_queue;

#define PROC_CHANGE_PENALTY 15

#define unlikely(x) x
/* Actually this is the original define */
#define prepare_to_switch() do { } while(0)

/* Assembler macro - This sets up the new task.
 * This is not needed in this simulator */
#define switch_to(prev,next,last) do { } while(0)

/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x) ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x) ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x) (x)
#elif HZ < 1600
#define TICK_SCALE(x) ((x) << 1)
#else
#define TICK_SCALE(x) ((x) << 2)
#endif

#define NICE_TO_TICKS(nice) (TICK_SCALE(20-(nice))+1)


int preemption_goodness(Task *prev, Task *p, int cpu)
{
   return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->
        active_mm);
}

void reschedule_idle(Task *p)
{
      int this_cpu = smp_processor_id();
      Task *tsk, *target_tsk;
      int cpu, best_cpu, i, max_prio;
      cycles_t oldest_idle;

      /*
       * shortcut if the woken up task's last CPU is
       * idle now.
       */
      best_cpu = p->processor;
      if (can_schedule(p, best_cpu)) {
            tsk = idle_task(best_cpu);
            if (cpu_curr(best_cpu) == tsk) {
                  int need_resched;
send_now_idle:
                  /*
                   * If need_resched == -1 then we can skip sending
                   * the IPI altogether, tsk->need_resched is
                   * actively watched by the idle thread.
                   */
                  need_resched = tsk->need_resched;
                  tsk->need_resched = 1;
                  if ((best_cpu != this_cpu) && !need_resched)
                        smp_send_reschedule(best_cpu);
                  return;
            }
      }

      /*
       * We know that the preferred CPU has a cache-affine current
       * process, lets try to find a new idle CPU for the woken-up
       * process. Select the least recently active idle CPU. (that
       * one will have the least active cache context.) Also find
       * the executing process which has the least priority.
       */
      oldest_idle = (cycles_t) -1;
      target_tsk = NULL;
      max_prio = 0;

      for (i = 0; i < smp_num_cpus; i++) {
            cpu = cpu_logical_map(i);
            if (!can_schedule(p, cpu))
                  continue;
            tsk = cpu_curr(cpu);
            /*
             * We use the first available idle CPU. This creates
             * a priority list between idle CPUs, but this is not
             * a problem.
             */
            if (tsk == idle_task(cpu)) {
/* Added in 2.4.17
#if defined(__i386__) && defined(CONFIG_SMP)
```

```
                 *//*
                   * Check if two siblings are idle in the same
                   * physical package. Use them if found.
                   *//*
                  if (smp_num_siblings == 2) {
                       if (cpu_curr(cpu_sibling_map[cpu]) ==
                          idle_task(cpu_sibling_map[cpu])) {
                             oldest_idle = last_schedule(cpu);
                             target_tsk = tsk;
                             break;
                       }

                  }
#endif
*/
                  if (last_schedule(cpu) < oldest_idle) {
                        oldest_idle = last_schedule(cpu);
                        target_tsk = tsk;
                  }
            } else {
                  if (oldest_idle == -1ULL) {
                        int prio = preemption_goodness(tsk, p, cpu);

                        if (prio > max_prio) {
                              max_prio = prio;
                              target_tsk = tsk;
                        }
                  }
            }
      }
      tsk = target_tsk;
      if (tsk) {
            if (oldest_idle != -1ULL) {
                  best_cpu = tsk->processor;
                  goto send_now_idle;
            }
            tsk->need_resched = 1;
            if (tsk->processor != this_cpu)
                  smp_send_reschedule(tsk->processor);
      }
      return;
}

/*
 * Wake up a process. Put it on the run-queue if it’s not
 * already there. The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you’re allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
static inline int try_to_wake_up(Task *p, int synchronous)
{
   //unsigned long flags;
      int success = 0;

