Großer Beleg:
Steps Towards Porting a Unix Single Server to the L3 Microkernel

Michael Hohmuth, Sven Rudolph

April 9, 1996

Contents

1 Introduction **5**
- 1.1 Motivation . . . 5
  - 1.1.1 Why Porting a Unix Single Server To L3? . . . 5
  - 1.1.2 Which Single Server Project Should We Start With? . . . 5
- 1.2 How To Tackle the Project? . . . 5
- 1.3 Typographic Conventions Used in This Document . . . 6
- 1.4 Thanks . . . 6

2 State of the Art **7**
- 2.1 Microkernel Technology . . . 7
  - 2.1.1 L3 . . . 7
  - 2.1.2 Mach . . . 11
  - 2.1.3 Comparison . . . 15
- 2.2 Unix Operating Systems as Microkernel Application Programs . . . 17
  - 2.2.1 Approaches . . . 17
  - 2.2.2 Implementation of Lites on Mach . . . 17
- 2.3 Development Environment for L3 programs . . . 18

3 Design **19**
- 3.1 Steps Towards Porting Lites to L3 . . . 19
- 3.2 C Threads on L3 . . . 19
  - 3.2.1 Alternatives . . . 19
  - 3.2.2 Discussion . . . 20
- 3.3 Ports Emulation . . . 20
  - 3.3.1 Alternatives . . . 20
  - 3.3.2 Discussion . . . 20
  - 3.3.3 Design of the Port Emulation Library . . . 21
- 3.4 IPC . . . 25
  - 3.4.1 Alternatives . . . 25
  - 3.4.2 Discussion . . . 26
  - 3.4.3 Design . . . 27
  - 3.4.4 Alternative Design . . . 29
- 3.5 Emulating Mach's VM Subsystem . . . 30
  - 3.5.1 External Pager Interface . . . 30
  - 3.5.2 Memory Region Manipulation Interface . . . 30
- 3.6 Infrastructure . . . 31
  - 3.6.1 Requirements . . . 31
  - 3.6.2 Programs . . . 31
  - 3.6.3 Possible Approaches . . . 31

3.6.4 Decision 32
3.6.5 Running the compiled programs 32

4 Implementation **33**
4.1 Libmach Emulation 33
4.1.1 Ports 33
4.1.2 IPC 34
4.1.3 Virtual Memory Interface 35
4.1.4 Utilities 35
4.2 C Threads Library 36

5 Performance **37**
5.1 Measurement Environment 37
5.2 Message Passing Benchmark 37
5.2.1 Environment 37
5.2.2 Results 37
5.2.3 Discussion 38
5.3 RPC Benchmark 39
5.3.1 Environment 39
5.3.2 Results 39
5.3.3 Discussion 39
5.4 Evaluation 40

6 Summary and Conclusions **41**

Bibliography **43**

Chapter 1

Introduction

1.1 Motivation

1.1.1 Why Porting a Unix Single Server To L3?

TUD's Operating Systems Group is doing various projects based on microkernel operating systems, currently L3, and, perspectively, L4 and other kernels. In order to provide a microkernel based environment for research, development and day-to-day use, writing a Unix emulation running on top of the microkernel is an important strategic step towards our goals.

There are several ways to achieve Unix emulation, but for the sake of having access to a Unix environment as soon as possible (which would also have the advantage of replacing the somewhat "esoteric" L3 development environment), we have been considering porting an existing Unix single server from another microkernel to L3.

1.1.2 Which Single Server Project Should We Start With?

We ultimately want to make our results available as free software. Hence, only free, "unencumbered" systems are considered. We have been looking at:

- The Lites single server, a 4.4BSD-Lite based server and emulation library running on Mach, looks like the most promising candidate for such a project.

- The Linux single server of Louis-Dominique Dubeau's Linux on Mach project. (In the meantime, this project has been obsoleted by a similar project at OSF.)

We have been focusing on Lites for several reasons: It was the first single server system we were able to install, it can run an existing Unix base system (e.g. NetBSD) almost without any modification, and it seems to be relatively well-supported.

1.2 How To Tackle the Project?

In this reoprt, we will first compare Mach with L3. Later, we will investigate how Lites has been implemented on Mach. We will identify common and different designs employed in these microkernels.

Beginning with this analysis, we will try to develop a strategy for porting Lites to L3: Which Mach facilities should be emulated? Which parts of the Lites kernel should be rewritten or changed?

1.3 Typographic Conventions Used in This Document

Throughout this document, certain typographic conventions will be used:

Emphasized text will be used to introduce new concepts.

`Typewriter font` marks program code.

This special font will be used to emphasize C function names and type class names. Variable and object names will be written in normal font, however.

1.4 Thanks

We would like to thank Prof. Hermann Härtig and the real-time operating systems group at TU Dresden for their support, Jochen Liedtke from GMD for useful dicussions, Bryan Ford from Utah University for his advice, and Jean Wolter for proof-reading this document.

Chapter 2

State of the Art

2.1 Microkernel Technology

2.1.1 L3

L3 is a microkernel based operating system developed by Jochen Liedtke and others at GMD's SET institute. Its key features are persistence, minimality and speed. L3 features synchronous message-based IPC, a simple-to-use external pager mechanism and a domain-based security design. [1] [2]

L3 knows about only a small number of kernel objects: tasks, threads and messages. (*task* denotes a protection domain provided by the (μ-)kernel, and *thread* means an activity within a task.)

IPC Architecture

A message is a collection of data that is passed to the kernel in one system call. The kernel is responsible for delivering this message to the intended recipient.

In L3 messages are sent by one thread to another thread without intermediate objects like ports. IPC is synchronous, i.e. the sender thread blocks until the receiver thread has received the message. These design decisions are a good base for enhancements of IPC performance. [3]

A message can contain components of four different types:

direct string These values are stored in the message buffer. Two DWords (one DWord is 4 Byte long) are mandatory.

indirect strings Indirect strings are specified in the message buffer by a pointer and the length of this memory area.

flex pages Flex pages are a mechanism for transferring memory pages between tasks. [4]

data spaces Data spaces are abstract memory objects. Sending a data space employs lazy copying.

A *message dope* is a structure that describes how many components of each of these types are to be processed. The *buffer dope* describes the layout of the provided message buffer and therefore how many items can be received, the *send dope* describes how many items are to be sent. (The two dopes differ when you send less data than you are prepared to receive.)

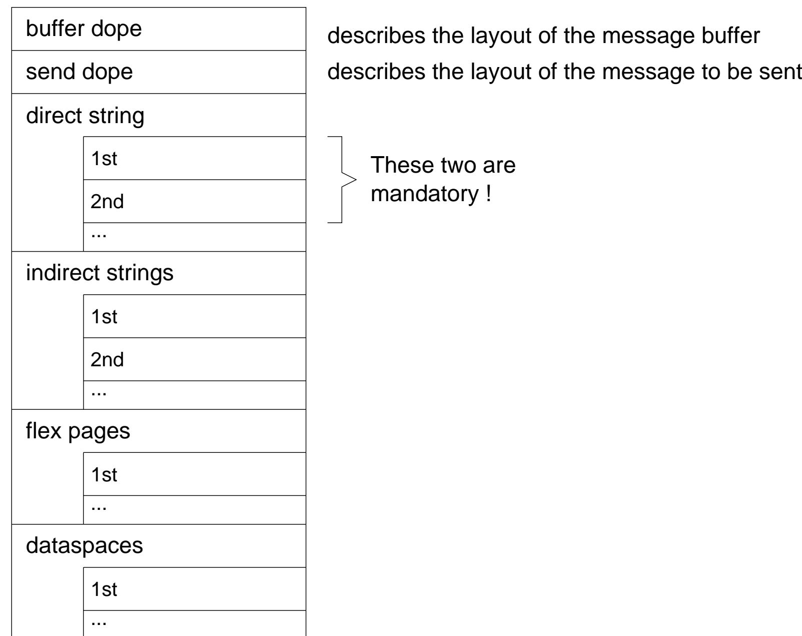

Figure 2.1: Message Structure in L3.

Design of the Virtual Memory Subsystem

During preparation of this document, L3's virtual memory (VM) subsystem has been comprehensively documented for the first time. This section will only explain the most important concepts of this subsystem. For a more detailed discussion please refer to the L3 documentation [5] and Jean Wolter's thesis [6].

Address Spaces An *address space* (aka *virtual* address space) defines the set of valid virtual addresses that any thread executing within the task owning the address space is allowed to reference. An address space is owned by exactly one task, and is created and destroyed along with it.

The semantics associated with a valid region of virtual memory are provided by memory managers which implement abstract memory objects called data spaces (discussed below). When a new memory region is established in an address space, a thread id of a manager providing those semantics needs to be specified.

Data Spaces A *data space* is an abstract memory object which can be mapped into the address space of tasks. The semantics associated with a data space are provided by the memory manager that backs it. Data spaces can be of variable sizes up to 4 GB.

From the kernel's perspective, data spaces are uniquely identified by a <user_task, memory_manager, data_space_number> triple, where the data space number is an identifier which is never interpreted by the kernel. It is the manager's job to decide whether the same data space number in two different user tasks refer to the same memory object or not (in the standard memory manager, it does not).

The kernel should be viewed as using main memory as a (directly accessible) cache for the contents of the various data spaces.

Memory Managers A *memory manager* (sometimes also called a *pager*) provides the semantics associated with the act of referencing memory regions which have mapped data

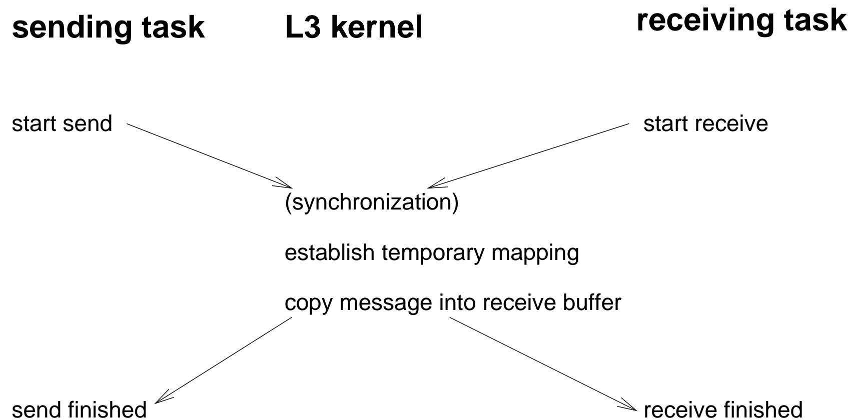

Figure 2.2: Message Transfer in L3.

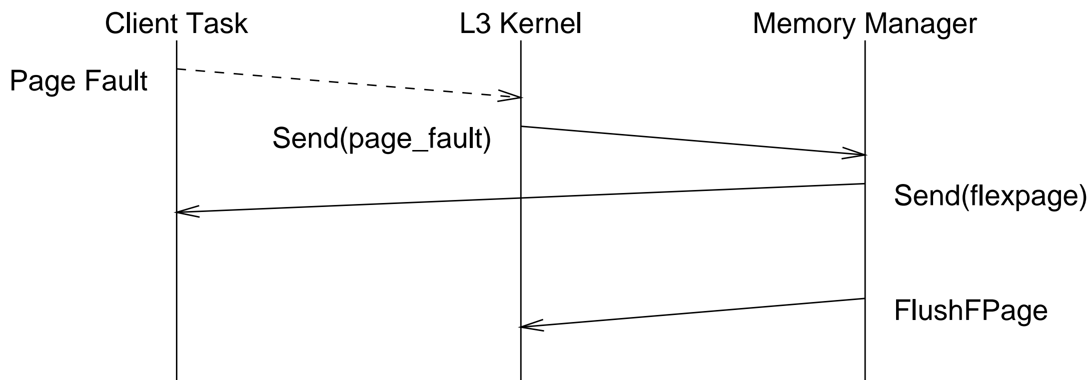

Figure 2.3: Memory object manipulation in L3.

spaces backed by it. It provides (or denies) access to pages of data that have been referenced by user tasks.

