Mach-US: UNIX On Generic OS Object Servers

J. Mark Stevenson
School of Computer Science
Carnegie Mellon University

Daniel P. Julin
ISIS Distributed Systems

Abstract

This paper examines the Mach-US operating system, its unique architecture, and the lessons demonstrated through its implementation.

Mach-US is an object-oriented multi-server OS which runs on the Mach3.0 kernel. Mach-US has a set of separate servers supplying orthogonal OS services and a library which is loaded into each user process. This library uses the services to generate the semantics of the Mach2.5/4.3BSD application programmers interface (API). This architecture makes Mach-US a flexible research platform and a powerful tool for developing and examining various OS service options.

We will briefly describe Mach-US, the motivations for its design choices, and its demonstrated strengths and weaknesses. We will then discuss the insights that we've acquired in the areas of multi-server architecture, OS remote method invocation, Object Oriented technology for OS implementation, API independent OS services, UNIX API re-implementation, and smart user-space API emulation libraries.

1. Introduction

More and more, operating systems are being developed with a micro-kernel based multi-server structure. Such systems use an operating system kernel to supply system primitives such as tasks, virtual memory, and IPC. Separate servers are used to support derived abstractions (*OS-items*) such as files, TTYs, and pipes. Additional client-side code may be used to do additional OS computation. Examples of such architectures can be seen in systems such as "V" [Cheriton88], Chorus/MIX [Batlivala+92], IBM's "Workplace OS" [Phelan+93], OSF1-AD [Zajcew+93], FSF "GnuHurd", Sun's "Spring" [Khalidi&Nelson93], and Mach-US from Carnegie Mellon University [Julin+91].

The goal of this separation is to achieve several kinds of flexibility in system configuration and development. These include: simplified development and debugging of individual services, replaceable system services, support for multiple API's, efficient use of multi-processor systems, and "loosely coupled" OS configurations. One approach to achieve these goals is to use object-based software with transparent remote method invocation. Mach-US is such a system.

This paper briefly describes Mach-US, the motivations for its design choices and its demonstrated strengths and weaknesses.

While Mach-US is a natural step in the development of micro-kernel based operating systems, it is revolutionary in many ways. Instead of manipulating previous UNIX implementations or even supporting the basic internal structure or semantics of BSD UNIX, Mach-US is a redesign of the entire OS outside the micro-kernel itself. It supplies a fresh look at a series of OS design options without heavily clouding those options with unnecessary constraints of previous systems. OS design features demonstrated by this system are:

- Multiple System Servers On a Micro-Kernel
- Application Programmer Interface Neutral OS Services
- Remote Method Invocation for OS Services
- UNIX API Re-Implementation
- Intelligent API Emulation Libraries

This research is sponsored in part by the Advanced Research Projects Agency under contract number DABT63-93-C-0054. and in part by the Open Software Foundation(OSF).
The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, OSF, or the U.S. government.

Architecture

Revolution, not evolution

Key ideas

- large-granularity servers
- multiple modular servers
- fine-granularity services
- coordination through emul. lib.
- everything out of kernel
- generic services / layering
- client-state processing
- communications through IPC + shared memory

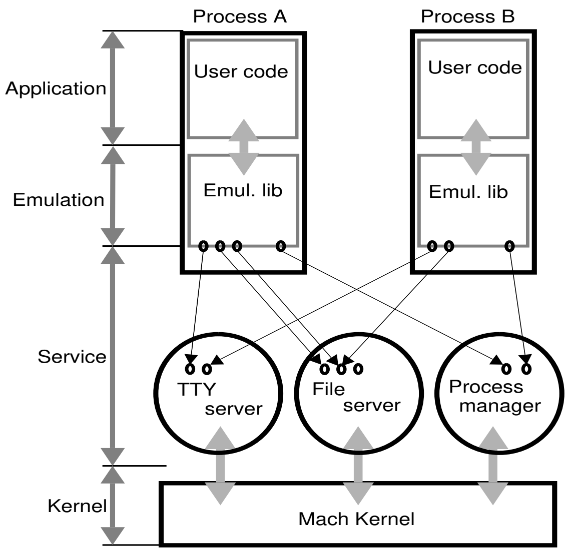

Figure 1: Overall System Architecture

2. Architecture

2.1. Basic Mach-US components

Mach-US is an object-oriented symmetric multi-server OS which runs on the Mach3.0 OS kernel. It is a multi-server system in that it has several servers with orthogonal functions in separate address spaces. These servers supply OS services such as file systems, network support, local sockets and pipes, process management, TTY service, authentication, and configuration.

Mach-US also includes an emulation library which is dynamically loaded into the address space of each user task. This library uses the system servers to generate the semantics of the Mach2.5/4.3BSD application programmers interface (API). It is binary compatible with 4.3BSD. Most servers are essentially independent of the application programmers interface. The few that are UNIX specific were designed to be easily modified or replaced to emulate different API's.

Mach-US is a symmetric OS in that there is no "central" server either to supply an API or for general traffic control. All multi-service actions are controlled by the emulation library.

The overall Mach-US architecture is shown on figure 1.

2.2. System Features

An important aspect of Mach-US is its flexibility. It offers a highly modifiable OS architecture without significant structural impediments to speed. It does this through the following mechanisms:

Object-Oriented Generic OS Interfaces: There are a series of C++ object-oriented interfaces (abstract classes/methods for multiple inheritance) that define semantics supplied by the system servers: access mediation, naming, I/O, network control(OSI-XTI based), and event notification. The various system servers support a combination of these interfaces to do their work and the emulation library uses them to supply a UNIX API.