      /*
       * We want the common case fall through straight, thus the goto.
```

```
        */
       /* spin_lock_irqsave(&runqueue_lock, flags); */
       p->state = TASK_RUNNING;
       if (task_on_runqueue(p))
             goto out;
       add_to_runqueue(p);
       if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
             reschedule_idle(p);
       success = 1;
out:
       /* spin_unlock_irqrestore(&runqueue_lock, flags); */
       return success;
}

int wake_up_process(Task * p)
{
       return try_to_wake_up(p, 0);
}


/* Calculate the goodness of the process */
int goodness(Task* p, int this_cpu, int this_mm)
{
   /* This is soly based on quantum and nice-level */
   int weight = -1;
   if (p->policy & SCHED_YIELD)
      return weight;

   if (p->policy == SCHED_OTHER) {
      /*
       * Give the process a first-approximation goodness value
       * according to the number of clock-ticks it has left.
       *
       * Don’t do any other calculations if the time slice is
       * over..
       */
      weight = p->counter;
      if (!weight)
         return weight;

      if (p->processor == this_cpu)
         weight += PROC_CHANGE_PENALTY;

      /* .. and a slight advantage to the current MM */


      if (p->mm == this_mm || !p->mm)
         weight += 1;


      weight += 20 - p->nice;

      return weight;
   }

   /*
    * Realtime process, select the first one on the
    * runqueue (taking priorities within processes
    * into account).
    */
   weight = 1000 + p->rt_priority;
   return weight;
}

void __schedule_tail(Task* prev)
{
   task_release_cpu(prev);
}



void schedule_tail(Task* prev)
{
   __schedule_tail(prev);
}



void schedule()
{
   int this_cpu = smp_processor_id();
   struct schedule_data * sched_data;
   Task *prev, *next;
   int c;

need_resched_back:

   prev = get_current();
   next = idle_task(this_cpu);


   /* This is not modeled in the simulator
   if (unlikely(in_interrupt())) {
      printk("Scheduling in interrupt\n");
      BUG();
   }
   */

   /* release_kernel_lock(prev, this_cpu); */

   /*
    * 'sched_data' is protected by the fact that we can run
    * only one process per CPU.
    */
   sched_data = &cpu_schedule_data[this_cpu];

   /* spin_lock_irq(&runqueue_lock); */

   /* move an exhausted RR process to be last.. */
   if (unlikely(prev->policy == SCHED_RR))
      if (!prev->counter) {
         prev->counter = NICE_TO_TICKS(prev->nice);
         move_last_runqueue(prev);
      }
   /*
   switch (prev->state) {
      case TASK_INTERRUPTIBLE:
         if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
         }
      default:
          del_from_runqueue(prev);
      case TASK_RUNNING:;
   }
   */
   /* Since tasks themself set the state running, no need to check for
    * a pending signal. All in all this boild down to the following: */
   if (prev->state != TASK_RUNNING)
      del_from_runqueue(prev);

   prev->need_resched = 0;

   /*
    * this is the scheduler proper:
    */

repeat_schedule:

   /*
    * Default process to select..
    */
   c = -1000;
   list<Task*>::iterator iter;
   for (iter = run_queue.begin(); iter != run_queue.end(); iter++)
   {
      if (can_schedule((*iter), this_cpu)) {
         int weight = goodness((*iter), this_cpu, prev->active_mm);
         if (weight > c)
            c = weight, next = (*iter);
      }
   }

   /* Do we need to re-calculate counters? */
   if (!c) {
      /* Recalculate all counter on all tasks. */
      list<Task*>::iterator p;
      /* spin_unlock_irq(&runqueue_lock);
       * read_lock(&tasklist_lock);
       */
      for_all_tasks(p)
         (*p)->counter = ((*p)->counter >> 1) +
         NICE_TO_TICKS((*p)->nice);

      /*
       * read_unlock(&tasklist_lock);
       * spin_lock_irq(&runqueue_lock);
       */
      goto repeat_schedule;
   }

   /*
    * from this point on nothing can prevent us from
    * switching to the next task, save this fact in
    * sched_data.
    */

   /* Add the task to the running list */
   /* Mark which cpu the task is running on */

   /* Sched data is locale for the cpu itself */

   //sched_data->curr = next;
   schedule_tail(prev);
   task_set_cpu(next, this_cpu);
   /* spin_unlock_irq(&runqueue_lock); */

   if (unlikely(prev == next)) {
      /* We won't go through the normal tail, so do this by hand */
      prev->policy &= ~SCHED_YIELD;
      goto same_process;
   }