Every thread can be a memory manager for every other thread. There is no special initialization necessary to become a memory manager, and the kernel provides no special access protection for them (as with all L3 IPC). Once the id of some thread has been specified in a map system call, the kernel starts treating the thread as the memory manager for the region (and send page fault messages to it, which, of course, can be ignored or refusingly replied to by the thread).

Memory managers willing to back a data space are involved in a simple IPC dialogue with the kernel [5]. Basically, the kernel reports page faults of client tasks, and the memory manager responds with a flexpage mapping to resolve the page fault and allow the kernel to establish a temporary page mapping.

Flexpage mapping Temporary mappings of memory regions from a memory manager to a client task are not limited to single hardware memory pages. Instead, the manager may specify any memory region to be mapped into the client (a *flexpage*). This is a simple optimization which may result in fewer page faults from the client because several pages can be mapped at once. [4]

Figure 2.3 shows the basic memory object manipulation dialogue between the kernel and a memory manager: A client task references a portion of a memory range for which a temporary

mapping has not yet been established. The kernel will send a page fault message to the respective memory manager and set up a receive operation on behalf of the client. The memory manager analyzes the request and responds with a flexpage directly to the client.

The kernel is free to forget the flexpage mapping at any time (perhaps when evicting the page from main memory) which may result in making clients send page fault messages requesting previously supplied pages again.

The memory manager may also forcibly remove a flexpage mapping established earlier (with a system call), for instance, when the manager is low on memory and wants to recycle memory regions previously committed as flexpages.

Kernel Resource Management

Except the CPU and the kernel clock, the L3 kernel doesn't manage system resources at all; that's done by of kernel-external tasks (discussed below).

The clock is managed by the kernel for two reasons: First, the system clock can well be thought of as a part of the CPU, and secondly (and more importantly) the kernel requires access to the clock to perform IPC timeouts. [7]

Kernel-external Standard Facilities

L3 comes with a number of kernel-external standard facilities which handle many of the resources conventionally managed by an operating system kernel.

Supervisor The supervisor is L3's root task. It manages host control (definition of a station's name, and spawning sub-tasks which manage checkpointing, devices and other system services) and controls a system's task tree. The supervisor is the only task which may create new tasks and threads; other tasks have access to these services through a special message-based (IPC) protocol, the *Supervisor Protocol.*

Device drivers Device drivers are (from the kernel's point of view) normal user tasks. (Well, almost: As they must not be paged out, they're memory-resident and non-persistent. Consequently, they must be re-started every time the system reboots.)

Device drivers are controlled in a uniform fashion using the *Generic Driver Protocol.*

Embedded ELAN compiler Most non-kernel operating system facilities have been implemented in ELAN. The standard ELAN environment contains procedures encapsulating L3 system calls and IPC protocols to manager tasks (for example, the Supervisor Protocol and the Generic Driver Protocol).

Please note that L3 currently doesn't come with any C libraries to access system services. That's why our group has developed several libraries providing C bindings to many system services [5]; these libraries are now being exploited regularly and gladly:

libl3sys encapsulates L3 system calls and calls to the standard memory manager.

libl3prot encapsulates L3's standard protocols: the Supervisor Protocol and the Generic Driver Protocol, among others.

libl3serv contains support generally required by L3 servers: startup code, trap handlers, and global initialization.

2.1.2 Mach

Mach is a microkernel originally developed at CMU. It features an object-oriented design approach, sophisticated memory management and message-based IPC.

Mach is meant as a foundation for operating systems built up on it and provides several kernel abstractions commonly required, its key elements being tasks and threads (similar to L3's), ports, messages and memory objects.

IPC

Messages are sent via unidirectional communication channels, so-called ports. The ability to send messages to ports and to receive messages is stored in *port rights*. Port rights are kernel-protected capabilities, the kernel enforces these capabilities and controls the transfer of port rights between tasks. Due to this access control properties ports also used to name other kernel resources: tasks, threads, devices, memory objects, among others. [8]

Unlike L3, IPC in Mach is asynchronous. When the send operation cannot deliver the message immediately it stores the message in a message queue and returns. The receive operation then dequeues this message.

Ports A port is a unidirectional communication channel between tasks. A port has a single receiver and (potentially) multiple senders.

The state associated with a port is:

- The message queue.
- The reference counter. (It counts the number of send rights that currently exist for this port.)
- Limits for the amount of received data.

Figure 2.4 shows a port and associated objects. These are described in detail below.

Messages A message is a collection of typed data.

Available message types are:

- pure data (integers, floats, strings, ...)
- port rights
- memory regions

Message Queues A message queue is associated with a port and stores all messages that were sent to this port.

Port Rights A port right is a secure entity that indicates the right to access a specific port.

There are three kinds of port rights:

receive right A receive right allows the task to receive messages from this port. Only one receive right exists for a port.

send right A send right allows the task to send messages to this port. The number of send rights per port is not restricted.

send-once rights A send-once right allows the task to send one message to this port. The send-once right is destroyed after the message is sent.

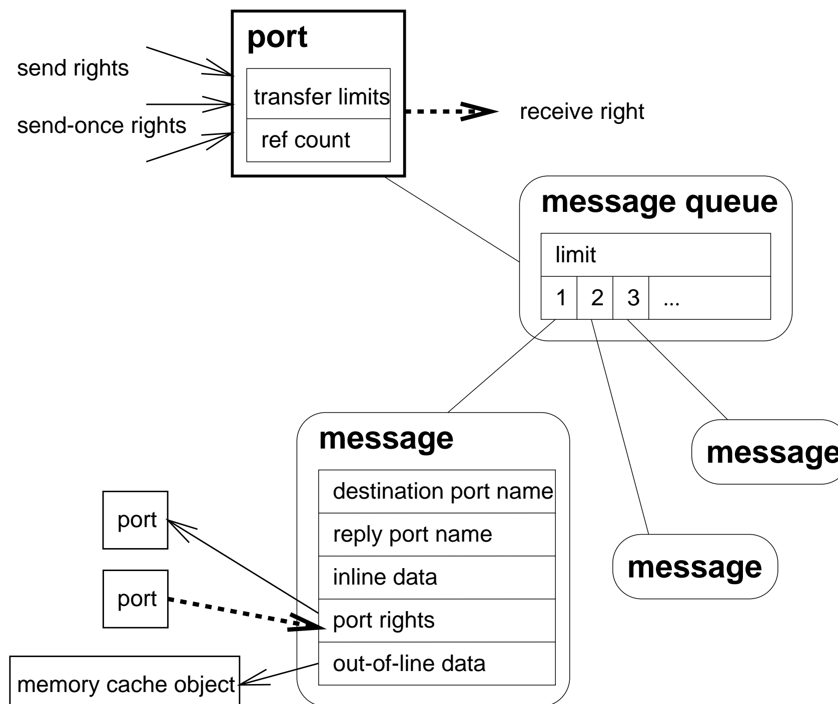

Figure 2.4: IPC structure in Mach.

Port rights are system-wide identifiers for ports. They are stored in kernel-protected port name spaces and can only be manipulated by tasks via port names.

Port Names A port name is used to name a port right in a task and points to one entry in the *port name space* where all port rights of one task are stored. Every port name space is protected even against its task.

Port Sets A port set is a common access point for receiving messages from ports. A receive operation retrieves a message from one of the queues of the member ports unless all of these queues are empty. A port cannot be a member of more than one port set.

Design of the Virtual Memory Subsystem

Mach's virtual memory design is one of the most remarkable aspects of the system. It is layered in a simple hardware dependent portion and a hardware independent portion providing a complicated set of high-level abstractions. It claims to feature high performance, exploiting optimizations like lazy copying and shared memory. [8]

The abstractions provided are:

Virtual Address Spaces *Virtual address spaces* in Mach share many aspects with L3's address spaces: They are owned by tasks and are created and destroyed along with them. Ranges of memory within a virtual address space can be associated with abstract memory objects backed by memory managers (discussed below).

Abstract Memory Object *Abstract memory objects* can be mapped into the virtual address space of tasks. The semantics associated with an abstract memory object are provided by the memory manager that backs it.

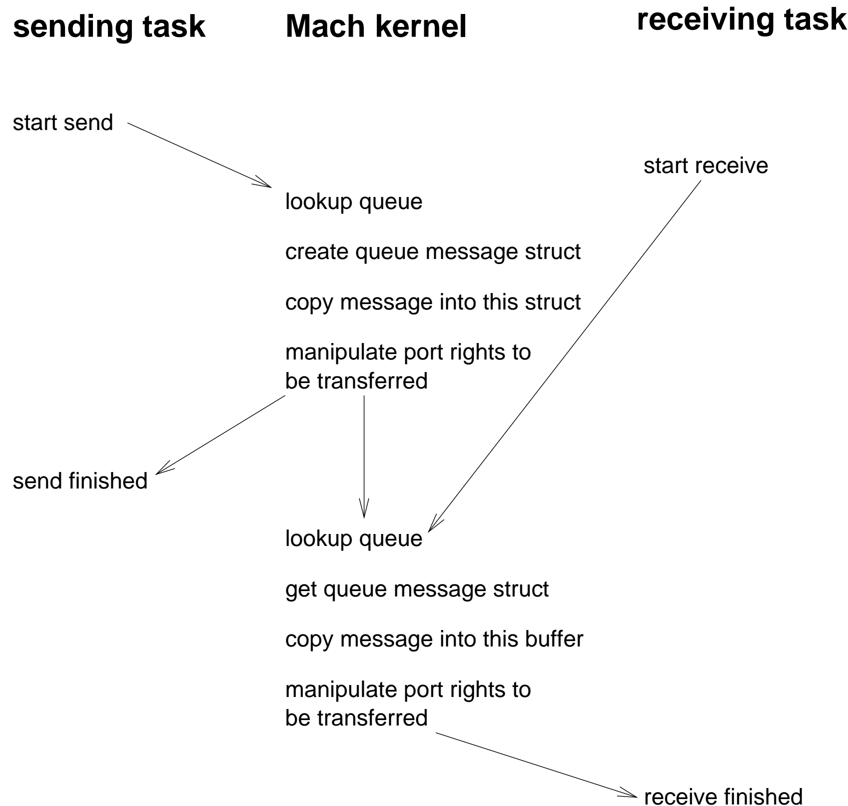

Figure 2.5: Message transmission in Mach.

Memory objects are named by abstract memory object representative ports (the port specified to the kernel when mapping a memory object into a client task's virtual address space) and abstract memory object ports (the port the memory manager presents to the kernel when initializing a memory object; see below).

Memory Cache Object The kernel should be viewed as using main memory as a (directly accessible) cache for the contents of the various abstract memory objects. In order to be able to control this cache's contents (i.e., to map or invalidate memory ranges), the memory manager gets a send right for a memory cache control port from the kernel at memory object initialization.

A *memory manager* is a task holding a send right to a memory cache object port. It is involved in a dialogue with the kernel to maintain the main memory cache of the abstract memory object backed by it. In this dialogue, the kernel sends requests to the memory manager's abstract memory object port, and the memory manager supplies requested data using IPC to the kernel's respective memory cache control port (see figure 2.6).

Memory is passed back and forth between the kernel and the memory manager as out-of-line data in usual Mach messages. The kernel may return data it was supplied earlier at any time; when doing so, the returned data becomes backed by the default memory manager (a manager built into Mach, the “pager of last resort”) until the manager explicitly deallocates it.

