Linux on the OSF Mach3 microkernel[1]

François Barbou des Places
Nick Stephen
Franklin D. Reynolds
(OSF Research Institute, Grenoble and Cambridge)

Ever since the selection of Mach2.5 as the basis of the OSF/1 operating system, OSF intended to base its OS developments on the Mach3.0 microkernel, which provides a scalable, extensible, OS-neutral set of abstractions. The OSF Research Institute has made significant improvements and extensions to the original CMU Mach3.0 microkernel, and the result, named OSF MK, is still available for free. The latest versions of OSF/1 are based on OSF MK but are encumbered by commercial licenses. We decided to produce an unencumbered UNIX-like server on top of OSF MK, providing our members and the research community with a fully unencumbered development environment for their microkernel developments.

In this paper, we first describe the functionalities that were added to OSF MK by the OSF Research Institute and some performance improvements. We describe the architecture of the Linux server, emphasizing the areas requiring interaction between the Linux code and the microkernel. We present some performance figures for the Intel x86 platform and introduce the port of OSF MK and the Linux server on the Apple PowerMac platform.

OSF MK and the Linux server are or will shortly be available for free from the OSF Research Institute for both the Intel x86 and Apple PowerMac platforms.

1. Introduction

In 1989, OSF chose the Mach2.5 kernel [1] to be the basis of the OSF/1 operating system. Mach is a modern, message passing, operating system microkernel. It is a scalable kernel intended for systems ranging from desktop workstations to multiprocessing supercomputers. As a microkernel, Mach provides a subset of the functionality found in the typical operating system. File systems, network protocol services such as sockets, administration tools such as quota services and security policies are not provided. Instead, Mach strives to provide a core collection of powerful OS neutral abstractions upon which can be built operating system servers. These abstractions, tasks, threads, memory objects, messages and ports, provide mechanisms to manage and manipulate virtual memory (VM), scheduling and inter-process communication (IPC).

In addition to the advanced features already present in the Mach microkernel, such as SMP support, network transparent IPC and support for application specific paging policies, the promise of a microkernel architecture was very appealing. OSF's customers have interests in many different operating systems. We needed a way to pursue a program of operating systems research that was not unduly operating system specific. Operating system specific behavior needed to be separated from OS independent functionality. Our hope was that a microkernel based architecture would prove to be more portable and modular than the monolithic systems that were prevalent at the time.

Currently OSF provides two versions of OSF/1. Each version of OSF/1 is hosted on top of the microkernel. OSF/1 1.3 is a system suitable for workstations and minicomputers. Its performance is very competitive with other versions of Unix [7]. OSF/1 AD, available to OSF RI customers, is a system intended for massively parallel processing supercomputers and clusters.

We are interested in promoting the use of our systems technology within the industrial and research communities. The OSF MK kernel is freely available but the OSF/1 server is encumbered by commercial licenses including the SVR2 Unix license. For many organizations this is not a problem because they

1. The work described in this paper (the Linux server and the port of the microkernel to the PowerMac) was partly supported by Apple Computer, Inc. who also provided useful technical information on the PowerMac.

already have a SVR2 license but for others the licenses have proved to be serious obstacles. In an effort to remove these obstacles we decided to produce a free UNIX-like server that would suit the needs of Mach developers.

We chose Linux for several reasons. It is one of the most popular free implementations of UNIX. Some of our members had expressed an interest in a Linux server. Linux is efficient and has very competitive performance. It provides a very attractive and effective development environment (GNU tools). There are several research projects based on Linux that could potentially benefit from a microkernel. Finally, it was an opportunity to create and experiment with an operating system server that was not derived from BSD. CMU's UX, BSD-Lites and OSF/1 are all descended from BSD.