/*#ifdef CONFIG_SMP*/
   /*
    * maintain the per-process 'last schedule' value.
    * (this has to be recalculated even if we reschedule to
    * the same process) Currently this is only used on SMP,
    * and it's approximate, so we do not have to maintain
    * it while holding the runqueue spinlock.
    */

//   sched_data->last_schedule = get_cycles();

   /*
    * We drop the scheduler lock early (it's a global spinlock),
    * thus we have to lock the previous process from getting
    * rescheduled during switch_to().
    */

/*#endif*/ /* CONFIG_SMP */

   /* kstat.context_swtch++; */
   /*
    * there are 3 processes which are affected by a context switch:
    *
    * prev == .... ==> (last => next)
    *
    * It's the 'much more previous' 'prev' that is on next's stack,
    * but prev is set to (the just run) 'last' process by switch_to().
    * This might sound slightly confusing but makes tons of sense.
    */
   prepare_to_switch();
   /*
   {
      struct mm_struct *mm = next->mm;
      struct mm_struct *oldmm = prev->active_mm;
      if (!mm) {
         if (next->active_mm) BUG();
         next->active_mm = oldmm;
         atomic_inc(&oldmm->mm_count);
         enter_lazy_tlb(oldmm, next, this_cpu);
      } else {
         if (next->active_mm != mm) BUG();
         switch_mm(oldmm, mm, next, this_cpu);
      }

      if (!prev->mm) {
         prev->active_mm = NULL;
         mmdrop(oldmm);
      }
   }
   */

   /*
    * This just switches the register state and the
    * stack.
    */
   switch_to(prev, next, prev);
   __schedule_tail(prev);

 same_process:

   /* reacquire_kernel_lock(current); */
   if (get_current()->need_resched)
      goto need_resched_back;
   return;
}


/* Called for each timer interrupt on each processor. */
void do_timer()
{
   int cpu=smp_processor_id();
   /* A tick actually shouldent be cpu specific. */
   /* For now we work out the single cpu problem. */
   Task *p = get_current();
   if (p == idle_task(cpu))
   {
      p->need_resched = 1;
   }

   if (--p->counter <= 0) {
      p->counter = 0;
      p->need_resched = 1;
   }
}


/* Return the idle task for the spec. cpu */
Task* idle_task(int cpu)
{
   if (!idle_tasks[cpu])
      printf("## No idle task for PE:%d\n", cpu);
   return idle_tasks[cpu];
}

/* Return the current task on this cpu */
Task* get_current()
{
   return current[smp_processor_id()];
}


/* Remove a task from the run_queue */
void del_from_runqueue(Task *task)
{
   run_queue.remove(task);
}

/* Set the mask for the task */
void task_set_cpu(Task *task, int cpu)
{
   task->set_state(TASK_RUNNING);
   current[cpu] = task;
   task->processor = cpu;
   task->cpus_runnable = 1UL << cpu;
}

/* Change to mask to have no cpu */
void task_release_cpu(Task *task)
{
   task->cpus_runnable = ~0UL;
}

/* Examine if the task is running on a cpu */
int task_has_cpu(Task *task)
{
   return (task->cpus_runnable != ~0UL);
}

/* Move a task to the last of the runqueue */
void move_last_runqueue(Task *task)
{
   del_from_runqueue(task);
   run_queue.push_back(task);
}

/* Move a task to the front of a runqueue */
void move_first_runqueue(Task *task)
{
   del_from_runqueue(task);
   run_queue.push_front(task);
}

/* Add a task to the runqueue */
void add_to_runqueue(Task *task)
{
   run_queue.push_front(task);
   /* nr_running++; */
}

/* Examine if a task can run on the spec. CPU */
int can_schedule(Task *p, int cpu)
{
   return (p->cpus_runnable & p->cpus_allowed & (1 << cpu));
}
```

