Design and Implementation of the L4 Microkernel for Alpha Multiprocessors

Daniel Potts, Simon Winwood, Gernot Heiser

UNSW-CSE-TR-0201
February 2002

disy@cse.unsw.edu.au
http://www.cse.unsw.edu.au/~disy/
Operating Systems and Distributed Systems Group
School of Computer Science and Engineering
The University of New South Wales
UNSW Sydney 2052, Australia

Abstract

This report gives an overview of the techniques used in the multiprocessor implementation of the L4 microkernel on the Alpha processor family. The implementation is designed to be scalable to a large number of processors, which is supported by keeping kernel data processor-local as much as possible, and minimising the use of spinlocks and inter-processor interrupts.

Contents

1 Introduction

This document describes the mechanisms used within the L4/Alpha microkernel to extend the L4 API to systems with multiple processors. Basic knowledge of the L4 microkernel is assumed; for a more general discussion of L4, see [Lie95, Lie96], as well as the user manual [AH98]. Basic knowledge of the Alpha family of microprocessors is also assumed; see [Com98] for more information.

For an introduction to L4/Alpha see the L4/Alpha reference manual [PWH01].

1.1 Goals

The development of this microkernel was done with the following goals in mind:

- Extend the kernel API to support SMP in an neat, unintrusive way;
- Minimise on an SMP-aware kernel the overhead to processor-local operations (i.e., operations that do not multiple processors);
- Minimise communication between different processors as much as user level programs permit;
- Provide a scalable kernel for SMP and ccNUMA machines. Sharing between processors needs to me minimised to ensure scalability.

2 Communication Mechanisms

This section describes the low-level communication mechanism used within L4. This takes the form of *mailboxes*, a form of inter-processor message passing.

Section 2.1 discusses the thread control block (TCB) and goes on to discuss new thread attributes *ownership* and *CPUID*. These form an essential part of concurrency protocols used within L4, and have a direct bearing on the system call implementations.

Section 2.2 discusses the mailbox mechanism, including the implementation and a usage example.

Section 2.3 discusses a hypothetical implementation of an interprocessor system call.

Section 2.4 discusses the various special cases related to these mechanisms.

2.1 Thread Control Blocks

Modifying TCB state is one of the most critical components when adding SMP support to L4. It is no longer safe to assume that TCB-related operations in the kernel can be maintained atomically without some form of locking. Having to use some form of locking mechanism for each TCB operation that needs to be performed atomically would impact on the fast case where interacting threads are on the same processor. On Alpha, depending on hardware configuration, locking can take almost 200 cycles.

To avoid the use of locking a TCB, we only allow the processor that is currently the *home* processor of the thread to modify the TCB state. When a thread on the local processor wishes to modify another local thread's state, it may do so using the current uniprocessor mechanism. A remote thread must send a request to the destination thread's processor to complete the operation on its behalf.

2.1.1 Ownership and CPUID

This section discusses the TCB variables *ownership* and *CPUID*, both of which refer to processors within the system (possibly the same processor).

The CPUID variable refers to the processor on which the thread is currently scheduled to execute user code — it is its *home* processor, where the user has bound this thread to execute. The CPUID processor may modify only those variables which relate directly to it, such as those in relation to timeout queues. This includes the *send root* of that thread, which is the head of a queue containing all threads waiting to send to that thread; the *present* queue which contains a list of threads which have that processor as their CPUID, and the *CPUID* variable itself.

The *ownership* variable refers to the processor which has the right to modify the thread's state, including its stack. This is the processor upon which the thread is currently executing some operation. This may be different to the CPUID *home* processor only if the thread is executing a system call. The owner may modify any variables in the TCB apart from those which belong to the *home* processor. Only this processor may modify the *ownership* variable, so only this processor can hand another processor ownership of a thread. Ownership transfers, along with requests for ownership, form the bulk of thread-related messages between processors.

These variables remove the necessity for locks within the TCB, as only one processor is allowed to modify specific TCB variables, and only that processor can pass on that right to another processor.

These variables are in separate cache lines in the TCB to ensure a minimal impact when accessing these variables.

2.2 The Mailbox Abstraction

The central kernel-to-kernel communication mechanism in L4/Alpha is the *mailbox*. The mailbox mechanism provides temporal ordering of messages to a processor, as well as some concurrency control between processors.

