Investigation of the portability of the GNU HURD

Matthew Wilcox

March 11, 1998

Abstract

The GNU HURD is currently only available for the Intel i386 and compatible processors. It was always intended to run on as many architectures as possible. This report examines the portability of the GNU HURD. It initially describes the approach of attempting to port the Mach microkernel, on which HURD is based, to the ARM processor. It then addresses the question of attempting to port HURD to the L4 microkernel.

Contents

Chapter 1

Introduction

1.1 What is portability?

The word portability has many meanings when used with respect to computer programs. Here, I interpret it to be the amount of modification which must be made to a program before it will run natively on a system other than that on which is was written. The original aim of the GNU system was to have a system which would be source-code compatible between many varying machines. Sun's Java system aims to be binary-compatible between systems, but the Free Software Foundation do not have this aim since they believe that source code should be available for all programs.

1.2 The history of HURD

The Free Software Foundation have been working on their GNU[1] operating system since 1985. It was designed to be a replacement for Unix. This had the advantage that when it was released, there would already be a large number

[1]GNU stands for GNU's Not Unix

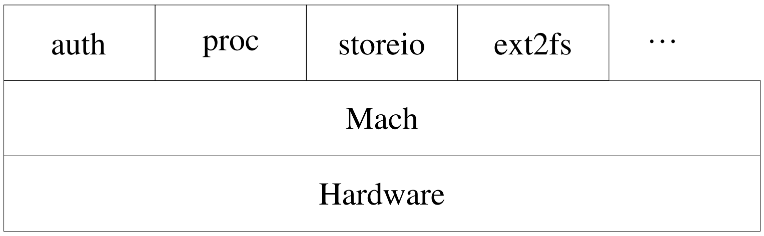


Figure 1.1: Block diagram of HURD.

of people who were able to use the system, and it meant that programmers could work on implementing small pieces of the system independently of each other. Many hundreds of utilities have been released by GNU and they run on a wide variety of Unix and Unix-like platforms, and even some totally unrelated platforms. Because they are so widely available and used, they have received a great deal of testing.

The HURD[2] is the GNU replacement for the Unix kernel. It was first released on the 6th August 1996.

1.3 Features of HURD

HURD provides a Unix-like interface to applications that run under it. HURD is structured as a number of communicating user-level processes that run on top of the Mach microkernel. There is almost no machine dependent code in HURD, so all that is necessary is to port the microkernel that supports it to new hardware. In order to do this, it is first necessary to port the GNU C compiler and the binutils — the assembler, the linker and various related utilities. The structure of the HURD is illustrated in figure 1.1 and discussed further in Section 5.1

[2]HURD stands for HIRD of Unix Replacing Daemons. HIRD stands for HURD of Interfaces Representing Depth

1.4 The goal of this project

The goal of this project is to determine how easy it is to make HURD run on differing architectures. In order to achieve this, I shall attempt to make it work on an architecture that it has never previously run on and one that I am familiar with — the ARM processor. Since HURD is based on Mach, porting the Mach microkernel to the ARM processor would be sufficient to run HURD, but Mach is not currently available for the ARM.

1.5 Report layout

This report starts by discussing microkernel based architectures. I then discuss my attempt to port the Mach microkernel to run on the ARM. Subsequently I examine the possibility of making HURD run on the L4 microkernel.

Chapter 2

Review of Microkernels

2.1 What is a Microkernel?

An Operating System is generally considered to be a set of standard utility programs, plus a kernel which provides services to allow programs to run. The kernel provides abstraction from the hardware and presents a higher level interface to user programs. It also normally provides protection from other tasks and controls communication between tasks.

The fundamental goal of a microkernel-based system is to remove as much functionality as possible from the kernel. As many services as possible should be provided by tasks external to the microkernel.

This has two main advantages, firstly it allows testing and debugging of the kernel to occur in an environment which provides greater functionality.

Secondly, it can allow for multiple operating system 'personalities' to be run concurrently on the same machine. This benefits users by allowing scarce or expensive physical resources to be shared. If this is allowed by the specific microkernel, it can also permit new versions to be tested without disruption to other users of the machine.

The principle disadvantage of the microkernel approach is that it will be slower than a monolithic kernel. This can be minimised by suitable design decisions, and we shall see later how great an effect this has.

2.2 Classification of Kernels

It seems to be possible to classify kernels into three different types. Firstly, there are the monolithic microkernels where all services are provided by the kernel and there is a big distinction between kernel and non-kernel systems. Kernels are traditionally non-pageable for engineering reasons and thus desirable additional features, such as a filing system based on the ftp protocol do not get added as this would lead to unacceptable memory usage by the kernel.

Once we have the concept of a microkernel, the obvious approach is to run an operating system emulator on top of it. Effectively, the microkernel provides a virtual processor which the emulator runs on. This design is called a single-server and these systems form the second category. The microkernel will not be pageable and the single-server may or may not be. It is still monolithic to a great degree however and it will still be very difficult to add functionality to it. Given this base, it is now feasible to run and debug alternative versions of the single-server concurrently with the base one, allowing for faster, more convenient prototyping.

Thirdly, we can take the multiserver approach and split the single-server into a collection of servers. This allows for some servers to be paged out if they are not currently being used while others remain paged in. If one server contains bugs, its ability to affect other tasks is greatly diminished and the problem can be isolated and dealt with much more efficiently. Again, replacement servers can be developed concurrently with a 'production' collection of servers handling everyday usage.

2.3 Examples of Kernel design

There are many examples of the first class of kernel available for study, such as Linux, NetBSD, FreeBSD and OpenBSD. We are not particularly concerned with these in this project.

There are several microkernel designs which lend themselves well to the second category (single-server) approach. Mach with the Lites single-server is probably the most popular example of this type. There is also an ongoing project at GMD[1] to port Linux to the L4 microkernel.

