UNIVERSITY OF NEW SOUTH WALES

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Performance Limits of Darwin on L4

Thesis Part B

Tom Birch

Bachelor of Science (Computer Science)

May 2005

Supervisor: Prof. Gernot Heiser

Abstract

Darwin is a modern operating system that forms the basis of Mac OS X, yet there are concerns about is performance due to its foundation on the Mach kernel. The L4 microkernel has taken many of the original goals of Mach and implemented them correctly, achieving high levels of performance. Darwin has been ported to run on top of L4, in order to take advantage of L4's high performance. This thesis details several areas of Darwin that have been modified to take advantage of L4's, and provides recommendations for further increases in the performance of L4/Darwin.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Darwin is a POSIX-compliant operating system derived from NeXTSTEP. It is part of Apple's Mac OS X, one of the 3 major desktop operating systems today. OS X is a closed source operating system, but Darwin is made available as an open source project by Apple. OS X is used by millions of people, and is heralded for its impressive features and ease of use. Despite this, there have been repeated concerns about the performance of OS X in various benchmarks when compared to Linux and Windows [IBM04].

Performance has always been a matter of great importance when designing operating systems. As with any engineering task, tradeoffs are required to achieve a balance between multiple goals. There are many goals or requirements of a modern operating system, namely convenience, abstraction, efficiency, ability to evolve and protection. Balancing all these requirements is becoming increasingly hard with the ever-growing demands of todays computers.

Many of the performance problems in Darwin are blamed on its foundation on Mach, one of the first microkernel attempts. While popular in its own time, it is now considered sub-standard when compared to current microkernel projects such as L4 and even monolithic operating systems such as Linux.

Recently Darwin was ported to the L4 Microkernel by the ERTOS group at NICTA [Emba]. It is still a work in progress, but is quite functional and able to run many existing Darwin binaries. Currently the port is only targeted at x86-based systems.

The aim of porting Darwin to L4 is to make Darwin take advantage of L4's improvements over Mach in areas such as IPC performance and memory management. While there is potentially much to be gained from porting to L4, care must be taken to preserve the original interfaces presented by the Darwin kernel in order for existing Darwin programs to function correctly.

Because Darwin has been ported with very little modification, there are many situations in which operations in L4/Darwin are more expensive than the same operation in native Darwin. This is because in addition to performing all the work necessitated by Darwin, there is also work that must be done by L4 or the glue code to tie everything together. An example of this is that Darwin system calls now travel over L4 IPC (Section 4.6), thus additional processing is required to pack and unpack the arguments to the system call into an L4 IPC message.

There are three goals of this thesis:

- To study the performance characteristics of the L4/Darwin port and identify areas where improvements can be made.
- To minimise any overheads introduced by L4 (e.g. for system calls) to ensure that the ported system is not worse than native Darwin.
- To explore various means of significantly modifying parts of Darwin to take advantage of features of L4.

I plan to use a wide selection of benchmarks to achieve the first goal and to help understand my progress in the second and third goals.

In the following sections I will cover

- The L4 Microkernel
- The motivation for porting Darwin to L4
- Darwin operating system and some its major components
- The design of L4/Darwin
- Implementation details and analysis of optimisations

Chapter 2

The L4 Microkernel

Before covering the specifics of the L4/Darwin port and the planned optimisations, it is necessary to first understand what L4 is and why it makes a good basis for a modern operating system. I will give a brief overview of the L4 Microkernel, and introduce two existing L4-based operating systems: L4/Linux and Sawmill Linux.

2.1 What is L4?

The microkernel approach to operating system design is aimed at removing functionality such as the file system and network stack from the traditional monolithic kernel and running it in userland. The main functionality in a microkernel-based system lies in user-level servers, and the microkernel simply provides a means for programs and servers to communicate and protect themselves from each other.

L4 is a 2nd-generation microkernel. Having learned from the mistakes of previous microkernel efforts [HHL+], the design of L4 emphasises a small *trusted computing base* (TCB), high performance and scalability [Embe]. L4 is designed to have a low cache footprint [Lie95] thus the overheads of trapping into L4 are reduced, which was just one of the problems with Mach's *inter-process communication* (IPC) primitive. L4 also emphasises only implementing the necessary primitives required for security, such as lightweight threads with unique identifiers and recursive address spaces.

Recursive address spaces in L4 are an interesting new concept. Processes may grant (and relinquish control of) or map (i.e. share) regions of their address space to other address spaces with L4 IPC. Processes can also downgrade previously mapped regions of memory or discard them completely. In Figure 2.1 we see *std pager* mapping a page to f_1, which then maps this page to F. At this stage *std pager*, f_1 and F all share the same page data. When F performs a grant operation, it releases control of the page, which is now shared between *std pager*, f_1 and *userA*.

L4 provides a synchronous (i.e. a send operation will block until the destination has received the message) IPC primitive to allow threads to exchange messages. IPC messages have a maximum length of 64 words (word length defined by the architecture) so in order to send long messages, shared memory must be used. The fact that L4 IPC

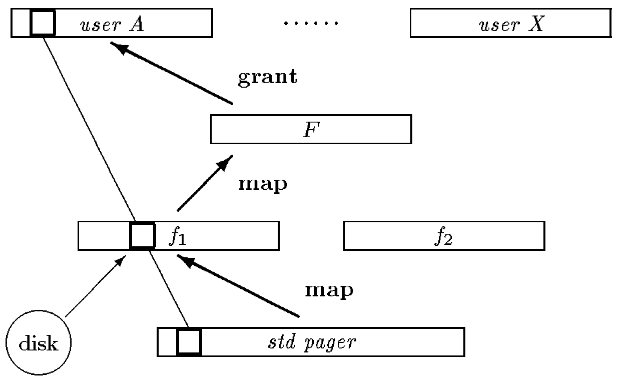


Figure 2.1: Recursive address spaces in L4 [Lie95]

is synchronous and has bounded message lengths means that the kernel can avoid any buffering of messages, reducing overhead and complexity in the kernel.

Interrupts in L4 are modelled as IPC messages. A thread registers to handle a specific interrupt number, and whenever that interrupt occurs, the kernel fabricates an IPC from a thread with the same id as the interrupt number. This is one of the properties of L4 that allows for efficient implementation of drivers on L4-based systems.

2.2 L4/Linux

There are two main research projects aimed at porting Linux to L4: the L4Linux project at TU Dresden [OSG] and the Wombat project by the ERTOS group at NICTA [Embf]. Both projects aim at minimal modifications to the Linux kernel, although Wombat is aimed towards embedded devices.

The Linux kernel runs in one address space and userland programs run in their own address spaces. IPC or exception handling is used for system calls (depending on which version of L4 is used) and L4 mappings are used to control memory access.

The work on L4/Linux is very similar to the L4/Darwin project as they are both POSIX [Ins05] operating systems with monolithic kernels, thus many challenges are common to the two projects (e.g. signal handling, system call despatching, scheduling). L4/Darwin borrows a lot form L4/Linux in terms of its approach to these problems.

2.3 Sawmill Linux

Sawmill Linux was a research project at the IBM T.J. Watson Research Center aimed at constructing a multi-server operating system on L4. Whereas in traditional Linux, all kernel functionality resides in the same address space, the Sawmill project has split up the Linux kernel into various modules, e.g. *Virtual File System* (VFS) and *Pluggable*

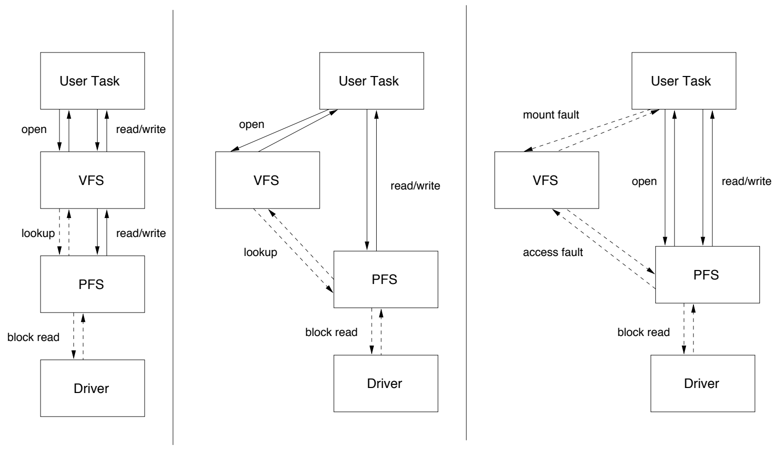


Figure 2.2: Multiple servers in Sawmill Linux [GJP+00]

File System (PFS) modules reside in different address spaces [GJP+00]. Sawmill also places a strict set of protection and semantic requirements on servers, and one of the goals is to implement these without significant performance degradation.

The different servers communicate via L4 IPC, and in order to solve the problem of a long chain of servers required for tasks such as VFS operations, user libraries were modified so that some operations communicate directly with the server responsible. For example, following the traditional VFS interface in Linux, a user task would have to contact the VFS server, which would in turn forward requests to the specific PFS server, which would then communicate with the driver (left side of Figure 2.2). This can be optimised however by modifying the user task to talk directly to the PFS server for read/write operations (middle of Figure 2.2). A further optimisation on top of this is to consult the VFS server only on mount operations, and talk directly to the PFS server for everything else (right side of Figure 2.2).

While this work provides a good set of guidelines to consider when implementing systems on top of microkernels, it is in contrast to the work on L4/Darwin as we are only considering a monolithic server approach.

Chapter 3

Darwin

In order to understand the motivation for porting Darwin to L4, it is necessary to understand exactly what Darwin is. Here I will introduce Darwin and discuss the two major components — BSD and Mach — and how they are combined in Darwin. I will also discuss some reasons why Darwin makes a good choice for a system to port to L4.

3.1 What is Darwin?

The Darwin operating system is an open-source subset of Apple's Mac OS X, which is derived from the NeXTSTEP operating system [App06c]. It is a full operating system by itself, capable of running many applications including the apache web server, MySQL and a whole host of other UNIX applications.

Because Darwin is a subset of OS X, it shares many components such as the kernel — XNU, dynamic linker — dyld, support for the Mach-O binary format and the driver framework — IOKit. Darwin is different in that it lacks many proprietary closed-source components of OS X (Figure 3.1) such as the window server, some proprietary kernel functionality (such as support for PowerPC binaries on i386), various user-level API's, and many drivers (although some are included in binary-only form). XNU, as shown in Figure 3.2, is a combination of the 4.4BSD kernel, the Mach 3 microkernel and IOKit

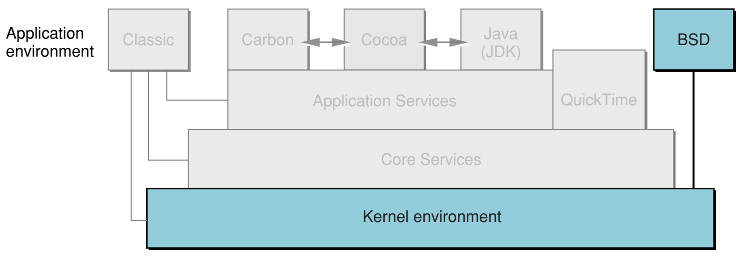


Figure 3.1: Mac OS X and Darwin architecture [App06b]

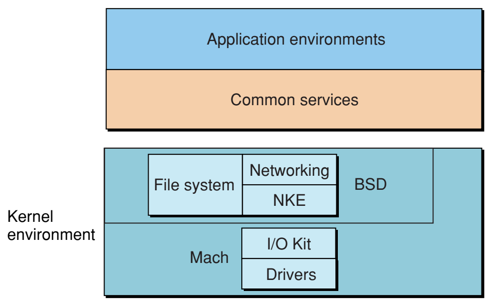


Figure 3.2: XNU Kernel Architecture [App06b]

(Section 4.7). Whereas most Mach-based systems run the guest OS in userland (with exception of Tru64 UNIX [OS] and NeXTSTEP [McC]), due to performance issues, the BSD part of XNU runs alongside Mach in protected mode, a technique known as 'colocation'.

Darwin is POSIX [Ins05] compliant and capable of running the majority of BSD programs with a simple recompile. There are also a large collection of Linux applications that have been ported to Darwin [Thea, Por].