Each processor owns a number of mailboxes[1] which other processors may fill with messages. A mailbox consists of a message type, a valid bit, and three 64-bit message-dependent words of data (one cache block), as depicted in Figure 1.

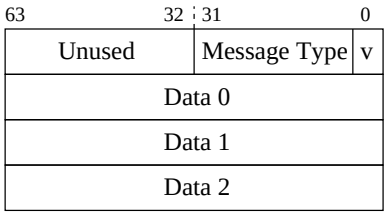

Figure 1: The mailbox data structure

Mailbox allocation on Alpha is performed by atomically incrementing a counter that indicates how many mailboxes are in use. This is done via a simple load-locked/store-conditional pair on the counter. If this fails, it will simply try the atomic load/store again. Once a mailbox has been allocated, filling in the data items may be performed using normal load / store operations. The message will not be acted upon by the remote processor until it sees the valid bit set. A memory barrier instruction on the local

[1] Currently, this number is fixed at compile time.

processor is executed to ensure this data is coherent with memory and so that the message and valid bit are visible to the remote processor; no other locking related programming is required.

Mailboxes are allocated in FIFO order to assist in providing some ordering of messages relative to the remote processor. This helps in reducing the number of possible states a thread may be in when it is performing remote system calls as messages become serialised. Without such serialisation, it is possible that a request to perform an operation on a remote thread arrives at the remote processor before the thread does, for example, in the case that it has recently been migrated to the remote processor.

The general usage of a mailbox is as follows:

1. Allocation of the mailbox.

   The processor allocates a mailbox owned by the remote processor. Any message sent by another processor to the target processor will be processed *after* this processor's message. This gives temporal ordering between processors, but note that the target processor is not included in this ordering.

2. Fill in the data items.

   The processor fills in any message data that needs to be communicated to the remote processor. This does *not* include the message type.

3. Commit the message.

   The message type is filled in, and the message is marked as valid. Until the message is marked as valid and the remote processor sees it, the remote processor will not process the message.

4. (Optionally) send the interprocessor interrupt.

   This notifies the remote processor that the message needs to be processed. This step may be omitted for performance optimisation reasons. For example, if the processor is sending multiple messages to the remote processor, if the processor is sending multiple messages to multiple processors and wants to group the interrupts, or if the message is not urgent.

When a processor receives an interprocessor interrupt (IPI), it starts processing mailboxes in order, until there are no valid mailboxes remaining.

Note that the processing of these mailboxes is done within the context of the thread that happened to be running at the time the processor was interrupted. This is an attempt to reduce costs related to a thread switch. This has two implications: firstly, any message that moves a thread off a processor needs to ensure that the thread is not the current thread; secondly, the message itself cannot have any context, so all processing must be done immediately, with interrupts disabled.

Typically these operations have been designed to be only a small number of instructions with the majority of the operation performed at a later time in the context of the thread which it affects.

The time between when the IPI is sent and when the remote processor is interrupted can be significant, so any code that uses the mailboxes must take this delay into consideration. Latency on Alpha has been measured to be several thousand cycles for an IPI. The actual overhead, i.e. the number of cycles actually executed by the processors involved in the IPI, is much smaller.

2.3 Basic Interprocessor System Call

This section describes a basic system call which requires an interprocessor message. For simplicity, the system call will read the number of timer interrupts that the remote processor has received (for more realistic examples, see Section 3).

2.3.1 Operation in general

The time line for this system-call is shown in Figure 2.