The Chorus [CDK94] operating system was designed for a multi-server approach. In Chorus terminology, the tasks are called actors. In an attempt to improve performance, actors may be co-located with the microkernel and run with kernel privilege. This is because switching from kernel mode to user mode is extremely expensive with some processors, notably the Intel x86 family. One study [Lie95] claims that switching to kernel mode and back to user mode takes 107 cycles with the Intel 486.

Nevertheless, the Chorus microkernel and the Mach microkernel share many similar concepts. Both use the concept of ports to implement interprocess communication (IPC), but they implement protection in a different manner. In Chorus, a 64 bit key is used in addition to a port ID to make it difficult for a malicious actor to send a message to an unsuspecting actor. In Mach, port send rights are administered by the microkernel which allows tasks to revoke send rights from tasks they no longer trust, and additionally to implement send-once rights, where a client may allow a task to send it a reply, but send no more messages. L4 takes a radically different approach by not attempting to implement protection within the microkernel, but to adopt a distributed protection scheme.

All three microkernels have very different ideas about implementing device drivers. Mach retains the device drivers within the microkernel. Chorus removes them into actors, which are co-located with the microkernel, but they can still be developed externally to the microkernel since it presents the

[1] The German National Research Center for Information Technology

same API to the actor. In L4, device drivers are not part of the microkernel at all. They are implemented as processes which claim any interrupts they require and request portions of the memory map which correspond to memory mapped I/O regions. Interrupts are implemented by the microkernel sending a message to the driver which has claimed that interrupt. This is similar to Chorus, but L4 does not require colocation in order to achieve acceptable performance. This is probably due to the extremely efficient IPC in L4.

Writing new device drivers is one of the most common requirements when developing, maintaining and porting kernels, so a system which permits drivers to be developed, tested and debugged more efficiently and safely is extremely useful.

Mach and Chorus were developed in a very different way to L4. Their designers implemented features that they thought would enable people to implement other operating systems on top of the microkernel in an effective manner. Instead, L4 implements what its designers consider to be a minimal set of features that are sufficient to implement an operating system. The designers are then able to expend much more effort on optimising the few remaining operations which are used frequently.

For example, when IPC was designed in L4, the designers calculated the minimum time possible for a message to be delivered, assuming an optimum scenario. They then set themselves a target of double this. A full description of this method can be found in [Lie93].

2.4 Why build multiservers?

Once the decision has been taken to remove functionality from the kernel and place the functionality of the operating system into a user-level task, the next natural inclination is to split the kernel up into separate bits. This has the advantage of shielding one portion of the kernel from another and allows for replacement of part of the kernel while leaving other portions undisturbed. It can also allow for operating systems with vastly different requirements to run on the same machine. For example, a real-time scheduler can coexist with a typical Unix scheduler, allowing non-critical tasks to run when the real-time systems are idle.

There may be additional overheads involved with this approach. In particular if task switching is slow, communication between the different servers involved will be slow.

Chapter 3

The ARM processor

3.1 Overview of the ARM

The ARM CPU is a 32 bit RISC processor originally designed by Acorn Computers Limited. Acorn formed Advanced RISC Machines Ltd to designing future versions of the processor. ARM Ltd now licence the designs to a large number of semiconductor, consumer electronics and other companies worldwide. Details about the ARM710a processor which is typical of the processors currently available may be found in [Adv95].

It has a largely orthogonal instruction set with a load/store architecture. The ARM has 16 general purpose 32-bit registers and a Program Status Register (which contains arithmetic result flags, processor mode and interrupt status) accessible at any time. Some of these registers are shadowed in other processor modes. The processor has an unprivileged user mode (USR) and several privileged modes (SVC, IRQ, FIQ, ABT, UND), the last two of which are not available on ARM CPUs before the ARM6. This presents some problems for virtual memory systems, as we shall see later.

All instructions are conditionally executed, not just branch instructions. This allows for a large reduction in the number of branch instructions required

Number	APCS	Description
r15	pc	program counter
r14	lr	link register
r13	sp	stack pointer
r12	ip	scratch register
r11	fp	frame pointer
r4-r10	v1-v7	variable registers
r0-r3	a1-a4	argument registers

Figure 3.1: The APCS register bindings

which makes pipeline refills much less common.

Although 15 of the 16 registers are general purpose, in order to ease the job of the compiler, an ARM Procedure Call Standard is defined which assigns additional meaning to specific registers. This is shown in Figure 3.1

The program counter is automatically copied to the link register by the ARM when it performs a function call (Branch with Link). The caller of the function places the first 4 arguments into a1-a4 and any further arguments go on the stack. The called function may corrupt all of a1-a4, but must preserve v1-v7, fp and sp. ip may be corrupted by the procedure call before entry to the procedure proper and may not be used as an argument register or to save data over the call.

3.2 Software Interrupts

The ARM has a number of ways to enter privileged modes in order to perform operations which are not permitted to tasks running in USR mode. One of these ways is via a Software Interrupt instruction, or SWI for short which causes the processor to enter SVC mode and jump to address 0x08. This would normally be a branch into the part of the kernel that will take appropriate action. 8 bits of the 32-bit instruction are used for the condition code and to indicate that this is a SWI. This leaves a 24-bit field in this instruction which is not interpreted by the processor, and this can be used to decide what action to take. SVC mode has its own private R13_SVC and R14_SVC. The address of the instruction following the SWI instruction is placed in R14_SVC by the processor. R13_SVC should have been previously set up to point to the kernel stack; this is generally part of the bootstrap code.

3.3 Hardware Interrupts

There are two types of hardware interrupt on the ARM, IRQ and FIQ. FIQ is an abbreviation of Fast Interrupt. The ARM has 2 interrupt lines entering it, one for each interrupt. An IRQ cannot interrupt a FIQ. When an interrupt is signalled on one of these lines, the ARM switches into the corresponding privileged mode and will typically enter the kernel, indirected via 0x18 for IRQ and 0x1C for FIQ. These modes also have their own private registers, R13_IRQ and R14_IRQ; and R8_FIQ to R14_FIQ. Again, R14 is set up by the processor to point to the appropriate return address once the interrupt has been handled.

