Flexible scheduling mechanisms in L4

Supervisor: A/Prof. Gernot Heiser

Assessor: Dr. Manuel Chakravarty

A Thesis submitted to the School of Computer Science and Engineering of The University of New South Wales in partial fulfillment of the requirements for the degree of Bachelor of Engineering (Computer Engineering)

Simon Winwood (2210220)

November 2000

Abstract

This thesis presents the design, implementation, and evaluation of a new scheduling mechanism for the L4 μ-kernel, running on the Alpha 21264 processor.

This design concentrates on hierarchical resource management, by which an application is free to manage its own resources, enabling it to take advantage of domain specific knowledge.

The final design, *recursive scheduling*, extends easily to multi-processor systems, and benchmarking shows that the goals of the design were met successfully.

Acknowledgements

I would like to thank all those people who have through their support enabled me to complete this thesis. Firstly, I would like to thank my supervisor Gernot Heiser for his support and guidance throughout the course of this thesis, and Daniel Potts for helping me with the mysteries of the Alpha implementation of L4, and allowing me to preempt his cubicle (more than) occasionally.

I would also like to thank the many Keg people who helped directly or indirectly with this thesis, especially Luke Deller, Alan Au and Adam Wiggins for always giving me someone to laugh at.

I would like to thank my girlfriend, Camille Scaysbrook, for her continued love and support and especially for her patience.

Finally, I would like to thank my parents for the various things that made doing this thesis possible.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Originally, the main aim of this thesis was to design and implement a scalable scheduler for L4/Linux, a port of the Linux operating system to the L4 μ-kernel. In designing this scheduler, I noticed that the scheduling abstractions provided by L4 were insufficient for a general purpose scheduler such as Linux's, and several other scheduling schemes that I thought to experiment with.

The direction of this thesis then turned to the scheduling mechanisms exported by L4 itself.

1.2 Goals

The aim of this thesis is to design, implement, and evaluate a secure, flexible, and high performance scheduler for the L4 μ-kernel.

Security The main design goal of L4 is a minimalist kernel with respect to security, as espoused in [Lie95]. Thus, security should be the major concern of any scheduling mechanisms implemented by L4.

Flexibility As L4 is a microkernel, it must support a wide range of different OS personalities, sometimes concurrently. This means that any mechanism must allow different, competing scheduling policies at the same time.

Efficiency Any scheduling mechanism should have a low overhead, in both time and cache pollution. This is further reinforced by the high frequency of scheduling decisions.

The actual meaning for these goals is explored in Chapter 3

1.3 Thesis structure

This thesis is organised into the following chapters:

Introduction This chapter presents the motivation, goals, and structure of this thesis.

Background This chapter presents the necessary background required for the remainder of this thesis, including an overview of L4/Alpha and its current scheduling mechanisms.

Theory and Practice This chapter contains theory pertinent to this thesis, including scheduling support in other systems, and resource management theory applicable to this thesis.

Design This chapter presents the conceptual and concrete design, including reasoning behind design decisions.

Implementation This chapter presents the implementation of the scheduling mechanisms designed in the previous chapter in the Alpha implementation of L4.

Results This chapter presents the results from tests run on the modified μ-kernel.

Conclusions This chapter wraps up the thesis and presents the conclusions derived from the previous chapters.

Chapter 2

Background

This chapter introduces the background material behind this thesis, namely basic scheduling concepts, an overview of the Alpha architecture, an overview of the L4 μ-kernel, and the L4/Alpha implementation.

2.1 Basic scheduling concepts

As operating systems evolved from the simple batch monitors, the process management mechanisms of these operating systems has also evolved. With the introduction of interactive systems, a whole new set of problems have cropped up — how to multiplex the available CPU resources among the applications that require execution, in a fair and efficient manner.

For the purposes of this thesis, a *thread* is the basic unit of execution. A *process*, or *task* (the preferred term) is a collection of threads that share the same resources, essentially the same address space, or protection domain. This thesis will only deal with so called *kernel threads*, threads which the kernel is responsible for managing, as opposed to *user threads* which are usually multiplexed onto 1 or more kernel threads, and are scheduled by a user level library.

The selection of a thread, and the way in which that thread is allocated CPU time is called *scheduling*, and is the main focus of this thesis. The following are the desired characteristics of any interactive, or "online"[1] scheduler:

[1]While "real time" would be a more appropriate term for a system in which time is important, this has further connotations, described in Section 2.1.1

Responsiveness Responsiveness is the perceived reaction time between some event and the systems reaction to that event, for example, the amount of time that passes between a key press and the character being echoed on the screen.

This is one of the most important characteristics of an interactive scheduler, as it effects the users perception of the system. Note that different events have different responsiveness requirements — the time to process a keystroke should be much lower than that required to process a button click, for example.

Fairness Fairness determines the amount of processor time that a thread should be able to consume. In an interactive system, it is desirable that every user gets their fair share of the CPU. It is also desirable that low priority threads also get to execute; that is, the system should prevent *starvation*.

Utilisation Utilisation is a measure of the effectiveness of a scheduler in taking advantage of the available resources. An effective scheduler will minimise the total idle time of the system.

Throughput Throughput is the ability of the system to process jobs, usually measured in jobs per unit of time. While this can be effected by the length of jobs, the scheduler determines the ordering of the jobs, and hence the throughput rate.

The *dispatcher* is responsible for actually running the thread the scheduler selected, as shown in Figure 2.1.

In order to provide a responsive system, modern interactive systems have preemptive multi-tasking schedulers, meaning that threads get a certain amount of CPU time, called a *quanta*, and are then preempted so that another thread may execute.

The following are thread properties which effect the responsiveness of a system:

Priority The priority of a thread effects who can preempt it, and who it can preempt. If a thread of a higher priority becomes ready to execute,

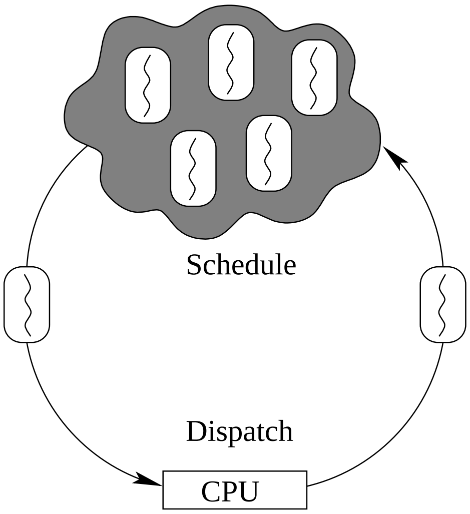

Figure 2.1: Roles of the scheduler and dispatcher

it will preempt the CPU from the currently running thread and start executing.

The priority effects the *latency* (time between a thread becoming ready, and when it starts execution) that a thread will experience if it becomes ready, and hence how responsive it is. The number of thread at a specific priority can also effect the latency which a thread experiences, as well as the time spent waiting for the CPU after preemption.

In a system with fixed priorities, starvation can also be an issue, as a low priority thread will not run if there is always a thread of a higher priority runnable.

Quantum size The quantum size of a thread determines the amount of time for which it may run without being preempted by a thread of the same priority, and hence how much work it may get done.

The quantum size also effects the responsiveness of a system — a large quantum will give a low number of threads executing per unit of time, and hence some applications will behave sluggishly. A small quantum will allow a larger number of threads to execute per unit of time, but may result in a high overhead as threads interfere with each other in the TLB and caches.

Execution rate The execution rate of a thread is the (relative) number of times it is given the CPU per unit of time. A thread that has a high execution rate will have high responsiveness, however this will impact on the responsiveness of other threads.

Note that each of these attributes (priority, quantum size, and execution rate) is orthogonal. A thread that has a high priority need not have a large quantum, or a high execution rate — this might be beneficial for an interactive thread, in that it is responsive, but cannot dominate the system. Some operating systems, such as Linux, tend to merge the three attributes into "priority" so that higher priority threads get more CPU time and are more responsive, but the distinction is important.

2.1.1 Real-Time scheduling

There is a class of applications which are time sensitive, called *real-time*. These applications require special scheduling support in order to achieve their aims. An example of a real-time application is the task within a heart monitor that monitors the state of the patient. If the monitor does not detect and fix any abnormalities within a certain period, the patient may die.

This is a *hard real-time* system — if the deadline is not met (the patient may need to be shocked less than 5 seconds after the attack, for example) then there may be severe consequences.

An example of a *soft real-time* system is video decoding. Each frame must be displayed a certain time after the last image, with minimal variation. If a frame is missed, however, then the only consequence may be slight jitter in the displayed video.

Although real-time scheduling is not a major goal of this thesis, any scheduling mechanism in L4 needs to support real-time applications, so the general requirements of a real-time application follows:

Deterministic Resources An application may require a certain amount of resources to be available to it at any one time, for example it may require 5ms per 50ms to meet its requirements.

Bounded Latency An application may require a bounded response time to an external event, for example an interrupt.

Deadlines An application may need to begin execution by a certain point in time, or may be required to complete by a certain point in time.

2.2 The Alpha Architecture

The Alpha Architecture [Com99a] is a 64 bit load-store RISC architecture designed with the following goals in mind:

- Longevity
- High Performance
- Scalability
- Adaptability

2.2.1 PAL code

Adaptability requires that multiple operating systems can be ported to the architecture with reltive ease. The way in which this is achieved is through the *Priveleged Architecture Library (PAL)*. PALcode is a set of abstractions such that the underlying complexity and peculiarities of the hardware are transparent to the operating system running on top of it.

PALcode is responsible for handling interrupts, exceptions, page faults, and system initialisation. PALcode is also subject to a number of restrictions with respect to code scheduling.

PALcode operates in the following environment:

- Instruction stream mapping disabled.
- Data stream mapping enabled.
- Access to all memory and hardware.
- Interrupts are disabled.

2.2.2 The Alpha 21264 Processor

The Alpha 21264 processor [Com99b] is the third major revision of the Alpha architecture commercially available. The features of this processor are in line with the high performance applications that the architecture is expected to run:

- The ability to issue 4 instructions per clock cycle, and execute 4 instructions out of order each cycle.
- Separate 128 entry data and instruction translation buffers.
- Separate 64Kb L1 data and instruction caches.
- 44 or 48 bit virtual addresses and 44 bit physical addresses.

The following are the features of the 21264 which effect this thesis:

- Branch mis-predict costs 7 cycles.
- In PAL mode, all conditional branches are mis-predicted.
- The memory access latencies between the processor and the L1 cache, the processor and the L2 cache, and the processor and the L3 cache are 3, 16, and 80 cycles respectively.

  This means that loads should be minimised, and where possible, group commonly used date in the same cache line.
- The processor executes instructions out of order.