In 1995 we began a project to port OSF MK to the Apple PowerMac and to create a Linux server that could run on top of OSF MK. The server was hosted on both Intel x86 and PowerMac platforms. Our goal is to produce a free system that has competitive performance, is usable on multiple, inexpensive hardware platforms and is of interest to both our members and the research community. In this paper we will describe the project in detail and some of our future plans. We will also describe some of the important differences between OSF MK and Mach 3.0.

2. OSF MK, the OSF microkernel

Our original interest in Mach was due to the powerful abstractions provided by the kernel, its operating system neutrality and the promise of greater portability and modularity. To a large extent, the microkernel has lived up to our expectations. OSF and some of our customers have ported the kernel to several platforms without undue difficulty. We and our collaborators have hosted different operating system personalities on top of the kernel. And for the last few years we have had an active research program that exploits and extends the abstractions provided by the microkernel.

Portability

The kernel itself is somewhat complex, yet the task of porting it to a new hardware platform is fairly straightforward. We support our version of the microkernel on several different hardware platforms. These include the Intel x86 family, the Intel i860, the DEC Alpha, the HP PA-RISC and the Apple/IBM/Motorola PowerPC. The microkernel has a clean separation of hardware dependent and hardware independent functionality. Writing the hardware dependent code for the microkernel usually requires approximately 4 to 6 months. The difficulty in creating an adequate suite of device drivers can easily exceed the effort necessary to port the rest of the kernel. In some cases the effort can be reduced by converting pre-existing drivers. We will discuss the effort to port the kernel to the PowerMac later in the paper.

Server Performance

In addition to different versions of Unix, MacOS, and MS/DOS have been hosted on Mach. IBM has hosted OS/2 on their own version of the Mach microkernel. Early efforts to layer OS personality servers on top of the microkernel have had disappointing performance due to the extra message-based communication between the system components. Often, as much as a 40% performance cost has been reported.

Our recent experience has led us to believe that most of the problems are due to a lack of attention to performance related issues. After creating the OSF/1 server and noting its disappointing performance, we embarked on an effort to improve its performance.

Thread Migration

Thread migration was derived from work done on Mach4.0 at the University of Utah [3]. It aims at reducing the cost of switching context between the sender and the receiver of an RPC.

The "thread" abstraction has been split into two new entities:

- the "activation", which controls the resources associated with the thread
- the "shuttle", which controls the flow of execution.

During an RPC, the shuttle part of the thread migrates to an empty (with no shuttle) activation in the receiver's task.

Short-Circuited RPC

We can optimize the RPC further by avoiding the construction of the Mach message and the marshalling and un-marshalling of the RPC arguments. This is only possible under certain circumstances. It requires thread migration: the sender's shuttle can migrate into an empty activation in the receiver's task and start executing the remote procedure directly.

In the current implementation, it also requires that the sender and the receiver are collocated (see below), i.e. that they are in the same address space.

Collocation

A collocatable server can be dynamically instantiated as a user task or as a kernel task. When a server runs as a kernel task, it has direct access to the kernel's address space and can use the kernel interfaces with little more code than a simple procedure call.

Modularity is preserved because all interactions between the kernel and the server are still done through MIG (Mach Interface Generator) interfaces at the source level, but the generated code can dynamically detect if the sender and the receiver are in the same address space and avoid unnecessary data copying.

Collocating a server as a kernel task should not be done lightly because an incorrect or malicious collocated server could corrupt the kernel. But once a server has been debugged in user-mode, it can be loaded into the microkernel for best performance.

The collocatable servers of OSF MK are similar in spirit to the "system actors" found in Chorus [4], another microkernel based system developed originally at INRIA.

Combined Effects

Used together, thread migration, short-circuited RPC and collocation can almost reduce an RPC to a simple procedure call. The kernel to server system call exception RPC and the server to kernel system calls benefit from this optimization, greatly improving the overall system performance.

The performance of OSF/1 as a kernel task is very competitive with other Unix systems, including the original OSF/1.

