The University of New South Wales

School of Computer Science and Engineering

Mac OS X on L4

Ka-shu Wong

Bachelor of Engineering (Software Engineering)

December 2, 2003

Supervisor: Prof. Gernot Heiser

Abstract

Mac OS X is an operating system with a kernel based on Mach, a first-generation microkernel with a reputation for being slow and lacking in flexibility. L4 is a second-generation microkernel that overcomes these limitations. This thesis examines the internal structure of the Mac OS X kernel and its reliance on Mach, and presents the design of L4-Darwin, a port of the Mac OS X kernel to L4 without Mach. There has been some progress in the implementation of L4-Darwin, and the implementation and preliminary benchmark results are discussed.

Contents

Chapter 1

Introduction

Mac OS X is Apple's newest operating system, consisting of a number of proprietary components layered over an open-source core, called Darwin, which forms a complete Unix operating system in itself. Both Mac OS X and Darwin are based on the same kernel, XNU, which is a combination of BSD and Mach.

Microkernel-based systems, such as systems based on Mach, have a reputation for exhibiting poor performance. The basic idea of the microkernel approach is to implement as much functionality as possible outside the kernel. It was believed that poor performance was inherent in this approach, and therefore a strategy used in many systems was to move such functionality back into the kernel. This was seen to have improved performance compared to a pure microkernel-based system. In creating XNU, Apple has taken this to the extreme, merging BSD with Mach and forming what is essentially a monolithic kernel, but with some Mach functionality built in.

However, Liedtke [11] and others have argued that the reason for microkernel performance problems was that existing microkernels were too large; by rigorously aiming at minimality, microkernels such as L4 [12] have been able to achieve much greater performance, particularly in the area of interprocess communication (IPC).

Given that the performance problems with Mach is because of its size, it seems that incorporating Mach into a monolithic BSD kernel would only decrease its performance compared to a plain BSD kernel. This raises the question of what features of Mach is used by Darwin and Mac OS X, and whether its performance can be matched or exceeded by using a pure microkernel system on a faster and smaller microkernel such as L4. In particular, if user-level programs in Darwin and Mac OS X make extensive use of Mach IPC, then it is possible that by replacing this with L4 IPC significant performance benefits can be achieved.

The goal of this work is to analyse the structure of XNU to see how Mach is used, and to produce a port of XNU to L4, which we call L4-Darwin, to determine whether:

1. Similar performance can be achieved despite the additional overhead imposed by running in user-space on top of the L4 microkernel, and
2. Performance benefits can result from removing Mach and using faster L4 abstractions such as L4 IPC.

A key feature of microkernels such as L4 is their flexibility – the ability to implement a wide variety of systems based on the microkernel. A secondary goal of this work is to demonstrate the flexibility of L4. To satisfy this goal, there are to be no modifications to L4 itself.

Since source is not available for many Mac OS X programs, it is necessary for L4-Darwin to have binary compatibility with XNU. Also, since Mac OS X is only available for PowerPC, the initial implementation of L4-Darwin will be for the PowerPC architecture.

The issue of L4-Darwin device drivers will only be dealt with briefly in this thesis, as there is another thesis project currently in progress which is investigating this in more detail.

1.1 Outline

- Section 2 explores general issues relating to the porting of monolithic Unix systems to microkernels.
- Section 3 analyses XNU and its use of Mach features, to investigate what needs to be done for L4-Darwin.
- Section 4 describes a design for L4-Darwin.
- Section 5 gives implementation details of L4-Darwin.
- Section 6 presents what has been achieved with L4-Darwin, and analyses the performance results that have been achieved.

1.2 Related work

There have been many attempts at creating and porting Unix and Unix-like systems to run on microkernels. This is covered in section 2.

1.2.1 NetBSD Darwin binary compatibility

The latest versions of NetBSD include binary compatibility support for Darwin binaries on the PowerPC architecture [4]. Currently, it is able to run most Darwin command line programs. The following features have been implemented:

- Mach-O loading, including support of shared libraries.
- Mach system call handling.
- Mach IPC emulation, including transfer of port rights.
- Mach thread system calls.
- Minimal IOKit emulation.
- Signal handling.

The achievements of NetBSD Darwin compatibility demonstrates that it is possible to emulate the Mach functionality required by Darwin and Mac OS X programs without incorporating Mach itself.

Chapter 2

Microkernel-based Systems

2.1 The microkernel concept

The microkernel, as an operating systems design concept, was first conceived as a way to deal with the increasing complexity of operating system kernels [14]. The idea is to move much of the functionality of a traditional monolithic kernel, such as filesystems and network stacks, into user-level servers which run alongside application programs. The kernel itself is then reduced to providing the minimum functionality required in order to support these servers. The advantages of this approach include:

- An error in a server only affects the server itself and can be isolated; the server can then be restarted without requiring a restart of the entire system.
- A smaller kernel is easier to maintain and is less likely to contain errors.
- A variety of different operating system environments, including differing application programming interfaces (APIs), even differing operating system design concepts, can be implemented as servers which co-exist and run on top of the same microkernel.
- A clean, well-designed interface enforces modularity of servers, which results in a modular operating system design that is both easier to maintain and to extend.
- The parts of the system that must be trusted, the trusted computing base, is reduced to the hardware, the microkernel, and some basic device drivers.

A typical microkernel-based system is structured as a number of servers on top of a microkernel. The microkernel typically provides support for threads, address spaces and inter-process communication. Traditional operating system functionality is implemented as a server or a number of servers running on top of the microkernel, and user programs make system calls through the inter-process communication mechanism provided by the microkernel.

2.2 Mach

Mach [14] is a microkernel originally developed at Carnegie-Mellon university, It is perhaps the most well-known and widely used microkernel, having been used in a number of research and commercial

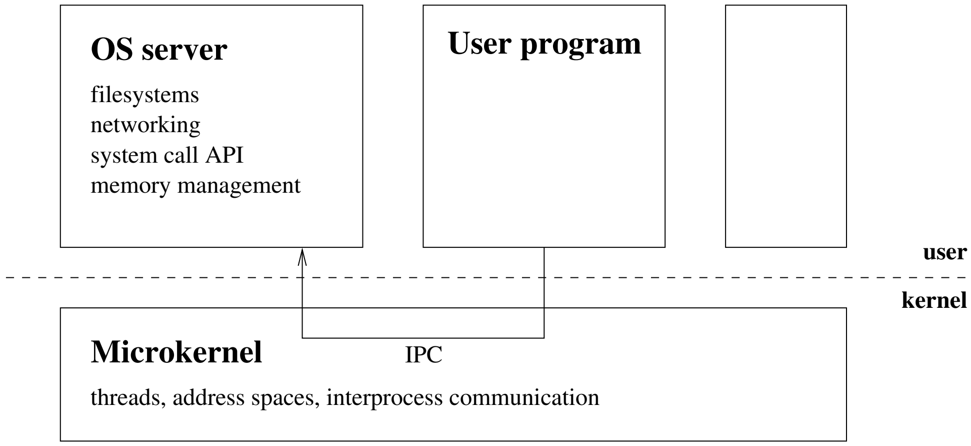

Figure 2.1: Structure of a typical microkernel-based system

systems. Mach was designed to provide a small set of basic functionality which would allow a variety of more complex systems to be constructed on top. The basic functionality that Mach provides are:

- Tasks with separate address spaces.
- Multiple threads running within a task.
- Virtual memory management system, including support of external pagers.
- Capability-based message passing interprocess communication (IPC), including copy-on-write transfer of memory buffers.

Some of the more notable features of Mach are:

External pagers An external pager allows page fault handling to be moved to a user-level task, while the microkernel continues to manage physical memory and virtual memory mappings. An external pager handles interactions with the backing store of a virtual memory region by loading pages into memory in response to page faults, and writing back pages as needed. The microkernel handles the allocation of physical memory as well as the virtual-to-physical mappings. The external pager technique allows memory-mapped files without integrating filesystem functionality into the kernel. Distributed memory can also be implemented using this model.

Capability-based IPC Mach uses a message passing interprocess communications (IPC) system, with *ports* as the basic addressing unit. Ports are used to represent services and objects, and provides a layer of indirection between clients and servers. Accesses to ports are governed by *port rights*, which gives tasks permission to either send or receive on a port. Port rights are granted via special messages containing a port right. Port rights can be thought of as capabilities to objects represented by the port, and are used as the main access protection mechanism on Mach-based systems.

IPC/MM integration For efficient transfer of large amounts of data in a message passing system, the message passing facility is integrated with virtual memory management, such that the data is transferred to the receiving task via memory mapping rather than by copying the data. This allows the transfer of large amounts of data at very low cost, and is key to the efficiency of servers such as file system servers.

Portability Mach is designed to be portable to a variety of architectures, and has a well-defined interface between its machine-dependent and machine-independent components.

Transparent shared libraries A transparent shared library is a library that can be loaded into the address space of a task without its knowledge. Mach can be told to redirect system call traps made by that task into its shared library, which can then either emulate the system calls directly or convert it into a message to a system server. This is useful for implementing binary compatibility with non-Mach operating system environments.

2.2.1 History of Mach and Mach-based systems

Mach was originally developed at Carnegie-Mellon University (CMU) in the 1980s. Its main purpose was to support operating systems research, functioning as a base on which various operating system environments can be constructed. Mach 3.0 is the first version of Mach that is a true microkernel; previous versions of Mach contained 4.3BSD functionality and are not pure microkernels.

Mach-UX: 4.3BSD-based single server

Mach-UX [7] was the first implementation of Unix on Mach 3.0, developed at CMU. It consists of two main parts: a Unix server and a transparent system call emulation library. The Unix server is implemented as a single multithreaded Mach task. Threading and synchronisation are handled using the Mach C threads threading library.

A pool of threads within the server are allocated to handle incoming system calls. Requests from user programs to the Unix server are sent through Mach IPC. For each incoming message, a thread from the pool is dispatched to handle the requested operation. Most request messages to the Unix server corresponds to a 4.3BSD system call.

To allow 4.3BSD applications to run unmodified on Mach-UX, the transparent system call emulation library is used. This library is loaded into the address space of the first user process and inherited by subsequent user processes. It makes use of the "trampoline" mechanism provided by Mach, which operates by transferring control to the emulation library within the calling task whenever a designated set of system traps are issued. The emulation library then emulates the system call, in most cases translating it into a remote procedure call to the Unix server.

Mach-UX supports two different modes of file access. File access can be performed through a message passing interface. However, file access can also be provided through memory objects, using an inode pager implemented in the Unix server. If the memory object access method is used, the emulation library maps the file into its address space. The emulation library then performs file operations such as read and write by accessing the address space region where the file is mapped, without sending requests to the Unix server. It was found that memory-mapped file access provides a significant performance improvement over the use of message passing.

Mach-US: Multi-server OS on Mach

Mach-US [16] is a multi-server operating system on Mach 3.0. It consists of a set of separate servers supplying OS services, and an emulation library loaded into each user process. Mach-US is a redesign of the entire OS around the microkernel, and because of this, its internal design bears no relation to the structure and design of BSD Unix.

The interface to the servers are API-neutral, and are designed with the goals of supporting multiple OS APIs, reducing the amount of inter-server interaction, and maximising the amount of processing performed in the emulation library. The interface is defined in an object-oriented manner. An *OS-item* represents an OS object, such as a file or a process, and has a well-defined set of operations. A server in Mach-US normally implements a number of OS-items.

Emulation libraries are used to implement OS APIs. An emulation library is loaded into a user process, and system call traps are redirected to the emulation library using the Mach system call "trampoline" mechanism. The emulation library supplied in Mach-US emulates the 4.3BSD system call interface. These system calls are implemented in the emulation library itself, using the API-neutral OS interfaces offered by the servers.

A surprising finding of the Mach-US development effort was that there is almost no need for inter-server communication in a multi-server system. This means that there should be little performance penalty for multi-server systems resulting from extra server-to-server IPC costs.

The Mach-US system was found to have about 10-20% slower performance than Mach-UX in general. However, on highly parallel operations its performance was found to be on a par with Mach-UX.

Lites: 4.4BSD Unix on Mach

Lites [9] is an implementation of 4.4BSD Lite Unix on Mach, with the goal of providing a free Unix for Mach that bridges the gap between different Unix flavors by supporting binaries from the different Unix variants. It consists of a server based on 4.4BSD Lite and an emulation library, and provides binary compatibility with several versions of BSD as well as Linux on the x86 platform.

The emulation library is considered as part of a user program, as a shared library, and it runs on the user program's stack. This is unlike other systems such as Mach-UX where the emulation library is more like a user-mode kernel extension than a library and runs with its own stack. The emulation library provides some services directly and contacts the server if required.

The Lites server is designed as a single multithreaded Mach task. This creates a number of issues in using the 4.4BSD Lite code. The BSD code is designed for single processor systems. As such, the BSD code is not reentrant, which poses problems as threads in the Lites server can be preempted at any point. To solve this, a master lock is used to protect the parts of the BSD code that are non-reentrant.

The proc structure in Lites is split into 2 structures, proc and proc_invocation, to support multi-threaded processes making system calls. The proc_invocation holds system call state and is local to a server thread. When a server thread services a system call, its invocation is linked to the process that made the call. In this way each server thread has a "current process"; there is no global current process in Lites, unlike 4.4BSD. The server thread handles the system call until completion.

It was found that Lites is somewhat slow compared to native FreeBSD and Linux. The slowness was found to be mainly due to the IPC costs – system calls implemented in the emulation library are fast,

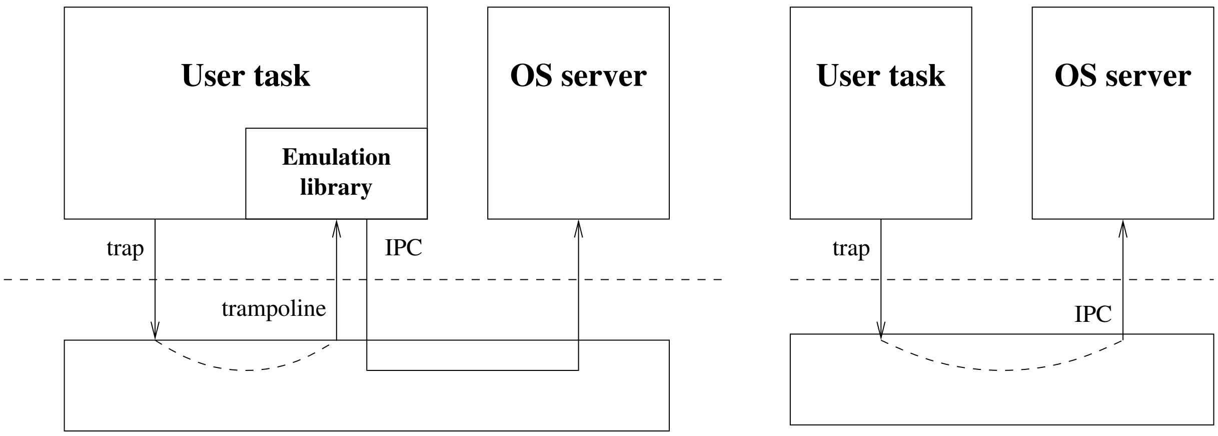

Figure 2.2: Emulator (left) vs no-emulator design

while system calls that require the server are slow.