Devices are multiplexed onto these two lines by the IO controller (IOC in old machines and IOMD in newer machines). In order to find out which device triggered the interrupt, it is necessary to read the status registers from IOC (which is memory mapped). The I/O map is illustrated in Figure 3.2

3.4 Aborts

There is an abort line entering the ARM processor which can be pulled high by an external memory manager when the ARM attempts an illegal access to memory. The ARM has two abort traps, depending on what it was attempting to do when it received the abort. If it was attempting to fetch data from an illegal address, it enters ABT mode and jumps to 0x10, and if it attempts to execute an instruction which is marked as having been from an

Address	Read	Write
0x00	Control	Control
0x04	Keyboard	Keyboard
0x08		
0x0C		
0x10	IRQ A status	
0x14	IRQ A request	Clear IRQ
0x18	IRQ A mask	IRQ A mask
0x1C		
0x20	IRQ B status	
0x24	IRQ B request	
0x28	IRQ B mask	IRQ B mask
0x2C		
0x30	FIQ status	
0x34	FIQ request	
0x38	FIQ mask	FIQ mask
0x3C		

Figure 3.2: Memory map of interrupt sources

illegal address[1] then it enters ABT mode and jumps to 0x0C. In either case, it preserves the return address in R14_ABT.

When the ARM attempts to execute an instruction which it does not understand it enters UND mode, stores the address of the instruction following the undefined one in R14_UND and jumps to address 0x04. This is normally used to implement a software floating point emulator in machines with no floating point hardware. Unfortunately, there is no freely available floating point emulator for the ARM, and this is someting that would need to be implemented.

Versions of the ARM before the ARM6 did not have ABT or UND modes. In the ARM2 and ARM3, when illegal memory accesses or undefined instructions occur, the ARM switches into SVC mode instead. This makes it very difficult to implement a virtual memory system since if the processor is

[1]The abort will not be taken immediately since the abort should not be occur if the aborting instruction enters the pipeline but is not subsequently taken

in SVC mode and it accesses memory which is not currently paged in then R14_SVC, which would normally contain the return address from the system call, will be overwritten with the address of the aborting instruction. To get around this, it is necessary to preserve the return address into a different register before attempting to access any memory, possibly including the SVC stack. Acorn's RISCiX (a derivative of 4.3BSD Unix) works in this manner. Acorn's RISC OS does not bother, and simply does not implement virtual memory. Under RISC OS, it is also not normally permitted to issue floating point instructions while in SVC mode since this will also overwrite R14_SVC.

3.5 Memory Management

The family of ARM processors have been attached to several different memory management systems. Acorn originally designed the MEMC to go with the ARM2, and this was retained for the ARM3. The ARM6 core is available with an MMU in the ARM610 chip, and without an MMU in the ARM60 chip. The MMU in the ARM610, ARM710 and StrongARM can be considered to be roughly equivalent. The MEMC chip is primitive by today's standards and it would be extremely difficult to implement a sophisticated memory management system with the MEMC. I will consider only the intersection of the feature sets of the MMUs contained in the ARM610, ARM710 and StrongARM since this produces a design which is compatible with all current production processors.

The MMU contains a Translation Look-aside Buffer (TLB), access control logic and translation table walking logic. The MMU translates virtual addresses generated by the ARM into physical addresses which are output onto the address lines. Before the MMU is activated, it is necessary to prepare a Translation Table which is 16k of Descriptors. Descriptors allow for either single-indirection (Sections) or double-indirection (Pages). Sections contain a pointer to 1MB of memory, and Page Descriptors contains a pointer to 4k of memory. The advantage of using Sections is that they are quicker to translate and only take 1 entry in the processor's TLB for an entire Megabyte.

When translating an address, the MMU uses the top 12 bits to index the

Translation table. If it finds a Section descriptor, it replaces the top 12 bits with the reference that it finds in the table. If it finds a Page Descriptor, it uses the next 8 bits of the virtual address to index the Page Table that the Page Descriptor points to, which contains the top 20 bits of the new physical address.

Chapter 4

Porting Mach to the ARM

4.1 Goal

The three critical components to getting the GNU system running on new types of machine are GCC (the GNU C Compiler), binutils (the GNU binary utilities) and Mach. GCC and binutils are already widely ported, so the obvious next step is to assess the portability of Mach. Mach as distributed by GNU supports only the i386 (and similar) processors. In order to determine how easy it is to port Mach, I decided to attempt to port it to run on the ARM architecture which I am reasonably familiar with programming.

