Mach: A New Kernel Foundation For UNIX Development

Mike Accetta, Robert Baron, William Bolosky, David Golub,
Richard Rashid, Avadis Tevanian and Michael Young
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pa. 15213

Abstract

Mach is a multiprocessor operating system kernel and environment under development at Carnegie Mellon University. Mach provides a new foundation for UNIX development that spans networks of uniprocessors and multiprocessors. This paper describes Mach and the motivations that led to its design. Also described are some of the details of its implementation and current status.

1 Introduction

Mach[1] is a multiprocessor operating system kernel currently under development at Carnegie-Mellon University. In addition to binary compatibility with Berkeley's current UNIX[2] 4.3BSD release, Mach provides a number of new facilities not available in 4.3:

- Support for multiprocessors including:
  - provision for both tightly-coupled and loosely-coupled general purpose multiprocessors and
  - separation of the process abstraction into *tasks* and *threads*, with the ability to execute multiple threads within a task simultaneously.
- A new virtual memory design which provides:
  - large, sparse virtual address spaces,

[0]This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864, monitored by the Space and Naval Warfare Systems Command under contract N00039-85-C-1034.

[1]Mach is *not* a trademark of AT&T Bell Laboratories (so far as we know).

[2]UNIX, however, *is* a trademark of AT&T Bell Laboratories.

  - copy-on-write virtual copy operations,
  - copy-on-write and read-write memory sharing between tasks,
  - memory mapped files and
  - user-provided backing store objects and pagers.

- A capability-based interprocess communication facility:
  - transparently extensible across network boundaries with preservation of capability protection and
  - integrated with the virtual memory system and capable of transferring large amounts of data up to the size of an address space via copy-on-write techniques.

- A number of basic system support facilities, including:
  - an internal adb-like kernel debugger,
  - support for transparent remote file access between autonomous systems,
  - language support for remote-procedure call style interfaces between tasks written in C, Pascal, and CommonLisp.

The basic Mach abstractions are intended not simply as extensions to the normal UNIX facilities but as a new foundation upon which UNIX facilities can be built and future development of UNIX-like systems for new architectures can continue. The computing environment for which Mach is targeted spans a wide class of systems, providing basic support for large, general purpose multiprocessors, smaller multiprocessor networks and individual workstations (see figure 1. As of April 1986, all Mach facilities, with the exception of threads, are operational and in production use on uniprocessors and multiprocessors by both individuals and research projects at CMU. In this paper we describe the Mach design, some details of its implementation and its current status.

2 Design: an extensible kernel

Early in its development, UNIX supported the notion of objects represented as file descriptors with a small set of basic operations on those objects (e.g., read, write and seek) [9]. With pipes serving as a program composition tool, UNIX offered the advantages of simple implementation and extensibility to a variety of problems. Under the weight of changing needs and technology, UNIX has been modified to provide a staggering number of different mechanisms for managing objects and resources. In addition to pipes, UNIX versions now support facilities such as System V streams, 4.2 BSD sockets, pty's, various forms of semaphores, shared memory and a mind-boggling array of ioctl operations on special files and devices. The result has been scores of additional system calls and options

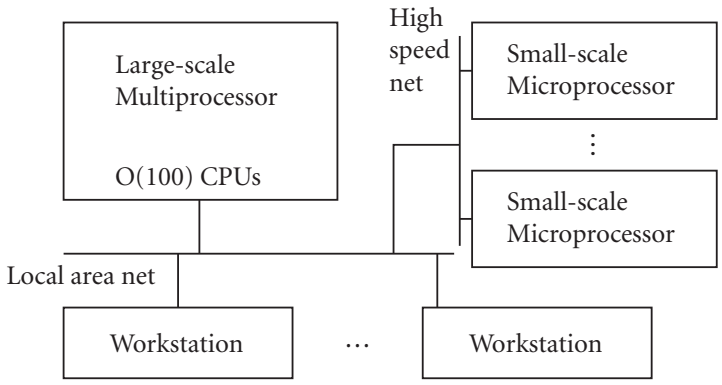

Figure 1: The Mach computing environment

with less than uniform access to different resources within a single UNIX system and within a network of UNIX machines.

