The University of New South Wales

School of Computer Science and Engineering

L4 on Uni- and Multiprocessor Alpha

Daniel Paul Potts (2192966)

Bachelor of Engineering (Computer Engineering)

October 1999

Supervisor: A/Prof. Gernot Heiser
Assessor: Dr. Jerry Vocheteloo

Abstract

The Alpha RISC architecture has established itself as one of the leading microprocessors in today's high end workstation market. The Alpha microprocessor boasts as being the world's fastest 64-bit processor and looks likely to remain this way for several years to come.

Recent trends in increased use of parallelism of processors in multiprocessor machines has seen the desire to produce a μ-kernel that is capable of supporting such hardware.

This thesis examines issues related to supporting multiprocessors in the L4 μ-kernel and proposes solutions to allow L4 to support such systems in a scalable, fast and efficient way.

This thesis also examines the current L4/Alpha that provides an implementation for uniprocessor Alpha microprocessors.

Acknowledgements

This thesis is the result of many hours spent with the Operating Systems group in Keg Lab at the University of New South Wales and with the Operating Systems group at Dresden University of Technology, Germany.

I'd firstly like to thank my supervisor Gernot Heiser for always providing support, guidance and the opportunity to work on this project.

I would like to thank Sebastian Schönberg for providing the initial port of L4 on Alpha and assisting me throughout the implementation of other components of L4 on Alpha. His work on the μ-kernel has been truly remarkable - relatively bug free and a pleasure to read (remember it is assembler!).

The Operating Systems Group at the University of Dresden in Germany, in particular Hermann Härtig for providing me with the opportunity to work with his students. Their assistance and feedback has provided a strong influence in the direction of the research in this thesis. Also, Adam Wiggins whom I was given the pleasure of travelling and studying with in Dresden.

I'd also like to thank Stig Telfer of Alpha Processor Inc. for his assistance on some of the more troublesome issues I have had related to the Alpha.

I'd also like to thank my DiSy group colleagues for their assistance and feedback needed to develop ideas and for putting up with me in the Keg Lab for endless hours. Thanks to Alan Au for assisting me in understanding the various workings of L4 internals.

I'd like to thank Belinda Hannan for her love, support and patience over the past few years.

Finally I'd like to thank my parents Brian and Marianne Potts for their continued love, support and the opportunity to attend university.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

This thesis is divided into two major parts which deal with the completion of the uniprocessor implementation of L4 on the Alpha architecture (L4/Alpha). Research and implementation of a scalable multiprocessor L4/Alpha kernel that is planned to be implemented in December 1999.

Today we are beginning to see a trend towards multiprocessor desktop machines and even larger multi-way multiprocessor servers in use. This thesis has been motivated by the need to provide a μ-kernel implementation for such machines that offers the same efficiency and flexibility that L4 [24] gives to uniprocessor machines. L4 is a high performance second generation μ-kernel.

1.2 Structure of the Thesis

This thesis is divided up into two main parts. The first part covers relevant research into the design and implementation of a multiprocessor version of the L4/Alpha μ-kernel. The second part presents implementation details of features added to the current L4/Alpha μ-kernel.

Chapter 2 covers relevant background information required to build an understanding of the discussion presented in this thesis. This includes an overview of related work, the Alpha architecture, the L4 μ-kernel and a look at the work previously completed on L4/Alpha.

Chapter 3 covers the multiprocessor design and implementation problems and presents solutions.

Chapter 4 presents implementation details of the L4/Alpha uniprocessor μ-kernel that I have contributed.

Chapter 5 provides a conclusion and direction for future work.

Chapter 2

Background

This chapter gives an overview of related work performed in relation to multiprocessor kernels, a short overview of the Alpha processor, an introduction to L4 μ-kernel structure and an overview of the previous L4/Alpha work completed externally to this thesis.

I also discuss some of the hardware we expect the multiprocessor L4/Alpha implementation to run on.

2.1 Related Work

The design and implementation of a μ-kernel such as L4 involves close examination of techniques used to produce a lean and fast μ-kernel. The L4 μ-kernel designed by Jochen Liedtke provides the basis for all L4 implementations. Currently implementations exist or are being developed for Alpha, ARM [37], MIPS [16] and x86 [25] [20].

This thesis outlines proposed plans for the first available multiprocessor L4 μ-kernel to be implemented specifically on the Alpha.

2.1.1 Multiprocessing

With multiprocessors being common today in the desktop and server market, several commercial operating systems support such machines. Solaris [33] and Windows NT [11] are examples of two operating systems that support this hardware.

Both operating systems provide mechanisms to execute threads across multiple processors of the system. Each provides for scheduling of kernel-level threads. The kernel-level threads are assigned user-level threads via the application programmer or a library interface.

Both operating systems use a priority based, time sliced, preemptive multitasking algorithm to schedule the kernel-level threads.

Solaris assigns kernel-level threads to a scheduling class with priorities inside each class. These classes define the type of scheduler which can be system, time shared, real-time or gang scheduled. These kernel-level threads are known as "light-weight processes" (LWP) to distinguish them from user-level threads, which only exist at the library level. The Solaris kernel uses a global ready queue to dispatch threads on a free CPU [17].

In order to schedule user-level threads the library is used to map them onto LWPs. As such the Solaris kernel is not aware of the existence of user-level threads but provides mechanisms to manage them.

The user-level thread library of Solaris allows threads to be bound to a particular LWP or unbound in which case it is scheduled from a pool of threads. Further more this library is responsible for adjusting the number of LWPs for a given process which complicates matters, although this does reduce kernel data structure sizes and simplifies scheduling.

Processes in Windows NT consist of one or more kernel-level threads. It provides user-level threads named "fibers" that are not directly managed by the operating system. The user is responsible for scheduling and managing fibers, particularly when the user desires to map more than one fiber to a kernel-level thread. The NT kernel dispatches kernel-level threads from a global ready queue similar to the method used in Solaris.

These systems schedule their threads on a systemwide basis utilising global priorities to ensure correct scheduling across the whole system.

Both operating systems and others that support scheduling on multiple processors will usually attempt to optimise their method of scheduling from a global ready queue. However this is certainly not optimal and still subject to system conditions at any given time. It has been shown that a global ready queue limits scalability [36]. Scalability is limited by several factors in relation to scheduling. Possibly the most significant influence is that when moving a thread between processors, it is restarted with cold caches. This is important considering the ever increasing gap between memory and CPU speed. Shared data usage has also been increased inside the kernel through the use of a shared global queue.

For machines with non-uniform memory access (NUMA) this effect is an even greater penalty. On these systems a node's local RAM essentially serves as a cache for a distributed-shared-memory (DSM) system. Sharing kernel structures over such a system should certainly be minimised.

Solaris and Windows NT show that much operating system design work revolves around extending the idea of a UNIX process or address space to threads by utilising their execution by the kernel's scheduler as discussed in [38]. These scheduling strategies have been designed for a particular purpose and only allow small changes in scheduling parameters.

Mach [27] also uses a global scheduling queue and like the other kernels examined attempts to be intelligent about sharing of such queues across a large system. Periodic scan of all threads in the run queues is performed to avoid starvation, although this is done more efficiently by having each thread maintain its own usage information. This is described in [6] with a method that appears complex and tailored for a specific scheduler design. It provides load balancing in the kernel which, while being efficient if well implemented, may restrict user implementation of alternate scheduling procedures. This method would be of concern particularly for the NUMA machines they support.

2.2 Alpha Architecture

Alpha [8] is a 64-bit load/store RISC architecture that has been designed for high performance. Instruction codes are all 32-bits long supporting five different instruction formats. The processor has 32 integer and 32 floating point 64-bit registers.

Current Alpha processors implement a 43-bit or a 48-bit virtual address space. These processors have up to four integer and four floating point units.

Current and future Alpha processors have been designed for multiprocessing with separate and very high speed system and memory buses. Next generation Alpha processors will support simultaneous multi-threading at the CPU level including support for very large (512-way) cache coherent NUMA machines.

2.2.1 Privileged Architecture Library (PAL)

Alpha's flexible Privileged Architecture Library (PALcode) is a set of subroutines used to implement some complex instructions that are not implemented in the processor core such as primitives for context switching, interrupts, exceptions and memory management.

PALcode is written using standard Alpha instructions including some privileged operations that provide access to low-level hardware. Typically PALcode is tailored to a particular operating system and Alpha processor revision.

The Alpha has three main modes of operation: user mode, kernel mode and PALcode mode (known as PALmode). In PALmode (and kernel mode) we have access to an extra set of registers known as PALtemp registers which we use to prevent the need to save user registers before using them. L4 uses these registers to store some state information.

PALcode is invoked at specific entry points. These are:

- On system reset.

- Hardware raised asynchronous events.
- Synchronous software events, raised by the *callpal* instruction or software faults.

On the Alpha 21164 (current processor being used for the uniprocessor port), a switch to PALcode can be done in between 4 and 9 cycles.

PALcode requires that careful attention is paid to the rules of code scheduling. For example, the result of reading a value from a PALtemp register cannot be used within two cycles after the read instruction itself. Violating such rules can lead to unpredictable behaviour of the processor.

PALmode has the following features:

- Instruction stream memory mapping is disabled.
- Data stream mapping is enabled.
- Privileged and unrestricted access to all hardware parts.
- Interrupts are disabled.

2.2.2 Internal Processor Registers

A set of Internal Processor Registers (IPR) are used to control different parts of the Alpha processor. These registers are only accessible in PALmode or in kernel mode given the right permissions. Typically accessing such registers has a delay of one cycle in current implementations.

2.2.3 Translation Look-aside Buffers

Translation Look-aside buffers (TLBs) are used to cache recently used page translation references.

An Alpha implementation may choose to have combined or separate data and instruction TLBs.

Typically an Alpha contains two independent TLBs, one for data and one for instruction pages. In the case that the processor does not find an entry for a given virtual address, an exception (trap to PALcode) is raised.

When a non-software field of a valid page table entry (PTE) is modified, copies of that PTE must be made coherent. PALcode mechanims are available to clear all TLBs, both DTLB and ITLB for a given virtual address, specific entries in a TLB, or all entries with the address space match bit clear.

2.2.4 Interrupts

The Alpha has three sources of interrupts:

- Hardware interrupts - level sensitive interrupts triggered by external signals
- Prioritised software interrupts - sourced and independently maskable by the software interrupt request register
- Asynchronous system traps - sourced by the asynchronous system trap request

If enabled, interrupts are performed by a branch to PALcode with relatively low latency.

Multiprocessor interrupt usage

On multiprocessors, an inter-processor interrupt exists. On most Alpha chipset implementations external interrupts can be programmed to goto one or more processors.

2.2.5 Memory Management

The Alpha physical address space is divided up into four equal size regions that are distinguished by features such as the coherency and whether the region exhibits memory-like or non-memory-like behaviour.

Physical memory exists in a memory-like cached region. Other regions such as the sparse PCI address space exist in memory-like non-cached regions. Chipset configurations exist in non-memory-like regions. This is defined by the fact that some addresses may be read or write only.

Cache Coherency

The Alpha architecture states that all processors or I/O devices must provide a coherent view of memory. That is, for all physical address space regions a coherency point (which may move) is specified. Each write by a processor or I/O device becomes visible to all other processors. In terms of physical memory reads and writes, the coherency point is specified to be the processor's cache.

Each of the four physical address space regions may provide coherency in different ways such as write-though, write-back or no caching.

The Alpha maintains a cache hierarchy as shown in Figure 2.1. In order to provide a coherent view of memory from each processor, system logic maintains that each Dcache (data cache local to each processor) is kept coherent with the Bcache (backup cache, typically found on the motherboard), that it is a subset of. The Bcache is kept coherent with main memory by system logic.

The Icache (instruction cache) is not considered a subset of main memory and as such is not maintained coherent by the hardware, and must be done in software.

On multiprocessors cache coherency is of particular importance. In order to maintain coherency between separate CPU caches the hardware implements cache coherency by a given protocol.

