THE MACH KERNEL

Akshay Sharma

Anshu Vashisht

Deepika Yadav

Abstract

Mach is an operating system kernel developed at Carnegie-Mellon University to support operating system research, primarily distributed and parallel computation. The Mach operating system was designed to incorporate the many recent innovations in operating system research to produce a technically advanced and fully functional system. Mach incorporates multiprocessing support throughout which is extremely flexible and ranges from shared memory systems to systems with no memory shared between processors. Mach provides a new foundation for UNIX development that spans networks of uniprocessors and multiprocessors. This paper describes Mach and its design. Also described are some of the details of its implementation and current status.

Keywords: *Multiprocessing, Mach Operating System, Shared Memory, Uniprocessors*

I. INTRODUCTION

Mach is an operating system kernel developed at Carnegie-Mellon University to support operating system research, primarily distributed and parallel computation. The project at CMU ran from 1985 to 1994 ending with Mach 3.0, which was finally a true microkernel. The Mach operating system was designed to incorporate the many recent innovations in operating system research to produce a technically advanced and fully functional system [1]. Mach was developed as a replacement for the kernel in the BSD version of UNIX, so no new operating system would have to be designed around it [2].

Several factors were considered in making the Mach operating System. It was important that the operating system be:

- Multi-user and multitasking
- Network compatible
- An excellent program-development environment
- Well-represented in the university, research and business communities.
- Extensible and robust
- Capable of providing room for growth and future extensions

II. THE MACH MICRO KERNEL ARCHITECTURE

The Mach microkernel has been built as a base upon which UNIX and other operating systems can be emulated. This emulation is done by a software layer that runs outside the kernel, as shown in Fig 1. Each emulator consists of a part that is present in its application programs' address space, as well as one or more servers that run independently from the application programs. It should be noted that multiple emulators can be running simultaneously, so it is possible to run 4.3BSD, System V, and MS-DOS programs on the same machine at the same time.

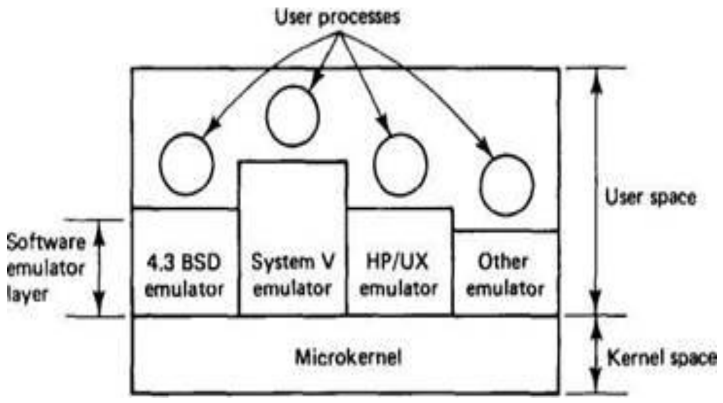

Fig. 1: The abstract model for UNIX emulation using Mach.

Like other microkernels, the Mach kernel, provides process management, memory management, communication, and I/O services. The idea behind the Mach kernel is to provide the necessary mechanisms for making the system work, but leaving the policy to user-level processes. Mach minimises kernel size by moving most kernel services into user-level processes.

A. *Mach kernel features*

The key features of Mach are:

1. Task and thread management: Mach supports the task and thread abstractions for managing execution. Computation within a task is performed by one or more threads; these threads share the address space and all other resources of the task. Threads are scheduled to processors by the Mach kernel, and may run in parallel on a multiprocessor [3].
2. Interprocess communication: Mach provides Interprocess communication among threads via constructs called ports. Ports are protected by a capability mechanism so that only Mach tasks with appropriate send or receive capabilities can access a port. Mach tasks, threads, memory objects, and processors are, all manipulated by sending messages to ports which represent them [4].
3. Memory object management: The address space of a Mach task is represented as a collection of mappings from linear addresses to offsets within Mach memory objects. The primary role of the kernel in virtual memory management is to manage physical memory as a cache of the contents of memory objects. The kernel's representation for the backing storage of a memory object is a Mach port to which messages can be sent requesting or transmitting memory object data [5].