OSF/1 MK

OSF/1 is a version of Unix developed by the Open Software Foundation (OSF) in the early 1990's. It has a monolithic kernel based on Mach 2.5. OSF/1 MK is a version of the OSF/1 operating system, implemented as a server running on Mach 3.0 by the OSF Research Institute (OSF RI). The design of the OSF/1 MK server was originally based on the design of the Mach-UX server from CMU. OSF subsequently made many improvements upon this design, which include a number of modifications to Mach:

No-emulator One of the changes made by OSF was to remove the emulation library and to disallow Mach calls from user processes [13]. It was claimed that allowing unprivileged processes direct access to Mach calls was a potential security violation, because the server is bypassed and so it cannot enforce its security policies. Also, direct Mach calls may confuse the server as they may put the system in a state that it does not expect. However, the main reason was probably because the emulation library increased maintenance costs. This is because the system call handler of OSF/1 MK using the emulation library is significantly different from monolithic OSF/1, and OSF needs to maintain both OSF/1 and OSF/1 MK in parallel.

To implement system call handling without using an emulator, a new exception protocol was defined in Mach, such that on an exception or a system trap a message with the thread state is sent to the thread's exception port. This message is received by the server. A modified state is returned with the reply, which the kernel sets before the thread resumes. The result of this change was that the internal structure of the OSF/1 MK server became more similar to that of the monolithic OSF/1, such that even the system call handler could be used with only minor changes.

An issue with removing the emulation library is that copyin and copyout become much more cumbersome to implement. Copyin and copyout are routines used for copying data between kernel and user space. Previously, copyin and copyout are implemented in the emulation library, which resides in the user process address space. Without the emulation library, copyin and copyout have to be implemented in the Unix server. This is difficult since the Unix server does not have direct access to the address space of the user process. There were two solutions to this, both of which involved extending the Mach VM system. The first solution was to extend the vm_read and vm_write calls, which allow access to the address spaces of other tasks, so that they support non-page-aligned access. The second solution was to add a new call, vm_remap, which maps part of the address space of a task into another task. The server uses this to map part of the user process address space into the server address space.

Co-location The performance of OSF/1 MK was found to be less than satisfactory, even after extensive optimisations. Condict et al. [2] claimed that the reason for the unsatisfactory performance is due to RPC costs. They also claimed that the reason for using a microkernel is for improved modularity of the system. The solution they proposed is a technique called *co-location*, where the server is placed in the kernel address space and a *short-circuiting RPC* mechanism is used where a RPC is converted to a procedure call whenever possible. The short-circuiting RPC mechanism is implemented in the stubs generated by the Mach interface generator (MIG), and works as follows:

- If the client and server are in the same task, the RPC stub makes a direct procedure call.
- If the client is a user task and the server is the kernel, the RPC stub makes a direct kernel request using a special system call trap.
- Otherwise, the RPC stub performs a normal message-based RPC.

The RPC stubs are implemented in such a way that the same binary can be loaded as a user-level server or as a co-located server without modification.

For a system trap on a system using the no-emulator model, a system trap is converted into a direct procedure call inside the Mach kernel. A typical RPC (or emulated system trap) to a co-located server has the following steps:

1. A user task traps into the Mach kernel.
2. The Mach kernel makes a procedure call to the correct procedure in the co-located server. This is done in the user thread that makes the request.
3. The server handles the request, possibly making some Mach calls by direct procedure calls.
4. The procedure returns, and Mach returns control back to the user task.

This is essentially the same as the steps taken when a monolithic kernel takes a system trap. Based on this, Condict et al. claim that a co-located server using short-circuiting RPC has a negligible performance penalty compared to a monolithic "integrated" kernel (an example of which is the Darwin/Mac OS X kernel).

The key difference between a co-located server and a integrated kernel is that interactions between the co-located server and Mach must go through the MIG-generated stub code. This enforces modularity, which Condict et al. identified as the reason for using the microkernel approach. It is not possible for the co-located server to "accidentally" access internal Mach structures, since it is not linked with Mach, but merely loaded into the same address space.

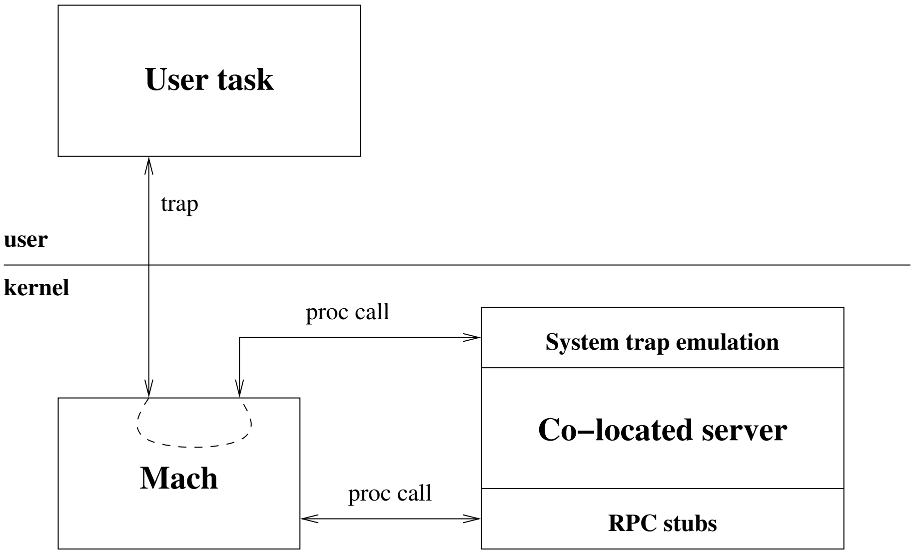

Figure 2.3: System trap emulation in a co-located server

The method of co-location removes the security advantages of running a microkernel. Condict et al. makes the assumption that the OS server is a trusted component, and so the protection provided by running the server in user-level is not important. They observe that co-location is not appropriate for highly secure or high availability systems where this additional security is important.

Thread migration To reduce the cost of context switching in an RPC between two user-tasks, OSF adopted a *migrating thread RPC* model developed by Ford and Lepreau [5]. The thread structure is split into 2 structures, the *shuttle* and the *activation*. The shuttle represents the actual thread of execution and the activation represents a user thread.

The server is modified so that instead of having a pool of threads waiting for messages, there is a pool of empty activations instead. These activations contain a stack pointer and an entry point. When an RPC operation occurs, the shuttle migrates from the calling thread to an empty activation in the server. Execution commences in the server at the activation's entry point. When the RPC operation completes, the shuttle migrates from the server back to the calling thread.

The advantages of this approach is that the complexity of the RPC path is greatly simplified. This results in a substantial speedup of RPC operations, with the null RPC operation being 3.6 times faster using migrating threads than using static threads.

MkLinux

MkLinux [3] is a port of Linux by OSF and Apple to run on OSF's version of Mach (OSFMK). The version of Linux initially ported was 1.2.13. MkLinux is a single server system, since implementing MkLinux as a multi-server system would require significant changes from the original Linux code. The MkLinux server is multithreaded, with separate threads to handle such things as replies to device requests, incoming network packets, and pager requests.

There is a pool of system call threads which wait on a port set of all user task exception ports. System call threads are not dedicated to a particular task. Since the Linux code is not designed for parallel execution, there is a global mutex used to serialise execution in the server.

MkLinux does not use emulation libraries residing in the user address space. The reason given for this is the same as that for OSF/1 MK: that the server functionality is split between the emulation library and server, increasing the complexity and potentially causing security violations. MkLinux therefore uses the same no-emulator system call mechanism as OSF/1 MK.

Since device drivers are supplied by the Mach kernel, MkLinux was modified to use these drivers. Linux has device drivers built in, with a device numbering scheme and device operation routines. In MkLinux, a generic emulation layer was added, which translates between Linux device numbers and Mach device names, and replaces the device operation routines with generic routines which use the Mach devices.

In Linux and most Unix systems, pending signals are delivered to a process when the kernel returns to the process upon completion of a system call, an exception, or an interrupt. For a process which loops and does not call a system call or cause an exception, the kernel relies on interrupts occuring to deliver the signal. This is a problem for MkLinux since interrupts are handled by Mach and never delivered to MkLinux, and so such a thread will never have signals delivered. This problem was solved by having a "fake" interrupt when a signal needs to be delivered, which triggers the signal delivery mechanism.

Since Mach provides memory management functions, the Linux VM operations and structures were emulated using their Mach equivalents. This created problems as the Linux code accesses the page tables extensively via macros. Stubs were provided for these macros to allow most of the Linux code to compile and run without modification.

Linux uses a dynamic buffer cache. The nature of a dynamic buffer cache requires it to interact with the VM system. When the system runs out of pages, the pages in the buffer cache should be written out to disk and discarded. The problem is that Mach does not know this, and it would page out pages when there are still pages which can be freed from the buffer cache. The solution was to add an *advisory page out* notification for pagers. This allows the pager to select the page to free, so that it can free a page other than the one selected by the microkernel.

2.3 L4: a second-generation microkernel

Despite years of research towards enhancing the performance of first-generation microkernels such as Mach, their performance still fell well short of that offered by traditional monolithic kernel designs. It was also widely believed that microkernels are not sufficiently flexible. Liedtke [12] argues that the inefficiency and inflexibility are not inherent in the microkernel concept, but can be avoided with proper design. L4 [11] was developed with the idea that for a microkernel to be efficient and flexible, a minimal, orthogonal set of kernel abstractions are required. The L4 microkernel (or more accurately, the L4 family of microkernels) is based on three basic abstractions: address spaces, threads, which execute inside address spaces, and message-based inter-process communication (IPC).

The description below is based on the L4 X.2 specification, which is implemented by the L4Ka::Pistachio microkernel.

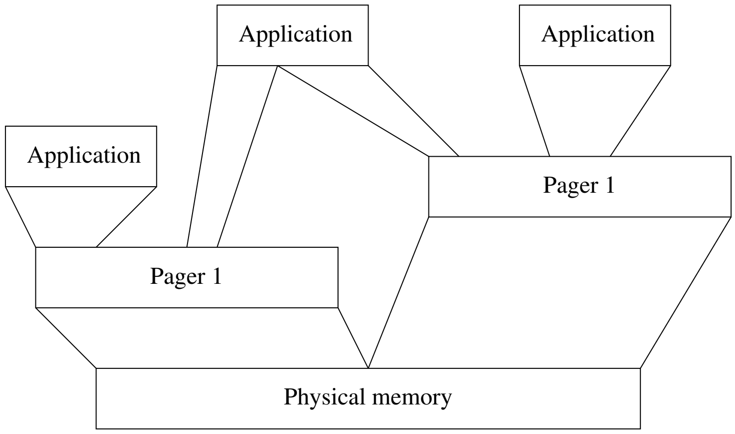

Figure 2.4: L4 recursive address space mapping

Threads A thread in L4 executes within an address space. Each address space can have multiple threads.

Address spaces Address spaces in L4 are constructed recursively. The initial address space is σ_0, and represents physical memory. Other address spaces are constructed by three operations, *grant*, *map* and *unmap*.

- The *grant* operation unmaps a region from the caller's address space and transfers it to another address space.
- The *map* operation maps a region from the caller's address space into another address space, resulting in the region being accessible in both address spaces.
- The *unmap* operation "undos" a map, unmapping the pages from all address spaces that received the mapping directly or indirectly from the caller.

All three operations operate on virtual pages in the caller's address space, not on physical pages. So these operations can only affect pages that are mapped into the caller's address space. Furthermore, map and grant operations are sent as messages, which a thread in the target address space has to explicitly receive. This ensures that a thread cannot arbitrarily map pages into other address spaces.

Page faults are handled by specially designated threads called *pagers*. Pagers receive page fault notifications as messages, and they handle the page fault by issuing a map or grant operation to map the required page in the faulting address space.

Using this recursive address space mechanism, memory management can be handled by user-level servers.

Message-based IPC Inter-process communication on L4 is provided by a message-based IPC service. IPC performance is maximised by avoiding unnecessary memory references. The use of synchronous IPC simplifies the IPC implementation and avoids the unnecessary use of buffers.

IPC messages are passed in *message registers*. A thread sends an IPC by writing the message into its message registers. The receiver then reads the message from its message registers. Message registers can be implemented as hardware registers to maximise IPC performance.

Interrupts L4 supports the use of user-level interrupt handlers. A interrupt handler thread can be registered for each interrupt. When L4 receives an interrupt, it is delivered as an IPC message to the corresponding interrupt handler.

This is the key mechanism which allows the implementation of device drivers outside the kernel as user-level servers.

Exception handling For each thread, an exception handler and a system trap handler can be specified. Exceptions and system traps made by a user thread are converted by L4 into IPC messages to the thread's exception or system trap handler. The IPC message contains thread state, including registers. The exception or system trap handler resumes the thread by sending a reply. The reply message contains the modified state, which the kernel sets before the thread resumes.

Protection mechanism In the current version of the L4 X.2 specification, there is no mechanism for restricting IPC messages between threads; any thread can send a message to any other thread. This is a shortcoming of the current L4 specification and will be addressed in future revisions.

In previous versions of L4, a *clans and chiefs* mechanism was used. Each thread belongs to a *clan*, which has a thread which is specified as the *chief*. Messages which cross the clan boundary are intercepted by the chief, which decides whether to block the message, forward the message unchanged, or forward the message with modifications. This method was abandoned as being too inefficient.

2.3.1 L4Linux: Linux single server on L4

To validate the claims made by Liedtke [12] and others about the performance of second-generation microkernels, a Unix OS server was needed for L4. Härtig et al. [8] chose Linux to be ported to L4 for this purpose. The version of Linux originally ported is 2.0. L4Linux has since been updated to Linux 2.2.

Linux is divided into architecture-dependent and architecture-independent components, with a well-defined interface between them. The architecture-dependent component implements such things as interrupt service routines, low level device driver support, context switching, system call mechanisms, signalling, copyin/copyout, and map/unmap operations on address spaces.

To minimise the changes that needed to be made to Linux, L4Linux is implemented as a server in a single L4 task. Modifications to Linux are restricted to the architecture-dependent part. Each user process in L4Linux runs in a separate L4 address space.

L4Linux is responsible for managing memory. Upon boot, L4Linux requests all memory from its pager. This is usually σ_0, and the result is that all physical memory is mapped to L4Linux. The L4Linux server then acts as a pager for its user processes. Since L4Linux cannot access L4 page tables, it needs to maintain its own record of page mappings. This is done by using the the original Linux page tables as logical page tables, which mirror the real page mappings within L4.

A single L4 thread is used to handle system calls and page faults. To deal with blocking within Linux, L4Linux uses multiple user-level threads within the server. Although using multiple L4 threads is arguably faster and more elegant, this was not done because it would require substantial changes to Linux.

L4Linux top-half interrupt handlers are implemented as threads waiting for L4 interrupt messages. Bottom-half handlers are executed in another thread. These threads are given an elevated priority, to ensure that they do not run concurrently with the rest of L4Linux. Using this mechanism, all Linux device drivers could be used without modification.

For binary compatibility, L4Linux uses an emulation library which is mapped into the user process address space. A trampoline mechanism is used to redirect system calls into the emulation library, where it is converted into an IPC to the L4Linux server. To minimise the changes required to Linux, the emulation library does not emulate Linux system calls locally; it is only responsible for translating system calls into IPCs to the server. For dynamically linked programs, a modified version of the standard shared C library is also used, which uses L4 IPC directly and bypasses the trampoline mechanism. It is intended that most Linux programs will use the modified C library.