Write invalidate protocols maintain consistency by ensuring that only a single

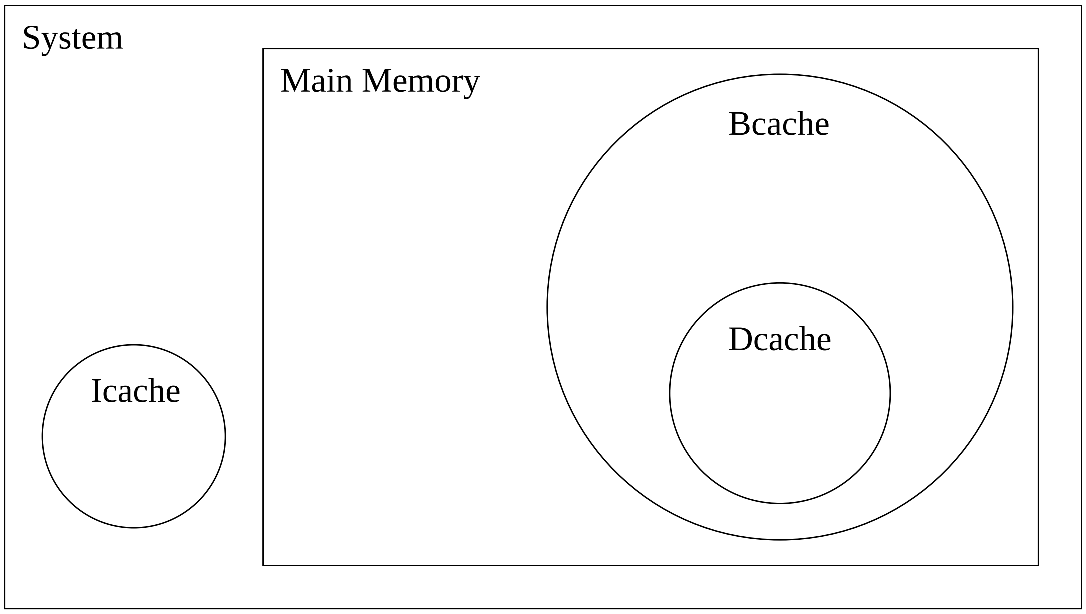

Figure 2.1: Alpha Memory Model

cached copy of a line from main memory exists at the point where a store occurs to that line. One such protocol is the MESI protocol discussed below. Flush cache coherency protocols may also be used to achieve the same effect.

One cache consistency protocol that provides full support for multiprocessing is the MESI protocol consists of four states that define whether a block is modified (M), exclusive (E), shared (S), or invalid (I).

The Alpha 21264 [9] supports several cache block states. These include:

- Invalid - 21264 does not contain a copy of the block.
- Clean - 21264 holds a read only copy of the block, exclusively.
- Clean/Shared - 21264 holds a read only copy of the block, it may exist else where.
- Dirty - 21264 holds a read-write copy of the block, exclusively, that must be written back to memory when evicted.
- Dirty/Shared - 21264 holds a read-only copy of the dirty block, it may exist else where, must be written back to memory when evicted.

Making the caches and translation buffers (TLBs) coherent on only one processor is not always sufficient. An operating system must arrange to ensure each processor has a coherent view of the system.

Read/Write buffers

Write buffers may be used to delay and aggregate writes. From the viewpoint of another processor or I/O device, buffered writes appear not to have happened yet. This means that we cannot rely on a load or store operation containing the most up to date value. Internally in a processor, loads and stores check buffers for the most up to date contents.

To deal with these issues the Alpha has:

- load locked and store conditional instructions to allow implementation of locks.
- memory barrier instructions.

Ensuring order on load and store instructions

Alpha provides memory barrier instructions to ensure partial ordering of reads and writes. This means that any load/stores executed before the memory barrier will complete (buffered operations will be flushed) before those after the memory barrier. Otherwise the load and stores can be reordered as per the Alpha architecture ordering rules. It is important to issue a memory barrier instruction after modifying a shared memory region if ownership of it is to be passed to another thread on another CPU. For example, the following sequence needs to be executed when a thread on a CPU gains access to a shared memory region.

```
obtain_lock(X);
memory_barrier();
<modify shared region X>
memory_barrier();
release_lock(X);
```

The first memory barrier instruction ensures that our local CPU buffers contain up to date contents. That is, we aren't reading a value from one of the load/store queues. This prevents any reads inside the critical section from being pre-fetched before the software lock is acquired. The second one, ensures that our changes get written out to the cache before another CPU may attempt to access it. Memory barriers do not ensure that remote CPUs are coherent, unless executed directly on the CPU.

Read-Modify-Write Atomic operations on Alpha

The Alpha architecture defines two important load and store operations that allow implementation of safe modification of variables. These are:

- *ldq_l* - load locked.
- *stq_c* - store conditional.

These are defined to load the contents of a memory address at which stage an internal lock bit is set and the lock address set to the memory address. The store is then conditional on this lock bit remaining set for this memory address. If it fails the operation needs to be repeated. A reasonably small arbitrary number of instructions can be placed between these two instructions, to implement the desired operation. Using this method several lock structures can be implemented.

Example:

```
try_again:
    LDQ_L  R1, x
    <modify R1>
    STQ_C R1, x
    BEQ  R1, no_store     // if store not successful goto no_store
    ...
no_store:
    <code to check for excessive iterations, perhaps a
     thread switch>
    BR try_again
```

In this example, 'x' is the memory address we which to modify. This is loaded into register R1 which is then modified and stored back to 'x' if the lock has been maintained.

This code provides an example of using these instructions to modify a single datum. Similar techniques can be employed to implement other atomic update procedures.

Read-Modify-Write Atomic operations on other architectures

Other architectures such as IA-32 provide atomic instructions that make implementing locks possible. MIPS R4000 provides the load locked and store conditional operations, ARM provides a swap instruction and IA-32 has CAS (Compare And Swap) instructions.

2.3 Multiprocessor machines

With the trend being towards multiprocessor machines, we examine two important types of such machines. In particular, we look at cache coherent machines. These machines maintain a level of memory coherency throughout processors of the system. These are of particular use for multi-threaded kernels.

2.3.1 Symmetric multiprocessor machines

Symmetric multiprocessor (SMP) machines consist of more than one processor, with each processor configured to have identical access to a shared memory and resources.

Current multiprocessor Alphas have been configured in this manner.

2.3.2 Cache coherent non-uniform memory access machines

Current SMP hardware do not scale well for high numbers of processors. Coherency protocols become more complex and memory access times suffer as the number of processors in the system grows.

Cache coherent non-uniform memory access (ccNUMA) systems, such as the SGI Origin [21] provide a coherent view of a distributed memory system. Each local node with it's own local memory is connected via a high speed interconnect such as a hyper cube network of busses.

In the memory system, this means a change from a structure in which all memory is equally close to all processors to one in which some memory is close to each processor, but different processors have different access times to the same memory.

Coherency is implemented in the hardware, requiring software to do little or no work to ensure a coherent view of memory.

Next generation Alphas will be offered in this configuration to support larger numbers of CPUs than conventional SMP systems.

2.4 Structure of L4

L4 [24] is a μ-kernel that implements a minimal set of abstractions with a view that anything that won't compromise security should be moved out of the μ-kernel. There are three major abstractions in L4: address spaces, threads and inter-process communication (IPC). The following sections outline relevant features of L4 that apply to this thesis. For further information regarding L4 see [4].

Address Spaces

Address spaces in L4 are recursively constructed. An address space maps each virtual address onto a physical page or another address space's virtual page. At the lowest level of mappings is Sigma0, which essentially provides mappings of physical pages to subsequent address spaces.

The concept of a task is essentially equivalent to that of an address space. In L4, a task is the set of threads sharing an address space. Creating a new task creates an address space with one running thread.

The following operations are used to create and maintain address spaces.

Grant - The owner of an address space can grant any of its pages to another address space, provided the recipient agrees. This page is then no longer available to the granter.

Map - The owner of an address space can map any of its pages to another address space, provided the recipient agrees. The page remains in the mapper's address space.

Flush - The owner of an address space can flush any of its pages. The flushed page is removed from any address space that has received the page directly or indirectly from the flusher. The owner may also flush the page from its own address space.

Threads

A thread is a basic execution abstraction that is extremely light-weight and cheap to create, start and stop. A thread executes in an address space shared with other threads belonging to the same address space. A thread is linked to exactly one address space and cannot migrate to another.

Each thread in a task has its own Thread Control Block (TCB), where information on the state of the thread is kept. Also, a kernel stack is placed in the TCB so that we can handle thread switch while in kernel mode.

IPC

Inter-process communication (IPC) in L4 is used to pass data by value or by reference. It is also used for synchronisation, page fault handling, interrupt handling and exception handling. IPC is designed and implemented to be extremely fast and efficient.

The IPC is strictly synchronous. Messages are only transfered when both the sender and receiver are ready.

IPC message types can be composed of the following:

Register message - On Alpha up to 8 registers are used to pass the content of a message between two threads. Using only this method is the fastest way to transfer data.

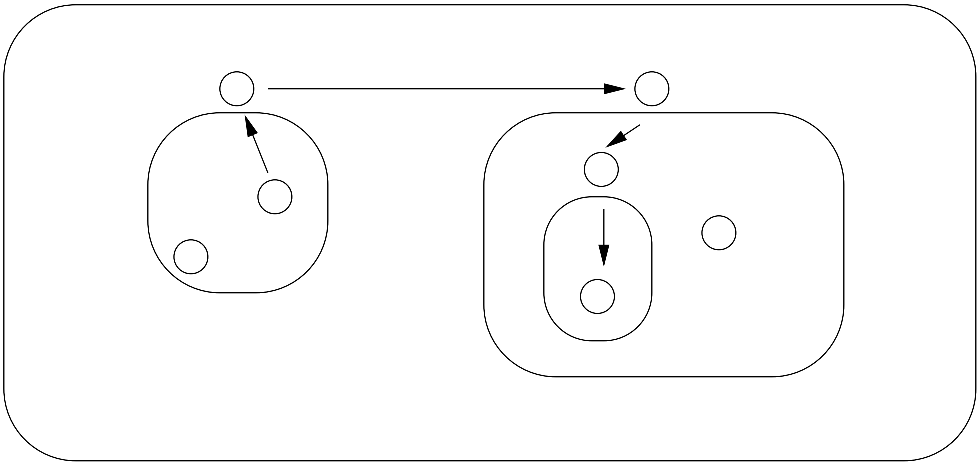

Figure 2.2: Clans and Chiefs
This figure depicts the path of a message as it is redirected to each chief and resent towards the destination.

Flex pages - These are also known mappings. In this case, it is the transfer of entire pages from the sending thread's address space to another. Pages can be either mapped or granted as discussed in 2.4.

Direct strings - Data transfered directly in the message, copied word by word to the address given by the receiver thread.

Indirect strings - Data transfered indirectly via a pointer. The message body contains details on the address and length of the data.

2.4.1 Security

The *Clans and Chiefs* concept implements a mechanism for security. A *clan* is a set of tasks with the creator of tasks in the clan as the *chief*. This is depicted in Figure 2.2.

All messages to recipients within the same clan are transfered freely. Messages to recipients that cross clan boundaries are always redirected to the chief of the border crossing. It is then left up to this chief to decide what to do with the message.

2.4.2 Scheduling

L4 implements a hard priority, round-robin internal scheduler. Thread scheduling is controlled by two parameters: time slice length and thread priority.

The time slice length can be specified for each thread in L4. A time slice length of zero is valid. A thread with a time slice length of zero is taken out of the ready queue and as such is never scheduled.

Thread priority determines which round-robin queue a thread belongs to. Changing a thread's priority moves it to the round robin queue associated with this priority. Threads in a particular round robin queue are only scheduled if there is no thread with a higher priority runnable.

2.5 Previous Work

The relevant work done by Sebastian Schönberg and Volkmar Uhlig is outlined here.