4. System call redirection: The Mach kernel allows a designated set of system calls or traps to be handled by code running in user mode within the calling task. The set of emulated system calls needs to be set up only once; it is inherited by child tasks on fork operations. This feature allows the binary emulation of operating system environments such as UNIX. It also allows for monitoring, debugging.
5. User multiprocessing: A user-level multithreading package, the C Thread library [6], facilitates the use of multiple threads within an address space. It exports mutual exclusion mutex locks and condition variables for synchronization via condition wait and condition signal operations.

III. PROCESS MANAGEMENT

Process management in Mach deals with processes, threads, and scheduling.

A. *Processes*

A process in Mach consists primarily of an address space and a collection of threads that execute in that address space. Processes are passive. Execution is associated with the threads. Processes are used for collecting all the resources related to a group of cooperating threads into convenient containers.

Fig 2 illustrates a Mach process. In addition to an address space and threads, it has some ports and other properties. The process port is used to communicate with the kernel. The bootstrap port is used for initialization when a process starts up. The very first process reads the bootstrap port to learn the names of kernel ports that provide essential services.

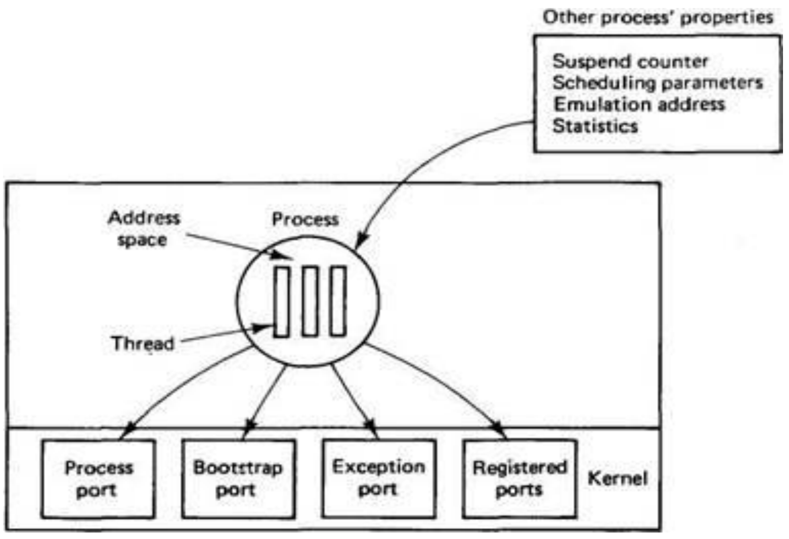

Fig. 2: A Mach process.

The exception port is used to report exceptions caused by the process. The port tells the system where the exception message should be sent. Debuggers also use the exception port. The registered ports are normally used to provide a way for the process to communicate with standard system servers.

A process can be runnable or blocked, independent of the state of its threads. If a process is runnable, those threads that are also runnable can be scheduled and run. If a process is blocked, its threads may not run, no matter what state they are in. The per-process items also include scheduling parameters. These include the ability to specify which processors the process threads can run on. Emulation addresses can be set to tell the kernel where in the process' address space system call handlers are located. The kernel needs to know these addresses to handle UNIX system calls that need to be emulated.

Mach provides a small number of primitives for managing processes. The most important of these calls are shown in Table I.

Table I. Selected process management calls in Mach.

Call	Description
Create	Create a new process, inheriting certain properties
Terminate	Kill a specified process
Suspend	Increment suspend counter
Resume	Decrement suspend counter. If it is 0, unblock the process
Priority	Set the priority for current or future threads
Assign	Tell which processor new threads should run on
Info	Return information about execution time, memory usage, etc.
Threads	Return a list of the process' threads

B. *Threads*

The active entities in Mach are the threads. They execute instructions and manipulate their registers and address spaces. Each thread belongs to exactly one process. All the threads in a process share the address space and all the process-wide resources.