3.2 BSD

The Berkeley Software Distribution (BSD) operating system was one of the first UNIX implementations, originating from the University of California, Berkeley in the 70s. Today there are many flavours of BSD including OpenBSD, NetBSD and FreeBSD which all derive from 4.4BSD. The BSD kernel implementation used in Darwin is 4.4BSD, with heavy modifications to run next to Mach (Section 3.4) while the userland components, including most of the libraries and common utilities, are from FreeBSD.

3.3 Mach

Mach is one of the earlier microkernel attempts, first developed at Carnegie Mellon University beginning in 1985. It was quite popular in its day, and there are many operating system projects based on Mach [Hel94, RMGB91, dPSR96, GDFR90, OS , McC], all of which are monolithic guest OSes running on top of Mach in userland or with colocation. In light of 2nd generation microkernels such as L4 however, it is clear that Mach has performance problems [HHL$^+$, Lie95].

Mach provides many facilities, including:

- an asynchronous, capability-based IPC primitive (Section 3.3.1)
- an *Interface Definition Language* (IDL) compiler (Section 3.3.2)

- a complex *virtual memory* (VM) system (Section 3.3.3)

In the following sections I will cover each of these in detail.

3.3.1 IPC

One of the most important features of Mach is its IPC functionality. Mach provides a very complex mechanism for communicating with other threads, allowing for not only data to be transferred, but also access rights to objects such as tasks and threads, drivers, regions of virtual memory, etc. Mach's IPC primitive is asynchronous and capability based, a contrast to L4's synchronous, thread-ID based IPC primitive (Section 2).

In Mach IPC, the most fundamental concept is a 'port'. A port represents a message box, to which messages can be written to and read from. Access to ports is capability based, and the capabilities in Mach IPC are represented by 'port rights'. There are three types of port rights:

- Send: there may be multiple send rights for a given port. All send rights grant the owner the ability to send an unbounded number of messages to the port.

- Send-once: a modified send right that becomes invalid once used. Send-once rights grant the owner the ability to send at most one message to a port. They are commonly used to implement replies.

- Receive: there is only ever one receiver for a port. Receive rights grant the owner exclusive access to read messages sent to a port, and use any typed data contained in such messages (such as port rights or out-of-line data discussed below).

Port rights are stored in a task-local namespace, thus copying a port right to another task without using the Mach API is meaningless. The kernel checks port rights on every IPC system call.

It is important to note that Mach IPC is asynchronous. Send calls return immediately, and the message is buffered in the kernel until the receiver performs a receive operation.

There are 3 types of data that can be sent through Mach IPC:

- port rights, which is how capabilities are transferred

- inline data that is contained within the message

- out-of-line data, represented by a pointer and a length in the message, which is interpreted by the kernel.

All data transfers in Mach IPC use copy semantics, i.e. there there is simple way to share a region of memory with Mach IPC. Memory sharing is possible however by creating a memory object and sending a port right for that memory object to another task.

```
routine task_info(
                target_task     : task_name_t;
                flavor          : task_flavor_t;
        out     task_info_out   : task_info_t, CountInOut);
```

Figure 3.3: An example MIG function definition

Call method	**Number**
BSD trap	∼270
Mach trap	35
MIG RPC	∼346

Table 3.1: Number of exported functions [App]
Numbers may vary depending on kernel version, target architecture and compile-time options

3.3.2 Remote Procedure Calls

One of the major uses of Mach IPC is *remote procedure call* (RPC). In a traditional operating system, kernel functionality can only be exported through system calls, whereas in Mach, another option is to export functionality as RPC over Mach IPC. The advantage of RPC over system calls is that functionality can be transparently implemented outside the kernel. For example, processes use Mach-IPC-based RPC to make calls to the window server.

Mach facilitates the use of RPC through the *Mach Interface Generator* (MIG). MIG is an IDL compiler that greatly simplifies the process of calling functions in other address spaces. Functions can be described by a small interface definition file, which specifies the arguments that function takes. An example stub for `task_info` is shown in Figure 3.3. MIG processes these files and generates client and server-side stubs.

The role of the client-side stub is to present a C function with the same prototype as the exported function. This function marshalls its arguments into a Mach IPC message, sends it to the appropriate port, gracefully handle any errors that may occur and returns an appropriate return value.

The role of the server-side stub is to handle incoming requests, un-marshall arguments and use them to call the actual implementation of the exported function, then pass return values and/or error status back through Mach IPC to the caller.

Many kernel functions are exported to userland through MIG RPC. Table 3.1 shows the number of these functions compared to system calls. Of particular note is the VM subsystem interface, including functions to allocate and alter the properties of regions of an address space. While adding to the cost of RPC when compared to system calls, due to the indirection through MIG stub code, MIG provides a lot of flexibility when defining cross address-space interfaces.

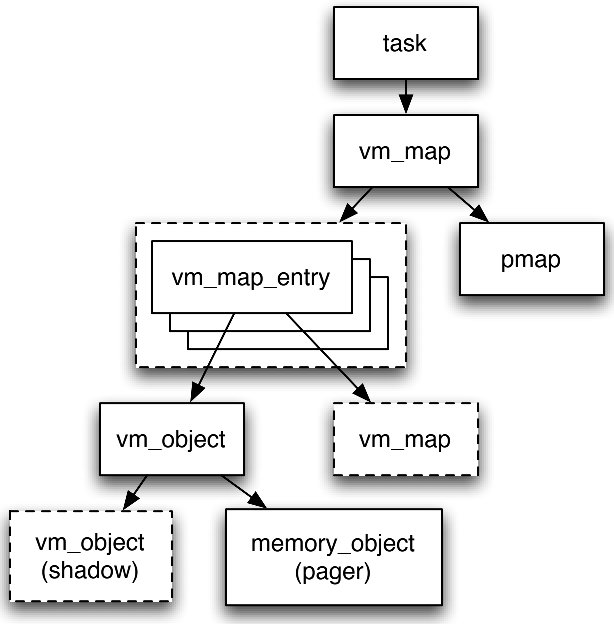


Figure 3.4: Mach VM structure

3.3.3 Virtual Memory

Mach has a very complex VM system [RTY+88, App06b] as shown in Figure 3.4. Mach VM is based around the idea that physical memory is a cache of the complete virtual address space. It aims to manage everything to do with the virtual address space, including paging functionality and shared memory. Mach VM in Darwin also provides extensive copy-on-write functionality that is used throughout many areas of Darwin (including IPC) in order to avoid unnecessary copying.

The Mach VM system is split up into its machine dependent (the `pmap` layer) and machine independent parts (the `vm_` interfaces) such that very little has to be modified in order to adapt it to a new architecture.

Each task contains a single `vm_map` which represents its address space, and this `vm_map` contains:

- a single pmap, the Mach structure for a machine dependent page table (which maps virtual address to physical addresses)
- a list of `vm_map_entry` structures

A `vm_map_entry` may may reference either a `vm_object` or another `vm_map` (for the purpose of sharing). This is commonly used for the shared text segment of a dynamic library: there will be a `vm_map` representing one or more shared libraries, and each task that wants access to those libraries will contain a reference to that `vm_map`.

Copy-on-write functionality is achieved through shadow `vm_object`s. When a `vm_object` is copied, an empty `vm_object` is created and its 'shadow' is set to point to the original `vm_object`. Read operations are forwarded to the shadow `vm_object`, whereas write operations cause data to be copied from the shadow and then modified. A shadow `vm_object` may itself have a shadow `vm_object`, this is known as shadow-chaining. Shadow chains can be arbitrarily long.

The actual data in `vm_object`s is provided by `memory_object`s, which are responsible for transferring page data between memory and the physical backing store, such as a hard disk or network. A `memory_object` can represent a memory-mapped file or the VM backing store.

The pmap layer manages all the hardware-specific details of virtual to physical and physical to virtual (ptov) translation of memory addresses. A pmap contains a page table and a ptov mapping, and provides an appropriate interface to modify these. The pmap layer is also responsible for managing the *translation look-aside buffer* (TLB). All TLB insertion and deletion operations are performed by the pmap layer, providing a convenient abstraction differences in the highly machine-specific TLB. Pmap removal and protection operations for example, translate directly into TLB remove operations.

When a pagefault occurs, the tasks pmap is first consulted. If it does not contain a mapping for the faulting virtual address, the corresponding `vm_map_entry` is looked up, and depending on whether it references a `vm_object` or `vm_map`, the object is consulted or the search starts again in the submap. If a `vm_object` is referenced, the search continues through the shadow `vm_object`s to find a translation, or if one isn't found, the referenced `memory_object` is used to retrieve the page data from disk.

3.4 Mach and BSD in Darwin

Mach and BSD are both free-standing operating systems, each implementing their own solutions to many issues that come up when implementing an operating system, such as abstracting threads and address spaces. When these two operating systems were combined in a monolithic kernel in Darwin, there were several cases where both Mach and BSD implemented abstractions or policy for the same concept, and each of these had to be merged in order to make a functional system.

One example of this is that the Mach and BSD kernels both have their own concept of an address space (Mach task vs BSD process) and a thread (Mach threads vs BSD kernel threads). In order for BSD and Mach to cooperate, each BSD process is mapped to exactly one Mach task, and the kernel threads are replaced with Mach threads.

Scheduling policy is another example, as both Mach and BSD implement their own schedulers. Since BSD threads and processes are mapped onto Mach threads and tasks respectively, the choice was made to use the Mach scheduler, and modify the BSD scheduler to make calls into Mach when it needs to change the priority of a thread or modify the run queue.

Another similar problem is that of paging and caching policy: Mach implements its own page cache, which is used to cache memory mapped files, whilst BSD has its own filesystem cache to speed up file accesses through read and write operations. The potential exists for a page to be cached in both the Mach and BSD caches, effectively wasting a page of physical memory. The solution in Darwin is the *unified buffer cache* (UBC) and the *universal page list* (UPL) interface [App06a]. The BSD filesystem cache and the Mach VM cache are combined in the UBC. Modifications to the UBC are achieved through the UPL interface. For example, VFS operations use the UPL interface to allocate pages for read and write operations.

There is a history of other projects involving a guest OS running on Mach (either as a user-level server or with colocation) that have had problems with matching policy implemented in Mach with the policy of the Guest OS, these include the MkLinux project [dPSR96], the Lites Server project [Hel94] and the DOS on Mach project [RMGB91]. Darwin has managed to resolve these problems by colocating Mach and BSD and heavily modifying both of them in order to resolve the duplication of abstractions. This colocation however prohibits any advantages derived from using a microkernel.

3.4.1 System calls in Darwin

XNU publishes both the full set of BSD system calls (`read`, `write`, `fork`, `exec`, etc.) and the full set of Mach system calls (`mach_msg`, `thread_self`, `task_self`, `thread_switch`, etc.) plus some extra system calls for managing the mix of Mach and BSD, e.g. `task_for_pid` for looking up a Mach task given a BSD process id. Both types of system calls are implemented as traps through the same exception handler (Mach traps all have a negative system call number, BSD traps are all positive) and are dispatched to the relevant routine in the Mach or BSD system call table. There are also about 350 other kernel functions exported to userland through RPC stubs generated by MIG(Section 3.3.2), including `host_page_size`, `thread_resume` and `memory_object_data_request`. The distribution of published kernel functions is shown in Table 3.1.

Chapter 4

L4/Darwin structure

Before covering methods of optimisation for L4/Darwin, I will first cover various implementation details that need to be understood in order to appreciate some of the optimisations. Here I will cover the basic design, and touch on various important areas in Darwin that were modified to work on top of L4.

4.1 The basic architecture

Darwin has been ported to run on top of L4 such that the modified XNU runs as a userland program in its own address space. Because Darwin running on L4 has the same privileges as any other user program and cannot directly control any hardware, it is now much easier to run multiple instances of it on the same machine. There are plans to do just this, as well as plans to run instances of the Wombat [Embf] server at the same time. This would allow clean separation of userland environments into separate virtual machines, and permit fast communication between a Darwin and Linux environment running on the same machine. There is still much work to be done on this however and it is not within the scope of this thesis.

Changes have been made to many of the architecture-specific parts of Darwin (mostly XNU and libc) in order for it to run on top of L4. Most of the changes are fairly straightforward, but there are some interesting problems when it comes to thread interaction that have proven challenging. Almost all of the modifications to XNU were made to the Mach component, as there were very little architecture specific parts of BSD that needed changing.