2.5.1 L4/Alpha μ-kernel

The L4/Alpha μ-kernel [31] was started by Sebastian Schönberg who completed a majority of the kernel. This included work done by Volkmar Uhlig on a preemptable mapping database.

The μ-kernel has been entirely written in Alpha assembler for speed and optimisation reasons.

Sigma0 has been written in C and exists entirely in user space.

System initialisation

System initialisation code includes routines to get the system in a state to run L4. This includes related motherboard chipsets and the CPU. This has been done in an Alpha generic way directly supporting the EB164 [12] evaluation motherboard. This included general interrupt initialisation including the timer interrupt set to a rate of 2 interrupts per millisecond.

The L4 μ-kernel is booted into a predefined boot address utilising an evaluation board debugger supplied by Digital. The kernel can also be booted directly by the SRM BIOS [13] supplied with most Digital Alphas.

Page Tables

Guarded Page Tables (GPTs) [23] are efficient for sparsely filled address spaces and is particularly suited to the 64-bit address spaces present on the Alpha.

In the L4/Alpha implementation [30] each Page Table Entry (PTE) is represented by two 64-bit wide entries. The first contains the guard, the second points either to the next level of page tables or is a leaf entry in the tree. Leaf entries contain the word that is written beside the virtual address into the translation buffer. Kernel data contains an array of entry points into each tasks GPT to make access to these fast.

The guard field is a mask which must partly match the virtual address in question. The page size currently used by L4/Alpha is 8KB, so the upper 51-bits are used for the guard.

Protection information is stored in the free lower bits of the guard and is used to specify whether a page is:

- User or Kernel - which mode is required to access the page. This is required as kernel pages are mapped into a user's virtual address space. See Section 2.5.1.
- Executable - instructions on page can be executed.
- Writable - decides whether the page can be written to.

Since translation must take place in PALcode, a PALtemp register is reserved to point to the base of the GPT. Page tables are accessible only via physical memory and hence only available in PAL or Kernel mode. Future versions of L4/Alpha may utilise a virtual address space page table.

Alpha implements Address Space Numbers (ASNs) to tag TLB entries. In the current L4/Alpha implementation the task number is equal to the ASN. The 21164 processor implements 7-bit Address Space IDs (ASIDs) giving a total of 128 ASIDs. This results in inefficient flushing for L4 tasks greater than 128 due to the need to flush such address spaces on every context switch.

Memory Management

Memory management in L4/Alpha provides address spaces in which threads may run.

The following were provided in L4/Alpha:

- Cache, translation buffer and page table management.
- Handling of page faults.
- Mapping and unmapping of pages.
- Creation of entire address spaces (tasks).
- Context switching.

Task deletion was not implemented.

Cache management is handled mostly by hardware. The programmer only needs to ensure that I/O access is synchronised with the caches and that the instruction cache is kept coherent when necessary. This is discussed above in Section 2.2.5. L4/Alpha provides a *callpal* routine to synchronise the instruction cache on a uniprocessor, useful only for self-modifying code.

L4/Alpha supports up to 1024 address spaces, with address space 0 reserved for kernel use. The user virtual address space is structured with the lower region of each space reserved for user mode threads. The 4GB region above this space is reserved for kernel access only. Hence kernel access is defined to be the region starting at 0xffff.ffff.0000.0000. This is depicted in Figure 2.3.

This structure is suitable for several reasons. Firstly, when future Alpha processors support larger virtual address spaces, no changes will need to be made. As discussed above, current Alpha processors support 43-bit virtual address spaces with newer ones supporting up to 48-bit spaces. Secondly, this provides for a very large user address space, of 8TB.

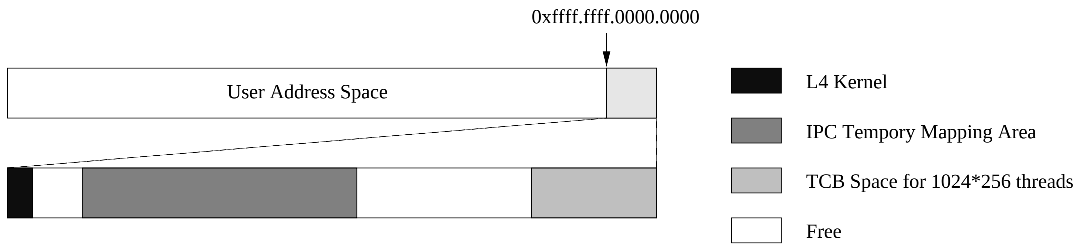

Figure 2.3: User Virtual Address Space Structure

Within the kernel reserved region of the user virtual address space is mapped the kernel code, a temporary mapping area and the TCBs for all possible threads in the system.

Mapping Database

The Alpha mapping database [32] provides a structure to allow high concurrency, fast access and time guarantees for real-time systems.

The mapping database is used to maintain information about mappings of pages from one address space to another. Such mappings originally belong to exactly one address space, but can be transferred to another address space with less or the same rights. L4/Alpha also supports mappings to the same address space, which can be disabled.

The information stored in the mapping database is particularly useful for the flush operation. This operation is responsible for revoking mappings of pages that exist from one address space to another. Operations on the mapping database have been implemented in a non-blocking fully preemptable way using no recursive algorithms (where appropriate) in order to provide real-time guarantees.

A more detailed outline is discussed in [32].

Thread management

The current L4/Alpha port provides support for a fixed number of 256 threads in an address space. Thread IDs are 64-bit and are structured as shown in Figure 2.4.

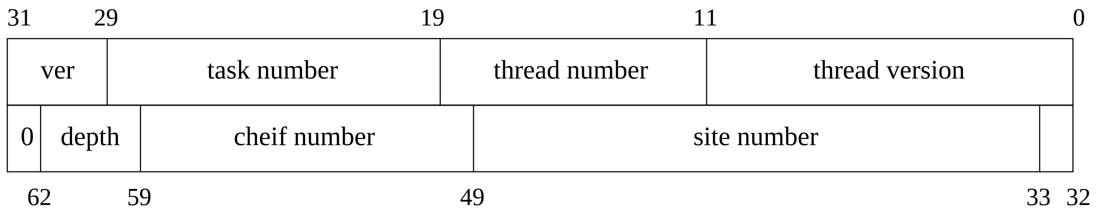

Figure 2.4: L4/Alpha 64-bit Thread ID

Each thread has it's own *thread control block* (TCB) shown in Appendix A, where state information on the thread is kept. Each TCB also contains a kernel stack to aid preemption while in kernel mode. TCBs exist in an array in virtual memory. Each TCB is 2048 bytes so that it can support the kernel stack including possible floating point registers storage.

This version of L4/Alpha initialises the TCBs and thread IDs, but does not have direct support for clans and chiefs, hence these bits are left blank in the thread ID. Furthermore, this version provides the *l4_thread_ex_regs()* L4 syscall, however does not correctly implement thread aborts. That is, upon thread state changes that affect other threads, thread operations are not aborted. L4 defines thread aborts to happen in certain cases such as change of thread's stack or instruction pointer and includes operations to abort any threads associated in IPC with this thread.

Inter process communication

Short and long messages are implemented including flex page (mappings) support.

Three types of flex pages are supported including virtual memory, physical memory and I/O memory. The last two are only allowed directly for Sigma0. Since Alpha uses memory mapped I/O the last two are also considered equivalent. The smallest flex page size corresponds to the smallest supported Alpha page size of 8KB.

Flex pages can be mapped, granted, set read only or read write. There is currently no support for an execute bit.

The fast path of IPC executes in PALmode and as such is not interruptible. Other sections such as direct string copying in the long IPC path are executed in kernel mode and are interruptible.

Correct message dope return values are not implemented. Clans and chiefs support is not implemented. Timeouts don't work as expected. This is discussed in Section 4.4.

Kernel Debugger

Supplied with L4/Alpha is a kernel debugger capable of displaying information on each thread including mapped memory, thread control block (TCB) information, registered device interrupts and other useful information.

This debugger is invoked either directly in user code via a *callpal* routine or internally by the kernel in panic states.

Chapter 3

Multiprocessor Design and Implementation

The motivation of this chapter is to outline a highly scalable L4 μ-kernel targeted at multiprocessors. In particular target Alpha SMP and ccNUMA machines.

Non-cache-coherent NUMA multiprocessors are not considered in this thesis due to the complex memory management required in both kernel and user space. These would require different abstractions about the system than those considered in this thesis.

There are several issues that we need to be aware about to aim for near linear scalability.

- TLB, memory and hardware cache coherency.
- Methods of implementing critical sections.
- Appropriate choice of locking methods.
- Bus traffic between processors.
- How to communicate efficiently between processors.
- Structure of L4 internals, particularly the scheduler.

In order to produce the desired outcome, the rules are simple:

- Minimise data sharing between processors.
- Fine grained data structures where appropriate.

This chapter discusses the relevant L4 kernel internals that need to be modified and outlines proposals for implementing them. In particular this includes scheduling methods, Inter-process communication, memory management and methods for handling L4 internals in a safe and efficient manner.

3.1 Scheduler

The desire is to have a scheduler that does not break the definition of a scheduler in L4. That is, a hard priority round-robin scheduling of threads. It is also desirable to maintain the flexibility of the uniprocessor scheduler so that users can implement their own scheduling policy.

3.1.1 Possible scheduling techniques

There are several ways that L4's internal scheduler can be structured. In choosing an appropriate method it is important to consider issues including flexibility, scalability, speed and the performance impact the scheduler may place on the system.

In general, the method used for scheduling can be chosen from the following:

One scheduler per processor, sharing a global ready queue

This can work effectively when carefully implemented, however has a high impact on the caching subsystems.

Mach uses a global ready queue for scheduling, however also has a queue for each processor. Each thread is responsible for maintaining state information regarding how it is scheduled. Periodically this information is checked and the local processor queues are updated. This method could be used to make using a global queue more efficient, however there is no need to place this policy inside the μ-kernel.

It restricts possible scheduling algorithms that the user can implement. Providing mechanisms to allow such flexibility will add complexity that isn't desirable.

It is also possible that when threads are re-run they frequently get cold caches. This results from lack of process affinity. Adding a feature to support process affinity will again add complexity.

Questions also arise on how to deal safely with scheduling threads of different priorities across multiple processors. The correct method would be to only schedule threads of the same priority, potentially resulting in idle processors.

One scheduler which schedules a thread per processor

This method solves problems of concurrent access to a global ready queue, however results in large amounts of time consumed on the scheduling processor. This may provide a large bias against threads scheduled on this processor.

Methods of interrupting other processors and notifying them of the next thread to schedule is a minor problem. This may be as expensive as the previous methods suggested above due to implementation issues. Such issues should be left up to the scheduling method selected by the user.

As discussed above, we have to deal with a safe method of scheduling threads of different priority.

One scheduler per CPU, each scheduler has its own queue

This method solves problems of having to share a global queue by implementing completely separate queues. In terms of scalability, this is the most efficient method. We no longer need to share data related to scheduling in the kernel.

Issues arise as how to schedule threads of different processors, how to load balance and how to deal with priorities.

This is the chosen method of multiprocessing L4, discussed below.

3.1.2 Scalable scheduler

The proposed solution for efficient scheduling is to have one scheduler per processor each with their own queue of threads. In essence each scheduler behaves like the hard priority round robin scheduler of uniprocessor L4. They do not interact with each other or schedule threads that exist in other scheduling queues.

Each processor is given an numeric ID so that it can be identified in some way.

In order to deal with the desire to execute threads on a different processor we provide a migration mechanism discussed below in Section 3.1.4. The scheduling mechanism leaves other scheduling methods such as load balancing to user level schedulers.

This method solves several potential problems that other scheduling schemes may have including complex routines and implementations that slow the μ-kernel down and limit scalability. For example, scheduling from a global queue has the potential to generate a lot of bus traffic between processors as they attempt to keep their cached copy of the queue up to date via internal cache coherency protocols.