2.3 The L4 μ-kernel

A μ-kernel is a set of abstractions that execute in a processors privileged mode. The goal of a μ-kernel is to provide a minimal kernel upon which operating system personalities can be constructed in a flexible and efficient manner.

So called first generation μ-kernel's were adaptations of existing operating systems (e.g. Mach) and attempted to provide a semi-complete operating system, containing such elements as device drivers and paged virtual memory.

The L4 μ-kernel is a second generation μ-kernel developed originally at GMD. L4 provides minimal functionality with respect to security: a mechanism is only included in L4 if placing it in user space would compromise security.

Implementations of L4 exist, or are currently under development, for the following architectures: x86[Lie96], MIPS [EHL97], Alpha[Pot99, Sch96], PowerPC, and StrongARM [Wig99].

L4 provides the following abstractions:

Address spaces An address space is a set of mappings from *Virtual Memory* to physical memory. L4 exports the entire physical address space to user level, including device memory.

Initially, all physical memory is owned by σ_0, the initial address space. A pager running in this address space maps each page in its address space on a first-come first-serve basis.

Address spaces in L4 are constructed recursively by *user level pagers*. These pagers are responsible for exporting portions of their address space to client address spaces, via mapping operations.

Threads Threads in L4 are the basic unit of execution and scheduling. Each thread has a unique thread id, and belongs to one and only one address space. Threads may not move between address spaces in the current implementation of L4.

The combination of an address space and the group of threads that run within it is called a *task*, and is the basic unit of protection in L4.

To create another task, a task needs a right to that task, Initially all task rights belong to σ_0. At initialisation it will grant all tasks to the initial resource manager[2]. Tasks may be granted to other tasks, conferring upon them the ability to activate that task.

Inter-Process Communication L4 provides highly efficient inter- and intra-task communication facilities in the form of IPC. IPC in L4 is strictly synchronous, and can be used to transfer data directly (in registers) or

[2]This is somewhat implementation dependent; in the MIPS implementation, tasks are given out in a similar fashion to memory, in a first-come first-served fashion

by reference (copied via the kernel). In both cases an IPC can only take place if both parties are ready.

IPC is used in L4 to deliver pagefaults, interrupts and exceptions. If a thread sends an IPC to another thread, that thread inherits its remaining timeslice and priority.

2.3.1 Scheduling within L4

Given the emphasis of this thesis, I will only provide a brief overview of L4's scheduling abstractions. These will be further investigated in later chapters when sufficient background information and theory have been introduced.

L4 provides a hard-priority round-robin scheduler with variable timeslice length (as in Figure 2.2).

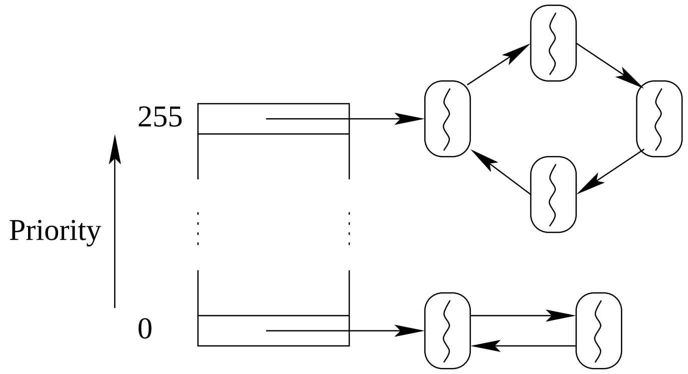

Figure 2.2: Prioritised round robin scheduler in L4

A thread has the following scheduling parameters associated with it:

Current Priority This determines whether a thread will be scheduled by L4 if it is runnable. If there are no threads with a higher (numerically larger) priority, the thread may be scheduled.

Timeslice length This determines the amount of time for which a thread may execute without begin preempted by the kernel, assuming no threads of a higher priority become ready during this period of time.

Maximum Controlled Priority (MCP) This determines whether a thread may modify the scheduling parameters of another thread. A thread A may only modify the parameters of a thread B if $MCP_A \geq MCP_B$,

External Preemptor This determines a threads behaviour when it is preempted (either via timeslice expiration or if a higher priority thread becomes runnable). If this is valid, the thread will execute an RPC to its external preemptor to notify them of the preemption.

A thread's scheduling parameters may be modified using L4's `thread_schedule()` system call, constrained by the MCP restrictions mentioned above.

In addition to the above mechanisms, a thread may explicitly schedule another thread, via either IPC or L4's `thread_switch()` system call. In both cases the destination thread receives the remainder of the threads timeslice and priority. Along with the external preemptor mechanism, these form the mechanisms by which a thread may control another threads scheduling behaviour.

2.3.2 The L4/Alpha implementation

The Alpha implementation of the L4 μ-kernel runs on every available generation of the Alpha architecture. It is currently the highest performing L4 implementation, with a minimum IPC time of 45 cycles[LES+97].

Originally developed by Sebastian Schönberg and Volkmar Uhlig, the L4/Alpha μ-kernel has been ported to the 21264 Alpha processor by Daniel Potts and is currently being extended to support SMP Alpha systems.

As this thesis involves considerable modifications to the scheduler implementation within the L4/Alpha μ-kernel, this section will detail the current scheduler implementation and other relevant mechanisms. For a more general discussion of the L4/Alpha implementation, see [Pot99] and [Sch96].

Overall design

The L4/Alpha μ-kernel is written completely in assembler, and is compiled using a modified assembler from Digital Equipment Corporation[3]. The majority of the kernel runs in PAL mode; L4 runs in kernel mode if full access to

[3] Now owned by Compaq Computer Corporation

user registers is required (some registers are shadowed in PAL mode) or code execution time is significant and should be interruptible (fpage mapping, long IPC, etc.)

Approximately 40% of this code is CPU specific, and of that approximately 10% is platform specific; this includes low-level interrupt handling, memory sizing, and platform initialisation.

There can be up to 1024 active tasks in the system at any one time, with 256 threads per task.

Internally, L4 maintains a number of lists, with each thread belonging to lists as follows:

Busy Queue This global queue contains *at least* all runnable threads in the system. It may contain threads which are not runnable due to *lazy scheduling*[Lie93], whereby a thread is left in the busy queue but marked as not runnable, if it needs to be blocked during IPC (i.e. if the partner is not ready). The scheduler is responsible for removing a thread from the busy queue if it is not runnable.

Interrupted queue This global queue contains threads that have been preempted prematurely (due to an interrupt or a thread of a higher priority becoming runnable). It is not used in the current implementation.

Soon wakeup queue This global queue contains all threads who have blocked on IPC with a valid timeout (not zero or infinite), less than a specific value. In the current implementation, this is an ordered list containing *all* threads waiting on a timeout.

Late wakeup queue This global queue is similar to the soon wakeup queue except that it contains thread who are to be woken up at some point in the future greater than that required to be in the soon wakeup queue. This is currently unused as all threads go into the soon wakeup queue.

Polling queue This per-thread queue contains all threads waiting to send to the owner of this queue. A thread is inserted into this queue if it tries to send to the queue's owner and the owner isn't ready to receive.

Present queue This global queue contains all active threads in the system.

Scheduling mechanisms

The current implementation of the L4/Alpha μ-kernel implements a preemptive fixed-timeslice fixed-priority round-robin scheduler. The Real Time Clock generates a timer interrupt approximately twice every millisecond[4]. There is one sorted timeout queue, which is checked every 4 ticks, or 2ms. A thread is rescheduled every 32 ticks, or 16ms.

The following are the actions taken on each timer interrupt:

1. Internal L4 time datastructures are updated and the current thread has its accounting information updated.
2. If the soon wakeup queue hasn't been parsed for 4 ticks, it is parsed, and if a thread is woken up the scheduler is invoked.
3. If 32 ticks have passed since the last reschedule[5], the scheduler is invoked.

The soon wakeup queue is searched even if there are no threads that need to be woken up.

The following are the actions taken on each scheduler invocation:

1. The current thread context is saved on the stack.
2. The priority list is searched from highest priority until a runnable thread is found. If a thread that is in the busy queue but isn't runnable is found, it is removed.
3. The current context is switched to the target threads; the stack pointer (SP) and program counter (PC) are saved to the TCB of the current thread, and the SP and PC of the new thread are loaded.
4. The PC of the new thread is branched to.

This is shown in Figure 2.3.

A scheduling decision needs to be made in the following circumstances:

- The current timeslice expires.

[4]This may be modified for Alpha CPUs with higher clockspeeds such as the 21264 to give better responsiveness

[5]Actually, if the lower 5 bits of the tick counter are clear

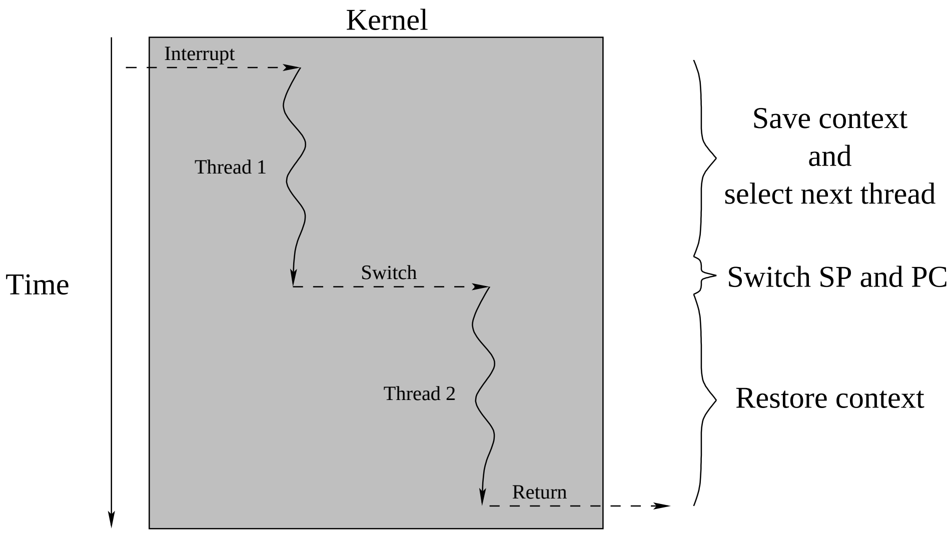

Figure 2.3: Rescheduling behaviour

- An external interrupt.
- A thread yields the processor without specifying the destination.

2.4 Discussion

While the current implementation of L4/Alpha provides a reasonably stable environment, the scheduler is does not conform to the current API and provides only the bare minimum scheduling mechanisms.

Chapter 3

Theory and Practice

This chapter describes the theoretical basis of this thesis: resource management, multiple processor support. The scheduling mechanisms implemented in other systems are also presented.

This chapter also extends the goals given in the previous chapter to actual design goals, in terms of the presented theory.