This uniformity of access makes it easy for new servers to supply additional functionality by sliding into the name-space under the known interfaces. That functionality is then generally available to the system users through their pre-existing software. Additional interface semantics can be achieved through class inheritance and specialization or through the introduction of further orthogonal interfaces.

Modular Services: Different functionality is separated into various servers: configuration, authentication, pathname management, diagnostic, local connections and pipes, UNIX file system, process management, TTYs, and network connections. This separation makes it simple to develop and debug OS services. One can add and subtract services as needed for a given invocation of the system. Furthermore, a bug in one service doesn't crash or corrupt the entire system.

Object Library/Code Reuse: There is an extensive object library for support/implementation of the various generic interfaces and for use as general server building blocks. This enables faster prototyping of a new server and eases creation of servers from foreign code. Some examples of this support are: namespace manipulation, protection, mapped-files/shared-mem/pager-objects, I/O and bulk-data transfer, client-server binding control and remote method invocation, and extensive interrupt/event/signal support.

3. Interface Model

To capitalize on the idea of using generic services, most high-level functions are defined in terms of an object-oriented framework. Each operating system service is represented by one or more abstract *OS-items* exporting a well-defined set of operations. Examples of logical OS-items under UNIX are files, pipes, sockets, TTYs, processes, etc. Each such abstraction is represented in Mach-US by a more API neutral OS-item. Each OS-item corresponds to a generic service that is specialized by the emulation library for the more familiar UNIX API semantics. Each server normally implements a large number of similar, but independent, OS-items. Note that the various servers and items are themselves implemented using object-oriented techniques; the word *OS-item* or *item* is used to avoid confusion with the actual objects used at the implementation level.

The operations exported by each OS-item are categorized into several independent functional groups which are used throughout the system. Each such group of functions is represented by a specific "standard" interface. Each defines a communication paradigm used for conveying a specific flavor of information under specific semantic and synchronization constraints between the emulation library and the OS-item implementations. The general interface paradigms currently defined are:

Access Mediation: access to all OS-items is mediated through a standard facility using general-purpose access control lists and a uniform representation for user credentials. The requirements of a particular API are handled with special initialization and interpretation of the standard abstractions.

Naming: all OS-items in the system are named and accessible through a uniform name space. There is a common name resolution protocol to navigate the global name space, locate servers and locate individual items inside a server (e.g., files in a file server, pipes in a pipe server, etc.). It supports user-centered as well as OS-centered naming through the use of prefix tables [Welch86].

I/O: the I/O interface provides a model for the identification and transfer of data. It supports both byte-level and record-level data access as well as sequential and random access operation. Most issues of synchronization are left to the clients (emulation libraries) for maximum flexibility: when necessary, asynchronous style operations are implemented by the client with additional (waiting) threads. Various caching and buffer management policies can be used or created within the same I/O framework.

Network Control: the network control operations deal with the creation and management of transport endpoints, to the exclusion of actual I/O on those endpoints. This interface is largely inspired by XTI[XTI90], with some changes to facilitate sharing of endpoints between multiple clients, and for integration in the uniform name space.

Asynchronous Notifications: this subsystem can be used by servers to deliver notifications, such as UNIX signals, to clients.

Each of these basic interfaces does not directly define the complete functionality exported by any individual OS-item; rather, they correspond to lower-level services that must be combined to define the complete item. For example, network endpoints export operations from both the I/O and network control interface categories.

3.1. Interface Design Guidelines

Interface Objectives: There are many different objectives to be examined when generating new OS interfaces. While these objectives may overlap or partially conflict, we chose to formalize them for Mach-US to help in its development as well as to clarify the system's goals. The previously described interfaces are defined to achieve these goals:

- Separate interface semantics from previous OS API semantics with the intent of supporting multiple different OS APIs, possibly at the same time.
- Make user specific data reside in the emulation library to be passed to servers.
- Ensure that inter-server coordination can be supplied by the emulation library. This avoids forcing unnecessary inter-server communication.
- Enable as much calculation in the emulation library (versus servers) as possible. This lowers the amount of process/server communication and enables more load distribution for multi-processor systems.

- Enable transparent caching of information in user-side proxies when useful and safe, again to reduce the amount of process/server communication.
- Supply simple and implementable interfaces that can be used directly by sophisticated users. It is important to make it possible for system server implementors to support these interfaces and to make the power of these interfaces available directly to serious developers for special uses and extra speed.
- Make it fast. This is a partial goal of all the preceding objectives.

Absolute Requirements/Constraints: The above objectives are tempered by these absolute restrictions:

- No data/ports/etc. in the emulation library can be usable to crash a service or other client. If a process directly modifies its emulation library, it may corrupt itself, but it may corrupt no other processes.
- No secure data in the emulation library. A process must not be able to change system-maintained information, that it has no right to change, by modifying its emulation library.
- Must be sufficient to support UNIX API requirements.

4. Client-server model

Mach-US has a remote method invocation package composed of three parts: a run-time system that transports invocations and OS-item references between client and server, a set of client-side objects (*proxies*) that interface between the clients and the run-time system, and a set of server-side objects (*agents*) that interface between the run-time system and the code implementing each OS-item in a server. More details about this package can be found in [Stevenson&Julin94] and [Guedes&Julin91].

4.1. Remote Method Invocation System Properties

The Remote Method Invocation (RMI) system supplies the following features to make OS development possible and simpler:

Communication Flexibility: Through the use of client side proxies [Shapiro86], Mach-US supports various modes of communication within the common invocation facility, so as to use the best approach in each case. There is a default message-passing RMI mechanism and alternate methods such as buffers of shared memory between specific clients and servers. Proxies also support specialized client-side caching and client-side processing to maximize the amount of control available to the implementors of OS-items.