Sharing a global scheduling queue also implies that each processor needs to first lock the queue when making modifications to it such as scheduling a thread. Since this method doesn't have a global scheduling queue no locks for queues is required.

The internal schedulers do not place any significant amount of overhead on the system with respect to the number of processors (hence schedulers) present or the layout of processors and memory of the system.

Users cannot make assumptions about thread execution across the whole system in relation to priorities. As stated above, L4's hard priority round robin scheme is limited to each processor.

For example, while a high priority thread is executing, no lower priority thread will be scheduled. This is true per processor but cannot be assumed when such threads are running on separate processors. This scheme does not break current uniprocessor applications. They are not aware of multiprocessors and hence won't utilise it.

3.1.3 Processor restriction

It is desirable to limit the range of processors a new task has available to it. For example, we may wish to limit a certain task to only a small range of processors. This prevents a task scheduling a thread on a processor it has no privileges for.

As a result, processor resources are based on the same policy as task rights that are passed on from a task's chief.

In order to implement this, each task has a bit array of processor ID. When set, access is permitted. As for the maximum controlled priority of a task, these permissions can be no greater than the creator of the task. This information is set when a task is granted, specifying it's processor rights for it's thread and subsequent child tasks.

For a 512 way multiprocessor this results in a one page (8KB) array to store these permissions for 1024 tasks.

L4 relies on hard priorities to make assumptions about what can be scheduled at any one time on a processor. Assumptions can be made that no thread with a lower priority will be scheduled while there is a thread with a higher priority in the scheduling queue. As a result, we can argue that limiting priorities of certain threads, we can do away with this feature. The user utilises responsible priority methods and user level schedulers. Efficiency is a major reason for the desire to have such a policy.

3.1.4 Thread migration

Threads can either be bound to a processor (default) or unbound. Threads that have time slice lengths of zero are removed from their assigned processor scheduling queue and are then free to accept time donations via the l4_thread_switch() syscall. This allows a user's implemented scheduling method to be more efficient as the need to explicitly migrate a thread is not there, however the cost of context switching overhead associated with time slice donation needs to be considered.

Time donation to a bound thread is equivalent to a yield, unless time donation occurs on the same processor.

For bounded threads migration is left up to the user. It can be performed explicitly via a call to l4_thread_schedule() specifying the appropriate ID of the processor to migrate to. The thread then gets removed from the current schedulers queue and then added to the new scheduling queue if the right permissions exist.

The other alternative is to give threads a zero time slice, in which case they do not exist in any runnable scheduling queues. Hence calls to l4_thread_schedule() are not necessary, rather time slice donation is all that is needed.

Normally each L4 thread is assigned to a particular processor (a particular internal scheduling queue). This means that when Linux's [35] scheduler (for example) or another scheduling method that wants to run a thread on a processor that it is not currently assigned to, it must first migrate it to the processor that it wants to execute on. For this reason, we allow threads that are going to be managed by user level schedulers, not to exist in any particular queue. Note however, that a good multiprocessor scheduler implementation would use a method to similar to that used with Mach. This is, it will use L4's internal scheduling queues to schedule a short list of threads.

3.1.5 Multi-threading CPU support

Simultaneous multi-threading on future Alphas will allow multiple threads to compete for and share available processor resources every cycle. A processor's pipelines are divided up and utilised as if they are separate processors. This results in more efficient use of processor pipelines. This is discussed in detail in [26].

To schedule on such processors we simply set a series of program counters depending on the number of threads we wish to schedule at the same time and the number of threads the processor can schedule at the same time.

Multiprocessor L4 will schedule threads only of the same priority on such systems. This is for several reasons. Firstly, allowing multiple priorities to run at once would require existence of multiple schedulers as defined above for multiple processors. This abstraction should not be carried out due to the extremely tightly coupled nature of such a processor.

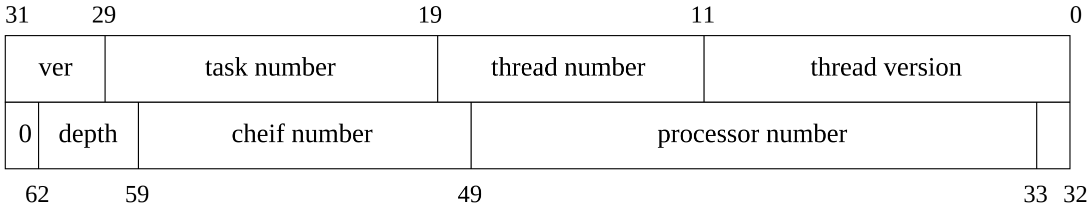

Figure 3.1: Multiprocessor L4/Alpha 64-bit Thread ID

3.1.6 Internal Thread ID changes

Task and Thread IDs have been changed to show information regarding CPU ID as shown in Figure 3.1. This Figure shows how the thread ID is stored internally in the TCB of the thread. The *site* field of uniprocessor L4 has been replaced with a processor number. Strictly speaking this information is not required by the user but will be present in syscalls such as l4_myself(). When thread IDs are used as arguments, this field is ignored.

3.1.7 API changes

In order to facilitate scheduling of threads and tasks on different processors, the following extensions have been proposed to the L4 API.

l4_thread_schedule() adds an extra argument specifying the CPU ID of the CPU to bind this thread to. This call can be made to a thread that is currently inactive (i.e., has never been started).

l4_task_new() adds an extra argument specifying the CPU ID of the CPU to bind this task (namely lthread0) and acts as a default for other threads in this task.

Other L4 syscalls do not require any modification. Although it may be nice cosmetically to add a CPU ID argument for l4_thread_ex_regs(), this syscall should remain as cheap as possible.

3.1.8 Examples

The aim of this example is to show that the proposed model supports implementation of arbitrary scheduling methods on top of a multiprocessor L4 μ-kernel. The following example shows a range of schedulers working across a small multiprocessor system. These schedulers are implemented as user-level schedulers on L4. Several operating system personalities are running on the system, utilising their own scheduling policy.

These examples encompass a single 8 CPU SMP system that we wish to have running a gang scheduler across all eight CPUs, Linux [28] running on four CPUs and Mungi [19] running on one CPU.

This system is set up such that all of the user-level schedulers run at the same priority to ensure that neither the gang schedulers or operating system schedulers suffer from CPU starvation while both are in use.

Gang scheduling

Gang scheduling schedules a set of related threads to run on a set of processors at the same time. Scheduling threads to execute at the same time reduces the overall time spent waiting for locks in most cases.

The gang scheduler is utilised by Linux and Mungi to schedule special jobs via a library interface. Each thread set is placed onto a scheduling queue. When the gang scheduler's user level schedulers gets scheduled, they examine this queue for the next set of threads to schedule together. At this stage the threads get scheduled via time slice donation and execute concurrently until their time slice expires. Implementation of such a scheduler should probably use a higher priority than other schedulers to ensure threads get scheduled at the same time.

Mungi scheduling

Mungi relies on the priority based round robin scheduling of L4. As such it can use the internal schedulers directly.

It has not been decided how Mungi plans to manage multiprocessor scheduling, so for this example we restrict Mungi to a single processor. Of course we could implement a load balancing scheduler under Mungi and run Mungi on several processors without modification, or extend the Mungi API much the same way as the L4 API has been done.

Linux scheduling with load balancing

The scheduler in Linux schedules threads from a queue of runnable threads. We can implement this load balancing scheduler on top of L4 also.

One way to achieve load balancing across a few processors is to let each of L4's internal schedulers schedule threads and occasionally migrate threads via explicit l4_thread_schedule() calls. An idle thread can be utilised to monitor the load in the advent that other user level schedulers may be running on this system also. This method is good for achieving a high level of system performance as the extra context switching overhead of user level schedulers is minimised.

Another way to achieve load balancing is by direct management of threads. A global scheduling queue can be used from which each user level scheduler running on each processor picks a thread to schedule. These threads can be more efficiently managed by setting their time slices to zero so that explicit thread management by the kernel is not required. This method works best on systems with a small number of processors due to the penalty of sharing a single queue.

Overall system

These systems are started up by an initial server which sets the priorities of each user level scheduler as appropriate and then starts up each operating system personality on a designated set of processors. Processor rights are set accordingly to prevent Linux, for example, from scheduling threads on the processor that Mungi is running on.

When Linux or Mungi have jobs they wish to gang schedule, they program this via the library interface which then sets the job to run. The library interface may implement gang scheduling via IPC to local processor schedulers.

3.1.9 Summary

Efficient scheduling of threads across a multiprocessor system is of vital importance for system scalability. The scheduling techniques discussed above should provide a good hardware abstraction layer that will not hinder the implementation of user policies nor result in system performance degradation when different techniques are used across a system. Obviously optimisations of these techniques will result from actual implementation and benchmarking.

3.2 Inter-process communication

The speed at which IPC is performed is critical for a good μ-kernel. With multiprocessor machines there are several more circumstances that can lead to performance penalties when performing IPC. Both writers of multiprocessor versions of L4 and the users of the system should be aware about such issues.

It is no longer the case that IPC always occurs on a single processor. We have to deal with handling IPC between threads on another processor that can not be performed via L4's conventional IPC method. Uniprocessor L4 uses a set of reserved registers to transfer a message on a context switch. In the multiprocessor implementation it is possible to transfer the message using the same technique by migrating the receiver to the senders CPU (or vice versa) for the message transfer. This would result in the cache coherency mechanisms taking care of the memory transfer once the thread is rescheduled on its assigned processor. However it is advantageous to allow more flexibility than this and consider alternatives for IPC when threads exist on different CPUs.

The speed at which IPC is performed is important. It is particularly effected by the hardware cache coherency mechanisms (see Section 2.2.5). This includes inherent delays caused by having data migrate from one cache to another. The method for synchronising the threads doing an IPC operation also needs to be optimised well.

On ccNUMA machines with coherency protocols and memory transfers having extra overhead these issues become even more important.

For short IPC (a register transfer of 64 bytes), we can perform the IPC operation via a memory region, such as the receiver stack (in their context).

The method chosen for message transfer is not particularly important in the case of short messages. As stated above, eventually cache coherency mechanisms will take care of dirty data. This results in temporary migration of threads to the same processor having little difference to a direct memory transfer.

In terms of implementation it is important to consider the overall impact certain methods will have on a given system configuration and actual ease of implementation. As a result, pushing/popping messages off relevant thread stacks is likely to be the chosen method (as chances are, the message is already on the senders stack).

To implement optimal IPC paths, benchmarks will have to be performed to work out the best solutions. Currently, due to a lack of data, this is not possible. For example, utilising a memory region for shared data is likely to be cheaper than relying on cache coherency protocols.

The IPC path can no longer be assumed to be single-threaded and needs to be made re-entrant. Fine grained locking methods are used to assist in performing IPC with a low impact, as discussed in Section 3.4.3.

Here we have a simplified solution that would solve the problems related to ensuring atomic operations on TCBs when IPC operations are taking place.

```
SENDER:
    <prepare ipc>
    obtain_lock(R)
    if receiver waiting (for me)
        change state of receiver to busy
        release_lock(R)
        perform IPC
    else
        enqueue in queue
        release_lock(R)
        sleep
    endif

RECEIVER:
    obtain_lock(R)
    if open_wait
```

```
        if someone waiting
            change state of receiver to busy
            release_lock(R)
            perform IPC
        else
            release_lock(R)
            sleep
    else if sender waiting for me
        change state of receiver to busy
        release_lock(R)
        perform IPC
    else
        release_lock(R)
        sleep
    endif
```

'R' represents receiver's lock granting exclusive access to the whole TCB. A more fine grained approach is possible, see the fine-grained TCB locking Section 3.4.3.

"Perform IPC" is the operation to actually do the message transfer. This involves either the message transfer via a memory region or by executing the receiver on the sender's CPU for the duration of the IPC call. This operation needs to ensure it manages the state of the receivers TCB correctly.