As the complexity of distributed environments and multiprocessor architectures increases, it becomes increasingly important to return to the original UNIX model of consistent interfaces to system facilities. Moreover, there is a clear need to allow the underlying system to be transparently extended to allow user-state processes to provide services which in the past could only be fully integrated into UNIX by adding code to the operating system kernel.

Mach takes an essentially object-oriented approach to extensibility. It provides a small set of primitive functions designed to allow more complex services and resources to be represented as references to objects. The indirection thus provided allows objects to be arbitrarily placed in the network (either within a multiprocessor or a workstation) without regard to programming details. The Mach kernel abstractions, in effect, provide a base upon which complete system environments may be built. By providing these basic functions in the kernel, it is possible to run varying system configurations on different classes of machines while providing a consistent interface to all resources. The actual system running on any particular machine is a function of its servers rather than its kernel.

The Mach kernel supports four basic abstractions:

1. A *task* is an execution environment in which threads may run. It is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources (such as processors, port capabilities and virtual memory). The UNIX notion of a process is , in Mach, represented by a task with a single thread of control.

2. A *thread* is the basic unit of CPU utilization. It is roughly equivalent to an independent program counter operating within a task. All threads

within a task share access to all task resources.

3. A *port* is a communication channel – logically a queue for messages protected by the kernel. Ports are the reference objects of the Mach design. They are used in much the same way that object references could be used in an object oriented system. *Send* and *Receive* are the fundamental primitive operations on ports.

4. A *message* is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain pointers and typed capabilities for ports.

Operations on objects other than messages are performed by sending messages to ports which are used to represent them. The act of creating a task or thread, for example, returns access rights to the port which represents the new object and which can be used to manipulate it. The Mach kernel acts in that case as a server which implements task and thread objects. It receives incoming messages on task and thread ports and performs the requested operation on the appropriate object. This allows a thread to suspend another thread by sending a suspend message to that thread's *thread port* even if the requesting thread is on another node in a network.

The design of Mach draws heavily on CMU's previous experience with the Accent [8] network operating system, extending that system's facilities into the multiprocessor domain:

- the underlying port mechanism for communication provides support for object-style access to resources and capability based protection as well as network transparency,

- all systems abstractions allow extensibility both to multiprocessors and to networks of uniprocessor or multiprocessor nodes,

- support for parallelism (in the form of tasks with shared memory and threads) allows for a wide range of tightly coupled and loosely coupled multiprocessors and

- access to virtual memory is simple, integrated with message passing, and introduces no arbitrary restrictions on allocation, deallocation and virtual copy operations and yet allows both copy-on-write and read-write sharing.

The Mach abstractions were chosen not only for their simplicity but also for performance reasons. A performance evaluation study done on Accent demonstrated the substantial performance benefits gained by integrating virtual memory management and interprocess communication. Using similar virtual memory and IPC primitives, Accent was able to achieve performance comparable to UNIX systems on equivalent hardware [3].

3 Tasks and Threads

It has been clear for some time that the UNIX process abstraction is insufficient to meet the needs of modern applications. The definition of a UNIX process results in high overhead on the part of the operating system. Typical server applications, which use the fork operation to create a server for each client, tend to use far more system resources than are required. In UNIX this includes process slots, file descriptor slots and page tables. To overcome this problem, many application programmers make use of coroutine packages to manage multiple contexts within a single process (see, for example, [2]).

With the introduction of general purpose shared memory multiprocessors, the problem is intensified due to a need for many processes to implement a single parallel application. On a machine with N processors, for example, an application will need at least N processes to utilize all of the processors. A coroutine package is of no help in this case, as the kernel has no knowledge of such coroutines and can not schedule them.

Mach addresses this problem by dividing the process abstraction into two orthogonal abstractions: the *task* and *thread*. A task is a collection of system resources. These include a virtual address space and a set of port rights. A thread is the basic unit of computation. It is the specification of an execution state within a task. A task is generally a high overhead object (much like a process), whereas a thread is a relatively low overhead object.