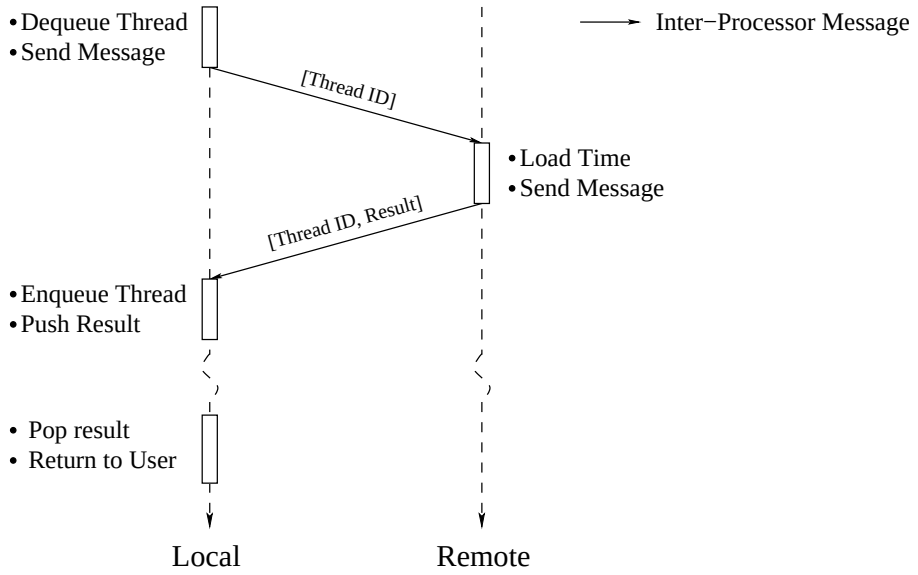

Figure 2: Basic interprocessor system call example

The system call is executed in 4 stages:

- Dequeue the thread and send the message.

  The thread is dequeued on its *home* processor for the duration of the system call by removing it from the scheduling busy list, and a message is sent to the remote processor. This message includes the thread ID, so the local processor knows which thread executed the system call when the remote processor replies.

- Perform the action and reply.

  The local time is read and a message is sent to the thread's processor. This message includes the thread ID and the result.

- Enqueue the thread on the *home* processor's scheduling busy list and push the result onto its stack.

  The result needs to be pushed onto the stack as we can't just restart the thread; there may be more messages to process, or there may be a higher priority thread running.

- Pop the result and restart the thread.

  When the thread is next scheduled, the kernel will pop the result off the stack and return the thread to user mode.

2.3.2 System Call Code

Assuming that the system call accepts the destination processor in a0 and returns the result in v0 (this is the usual semantics), the system call code[2] would be as in Figure 3.

This forms the bulk of the system call that is executed when the caller enters the kernel. The code is quite simple, with the majority of the complexity hidden behind macros. The code is essentially a boilerplate for any system call which needs to send an interprocessor message. The destination processor is checked, the thread is removed from the scheduling queue, the mailbox is allocated, filled in, committed, and an IPI is sent as per Section 2.2. Finally, the processor is yielded to the dispatcher, and the thread's restart address (the address at which it begins executing when next scheduled) is set to the `remote_ticks_return` address.

An important point to note regards the label given as the last parameter to the `alloc_mailbox` macro. This is the target to branch to if there are no free mailboxes. A sensible policy here is to return a failure code to the user, or to try again (interrupts should be enabled and disabled so that any pending interrupts can be processed). This macro allows for not specifying an error label. This is for compatibility with old code and should not be used. The kernel will panic if there is no free mailbox and no error label is provided.

2.3.3 Remote Code Executed by IPI Handler

The remote processor executes the `ipi_remote_ticks` code to process the IPI for the given message type as shown in Figure 4.

The mailbox should be freed as soon as possible so that another processor may use it. This is done using the `free_slot` macro.

It is important to note here the way in which the thread's processor is determined. The *CPUID* variable is used, not the *ownership* variable. The *ownership* variable is *not* guaranteed to be correct, and may change between the time the message is sent to when it is received. The thread is not even guaranteed to exist any more, or be executing the system call (another thread may have deleted the task, migrated the thread, etc). This needs to be checked on the thread's *home* processor. If the thread's *home* processor is no longer the same as CPUID when the message arrives at the processor designated CPUID, the message is forwarded or a negative acknowledge may be sent. This result depends on the operation.

2.3.4 Sending the Interprocessor System Call Result

The last major piece of code is that which restarts the thread and stores the result on the stack as shown in Figure 5.

The thread id and CPUID both need to be checked to make sure that the thread exists, and belongs to the executing processor. If the thread has changed, the message is dropped. If the thread has been migrated off this processor, the message is forwarded (`ipi_forward_message` is a global routine which copies the contents of the mailbox in pp3 and re-sends the message to the correct processor).

If all is correct, the thread is enqueued in the scheduling queue, and the result is stored on the stack. The next time this thread is scheduled, it will execute the `remote_ticks_return` procedure, which just pops the return value off the stack and returns to user mode.