The thread port is analogous to the process port. Each thread has its own thread port, which it uses to invoke thread-specific kernel services, such as exiting when the thread is finished. Mach threads are managed by the kernel, that is, they are sometimes called heavyweight threads rather than lightweight threads (pure user space threads).

Like a process, a thread can be runnable or blocked. A counter per thread can be incremented and decremented. When it is zero, the thread is runnable. When it is positive, the thread must wait until another thread lowers it to zero. This mechanism allows threads to control each other's behaviour.

Mach's approach is the C threads package. This package is intended to make the kernel thread primitives available to users in a simple and convenient form. The thread system calls are listed in Table II.

Table II. The principal C threads calls for direct thread management

Call	Description
Fork	Create a new thread running the same code as the parent thread
Exit	Terminate the calling thread
Join	Suspend the caller until a specified thread exits
Detach	Announce that the thread will never be jointed (waited for)
Yield	Give up the CPU voluntarily
Self	Return the calling thread's identity to it

C. *Implementation of C Threads in Mach*

Various implementations of C threads are available on Mach. The original one did everything in user space inside a single process. This approach timeshared all the C threads over one kernel thread, as shown in Fig. 3(a). This approach can also be used on UNIX or any other system that provides no kernel support. The threads were run as co-routines, which mean that they were scheduled non-pre-emptively. If one thread makes a blocking system call, such as reading from the terminal, the whole process is blocked. To avoid this situation, the programmer must avoid blocking system calls.

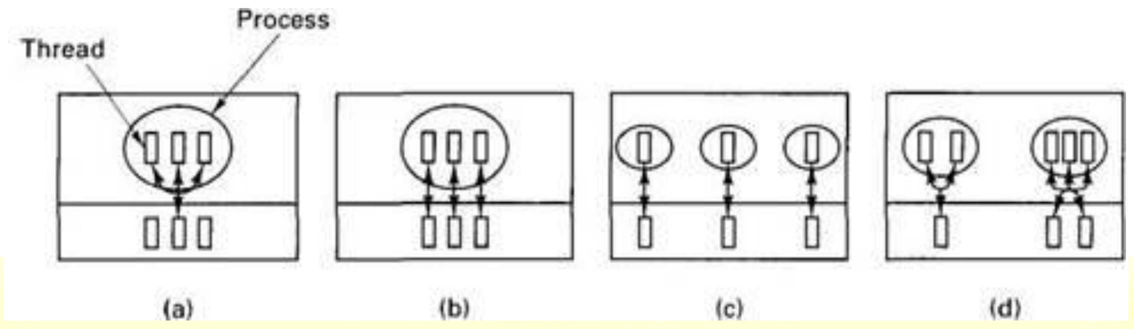

Fig 3. (a) All C threads use one kernel thread. (b) Each C thread has its own kernel thread. (c) Each C thread has its own single-threaded process. (d) Arbitrary mapping of user threads to kernel threads.

A second implementation is to use one Mach thread per C thread, as shown in Fig. 3(b). These threads are scheduled preemptively. Furthermore, on a multiprocessor, they may actually run in parallel, on different CPUs. In fact, it is also possible to multiplex *m* user threads on *n* kernel threads, although the most common case is *m* = *n*.

A third implementation package has one thread per process, as shown in Fig. 3(c). The processes are set up so that their address spaces all map onto the same physical memory, allowing sharing in the same way as in the previous implementations. This implementation is only used when specialized virtual memory usage is required. The method has the drawback that ports, UNIX files, and other per-process resources cannot be shared, limiting its value appreciably.

The fourth package allows an arbitrary number of user threads to be mapped onto an arbitrary number of kernel threads, as shown in Fig. 3(d). It gives the greatest flexibility and is the one normally used at present

IV. COMMUNICATION IN MACH

The goal of communication in Mach is to support a variety of styles of communication in a reliable and flexible way. It can handle asynchronous message passing, RPC, byte streams, and other forms as well.