The "sleep" operation inserts the thread's timeout into a per processor queue.

The "enqueue" operation inserts the timeout of the sender into the receiver's send queue. This is implemented via an atomic operation as discussed in Section 3.4.3.

We define atomic IPC operations with a success or failure result. The lock in the above pseudo code simulates this. Once the lock has been obtained to perform the IPC, it can assumed to complete, particularly for short IPC. In this case each processor can be considered to be operating un-interruptible as in the uniprocessor implementation. Similarly long IPC operations that normally execute in kernel mode with interrupts enabled may be cancelled as for the uniprocessor implementation.

For a neat implementation separate methods for performing short and long IPC are used in the most efficient way possible. While we can provide the user with a set of syscalls depending on how they want the IPC delivered, there shouldn't be any significant need to allow the user to specify which method to use as this leads to a more complex IPC path. The aims are to perform IPC as fast as possible. This includes minimising caching impacts and overall IPC latency due to the introduction of memory barriers to flush load and store queues and inter processor interrupt latency (if this is used).

3.2.1 Interrupt handling

Threads that register themselves as interrupt handlers for a particular interrupt must be bound to a particular CPU. It is possible for this thread to move to another CPU and take the interrupt with it if the system supports such an action.

3.3 Memory management

Memory management is one of the most important areas in multiprocessor kernels. The ways in which processors share and manipulate data can have a huge impact on the μ-kernel. This includes management of address spaces, mappings, page tables, and tlb entries.

In this section, several memory related issues and solutions are presented.

3.3.1 Address space management

Coherent mappings particularly in the TLB of each CPU need to be maintained on a multiprocessor system. Problems can result from page flush operations as these require any valid mappings for a page to be invalidated from all TLBs On a uniprocessor system (and multiprocessor) the mapping database maintains the bookkeeping information required to successfully perform flush and unmap operations.

In multiprocessor systems we have to ensure address space coherency across the whole system. There are two methods for handling address spaces that can be considered for multiprocessor systems. Either we allow only one address space per processor, or we allow address spaces to span processors. The discussion that follows relates to performing flush operations on one processor which also needs to ensure that other processors are in a coherent state. This is a major concern for address space coherency.

One address space per processor

Restricting address spaces to a single processor avoids the problem of having mappings for flushed pages in remote CPU's TLBs. This is the method utilised by several SMP kernels, however it is not really suitable if we wish to provide a good multi-threading semantic. For example this would restrict gang scheduling. In this case we would have to create separate address spaces for each thread (per CPU) we wish to schedule at the same time and then share memory pages in each. The mapping database provides the required book keeping by tagging each page with a CPU identifier. This makes flushing straight forward, however using this method for gang scheduling is clearly not very appropriate.

Multiple processors per address space

The desired use of address spaces in an multiprocessor version of L4 is not to restrict them to single processors at any given time. This poses a challenge and creates potential for serious performance penalties in L4 due to management of the address space mappings for each processor. We can consider the following solutions:

- Interrupting each CPU and flushing the required page or ASID (upon task deletion) in its TLB. This method is probably acceptable assuming flushing is rare, however it would be extremely inefficient particularly on larger SMP systems when every time a flush operation occurs we have to interrupt each CPU.
- Maintaining state information for each mapping in each TLB. This allows us to have some knowledge about each page that has been mapped in the TLB. However this method relies on us recording the state information upon each TLB refill which makes this operation a lot more expensive, particularly due to the frequency of this operation compared to flush operations.
- Utilise the scheduling mechanisms to record a set of processors that may have mapped pages for a particular task ID and only interrupt and flush required pages of the specified processors. This option is discussed further below.

The overall choice to be made for an implementation requires us to consider how to manage the resources effectively. The following need to be considered:

- Bookkeeping per page or per task ID and the cost of reading and writing such structures.
- Cost of interrupting a set of or all CPUs.
- Life of a TLB entry for a given mapping.
- Cost of probing bookkeeping information for each processor to determine if it needs to be flushed.

We wish to minimise the cost of maintaining the required information and consider how suitable such maintenance may be based on cost. For example, we can consider recording existence of a TLB mapping during each TLB fault to be a severe penalty in performance for a rarely performed flush operation. As such, maintaining state information for each page is not a viable solution.

The aims for the flush operation are to maintain a minimal set of state information, only interrupt CPUs which are highly likely to have a TLB mapping for a flushed page and to store bookkeeping information in such a way that is fast and efficient in relation to the hardware cache coherency mechanisms. This should allow for a minimal impact on the system for ensuring a required level of TLB coherency.

TLB coherency management

We can utilise the design of the internal L4 scheduler to efficiently maintain the desired state information we wish to have in order to provide for a level of TLB coherency in an multiprocessor system.

For each processor we maintain a list of task IDs of threads that have executed on this processor. When a flush operation is executed we first remove the flushed page's mapping/page table entry and flush each required CPU's TLB based on the state information maintained by the scheduler. This operation does not need to be made atomic as no further TLB entries will be added on remote processors once the mapping is removed from the page tables.

When we do a flush operation on one processor, we need to make sure that no other processor has a valid TLB entry for the removed entry. Inter-processor transactions as discussed in Section 3.2, can be used to signal other processors the need to do something, in this case flush certain TLB entries.

This proposed method allows us to use the current L4 mapping database without any multiprocessor specific changes. Further more it provides a reasonable solution in efficiently determining a set of processors that may require TLB flushing while ensuring that all TLB mappings for an invalid page are removed.

3.3.2 Address Space ID management

The Alpha architecture [8] defines the maximum number of ASIDs to be 65536 (16-bits). Currently the 21264 Alpha processor supports 8-bit ASIDs. L4/Alpha supports up to 1024 address spaces active concurrently, although this number may be increased. It is possible that the number of active address spaces is greater than the number of supported ASIDs, as such we need to provide a method for efficiently dealing with this case.

The solution is to provide ASID recycling similar to that discussed in [15]. When we reach a state of no free ASIDs, we use a method of FIFO replacement (or other suitable strategies) to select and reuse a currently allocated ASID. This method can be used for both uniprocessor and multiprocessor systems.

For a multiprocessor implementation we have the choice of sharing the ASIDs across the whole system (having a single mapping from task id to ASID for all processors), or allocating them locally per processor (there exists an ASID per processor for a particular task that may not be equivalent to each other, and have no meaning outside the realm of a processor and local ASID assignment entity). Choosing the second method provides greater ease of scalability by ensuring that there needs to be no negotiation of ASIDs across the whole system. It also reduces the overall amount of ASID recycling that would occur otherwise.

The internal L4 scheduling routines take care of assignment and management of these ASIDs. For each processor we maintain a 'one bit per task' task bit-field to indicate if a page of this address space has been accessed (indicating the possible need to flush the TLB for this task eventually). It also maintains a mapping table for each of task to an ASID. When a thread gets scheduled (or at other suitable times), it's address space is marked as being accessed in task bit field. An ASID check is also done, and if it has not been assigned an ASID, this is performed also.

3.3.3 Page Tables

Cristan Szmajda has proposed a new page table structure, the *level-and-path-compressed trie* (LPC-trie) [34]. This structure adapts to both sparse and dense address distributions. It is planned that this page table structure will be used in this multiprocessor implementation.

Page table coherency will be dealt with using similar methods as discussed in Section 3.3.1.

3.4 Atomic state changes

In the uniprocessor implementations of L4 we are able to exploit the fact that while interrupts are disabled, critical sections are safe. In multiprocessor kernels, we can no longer assume that disabling interrupts on one processor is enough to ensure that a critical section will not be entered, particularly by code executing on another processor.

To provide correctly implemented critical sections, we need to rely on other methods such as use of semaphores to ensure that only one thread of execution can enter a critical section at any one time.

The methods chosen to safely protect each critical section are extremely important for system efficiency. An analysis has been performed on system structures that are shared between processors and has shown that in the majority of cases locks are of low contention. Furthermore, with correct sequencing of code, critical sections are typically very short. Benchmarking will later optimise the approaches chosen.

3.4.1 Locking methods

Using the methods and instructions discussed in Section 2.2.5, we can implement several locking methods to safely protect critical sections.

At this stage, it does not appear that there need to be locks that are held for extended periods of time, or any major situations that may result in deadlock. With systems that employ fine-grained locking, there is always a risk that deadlock may occur.

The following methods are used to implement critical sections in certain cases, depending on the type of data or code sequence being protected.

Method 1 - multiple data protection

This method relies on a lock *'x'* to protect a critical section. This method is flexible enough to protect any section of code of reasonable execution length, however it is desirable that such critical sections are kept as short as possible.

```
method_1:
    obtain_lock(x)
    memory_barrier()
    <critical section>
    memory_barrier()
    release_lock(x)
```

As discussed in 2.2.5, the memory barriers ensure coherent data is being viewed.

obtain_lock() is implemented as follows and assumes that in the common case the lock is not being held:

```
obtain_lock:
    R0 = load_locked(x)
    if R0 != 0                          // if lock being held
        goto obtain_lock
    R0++
    if store_conditional(R0,x) == fail
        goto obtain_lock                // try again

release_lock:
    R0 = load(x)        // we already have coherent value
    R0--
```

```
    store(R0.x)
    memory_barrier      // send it to the caches
```

release_lock() can be implemented with the load locked instruction to avoid the explicit memory barrier, however this appears misleading.

Method 2 - direct datum modification

In this method we want the ability to modify a single datum until successful, recalculating the critical operation if necessary. We exploit the atomic load/store instructions (Section 2.2.5) to implement fast critical sections, using the datum itself as the lock.

```
method_2:
    R0 = load_locked(x)
    <critical section operates on R0>
    if store_conditional(R0,x) == fail
        goto method_2
```

Method 3 - direct data modification with abort procedures

This method is simply an extension to method 2. Here we provide the ability to yield our time slice or abort after a certain period of time. Typically this code is used when we risk spinning for long periods of time. Back off is a technique used to wait for extending periods of time if we continually fail to obtain the lock.

```
method_3:
    R0 = load_locked(x)
    <critical section operates on R0>
    if store_conditional(R0,x) == fail
        <perform back off/abort check>
    else
        goto method_3
```

3.4.2 Inter-CPU transactions

Sometimes there is a need to do more than simply lock and modify a critical section. In certain cases we wish to modify internal processor data on another processor. For example, notifying a processor's scheduler to schedule a thread that is waiting on an IPC.

In order to invoke operations on remote processors, we define a series of transactions. Currently we have transactions for:

- Inter-processor IPC as discussed in Section 3.2. Such a transaction is used to signal a processor that an IPC related operation needs to be invoked.
- Scheduler operations, namely busy queue insertion and removal.
- Flush operations as discussed in Section 3.3.1.

We define two functions to manage transactions:

allocate_transaction() - Grabs a small memory block in which the transaction contents can be stored in. This structure contains details of the operation to be performed, the required data and is placed in a predefined location that the recipient of the transaction knows where to look.

free_transaction() - This function simply returns the allocated memory back into the kernel free pool.

Choosing appropriate methods to implement these transactions can be done from one of the following:

1. Inter-processor interrupts - when we wish for another processor to do something, we send it an interrupt and have it process the transaction immediately. This method is too disruptive to normal scheduling operations and should be used wisely.
2. Per-processor queue - each processor has a transaction queue that is parsed on timer interrupts to see if any operation needs to be performed.

Lock	Protects	Implements
TCB_THREAD_STATE	TCB_COMM_ADDR, TCB_COMM_PARTNER	IPC in progress state operations
TCB_LIST_STATE	TCB_LIST_STATE	Atomically update thread list states.
LOCK_TASK_STRUCT	TCB_CHILD_TASK, TCB_SISTER_TASK, TCB_PRESENT_QUEUE	Task creation and deletion, maintained in lthread0.
TCB_THREAD_NEXT	TCB_THREAD_NEXT	Active thread queue.
LOCK_SEND_ROOT	TCB_SEND_ROOT, linked TCB_SEND_QUEUEs	Send queue operations.
LOCK_EX_REGS	TCB_PAGER, TCB_PREEMPTER, TCB_SCHED_PARAM, TCB_KSP, TCB_RESTART	*l4_thread_ex_regs()* state information.