[2]This example code needs further code to actually make it callable by a user thread. A discussion on how this is done is beyond the scope of this document.

```
remote_ticks:
        ;; Check that the processor exists
        check_cpu_exists        a0, t0, t1
        beq     t0, remote_ticks_out

        disable_int t4

        ;; Make some space on the stack for the result
        addq        sp, #8, sp

        ;; Dequeue the thread.
        get_current_tcb     t0
        dequeue_busy  t0, t1, t2, t3

        ;; Send the message (mailbox result in t0)
        alloc_mailbox  t0, a0, t1, t2, t3, remote_ticks_full

        ;; We have a mailbox here, so put record my TID so we know
        ;; who to return to when we are finished.
        tid_internal   t1, t1, t2
        stq_p   t1, MB_DATA0(t0)

        ;; Finally, commit the mailbox and send the IPI.
        commit_mailbox MB_REMOTE_TICKS, t0, t1
        send_ipi   a0, t0, t1

        enable_int t4
        cpu_tcb_dispatcher  t0, t1
        switch_thread_kernel  t0, <remote_ticks_return>, t1, t2, t3

        ;; We return -1 to indicate failure.  A more robust
        ;; implementation would have error codes.

remote_ticks_out:
        subq    zero, #1, v0
        return_frame

remote_ticks_full:
        tcb     t0
        enqueue_busy  t0, t1, t2, t3

        subq    zero, #1, v0
        return_frame

remote_ticks_return:
        ;; Pop the result off the stack and return.
        pop        p_v0
        return_frame
```

Figure 3: System call message dispatch

```
ipi_remote_ticks:
        ;; Free the mailbox (in pp3) as soon as possible.
        ldq_p   pp0, MB_DATA0(pp3)
        free_slot  pp3, pp1, pp2, pp4

        ;; Get the destination processor.
        tcb_ptr  pp0, pp2
        ldq_a   pp2, TCB_CPUID(pp2)

        ;; Send the reply.
        alloc_mailbox  pp3, pp1, pp2, pp4, pp7

        stq_p   pp0, MB_DATA0(pp3)

        ;; Get the number of ticks
        ptldq   pp1, ptCurrentTicks
        stq_p   pp1, MB_DATA1(pp3)

        commit_mailbox  MB_REMOTE_TICKS_REPLY, pp3, pp4
        send_ipi  pp2, pp3, pp4

        ;; Process next message
        br    zero, ipi_message_loop
```

Figure 4: System call message dispatch remote handler

2.3.5 Adding IPI Handler for a New System Call

The only remaining modifications are the addition of the `MB_REMOTE_TICKS` and `MB_REMOTE_TICKS_REPLY` message types to the enum defining the message types and the addition of the `ipi_remote_ticks` and `ipi_remote_ticks_reply` entries to the IPI message vector. The IPI message vector is a straight forward jump table, which uses the message type value to determine which IPI handler routine to jump to.

Although this system call is relatively simple, a substantial amount of code is needed. Fortunately, the majority of this is boilerplate code, and can be taken from existing system calls.

2.4 Special Cases

There are cases that occur when threads are no longer where they are thought to be due to the mechanisms used to communicate between processors. For example, threads may no longer exist or may have moved to another processor.

Thread migrate requests that are destined for a processor that is no longer the *home* processor (also known as the *CPUID* field of TCB) of that thread are forwarded to the current *home* processor or rejected.

Any operation that requires thread state modification, such as thread migrate, has the potential to cause the destination thread to abort what it is doing. As for destructive ex_regs() system-call operations in the uniprocessor kernel, system calls such as thread migrate can cause an in progress IPC to be aborted along with any further pending IPCs. This reduces the complexity of calls such as thread migrate, and reduces the possible number of states a thread can be in at any one time.

```
ipi_remote_ticks_reply:
        ldq_p   pp0, MB_DATA0(pp3)
        ldq_p   pp1, MB_DATA1(pp3)

        ;; Check everything is OK
        check_internal_tid  pp0, pp2, ipi_rtr_out, pp4, pp7

        ;; TCB is in pp2
        check_cpuid   pp2, pp0, ipi_forward_message, pp4, pp7

        ;; If we get here, everything is OK, and we can continue.
        free_slot  pp3, pp0, pp4, pp7

        ;; Store the result on the stack of the thread.
        ldq_a    pp4, TCB_KSP(pp2)
        stq_a    pp1, 0(pp4)

        enqueue_busy  pp2, pp1, pp3, pp4, pp6

        br   zero, ipi_message_loop

        ;; If the thread doesn't exist, just drop the message.
ipi_rtr_out:
        free_slot  pp3, pp0, pp4, pp7
        br   zero, ipi_message_loop
```