In a subsequent experiment we ported OSF/1 1.3 and the microkernel to an HP PA-RISC workstation. Then we layered an HP-UX (HP's version of Unix) emulation library on top of OSF/1. This allows us to run HP-UX applications unchanged on our system. We then performed a variety of performance tests including AIM III (a common multi-user Unix benchmark), TTCP and others. Most of the tests indicate that OSF/1 using HP-UX emulation libraries has equal or superior performance to HP-UX [11]. This result came as a pleasant surprise and is generally considered a vindication of our performance efforts.

A Real-Time microkernel

In addition to its portability and its operating system neutrality, we were interested in the microkernel as a foundation for research into real-time and distributed computing issues. Mach was not designed as a real-time operating system. The original design focus was for scalable, multiprocessing timesharing systems. Extensive use of lazy evaluation techniques were used in the design of the VM and scheduling subsystems. In order for OSF MK to be a suitable foundation for real-time application we had to make enhancements to the microkernel ranging from the prosaic, such as pre-emption, clocks and alarms to the innovative, such as real-time RPC, the CORDS framework for network protocols and "paths".

Pre-emption

Before the microkernel can be suitable for real-time applications it must provide reasonable, predictable behavior. Complex real-time operating systems, like the various real-time Unix systems and the OSF microkernel all use pre-emption to avoid indeterminate event latencies because there are certain features that, though desirable, are inherently unpredictable. Our pre-emption strategy exploits the fine-grained locks already in the kernel to provide Symmetric Multi-Processing (SMP) support. This naturally led to a fully preemptible system [5].

Mach 3.0 had simple and complex locks [6]. Simple locks provided mutual exclusion and complex locks provided multiple reader, single writer semantics. In OSF MK, simple locks were enhanced to prevent pre-emption. This resulted in a working system but because of the original code's extensive use of simple locks the resulting system had unacceptable event latencies. To deal with this problem we added a new type of lock: a mutex lock. A mutex lock is an inexpensive, mutual exclusion blocking lock. The difference between a simple lock and a mutex lock is that a kernel thread can be preempted while holding a mutex. Most of the algorithms that used simple locks were converted to the new mutex locks.

In the initial version of the system it was possible to have unwarranted context switching between timesharing threads due to pre-emption. This problem was corrected by a simple modification to the pre-emption code. Preemption only occurred if the higher priority thread was a fixed priority thread. With this change, the cost of enabling pre-emption in an SMP environment was negligible when measured by a standard benchmark like AIM III. In a uni-processor environment, the cost of pre-emption is identical to the cost of enabling SMP locks, i.e., approximately 10%. Since our preemption mechanisms are integrated so closely with the SMP locking mechanisms this is not surprising.

Priority Inversion

Kernel pre-emption created a new problem - a type of scheduling anomaly sometimes referred to as a priority inversion [7]. Priority inversions can occur when a high priority thread becomes dependent on or blocked by a lower priority, preempted thread. We designed a straightforward priority boosting protocol inside the kernel to deal with priority inversions. Priority boosting propagates across dependencies, not just locks. If a thread blocks and becomes dependent on another thread, then the thread controlling the dependency is boosted. If the boosted thread is blocked by another dependency then the boosting propagates down the dependency chain. A thread remains boosted until it releases its last dependency.

This algorithm is not perfect in that some threads remain boosted longer than absolutely necessary. But it is very simple and inexpensive.

Real-time RPC

The Real-Time RPC [8] is not layered on top of message based IPC. Implementing RT-RPC as a new kernel service had important advantages:

RPC specific optimizations can be made along the entire RPC path (our RPC is twice as fast as the Mach3.0 RPC optimizations). Real-time RPC specific behaviors, such as alerts, orphan detection, predictable delivery and nested time constraint propagation are possible. An efficient, unified programming model for invoking operations across module boundaries within a task, across the task/kernel boundary or across task boundaries is possible.