3.1 Multiple CPU scheduling issues

Systems with more than one CPU are becoming cheaper, and hence, more common. Along with lower cost, these systems also have increasingly larger numbers of general purpose CPUs. For example, it is now possible to purchase a 32-way ccNUMA Alpha machine from Compaq Computer Corporation.

In a multiple CPU system, the organisation of the run-queue is of major importance. The two alternatives are a single global run queue, or a per-processor run queue.

A single run queue that is shared amongst multiple processors needs to have concurrency mechanisms to keep it consistent. This is simpler than per-processor run-queues, however it can lead to poor performance due to cache affinity information being underutilised.

Per-processor run queues introduce the problem of load balancing, and the meaning of priority — in a system with multiple run queues, a thread may execute while a higher priority thread is waiting to execute on another processor.

Cache affinity information if becoming increasingly important — [CDV+94] notes that a scheduler which takes advantage of this information can get a large improvement over a standard UNIX scheduler. This is further re-enforced in [SL93], looking also at the problems that may arise due to excessive bus traffic due to reloading cache working sets. Another problem raised is that of load balancing — there is a confilicting requirement of throughput and resource utilisation; a scheduler needs to ensure that there are no idle processors when the system is loaded, however it is also important that the scheduler minimises overhead due to migration.

3.2 Resource isolation and reservation

The basic function of an operating system is to manage the available resources, or, in a μ-kernel system, to provide mechanisms with which an operating system personality can safely multiplex the available resources among its clients.

As systems are becoming multi-functional, it is becoming increasingly important to isolate entities from each other, whether they are users, applications, or administrative groups.

For example, with the popularity of languages such as Java, it is now common for untrusted content to be executed on a shared system. To allow such applications to utilise arbitrary amounts of shared resources, such as CPU time, memory, etc. is unacceptable, as this will impact unfavourably on other entities within the system.

As another example, in a system which acts as a host to multiple services (virtual hosting of web sites, for example), if a minimum quality of service can be guaranteed, then those services can be charged a higher rate than a host offering only best-effort service.

In a multiple user system, such as that found in many universities, a user can mount a Denial of Service (DoS) attack on the other users on the system, intentionally or unintentionally, using a fork bomb, for example, where as many processes as possible are started up as fast as possible (essentially, each process continuously starts up copies of itself).

One method for isolating entities is proposed in [VGR98]. This method uses Software Performance Units (SPU) as the unit of isolation. A thread that belongs to one SPU will not be effected by the resource usage of a thread in another SPU (although thread is no protection between threads in the same SPU).

In this system a SPU is allocated an integer number of CPUs, and then a portion of another CPU as appropriate. Within each SPU, the normal Irix (the prototype platform) scheduler is used to schedule threads.

An important attribute of this system is that of sharing — an SPU may elect to share its idle resources with other SPUs within the system. This allows full system utilisation, but also means that the donator may be disadvantaged: the system ensures that a SPU gets the resources it is entitled to by preempting them from the SPU to which they were donated, but on SMP systems this may result in overhead due to cache and TLB pollution by the preempted SPU.

3.2.1 Proportional Share Scheduling

The Share fair share scheduler as described in [KL88] recognises that the discrimination unit of resource allocation is the user, not the process. Each user is given a certain number of shares, and processes are scheduled according to their owners remaining number of shares. At each reschedule, the owner of the current thread is charged according to the amount of CPU time that it has used.

The Share scheduler also allowed users to be grouped, so that in a system owned by multiple parties, each party was given usage of the machine according to the amount they owned. Users belonging to those groups are then allocated resources from those allocated to the group.

3.2.2 Lottery scheduling

Lottery scheduling is a method for achieving proportional allocation of CPU time, as described in [WW94]. The basic idea is that each thread in the system holds a number of tickets, and at each scheduling event, a random number is chosen. The thread that holds that ticket is then chosen to execute.

This is implemented by using a pseudo-random number generator and searching through each thread, until the thread that owns that ticket is found

(Note that while this is an $O(n)$ implementation, using an intelligent datastructure such as a tree, this can be reduced to $O(\ln n)$). This algorithm is *probabilistically fair*, as each thread has a $\frac{t}{T}$ chance of being chosen, where t is the number of tickets it holds, and T is the total number of tickets.

This system also allows hierarchical scheduling, where each group of threads has a separate currency. Tickets issued in one currency, are converted into the base currency. This is shown in Figure 3.1, with the number of base tickets shown in brackets.

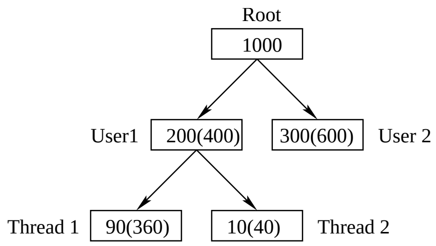

Figure 3.1: Lottery scheduling

This scheme has been implemented in FreeBSD[PMG99], with slight modifications. The modifications introduced in this paper are such that `nice` semantics can be achieved, essentially by modifying the number of (user) tickets that a process receives.

Although this system allows hierarchical scheduling and proportional share scheduling, it only gives probabilistic fairness, and hence cannot be used for real time threads where latency is an issue. Another possible performance issue with this scheme is that each user currency needs to be converted to the base currency, a non-trivial operation.

3.3 Application controlled scheduling behaviour

Systems are beginning to support a broader range of applications, especially soft real-time (for example video and audio decoding in software is now commonplace, as are soft-modems and other latency critical software-intensive devices), and no single scheduling policy meets the requirements of these applications, as discussed in [RSH00].

Supporting real-time applications and interactive applications in the same system requires multiple scheduling policies in an operating system, and hence some kind of scheduler extensibility. However, "normal" applications will also benefit from increased scheduling flexibility — for example, a web server can allocate a larger amount of time to high priority clients.

An operating system should enable an application to take advantage of domain specific knowledge; an application has a greater awareness of its resources requirements than does the operating system which is responsible for allocating those resources. An application knows which of its resources are critical and which can be reduced gracefully.

It is therefore sensible to expect an operating system to export these decisions to the application. One method for allowing an application to modify OS scheduling policies is through kernel extensions, as with Vassal [CJ98].

The Vassal system employs Windows NT's extension mechanisms to add arbitrary scheduling policies to the kernel, implemented as device drivers. When the dispatcher needs to choose a thread, it queries each loaded scheduler until a runnable thread is found. If no policies have a runnable thread, the dispatcher consults the normal NT scheduler.

The Vassal system supports only one extra scheduling policy at any one time, as there are currently no mechanisms for choosing which order to query policies. Any extension to this system would need to include arbitration mechanisms so that a desired system-wide policy is enforceable.

At the other end of the spectrum is CPU Inheritance Scheduling, described in [FS96]. CPU Inheritance Scheduling achieves a given scheduling policy by a scheduler thread explicitly donating the CPU to a client thread. The kernel implements only a simple dispatcher, used for switching between threads, and the remainder of the scheduler is implemented in user space.

The advantage of such a scheme is that any thread can schedule any other thread, simply by donating the CPU to it. The kernel does not need to know about priorities or timeslice lengths — a thread runs until an external event (timer interrupt) preempts the CPU from it; when this happens, the owner of the interrupt will be given the CPU, and it is then free to donate it to some other thread.

A major disadvantage of this system is that interrupts have an implicit high priority. This means that high priority threads, such as real time threads, may get interrupted, even if this is against the design of that system. Additionally, each scheduling event requires multiple context switches, which is expensive.

3.4 Practice: Current scheduler implementations

3.4.1 Linux

Linux [Lin] is a freely available UNIX clone. Linux implements a fairly standard UNIX scheduling policy, which attempts to give interactive threades a low latency. An interactive thread is one which uses only a small portion of its allocated timeslice.

The scheduler is implemented using a global run queue, which is scanned on every scheduling event to calculate the 'goodness' of each thread. The goodness of a thread is related to the number of ticks it has remaining. When there are no threads with ticks remaining, the scheduler resets the ticks for each thread using a decayed usage scheme, based also on that threads nice value.

Linux has basic support for real-time applications; real-time threads are given a static priority, and can be designated as either First-In First-Out, or Round-Robin. A real-time thread will be scheduled over any interactive thread in the system. Note that a user needs superuser privileges to set a thread to be real-time.

While this scheduler is good for small, lightly loaded systems, as soon as the system grows beyond a single processor, or a large number of threads, the scheduling overhead becomes unacceptable (See [BH00] for an analysis of the linux scheduler under load.)

Linux offers basic per-thread resource isolation — a thread may have a CPU limit and is killed if it is exceeded.

3.4.2 The Mach μ-kernel

Mach is a first generation μ-kernel, developed at CMU. Mach provides a scheduling mechanism similar to that of Linux, as described in [Bla90]. The global scheduling datastructure is a table indexed by priority, with each entry containing a pointer to the next runnable thread at that priority.

The priorities are dynamic, in that a thread's usage directly effects that thread's priority; each thread is responsible for their own aging, with a thread searching through the run queue every few seconds to avoid starving low priority threads that haven't had a chance to modify their priority.

Although the principal scheduling data structure in Mach is the global run queue, each processor in the system also has a local run queue. This is to enable legacy UNIX compatibility code that has not been parallelised to execute, and to enable binding of threads to a specific set of processors.

Mach provides no resource management mechanisms, other that a high level processor-set based access mechanism. A user level server is responsible for allocating threads and processors to processor sets — a thread may only execute on a processor in the same processor set as itself.

3.4.3 L4 revisited

L4 provides only minimal scheduling mechanisms, however there is currently no mechanism to limit the influence of one thread on another. While the external preemptor mechanism allows a thread's scheduling behaviour to be controlled, there are no constraints on the preemptor to follow a system wide policy.

The *thread_switch* mechanism allows a similar hierarchical implementation to that presented in [FS96], however the priority inheritance aspects of this may interfere with the desired policy.

Mechanisms need to be added to L4 so that it can guarantee resource isolation to the systems built on top of it. Such mechanisms will form the focus of the next chapter.

3.5 Thesis goals revisited

Chapter 1 gave the goal of this thesis as the design of a secure, flexible, and efficient scheduling mechanism. With the above theory, the actual meaning of this can be expanded into design goals.

3.5.1 Security

An operating system is *secure* if entities within that system can only effect other entities within the constraints of the controlling security policy for that system. In a scheduling context, this means that an entity (user, application, thread, etc.) can only utilise as much CPU time as the systems policies allow, under the timing constraints of those policies.

For example, if a scheduling policy determines that thread A is to execute with higher priority than thread B, but only for 5ms per 100ms, then security has been violated if thread A runs while thread B is ready, or if thread B executes for more than its allocated 5ms per 100ms.