To overcome the previously mentioned problems with the process abstraction, Mach allows multiple threads to exist (execute) within a single task. On tightly coupled shared memory multiprocessors, multiple threads may execute in parallel. Thus, an application can use the full parallelism available, while incurring only a modest overhead on the part of the kernel.

Operations on tasks and threads are invoked by sending a message to a port representing the task or thread. Threads may be created (within a specified task), destroyed, suspended and resumed. The suspend and resume operations, when applied to a task, affect all threads within that task. In addition, tasks may be created (effectively forked), and destroyed.

Tasks are related to each other in a tree structure by task creation operations. Regions of virtual memory may be marked as inheritable read-write, copy-on-write or not at all by future child tasks. A standard UNIX fork operation takes the form of a task with one thread creating a child task with a single thread of control and all memory shared copy-on-write.

Application parallelism in Mach can thus be achieved in any of three ways:

- through the creation of a single task with many threads of control executing in a shared address space, using shared memory for communication and synchronization,
- through the creation of many tasks related by task creation which share restricted regions of memory or
- through the creation of many tasks communicating via messages

These alternatives reflect as well the different multiprocessor architectures to which Mach is targeted:

- uniform access, shared memory multiprocessors such as the VAX[3] 11/784, VAX 8300 and Encore MultiMax[4],
- differential access shared memory machines such as the BBN Butterfly and IBM RP3,
- loosely-coupled networks of computers.

In fact, the Mach abstractions of task, thread and port correspond to the physical realization of many multiprocessors as nodes with shared memory, one or more processors and external communication ports.

4 Virtual Memory Management

The Mach virtual memory design allows tasks to:

- allocate regions of virtual memory,
- deallocate regions of virtual memory,
- set the protections on regions of virtual memory,
- specify the inheritance of regions of virtual memory.

It allows for both copy-on-write and read/write sharing of memory between tasks. Copy-on-write virtual memory often is the result of form operations or large message transfers. Shared memory is created in a controlled fashion via an inheritance mechanism. Virtual memory related functions, such as pagein and pageout, may be performed by non-kernel tasks. Mach does not impose restrictions on what regions may be specified for these operations, except that they be aligned on system page boundaries (where the definition of the page size is a boot-time parameter of the system).

The way Mach implements the UNIX fork is an example of Mach's virtual memory operations. When a fork operation is invoked, a new (child) address map is created based on the old (parent) address map's inheritance values. Inheritance may be specified as *shared*, *copy* or *none*, and may be specified on a per-page basis. Pages specified are shared, are shared for read and write access by both the parent and child address maps. Those pages specified as copy are effectively copied in the child map, however; for efficiency, copy-on-write techniques are typically employed. An inheritance specification of none signifies that the page is not passed to the child at all. In this case, the child's corresponding address is left unallocated. By default, newly allocated memory is inherited copy-on-write.

[3] VAX is a trademark of Digital Equipment Corporation.
[4] MultiMax is a trademark of Encore Computer.

Like inheritance, protection may be specified on a per-page basis. For each group of pages there exist two protection values: the current and maximum protection. The current protection controls actual hardware permissions. The maximum protection specifies the maximum value that the current protection may take. The maximum protection may never be raised, it may only be lowered. If the maximum protection is lowered to a level below the current protection, the current protection is also lowered to that level. Either protection is a combination of read, write, and execute permissions. Enforcement of these permissions is dependent on hardware support (for example, many machines do not allow for explicit execute permissions, but those that do will be properly enforced).

Consider the following example: Assume that a task with an empty address space has the following operations applied to it:

Operation	Arguments	Comments
allocate	`0-0x100000`	allocate from 0 to 1 megabyte
protect	`0-0x10000` read/current	make 0-64K read only
inherit	`0x8000-0x20000` share	make 32K - 128K shared on fork

The resulting address map will be a one megabyte address space, with the first 64K read-only and the range from 32K to 128K will be shared by children created with the fork operation.