Figure 5: IPI reply handler example

Figure 6 shows the execution of an ex_reg() on a thread that is currently executing an IPC call. *Requester* is the CPU from which the ex_reg() is being executed. *Owner* is the CPU that the thread to be ex_reg()'ed is currently running on, performing a remote IPC operation. *Home* is the CPU that owns the thread the ex_reg() is being performed upon. The steps involved are as follows:

- The *requester* determines the thread is currently performing an operation (IPC) that needs to be aborted. It sends a "thread abort and ownership request" to the *owner*.
- The *owner* receives the request. If the thread exists on this CPU and is performing an operation that needs to be aborted, it does so. The thread is then migrated to the *Requestor*. If the thread does not exist on the *owner* CPU, the request is forwarded to its next apparent location.
- The ownership change arrives at the *requester* which then performs the required ex_reg() operation. The caller of ex_reg() is now free to resume execution, having obtained the required results that ex_reg() returns. If the thread is not on its *home* CPU it is migrated to *home* where the final component of the abort process required by ex_reg() is performed. This involves aborting any thread attempting to perform an IPC to the ex_reg()'ed thread.
- The ex_reg()'ed thread returns to its *home* processor where it resumes execution at the new PC.

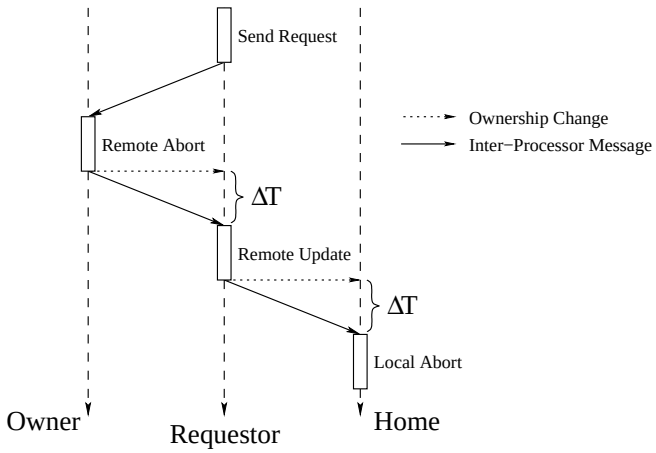

Figure 6: Time-line for multiprocessor ex-regs operation

3 System Call Implementation

3.1 Interprocessor Inter-Process Communication

An IPC operation that is requested between threads on different CPUs utilises the mailbox mechanism as discussed in Section 2.2.

A concise description of the procedure is as follows:

- Sender detects destination thread is not on local CPU.
- Register based component of message is pushed onto sender's stack.
- A mailbox message is built containing a pointer to sender's thread context.
- The mailbox message is committed (sent) as described in Section 2.2, the thread is made unrunnable and the scheduler is invoked to run the next runnable thread.
- Destination CPU decodes mailbox message and takes appropriate action.

  This component needs to be fast as not to interfere too heavily with the current thread that has been interrupted. This operation checks the state of the receiver. If the receiver is waiting to receive this IPC, it is made runnable and its start address is set to that of the sender. If the receiver is not ready to receive the message, the sender is enqueued in the receiver's send queue as in the normal uniprocessor IPC case.

  No actual message transfer is performed during this time. In order for the IPC operation to complete, the receiver must receive a time slice on its local processor. Note that time slices are not donated between processors.

- Sender thread is invoked on receiver's CPU via a ready receiver thread (thread expecting message and has obtained a time slice) and IPC message transfer is performed using the IPC routines.

  Before switching to the receiver to transfer register-based message contents, a mailbox message is built and committed containing IPC success status back to the sender's *home* CPU.

- Sender's CPU decodes mailbox message and makes sender runnable.