Thus, any scheduling mechanisms for L4 should allow the interaction between entities to be strictly controlled: it should export mechanisms to limit the amount of time for which a thread can execute, as well as the relative priority of that thread.

Note that this view of security is somewhat different to that posed by the confinement problem — in this context, security is related to the amount of access that a thread is given to a particular resource, in this case the CPU.

3.5.2 Flexibility

An operating system is *flexible* if it allows its clients to control their own resource utilisation. In a scheduling context, this means that an entity should be able to define its own scheduling policy: to define the relative priorities of the threads it is responsible for, as well as the allowed resource utilisation of each thread.

The security goal described in the previous section implies that an entity can only define its own scheduling policy within that defined by its controlling entity, and so forth.

For example, given the example in the previous section, but let thread A be replaced by entity A containing threads A_0 and A_1. A desirable policy might be that thread A_0 may execute whenever it is ready, while thread A_1 may execute if thread A_0 is blocked. Entity A should be able to allocate its time between its two threads, constrained by the 5ms originally given to it, such that the sum of execution time for threads A_0 and A_1 should be 5ms per 100ms at most.

Thus, any scheduling mechanisms for L4 should allow composition of scheduling policies: an entity should be able to multiplex its available resources in any fashion, constrained by the scheduling policy of its controlling entity.

3.5.3 Efficiency

An operating system is *efficient* if it achieves its goals with a minimum of resource utilisation. In a scheduling context, this means that entities should be able to make scheduling decisions only when necessary to enforce a given scheduling policy. In addition, the overhead imposed by the execution of these scheduling policies should be minimised.

Thus, any scheduling mechanisms for L4 should allow an application to control the granularity of its scheduling decisions, and should dispatch threads with minimal overhead.

Chapter 4

Design

This section presents the design of a scheduling mechanism for L4 that allows user controlled scheduling decisions in a flexible, secure and efficient manner.

4.1 Design Goals

As discussed in the previous chapter, the goals of this design are security, flexibility, and efficiency. For maximum flexibility, a hierarchical scheme was chosen. As to why this is desirable (apart from the reasons given in the previous chapter), consider the system in Figure 4.1.

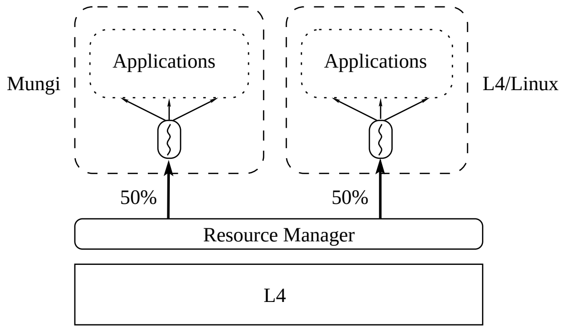

Figure 4.1: Linux and Mungi running concurrently

Each operating system personality should be able to schedule its own threads, and the resource manager should be able to schedule each personality (in this case such that each personality receives 50% each).

More generally, L4 systems are usually server based systems, and each server may consist of multiple threads, so it would be desirable for each server to be able to allocate its resources amongst those threads according to its own policy.

4.2 Conceptual design

Given that each thread in the system should be able to grant its resources to other threads, and only up to the amount which it had granted, the desired effect is similar to that provided by lottery scheduling. Lottery scheduling implies too much policy, however, and is not a flexible enough scheduling mechanism for a μ-kernel such as L4.

A naive approach to this problem, as in Figure 4.2, would be to give a large amount of time to the first top-level scheduler thread, who is then responsible for splitting this up among its threads, and so forth. After that quantum has been used up, the next top-level scheduler thread receives the next quanta, and it then divides the time amongst its threads, and so forth.

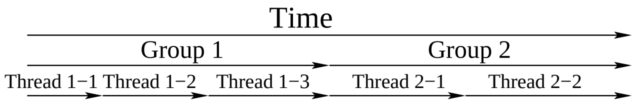

Figure 4.2: Naive approach to a hierarchical scheduler

While this approach is simple to implement, and has a low overhead due to context switches, it has a fundamental flaw: the maximum latency experienced by a thread can be very large. If a thread wakes up just after its groups quantum expires, it will have to wait until its group gets the CPU again, even if it is a high priority thread.

As a second approach, consider a system in which a thread is granted resource rights, called *potential quanta*, such that it can only execute for up to its potential quanta, in multiple timeslices. Given a method for threads to grant portions of their potential quanta to other threads, this gives the desired properties: a thread can only utilise a certain amount of the CPU, and the thread can pass on this resource right to other threads.

The problem then becomes how to achieve the granting of resource rights. Conceptually, every time a thread is executing, the threads that gave it its potential quanta should also be considered to be executing. A simple way to achieve this would be to consider granting of resource rights to be equivalent to a transfer — the grantor's potential quanta is decremented by the amount donated, and the grantee's potential quanta is incremented by the same amount.

Although this method is again straightforward, there are still problems. If a thread can only donate up to its potential quanta, the amount it donates is directly coupled to the amount it receives. A larger problem exists with revocation. The thread has been granted rights to future resource usage, but the grantor may have a need to revoke those rights (for example, if a thread of a higher priority requires service), but if the thread has already granted those rights to other threads, the original grantor cannot revoke them, unless a record of every transaction is kept. Obviously, this is undesirable — a thread should be able to revoke the resources it has given another thread at any time (an extreme case is if a thread needs to be suspended).

These problems arise due the charging method; a thread is charged when it grants resource rights, not when those rights are exercised. What is really required is for a thread to be charged when any thread it has allocated potential quanta executes, or any thread to whom it has further granted those rights.

To achieve this, every thread is associated with one and only one scheduler thread, which is responsible for allocating its potential quanta. As shown in Figure 4.3, threads form a tree; a thread is a parent of another thread if it schedules it directly. The root of the tree is τ_0 and has infinite potential quanta.

Whenever a thread is scheduled, its potential quanta is charged for its timeslice. Its scheduler is also charged, and its scheduler's scheduler, up to τ_0. This implies that a thread that is not a descendant of τ_0 will not be scheduled, as it does not have access to CPU time. If a thread needs to revoke the access rights of a child thread, it need only change the value of the child's potential quanta.

This mechanism is called *recursive scheduling.*

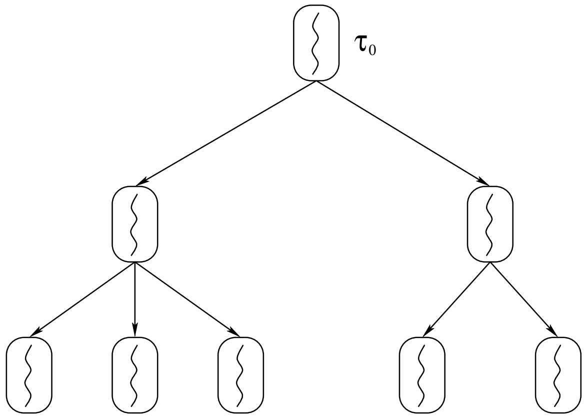

Figure 4.3: Scheduling hierarchy

There is, however, another issue: a thread can only execute as long as it has non-zero potential quanta. What should happen when a thread attempts to execute with a zero potential quanta? There are 2 possible options: remove it from the run queue until its scheduler notices and allocates it more potential quanta, or notify its scheduler in some fashion.

While the first option is simpler, the second option is attractive in that the scheduler thread needs some way of determining when a thread has exceeded its allocated potential quanta. The only other way in which it could ensure this is by polling on the threads remaining potential quanta, which is unattractive at least.

While this solution has the desired attributes of hierarchical resource management, it still does not specify the manner in which threads are scheduled by L4, or the manner in which a scheduling policy may be enforced — that is the focus of Section 4.3.

4.3 Concrete design

The low level scheduler within L4 is implemented as a prioritised round-robin scheduler with variable timeslice length. This is to allow backwards compatibility with previous L4 applications, and allows schedulers to give different client threads different priorities depending on their latency requirements.

Each thread in the system is associated with a scheduler thread, which is responsible for the amount of time that the thread receives and the way in which the thread can use it. This is achieved by manipulating the following thread attributes:

Priority This determines whether a thread will be scheduled if it is runnable. A thread will only be scheduled if all runnable threads in the system have a priority less than or equal to its own. Threads with the same priority are scheduled in a round-robin fashion.

Timeslice This is the amount of time for which a thread may execute without being preempted by threads at the same priority. The thread may be preempted at any time by threads of a higher priority.

Potential Quanta This is the total amount of time for which a thread may execute without invoking its scheduler. This may span multiple timeslices.

A scheduler is responsible for determining the relative amounts of time that its children (all those threads which it is directly responsible for) receive. This is achieved by setting the potential quanta of each of its children. By limiting the time that its children receive, a scheduler limits the amount of time that all its descendants receive.

Because a scheduler thread's grandchildren and their descendants are limited by the potential quanta that the thread give its child (their parent), the scheduler doesn't need to know about the descendants of its children in order to isolate each of its children from the others. This allows schedulers to guarantee a minimum amount of service to its children.

4.3.1 Security

A thread will violate the security of the system if either of the following occur[1]:

1. An entity receives more cumulative execution time than has been allocated to it according to the policy of its controlling entity, and all that entities ancestors.

[1]This is not an exhaustive list of the ways in which a thread can compromise system security, but it addresses the main issues raised by the inclusion of the proposed mechanisms.

2. L4 permits an entity to charge the potential quanta of a thread which is not related to the thread that set that allotment.

3. The scheduling parameters of a thread are allowed to be given arbitrary values.

A thread may only execute if it has enough potential quanta, and all its ancestors also have sufficient potential quanta. This means that the first violation cannot occur.

As a thread modifies the potential quanta of its scheduler, it is possible for a denial of service attack to take place if a thread is associated with a scheduler without that schedulers knowledge. This will only happen if a thread's scheduler is changed to the target thread, and the thread has a non-zero potential quanta.

The problem is addressed by setting a thread's potential quanta to zero whenever its scheduler is changed. The target scheduler will receive a timefault IPC from the thread when the thread is next scheduled — the thread will not be dispatched until its scheduler gives it more potential quanta.

With this constraint, a thread can only modify another threads potential quanta with that threads permission, and so the second violation cannot occur.

The third violation requires a limit to the maximum priority and timeslice that a thread may be given — the priority and timeslice length of a thread needs to be bounded as they effect the latency of other threads in the system. To do this, the following attributes are associated with a thread:

Maximum Schedulable Priority (MSP) This determines the extent to which the thread can schedule other threads (including itself). A thread can only set the priority of a thread it schedules to (strictly) less than its MSP. Hence, a thread with a MSP of 0 cannot schedule another thread.

Maximum Schedulable Timeslice This is similar to the MSP in that a thread can only set the timeslice of the threads it schedules (including itself) to less than or equal to its maximum schedulable timeslice.