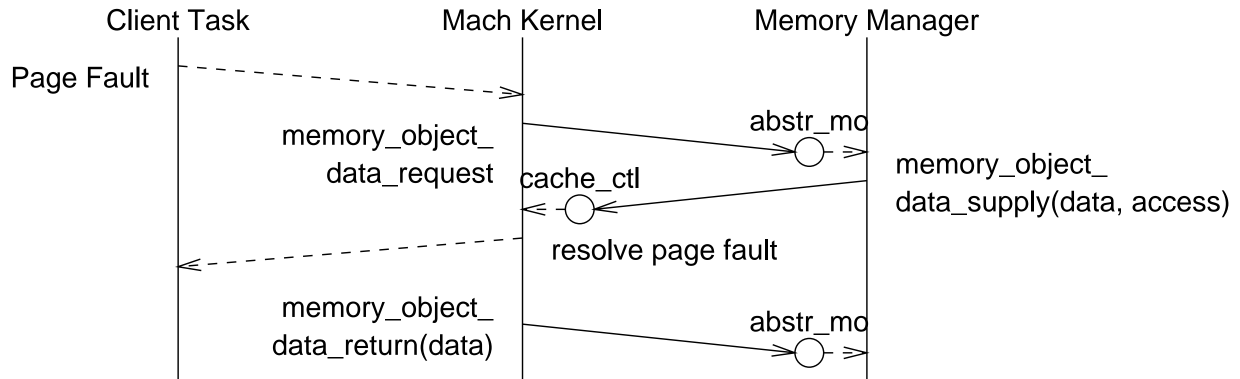

Figure 2.6: Memory object manipulation in Mach.

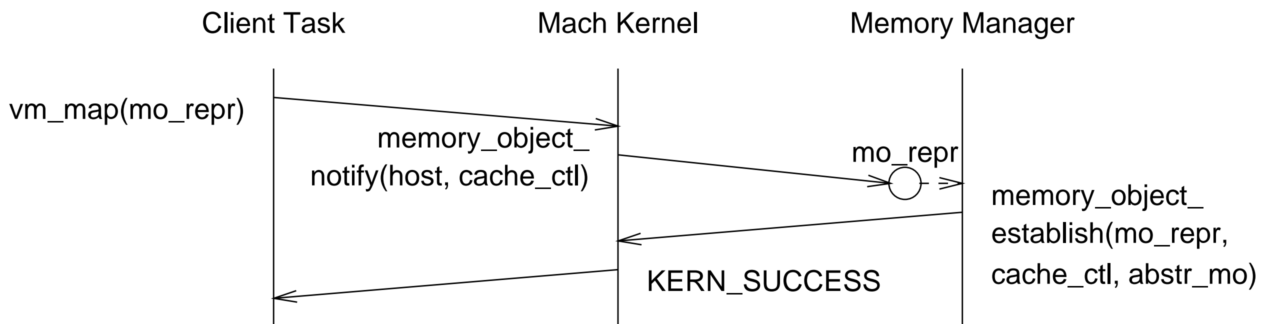

Figure 2.7: Memory object setup dialogue in Mach.

The protocol to initialize an abstract memory object has a special meaning: It is used to authenticate the kernel and the memory manager to each other. For details about the employed authentication scheme, please refer to Mach Kernel Principles [8].[1]

During this dialogue (figure 2.7), the kernel needs to establish an association between the memory object representative port right presented in the **vm_map** system call and the memory object's abstract memory object port. That's why the kernel communicates a send right for its memory cache control port to the owner of the specified representative port and believes any task which can present the rights of both the representative port and the cache control port to be the memory manager for said representative port. This dialogue also has the side effect of exchanging the cache control port and abstract memory object port rights between the kernel and the memory manager so that the memory object manipulation dialogue discussed above can begin.

Kernel Resource Management

The Mach kernel virtualizes many physical resources conventionally managed by operating systems: clocks, devices, hosts, nodes, processors and processor sets (see Mach Kernel Principles [8] for an explanation of these resource classes). The kernel also provides for resource accounting

[1] The initialization dialogue discussed here is unique to the OSF version of Mach 3.0. Other Mach version employ a simpler scheme; however, with those schemes the memory manager is unable to protect itself properly.

using *ledgers,* a special kernel resource providing a mechanism to limit consumption of another resource.[2]

Kernel-external Standard Facilities

Mach comes with extensive C libraries and an RPC interface generator which make interfacing the microkernel easy.

Libmach The C library *Libmach* provides C bindings to system calls and IPC to kernel-implemented objects. [9]

Libcthreads The *Libcthreads* library is a user-level threads library. It implements *C Threads* which are not to be confused with kernel threads: C Threads are user-level threads, and Libcthreads manages the mapping from C Threads to kernel threads. It also offers tools to synchronize threads. [10] [11]

MIG *MIG*, the Mach Interface Generator, is Mach's RPC stub generator. It generates code for communication between two tasks from an interface description. The interface is described as a set of procedures with input and output parameters. Libmach contains some MIG-specific functions that are required by MIG-generated programs. [10]

2.1.3 Comparison

This section will compare the two microkernels Mach and L3. We will identify features available in both microkernels and features available in one but missing in the other.

Later in this section, we will take a closer look at the virtual memory subsystems of the kernels.

Features Available in Both μ-kernels

- Tasks, threads.
- Virtual memory subsystem with external pager interface; however, the implementation and kernel interface of the VM subsystems differ a lot between the two kernels; see below.
- IPC via messages. However, there is a fundamental difference between the kernels' IPC schemes: L3 implements synchronous IPC between threads while Mach offers asynchronous IPC using ports as communication channels.
- Interface to clocks and other physical resources, although implemented very differently.

If one system were to be emulated on the other, a fair (albeit varying from feature to feature) amount of emulation glue would be required.

Features Only Available in Mach

The following fundamental Mach features don't have an equivalent on L3:

- Ports.
- MIG, the RPC interface generator.

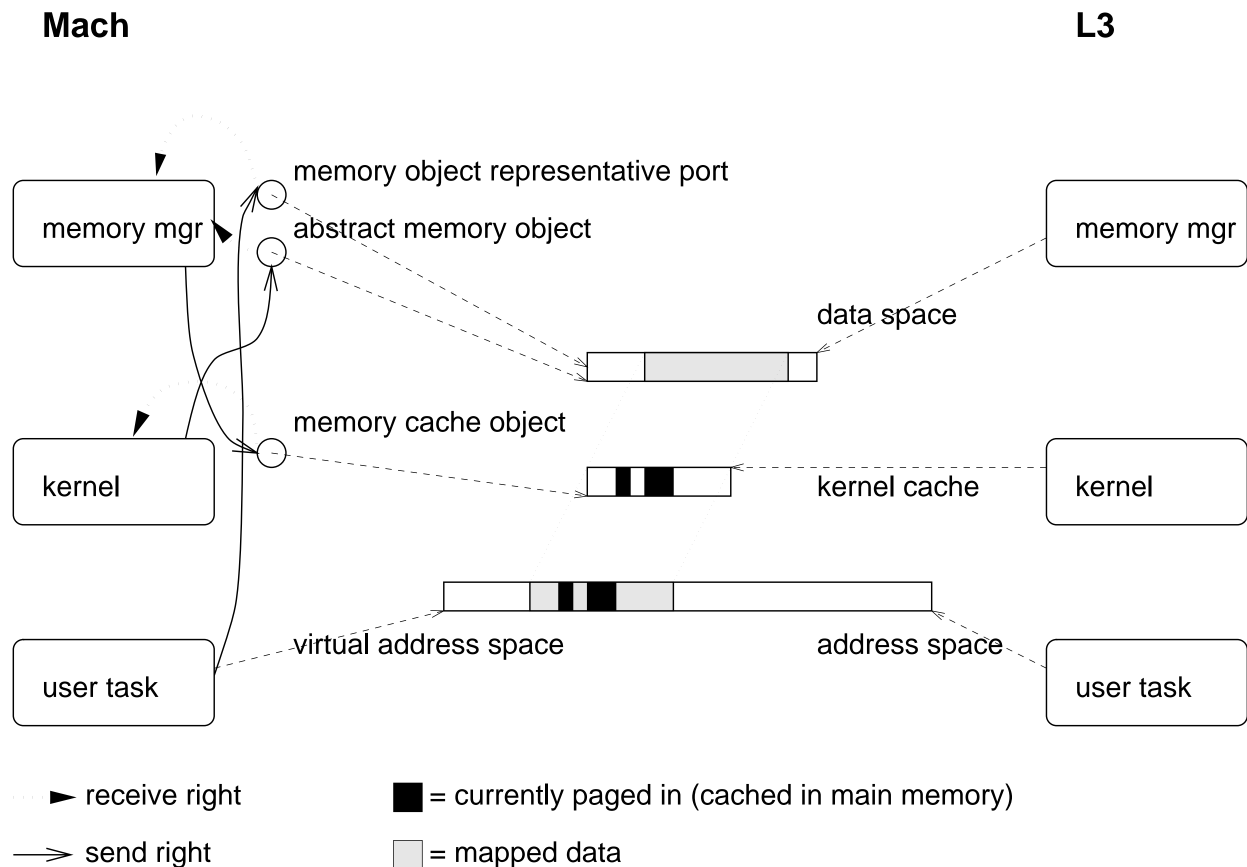

Figure 2.8: External Pagers in Mach and L3. The figure shows the abstractions (objects) involved in the external pager mechanisms, and the various types of references the active elements (tasks and kernel) hold, as explained in sections 2.1.1 and 2.1.2

Comparison of the Virtual Memory Subsystems of Mach and L3

Both microkernels possess virtual memory subsystems which fit well in their respective kernel design philosophies: L3 strikes for minimality, simplicity and speed, while Mach goes for object-orientation, elegance and kernel-enforced reference protection.

Figure 2.8 compares the elements involved in Mach's and L3's external pager mechanisms. We see that these elements have similarities to their respective counterparts in the other kernel, but their interconnections (and interactions) differ a lot.

There are further, more subtle differences:

- In Mach, when resolving page faults, the memory manager passes pages to the kernel in terms of out-of-line data in **memory_object_data_supply** messages sent to the kernel's cache control port. After the data has been supplied, the memory remains under kernel control until returned in a **memory_object_data_return** message. [8]

  In L3, it works differently: Memory managers merely pass their data by reference using flexpage mappings; the memory remains under control of the memory manager. Both the kernel and the memory manager can remove the mapping at any time. [4]

[2]Ledgers are only available in OSF Mach.

- Mach provides an interface to operate directly on virtual memory regions (**vm_***): For instance, it is possible to unmap a specified memory region, no matter which memory manager(s) backs the region. [8]

  L3 provides no equivalent interface: Region management is carried out entirely within memory managers (hence only available on single data spaces, not on virtual memory regions having any number of data spaces mapped), so client tasks need to communicate with their memory manager to manipulate memory regions.

2.2 Unix Operating Systems as Microkernel Application Programs

2.2.1 Approaches

There are two approaches to implementing Unix operating systems on microkernels: multi-server and single-server. This work (and this section) focus on the single-server approach; for a general discussion of multi-server vs. single-server vs. monolithic Unix design, please refer to Jean Wolter's thesis [6] and the Lites Server document [12].

2.2.2 Implementation of Lites on Mach

Lites is a free Unix implementation implemented as a single-server and an emulation library running under Mach. It has been developed by Johannes Helander at HUT and borrows code from CMU's Unix single-server implementation and UCB's 4.4BSD Lite. [12]

In this section we will look at certain aspects of the Lites implementation, namely the aspects relevant to porting the Lites server to another microkernel.

Types of IPC Used

Messages through ports are used for RPC between the Unix server and the emulated Unix processes (the clients), and for delivering signals to a signal handling thread running within the task emulating the Unix process.