Transparency: Code for implementing a feature/method of an OS-item may reside either in the client's address space, a server's address space, or both, and may change on a per item basis. The caller does not need to know the details. All actions against OS-items are standard C++ method invocations.

Forking: The facility provides an efficient and secure mechanism for object reference (binding) inheritance and reconstruction during process creation.

Interruptibility: OS semantics stipulate that some ongoing operations can be asynchronously aborted. The RMI system provides complete support to interrupt pending invocations. Extra care is taken so that methods either complete or are interrupted atomically. A further package is supplied so that servers and the emulation library may control where, when, and how they are interrupted.

Access mediation and client identification: Each invocation of a specific operation upon an OS-item is subject to access mediation. It is not possible for users to perform unauthorized or forged invocations. Also, the identity of the client performing a given operation is available to the OS-item manager. Although these functions could be provided in an ad-hoc fashion, integrating them with the central invocation facility offers efficiency, simplicity, and uniformity.

Garbage collection: The invocation facility uses Mach3.0 IPC ports [Draves90] with a "no-more-senders" notification to inform a server when there are no more clients of a given OS-item.

4.2. Proxies, Agents, and Agencies

Client view — ***proxies*****:** Each OS-item is represented on the client side by a special object that functions as a *proxy*[Shapiro86]. A Mach-US proxy is a body of code that is the client-side part of a given OS-item. All operations on the item are invoked by the client as local operations on the proxy instead of being sent directly to the item's server. In simple cases, the proxy forwards all client invocations to the server via Mach IPC, but in other cases, the proxy may perform most or all of the processing locally, thereby reducing the communications overhead and the load on the server. One typical application is to use a proxy to cache information on the client side of a client-server interface.

A new proxy object is automatically instantiated in a client's address space by the run-time system when an "out arg" object is returned by a remote invocation. The nature and implementation of any given proxy is specified by the designer of each individual server and not by the clients.

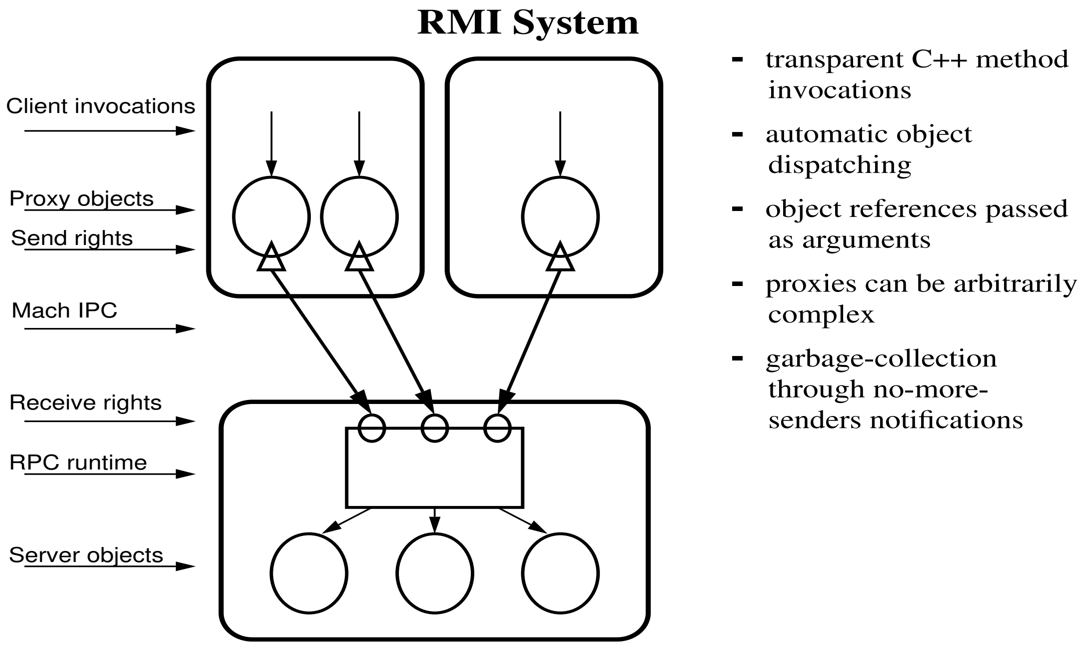

Figure 2: Remote Method Invocation System

The set of methods exported by a proxy need not be the same as the set of method invocations or messages exchanged between a proxy and its corresponding OS-item manager(server). Only the exported set of methods is visible to clients of the system and constitute the actual OS-item's interface.

Server View — *agents* **and** *agencies***:** An *agent* object is used as a filter between the remote invocation system and the actual server object implementing a given OS-item. Such a server object is called an *agency*. Remote invocations on each OS-item are dispatched to the user's agent object of the item instead of being dispatched directly to the agency. The agent verifies that a given invocation is allowed under the current access mediation policies and forwards the invocation to the agency for that OS-item. An agent can be queried for further access information as needed.

The structure of the remote invocation system is shown in figure 2.

5. Emulation Model

As mentioned earlier, one of the base goals of the system was to be able to run application binaries without re-compiling or re-linking. This is done by having syscall traps re-directed by the Mach3.0 kernel to the task's emulation library. The emulation library is not a library in the standard sense. It is an executable image that is loaded into the address space of the initial process at runtime and inherited by its children. Yet, it is not called directly by user processes but instead is invoked by the syscall redirection. While it would be possible for an application to make direct invocations into an emulation library, this has not been done because there has been no real demand for this functionality to date. However, there are special applications that are linked with the system interface libraries used by the emulation library. These applications make calls directly upon the proxy interface.