The following sections detail the most important areas of Darwin that needed modification to work on L4. Throughout the rest of this document, *kernel* refers to the L4/Darwin kernel server, i.e. the modified XNU kernel running in a single user-land address space on top of L4.

4.2 Virtual Memory

The Mach VM subsystem is very portable. All of the architecture dependent code lives in the pmap layer - an abstraction layer in Darwin to hide hardware-specific memory address translation discussed in Section 3.3.3. All that was needed to make Mach VM operational was to rewrite this code for L4's memory primitives.

The pmap operations map quite cleanly onto L4 primitives: a `pmap_remove` causes an `L4_Unmap`, `pmap_protect` and `pmap_page_protect` both cause an `L4_Unmap` of the revoked permissions, and a `pmap_enter` just inserts into the page table and ptov mapping, and the page table is looked up on a pagefault IPC, which causes an `L4_Map_Item` to be sent to the faulting thread. The operations on the kernel pmap translate to `L4_Flush` as pages must be unmapped/protected in the current address space. In order to save the cost of calling `vm_fault` on every page fault, we added a `pmap_fault` routine that checks if there is an entry in the pmap that satisfies the given permission, and returns that. We call this routine before `vm_fault` on every pagefault.

Mach VM is not fully functional on L4 yet, as reference/dirty bits are not yet implemented and paging in/out is not operational.

4.3 Threads

Darwin uses Mach threads as the fundamental thread abstraction, and managas all scheduling in the context of Mach threads. Darwin assumes that Mach threads run on raw hardware and be managed by the Mach scheduler. Since L4 provides its own abstraction of threads that run directly on hardware, Mach threads must be modified to accommodate this.

Each userland Mach thread maps onto exactly one L4 thread. In the kernel however, all threads share a single L4 thread and stack switching is used to change between them. This is due to the way interrupts are handled in an L4-based system. Interrupts have to be sent as L4 IPC messages to a particular thread, as opposed to interrupts being handled by the exception handler regardless of the current thread. Having multiple Mach threads hosted by a single L4 thread greatly simplifies the interrupt handling logic in L4/Darwin.

4.4 Scheduling

Darwin makes strong assertions about which thread is running at any given time. This makes sense as the Mach scheduler can enforce control over all switching between threads through its exclusive access to hardware. These assertions fail when Darwin is running on top of L4, which provides its own thread abstraction. To prevent unexpected events occurring, the Mach scheduler has been modified to keep every userland thread in a stopped state until it is chosen to run. This ensures that only the Mach scheduler's *current thread* may be running on an L4 thread at any point in time.

This implementation is sub-optimal however, as the L4/Darwin kernel server must be consulted for every scheduling decision. It would be preferable to take advantage of L4's own scheduler and allow it more freedom to choose the next thread to run. This is very difficult as L4 has a very simplistic fixed-priority round-robin scheduler, whereas Mach employs a much more complex scheduler policy. There is current (though yet unpublished) work on a more extensible scheduler for L4 however, and it is hoped that we will be able to get more efficient scheduling in the future by expressing the complex the policies of the Mach scheduler using L4's scheduling primitives.

4.5 Tasks

In L4/Darwin, Mach tasks are complemented by L4 address spaces, but some modifications to Darwin were required to make everything work correctly. Each Mach task maps onto exactly one L4 address space. The pager for each address space is set to be the L4/Darwin kernel server, and the L4 *user thread control block*(UTCB) area is decided by the kernel server. The number of threads in an L4 address space is currently limited by the size of the L4 UTCB area (i.e. the maximum number of threads is the size of the UTCB area, divided by the size of a UTCB, which for a given architecture is a fixed in size).

The address space layout of a task in L4/Darwin is shown in Figure 4.1. Currently significant portions of the address space are reserved by L4. The *kernel interface page* (KIP) and UTCB area are small enough to be insignificant in a virtual address space with 2^{20} pages, but the more than 10% of the address space used for the L4 kernel is unacceptable. It is hoped that future work on L4 will greatly reduce this, possibly even a transition to a physically-addressed L4 kernel [Nou05] to completely free up the virtual address region reserved by L4.

4.6 System calls

System calls require special attention in L4/Darwin in order for them to function correctly. The L4 model for dealing with events such as system calls and interrupts is vastly different from the traditional approach used in many other operating systems. As mentioned in Section 2.1, L4 models these events as L4 IPC messages. This is in contrast to the trap-handler model used in Darwin where system call execution in the kernel begins at the exception handler. In L4/Darwin a *system call loop* is used to handle all incoming interrupts and system calls. This loop accepts L4 IPC messages, processes them and replies accordingly. Currently system calls, pagefaults, timer ticks and console input all handled this way by the same thread.

Mach and BSD system calls in L4/Darwin are provided through L4 IPC. As shown in Figure 4.2, the system call stubs in libc have been modified to call a C function. This function packs all necessary information (including system call number and stack pointer) into an L4 IPC message which is sent to the L4/Darwin kernel server and handled by the system call loop. The system call number and stack pointer are unpacked, the

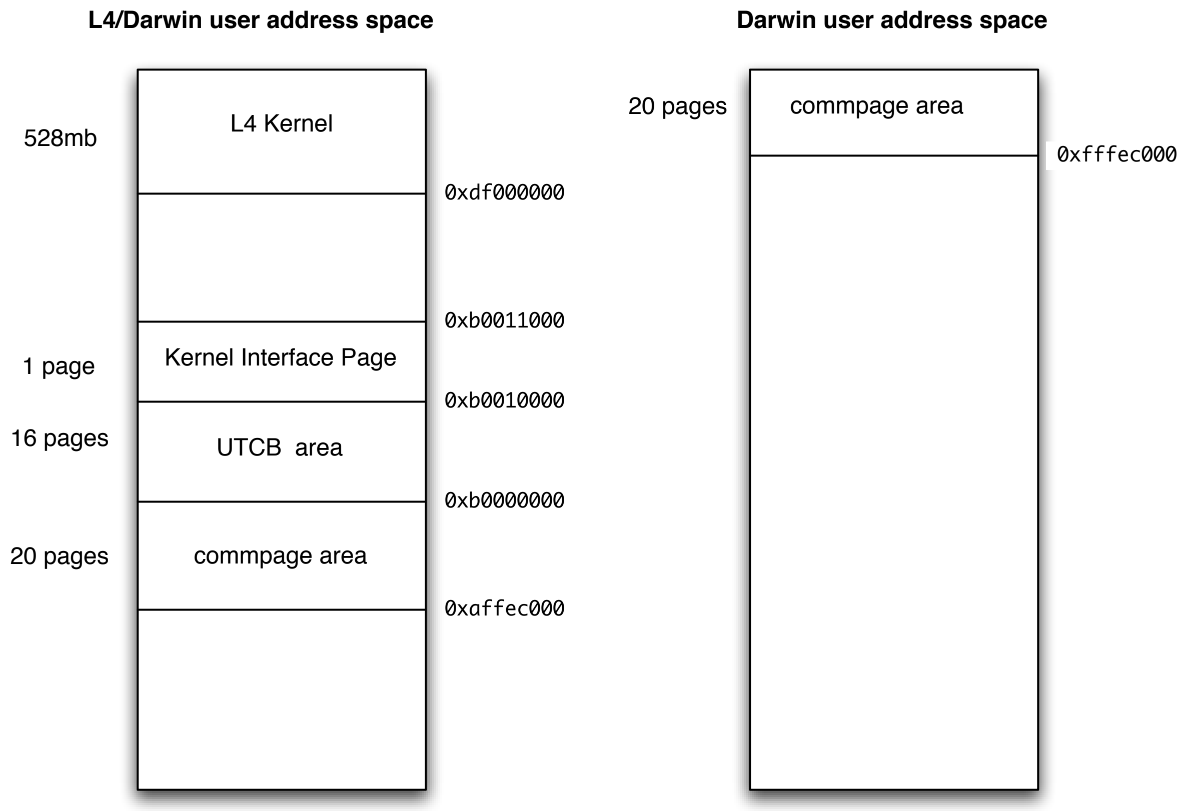


Figure 4.1: Address space layouts of userland processes in L4/Darwin and native Darwin

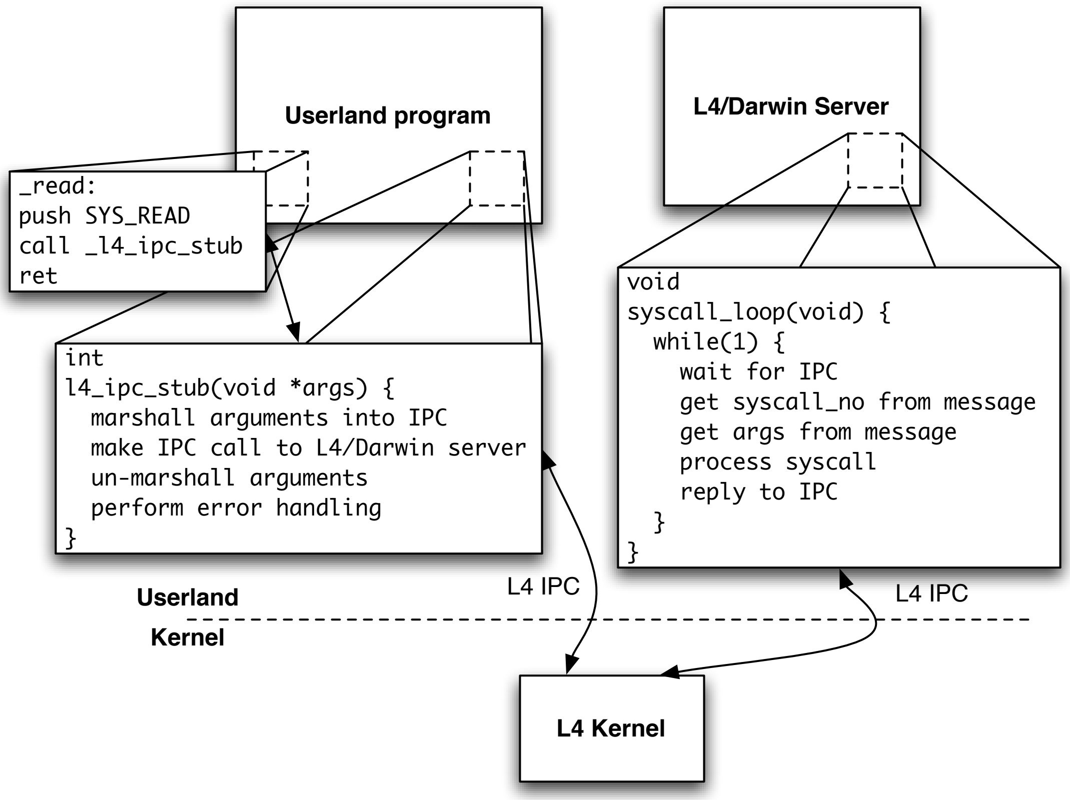
arguments are copied in from the stack and the appropriate call is made. A reply is sent back containing the return value and a flag indicating the error status.

There is a system call used by pthreads(described in Chapter 8) to set the thread-specific storage pointer. In Darwin this pointer is stored in a special IA32 segment register `%gs:0`. Since this register is reserved for use by L4, the pthread thread-specific storage pointer is stored in the `USER_DEFINED_HANDLE` field of the thread's UTCB.

4.7 IOKit

IOKit is the driver framework in Darwin and Mac OS X. It is separate from both the Mach and BSD parts of Darwin, is written in a stripped-down version of C++, and does not depend on any specific features of the Darwin kernel. All drivers in Darwin and OSX are provided through IOKit.

IOKit has been ported to L4 by Geoffrey Lee, and his undergraduate thesis covers its implementation [Lee05]. IOKit runs directly on top of L4, and makes use of L4 IPC for communication with the L4/Darwin kernel server. IOKit in L4/Darwin is able to load



```
_read:
push SYS_READ
call _l4_ipc_stub
ret
```

```
int
l4_ipc_stub(void *args) {
  marshall arguments into IPC
  make IPC call to L4/Darwin server
  un-marshall arguments
  perform error handling
}
```

```
void
syscall_loop(void) {
  while(1) {
    wait for IPC
    get syscall_no from message
    get args from message
    process syscall
    reply to IPC
  }
}
```

Figure 4.2: Current syscall implementation

and use binary drivers that ship with Mac OS X so supporting new Apple hardware as it comes out does not pose any significant problem in this respect.