Shared memory is employed for sharing various data between the server and the clients, both in read-only and read-write fashion: u area, time information and mapped files ("vnode paging"), among others. The server acts as an external pager for the clients.

Faults, exceptions and low-level Mach system calls Besides the usual semantics of invoking Mach mechanisms like page in/out requests to pagers and Mach system calls, these types of IPC are also used to emulate certain Unix mechanisms: Unix system call redirection/emulation and raising Unix signals. [12]

What Ports Are Used For

Ports are not just used as communication channels; they're exploited in various other ways, too:

Object referencing Besides utilization as references to Mach kernel objects as explained in section 2.1.2, ports are also used by the clients' emulation library to reference Lites (Unix) kernel objects like process structures and files, and references to Lites' ttys, Unix devices and network interface handlers are passed to Mach's device drivers.

Mach-enforced reference protection The Mach kernel protects port references (send and receive rights). The utilization of port rights as capabilities regarding Lites kernel objects simplifies access checking in the Lites server. [12]

Other Mach Facilities Employed

Besides ports and Mach's IPC facilities, Lites makes use of various other Mach facilities:

Libcthreads The C Threads library is utilized to gain concurrency in the Lites server and to synchronize accesses from threads to kernel data structures.

MIG The interfaces for communication between the server and the emulation library mapped into the user task's address space are described as MIG specification files. MIG generates the RPC stubs from these files.

2.3 Development Environment for L3 programs

The L3 kernel is implemented using a DOS 32-bit assembler by Pharlap. This assembler is run in the DOS environment and we cannot give it away at no cost because of licensing restrictions. The memory management subsystem is written in CDL, but we did not consider to use this non-widespread language. (Our group even started a project to rewrite the memory management in C.)

ELAN, an ALGOL-like language, is used as command language, for scripts and as implementation language for about all non-kernel programs. The ELAN compiler translates very efficiently into i386 assembler. [13]

A port of gcc (the GNU C Compiler) version 1.37 is available. Compared to the current gcc 2.x it lacks some features, among them limited inline assembler facilities and no C++.

We didn't find a make-like programs .

Chapter 3

Design

This chapter describes the various steps to be tackled to port Lites to L3, and alternative approaches.

3.1 Steps Towards Porting Lites to L3

In this section, we will list the subtasks we've identified which will ultimately lead to our goal, a Lites implementation running on L3. All steps will be discussed in greater detail later in this chapter.

The following list is based on some of the findings of section 2.2.2, namely that Lites depends a lot on Mach ports, the C Threads Library and Mach's IPC facilities.

1. Create a C development environment for L3. (Section 3.6)
2. Write a C library providing bindings to L3 system calls. (This step has already been carried out by other members of our group, so we didn't have to spend much time on it. [5])
3. Provide the C Threads user-level threads interface on L3. (Section 3.2)
4. Replace utilization of Mach IPC with corresponding types of L3 IPC. (Section 3.4)
5. Provide a certain level of Libmach emulation; at least ports must be emulated on L3 because Lites depends heavily on them. (Sections 3.3 and 3.5)
6. Replace references to Mach's device drivers with L3's. (This step is beyond the scope of this work.)

3.2 C Threads on L3

3.2.1 Alternatives

To resolve Lites' dependence on Mach's C Threads library, we've identified the following alternatives:

1. Re-implement C Threads based on L3 kernel threads, or implement a Pthreads library on L3. (The Pthreads interface is very similar to C Threads, and it is easy to emulate C Threads on the top of Pthreads.)

2. Mach's Libcthreads is based on Mach's kernel threads interface. An easy way to port Mach's Libcthreads to L3 would be to emulate Mach's kernel threads interface on L3.

3. We could modify Lites in such a way that it is no longer dependent on C Threads but uses L3 threads directly.

3.2.2 Discussion

As discussed in section 2.1.2, the C Threads library provides user-level threads which it multiplexes onto Mach kernel threads. It provides a higher-level interface than the kernel thread interface.

The L3 kernel thread interface is comparable to Mach's. Both interfaces implement similar operations: creation and destruction of threads, and definition of a thread's properties. [5] [8]

To summarize, Option 3 would involve changing Lites to use a more lower-level interface, Option 2 would require emulation of Mach's low-level thread interface on L3's similar interface, and Option 1 means the re-implementation of a high-level interface.

We favour the second option (emulate Mach's kernel thread interface) because we estimate it to be least expensive.

3.3 Ports Emulation

The central aspect of Libmach emulation is resolving Lites' dependence on ports. As explained in section 2.2.2, Lites depends on Mach's port functionality in various ways.

3.3.1 Alternatives

1. Modify Lites to get along without ports.

2. Emulate a subset of Mach's port functionality on L3:

   (a) Write an external port emulation server as an L3 task. This server would emulate Mach port semantics.

   (b) Handle port emulation locally within tasks which depend on it.

3.3.2 Discussion

Ports Vs. No Ports

The advantage of a Lites implementation which doesn't require a port layer seems obvious: A version of Lites optimized for L3's synchronous message passing scheme promises much better performance than one which requires an emulation layer. However, in section 2.2.2 we've seen that Mach ports are utilized not just as communication endpoints but for a great variety of purposes. It follows that doing without ports would require a lot of changes to Lites; therefore, we've refrained from this option.

External Server Vs. Task-local Port Emulation

The main advantage of having an external port emulation server as a self-contained L3 task is basically the same which holds for any multi-server solution: The solution is more modular, and components have better protection against each other. In our case, the clean separation of functionality traditionally provided in the Mach kernel into an external server would be an

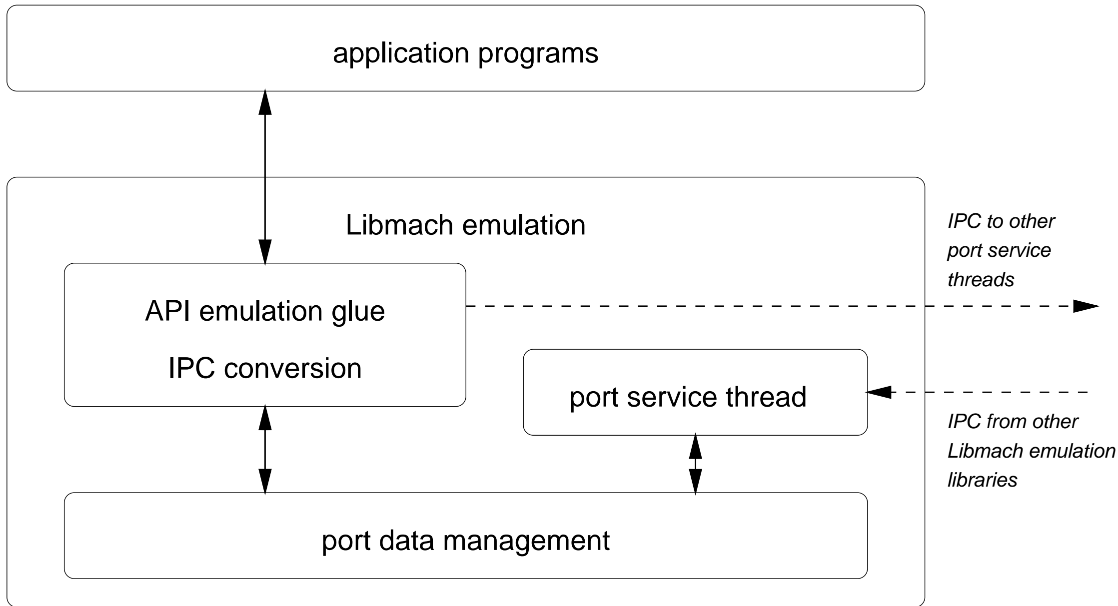

Figure 3.1: Overall Structure of Libmach Emulation.

additional advantage: There would be only one entity which would need to parse Mach messages in order to manipulate (and protect) port rights and memory objects and translate port names from one task's namespace into another's.

On the other hand, the external server solution bears disadvantages. If the server were to provide port reference protection and port name translation, it would require sending every message twice: once when sending it to the server for approval and a second time when delivering it to the recipient. As L3 doesn't support lazy copying of random memory ranges[1], this incurs a performance hit. While the amount of copied data per sent message may be relatively small, it certainly sums up to considerable quantities because the extra copy happens at every system call via RPC to the Lites server.

The task-local port emulation solution avoids copying IPC data several times. However, it possesses a problem of its own: Each task would have to maintain its own port name space, i.e. port names and port references for all known ports. When updating port references for a task, the respective partners involved need to update their reference information: When transferring send rights, only the owner of the receive right must be informed, but when transferring receive rights, *all* send right owners need to be passed the new recipient information. The latter operation would be considerable cheaper when using a central server.

In practice, however, migration of receive rights between tasks is a rare operation. We've analyzed the Lites code and found that it doesn't use transfers of receive rights at all; thus, the reference update problem doesn't apply to us. That's why we've decided to implement the task-local emulation solution.

3.3.3 Design of the Port Emulation Library

The goal is to create a library which provides a subset of the Libmach API, namely the **mach_port_*** routines. Further objectives are efficiency and easiness. Starting with Lites' pattern of port use, we want to provide just the functionality Lites requires.

[1]Unlike Mach, L3 supports lazy copying only on the data space level.

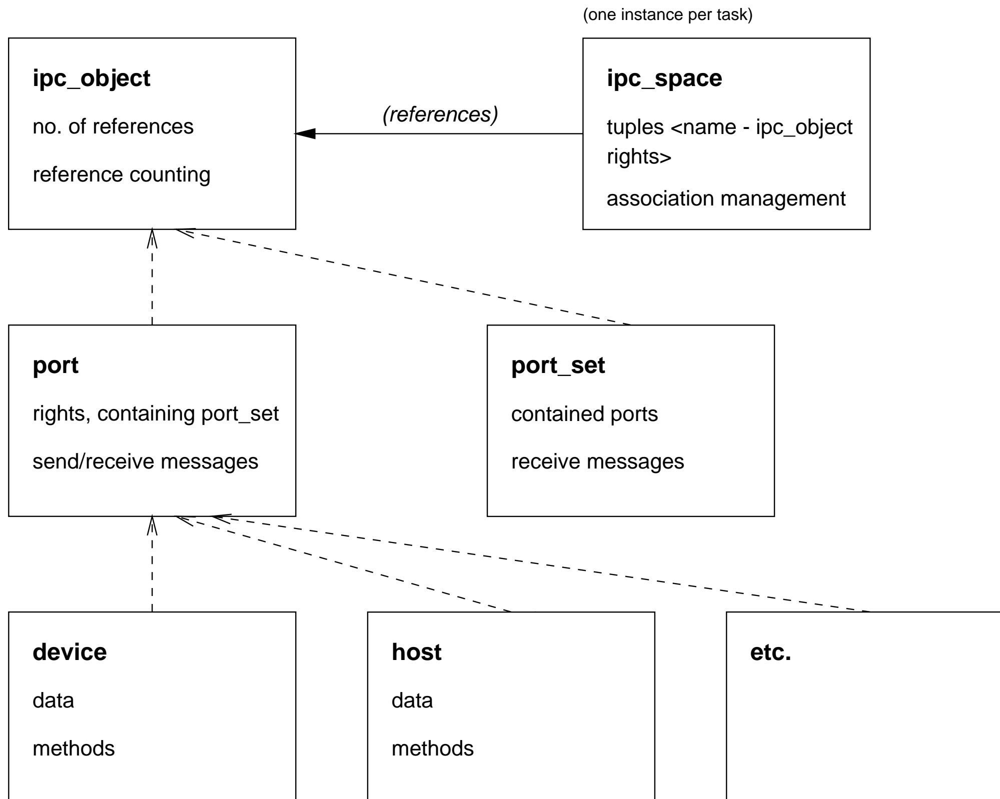

Figure 3.2: Class Hierarchy of Port Implementation in Mach Kernel. Each box lists a class name, the specific data that objects of these classes hold, and typical methods applied to members of the class

Overall Design

Figure 3.1 shows an overall sketch of the Libmach emulation design.

