The L4 Microkernel on Alpha

Design and Implementation

Sebastian Schönberg

September 27, 1996

Preface

The purpose of a microkernel is to cover the lowest level of the hardware and to provide a more general platform to operating systems and applications than the hardware itself. This has made microkernel development increasingly interesting. Different types of microkernels have been developed, ranging from kernels which merely deal with the hardware infterface (Windows NT HAL), kernels especially for embedded systems (RTEMS), to kernels for multimedia streams and real time support (Nemesis) and general purpose kernels (L4, Mach).

The common opinion that microkernels lead to deterioration in system performance has been disproved by recent research. L4 is an example of a fast and small, multi address space, message-based microkernel, developed originally for Intel systems only. Based on the L4 interface, which should be as similar as possible on different platforms, the L4 Alpha version has been developed.

This work describes design decisions, implementation and interfaces of the L4 version for 64-bit Alpha processors.

Acknowledgement

I would like to thank to the Computer Laboratory at the University of Cambridge, providing me a desk in a very pleasant environment, to the members of the System Research Group, in particular Simon Crosby, Steven Hand and Robin Fairbairns for reading and correcting this paper as well as Richard Black, Chai-Keong Toh, Herbert Bos, Ian Pratt and Paul Barham for helpful discussions.

My work was also very well supported by the Lab. system team, in particular Martyn Johnson.

I would also like to thank Prof. Hermann Härtig at the University of Technology, Dresden, for his support during my project work 'L4 on Alpha' and the opportunity to stay for six month in Cambridge.

Last but not least to my parents for their understanding, advice and all their love...

This work is submitted only for achieving the academical degree *Diplom-Informatiker* at the University of Technology, Dresden.

This thesis is result of my own work and not outcome of work done in collaboration with other people.

All trademarks used in this work are hereby acknowledged.

Contents

Chapter 1

Introduction

1.1 Motivation

Microkernel interfaces should provide a good abstraction of the hardware to various levels above them. The L4 kernel [11] interface covers a wide range of this required functionality and flexibility. With only a few system calls, the kernel provides a maximum of stability and integrity for applications.

The algorithms used, the structures and the position very close to the hardware of the kernel makes it impossible to develop a generic kind of microkernel without loss of performance. However, the interface should be as portable as possible and can be used on very different platforms.

At the moment, the Digital Alpha 21164 series belong to the fastest microprocessors which are available. Combined with an acceptable price, machines based on this processor type are suitable for every kind of application, especially for multi-media and real-time applications.

Despite various differences between Intel and Alpha based machines, this work shall show how a well designed interface can be used without significant changes on different machines. Furthermore, it shall prove that low-level programming in assembly language is necessary in order to achieve high performance.

1.2 Synopsis

Chapter 2 gives an overview about related work on microkernel operating systems. Chapter 3 introduces the architecture of the Alpha processor and the design of L4. In addition, the Alpha based version of L4 is described. Thereafter, the next few chapters present the design and implementation of the L4 kernel on the Alpha. The components implemented are *Memory Management*, *Thread Management*, and *IPC*. Chapter 8 discusses the optimization of code and data structures for the Alpha processor. Chapter 9 contains some special implementation details and extracts of the code. Chapter 10 summarizes the important conclusions of this paper, presents some performance results and discusses the scope for further work.

Chapter 2

Related Work

2.1 Nemesis

Nemesis, described in [9] is a single address space operating system, developed at the University of Cambridge and is available for MIPS, ARM and Alpha 21064 based systems. Ports are underway for Alpha 21164 and Intel x86 based systems.

Nemesis is structured to fulfill the requirements of a fine-grained resource control system. Domains, which are scheduled by a very small kernel, are similar to processes in conventional operating systems. During execution of Nemesis Trusted Supervisor Code (NTSC) a domain is not preemptable. Hence, the kernel has only one stack and need not to be reentrant.

Activating a domain can either lead to a continuation after the interrupted instruction or the invocation of an user level activation handler. This can be used for intra domain thread scheduling and allows implementations of various scheduling strategies. Inter Domain Communication (IDC) is object-based done via channels. Every domain contains a Domain Control Block (DCB), consisting of a user read-only and a read-write part. Important information for scheduling are stored in here.

Nemesis, like Spring and other RPC-based systems uses strongly typed interfaces defined in an interface definition language, called MIDDL.

The Nemesis Alpha versions use a modified OSF PALcode.

2.2 Linux on the Alpha

Linux is a classical UNIX compatible operating system. All device drivers, file systems and operating system related parts are integrated into the kernel. This gives rise to robustness, provided the driver and file system code is error free. Errors in any part of the kernel can influence its stability and more often lead to a crash of the entire system. Despite the presence of loadable modules, changing the driver for any hardware device requires the system to be rebooted.

Linux on the Alpha processor uses the PALcode, supplied by Digital, and runs exclusively in the kernel mode.

2.3 The SPIN Kernel

The SPIN kernel [10] is being developed at the University of Washington. SPIN is written in Modula 3 and provides mainly virtual memory. Networking is incorporated in the kernel and enables fast commu-

nication with other machines. The kernel does not define an address space model directly, but can be used to implement one.

Furthermore, SPIN defines a structure on which thread implementations, in combination with the dynamic extension of the kernel, can be made. This allows the use of user specific scheduling strategies and a small overhead in the kernel for dispatching of threads. The kernel itself contains threads, based on a trusted thread package, exporting the Modula 3 thread interface.

SPIN supports dynamic incorporation of linked modules into the operating system kernel, which allows fast interaction between user applications and hardware events.

The protection model used is based on capabilities and supports fine-grained access control of resources. Events are used to indicate memory management faults to higher level memory managers, which can define services such as demand paging, garbage collection or shared memory.

2.4 The Spring Kernel

The Spring kernel [8] has been designed and implemented at the Univerity of Massachusetts to support and provide predictability, on-line dynamic guarantees, atomic guarantees, end-to-end scheduling and resource reservations. In effect, it supports a 'call/task admission' paradigm.

It utilizes a micro-kernel design for multiprocessor architectures and provides an interface to remote processes, support for distributed shared memory, and predictable low level communication. The kernel exists as a component of Spring's integrated environment that includes compilers, system description languages, etc. This environment extracts significant semantic information from programms which are used at runtime to provide flexibility and predictability.

Spring runs on an local area network, called SpringNet which consists of a set of multiprocessor nodes. Each Spring node contains system processors (SP) and application processors (AP), an I/O subsystem, and globally replicated memory. The SP insulates the application from the non-deterministic aspects of the environment and performs scheduling (in a planning mode) while the APs execute tasks guaranteed by the scheduler. The I/O subsystem handles non-critical I/O, slow I/O devices, and fast sensors. The fiber optic network supports predictable real-time distributed communication.

For scheduling Spring uses a VLSI-based scheduling co-processor, which can be used on- or off-line, and it constructs schedules based on deadlines, resources, precedence contraints, values, etc.

Unlike L4 and Nemesis, the Spring kernel requires special hardware. This avoids the influence of unpredictable hardware behaviour, such as memory caching strategies, during time critical sections but it is not suitable for general purpose computing platforms.

Chapter 3

Background

This chapter gives a short overview of the Alpha processor and the structure of the L4 μ-kernel.

3.1 Alpha Architecture

All Alpha processors are typical RISC machines with register to register data manipulation only. Operation codes are 4 bytes long with five different types of instruction formats. Three different data types, namely integer with 8, 16, 32, and 64 bits representations and IEEE and VAX floating point formats are supported. The basic addressable unit is the 8-bit byte, and a 43-bit virtual address space is supported. The processor has 32 integer and 32 floating point registers, each 64-bit wide. Register R31/F31 always contains the value 0 and cannot be changed.

At the moment, there are two different Alpha processor types on the market. The older and slower processors, 21064 and 21066, called EV4 with a frequency up to 275MHz have one unit for integer and one unit for floating point instructions. The second generation, 21164 called EV5 has two integer and two floating point units and runs a frequency up to 500MHz.

The following subsections list the most important features of the Alpha processor, which influence the design of the machine dependent aspects of the L4 kernel.

3.1.1 Load Store Architecture

Like every typical RISC processor, arithmetic operations can only be performed on registers. Hence, every modification of data leads to two memory accesses, a read and a write cycle. To accelerate the execution, frequently used data should be held in registers whenever possible. A two-level cache system and the use of write buffers increase the memory access performance of the processor.

3.1.2 Privileged Architecture Library (PAL)

Since complex instructions are not implemented in the RISC processor core itself, it must provide a solution to implement them in software. This is achieved on the Alpha processor by use of PALcode. PALcode has characteristics that make it appear to be a combination of microcode, ROM BIOS, and system service routines. This provides much flexibility in implementing application or system specific solutions.

In any operating system, some functionality must be implemented in PALcode, since only this environment allows unrestricted access to all parts of the underlying hardware or special functionality. PALcode has privilege to use five special instructions to

- access physical memory and
- to modify internal processor registers.

PALcode runs in a special environment, called PALmode, which is defined as follows:

1. Instruction stream (Istream) memory mapping is disabled,
2. Data stream (Dstream) mapping is enabled.
3. Privileged and unrestricted access to all hardware parts, and
4. Interrupts are disabled.

The first two points have the consequence that physical addresses are used for addressing instructions. To get access to data, located in virtual memory, address translation from virtual addresses to physical addresses is done in the same way as it is done in non-PALcode procedures.

In addition to the general purpose registers, the Alpha provides a second register file, which is available in PALmode and in kernel mode. It can be used to store information temporarily and consequently aids this in avoiding memory accesses.

The 21064 supports 32 independent PALtemp registers, whereas the 21164 only has 24 PALtemp registers. However, it has 8 further registers, called PALshadow which overlap some of the user integer registers. This permits the use of these registers without explicit saving and restoring into or from memory.

PALcode is invoked at specific entry points at well defined conditions. There are three different kinds of invocations.

1. After system reset
2. Hardware raised, asynchronous events
3. Synchronous software events, raised by the `callpal` instruction.

To achieve consistency between cache, memory and Istream, in all these cases all pipelines are flushed and the instruction pointer is set to a fixed address, which depends on the processor and on the cause of the event.

In the best case, the 21164 is able to switch to PALcode within 4 cycles, and generally within 9 cycles. On the 21064 up to 19 cycles have been measured in the worst case.

The position very close to the processor hardware of PALcode requires that attention is payed to rules of code scheduling. Most of these rules are related to a delay for 'n' cycles between instructions producing and consuming a result (Producer Consumer-latency). For instance, the result of reading a value from a PALtemp register cannot be used within two cycles after the read instruction itself.