UNIX API syscalls are implemented in the emulation library itself. Each syscall may make zero, one, or several method invocations against OS-item proxies, and each proxy may or may not need to make a call to a system server to do the requested action. Note that the syscall implementations have the advantage that they actually reside in the address space with the caller, and can access argument memory directly.

The basic responsibilities of the emulation library are:

Syscall: Implements all UNIX API syscalls using OS-item proxies associated with the various system servers.

User State Management: Maintains umasks, sigmasks, seekkeys, and other user-specific parameters.

API Value Translations: Translates values such as *errno* and *uid* from Mach-US general forms into UNIX-specific forms.

API Semantics Objects: Maintains the API-specific object such as UNIX IO objects, file descriptor table, etc.

Name Resolution: Follows path names across servers using prefix caching. This caching causes most name lookups to only communicate with the server that supports the OS-item for which one is searching.

Forking: Creates the new child process and reinitializes its state. The process manager is advised of the intent to fork, and of the occurrence of the fork, but is basically an observer.

The gist of forking is that the emulation library creates a new task that shares the state of the current process and then sends it on its way. The problems come with the cloning of that state. In a multi-threaded system, various parts of the emulation library may not be appropriate to just be just copied at any time. These problems are solved through the "cloning" methods supplied by each object in the emulation library.

6. Server Implementation

Any given server supplies a set of OS-items that export some generic OS interfaces to provide their individual functionality. Specialized interfaces may also be exported by an OS-item that has some unique semantics (e.g., process manager). Every server supports a portion of the name space. Its OS-items are found in a tree of directories supplied by the server under the "/servers" directory. A server is responsible for the proxy and agency implementations for each OS-item and generally uses the common agent class to handle OS-item protection.

There is an extensive set of implementation classes used to support the various generic interfaces. Some simple servers (e.g., pipenet and pathname servers) just glue these common implementations together to support their OS-items, whereas other servers (e.g. ufs and net servers) supply their own base item implementations and use the general implementation classes to handle the required client-server communication paradigms. Call-through proxies are automatically generated for each OS-item and some servers inherit these into their more specialized proxies, if any.

The OS servers are written to a UNIX-style environment. While they can never make syscalls, they use a specialized library that provides functionality like malloc, strcpy, and printf. The rest of their work is done by self-contained computation and direct calls to the Mach3.0 kernel. Each server is multi-threaded with individual threads to service individual user requests and to control possible device activity.

6.1. Server Descriptions

Below are descriptions of the specific system servers supplied at this time. There is no reason why other OS-items cannot be supplied by other system servers in the future, and all of the servers supplied need not be used.

- The **PathName** server manages the root of the name space hierarchy under which all other servers and naming subtrees are mounted. It supplies directory, link, and mount-point OS-items via the naming interface. In particular, it supplies the "/servers" directory.
- The **UFS** server implements a UNIX file system. It exports the naming interface for directories, mount points, links, etc., and also supports the I/O interface for file objects. Each UFS server currently supports one UNIX disk partition. Several independent UFS servers may be run to support multiple partitions. The emulation library prefix table is used to create the standard UNIX "/" directory tree. Internally, the UFS server contains a C++ vnode style interface [Kleiman86]. It uses the Mach3.0 external pager [Young89] mechanism to map files into a complex proxy to support the read/write semantics for each file.
- The **PipeNet** server supplies all local IPC endpoints and communication. It is used to implement pipes and UNIX-domain sockets. It supports the naming, block I/O, byte I/O, and network interfaces.
- The **Net** server supplies network endpoints and communication. It uses a protocol engine from the xKernel project at the University of Arizona [Hutchinson&Peterson91] and supports the same set of interfaces as the PipeNet server.
- The **TTY** server supplies TTY and PTY OS-items. It supports the naming and I/O interfaces as well as a special interface for bsd_tty_ioctls. Its base implementation is largely derived from 4.3BSD UNIX.
- The **Process Manager** manages task and task-group OS-items. It supports the naming interface and a special task interface. It is responsible for assigning ids, monitoring task lifetime, and forwarding user interrupts. It does not create a client's actual tasks; that responsibility is held by the emulation library.

The following servers do not fit the standard interface model and supply system support functionality. They supply no OS-items to the system namespace.

- The **Configuration** server is used for system startup and general environment support.
- The **Authentication** server supplies user authentication for OS-item access based on unforgeability

Timing Results			
Test	Mach-UX	Mach-US	Slowdown
Compile	7.3sec	8.9sec	22%
Null FTP Get/Put	260Kbytes/sec	240Kbytes/sec	7.7%
Parallel Compile (1)	39	49sec	26%
Parallel Compile (10)	146sec	139sec	-4.8%

Figure 3: Time tests, Mach-UX vs Mach-US

of ports in Mach3.0. This work is derived from [Sansom88].

- The **Diagnostic** server acts as a system output console. Server and emulation library printf, log, and error messages are sent to this server for display on the console.

7. Status and Performance

The Mach-US system is operational and available for distribution to all licensed parties. Near complete distributions are available via anonymous FTP. Mach-US, although not a complete UNIX emulation, is sufficient to support many of the day-to-day tasks performed by typical UNIX users. Most of the common UNIX utilities we know and love (csh, pwd, cc, gnu-emacs, find, ftp(d), telnet(d), inetd, ...) are commonly used. There is full TTY and job control, pipes and signals, file access and access control. One can log into this UNIX system and do useful work today. The entire system has been compiled on itself, using sources stored on a Mach-US file server. Yet, first runs of large applications on Mach-US sometimes uncover problems, generally with API conformance.