Timeouts are always added on the message destination CPU's timeout queue. This avoids potential race conditions if the timeouts where queued on the sender's *home* CPU.

The majority of the interprocessor IPC operation can be performed using a shared path of IPC code also used by a local processor IPC. As a result, SMP functionality does not greatly impact on code size.

Benchmarks indicate that interprocessor IPC is bound by IPI latency. On a dual Alpha 21264 system, a short IPC operation measures approximately 3300 cycles. The measured raw IPI and device interrupt time does not appear to differ greatly from this value.

3.2 Exchange Registers

In order for classic exchange register system call operations to be performed on a multiprocessor system some changes were needed.

When the thread is not on the same processor as the caller the following occurs:

- A thread abort and ownership request IPI message is sent. This causes any operations that need aborting (such as IPC) to be aborted. A success message is sent back to the caller thread's processor.
- The caller thread is restarted and the exchange register occurs.
- Any final thread clean up is performed that causes any threads that are polling the callee to be aborted.
- The thread is migrated to its *home* processor if it is not already on it.

When the thread is on its *home* processor the exchange register occurs the same as for the uniprocessor case.

3.3 Scheduling

In order to minimise the level of complexity and the amount of communication that takes place when maintaining global scheduling queues, we chose to implement separate scheduling queues per processor. The user is able to modify these scheduling queues by the thread_migrate() and thread_schedule() system calls.

3.3.1 Migrate

Migrate is currently implemented as a separate system call in L4/Alpha. The core of the migrate operation is performed as follows:

- If the thread is not on the caller's processor, forward the request to the processor it is currently executing on.
- If the thread is migrating itself then switch to the dispatcher and perform the operation on its behalf.
- Abort any in-progress thread operations such as IPC. This performs the same steps an exchange register system call would do to abort a thread. See Section2.4 for details on the abort process.

- Remove thread from current processor's run queue.
- Send migrate enqueue message to destination processor.
- Destination processor receives message and enqueues the thread in the processors busy list and makes it runnable.

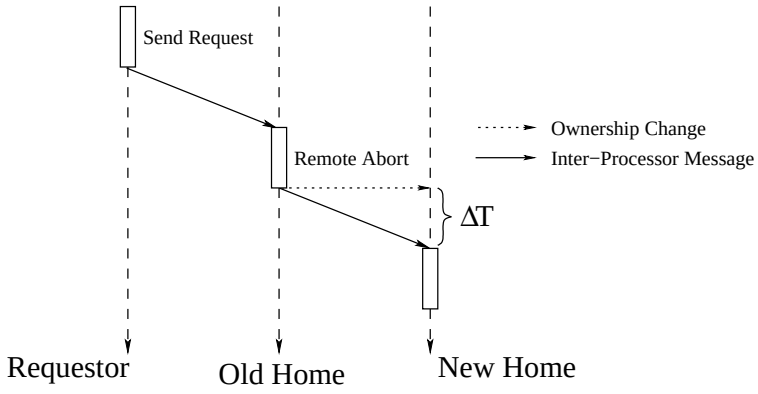

Figure 7: Timeline for multiprocessor migrate operation

Figure 7 shows a migrate operation in which the caller performing the migrate system call is not on the same processor as the thread being migrated. The request is forwarded to the processor the thread is currently executing on called *Old Home*. Once the migrate request arrives at the current *home* processor of the thread (*Old Home*), the migrate operation is performed. The thread is migrated to *New Home* where it resumes execution. This diagram represents a simplification of the possible execution path that may be required to migrate a thread. Specifically, in this example, the thread to be migrated is not currently executing any operation, such as IPC, that needs to be aborted.

3.4 Task Creation

The task_new() system call was modified to include a CPUID parameter that allows the user to choose a CPU that the new task's active thread will start executing on. This avoids cache thrashing that may occur from using migrate to move the running thread.

The system call behaves the same as the uniprocessor implementation, however, if the active thread is to be scheduled on a different CPU, a thread enqueue request is sent to the destination processor. The thread will then get enqueued in the remote processors scheduling queue.

3.5 Minimising IPI Use

In order to minimise the use of IPIs and the interruptions they may cause, some rules are followed:

For operations that are not on the fast path, an IPI is not sent with a message. Instead, the message will be processed by the remote CPU when it next receives an IPI, or between preemption points.