```
Task* cpu_curr(int cpu)
{
   return current[cpu];
}

/*
 * On x86 all CPUs are mapped 1:1 to the APIC space.
 * This simplifies scheduling and IPI sending and
 * compresses data structures.
 */
int cpu_logical_map(int cpu)
{
   return cpu;
}

int cpu_number_map(int cpu)
{
   return cpu;
}

/* Check if the task is on any runqueue */
int task_on_runqueue(Task *p)
{
   /* Go through the runqueue */
   /* In linux, the thread would know itself through p->run_list.next */
   return (find(run_queue.begin(), run_queue.end(), p) != run_queue.end());
}

/* Retrive number of ticks since last schedule in this cpu */
/* This is updated for every tick */
cycles_t last_schedule(int cpu)
{
   return cpu_schedule_data[cpu].last_schedule;
}

void sched_init(Task *idle_task)
{
   int cpu = smp_processor_id();
   if (!cpu)
   {
      current = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      idle_tasks = (Task **) malloc(sizeof(Task*)*smp_num_cpus);
      cpu_schedule_data = (struct schedule_data*)
         malloc(sizeof(struct schedule_data)*smp_num_cpus);
   }

   idle_tasks[cpu] = idle_task;
   /* set the idle task as the active task the CPU */
   task_set_cpu(idle_task, cpu);
}
```

sched.h

```
#ifndef __SCHED_H__
#define __SCHED_H__
```

```
#include <list>
#include <vector>
#include "globals.h"
#include "task.h"
#include "sched.h"
/* Code generation optimation - Not needed here */
#define unlikely(x) x

/* Actually this is the original define */
#define prepare_to_switch() do { } while(0)

/* Assembler macro - This sets up the new task.
 * This is not needed in this simulator */
#define switch_to(prev,next,last) do { } while(0)

#define for_all_tasks(p) for (p = tasks.begin(); p != tasks.end(); p++)

struct schedule_data {
   cycles_t last_schedule;
   Task* curr;
};

/* A list of running programs on the cpu’s */
extern Task ** current;
extern Task ** idle_tasks;
extern struct schedule_data *cpu_schedule_data;
extern cycles_t jiffies;

/* The run queue */
extern list<Task*> run_queue;

/* Specify idle tasks */

int goodness(Task* p, int this_cpu, int this_mm);

Task* idle_task(int cpu);

Task* current_task();

/* Remove a tash from the run_queue */
void del_from_runqueue(Task *task);

/* Check if the task is on any runqueue */
int task_on_runqueue(Task *p);

void task_set_cpu(Task *task, int cpu);

void task_release_cpu(Task *task);

int task_has_cpu(Task *task);

void move_last_runqueue(Task *task);

void move_first_runqueue(Task *task);

void add_to_runqueue(Task *task);
void add_to_runqueue_tail(Task *task);
```

```
int can_schedule(Task *p, int cpu);

cycles_t last_schedule(int cpu);

Task* cpu_curr(int cpu);

int cpu_logical_map(int cpu);

int cpu_number_map(int cpu);


#endif /* __SCHED_H__ */
```

sched_interface.h

```
#ifndef __SCHED_INTERFACE_H__
#define __SCHED_INTERFACE_H__

#include "simulator_interface.h"

/* Get the id of the "current" processor. */
int smp_processor_id();

/* Initialize structures and specify idle tasks */
void sched_init(Task *idle_task);

void do_timer();

void schedule();

int wake_up_process(Task *p);

int task_has_cpu(Task *task);

Task* get_current();

#endif /* __SCHED_INTERFACE_H__ */
```

simulator.h

```

/* Give all tasks a tick */
/* Call each cpu, and all the tasks */
void tick();

/* Wakeup a specific task */
/* (Tell the scheduler that the task is ready) */
void wakeup();
```

simulator_interface.h

```
#ifndef __SIMULATOR_INTERFACE_H__
#define __SIMULATOR_INTERFACE_H__

#include "task.h"
#include "globals.h"
#include <list>

void smp_send_reschedule(int cpu);

/* Access to all tasks in the system */
extern list<Task*> tasks;

extern unsigned int jiffies;