Linux always maps the user address space into the kernel address space. However it was found that mapping a user address space into L4Linux resulted in significant performance penalties. Therefore instead of direct copies, copyin and copyout copy to "physical" addresses using page table lookups.

Signal delivery in Linux is done by directly manipulating the stack pointer and instruction pointer of the user process. This is not possible in L4Linux, because manipulations of thread state are restricted to other threads running in the same address space. This is solved by adding a signal handler thread to each user process. Signals are delivered as IPC messages to this thread, which then manipulates the state of the main thread to deliver the signal. This is safe as this mechanism is only used to deliver signals; the effects of a signal such as SIGKILL is handled by the server and does not involve the signal handler thread.

L4Linux was found to be about 8% slower than native Linux, compared to 29% for co-located MkLinux and 49% for user-mode MkLinux. This contradicts the argument of Condict et al. [2] that co-location offers similar performance to a native monolithic kernel. It is not clear why co-located MkLinux is slow, but it seems that co-location is not sufficient to overcome the performance penalties of a slow microkernel.

2.4 Other microkernel-based systems

2.4.1 Lites on L3

Lites on L3 is a port of the Mach-based Lites single server to run on L3. Hohmuth and Rudolph [10] identified a number of Mach functions used by Lites that have no equivalent on L3:

- *Ports*, the IPC addressing mechanism used by Mach, are used as references to kernel objects such as files.
- *Port rights* are used as the protection mechanism in Lites to prevent unauthorised accesses to kernel objects.

- The *Mach interface generator* (MIG) is used to produce the interface between the server and emulation library.
- The *Mach C threads* library is used extensively in the Lites server.

To port Lites to L3, an emulation library was created to emulate this functionality.

Mach IPC messages are sent using the mach_msg system call. The emulation library emulates Mach IPC at this level, by emulating the behaviour of the mach_msg system call. To avoid extra IPCs, the emulation is performed in a decentralised manner in the emulation library of each process, rather than in a central server. A complicated protocol is used to emulate the behaviour of port rights and the sending of port rights over IPC messages. Emulating mach_msg eliminates the need to emulate MIG, since the original MIG-generated stubs which call mach_msg can be used. However, in doing so the possible performance benefits resulting from a stub generator tailored for L3 IPC is lost. The reason that Hohmuth and Rudolph gave for this decision was that a suitable stub generator is not yet available.

The Mach external pager interface is not emulated, as doing so would increase complexity significantly. Instead, the Lites server is modified to use the L3 external pager interface. The only Mach VM calls emulated are vm_allocate and vm_deallocate, which are used to provide simple memory allocation and deallocation.

It was found that message transfers using the mach_msg call are slightly faster in the L3 emulation compared to native Mach if the message only contains uninterpreted data, and significantly slower if the message contains port rights. Native L3 IPC was significantly faster than both native Mach IPC and emulated Mach IPC on L3. For remote procedure calls, native Mach was found to be faster than the Mach emulation on L3. This is attributed to the RPC path on Mach having been specially optimised while no such optimisation has been performed on the emulation library.

It turns out that the penalty of the decentralised protocol for the sending of port rights was more than the penalty of the double IPC required if a central server was used. Therefore this decentralised approach may not be suitable for a system in which the sending of port rights is used extensively.

2.5 Summary

In the survey of microkernel-based systems, there are a number of issues that are common to many of the systems. This section provides a summary of these issues and the approaches that different systems take in solving them.

2.5.1 System call emulation

A common requirement of microkernel implementations of Unix systems is binary compatibility with existing Unix programs. Unix programs make system calls with system trap instructions, but Unix servers typically have an IPC interface based on the IPC mechanisms of the underlying microkernel. For binary compatibility, there needs to be some way of converting system traps to an IPC to the server.

The solution adopted by early Unix servers such as Mach-UX [7] and Mach-US [16] was to use an emulation library. This library resides in the address space of every user process. Using a mechanism provided by the microkernel, system call traps are redirected to the emulation library where it is emulated.

The emulation library performs some local processing, and the server is accessed only if required. Such an emulation library is able to provide excellent performance for system calls which do not require the server.

Patience [13] claimed that using an emulation library creates a potential security violation, because user programs are allowed to access and modify emulation library code as well as use microkernel services. The no-emulator design was introduced, where the microkernel has the responsibility of translating system traps and exceptions into IPC messages. This design eliminates the need of having an emulation library within the user address space. The no-emulator design was applied to OSF/1 MK, where it was found that fewer changes are required to adapt a monolithic kernel to run on a microkernel compared to designs which use an emulation library. The no-emulator design was also used in MkLinux, to reduce the complexity of the implementation and to prevent security violations.

Helander [9] argued against the no-emulator approach. He claimed that the security problems can be solved by proper design, and that the no-emulator design introduced additional problems in accessing user address spaces. He did not consider it important to minimise the number of changes required to adapt a monolithic kernel to run as a server.

L4Linux [8] uses an emulation library which does not do any processing locally. In doing so the number of changes required to Linux was minimised, however this approach forgoes the performance benefits which can result from the use of an emulation library.

2.5.2 Accessing user address spaces

Traditional monolithic Unix systems such as BSD and Linux are designed to access user address spaces directly. Direct access to user address spaces is difficult in microkernel-based systems, since the Unix server runs in its own user address space. Typically, it is system calls with buffer arguments that require the kernel to access a user address space.

In systems which use an emulation library, such as Lites [9], the buffer arguments of a system call are transferred by the emulation library over IPC messages. This is not possible in no-emulator systems and on systems where the emulation library performs no processing, since identify and copying the buffer requires the emulation library to understand system call semantics. The solution used in OSF/1 MK and MkLinux, which use no-emulator designs, was to modify the microkernel to provide calls for accessing user address spaces.

In L4Linux the problem was solved differently. It was found to be impractical to map the user address space to the server. However, since L4Linux performs memory management for user processes and has access to logical page tables, access to user address spaces can be implemented using page table lookups and accesses to "physical" memory. This was the solution chosen for L4Linux.

2.5.3 System call handler threads

A problem encountered when adapting monolithic kernels to run on microkernels is that the implementation of some system calls are designed to block. This poses a problem for single-threaded servers since if a system call blocks, the thread cannot return to the system call loop to process additional system calls while this call is blocked. For a server which uses multiple threads to handle system calls, there are no issues with blocking threads, but there is the problem of choosing a thread that is not blocked to deliver an incoming system call.

Mach-based systems generally use multiple threads to handle system calls. These systems make use of the Mach facility to receive messages on port sets and rely on Mach to deliver messages to an available thread.

No such facility is available on L4 since IPC messages are addressed by thread ID. The solution used in L4Linux [8] is to use a single thread, where all system call IPCs are delivered. Blocking system calls are handled by multiplexing the thread at user-level.

2.5.4 Performance

A major problem encountered in early microkernel-based systems was their poor performance compared to monolithic systems. This was found to be mainly the result of the cost of IPC operations. A number of techniques to improve performance were developed in response to this:

- Transferring functionality to the emulator library, avoiding IPC operations whenever possible.
- Migrating thread RPC [5].
- Co-location [2].

Liedtke [12] argued that the performance problems result from the design of the microkernel. He produced the L4 microkernel which avoids the design problems that were identified. This was demonstrated in L4Linux [8], where the performance was found to be significantly better than that achieved with MkLinux, even when co-location was used to improve performance.

2.5.5 Signal delivery

Signal delivery in traditional Unix systems is performed by directly modifying the state of the target process, usually just before the target process resumes.

This was a problem in L4Linux since L4 does not provide a mechanism for modifying the state of a thread outside the current address space. The solution was to add an extra signal handler thread to every user process. This thread receives signals as IPC messages, and delivers the signal by modifying the state of the main thread in the process.

Chapter 3

XNU, the Darwin/Mac OS X kernel

3.1 Overview

Darwin is the open-source core of Apple's Mac OS X operating system, which includes a suite of standard Unix tools, as well as a kernel called XNU [1], which is based on 4.4BSD-Lite2 and Mach.

The version of Mach which XNU is based on is derived from a version of Mach 3.0 adapted to the PowerPC platform by Apple and the Open Software Foundation. This adapted version was known as OSFMK, and was used in MkLinux before being incorporated into XNU.

In previous implementations of UNIX on Mach, the method of placing servers within the kernel, called *co-location* [2], was used to improve performance to acceptable levels. Using co-location, servers are placed in the kernel address space. However the kernel and server are not linked together, and all interaction goes through the interfaces generated by the Mach interface generator (MIG). This is the same interface as that for user-level servers, albeit with extra code which allows a faster IPC path for co-located servers. In XNU this is taken a step further: the Mach code is linked with the BSD code, and calls from BSD to Mach are done as direct function calls, which do not use a MIG generated layer.

In addition to the BSD and Mach components, a major component of XNU is the IOKit, an object-oriented device driver framework, implemented in a restricted subset of C++ based on the Embedded C++ specification. IOKit drivers are loaded dynamically into the kernel.

XNU relies on Mach for a number of essential operating system services, which are:

- Tasks and threads
- Virtual memory management
- Inter-process communication (IPC)

This section is based on the version of XNU which is part of Darwin 6.6 and Mac OS 10.2.6.

3.2 Structure

XNU contains three major components linked into one address space: Mach, BSD and IOKit. The boundaries and interfaces between the components are not well defined: there are many undocumented

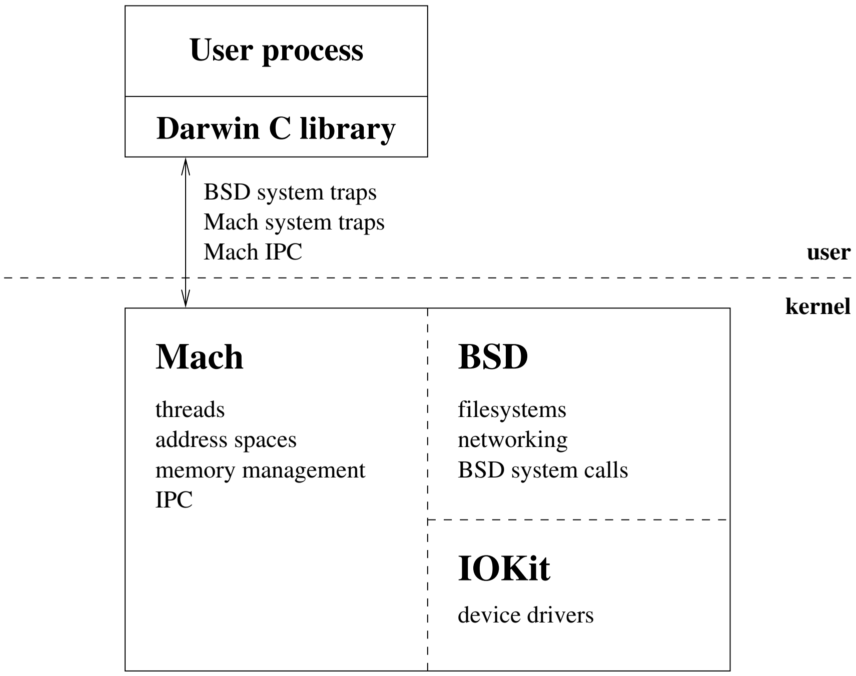

Figure 3.1: XNU system structure

function calls between the components, and there are instances where data structures are shared between components. Figure 3.1 shows the overall structure of the system, where the dotted lines inside XNU represents "fuzzy" boundaries which are not well defined.

3.3 Processes and tasks

XNU implements Unix-style processes and threads on top of Mach tasks and threads, with a one-to-one mapping between XNU processes and Mach tasks, and between XNU threads and Mach threads. A task in Mach is essentially a collection of system resources, represented by a set of port rights, as well as an address space. A task can contain a number of threads. Every thread belong to exactly one task, which defines its address space and access rights to system resources. Unix processes in XNU are built on top of Mach tasks, and add file descriptors, Unix users/groups, and parent/child process relationships.

3.3.1 Thread shuttles and activations

As part of the changes made to Mach to use a migrating thread model for RPC [5], the thread data structure was split into 2 structures, the *thread shuttle* and *thread activation*. The thread shuttle is the actual thread of execution, and contains such things as a kernel stack and scheduling and accounting information. The user state of a thread (i.e. registers, instruction and stack pointers) is contained in the thread activation, which represents a user thread and is tied to a specific task.

A thread shuttle maintains an *activation stack* of thread activations. When a thread shuttle migrates to a user task, an activation is pushed onto the activation stack. When it returns from the migration, an activation is popped off the activation stack. The activation on the top of the stack is considered to be the "current" activation, and represents the user thread that is executed when the thread shuttle goes into user mode.

3.3.2 BSD processes and threads

The proc structure represents a BSD process in XNU. It contains file descriptors, user/group information as well as parent/child process relationships. The relationship between BSD processes and Mach tasks are recorded using a pointer in the proc structure to point to the Mach task, and a pointer in the Mach task to point to the proc structure.

The uthread structure represents a thread in XNU. It contains information for BSD system calls, including parameters and return values, signals, and state for blocking system calls which use continuations. There is a one-to-one relationship between Mach threads and uthreads, which is maintained with a pointer in a thread activation pointing to the corresponding uthread structure.

3.4 System calls

User processes running on XNU make system calls using a variety of mechanisms. There are three main types of system calls: BSD system calls, Mach system traps, and system calls over Mach IPC.

Unlike other Unix-style systems built on top of Mach, BSD system calls in XNU do not use Mach IPC or use a trampoline mechanism to translate system call traps into messages. The system call mechanism in XNU is essentially that of traditional BSD systems: A system call is made using a system call trap instruction, with the parameters of the system call in registers. Inside the kernel, the system trap handler is looked up in the sysent table, a table of function pointers, and called directly. The return values of the system call are then returned in registers.

The basic Mach system calls, such as the mach_msg_overwrite call, used for sending IPC messages, are made using a system call trap instruction in the same way as the BSD system calls. They are distinguished from BSD system calls by using negative system call numbers; BSD system calls use positive system call numbers.

However, many system calls in XNU occur over Mach IPC using remote procedure call (RPC) stubs generated by the Mach interface generator (MIG). This include most Mach system calls, such as calls to the Mach VM system and calls for manipulating Mach tasks and ports, as well as all IOKit calls. A MIG-generated stub makes a system call by sending Mach IPC messages to a kernel port. The system call is specified in the ID field in the message header, and the parameters are passed in the body of the message. A reply message delivers the return values of the system call.

There are also a number of special "fast" system traps in XNU. These are the pthread_self and pthread_set_self calls, which are used to retrieve and store a data pointer in the Mach thread structure, respectively. These calls are used by the Darwin POSIX threads library.

3.5 Mach inter-process communication

The Mach inter-process communication (IPC) mechanism in XNU is used for communication between user processes, as well as for Mach system calls. It is not used for BSD system calls; these are implemented as simple traps into the kernel.

The Mach IPC interface in XNU is of particular importance for L4-Darwin as it is the only Mach interface that Apple does not discourage the use of by applications [6]. Although applications can be written with minimal use of Mach IPC, which is the case for most command-line applications in Darwin, Mach IPC appears to be used extensively for the Mac OS X GUI and applications which use it. This can be seen from the data in section 3.9.