Another possible security problem is if a thread's scheduler is changed and the thread's attributes are not. As an example of why this is a problem, consider a system in which there are 2 schedulers, a low priority, large timeslice scheduler, and a high priority, small timeslice scheduler. If a thread with a large timeslice is moved from the low priority scheduler to the high priority scheduler, then it is possible for it to have a high priority and a large timeslice. This is clearly a security violation.

The prevent this, whenever a threads scheduler is changed, the thread's attributes are modified so that they are within that allowed to the new scheduler.

Given the above restrictions, there is no reason why a thread, or any thread in its task, cannot modify its own scheduling attributes (not including potential quanta, of course), constrained by the normal thread scheduling rules.

In summary, a thread may be scheduled if and only if the following conditions are met:

1. The thread is runnable.
2. The thread's priority is greater than or equal to all other runnable threads.
3. The thread has sufficient potential quanta.
4. Each ancestor scheduler also has sufficient potential quanta.
5. τ_0 is an ancestor of the thread.

To prevent malicious charging of time to another thread, whenever a thread's scheduler is changed, the threads potential quanta is set to zero.

To prevent an illegal attribute state, a threads scheduling attributes are constrained by those of its scheduler at all times.

4.3.2 SMP and NUMA

For performance and scalability reasons, each CPU will have its own run queue(s), meaning that priority is only respected on that CPU.

A problem with systems with multiple processors is that of access control – which thread may execute on which processor. The current proposal [Pot99] is for each task to be associated with a processor set; a thread may only execute on a given processor if it is in that thread's processor set. In the design detailed in this thesis, a threads scheduler would grant the thread a subset of its available processors, and this set would be checked whenever a thread wanted to migrate between processors.

A problem exists here with resource revocation: if a thread wishes to revoke access to a processor for a child thread, it needs to also revoke access for all that threads children. This implies that the scheduler stores explicitly (in the kernel) the threads it schedules, and that revocation would need to recursively traverse this scheduling tree.

A closer look at this design, however, shows that processor sets are not necessary, with some extra constraints. If a thread allocates another threads potential quanta, then it is conferring upon it the right to execute. Restricting potential quanta to the processor on which it was allocated has the same effect as processor sets — a processor's τ_0 determines which threads get time on their CPU, and hence whether those threads and their descendants are allowed to execute.

At any time a processor may be preempted from a thread and its descendants by its scheduler, either by migrating it to another CPU or by setting its potential quanta to zero. All the child threads of that thread will also have that processor preempted from them, as their scheduler is no longer runnable on that processor.

As described in Section 4.3.1, whenever a thread's scheduler is changed, the thread's potential quanta is set to zero. This means that a thread may only be scheduled on a processor if all of its ancestors belong to that processor.

4.3.3 Implementation details

Whenever a scheduling decision needs to be made, the dispatcher finds the next thread that is ready to run which has the highest priority (threads at the same priority level are scheduled in a round-robin fashion). This thread then has its potential quanta reduced by its timeslice. If it has insufficient potential quanta, the kernel does a 'call' type IPC (send and closed receive) to its scheduler on its behalf.

If the thread has sufficient potential quanta, the threads scheduler also has its potential quanta reduced by the same amount. If the scheduler has insufficient potential quanta, it does an implicit IPC to its scheduler.

If a scheduler thread has insufficient time and is blocked on IPC, that IPC is cancelled and the thread executes the time fault to its scheduler (this is to avoid deadlocks).

This behaviour continues until τ_0 is reached, at which time the thread is dispatched.

If a thread needs a scheduler that is blocked on a time fault IPC (i.e. some other thread, or the scheduler, has attempted to reduce the schedulers potential quanta and caused it to fault to its scheduler), it is enqueued until that thread's scheduler gives it more potential quanta.

If a thread A is scheduled by L4 and needs to reduce the potential quanta of its scheduler thread, or one of its ancestors, thread B, but thread B is enqueued waiting on another thread C (i.e. thread C has executed a timefault IPC to its scheduler), then thread A is inserted into thread C's queue after thread B. This ensures that thread A is woken up when it has a chance of successfully reducing thread B's potential quanta.

4.3.4 Time

In any scheduling system, the notion of time, and its representation, is extremely important. Time is central to the services provided by a scheduler, and as such needs to be carefully defined.

There are 2 possible representations of time: in microseconds, and in *ticks* (the number of microsecond between the periodic timer interrupt). The first, microseconds, has the distinct advantage of being standard across all platforms, and a universal standard,

The second alternative, ticks, is also desirable in that it is central to the granularity of services provided by the system, especially timeslice length and minimum timeouts.

In deciding between the two for the scheduler implementation in L4, the philosophy of L4 needs to be taken into consideration. In [Lie95], an efficient μ-kernel is shown to be inherently non-portable, as portability requires a tradeoff with performance. Also, a feature is included in the kernel only if exporting it to user space would constitute a security flaw.

The granularity of scheduling should be exported to the systems built on top of L4, as it enables an OS personality or application to make intelligent resource allocation decisions — for example, is it fair for an application to request a timeout of half a millisecond? Will this be honoured by the operating system? What happens if this is less than the tick period — does the kernel return immediately, or block for much longer than anticipated? If the scheduling and timeout granularity is exported to the user, such problems disappear.

If the user gives its times to the kernel in ticks, there is no need for the kernel to keep track of the number of microseconds that have passed, only the number of ticks. This makes interrupt handling more efficient, and allows easy operations on time (for example, if a list needs to be scanned every n ticks, or every m microseconds, it is much easier to determine whether the list needs to be scanned if n is a multiple of 2, as opposed to if m is a multiple of 488.)

Finally, if the number of microseconds per tick is exported, the application can make its own time-accuracy tradeoffs if it wants to store time as microseconds internally. For example, the timer generates an interrupt every $488\mu s$ in the Alpha implementation of L4. This can be represented quite simply as a divide by 512, and generated in one shift instruction. If the application is not interested in exact time, then this is a cheap way to convert between microseconds and ticks. If the application is concerned with accurate time keeping, and is willing to pay for it, it can use a complicated algorithm, such as the one shown in Figure 4.3.4[2]. Applications can also cache the results of the conversion; this is infeasible for L4.

Thus, the microsecond representation becomes an unnecessary abstraction — L4 should use the tick as the unit of time to allow systems built on top to take advantage of the information this imparts. All times within L4 will be stored a ticks, and interfacing with L4 will also be in ticks. This allows a more efficient implementation without compromising security.

[2]This uses fixed point division as the Alpha architecture does not feature an integer divide instruction. The comments indicate the sub-cluster which executes that instruction; the total execution time is 6 cycles. Note that this level of precision is only needed for large (32 bit) integers; smaller integers can use lower precision, and hence faster code.

```
sll    t1,  6, t0      ; U(0): 0.0000000000000000000000001000000
s8addq t1, t1, t3      ; L(0): 0.0000000000000000000000000001001
s4addq t1, t1, t4      ; U(1): 0.0000000000000000000000000000101

sll    t1, 17, t5      ; U(0): 0.0000000000000100000000000000000
addq   t0, t4, t4      ; U(1): 0.0000000000000000000000001000101
addq   t0, t3, t3      ; L(0): 0,0000000000000000000000001001001

sll    t4, 16, t2      ; U(1): 0.0000000010001010000000000000000
sll    t3, 7, t3       ; U(0): 0.0000000000000000010010010000000
s4addq t0, t4, t4      ; L(1): 0.0000000000000000000000101000101

addq   t3, t4, t3      ; U(0): 0.0000000000000000010010111000101
subq   t2, t5, t5      ; U(1): 0.0000000010000110000000000000000

addq   t3, t5, t0      ; U(0): 0.0000000010000110010010111000101

srl    t0, 31, t0      ; U(0): t0 = (t1 * 4400581) >> 31
```

Figure 4.4: Fixed point approximation to division by 488.

Chapter 5

Implementation

This chapter presents the implementation of the design presented in Chapter 4. The chapter includes only a subset of the functions implemented; for example, the `thread_schedule` syscall has not been included as it does no 'interesting' work (merely updating variables in the TCB). The majority of the code presented has been optimised for clarity rather than performance.

5.1 Timeouts

The current implementation supports a single sorted timeout queue, which is consulted every 4 ticks. This scheme has performance and accuracy issues:

Expensive insertion The insertion operation is $O(n)$. As *all* wakeups are inserted into this list, n can be large.

Expensive removal As the current implementation utilises a singly linked list, the entire list needs to be searched for the thread to be removed.

Low wakeup granularity Because the soon wakeup queue is consulted every 4 ticks, the minimum wakeup granularity is 2ms. There is no good reason for this, only that scanning the queue every tick is inefficient.

Unnecessary interrupt overhead Even though the wakeup queue is only looked at every 4 ticks, this requires multiple expensive instructions (conditional branches and loads) even if no thread is ready to be woken up.

To solve these issues, multiple doubly-linked timeout queues were introduced, as described in [Lie93]. The timeout is logically split, as shown in Figure 5.1, into 3 sections. The lower r bits specify the range of each timeout bucket, the next i bits specify the number of buckets, and the last x bits are used to determine if the timeout is within the short wakeup range of 0 to $2^{i+r} - 1$ ticks.