Table 3.1: Locking states of TCB
The lock is used to protect certain TCB entries as shown.

The second choice appears more reasonable, implemented using the atomic operations discussed in Section 2.2.5. Further more, it is not as disruptive as the first case, allowing transactions to be processed based on priority. The series of transactions defined above work well with this case. None require immediate handling of operations, however some may be prioritised.

If the transaction is to be performed on the same processor, optimisation can be made to avoid locking.

3.4.3 Fine-grained TCB locking

The *thread control block* (TCB) maintains the state information of each thread. In a multiprocessing system we can not assume that state changes in the TCB are atomic. Table 3.1 shows sets of TCB elements that need to be modified atomically to ensure correct thread manipulation. The *lock* defined to protect a related set of TCB elements is either a TCB element or a newly defined lock.

Table 3.2 shows several of the operations that we wish to perform on the TCB safely.

Operation	**Method**	**Sequence**
IPC enqueue operations (insertion of thread into receiver's send queue)	Method 3	Operation relies on successful change of TCB_THREAD_CHANGE before proceeding modification of any other variables. Any failures require a check to ensure operation can be retried.
Insertion/Removal from a send queue	Method 1	Use LOCK_SEND_ROOT as lock, critical section involves traversing queues for insertion and pointer manipulation for insertion and removal operations. It is important to note that the lock exists in the TCB of the thread whose queue we are adding to. No other linked TCBs need to be locked, as they too will be required to obtain this lock.
TCB_THREAD_NEXT insertion/removal	Method 2	Use datum as lock, critical section operates on contents depending on operation

Table 3.2: Safe TCB operations
Examples of appropriate methods to use for certain operations.

These operations may be invoked from kernel code including the IPC path, task create and delete operations and thread manipulation operations.

Several TCB elements are only accessed by the thread itself, as such do not need locking. More importantly, certain TCB elements are locked implicitly by locking other TCB elements or the appropriate lock first. In these cases, operation ordering needs to be carefully adhered to.

3.5 Summary

This chapter has highlighted and presented solutions to several problems that we encounter when attempting to provide a scalable high performance version of L4 for multiprocessors. It is hoped that the methods presented here provide a view that conforms with the goals of uniprocessor L4 design.

With the research covered here, I feel that implementation can begin on providing a scalable μ-kernel for Alpha multiprocessors. During implementation it is expected that some minor implementation issues and optimisations will be worked out. At this stage it is desirable that the polices presented here have general acceptance of L4 implementers.

Chapter 4

Uniprocessor Implementation

This chapter discusses the relevant implementation details of the uniprocessor version of L4/Alpha that I have contributed. For previous work on this μ-kernel, see Section 2.5.1.

Implementation using someone else's code is a particularly time consuming job. This involves understanding small blocks of code, data structures and how they are tied together. A lot of time has been spent on doing this and in particular examining what works correctly, what needs fixing and what isn't implemented. Optimisation of implemented code has been performed for critical code segments, such as those that affect the "fast path" of IPC.

I have attempted to make a tool base for managing and loading the kernel across a range of different systems. These are commodities which make the kernel manageable in a real environment. A more complete tool set needs to be worked on this is discussed below.

4.1 Thread manipulation

In this section changes to relevant thread related procedures is discussed.

L4/Alpha lacked a correct and complete l4_thread_ex_regs() system call. This included lack of support for clans and chiefs and a correct implementation of thread abort code.

The rest of the l4_thread_ex_regs() code involves initialising and/or returning of thread state for a particular thread which was previously implemented so that threading is supported. Most of this information is placed in the TCB (see Appendix A). This syscall is fairly light weight, except for the thread abort code. This syscall is discussed below.

4.1.1 Soon wakeup queues

When a thread attempts to perform an IPC on another thread that is not ready, the thread is placed with a given timeout value into the soon wakeup queue. On each timer interrupt, this queue is checked for any threads whose timeout has expired. When this happens, the thread's IPC operation is aborted.

The soon wakeup queue insertion and removal methods have been re-implemented for more optimal usage of this queue. Threads are now inserted in this queue sorted on timeout. As a result, the whole queue does not need to be traversed when checking for threads that have timed out. Having an unsorted queue results in far too much time spent traversing the list on each timer interrupt. At the cost of inserting into a sorted queue, we get an overall system performance increase. A more fast-IPC-path friendly method would be to define several unordered queues and insert threads into a particular queue based on their thread ID. This method is discussed further in [22].

Previously, timeouts where not handled correctly due to a bug resulting in threads blocking endlessly. My implementation provides easy methods for removing and inserting threads that is useful for IPC, thread aborts and timer interrupt code.

4.1.2 l4_thread_ex_regs()

Several changes were made to this syscall to include support for the clans and chiefs mechanism. In order to support this mechanism, correct and complete generation of thread IDs was added as discussed below.

Correct thread abort code has been added to this syscall.

Thread ID generation

Correct thread IDs are required for the clans and chiefs security model. In particular correct generation of *depth* and *chief* bits (see Figure 2.4) in the thread ID. The *depth* bits specify the level of recursive clans that the thread is in. The chief bits are equal to the task number of the creator of the task a thread is in.

This information is easily obtainable at the time of task creation and used each time a new thread's TCB is generated. This information containing the complete thread ID is placed into the TCB_MYSELF entry inside the TCB. While the depth and chief fields not strictly required by the user, they are easily obtainable using the l4_myself() and other appropriate syscalls as part of the thread ID.

Thread abort

When the state of a thread is changed using l4_thread_ex_regs(), particularly its instruction pointer or stack pointer, any current operations on this thread need to be aborted. This includes IPC operations that are in progress or any other state that should result in an abort.

Testing the l4_thread_ex_regs() syscall in the test suite (see Section 4.9) revealed that threads performing or waiting to perform IPC to the target thread were not aborted correctly. This code was only aborting the thread that l4_thread_ex_regs() was performed on, which is not required (nor does it make sense).

If l4_thread_ex_regs() is called on a thread, any other thread must be aborted if it is in one of the following states with respect to the target of the l4_thread_ex_regs():

Pending threads or waiting threads, are those that are waiting to receive an IPC from either any thread, a specific thread or an interrupt. We don't need to abort these threads (because there aren't any) but the l4_thread_ex_regs()'ed thread may be in this state.

Polling threads are those that are waiting to the target thread. These threads are linked in the send queue of the desired receiver.

In-progress IPC - a message transfer in progress.

In order to correctly clean up in this operation we need to remove threads from the appropriate queues that are affected. This includes unlinking polling threads from send queues and timer wake up queues. We also need to ensure that the thread being l4_thread_ex_regs()'ed is not in any queues, particularly soon wakeup queues (see Section 4.1.1 for a description of *soon wakeup queues*).

Threads that have been aborted need to be cleaned up correctly. This means, we have to unwind the thread's stack to the correct position, which can be messy in some cases. Routines are defined in the IPC path code to handle this. Failed IPCs that have been aborted are returned with the appropriate result to the user.

The following pseudo code illustrates the operation, where *thread* is the thread that has been l4_thread_ex_regs()'ed:

```
thread_abort(thread)
    disable_interrupts
    if thread is polling
        remove entry from receiver's receive queue
        if thread in soon wakeup queue, remove

    for each t in thread->send_root          // polling threads
        if t not in dying state
            clean up stack
            abort to cancel_polling routine
            if t in soon wakeup queue, remove
            enqueue in busy queue
    if thread is locked                      // IPC is in progress
        t = communication partner
        if t not in dying state
            clean up stack
            abort to cancel_ipc routine
            if t in soon wakeup queue, remove
            enqueue in busy queue
```

Interrupts are disabled to ensure the procedure executes as a critical section.

4.2 Task manipulation

Creation and deletion of tasks is done by the l4_task_new() syscall. This syscall did not need any significant changes, except those required to support the clans and chiefs model. One major addition is the implementation of task delete discussed below. The interface of this syscall has been changed to include an initial stack pointer so that this doesn't need to be initialised by the user in a startup stub. Previously code linked as the startup code for a new task set the stack pointer to a statically defined location in the text segment of the given code.

4.2.1 Task delete

Task deletion is performed whenever l4_task_new() is called on a task with a new valid pager in which case a new task is then generated. Alternatively, when an invalid pager is specified in which case the task is only deleted if it previously exists.

Implementation of task delete involves several stages to clean up all data structures including mappings, page tables and subtasks. The method used first recursively deletes any child tasks. This operation uses very little stack (2 64-bit words per clan boundary crossed), which is limited by the allowed depth of the clans and chiefs model. The next stage involves marking threads as dying so that the thread abort routine (discussed above in Section 4.1.2) does not reschedule these threads. After all threads have been made inactive, we can clean up the task data structures. Any mappings made in its address space are flushed. Page tables are then unlinked from the internal task table. Task creation rights are reserved for the deleter and we can then continue the time consuming task of cleaning up the page tables. This involves following all page table pointers, freeing and returning any used memory to the free memory pool.

This is shown below:

```
delete_subtree(task):
    child = task->first_child
    while(child)                    // recurse deleting subtasks
        delete_subtree(child)
        child = child->sister
```

```
if deleting initial task
    unlink from parent
for each thread in task         // mark as dead so we don't
    mask thread as dead         // schedule again
    dequeue from busy queue     // remove from scheduling queue
for each thread in task
    thread_abort(thread)
for each mapping in task->mapping_database
    flush(mapping)
flush TLBs
free TCBs
unlink root page table pointer
set task creation rights to the deleter
enable_interrupts
mem_free_tree(task->page_table_root)
```

The L4/MIPS [16] implementation was found to disable interrupts for a significant amount of time thereby affecting system performance and causing problems for any real time systems. The new implementation attempts to run with interrupts enabled as much as possible. However the overall L4 design does not easily permit interruptible code as much as it should. This implementation allows interrupts to be enabled between each recursive task delete and when cleaning up page tables after the page table root has been removed. Freeing memory taken up by page tables should in particular be interruptible. The stages at which interrupts can be enabled is still a bit fuzzy. We should be able to enable them after marking each thread of a task as dead, however this does not prevent other threads, such as those doing IPC operations, from scheduling a thread of a task being deleted. Being able to support this would involve extra checks in any scheduling queue insertion code, thereby adding a small amount of overhead. Enabling interrupts before removing mappings may be safe due to the preemptable implementation of the mapping database routines and should probably be fixed in the future.

4.3 l4_id_nearest() syscall

l4_id_nearest() is the basis of the clans and chiefs security model. This syscall returns the thread ID of the actual receiver when sending a message to the specified destination. Previously it was not implemented.

There are three possible return values for this syscall:

OUTER - destination thread is outside the callee's clan boundary. Returns the callee's chief ID.

INNER - destination thread is inside the callee's clan, requiring the cross of at least one clan boundary to reach it. Returns the ID of the task in the caller's clan which is the direct or indirect chief of the destination.

SAME - destination resides in the same clan.

The implementation places common cases first. It runs in kernel mode with interrupts enabled, as such it is preemptable.

```
if myself->chief_id == dest->chief_id     // check if same chief
    return (SAME, dest)
if dest->task_id == myself->chief_id      // check if receiver chief
    return (OUTER, dest)
while( myself->depth < dest->depth)
    if dest->depth > myself->depth
        if  myself->taskid == dest->chief_id
            return (INNER, dest)
        dest = dest->chief
if myself->chief_id == dest->chief_id    // check same chief of new dest
    return (INNER, dest)
return (OUTER, myself->chief_id)
```

4.4 IPC

The IPC path is where the most implementation time has been spent. The work includes:

- IPC timeout bug fixes. Registers were being trashed in the original version of the timeout calculation code. As a result timeouts where not set correctly.