3.5.1 Ports as kernel object handles

In addition to the usual use of Mach ports as the addressing mechanism for IPC, Mach ports are also used as handles to kernel objects. Ports can be associated with a pointer to a kernel data structure and a kernel object type.

3.6 Mach virtual memory subsystem

XNU uses the Mach memory management system for virtual memory. The Mach memory management system [15] is a portable memory management system originally developed by CMU as part of Mach. It had several features which was notable in its time, in particular its support for user-level pagers, as well as its portability which derived from the use of a thin machine-dependent layer with the bulk of the code being machine-independent.

In XNU, a number of changes have been made from the original Mach sources:

- Support for external pagers have been removed. However, in the Apple developer documentation [1] Apple states that support is planned for a future version of XNU.
- A new abstraction, called the *Universal Page List* (UPL), has been added, together with a number of routines for operating on UPLs. UPLs are used extensively by the paging code for communication with the memory management system.
- *Named entries* have been added. A named entry is a handle to a VM object or a submap, visible to user tasks. Named entries can be mapped into an address space using vm_map. This is intended as a means for tasks to share portions of their address space.

3.6.1 Memory objects and address spaces

The basic unit of Mach memory management is the *memory object*. A memory object represents data which can be mapped into memory, and is backed by a *pager* which supplies the data for the memory object.

Each Mach task has an address space. A Mach task can allocate anonymous memory, map memory objects, and deallocate memory in an address space. Memory objects can be mapped with some combination of read, write and execute permissions, subject to hardware support. Child tasks inherit the address space of its parent. Regions of the address space can be specified to be either shared, copied or not shared, which determines whether it is shared, copied or discarded when a child task's address space is created.

3.6.2 Data structures

A *VM object* is an internal abstraction which represents the mapping of a memory object into memory. It contains a set of pages allocated to the VM object, and information about the memory object that it maps. Physical addresses can be looked up by VM object and byte offset. For fast lookup, memory mappings are recorded in a hash table structure, indexed by VM object and byte offset. Information about physical pages are maintained in a table indexed by physical page number. Each page may be linked into a number of lists:

- *Memory allocation lists* are used to keep track of free and allocated pages. There are three such lists; each page is linked into exactly one of these three lists:
  - The *free list* contain pages which are not used and are available for allocation.
  - The *active list* contains pages which are mapped to an address space.
  - The *inactive list* contain pages which are not mapped to any address space, but are allocated to a VM object. Pages on the inactive list may be dirty.
- A *memory object list* is part of a VM object, and contains all physical pages allocated to that VM object. Each physical page belongs to at most one VM object.

  This list is not used for page lookup, but it is used to speed up operations which operate on all pages of a VM object, such as object deallocation.
- Each physical page allocated to a VM object is also linked into a *object/offset hash bucket*, used for looking up physical addresses by VM object and byte offset.

Virtual addresses in an address space are mapped to byte offsets into VM objects using an *address map*. An address map is a doubly-linked list of *address map entries*, sorted by ascending order of virtual address. An address map entry maps a range of virtual addresses into a range of offsets into a VM object, and virtual addresses mapped by different entries in the same address map may not overlap. Each address map entry contains a reference to a VM object as well as inheritance and protection flags.

Copy-on-write duplication of address spaces is supported by the use of *shadow objects*, which hold modified pages resulting from copy-on-write faults. A shadow object is created as the result of a copy-on-write fault. It is a VM object which initially contains no pages, but has a reference to another VM object. Shadow objects can themselves be shadowed, forming a chain of VM objects, known as a *shadow chain*.

If a page in a shadow object is written to, it is copied and added to the shadow object before it is modified. Page lookups on shadow objects are performed by first looking in the shadow object. If the page is not found there, the search proceeds up the shadow chain until the page is found.

Shadow chains can potentially become very large from repeated copy-on-write duplication of VM objects, which can cause page lookups to be very slow. To deal with this, the Mach memory management system removes intermediate VM objects when it recognises that they are no longer needed.

For *anonymous memory*, the Mach memory management system supports lazy creation of VM objects and memory objects. When anonymous memory is allocated, an address map entry with no VM object is created. If the memory region is accessed, a VM object with no associated memory object is created. If the memory needs to be paged out, a memory object backed by the default pager is allocated.

Instead of a reference to a VM object, an address map entry can point to a *sharing map*. The sharing map data structure is identical to the address map. This is used for sharing of address space ranges, which can potentially contain a large number of objects.

3.6.3 Machine dependent component, Pmap

One of the key features of Mach is its separation between machine dependent and machine independent components, with well-defined interfaces between them. Pmap is the machine dependent component of the Mach memory management system. It provides operations for working with *physical map* (pmap) data structures, which represents a set of virtual-to-physical mappings for an address space.

The actual data structure used is dependent on the hardware; on most architectures, such as PowerPC, it is a page table. As well as maintaining the pmap data structure, the pmap component is responsible for maintaining the state of the hardware MMU to ensure that it is consistent with the currently active pmap.

The pmap acts as a cache for virtual-to-physical mappings; all mappings can be determined from the machine independent data structures alone. A consequence of this is that the interface does not require the pmap to keep track of all mappings; most mappings can be discarded at any time.

3.6.4 Universal Page Lists and Pagers

Universal Page Lists (UPL) are data structures in XNU which are used extensively by pagers. A UPL is a list of physical pages corresponding to a a range of pages within a VM object. It represents a currently active paging operation on a VM object. Operations can be performed on UPLs, such as modifying flags on the pages, and the changes can be either applied to the actual pages in the VM object or discarded. See section 3.7 for a more detailed description of the UPL data structure and interfaces.

Each memory object in Mach is backed by a pager which provides calls for populating a VM object with data and for flushing the data in a VM object to a backing store.

In XNU, the only pagers currently supported are internal pagers, since the *External Memory Management Interface* for using user-level pagers is not yet implemented. The pagers provide two main calls, a *data request* call and a *data return* call. For both calls, the pager is passed a reference to a memory object and a *memory object control*, which in XNU contains a VM object. It is also passed an offset into the memory object and a size, which specifies a range of addresses. The data request call is expected to populate the range by causing the pages to be mapped in the VM object and filling them with data. The data return call is expected to write the data in the specified range to a backing store.

There are three different pagers available in XNU: the *vnode pager*, the *device pager*, and the *default pager*.

- The vnode pager is used for the mapping of files into memory. The vnode pager code is a wrapper around the vnode_pagein and vnode_pageout calls in the BSD component, which are in turn wrappers around the pagein and pageout VFS operations on the vnode. These operations are described in section 3.7.
- The device pager is used for mapping device memory into a task's address space. The device pager code is a wrapper around an IOKit IOMemoryDescriptor object, which represents a range of memory, specified as one or more physical or virtual address ranges. Mapping an object backed by the device pager into a tasks's address space allows the task to access this memory directly.
- The default pager is used for the paging of anonymous memory. Anonymous memory is not associated with a memory object until it needs to be paged out (for example, when the system is low on memory). When this happens, a memory object backed by the default pager is allocated.

3.7 Filesystems and the Unified Buffer Cache

XNU uses a *Unified Buffer Cache* (UBC) for caching of files.

In modern UNIX and UNIX-like systems, there are two ways of accessing files - either through read/write system calls, or through mapping the file contents into memory using mmap. In a typical system without a UBC, all disk accesses are cached using a *buffer cache*. The buffer cache is used by both the read/write system calls as well as mmap. To map file data into memory, mmap stores the file data in a *page cache*, managed by the VM system, which can then be mapped into a process's address space.

The problem with this is that file data can end up being cached twice, in the buffer cache as well as in the page cache. This wastes memory, reduces performance because of extra copying, and creates the possibility for inconsistencies between the two caches. A UBC solves this problem by providing a single mechanism for caching files accessed through memory mapping and through UNIX-style system calls.

3.7.1 UBC design

In XNU, the UBC is implemented using Mach memory objects. A memory object is associated with each vnode, which is used to hold the pages which are currently being used to cache the vnode. The UBC has functions which read in and write out these pages.

The UBC code interacts with the memory management system using the *Universal Page List* (UPL) interface, which allows the UBC code to allocate pages to the memory object, and to signal the beginning and end of operations on the memory object. A typical UBC operation to read in or write out a region of the file proceeds as follows:

1. memory_object_upl_request is called, marking the beginning of a UBC operation. If the operation is for reading in, pages are allocated for the "holes" in the region, and these pages are marked as absent; if the operation is for writing out, the holes are not filled. In both cases, a list of the physical addresses of the pages is returned.
2. The UBC code fills the absent pages with data if it is a read operation, or writes out the pages if it is a write operation.

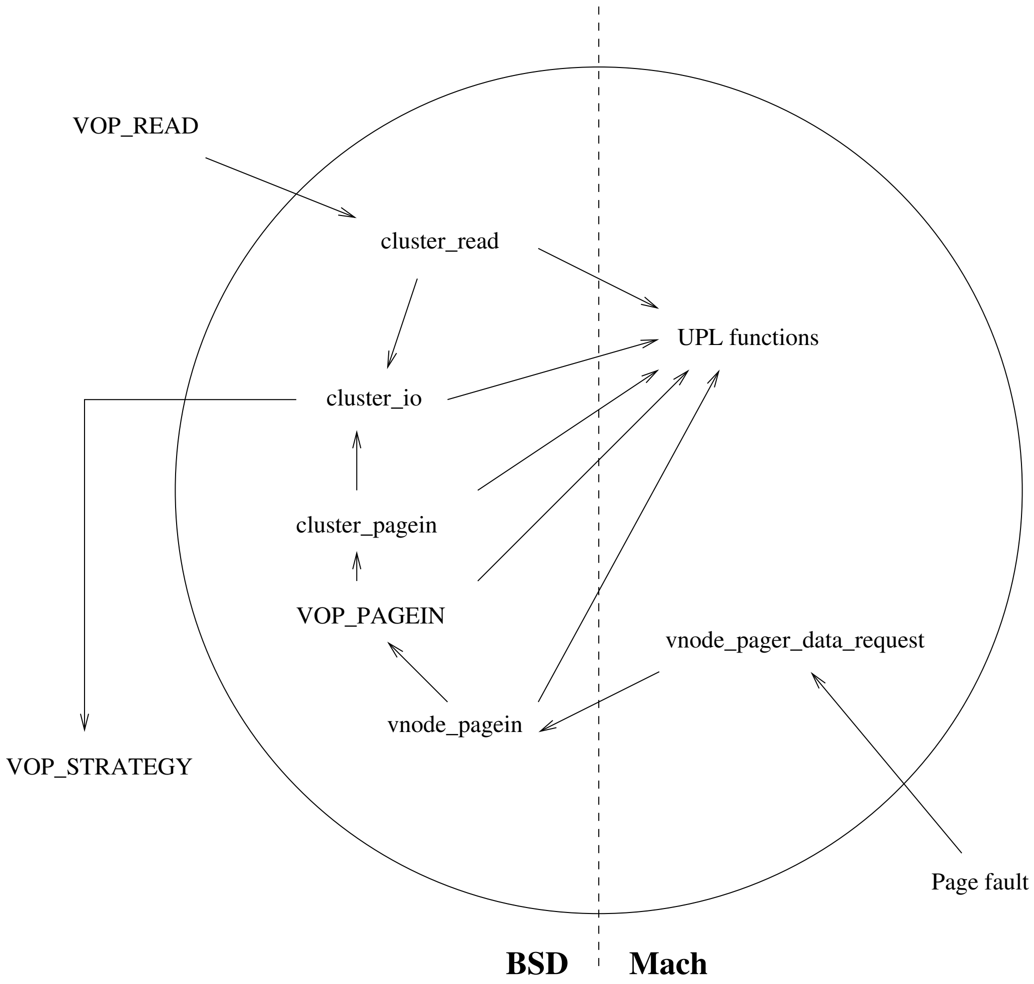

Figure 3.2: Simplified call graph for read and page-in using the UBC

3. The upl_commit function is called to mark the end of the UBC operation. If it is a read operation, the absent flag is cleared on the pages that are read in.

   The UBC operation may not be completed all at once. In this case the upl_commit_range function can be used to mark the completion for a subset of the pages.

The actual UBC code is located in the VFS operations of each filesystem, mainly in the read, write, pagein and pageout operations. For filesystems on block devices, much of this code is identical, and the common portions are placed in the *cluster IO* routines. These routines are described below in section 3.7.5. The NFS filesystem has a separate implementation of these operations.

The memory object used for the UBC is actually a *vnode_pager* structure. This allows it to double as a memory object which can be mapped by the VM system into an address space. Paging requests on this object cause UBC operations to be called on the vnode, via the *vnode pager* wrapper functions.

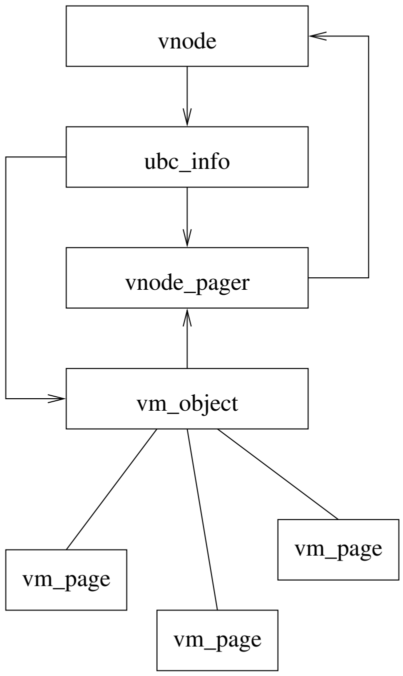

Figure 3.3: UBC data structure relationships

3.7.2 Data structures

ubc_info The ubc_info structure is used to hold the memory object information for a particular vnode. It contains a memory_object_t, which is a reference to a pager, and a memory_object_control_t, which in XNU is a reference to a vm_object.

upl A UPL is created either for paging in or for paging out. Logically, a UPL contains a list of physical pages corresponding to a a range of pages within a VM object.

A UPL also contains a VM object containing the pages; in some situations the UBC read and write functions maps this into the kernel address space to read and write the pages of the UPL.

upl_page_info_t A upl_page_info_t is an entry in the UPL page list. Each entry in the UPL page list contains the physical address of the page, plus a number of flags: pageout, absent, dirty, precious, and device.

3.7.3 UPL interfaces

memory_object_upl_request The memory_object_upl_request function is used to obtain a UPL for the VM object corresponding to a memory object. The function is given the offset and size of the range for which the UPL is to be created, as well as a flag which determines whether the UPL is created for page in or page out.

When a UPL is created for page in, pages that are not present are allocated and marked as absent, then included in the UPL. When a UPL is created for page out, pages that are not present are not included in the UPL.

upl_abort, upl_abort_range These functions are used to abort a paging operation on the given range of pages, returning the pages to the state before the UPL was created. The upl_abort_range function operates on part of a UPL; the upl_abort function operates on the entirety of the UPL.

upl_commit, upl_commit_range These functions are used to signal the completion of a paging operation on the given range of pages. The upl_commit_range function operates on part of a UPL; the upl_commit function operates on the entirety of the UPL.

upl_page_present, upl_dirty_page, upl_valid_page, upl_phys_page These functions are used to request information about an entry (i.e. a physical page) in the UPL. These functions are given an UPL and an offset into the list of physical pages of the UPL. upl_page_present returns true if the page is present in the UPL; upl_dirty_page returns true if the page is dirty; upl_valid_page returns true if the page is not marked as absent, and upl_phys_page returns the physical address of the page.