Chapter 5

Approach and methods

In the following chapters I will cover the strategies I have chosen to optimise L4/Darwin, but first is necessary to discuss the overall strategy for optimisation. In the sections below I will discuss the areas where Darwin has performance problems, the areas where L4 excels in terms of performance, and the reasons for optimising certain parts of Darwin by taking advantage of L4. I will also detail the benchmarking methods used throughout this thesis.

5.1 Darwin's performance

One of the most notable features of Darwin is that it is based on the Mach kernel. Mach was originally designed as a microkernel, with a memory subsystem and an IPC primitive to provide all the abstraction required to support a guest operating system running in userland. In light of new research on operating systems, Mach's VM and IPC subsystems are very complex and inefficient. Mach's VM system is very complex, and its use is too pervasive throughout Darwin to consider modifications to it within the scope of this thesis. Mach IPC however is more suitable for modification as it is used in relatively isolated instances throughout Darwin.

Mach's IPC primitive is very complex and expensive. There are approximately 18,000 lines of code [App] devoted to processing Mach IPC requests in the kernel, 1200 of which make up the "HOTPATH" for the trap handler. Because it is capability based, checking must be performed in the kernel to authenticate each IPC operation, and in Mach this is an expensive operation. Being an asynchronous IPC primitive, if the receiver is not blocked waiting for a message, the message is stored in the kernel until is requested, which adds even more overhead and complexity.

Mach IPC allows data to be sent inline — the message contains the data, or out-of-line — the message contains a pointer and length that is understood by the kernel. Data can be copied up to 4 times between the sender making the trap and the receiver being able to access the data [GDFR90]. This is a gross inefficiency, especially when sending large amounts of data inline, which is possible in OS X. The Mach IPC abstraction in the CoreFoundation framework(a library in OS X that provides a common interface to much system functionality, as well as a collection of common data structures) will inline up to 10 pages of data before switching to use out-of-line transfers in Mach IPC.

The high cost of Mach IPC has been noted many times [Hel94, RMGB91, dPSR96, GDFR90], and in most attempts to port a guest OS to Mach, the authors have resorted to many techniques to avoid using Mach IPC. An example of this is implementing kernel functionality of the guest OS in userlevel libraries in order to not have to contact the kernel nearly as often [Hel94]. This type of optimisation is not limited to microkernel-based systems, it would bring performance benefits to any system — while at the same time introducing more complexity and security issues [dPSR96]) — but the fact that it has been used in many Mach projects shows just how prohibitively expensive Mach IPC is in practice.

The use of MIG RPC in Mach-based systems is also cause for concern. Traditionally, RPC is a synchronous operation, emulating the function-call semantics of a programming language, but in Mach this synchronous operation can only be implemented with asynchronous primitives, which adds extra overhead. The fact that so much of Mach's functionality is exported to userland via MIG RPC stubs means that there is a high cost for most interaction with the kernel.

5.2 L4's performance

In contrast to Darwin and Mach, L4's IPC primitive is one of the major reasons why L4 is considered so efficient [HHL$^+$]. L4 IPC is far from feature-compatible with Mach IPC, and it is fundamentally different in several ways, but due to its simplicity, it is possible to efficiently emulate Mach IPC with a layer on top of L4 IPC.

One of the critical differences of L4 is that it is synchronous — an attempt to send a message to another thread will block until the receiver performs a receive operation. This ensures that no messages are buffered in the kernel, and thus keeps the overhead of sending messages with L4 IPC relatively low.

Another difference in L4 IPC is that it is not capability based — the destination for an IPC message is specified only by the thread-ID for that thread. As a result, any thread may send an IPC message to any other thread in the system, and L4 does not have to perform any security checks to ensure that such an operation is valid. Security mechanisms can of course be built on top of L4 IPC and there is work on security mechanisms built into L4 itself [Embd].

One of the important features of L4 IPC is that it has a very tight restriction on what can be transferred in a message. Messages are limited to a total of 64 words, with the word size dependent on the underlying architecture. This restriction ensures that very little is copied on an IPC operation by the L4 kernel itself, and enforces the use of shared memory for copying large amounts of data.

5.3 Taking advantage of L4

Because of the overwhelming evidence in favour of L4's IPC primitive, the clear way to increase the performance of Darwin running on top of L4 is to take advantage of L4 IPC as much as possible. I have chose three distinct areas to examine the potential of L4 IPC in L4/Darwin, each for different reasons.

The cost of crossing traditional kernel/userland boundary is fundamental to overall system performance, thus exploring the use of L4 IPC as the means for crossing this boundary is an important step in achieving maximum performance for the whole system. I discuss the kernel/userland boundary and how it fits in with L4 in depth in Chapter 6.

Using L4 as a replacement for Mach IPC would give an insight into how much advantage can be gained by completely replacing all uses of Mach IPC with L4 IPC in L4/Darwin. I discuss this in the context of remote procedure calls and provide some insight into the where the costs of Mach IPC actually come from in Chapter 7.

Finally, as L4 allows new efficiencies such as IPC transfer between threads in the same address space without an address-space switch, it is necessary to explore methods to take advantage of this. In Chapter 8 I discuss the use of intra-address-space IPC as the foundation for thread synchronisation primitives.

5.4 Benchmarking

In order to examine the performance of various optimisations, it is necessary to have a precise benchmarking approach that can show with as much accuracy as possible the true cost of performing a given task. I have ensured that each benchmark is run multiple times and that the results from these runs are combined correctly to arrive at an accurate estimate of the true cost of the benchmark, this is explained below.

5.4.1 L4bench

For all benchmarks used throughout this thesis except those in Table 7.1, I have applied used L4bench [Embb] — an extensible benchmarking framework — to generate the numbers. L4bench combines multiple 'tests' and 'counters', and automatically runs large batches of benchmarks, outputting the results in the form of python commands for post-processing.

Tests represent the actual benchmark workload to be executed, and are specified by a set of functions for setup, teardown and execution of the benchmark, as well as a set of parameters. Each parameter is specified by a lower and upper bound, a step size and a step function. L4 bench then takes this parameter specification and executes the test with each possible value of the parameter. For example, if the lower bound is 10, the upper bound is 10,000, the step size is 10 and the step function is 10, then L4bench will execute the test with parameter values of 10, 100, 1000 and 10,000. This concise representation of parameters allows for a large multi-dimensional parameter space (i.e. multiple parameters) to be specified and then tested with the specified test function.

Counters represent the way to measure the numbers generated from running a benchmark. They are specified by start and stop functions, plus a function to return the value of the counter. I implemented two counters to use for all benchmarks: one to count cpu cycles and on one to use `gettimeofday` to show an output in microseconds.

5.4.2 Test environment

All benchmarks in this thesis were performed on a 1.83GHz Apple iMac with a dual-core Intel "Core Duo" processor. Because L4/Darwin only utilises a single core on the iMac, benchmarks performed in OS X were done so using the "cpus=1" kernel boot argument to limit the number of utilised cores to one.

5.4.3 Analysis

The python commands output by L4bench contain all the raw number values returned by the counters for each run. These are input into a python script that puts the values into a table and then uses linear regression to arrive at a unit-cost for each test. Currently linear regression is used when there is only one parameter, as there are many models that may fit a 2-variable equation. Despite this, however, all benchmarks used for this thesis were able to be constructed as 1-parameter tests. Using linear regression, running a test with multiple values for each parameter can produce an accurate estimate for the cost of a test with a parameter value of one.

Chapter 6

Optimised kernel entry

In Darwin, as in Windows, Linux, Solaris and many other operating systems, services are provided to userland applications by means of system calls. System calls allow application control flow to enter the kernel in order to accomplish certain tasks. These system calls provide an interface to file-system operations, network connections, the virtual memory subsystem and many other parts of the kernel. Because access to these services is a necessary part of normal program execution, the time taken to make system calls directly affects the running time of the program making those calls. Therefore, in striving for high overall system performance, it is important to minimise the cost (in terms of time) of making system calls.

6.1 Background

In order to understand the different techniques for optimising system calls on L4/Darwin, it is important first to understand how they work in general, how they are implemented in Darwin, and they can be implemented in an L4-based system.

6.1.1 User/Kernel system call boundary

To userland programs, system calls are just like any other standard library function, except that they perform some privileged task that can't simply be reimplemented in userland in the way a maths library could be. System calls are executed by the kernel in order to provide encapsulation of services, and since only the kernel can perform secure resource management and privileged operations such as interacting with hardware. Because of this elevated responsibility of the kernel, a method of transferring control from the userland to the kernel is needed. In addition to transferring control to the kernel, enough information must also be transferred to specify exactly what operation must be performed, for example, which file to write to, where to get the data from and how much to write.

Traditionally, system calls are implemented by small assembler stub functions that are the entry point for userland programs making the calls. These stub functions store the arguments and system call number either in registers or on the stack, and then trap into the kernel. There are three popular methods on the IA32 architecture to trap into the kernel: call gates, software interrupts and `sysenter`. Call gates are call instructions to a predefined address which, when executed, cause the CPU to switch to kernel mode before executing code at the address. Software interrupt are just like hardware interrupts, but generated by the `int` instruction. They handled by the kernels exception handler. A relatively new instruction, `sysenter` is specifically designed to bypass the interrupt descriptor table required to handle software interrupts, and provides a slightly faster means of invoking system calls.

Once control has been transferred to the kernel, the userland register state is usually inspected to extract the system call number and arguments. If the arguments are all in registers, they are just copied from the saved register file. If the arguments are on the stack however, they are retrieved through a copyin operation, using the userland stack pointer as the address to start copying from. With the system call number and arguments ready, the kernel can perform any checking, and invoke the actual system call implementation.

When the work of the system call is finished, either due to successful completion or an error, execution is returned to userland. This is achieved through modifying the userland register state to hold the return value or error code. Once control as returned to userland, it is the job of the assembler stub to decode any return or value, and take appropriate action. On UNIX systems, system call errors are stored in the `errno` variable, which is a special thread-local variable accessed through a C macro.

6.1.2 How Darwin implements system calls

The way Darwin implements system calls is relatively straightforward. Userland programs call an assembler stub function which stores the system call number in the `EAX` register and traps into the kernel. In early versions of Darwin on IA32, call gates were used but in the latest version, sysenter is the mechanism by which control is transferred to the kernel. Arguments are passed on the stack, since IA32 has very few registers, and because the C calling convention in Darwin passes function arguments on the stack, nothing needs to be copied by the stub to prepare arguments for the kernel.

The assembler stubs are implemented through C preprocessor macros that expand out to assembler that stores the system call number, generates the trap, etc. An example of such a stub is shown in Figure 6.1.

Once in the kernel, checks are performed to make sure the system call number is valid, a kernel funnel used to serialise access to the BSD part of the kernel is locked, and trace information is recorded. The arguments are then copied in to the kernel and the system call is made. The return value is stored in `EAX`, and `EDX` as well for 64-bit return values according to the IA32 C calling convention. If an error occurs at any point, the error code is stored in `EAX` and the carry bit in the status register is set. Once this is completed, control returns to the exception handler which switches back to userland.

The assembler stub in userland uses the `JNB` instruction to branch on the value of the carry bit. If it is set, code will be called to copy the value from `EAX` to `errno`, otherwise

```
#define UNIX_SYSCALL_TRAP   lcall   $0x2b, $0

#define UNIX_SYSCALL(name, nargs)        \
    .globl  cerror                      ;\
LEAF(_##name, 0)                        ;\
    movl    $ SYS_##name, %eax          ;\
    UNIX_SYSCALL_TRAP                   ;\
    jnb 2f                              ;\
    BRANCH_EXTERN(cerror)               ;\
2:

_read:
    UNIX_SYSCALL(SYS_read, 3)
    ret
```

Figure 6.1: A typical system call stub in Darwin

the function just returns as return value is right where it should be according to the C calling convention.

6.1.3 Using L4 IPC for system calls

As mentioned in Section 4.6, system calls are implemented slightly differently in an L4-based system. The kernel-side implementation is vastly different as it requires a synchronous event loop as opposed to the traditional asynchronous exception handler. The userland implementation requires fewer changes, however, and it is possible on some systems for userland stubs to work transparently. This technique is used to implement system calls in the Wombat project.