A. *Ports*

The basis of all communication in Mach is a kernel data structure called a port. When a thread in one process wants to communicate with a thread in another process, the sending thread writes the message to the port and the receiving thread takes it out. Each port is protected to ensure that only authorized processes can send to it and receive from it. Ports support unidirectional communication. A port that can be used to send a request from a client to a server cannot also be used to send the reply back from the server to the client. A second port is needed for the reply.

Ports support reliable, sequenced, message streams. If a thread sends a message to a port, the system guarantees that it will be delivered. Messages sent by a single thread are also guaranteed to be delivered in the order sent. Unlike pipes, ports support message streams, not byte streams. When a port is created, 64 bytes of kernel storage space are allocated and maintained until the port is destroyed, either explicitly, or implicitly under certain conditions. The port contains the fields shown in Fig 4.

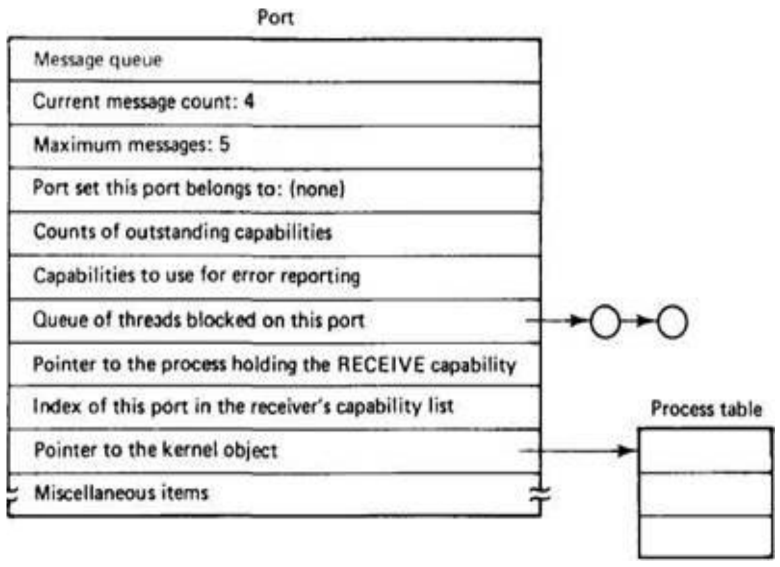
Port
Message queue
Current message count: 4
Maximum messages: 5
Port set this port belongs to: (none)
Counts of outstanding capabilities
Capabilities to use for error reporting
Queue of threads blocked on this port
Pointer to the process holding the RECEIVE capability
Index of this port in the receiver's capability list
Pointer to the kernel object
Miscellaneous items
Process table

Fig 4: A Mach port.

When a thread creates a port, it gets back an integer identifying the port. This integer is used in subsequent calls that send messages to the port or receive messages from it in order to identify which port is to be used. Ports are kept track of per process, not per thread. The kernel, in fact, does not even maintain a record of which thread created which port. Ports may be grouped into port sets for convenience. A port may belong to at most one port set. It is possible to read from a port set (but not write to one). A server, for example, can use this mechanism to read from a large number of ports at the same time. The kernel returns one message from one of the ports in the set. If all the ports are empty, the server is blocked.

B. *Capabilities*

To a first approximation, for each process, the kernel maintains a table of all ports to which the process has access. This table is kept safely inside the kernel, where user processes cannot get at it. Processes refer to ports by their position in this table, that is, entry 1, entry 2, and so on. We will refer table entries as capabilities and will call the table containing the capabilities a capability list. Each process has exactly one capability list. When a thread asks the kernel to create a port for it, the kernel does so and enters a capability for it in the capability list for the process to which the thread belongs. The calling thread and all the other threads in the same process have equal access to the capability. Each capability consists not only of a pointer to a port, but also a rights field telling what access the holder of the capability has to the port. Three rights exist: RECEIVE, SEND, and SEND-ONCE.