The *port data management* layer is the heart of the emulation library: It manages data structure bookkeeping and synchronization between the application and the library's port service thread and is discussed below in greater detail.

The *port service thread* handles incoming Mach IPC messages sent by instances of the Libmach emulation in other tasks.

The API emulation glue and IPC conversion layer does the real emulation work: transforming Mach IPC calls into L3 IPC (discussed below in section 3.4), and emulate Mach kernel object manipulations using the port data management layer.

Furthermore (and not shown in the figure), the emulation library contains utilities for L3 virtual memory management, data structure locking, initialization and debugging aids.

Design of the Port Data Management Layer

The port data management layer manages the data structures used for bookkeeping ports, port references and outstanding (queued) Mach messages.

It has been designed in object-oriented fashion. The rationale is that the respective portion of the Mach kernel has been designed in object-oriented fashion, too, and we tried to resemble their design as much as sensible.

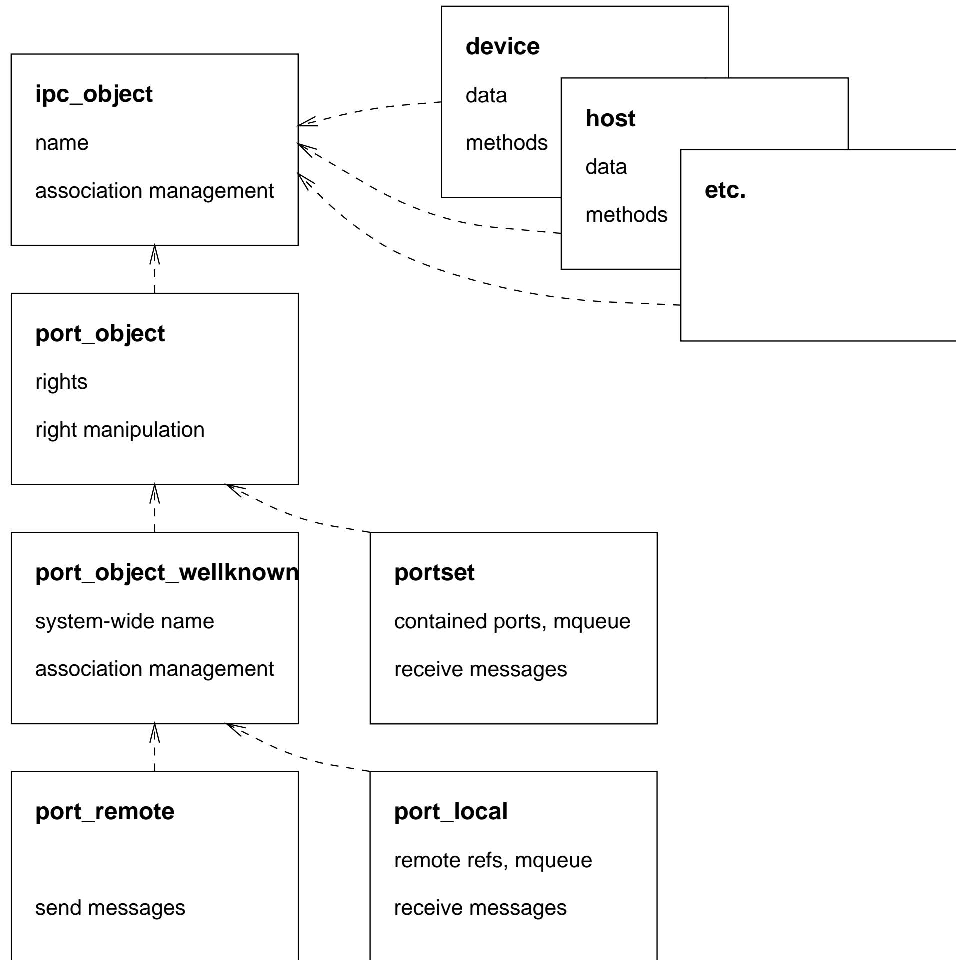

Figure 3.3: Class Hierarchy of Libmach (and Port) Emulation.

Figure 3.2 shows part of the class hierarchy as implemented in the Mach kernel. We see that the tasks' port name spaces (**ipc_space**) contain references to members of **ipc_object** (or subclasses thereof). **ipc_object** is the internal notion of objects which are named by port names; they can be **port**s or **port_set**s. All other kernel objects (devices, hosts and so on) are basically implemented as subtypes of ports; they're controlled with specializations of the generic Mach IPC interface.

The class hierarchy as implemented in the Libmach emulation is shown in figure 3.3. There are several differences to Mach's hierarchy:

- Because all data structures are local to the application (as opposed to Mach's central management), there's no need to map from a per-task name space to **ipc_object**s. Instead, the port name management can be done within the **ipc_object** class.

- We need to distinguish between local ports (those a task owns the receive right for) and remote ports. Furthermore, we wish to distinguish between port objects which need to be known system-wide and those which don't. This leads to the various sub-classes of **port_object**.

- To resolve references to ports between remote tasks, we need a naming scheme which is valid system-wide. This naming scheme operates on objects of type **port_object_wellknown**. Unique system-wide names are created along with new ports.
- Mach system objects like Mach devices can be emulated locally. Because we don't need to allow remote access to the device interface, we don't need to control them via Mach IPC, and consequently we don't require their class implementations to be inferior to any of the **port_*** classes. It is sufficient to make them inferior to class **ipc_object** as they're still named in the same way as ports (that is, those objects are named with port names, but they're not really ports).

Mapping Mach Semantics to the Port Data Management Layer

1. The **mach_msg** library function transmits messages to remote port service threads as follows:
   - Using the port name specified to **mach_msg**, look up the corresponding port_remote element.
   - Encapsulate the Mach message in an L3 message, address it with the port_remote's system-wide name and send it to the port service thread indicated in the port_remote object.
2. A port service thread which has received the encapsulated Mach message using L3 IPC needs to append the message to the appropriate message queue:
   - Extract the addressee from the message; this is the system-wide name of the port to which the message has been addressed.
   - Try to find a corresponding port_local data structure using the appropriate lookup function.
   - If it finds one, check if the sender holds an appropriate right to send into this port.
   - Check if the port belongs into a port set. If so, queue the message in the message queue attached to the port set, otherwise in the port_local's queue.
3. To dequeue the message, the **mach_msg** library routine needs to lookup the portset or port_local element corresponding to the port name it was specified, and can then remove the message from the attached queue.
4. A major Mach concept emulated by the port emulation library is the transfer of send rights to Mach ports via Mach messages. This is accomplished using a 3-phase-commit-like protocol: In order to prevent race conditions, we have to make sure neither the sender nor the receiver of the send right can use the send right before the transfer message has been successfully sent.
   - When the **mach_msg** routine detects that a send right should be copied or moved, the corresponding port's port_local object is notified of the planned transfer operation. If the current task (that wishes to transfer the right) also owns the corresponding port's receive right (i.e. the port_local object in question is local to the task), this can be done by directly invoking a routine provided by the **port_local** class. Otherwise (the port_local object is remote), an RPC is sent to the port owner's port service thread requesting it to run the routine on behalf of the current task.

- The port_local object checks the request and stores it for later execution, but does not actually update its reference lists yet. This is done so that **mach_msg** can cancel the transfer of rights should the message transfer to the target task fail. Then it returns to the invoking **mach_msg** routine, indicating whether the operation was approved, and also returning a handle which can later be used to commit or cancel the rights transfer.
- If the port_local object has approved the modification, the **mach_msg** routine tries to send the message and the commit handle to the target task as usual. (If the transfer fails, it invokes the port_local object again to cancel the rights transfer using the commit handle.)
- If the message has been sent successfully, the receiver invokes the port_local object, supplying the commit handle, to commit the rights transfer. The port_local object updates it list of rights, by inserting new rights, deleting rights and updating reference counters, as requested.

In the special case when the sender of the receive right also holds the receive right to the port in question, this protocol can be short-circuited by a normal message send operation (as long as the port data structures remain locked by the sending thread).

3.4 IPC

Lites uses MIG-generated stubs, these in turn employ Mach IPC via **mach_msg**. We have to enable Lites to use the L3-provided message facilities instead.

3.4.1 Alternatives

We have seen in section 2.1.2 how the message transfer works in the Lites-on-Mach environment.

The main difference between the following alternatives is the level at which the message flow is intercepted and the message is converted from the Mach form into a L3-specific one.

Using L3 IPC System Calls

All code fragments referring to IPC could be adapted manually in order to interface to the L3 IPC system calls.

IPC Stub Generation

The IPC interfaces used by Lites are described by MIG interface definition files. A stub generator could generate stubs that use L3 IPC instead of Mach's **mach_msg**.

mach_msg Emulation

All MIG-generated stubs call **mach_msg**. **mach_msg** is the only entry to the kernel's IPC transmission facilities. So we could provide our own **mach_msg** that converts its parameters and transmits the resulting message using the L3 message transmission facilities.

3.4.2 Discussion

Using L3 IPC System Calls

We considered the technique of replacing the many occurences of Mach IPC calls to be cumbersome and error-prone. As described in section 2.2.2 Lites relies on Mach IPC features (especially ports and port rights) that are not present in L3. This adds more complexity to this approach.

Stub Generation Vs. mach_msg Emulation

The generated stubs replace the MIG-generated ones, so we don't need to add an additional layer, and we get rid of the inefficient MIG-generated stubs. Modern stub generator optimization techniques ([14] and [15]) should enable us to gain better performance than in the current MIG-using Mach setups.

Implementing such a sophisticated stub generator is beyond the scope of our work. The authors of the papers mentioned above work on a stub generator framework but the implementation is still in progress, therefore we refrained from using it yet.

The primary advantage of the **mach_msg** emulation approach is that the messages will be intercepted at the lowest level. **mach_msg** is the common point where every message has to pass through. On the other hand this may be a substantial drawback because we have to add this additional emulation layer and each message must be processed by this layer. This can cause a considerable performance loss.

Message Encapsulation Vs. Message Transformation

In the *Message Encapsulation* approach the message to be transmitted contains a verbatim copy of the original message buffer. For some kinds of message components the plain content of the messge buffer is not sufficient (some of the data is only valid in the task's scope, e.g. port names). For these message components it is necessary to create auxiliary information and send this in addition to the verbatim buffer copy.

Message Transformation means that the message is converted at the message component level. Each message component has one (or sometimes) more converted counterparts; no message components share one.

The structure of a message differs substantially between L3 and Mach, so Mach has a great variety of message component types, whereas L3 only has a direct string, indirect strings and dataspaces. So e.g. all Mach integer, character and boolean message component types are mapped to L3 direct strings, and the Mach-specific type information is lost.

The **mach_msg** emulation does not know the structure of the Mach message in advance. In contrast the L3 syscall that waits for the message has to impose upper limits on the amount of message components of each type (direct, indirect etc.), because it has to provide preallocated receive buffers. The encapsulation gives us a structure of the L3 message that does not depend on the structure of the original Mach message, whereas a Mach message transferred via transformation would have to obey some arbitrary restrictions.

In the transformation approach each data element of a Mach message will be converted into a component of an L3 message. This allows more interoperability between programs using this **mach_msg** emulation and other L3 programs. It is questionable whether this improved interoperability is of practical use. In addition L3 requires the components of a message to appear in a certain order, so the transformation has to reorder the message components.