The client side of a RT RPC is very similar to a message based RPC. In both cases threads invoke RPCs using ports and the client thread waits for the server to process the request and reply. The server side is somewhat different. Instead of a pool of threads waiting in a message receive loop, a server creates a pool of “empty threads”. These threads have no scheduling state. When a client invokes a server and a server thread is available, the kernel chains the client and server threads together and upcalls the server immediately. Many client thread attributes, such as scheduling attributes, are propagated as well as the normal RPC parameters associated with the server’s operational interface. When the server completes the service requested by the RPC and replies, the server thread becomes empty and a candidate for future upcalls. The reply parameters are propagated back to the client which returns from the invoke and resumes execution.

If no server threads are available then the client thread is blocked. When a server thread eventually becomes available then the scheduling policy selects the appropriate client thread based on its scheduling attributes and it is chained to the server thread. In this way, client threads are serviced in the correct order. This avoids the scheduling anomalies introduced by Mach IPCs port queues and ordered message delivery guarantees and it makes it possible for servers to provide service according to client specified time constraints.

Alerts

Sometimes it is important to signal or generate an exception at the head of an RPC chain rather than a thread somewhere in the middle. One reason for doing this could be the elapsing of a deadline specified by one of the threads in the chain. Alerts are the mechanism used, by either the kernel or an application, to generate a timely exception at the head of an RPC chain.

When an alert is posted to thread A, which is not the head of a chain (suppose a A->B->C->D RPC chain), the kernel propagates the alert towards the head. When the thread that is the current head of the chain is located, in this case thread D, it is suspended and an alert exception upcall is made to the thread’s exception port. This gives the task with thread D a chance to respond in a timely fashion to the event that triggered the alert. By using upcalls instead of messages the time constraints of the target thread can be propagated to the exception handler. In this way, the exception processing can proceed without risk of a scheduling anomaly. The return value in the reply from the exception upcall indicates to the kernel whether the alert was successfully handled or not. If it was not handled then thread D is terminated (just as with any unsuccessfully handled exception) and the alert is raised on thread C, the new head of the chain. Alerts will be back propagated up the chain until the thread originally alerted is reached.

Orphans

Node failures or other events such as task or thread termination can result in broken chains. Detecting and eliminating the orphaned chain fragment in a timely fashion is important in real-time systems. Responding to whatever failure or event caused the chain to break is as important as responding to any other external event. In some systems, timely response to failures is more important than the processing of ordinary events.

When a chain is broken the rooted portion of the chain is immediately restarted. It returns from the RPC invocation with an error indicating the chain was broken and takes whatever action is deemed appropriate by the application.

An orphan alert is posted to the head of the orphaned chain. The alert is propagated to the head of the chain where an orphaned exception is generated. This is a fatal alert than cannot be handled correctly, i.e., when the orphaned exception handler returns the thread is terminated and the alert is raised on the next thread up the chain. This gives each thread on the chain a chance to clean-up (release locks, undo or perform compensating actions) before being terminated.

Characterization Tools (ETAP)

ETAP (Event Trace Analysis Package) [14] is a tool for characterizing the performance and behavior of real-time applications as well as the system software. ETAP is straightforward in design. The kernel reserves a block of memory as a circular message buffer. The size of the buffer is configurable. The kernel has been instrumented with a variety of probes. When activated these probes create entries in the circular buffer. Probe entries contain a type field, a timestamp, a thread ID tuple, and probe specific information. Probes can be used to capture a wide range of information such as context switching events, system calls, lock events, device events, etc. There are global and per thread probes. Any subset of the probes can be dynamically activated or deactivated. Applications with probes write to the buffer. There is a second task that reads the buffer and records it on disk where the information can be subsequently analyzed using different report generation programs. When configured into the kernel, inactive probes incur an insignificant overhead. Approximately 1 percent when running AIM III.

The thread ID tuple identifies the thread and its shuttle or RPC chain. The RPC chain identifier allows us to determine which client thread a server is acting for. This is an invaluable tool for tracking the causal dependency of events in a client/server system. It has also become a valuable tool for debugging the kernel and applications.