The RECEIVE right gives the holder the ability to read messages from the port. At any instant only one process may have the RECEIVE right for a port. A capability with a RECEIVE right may be transferred to another process. A capability with the SEND right allows the holder to send messages to the specified port. The SEND-ONCE right also allows a message to be sent, but only one time. After the send is done, the kernel destroys the capability. This mechanism is used for request-reply protocols. For example, a client wants something from a server, so it creates a port for the reply message. It then sends the server a request message containing a (protected) capability for the reply port with the SEND-ONCE right. After the server sends the reply, the capability is deallocated from its capability list and the name is made available for a new capability in the future.

In Fig. 5 both processes have a capability to send to port *Y*, but in *A* it is capability 3 and in *B* it is capability 4. A capability list is tied to a specific process. When that process exits or is killed, its capability list is removed. Ports for which it holds a capability with the RECEIVE right are no longer usable and are therefore also destroyed, even if they contain undelivered (and now undeliverable) messages.

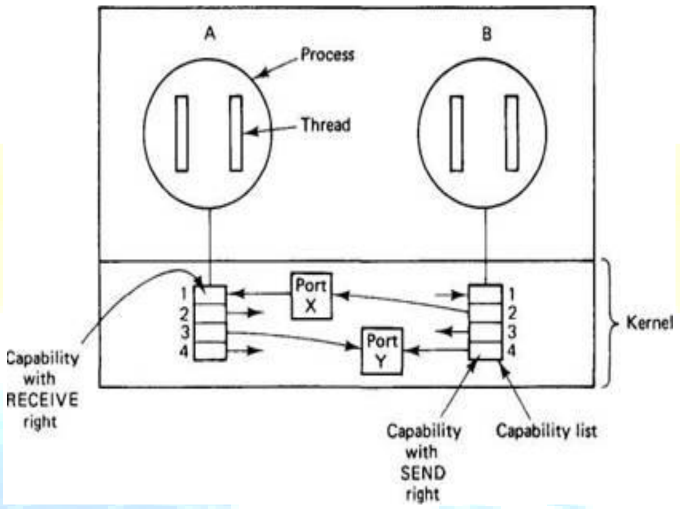

Fig 5: Capability lists

Primitives for Managing Ports: Mach provides about 20 calls for managing ports. All of these are invoked by sending a message to a process port. A sampling of the most important ones is given in Table III.

Table III. Selected port management calls in Mach

Call	Description
Allocate	Create a port and insert its capability in the capability list
Destroy	Destroy a port and remove its capability from the list
Deallocate	Remove a capability from the capability list
Extract-right	Extract the nth capability from another process
Insert-right	Insert a capability in another process' capability list
Move-member	Move a capability into a capability set
Set_qlimit	Set the number of messages a port can hold

C. *Sending and Receiving Messages*

The purpose of having ports is to send messages to them. Mach has a single system call for sending and receiving messages. The call is wrapped in a library procedure called *mach_msg*. It has seven parameters and a large number of options. To give an idea of its complexity, there are 35 different error messages that it can return.

The *mach_msg* call is used for both sending and receiving. It can send a message to a port and then return control to the caller immediately, at which time the caller can modify the message buffer without affecting the data sent. It can also try to receive a message from a port, blocking if the port is empty, or giving up after a certain interval. A typical call to *mach_msg* looks like this:

```
mach_msg(&hdr,options,send_size,rcv_size,rcv_port,timeout,notify_port);
```

The first parameter, *hdr*, is a pointer to the message to be sent or to the place where the incoming message is put, or both. The message begins with a fixed header and is followed directly by the message body. This layout is shown in Fig 6.