The following measurements of Mach-US and its UNIX API are supplied to demonstrate that this system, with its unique and flexible architecture, has acceptable end-to-end performance relative to other UNIXs. They do not offer a detailed timing analysis of the Mach-US system.

The timings given compare the Mach-US UNIX emulation with the same benchmarks for the Mach3.0 single server (Mach-UX) system [Golub+90].

- The **compile-test** is a Mach classic that compiles nine small programs. One run of this test creates 48 emulated processes which make a total of 9290 system calls. Because of caching and client-side processing in intelligent proxies and the emulation library, there are only 2430 outgoing remote method invocations. Table 3 shows a comparison of compile tests run on an Intel Xpress i486/50Mhz with a 50Mhz memory bus and 24Meg of memory. The Mach3.0 kernel version used was MK83. The Mach-UX times are for version UX41 and the Mach-US times are for version US50. Our current belief is that the 22% slowdown is caused mostly by new, under optimized Mach-US coding details rather than by its architectural structure. See section 8.1 for further explanation.

- The **parallel-compile-test** is a related test that runs several compile-tests at the same time. Two tests shown in table 3 were made using one compile test at a time and ten compile tests at a time. These tests were run on Sequent Symmetry hardware using 18 of 20 i386/16Mhz processors and 32Meg of memory. This hardware was used to demonstrate relative strengths of the Mach-US distributed computation on a multi-processor system. Because of recent hardware problems, the timings reported are for slightly older revisions of the system. The Mach3.0 Kernel version was MK78, the Mach-UX version was UX38 and the Mach-US version was US48.

  It is significant to note that there is a 31% relative speedup for Mach-US over Mach-UX between the single test and the ten parallel tests. While some of this difference may be caused by arbitrary constraints within Mach-UX, the bulk of the difference can be attributed to a better separation of services in Mach-US than in Mach-UX and a greater client autonomy thanks to the intelligent emulation library and proxy implementations.

- **FTP** is a very high level test of the Mach-US network service. The FTP client is run on the same system as the compile test. The FTP server (FTPD) is run on a i486/25Mhz with 16Meg of memory using the same kernel and UX as the client system being timed. The FTP server system does not change during the tests. "Get" and "put" operations were done for a 750Kbyte file to /dev/null. /dev/null is used to eliminate disk writes from the tests. The first test was thrown out to ensure that the input file was in the disk cache of each OS. Even though Mach-US uses an xKernel protocol engine, the xKernel uses the same BSD code base used by Mach-UX. The throughput differences demonstrated are fairly small.

The compile-test and other straight-line timings have shown that Mach-US generally runs 10-25% slower than Mach-UX for high level benchmarks, but shows significant potential for improvement on multi-processor platforms.

The UNIX API portion of the system is largely un-tuned. This is because the chronically limited Mach-US resources have been concentrated on improving its functionality and robustness. Despite this, some macro-benchmarks show similar or better than those obtained with Mach-UX. There are a number of other known areas where significant performance gains can be achieved through straightforward engineering efforts of bottleneck analysis and RMI information piggy-backing. Additional sophisticated proxies could be implemented to further increase system speed.

The performance gap is real, yet Mach-US achieves the level of performance it does in a more complex, and much richer, environment than other systems available today, and many of the speed differences can be minimized or eliminated using known techniques. Hence, the test results suggest that performance similar to the commonly used Mach-UX single server can be achieved by this multi-server system. We submit that the architecture of Mach-US is not a serious impediment to system usage for speed reasons.

8. Analysis of OS design features

8.1. Multiple System Servers On A Micro-Kernel

The general idea of using multiple separate servers worked very well. The separate servers make debugging new functionality far simpler than dealing with a more monolithic running system. There is little need for concern about corrupting or deadlocking the system under test. One can simply use a standard user-level debugger, such as GDB, on the part of the system in play, leaving the rest of the system running peacefully.

One important research result is that **there is almost no need for inter-server communication**. The one and only inter-server call in the system is needed to implement keyboard interrupts. This means that a multi-server system need not be crippled by such communication, and does not need to run significantly slower than a single server system. While it could be argued that the server coordination role of the emulation library prevented some inter-server calls, this is not a valid argument. It is true that there are a few functions, such as name resolution, that could be implemented by servers calling servers (for links and mount points). But that approach would yield no fewer inter-address space actions. Mach-US does name resolution by using partial path results for further resolution by the emulation library. There appears to be no real advantage to nested inter-server calls versus iterative emulation library calls, and iteration gives the emulation library the opportunity of doing additional user space computation and caching.

8.1.1. Mach3.0 Suitability

Running on the Mach3.0 micro kernel itself worked acceptably well. It supplied a rich enough environment that the various system servers were able to do their work and it supported the syscall redirection needed to do the user-space implementation of syscalls. We often run the multi-server beside the Mach-UX single server for even greater ease of debugging.

Mach-US probably exercises the features of the Mach3.0 micro-kernel more than any other system. It became a valuable test case to ensure the correct functioning of the kernel. There were some very winning features of the kernel that helped immensely in Mach-US implementation:

Syscall Redirection is used to cause syscall traps into the kernel to invoke the appropriate code in the emulation library. Without such redirection, re-compilation of all user applications would be required, thus greatly limiting the usefulness of the system.

