Microkernel Operating System Architecture and Mach

David L. Black* David B. Golub Daniel P. Julin Richard F. Rashid
Richard P. Draves Randall W. Dean Alessandro Forin Joseph Barrera
Hideyuki Tokuda Gerald Malan David Bohman †

DRAFT of June 16, 1991

Abstract

Modular architectures based on a microkernel are suitable bases for the design and implementation of operating systems. Prototype systems employing microkernel architectures are achieving the levels of functionality and performance expected and required of commercial products. Researchers at Carnegie Mellon University, the Open Software Foundation, and other sites are investigating implementations of a number of operating systems (e.g., Unix[1], MS-DOS[2]) that use the Mach microkernel. This paper describes the Mach microkernel, its use to support implementations of other operating systems, and the status of these efforts.

1 Introduction

Microkernel architectures offer a new approach for operating system implementation. These architectures separate the portions of the operating system that control basic hardware resources (often called the operating system 'kernel') from the portions that determine the unique characteristics of an operating system environment, for example, a particular file system interface. In contrast, the traditional monolithic approach to operating system implementation spreads knowledge about the basic system structure throughout a single large kernel. By modularizing the implementation, a microkernel architecture offers improved support for constructing new system services, and configuring systems for specialized environments. These architectures can also simplify porting a system

*Dr. Black's address is: Research Institute, Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142, the address for the other authors is: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

†This research was supported by the Defense Advanced Research Projects Agency (DOD) and monitored by the Space and Naval Warfare Systems Command under contract N00039-87-C-0251, ARPA Order No. 5993. 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 DARPA or the U.S. government.

[1]Unix is a trademark of Unix Systems Laboratories, Inc.

[2]MS-DOS is a registered trademark of Microsoft Corporation.

to a new hardware platform because almost all of the machine-dependent code is isolated in the microkernel. Finally, the use of common underlying services provides support for the coexistence and interoperability of multiple operating system environments on a single host. Experience with implementations based on the Mach system demonstrates that these advantages can be achieved without sacrificing performance.

The Mach kernel [1] is an example of this modular layered approach to operating system design. Mach is a multiprocessor kernel that incorporates in one system a number of key facilities that allow the efficient implementation of those functions necessary to support binary compatibility with existing operating system environments. These mechanisms are intended not simply as extensions to normal operating system facilities but as a foundation upon which Unix and other operating system facilities can be built. We have built and continue to enhance implementations of Unix that achieve levels of performance and functionality competitive with commercial products.

This paper describes research implementations of operating systems based on the Mach microkernel. The results of this work range from mature prototypes that achieve high levels of functionality and performance to more experimental efforts that explore different areas of the design space. The next section describes the overall architecture of operating system implementations based on Mach. Sections 3 and 4 describe the basic Mach microkernel and some of the important concepts and components common to the various implementations. Sections 5, 6, and 7 describe the prototypes that we have constructed using three different implementation approaches, including their status, advantages, and disadvantages.

2 System Architecture

Mach implementations of other operating system environments are based on the observation that modern operating systems such as Unix and OS/2 are primarily distinguished in the programming environment they provide rather than the way they manage or manipulate hardware resources. It is both possible and rational to think of such systems not as operating system kernels but as application programs – servers or sets of servers that can provide client programs with specific programming abstractions.

These traditional operating system environments can be treated as a set of services implemented in terms of a more primitive set of system abstractions. This layered approach allows the most complex software layers to be built above a relatively simple system software kernel which can directly manage hardware resources in ways that can meet the needs of specialized environments. In particular, this approach implies:

Tailorability – Versions of traditional operating systems such as 4.3BSD, OS/2, and System V.4 can be treated simply as different applications which can be purchased separately and potentially run simultaneously with other OS environments. The Mach kernel also provides a base for the implementation of special purpose or customized operating system environments for applications such as network servers or databases.

Portability – Nearly all of the code which constitutes the operating system environment is independent of a machine's instruction set, architecture, and configuration.

Network accessibility – A complex operating system environment need not reside on the same machine as its clients. This allows specialized systems and embedded applications to take advantage of general operating system features without having to incorporate those features in their own system.

Extensibility – New operating system environments and versions can potentially be implemented or tested alongside existing versions.

Real-time – Traditional barriers to real-time support in Unix can be removed both because the kernel itself does not have to hold interrupt locks for long periods of time to accommodate Unix system services and because the Unix services themselves are preemptable.

Multiprocessor support – The underlying kernel does not need to support complex system functions (e.g. file systems or network protocols) which may be defined in ways which limit parallelism. Thus its functions can be more completely parallelized and its features tailored to parallel applications.

Multicomputer support – Efficient operating system support for multicomputers (multiprocessors whose processors do not share physical memory) is greatly simplified. Since the kernel only provides a small number of basic abstractions, it can optimize the mapping of each abstraction onto the distributed hardware. This mapping can be transparent to the implementation of operating system environments, because it does not change the functionality of the kernel abstractions.

Security – Many traditional operating system environments have defined features which are inherently insecure. A more minimal kernel can be defined and implemented in a secure fashion such that trusted computing environments can be implemented in terms of its secure abstractions. The modular architecture of such a layered system is better suited to trusted systems than the structure of traditional kernels.