Miscellaneous

In addition to the work already described we have made a variety of relatively small changes and additions to the microkernel. These include:

- POSIX compatible fixed priority scheduling policies (FIFO and RoundRobin)
- Configurable number of priorities (32 to 1024)
- Support for multiple clocks and an alarm (timeout) service.
- Counting Semaphores and Locks for inter-task synchronization
- Real-time threads library (cthreads assumes a timeshare scheduling policy)
- Extensions to IPC to provide control over the lazy evaluation of buffer copies.

We are currently experimenting with a scheduling framework [15]. This framework both simplifies the task of creating a new scheduling policy, such as Earliest Deadline First or Best Effort and coordinates the scheduling of threads and synchronizers. Including the synchronizers in the framework enables the development of scheduling policies that can deal with scheduling anomalies such as priority inversion.

Networking with CORDS

The support for networks in Mach3.0 was limited to the packet filter. Protocols and network transparent IPC were expected to be implemented as user space servers. This had a negative effect on the performance of most uses of the network. The architecture also presented significant obstacles to a correct implementation of network IPC. In OSF MK we have added an object oriented framework for network protocols, Communication Objects for Reliable, Distributed Systems (CORDS) [9]. CORDS is derived from the xkernel, developed by the University of Arizona [10].

The CORDS framework has many features to simplify the task of implementing network protocols. Complex protocols can be decomposed into a graph of “micro-protocols”. A protocol graph can be extended across protection boundaries permitting portions of a protocol graph to exist in a task while other parts exist in the kernel.

There is a notion of a “path” that describes the route a message will take through the protocol graph. Resources such as message buffers and threads can be attached to paths allowing the protocol designer to manage the resources needed to guarantee the end-to-end quality of service needed by the application. Paths also provide a natural means for the treatment of protocol parallelism. We have used the framework to develop a real-time distributed clock protocol based on the Cristian algorithms, node-alive protocol, ordered reliable broadcast protocols, Mach IPC and RPC protocols among others.

Multi-Computers and Clusters

DIPC (Distributed IPC) and XMM (eXtended Memory Management) [13] provide transparent inter-node communication and shared memory on NORMA (No Remote Memory Access) architectures.

DIPC extends Mach IPC in a way that permits applications running on any node to view Mach abstractions such as tasks, threads, memory objects and ports in a transparent way. XMM supports distributed shared memory.

The OSF/1 AD system uses these two Mach subsystems to provide a scalable, single system image of UNIX. It is intended for massively parallel processing environments, such as the MPP Intel Paragon, but also for clusters of interconnected workstations.

Configurable Kernel

With all these extensions, the microkernel has grown to become unacceptably large. We want the microkernel to run on low-end machines and we also want to target embedded systems, so we embarked on a program to make most of the microkernel's features configurable[12]. The target is a minimal microkernel that could run on a compute-only node and would only perform basic scheduling and IPC.

Miscellaneous

We are developing or planning other projects on OSF MK, which are less relevant to the Linux server project because they are not integrated in the mainline microkernel or not freely available. These projects include:

- fault tolerance (still in the design phase),
- high trust: this implies a complete re-implementation of OSF MK in C++ with strict layering for even better modularity [16].

3. Port of Mach to PowerMac

Introduction

The OSF MK microkernel is a mature technology on various machine types and processor architectures. Apple Computer Inc. asked the OSF to make available a free, microkernel based operating system on their PowerMacintosh series of computers. The major part of the port to PowerMacintosh was that of porting the OSF MK microkernel.

The range of PowerMacintosh machines use various different PowerPC processors, together with very different machine architectures, ranging from a board architecture very close to that of the 68000-based Macs through to the latest CHRP machines. We initially targeted the PowerMac 8100/80 machines which contain the PowerPC 601 microprocessor and a board architecture similar to that in the 68000 Macs.