Violated rules lead to unpredictable behaviour of the processor, which may include processor shutdown. Digital has developed a tool, called the PALcode Violation Checker (PVC) which tests compiled code for these PALcode restrictions.

3.1.3 Translation Buffers (TB)

Generally speaking, RISC processors contain only the most important and simplest instructions. 32-bit address space management can be handled quite simply using a two-level hierarchical page table. Using the same technique on a 64-bit address space would require at least five page table levels and a more complex hardware. Hence, the processor only caches recently used page translation references in an internal table, called the Translation Buffer (TB). Each table entry stores the virtual address. The translation itself must be done by the operating system.

The Alpha contains two independent TBs, one for references to instruction pages (ITB) and one for data (DTB) pages. In case of the processor does not find an entry for a virtual address in the Translation buffer, an exception, called 'TB miss' is raised.

3.1.4 Internal Processor Registers

The Alpha contains a set of Internal Processor Registers (IPR). These IPRs are used to control the different parts of the processor and can be accessed in PALmode or kernel mode. IPR used to manipulate the translation buffer are accessible in PALmode only.

3.1.5 Interrupts

The Alpha supports three sources of interrupts:

- Hardware: Six level-sensitive on the 21064, or seven level-sensitive on the 21164, triggered by external signals
- Software: Fifteen software interrupts, sourced by the Software Interrupt Request Register
- Asynchronous System Trap (AST), Four traps, sourced by the asynchronous system trap request register (ASTRR)

All software interrupts are independently masked by on-chip enable registers to support a software controlled mechanism for prioritization. If enabled, interrupts are performed by a branch to PALcode. The interrupt latency is relatively low. For hardware interrupts, the 21064 supports a bit-mask to enable or disable each interrupt, the 21164 supports a 32-level interrupt prioritization scheme.

3.2 Structure of L4

This section briefly describes the structure of L4 and its components. It then addresses issues of how to port the components of L4 to the Alpha Processor. There are four principal parts of L4 which are interrelated as shown in figure 3.1. They are:

- *Thread Management*: responsible for all active parts in an address space. Provides a basic, Round Robin like, scheduling; activates and preempts threads.
- *Memory Management*: provides the passive part, similarly to a container, for threads, handles mappings, and access rights to pages.
- *Communication Management*: Provides **Inter Process Communication** (IPC).
- *Machine Management*: manages caches, interrupts, exceptions and covers the lowest level of the processor.

3.2.1 Thread Management

Threads are the active components in L4. Each thread is identified by its identifier, called its Thread ID. This number is unique in *time* and *space*. This means that after starting the system, no two threads are in ever given the same Thread ID. A thread is linked exactly to one address space and cannot migrate to another.

At any time, all 256 (the maximum number) threads exist in each address space. These threads can either be active or inactive. There is no service to create or destroy a thread as is commonly found in

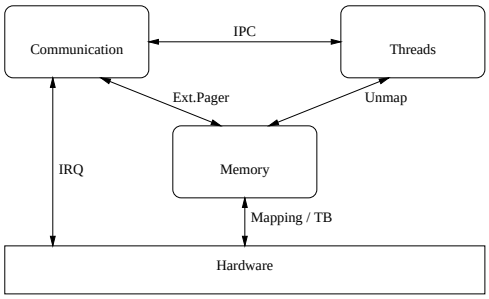

Figure 3.1: L4 Kernel Parts

traditional OS. Simply changing the thread instruction pointer to a valid address leads to scheduling of the apropriated thread at the next possible time.

Every thread has its own Thread Control Block (TCB), where information about the current thread state, the consumed time, and data necessary for IPC are stored. Since threads are also preemptable while running in kernel mode, a kernel stack per thread is required; the TCB is used therefore. This allows to use the stack pointer register to find the Thread Control Block that belongs to a thread. All TCBs are arranged as an array in virtual memory.

The scheduling strategy used is a simple Round Robin policy with static priorities. An interface for external schedulers provides the possibility to change this scheduling policy. External schedulers are user level threads, running in the same or a different address space as the thread, managed by it. The external scheduler of a thread always receives an automatic, kernel-initiated message when this thread loses the processor. Hereafter, the preempted thread is blocked until the external scheduler sends its reply. Figure 3.2 illustrates the control flow for an external scheduler.

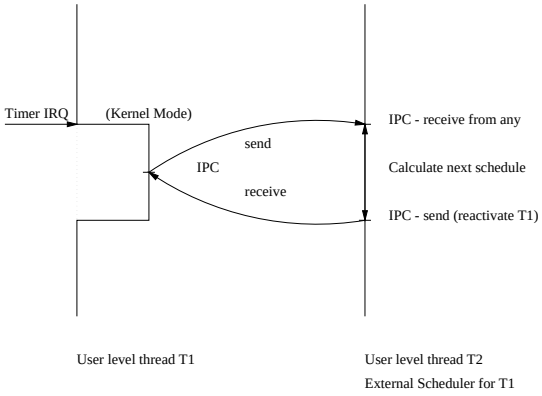

Figure 3.2: External Scheduler Control Flow

The concept of *Clans and Chiefs* is used to achieve a maximum of security. A clan is a set of tasks headed by a chief task; clans can be nested. All messages to recipients within the same clan are transferred freely. Messages crossing the border of a clan are always redirected to the chief of this clan. It can decide by use of the receiver ID, the sender ID, or the message itself, what should be done with this message. Figure 3.3 illustrates this redirection. The concept of 'Clans and Chiefs' is explained shortly in [11].

3.2.2 Memory Management

The Memory Manager controls all parts of memory. This is necessary to guarantee the integrity and autonomy of threads in different address spaces. Each one of the 1024 possible address spaces contains

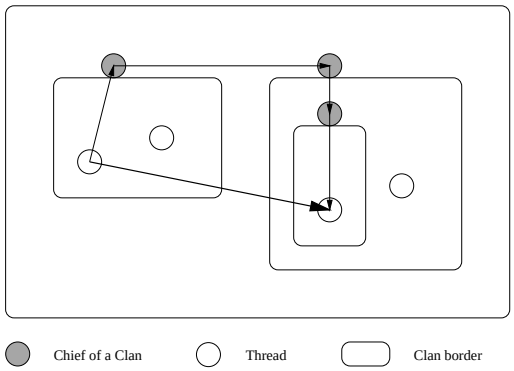

Figure 3.3: Clans and Chiefs

256 threads. Creating an address space includes the creation of all 256 threads. Except for thread 0 which is active, all other threads are initially inactive and do not occupy memory.

Besides this service, the Memory Manager *maps* and *unmaps* pages, transforms page faults into messages and sends them to *external pagers*.

The L4 memory management is strongly based on the concept of external pagers. External pagers are threads which are able to handle page faults of other threads. In the case of a thread causes a page fault, the external pager of this thread receives a message and must reply to this message with a memory object, called Flexpage. The first pager, or root pager of a system is called *Sigma 0*. During system startup, Sigma 0 receives all memory, not internally used by the kernel. Hence, later on, Sigma 0 is responsible for sharing the physically existing memory between address spaces.

3.2.3 Communication Management

The only way for a thread to communicate kernel-controlled with threads in different address spaces, including those residing on different machines, is by Inter Process Communication (IPC). To avoid the typical effect that IPC is the 'bottleneck' of the microkernel, this service has to be well optimized and adapted to the underlying hardware. Due to the different numbers of integer execution units of the 21064 and the 21164[1], it can be assumed that optimized IPC versions for the 21064 and the 21164 look completely different.

Fast IPC can be used for services like:

- Communication between threads (residing in different address spaces as well as for threads of the same address space)
- Announcing page faults to external pagers,
- Thread activation by external schedulers,
- Announcing Interrupts to threads.

L4 IPC has some important features;

- It is direct; neither a channel nor a port is used. The peer is specified by its thread ID.
- The IPC is strictly synchronous. Messages are only transferred if both sender and receiver are ready to perform the act.

[1]The 21064 can execute only one integer instruction, the 21164 can execute two in parallel

- It uses a description structure, which allows 4 types of data transfer in one message:
  - *Register words*: depending on the machine used up to 8 registers are used to pass their content to the communication partner. Using only register transfer is the fastest way to exchange data and should be used whenever possible.
  - *Direct strings*: data behind the descriptor called *dope* are copied word by word to the address given at by receiver thread.
  - *Indirect strings*: two words in the message body describe the address and the length of a string.
  - *Flex pages*: Allows transfer of entire pages from one address space to another. The page can be *mapped* [2] or *granted* [3].
- It uses one system call to implement five different behaviours:
  - Send a message to a certain thread (Send).
  - Receive a message from a certain thread (Receive).
  - Receive a message from any thread (Wait).
  - Send a message and wait for the dependent reply (Call)
  - Reply a message and wait for any thread (ReplyAndWait)

The general structure of the dope principle is illustrated in 3.4.

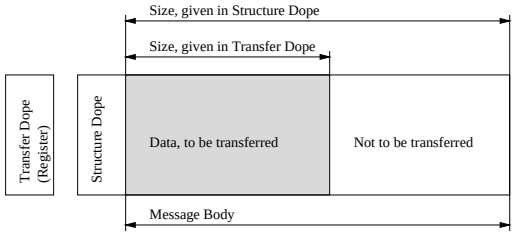

Figure 3.4: Principle of Message Dopes

3.2.4 Machine Management

This part of the L4 kernel covers the low level hardware. It manages

- hardware interrupts and exceptions,
- fills the TB on demand,
- modifies processor internal registers.

This part is different for each version of the processor and for each complete system.

[2] appears in both address spaces
[3] vanishes in the sender's address space and appears in receiver's address space

Chapter 4

Alpha Processor Modes and L4

In the L4-Alpha port, all microkernel service procedures have a stub in PALcode [7]. Depending on their complexity, either the complete function is executed in PALcode, or the stub is used only to invoke the specific procedure, running in kernel mode. The exact interface is described in Appendix A.

4.1 Privileged Architecture Library

On the 21064, PALmode provides an additional register file with 32 private registers, called PALtemp registers. This file can be used, only for storing information. L4 uses these registers for keeping information which must be readily available, such as the Page Table Base. Integer registers are stored in here temporarily during PALcode service procedures.

Since internal processor registers on the 21064 often have a different read and write format, PALcode uses some of the PALtemp registers to couple them with an IPR. Both registers are updated while a write operation, for reading the internal processor register, the PALtemp register is used. Hence, the read format of the IPR is the same as the write format.