- Addition of message dopes, both internally and in the syscall library interface. Message dopes return information about the success or failure of the IPC.
- Addition of chief syscall support. These calls are designed to give the receiver a full set of information about the received IPC, in particular the original destination of a redirected message. The IPC path needs addition of redirection and deceiving code.

For a complete integration of the clans and chiefs security mechanism into L4 we need to add support for redirection, deceiving IPC and support these though the library interface. This required a minor re-write of parts of the IPC path, discussed below.

4.4.1 Redirection

Redirection occurs when a clan boundary is crossed. That is, the destination thread is not directly accessible via IPC. To make IPC secure, we need to check for redirection before sending to the appropriate destination thread.

A single check in the IPC path determines if we need to do redirection. If so, we break out of the fast path to perform a series of checks. The first one determines the redirected destination thread that this message should be sent to.

The implementation makes a call to the l4_id_nearest() routine directly to determine the correct thread for redirection. This runs in kernel mode, hence we break out of PALmode and are preemptable for the duration of this call. Upon returning from l4_id_nearest(), we continue as normal from the point we left off in the IPC.

The original destination thread is kept in a register and returned to the caller, so as to aid forwarding the message on via deceiving IPC discussed below.

4.4.2 Deceiving IPC

In order to be able to forward a message on a thread's behalf we need to perform *deceiving IPC*. When performing a deceiving IPC, a *virtual sender* is specified as the thread the message should appear to have originated from. The virtual sender and destination threads are then checked to see if direction is preserved. Direction preservation ensures that a deceit operation is not performed by someone that does not have rights to deceive for a particular virtual sender and destination. If direction preservation is not honoured, the IPC is delivered normally, ignoring the virtual sender specified.

The receiver of a deceived IPC is alerted to the deceit by a bit set in the message dope and can determine the real sender by use of the l4_id_nearest() syscall.

The check for direction preservation results in two separate calls to l4_id_nearest() as follows.

```
ipc_deceit_long()
    check virtual-sender valid          // via call to l4_id_nearest
    check destination valid             // via call to l4_id_nearest
    if direction is valid
        set deceit bit in msg dope
        apparent sender = virtual sender
    else
        apparent sender = real sender
```

This section of code was problematic to implement. Care needs to be taken to ensure that when deceiving operations are performed, correct thread IDs are checked. For example, we have a chief sending to a destination thread in side its clan goes first. This chief is send deceiving a message from someone outside this clan. The chief will see that the receiving thread is not yet ready and will insert itself into the receivers send queue. When the receiver attempts to receive directly from the virtual sender, it checks to see if this task is polling (trying to send to) it. The wait operation will block endlessly waiting to receive an IPC from the original sender. This is corrected by checking each thread and its virtual sender if there is one, attempting to send to the receiver.

4.4.3 IPC (library) calls

In order to support the addition of clans and chiefs into the IPC path some library calls where added as in other L4 implementations.

Deceiving IPC calls give the ability to provide a virtual sender for the send component of an IPC. *Chief* IPC calls give the ability to provide a virtual sender for the send component of an IPC and return the real destination of a redirected message for the receive component.

The interface for all IPC calls is the same as L4/MIPS.

4.4.4 Summary

L4/Alpha now provides a more complete IPC implementation with support for the clans and chiefs security model and several bug fixes to the overall IPC path.

4.5 Alpha compatibility support

To support a large range of differing Alpha processors and in particular Alpha motherboards we need to implement support for these directly. Processor features, interrupt sources, memory configurations and different BIOSes are a small list of system specific areas that support is required for.

4.5.1 BIOS independence

The BIOS typically provides PALcode for the desired system. For example, SRM BIOS provides PALcode for Digital Unix; ARC/AlphaBIOS provides PALcode to boot Windows NT. L4 provides its own PALcode independent of that supplied in the BIOS.

It is normally the responsibility of the BIOS to provide system information including RAM configuration in a page in memory.

Interfacing the BIOS is problematic enough to warrant not relying on it in L4. As a result, it is required that L4 find out system configuration information on its own. This is particularly important for memory sizing. This is done at system reset time by asking the hardware directly for configuration information.

This had been problematic in some cases as it is hard to find information on some motherboards, while the chipsets are relatively easy to obtain documentation on. As a result it appears that motherboards utilising the PYXIS chipset do not receive correctly intervaled interrupts, even though this information appears to be set correctly.

Independent BIOS booting

By default a BIOS can load an executable image into any location in memory. Different BIOSes load images into different default addresses. Loading from flashable ROM as opposed to loading from disk can also result in the image being placed at different locations. Sometimes this is not suitable for a program which has been linked at a particular address, in particular L4 which is linked at 0x20.0000.

To get around the problem of not knowing where an image will be initially located, extra support to build a self relocating L4/Alpha image has been added. The relocating code executes in PALmode with support for a compressed L4 image. It copies L4 to the correct link location and then executes a system reset with a new PALcode entry point in L4. If the image destination is overlapping the re-locator, it is moved first before extracting the L4 image.

Having a relocatable L4 allows us to get around limitations of several BIOSes by providing a BIOS independent loader.

The relocation code was taken from MILO [1] the Alpha Linux loader and modified to support L4. This included support for booting L4 and the correct location. The actual MILO code was removed and replaced with L4. What is left is the relocation code as MILO too requires to be linked in at a specific address. In the future it would be nice to provide L4 boot support at the MILO prompt. Work on this began, including adding network boot support to MILO, however is not complete.

4.5.2 CPU and Chipset Support

There exist different chipsets and different motherboard designs that we need to support for a working system. For example, timer interrupts can be sourced from different locations.

Support has been added directly for PYXIS [14] chipset based motherboards including the UX [29] and Miata [10] machines. This support provides handlers in assembler for the timer interrupt. The methods used for each motherboard are different, as a result unique handlers exist for both. Memory sizing routines are also required for each motherboard.

The Alpha 21164 CPU core exists in several different flavours some that implement extended instruction sets. In order for these to be used they have to be enabled. In L4 use of byte operations such as load and store byte have been enabled.

Some CPU internal registers need to be defined on startup so that the CPU is in a state to function correctly. Unfortunately different BIOSes initialise the values in different ways, in the case that the Alpha architecture manual states them as being undefined.

Close examination of L4/Alpha startup initialisation code reveals cases where some registers are not initialised correctly. Once these were fixed, L4 was able to boot correctly. Future implementations need to ensure all internal processor registers are preset to sane states.

4.6 Image builder

In order to produce an image suitable for booting, Digital provides a tool called *sysgen* which is able to take several images including ecoff and a.out executables and place them into a single image. Its basic functionality is to size and pad to specified boundaries.

This tool lacked some features handy to build an L4 image with such as storing information on the files contained within the image, in the image.

The sources to this tool where taken and modified to provide extra functionality so that information on the contents of the generated image can be recorded and used in a similar manner to the Down-loadable Image Tool (DIT) provided with L4/MIPS.

The L4 image as shown in Figure 4.1 is a result of utilising this tool to concatenate a series of files together to form the bootable L4 image.

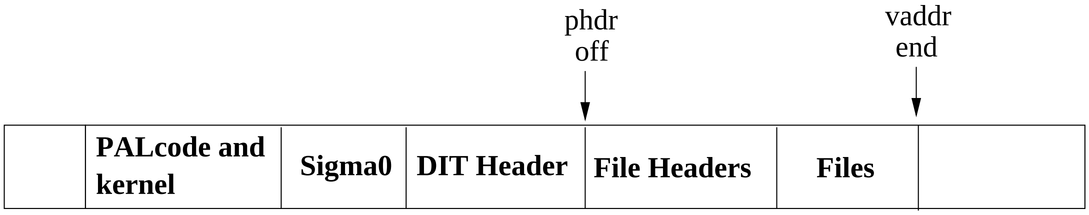

Figure 4.1: L4 bootable image layout

4.6.1 Features

The *sysgen* tool supports the following features, some of which have been added:

- a.out files,
- OSF/1 coff object files,
- stripped/raw files,
- specification of entry points of an image,
- include and generate DIT block,
- set image to runnable in DIT block,
- set image as resource manager in DIT block.

For example, to build an L4 image comprising of the L4 PALcode, a sigma0 image, a DIT block and an application:

```
sysgen -v -a -e0 pal/l4pal -e20000 -c sigma0/sigma0 \
       -d -e22000 DIT -e  24000 -r -b -c user/user -o l4image
```

Currently, the DIT block is expected to be at a specific location. As such the first three image's entry points and positions are not usually changed. PALcode, Sigma0 and the DIT sections are required.

The DIT block is allocated on an 8KB page, which includes space for the l4_kernel_info structure (used by L4 to maintain system state information) and DIT header, with the remaining space allocated for file headers.

~ (30)	b	r

Figure 4.2: DIT file header flags

As for L4/MIPS, each file header contains relevant information on a file in the image. The *flags* (see Figure 4.2) bit field has been modified to include a bit *'b'* specifying if this particular file is to be designated as the resource manager. In this case, this initial server is given rights to resources such as creation of tasks. The *'r'* flag specifies a file as an image server.

4.6.2 Support in Sigma0

It is desirable to have a standard way of loading initial servers such as device drivers and the other executables at startup. Support to do this has been added to Sigma0.

The information stored in the DIT header is used to determine which files to start as initial servers, such as serial drivers. Once this is done, Sigma0 allocates resource rights (namely task creation rights) to the file designated as the resource manager and then starts this server.

4.7 Execution on L4/Alpha

This section discusses some user-level support routines that have been added to the L4/Alpha source tree.

4.7.1 Pseudo serial driver

Previously L4 offered the following print routines:

- print_string()
- print_hex()

These functions implemented as *callpal* routines were used by user-level programs to output to the serial device. Using these routines makes it hard to print complex strings without risk of having them interleaved with serial output of other threads.

A serial output thread has been added to the kernel source tree. It accepts short messages (64 bytes) and outputs them to the serial device. This thread makes direct calls to *print_string()*. Currently it is not implemented as a real driver.

This in addition to libc support discussed below, makes it easier to produce valid output.

4.7.2 Libc support

Support has been added to the kernel source tree to assist user-level programs in displaying valid output. This includes implementation of several standard output functions such as *printf() and strcpy()*, a subset of libc. Currently, no floating point support is included with libc.

Alpha does not implement integer divide and remainder routines that are used frequently in the libc code. These instructions are emulated by routines that the compiler inserts (specified by the user) in order to be able to use these routines.

Users are recommended to build their own libc at this stage, with the support they desire. For example, Mungi includes its own libc.

4.8 Mungi

Mungi [19] version 1.2 has been ported to L4/Alpha in a mostly working condition.

In order to get Mungi working on L4/Alpha relatively few code changes were required. L4 macros that were assumed to exist had to be defined in some cases. Further more, a fairly consistent L4 API between L4/MIPS and L4/Alpha assisted in supporting common source trees.

Separate trees were created in the appropriate locations for Alpha specific code. A few Alpha specific assembly stubs had to be written including stubs to ensure that the stack was set up correctly across system calls going in and out of the Mungi kernel.

Certain components for Mungi's libc (string functions) equivalent had to be ported over. This included integer divide and remainder routines and implementations of memcpy and other related routines. The majority of this code is independent of MIPS or Alpha.

To make porting easier, many minor differences between L4/MIPS and L4/Alpha kernels were minimised by addition of the given functionality into L4/Alpha. This includes similar kernel information page handling and other features added to L4 as discussed in previous sections such as DIT.

Porting Mungi over to Alpha produced evidence of some bugs present in the Mungi code which interestingly did not appear when executing on MIPS. This includes use of macros to determine if Mungi objects overlap. Also the kernel memory allocation code did not correctly initialise a pointer to NULL. L4/Alpha does zero out memory at boot time, ruling out possibility of uncleared memory producing invalid pointers. These two bugs have been fixed and added to the generic Mungi source tree.