An important feature of Mach's virtual memory is the ability to handle page faults and page-out data requests outside of the kernel. When virtual memory is created, special paging tasks may be specified to handle paging requests. For example, to implement a memory mapped file, virtual memory is created with its *pager* specified as the file system. When a page fault occurs, the kernel will translate the fault into a request for data from the file system.

Mach provides some basic paging services inside the kernel. Memory with no pager is automatically zero filled, and page-out is done to a default pager. The current default pager utilizes normal file systems, eliminating the need for separate paging partitions.

5 Virtual Memory Implementation

Given the wide range of virtual memory management built by hardware engineers, it was important to separate machine dependent and machine independent data structures and algorithms in the Mach virtual memory implementation. In addition, the complexity of potential sharing relationships between tasks dictated clean separation between kernel data structures which manage physical resources and those which manage backing store objects.

The basic data structures used in the virtual memory implementation are:

address maps: doubly linked lists of map entries, each entry describing the properties of a region of virtual memory. There is a single address map associated with each task.

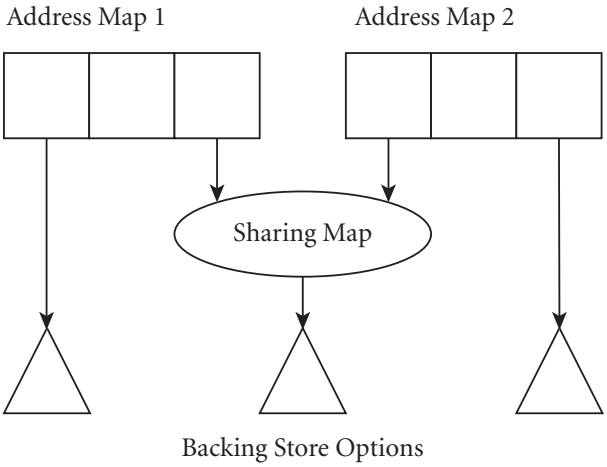

Figure 2: Task address maps

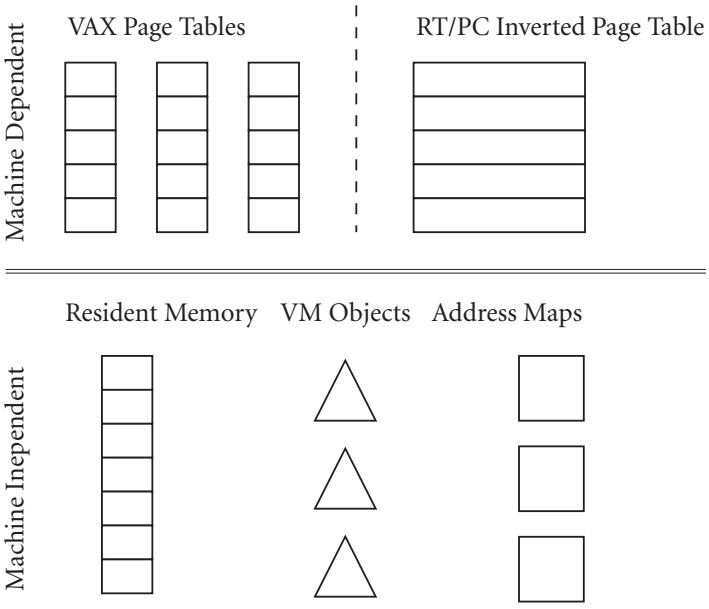

Figure 3: Task address maps

share maps: special address maps that describe regions of memory that are shared between tasks. A sharing map provides a level of indirection from address maps, allowing operations that affect shared memory to affect all maps without back pointers.

VM objects: units of backing storage. A VM object specifies resident pages as well as where to find non-resident pages. VM objects are pointed at by address maps. *Shadow* objects are used to hold pages that have been copied after a copy-on-write fault.

page structures: specify the current attributes for physical pages in the system (e.g., mapped in what object, active/reclaimable/free).

The virtual memory implementation is split between machine *independent* and machine *dependent* sections. The machine independent portion of the implementation has full knowledge of all virtual memory related information. The machine dependent portion, on the other hand, has a simple page validate/invalidate/protect interface, and has no outside knowledge of other machine-independent related data structures.