On the 21164 the method of using PALtemp registers as shadow registers is not required. All important IPRs have the same read and write format. Hence, only 24 PALtemp registers are supported. The remaining 8 registers, called PALshadow registers, overlay the integer registers R8-R14 and R25 in PALmode. Thus, PALcode can consider these integer registers as local scratch and does not need to store them explicitely.

Some of the PALtemp registers are used to store L4 dependent information permanently. These are shown in figure 4.1

PALtemp register	**Name**	**Usage**
pt0	ptIntMask	Saves the current interrupt mask.
pt1	ptCurrentTime	Current Timer Tick value
pt3	ptPageBase	Contains the root of the current page table tree.
pt4	ptKSP	Kernel stack pointer of the current thread during user mode

Figure 4.1: PALtemp Registers and Usage

On the 21064, the registers pt12 to pt23 are used for temporary use during PALmode and Kernel mode operation. Registers pt24 to pt31 are used while filling the Translation Buffer.

A 21164 implementation will use the PALshadow registers for this purpose. This avoids saving and restoring the integer registers at the beginning and the end of the TB miss handler.

The **User-PALcode interface** provides the interface for all user threads to call an L4 service. It builds a stack frame, which is uniform to all L4 microkernel services, saves the user stack pointer, the mode where the call came from, and the return address on top of the kernel stack. Hereafter the stack pointer is set to this kernel stack. The layout is shown in figure 4.2

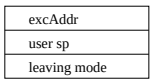

Figure 4.2: Kernel Stack Layout

4.2 Kernel Mode

Since PALcode is very hardware specific and not interruptible, implementing all L4 functionality in PALcode would result in a complex and unmodularized kernel, as well as in problems of predictability and fastly delivering interrupts. Hence, less time critical procedures or procedures which have to be interruptible like copying of long messages run in kernel mode.

The **PALcode-kernel interface** links the part, written in PALcode and this in kernel mode together. This is necessary to have a fast method to switch from PALmode to kernel mode. The used memory mapping of L4 enables the kernel to access the physical memory, where the PALcode is located in, as virtual memory. To jump to kernel mode, PALcode simply sets the processor mode to kernel, calculates the kernel entry address and branches to kernel mode via `hw_rei`. Returning from kernel mode to PALmode is done via the privileged operation `call_pal PAL_RET`. It continues at the address given in the `AT` register, that was chosen, since it is only used for temporary calculations and therefore conflicts with procedure variables are avoided. A round trip costs about 20 cycles.

4.3 Summary

Only the combination of PALcode and kernel mode provides the best and fastest possibility to implement a microkernel on the Alpha. Only a few and highly specialized procedures take fully advantage of the PALcode, the more general and interruptible parts of the L4 kernel use the privileged kernel mode.

Chapter 5

Memory Management

Memory management provides *address spaces*, which are the passive base, in which threads may run. The following functionality is provided by the memory management mechanism in L4.

- Cache synchronization management,
- Translation Buffer management,
- Page tables management,
- Handling of page faults,
- Mapping and unmapping of pages, and
- Creation and deletion of entire address spaces

5.1 Cache Synchronization

Since Alpha systems support an automatic synchronization between first and second Level caches, the programmer must only ensure, that slow IO access is synchronized with Dcache or Icache. To assist this, the Instruction Memory Barrier `IMB` and Memory Barrier `MB` instructions were introduced. While IMB synchronizes the Icache with the host memory and has to be implemented as user callable PALcode (entry 0x86), `MB` is a processor instruction and synchronizes the Dcache. `IMB` is only required for self-modifying code. `MB` should always be used before and after IO manipulation.

5.2 Translation Buffer Management

The Alpha 21064 ITB has twelve entries. Eight entries are used for small 8KB pages; four entries are used for large 4MB pages. The ITB entry to be replaced is chosen by the processor for each region independently, using a Not Last Used (NLU) algorithm.

The DTB has 32-entries, which are used for all sizes of referenced data pages. Similar to the ITB, the mechanism to fill the DTB uses an NLU replacement algorithm. In future processor implementations, the strategy may be changed to a Robin Round (RR) algorithm.

All entries are untagged, but each contains an Address Space Match (ASM) bit to exclude it from a TB flush operation. This avoids TB misses for pages which are shared between different address spaces after a context switch.

The 21164 has two independent TBs, each of this is tagged by a 7-bit address space number (ASN). This improves performance, since translations for an address space need not to be flushed during a

context switch. The ITB is a 48-entry, fully associative translation buffer. Unlike the 21064, the 21164 combines all references, independent of the page size, in one translation buffer. The DTB is a 64-entry, fully associative translation buffer. Both buffers also use an NLU replacement strategy and must be maintained in PALcode.

5.3 Page Tables

As described in [6], Guarded Page Tables (GPT) are very efficient for sparsely filled address spaces. This together with the simple and fast algorithm to translate a virtual address into a physical address were the basic reasons for the choice of Guarded Page Tables. Other alternatives such as static n-level page tables or inverted page tables are not suitable for 64-bit address spaces, since more resources, either translation time or storage space, are required. Using software TLBs can reduce the costs for address translation.

5.3.1 Address Translation

Address translation to fill the Translation Buffers (TB) must be implemented in software. L4 uses Guarded Tables (GPT), since they use memory and processor resources more efficiently than conventional translation algorithms, whilst preserving all advantages of hierarchical page tables.

In sparsely filled address spaces, many tables of an n-level page table tree are filled with only one entry and used to go on to the next level. Guarded Page Tables skip these intermediate levels using a guard. Figure 5.1 illustrates this. This means, some of the bits of the virtual address are skipped, if they match a specific pattern. This method minimizes the time to translate a virtual address and saves memory. Furthermore, Guarded Page Tables support pages and page tables of different size[1].

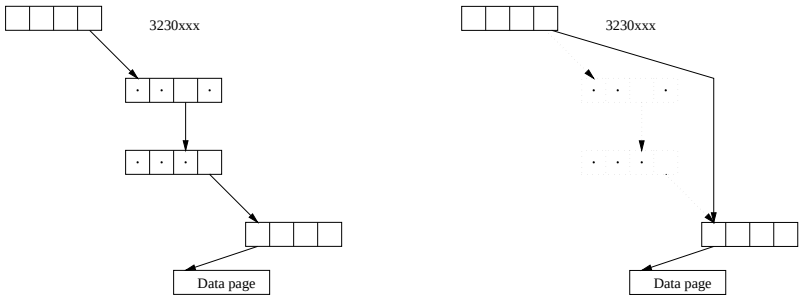

Figure 5.1: N-level page table tree vs. Guarded Page Table tree

5.3.2 Page Table Structure

Each Page Table Entry (PTE) is represented by two, 64-bit wide entries. The first contains the guard, the required protection and the number of guard bits which must match the address for a valid translation. The second entry points either to the next level in the page table tree and codes the size of this table, or if the entry is a leaf entry in the tree, it contains the word that is written beside the virtual address into the TB. Figure 5.2 shows the structure of these two 64-bit words, containing the Guard, Protection bits, and the pointer to the Next Frame Address. L1 represents the the remaining bits, used for the next level address translation, L0' contains in coded form the size of the next page table.

Figure 5.3 represents one virtual address at *level n* and *level n+1*. Furthermore, it shows which size information of the address are stored in the page table entry. u and u' are the bits of the address used for indexing the table. g and g' are the guard bits which must match these bits stored in the page table

[1] Page tables with a size of $size = 2^n$ for n greater than 4 are supported

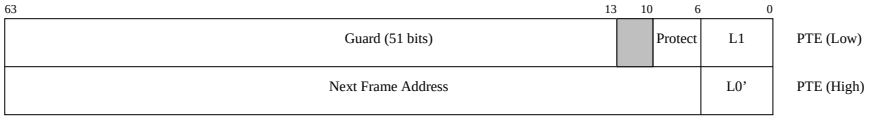

Figure 5.2: Guarded Page Table Entry

entry. A conventional n-level hierarchical page table can be simulated by using always the same size for u and a guard length g of 0.

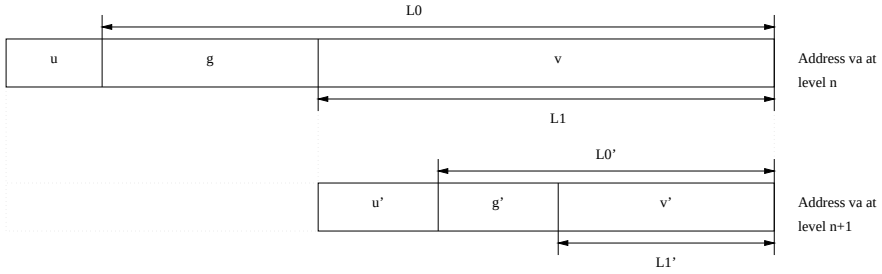

Figure 5.3: Virtual Address Translation

5.3.2.1 The Guard Field

The guard is a mask, which must match partly the virtual address at each stage in the page translation. Since the smallest page size supported by the Alpha is 8KB, the upper 51 bits are used for the guard.

Protection information is necessary to figure out whether a page access is valid or not. It must be included in each PTE to benefit from the advantages of hierarchical page tables. This allows the changing of the protection of a range of n continuous pages by changing only one bit.

To adapt to all variants of information supported by the Alpha hardware the following bits are required:

- *User-Kernel:* decides which mode is required to access this page.
- *Executable:* this page contains instructions which can be executed.
- *Writable:* decides whether the page can be written or not.

The protection field is enhanced by one bit, called 'InTree' which decides between a leaf and a node of the tree. Bits 6...9 are used to store this information.

The lowest 6 bits are used to indicate how many bits of this guard are valid. The number is coded in the form of *64-guard_length*. Strictly speaking, it represents the number of bits remaining for the next page table tree level.

5.3.2.2 The Pointer Field

The pointer field contains two pieces of information: the size of the page table at the next level and the starting address of this table.

To indicate the number of entries of the next page table tree level, six bits are required. The information is stored in the form of $64\text{-}log_2 size$.

It is possible to use the lowest six bits without restricting the functionality. It can be assumed that all page table entries are aligned to their size (16 bytes), hence the lower 4 bits are always zero. If it is

decided that page tables with less than 4 entries are useless and not required, then two further bits are zero for each base address of a page table.

The upper 58 bits of this entry are the pointer to the next page table.

If this entry is a leaf of the tree, all 64 bits of the field contain the entry, which is written in the TB.

5.3.2.3 The Page Table Base Register

Translation must take place in PALcode. One of the PALtemp registers is dedicated to point to the root of the page tree. It is structured in exactly the same way as the pointer to a PTE.

5.3.2.4 The Operation

The operation itself is very simple. Figure 5.4 shows the steps of the operation.