4.2 History of Mach

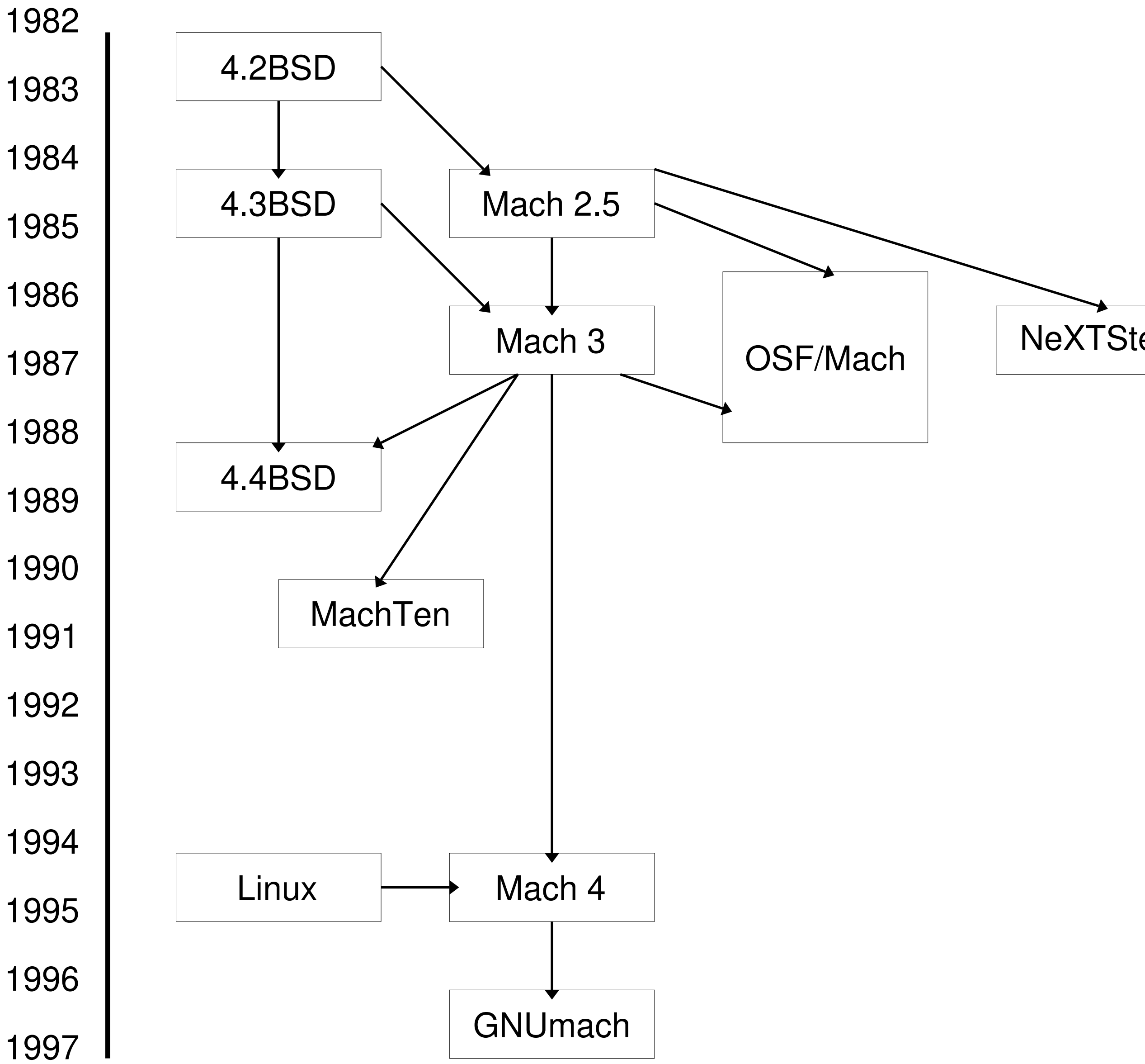
Mach was first described in a paper [ABB+86] to Usenix in 1985. It was developed at Carnegie Mellon University to form the base for their operating system research. It was initially based within 4.2BSD, replacing its components with Mach components as they were completed. When 4.3BSD was released, the remaining BSD components were updated. The first release of Mach, Release 0 took place in 1987. Several more releases followed until

Release 3 in 1990, by which stage the BSD components had all been removed from the server to run as a single-server on top of the 'bare' Mach microkernel. The University of Utah took over development of Mach in 1995 to form the basis of their research into operating systems and added new features.

- Additional device drivers (ported from Linux)
- Migrating Threads
- Kernel Activations
- Presentation/Interface RPC

They released Mach 4 in 1996. They have since continued in their research with a project called Fluke. GNU now distribute a version of Mach that is based upon the Mach4 release from Utah.

Mach has been used as the basis for commercial operating systems; Version 2.5 was used as the basis of the NeXTStep operating system and also the basis of OSF/1 Unix. The OSF still maintain a separate copy of Mach, now based on Mach Version 3. Recently, the OSF have ported Linux to run on top of their version of Mach.


1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
4.2BSD
4.3BSD
Mach 2.5
Mach 3
OSF/Mach
4.4BSD
MachTen
Linux
Mach 4
GNUmach

4.3 Support for the ARM

Most GNU tools support the ARM, including GCC (The GNU C Compiler), binutils and glibc[1].

4.4 ARM support for Mach features

Mach can be thought of as providing a virtual machine to tasks which run on top of it. This section shows how some of the concepts which are part of the virtual machine can be implemented on the ARM architecture.

4.4.1 Tasks and Threads

In Mach, the notion of a process is split into a *task* which is a container for all the resources that are allocated to the process and *threads* which act as points of control for the process. A thread is a very lightweight entity then, consisting only of its register state, some thread-specific communication port rights, scheduling state and any statistics which the kernel is collecting. Tasks contain much more state; they contain threads, have an associated address space, hold a set of port rights and can intercept and system calls made by threads.

4.4.2 Device drivers

There are several possible approaches to the design of the IRQ handler. The one used in Acorn's RISC OS [Aco92] is to simply call the device driver directly, with the processor remaining in IRQ mode and further interrupts disabled. If the device driver thinks it will take an unacceptably long time to execute, it may reenable interrupts, but must then be able to cope with

[1] Currently only in developmental versions

being called reentrantly. A better solution is to use a queue of pending interrupts. In this system, interrupts are left disabled for only very short periods of time during the kernel interrupt handler, the device drivers are called with interrupts enabled and are called in SVC mode instead of IRQ mode. The kernel IRQ handler can then protect the device drivers against being reentered by allowing the existing instance to complete and then calling it when it finishes dealing with the old IRQ.

A consequence of the device driver being called with IRQs enabled means that a priority system is needed to ensure that time critical interrupts are not missed. In order to not lose the benefits of FIQ mode (ie having a lot of registers which do not need saving; plus devices that are connected to the FIQ lines often require very fast interrupt handling, I think a hybrid system is required which allows FIQ routines to execute as in the RISC OS style system, but IRQs to be done in this new fashion.