Other such implementations have, however, frequently started from a rather different notion of the relationship between the system kernel and the supported OS environment: virtual machines (IBM's CP/67 [13]), layering the kernel on a simple message engine (AT&T's MERT[11]), using a global shared communication area (Taos[16]), or loading operating system environment specific emulation-assist code into the kernel (Chorus[2]). In contrast, our approach takes advantage of Mach's support for the manipulation of system resources via a small set of machine-independent abstractions and its integration of memory management and communication functions. All functionality pertaining to the implementation of specific operating system services is performed by Mach tasks (including the application itself) which take advantage of the Mach IPC, scheduling, and virtual memory services. No functionality specific to Unix or other emulated operating systems exists within the kernel.

3 The Mach Microkernel

Mach provides an unusually flexible execution environment for both system and user applications. It exposes the management of CPU, communication, virtual memory, and secondary storage resources in a way that allows system applications to use those resources efficiently.

3.1 Mach Kernel Features

The key features of Mach are:

- Task and thread management
- Interprocess communication
- Memory object management
- System call redirection
- Device support
- User multiprocessing support
- Multicomputer support

A more detailed description of Mach and its abstractions can be found in [1].

Task and thread management Mach supports the task and thread abstractions for managing execution. A task is a passive resource abstraction, consisting of an address space and communication access to system and server facilities. 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. Multiple scheduling classes can be defined for threads. At the present time two classes are provided: fixed priority and timesharing. Timesharing threads are scheduled by the Mach kernel using a multi-level feedback queue scheduler with 32 priority levels. The schedulability of tasks, their threads, and even processors can be controlled by user-state programs. In particular, it is possible for a privileged user-state program to directly control the mapping of threads to processors and thus fully determine system scheduling policy. This feature has been used at CMU to implement multiprocessor scheduling policies such as gang scheduling and virtual processor speedup[3].

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. All services, resources, and facilities within the Mach kernel, as well as those exported by particular Mach tasks or servers are represented

as ports. Mach tasks, threads, memory objects, and processors are, for example, all manipulated by sending messages to ports which represent them. As such, the Mach port facility can be thought of as an object reference mechanism. In addition, this facility is designed to be transparently extended over a network through the use of external communications servers[14].

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[20]. Memory object backing store can thus be implemented by user-state programs such as file system servers, database applications or AI knowledge stores.

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, and transparent extension of existing operating system functions. Similar facilities are provided for redirecting exceptions; this is used to implement redirection for operating systems whose system call linkages are treated as exceptions by Mach (e.g., DOS).

Device support The Mach kernel provides all low-level device support. Each device is represented as a port to which messages can be sent to transfer data or control the device. Data is transferred through read and write operations; the request and reply messages are exported separately, allowing both synchronous and asynchronous styles of I/O. The external memory object protocol allows a user to map the frame buffer for a graphics device directly into its address space.

User multiprocessing A user-level multithreading package, the C Thread library [4], 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. This library has recently been improved to optimize the use of Mach kernel threads by multiplexing C threads onto kernel threads and implementing user-mode context switches between C threads that do not involve the kernel.

Multicomputer support Mach supports multicomputers (multiprocessors which use an interprocessor network instead of shared physical memory) by transparently mapping the Mach abstractions onto the distributed hardware. For example, when a new task is to be created, Mach locates an appropriate node in the multicomputer upon which to create the new task. There are several advantages gained from kernel management of the distributed hardware. Existing operating system environment implementations can be used without change because the multicomputer kernels

support the standard Mach abstractions. The Mach kernel also serves as a single location for optimizations appropriate for multicomputers; all users of its abstractions can then benefit from these optimizations. For example, when data is copied using Mach operations, Mach can use optimization techniques such as copy-on-reference and copy-on-write to greatly reduce the amount of physical memory which is actually copied. Finally, since Mach abstractions are location transparent, the kernel is free to dynamically balance resource usage without user or operating system intervention or knowledge. For example, the kernel can provide load balancing by migrating tasks from one node to another. Mach's multicomputer support is applicable to conventional multicomputers, such as hypercubes and meshes, as well as *processor pools* formed by using a high performance network (e.g., FDDI) to connect workstations or shared memory multiprocessors.

3.2 Evolution from Mach 2.5

The Mach 3.0 Microkernel has evolved from the Mach 2.5 system that is the basis for commercial systems from NeXT, Encore, OSF, Omron, and others. The Mach 2.5 system contains compatibility code for BSD Unix in the kernel and depends heavily on that code. For example, it is not possible to create a Mach task in a 2.5 system without also creating a Unix process. All of this compatibility code has been removed from the kernel in the Mach 3.0 system; this has resulted in the addition of Mach interfaces in areas that did not exist in 2.5 (e.g., devices). The major change to the Mach code has been a complete rewrite of the IPC implementation to achieve improvements in both memory usage and performance. By optimizing the representation of ports and port rights, the amount of memory used for IPC data structures was reduced by 50% for a system running a Unix emulation. New algorithms and other optimizations to favor the common remote procedure call case in both the IPC and scheduling code have doubled the speed of a null RPC[6, 7]. Mach 3.0 now executes a null RPC on a DecStation 3100 (16.67Mhz R2000 cpu) in 95 microseconds.