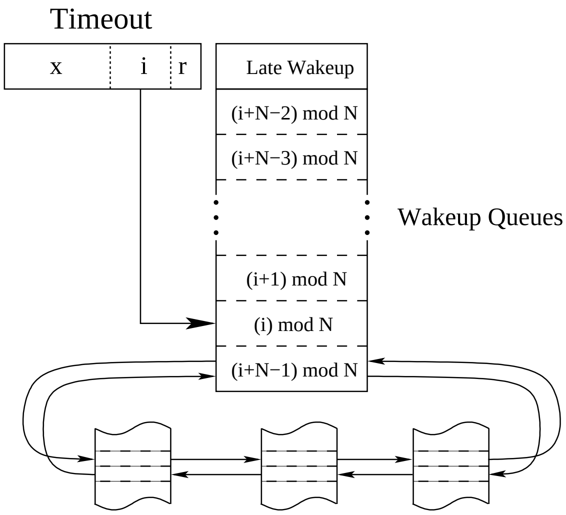

Figure 5.1: Multiple Timeout Queues

Choosing r to be 1, 2, or 4 gives 2, 4, or 16 ticks per timeout bucket respectively, and is efficient as it means that the timeout can be converted into an index by masking out the lower r bits and the upper x bits and doing an `s8addq`, `s4addq`, or `addq` to obtain the final offset (otherwise an extra shift is required — each bucket requires 16 bytes). The insertion macro is shown below. A bucket size of 4 is assumed.

```
.macro insert_timeout tcb, timeout, now_t, tmp1, tmp2, tmp3, pcpub
        ; Find maximum possible soon wakeup value
        addq    now_t, #<TO_I_NUM << TO_I_SHIFT>, tmp1

        ; Select relevant soon wakeup list
        and     timeout, #TO_I_MASK, tmp2

        ; Choose late wakeup list if timeout is too far into the future
        cmplt   timeout, tmp1, tmp1
        cmoveq  tmp1, #PCPUB_LATE_WAKEUP, tmp2
        addq    pcpub, tmp2, tmp2
```

```
        ; Update TCB pointers
        ldq     tmp3, PCPUB_SOON_WAKEUP(tmp2)
        stq     tpm3, TCB_WAKEUP_QUEUE(tmp2)

        lda     tmp1, <PCPUB_SOON_WAKEUP + 8>(tmp2)
        stq     tmp1, <TCB_WAKEUP_QUEUE + 8>(tcb)

        ; Update next link's previous pointer.
        lda     tmp1, <TCB_WAKEUP_QUEUE + 8>(tcb)
        stq     tmp1, 8(tmp3)

        ; Update PCPUB's next pointer
        lda     tmp1, TCB_WAKEUP_QUEUE(tcb)
        stq     tmp1, PCPUB_SOON_WAKEUP(tmp2)

        lda     tmp1, TCB_WAKEUP_QUEUE(tcb)
        stq     tmp1, PCPUB_SOON_WAKEUP(tmp2)

        ; Update next timeout
        ldq     tmp3, <PCPUB_NEXT_TIMEOUT>(tmp2)
        cmpult  timeout, tmp3, now_t
        cmoveq  now_t, timeout, tmp3
        stq     tmp3, <PCPUB_NEXT_TIMEOUT>(tmp2)
.endm
```

Note that the insertion macro maintains `PCPUB_NEXT_TIMEOUT`, a per-timeout-queue hint about the next valid timeout. There is no need to update it if an entry is removed, although it needs to be less than or equal to the next valid timeout. If the hint is less than the next wakeup, it will cause a spurious search of the wakeup list, but there will be no other side-effects. This allows fine grained timeouts as it the next timeout is checked every tick, without slowing down the critical interrupt path with unnecessary checks (most of the time).

To delete an entry from the linked list, the `delete_ll` macro is used, as no other bookkeeping needs to be done.

The late wakeup queue is parsed every 2^{i+r} ticks, and any threads that have a wakeup that is valid are inserted into the relevant soon wakeup queue.

5.2 Interrupt handling

At each timer interrupt, L4 decodes the interrupt reason, and branches to the timer interrupt routine, shown in the code below:

```
        ALIGN_FETCH_BLOCK
sys_int2_handler:
        ; Save user SP and PC
        open_frame

        ptldq   pp2, ptCurrentTicks
        ptldq        pp7, ptCurrentTimeslice

        ; Acknowledge interrupt to RTC
        AckRTC        pp4, pp3, pp1, pp6
        tcb        pp1

        ldq_a        pp3, TCB_SCHED_ACCOUNT(pp1)

        addq        pp2, #1, pp2
        ptstq        pp2, ptCurrentTicks

        addq        pp3, #1, pp3
        stq_a        pp3, TCB_SCHED_ACCOUNT(pp1)

        and     pp2, #TO_I_MASK, pp3
        beq     pp3, parse_late_wakeup
parse_late_wakeup_return:

        ; Load the next valid timeout
        and     pp2, #TO_I_MASK, pp3
        addq    pp3, pp6, pp3

        ; Note that this can be -1ull
        ldq_p        pp3, PCPUB_NEXT_TIMEOUT(pp3)

       ; Update remaining timeslice
       ; Note that it may be 0 before
       ; the subtraction
        mov        pp7, pp4
        subq        pp7, #1, pp7
        cmovne        pp4, pp7, pp4
        ptstq        pp4, ptCurrentTimeslice

       ; Need reschedule?
        cmpeq        pp4, zero, pp7
```

```
        ; Do we need a wakeup?
        cmpule       pp3, pp2, pp2
        bne        pp2, parse_soon_wakeup
parse_soon_wakeup_return:
        ; Do we need to reschedule?
        blbs    pp7, parse_schedule
        close_frame

        ALIGN_FETCH_BLOCK
parse_schedule:
        ; Prevent recursive timer irq's
        disable_int        pp0
        push        p_gp

        kernel        k_switch_thread
```

The timer interrupt handler is platform independent (originally there was a handler per architecture). The only platform-specific code is abstracted into the `AckRTC` macro, which is responsible for acknowledging the timer interrupt to the Real Time Clock (RTC), and the `ptldq/ptstq` macros (the 21264 does not include any spare PAL registers, so they must be simulated in the PCPUB, while the other generations do).

In the fast case the interrupt handler does a minimum amount of work:

1. Update internal datastructures.
2. Update TCB accounting.
3. Update reamaining ticks.
4. Parse late wakeup queue if necessary.
5. Parse soon wakeup queue if necessary.
6. Reschedule if necessary.

Note also that the fast case is the fall through case. This is important, as in PAL mode on the 21264 all conditional branches are mispredicted, incurring a 7 cycle penalty.

5.3 Rescheduling

When a reschedule is necessary, due to a timeslice expiration, or another thread being woken up, the `k_switch_thread` function is called (the 'k' means that it runs in kernel mode). The relevant parts of this function are shown in the code below, essentially the loop to select and dispatch the new thread.

```
REPEAT
        EXITZ  t7
        subq   t8, #4, t8
        subq   t7, #1, t7
        ldl_p  t0, 0(t8)
        CONTZ  t0

        ldq    t0, TCB_BUSY_NEXT(t0)

        REPEAT
                ldq   t1, TCB_THREAD_STATE(t0)
                bic   t1, #TFS_RUNNING, t1
                IFZ   t1
                        switch_context t0, v0, t10, t1, t3
                        ldq   t2, TCB_TIMESLICE_LEFT(t0)

                        tcb   t1
                        small_switch  t1, t0, AT, switch_preempted

                        bne   t2, restart_thread
                        br    zero, refresh_timeslice
                XENDIF
                dequeue_busy t0, t1, t2, t3, v0
                mov    t0, t1
                ldl_p  t0, 0(t8)
                CONT   t0
        ENDR
        CONT    t7
ENDR
```

The `switch_context` macro sets the internal processor registers, essentially the Address Space Identifier (ASID) and the page table root used in pagefault handling. The `small_switch` macro saves the target (in this case `switch_preempted`, a function that restores all saved registers and returns to user mode) and current stack pointer in the current thread's TCB, and loads the saved stack pointer and program counter from the target thread.

If the target thread has any remaining time (if it was preempted in the middle of a timeslice, for example) it is restarted immediately, otherwise the `refresh_timeslice` entry point is branched to.

Note that this loop removes a thread from the run queue if it is not runnable — this can come about due to the *lazy scheduling* [Lie93] mechanism employed during IPC — a thread is not removed from the busy queue, only marked as not runnable if it needs to block.

5.4 Timeslice acquisition

When a thread is selected, it needs to acquire a timeslice, as described in the Chapter 4. The fast case code is shown below:

```
        ALIGN_FETCH_BLOCK
refresh_timeslice:
        ldq   t2, TCB_SCHED_TIMESLICE(t0)
        mov   t0, t1
        ldl   t3, TCB_LIST_STATE(t0)
        GET_16CONS t4, LLS_TIMEFAULT_NOW
        and   t3, t4, t3
        beq   t3, refresh_retry
        ret   zero, (AT)

refresh_retry:
        get_PCPUB v0, t4
        REPEAT
                cpu_tcb_dispatcher t3, t4
                cmpeq   t1, t3, t3
                EXIT    t3

                ldq   t3, TCB_SCHED_QUANTA(t1)
                cmpule  t2, t3, t4
                beq     t4, timefault_execute

                addq    t3, #2, t4
                subq    t3, t2, t5
                cmoveq  t4, t3, t5
                stq     t5, TCB_SCHED_QUANTA(t1)

                ldq     t3, TCB_SCHEDULER(t1)
                tcb_ptr  t3, t4, t5

                ldq     t5, TCB_MYSELF(t4)
```

```
                cmpeq   t5, t3, t3
                beq     t3, invalid_scheduler

                mov     t4, t1

                ldl     t5, TCB_LIST_STATE(t1)
                GET_16CONS t6, LLS_TIMEFAULT_QUEUE
                and     t5, t6, t5
                bne     t5, timefault_enqueue

                CONT
        ENDR

        ldl   t3, TCB_LIST_STATE(t0)
        GET_16CONS t4, <LLS_TIMEFAULT_QUEUE!LLS_TIMEFAULT_NOW>
        bic   t3, t4, t3
        stl   t3, TCB_LIST_STATE(t0)

restart_thread:
        ptstq t2, ptCurrentTimeslice, v0

        blbs  AT, back_to_pal
        ret   zero, (AT)

        ALIGN_FETCH_BLOCK
back_to_pal:
        l4_call_pal    PAL_RETPAL
```

The portion of code at the start is to handle threads that have timefaulted previously and have been restarted. The code checks to see whether the thread belongs to a time queue, and if so jumps to the relevant restart address within the kernel. This is to ensure that a thread does not reenter the timeslice acquisition code — obviously, to do so would be incorrect.

The main loop does the following, for each level in the hierarchy, starting with the thread to be dispatched:

1. Ensure that the thread has sufficient potential quanta. If the thread's timeslice is greater than the potential quanta of the current thread (itself or one of its ancestors), the `timefault_execute` function is branched to, which executes a timefault from the current thread to its scheduler.

2. Update the thread's potential quanta. If the thread has infinite potential

quanta, represented as -2, then no modification is made to the potential quanta, otherwise the timeslice is deducted.

3. Verify the thread's scheduler. The thread ID of the scheduler must be checked to ensure that it is the same as that stored in the thread's TCB. This is necessary, as the scheduler's task may have been recycled, in which case the version number will have been incremented, and the scheduler thread will no longer be the scheduler (it will belong to some other application).

4. Enqueue the thread after its scheduler if necessary. If the thread is a descendant of a thread who has timefaulted, then the thread needs to be enqueued in that ancestors timefault queue. This is to prevent multiple timefaults by a thread.

In an SMP implementation, there would also need to be a CPU check for the scheduler thread after the thread ID check, If the scheduler was on a different CPU, then a function similar to `timefault_execute` would need to be written, which would generate an IPC to the threads scheduler.

When the loop has reached τ_0, in this case the dispatcher, the loop will finish and the thread will be dispatched. This involves cleaning up the list state for that thread, setting the `ptCurrentTimeslice` register to the threads timeslice, and branching to the threads restart address (if the lower bit of the restart address is set, then the restart address is in PALcode, probably IPC, and so a return to PAL mode is needed).