3.7.4 VFS operations

Most of the UBC functionality is implemented in the VFS operations of each filesystem, mainly in the read, write, pagein and pageout operations. The read and write operations are given a user buffer and are expected to read in or write out the buffer to a file. In doing so the read and write operations are expected to keep the UBC up-to-date. The pagein and pageout operations are given a UPL and are expected to read in or write out the pages in the UPL. The UPL is expected to have been created from the VM object which is the cache for the file, such that the pagein or pageout operation actually reads in or writes out the cache for the file.

3.7.5 Cluster IO routines

The cluster IO routines, located in the VFS code of the BSD component, form the core of the UBC implementation for most filesystems. There are 4 main routines: cluster_read, cluster_write, cluster_pagein, and cluster_pageout.

The cluster IO routines operate on vnodes with ubc_info structures which are on filesystems which support the cmap VFS operation for mapping file offsets to disk blocks. Through the UPL interface, the routines allocate pages for the required region of the file. The routines get the physical addresses of the region from the UPL and read or write the file data to the physical addresses through calls which write or read the underlying block device.

The cluster_read and cluster_write routines are for the use of the read and write VFS operations. These routines construct a UPL of the region from the VM object associated with the vnode. For cluster_read, the physical addresses of each page in the region are obtained, and data is read in and written to the physical addresses, using a call to the strategy VFS operation of the underlying block device. The data is then copied from memory to the user buffer. For cluster_write, the data in the affected pages in

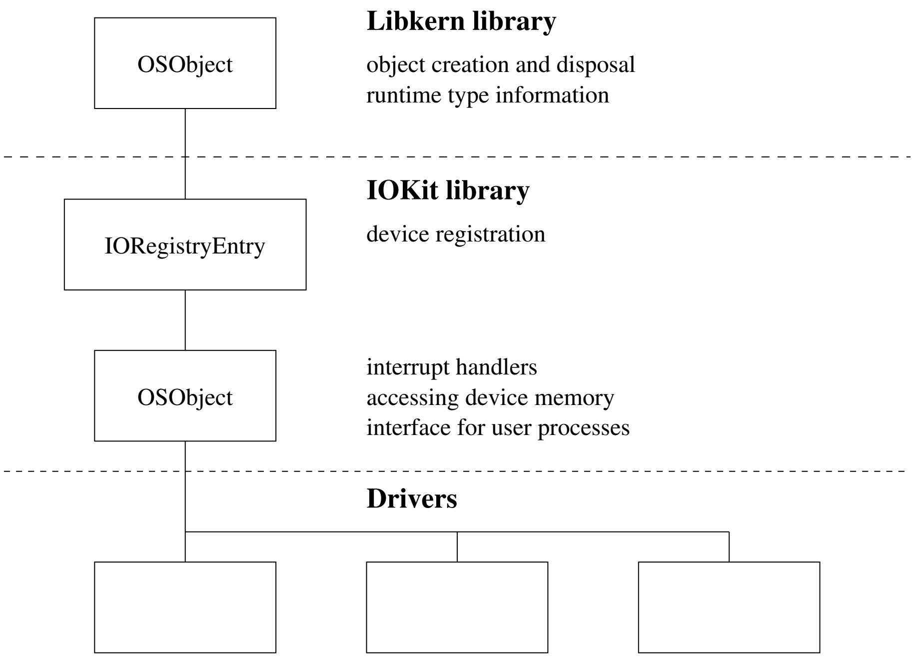

Figure 3.4: IOKit class hierarchy

the VM object are read in if they are not already in memory, then the data to be written is copied into the pages. When a sync operation occurs, the data in the VM object is written out to disk.

The cluster_pagein and cluster_pageout routines are used by the pagein and pageout VFS operations, which are called by the vnode pager. The vnode pager creates a UPL of the pages for which page in or page out is needed. This is passed to the pagein or pageout operation, and to cluster_pagein or cluster_pageout, which obtain the physical addresses of the pages through the UPL, then read or write to it.

3.8 IOKit device drivers

IOKit is the device driver framework used in XNU. IOKit drivers are kernel modules which are dynamically loaded into and linked with the kernel. They are written in Embedded C++ and make use of two class libraries, libkern and IOKit, which provide the base classes for all IOKit classes. These libraries are implemented within the XNU kernel, and are linked dynamically with the device drivers when they are loaded.

The libkern class library contains the OSObject and OSMetaClass classes. OSObject is the root class of all IOKit classes and provides object creation and disposal functionality. OSMetaClass implements a runtime type information system for IOKit classes.

The IOKit class library contains the IORegistryEntry and IOService classes. IORegistryEntry is responsible for device registration. IOService is the base class of all driver objects. The functionality that it provides include registering and controlling interrupt handlers, accessing device memory, and implementing interfaces for user processes.

3.9 Mach usage statistics

Some measurements were conducted on the XNU kernel in Darwin and Mac OS X systems to determine how the Mach component is used. OpenDarwin 6.6.1 and Mac OS X 10.2 were used. In all case the measurement period is from system boot until a shell is available. This period was chosen as calls made during this period represents the necessary functionality that L4-Darwin must provide; if these calls are not implemented then the system will not boot. These numbers provide a rough guide to the porting effort required.

The Mach IPC mechanism is used for communication between user processes as well as Mach and IOKit system calls. The total number of calls to Mach message sending system calls, mach_msg and mach_msg_overwrite, were counted, as well as the number of calls which were determined to be messages to a kernel port. Messages addressed to kernel ports correspond to Mach and IOKit system calls. The results are shown in table 3.1.

	Mac OS X	Darwin
Message calls	102885	29895
Kernel messages	68517	26780

Table 3.1: Number of Mach message sending calls to number of kernel messages

It was found that 90% of the calls to mach_msg and mach_msg_overwrite in Darwin were for kernel messages, while the number for Mac OS X was 67%. This shows that Mach IPC is used for communication between user processes in both Darwin and Mac OS X, and that it is more extensively used in Mac OS X. Since Mac OS X is comprised of Darwin plus the Aqua windowing system and related system libraries, it appears that it is those components which make extensive use of Mach IPC.

It was found that many of the Mach IPC calls are for accessing Mach and IOKit system calls. To determine the distribution of these system calls, the mach_msg_overwrite function was again modified to record the message identifiers of messages delivered to the kernel. These identifiers are generated by MIG and are used to identify the routine which is called. By examining the MIG-generated files, the routines corresponding to the identifiers were found. These routines were then put into categories based on the prefix of their names. The distribution of calls are listed in table 3.2.

The results show that the majority of the calls are to the VM system (31%), the IOKit (16% and 31%), and for manipulating Mach ports (35% and 25%).

The usage of Mach functions by direct function calls from the rest of XNU was then investigated. To do this, the XNU kernel was linked in a way that caused all calls to Mach functions from outside Mach to go through wrapper functions. These wrapper functions were then used to log the calls.

The list of wrapper functions required was determined by taking the intersection of undefined symbols from outside Mach and defined symbols from within Mach. Header files were then parsed to determine the actual functions from within this list and to generate the stubs. There were a total of 228 stubs generated. These are listed in the table on the next page, together with the number of times each one is called during the measurement period.

These numbers show that there are a lot of interactions between Mach and the rest of XNU, and that the interface appears to be not as cleanly defined as that for a pure microkernel-based system, or even for a co-located server.

	Mac OS X	Darwin
clock	2.73%	0.03%
host	5.94%	11.07%
io	30.68%	16.46%
kmod	0.16%	0.35%
mach_port	25.13%	34.63%
processor	0.01%	0.01%
semaphore	0.23%	0.06%
task	2.99%	5.48%
thread	1.12%	1.00%
vm	31.00%	30.91%

Table 3.2: Distribution of Mach and IOKit system calls

Mach function	Count
Assert	0
Debugger	0
_disable_preemption	0
_enable_preemption	0
absolutetime_to_nanoseconds	123
act_machine_get_state	537
act_machine_set_state	531
act_thread_catt	8
act_thread_csave	8
assert_wait	6935
assert_wait_timeout	160
bcopy	48579
bcopy_phys	251
bzero	108830
clear_wait	0
clock_absolutetime_interval_to_deadline	881
clock_get_calendar_value	15260
clock_get_system_value	30
clock_get_uptime	2566
clock_initialize_calendar	0
clock_interval_to_absolutetime_interval	1800
clock_interval_to_deadline	15295
clock_set_calendar_adjtime	0
clock_set_calendar_value	0
cngetc	0
cnputc	0
cnputcusr	4604
conslog_putc	807
convert_port_to_act	257
convert_task_to_port	44
copyin	25864
copyin_shared_file	0
copyinstr	8720
copyout	40713
cpu_number	1866
default_pager_add_file	1
default_pager_backing_store_create	1
default_pager_backing_store_delete	0
delay	1885
device_pager_deallocate	0
find_user_regs	83360
find_user_vec	179
flush_dcache	139
funnel_alloc	0
get_bsdtask_info	452731
get_bsdthread_info	499432
get_task_map	279
get_task_pmap	1601
get_thread_waitresult	2575
get_threadtask	1296
get_user_regs	9

Mach function	Count
host_default_memory_manager	1
host_info	1521
host_priv_self	26
host_reboot	0
host_security_self	19
host_security_set_task_token	19
host_set_exception_ports	1
initialize_screen	1
invalidate_icache	6
io_map	0
ipc_object_copyin	1
ipc_port_copyout_send	44
ipc_port_release_send	19
is_thread_idle	11804
kalloc	195077
kalloc_noblock	797
kernel_memory_allocate	69
kernel_task_create	1
kfree	193959
kmem_alloc	494
kmem_alloc_contig	4
kmem_alloc_pageable	235
kmem_alloc_wired	1
kmem_free	687
kmem_realloc	0
kmem_suballoc	0
kmod_create_fake	1
kmod_destroy	7
kmod_load_extension	21
kmod_lookupbyname_locked	196
kvtophys	124
lock_done	88
lock_write	88
lsf_mapping_pool_gauge	0
mach_make_memory_entry	0
mach_make_memory_entry_64	9
mach_memory_object_memory_entry_64	1
mach_msg_send_from_kernel	0
mach_port_allocate	993
mach_port_deallocate	3643
mach_port_move_member	13
mapping_make	0
mapping_remove	0
memcpy	423810
memory_object_change_attributes	28
memory_object_control_deallocate	28
memory_object_default_deallocate	1
memory_object_destroy	0
memory_object_lock_request	30
memory_object_page_op	33942
memory_object_recover_named	0

Mach function	Count
memory_object_release_name	28
memory_object_upl_request	3246
memset	110093
mutex_alloc	52
mutex_free	8
mutex_lock	218322
mutex_pause	0
mutex_try	0
mutex_unlock	218322
pmap_change_wiring	0
pmap_create	462
pmap_enter	0
pmap_extract	1660
pmap_modify_pages	127
pmap_remove	0
pmap_switch	227
port_name_to_act	0
processor_start	0
scc_putc	0
semaphore_create	17
semaphore_destroy	2
semaphore_signal	330
semaphore_timedwait	1
semaphore_wait	328
set_bsdtask_info	581
shared_region_mapping_dealloc	0
shared_region_mapping_info	237
shared_region_mapping_ref	1
shared_region_object_chain_attach	1
stack_privilege	7
strcat	572
strcmp	2941508
strcpy	17280
strncmp	23110
strncpy	459
swap_act_map	227
swap_task_map	227
switch_to_old_console	0
switch_to_serial_console	0
task_create_local	303
task_deallocate	2593
task_halt	227
task_hold	0
task_importance	0
task_info	560
task_reference_try	44
task_release	0
task_resume	0
task_set_exception_ports	1
task_suspend	278
thread_abort	0
thread_abort_safely	40
thread_block	8067

Mach function	Count
thread_cancel_timer	372
thread_create	303
thread_deallocate	0
thread_enable_fpe	5
thread_exception_return	83637
thread_funnel_get	180194
thread_funnel_merge	0
thread_funnel_set	91501
thread_get_cont_arg	4636
thread_getstatus	0
thread_hold	1
thread_read_times	1
thread_release	1
thread_resume	303
thread_set_cont_arg	4638
thread_set_timer_deadline	373
thread_setstatus	470
thread_sleep_mutex	1
thread_sleep_mutex_deadline	0
thread_sleep_usimple_lock	0
thread_stop	1
thread_terminate	105
thread_unstop	1
thread_wakeup_prim	13763
upl_dirty_page	1877
upl_get_internal_pagelist_offset	1
upl_page_present	14303
upl_phys_page	5024
upl_valid_page	7710
usimple_lock	406031
usimple_lock_init	2670
usimple_unlock	406031
vec_save	179
vm_allocate	3028
vm_behavior_set	0
vm_conflict_check	16996
vm_deallocate	2506
vm_fault	1
vm_get_shared_region	471
vm_inherit	3
vm_map	2669
vm_map_64	2756
vm_map_copy_discard	0
vm_map_copy_overwrite	2217
vm_map_copyin_common	2254
vm_map_copyout	8
vm_map_create	462
vm_map_deallocate	8614
vm_map_get_upl	0
vm_map_page_query	0
vm_map_protect	0
vm_map_region_replace	2
vm_map_remove	0

Mach function	Count
vm_map_switch	16
vm_map_unwire	605
vm_map_wire	607
vm_msync	131
vm_protect	5166
vm_region	688
vm_region_clone	2
vm_region_recurse_64	0
vm_set_shared_region	2
vm_upl_map	1963
vm_upl_unmap	1963
vm_wire	0
vnode_pager_deallocate	423
wait_queue_assert_wait	972
wait_queue_init	33
wait_queue_wakeup_all	1015
zalloc	17148
zalloc_noblock	0
zfree	13632
zinit	0

Chapter 4

Design

4.1 Overview

Because Mach is linked at compile-time with the other components of XNU, it is essentially a monolithic, or "integrated", kernel. To create L4-Darwin, there is the option of porting the entire XNU kernel, including Mach, in a similar way to the porting of L4Linux [8]. Alternatively, the Mach component can be removed, so that L4-Darwin contains the BSD component of XNU without Mach. There are a number of points in support of including Mach:

- If XNU is ported as a monolithic kernel, then most of the changes to XNU will be restricted to the architecture-dependent components, which are already well-defined as XNU supports 2 different architectures. This simplifies the porting process, and reduces the effort required to update L4-Darwin to newer versions of XNU.
- In examining the source code of XNU, it was found that there are many interdependencies between the BSD and Mach components. Therefore it may be difficult to remove the Mach component from XNU.

There are also a number of points in support of not including Mach:

- If Mach is used in L4-Darwin, performance benefits which may result from not using Mach will not be realised. Such a performance benefit was seen in Lites on L3 [10], where there was a measured performance improvement for some cases of emulated Mach IPC over native Mach IPC.
- The implementation of Darwin binary compatibility in NetBSD [4] demonstrates that it is feasible to emulate Mach functionality, at least for the parts of Mach that are used by Darwin user programs.