Rather than using the mailbox and IPI mechanism for kernel data structure maintenance, locking is used. In general, this is limited to simple operations such as linked list manipulation.

4 Kernel Internals

This section discusses several kernel internals that have not been discussed so far.

4.1 Address Space Identifiers

On Alpha, Address Space Identifiers (ASIDs) are a per-processor entity. Rather than using a global ASID to tag tasks across a multiprocessor system, L4 maintains that an ASID is only relevant on a local processor. This ensures that there is no impact on scalability while effectively allowing for a greater working set of address spaces.

4.2 TLB Consistency

While cache and memory are kept coherent by hardware across all processors on Alpha, TLB coherency is not maintained across multiple processors. This causes a problem when a page table mapping that is removed on one processor, may be cached in the TLB of another.

The unmap system call performs the following set of operations to ensure that any address space unmap is global across the whole system:

- Remove mapping from page table.
- Flush mapping from local processor's TLB.
- Determine which processors may have this mapping in their TLB and send them a message requesting it to be removed.

  A table is maintained for each processor that contains a list of currently active L4 task ids. This table is looked up on each processor of the system to determine if a message requesting an address space be flushed from the TLB, needs to be sent.

  This component is not currently implemented. It is likely to be a bit array with each bit corresponding to a L4 task id. The scheduler sets a relevant task id bit in this array when a thread from this task is scheduled. This bit get cleared, when the ASID corresponding to this task id, gets flushed.

  This method believed to be cheaper and less intrusive than sending a message to each processor in the system requiring them to flush an address space, possibly unnecessarily.
- Remote processors receive the request, flush mapping from their TLB and send acknowledgement message back.
- Once all acknowledgement messages have arrived at the caller, the system call returns.

4.3 Kernel Memory

Allocation of kernel memory for use in page tables, TCBs and other kernel data structures requires some form of cooperation between processors on a multiprocessor system.

L4 Alpha maintains a global free list which is initialised by the primary CPU at system start up. When some memory is needed by the kernel, this list is first locked, some frames allocated, then the list is unlocked.

To minimise locking of this global free list, each processor also maintains a free list which it attempts to use first when requiring some memory. This list is replenished by allocating several pages from the global free list at once, in order to reduce contention of the global free list. When a processor frees a page, it is freed to the local processor's free list.

Currently no mechanism has been implemented to deal with an empty global free list.

4.4 Multiprocessor Page Tables

Currently there is no implemented page table structure that takes multiprocessor issues, such as the need to lock page table entries, into account. A generalised virtual memory management module for 64-bit versions of L4 is presently under development at UNSW by Chris Szmajda, we therefore left this issue unresolved until that work is progressed further.

4.5 Interrupts

Each Alpha CPU has five interrupt lines, some of which may be registered by user level threads to receive interrupts from. This includes one device interrupt, which multiplexes all PCI/ISA device interrupts. In most motherboard designs, this device interrupt is routed to all processors on a multiprocessor system.

L4 allows these interrupt lines to be directly registered by user level threads running on each local processor, as it does in uniprocessor kernels. As a result, systems may be built in which load balancing of device driver interrupts may be implemented. This is left up to user level threads to cooperate on a suitable way for this to be done.

No restriction is placed on a thread that has an interrupt line associated with it from migrating to another CPU. It will still be responsible for handling the interrupt from the CPU with which it originally requested the association. Responsible user level programming should avoid this.

5 Benchmarks

This section shows some of the cycle counts of various operations in the L4/Alpha kernel and show that support for multiple processors does not significantly impact operations that do not span multiple processors.

These cycle counts are not meant to show the results of an optimal kernel. This kernel has not been optimised and includes code that impacts as a performance hit on 21264 systems due to the nature of the PAL code environment on this system. It has been verified that some operations such as reading a word back that was just written adds an extra 200 to 300 cycles to the overall result. Some parts of this code does such operations for validation purposes. On 21264 systems it is also possible to use caching hints to prefetch and evict data that should provide much lower cycle counts.

Storing the cycle counter value on this system appeared to add between 40 and 120 cycles to the overall results. This value has not been deducted from the results below. The out-of-order execution of this code also highlighted another shortcoming of these benchmarks when measuring the cycle count. For example, adding no-op instructions when measuring cycle count altered the results favourably.