There are two facilities available in L4 to transfer control to a kernel server: L4 IPC and exception IPC. Both use L4 IPC to communicate with the kernel server, but the way they are invoked from userland is different.

L4 IPC is just the standard IPC interface discussed in Section 2.1. System calls over L4 IPC have userland stub functions that pack the system call number and arguments into an IPC message and call the kernel server. A reply IPC message contains the return value and error status.

Exception IPC works much the same way as a traditional kernel trap. Once the trap instruction is executed, an IPC message is sent to the thread's exception handler with the thread's register file contained within the message. From this the process of extracting arguments is the same, and modifying the register state on return is achieved though the IPC reply to the exception message.

System call	Darwin	L4/Darwin	Delta	Linux
mach_msg(0)	2523	3038	515 (20.4%)	-
pid_for_task	4328	4749	421 (9.7%)	-
flock(no args)	3678	4239	561 (15.3%)	674
mach_msg send/recv	5898	6582	684 (11.6%)	-
L4 IPC call	-	1830	-	-

Table 6.1: Cycle costs of various system calls on a 1.8GHz iMac Core Duo

6.2 Methods for optimisation

There are several techniques available for optimising system calls, some of which apply to all systems, and others which apply only to L4. I will discuss these in turn and then evaluate the methods suitable to L4.

6.2.1 Address-space layout

In Linux, like most other operating systems on 32bit IA32 machines, a "3:1 split" is used to save the cost of kernel entry. The kernel is mapped into 1GB of the 4GB userland address space, but only accessible when in kernel mode. This is in contrast to a "4:4 split" where both the kernel and userland applications each have their on 4GB address space. On a kernel call in with a 3:1 split, an address-space switch is not needed because the kernel region is already mapped into the current address space. A 3:1 split can bring major speed improvements as shown by the relatively low figure for the `flock` system call shown in Table 6.1. Given that the difference between the Linux and Darwin implementations is over 3000 cycles, and this is well above the cost of an L4 IPC call (which approximates the cost of two address-space switches), it is clear that there is no address-space switch overhead for Linux system calls. Since Darwin and OSX use a 4:4 split and assume a full 4GB userland address space on 32bit systems, this approach is not suitable as it would break binary compatibility.

6.2.2 L4 IPC

IPC is one of the most heavily-optimised facilities of L4 as it is also one of the most highly used. The cost of L4 IPC for x86 is shown in the last row of Table 6.1. This is significantly lower than the cost of normal system calls, but is still relatively expensive compared to L4 IPC on other modern architectures such as IA64, where the cost can be as low as 36 cycles [GCC$^+$05]. Optimising system calls for L4 IPC guarantees that as L4 improves, so will system call performance.

The first attempt at using L4 IPC to perform system calls involved rewriting the `UNIX_SYSCALL` macro to call a C function. This function used the L4 convenience interface to pack they system call number and arguments into an L4 IPC message to the L4/Darwin kernel server. While this C function proved to be very easy to maintain

System call	Darwin	L4/Darwin using L4 IPC
mach_msg(0)	2523	2393
pid_for_task	4328	3886
flock(no args)	3678	3509
mach_msg send/recv	5898	5761

Table 6.2: Cycle costs various system calls on a 1.83GHz iMac Core Duo

and extend, it significantly increased the time required to make system calls and was therefore not a viable option.

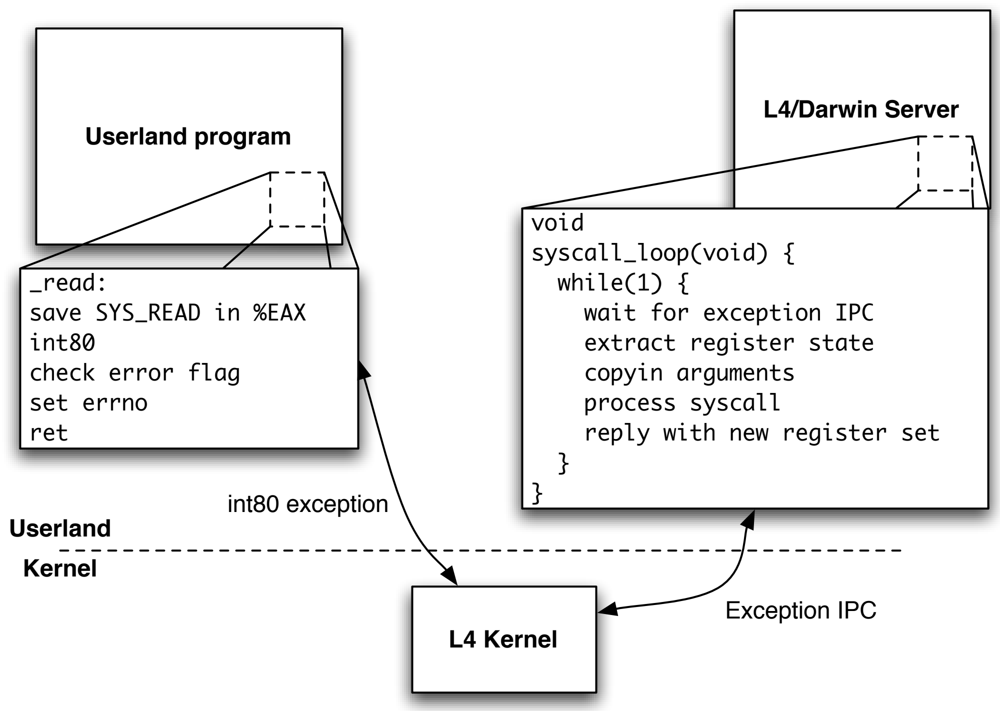
The second attempt was a function written entirely in assembler, manually filling in an L4 IPC message before invoking the L4 IPC trap into L4 itself. Each message contained the system call number and the stack pointer, allowing the L4/Darwin kernel server to "copyin" the system call arguments from the user program's stack. This method proved to be much more efficient that using the L4 convenience interface, but was still slower than native system calls on a pure Darwin system.

In order to further reduce the cost of system calls, I modified the assembler stub to take advantage of L4's message registers when transferring the system call arguments. The assembler function copies arguments off the stack into L4 message registers in the userland address space, which are then copied by L4 to the L4/Darwin kernel server. The number of arguments copied is determined at compile time by the second argument to the `UNIX_SYSCALL` macro. Having the arguments copied by L4 avoids having to make a relatively expensive "copyin" call inside L4/Darwin. The results of this optimisation are shown in Table 6.2.

6.2.3 Exception IPC

The way in which exception IPC would be used in L4/Darwin is shown in Figure 6.2. The process is summarised below:

1. Userland process saves system call number in a register and raises a software interrupt.
2. L4 handles this interrupt and generates an exception IPC to the L4/Darwin kernel server.
3. L4/Darwin kernel server extracts system call number and stack pointer from the register state in the exception IPC.
4. System call arguments are retrieved via copyin.
5. The requested operation is performed.
6. L4/Darwin kernel server replies to the exception IPC with the new register state, containing the return value.



```
_read:
save SYS_READ in %EAX
int80
check error flag
set errno
ret
```

```
void
syscall_loop(void) {
  while(1) {
    wait for exception IPC
    extract register state
    copyin arguments
    process syscall
    reply with new register set
  }
}
```

Figure 6.2: Syscalls with an exception handler

7. Userland stub sets `errno`

The options for speeding up exception IPC are relatively limited, as the interface for both the userland stubs and the kernel server handler are almost exactly the same as in native Darwin, and thus any possible optimisations would have hopefully been employed. The advantage of exception IPC however is that userland system can be supported without modification. As the process of building Darwin's standard library, libSystem, is quite complex, upgrades to newer source versions are much easier when system calls are delivered via exception IPC.

The fact that no available version of Darwin uses software interrupts for system calls proves difficult, as call gates and the `sysenter` trap are not reflected back up to userland by by L4. It is hoped that with future work on L4, the `sysenter` trap will be able to be virtualised, and thus system calls based on it will be able to be tested on L4/Darwin. The performance of system calls via exception IPC is not expected to be any greater than those using L4 IPC because the relatively expensive copyin operation must be performed on each system call.

Chapter 7

L4-IPC-based RPC

The use of Mach-IPC-based remote procedure calls is prevalent in Darwin. Approximately 90% [Won03] of the calls to Mach IPC in Darwin are MIG-based remote procedure calls to services in XNU. Much of the VM subsystem, for example, can only be controlled through MIG RPC. The usage patterns of different exported RPC calls is detailed in [Won03].

These functions are essentially system calls, but instead of being provided by the traditional trap mechanism, they are exported through MIG generated stubs. With MIG, the message passing overhead of Mach IPC still applies even if all the features are not required. Such tasks as checking capabilities to send to the globally accessible MIG kernel port must be performed on all RPCs, even though the same level of security can be achieved with a system call style interface. On a 1.8GHz iMac Core Duo, the MIG overhead was measured to be 3896 cycles as shown in Table 7.1. It is important to note that all measurements are taken in the L4/Darwin kernel server, at the points marked A, B, C and D in Figure 7.1, and that no address-space switch takes place that would possibly distort the measurements.

	Number of Cycles
from: start of mach_msg to: start of task_info	2655
from: start of task_info to: end of task_info	68
from: end of task_info to: end of mach_msg	1241
total time spent in MIG code	3896

Table 7.1: Cycle costs of MIG overhead on a task_info call on a 1.8GHz iMac Core Duo

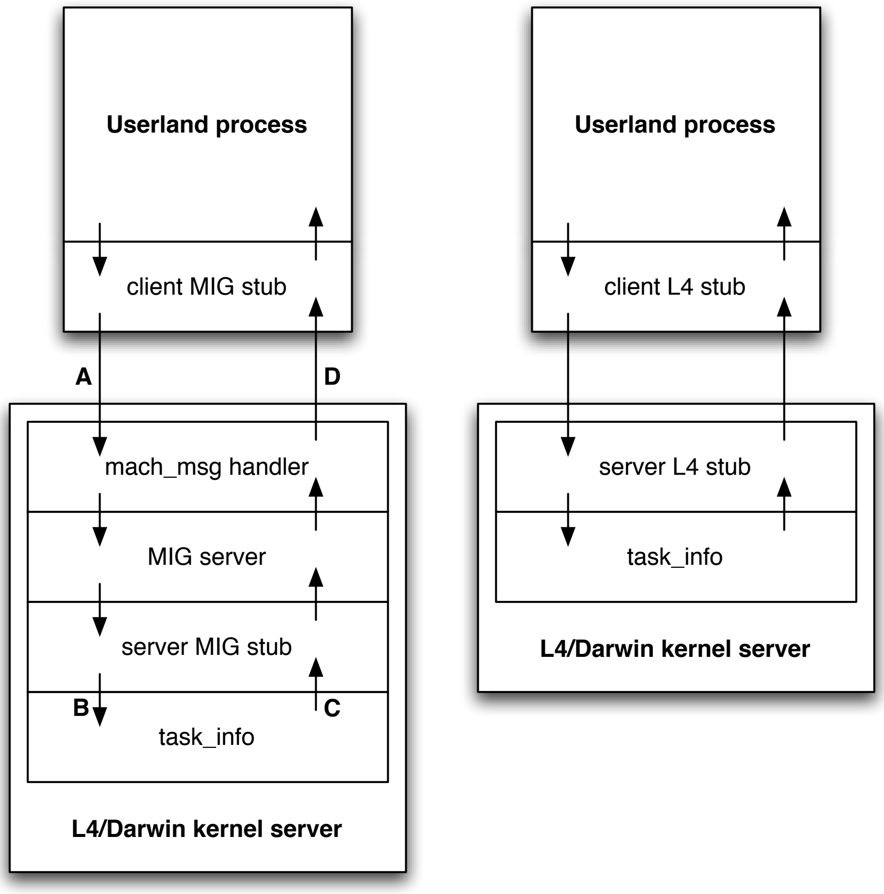


Figure 7.1: Call path of a function exported by MIG (left) and the optimised version (right)

7.1 Removing the MIG overhead

As an experiment to to see how much of the overhead on `task_info` could be removed, I traced through its execution to find the critical operations. It turned out the only operations necessary to the execution of `task_info` within the kernel were the conversion from a Mach port name to a `task_t` structure, and the copyout of the info structure filled in by `task_info` itself. These operations were easily replicable without the entirety of `mach_msg` being executed. The marshalling of data into and out of a Mach message structure and the lookup of the kernel MIG port are the two most costly operations which can be avoided. There is no reduction in security by shortcutting the kernel MIG port, as every task has a send right to this port.