To illustrate the clean separation between machine-dependent and machine-independent code, we note that in porting OSF MK to the PowerMac, the modifications to the generic parts of OSF MK consisted of two minor source file additions to preprocessor options.

The Early Stages of a Kernel's Life

When porting an operating system to any new processor, the development environment and compiler tool chain is the first thing which needs to be in place. Since the PowerMac did not have any Unix-like environment available, we set up a cross-compiling environment to build both Mach and, later, the Linux server. The host machines used for the development were x86 machines running Linux, and HP700 machines running OSF/1. Midway through the development, the x86 machines were switched from the monolithic Linux to running the Linux server, in order to provide a self-hosting alpha test site.

The compiler tool chain chosen was GCC 2.7.1, which supports the cross-compilation of PowerPC code using the elf binary format. We also used a crossed version of GDB for remote debugging via a serial line, allowing symbolic debugging of the kernel from an extremely early stage.

Once the tool chain is in place, work can begin on porting the kernel. Initially, the only machine-dependent requirements are a means of I/O, traditionally done via a serial port connected to the development machine. Below is a list of the steps taken in the kernel port:

- simple program cross-compiles and links.
- program loads and performs minimal polled I/O on the serial line
- `printf` works
- remote debugging stubs work
- kernel code linked in to remote debugger stubs
- kernel VM initializes
- traps and exception handlers (first VM fault)
- building user libraries and first user process
- bootstrapping first user process
- first system call from user process works
- clock interrupts implemented
- interrupt driven console device runs
- testing and debugging

The first user task that was executed was a kernel benchmarking and testing suite called MPTS (Microkernel Performance Test Suite) [19] and it took approximately 5 engineer-months to have a booting system which correctly executed the full benchmarking suite.

Once the minimal kernel functionality is complete, there remains the issue of device drivers. Device drivers are more platform dependent than they are processor dependent, and on many platforms device drivers may be reused or easily adapted from those written for previous ports. This was the case for both the serial line driver and for the SCSI controller on the

PowerMacs. Writing a small stub of PowerMac-dependent DMA code meant that the original drivers could be used with little modification.

Trade-offs

In any first implementation of an operating system, the trade-off of simplicity and debuggability is made against those of performance and functionality.

Once an initial version of the kernel is working, more time can be spent padding out stub routines and in optimizing those routines which were written for simplicity instead of performance (routines such as `bcopy` plus bit testing and setting routines start out written in C, before being optimized into hand-coded assembler).

Another trade-off made was to concentrate on obtaining functionality on the available test hardware. Minimal effort was made to cater for other processors in the PowerPC family or for other PowerMac machine architectures, since testing on these other machines was not possible. However, the assembly code and low-level exception handlers have been written so that it should be simple to incorporate the behavior of the other PowerPC processors (the 603 and 604 in particular). Additional device drivers and interrupt handling code will have to be written for the other machine architectures when porting to those machines.

4. Linux Server Architecture

A Single Server

Our Linux server is a “single server”, meaning that the entire Linux functionality resides in one single Mach task. The alternative to this design is the “multi-server” design, where functionality is split between smaller specialized tasks communicating though Mach RPC.

The multi-server design takes better advantage of the Mach architecture and allows one to re-use more code between different OS personalities: a generic terminal server could be shared by most OS servers for example. The drawbacks are performance and complexity. Performance is impacted by the cost of the extra communication between the various servers. Because servers can call each other in random ways, complex RPC chains are created, making it hard to implement some aspects of the OS functionality, like interrupting a system call for example. One has to implement potentially complex mechanisms to chase the system call through the various servers and abort it in a sensible way.

Although we are convinced that multi-servers are the way to go to produce high-quality operating systems on top of Mach, this strategy was not applicable in our case because we started from an existing monolithic kernel and we wanted to maximize the code reuse ratio to make it easier to track new releases of Linux and leverage the Linux community effort.