Threads are used to shepherd user requests through the system servers, syscalls through the emulation library, and wait for various asynchronous events. Since we required more "fairness" than co-routine threads could offer, system supported and scheduled threads were very useful to such a server model. They are essential for optimal use of multi-processor shared-memory systems such as the Sequent system mentioned earlier.

Ports supply two basic features: unforgeable identifiers and IPC endpoints. By combining the concepts, Mach IPC makes it possible for Mach-US to use a port as an OS-item identifier. A separate identifier is used for each user referencing an OS-item. Clients do RMIs to the port itself and the server has a secure ID of the caller. Hence, this basic port model proved to be a powerful weapon in developing an integrated system wide RMI facility.

External Pagers are servers which the Mach3.0 kernel uses to acquire and update pages of data have been logically mapped into a clients address space [Young89]. Hence the external pager mechanism enables sharing of data between the server and a client, where that data is requested by a simple memory reference.

Both the UNIX file server and the process manager act as external pagers. The file proxies page-fault in file pages as needed. The process manager shares memory with each emulation library to avoid un-necessary server notification of transient process information.

The external pager mechanism supports these two services more efficiently than standard RMI would have.

No-More-Senders port notification can be used to notify a server when there are no remaining external references to a given port. Mach-US uses it for garbage collection of OS-items. It is especially useful since it works automatically for all OS-item references a client holds when the client exits or crashes. This occurres in the same fashion as if the client had explicitly closed the reference.

Scheduling Hand-Off is a Mach kernel mechanism that causes a receiver of a message to continue execution immediately using the remainder of a clients scheduling time slice without waiting to just be rescheduled at some later date. This feature was designed to let small IPCs execute quickly without impacting system fairness, and Mach-US runs significantly faster when such hand-off is utilized. This form of hand-off was a useful step toward the concept of threads that migrate with messages between processes (a feature not in CMU Mach3.0).

There were however some problems when using the Mach3.0 kernel:

Inter Process Communication: The IPC system was not really designed to support a remote method invocation (RMI) paradigm that included interruption. Mach3.0 IPC itself is fairly heavy weight and complex. This comes partly from a "second system effect" and partly from the use of a communication model that attempts to handle every combination of independent message/port transmissions possible, supporting loosely coupled remote procedure calls (RPC) and still making some "fast-path" messages run as fast as possible. Achieving a correct RMI implementation with "fast-path" communication for normal invocations was not simple using the IPC system and there is some reason to believe that a RMI-based IPC could be as effective and easier to use.

The need to interrupt remote method invocations caused headaches. The mechanism for "aborting" long running Mach3.0 requests (e.g., paging and messaging) is crude at best, and deadlocks in some cases. Furthermore, there is no way to identify an individual invocation without taking all invocations off of the fast path or adding inappropriate additional complexity to normal communication. Departure from accepted coding practice was necessary solve these problems. The solutions involved very rare interrupt/restart of uninterrupted invocations, creative use of the "mach_msg" error states, and cautious use of the "abort" mechanism.

Forking: There is no support for forking of kernel entities beyond memory inheritance. There is a real need to support a coherent inheritance model for kernel ports. Furthermore, even memory inheritance can quickly become hard to manage in a complex address space that contains an emulation library, multiple stacks from various threads both in the user application and in that emulation library, out-of-line memory segments created by the memory system, arbitrary buffers, and shared memory areas allocated by various proxy objects or other emulation library components. Long "shadow chains" supporting "copy on write" inherited memory objects became a problem requiring careful memory use.

Even though there were some problems when using the Mach3.0 micro-kernel, it supplied several features that were either useful of essential for the implementation of Mach-US. It was a good and appropriate environment for implementing a multi-server operating system.

8.2. Application Programmer Interface Neutral OS Services

The redesign of the interfaces to OS-items enabled computation to be shifted to the user's address space. This lowered the level of inter-process communication and also speeds computation on multi-processor system configurations.

While we supported the 4.3BSD API on what is believed to be neutral interfaces, that API neutrality is not truly proven. Because of time restrictions, there exists only one emulation library today. However, the same interface calls were often used to support quite disparate syscalls and all API-specific user data is maintained in the emulation library. Hence, there is some evidence that interface neutrality was at least partly achieved.

There are areas where this neutrality was not practical. We did not try to implement a general version of TTY controls. While Mach-US TTYs support the generic ByteIO interfaces, a neutral ioctl system seemed improbable and not worth pursuing. The process manager is also partly API-specific. By the nature of event/signal delivery and its associations with process lifetime, the process manager was implemented to support the specific semantics of 4.3BSD signal/wait. However, its general design and structure are not API-specific, and could be easily modified for another API.

Supporting the generalized interfaces caused additional work, but the redesign effort showed some real fruit. Additionally, to meet the initial design goal of supporting different APIs on a common set of servers, this generality was essential. Such well-defined interfaces proved useful when we added new features to the system while creating servers from imported software. Only limited adaptations were needed to fit said interfaces using the support libraries provided. Standard UNIX applications could immediately and transparently use the new features provided.

8.3. Remote Method Invocation For OS Services

The general concept of OS-items appears to be common to many OS definitions. Therefore, it is natural and useful to represent them as programming language objects and to support remote method invocation for their access.

The Mach-US libraries and runtime supply a uniform policy and implementation for transparent remote method invocation, combined client and server side OS-item implementation, interruption, access mediation, forking, and immediate garbage-collection of unreferenced items. Such a powerful unified system for cleanly handling these specific issues for all OS-items was invaluable to the development of a multi-server OS. Using the RMI system was not burdensome to the implementors of OS-items, and it eased their work while supplying the needed flexibility and base implementation to achieve good system speed. The RMI system supplied should be usable and useful to other systems that need to tackle these issues.