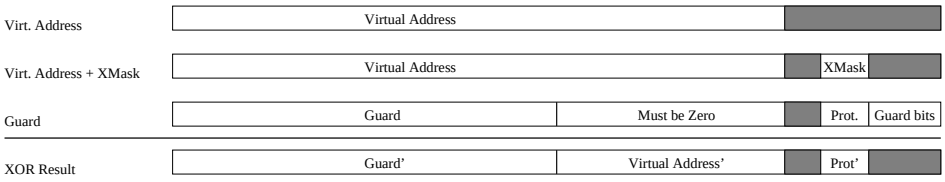

Figure 5.4: Guarded Page Table Operation

1. The protection bits of the virtual address are merged with the constant XMask. This sets bits 4..9 of the virtual address to a defined state.

2. The result is combined with the guard field by an XOR instruction. All bits which are identical in the guard and in the address will be cleared. If the *guard'* field is equal to zero, the address matches the guard.

3. By setting the XMask in the original virtual address, the resulting protection field *prot'* is not zero, if either the translation is finished (only the InTree bit is set) or the access is not permitted.

5.3.3 Address Translation Faults

There are three possible causes of address translation faults:

1. *Table Protection Violation*: The pair (virtual address, TLB entry) exists in the GPT, but is not valid for the current mode (e.g. write to a read-only page). In this case, the PALcode is left and execution is continued in kernel-mode at *tb_protection_fault[2]

2. *Guard Fault*: There is no usable entry for this virtual address. The execution is continued in kernel-mode at *tb_guard_fault.

3. *Access Violation*: belongs to the type 'Table Protection Violation'. This fault occurs not during parsing the GPT, but if the processor detects a mismatch between the access permitted by the TB entry and the required type of access. Execution is continued in kernel-mode at *tb_access_fault.

[2] *tb_... entries represent the entry, callable as itb_... while ITB misses or as dtb_... while DTB-misses

In all three cases, the PALcode creates a stack frame, restores the used registers from the PALtemp register file and saves the registers, containing the return address, the faulting address and the error reason on the stack.

Since the translation procedure does not use virtual addresses, no nested DTB misses can occur. After entering kernel-mode, the code is reentrant and further TB misses are no problem.

After all work in kernel mode is done, the kernel has to return to one of the following points in PALmode:

- *ret_*tb_fill*: the given argument pair is written to the translation buffer and the faulting instruction is executed once again. This entry is used, if a valid translation should only enabled until the next address space switch.
- *ret_*tb_schedule*: the scheduler is called. There are no changes in the TB. This function is used, if the state of the faulting thread has been changed.
- *ret_*tb_rerun*: the faulting instruction is executed once again without any changes regarding the TB. This entry point must be called, if PTEs have been changed. Depending on the arguments, the entire depending TB may be flushed.

5.3.4 Page Table Management

Page tables are accessed only via physical memory; there are no virtual memory mappings to page tables. PALcode and kernel can access physical memory by using load physical (ldl_p and ldq_p) or store physical (stl_p and stq_p) instructions. Inserting new entries into GPTs sometimes requires the tree to be split. This means that the guard of the existing entry is shortened to the common base of the old and the new entry, and a new level is inserted into the tree. This 'split' operation runs in kernel mode and takes up to 400 cycles. If a new page has to be allocated, the kernel talks to Sigma 0 via IPC and requests a page. This page has to be granted to the kernel, since for security reasons the kernel needs exclusive access to pages used for page tables.

Freeing of page table entries can join two levels of page tables. This happens if affected page tables contain only one remaining entry. This 'join' operation can either be done during the free operation or, in case of less free memory in a kind of a garbage collection. This second method should be preferred, since garbage collection can be done in parallel, when the system is idle or all threads of an address space are not runnable.

Freeing page table entries requires synchronization with the TB, which is achieved by simply flushing the translation entry for the affected virtual address.

To get the page table root of an address space quickly, all pointers to page tables are stored in an 8KB array, located in the kernel data. The task number that is equal to the address space number, is used as an index into this table. Since all page table references including the page table root are pointing to physical addresses, only the lower 32 bits are used. The higher 32 bits point to the first MemMapTree-node (see 5.5.4) which links all mappings of this address space.

5.4 Page Fault Handling

Section 5.3.3 explains, that PALcode branches to kernel mode in the case of faults during address translation. A third type of exception is raised if the TB already contains an entry that does not allow the requested access.

The kernel decides by use of the faulting address the action to be executed. In case of a common page-not-present fault it initiates a Flexpage IPC call with the external pager of the faulting thread. Otherwise it prepares mappings for cases which are:

- Access of Thread Control Block (TCB) space without a special TCB pager (Sigma 1)

- Access of the Temporary Mapping Area (TMA) (see section 7.3.1).

5.4.1 Page Fault in User Space

This kind of page fault occurs always, accessing a page which is not present in physical memory. If the causing thread has an *external pager*, it receives a page fault message and has to reply with a Flexpage message. If such a pager does not exist, the thread is cancelled by the kernel.

5.4.2 Page Faults in the TMA

Every long message transfer touches the TMA and causes at least one TB miss during execution. The page fault handler for the TMA looks up the frame of the virtual address in the receiver's address space, to where the data should be copied and puts this value into the TLB only. Since a temporary mapping is only valid during the time taken to transfer message, it would be much too expensive to enter PTEs into the page table and delete them after performing the transfer. Furthermore, if an average bus transfer rate of *60MB/s* is assumed, 750 pages could be copied within one schedule.

$$60\frac{MB}{s} * 100\frac{ms}{timeslice} = 6\frac{MB}{timeslice} \approx 750\frac{Pages}{timeslice}$$

Hence, the need to look up an entry twice will occur very rarely and only for very long message.

All these TMA translation buffer entries have a cleared ASM bit. This enables the PALcode after the transfer to clear the TB simply by flushing the translation buffer. This is less expensive than an entry-by-entry clearing of TMA mappings in the TB. Furthermore, usually a following task switch to the receiver would automatically flush the translation buffer.

5.4.3 Page Faults in TCB Space

Since TCB space does not belong to the user accessible memory and contains the kernel stack, where wrong values influence the stability and integrity of the kernel, Sigma 0 is not trustworthy enough to manage this area. In fact, Sigma 1 is responsible for any faults accessing the TCB space. During system startup where no Sigma 1 is available, the kernel simply maps an empty frame at this address.

5.5 Address Spaces

In general, the L4 semantic defines an address space as a region of virtual memory, containing as set of pages. Such a page can refer to no frame, or to exactly one frame, a frame can be mapped to several pages. If a page refers to no frame, this page is neither readable nor writable. Accessing such a page causes a page faults.

Entire address spaces can be created and deleted. At the page level there are services for mapping and flushing of single pages.

Pages of address spaces can be inherited recursively. This means, every thread can map every page of its own address space into other address spaces using IPC.

Alpha L4 supports up to 1024 address spaces; address space 0 is reserved for the kernel and not used for user threads. Every address space is completely protected against access from other address spaces. The region of the kernel is not accessible for user theads.

5.5.1 Address Space Switches

Since context switches lead to flushing of TB entries, the kernel does not switch the context, if the next thread is using the same page table tree for address translation. The code for a context switch loads the new page table root from the page table root array and stores this value in the *ptPageBase* register and flushes the DTB and ITB.

Due to the high cost of context switches, they should be avoided, whenever possible. For this reason, the system scheduler thread is shared between all address spaces and can execute in every context.

5.5.2 Address Space Structure

The lower region of each address space is accessible for user mode threads. Above these addresses, a region of 4GB virtual memory is reserved for kernel access only. This, so-called kernel memory, starting at address `0xffffffff.00000000` contains the kernel code and some structures, bound to fixed addresses. This start address was chosen, since

- it is easy to calculate by

```
subq    zero, 1, addr     ; create 0xffffffff.ffffffff
zap     addr, 0x0f, addr  ; delete lower 32 bits
```

- All current Alpha processors support only 43-bit address space. All bits from 43 to 63 must be equal to bit 42. The chosen address meets this requirement. If a future Alpha processor supports 64-bit address spaces, no changes will have to be made.
- It provides a huge user address space of up to 8188GB (8TB) virtual memory.

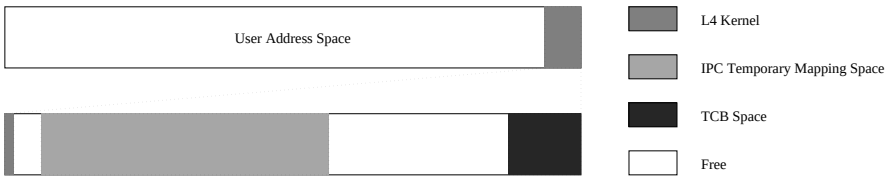

Figure 5.5: Address Space Structure

The parts of the kernel memory and their addresses[3] are shown in figure 5.6:

Address from	Address to	Usage
`0x00000000`	`0x001fffff`	Kernel Code and statically linked data
`0x10000000`	`0x8fffffff`	Temporary Mapping Area (TMA) for 256 threads
`0xe0000000`	`0xffffffff`	TCB Area for 1024*256 threads

Figure 5.6: Kernel Address Space Parts

5.5.3 Address Space Modification

An address space is created automatically when the first thread of a task is started. At this time, the address space includes the all shared kernel pages. All addresses below this address are not mapped to

[3] Only the lower 32 bits are given, the upper 4 bytes are `0xffffffff`

any frame. Accessing one of these pages causes a page fault and a messages is sent to the responsible external pager.

Deleting an address space is done by deleting the task. All mappings in this address space are released hereafter. All address spaces created by threads of this address space will be deleted as well.

Creation and deleting of address spaces are provided by the *task_new* system call.

After creation of an address space *Inter Process Communication (IPC)* is used to send pages from one address space to another. A page can be granted or mapped. The inverse operation, *unmap*[4] which allows to remove all mappings in other address spaces initiated by the flush-calling thread, requires bookkeeping about the initiated mappings of a thread.

The **map** operation is performed by sending a message to a thread of the affected address space, which has to accept a Flexpage message. Afterwards, the page can be accessed in both address space. Sometimes it is useful to pass mappings, through controlling systems. This can be done by granting a page, where the page disappears in senders address space.

A thread can **flush** or **unmap** any page of its address space at any time; no other thread can deny this action. The page concerned is removed from all address spaces which had received this page from the caller directly or indirectly. While the flush service includes mappings to the frame in caller's address space, unmap does not.