A Multi-Threaded Server

A server on top of Mach simply receives and replies to requests from user tasks or from the microkernel. It has no explicit control on scheduling nor hardware interrupts, so it cannot decide what it needs or wants to do at a given time. We do not want to add code everywhere to check if there is something more important to do (like receive an incoming network packet or disk block) or to manage explicit context switches when we can rely on Mach threads and the user-mode “cthreads” library. This library offers various synchronization primitives (simple locks, mutexes and condition variables) and hides most of the necessary synchronization of the underlying Mach kernel threads.

The Linux server has dedicated threads to handle the following tasks:

- keep track of time (“jiffies”)
- process replies to asynchronous device requests
- process incoming network packets
- process pager requests
- dispatch fake interrupts
- idle thread, to wake up timed out tasks and tasks with pending signals
- process system call and exception messages from user tasks.

Most of these threads wait on a dedicated Mach port to receive a messages. The system call threads are managed as a pool and wait on a port set regrouping all the user tasks’ exception ports. A system call thread is not dedicated to a given user task.

The system is serialized by a global mutex. Server threads must acquire it before doing anything sensible and release it when about to block.

System Call Redirection

As we mentioned earlier, communication between the user applications and the server is a critical issue for performance. Mach3 offers a system call emulation facility based on the redirection of the control flow to an *emulation library,* a piece of code that resides in the user address space and is able to communicate with the server via Mach RPC. For performance reasons, this library can implement some system calls (getpid, signal mask operations, etc...) locally without any interaction with the server [18].

Although this provides excellent performance, it means that the server functionality is shared between this emulation library and the server itself, leading to extra complexity and consistency problems; the server cannot really consider the emulation library like the rest of the user code, especially with respect to signal handling. The emulation library is not protected from user access and is therefore a potential Trojan horse for a malicious user. It is extremely complex (and inefficient) to protect the server against malicious usage of the emulation library's privileged communication to the server. Furthermore, multi-threaded applications imply even more complexity for the emulation library, which has to be fully-reentrant and has to identify the user threads.

For these reasons, we decided against the use of such an emulation in our servers. We extended the Mach exception mechanism to be more flexible and efficient [17]. With OSF MK, a system call from a user task raises an exception and enters the microkernel, which sends an exception RPC to the server, providing the user thread's state. This is similar to the way a system call enters a traditional UNIX system.

Combined with the collocation, thread migration and short-circuited RPC microkernel improvements, this method has proved to have competitive performance.

User Memory Access

Having the Linux server running as a regular user task makes it harder for it to access the memory of its user processes. The monolithic Linux kernel just uses segment registers to get inexpensive access to the user address space, but the Linux server has to use the Mach VM interfaces. Since this is also a critical aspect for the overall system performance. We cannot afford to suffer the overhead of switching to the microkernel for each access to user memory. We map the necessary user memory areas into the server's address space using Mach VM services. Once the mapping is done, the server can access the memory without any performance penalty.

When the Linux server is collocated in the microkernel's address space, it can even avoid to setting up the mapping and use the microkernel's copyin and copyout mechanisms, which are similar to the monolithic Linux memcpy_fromfs and memcpy_tofs interfaces. There is still an extra cost because the server does not have direct access to the microkernel routines (it is a separately linked task) and has to go through short-circuited-RPC-like interfaces. This overhead is not a problem in itself, but is emphasized by Linux's habit of doing lots of very small (byte or word) copies at a time. By re-organizing some pieces of code in critical places (mainly in the exec path), we managed to get reasonable performance.

Device Access

The device drivers are in the microkernel, but the server has to access them and let its processes use them. Linux handles device numbers and uses its own device operation routines. Mach names its devices with regular names ("console", "hd0a", "fd0a", "sd0a", etc...) and offers its own device interfaces. In the Linux server, we just added a generic emulation layer, replacing the bottom half of most Linux device drivers.