The use of continuations is an important contribution to the performance of Mach 3.0. As used by the Mach scheduler, a continuation is the address of the routine to call when the thread continues execution plus a small data structure that contains local state needed by that routine. This local state corresponds to the local variables that would normally be saved (e.g., in a control block structure), and in particular, saving and restoring this local state replaces the saving and restoring of processor registers during a context switch; this represents a significant reduction in the amount of information to be manipulated, especially for modern processor architectures with large numbers of registers (e.g., 32). The use of a continuation also replaces the execution context (the routine that invoked the context switch and its callers) that is normally saved on a kernel stack. Hence, a thread blocked with a continuation does not require a kernel stack; this eliminates the needs to maintain many of these stacks and switch between them during most scheduling operations. Continuations have been applied to the IPC, exception, and page fault handling facilities of Mach 3.0, resulting in both performance improvements and a reduction in kernel memory requirements (by reducing the number of kernel stacks).

3.3 Real-Time Mach

The Mach kernel is similar in structure to real-time message passing kernels. It contains no built-in file system or other higher level facilities which could interfere with interrupt handling or real-time performance. As such, it provides a useful vehicle for experimental work in real-time scheduling and resource management. Due to the portability of the Mach kernel, Real-Time Mach should be able to provide a common real-time computing environment for various machine architectures including single board computers and embedded systems.

The Advanced Real-Time Technology (ART) group at Carnegie Mellon has developed real-time scheduling and resource management enhancements for the Mach kernel as well as a real-time application development toolset. This research effort is based on experiments carried out over a number of years on the ARTS distributed real-time kernel[19] and its real-time toolset, Scheduler 1-2-3[17] and Advanced Real-Time Monitor[18].

The objective of the Real-Time Mach project has been to develop a real-time version of Mach that can support a predictable real-time computing environment and to develop an associated real-time toolset. Real-Time Mach supports the following real-time features: a real-time thread model, an integrated real-time thread scheduler including multiple policies, real-time synchronization mechanisms, and a memory resident memory object manager. Prototype Real-Time Mach kernels with these features have been implemented and are in use at Carnegie Mellon University.

4 Operating Systems as Application Programs

The basic facilities provided by the Mach kernel support the implementation of operating systems as Mach applications. The memory object management mechanisms allow paging functionality to be implemented outside the kernel, and provide system programmers with control over data cached by the virtual memory system. The system call redirection mechanism makes it possible to support applications that have the system call traps linked into their executable binaries without modifying the kernel. Hence, the structure of a user-mode operating system emulation consists of an emulation library for each application plus one or more servers. Operating systems that have been implemented in this fashion include 4.3BSD Unix, MS-DOS, and the Macintosh[3] operating system. Efforts to implement other operating systems (e.g., OSF/1) using this technology are underway.

4.1 Emulation Libraries

Emulation libraries are a key architectural component that support the implementation of non-native operating system environments. An emulation library functions both as a translator for system service requests and as a cache for their results. System service requests from an application in the non-native environment are translated to requests for the Mach microkernel or other servers

[3]Macintosh is a trademark of Apple Computer, Inc.

that are used to emulate the target environment. Results returned by these requests may be cached for future use. Emulation libraries are transparently loaded into otherwise unused portions of application address spaces; Mach's memory inheritance mechanism is used to implement inheritance of these libraries for child tasks created by such applications. It is not strictly necessary for the emulation library to occupy part of the application's address space, but doing so optimizes data transfer between the library and application.

Mach's system call redirection mechanism is used to invoke emulation libraries from fully linked application binaries. These binaries request operating system services by executing hardware trap instructions (e.g., SVC); Mach's system call redirection facility forwards these trap invocations to the emulation library. This trap redirection is not necessary if the base library containing the system call stubs (e.g., libc) can be dynamically linked to the application at execution time. In this case, a new version of the base library can be substituted that translates invocations of these stubs directly into operations for the Mach kernel or other servers. Current emulation library code is always executed with its own stack instead of using the stack of the application making the system call. This allows emulation libraries to be used with applications which may be doing their own stack management or providing their own lightweight process mechanisms.

Emulation libraries are vulnerable to tampering by application programs because they reside in user applications' address spaces and it is well within the capabilities of a user program to either read/write or remove that region of memory. As a result, care is taken to ensure that the correct functioning of the servers cannot be affected by malicious or unintentional tampering with emulation libraries. It is also important that information managed by the library not be more security sensitive than information otherwise available to the user. In this regard, the library must operate under restrictions similar to that of the standard C runtime library.

System call implementations using emulation libraries and system call redirection within a single address space do not suffer a performance disadvantage with respect to in-kernel system call implementations. These performance comparisons were made between Mach 2.5 and a Mach 3.0 system using an emulation library on a 25MHz HP-Vectra (80386 processor). The basic mechanism costs of calling and returning from a system call are identical for the in-kernel and emulation library cases (35 microseconds). This is to be expected because both mechanisms involve a single kernel entry and exit; when a system call is redirected to an emulation, the return to the caller is performed entirely in user mode without further kernel involvement. When the additional work of handling a simple system call (e.g., getpid) is added, the emulation library approach is slightly faster (64 microseconds) than the in-kernel approach (68 microseconds). This work represents the additional state that must be set up and torn down to create the environment for handling a Unix system call.

4.2 OS Environment Architectures

There are several possible alternatives for the structure of operating system environment implementations using Mach. The simplest approach is to implement all of the functionality in an emulation library. This is best suited to single user systems, as service requests from such systems are often easy to translate into requests for Mach or other servers. Examples of systems that have been

emulated in this fashion include MS-DOS and the Macintosh operating system. Our work with this approach has made extensive use of the native operating system code for the original system.