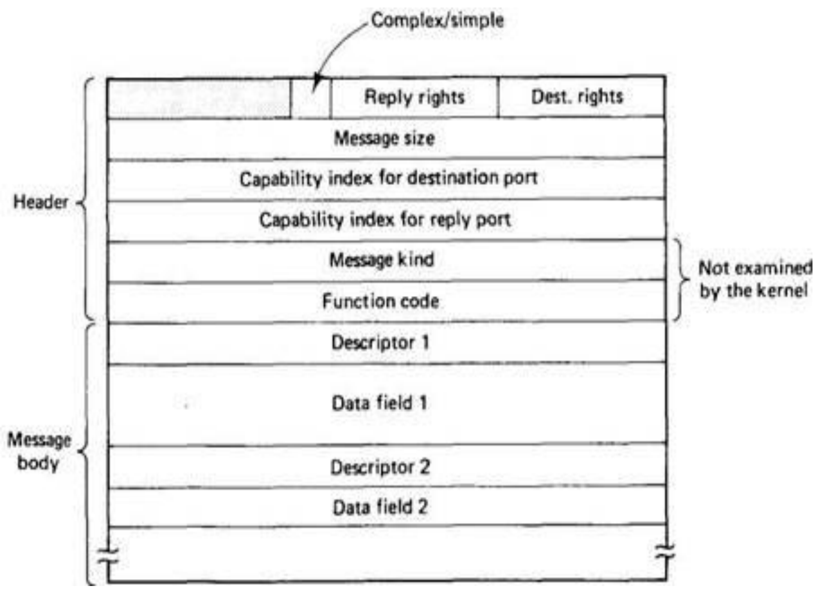

Fig 6: The Mach message format

This information is needed so that the kernel can tell where to send the message. The second parameter, *options,* contains a bit specifying that a message is to be sent, and another one specifying that a message is to be received. If both are on, an RPC is done. Another bit enables a timeout, given by the *timeout* parameter, in milliseconds. Other bits in *options* allow a SEND that cannot complete immediately to return control anyway, with a status report being sent to *notify_port* later. All kinds of errors can occur here if the capability for *notify_port* is unsuitable or changed before the notification can occur. The *send_size* and *rcv_size* parameters tell how large the outgoing message is and how many bytes are available for storing the incoming message, respectively. *Rcv_port* is used for receiving messages. It is the capability name of the port or port set being listened to.

When a message is sent and successfully received, it is copied into the destination's address space. It can happen, however, that the destination port is already full. What happens then depends on the various options and the rights associated with the destination port. One possibility is that the sender is blocked and simply waits until space becomes available in the port. Another is that the sender times out. In some cases, it can exceed the port limit and send anyway.

V. EXCEPTION HANDLING IN MACH

Exceptions are caused by the occurrence of unusual conditions during program execution; raising an exception invokes the operating system to manage recovery from the unusual condition. This paper concerns the design and implementation of an exception handling facility for the Mach [7] operating system. Mach supports debuggers via a combination of independent kernel facilities instead of concentrating debugger support in a single kernel component (e.g. ptrace). This approach avoids duplicating functionality within the kernel by designing debugger support facilities to allow use by other applications. This results in increased flexibility and functionality which benefits all applications using these facilities, including debuggers. In addition the kernel is simplified by the corresponding reduction in special purpose debugger support code. *There are four major classes of applications that use exceptions:*

1. Debugging. Debuggers rely on exceptions generated by hardware trace and breakpoint facilities. Other exceptions that indicate errors must be reported to the debugger; the presence of the debugger indicates the user’s interest in any anomalous program behaviour.

2. Core dumps. In the absence of a debugger, a fatal exception can cause the execution state of a program to be saved (in a file) for later examination. Unix[2] systems refer to these files as 'core dumps' for historical reasons [8].
3. Error handling. Certain applications handle their own exceptions (particularly arithmetic) under some circumstances. For example, a handler could substitute zero for the result of a floating underflow and continue execution. Error handlers are often required by high-level languages [9].
4. Emulation. Virtually all modern machines generate exceptions upon encountering operation codes that cannot be executed by the hardware. Emulators can be built to execute the desired operation in software [10].

A. *A Model for Exception Handling*

The Mach exception handling facility is based on a general model that describes the use of exceptions. This model is derived from the requirements of applications that use exceptions. The generality of the resulting model is sufficient to describe virtually all uses of exceptions.