Table 1 shows the cycle count for the task_new() system call, and the IPC receive system call in order to show the impact of the multiprocessor support. The uniprocessor kernel (*UP kernel*) tests show costs (in cycles) for a kernel compiled with support for only one processor. This results in a kernel that does not have any multiprocessor code compiled in. The next test is for the same operation on a multiprocessor kernel (*MP kernel*). In this test the operation is kept on a single processor. The final test shows how the operation behaves when the operation occurs between two processors.

Operation	Type	Average cycle count
Task new	UP kernel	1843
	MP kernel, same processor	2025
	MP kernel, remote processor	2521
IPC receive - sender ready	UP kernel	231
	MP kernel, same processor	231
	MP kernel, remote processor	1104
Raw IPI latency		3300

Table 1: System call cycle times

In the multiprocessor task_new() test, cycle times for the operation of creating a new task on a remote processor are measured. Effectively, this calculates the time to allocate the new TCB for the address space and first active thread then send the IPI message off to the remote processor.

The results for task_new() show the overhead required when supporting multiple processors due to the requirement to verify that the task can be created on a particular processor. In the case that it is being created on a remote processor, the added time for allocating a *mailbox* and sending the IPI are included. Note that while an IPI is sent for the task_new() system call, due to the nature of such a system call, it may be reasonable not to waste an IPI in this case for the reasons mentioned in Section 3.5.

In the multiprocessor IPC test, the receiving thread is waiting on a CPU different to the sender's. This was useful to measure as it highlights the cost of the message transfer from one CPU's cache to another. For a single transfer of 8 bytes, this is believed to be around 80 to 120 cycles.

The receive case for both uniprocessor and multiprocessor kernel is identical code, hence we see the same average cycle time when the sender and receiver are on the same processor.

The IPI latency measured appears to correspond to interrupt latency for all system interrupts (including device interrupts) on a typical Alpha system.

6 Future Work and Discussion

It appears that performance optimisations may be possible for an architecture once it is known how it responds to different methods of synchronisation. As a result, further experimentation should involve choices of utilising locking semantics and IPI's based on the respective performance of a particular architecture.

The SMP API needs to be carefully developed to ensure that an increase in kernel complexity occurs as little as possible. While a goal of this project was to minimise complexity in the kernel, the addition of SMP support has shown that the number of possible states a thread can be in at any one time is greatly increased leading to the risk of missed corner cases.

Other issues such as how to handle time slices during long interprocessor IPC need to be dealt with carefully. Time slice donation policies will have a significant impact on SMP kernel implementation.

We have not discussed the virtual memory model in this paper. A future area of work will involve looking into ensuring that maintaining coherent page table structures is scalable for large multiprocessor systems, especially ccNUMA machines. The memory management of the kernel should also support an arbitrary number of memory segments such as those present in ccNUMA machines and visibly export them to the user via sigma0.

It appears that there needs to be a mechanism in place to prevent any threads in the system from creating new threads or migrating threads to CPUs that may not be intended for it. The use of a priority

based scheduling scheme is not easily manageable on several processors.

Possible solutions include the use of CPU sets that associate each task or thread group with a list of processors that it may execute on. Other solutions could include alternate schedulers that rely on time slice donation, or hierarchical schedulers [Win00].

References

[AH98] Alan Au and Gernot Heiser. *L4 User Manual*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, January 1998. UNSW-CSE-TR-9801. Latest version available from http://www.cse.unsw.edu.au/~disy/L4/.

[Com98] Compaq Computer Corp. *Alpha Architecture Handbook Version 4*, 1998. http://www.support.compaq.com/alpha-tools/documentation/current/alpha-archt/.

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

[Lie96] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[PWH01] Daniel Potts, Simon Winwood, and Gernot Heiser. *L4 Reference Manual: Alpha 21x64*. University of NSW, Sydney 2052, Australia, March 2001. UNSW-CSE-TR-0104. Latest version available from http://www.cse.unsw.edu.au/~disy/L4/.

[Win00] Simon Winwood. Flexible scheduling mechanisms in L4. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2000. Available from http://www.cse.unsw.edu.au/~disy/papers/.