The alternative approaches use one or more servers in addition to the emulation library. These server-based approaches are distinguished by the granularity of the implementation's decomposition into servers. At one end of the spectrum are large granularity decompositions that implement most of the emulated functionality in one server. The 4.3BSD Unix system has been emulated using this structure, and an emulation of OSF/1 is in progress. At the other end of the spectrum are small granularity decompositions that employ a family of functionally specialized servers in addition to the emulation library. This approach allows the reuse of servers among different operating system environments, and supports the coexistence and sharing of resources among such environments.

5 Native OS Systems

Native OS systems utilize an emulation library and the code of the original operating system to implement the system services used by applications. This approach has been used to emulate the DOS and Macintosh operating systems on top of Mach. In both cases, significant portions of the native operating system are allowed to run directly because many service requests for single user systems do not involve operations that require intervention by the Mach kernel. An example from the DOS implementation is that native DOS filesystem code is allowed to execute, because its invocations of low level BIOS functions (the only code that actually accesses the I/O device) are intercepted. In the Macintosh system, the graphics primitives that write to the display are allowed to execute directly from the Mac ROMs once the display has been appropriately mapped.

The Mach exception mechanism is used to intercept and redirect service requests for both systems because their system call invocation mechanisms differ from Mach and/or Unix on the same hardware. The Mach kernel does not recognize these as system call invocations, and instead treats them as exceptions. In addition to system calls, both systems require interception of certain low level functions internal to the native OS. Intercepting BIOS invocations from DOS is straightforward because DOS uses a system call-like mechanism (software interrupt) to invoke BIOS operations[4]; these invocations generate Mach exceptions that are redirected to the emulation library. In the Macintosh implementation, certain routines in the Macintosh ROMs must be intercepted. This was done by taking advantage of a ROM patch facility. The patch facility supports replacement of ROM operations by RAM equivalents so that changes can be made if bugs should be found in the ROM routines. The Mach emulation uses this feature to replace certain ROM routines with emulation library implementations.

The emulation library plays a dual role in these systems. It is responsible not only for implementing system services (or invoking native OS code to do so), but also for virtualizing access to hardware devices. This is because many applications in these systems expect to access hardware devices (e.g., the display) directly without invoking a system service. As part of its support for multi-user and multi-application environments, the Mach kernel must protect and control access

[4]Not all BIOS invocations are intercepted; those that do not cause protection or device access conflicts are allowed to execute using the native BIOS code.

to devices. Virtual memory techniques are used to make devices accessible to applications. For example, an inaccessible region of memory can be placed in the region of address space where an application expects to access the control registers of a device. Attempts to access these registers cause exceptions, which allow the emulation library to perform the appropriate functions. In many cases (e.g., displays), the Mach device pager can be used to map the device buffer memory directly into applications at the expected location. An additional area of virtualization is the use of threads to emulate asynchronous device interactions. For example, emulation library threads are used to handle both disk and keyboard accesses.

Both the Macintosh and DOS systems expect to execute on a Mach kernel that is also emulating the Unix operating system. The respective emulators are initially loaded from a Unix file system, and Unix file services are transparently available to applications using both emulated systems (e.g., DOS uses the Network Redirector to access Unix files). In addition, the emulated systems can manage disk partitions using their private (non-Unix) on-disk filesystem layouts. The Macintosh system also supports the use of Unix and Mach applications under MultiFinder; a Macintosh application has been written that provides access to a Mach/Unix C shell. A similar application is being implemented to provide shell access to Windows 3.0 under DOS.

These system implementations are functionally complete, and support virtually all Macintosh and DOS applications without change and with similar performance. Among the DOS applications that we have used on top of Mach are business applications such as Lotus 1-2-3[5], WordPerfect[6], and Windows 3.0[7], and games such as Wing Commander[8] and Space Quest IV[9]. The Macintosh system supports Multifinder, business applications (e.g., MacDraw 2.0[10], Excel[11], Powerpoint[12]), games (e.g., Beyond Dark Castle[13]), etc. Application performance is essentially indistinguishable from the native system for both systems; the use of the continuation RPC enhancements described in Section 3.2 has been a major contributor to this achievement. The use of virtual memory in the Mach kernel and these implementations imposes minimum hardware requirements on these systems. For DOS systems, an 80386 or 80486 processor is required, as other compatible Intel processors do not support virtual memory. Most display types (e.g., VGA, EGA) are supported, as are third party sound boards. The corresponding requirement for Macintosh systems is a Mac II, and sound is also supported.

6 Large Granularity Server Systems

Our implementations of large granularity server decomposition employ an emulation library that communicates with a multithreaded server specific to the operating system being emulated. This

[5] Lotus 1-2-3 is a registered trademark of Lotus Development Corporation.
[6] WordPerfect is a trademark of WordPerfect Corporation.
[7] Windows 3.0 is a trademark of Microsoft Corporation.
[8] Wing Commander is a trademark of Origin Systems.
[9] Space Quest IV is a trademark of Sierra On-Line.
[10] MacDraw is a trademark of Apple Computer, Inc.
[11] Excel is a trademark of Microsoft Corporation.
[12] Powerpoint is a trademark of Microsoft Corporation.
[13] Beyond Dark Castle is a trademark of Silicon Beach Software, Inc.

server, contained in a single task, is typically invoked via a Mach message exchange for each system call issued by application processes. In addition to managing system call emulation, the server may also act as an external memory manager for file and other data.