One advantage of this separation is the fact that the "page size" for different sections of the implementation need not be the same. For example, the machine dependent page size on a VAX is 512 bytes. The machine independent page size is a boot time variable that is a power of two of the machine dependent size. The backing storage page size may vary with the backing store object.

The actual data structures used in a machine dependent implementation depend on the target machine. For example, the VAX implementation maintains VAX page tables, whereas the RT/PC implementation maintains an Inverted Page Table. Since the machine independent section maintains all data structures, it is possible for a machine dependent implementation to garbage collect is mappings (e.g. throw away page tables on a VAX). The machine independent section will then request the machine dependent section to map these pages again when the mappings are once again needed.

In addition to the normal demand paging of tasks, the Mach virtual memory implementation allows portions of the kernel to be paged. In particular, address map entries are pageable in the current implementation.

6 Interprocess Communication

Interprocess communication in 4.3BSD can occur through a variety of mechanisms: pipes, pty's, signals, and sockets [7]. The primary mechanism for network communication, internet domain sockets, has the disadvantage of using global machine specific names (IP based addresses) with no location independence and no protection. Data is passed uninterpreted by the kernel as streams of bytes. The Mach interprocess communication facility is defined in terms of *ports* and *messages* and provides both location independence, security and data type tagging.

The *port* is the basic transport abstraction provided by Mach. A port is a protected kernel object into which messages may be placed by tasks and from which messages may be removed. A port is logically a finite length queue of messages sent by a task. Ports may have any number of senders but only one receiver. Access to a port is granted by receiving a message containing a port capability (to either send or receive).

Ports are used by tasks to represent services or data structures. For example, Flamingo [11], a window manager running under Mach on the MicroVAX II, uses a port to represent a window on a bitmap display. Operations on a window are requested by a client task by sending a message to the port representing that window. The window manager task then receives that message and handles the request. Ports used in this way can be thought of as though they were capabilities to objects in an object oriented system [4]. The act of sending a message (and perhaps receiving a reply) corresponds to a cross-domain procedure call in a capability based system such as Hydra [12] or StarOS [5].

A *message* consists of a fixed length header and a variable size collection of typed data objects. Messages may contain both port capabilities and/or embedded pointers as long as both are properly typed. A single message may transfer up to the entire address space of a task.

Messages may be sent and received either synchronously or asynchronously. Currently, signals can be used to handle incoming messages outside the flow of control of a normal UNIX style process. A task could create or assign separate threads to handle asynchronous events.

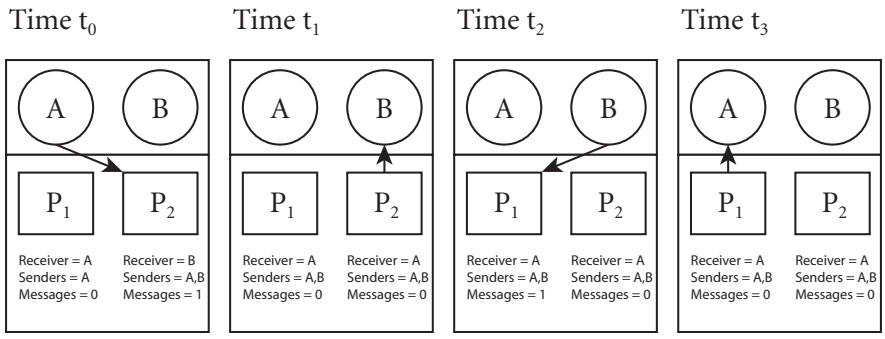

Figure 4: Typical message exchange

Figure 4 shows a typical message interaction. A task A sends a message to a port P_2. Task A has send rights to P_2 and receive rights to a port P_1. At some later time, task B which has receive rights to port P_2 receives that message which may in turn contain send rights to port P_1 (for the purposes of sending a reply message back to task A). Task B then (optionally) replies by sending a message to P_1.

Should port P_2 have been full, task A would have had the option at the point of sending the message to: (1) be suspended until the port was no longer full, (2) have the message send operation return a port full error code, or (3) have the kernel accept the message for future transmission to port P_2 with the proviso that no further message can be sent by that task to port P_2 until the kernel sends a message to A telling it the current message has been posted.