5.5.4 Bookkeeping of Mappings

Since flushing and unmapping requires information, about which thread has sent pages to other address spaces, and where these frames are mapped, the kernel must keep track of all of this information. The kernel stores this data in a tree, called the *MemMapTree (MMT)*, sorted by the physical address of the frame, which contains the page frame number. Each node contains beside the page frame number the linked virtual address, and the initiator of this mapping. Furthermore, links exist for mappings within the same address space and to the same frame. All MMT nodes which describe mappings in the same address space are linked together in a list. The higher 32 bits of the page root pointer in the page table root array points to the head of this list.

Each node contains the following data:

- *Page Frame Number*: This is the upper part of the TLB entry and is used to sort the mapping in the tree.
- *Same Physical Address*: link to a node, which describes a mapping to the same frame. This is used, if a frame has to be flushed.
- *Same Address Space*: Links to the next mapping node, which describes a mapping into the same address space.
- *Initiator*: TCB address of the thread, which has initiated this mapping.
- *Left and Right*: points to nodes with a frame number less or greater than the number of this node.
- *Virtual Address*: contains the virtual address and the task number of this mapping.

All pointers in this list refer to physical addresses and can cover 2GBytes of physical memory.

[4] unmap may include the calling address space

5.6 Summary

Microkernel memory management is responsible for handling address spaces, address translation and low level hardware requirements like filling the TLB.

The Alpha L4 uses Guarded Page Tables and fills the TLB on demand. This code runs completely in PALmode and takes *18+26+13*level* cycles.

Inserting and removal of an entry into the GPT is done in kernel mode and can cause splitting GPT entries into subtrees or joining them. This operation can cost up to 400 cycles plus costs for IPC to Sigma 0.

Entries, concerning the Temporary Mapping Areas are not inserted into the page tables since this operation is more expensive than eventually looking up an entry twice.

A binary tree is used to save information, which thread has initiated a mapping into which address space and at which address. Each node contains links to mappings of the same address space and references to same frames.

Creating an active task implies creating the address space.

Chapter 6

Thread Management and Scheduling

Threads are the active parts of an address space. While an address space provides sharing of the memory, thread management virtualizes the CPU resource. Every thread has a complete CPU register file including one instruction pointer and a set of state information which provides for the illusion of uninterrupted execution.

Based on the protection scheme of address spaces, all threads running in the same address space can influence each other, threads running in different address spaces cannot do so.

L4 supports a fixed number of threads per address space; the Alpha version provides 256. If more than 256 threads are required, an external pager can create a second address space which is entirely mapped to the first address space.

6.1 Thread Identification

Each thread can be identified by an unique number, its Thread Identifier (TID). This value is unique in *time* and *space*. In a running system, once a TID is used, it will never used again as long as the system runs. Several pieces of information are assembled and stored in the L4 TID. These are:

- *Thread Version*: The number of the thread, using this Control Block.
- *Thread Number*: Index the thread in this address space.
- *Task Number*: This is a synonym for the address space number. Threads with the same task number share the same address space.
- *Site*: With a view to supporting a multi node system, it makes sense to store the number of the node on which this thread is running. All threads with the same Site number run on the same node.
- *Chief Number*: Creating a task from within a task creates a hierarchy. This field contains the creator's task number. Thread 0 of this task is called the *Chief* of this thread.
- *Depth*: The level of chief nestings.

The complete Alpha L4 Thread ID is shown in Figure 6.1.

The *l4_id_nearest* system call returns either the TID of the caller or the TID of a thread which is chief of the TID argument to the call.

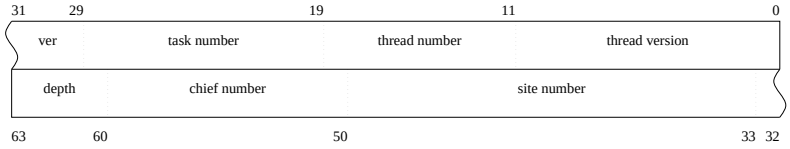

Figure 6.1: Alpha L4 Thread ID

6.2 Thread Control Block

As described above, during processor preemption all important informations must be saved. To this end, every thread has its own structure, called the *Thread Control Block* (TCB) where this data is stored.

Since a preemption is possible while the thread is running in kernel mode, a kernel stack per thread is necessary. The TCB is a good place for this stack. Furthermore, it allows a fast (2 cycles) calculation of the current thread's TCB start address while in kernel mode.

All TCBs are arranged in an array in virtual memory. The task and thread number of the TID are both used to index this table. All TCBs of existing threads are linked in a doubl linked list.

To decide on an optimal TCB size, the following consideration was made: Beside the complete integer registers file, which uses *32*8* bytes and is stored on the kernel stack during preemption, 32 floating point registers, each of 8 bytes must be stored. In 32 fields, information about the thread and linked lists are kept. This results in a TCB of 768 bytes. Return addresses of procedures and temporary registers used are also saved on the stack. For this reason, a stack of size 1024 bytes seams to be insufficent and consequently the TCB used of size 2048 bytes.

The structure of the entire Thread Control Block is explained in Appendix B.

6.3 Thread Operations

L4 provides only one, but very powerful system service, for thread manipulation, called **l4_thread_ex_regs**. This service can be used for

Starting threads:
By default, creating an address space creates all 256 possible threads, virtually. Except for thread 0 in an active task, these threads initially are passive but may be activated using l4_thread_ex_regs. Hence, instead of *starting a thread* it is more accurate to say that a thread is *activated*. This is done by setting a valid instruction pointer which points to a 4 byte aligned address in user mode.

Signalling threads:
Since IPC is strictly synchronous and requires always a peer, it cannot be used for signalling. Within an address space, the following technique can be used for signalling.

1. call l4_thread_ex_regs and set the thread to a procedure which does not modifies any registers. This keeps the current thread state and the previous instruction pointer and stack pointer are returned.
2. save these values on top of an exception stack.
3. call l4_thread_ex_regs to set this thread to the exception procedure.

Ending threads:
L4 threads cannot be destroyed as in common systems. Since ending a thread only means that the thread is withdrawn from the processor, this result can be achieved by invoking an IPC with itself and a timeout 'never'. Due to the strictly synchrounous IPC, receiving a message from itself is not possible, but the thread is blocked and does not get a timeslice; this results in exactly the

same behaviour as a not existing thread. To start the thread again, set a new instruction pointer with the l4_thread_ex_regs system call.

6.4 Scheduling

Scheduling is responsible for a fair sharing of the processor between threads. Since many different algorithms for various tasks exist, schedulers built into the kernel could only cover a small subset of possible algorithms. For this reason, L4 contains only a very simple, priority based Round Robin (RR) scheduler in the kernel.

An *External Scheduler Interface* (ESI) provides the opportunity to use many different scheduling policies. At the end of every time slice, the kernel initiates an IPC-based Remote Procedure Call (RPC) to the External Scheduler (ES). This stops the thread to be preempted, until the ES replies to this message.

Internally, all inactive threads are stopped in PALcode. After entering PALcode, the stack frame is built up, eventually the registers are pushed and the new schedule-in address is always saved on top of the stack. This allows PAL procedures to continue at the next time slice at another instruction than the next one. The switch itself is done by loading the kernel stack pointer from the target thread and continue at its appropriated address.

Floating point registers are not stored during preemption; instead the kernel only makes the FPU unavailable for the next thread. Later on, if the thread tries to use the FPU at the first access, an exception is raised and the kernel handler can save and restore the FPU context.

Scheduler Interface

The scheduler interface permits the reading and modification of thread data which are important for scheduling such as:

- Time Slice, Priority, the overall consumed time,
- the External Preemptor, and
- the current IPC peer, if the thread is performing an IPC.

Preemption

A thread is preempted, if it has completed its time slice. In this case *all* registers are saved in the Thread Control Block and the next 'ready' marked thread is scheduled. This operation costs about 100 cycles. About 70 of them are used to save and load the integer registers.

Yield

This occurs when a thread gives up its time slice voluntarily and the processor executes another thread of the same or of a different address space. If the calling thread dedicates its timeslice to no appropriated thread, the scheduler picks the next ready thread, which gets an entire time slice. Otherwise, the specified thread get the remaining time slice of the current thread in addition to an ordinary time slice.

Since *yield* must be initiated by the threads themselves, *no* registers, except stack and instruction pointer are saved. The calling thread can do this more efficiently by saving only a few important registers than the kernel, which would have to save all registers.

6.5 Summary

Threads are the active part in an address space. L4 supports up to 256 threads per task. The kernel uses an identifier, which is unique in time and space for an identification of threads. This ID consists, beside other information, of the logical thread number in this task and the address space number. Both values are used as an index into the Thread Control Block array.

The lower part of the TCB is used for thread state information, the higher part contains the kernel stack.

Creating and signalling of threads is done by the *l4_thread_ex_regs* system call.

The kernel provides an interface for external schedulers. Internally, a Round Robin algorithm is used to select the next thread to run.

Chapter 7

IPC

The L4 IPC provides structured message transfer. This means that the kernel understands different types of messages and is able to handle them. This avoids unnecessary copying or multiple IPC calls to transfer complex messages. Five different behaviours of one system call provide *sending*, *receiving*, *waiting*, *call* or *reply_and_wait* semantics for IPC. The operation is chosen by means of the send and receive vector.

7.1 Interface

The IPC interface is designed for optimal use by compilers. All mandatory arguments are given in the argument registers a0...a5 (r16...r21).

```
quad ipc (TID target : a0,
    DOPE snd_msg_dope : a1, void *snd_vect : a2,
    DOPE rec_msg_dope : a3, void *rec_vect : a4,
    quad time_outs : a5,
    [quad flexpage : fp],
    [quad w0, w1, w2, w3, w4, w5, w6, w7]);
```

Both timeout values are packed into one 64-bit value. The precision on Alpha L4 is much higher than that of the Intel version, in which both timeouts are packed into one 32-bit word. Each timeout value consists of an IPC timeout and a page fault time out. This is used to prevent blocking an IPC by a non trust-worthy external pager, if it does not reply to a page fault message.

7.2 Short Messages

Transfer of short messages occurs most frequently. Typically at least one message from or to the server is a short acknowledgement, which is typically not longer than 2 quadwords. Up to 8 quadwords in registers can be passed directly. The decision to restrict to 8 quadwords results from the requirement that all registers to be passed to the receiver have to be saved if the receiver is not waiting for a message.

All **Interrupts**, both software and hardware, are announced by Inter Process Communication. Calling IPC with an interrupt number as receiver allocates the interrupt. Hereafter, the kernel initiates an IPC to the thread every time this interrupt occurs. Each thread can only handle *one* interrupt.