A large granularity operating system emulation of this kind is attractive for several reasons:

- The server is solely responsible for performing the emulation of all OS environment semantics. The structure of the server is, in fact, similar to that of an in-kernel implementation; it has global knowledge of all the information needed for the emulation. Internal context switching between threads can be extremely fast.
- The OS server is completely pageable and can make more efficient use of memory (by sharing data structures and stack space) than can a multiple server implementation.
- It can be relatively straightforward to transform an existing in-kernel OS implementation into such a server, because most of the code can be simply carried over. This can make it easy to preserve both existing code and semantics. This could allow vendors with proprietary OS environments to more quickly take advantage of Mach as a basis for their systems.

The feasibility of this approach has been demonstrated by implementing a Unix server and associated emulation library for 4.3BSD Unix[8]. In practice, this single task Unix server works well and is in regular use. It currently runs on a variety of platforms, including DECStations (2100, 3100, and 5000), i386 PCs from multiple vendors, and Sun 3 machines. This system is functionally interchangeable with existing versions of 4.3 BSD/Mach on those machines. A similar system is under development for OSF/1. The remainder of this section describes the implementation of these systems and how they take advantage of the features that Mach provides.

6.1 Unix Server

The bulk of Unix services are provided by the Unix Server. It is implemented as a Mach task with multiple threads of control managed by the Mach C Threads package. Internal synchronization and process-switching within the Unix Server (e.g., sleep, wakeup, spl) are implemented by using the C Threads package's mutex, condition_wait, and condition_signal functions. A typical system configuration will have dozens of C Threads allocated within the Unix Server. Most threads belong to a common pool which handle incoming requests from user processes. Several threads are dedicated to routines that, in a BSD kernel, would be driven by hardware interrupts (device IO completion, timeout, network code). All communication with hardware devices is done through Mach's IPC facility. Figure 1 shows the organization of the Unix Server and its relationship to the Mach kernel.

The primary tool for communication between the Unix Server and a Unix application program is Mach IPC. Most requests for service arrive in the form of Mach messages requesting that a Unix operation or service be performed. For each incoming message, a c thread is dispatched from the pool to handle that operation. That thread then determines which Unix process requires service, what operation is to be performed and finally parses the message to obtain the arguments for that

Figure 1: Unix Server System Architecture

operation. Many, but not all, messages to the Unix Server correspond directly to system calls normally present in 4.3BSD.

The main departure from this style of interaction between the Unix Server and a Unix application can be found in the handling of 4.3BSD file access. Access to Unix files can be provided either through a pure message passing interface or through the Mach memory object facility. The decision of which interface to use can be made either by the Emulation Library or the Unix Server. The primary reason to choose a pure message passing interface would be performance in a network environment where a message passing interface corresponds more precisely to the natural implementation technology of a network. In a tightly coupled multiprocessor or a uniprocessor a memory object interface is a more efficient way to transfer large amounts of data.

In the case of a memory object implementation of file access, the Unix Server acts as the memory object manager (or 'inode pager') for 4.3BSD files. When a file is opened by a Unix application its data is mapped directly into the portion of the Unix application address space occupied by the Transparent Emulation Library. That library then directly provides read, write, lseek, etc. system call access to the file's data. In order to ensure Unix file sharing semantics, the Transparent Library must hand-shake with the Unix Server through messages whenever conflicts with other applications could arise. See Figure 2. The major drawback of this approach is increased costs for open and close operations on files. These operations can require memory mapping and deallocation, which are reasonably expensive in practice. Our performance results have shown that open/close costs are important, but not nearly as important as the cost of read/write/lseek operations.

Figure 2: Unix Server Input/Output Architecture

6.2 Transparent Emulation Library

The Transparent System Call Emulation Library for this system contains the equivalent of the Unix system call handler and glue routines necessary to transform system calls into remote procedure calls to the Unix Server. All application system calls are intercepted by the library using the Mach system call redirection facility. These calls may be directly handled by the library or transformed into remote procedure calls to the server process. Most of the machine dependencies in the BSD code are handled by the Library. These include manipulating the application's stack for signal handling and forking a new process.

Many common Unix requests are handled exclusively by the Transparent Library. For example, a Unix file which has been mapped into the Transparent Library memory region as part of an open call can be directly read by the Transparent Library without requiring the intervention of the Unix Server. Shared memory techniques are used to allow most signal operations that set or read signal state (e.g., the signal mask) to be directly handled in the library. When the server forks a new process it creates two regions of memory shared between the server and the process, a read-only region for server to library communication and a read/write region for library to server communication[14]. The use of these techniques results in approximately one-half of application system calls being directly handled by the library without communicating to the server.

The Transparent Emulation Library is loaded into the address space of the first user process (/etc/init). The Unix Server uses Mach's memory inheritance facility to cause inheritance of the

[14]Use of this shared memory is optional; while it improves performance on uniprocessors and tightly coupled multiprocessors, it may not be appropriate for other architectures. If the memory is not configured, a message passing interface to the server is used instead.

Transparent Library by each child process from its parent on a fork operation. Server implementations of execve and similar operations which reload an address space with a new application program are careful to preserve the Transparent Library portion of the address space. One advantage of this technique is that it allows multiple Transparent Libraries with perhaps different behavior (such as the support of somewhat different Unix variants) to co-exist with the same Unix Server.