The fast path for this code does 4 loads, and 1 store. The 2 scheduling attributes, `TCB_SCHED_QUANTA` and `TCB_SCHEDULER` should be on the same cache line, and the `TCB_LIST_STATE` and `TCB_MYSELF` are both commonly used variables, so there is a good chance that they are also in the cache.

The `timefault_execute` function is responsible for notifying a thread's scheduler (the thread need not be the thread that is being dispatched; on the other hand, it may be) that it has insufficient potential quanta. The code is shown below:

```
        ALIGN_FETCH_BLOCK
timefault_execute:
        push        p_t0!p_t1!p_t2!p_AT
```

```
        insert_l  t0, t1, t3, TCB_TIMEFAULT_ROOT

        ldl        t3, TCB_LIST_STATE(t0)
        GET_16CONS t3, <LLS_TIMEFAULT_QUEUE!LLS_TIMEFAULT_NOW>
        bis        t3, t4, t3
        stl        t3, TCB_LIST_STATE(t0)

        GET_16CONS  t3, TFS_LOCKED_WAITING
        stq     t3, TCB_LIST_STATE(t0)

        switch_context t0, v0, t5, t3, t4
        small_switch        t0, t1, AT, timefault_retry

        ;; If the faulter is doing an IPC, cancel it.
        ldq        t3, TCB_THREAD_STATE(t1)
        bic     t3, #TFS_RUNNING, t3
        bne        t3, timefault_abort_ipc
timefault_abort_ret:

        push        p_t0!p_t1!p_t2!p_AT

        ;; Set up faulter thread
        ldl        t2, TCB_LIST_STATE(t1)
        GET_16CONS t4, <LLS_TIMEFAULT_QUEUE!LLS_TIMEFAULT_NOW>
        bis        t2, t4, t2
        stl        t2, TCB_LIST_STATE(t1)

        ;; Prepare for IPC timefault.
        ldq        a0, TCB_SCHEDULER(t1)

        clr        a1
        GET_16CONS a2, <<0 @ 13>!<63 @ 2>!<MVR_CLOSEDWAIT>>
        GET_16CONS a3, IPC_TIMEOUT_NEVER
        clr        a4
        clr        a5

        l4_call_pal        PAL_TIMEFAULT

        pop        p_t0!p_t1!p_t2!p_AT
        ret        zero, (AT)
```

This function is relatively simple, although there are a few subtle issues. Firstly, the thread that is executing the timefault may or may not be the thread to be dispatched. Secondly, the timeslice needs to be stored. If it is not (i.e. if it is read from the TCB), it is possible for a thread to execute a denial of service attack on its scheduler (the actual sequence of events is somewhat complicated).

This function does the following:

1. Saves the necessary state on the stack — thread to be dispatched, faulting thread, timeslice length, and the restart address.

2. Marks the thread to be dispatched as non-runnable, and inserts it into the faulting thread's timefault queue.

3. Switches context to the faulting thread, including stack pointer and program counter. The thread to be dispatched has its return address set to `timefault_retry`, a function which restores the state of the thread to be dispatched and takes up where it left off in `refresh_timeslice`.

4. Cancels any pending IPC. If the faulting thread is currently waiting to send or receive the IPC is cancelled, and the threads return address modified accordingly.

5. Executes the timefault. The IPC parameters are setup in this function, and PAL mode is entered. The `PAL_TIMEFAULT` PAL call then does an IPC `call_pal`. When the IPC returns (i.e. the faulting threads scheduler replies), all the threads waiting on the faulting thread are woken up.

   The IPC needs to be done in PAL mode as the code directly after it needs to be atomic with respect to timer interrupts; if an interrupt occurs before the threads state can be modified, then the faulting thread may reenter the timefault handling code, which would lead to incorrect behaviour.

6. The function then returns to the threads restart address (saved on the stack before the IPC).

5.5 Discussion

The addition of the mechanisms designed in Chapter 4 have been presented in this chapter. As can be seen from examining the amount of code presented, the fast case (where all threads have sufficient quanta) adds very little overhead, approximately 4 loads and 1 store for each level in the hierarchy. As the number of levels is expected to be quite small (certainly less than 5), the additional code should have minimal impact on the caches, and should have a minor impact on the reschedule time.

The modification of the timeout mechanisms should also reduce the amount of cache pollution, as insertion into and remove from the wakeup list(s) need only touch the TCB of the thread at the head of the list, not each waiting thread. These optimisations should speed the common case where a thread completes the IPC successfully, and needs to insert and remove itself from the timeout queue.

Chapter 6

Results

This section presents the benchmarks used to determine the feasibility of the design and implementation presented in previous chapters.

Figure 6.1 shows the general system setup used to exercise the new scheduling mechanisms. Thread 0 is the root scheduler for this system, and can be considered to have contiguous, infinite time.

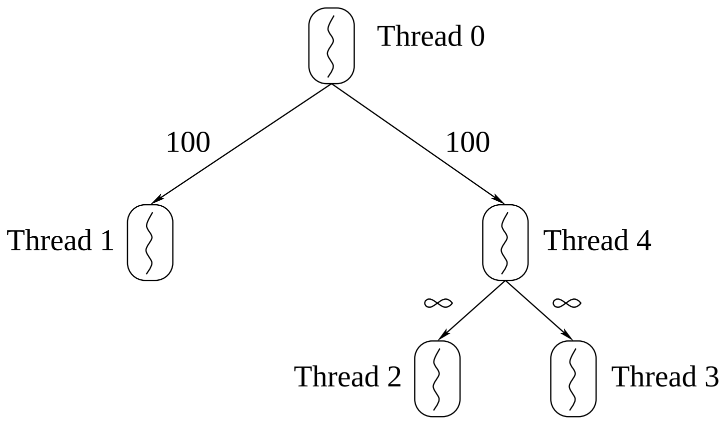

Figure 6.1: Benchmarking system

The potential quanta of thread 2 and thread 5 has been set to 100, so that they should receive equal amounts of CPU time over the measured interval.

The basic function followed by the leaf threads (threads 1, 2, and 3) is an infinite loop. For profiling purposes, every 1000 loops each thread increments a per-thread counter that is visible to the rest of the threads in the system. Every 100ms, a high priority thread wakes up and records these values. After

100 iterations (so after 10 seconds has passed), the measuring threads outputs its measurements and stops the benchmark.

6.1 Hierarchical scheduling

This benchmark examines the hierarchical scheduling mechanisms implemented. Each leaf thread executes the infinite loop described above, and the root scheduler allocates threads 1 and 4 potential quanta after both have generated a timefault. The results are shown in Figure 6.2.

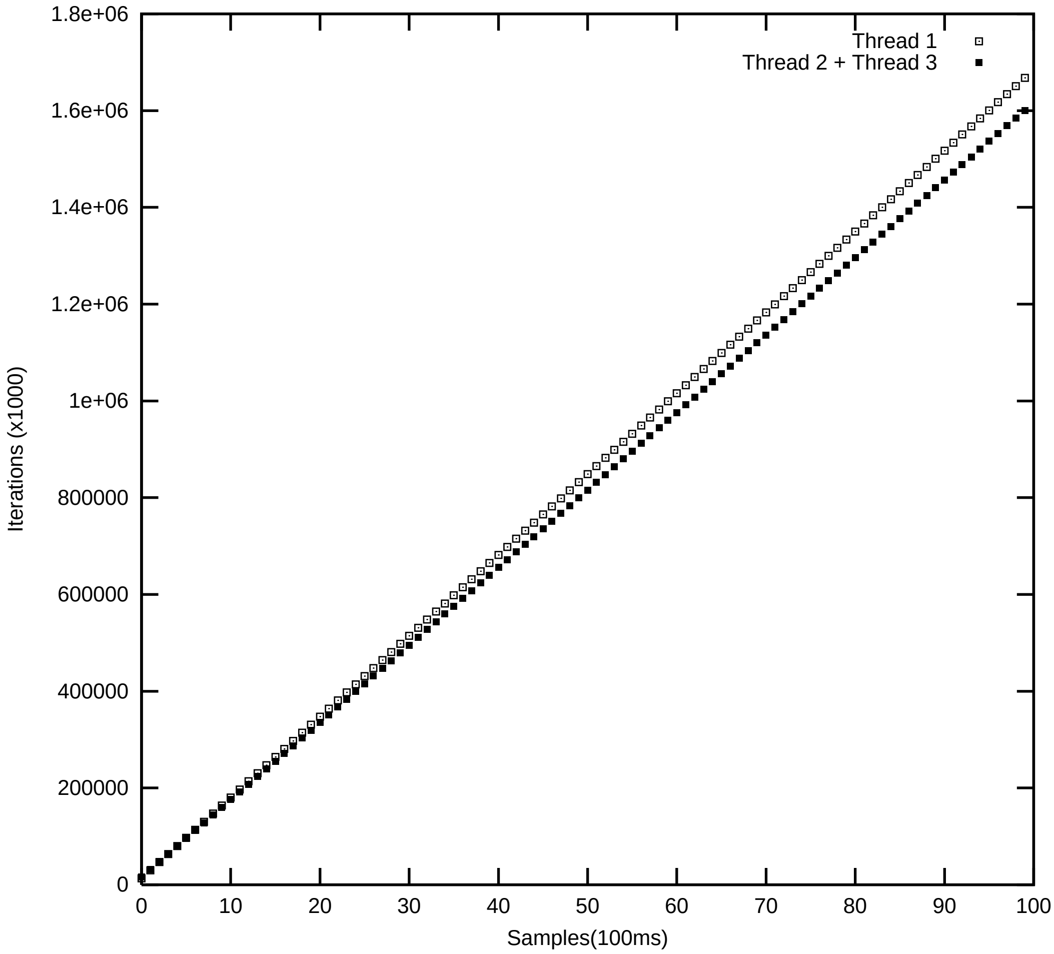

Figure 6.2: Hierarchical scheduling results

The top line is thread 1's counter, while the lower line is the sum of threads 2 and 3. While the two lines are very close, they are diverging by a small amount. The cause of this is most probably the time donated by thread 0 on timefaults — thread 5 also receives the donated timeslice, but the time it spends executing is not recorded in this benchmark.

Overall, this result is quite encouraging, as the results are basically what was predicted. The individual counters for threads 2 and 3 are not shown as they are basically identical.

6.2 Resource Isolation

This benchmark examines the resource isolation mechanisms implemented within L4. This benchmark is similar to the previous one, but thread 3 sleeps until halfway through the benchmark (as shown by the flat portion of the graph). The results for this are shown in Figure 6.3.