Applications that use exceptions can be divided into two major classes:

1. Error Handlers: These components perform recovery actions in response to an exception and resume execution of the thread involved. This class includes both error or exception handlers and emulators. Error Handlers typically execute in the same address space as that thread for efficiency reasons (access to state). The term 'handler' is used to refer to any application that uses exceptions.
2. Debuggers: These components examine the state of an entire application to investigate why an exception occurred and/or why the program is misbehaving. This class includes interactive debuggers and servers that produce core dumps; the latter can be viewed as front ends to debuggers that examine core dumps. Debuggers usually execute in address spaces distinct from the application for protection reasons.

Our model is derived by examining the requirements common to error handlers and debuggers. The occurrence of an exception requires suspension of the thread involved and notification of a handler. The handler receives the notification and performs some computation (e.g. error handler fixes the error, debugger decides what to do next), after which the thread is either resumed or terminated. On this basis we propose the following model to cover all uses of exceptions: the occurrence of an exception invokes a four step process involving the thread that caused the exception (Victim) and the entity that handles the exception (Handler, the operating system):

1. Victim: **raise** -- cause notification of an exception's occurrence.
2. Victim: **wait** -- synchronize with completion of exception handling.
3. Handler: **catch** -- receive notification. This notification usually identifies the exception and the victim. Some of this identification may be implicit in where and how the notification is received.
4. Handler: take action. There are two possible actions:

- **clear** -- clear exception causing victim to return from **wait**.
- **terminate** -- cause termination of victim thread.

The boldface primitives in this model constitute the high-level model interface to exceptions and can be viewed as operating on "exception objects". The handler will usually perform other functions between the **catch** and **clear** or **terminate** steps; these functions are particular to the handler application itself and are not part of the exception model.

Any exception handling facility must implement these primitives in some form; as an example we consider signal handling in Unix. An exception that invokes a signal handler uses the following implementation of the model:

1. **raise** - Internal kernel code that translates the hardware exception to a signal and sends the signal to the process. The kill() system call can be used to mimic this for user-detected exceptions.
2. **wait** - Implicit because the handler and victim execute in the same process; execution of the victim cannot resume until the handler completes.
3. **catch** - Internal kernel code that invokes the handler and sets up its stack. Both the exception and the entity that caused it are implicit in the choice of the handler and the process context that it executes in.
4. **clear** - sigcleanup() or sigreturn() system call invoked when handler exits. These calls unwind the stack and clear the kernel state associated with a signal handler.
5. **terminate** - exit() system call.

The model also applies to UNIX support for debuggers. This exception handling model serves as a guide for future exception handling facilities.

VI. UNIX EMULATION IN MACH

Mach has various servers that run on top of it. Probably the most important one is a program that contains a large amount of Berkeley UNIX inside itself. This server is the main UNIX emulator.

The implementation of UNIX emulation on Mach consists of two pieces, the UNIX server and a system call emulation library, as shown in Fig 7.

When the system starts up, the UNIX server instructs the kernel to catch all system call traps and vector them to addresses inside the emulation library of the UNIX process making the system call. From that moment on, any system call made by a UNIX process will result in control passing temporarily to the kernel and immediately thereafter passing to its emulation library. At the moment control is given to the emulation library, all the machine registers have the values they had at the time of the trap. This method of bouncing off the kernel back into user space is sometimes called the trampoline mechanism. Once the emulation library gets control, it examines the registers to determine which system call was invoked. It then makes an RPC to another process, the UNIX server, to do the work. When it is finished, the user program is given control again. This transfer of control need not go through the kernel. When the *init* process forks off children, they automatically inherit both the emulation library and the trampoline mechanism, so they, too, can make UNIX system calls. The EXEC system call has been changed so that it does not replace the emulation library but just the UNIX program part of the address space.