At this stage Mungi correctly initialises Mungi internals to a complete state ready to run user applications. The process of running user level applications includes firstly starting up a new task which begins execution of an upcall thread that is responsible for setting up the actual application to run. This upcall thread successfully starts the application thread which begins executing. This application which tests Mungi, fails when performing an L4 page flush. Due to time constraints this problem has not been looked into. It is likely that this call fails due to a difference in the L4 page flush (unmap) syscall.

Mungi tested several key features in L4/Alpha including deceiving IPC, flex-pages and threads. Also Sigma0 features were tested including DIT and kernel information pages and allocating of memory via flex-pages (mappings).

4.9 Testing

This section provides a small summary of the testing performed. This code exists in the L4/Alpha kernel source tree as a collection of several small test programs.

In order to determine what was implemented in the original uniprocessor implementation, in particular what appeared to work correctly, a lot of testing was done.

Thread preemption was tested. This test simply starts several threads and attempts to display serial output from each of them. Initially this did not work due to lack of PYXIS chipset support, required to initialise the timer.

All IPC calls were tested in varying scenarios. This included the chief calls, deceiving calls and calls involving redirection (clans and chiefs tests) done using short IPC. Testing revealed bugs in l4_id_nearest() syscall for correct clans and chiefs operation, incorrect timeouts when wait calls were queued and incorrect library stubs. These were all fixed and behave correctly. Correct message dope generation has been tested for all but receiving flex pages.

L4/MIPS provides a comprehensive test suite for testing L4 syscalls. Test code was taken from this suite including tests for all IPC calls except chief calls, l4_thread_ex_regs() and l4_task_new(). This suite assisted in debugging these syscalls. Unfortunately porting the test code to L4/Alpha proved to be too time consuming (even though it was cleaning up bugs). The test suite relies on too many hard-coded assumptions such as link addresses of executables.

In the future a generic test suite that is capable of correct bounds checking of syscalls will be implemented.

Testing of l4_thread_ex_regs() in the L4/MIPS test suite involved testing the new thread abort code. This test involves aborting a thread attempting to send to the thread we call l4_thread_ex_regs() on. The send operation aborts correctly and returns the correct result.

The l4_id_nearest() syscall was tested extensively with several levels of clans. This also implies that testing of IPC redirection calls was required in to correctly test these syscalls.

Chapter 5

Conclusion

Implementation and bug fixes to the uniprocessor L4/Alpha μ-kernel have been done to provide a more complete and functional μ-kernel.

A multiprocessor implementation of L4/Alpha has been broken down into components and analysed for ways to provide a scalable, fast and efficient μ-kernel. It is hoped that this research is successful in producing a μ-kernel that is capable of performing extremely well on both small and massively parallel multiprocessors. I am reasonably confident this work has produced the basis for a good implementation.

5.1 Future Work

In further work, the current uniprocessor kernel will need to be further stabilised.

It is planned that multiprocessor L4/Alpha will begin shortly which should provide a fall back and possible replacement to the current uniprocessor L4, however at this stage I am not certain if the kernel will support optimal uniprocessor execution. This would require separate uniprocessor and multiprocessor macros and possibly separate sub trees of some code.

A back port to the 21064 series of Alpha processors may be done, based on the uniprocessor version of L4/Alpha.

A proper L4 test suite for verifying the correctness of the L4 API. Mungi has shown that a system (albeit small) can be implemented in a generic fashion and work on L4 of different architectures.

Completion of a port of Linux to L4/Alpha.

Appendix A

L4/Alpha Thread Control Block

Below is defined the Thread Control Block (TCB) as defined in L4/Alpha. Each offset includes a description. L defines a long word (32-bit) data type; Q defines a quad word (64-bit) data type; QQ defines a double quad word entry used for double linked lists.

```
#define TCB_MYSELF          0x0000  Q: myself ID
#define TCB_WAIT_FOR        0x0008  Q: Polling for ...
#define TCB_KSP             0x0010  Q: Kernel Stack
#define TCB_RESTART         0x0018  Q Restart address
#define TCB_COMM_ADDRESS    0x0020  Q: Partners mapping address
#define TCB_THREAD_STATE    0x0028  Q: Thread State
#define TCB_COMM_PARTNER    0x0030  Q: Communication Partner
#define TCB_LIST_STATE      0x0038  L: State of lists
#define TCB_COARSE_STATE    0x003C  L: Coarse state

#define TCB_TIMEOUT         0x0040  Q: When to wake up
#define TCB_WAKEUP_QUEUE    0x0048  Q: Wakeup Queue

#define TCB_BUSY_QUEUE      0x0050  QQ: Busy Queue
#define TCB_BUSY_NEXT       0x0050
#define TCB_BUSY_PREV       0x0058

#define TCB_SEND_ROOT       0x0060  QQ: Send Queue Root
```

```
#define TCB_SEND_QUEUE      0x0070  QQ: Send Queue

#define TCB_PAGER           0x0080  Q: Pager of this thread
#define TCB_PREEMPTER       0x0088  Q: Preempter of this thread

#define TCB_SCHED_PARAM     0x0090  L: Schedule parameter
#define TCB_SCHED_SCP       0x0090  L: Schedule low
#define TCB_SCHED_MCP       0x0094  L: Maximum Controlled Priority
#define TCB_SCHED_ACCOUNT   0x0098  Q: Accounting Information

#define TCB_PRESENT_QUEUE   0x00A0  QQ: Present Queue Next + Prev
#define TCB_IDT             0x00B0  Q: Interrupt Table
#define TCB_INTERRUPT_MASK  0x00B8

#define TCB_MDB_ACCOUNT     0x00C0  Q: Thread 0: used for accounting
#define TCB_CHILD_TASK      0x00C8  Q: Thread 0: first child task
#define TCB_SISTER_TASK     0x00D0  QQ: Thread 0: next + prev task in clan
#define TCB_THREAD_NEXT     0x00E0  Q: what the present queue should be

#define TCB_IDENT           0x0100  L: ident TCB

#define TCB_STACKTOP        (SIZE_TCB-4*8)
```

Appendix B

Sysgen-dit image builder

This tool is used to build images as discussed above in 4.6. This is the output produced with *sysgen –help*.

```
sysgen - generates system images

> sysgen [flags] [[fileflags] file] ... [[fileflags] file]

    Where [fileflags] is one or more of these:
            -a: a.out file (produced by GAS)
            -c: OSF/1 coff object file
            -s: stripped file (no header)
            -eNNN: override/supply the entrypoint/base of an image
            -o: file is the output file
            -d: calculate and include a DIT block here
            -r: set this image to runnable in the DIT block
            -b: set this image as the resource manager
                - rights to resources
                This option implies option r also
    The files default to type -a

            If output file is not supplied stdout is used
    Flags: -hv
            -h: print this help text
```

```
            -v: verbose
```

Example

```
sysgen -v -a dbmpal -a -e40000 dbmhwrpb -s eb66_rom.nh \
    -o eb66_rom.img
```

Example

```
sysgen  -v -a -e0 pal/l4pal -e20000 -c sigma0/sigma0 -d -e22000 DIT \
    -e24000 -r -b -c user/user -o l4image
Stripped files must have an entrypoint/base specified
```

Bibliography

[1] MILO - Alpha Linux loader. Available from ftp://genie.ucd.ie/pub/alpha/milo.

[2] Solaris SunOS 5.0 Multithread Architecture, A White Paper. Technical report, Sun Microsystems Inc., 1991. Available from http://www.sun.com/workshop/threads/usenix.html.

[3] J. H. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. Department of Computer Science, University of North Carolina at Chapel Hill.

[4] A. Au and G. Heiser. *L4 User Manual.* School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Jan. 1998. UNSW-CSE-TR-9801. Latest version available from http://www.cse.unsw.edu.au/∼disy/L4/.

[5] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. Institute of Computer Science, The Hebrew University of Jerusalem, 1998.

[6] D. L. Black. *Scheduling and Resource Management Techniques for Multiprocessors.* School of Computer Science, Carnegie Mellon University, Pittsburgh, 1999.

[7] D. R. Cheriton and K. Duda. A caching model of operating system functionality. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 14–17, Monterey, CA, USA, Nov. 1994. USENIX/ACM/IEEE.

[8] Compaq Computer Corp. *Alpha Architecture Handbook*, 1998.

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

[10] Compaq Computer Corp. *Miata System Reference*, 1999. Available from http://www.compaq.com/products/workstations/digital/aseries/userguide.html.

[11] H. Custer. *Inside Windows NT.* Microsoft Press, 1993.

[12] Digital Equipment Corp. *Digital Semiconductor 21164 Alpha Microprocessor Evaluation Bboard - User's Guide*, 1996.

[13] Digital Equipment Corp. *Alpha SRM Console for Alpha Microprocessor Motherboards - User's Guide*, 1997.

[14] Digital Equipment Corp. *DIGITAL Semiconductor 21174 Core Logic Chip - Technical Reference Manual*, 1997.

[15] K. Elphinstone. *Virtual Memory in a 64-bit Microkernel.* PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Mar. 1999. Available from http://www.cse.unsw.edu.au/~disy/papers/.

[16] K. Elphinstone, G. Heiser, and J. Liedtke. L4 reference manual — MIPS R4x00 — Version 1.0. Technical Report UNSW-CSE-TR-9709, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Dec. 1997. Latest version available from http://www.cse.unsw.edu.au/~disy/L4/.

[17] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond multiprocessing... multithreading the SunOS kernel. In *Proceedings of the 1992 Summer USENIX Technical Conference*. Sun Microsystems Inc., 1992.

[18] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proceedings of the 1996 USENIX Technical Conference*, pages 91–105. Department of Computer Science, University of Utah, 1996.

[19] G. Heiser, K. Elphinstone, S. Russell, and J. Vochteloo. Mungi: A distributed single-address-space operating system. In *Proceedings of the 17th Australasian Computer Science Conference*, pages 271–80, Christchurch, New Zealand, Jan. 1994.

[20] M. Hohmuth. *The Fiasco Kernel: System Architecture.* Dresden University of Technology, 1999. In progress paper.

[21] J. Laudon and D. Lenoski. The SGI origin: A ccNUMA highly scalable server. Technical report, Silicon Graphics, Inc., 1998.

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

[23] J. Liedtke. Some theorems about guarded page tables. Arbeitspapiere der GMD No. 792, German National Research Center for Computer Science (GMD), Sankt Augustin, Germany, 1993.

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

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

[26] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, , and D. M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. Technical report, Digital Equipment Corp., 1996.

[27] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: a system software kernel. *Spring COMPCON*, pages 176–8, 1989.

[28] D. A. Rusling. *The Linux Kernel.* Linux Document Project, 1999. Available from http://metalab.unc.edu/pub/Linux/docs/linux-doc-project/linux-kernel.

[29] Samsung Electronics. *AlphaPC 164UX Motherboard - Technical Reference Manual*, 1998.

[30] S. Schönberg. Guarded page tables on Alpha 21064. Technical report, Cambridge University, 1996.

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

[32] S. Schönberg and V. Uhlig. Micro-kernel memory management. Technical report, Dresden University of Technology, 1998.

[33] Sun Microsystems Inc. *Multithreaded Programming Guide*, 1994. Available from http://www.sun.com/workshop/threads/usersGuides.html.

[34] C. Szmajda. Optimisation of guarded page tables. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Nov. 1999. To be submitted.

[35] L. Torvalds and et al. Linux kernel 2.2.13. 1999. Available from ftp://ftp.kernel.org/pub/linux/kernel/v2.2/linux-2.2.13.tar.gz.

[36] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the 13th ACM Symposium on OS Principles*, pages 26–40, Pacific Grove, CA, Oct. 1991. ACM.

[37] A. Wiggins. Design and implementation of the L4 microkernel on the strongARM SA-1100. To be submitted, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Nov. 1999.

[38] F. Zabatta and K. Ying. A thread performance comparison: Windows NT and Solaris on a symmetric multiprocessor. In *Proceedings of the 1998 USENIX Technical Conference.* Brooklyn College and CUNY Graduate School, Aug. 1998.