#endif /* __SIMULATOR_INTERFACE_H__ */
```

task.cc

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sched.h>
#include <list>
#include "task.h"
#include "globals.h"
#include "sched_interface.h"
#include "iosched.h"

using namespace std;

/* Make unique mm's */
static int mm_count = 0;

/*************** TASK **************/
Task::Task(int pid, int grp, int nice, int policy, int run_prob)
{
   /* Default initializers */
   state = TASK_RUNNING;
   policy = SCHED_OTHER;
   rt_priority = 0;
   counter = 0;
   processor = 0; /* No previous CPU */
   cpus_runnable = ~0UL;
   cpus_allowed = ~0UL;
   need_resched = 0;
   mm = mm_count++;
   active_mm = mm_count++;
   quantum = 0;
   /* Put in next most CPU-bound queue */
   run_queue = -1;
   /* Dont relocate before it has been active for a whole scheduling cycle */
   moved = 1;
   /* Reset statistics counters */
   life_time = 0;
   cpu_time = 0;
   io_time = 0;
   wait_cpu = 0;
   wait_io = 0;

   this->pid = pid;
   this->grp = grp;
   this->nice = nice;
   this->policy = policy;
   this->run_prob = run_prob;

   io_start=-1, io_length=-1;
   cpu_start=-1, cpu_length=-1;
   sched_stamp=-1;
   util_add=0;

}

Task::~Task()
{
   //delete all stored properties.
   /*list<TaskProp*>::iterator iter;
   for (iter = prop_list.begin(); iter != prop_list.end(); iter++)
   delete (*iter);*/

}
/* Construct a task from a string */
Task* Task::read(char* line)
{
   int pid, grp, nice, policy, run_prob;

   /* Default run value (run is optional)*/
   run_prob = 0;
   if (sscanf(line, "pid:%d, grp:%d, nice:%d, policy:%d, run:%d", &pid, &grp
        , &nice, &policy, &run_prob) >= 4)
      return new Task(pid,grp,nice,policy,run_prob);
   else
      return NULL;
}

/* Printout all information */
void Task::print(int tick)
{
   printf("tick:%d pid:%d grp:%d nice:%d policy:%d processor:%d "
        "rt_priority:%d counter:%d state:%d "
        "cpus_runnable:%x cpus_allowed:%x ",
        tick, pid, grp, nice, policy, processor,
        rt_priority, counter, state,
        cpus_runnable, cpus_allowed);

   /* Print stats */
   float cpuiocpu=1.0*cpu_time/(wait_io+io_time+cpu_time);
   float test=1.0*cpu_time/(io_time+cpu_time);
   float wait=1.0*wait_cpu/(wait_cpu+wait_io);
   float cpu_eff=1.0*cpu_time/(cpu_time+wait_cpu);
   float io_eff=1.0*io_time/(io_time+wait_io);
   int real_time=cpu_time+io_time;

   printf("life_time:%d real_time:%d "
        "cpu_time:%d io_time:%d wait_cpu:%d wait_io:%d "
        "cpuiocpu:%f cpu_efficiency:%f io_efficiency:%f "
        "task_char:%f wait:%f ",
        life_time, real_time,
        cpu_time, io_time, wait_cpu, wait_io,
        cpuiocpu, cpu_eff, io_eff,
        test, wait);

   list<TaskProp*>::iterator iter;
   for (iter = prop_list.begin(); iter != prop_list.end(); iter++)
   {
      (*iter)->print(tick);
      printf(" ");
   }
   printf("\n");

}

void Task::set_state(enum task_state state)
{
   /* Break if this is really a state change */
   if (state == this->state) return;
   /* Set the new state */
   this->state = state;

   /* let all properties know about this change.*/
   list<TaskProp*>::iterator iter;
   for (iter = prop_list.begin(); iter != prop_list.end(); iter++)
   {
      (*iter)->statechange(state, this);
   }
}

/* Do one tick in the system. */
/* @param: none */
/* @returns: true if resched should be called */

int Task::tick()
{
   /* Update the statistics counters */
   switch (state)
   {
   case TASK_RUNNING:
      life_time++;
      if (task_has_cpu(this))
         cpu_time++;
      else
         wait_cpu++;
      break;
   case TASK_SUSPENDED:
      life_time++;
      if (task_has_io(this))
         io_time++;
      else
         wait_io++;
      break;
   default:
      break;
   }

   list<TaskProp*>::iterator iter;
   for (iter = prop_list.begin();iter != prop_list.end(); iter++)
   {
      if ((*iter)->tick(this))
         need_resched = 1;
   }
   /* Done sending tick */
   return need_resched;
}

void Task::add_task_prop(TaskProp *task_prop)
{
   prop_list.push_back(task_prop);
}

/* Utility function */
int set_nice(int val)
{
   return nice(val);
}

int Task::run()
{
   fflush(stdout);
   struct sched_param p;
   int i, pid, res;
   /* Dont run if prob < 0 */
   if (run_prob < 0)
      return 0;
   /* Spawn a new process */
   pid = fork();

   if (pid)
      return pid;

   p.sched_priority = rt_priority;
   res = sched_setscheduler(getpid(), policy, &p);
   if (res)
      fprintf(stderr, "Unable to set scheduler for pid:%d\n",getpid());
   res = set_nice(nice);
   if (res)
      fprintf(stderr, "Unable to set nice level for pid:%d\n",getpid());

   /* Set process properties */
   /* Find the correct process */
   list<TaskProp*>::iterator iter;
   for (iter = prop_list.begin(), i = 0;
       iter != prop_list.end() && i < run_prob; iter++,i++);

   /* Run the process */
   (*iter)->run();
   return 0;
}
```

```
/******* Utility functions *********/

/* Use more n cycles */
/* Just remember to compile with -O2 */
void cputime(int cycles)
{
   for (int i=0;i<cycles/2;i++);
}
```