I implemented a new userland stub ('client L4 stub' in Figure 7.1) shown in Appendix B to pack the arguments of `task_info` into an L4 IPC message and make the call. The server side implementation ('server L4 stub' in Figure 7.1) is shown in Appendix C. The code path to call `task_info` is shown in Figure 7.1, with the original implementation on the left and the optimised version on the right. The cost of the optimised implementation was measured to cost 3970 cycles. Compared to the original implementation costing 6023 cycles, this is a significant improvement. Note that both these were obtained using the benchmarking process described in Section 5.4.

The runtime saving of re-implementing MIG RPC in this way could have significant benefits for overall system performance, as a significant proportion of exported kernel functionality is available only through MIG. Particular services such as the Mach VM system and Mach port allocation and inspection are only available through MIG. It would be interesting to see the change in overall system performance if the overhead of MIG-based RPC was significantly reduced.

As further work I would like to develop an automated process to read MIG definition files and generate these optimised stubs. Work at ERTOS in the summer of 2005/2006 showed this to be quite difficult due to the type system supported by MIG and the way this is used throughout various RPC's. A partial implementation to solve the trivial cases may be an appropriate solution if overcoming the type system problems proves too hard. After completing this I plan to run system-wide benchmarks to evaluate the change overall system performance with these optimised RPC calls.

Chapter 8

Optimised POSIX threads

The majority of UNIX operating systems provide the pthreads (POSIX Threads) library. Pthreads provides a set of functions for creating threads within a process, manipulating them and synchronising them. It is available on Darwin/OSX, Solaris, Linux, FreeBSD, OpenBSD, NetBSD and even Windows [Mic], and thus is a convenient abstraction layer for writing portable multi-threaded applications.

One of the fundamental synchronisation primitives provided by pthreads is the mute: a basic lock used to achieve mutual exclusion. A mutex is locked by calling `pthread_mutex_lock`, and while a mutex is locked, any thread that tries to lock it will be suspended until the mutex is unlocked with `pthread_mutex_unlock`. Mutexes are used throughout Darwin to guarantee mutual exclusion. For example, a pthread mutex is attached to each `FILE` structure. This mutex is locked whenever an operation such as `fprintf` is performed on that file, and unlocked when the operation is complete. Mutexes are also heavily used in non-library code to control access to data structures that are shared between threads.

In Darwin, when there are more than two threads needing access to a mutex, the procedure followed to acquire the mutex becomes quite complex. The process in Figure 8.1 summarises the steps required to lock the mutex. Note that the semaphore operations in steps 2 and 4 are system calls, and thus require a costly address-space switch from the current thread to the kernel. The order semaphore is used to ensure that when multiple threads are contending for a mutex, it will be granted to threads in the order they request it. This avoids starvation in cases where one thread performs `pthread_mutex_lock` at the start of a loop and `pthread_mutex_unlock` at the end. Although the mutex is unlocked frequently, the time from the unlock at the end of the loop to the lock again at the start of the loop is very short. Without any semaphores the thread could unlock and then lock again without being preempted, thus any competing threads would never acquire the mutex.

The cost (in terms of time) of thread synchronisation is an important factor for multi-threaded applications. Shared resources act as bottlenecks preventing further execution, so the faster a thread can gain exclusive access to a resource, perform its operation and then release access, the faster the overall program will run. In the following sections I

1. Acquire the spinlock for the mutex
2. Wait on the ‘order’ semaphore for that mutex.
3. Release the spinlock for the mutex

    4a. Wait on the ‘waiters’ semaphore.

    4b. Signal the ‘order’ semaphore.

    Note that 4a and 4b are performed in one atomic call

4. Acquire the spinlock
5. Set the mutex owner to itself
6. Release the spinlock

   At this point, `pthread_mutex_lock` returns successfully

Figure 8.1: Pthread mutex lock protocol in Darwin

will discuss my solution to speeding up pthread synchronisation primitives.

8.1 Mapping multiple pthreads onto one Mach thread

In L4/Darwin, each Mach thread maps to exactly one L4 thread. Because the underlying Mach thread for each pthread is transparent for all UNIX applications and most Darwin/OSX-specific applications, the possibility exists to map each pthread to its own L4 thread without allocating a Mach thread. Doing this makes possible several performance optimisations, but also raises a number of issues which are discussed in this section.

There are extensions to pthreads in Darwin that allow conversion from a `pthread_t` reference to a Mach `thread_t` reference, and there is also the `mach_thread_self` system call which can be called by any thread. Therefore in order to provide complete compatibility with the existing APIs, these interfaces must present each pthread as being backed by its own Mach thread.

If multiple pthreads share a single Mach thread, the current implementation of the Mach scheduler requires a Mach thread to only ever be executing at most one system call at any one time. Without modifications to the scheduler, deadlocks may occur. For example, if two threads are using a UNIX socket to communicate, a blocking receive operation on the socket will prevent that Mach thread from executing another system call until the socket operation finishes. If the only threads sending to that socket are pthreads sharing the same Mach thread, the process will cease to make progress.

Another complication is how signals are delivered. The BSD implementation in Darwin allows signals to be sent to a particular Mach thread. There are also pthread

APIs to facilitate changing the signal mask on a per-thread basis. If multiple pthreads were multiplexed onto a single Mach thread, there would be ambiguity as to which signal mask was being modified on per-thread signal mask changes, and also ambiguity as to which thread to deliver a signal to.

From these issues it is clear that much work is involved in hosting multiple pthreads on top of a single Mach thread. A solution to all these problems is to still allow each pthread to have its own Mach thread, but allocate the Mach thread structures dynamically when they are needed, and relax the Mach scheduler to allow L4 to make more scheduling decisions. I have investigated doing this, but there is a significant amount of understanding of the Mach scheduler required to make this modifications. As a result, there is currently only one Mach thread allocated no matter how many pthreads are created. Further work is required to address the issues with the Mach scheduler described above, this is discussed in Section 11.2.

8.2 Fast mutex operations

Due to pthread mutex operations requiring at least 4 full address-space switches in a contentious environment, a performance boost can be achieved by eliminating the need for these switches. By completing all semaphore operations within the same address space, it would possible to achieve mutual exclusion for many threads sharing a resource without ever having to switch address spaces. This is achieved with L4 by running a special thread in each address space that handles all semaphore operations. What were system calls are now redirected to this semaphore server thread, and since it is running in the same address space, L4 can allow communication with pthreads and the semaphore server via L4 IPC without incurring a costly address-space switch.

I have implemented a semaphore library with the same semantics as Mach semaphores to explore the advantages of fast mutex operations. This semaphore library is currently only used by the pthread mutex lock/unlock operations, thus the only operations required are:

- semaphore_create
- semaphore_destroy
- semaphore_wait
- semaphore_signal
- semaphore_wait_signal (atomic wait and signal)

As mentioned before, the crucial difference of this semaphore library is that a dedicated 'semaphore server' thread runs in the same address space, and all operations are achieved through L4 IPC calls to this thread. This is shown in Figure 8.2. Blocking on a call to `semaphore_wait` is achieved through the semaphore server not replying to the IPC call until a corresponding `semaphore_signal` on the same semaphore is received.

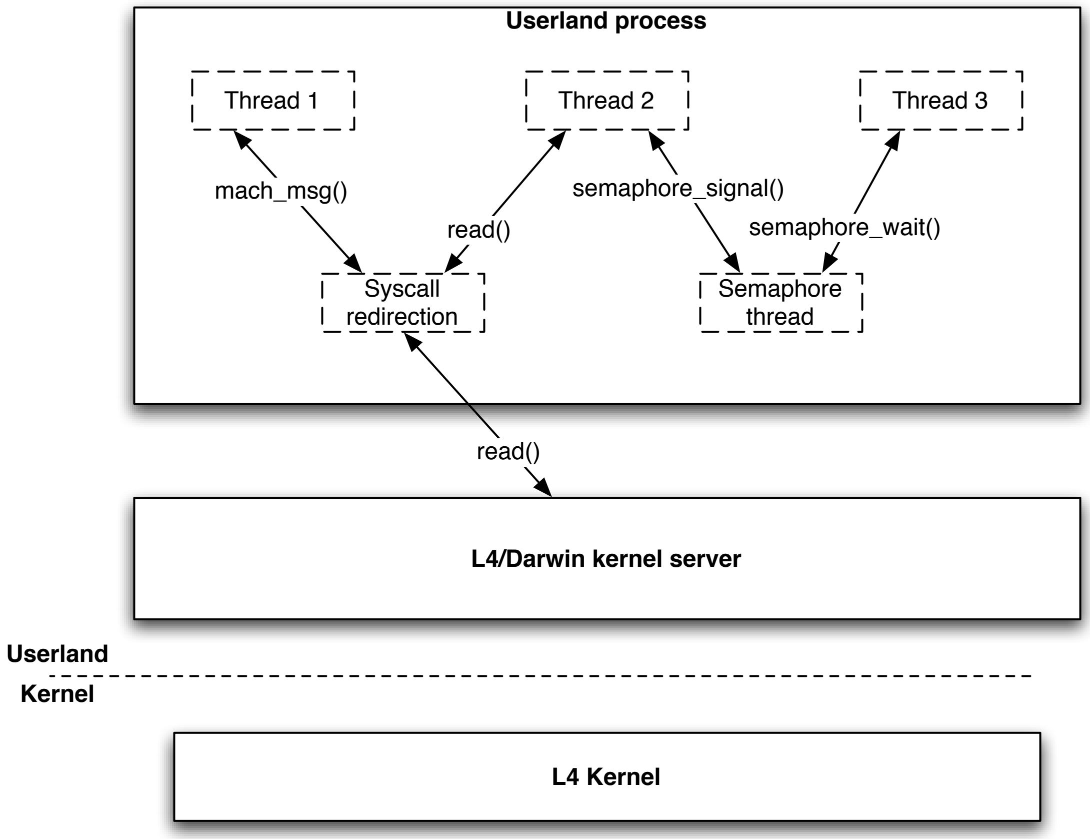


Figure 8.2: Syscall redirection and a semaphore server for lightweight pthreads

The advantage of this dedicated semaphore thread running in the same address space is that no address-space switch is required to perform semaphore operations, thus the four address-space switches required by a mutex lock operation can be reduced to zero.

8.3 Current implementation

To test the performance benefits of lightweight pthreads, I modified the pthread creation routine to bypass the Mach internals and create a new L4 thread without an associated Mach thread. While this shortcut does not allow many applications to run, it does give great insight into the possibilities of allowing L4 to make scheduling decisions.

For multiple pthreads to share one Mach thread, their system calls must be synchronised. I achieved this by redirecting all system calls through a dedicated system call thread within the same address space, as shown in Figure 8.2. Whenever a thread makes a system call, it uses L4 IPC to call the system call thread, which forwards the message to the L4/Darwin kernel server and forwards the reply back to the original calling

System call	L4/Darwin	L4/Darwin(redirection)	Overhead
mach_msg(0)	2393	4012	68%
pid_for_task	3886	5735	48%
flock(no args)	3509	5243	49%
mach_msg send/recv	5761	8020	39%

Table 8.1: Overhead for system call redirection

Thread count	**Darwin**	**L4/Darwin**	**Relative cost**
2	12089	6426	53%
3	13944	6433	46%
4	15878	6479	40%
5	16582	6522	39%
6	16957	6515	38%
7	17113	6544	38%
8	17362	6535	38%
9	17256	6531	38%

Table 8.2: Cycle costs of pthread synchronisation
All benchmark code was compiled with gcc -O2 and run on a 1.8GHz iMac Core Duo with one core disabled in both Darwin and L4/Darwin

thread. This ensures that only one pthread is making a system call at any time. The overhead of indirecting through this system call thread is shown in Table 8.1. While this overhead is significant, redirection is only a temporary measure and will not be required in any final design.

8.4 Evaluation

To evaluate this library, I wrote a benchmark to stress the `pthread_mutex_lock` and `pthread_mutex_unlock` operations. A specified number of threads are created, and each thread increments a shared counter until the counter reaches a predetermined maximum. The source code for this benchmark is shown in Appendix A. This benchmark uses only pthread library calls, so it can be compiled to run on native Darwin to achieve a fair comparison.