8.4. Object Oriented technology for OS implementation

Mach-US uses GNU C++ as its primary implementation language with some plain C where appropriate and some imported C code. Since there has been some controversy on the topics of object oriented (OO) software development and OS implementation, we feel it is appropriate to offer some personal perspective. A large system needs to be carefully architected, independent of whether it is being implemented using OO technology or not. OO technology alone does not solve design problems, yet OO methodology was a very useful tool for interface clarification, data encapsulation, and code re-use through inheritance. Early Mach-US implementation was done in simple C, and we switched to OO technology for those features. Yet it is just a tool, and just like any tool, there are times when it is the right answer, times when it can be made to work, and times where its misuse can be injurious. All of these were experienced during the implementation of Mach-US.

We did not find any real disadvantages to the use of OO methodology for OS development while designing and implementing Mach-US. Its advantages helped in OS development as it would in any other large system, and OO methodology was a good clear choice for representing and implementing OS-items.

8.4.1. Problems with C++

GNU C++ created many problems for us, but was the only appropriate and available alternative. We chose C++ because of its C compatibility, acceptance, availability. We used the "abstract class" feature of C++ to define the generic service interfaces of the system. This made it possible to define clean interfaces for client use, yet leave the implementations to be filled in later by the various implementation classes that lived in their own inheritance trees. Multiple inheritance was used to combine those abstract interface classes and fill in their methods from the implementation classes.

This constituted a logically clean and understandable mechanism that had the flexibility and transparency properties we needed. It relied on the C++ strict type checking to insure interface conformance at every implementation and call site. It was also a bit of a nightmare. These features of C++ are not the most commonly used features. It was sometimes difficult to determine the precise semantics of the combination of multiple inheritance, abstract classes, the "virtual" class property, and object initialization/destruction. More costly problems were that these and other complex features of the language often did not work well, or were only partially functional. The nature of these bugs changed with each revision. We resolved these problems with a combination of syntactic twiddling, work arounds, localized redesign, and replacing parts of the C++ runtime system.

8.5. UNIX API Re-Implementation

A large part of the problem with UNIX API re-implementation is defining what that API is. There are many different documents which describe what syscalls are expected to do, but what is really desired is to have an ever-changing set of applications run on the new system. Each application was written based on what worked for the system upon which it was developed, not the specifications, manual pages, or common knowledge about the system. For example, "Inetd" knows fields of the various socket arguments were not used, and those arguments can't be checked or used by the new system. Our methodology to determine conformance was to run two releases of the common UNIX tools (4.3BSD Tahoe, MtXinu 2.62MSD [MtXinu90]), some formal 4.3BSD compliance tests [Perennial87], and GNU-EMACS. We have built the system from sources upon itself. The largest amount of time spent on any one aspect of the system was spent on API conformance.

While it is difficult, some level of API re-implementation is necessary. OS researchers have a responsibility to explore different OS architectures. In order to be practical, we must write software that supports the semantics of an existing OS, ensuring that we have applications to run. Hence, there is no avoiding partial re-implementation.

We developed our UNIX API emulation library from scratch to fit the initial design objectives of eliminating licensed BSD code and exploring alternate OS-item interfaces that support API neutrality. Conformance was time consuming, yet few of the problems encountered were caused by the Mach-US architecture. The problems encountered would be the same for any system trying to achieve UNIX conformance with a new code implementation.

Our experience implies that an API neutral multi-server symmetric object oriented OS is just as good, or better than other architectures for doing such a UNIX API re-implementation, but such a re-implementation is painful.

8.6. Intelligent API Emulation Libraries:

Except for the difficulty of re-implementing a pre-existing ill-defined API, intelligent emulation libraries are quite a powerful tool. Mach-US uses its emulation library to maintain the user-specific API information and implement active client side proxies. For real world test cases, that library makes approximately one-fourth as many server calls as a system that makes a server call for every syscall. This difference is important since newer multiprocessor architectures make the relative costs of some inter-processor communications increase. A significant amount of the responsibility for system computation and coordination is shifted to the user process further distributing system computation across the system for such machines.

There are drawbacks:

- It is not always easy to determine if a given bug is in code in the emulation library or in a server. This problem can be overcome with some additional effort and is analogous to the difficulty of determining which "class" in an inheritance tree of any object system may contain a specific bug.
- Forking the emulation library state to a child process is tricky and required a fair amount of overhead and scaffolding to synchronize and reinitialize.
- The emulation library is a part of Mach-US with a complex address space that contains a multitude of threads and stacks, some passed in from the application at syscall time and some that are specific to emulation library execution. This is compounded by the possibility that the application and the emulation library are using different threads packages. Furthermore, it uses all of the interfaces to the other services.

We found that an intelligent of client side library for system code execution is a complex yet powerful and workable solution to ensure needed flexibility to research the interface between the client process and the system services.