7.3 Long Messages

Long messages are described by two words which are called the 'structure dope' and the 'message dope'. These messages include *direct strings*[1] or *indirect strings*[2] and are memory based. The structure dope describes the static appearance of the message, the message dope the data to be transferred. This allows to send only parts of an entire structure. Both dopes have the same format, which is shown in figure 7.1. The structure dope is expected at the beginning of the message in memory; further implementation can combine both dopes in the argument register, if they are still 32-bit.

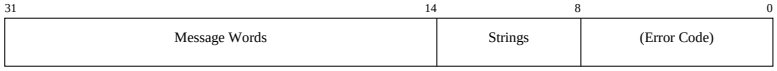

Figure 7.1: Message Descriptor - Dope

7.3.1 IPC and Temporary Mapping

Old-fashioned kernels transfer data from one address space to another by copying the data first into a kernel buffer and later on, from this buffer to the destination address space. L4 uses temporary mapping for the transfer; parts of the receiver's address space are mapped at an appropriate position into the temporary mapping area in the address space of the sender while transferring the message. Figure 7.2 illustrates this. Copying the data to this region implies a cross-address space copy of the data to the receiver. Every thread of an address space has its own, 8MB big slot.

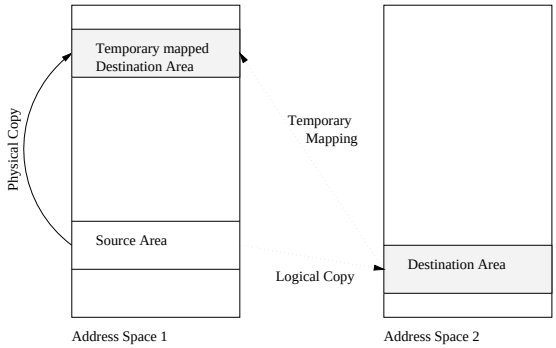

Figure 7.2: Message transfer and Temporary Mapping

Every thread of a task has its own send area. Since mappings there have to be changed very often, exactly each time a new transfer is performed, page tables do not contain entries for this region (see section 5.4.2). Instead, the kernel generates the data which have to be filled in the Translation Buffer on demand by means of the receivers page table. This avoids complex modifications of page tables.

Hence, short 'Long Messages' which are transferred within one timeslice are faster. Really long messages take a little more time due to the overhead of calling the kernel and possible multiple lookups of addresses. An obvious optimisation could be implemented in a future version, in which the PALcode would handle IPC region TB misses directly and not call the kernel mode page fault handler.

7.3.2 Direct Strings or Message Words

Direct Strings or Message Words are 64-bit wide and located at the beginning of the message. They are transferred directly to the address pointed to by the message vector of the receiver. If the Flexpage bit

[1] quadwords located directly behind the dope
[2] the message consist of a set of pairs of a pointer and the string length

in the send vector is set, all these Message Words are copied and treated in addition as Flexpages. This continues as long the Flexpage descriptor is valid; hereafter the words are treated as sequence of integers and are only copied.

7.3.3 Indirect Strings

The message itself contains only references to indirect strings. Since a message buffer is used for sending and receiving of messages, pointers to strings to be sent and pointers to regions for receiving strings alternate with each other.

7.3.4 Flexible Pages

Flexible Pages, or Flexpages for short, are memory objects which can be passed via IPC from one address space to another. This data type is base for all external pagers to reply to requests on page faults. Flexpages can cover a set of 2^n continuous pages. The smallest page size is 8KB, which is equal to the hardware supported page size.

The Intel L4 version uses the given send address to decide between sending an ordinary memory page or I/O space. The Alpha version, however, allows 3 types of Flexpages.

If the flexpage bit in the send vector is set, all Message Words are treated as Flexpages until such a message word is a not valid Flexpage descriptor. Figure 7.3 illustrates the descriptor used in the message, table 7.4 lists a descrtiption of the descriptor fields.

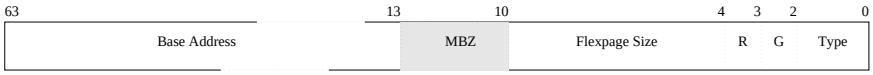

Figure 7.3: Flexpage Descriptor

Sigma 0 benefits most from this extension. While the Intel version has to build up page tables for all frames which should be accessible, the Alpha version does not. Sigma 0 can do this as follows;

1. Create a receiver thread, called SigmaR 0, which accepts Flexpages from Sigma 0 only.
2. If a mapping is required for a frame, send a Physical Memory Flexpage to SigmaR 0. This establishes a mapping in SigmaR 0's address space, which is equal to Sigma 0's.
3. Now, Sigma 0 is able to handle this page as appropriate, i.e. write to it or send it to a device driver.

Only Sigma 0 should be allowed to use Physical or I/O Memory Message and only for frames which are not used by the kernel.

7.4 Summary

L4 IPC appears in five variations and allows threads to send structured messages. Every message is described by a message dope. All mandatory arguments are passed in registers.

Indirect strings provide an easy way to send memory based data from various locations within the address space without copying them together.

Flexpages are the base for external pagers and make it possible to transfer memory to other address spaces without touching the data inside the pages. Three types of Flexpages can be sent, which are virtual memory, physical memory, and I/O memory.

Field	Name	Description
`type 00`	Invalid	Used for compatibility
`type 01`	Virtual Memory	Describes an ordinary page of virtual memory, mapped in the sender's address space by its virtual address.
`type 10`	Physical Memory	The address points to a frame instead of a page. This permitts the establishment of mappings, eg. for graphic adapters, which do not have pre-established kernel mappings. This capability should be restricted to Sigma 0.
`type 11`	I/O Memory	This type is especially for systems with separate I/O space like the Intel, or systems which require hints about caching strategies for TB entries, like the MIPS. Since the Alpha uses memory mapped I/O to access I/O ports and is able to handle caching to these regions without software hints, this type is completely equivalent to Physical Memory.
G	Granting	If this bit is set, all pages will be granted, otherwise mapped.
R	Read-Only	If set, all pages will be mapped read-only, otherwise read-write. An 'executable' bit may be added at a later time.
Size	Flexpage Size	The size describes the region to be transferred in 2^n bytes. (I.e. 13 means 8KB and is the smallest valid size). A value of zero covers the entire address space.
Base	Base Address	Address of the first Flexpage.

Figure 7.4: Flexpage Types

Chapter 8

Optimization

Optimization assumes good understanding of the hardware structures. Despite very good optimizing compilers, the best code is always created by hand. To begin with the algorithm used should be as good as possible, before the code is optimized.

8.1 Istream Considerations

I-stream considerations regard packing and alignment of code, instruction issues, or branch prediction.

Procedure entries should be octa-word aligned, frequently used branch targets at least quadword aligned.

Some Alpha implementations are able to execute instructions in parallel. This depends on the availability of execution units for the required instruction. Two instructions can be executed in parallel only if they are independent of each other.

An unexpected change of the I-stream address results, in many Alpha implementations, in about 10 additional cycles. Unexpected means, in general, missing predicted branches. Even a correctly predicted branch is slower than straightforward code.

Loops should be unrolled. In the more often used case a jump should have a target which has an address less than the jump instruction. Infrequently used code should be put completely out of the frequently executed instruction stream. and *behind* the straightforward code stream.

Since Alpha code tends to be very large; effort spent in increasing the compactness of code (and hence the density of I-stream) may be well spent.

Instruction scheduling influences the performance. Some instructions on the Alpha have a latency of two cycles, which means the result can be used without delay only two cycles later.

8.2 Data alignment

Data alignment is one of the most important facts for improving performance. The basic rule is to align data to their natural size.

Data which are often used together should be arranged near each other. Avoiding cache conflicts is achieved by adjusting data with different offsets within the cache lines.

The code should use ascending memory addresses for consecutive reads and writes. Intercept consecutive reads and writes after 8 cycles to avoid memory bandwidth overrun.

Chapter 9

Implementation

The following sections describe some of the interesting implementation details of various parts of the L4 kernel. The kernel consists of two virtual parts. The first is written in PALcode and contains the interface and some highly optimized procedures for address translation, interrupt handling, and short messages. The second part runs in kernel mode and contains the code which has to be interruptible.

9.1 Address Translation

Address translation is done completely in PALcode. An average translation, which includes entering and leaving the PALmode, traversing a three level page table tree and filling the Translation Buffer is done in 86 cycles.

If a translation is performed successfully, the pair (virtual address, TB entry) is written to the Translation Buffer. All page tables are accessed via physical memory only. The root of a page table tree is stored in the Internal Processor Register 'ptPageBase'. All 1024 page table roots are stored in an array.

Non resolvable translation faults lead to a jump to three different entry points in kernel mode, depending on the cause. This code either sends messages to external pager threads, or handles access faults to special parts of the kernel address space. Three different return points for these handlers allow them to write a value into the TB, rerun the causing instruction or start the scheduler.

The algorithm, explained in subsection 5.3.2.4 is implemented as follows

```
        HDW_VECTOR (PAL_ITB_MISS_ENTRY)
pal_itb_miss_entry:
        mtpr    a0, pt7                         // pt7 is PAL_TEMP reg 7
        mtpr    t0, pt0
        mtpr    t1, pt1
        mtpr    t5, pt5
        mtpr    t6, pt6
        mfpr    a0, excAddr                     // Get fault address
        mtpr    t3, pt3
        ldiq    t6, 0x3c0                       // Protection area in address
        ldiq    t5, 0x1c0                       // Bits to be inverted by XOR
        bis     a0, a0, t3

        mfpr    t0, ptPageBase                  // t0 = PageBase
        bic     a0, t6, a0                      // 0x3c0 - clear prot. area in address
        mtpr    t2, pt2
        bis     a0, t5, a0                      // XOR inverts only X, W, P bits
        mtpr    t4, pt4
```

```
        bic     t0, 0x3f, t2                    // t2 = old.frame start base
        srl     a0, t0, t1                      // t1 = table_bits (a0)
        ldiq    t6, PP_EXECUTABLE | PP_INTREE   // required protection
1:
        s4addq  t1, 0, t1                       // t1 = t1 * 4
        s4addq  t1, t2, t2                      // t1 = t1 * 4 + t2
        /*stall*/
        ldq_p   t1, 0(t2)                       // load lower part of gpte
        ldq_p   t0, 8(t2)                       // t0 = higher part of guard
        /*stall*/
        xor     a0, t1, a0                      // guard bits and flip prot.
        bic     t0, 0x3f, t2                    // t2 = old.frame start base
        srl     a0, t1, t4                      // shifted guard
        and     a0, t6, t1                      // P - check
        bis     a0, t5, a0                      // XOR inverts only X, W, P bits
        bne     t1, itb_protection_miss         // P - check
        srl     a0, t0, t1                      // t1 = table_bits (a0)
        beq     t4, 1b
        br      zero, itb_guard_fault

itb_protection_miss:
        // Depending on t1 bits error handling
        bic     t1, PP_INTREE, t1               // PP_INTREE is always set
        bne     t1, itb_protection_fault

itb_translation_valid:
        bne     t4, itb_guard_fault
        mtpr    t3, tbTag                       // Virt. Address
        mtpr    t0, tbCtl
        mfpr    t1, pt1                         // Required cycle
        mtpr    t0, itbPte                      // Page Table Entry
        mfpr    t2, pt2
        mfpr    t3, pt3
        mfpr    t4, pt4
        mfpr    t5, pt5
        mfpr    t6, pt6
        mfpr    a0, pt7
        mfpr    t0, pt0                         // Restore Scratch Register
        hw_rei
```