6.3 Performance

This section describes performance results comparing the 4.3BSD implementation with Mach 2.5 and commercial systems. These measurements show that the system is achieving comparable performance to commercial systems even though its performance tuning and improvement are far from complete. We expect to obtain similar or better performance from the server implementation of OSF/1 when it is complete. The measurement results come from several basic system call performance tests and two filesystem oriented tests, a compilation test and a more comprehensive file system test. The same disk with the same binaries and user environment was used for all direct comparisons.

The compilation test consists of a shell command file that runs nine compilations of small C source files. These files contained relatively few header file inclusions. Each compilation is separately timed using the 'time' command; the reported results are the total elapsed time. The resulting test stresses process creation/termination, program load and startup, file open/close and read/write costs for small files. The test performs approximately 2600 Unix system calls, including forking 57 processes, attempting to open 240 files and close 350 file descriptors, unlinking 100 files, and calling execve 160 times. Read, write, and lseek operations account for a large fraction of all system calls. Roughly 750 lseek operations, 450 read, and 230 write operations are performed.

The file system test was originally developed by M. Satyanarayanan for his performance evaluation of the Andrew File System[10]. Specifically, we used a version of the Andrew Benchmark modified by John Ousterhout[12]. This benchmark stresses directory and file creation, file copy, file search (using 'find'), and compilation activity. A complete description of this benchmark can be found in the cited papers.

Tables 1, 2, and 3 show some of the comparison results. The third column represents the relative speedup (or slowdown) of the Mach 3.0 plus server system against the other system. The variation in these numbers reflects the changed system architecture; moving the Unix implementation outside the kernel changes the relative speed of some operations with respect to each other. The important conclusion to draw from these results is that the overall performance of this system is comparable to Mach 2.5 and two commercial versions of Unix (and faster in some cases). Work is continuing on measuring and tuning the performance of this and other areas of the system. Among our preliminary results in the networking area is a test in which ftp bandwidth is essentially unchanged when substituting a Mach 3.0 system for a Mach 2.5 system.

Test	Mach 2.5	Mach 3.0	Mach 3.0 Speedup
create write (100K bytes) delete	634.1ms	596.0ms	*1.06*
lseek + read (8K bytes)	2.12ms	1.36ms	*1.55*
write (1M bytes)	0.26MB/sec	0.26MB/sec	*1.00*
read (cached)	4.40MB/sec	5.12MB/sec	*1.16*
read (uncached)	0.41MB/sec	0.38MB/sec	*0.92*
compilation	28.5sec	27.4sec	*1.04*
filesystem	400sec	405sec	*0.98*

Table 1: Mach 2.5 vs. Mach 3.0 + BSD Server: 8MB Sun 3/60 with Priam disk

Test	Mach 3.0	SunOS 4.1	Mach 3.0 Speedup
getpid	0.102ms	0.090ms	*0.88*
lseek + read (8K bytes)	1.36ms	1.56ms	*1.14*
read (cached)	5.12 MB/sec	5.68 MB/sec	*0.90*
compilation	26.1sec	28.9sec	*1.10*
filesystem	397sec	373sec	*0.94*

Table 2: Mach 3.0 + BSD Server vs. SunOS 4.1: 8MB Sun 3/60 with Wren V disk

Test	Mach 3.0	Ultrix 4.0	Mach 3.0 Speedup
create write (100K bytes) delete	281ms	436.72ms	*1.55*
write (10M bytes)	0.38MB/sec	0.43MB/sec	*0.88*
read (cached)	9.36MB/sec	5.82MB/sec	*1.60*
read (uncached)	1.01MB/sec	1.11MB/sec	*0.90*
compilation	11.4sec	14.1sec	*1.23*
filesystem	99sec	100sec	*1.01*

Table 3: Mach 3.0 + BSD Server vs. Ultrix 4.0: DECStation 5000/200 with Wren V disk

7 Small Granularity Server Systems

Our prototype system using small granularity server decomposition divides the responsibility for operating system support among an emulation library and a collection of servers specialized to particular functions such as naming, authentication, and file data access. The interfaces between the various system components, and those components themselves, are designed to be largely independent of the target environment. This offers benefits including interchangeable components, code reuse, and portability. This section describes research work to design and build a small granularity system employing multiple servers for the emulation of various operating systems, along with a prototype for Unix emulation. This research is still at an early stage.

The distinguishing feature of this system's structure is the use of a common object-oriented framework for server implementation. This framework is reusable, and does not have to be reimplemented for each new emulation system. This reuse extends to components of servers and even entire servers. The independence of this framework from the servers that inhabit it enhances configurability of the resulting systems, and makes it possible to support multiple system implementations concurrently on a single host.

7.1 Architectural Framework

Figure 3 shows the general organization of this system. Each target process is implemented by a Mach *task* that contains the program to be supported and a copy of the *emulation library* to intercept the program's system calls. Most of the target system's functionality is implemented by functionally specialized servers such as a fileserver, network server, and process manager. Their services are exported to the emulation libraries via special libraries or proxies that facilitate and optimize client-server interactions. The final component is the *Mach microkernel* that provides the basic facilities for the execution of those various components. This organization can be viewed as a combination of three independent software layers, namely the kernel, service (servers and proxies), and emulation libraries.