9. Conclusion

We have implemented and demonstrated a system that emulates the UNIX API on top of a micro-kernel with full binary compatibility. Its unique architecture provides many features to enhance the flexibility of system development. This research has provided insights in the following areas:

- Multiple System Servers On a Micro-Kernel is a viable way to strongly partition OS functionality. Such an OS architecture and does not force inter-server calls. Multi-server systems supply flexibility and debugging strengths not available in traditional, more unified OS architectures.
- Application Programmer Interface Neutral OS Services are useful as part of a highly valuable redesign of the interfaces to the OS system services. Yet the current system does not fully prove the API neutrality of its OS service interfaces.
- Remote Method Invocation for OS Services is a simple and consistent point to supply OS developers with the features of: call site transparency, authentication, interruption. It supplies Mach-US with the proxy paradigm for server controlled client-side caching, computation, and communication flexibility. RMI and its associated features has been an essential component for multi-server OS development.
- Object Oriented Technology for OS Implementation is highly useful in the same ways that object-oriented technology is useful for any large software system. Language objects are natural tools to represent OS-items. However, objective technology is only a component of OS implementation and is not a substitute for careful system design.
- UNIX API Re-Implementation is painful and total conformance is difficult to achieve for any system. Yet some API re-implementation is required in order to research new OS architectures and still support existing software.
- Intelligent API Emulation Libraries shift work from the OS to the users address space and drastically decrease the need for interaction between the user process and the OS itself.

Mach-US has demonstrated a unique effective system design with many features that enhance system development flexibility and make it a strong platform for further OS research.

10. Further Information

Additional publications and availability information for Mach-US is accessible via the world wide web URL: *http://www.cs.cmu.edu:8001/afs/cs/project/mach/public/www/projects/mach_us.html* or via anonymous FTP from host *mach.cs.cmu.edu* in the *doc/mach_us* and *src/mach_us* subdirectories.

11. Acknowledgments

Many people at CMU and at the Research Institute of OSF have participated in various stages of the design or implementation of the system. Beside the authors, they are: Robert Baron, Jonathan Chew, Paulo Guedes, Michael Jones, G. Robert Malan, Manish Modh, Paul Neves, Douglas Orr, Richard Rashid, Paul Roy, Richard Sanzi, and Mary Thompson. Further help and support for this paper was supplied by Patricia Jones and Wendy Elcesser.

References

[Batlivala+92] N. Batlivala, et.al. Experience with SVR4 Over CHORUS *Usenix Micro-kernels and Other Kernel Architectures Workshop Proceedings*, April 1992.

[Cheriton88] Davic R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314-333, March. 1988.

[Draves90] Richard P. Draves. A Revised IPC Interface *Usenix Mach Symposium Proceedings*, Oct. 1990.

[Golub+90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. UNIX as an application program. *USENIX Summer 1990 Conference Proceedings*, June 1990.

[Guedes&Julin91] Paulo Guedes and Daniel Julin. Object-Oriented Interfaces in the Mach 3.0 Multi-Server System. *Proceedings of the IEEE Second International Workshop on Object Orientation in Operating Systems*, October 1991.

[Hutchinson&Peterson91] N.C. Hutchinson and L.L. Peterson. The x-Kernel: An architecture for implementing network protocols. In *IEEE Transactions on Software Engineering* 17(1):64-76,Jan.1991.

[Julin+91] Daniel P. Julin, Jonathan C. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. Generalized Emulation Services for Mach3.0: Overview, Experiences and Current Status *Usenix Mach Symposium Proceedings*, November 1991.

[Khalidi&Nelson93] Y.A. Khalidi and M.N. Nelson. An implementation of Unix on an Object-oriented Operating System. *USENIX Winter 1993 Conference Proceedings*, January 1993.

[Kleiman86] Kleiman, S.R. Vnodes: An Architecture for Multiple File System Types in Sun Unix. *USENIX Summer 1986 Conference Proceedings*, 1986.

[MtXinu90] MtXinu, Inc. *UNIX User's Reverence Manual, 2.6 MSD Version*, January 1990.

[Perennial87] PERENNIAL Inc. PERENNIAL UNIX Validation Suite Manual, April 1987.

[Phelan+93] James M. Phelan, James W. Arendt, and Gary R. Ormsby. An OS/2 Personality on Mach. *Usenix Mach Symposium Proceedings*, April 1993.

[Sansom88] Robert D. Sansom. Building a Secure Distributed Computer System *Thesis CMU-CS-88-141*, May 1988.

[Shapiro86] Marc Shapiro. Structure and encapsulation in distributed computing systems: the Proxy principle. In *The 6th International Conference on Distributed Computing Systems*, Boston USA, May 1986.

[Stevenson&Julin94] J. Mark Stevenson, Daniel P. Julin. Client-Server Interaction in Multi-Server Operating Systems: The Mach-US Approach. *Carnegie Mellon University Technical Report CMU-CS-94-191*, September 1994.

[Welch86] B. Welch and J. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pp. 184–189, IEEE, May 1986.

[Young89] Michael W. Young. Exporting a User Interface to Memory Management from a Communication-Oriented Operation System. *Thesis CMU-CS-94-191*, September 1994.

[XTI90] Open Software Foundation. *OSF/1 Network Programmer's Guide*, 1990.

[Zajcew+93] R. Zajcew, P.Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, Netterwala. An OSF/1 Unix for Massively Parallel Multicomputers. *USENIX Winter 1993 Conference Proceedings*, January 1993.

12. Author Information

J. Mark Stevenson received his BS in Math/CS from Carnegie Mellon University in 1983. He was a member of the Osiris multi-processor distributed operating system project at Intel and did distributed engineering drawing image processing with Formtek. During the last five years, he has been on the Mach project at CMU with a primary emphasis on Mach-US. Email: jms@cs.cmu.edu

Daniel Julin has been a member of the Mach system design and development team at Carnegie Mellon University from 1984 to 1993. During that period, he has been primarily involved with the areas of networking, remote inter-process communication, distributed object-oriented systems, security, and operating system emulation. He is now with Isis Distributed Systems, Inc., focussing on high-performance networking, fault-tolerance and consistency in distributed systems. Email: dpj@isis.stratus.com