Regarding performance there seems to be no obvious difference, details depend on the actual implementation. The higher complexity of the transformation might cause some degradation, but we have no proof.

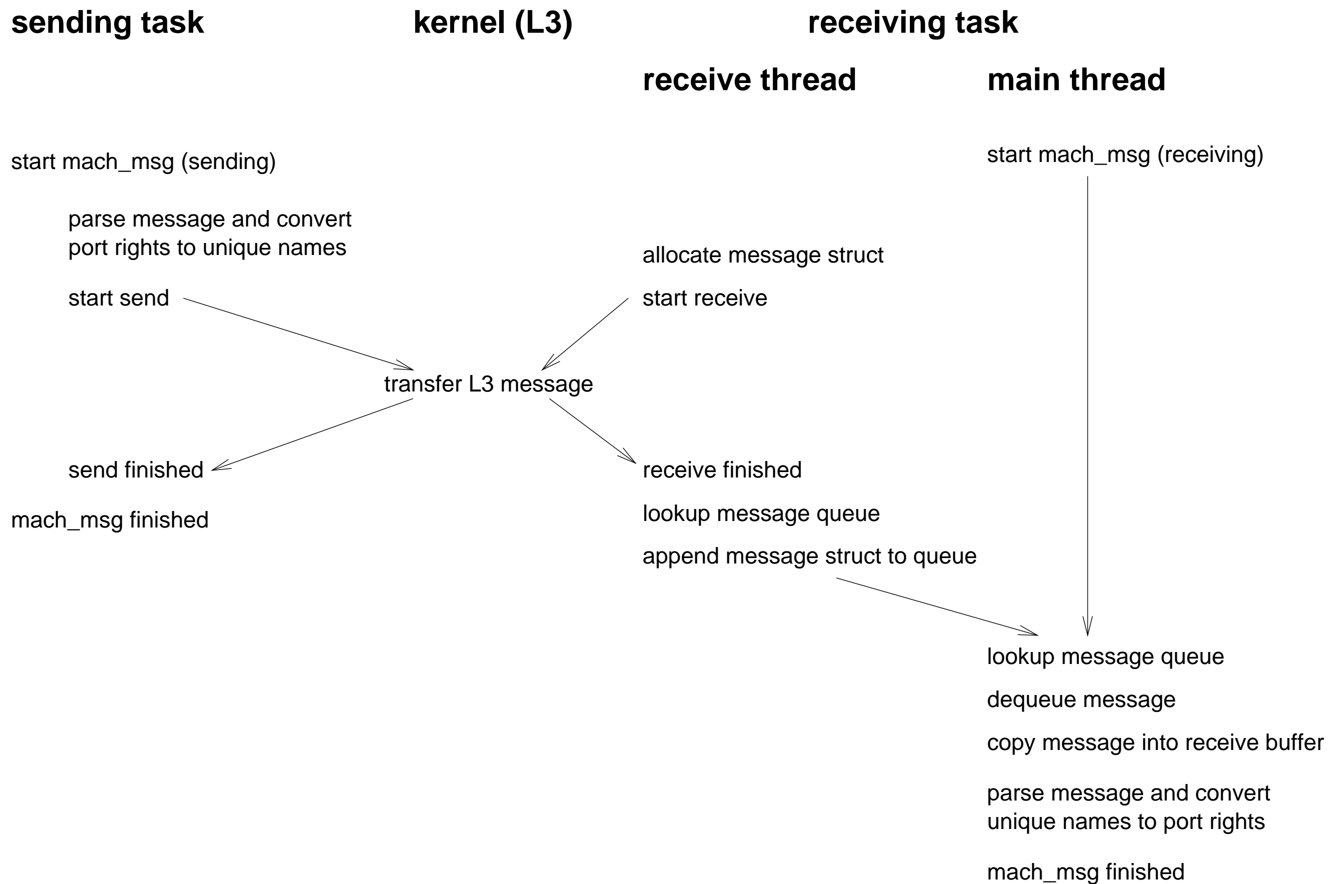

Figure 3.4: Message transmission in the mach_msg emulation.

There seems to be no enormous difference in the amount of data that has to be copied; both methods can take advantage of passing references instead of copying everything.

We decided to implement message encapsulation because the previous decision of writing a **mach_msg** emulation (and the above-described receive-buffer problem) enforced this. The following design describes some details.

An alternative design using a stub generator and message transformation is outlined after that.

3.4.3 Design

A general picture of the design is provided in figure 3.4. Compared to the L3 (figure 2.2) and Mach (figure 2.5) architecture the higher complexity is due to two reasons: the port rights are handled in user space and therefore require additional conversions, and Mach's message queuing is moved into user space. The asynchronity is provided by an additional thread, the port service thread.

Message Encapsulation

An example of a Mach message and its L3 counterpart created by encapsulation is shown in figure 3.5.

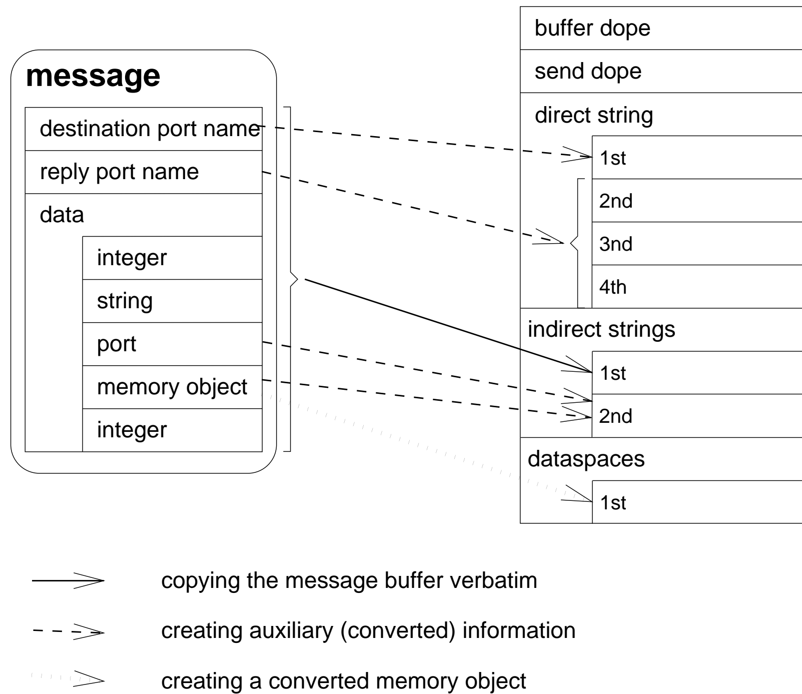

Figure 3.5: Message encapsulation example.

The port names must be converted to global unique names, because port names are only valid inside the task's scope. Currently the size of such a unique name is 12 byte, therefore it is stored in 3 direct string elements. If a unique name were much larger it would be appropriate to use a indirect string instead.[2]

The most similar component for a Mach memory object is a L3 dataspace. This isn't implemented because we didn't emulate Mach's VM subsystem; reasons are described below (section 3.5).

Port Service Thread

In Mach buffering is done in the kernel (see figure 2.5). In L3 the message transfer is synchronous, therefore message buffering and queuing has to be provided in user mode. Similar to the signal handling thread in Lites [12] we designed a port service thread.

This thread receives all encapsulated Mach messages. According to their destination port the messages are copied into the appropriate queue and the thread starts receiving again.

The queues are part of the receiving task and are shared by the port service thread and the main thread. Access to the queues is mediated by a global lock.

The main thread and the port service thread may be synchronized by condition variables. Each message queue has a condition variable that represents the condition that a message is available in this queue. When the main thread invokes **mach_msg**, it finds the associated condition variable and starts waiting for the condition. When the port service thread received and queued a message for this queue it raises a signal for the queue's condition variable. The main thread then wakes up and dequeues the message.

[2] The real implementation is different, the unique name is stored at the beginning of the second indirect string.

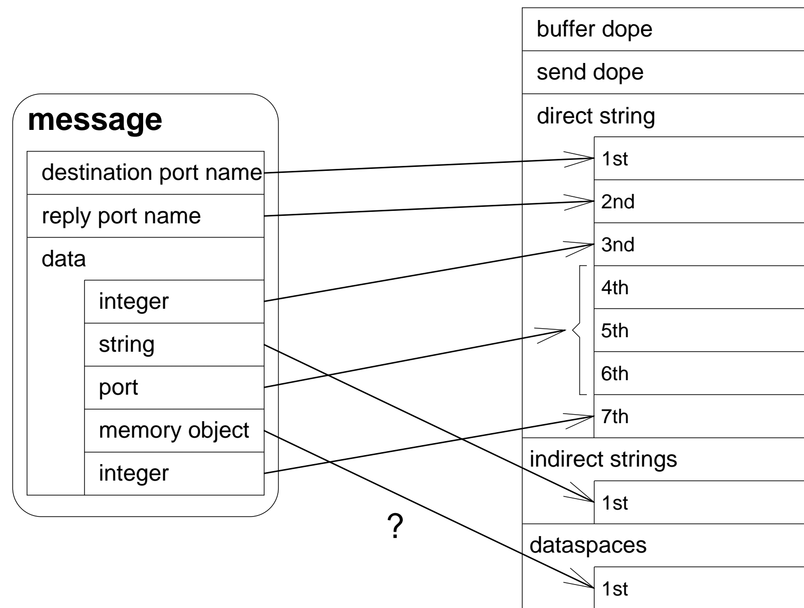

Figure 3.6: Message transformation example.

The receive buffer has to be provided before the message has been received. Therefore it is impossible to choose its size according to the size of the message. The ideal solution might be to allocate a very large buffer and truncate it after the message has been received.

3.4.4 Alternative Design

IPC Stub Generation

A new stub generator replaces MIG and its support library.

Message Transformation

An example of a Mach message and its L3 counterpart created by translation is shown in figure 3.6.

The mapping of the Mach message components to L3 message components is quite straightforward: all integer, character and boolean values are mapped to direct strings, the strings are mapped to indirect strings.

Combining IPC stub generation and Message Transformation

Message transformation causes the resulting messages to have a variable number of message elements. Therefore IPC stub generation should be used. All IPC definitons for one interface (all messages that a thread should be able to receive) must be known to the stub compiler in order to compute the maximal number of each message component types.

Converting messages by transformation is more complex. Stub generation moves this overhead from run-time to compile-time, so the impact on IPC performance is lowered.

3.5 Emulating Mach's VM Subsystem

As we've seen in section 2.1.3, Mach's and L3's VM schemes posses both similarity and differences: While the concepts implemented are quite similar, the methods to interface them differ a lot.

Lites utilizes the following portions of Libmach's VM interface:

memory_object_*, an interface to Mach's external pager mechanisms.

vm_*, an interface to operate on a given task's memory regions.

As both of these interfaces are pretty complex, our general design goal was to emulate only the semantics required by simple RPC-using servers and clients and otherwise to modify Lites to use the much simpler L3 interfaces.

3.5.1 External Pager Interface

In order to allow a Mach memory manager to run on L3 without modification, and a client program to use Mach's **vm_map** interface to attach to a memory manager, emulation glue code would be required to implement solutions to the following problems:

- Mach uses ports to reference memory objects, so the emulation library would have to maintain a mapping between data space numbers and port names to handle page fault messages in the memory manager.
- To the memory manager, the emulation library looks like the Mach kernel. So it would have to implement some heuristics on when to return (with **memory_object_data_return**) pages supplied by the manager. The L3 kernel offers no help because it doesn't tell the manager when a page is evicted from main memory, so the emulation library may end up working *against* the L3 kernel's page eviction algorithm.[3]
- The **vm_map** emulation needs support in the Mach emulation on the memory manager's side: It needs to translate its memory object representative port to a data space number and pager thread id before it can invoke the L3 **DSMapRegion** system call.