In the end, it was decided that the Mach component should be removed because the porting of a monolithic kernel to L4 has already been done in L4Linux, and porting the entire XNU kernel would essentially be a repeat of this exercise. In addition, one of the goals of L4-Darwin is to determine whether there are performance benefits from not using Mach, and this goal cannot be achieved if L4-Darwin includes Mach. Therefore L4-Darwin consists of the BSD component of XNU together with some "glue" code which emulates some Mach functionality.

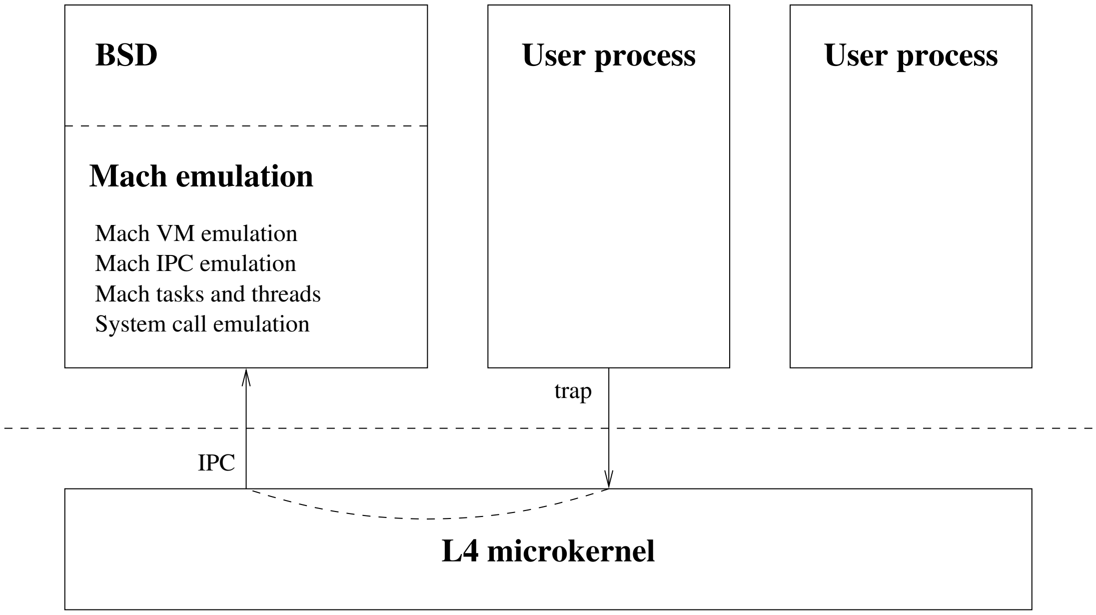

Figure 4.1: L4-Darwin design: system overview

4.2 Single-server vs multi-server

In other ports of monolithic kernels to microkernel systems, such as Mach-UX, Lites, MkLinux and L4Linux, a single-server design was chosen to simplify the porting process. This is because the design of monolithic kernels result in many dependencies within itself, with the result that to separate any one component out as a separate server would require extensive modifications. The few multi-server systems that do exist, for example Mach-US [16], are designed from the beginning to use multiple servers.

Because of the difficulty of splitting up monolithic kernels, it was decided that L4-Darwin is to be a single-server system. It may be possible to put device drivers into a separate server, however this has not been investigated since it is outside the scope of this project.

4.3 Server threads

A major decision in the design of L4-Darwin is whether the L4-Darwin server should should have one thread or multiple threads servicing system calls. A number of issues arise due to the design of the BSD code:

- Most of the BSD code is designed to be executed in the context of a user thread. This means that the BSD code relies on the concept of there being a “current” thread and process. Also, the BSD code expects to be able to block in such a way that other user threads are still able to make system calls and enter the BSD code. Since significant modifications to the BSD code are to be avoided, the chosen design for the server must support blocking in the BSD code.
- Most of the BSD code is not reentrant. Therefore if the L4-Darwin server is to be multithreaded,

there needs to be locking mechanisms to prevent the BSD code from executing in more than one place at the same time. However, such a locking mechanism exists already in XNU, in the form of *funnels*, and locking/unlocking calls already exist in the BSD code. If the L4-Darwin server is to be multithreaded, implementing the funnel locking mechanism will solve the reentrancy problem.

The options available are:

- *Single L4-Darwin server thread.* This has the advantage of being simple, but there needs to be some way of allowing the BSD code to block without stopping the system call loop. A solution to this is to use some kind of user-level threading technique. This is the approach taken by L4Linux [8], which uses a single server thread to handle system calls and page faults, and multiplexes this thread to avoid blocking in the L4Linux server.

  A problem with this approach is that on multiprocessor systems, this essentially restricts the L4-Darwin server to run on a single processor. This limits the scalability of L4-Darwin.

- *One L4-Darwin server thread per user thread.* The system call handler and pager for each user thread is set to its corresponding server thread. This has the advantage of mapping cleanly to the design of the BSD code. Blocking can be implemented in a straightforward manner by pausing the thread. The concept of a "current" thread is also easily implemented as each kernel thread corresponds to a single user thread. An advantage of this approach is better scalability compared to using a single server thread in the L4-Darwin server. However, the drawbacks of this method is that it essentially doubles the kernel resource usage for each user thread, as a server thread (which may be rarely used) needs to be created for every user thread.

- *Pool of L4-Darwin server threads.* With this option, a number of server threads are used to service system calls from user threads. A server thread can be allowed to block when handling a system call since other threads are available to receive system calls. The concept of "current thread" can also be implemented by simply setting a "current thread" variable to the thread that the system call was received from. The problem with this approach is that L4 IPC addressing is by thread ID, and a user thread has no way of knowing which server thread is available to receive system calls. Some kind of indirection is needed, possibly by having a "master" thread receive IPC messages, which are then passed to "worker" threads that do the actual work. However, this indirection step is likely to slow down system calls significantly, as an extra IPC is needed.

  This approach is similar to that used by most Mach-based systems. However, this is because Mach IPC uses ports for IPC addressing, and Mach provides a mechanism for multiple threads to wait for messages on the same port or set of ports. L4 does not provide such a mechanism.

It was decided that L4-Darwin is to use a single system call thread, since this was the approach successfully used in L4Linux. Using a pool of threads is problematic because of the need to have a indirection mechanism, and using a server thread per user thread is wasteful of resources.

4.4 System calls

There are BSD and Mach system calls in Darwin. BSD system calls, as well as some Mach system calls occur through the traditional system trap mechanism. A large number of Mach system calls go through an RPC mechanism over Mach IPC. These are discussed in section 4.5.

L4 X.2 provides a no-emulator system call mechanism similar to that provided by the OSF version of Mach, used by OSF/1 MK and MkLinux. The L4 kernel delivers system call traps and exceptions to a designated exception handler thread as IPC messages. Since this is the facility provided in L4 X.2 for system call emulation, L4-Darwin has to use a no-emulator design. This is different from the design used by L4Linux, which uses a different version of L4 that provides a trampoline mechanism instead.

The two types of system calls are distinguished by their system call numbers: BSD system calls have positive numbers, while Mach system calls have negative numbers. The BSD system calls are implemented in the BSD component. However, the Mach system calls are implemented in the Mach component, which has been removed, and so these system calls need to be emulated.

4.5 Mach IPC emulation

Mach IPC is used for accessing Mach and IOKit system calls, and for communication between user processes. From the system call counts (see section 3.9), it is clear that Mach IPC is used extensively by Darwin programs to access kernel system calls, and to an even greater extent by Mac OS X programs for communication between different user processes. Therefore it is necessary for L4-Darwin to emulate Mach IPC. There are two types of IPC that need to be emulated: IPCs to the Mach kernel, used for Mach and IOKit system calls, and IPCs between user processes.

One of the key features of Mach IPC is the use of ports as the addressing mechanism. Ports are used to identify kernel objects such as tasks and threads. This is often used in conjunction with system calls over Mach IPC; System call IPCs are sent to ports representing kernel objects, although not all such ports are for receiving system call IPCs. Ports are visible to user tasks as *port rights*, which represent the right to send or receive on a port. Port rights can be transferred between user processes via IPC. Both the use of ports as kernel object handles, as well as the port right transfer mechanism, need to be emulated in L4-Darwin.

Mach IPC emulation can be implemented in a central server which acts as a hub for all IPC messages, or in a decentralised fashion by using an emulation library and sending messages directly between user processes. Lites on L3 [10] emulates Mach IPC, including the port right transfer mechanism, in a decentralised fashion using a Mach emulation library. It was found that while message sending operations are fast, the port right transfer mechanism was complicated and hard to implement, and only the transfer of send rights were implemented.

The advantage of implementing Mach IPC in a decentralised fashion is that a Mach IPC message can be implemented with a single IPC operation of the underlying microkernel. However, this requires an emulation library to reside in the user task, which is not possible in L4-Darwin because of the system call emulation mechanism used. Therefore it was decided that Mach IPC emulation is to be implemented centrally in the L4-Darwin server. This has the additional advantage that it simplifies implementation, since all port operations, including port right transfers, can be performed by manipulating data structures within the server.

4.5.1 System calls over Mach IPC

A user process makes a system call over Mach IPC by constructing a message and then calling the mach_msg or mach_msg_overwrite system trap. Since L4-Darwin is a single server, these can be emulated within the L4-Darwin implementation of mach_msg or mach_msg_overwrite.

4.6 Virtual memory management

The BSD component of XNU has no virtual memory management (VM) capability of its own, but relies on the Mach VM for operations such as allocating and deallocating memory, manipulating address spaces, and memory mapping of files and devices. Darwin system libraries also rely on Mach VM calls to allocate memory. Because of this extensive reliance on the Mach VM, it was necessary to either port or emulate the Mach VM for L4-Darwin. There are a number of issues which are a factor in the decision to port or emulate:

- There are many interdependencies between the Mach VM subsystem and the rest of Mach, which makes the Mach VM subsystem difficult to separate out and port to L4.
- The Mach VM is designed to be machine independent, and has a machine dependent component, pmap, which manipulates hardware page mappings. The functionality of the pmap component almost maps cleanly to L4 address maps. The main problem with mapping pmap functionality to L4 is that on L4, new page mappings can only be added in a reply to a page fault message, or when the user process requests it, but in pmap, it is expected that new page mappings can be added at any time.
- The Mach VM system is large and complicated. This argument goes in both directions, as a large and complicated block of code is difficult to port, and also difficult to emulate correctly.
- There is likely to be some amount of redundancy in the Mach VM code, since some features have been disabled and are no longer used. This means that an emulation of the Mach VM is potentially simpler and smaller than the original.
- Basic Mach VM calls are well documented, however the extensions added by Apple are poorly documented and many calls are not documented at all. This complicates the emulation of the VM system.

An early attempt was made to port the Mach VM, however this attempt was aborted when little progress in resolving the dependencies was made after 2 weeks. It is possible that this failure was due to an insufficient understanding of the code, however there was no evidence to indicate that another attempt will succeed, so the decision was made to emulate the Mach VM. It is not clear that this is the correct decision, and this may be open to change in the future.

The L4-Darwin VM system runs within the L4-Darwin server rather than as a separate server. The reasons for this design decision were:

- The BSD component appears to access some VM data structures directly, such as the upl_page_info_t array in a upl.
- In XNU the Mach VM system is linked to the BSD component and VM calls are direct function calls. Doing the same for the L4-Darwin server simplifies the porting process.

- There are many calls between the BSD component and VM system. These calls occur in both directions, and many of them are undocumented. Implementing the VM system as a separate server would require working out the function of these calls in detail, and implementing RPC stubs for every call.
- Even if the L4-Darwin VM system runs as a separate server, it will not receive direct calls from user processes since user processes make Mach VM calls via Mach IPC, which use the Mach IPC system call trap, which has to be handled by the L4-Darwin server.

  However, the system libraries can be modified to make VM calls direct to the VM server instead of through Mach IPC.
- The unified buffer cache (UBC) implementation is split between the Mach VM and the BSD components, and there are frequent calls between them (see section 3.7). Therefore splitting the code into 2 servers will result in a significant performance penalty unless some changes are made to the UBC design.

4.7 Tasks and threads

XNU implements BSD processes and threads on top of Mach tasks and threads. For L4-Darwin, Mach task and thread functionality needs to be emulated. The straightforward design for this is to map Mach tasks one-to-one to L4 address spaces, and map Mach threads one-to-one to L4 threads.

4.8 Signals

Signal delivery in XNU involves modifying the state of a user thread directly. This is difficult to do in L4, because the L4 API does not permit the modification of the state of a thread in another address space except for special instances such as in the reply to a system call or exception message. Therefore it is unavoidable to have a signal delivery thread in the same address space as the user thread receiving the signal. This thread communicates with the L4-Darwin server using IPC, and modifies the state of the target thread when a system call IPC is received. This is essentially the same signal delivery mechanism used in L4-Linux.

This mechanism does make it possible for a user thread to prevent receipt of its own signals. However, this is not a problem as the thread can achieve the same effect by disabling its signal handlers via BSD system calls. Signals with external effects, such as SIGKILL, are unaffected since they can be implemented in the L4-Darwin server.

4.9 Device drivers

Device drivers are outside the scope of this project, since there is another project investigating the issue of providing drivers for L4-Darwin. It is intended that L4-Darwin eventually support IOKit drivers in some form.

L4Linux uses the L4 interrupt delivery mechanism to allow Linux drivers to be used without change. This suggests that it should be possible for L4-Darwin to support IOKit device drivers in a similar way.

4.10 Unified buffer cache

In XNU, the UBC implementation is split between the BSD component and the Mach component. To support the UBC on L4-Darwin, it is enough to emulate the parts of the UBC which are part of the Mach VM system in XNU. These parts include the UPL functions in the VM system used for manipulating the resident pages of a vm_object, and the vnode pager, which is a wrapper for UBC operations implemented in the BSD component. See section 3.7 for details.

4.11 Analysis of design

4.11.1 Shortcomings

A major shortcoming of this system is that it uses only one L4-Darwin server thread. This limits the scalability of L4-Darwin on multi-processor machines, since the L4-Darwin server can only run on one processor at a time. It is possible that this has little effect on the scalability of L4-Darwin, since the BSD component is already governed by two global locks which restricts BSD to run on at most two processors at once: network code on one processor and non-network code on the other processor. On a single processor system, this should be more efficient than using multiple L4 threads for the server, since user thread switches don't require trapping into the kernel.

4.11.2 Performance

Since the system call handlers within XNU and L4-Darwin are very similar, the performance of BSD system calls in L4-Darwin is dependent on the performance of the system call emulation code in L4. However, the performance of the PowerPC system call emulation code in the L4Ka::Pistachio kernel has not been measured before.

It is difficult to estimate the performance of Mach IPC and Mach system calls, since these components will be implemented from scratch, however it is hoped that some performance improvement will result from simplified implementation of these features.

Chapter 5

Implementation

This chapter covers the details of the implementation of L4-Darwin.