Results of this benchmark are shown in Table 8.2. From the table it is clear that reducing the number of address-space switches required to acquire a mutex greatly reduces the time to run this benchmark. It is also interesting to note that as the number of threads contending over the shared counter increases, the relative cost of running on L4 decreases. This suggests that the Mach scheduler is optimised for handling 2-3 threads contending over a critical chapter, but is not capable of handling a large number of threads concurrently.

While these results are substantial, there is still room for improvement. As discussed in Section 10.1, the version of L4 currently used by L4/Darwin lacks an implementation of the IPC fastpath for the IA32 architecture. It also lacks an implementation of Local IPC, which could bring a significant speedup to the large amount of IPC messages transferred between threads in the same address space on semaphore operations.

Chapter 9

A new L4 convenience interface

Throughout the course of my work I have noticed that speed improvements can be gained through not using the L4 convenience interface. Throughout this section I will discuss the reasons behind the poor performance of the convenience interface and propose a new interface.

While IDL compilers can provide the necessary optimisation for making userland IPC stubs fast, they require extra development time to utilise, as stubs must be written and the IDL compiler must be factored into the build process. The convenience interface should be just as its name implies — easy to pick up and use straight away, thus any performance increase that can be gained by modifying the convenience interface is worthwhile.

The typical flow of execution when using the L4 convenience interface is summarised in Figure 9.1. One of the important items to note is the copying of message data in steps 4 and 5 before the trap, and in steps 8 and 9 after the trap - all message data is copied twice. This may be unavoidable with a stack-based calling convention as in IA32, but with function inlining and a register-based calling convention in x86-64, it is possible to avoid this indirection of copying to a buffer first and just copy directly to the UTCB. There are other inefficiencies in using many functions to assemble the message. Even with function inlining, a compiler cannot do as good a job as hand-coded assembler optimisation.

A typical L4 IPC operation is shown in Appendix B. The lines of assembler generated for this function with various compile flags is shown in Table 9.1, along with the number of cycles required to execute.

There are several benefits of hand-coded stubs. The first is memory usage: the smaller the code size is, the more memory is free for other data. This effect is exaggerated when IPC calls are invoked from many different parts of the program. Since the L4 convenience interface relies on function inlining for its performance, code size quickly bloats. Optimised assembler stubs however, can be generalised to accept an arbitrary number of arguments. These generalised functions can then be stored in shared libraries and reduce the memory overhead of calling IPC in multiple places. Instead of bloating code with inlined functions for every IPC operation, user applications will only store the

1. A message structure is allocated
2. The message structure is cleared
3. The message label is set
4. Data is copied to the message from local variables
5. The message is copied to the UTCB
6. The L4 IPC syscall stub copies data from the UTCB to registers
7. The IPC trap is made
8. The L4 IPC syscall stub copies data from registers to the UTCB
9. The message is copied from the UTCB to a message structure
10. Data is extracted from the message structure.

Figure 9.1: Flow of execution when making an IPC call with the L4 convenience interface

stubs once in a shared library.

Another benefit is cache locality: as all 254 lines of assembler must be executed in the optimised case, each of these instructions will be cached. The much smaller 41-line stubs will have a smaller cache footprint, and will thus have much less chance of evicting other frequently used code from the cache.

The final benefit is of course raw speedup. In the last column of Table 9.1, the difference between the implementations become apparent — the convenience interface costs almost 4 times as much as optimised stubs. An 8% reduction in the total cost of IPC operations system-wide is not insignificant, and with an implementation of the L4 IPC fastpath and Local IPC(both discussed in Section 10.1), the relative cost of userland stubs will become greater. It is also important to note that the

GCC flags	**Lines of asm**	**Cycles to execute**	**% slower**	**Stub cost**
-DUSE_ASM_STUBS=1	41	2054	1.8%	-
-DUSE_ASM_STUBS=1 -O2	41	2017	0%	60
-DUSE_ASM_STUBS=0	427	2546	26%	-
-DUSE_ASM_STUBS=0 -O2	254	2181	8%	221

Table 9.1: Comparison of code generated by different compiler flags for Appendix B

Chapter 10

Discussion

10.1 Limitations of Pistachio N1

There are several limitations of the version of L4 used by L4/Darwin. Currently an older incomplete version implementing the NICTA N1 API is used. This version of L4 has been modified to support the L4/Darwin project. The two major changes are:

- Support for the Mach-O binary format, to simplify compilation and booting.
- EFI support to allow L4 to run on modern Intel-based Apple hardware.

The Darbat version of L4 however lacks two very important features: an implementation of the L4 IPC fastpath, and an implementation of the *Local IPC* system call.

The L4 IPC fastpath is a highly optimised piece of hand-code assembler that replaces the generic IPC implementation when certain conditions are met. These conditions include:

- the receiving thread is blocked waiting to receive
- the message only contains untyped words i.e. no map/grant items

Most of the time, these conditions are met for communication between userland processes and the L4/Darwin kernel server. A speedup is therefore expected once a version of L4 with the IPC fastpath is used for L4/Darwin.

Local IPC (LIPC) is similar in many ways to the fastpath, as it gives a speedup when certain conditions are met. LIPC is different however in that it optimises only for IPC transfers between threads in the same address space by eliminating the need for L4's intervention. From the L4 reference manual, LIPC can give a performance benefit when:

- a send phase is included
- a blocking receive phase is included
- the destination is a thread in the same address space

- the destination thread is running on the same processor
- the message contains only untyped words

A send phase simply means that a message is being sent from the thread calling LIPC. A blocking receive phase implies the thread wishes to receive a message, and will block if there is no thread waiting to send to it rather than returning an error.

LIPC would be of particular benefit to the dedicated semaphore server in Section 8.3, as all of these conditions, except the same processor condition, are guaranteed to be met. The speedup of performing semeaphore operations without even having to negotiate with L4 is expected to be significant.

10.2 The current status of L4/Darwin

Originally I had planned to evaluate my optimisations with large scale benchmarks such as LMbench [MS] and the AIM Multiuser Benchmark [Sak]. It became apparent that this was impossible due to the incomplete status of the L4/Darwin project. Many system calls do not function properly due missing kernel functionality, and several fundamental features such as signals, job control and preemption are only partially implemented. LMbench and AIM both rely heavily on signals and a wide variety of system calls to work correctly, and thus it is impossible to run them on L4/Darwin. The lack of support for paging out means that once physical memory is exhausted, nothing can be done to reclaim it. This is also compounded by the fact that the L4/Darwin kernel server does not have access to all physical memory, only a subset granted to it by L4. Paging is particularly important for large-scale benchmarks as they tend to stress the system by consuming large amounts of memory.

Chapter 11

Future work

11.1 Experimentation with the Pistachio N2 version of L4

The NICTA N2 API for L4 brings some significant changes. One particularly important change is the lack of recursive address spaces, which are described in Section 2, and the new MapControl API which replaces them. The MapControl API is different in two important ways:

The first is that address spaces are now populated by providing mappings from virtual addresses to physical addresses, rather than mappings from a virtual address in one space to a virtual address in another. This has two clear advantages: it is now much easier to support architectures where the physical address space is larger than the virtual address address space, and pagers such as the L4/Darwin kernel server are not required to have its virtual address space crowded by every page required by every userland process.

The second advantage of the MapControl API is that it is now possible to query L4's internal page tables to retrieve virtual to physical translations and protection bits. This makes redundant all of the page tables used by the `pmap` interface in Darwin. Since these page tables are no longer required, the space that would previously be allocated to them is now free for other uses. The `pmap` implementation would then be rewritten to perform insertion and lookup of virtual to physical translations using the MapControl API. On the x86 architecture with 32-bit virtual and physical address spaces, each processes page table can be anywhere from 4KiB to 4MiB in size. On a system with many processes, the removal of these page tables represents a significant saving. After L4/Darwin is ported to the latest version of L4 in the coming months, I plan to evaluate the performance increases gained from freeing up the memory used by the `pmap` layer.

11.2 Extensions to POSIX threads

Given the success of my current optimisations for the pthreads library, I plan to extend this work in two ways: modifying the Mach scheduler to give L4 more control over scheduling decisions as described in Section 8.1, and transferring all synchronisation primitives provided by pthreads to use my semaphore library from Section 8.3.

Modifying the Mach scheduler is a requirement for providing API binary compatibility to existing applications that use pthreads. The current system call redirection system described in Section 8.3 is a temporary measure and will lead to deadlocks that cannot be resolved in userland programs. This Mach scheduler currently has a rigid policy where the *current thread* is only changed when a thread is preempted and a new one is scheduled. The *current thread* is the only thread allowed to make system calls at any time. I would like to relax the concept of *current thread* in Mach, allowing userland threads to make system calls at any point in time, and having the value of *current thread* modified when system calls do occur. Relaxing the Mach scheduler would allow the more agile L4 scheduler to schedule threads from an available pool of runnable threads supplied by the Mach scheduler, without having to consult the Mach scheduler on every timer tick, increasing the overall performance of the system.

Modifying the pthread library to take full advantage of the fast semaphore library will also bring significant performance improvements. Currently only trivial semaphore operations are supported as these are all that is required for pthread mutex operations. There are another set of semaphore operations that allow timeouts - these are used by the pthread condition-variable primitives. Adding timeouts to the semaphore server would add complexity, but would also complete the implementation of the semaphore library, making it fully compatible with Mach semaphores.

With a fully implemented semaphore library, it would be possible to replace the existing implementation of the semaphore operations in libc with my library. This would allow mutexes for libc resources as well as user-created mutexes to take advantage of the speedup possible with a local semaphore server. Resources in libc that use mutexes include `FILE` structures - a data structure used to store metadata about an open file, and used as a file reference for operations such as `fprintf`. There is one design issue with this however - mutexes can be placed in shared memory regions and used to synchronise threads in two different address spaces. Since all operations on a semaphore must be negotiated with the semaphore server thread that was used to create it, a means of contacting the semaphore server thread for a given semaphore is needed when it is used outside the address space within which it was created. This can possibly solved by encoding the L4 thread id of the semaphore server and the semaphore id into a single 32bit value to conform to the current `semaphore_t` representation.

11.3 Large-scale benchmarks

In Section 10.2 I mentioned that it was currently impossible to run large scale benchmarks on L4/Darwin. As the project becomes more stable, I plan to run some large scale benchmarks to gain better insight into the affect of my optimisations on overall system performance. Two benchmarks I plan to use are LMbench and AIM

LMbench [MS] is a suite of UNIX benchmarks for measuring system performance of various tasks such as file system operations, context switching, process creation, and system calls. I plan to use LMBench extensively to measure the cost of these various operations on L4/Darwin with and without various optimisations, and on native Darwin to serve as a baseline.

The AIM Multiuser Benchmark [Sak] is a system-wide benchmark designed to measure the performance of multi-user systems. It runs many tasks that are common on a system with many users, and puts the system under quite a significant load I plan to use this to understand how my optimisations affect the overall performance of the system.

11.4 Automatic generation of MIG stubs

In Section 7.1 I showed that it was possible to reduced the cost of RPC by approximately 2000 cycles when not using MIG. The engineering effort in manually converting all existing MIG stubs to L4-based stubs would be significant, and this manual conversion would have to be applied with each new version of XNU. A better solution would be to automate the process of generating stubs.

There already exist two *interface definition language* (IDL) compilers for L4: IDL4 [Sys] and Magpie [Embc]. Neither of these support the archaic MIG syntax. IDL4 supports CORBA [Obj] and DCE IDL [Theb] definition languages, thus in order for to work correctly, a layer would have to be written to translate MIG definition files into either the CORBA or DCE IDL languages. Magpie however has an extensible front and back end, allowing a module to be written that allows Magpie to read MIG definition files and output optimised stubs for L4.

I plan to investigate the writing such an extension to Magpie, although there are several design issues to overcome. One major problem is MIG's heavy reliance on the Mach IPC primitive - all complex types such as tasks, threads and memory regions are represented in MIG with Mach ports. In addition, the `mach_msg` primitive used by MIG to invoke RPCs performs special error-handling functionality when, for example, an operation on a task port is performed when that task no longer exists. In order for Magpie to correctly handle MIG definition files, all of these complexities would need to be handled in a transparent way.