All in all, accurate emulation of Mach external pager semantics introduces a large overhead on L3's memory management.

That's why we've decided to do without emulation of Mach's pager interface. Instead, Lites should be modified to employ L3's memory management scheme directly. (The Lites server code which requires modification is well-discriminated already because of a large `#ifdef` which accounts for different implementations for different versions of the Mach kernel.)

3.5.2 Memory Region Manipulation Interface

As we've stated earlier, L3 does not provide a global interface to operate on random virtual memory regions; instead, region management is carried out entirely within memory managers. Thus, clients wishing to change the semantics associated with a given memory region must communicate directly with their memory manager.

Generally, an emulation library running under L3 can not keep track of which memory region is backed by which memory manager, except if the emulated program always uses interfaces

[3]This will not be a problem with the L4 micro kernel: Its more general memory management interface will allow user level pagers to control page eviction.

provided by the library to manipulate memory regions. In the case of random emulated Unix binaries, this can not be guaranteed, so it is a bad idea to rely on it.

That's why we chose to emulate only the most common memory region manipulation routines: **vm_allocate** to allocate some memory from the standard memory manager, and **vm_deallocate** to deallocate previously **vm_allocate**'d memory (a serious restriction with respect to Mach's original **vm_deallocate** semantics, which are to unmap a random memory region, not limited to previously **vm_allocate**'d memory regions).

We designed an Unix-**sbrk**-like interface which provides objects called *heap spaces* (based on data spaces) and a method to sequentially allocate memory blocks from them, and we ported FreeBSD's **malloc** library routine to operate on such heap spaces, forming a *heap*. Our Libmach emulation's initialization code allocates two heaps on startup: One for library-internal data management (e.g. for the port data management layer), and one for serving the emulated program's **vm_allocate** requests.

3.6 Infrastructure

3.6.1 Requirements

The intended implementation language is C, therefore the availability of a C compiler and corresponding tools (linker, usually an assembler) is essential. A way of integrating program parts written in assembler is required in order to implement certain low-level functions not available in C and to test and evaluate optimized code for performance-critical sections.

The development environment must be designed to provide a method for handling private source code files against a common source tree.

Some members of the research group are used to GNU Emacs, so there should be a way for using it as editing framework. A version control system is optional but highly desired in order to coordinate the collaboration.

3.6.2 Programs

The Gcc (the GNU C Compiler) is a high quality C/C++ compiler. Its source code is freely available and it can be built on all major Unix-like operating systems.

Together with the GNU Binutils, a set of assembler, linker, and other tools, it can be used to create a cross compilation environment. This enables us to create programs for an operating system that isn't stable enough yet to host the Gcc/Binutils tool chain itself. When we have a running and stable Unix environment on L3 we can built native versions of these development tools by recompiling them using the cross compiler.

This development tool chain is already successfully used by other operating system research groups, and its quality is even sufficient to be used by commercial Unix vendors.

CVS (the GNU Concurrent Versions System) is a version control system. CVS stores all files in a centralized repository. Each developer has one (or more) private copies of a source tree stored in this repository. During development changes are made in these private files. After testing these changes are submitted to the repository.

3.6.3 Possible Approaches

L3-delivered Tools

L3 provides Gcc version 1.37. This version is quite antiquated but usable.

The source code management was done by synchronizing the private copies manually and transferring the files either by floppy or via a TCP/IP connection and ftp.

CVS must be used in the Unix-based environment. Since version 1.5 CVS has an integrated mechanism for communicating with a CVS server via TCP/IP. A port of the CVS Client to L3 should be feasible, but we cannot foresee the pitfalls-to-be.

DOS-based development

L3 comes with an integrated DOS box. A DOS-based cross development would enable developers to run the cross-development concurrently to L3. Please note that this approach wasn't taken into consideration when the decision was made because a viable option for creating a critical component (the access to a common source tree) was not available at this time. Nevertheless this section shows a route through an alternative approach.

DJGPP is a C/C++ development toolbox consisting of DOS ports of gcc 2.x, GNU Binutils and GNU Make. There exists a version that creates a.out programs, so the steps needed for running its output on L3 should be similar to the ones described for the UNIX-based case. DJGPP uses the DPMI interface, so the DOS facility must provide this. DOSEMU (the DOS emulator for Linux and NetBSD) provides DPMI in an protected operating system, so this need not break security. We expect kernel modifications to be necessary.

A sufficient variety of editors is available for DOS. There is even a DOS port of a recent GNU Emacs.

When the DOS box allows communication via TCP/IP it is possible to run a not-yet-existing port of the CVS client in it. Otherwise we could try to port the CVS client to native L3.

Unix-based cross development

The GNU C Compiler and the GNU Binutils can be used to create a cross development environment. CVS is already available.

The compiled programs must be transferred to a machine running L3. The most automatic way is to transfer these files via FTP.

3.6.4 Decision

We decided to create a Unix-based cross development environment because it required the smallest amount of work and provided the most comfortable development environment.

The Linux a.out binary format was chosen because of its simple but sufficient structure.

(There are plans to use ELF on L3, but the ELF tools are not stable enough yet and ELF is more powerful, and hence complicated, than a.out.)

3.6.5 Running the compiled programs

The first approach was to compile and link the program using the cross tools. A loader was written (by another member of our group) that runs Linux-compiled a.out programs.

Another approach is to link the object files on L3 against the L3-provided Libc. This enables us to use the Libc on L3.

Chapter 4

Implementation

4.1 Libmach Emulation

The Libmach emulation has been written in C. Because of the object-oriented design of the port data management layer, we also considered using C++ for some of the internal parts, but we hesitated because we didn't have any experience with C++ cross compilers. It is well possible to write efficient object-oriented programs using C: The X11 toolkit library is a convincing example.

4.1.1 Ports

We've implemented a subset of Libmach's **mach_port_*** interface. The emulation routines manage a task's local port name space using the port data management layer. It currently is impossible to manipulate a remote task's port name space.

The routines implemented are:

mach_port_allocate and mach_port_allocate_name, routines to create a new port.

mach_port_destroy, a routine to destroy a port object and make its port name available for immediate reuse.

mach_port_deallocate, a routine to deallocate send rights associated with a port.

mach_port_mod_refs, a routine to modify the reference counts of rights associated with a port. [9]

Other routines of the Libmach interface will be implemented when necessary; their implementation is easy and straightforward.

The port data management layer consists of implementations of the classes shows in figure 3.3. All classes implement constructor and destructor routines (if the class is not an abstract class), *init* and *done* methods, routines to change their port name, and a "friend"[1] routine which looks them up (i.e. returns a pointer to them) using a given port name. The higher level classes also contain routines to modify their list of rights, and to look them up using their system-wide name.

The object-oriented flavour of the port data management library is achieved in the same way as in the X11 toolkit library. [16]

[1] "friend" in the sense of C++'s `friend` keyword

The three-phase-commit protocol described in section 3.3.3 hasn't been implemented, yet: Only the special case of transferring send rights of ports the sending tasks holds the receive right for works. Also, port reference counting hasn't been implemented, yet: So far, receive right holders don't get notified when a tasks manipulates reference counts for their send rights. Consequently, dead name notifications and port-deleted notifications also don't work.

4.1.2 IPC

The **mach_msg** emulation provides the original **mach_msg** interface. The following routines are involved in the emulation and encapsulation:

mach_msg, the original Mach wrapper around **mach_msg_trap**. It tries to restart aborted transmissions.

mach_msg_trap, calls **mach_msg_trap_send** and **mach_msg_trap_rcv** according to the header options.

mach_msg_trap_send, handles the whole conversion and encapsulation for the send case.

mach_msg_trap_rcv, handles the whole conversion and decapsulation for the receive case.

The layout of the encapsulated message follows the example given in figure 3.5.

```
struct msg_encapsulated {
  MessageDopeT message_size;
  MessageDopeT message;
  unsigned long un_name;
  int  msgnum;
  StrDopeT machmsg;
  StrDopeT machaux;
};
```

The original message buffer is transmitted in the machmsg field. So the message parsing in **mach_msg_trap_rcv** parses the same structure and is identical to the parsing in **mach_msg_trap_send**.

We only implemented the following port right transfer types: MACH_MSG_TYPE_MAKE_SEND and MACH_MSG_TYPE_MAKE_SEND_ONCE. The long form of message components (msgt_longform) is not supported. The msgt_deallocate flag is not supported, it is already depreciated in Mach p. 29. The msgaux string has a fixed size, because we cannot estimate its size before parsing the whole Mach message. [8]

Condition variables are not available yet, so busy-waiting is used for synchronization. In order to avoid the performance impact of busy-waiting the task yields its processor time (in L3: `ipc_switch`). The receive operation does not manage timouts, hence it always waits until a message has been received.

The port service thread receives messages. It has to provide preallocated buffers before it knows the required size of the buffers, so these buffers have a fixed size too. After the message has been received the buffers could be truncated, but the current memory management doesn't support this. When the receive buffer is too small the receive operation for this message fails.

The data type **mqueue_t** is intended to store received messages in a FIFO.

These routines manipulate message queues:

mqueue_new, a routine to create a new message queue.

mqueue_destroy, a routine to destroy a message queue (and free the allocated memory)

mqueue_queue, a routine to append a message to the message queue.

mqueue_dequeue, a routine to dequeue the first message from the message queue and return this element.

The MIG-generated stubs require some support routines that are part of Libmach. These routines provide allocation/deallocation of memory (**mig_allocate**, **mig_deallocate**), initialization (**mig_init**), port-specific functions and an improved **strncopy**.

These MIG-specific routines were copied from Mach's Libmach and adapted to the L3-specific functions. The memory management routines use **_libmach_malloc** and **_libmach_free** which allocate storage from the library's heap (see below).

4.1.3 Virtual Memory Interface

As explained in section 3.5, we've implemented only a small part of Mach's memory management interface:

vm_allocate, a routine to allocate virtual memory for a task.

vm_deallocate, a routine to release a memory region of a task's address space. [9]

Both routines have several restrictions compared to Mach's implementation, as explained in section 3.5.2; basically, they only provide the functionality of the traditional Unix **malloc/free** interface.

Internally, the library knows about two classes of memory objects:

heap_space, an object built upon an L3 data space, provides a routine to allocate memory blocks in an Unix-**sbrk**-like fashion. The implementation is currently limited to 8 heap_spaces (a compile-time constant).

heap, built upon a heap_space, provides the usual **malloc** and **free** interface. These routines have been ported from FreeBSD.

At startup, the library initialization code allocates two data spaces and creates two heaps upon them: One is used to resolve **vm_allocate** requests, and the other one for library-internal memory management: allocation of port data structures, rights, message buffers, and so on. The size and location within the task's address space of these two heaps are compile-time constants.

4.1.4 Utilities

The Libmach emulation library contains several general utilities and helper routines.

Locking routines To synchronize accesses to common data structures between the application and the port service thread, a simple spin lock implementation has been incorporated. Spin locks are utilized in two applications:

- All data structures of the port data management layer are protected by a single master lock. Routines wishing to access these structures must first acquire the lock and release it afterwards.

- The **heap** routines described above protect themselves from multiple invocation on the same heap by setting a heap-specific lock.

Simple byte string manipulation To ease programming, the library contains implementations of the ANSI C **memset**, **memcmp**, **memcpy** and **strlen** functions.

Debugging aids Debugging on L3 is very difficult because a decent debugger hasn't yet been ported, so we were mainly using just the "**printf** method" and L3's kernel debugger.

To support printing messages on an L3 terminal using L3's native **printf** implementation, our library contains a wrapper which allows linking cross-compiled programs against L3's standard C libraries.