When an interrupt occurs, it's evidently necessary to save the state of the thread that was interrupted. It is probably sensible to stop the currently running thread and put it back in the pool of runnable threads rather than returning directly to it as a higher priority thread that has been blocking for I/O may now be able to continue.

For an Unix-like OS such as Mach, the sensible solution seems to be to have two halves to device drivers: one half which is called directly when an interrupt is triggered, which performs the time-critical work, such as copying data from a register on the device into a buffer in ordinary RAM.

4.4.3 Clocks

Two of the IRQ sources are timers. These are loaded with an initial value and then count down to zero at a rate of 500ns per tick. When it reaches zero, an interrupt occurs. Clocks are controlled by the kernel in Mach as it must be able to preemptively multitask threads. One design decision which has to be made is how fast to run the clocks — how often to cause interrupts to occur. Acorn's RISC OS provides a centisecond timer and leaves one unallocated for special uses. Mach's clock interface allows for sophisticated control of these clocks. It allows for setting the resolution of the clock in nanoseconds and for reading clocks at nanosecond resolution. Alarms may also be set to wake up a thread at a given (absolute or relative) time.

4.5 Structure of Mach

Internally Mach is notionally organised into machine dependent and machine independent parts. However, there is no documentation about the purpose or function of each file. Additionally, the platform-dependent files within the i386 directory are split by author, not by purpose which makes it extremely difficult to locate the file that is required.

There are approximately 210 object files in a typical build of gnumach[2]. 53 of these are directly from the i386 directory. Unfortunately, this is not the full story since the non-machine specific files also include many header files which contain machine specific data, and in some cases even inline assembler. Assembly code is justifiable (particularly in kernels), but frequently there is no comment against it to indicate what it does and it is unreasonable to expect porters to understand 8086 assembler.

The internal layout is confused. The original build environment used files in a 'dummy' directory to control which features were added to the kernel, and vestiges of this system still remain. Some work has been done to convert the kernel to a GNU-style build environment where options are specified to a configuration script.

One of the major reorganisations that occured between Mach 3 and Mach 4 was that the build environment changed from the machine-dependent parts pulling in machine-specific components to a system where the machine-specific components treat the machine-dependent parts as a library of functions that are used in the cases where there is not a machine-specific function for the job. In theory this makes it easier to produce machine-specific components and aids the understandability of it. Unfortunately, it seems that

[2]this varies depending on which device drivers are selected, and increases if the kernel debugger is enabled

this reorganisation was not completed and there are still many components which work in the old way.

Porting the MIG (Mach Interface Generator) to the ARM is not a hard job, it merely needs to be told about the sizes of certain types — for example, it needs to be told the size of a machine word, and the size of a byte.

Mach pulls in some function from libc rather than providing its own. These functions are `htonl()`, `ntohl()`, `htons()`, `ntohs()`, `memcpy()`, `memset()`, `bcopy()`, `bzero()` and `strstr()`. The easiest way to provide these functions is to build glibc for the appropriate target. Since the build environment is not set up to build for a different processor (referred to as a cross-compilation), some manual tweaking of the Makefile is required.

4.6 Summary of Evaluation

I was unsuccessful in my attempt to port Mach to the ARM. This was due to a number of factors:

- The internal structure of Mach is not sufficiently well documented.
- Mach has a lot of code in it left over from previous incarnations. It needs to be tidied up so it is easier to understand.
- I did not have access to the University computer systems from my room, as I had been led to believe that I would.
- The build structure of Mach is extremely complicated.

I therefore decided to look for alternative methods of running HURD on the ARM.

Chapter 5

Porting HURD

5.1 The Architecture of HURD

In order to port HURD, it is necessary to have some understanding of the structure of HURD. As has previously been stated, HURD consists of multiple servers which cooperate to provide the functionality traditionally provided by the Unix kernel. This is not quite true as some functions are more appropriately provided by the C library. I summarise here the HURD servers and their purpose.

5.2 The HURD filesystem

As in traditional Unix, the filesystem is very important under HURD. Devices are represented by special files in the /dev directory and it is possible to mount filesystems by using the *settrans* command, which sets up a translator[1] on a directory.

[1] HURD sometimes refers to servers as translators when they are being used in this manner

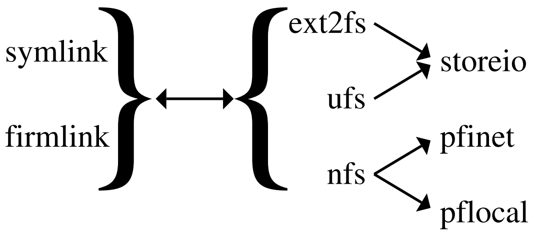


Figure 5.1: Communication between the HURD filesystem components.

A choice of two popular filesystems are available for disc-based filesystems: The *ext2* filing system that was developed for Linux by Remy Card and the *ufs* filing system which has a BSD heritage. The *nfs* filesystem [CPS95] is provided for accessing remote filesystem servers. Both the *ext2* and *ufs* servers use the *storeio* server. In its turn, the *storeio* server communicates with a device directly, or with a file on a filesystem. The *nfs* server communicates with either the *pfinet* or the *pflocal* servers and could be run over other protocol families as servers for them become available.

The *symlink* and *firmlink* servers are both called by and subsequently call the filesystem servers in order to resolve symbolic and firm[2] links. This completes the view of a traditional Unix filesystem that a process has. The communications are summarised in Figure 5.1.

5.3 HURD Processes

HURD processes are Mach tasks. HURD has various servers which provide process management. Servers that fall into this category include *auth* which handles the privileges of processes, *exec* which starts processes and *proc* which reports on the status of processes. The *crash* server handles processes which crash; it allows them to be attached to with a debugger or have images dumped to disc or simply killed off.

[2] A firm link is a concept not found in traditional Unix. It is conceptually half-way between a soft link and a hard link.