task.h

```
#ifndef __TASK_H__
#define __TASK_H__

#include <list>
using namespace std;

#define SCHED_YIELD 0x10
enum task_state
{
   TASK_SUSPENDED,
   TASK_RUNNING,
   TASK_STOPPED
};

/* Forward declaration */
class Task;
class TaskProb;

/* Pure virtual class TaskProb */
class TaskProp
{
 public:
   virtual void statechange(enum task_state state, Task *task) = 0;
/* Allow callback - a property can change the state of a task */
   virtual int tick(Task *task) = 0;

   virtual void print(int tick) = 0;

   /* Start a process using fork, and return the process id. */
   /* This is used when testing processes on a live kernel */
   /* This will not work if the process has more than one property. */
   virtual void run() = 0;
};

/* General class for tasks */
class Task
{
 public:
   /* The state of the task */
   enum task_state state;

   int pid;
   /* thread group */
```

```
int grp;
/* nice level (-20 -> 20) */
int nice;

/* List properties */
list<TaskProp*> prop_list;

/* Sched */
int policy;

int rt_priority;

/* The CPU the process was running on. */
int processor;

/* Bitmask for runnable cpu’s. ˜0UL means that the task is running.
 */
unsigned int cpus_runnable;

/* Bitmask for allowd CPU’s. Default ˜0UL */
unsigned int cpus_allowed;

/* Has this task used its quantum? */
int need_resched;

/* Quantum (dynamic priority) */
int counter;

/* Temporary quantum */
int quantum;

/* Runqueue (should be a pointer to the actual struct, instead of index)*/
int run_queue;

/* Indication weather the proces has been moved to another runqueu
   within this scheduling cycle */
int moved;

/* Memory */
int mm, active_mm;

/* On real testing, the n’th process is run. */
int run_prob;

/* Statistics */
int life_time;
int cpu_time;
int io_time;
int wait_cpu;
int wait_io;

/* Two Level scheduler */
int io_start, io_length;
int cpu_start, cpu_length;
int sched_stamp;
int util_add;
```

```
   Task(int pid, int grp, int nice, int policy, int run);
   ~Task();
   /* the change is the state cascades to all properties. */
   void set_state(enum task_state state);
   void add_task_prop(TaskProp *task_prop);
   int tick();
   void print(int tick);
   static Task* read(char* line);

   /* Run the tasks on the CPU's */
   int run();
};

/* use the cpu for n cycles*/
extern void cputime(int cycles);

#endif /* __TASK_H__ */
```