This code was examined using the DEC PALcode Violation Checker (PVC) tool and cycles were measured with the Alpha internal processor Cycle Counter (CC). The loop contains 12 instructions and takes 11 cycles + 2 stall. Two instructions can be issued in parallel. The complete code for an ITB miss, including parsing a three level page table, TB fill, entering and leaving PALcode takes 26 cycles for saving and restoring used registers, *3*13* cycles for the parsing loop and 18 for entering and leaving PALcode. On an 133 MHz 21064 Alpha, each TLB fill takes *0.6μs.* The resulting calculated and measured 83 cycles compares favorably with the 400 cycles for other software implementations, like Mach. All instructions in the code are ordered very carefully, to minimize required stalls. This led to a 25% faster version than the first implementation. The measured case does not consider cache misses during load of page table entries.

9.2 Thread and Context Switches

Thread switches perform only the reload of the kernel stack pointer and the continuation of the control flow at an appropriated address in the new thread. They do not perform any context switches, which are not necessary if the new thread is running in the same address space.

```
switch_thread (tcb, label)
    pal_addr (AT, label)        ; address in PALcode for label
    subq    sp, 8, sp           ; get space on stack
    stq_a   AT, 0(sp)           ; store the schedule in address
    ldiq    AT, TCB_MASK
    bic     sp, AT, AT          ; get my tcb address
    stq_a   sp, TCB_KSP (AT)    ; store my kernel stack pointer (ksp)
    /* SWITCH ! */
    ldq_a   sp, TCB_KSP (tcb)   ; load ksp from target thread
    ldq_a   AT, 0(sp)           ; load the schedule in point
    addq    sp, 8, sp
    ret     zero, (AT)          ; jump there
```

Context switches are performed if the next thread belongs to a different address space. On the 21064 with an untagged TB, this leads to a flush of all entries in the TB, except those with a set *Address Space Match* (ASM) bit. Hence, all kernel pages which are shared between all address spaces have this ASM bit set. The code for the context switch takes 14 cycles. Additional costs are expected due to the later occurring TB misses.

```
switch_context(tcb, tx)             ; tx is a temporary register
    task_nr (tcb, tx)               ; get the new task number
    pal_addr (AT, pal_ptroots)      ; pal address of ptroots
    s8addq  tx, AT, AT              ; build offset into table
    mfpr    tx, ptPageBase          ; get current PageBase
    ldl_p   AT, 0(AT)               ; load new PageBase
    xor     tx, AT, tx              ; are they equal
    beq     tx, 9f                  ; no, than continue
    mtpr    AT, ptPageBase          ; set new PageBase
    mtpr    zero, xtbAsm            ; flush the ITB and DTB
    STALL                           ; ... requires wait cycles
    STALL                           ;
    STALL                           ;
    STALL                           ;
9:                                  ; now in new address space.
```

9.3 Identification and Context Block

Each thread is identified by an unique number, called **Thread Ident** (TID). This value is unique in *time* and *space*. The most important parts of the TID are the thread and the task number. L4 supports a fixed amount of tasks and possible, parallel existing threads within them, these values are used to index a table, where all *Thread Control Blocks* (TCB) are stored.

Since preemption is possible in kernel mode, every thread contains its own kernel stack, which is located in the upper half of the TCB. The start address of the TCB can be determined by clearing the lower 11 bits of the stack pointer.

```
tcb (tcb)
    ldiq    AT, TID_M_VER0      ; Create the mask 0x7ff
    bic     sp, AT, tcb         ; Clear this bits in stack pointer
```

To calculate the TCB from a given TID, the following code is used. It sets the upper part of the address to `0xffffffff.exxxxxxx` and clears the lower eleven bits.

```
tcb_of_tid(tid, tcb)
    ldah    tcb, 0xe000(zero);
    lda     AT, TID_M_VER0 (zero);
    or      tcb, id, tcb;
    bic     tcb, AT, tcb;
```

9.4 Short Messages

Short messages are the fastest version and allow passing up to 8 registers from the sender to the receiver. The contents of the registers t0...t7 are passed to the receiver. This code runs completely in PALmode and takes 400 cycles for a RPC round trip. This apears to be greater than the 240 cycles required for IPC on Intel L4, but results in *1.45µs* on a standard 275MHz 21064 or about *0.92µs* on an 433MHz 21164, which compares favorably to the *2.4µs* on a 90MHz Pentium system.

The following extract shows the code for transferring short messages.

```
/*-------------------------------------------------------------------------
** Continuation of IPC
**
**      a0      - receiver id
**
**      a1      - snd_msg_dope
**      a2      - snd_msg_vect
**
**      a3      - rec_msg_dope
**      a4      - rec_msg_vect
**
**      a5      - msg_timeout = (receive = 63..32 | send = 31..00)
**
**      s6/fp   - receive flexpage register
**
**      t0..t7  - msg0 .. msg7
**
** SIDE EFFECTS
**      thread + task switches occures, so TLB misses cannot be avoided
*/

        ALIGN_BRANCH_TARGET
l4_ipc_cont:
        open_frame                              // Faults during touching TCBs
        // 3 * send_msg_vect != NIL -> run through, if data to be send
        addq    a2, 1, t8                       // test for send_msg_vect > VSPACE
        beq     t8, ipc_receive_only            // yes, jump to ipc_receive_only

        // 6 * inner clan transfer
        tcb     (t8)                            // test for different clans
        ldq_a   t10, TCB_MYSELF (t8)
        xor     t10, a0, t11
        srl     t11, TID_S_SITE, t11                    // Bits above 33 have to be identical
        bne     t11, ipc_send_chief             // no, jump to ipc_send_chief
```

```
        // 8 * valid target ID
        tcb_ptr (a0, t9)                        // t9 = TCB of target TID.
        ldq_a   t10, TCB_MYSELF (t9)            // t8 = ID, stored in TCB
        cmpeq   t10, t9, t11                    // target id == id of (target) ?
        bne     t11, ipc_send_dest_not_exist    // no, jump to ipc_send_dest_not_exist

ipc_send:
        // * partner in open or closed wait
        ldl_a   t10, TCB_FINE_STATE (t9)
        bic     t10, TFS_CLOSED_WAIT, t10
        beq     t10, 1f                         // It is a closed wait
        bic     t10, TFS_OPEN_WAIT, t10
        beq     t10, ipc_send_exec              // Open wait ? wonderful, so do the transfer
        br      zero, ipc_send_enqueue

        ALIGN_BRANCH_TARGET
1:
        // * partner in closed wait, does it wait for me ?
        ldq_a   t10, TCB_WAIT_FOR (t9)          // Wait for entry of partner
        ldq_a   t11, TCB_MYSELF (t8)
        cmpeq   t11, t10, t10
        beq     t10, ipc_send_enqueue           // No, then take first from send_queue

ipc_send_exec:
        // * Only a short message is handled here
        bne     a1, ipc_send_long               // send_dope must be zero for short msg

ipc_send_exec_from_long:
        // 1 * set partner to running
        bis     zero, TFS_RUNNING, t10          // ok, partner can now be started again
        stl_a   t10, TCB_FINE_STATE (t9)

        mark_busy (t9, t8, t10, t11)

        // * Only a send operation
        addq    a4, 1, t10                      // is rec_vect empty, then it's a send only
        beq     t10, ipc_send_only

        stq_a   fp, TCB_FLEXPAGE(t8)            // store my flexpage register
        stq_a   a3, TCB_COMM_RDOPE(t8)          // store my receive dope in my own TCB
        stq_a   a4, TCB_COMM_ADDRESS(t8)        // store my comm address
        blbc    a4, ipc_call_open               // is it an open cal

ipc_call_close:
        bis     zero, TFS_CLOSED_WAIT, t10      // No, we wait for partner's reply
        stl_a   t10, TCB_FINE_STATE (t8)        // We are in CLOSED_WAIT for a0
        stq_a   a0, TCB_WAIT_FOR (t8)

        switch_context (t9, t10)
        switch_thread (t9, ipc_send_ok)         // Switch to partner, if we come back, jump to ipc_se
        // --> to ipc_send_ok

        ALIGN_BRANCH_TARGET
ipc_call_open:
        ldq_a   t10, TCB_SEND_ROOT (t8)         // Is a thread waiting in queue ?
        bne     t10, ipc_is_sender
```

```
        bis     zero, TFS_OPEN_WAIT, t10        // No, set state to open wait
        stl_a   t10, TCB_FINE_STATE (t8)

        switch_context (t9, t10)
        switch_thread (t9, ipc_send_ok)         // Switch thread to receiver
        // --> to ipc_send_ok

        ALIGN_BRANCH_TARGET
ipc_send_only:
        bis     zero, TFS_RUNNING, t10          // Set partner running
        stl_a   t10, TCB_FINE_STATE (t9)

        mark_busy (t9, t8, t10, t11)            // Mark t9 in busy queue

        switch_context (t9, t10)
        switch_thread (t9, ipc_send_ok)         // Switch to t9
        /* NEVER REACHED */


        // * return with success
        ALIGN_BRANCH_TARGET
ipc_send_ok:
        bis     zero, IPC_SUCCESS, v0           // IPC, ok
        close_frame
```

Chapter 10

Summary

This chapter summarizes the most important achievments of this work and gives an overview of the scope for further work.

10.1 Performance

The first implementation of the Alpha L4 kernel was made on an Alpha 21064, EV4 workstation system called *Sandpiper*. All performance measurements are related to this 133MHz clocked machine, using the *Processor Cycle Counter*.

10.1.1 Page Table Translation

Page Table Translation is done in PALmode, using Guarded Page Tables. It takes about *18+26+13*level* cycles and compares favourably to other software implementations. Since Intel processors use a hardware built-in page translation, comparison of performance is not useful.