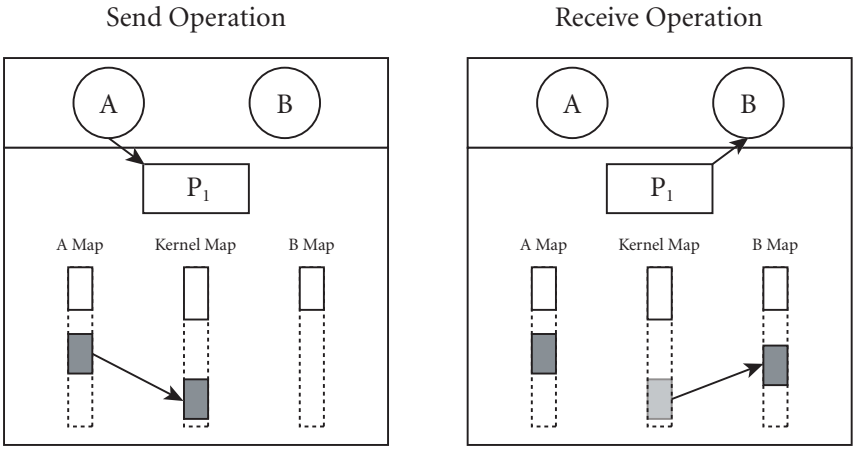

Figure 5: Memory mapping operations during message transfer

Figure 5 shows Task A sending a large (for example, 24 megabyte) message to a port P_1. At the point the message is posted to P_1, the part of A's address space containing the message is marked copy-on-write – meaning any page referenced for writing will be copied and the copy placed instead into A's virtual memory table. The copy-on-write data then resides in a temporary kernel address map until task B receives the message. At that point the data is removed from the temporary address map. The operating system kernel determines where in the address space of B the newly received message data is placed, allowing the kernel to minimize memory mapping overhead. Any attempt by either A or B to change a page of this copy-on-write data results in a copy of that page being made and placed into that task's address space.

6.1 Defining interprocess interfaces

Interprocess interfaces, including the interface to the Mach kernel, are defined using an interface definition language called Matchmaker [6]. Matchmaker compiles these interface definition into remote procedure call stubs for various programming languages including C, CommonLisp and a CMU variant of PASCAL. These stubs use the Mach message system as their basic transport facility. Matchmaker interfaces can perform runtime type-checking and provide

sufficient information in messages for network communication servers to perform routine data-type conversion and data re-alignment between machines of different architectural types.

6.2 Network communication and security

By itself, the Mach kernel does not provide any mechanisms to support interprocess communication over the network. However, the definition of Mach IPC allows for communication to be transparently extended by user-level tasks called *network servers*. A network server effectively acts as a local representative for tasks on remote nodes. Messages designed for ports with remote receivers are actually sent to the local network server.

When a task sends a message to a destination port on another node, the forwarding of the message is transparent to the sender. The sender has no direct means of determining whether the eventual destination port is local to its node or is actually on a remote node. The security guarantees of the Mach port capabilities can be extended into the network environment by network servers through the use of encryption [10].

Network servers collectively implement the abstraction of *network ports*. A network port is the network representation of a port to which tasks on more than one node have access rights. Each network port is known by its *network port identifier*. A network server maintains a mapping between network ports (accessible to tasks on its node) and their corresponding local ports.

In operation, when a network server receives a message from a task trying to send a message to a remote destination port, it maps the local destination port into a destination network port identifier. The network server then derives the address of the destination node from the network port identifier and sends the message over the network to this node. The destination network server, on receiving the network message, maps the network port identifier into a local destination port and forwards the message to its ultimate destination. Each network server holds receive rights to those network ports for which the receive rights to the corresponding local ports are held by local tasks. Send and ownership rights to network ports are handled in the same way except that send rights to a network port may be held by many network servers. Messages are typed collections of data objects and any message may contain port access rights. Network servers must examine the type tags of data sent or received in messages over the network to recognize the transmission of such access rights and take appropriate action. Currently Mach's network servers handle data-type conversion and re-alignment for three different machine architectures: the DEC VAX, IBM RT/PC[5], and PERQ Systems PERQ[6].