5.4 Other servers

Most of the remaining translators may fairly be filed under 'miscellaneous'. For example, there is a *null* server which provides /dev/null and /dev/zero. The *fifo* and *new-fifo* translators provide named pipes and the *ifsock* server provides a BSD-style sockets interface.

There are also some 'toy' servers: *devport*, *fwd* and *magic*. One might be tempted to categorise the NFS server task as a server since many implementations of NFS place the server inside the kernel in an effort to improve performance, but conceptually it lies outside the kernel activities in ordinary Unix so I would consider this wrong. Of course one of the effects of breaking the kernel up in this manner is to blur the distinction between kernel-provided services and user level services.

5.5 Libraries

The HURD distribution does not just contain servers. Since GNU wish the kernel to be easily extensible, they have provided a large number of libraries for programmers to use in order to help them write servers faster.

For example, the *ext2fs* server depends on the HURD libraries *diskfs*, *pager*, *iohelp*, *fshelp*, *store*, *ports*, *threads*, *ihash* and *shouldbeinlibc*.

Some of these libraries do fairly mundane things, for example, *libihash* provides an implementation of hash tables. Since this is supplied with the OS, there is no need for anyone to write their own implementation. *libshouldbeinlibc*, as its name suggests, contains about 40 different functions that are currently missing from libc but logically belong there.

Many are to do with implementing filing systems: *diskfs*, *fshelp*, *netfs* and *trivfs* all help in different ways. The general idea is that the author of a new filing system only needs to implement the parts which are specific to their filing system.

The *threads* library offers the interface to Mach CThreads which used to be provided as part of the Mach 4 distribution but is no longer part of the GNUmach distribution. *ports* provides a higher level interface to Mach ports.

ftpconn manages ftp connections, *pager* exists to help servers act as memory pagers. This is less specialist than it sounds; for example with the `mmap()` function, any filesystem server is acting as a pager. *store* helps tasks use other servers as pagers. *pipe* provides a high-level communications pipe between two processes. *ps* simplifies obtaining process information.

5.6 Other components

The only remaining things in the HURD distribution are essential system programs such as *getty*, *init*, *login*, *ps*, *settrans*, *su*, *vmstat* and so on. Most of these programs do the same as their counterparts in conventional Unix distributions, but they have enhancements to deal with specific HURD concepts. For example, *su* handles multiple simultaneous user IDs whereas Unix only allows a process to have one UID. Other programs have similar purpose to a Unix utility, but have a different name due to the difference syntax required. For example, the command which mounts a partition on the /home directory under Linux is

```
mount -t ext2 /dev/hda2 /home
```

Whereas Under HURD one would use the command

```
settrans /home /hurd/ext2fs /dev/hda2
```

This illustrates the philosophical difference quite nicely: under Linux, the mount command tells the kernel to mount a filesystem on /home, of type ext2, that it will find on the block device /dev/hda2. The HURD settrans command modifies the /home inode so that when /home is accessed, it starts a new *ext2fs* server with the given parameters. In this case, *ext2fs* will then start up a new *storeio* server to access the hda2 block device on its behalf.

Chapter 6

The L4 Microkernel

6.1 Why L4?

There are other good reasons for finding an alternative microkernel to Mach.

1. It's very big

   The precompiled kernel distributed with the 0.2 release of the GNU system is 1162k. Admittedly, this contains a large number of device drivers and it is possible to build a kernel specific to a particular machine which will contain only the appropriate device drivers and so will be smaller. However, the Linux (version 2.0.33) kernel I have on my PC is only 1055k, and that includes most of the functionality which must then be deployed by additional servers on top of Mach.

2. It is too slow

   Recent measurements [HHL+97] show the performance of Mach to be dramatically less than previously reported. In particular, comparing IPC performance between monolithic and Mach-based systems shows closer to 50% performance rather than the 90% often claimed.

3. It is too complicated

> Mach contains in excess of 200 system calls. The semantics of each call are complex and several of them would be better implemented without (direct) kernel intervention. For example, many of the vm_* interfaces could be directly implemented as an IPC from the task to its memory manager. The OSF Mach Kernel Interfaces document is in excess of 450 pages.

The L4 microkernel seems to suffer from none of these faults. L4 running on the x86 family is under 32k and outperforms Mach by a significant factor. It contains just 7 system calls and the reference manual is only 50 pages. It does not contain a default memory pager as Mach does, but I do not see this as a disadvantage since different operating systems have such different requirements for a pager that they normally provide their own in any case.

L4 is available for the Intel 486 (and compatible CPUs) and MIPS processors. A version for the DEC Alpha is in development. Some interest has been expressed in a version for the ARM.

6.2 Features of L4

The primary purpose of microkernels that are designed as bases for multi-server style operating systems is efficient and secure message-passing.

Mach and L4 have two significant differences between their IPC methods. First, Mach uses asynchronous message passing, which means that the kernel must buffer data (potentially large quantities of it). L4 uses synchronous message passing which involves much less work for the kernel.

Secondly, Mach has a centralised structure for security, where the kernel enforces the 'send rights' through a mechanism known as ports. L4 distributes security to external tasks through a mechanism known as clans. This is first discussed in [Lie92].