Other debugging aids include macros which allow priority-based tracing of events (using **printf**), a verbose exception handler (able to tell which exception killed a thread), a routine which can invoke the L3 kernel debugger, and assertion/panic routines which can trigger a core dump (which can later be analyzed in the cross development environment).

4.2 C Threads Library

The C Threads Library emulation discussed in section 3.2.2 hasn't been implemented, yet.

Chapter 5

Performance

In order to evaluate the performance of the implementation, we built a few benchmarks which we ran both on Mach and on L3 using our Mach emulation library.

5.1 Measurement Environment

The version of Mach used was UK02pl13 from Utah University running Lites 1.1 (using a FreeBSD base system). On the L3 microkernel we used the standard L3 operating system.

Both operating systems ran on a 66 MHz 486-DX2 machine.

The tests were done when the systems were quiescent. On Mach, we used the BSD `time` command to measure the programs. On L3, we used the `cputime` ELAN procedure.

5.2 Message Passing Benchmark

5.2.1 Environment

In the first series of measurements, two tasks synchronously exchanged a number of Mach messages. Before doing so, each task created a port and copied a send right for the port to the other task.

In the first variation, the messages didn't carry any data. A second version exchanged some unstructured data between the tasks.

The third version was slightly more complicated: Both tasks allocated a number of ports, and sent send rights for these ports to the other task, which immediately deallocated the send rights it received.

In order to relativize the performance achieved by both Mach environments (native and emulated), we also measured L3's raw IPC performance directly using the L3 microkernel's message passing primitives.

5.2.2 Results

The following table summarizes the results. The times given are round-trip times for one message exchange.[1]

[1]For the longest message transfer test, we chose to transmit 4095 longwords (instead of 4096) because that's the largest quantity supported by Mach's "short" message element description type, **mach_msg_type_t**.

message transferred	Mach (ms)	L3 (Mach emul) (ms)	L3 (native) (ms)
Null message	0.401	0.358	0.039
16 longwords	0.407	0.375	0.045
256 longwords	0.604	0.539	0.108
4095 longwords	4.34	4.05	1.43
1 port	0.628	0.689	
16 ports	2.29	4.00	
256 ports	29.69	72.35	

5.2.3 Discussion

As can be seen from the table, for a simple message round trip (null message or message containing uninterpreted data), the **mach_msg** emulation on L3 is somewhat faster than Mach's **mach_msg**, while transferring port rights takes substantially longer in the emulated environment.

Sending uninterpreted data takes advantage of L3's highly optimized IPC path. According to [3], a "null RPC" on L3 (implemented by two message transfers, i.e. in our context, a "null message") can be up to twenty times faster than on Mach. (In our test, we observed a ratio of about 10 for small messages and 3 for larger messages.)

It is obvious that we can't achieve the same performance with our **mach_msg** emulation:

- A null message transfer still copies the Mach message header.
- Message queuing and dequeuing costs extra time. So does message header parsing at both the sender's and the receiver's side.
- Message buffers must be allocated and deallocated.
- On the sender's side, the port name needs to be looked up to find the port's system-wide name; on the receiver's side, the system-wide name needs to be looked up to queue the message, and the local port name must be looked up to dequeue the message.

The performance penalty for transferring a port send right and deallocating it in the receiver on Mach is between about 55μs (for 256 rights) and 110μs (for 1 right), while on L3 it is between about 115μs (16 rights) and 165μs (1 right); in the 256 rights case, it is 140μs. (The round-trip penalties have been divided by 2.)

We attribute the extra cost on L3 to the distributed management of the port name space: To move a port right in the emulated environment, a corresponding port data structure needs to be looked up twice: once in the sender, and once in the receiver. In opposition to that, port right management is Mach is centralized in the kernel.

On Mach, manipulating port rights seems to become faster as more ports are to be manipulated at once. We don't know for sure why this happens; we suspect the kernel is building a cache working set when processing a long list of rights.

On L3, this effect is visible as well, to some degree. However, the transfer cost per port right for 256 ports is larger than for 16 ports. We know of two possible causes for this slowdown:

- In the 256 ports case, manipulation of the port name space takes longer than a time slice, and context switches to other L3 tasks may disrupt cache working sets.

- Our association management implementation used for maintaining the port name space and the space of system-wide port names is somewhat inefficient for large quantities of ports: It utilizes a hash table with only 32 bucket chains and 8 entries per bucket (these are compile-time constants). As these buckets overflow, the library starts allocating (and later deallocating) extra buckets, and the list of bucket entries that must be compared increases; all of these cost extra time.

5.3 RPC Benchmark

5.3.1 Environment

In this test, we used a simple client/server program where client and server (in separate tasks) communicate using MIG-generated RPC stubs. In the first version of this benchmark, the server procedure takes one integer argument which it passed back to the client ("null RPC"). In the second version, an array of integer data is being copied back and forth between the client and the server.

Before starting, the server created a service port and communicated the client a send right for it. Replies to the client were sent through send-once rights generated from a reply port created by the MIG client stub. This is MIG's standard way of setting up an RPC.

5.3.2 Results

The following table summarizes the results. The times given are round-trip times for one RPC. The table also lists the times for the "null message" benchmark from the previous section (5.2).

message transferred	Mach (ms)	L3 (Mach emul) (ms)
Null message	0.401	0.358
Null RPC	0.270	0.485
RPC, 16 longwords	0.294	0.502
RPC, 256 longwords	0.624	0.803
RPC, 4095 longwords	6.65	6.21

5.3.3 Discussion

The first eye-catching result is that on Mach, a null RPC using MIG-generated stubs takes considerably less time than a simple message exchange even though an RPC seems to be doing more work: creating/destroying a send-once right, and the overhead added by the RPC stubs.

This is because the IPC path most commonly used by MIG-generated stubs (request using a send right, passing a send-once right for the reply, only uninterpreted data to transfer, i.e. no ports and no memory objects) has been specially optimized in the Mach kernel to reduce the number of context switches and the complexity of handling Mach-internal data structures. Our emulation library hasn't been optimized for this special case.

For **mach_msg** emulation on L3, the performance decrease compared to simple message passing seems to be about normal, given the cost of inserting and deleting the send-once right for the server's reply (about 165μs) and some extra cost for copying the data in the RPC stubs.

As can be seen from comparing the results of this and the previous benchmark, RPC's transferring larger quantities of data between client and server take longer than the respective message-passing variant in both environments (and despite of the special RPC optimization in Mach). This is due to extra copies of the transmitted data by the MIG-generated RPC stubs.

5.4 Evaluation

The performance of our Mach emulation system lies within expected bounds: For many common cases, we achieve performance similar to or not dramatically worse than Mach. However, there is still room for a lot of optimization:

- The Mach IPC emulation lacks a sophisticated buffer management. (Currently, the port service thread dynamically allocates three buffers in order to receive a message which must be deallocated later.)
- The hash table implementation used to manage global and local port name spaces should be tuned for the number of ports commonly used in a Lites system.

In the case of transferring a lot of port rights via Mach IPC, our emulation library performs notably worse than the original system. In section 3.3.2, we argumented against central port emulation in an external server, saying that this would require sending every message twice. It turned out, however, that the performance loss imposed by the decentral port emulation we implemented is much higher than the cost of an extra message copy. In Lites, sending port rights is a relatively rare operation, so that doesn't matter a lot. However, should a different project require Mach port emulation, this should be taken into consideration.

Chapter 6

Summary and Conclusions

The objective of this work has been to advance our effort of porting the Lites single server to the L3 microkernel. We discussed various steps required to carry out the port, and the problems connected to them.

Emulation of the Mach port abstraction, and Mach IPC emulation With the development and Mach emulation environment we've created, it is possible to run simple single-threaded Mach servers (and clients) on L3. MIG now is the first RPC stub generator for the L3 microkernel.

The IPC and port emulation library works, but needs debugging and profiling. It proved tricky to emulate all facets of Mach IPC, mainly because of a lack of documentation (most Mach programmers seem to use MIG-generated stubs these days and don't care about the trifles of the underlying IPC system), and no doubt there are bugs in there.

Emulation of Mach's virtual memory and external pager interface We found emulating Mach's external pager interface to be an unnecessarily complex task. We estimated that it would be easier to completely replace its use with an adaption to employ L3's VM interface directly.

Because of the perceived complexity of the Mach interfaces, we decided not to extend this work beyond a study of the feasability of porting a Mach server to another microkernel.

We implemented an emulation environment to make IPC-based Mach servers run on L3, and studied the performance which can be expected from such a framework. We found that it is possible to achieve RPC performance similar to Mach's.

In retrospective, the decision to use a Mach single server to start a porting effort towards a Unix single-server system on L3 proved questionable.

There are indications that it is well possible and feasable to port a monolithic Unix system to a microkernel [17]. Compared to a port of Lites from Mach to L3, this approach may have some advantages:

- Judging from the experiences from this work, emulating Mach facilities on a different microkernel is a quite complex task. This becomes unnecessary if not porting a Mach server.
- Unlike Mach, L3/L4 don't have any device drivers built into the microkernel. Porting a monolithic Unix system might enable us to re-use the device drivers that come with the system.

Consequently, our group started a project to port the monolithic Linux kernel to the L4 microkernel.

Bibliography

[1] *J. Liedtke*: A Persistent System in Real Use - Experiences of the First 13 Years; *1993 I-WOOOS*

[2] *J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, G. Szalay*: Two Years of Experience with a μ-Kernel Based OS; *Operating Systems Review, January 1991*

[3] *J. Liedtke*: Improving IPC by kernel design; *1993 SOSP*

[4] *H. Härtig, J. Liedtke, J. Wolter*: Support for Variable Size Pages Through External Pagers; *to appear*

[5] *M. Schalm, J. Wolter, M. Hohmuth*: L3 Documentation; *TUD-internal document*

[6] *J. Wolter*: Emulation des UNIX-Prozeßkonzepts auf dem Mikrokern L3; 1995 *Thesis, Dept. of Computer Science, TU Dresden; Technical report TUD/FI/95/12*

[7] *J. Liedtke*: The L3 Microkernel; *TUD-internal presentation*

[8] *K. Loepere (Editor)*: OSF Mach Kernel Principles, Revision 4; *OSF Mach series, Open Software Foundation and Carnegie Mellon University*

[9] *K. Loepere (Editor)*: OSF Mach Kernel Interfaces, Revision 4; *OSF Mach series, Open Software Foundation and Carnegie Mellon University*

[10] *K. Loepere (Editor)*: OSF Mach Server Writer's Guide, Revision 4; *OSF Mach series, Open Software Foundation and Carnegie Mellon University*

[11] *K. Loepere (Editor)*: OSF Mach Server Library Interfaces Guide, Revision 4; *OSF Mach series, Open Software Foundation and Carnegie Mellon University*

[12] *J. Helander*: Unix under Mach: The Lites Server; 1994 *Master Thesis, Dept. of Computer Science, Helsinki University of Technology*

[13] *W. Metterhausen*: L3 Benutzerhandbuch; 1993 *Accommodat GmbH*

[14] *B. Ford, M. Hibler, J. Lepreau*: Separating Presentation from Interface in RPC and IDLs; 1994 *University of Utah, UUCS-95-018*

[15] *B. Ford, M. Hibler, J. Lepreau*: Using Annotated Interface Definitions to Optimize RPC; 1995 *University of Utah, UUCS-95-014*

[16] *J. McCormack, P. Asente, R. R. Swick, D. Converse (editor)*: X Toolkit Intrinsics – C Language Interface; X Window System; X Version 11, Release 6; 1994 *X Consortium, Inc.*

[17] *François Barbou des Places, Nick Stephen*: Linux on the OSF Mach3 microkernel; 1996 *FSF-sponsored Conference on Freely Distributable Software*