Although programming interfaces differ among operating systems, many of them provide similar sets of services in terms of functionality (e.g., file management, terminal I/O, network access, etc.). Hence the server layer can be structured as a set of components that can be assembled in various configurations. This enhances flexibility, modularity, security, etc. In many cases, the same components can be used for more than one implementation, based on the indirection provided by the emulation library. This decomposition has two aspects:

- Decomposition or *factorization* of the functionality to be provided by the server layer into as many independent *services* as possible. Each of these individual services is then implemented in a separate module, normally in the form of an autonomous *system server*. This decomposition isolates most problems and design decisions into separate, replaceable servers.
- Definition of standard interfaces and protocols for the interactions between the servers and the emulation library, and between the servers themselves. These interfaces do not map directly

Figure 3: General System Structure

into the set of services or servers; instead they correspond to various groups of operations or mechanisms that may be common to several services. Examples of such operations include high level functions such as access control or naming, and low level functions such as locking and synchronization. These standard interfaces facilitate the integration of servers into different system configurations. Many of the interfaces are relatively independent of the target operating systems; the emulation library is responsible for customizing them to specific target system interfaces. These interfaces are used only within the system, and are not exported to applications. As such, they form part of a System Programming Interface (SPI) including the Mach kernel interfaces (as opposed to an Application Programming Interface, or API).

To further simplify the use and combination of the various servers, their interfaces are defined in terms of an object-oriented model. Each operating system service is represented by one or more *items*. Examples from the UNIX domain are files, pipes, sockets, ttys, etc.; however, each of those UNIX abstractions may be represented by a more neutral item, corresponding to the generic services defined above. Each server normally implements a large number of similar, but independent items. Each item is opaque, and exports a well-defined set of operations. These operations can be invoked by the item's *clients*, which are normally the various emulation libraries operating on behalf of each application program. The abstraction of a server is not actually exported to clients; instead, those clients only have access to a number of items, and cannot directly determine which items are managed by which servers. In addition to the interfaces, the various servers and items may themselves be implemented using object-oriented techniques; the word *item* is used to avoid confusion with the actual objects used at the implementation level.

An important aspect of the design is the attempt to optimize client-server interactions by displacing some of the processing required to service various functions, from the system servers into the clients of these services themselves. One major tool for this optimization is the use of *proxy*

objects[15]. A proxy is a body of code loaded in a client's address space, which acts as a local representative to that client for a given item from a given server. All operations on the item are invoked by the client as local operations on the proxy instead of being directed to the server. In the simple cases, the proxy simply forwards all invocations to the server, but in other cases, the proxy may perform most or all of the processing locally, reducing the communication overhead and the load on the server. Proxies are often used to cache information on the client side of a client-server interface. A new proxy object is instantiated in a client's address space as part of the protocol that establishes the client's access to the corresponding item. The set of all the proxies for the items currently accessed by a given client can be viewed as an additional layer of software within the server layer. Although they reside in the client's address space, proxies are logically part of the servers for their items. Proxies are designed and implemented as part of the servers, and not the clients of the servers.

Another tool for the optimization of client-server interactions resides in the specification of the client-server interfaces themselves, and in the separation of responsibilities between the emulation library and the various servers. For example, many of the UNIX per-process attributes, such as the file descriptor table or the signal mask, are entirely managed within the emulation library without the intervention of any server or proxy.

7.2 Server Organization

The framework provides support for the construction and coordination of servers. There are two classes of servers in this system:

Application servers: These servers provide services directly to application programs (e.g., a fileserver).

System servers: These servers provide services primarily to other servers (e.g., basic nameservice, authentication).

The writer of an application server can assume the availability of the basic services provided by the system servers, and may take full advantage of them. In addition, the framework supports server implementation by providing a number of standard low level mechanisms, interfaces, and reusable code fragments. These implement functions such as server invocation, access mediation and asynchronous event notification.

The current system contains the following application servers for the support of Unix and other environments:

- One or more file servers supporting random-access collections of bytes stored in various formats on local disks or remote file servers.
- A terminal server ('TTY') for the management of logical and physical terminal devices.

- A local IPC server supporting basic communication between application programs (pipes, queues, UNIX-domain sockets, etc.).
- A process management server or 'task master' to keep track of application processes and allow them to be operated on by external agents.
- One or more network servers providing access to the network and implementing various protocol families.
- a device server to control user access to the physical devices managed by the micro-kernel.

The current system supports only a Unix environment, but others are anticipated.

The application servers are supported by the following system servers, some of which are still under development:

- One or more *root name servers*, responsible for integrating the various servers together into a single uniform name space from which they can be located by clients.
- An authentication server, acting as a secure repository for information on the identity of users.
- A blackboard server to manage any information that must be efficiently shared between several emulated processes and/or servers.
- A lock/semaphore server to handle synchronization functions between clients and/or servers.
- A configuration/startup server to handle the startup of all the other servers and of the system as a whole, and to keep track of the system configuration (which servers to start, which devices to use, etc.).
- A remote Mach IPC server ('netmsgserver') responsible for forwarding the Mach IPC facility over the network.
- A network shared memory server, providing uniform shared memory over a collection of nodes connected by a network.
- A diagnostics server responsible for logging all debugging, warning and error messages produced during the operation of the system.

This collection of application and system servers is sufficient to implement all of the functionality of Unix.

7.3 Status and Performance

The functionality of the prototype for small granularity 4.3BSD Unix emulation has reached the level of minimal utility. After starting the system it is possible to log in and obtain a shell. The

emacs editor is functional[15], as are the usual program development tools (make, cc, ld, etc.) This makes it possible to edit, compile, and execute programs. Unix sockets and the udp protocol are implemented, allowing files to be retrieved from and sent to remote systems via tftp. The majority of Unix network services await an implementation of the tcp protocol for this system. This functionality is sufficient to self-host the system, but the current performance and robustness were insufficient for self-hosting when this paper was written; improving these are important areas of ongoing work.