Tasks are organised into Clans with Chiefs. Within a Clan, the only protection is that imposed by the individual task based on the sender's ID (which is

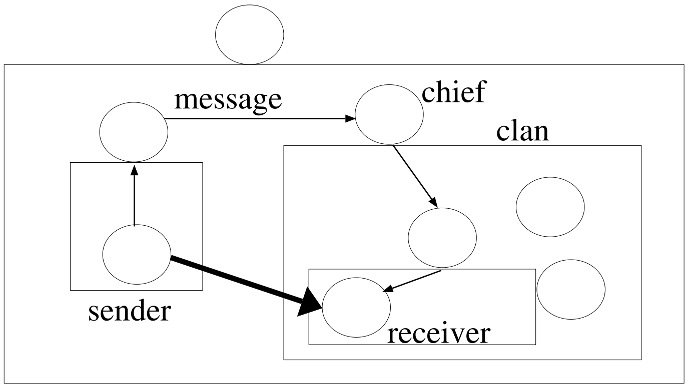


Figure 6.1: Message transmission between tasks in different Clans.

enforced by the microkernel) but between Clans, each Clan boundary that the message crosses incurs inspection and possible rejection by the Clan Chief. Clans are nestable, so a hierarchy of protection can be built. An example of message transmission is shown in Figure 6.1. Here, the rectangles represent Clans and the circles represent tasks. The thick arrow represents the message that is sent and the thin arrows represent the messages which are actually passed.

In a centralised system, the kernel is responsible for administering port rights which adds significant overhead to IPC calls. This conflicts directly with the requirement that IPC be fast. Additionally, it is philosophically superior since the point of a microkernel is to remove as many features as possible from kernel space. It does not harm speed when communication is intra-clan and simply multiplies the time taken by the number of clans traversed when communication is inter-clan. It should also scale better than an in-kernel regulated protection scheme since the protection mechanisms may be chosen on an arbitrary basis and changed arbitrarily frequently without requiring communication with the kernel. Transparent multiple node communication may be achieved using the clan mechanism since the task sees no difference between communicating with a task on a different machine and a task on the same machine in a different clan. In either case, the message is intercepted and potentially modified by the clan's chief.

6.3 Memory management in L4

Mach has an intricate memory management system which allows the task to communicate with its pager in great detail. L4 has no such interface. It has an extremely simple handler for the physical memory called σ_0. This provides no additional paging facility. It is intended to grant all of the available physical pages to a more sophisticated higher-level pager which is referred to as σ_1.

I do not see the advantage in placing σ_0 outside the kernel. It requires that the kernel pass considerable information about the physical state of the machine to σ_0 (though this is achieved in an efficient manner). It is necessary to define an IPC protocol to access σ_0 as part of the kernel definition as otherwise the task of writing σ_1 would be impossible. The σ_0 protocol definition notes that 'Special σ_0 implementations may extend this protocol' which is unwise in my opinion since it could lead to incompatible implementations.

Conceptually then, σ_0 may be considered to be part of the kernel. The only advantage to having σ_0 separate to the kernel is that it allows for separate compilation of σ_0 which may be convenient in certain situations. The design of σ_0 is such that it will not be required after the initial OS bootstrap, except to refuse requests for any further memory allocation. It might lead to a more efficiemt implementation to put σ_0 inside the kernel and have a system for removing initialisation code from the kernel as recent Linux kernels do.

6.4 Device drivers in L4/ARM

If L4 is ported to the ARM then device drivers present an interesting problem. On the ARM, there are only two types of interrupt, normal and fast. It is necessary to interrogate the I/O controller to determine which device caused the interrupt. This is not a problem as such, it is reasonable for the kernel to de-multiplex the interrupt and expose an interface to the drivers that masks this, but the real problem is that all the expansion card interrupts are multiplexed onto one of the I/O controllers lines. When that interrupt is triggered, each expansion card must be interrogated in turn to see if it caused the interrupt. There is a standard way for expansion cards to tell the kernel how to find out if their interrupt has been triggered, but not all cards support this method. For details, see [Aco92, page 4-126]

This is soluble in Mach — since the device drivers are in-kernel, interrupts can be passed around the built-in drivers until one claims it. However in L4, this is somewhat more difficult. Since the device drivers are outside the kernel, it is not possible for the kernel to tell *a priori* which expansion card has caused the interrupt. Unfortunately, L4 allows only one thread to be the recipient of any given interrupt.

In my opinion, L4 should be modified to allow an interrupt to be shared — ie the interrupt should be delivered to all of the threads which have requested it. It is then necessary to have a further protocol which permits the thread to tell the kernel whether or not it has dealt with the interrupt or wishes it to be passed on to other claimants.

However, there is a security problem with this. A thread needs no particular right to associate with an interrupt. Since it is already determined that security shall lie outside the kernel, it makes no sense to make an exception to this rule for device drivers. Any solution ought to be formulated in terms of clans and chiefs. Unfortunately, the kernel is an exception to the clans mechanism. Messages that are sent directly to the kernel bypass all chiefs. I consider this to be a flaw in the implementation of L4.

Another potential solution to this problem is for L4 to treat subsequent claimants of the interrupt specially, and pass them to the first claimant for checking, as if it were the chief for this particular thread. However, this idea is also flawed since the protection it provides can be circumvented by the following sequence:

1. A second (malicious) thread claims the vector and is approved.
2. The first thread is killed in order to be replaced by an improved version. The second thread then becomes the primary thread.
3. The second thread may now deny service to the replacement for the

first thread.

The only viable solution to this problem in terms of the current operation of L4 is to have a task external to the kernel which device drivers register themselves with.

6.5 HURD on L4

The HURD distribution contains a large number of libraries. This was a design decision taken early on, since it was thought likely that as a large number of similar services would be desired, abstracting as much as possible into libraries was a good idea. One of the libraries in the HURD distribution is *LibMOM* (Microkernel Object Module) which provides an abstraction layer between HURD processes and the underlying microkernel. The intention of this library is that to port HURD from one microkernel to another it should only be necessary to rewrite *LibMOM*.

However, careful examination of the sources show that none of the components of HURD currently use *LibMOM*. So the first step in porting HURD to any other microkernel must be to alter the various servers to use the *LibMOM* indirection layer.