5.1 Boot process

The L4-Darwin server is loaded as the root task by the boot loader. A single L4-Darwin server thread is created at startup, which becomes the system call handler and pager for all L4-Darwin user threads after initialisation is complete. The startup procedure, from the start of first L4-Darwin server thread to the start of the init process, is as follows:

1. The L4 environment library is initialised, and malloc is initialised with a statically allocated heap.
2. The Mach memory management emulation is initialised. This sets up the free list, then requests pages from σ_0 and populates the free list with the pages received. The initialisation code also reserves a special page for user process startup code (see section 5.2).
3. The thread and task structures are initialised. This sets up the list of all running threads, as well as the static thread and task structures for the kernel task.
4. The bsd_init function in the BSD component is called, which initialises all BSD data structures. It sets up the static structures for the current process, which becomes process 0, the kernel process.
5. The bsd_init function finds and mounts the root file system, as well as the devfs filesystem.
6. The init process is created by cloning the kernel process. This creates a new BSD process, as well as a new Mach task and new Mach thread within the task. The Mach emulation code creates a corresponding L4 address space and L4 thread. To set up the init process, act_set_astbsd is called on the newly created thread. This sets up the thread so that bsd_ast is called when the thread is about to run.
7. The thread is started, and bsd_ast gets called. The current process is now the init process. In L4-Darwin, this is emulated with no actual thread switch by calling bsd_ast directly from thread_resume, after changing the current thread pointer. The bsd_ast function detects (via a flag in the kernel) that the init process has not been started yet, and so assumes that the current process is the init process. The bsd_ast function then calls bsdinit_task to set up the address space of the init process.

8. The init program is loaded from /sbin/mach_init on the root filesystem, using a call to execve. The Mach-O loader sets up mappings in the address space, and also sets up the entry point and the stack pointer. In XNU, this is stored in the saved registers for the thread. This is emulated in L4-Darwin by storing the entry point and stack pointer in fields defined in the thread structure, and having the thread_resume function start the thread at the stored entry point with the stored stack pointer.

9. The bsdinit_task function creates stdin, stdout, and stderr file descriptors by opening /dev/console. In Darwin these are created in mach_init, however creating these file descriptors here allows testing without having to run mach_init.

10. The thread in the init process is started. The bsd_init function returns, and the L4-Darwin server thread enters a message loop.

5.2 System calls

In L4-Darwin, system call traps from Darwin binaries are handled by the L4 system call emulation mechanism, which catches system call traps made by the user task and dispatches it to a system call handler thread via L4 IPC.

To do this, the exception handler field in the user thread's UTCB is set to point to the system call handler thread. A problem arises because L4 does not provide a means for a thread outside the target thread's address space to set the exception handler. To solve this, a special page of initialisation code is used, which is mapped into the address space of every user task. This page is set to be the entry point for the user thread, and the real entry point is placed at the top of the user stack. The initialisation code sets the exception handler, then pops the real entry point address off the stack and jumps there.

An issue which arose in the system call emulation is that the user libraries in Darwin expect the link register (LR), which is used for storing the return address in a function call, to be preserved over a system call instruction. The then current version of Pistachio/PPC did not do this, which resulted in some unusual behaviour when a function attempted to return after a system call.

```
_write:
        li      r0,0x4
        sc
        b       1f
        b       2f
1:      b       cerror
2:      nop
        blr
```

Figure 5.1: Typical system call in the Darwin C library

Because of the way XNU is structured, there is the concept of a "current thread" when executing in the kernel. This is used by functions such as copyin and copyout to determine the target thread or task of the operation. Within system call handlers, it is assumed that the current thread is the thread where the system call was made. This is emulated in the L4-Darwin server by setting the "current thread" to be the thread that the last system call was received from.

5.3 Server threads

The user-level threading mechanism needed for multiplexing the single L4-Darwin server thread is not yet implemented.

5.3.1 BSD traps

BSD system calls occur through system call traps. A user process makes a BSD system call by executing a system call trap instruction. There are two “flavors” of system calls, one with the system call number in register R0 and parameters in R3 to R10, and the other with 0 in R0, the system call number in R3, and parameters in R4 to R10. BSD system call numbers are positive. The return value of BSD system calls are returned in registers R3 and R4.

An unusual “feature” of the BSD system calls in XNU is that the instruction pointer is incremented by 4 if the system call is successful. This is because in the Darwin user libraries, a call to cerror always immediately follows a system call instruction, and such a call is unnecessary if the system call is successful. This unusual behaviour is emulated in L4-Darwin.

5.3.2 Mach traps

Mach system call traps are distinguished from BSD system call traps by the use of negative system call numbers. A user process makes a Mach system call by loading the system call number in register R0, then issuing a system trap instruction. The first 8 parameters are passed in the registers R3 to R10. However, Mach system calls can have up to 9 parameters; the 9th parameter is passed on the user stack. The result of Mach system calls are returned in the register R3.

In L4-Darwin the calls host_self, thread_self, task_self, mach_reply_port, mach_msg_overwrite, and mach_msg are emulated so that basic system calls over Mach IPC are supported. The remainder of Mach system calls are not yet implemented.

There are a number of special “fast” system calls. L4-Darwin emulates 2 of these system calls, pthread_set_self and pthread_self. These system calls respectively store and retrieve a user data pointer for the current thread, stored in the thread data structure.

5.3.3 System calls via Mach IPC

A large number of system calls occur through Mach IPC to a kernel port. From the perspective of the user process, these calls are remote procedure calls (RPC) using stub code generated by the Mach interface generator (MIG). The MIG-generated stubs construct a message containing the parameters of the system call, and calls the mach_msg_overwrite trap to send the message and wait for a reply. The reply message contains the results of the system call.

L4-Darwin emulates these messages by catching messages to kernel ports in mach_msg_overwrite trap. The message is dispatched it to the appropriate function according to the ID field, which then extracts the parameters from the message, handles the system call, and constructs a reply. The reply message is then copied back to the user process.

The format of Mach system call messages are defined by the MIG. In the system call request message, the system call is identified by the ID field in the Mach message header. The body of the message consists of an 8-byte MIG header, followed by the parameters of the system call. In the system call reply message, the ID field contains the number of the system call plus 100. The body of the message consists of the MIG header followed by the results of the system call.

Some system calls may have more than one result value. For example, the vm_allocate system call returns both an error code and the address of the allocated block. The MIG-generated stub supports this using pointers; when calling the stub, the caller passes a pointer to a variable; upon return the variable is filled with the result value. These pointers are handled by the stub and are not passed over the RPC messages. The result value is returned in the reply message and the stub copies it to the appropriate location in memory.

The system calls that are emulated are vm_allocate, which calls vm_allocate in the VM emulation to allocate memory for the user task, and host_page_size, which returns the page size (4k).

5.4 Mach IPC emulation

A full emulation of Mach ports is not yet complete. The calls which return port rights, namely host_self, thread_self and task_self, were made to return a constant value. This allowed the emulation of the basic system calls which occur over Mach IPC, since for such calls the assumption that the call is for the calling thread is often correct.

Mach IPC between different user processes is not yet implemented, since it relies on a functioning port emulation.

5.5 VM implementation

The implementation of the Mach VM emulation in L4-Darwin is based on the design and structure of the actual Mach VM.

5.5.1 Emulated data structures and calls

The data structures used in the emulation code are roughly similar to that which is used in Mach.

A *memory_object* represents a memory object, and has a pager associated with it which has calls that can be used to obtain or write out data for the memory object. A memory_object that is currently cached in memory also contains a pointer to a vm_object.

A *memory_object_control* in Mach is originally a control port for a memory object. In XNU, memory_object_control appears to be used as a reference to a memory object, and is used as a parameter in some VM calls. In L4-Darwin, this is emulated by making a memory_object_control the same as a memory_object.

A *vm_map* represents an address space. Mapped regions in the address space are represented by vm_map_entry structures, which are maintained in a doubly-linked list sorted in order of virtual address.

Each vm_map_entry structure contains a pointer to a vm_object and a 64-bit offset, as well as protection flags which specify whether the region is mapped with read, write, or execute permissions. The vm_object pointer may be null, if the region is a newly created region of anonymous memory.

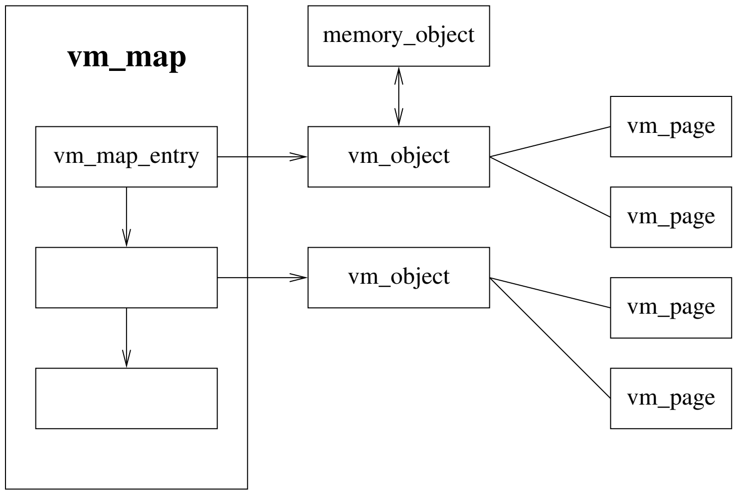

Figure 5.2: Example of a vm_map structure

A *vm_object* structure represents an in-memory cache of a memory object. It contains a pointer to a memory_object as well as a list of physical pages which are allocated to the vm_object. The page list is stored as a list of vm_page structures.

A *vm_page* structure is created for every physical page managed by the VM system. A vm_page object can be allocated to a vm_object. If it is, it contains a pointer to the vm_object and an offset into the vm_object. Each vm_page structure also contains a number of flags: a flag that marks whether the page is currently being paged in, and a flag that marks whether the page has been modified.

In figure 5.2 there is an example of a vm_map data structure with 3 regions. The first region maps a memory object and has some pages allocated to it. The second region is a region of anonymous memory, which also has some pages allocated to it. The third region is also a region of anonymous memory, however this region has not been used and so the vm_object for this region has not been created.

The *upl* structure is a list of physical pages for a contiguous range of offsets in a vm_object. It contains a pointer and an offset into a vm_object, as well as the list of the physical pages. The page list is visible from outside the L4-Darwin VM system, and so it needs to have the same format as the upl page list in XNU.

A *upl_page_info_t* is an entry in the list of physical pages in a upl structure, and contains the physical address of a page as well as flags that mark whether it is absent or dirty. This structure is visible from outside the VM system, and so the upl_page_info_t structure in L4-Darwin is identical to the upl_page_info_t in XNU.

The VM system keeps track of free pages in a linked list of vm_page structures.

In a page fault, the VM system needs to look up pages based on a vm_object and offset. For efficient lookup, a hash table is used. The hash table is indexed by a hash of the vm_object pointer and offset, and each hash bucket contains a linked list of vm_page structures. The number of hash buckets is chosen to be the smallest power of 2 greater than the total number of pages.

Copyin and out

The *copyin*, *copyout*, *copyinstr* and *copyoutstr* functions are used extensively by the BSD component to copy data in to and out of user address spaces. These functions rely on the existence of a "current" task: they do not take an address space as a parameter, but rather the target space is determined using the current_map function, which returns the address map of the current task. The internal function vm_fault is then used to look up the virtual-to-physical mappings of the address space.

Address map functions

The *vm_map* function is used to map a memory object into an address spaces. If the anywhere flag is set, the object may be mapped anywhere; otherwise the object is mapped at a specified address. The offset into the memory object is 32-bit for compatibility with code which expects the offset to be 32 bits. For 64-bit offsets, the *vm_map_64* function, which is identical to vm_map except that the offset is 64-bit, is used.

The *vm_allocate* function allocates a block of anonymous memory with a specified size. If the anywhere flag is not set, the address is taken as a hint of where to map the newly allocated block of memory. In L4-Darwin, this function creates a vm_map_entry with a null vm_object pointer, to allow for lazy creation of vm_objects.

The *vm_map_create* function creates a new vm_map. A parent vm_map can be specified, which causes the new map to inherit the address space of the parent vm_map. The inheritance feature is not yet implemented.

Memory object functions

From inspection of the XNU source, the *memory_object_create_named* function in XNU appears to be used for creating a memory_object_control for a memory object. This is emulated in L4-Darwin by simply returning the memory_object pointer as a memory_object_control pointer.

UPL functions

The *memory_object_upl_request* function is used for creating a upl for a region of the vm_object corresponding to a memory object. This function can operate in two modes, 'copy in' and 'copy out'. Copy in mode is used for paging in; pages that are within the range but are not allocated are allocated and marked as currently being paged in. Copy out mode is used for paging out; pages that are within the range but are not allocated are left unallocated, and is marked as invalid in the list. A call to memory_object_upl_request marks the beginning of a paging operation.

The *upl_commit_range* function is used to signal the end of a paging operation. If it is a page in operation, the flag marking pages as currently being paged in is cleared.

The *upl_phys_page*, *upl_page_present*, *upl_valid_page*, and *upl_dirty_page* functions are used to access the contents of an upl structure.

Page faults

The L4-Darwin server thread is specified as the pager thread for all L4-Darwin user threads. When a page fault message is received, the vm_fault function in the VM system is called to look up the physical page, and a reply message is sent.

The vm_fault function is given a vm_map structure and a virtual address to be looked up. It scans through the list of regions and locates the region which the virtual address falls in, then uses the object/offset hash table to look up the page.

The vm_map structure consists of a sorted linked list of vm_map_entry structures. The main reason for choosing this data structure is for simplicity, as it is easy to implement, and because it is similar to the data structure used in Mach. Since an address space usually contains a small number of regions, a linear search through the list should be sufficient. In Mach, this search is optimised by storing a hint pointing to the last region found; the search starts from the region that the hint points to. According to the principle of locality, subsequent page faults are likely to occur in the same region, so this method is effective in speeding up page faults lookups. This technique is not currently used in L4-Darwin, however it should be easy to implement in the future.

The object/offset hash table used is essentially the same structure as that used in Mach. This hash table duplicates the data stored in the page list of a vm_object. The purpose of the hash table is to allow fast lookups of pages by vm_object and offset, which is commonly required on page faults.

If the virtual address falls within a region in the address space, but the vm_object does not contain the page, then the page needs to be paged in. If the region is anonymous memory, then a page is allocated and zero-filled. If the region maps a memory object, the memory_object_data_request function is called to page in the required region. This function allocates the page and fills it with the required data.

Pager interface

The *memory_object_data_request* function is used to page in a region of memory in a memory object. This function is a wrapper around similarly named functions in the pagers. For example, if the memory object is a vnode object, then the memory_object_data_request function calls the vnode_pager_data_request function, which then does the actual paging in.

A typical data_request function in a pager is called with a memory object with a range within the memory object specified. The function then calls the memory_object_upl_request function to create a upl. This allocates pages within the range which are not yet allocated, and signal the beginning of paging for these pages. The data_request function then fills in the pages with data. When this is done, the function calls a upl_commit function to signal the end of paging if the operation completed successfully, or upl_abort if the operation failed and the pages have not been filled in with data.

The *memory_object_data_return* function is used to page out a region of memory in a memory object. This function is also a wrapper around functions in the pagers.

5.6 Tasks and threads

5.6.1 Emulated data structures and calls

A *task* structure represents a task, which has an address space, port rights, as well as a number of threads. To emulate this in L4-Darwin, a task structure contains a L4 space identifier, and a pointer to a vm_map. The Mach task structure in XNU also contains a pointer, used by the BSD component to keep track of which BSD process corresponds to a Mach task. This pointer is also stored in the L4-Darwin task structure.

The *task_create_local* function creates a new task. In L4-Darwin, this function allocates and initialises a new L4 space identifier, creates a new vm_map, and allocates and initialises a new task structure. This function should duplicate the address space of the parent task if requested to do so, using vm_map_create, however vm_map_create does not currently implement this functionality.