10.1.2 IPC

Table 10.1 shows IPC performance for transferring short IPC (first line) and long IPC messages[1]. The instruction cycles and timing results are very similar to the results on a 133MHz Intel Pentium, but the much higher system clock of new Alpha processors results in higher IPC throughput.

Message length	**Cycles**	**Xfer time (μs)**	*b (MB/s)*
0 + 64	368	2.7	-
8 + 64	2550	19.2	0.42
32 + 64	2618	19.7	1.62
256 + 64	3038	22.8	11.2
2048 + 64	6144	46.2	44.32
8188 + 64	17020	128.0	64.0
8192 + 64	18680	140.5	58.3
16376 + 64	44980	338.2	48.42
16384 + 64	46050	346.2	47.32

Figure 10.1: IPC Roundtrip Transfer Times

[1] +64 means 8 registers are passed to the receiver at any time

The left figure of 10.2 shows the cycles used for the different message sizes, the right figure presents the transfer bandwidth achieved.

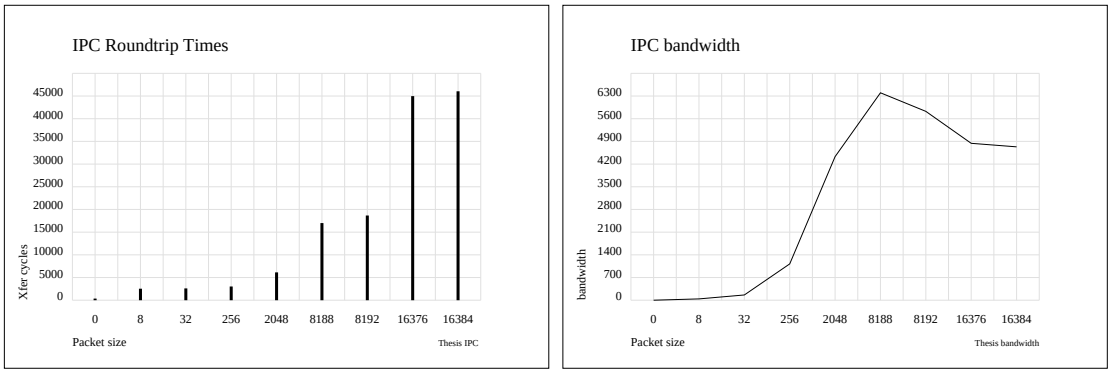

Figure 10.2: IPC Roundtrip and Bandwidth

Sending Flexpages is used for handling page faults by external pagers. The achieved results are acceptable; times for sending the first and the second page are quite high. This relies on that page tables must be created on the fly and existing entries in the page tables have to be splitted. Table 10.3 illustrates this values. Regarding the times to transfer 8 pages and 16 pages, it can be assumed that one page is transferred in appiroximately 580 cycles or 4.3μs.

Pages	**Cycles**	**Xfer time (μs)**	**time per page (μs)**
1	19283	144.9	144.9
2	33152	249.3	124.7
3	33611	252.7	84.2
4	34025	255.8	63.9
8	35700	268.4	33.35
16	40344	303.7	18.98

Figure 10.3: IPC Flexpage Transfer Times

10.2 Further Work

In further work, the remaining parts of L4 should be implemented completely. Some protection checks have to be added and the hardware related parts should be ported to the newer 21164A processor.

An implementation of the Guarded Page Tables could be done in hardware, for instance a VLSI chip.

Optimization of all parts for a more better performance should be done.

In addition to these, the Linux/L4-Intel would provide be a base for a Linux/L4-Alpha port.

Appendix A

Interface

This part gives an overview of the Alpha L4 kernel Interface. Arguments are given in the argument registers a0 .. a5, from left to right.

A.1 ipc

Transfers, depending on the arguments the registers `t0 .. t7` to the target thread. The `fp` register is used to receive flexpages. **Timeouts** contains the packed values for the send and receive operation.

Arguments:

```
thread_id target : a0
quad send_dope : a1
quad send_vect : a2
quad rec_dope : a3
quad rec_vect : a4
quad timeouts : a5
quad flexpage : fp
quad values : t0 .. t7
```

Results:

```
quad result : v0
quad partner : pv
```

Calling Sequence:

```
call_pal 0x0080
```

A.2 fp_unmap

Releases mappings, initiated directly or indirectly by the caller. Flush includes the caller's address space, unmap does not.

Arguments:

```
quad address : a0
quad attribute : a1
```

Results:

```
quad result : v0
```

Calling Sequence:

```
call_pal 0x0081
```

A.3 id_myself

Returns callers thread id, if next_thread is nil or the chief of the given thread.

Arguments:

```
thread_id next : a0
```

Results:

```
thread_id myself_or_chief : v0
```

Calling Sequence:

```
call_pal 0x0082
```

A.4 switch

Switches either to the given thread or if not possible to the next schedulable.

Arguments:

```
thread_id next : a0
```

Results:

```
none
```

Calling Sequence:

```
call_pal 0x0083
```

A.5 scheduler

Sets or reads the values for scheduling.

Arguments:

```
thread_id thread : a0
thread_id preempter : a1
quad parameter : a2
```

Results:

```
thread_id partner_id : t0
thread_id old_preempter : t1
quad old_paramenter : t2
quad time_slice : t3
```

Calling Sequence:

```
call_pal 0x0084
```

A.6 thread_ex_regs

Changes the IP register for a thread, defines the pager and starts this thread.

Arguments:

```
thread_id thread_no : a0
thread_id preempter : a1
quad parameter : a2
```

Results:

```
thread thread_id : v0
```

Calling Sequence:

```
call_pal 0x0085
```

A.7 task_new

Creates or deletes an address space.

Arguments:

```
thread_id thread_no : a0
thread_id chief : a1
thread_id pager : a2
quad initial_ip : a3
```

Results:

```
thread_id task_id : v0
```

Calling Sequence:

```
call_pal 0x0087
```

Appendix B

Thread Control Block

The Thread Control Block is a structure that contains all data which are necessary to reactivate or maintain threads. The most important fields are briefly described here.

`TCB_MYSELF` contains the Thread ID for this thread. It is necessary to keep this in the TCB, since the kernel uses only the address of the TCB to access them or to verify a user given TID[1].

The fields `TCB_PRESENT_QUEUE`, `TCB_SEND_QUEUE`, `TCB_BUSY_QUEUE` are used to maintain this thread. All existing threads are linked together in the 'Present Queue', runnable threads are linked in the 'Busy Queue'. Threads waiting for a Send-IPC to this thread are linked in the 'Send Queue'. The `TCB_LIST_STATE` reports in which queue the thread is linked in.

`TCB_COMM_PARTNER`, `TCB_COMM_ADDRESS`, `TCB_COMM_RDOPE`, and `TCB_FLEXPAGE` are used for Inter Process Communication. `TCB_PREEMPTER` and `TCB_SCHEDULE` contain information for the external scheduler interface.

`TCB_COARSE_STATE` keeps the state of the Thread Control Block. Possible states are *used*, *free*, or *blocked*.

`TCB_FINE_STATE` stores the state of the thread in a bitmask which are *running*, *locked_waiting*, *locked_running*, *polling*, *open_wait*, *closed_wait*, and *aborted*.

The complete structure is defined in `l4pal.h`.

[1] A TID can point to a valid TCB, but may not match the version number in the TCB

Appendix C

Glossary

ASN
Address Space Number, on the 21164 a seven bit wide field in the translation buffer to avoid flushing of the TB. A TB entry is only valid, if the ASN field in the TB and the ASN-IPR are equal.

ASM
Address Space Match, single bit in the translation buffer entries to allow restricted flushing of the translation buffer.

AXP
Trademarked name for the Alpha processor series.

DTB
Data Translation Buffer, TB for data page references.

ES
External Scheduler, thread which receives a message from threads if they are preempted. It can be used for implementing different scheduling strategies.

EP
External Pager, thread which receives a message from threads if they are causing a page fault. The EP replies a memory object to this message using standard IPC.

GPT
Guarded Page Table, structure by which a virtual address is translated into a physical address.

IPC
Inter Process Communication, allows data transfer between different address spaces.

IPR
Internal Processor Register, control register of the Alpha

ITB
Instruction Translation Buffer, TB for code page references.

MMT
Memory Mapping Tree, structure which saves information about mappings of pages in all address spaces.

PAL
Privileged Architecture Library, Code, which consists of standard instruction to perform hardware dependent functionalities.

ptReg
PAL_TEMP register, additional register, accessible in PAL and kernel mode, which can only be used for storing information.

PVC
PALcode Violation Checker, tool to check binaries for violation rules.

RPC
Remote Procedure Call, combined send and receive IPC operation, to send parameters and a command to a server and wait for the result.

Sigma 0
First External pager of the system. Maintains physically existing main memory and shares this between address spaces.

Sigma 1
Special external pager, responsible for page faults in the Thread Control Block Area.

TB
Translation Buffer, caches recently used address translations

TCB
Thread Control Block, 2048 byte big memory part, which contains thread state informations and a kernel mode stack.

TID
Thread Ident, 64-bit structure, used to identify a thread uniquely during the system up-time.

TMA
Temporary Mapping Area, 2GB part of the virtual address space where mappings of the receiver address spaces are established in the sender address space while copying memory based messages.

Bibliography

[1] *DEC* DECchip 21064 and DECchip 21064A, Hardware reference manual 1994 *Digital Equipment Corporation*

[2] *Richard L. Sites* Alpha Architecture Reference Manual 1992 *Digital Equipment Corporation*

[3] *Jochen Liedtke* Some Theorems About Guarded Page Tables 1993 *GMD SET-RS*

[4] *Jochen Liedtke* Page Table Structures For Fine-Grain Virtual Memory 1994 *GMD SET-RS*

[5] *Jochen Liedtke, Kevin Elphinstone* Guarded Page Tables on MIPS R4600 1995 *UNSW-CSE-TR-9503*

[6] *Sebastian Schoenberg* Guarded Page Tables on Alpha 21064 1996

[7] *Sebastian Schoenberg* Porting L4 to Alpha 1996

[8] *J. Stankovic, D. Niehaus, and K. Ramamritham* SpringNet: An Architecture for High Performance Distributed Real-Time Computing, Workshop on Parallel and Distributed Real-Time Systems 1993

[9] *Roscoe, Timothy* The structure of a Multi-Service Operating System 1995 *TR 376*

[10] *Bershad. B, et al.* Extensibility, Safety and Performance in the SPIN Operating System 1995

[11] *Jochen Liedtke* L4 Reference Manual, 486, Pentium, Pentium Pro 1996 *GMD SET-RS*

Index