6.5.1 Memory Management

The `vm_allocate()` Mach system call is replaced by the *LibMOM* functions `mom_allocate_memory()` and `mom_allocate_address()`. However, these functions only allow for allocating memory in the current task, whereas Mach's `vm_allocate()` allows tasks to allocate memory into the address space of another task. Unfortunately, HURD does use this feature of Mach and there is no defined *LibMOM* function to transfer memory from one task to another. It is not used frequently, of the 98 calls to `vm_allocate()`, only 15 do not refer to the invoking task. For example, the *exec* server allocates memory to the task that it is starting using `vm_allocate()`.

If a microkernel has external memory managers, then it must be possible for one task to give memory to another task. However, the precise procedure for this is likely to vary from kernel to kernel, so I would propose that a new call is required for *LibMOM*.

Some of the current *LibMOM* calls are actually common combinations of other calls. Whether a combined call is required that allocates memory to a different task is a question that could only be answered by profiling a system that did not have it and comparing it to one that does.

6.5.2 Interprocess Communication

The other main microkernel provided service is interprocess communication, normally abbreviated to IPC. Much of the IPC in Mach-based operating systems is already abstracted away from the raw Mach_Msg interface by MIG, the Mach Interface Generator. It is similar in action to Sun's rpcgen program in that it takes a high level representation of services provided into client and server stubs which can be linked against by ordinary programs. The operating systems group at Utah have written Flick [EFF+97] which is intended to provide a replacement for many different generators of this sort, including MIG and rpcgen.

I don't think it is worth investigating porting MIG to generate L4 calls, since Flick would provide a much better basis for emulating Mach-style IPC. Flick generates code that is 'between 2 and 17 times faster' [EFF+97] than other generators. Flick already supports interface descriptions written in CORBA, ONC RPC and MIG, and will generate stubs for IIOP, ONC/TCP, Mach ports or Fluke IPC. The authors claim that it is extremely flexible and extensible so it should not be hard to provide a back end that generates L4 calls.

6.5.3 Emulating Mach

The alternative approach taken in a project described in [HR96] is to provide an emulation of LibMach which provides a veneer over the Mach kernel. The conclusion of that report is that providing a Mach emulation on top of another microkernel is unnecessarily complicated and it is probable that altering the overlying operating system to work with L4 directly would be significantly faster.

This does not necessarily mean that a common microkernel abstraction layer such as that which *LibMOM* attempts to provide is going to be inefficient. Much of the overhead associated with the LibMach approach was consumed in emulating the exact semantics of Mach. This would not apply to *LibMOM* since it implements very simple primitives.

6.6 Experimental Evaluation

I was not able to perform any experimemts of my own, but the results for running Linux on the L4 kernel mentioned in [HHL+97] look promising for the performance problem associated with Mach. Additionally, they state that it took '14 engineer-months' to port the monolithic Linux kernel to run on L4.

Unfortunately, I am not able to persuade GNUMach to run on my computer, due to a bug in the device driver for my Adaptec SCSI card. I was therefore not able to test my modifications. I hope to do so at some stage in the future when a replacement driver appears.

There are many shortcomings in the *LibMOM* API compared to the Mach API. *LibMOM* evidently requires more work before HURD can be fully abstracted fom the Mach microkernel. In order to achieve its goal of abstraction from any particular microkernel, it must abstract all the services which HURD requires. It does not currently even attempt to deal with handling threads, clocks and most importantly, it has no interface which deals with access control or other security mechanism.

It could be argued that this type of interface should not be added to *Lib-MOM*; instead in order to port HURD to a new microkernel, *libthreads* and *libports* should be ported. I would disagree with this because it would then leave many libraries which had to be rewritten instead of one, which would make the task of porting HURD less clear.

Chapter 7

Project Evaluation

7.1 Conclusions

HURD is designed to be the kernel of the GNU Operating System. It is crucial that it runs on machines of as many different types as possible. Unfortunately, Mach is extremely difficult to port to new architectures. Due to the lack of performance which Mach currently exhibits, there is little or no incentive to port it to new architectures. So while HURD continues to run on Mach, it is unlikely to run on many types of machine.

There is hope for HURD. It seems to be possible to abstract the component servers away from the Mach interface. If this is done then it will allow the replacement of Mach with L4, or possibly another microkernel.

7.2 Further Work

During the course of this project, I identified a number of things that I would like to do. These are not all directly related to the GNU HURD and some of them would make interesting projects in their own right.

- Flick should be examined to determine how easy it is to modify it to generate L4 IPC calls.
- A followup project could investigate porting L4 to alternative processors.
- Further work should be done on abstracting HURD's use of port rights away from Mach and towards a more generic approach which would allow for it to be ported between microkernels more easily.
- Some work should be done towards providing a floating point emulator for the ARM. This would benefit Linux/ARM and NetBSD/ARM as well as Mach or L4.

Bibliography

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, 1986.

[Aco92] Acorn Computers Limited. *RISC OS 3 Programmer's Reference Manual*, December 1992.

[Adv95] Advanced RISC Machines Ltd. *ARM710a Preliminary Data Sheet*, ARM DDI 0022D edition, July 1995.

[CDK94] Coulouris, Dollimore, and Kindberg. *Distributed Systems Concepts & Design*. Addison-Wesley, 2nd edition, 1994.

[CPS95] Brent Callaghan, Brian Pawlowski, and Peter Staubach. RFC 1813. Technical report, Internet Engineering Task Force, June 1995.

[EFF+97] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 44–56, New York, June15–18 1997. ACM Press.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Lidtke, Sebastian Schönberg, and Jean Wolter. The Performance of μ-Kernel-Based Systems. In *Proceedings of the sixteenth ACM Symposium on Operating System Principles*, volume 31-5, December 1997.

[HR96] Michael Hohmuth and Sven Rudolph. Steps Towards Porting a Unix Single Server to the L3 Microkernel. Technical report, Dresden University of Technology, April 1996.

[Lie92] Jochen Liedtke. Clans & Chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305. Springer, 1992.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In Barbara Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 175–188, New York, NY, USA, December 1993. ACM Press.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 237–250. ACM Press, December 1995.