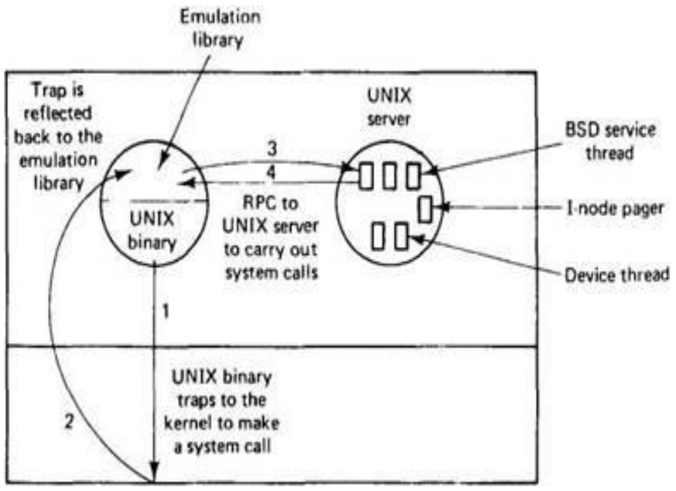
Emulation library
Trap is reflected back to the emulation library
UNIX server
BSD service thread
3
4
RPC to UNIX server to carry out system calls
UNIX binary
I-node pager
Device thread
1
2
UNIX binary traps to the kernel to make a system call

Fig 7: UNIX emulation in Mach uses the trampoline mechanism.

The UNIX server is implemented as a collection of C threads. The emulation library communicates with these threads using the usual Mach interprocess communication.

When a message comes in to the UNIX server, an idle thread accepts it, determines which process it came from, extracts the system call number and parameters from it, carries it out, and finally, sends back the reply. Most messages correspond exactly to one BSD system call.

One set of system calls that work differently are the file I/O calls. They *could* have been implemented like this, but for performance reasons, a different approach was taken. When a file is opened, it is mapped directly into the caller's address space, so the emulation library can get at it directly, without having to do an RPC to the UNIX server. To satisfy a READ system call, for example, the emulation library locates the bytes to be read in the mapped file, locates the user buffer, and copies from the former to the latter as fast as it can.

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 [9].

VII. CONCLUSION

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced operating system. BSD UNIX emulation is done by an emulation library that lives in the address space of each UNIX process. Its job is to catch system calls reflected back to it by the kernel, and pass them on to the UNIX server to have them carried out. A few calls are handled locally, within the process' address space. Other UNIX emulators are also being developed.

As we have shown, Mach is well on its way to achieving its goals. Mach 2.5 includes 4.3 BSD in its kernel, which provides the emulation needed but enlarges the kernel. Mach uses lightweight processes, in the form of multiple threads of execution within one task (or address space), to support multiprocessing and parallel computation. Its extensive use of messages as the only communication method ensures that protection mechanisms are complete and efficient. Finally, by having the virtual memory system use messages to communicate with the daemons managing the backing store, Mach provides great flexibility in the design and implementation of these memory-object-managing tasks.

By providing low-level, or primitive, system calls from which more complex functions can be built, Mach reduces the size of the kernel while permitting operating-system emulation at the user level, much like IBM's virtual machine systems.

IJMRA

REFERENCES

[1].http://www.writework.com/essay/mach-operating-system-mach-history-mach-factors-were-consi#8089

[2].http://en.wikipedia.org/wiki/Mach_(kernel)

[3].Black, D., Scheduling Support for Concurrency and Parallelismin the Mach Operating System, COMPUTER 23, 5, (May 1990), 35-43.

[4].Sansom, R., D. Julin, and R. Rashid, Extending a Capability Based System into a Network Environment, Proceedings of the ACM SIGCOMM 86 Symposium on Communications Architectures and Protocols, (August 1986), 265-274.

[5].Young, M., A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, (November 1987), 63-76.

[6].Cooper, E. and R. Draves, C Threads, Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, 1988.

[7].Accetta, M.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F., Tevanian, A., and Young, M.W. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer Usenix*. July, 1986.

[8].Bach, M. J. *The Design of the Unix Operating System*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

[9].The Mach System – Appendix to Operating System Concepts (8th ed) by Avi Silberschatz, Peter Baer Galvin and Greg Gagne.

[10]. Dobberpuhl, D.W., Supnik, R.M., and Witek, R.T. The MicroVAX 78032 Chip, A 32-Bit Microprocessor. *Digital Technical Journal* (2):12-23, March, 1986.