A *thread* structure represents a thread, which belongs to a particular task. In XNU, the Mach thread structure is split into separate *shuttle* and *activation* structures to support migrating thread IPC (see section 3.3.1). Since L4-Darwin is to emulate Mach IPC over L4 IPC, this split is no longer needed, and so in L4-Darwin the thread and activation structures are merged into one structure.

The L4-Darwin thread structure contains a L4 thread identifier. Like the task structure, a pointer is also present in the thread structure to maintain the mapping between Mach threads and BSD threads. The thread structure also contains a pointer to a task structure, for keeping track of which task this thread belongs to.

The Mach thread structure in XNU contains a user data pointer, which is used by the system pthread library. The emulated thread structure also contains a field to hold this user data pointer, which is accessed using the functions *pthread_self* and *pthread_set_self*.

The *thread_setuserstack* and *thread_setentrypoint* is used by the BSD component to set the initial stack pointer and entry point of a newly-created thread. In XNU this is implemented by directly modifying the saved registers of the thread, setting the instruction and stack pointers to the requested values. This is not possible in L4 since the instruction and stack pointers are set in the message sent to start the thread. To emulate the XNU behaviour, the instruction and stack pointers are saved in fields of the thread structure, which are then sent to the thread when it is started in the thread_resume call.

The *act_set_astbsd* function is used to set a flag on a thread, such that when the thread is about to run, the *bsd_ast* function in the BSD component is called. This is used by the BSD component to deliver Unix signals, as well as to initialise the init process when it is first run. This is emulated in L4-Darwin by adding a field to the thread structure, and modifying thread_resume to call bsd_ast if the field is set. This does not emulate the XNU behaviour completely, but it is sufficient for the init process to be initialised.

The *thread_create* function is used to create a thread. In L4-Darwin, this function allocates and initialises a new L4 thread, and allocates and initialises a thread structure. It also calls the *uthread_create* function located in the BSD code to initialise the Unix thread structure associated with this thread.

The *thread_resume* function in used to start a newly created thread, or restart a paused thread. In L4-Darwin, only the functionality for starting a new thread is implemented. It sets the current thread to be the resuming thread, calls bsd_ast if necessary, then sends the message which starts the thread.

To determine which thread a system call message is from, the L4-Darwin system call handler needs to be able to look up threads by their L4 thread identifier. This is done in L4-Darwin by keeping track of the thread structures in a linked list, so that a look up can be done by scanning through the list. This is suboptimal as a lookup is done on every system call and page fault, and should be replaced in the future with a hash table.

5.7 Unified buffer cache

The *cluster_read* operation, a UBC function, requires a special modification for reads from kernel space to work. The cluster_read function is implemented such that in some situations it performs the read by mapping a portion of the memory object into the kernel address space, then copying the memory from the mapped region. This is not possible in L4-Darwin because the L4-Darwin server does not have a pager which can perform these mappings. To get around this, the cluster_read function is modified to copy page-by-page from its physical address, without mapping the memory object.

5.7.1 The vnode pager

In L4-Darwin, the *vnode_pager* structure is a memory_object with an extra field, a pointer to the vnode for which it is a cache for.

The *vnode_pager_setup* function is used to create a new memory object backed by a file, represented by a vnode. In L4-Darwin this function is implemented by allocating a new vnode_pager structure, filling in the fields of the structure and returning it as a memory_object.

The *vnode_pager_data_request* and *vnode_pager_data_return* functions are respectively used to page in and page out memory objects backed by the vnode pager. These functions are wrappers around calls to vnode_pagein and vnode_pageout on the vnode, which are UBC functions located in the BSD component.

5.8 Mach-O loader

A Mach-O loader is present in the BSD component for loading Darwin user programs and libraries, all of which use the Mach-O format. A Mach-O file contains a header followed by *load commands*, which specify the layout of the file in memory as well as the state of the initial thread and information for dynamic linking. The Mach-O loader reads in the header and the load commands of the Mach-O file. According to the load commands, the Mach-O loader then sets up memory mappings for each segment, and sets the initial state of the first thread.

In L4-Darwin, the mapping operations use the emulated vnode pager (see section 5.7.1) and VM mapping (see section 5.5) operations.

The thread states stored in the Mach-O file include the contents of every register and not only the instruction and stack pointers. L4 does not support setting the initial state of a thread apart from setting the instruction and stack pointers, and so in L4-Darwin the state of the registers are ignored and only the instruction and stack pointers are loaded. This does not seem to affect the functionality of Darwin programs; it appears that Darwin does not actually make use of this feature of the Mach-O format.

However, the initial thread state is used to specify the entry point and initial stack pointer of the user program. The Mach-O loader calls the thread_setentrypoint and thread_setuserstack functions (see section 5.6) to set the entry point and stack pointer.

Shared library support is split between the Mach VM and the BSD code. The Mach VM part of the shared library code is undocumented and has not been implemented in L4-Darwin, and so L4-Darwin does not currently support shared libraries.

5.9 Filesystems

For the BSD boot process to complete successfully, a block device with a filesystem is needed for mounting the root filesystem and loading an init process. However, at the moment there are no real device drivers in L4-Darwin, so there are no block devices. To provide a filesystem for L4-Darwin development before real device drivers are available, a RAM disk block device was created. This block device treats a fixed region of memory as a RAM disk, and reads and writes to the block device are implemented as memory copies to and from that region. The filesystem image was placed into memory by incorporating it into the ELF image which is loaded into memory on boot. See section A for more details.

XNU supports a number of filesystems, including the BSD UFS filesystem. The UFS filesystem format was chosen for the image as there is a tool available for making UFS filesystem images.

XNU provides a devfs filesystem for device files in /dev. A devfs filesystem is a virtual filesystem in which device files appear when devices are loaded and registered. By default, the devfs filesystem is mounted as part of the boot process. The default mach_init in Darwin creates stdin, stdout and stderr by opening /dev/console, which is the device file for the console. The devfs filesystem was implemented in L4-Darwin so that stdin, stdout and stderr could be created.

Chapter 6

Results

6.1 Qualitative results

Currently, L4-Darwin is able to run some simple command-line programs. The following features of L4-Darwin are functional:

- UFS filesystems and devfs filesystems can be mounted.
- Loading of statically linked Mach-O binaries from a filesystem.
- The system call emulation mechanism is functional: the L4-Darwin server can receive and respond to system calls made by user programs.
- Character devices such as the console can be opened and written to.
- Some Mach system calls work: programs are able to call vm_allocate to allocate memory.
- Basic command-line programs statically linked against the C library are able to run. The Darwin C library initialisation code completes successfully, and most basic C library functions, such as printf, work.

There are a number of things which L4-Darwin does not yet do. The most important ones are:

- Forking of user processes is not supported.
- Most Mach system calls are not implemented. Most importantly, this includes the Mach IPC mechanism.
- L4-Darwin does not have any device drivers and so has no input or output mechanism apart from the L4 debug console.

```
        lis     r13, 0x100
loop:
        li      r0,0x14
        sc
        nop
        addic.  r13, r13, -1
        bne+    cr0, loop
```

Figure 6.1: The getpid benchmark loop

6.2 Benchmarks

A major problem with first-generation microkernel-based systems is the system call overhead imposed by the IPC mechanism. To measure the system call overhead, the "null" system call getpid was called a large number of times in a tight loop.

It was found that the Darwin C library caches the result of getpid, so to ensure that a system call is actually made in every loop iteration, the test program was coded in assembly. The test program was written to perform the getpid system call 2^{24} times in a loop and then exit. Both tests runs were performed on an Apple iMac with a PowerPC G3 processor running at 500MHz, a 512KB L2 cache, and 384MB of main memory. The tests were repeated 3 times and the results were averaged.

	Total time	Time per iteration
XNU	27.9s (1%)	1.66μs
L4-Darwin	23.0s (1%)	1.37μs

Table 6.1: Timings for getpid system call (standard deviation in parenthesis)

The results are shown in table 6.1. These results are somewhat surprising because since L4-Darwin imposes extra IPC overhead on the system call path, it should be expected that the L4-Darwin result is slower. However, this is not the case.

The special "fast" trap pthread_self was also tested. Again, a test program that makes the system call 2^{24} times was used. The results are shown in table 6.2. For comparison purposes, the system call handler code was modified so that it immediately returns a reply without performing any processing, and measurements were again performed; the results of this is shown as "null call" in the table.

	Total time	Time per iteration
XNU	1.6s (1%)	0.10μs
L4-Darwin	17.9s (1%)	1.07μs
L4-Darwin (null call)	15.0s (1%)	0.89μs

Table 6.2: Timings for pthread_self system call (standard deviation in parenthesis)

6.3 Analysis

To understand the reasons for the getpid result, the BSD system call handler functions in L4-Darwin and XNU were examined. The handler function was re-implemented in L4-Darwin since different methods are needed to access the user registers: L4-Darwin extracts user register contents from the L4_Msg_t structure with the system call message, while XNU reads them from the savearea structure which has saved user context. Apart from this, the handlers perform essentially the same function: both handlers store the system call parameters into the uthread structure, looks up the system call routine in the sysent table, then calls the routine. The major difference between the two is that XNU performs locking but L4-Darwin does not.

The extra locking in XNU may be sufficient to account for the difference in performance, however this is difficult to test since the locking routines in L4-Darwin are not implemented, and the locking mechanisms in XNU cannot be easily disabled.

In XNU the pthread_self trap is handled early in the system trap handler before full user context is saved. When L4 system call emulation is used, it is unavoidable that the full user context is saved. This is because the no-emulator design used requires L4 to save all user context and place user registers into a message before the system call handler can be triggered. The "null call" gives an estimation of the cost of saving user context and marshalling user registers into a message; it can be seen that this is much greater than the cost of the pthread_self trap in XNU. This shows that there are limits to the performance that can be achieved using a no-emulator design: significant performance loss for some system calls are unavoidable because of issues inherent in system call emulation.

Chapter 7

Conclusion

Although the L4-Darwin port is not fully functional, the preliminary results are promising. The benchmark results show that L4-Darwin is able to match XNU on system call overhead for normal BSD system calls, at least in the case of a single-processor system. The question of whether better performance can be achieved by removing Mach is still open, since a substantial amount of the Mach emulation has not been implemented.

7.1 Future work

There is much work that needs to be done before L4-Darwin can be used in a Darwin or Mac OS X system.

- *Forking and multiple user processes.* At the moment L4-Darwin only supports a single user process. To implement forking, there needs to be a way of setting the register state of the newly created thread. There is no interface in the L4 API to do this, so there will need to be some initialisation code that runs in the thread's address space to set the registers.
- *L4-Darwin server thread multiplexing.* To handle blocking in the server, the L4-Darwin server thread needs to be multiplexed. This can be done by switching stacks whenever a L4-Darwin routine needs to "block".
- *Mach interprocess communication.* Since binary compatibility is required, and there are many Darwin and Mac OS X programs which use Mach IPC, an emulation of Mach IPC at the system call level is required. Basic Mach IPC emulation has been achieved in the NetBSD Darwin binary compatibility module; Mach IPC emulation for L4-Darwin can be based on the NetBSD implementation.
- *Mach memory management.* Basic memory management capabilities exist, however there is still a lot of work to do. In particular, this implementation has not been thoroughly tested, and much more needs to be done before this becomes as stable and robust as the original Mach memory manager. The possibility of porting the Mach memory manager can also be explored further.
- *Device drivers.* L4-Darwin currently does not have any device drivers. A temporary solution to this is to use device drivers from elsewhere that are already adapted for L4, such as the Mungi

device drivers. Ideally, the solution to the device driver problem is to find a way of running the IOKit device drivers in userspace. The feasibility of this is yet to be investigated; however there does not appear to be any reason why this is infeasible.

- *Remote procedure call stubs.* It was shown that Mac OS X use Mach IPC extensively. For best performance on L4-Darwin, these programs should be modified to use L4 IPC. It is likely that most Mach IPC operations are RPCs using stubs generated by the Mach interface generator (MIG). Therefore to move away from Mach IPC, it is helpful to have a stub generator that uses the MIG definition files to generate stubs for L4 IPC. This can be done by modifying IDL4 to understand MIG definition files.

Appendix A

L4-Darwin Filesystem Image

A.1 Boot loader

The initial method of loading the filesystem image into memory was the feature of the psim PowerPC simulator where a file can be used to specify the initial contents of memory. This was unsatisfactory because it only worked with psim and could not be used when L4-Darwin is tested on real hardware. The solution was to incorporate the filesystem image into the boot loader ELF file.

L4-Darwin currently uses the piggybacker boot loader from the Pistachio distribution. To create a boot image, the ELF images of the Pistachio kernel, σ_0 server, and root task are included as data sections in a single ELF file together with the boot loader. To load the filesystem image, it is incorporated as another section in this ELF file. This extra section is loaded by OpenFirmware, but ignored by the piggybacker boot loader. In this way, the filesystem image is loaded into memory upon boot.

A.2 Makefs

The makefs tool available with NetBSD can be used to create BSD Fast File System (FFS) images. To create a filesystem image with makefs, a directory containing the required contents of the filesystem is created. When makefs is invoked, this directory is specified; makefs recursively traverses this directory and adds all files within it to the filesystem image.

It was necessary to port makefs to Linux since there were no NetBSD machines available. The main change made in the port was the removal of support for "mtree specification files". It was found that this feature relied on code from several other components of NetBSD, and since this feature was not needed it was removed.

A.3 Filesystem image requirements

For the current version of L4-Darwin, a 4096-byte block size is required. There also needs to be a /dev directory present in the filesystem image for mounting devfs, as well as a /sbin/mach_init binary that can be loaded as an init task.

Bibliography

[1] Apple Computer, Inc. *Inside Mac OS X: Kernel Programming*.

[2] Michael Condict, David Mitchell, and Franklin Reynolds. Optimizing performance of Mach-based systems by server co-location: A detailed design. Technical report, OSF Research Institute, August 1993.

[3] François Barbou des Places, Nick Stephen, and Franklin Reynolds. Linux on the OSF Mach3 microkernel. In *Proceedings of the Conference on Freely Distributable Software*, Boston, MA, February 1996.

[4] Emmanuel Dreyfus. Mach and darwin binary compatiblity for netbsd/powerpc and netbsd/i386. `http://hcpnet.free.fr/applebsd.html`.

[5] Bryan Ford and Jay Lepreau. Evolving mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*, pages 97–114, 1994.

[6] Louis Gerbarg. Advanced synchronization in Mac OS X: Extending Unix to SMP and real-time. In *Proceedings of the BSDCon 2002 Conference*, San Francisco, CA, February 2002.

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

[8] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on OS Principles (SOSP)*, pages 66–77, St. Malo, France, October 1997.

[9] Johannes Helander. Unix under Mach: The Lites server, 1994.

[10] Michael Hohmuth and Sven Rudolph. Steps towards porting a unix single server to the l3 microkernel.

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

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

[13] Simon Patience. Redirecting system calls in Mach 3.0: An alternative to the emulator. Technical report, OSF Research Institute, April 1993.

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

[15] 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.

[16] J. Mark Stevenson and Daniel P. Julin. Mach-US: UNIX on generic OS object servers. In *Proceedings of the 1995 USENIX Technical Conference*, New Orleans, LA, USA, January 1995.