11.5 Up-to-date comparisons

While best efforts were made to compare L4/Darwin to an equivalent version of Darwin, this has been impossible to do. When L4/Darwin was developed, the Darwin source available was far behind any current binary release from Apple. Further source releases have been made however, and thus it is now possible to obtain the source code for the version of Darwin that corresponds to the most recent release of Mac OS X. Unfortunately, significant effort is required to modify L4/Darwin to update to this new source code.

Given this difference in source code versions, some benchmark numbers may not accurately reflect how L4/Darwin compares to native Darwin. Although the numbers in Table 6.2 show L4/Darwin to be faster, they may be misleading due to the differences in source code. An accurate comparison would involve updating L4/Darwin to use the latest sources, though these numbers still show that L4/Darwin is competitive.

Chapter 12

Conclusions

Several modifications have been made to L4/Darwin that increase its performance and show where more performance can be gained. All of these modifications leveraged the highly optimised L4 IPC primitive.

System calls over L4 IPC were shown to be more efficient than the native implementation in Darwin. This was due primarily to the speed of L4 IPC and the use of L4 message registers to transfer system call arguments, avoiding a copyin operation.

POSIX thread synchronisation was shown to take as little 38% of the time when the semaphore operations were modified take advantage of L4 IPC, compared to the native implementation. While the current implementation of optimised pthreads is not suitable for deployment, it does show the magnitude of the speedup that can be achieved by modifying all of the pthread synchronisation primitives to take advantage of L4 IPC.

Modifying a MIG-based RPC stub to use L4 IPC has shown that a significant speedup can be achieved by not using the Mach IPC primitive to transfer arguments and control information. This result can be extrapolated to assume that the majority of MIG-based RPC calls in Darwin will gain a similar speedup from using L4 IPC. The L4 convenience interface was shown to be lacking in terms of raw speed and a large cache footprint. An alternative, hand-written stubs for all possible numbers of arguments, was shown to be advantageous both in terms of raw speed and a small cache footprint, and the fact that it can be stored in a library to avoid bloating code size with repeated use of L4 IPC.

Appendix A

Benchmark used for pthreads

```
void *thread_func(void *);
pthread_mutex_t mtx;
pthread_mutex_t *sharing;

int number_max = 0;
int number = 0;
int nthreads;

int can_die[9];

void
pthread_teardown(struct bench_test *test, int args[])
{
    number_max = number;
    number++;
    int i;
    for(i=0;i<nthreads-1;i++)
    {
        can_die[i+1] = 1;
    }

    pthread_mutex_unlock(&mtx);
    pthread_mutex_unlock(&sharing[1]);
    pthread_mutex_lock(&sharing[0]);
    free(sharing);
}

void
pthread_tester(struct bench_test *test, int args[])
{
    number = 0;
    thread_func(NULL);
}

void
pthread_init(struct bench_test *test, int args[])
{
    nthreads = args[0];
    number_max = args[1];
    sharing = malloc((nthreads)*sizeof(pthread_mutex_t));
    pthread_mutex_init(&mtx, NULL);

    number = 0;

    int i;
    for(i=0;i<nthreads-1;i++)
    {
        pthread_t thread;
        pthread_mutex_init(&sharing[i+1], NULL);
        pthread_mutex_lock(&sharing[i+1]);
        pthread_create(&thread, NULL, thread_func, (void *)(i+1));
    }

    pthread_mutex_init(&sharing[0], NULL);
}

void *
thread_func(void *arg)
{
    while(number <= number_max || can_die[(int)arg])
    {
        pthread_mutex_lock(&sharing[(int)arg]);
        pthread_mutex_lock(&mtx);
        if(arg == (void *)0x0 &&
                number >= number_max - ((number_max-1) % (nthreads)+1))
        {
            return NULL;
        }
        number++;
        can_die[(int)arg] = 0;
        pthread_mutex_unlock(&mtx);
        pthread_mutex_unlock(&sharing[((int)arg + 1 )% (nthreads)]);
    }
    return NULL;
}
```

Appendix B

Userland stub for optimised task_info

```
kern_return_t task_info(task_t task, int flavour, integer_t *out,
    mach_msg_type_number_t *count)
{
    kern_return_t retval;
#if USE_ASM_STUBS
    __asm__ __volatile__(
            "pushl %%ebx;\n"
            "pushl %%ebp;\n"
            "pushl %%esi;\n"
            "pushl %%edi;\n"
            "movl  %%gs:0, %%ecx;\n"
            // Load the args
            "movl  (0x0)(%%eax), %%esi;\n"
            "movl  %%esi,        (0x04)(%%ecx);\n"
            "movl  (0x4)(%%eax), %%esi;\n"
            "movl  %%esi,        (0x08)(%%ecx);\n"
            "movl  (0x8)(%%eax), %%esi;\n"
            "movl  %%esi,        (0x0c)(%%ecx);\n"
            "movl  (0xc)(%%eax), %%esi;\n"
            "movl  (0)(%%esi),   %%esi;\n"
            "movl  %%esi,        (0x10)(%%ecx);\n"
            "movl  (-40)(%%ecx), %%eax;\n"
            "movl  %%eax, %%edx;\n"
            "movl  $0x0547C004, %%esi;\n"
            "movl  %%esi, (0)(%%ecx);\n"
            "movl  (4)(%%ecx), %%edi;\n"
            "movl  (8)(%%ecx), %%ebp;\n"
            "call  1f;\n"
            "jmp   2f;\n"
            "1:\n"
            "popl  %%ebx;\n"
            "int   $0x30;\n"
            "2:\n"
            "movl  %%ebp, %%eax\n"
            "movl  %%edi, %%edx\n"
            "popl  %%edi;\n"
            "popl  %%esi;\n"
            "popl  %%ebp;\n"
            "popl  %%ebx;\n"
            :
            "=d" (retval), "=a"(*count)
            :
                "a" (&task)
                 );

    return retval;
#else
    L4_Msg_t msg;
    L4_MsgTag_t tag;

    L4_MsgClear(&msg);
    L4_Set_MsgLabel(&msg, 0x547);
    L4_MsgAppendWord(&msg, task);
    L4_MsgAppendWord(&msg, flavour);
    L4_MsgAppendWord(&msg, (L4_Word_t)out);
    L4_MsgAppendWord(&msg, (L4_Word_t)count);
    L4_MsgLoad(&msg);
    tag = L4_Call(L4_Pager());
    L4_MsgStore(tag, &msg);
    *count = L4_MsgWord(&msg, 1);

    return L4_MsgWord(&msg, 0);
#endif
}
```

Appendix C

Server-side implementation of optimised task_info

```
int
handle_task_info(L4_ThreadId_t tid, L4_MsgTag_t tag, L4_Msg_t *msg)
{
    L4_Word_t task_name, output_ptr;
    task_name = L4_MsgWord(msg, 0);
    task_flavor_t flavour = (task_flavor_t)L4_MsgWord(msg, 1);
    output_ptr = L4_MsgWord(msg, 2);
    integer_t count = (integer_t)L4_MsgWord(msg, 3);
    integer_t old_count = count;

    integer_t *array = kalloc(count*sizeof(*array));
    assert(array);

    ipc_space_t space = darbat_current_space();
    darbat_space_lock(space);
    struct ipc_entry *entry =
        ipc_entry_lookup(space, (mach_port_name_t)task_name);
    darbat_space_unlock(space);
    ipc_port_t task_port = (ipc_port_t)(*((void **)entry));
    task_t task = convert_port_to_task(task_port);

    kern_return_t retval = task_info(task, flavour, array, &count);

    if(retval == KERN_SUCCESS)
    {
        copyout(array, (user_addr_t)output_ptr, count*sizeof(*array));
    }
    kfree(array, old_count*sizeof(*array));
    L4_MsgClear(msg);
    L4_MsgAppendWord(msg, (L4_Word_t)retval);
    L4_MsgAppendWord(msg, (L4_Word_t)count);
    L4_MsgLoad(msg);
    return 1;
}
```

Bibliography

[App] Apple Computer Inc. Darwin source code. http://www.opensource.apple.com/darwinsource/.

[App06a] Apple Computer Inc. I/O Kit Fundamentals. http://developer.apple.com/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/, 2006.

[App06b] Apple Computer Inc. Kernel Programming Guide. http://developer.apple.com/documentation/Darwin/Conceptual/KernelProgramming/, 2006.

[App06c] Apple Computer Inc. Porting Unix. http://developer.apple.com/documentation/Porting/Conceptual/PortingUnix, 2006.

[dPSR96] Francois Barbou des Places, Nick Stephen, and Franklin D. Reynolds. Linux on the OSF Mach3 microkernel. First Conference on Freely Redistributable Software, Cambridge, MA, 1996.

[Emba] Embedded, Real-Time and Operating Systems Program, NICTA. Darbat. http://www.ertos.nicta.com.au/software/darbat/.

[Embb] Embedded, Real-Time and Operating Systems Program, NICTA. L4bench. http://www.ertos.nicta.com.au/software/kenge/l4bench/devel/.

[Embc] Embedded, Real-Time and Operating Systems Program, NICTA. Magpie. http://www.ertos.nicta.com.au/software/kenge/magpie/latest.

[Embd] Embedded, Real-Time and Operating Systems Program, NICTA. seL4 - Secure Microkernel Project. http://www.ertos.nicta.com.au/research/sel4/.

[Embe] Embedded, Real-Time and Operating Systems Program, NICTA. The L4 Microkernel. http://www.ertos.nicta.com.au/research/l4/.

[Embf] Embedded, Real-Time and Operating Systems Program, NICTA. Wombat. http://www.ertos.nicta.com.au/software/kenge/wombat/latest/.

[GCC$^+$05] C. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser. Itanium — a system implementor's tale. In Proc, 2005.

[GDFR90] David B. Golub, Randall W. Dean, Alessandro Forin, and Richard F. Rashid. UNIX as an Application Program. In *USENIX Summer*, pages 87–95, 1990.

[GJP$^+$00] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The sawmill multiserver approach. In SIGOPS European Workshop. ACM, September 2000., 2000.

[Hel94] J. Helander. Unix under Mach: The Lites Server. Helsinki University of Technology, Master's Thesis, 1994.

[HHL$^+$] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The Performance of μ-Kernel-Based Systems.

[IBM04] IBM Inc. Yellow Dog Linux on Power Mac G5. http://www-128.ibm.com/developerworks/library/l-ydlg5.html, 2004.

[Ins05] Institute of Electrical and Electronics Engineers. POSIX. http://standards.ieee.org/regauth/posix/, 2005.

[Lee05] Geoffrey Lee. I/O kit drivers for L4. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2005.

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

[McC] Thomas McCarthy. Intro to NEXTSTEP. http://www120.pair.com/mccarthy/nextstep/intro.htmld/.

[Mic] Microsoft Corpo-
ration. Pthread Support in Microsoft Windows Services for UNIX Version 3.5. http://www.microsoft.com/technet/interopmigration/unix/sfu/pthreads0.mspx.

[MS] Larry McVoy and Carl Staelin. LMbench. http://www.bitmover.com/lmbench/.

[Nou05] Abi Nourai. A physically-addressed L4 kernel. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, March 2005.

[Obj] Object Management Group. CORBA^{TM}/IIOP^{TM} Specification. http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[OS] OS Data. Digital UNIX/Tru64. http://www.osdata.com/oses/decunix.htm.

[OSG] TU-Dresden Operating Systems Group, Dept of Computer Science. L4Linux. http://os.inf.tu-dresden.de/L4/LinuxOnL4/.

[Por] Darwin Ports. http://darwinports.opendarwin.org/.

[RMGB91] Richard F. Rashid, Gerald R. Malan, David B. Golub, and Robert V. Baron. DOS as a Mach 3.0 Application. In *USENIX MACH Symposium*, pages 27–40. USENIX, 1991.

[RTY$^+$88] Richard Rashid, Avadis Tevanian, Jr., Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, C-37:896–908, 1988.

[Sak] Jonathan Saks. The AIM Multiuser Benchmark. http://sourceforge.net/projects/aimbench.

[Sys] System Architecture Group, University of Karlsruhe. IDL4 Compiler. http://l4ka.org/projects/idl4/.

[Thea] The Fink Project. Fink. http://fink.sourceforge.net/.

[Theb] The Open Group. OpenDCE. http://www.opengroup.org/dce/.

[Won03] Ka-shu Wong. MacOS X on L4. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 2003.