[5] RT/PC is a trademark of International Business Machines.
[6] PERQ is a trademark of PERQ Systems Corporation.

7 System Support Facilities

In addition to the basic system support facilities provided by 4.3, Mach provides a kernel debugger and a transparent remote file system.

7.1 Kernel Debugger

Kernel debugging has always been a tedious undertaking. UNIX systems traditionally have no support for kernel debugging, requiring kernel implementors to "debug with printfs" or other ad hoc methods. The Mach kernel has a built-in kernel debugger (kdb) based on adb[7]. All adb commands are implemented including support for breakpoints, single instruction step, stack tracing and symbol table translation.

In order to aid debugging, as well as study the performance of the kernel, the Mach debugger also supports functions not available in adb. For example:

enhanced stack traces: stack traces may contain the values of local variables and registers for each stack frame.

call/return trace support: single stepping may continue without intervention until the next call or return instruction.

instruction counting: the number of instructions executed between regions of code may be counted.

During the implementation of the system these features have proved invaluable in both debugging and performance tuning.

7.2 Transparent Remote Filesystem

The remote filesystem available in Mach was originally available in 1982 as part of CMU's locally maintained version of 4.1 UNIX. At that time, it supported only a small set of the functions required of a file system: it could read and/or write publicly accessible files. Over the years, the remote filesystem has undergone a steady increase in functionality. Currently, all UNIX functions, such as remote current directories and execution of remote files, are supported.

The remote filesystem is completely transparent to the user. Users may effectively login to a remote filesystem connection to receive all of their normal privileges on the remote filesystem, or they may elect to not login, and receive only "anonymous" access to the remote filesystem.

A small set of kernel hooks redirects remote file operations to remote servers transparently. Each machine wishing to allow remote requests runs a user-mode server process. The kernel sends requests corresponding to operations such as read, write, open and close. The client then performs the appropriate operation, and returns with a reply code and/or data. Data is not cached with one exception: remote execution of files causes a cached copy of the entire file

[7]This version currently only works on Vaxen.

to be read into an inode on a local disk. Subsequent executions of this file cause the kernel to check for a modification of the remote file; if no such modification has been made, then the locally cached copy is executed.

Links to remote filesystems are created using a special file type. While mount points have been used for this purpose in other remote filesystems [1, 2], it was felt that the restriction on the number of mount points (and the need to actually mount such a filesystem) made this option inappropriate. Using special links allows a machine to connect to an arbitrary number of other machines without the need for mounting all possible remote filesystems, and the fear of the mount table overflowing.

8 Implementation: a new foundation for UNIX

The Mach kernel currently supplants most of the basic system interface functions of the UNIX 4.3BSD kernel: trap handling, scheduling, multiprocessor synchronization, virtual memory management and interprocess communication. 4.3BSD functions are provided by kernel-state threads which are scheduled by the Mach kernel and share communication queues with it.

The spectacular growth in size of the Berkeley UNIX kernel over the last few years has made it apparent that continued expansion of UNIX functionality threatens to undercut the advantages of simplicity and modifiability which mad UNIX an attractive operating system alternative for research and development. Work is underway to remove non-Mach UNIX functionality from kernel-state and provide these services through user-state tasks. The goal of this effort is to "kernelize" UNIX is a substantially less complex and more easily modifiable basic operating system. This system would be better adapted to new uniprocessor and multiprocessor architectures as well as the demands of a large network environment. The success of this transition will depend heavily on the fact that the basic Mach abstractions allow kernel facilities such as memory object management and interprocess communication to be transparently extended. Figure 6 shows the eventual relationship between the Mach kernel and UNIX.