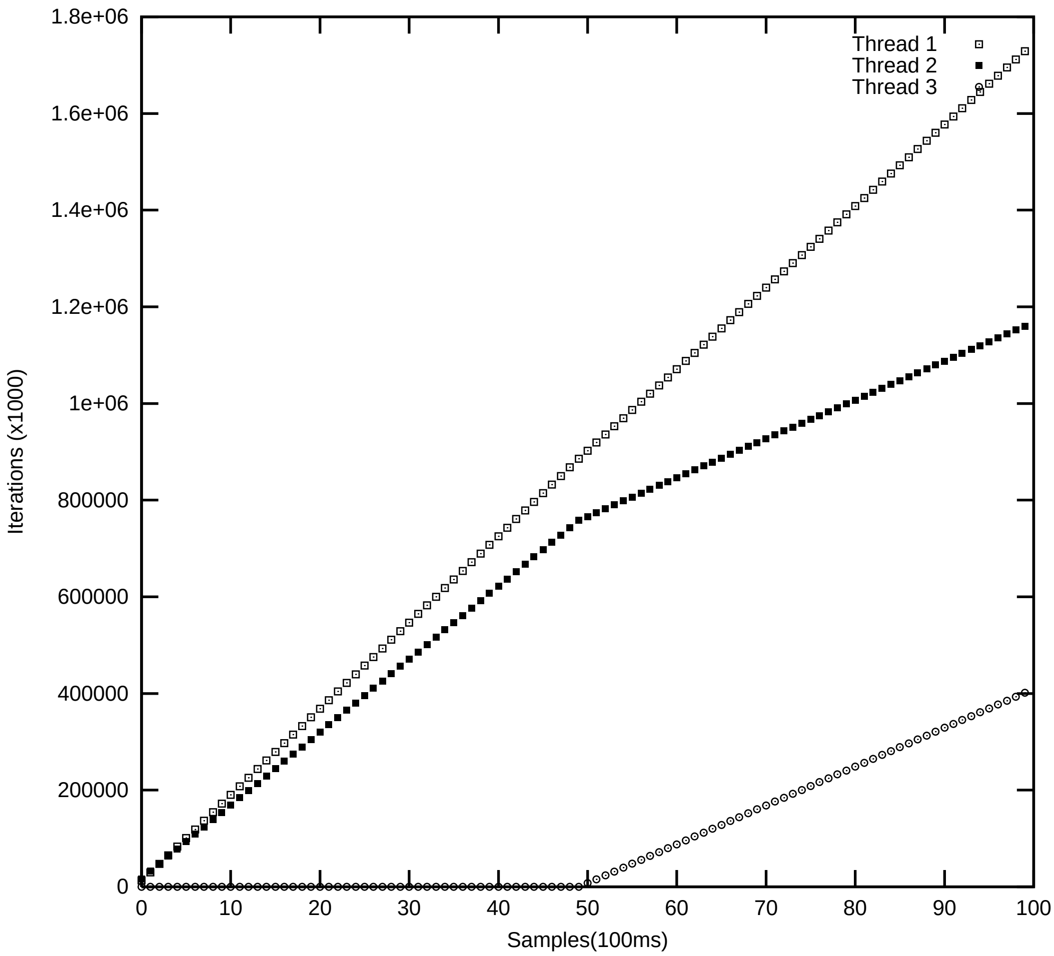

Figure 6.3: Resource isolation results

The top line is thread 1's counter. The middle line is thread 2's counter, while the bottom line is thread 3's counter. The major point to notice in this graph is that thread 2 is the only thread effected by thread 3 waking up. Thread 1 does not notice any changes in the amount of processor time that it receives.

Again, these results were as expected, with the other thread in thread 3's scheduling group losing half of its processor time to thread 2, and thread 1 receiving the same amount of time.

6.3 Resource Revocation

This benchmark examines the resource revocation mechanisms implemented within L4. The test is similar to the first benchmark, but halfway through the test, thread 4 is no longer granted potential quanta, and so it and its children should not be able to execute. The results are shown in Figure 6.4.

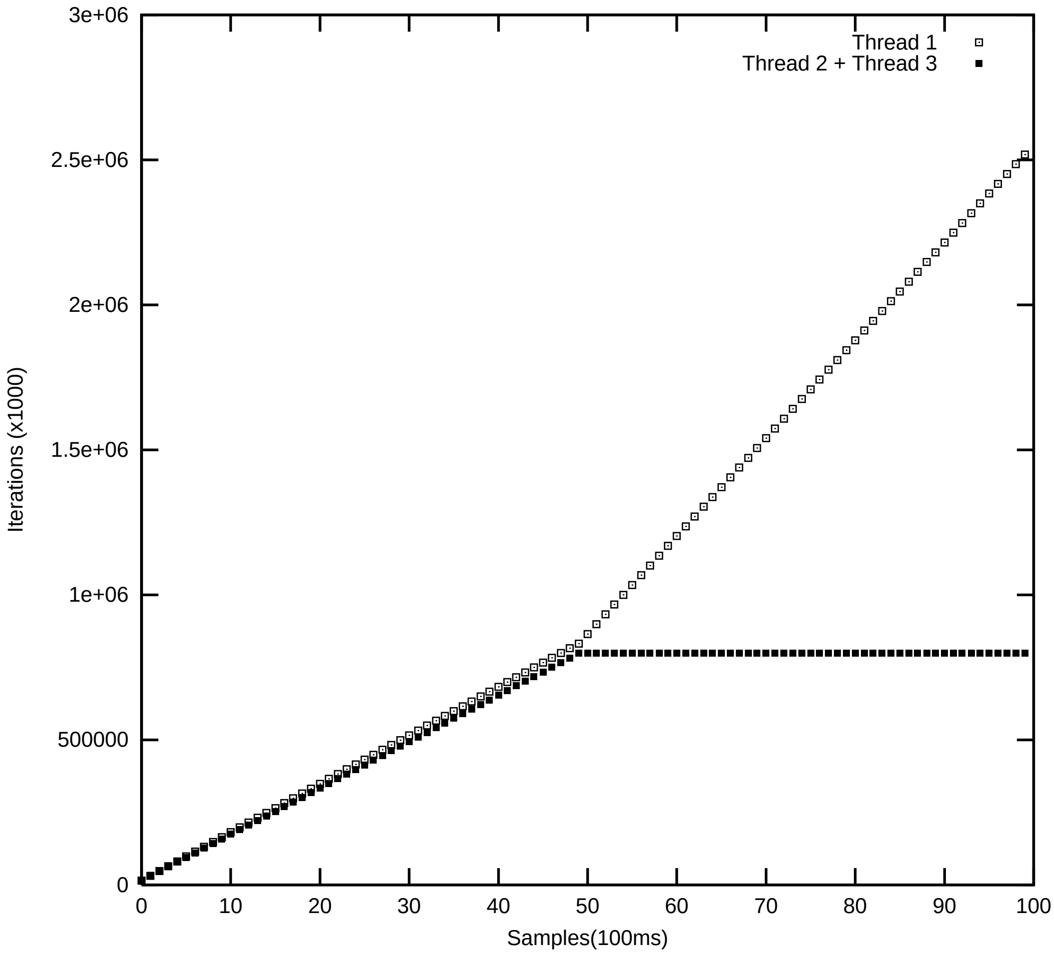

Figure 6.4: Resource revocation results

The top line is again thread 1, while the bottom line is the sums of threads 2 and 3. Halfway through the test, the processor is revoked from thread 4, and hence threads 2 and 3. Thread 1 is now the sole runnable thread on this processor, and its usage curve responds accordingly.

This test was as expected, with thread 1 receiving all the processor time after thread 5 had it revoked. This would be the expected use of the revocation mechanism — to allow a thread to receive a larger portion of a processor by removing resource rights from another thread.

6.4 Discussion

Overall these results were encouraging. The results were as expected, with some slight deviation, probably due to the donation semantics of L4's IPC mechanisms. The results shown in Section 6.2 are particularly important, in light of the resource isolation goals of this thesis.

Chapter 7

Conclusions

The aim of this thesis was to design flexible, secure, and efficient scheduling mechanisms for the L4 μ-kernel, and implement them in L4/Alpha. While the design of these mechanisms was undertaken with these goals in mind, the results from the benchmarks in the previous chapter validate those design decisions.

Bibliography

[BH00] Ray Bryant and Bill Hartner. Java technology, threads, and scheduling in linux: Patching the kernel scheduler for better java performance. http://www-4.ibm.com/software/developer/library/java2/, January 2000. IBM Linux Technology Center.

[Bla90] D. L. Black. Scheduling support for concurrency and parallelism in the mach operating system. *IEEE Computer*, 23(5:35–43, May 1990.

[CDV$^+$94] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *asplos*, October 1994.

[CJ98] George Candea and Michael B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Second USENIX Windows NT Symposium*, pages 157–166, Seattle, WA, August 1998. USENIX.

[Com99a] Compaq Computer Corp. *Alpha 21264 Microprocessor Hardware Reference Manual*, 1999.

[Com99b] Compaq Computer Corporation. *21264 Specifications*, 4.2 edition, Feb 1999.

[EHL97] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual — MIPS R4x00*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 1997. UNSW-CSE-TR-9709. Latest version available from http://www.cse.unsw.edu.au/~disy/L4/.

[FS96] Bryan Ford and Sai Susarla. Cpu inheritance scheduling. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, October 1996. USENIX Assoc.

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

[LES+97] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for efficiency). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Cape Cod, MA, USA, May 1997. IEEE.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 175–88, Asheville, NC, USA, December 1993.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[Lie96] Jochen Liedtke. *L4 Reference Manual — 486/Pentium/PentiumPro, Version 2.0.* GMD, Schloß Birlighofen, Germany, September 1996. Working Paper 1021.

[Lin] The linux kernel. http://www.kernel.org.

[PMG99] David Petrou, John W. Milford, and Garth A. Gibson. Implementing lottery scheduling: Matching the specialisations in traditional schedulers. In *Proceedings of the 1999 USENIX Technical Conference*, pages 1–14, Monterey, CA, USA, June 1999.

[Pot99] Daniel Potts. L4 on uni- and multiprocessor Alpha. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 1999.

[RSH00] John Regehr, Jack Stankovic, and Marty Humphrey. The case for hierarchical schedulers with performance guarantees. Technical Report CS-2000-07, University of Virginia, March 2000.

[Sch96] S. Schönberg. The L4 microkernel on Alpha - design and implementation. Technical Report 407, Cambridge University, 1996.

[SL93] Mark S. Squillante and Edward D. Lazowska. Using processor cache affinity information in shared-memory multiprocessor scheduling. *ieeetpds*, 4(2):131–143, February 1993.

[VGR98] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *asplos*, pages 181–192. acm, 1998.

[Wig99] Adam Wiggins. The design and implementation of the l4 microkernel on the StrongARM SA-1100. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 1999.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 1–11, Monterey, CA, USA, November 1994. USENIX/ACM/IEEE.