A comparison with the large granularity server system provides some evidence that tuning of this system will enable it to achieve competitive levels of performance. System calls in the large granularity system are handled either directly by the emulation library or by the library contacting exactly one server (the Unix server). The same statement is true for this small granularity architecture; many system calls are handled in the library, and those that aren't almost always contact only one server. The performance results in Section 6.3 show that the large granularity system is achieving commercial levels of performance, so this operational similarity suggests that similar levels of performance should also be possible for this small granularity system.

Our current research also involves extending the functionality of this system. An important aspect is investigating the suitability of the overall framework for implementing operating systems other than Unix (e.g., OS/2). In addition to reusing the basic system servers, this work has a goal of reusing some application servers (e.g., fileservers) among different operating system implementations. Another aspect is the development of a trusted or secure system based on this technology. This effort targets the B3 level of security in the Trusted Computer System Evaluation Criteria[5], and is being pursued in cooperation with Trusted Information Systems, Inc.[9].

8 Conclusion

The Mach microkernel supports a variety of approaches to operating system implementation. Native OS systems allow most of the original system's code to execute without change; they are primarily applicable to single user operating systems such as DOS. Large granularity server systems concentrate most of the non-native operating system's functionality in a single server; this simplifies development by making it possible to reuse much of the original system's code. Small granularity server systems spread the implemented functionality across a collection of functionally specialized servers. The framework for constructing such systems can support reuse of components among different emulations, and interoperability of different emulations on a single host. A common feature of all of these systems is the use of an emulation library to intercept and implement system service requests made by applications, providing support for the execution of unchanged application binaries.

Microkernel architectures offer an important and viable implementation alternative to monolithic operating system architectures. Operating system implementations based on Mach microkernel are achieving commercially competitive levels of functionality and performance. The maturity of these prototype implementations varies; some are in daily use (with acceptable performance),

[15] Except for certain advanced process control features.

while others are still under active development. These architectures can support a variety of systems (including real-time and secure systems) on a common kernel base.

9 Acknowledgments

In addition to the authors, the developers of Mach and the operating system implementations discussed in this article include: Avadis Tevanian, Jr., Michael W. Young, Richard Sanzi, William Bolosky, Michael Jones, Jonathan Chew, Mark Stevenson, Douglas Orr, Robert Baron, Gregg Lebovitz, the ART group at Carnegie Mellon University and the research staff of the Open Software Foundation's Research Institute in both Cambridge, Massachusetts, and Grenoble, France.

References

References

[1] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, Mach: A New Kernel Foundation for UNIX Development, *Proceedings of Summer Usenix*, (July 1986).

[2] Armand, F., M. Gien, M. Guillemont, and P. Leonard, Towards a Distributed UNIX System - The CHORUS Approach, *Proceedings of the European UNIX Systems User Group Conference*, (September 1986).

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

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

[5] U.S. Department of Defense, Trusted Computer System Evaluation Criteria, Department of Defense Standard DOD 5200.28-STD, (December 1985).

[6] Draves, R., A Revised IPC Interface, *USENIX Mach Workshop Proceedings*, (October 1990), 101-122.

[7] Draves, R., R. Dean, B. Bershad, and R. Rashid, Continuations: Unifying Thread Management and Communication in Operating Systems, To Appear.

[8] Golub, D., R. Dean, A. Forin, and R. Rashid, UNIX as an Application Program, *Proceedings of Summer USENIX*, (June 1990), 87-96.

[9] Graham, J., and W. Morrison, Trusted Mach Architecture, TIS report 324, Trusted Information Systems, (April 1990).

[10] Howard, J., M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, Scale and Performance in a Distributed File System, *ACM Transactions on Computer Systems* **6**, 1, (February 1988).

[11] Lycklama, H. and D. Bayer, The MERT Operating System, *Bell System Technical Journal*, (July 1978).

[12] Ousterhout, J., Why Aren't Operating Systems Getting Faster as Fast as Hardware?, *Proceedings of Summer Usenix*, (June 1990), 247-256.

[13] Parmelee, R., T. Peterson, C. Tillman and D. Hatfield, Virtual Storage and Virtual Machine Concepts, *IBM Systems Journal* **11**, 2, (1972), 99-130.

[14] 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.

[15] Shapiro, M., Structure and Encapsulation in Distributed Computing Systems: The Proxy Principle, *The 6th International Conference on Distributed Computing Systems*, (May 1986).

[16] Thacker, C., L. Stewart and E. Satterthwaite, Jr., Firefly: A Multiprocessor Workstation, *IEEE Transactions on Computers* **37**, 8, (August 1988), 909-920.

[17] Tokuda, H., M. Kotera and C. Mercer, A Real-Time Monitor for a Distributed Real-Time Operating System, *Proceedings of ACM SIGOPS and SIGPLAN Workshop on Parallel and Distributed Debugging*, (May 1988).

[18] Tokuda, H. and M. Kotera, Scheduler1-2-3: An Interactive Schedulability Analyzer for Real-Time Systems, *Proceedings of Compsac88*, (October 1988).

[19] Tokuda, H. and C. Mercer, ARTS: A Distributed Real-Time Kernel, *ACM Operating Systems Review* **23**, 3, (July 1989), 29-53.

[20] 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.