9 Current status: Mach-1

Mach is still under development and extensive performance comparisons with other systems have not yet been done. Although the system has yet to be tuned, current performance appears to be in line with 4.3BSD. Some early simplistic measures of virtual memory performance are encouraging. The MicroVAX II cost of touching newly allocated memory is less than 0.7 milliseconds per 1024 bytes of data (versus approximately 1.2 milliseconds for 4.3BSD). Operations typically expensive in UNIX, e.g. fork, are substantially faster with the new virtual memory support. Mach is currently in production use by CMU researchers on a number of projects including a multiprocessor speech recognition system called Agora and a project to build parallel production systems.

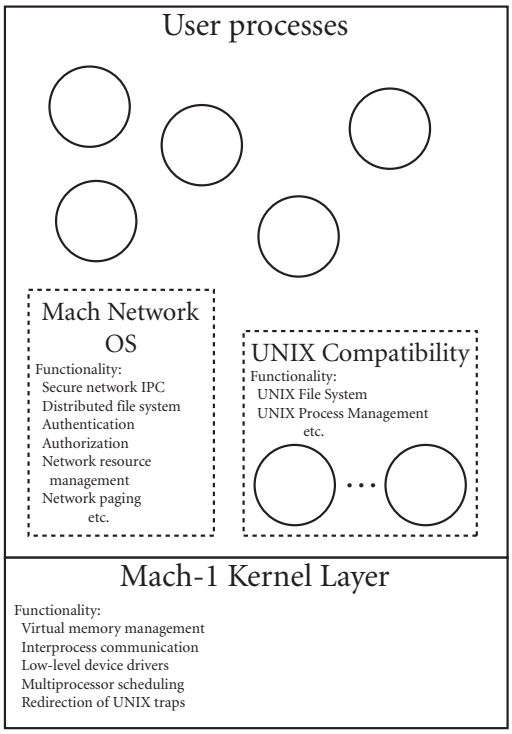

Figure 6: Mach with UNIX functionality in user-state tasks. As of April 1986 the box labeled "UNIX compatibility" still executes in kernel state and communicates with the Mach kernel layer through a shared communication queue.

As of April 1986, Mach runs on most VAX architecture machines: VAX 11/750, 11/780, 11/785, 8600, MicroVAX I, and MicroVAX II. In addition, Mach runs on four (11/780 or 11/785) processor VAX 11/784 with 8 MB of shared memory and the IBM RT/PC. The same binary kernel image runs on all VAX uniprocessors and multiprocessors. The same kernel source is used for both VAX and RT/PC systems. Work has begun on ports to the uniprocessor SUN 3, multiprocessor Encore MultiMax and VAX 8300. Implementation of the Mach thread mechanism is expected by Summer 1986.

References

[1] D. R. Brownbridge, L.F. Marshall, and B. Randell. The newcastle connection, or UNIXes of the world unite! *Software - Practice and Experience*, 20, 1982.

[2] M. Satyanarayanan et al. The ITC distributed file ssystem: Principles and design. pages 35–50. ACM, December 1985.

[3] R. Fitzgerald and R. F. Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems*, 4(2), May 1986.

[4] A. K. Jones. The object model: A conceptual tool for structuring systems. *Operating Systems: An Advanced Course*, pages 7–16, 1978.

[5] A. K. Jones, R. J. Cahnsler, I. E. Durham, K. Schwans, and S. Vegdahl. Staros, a multiprocessor operating system for the support of task forces. pages 117–129. ACM, December 1979.

[6] M. B. Jones, R. F. Rashid, and M. Thompson. Matchmaker: An interprocess specification language. ACM, January 1985.

[7] W. Joy. 4.2BSD system manual. Technical report, Computer Systems Research Group, Computer Science Division, University of California, Berkeley, Berkeley, CA, July 1983.

[8] R. F. Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. pages 64–75. ACM, December 1981.

[9] D. M. Ritchie and K. Thompson. The Unix time sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[10] R. Sansom, D. Julin, and R. Rashid. Extending a capability based system into a network environment. Technical report, Department of Computer Science, Carnegie-Mellon University, April 1986.

[11] E. T. Smith and D. B. Anderson. Flamingo: Object-oriented abstractions for user interface management. pages 72–78, January 1986.

[12] W. A. Wulf, R. Levin, and S. P. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, 1981.