The device emulation code fulfills two tasks:

- translate a Linux device (major, minor) into a Mach device name
- translate Linux device operations (open, read, write, etc...) into the appropriate Mach device requests.

Just as Linux devices get registered when they are detected, we register some additional information, for instance a couple of routines to translate a Linux device number into the equivalent Mach device name and the reverse. The "device operations" structure is replaced with more or less generic routines (we have a set of routines for block devices and another for character devices): accessing an IDE disk, a SCSI disk or a floppy does not make any difference for the Linux server once it has found which Mach device to use.

The device specific code is therefore reduced to the initialization routine, and most of these routines only differ by the Mach device name they register.

Scheduling

Scheduling is another of the microkernel's duties. The Linux "schedule" routine is still used to scan the task list for newly runable tasks, but no actual context switching between user processes is done in the Linux server. User tasks "block" when running in one of the server threads, using some Mach synchronization primitives provided by the "cthreads" library.

"schedule" calls "condition_wait" when the current task's state is no longer TASK_RUNNABLE and we call condition_signal whenever the task becomes runable again.

Fake Interrupts

In the monolithic Linux, like in most Unix systems, pending signals are delivered to a user process when it returns from kernel mode to user mode. A user process can enter kernel mode when issuing a system call, when causing an exception (page fault, arithmetic error, etc...) or when interrupted by a hardware interrupt (like a clock interrupt). The Linux server follows this scheme, but will never be interrupted by the hardware since interrupts are handled by the microkernel. This means that a user process looping in user-mode without doing any system calls or exceptions is virtually unkillable because signals will never be delivered to it.

This problem has affected all Unix servers on top of Mach3. Some decided to solve this issue by adding an extra thread in the emulation library, to listen for messages from the server, and forcing the real user thread to check for signals when required. Despite the fact that it adds even more complexity to the emulation library, this could not be applied to the Linux server because we rejected the emulation library solution in the first place.

Our solution was to implement *fake interrupts* to allow the Linux server to regain control of a user process even if does not cooperate. The server takes control of the user thread, gets its state and jumps to the system call return code where signals will be processed. Race conditions with a possibly incoming or returning system call are avoided by suspending the user thread and making sure that it's suspended in a safe place, using the "thread_abort_safely" Mach service. Of course, thread_abort_safely will fail if there's an exception message on its way to or from the server.

Linux Jiffies Emulation

In the monolithic Linux, the "jiffies" global variable counts clock ticks since the system start-up time. It is incremented in the clock interrupt handler and is widely used throughout the rest of the system.

The Linux server does not receive clock interrupts and the only way for it to count time is to use the Mach alarm services, which are obviously more expensive than a simple increment every 10 milliseconds. The OSF/1 server does not manage its own idea of the time and relies on the microkernel for that. It has a time-out thread which keeps on blocking and requesting to be woken up by Mach when the next OSF/1 time-out expires. Unfortunately, the wide usage of "jiffies" and our desire to maximize code-reuse forced us into emulating the Linux kernel's behavior.

A jiffies thread is woken up at regular intervals by the microkernel and increments "jiffies" by the amount of clock ticks that have passed during its sleep. The interval could be set to exactly one clock tick, in which case we would have the same clock precision as the monolithic Linux, at the expense of a context switch and some overhead every 10 ms. Although we have not measured the impact of this overhead yet, we currently use a 100ms interval.

The real time itself can be obtained more accurately from the microkernel by mapping it in the server's address space, and updating the Linux "xtime" global variable before sensitive uses.

Linux VM Emulation

The Mach VM interfaces allows a user task to create memory objects and map them in an address space. Page management is totally hidden by the microkernel. The obvious place to connect Linux VM with Mach VM is therefore the "vm_area" structure management, which is roughly the equivalent of the "vm_map" structure in Mach.

The Linux page table management code could be discarded if Linux did not reference the page tables so widely. To minimize changes to the original Linux code, we chose to provide a machine-independent dumb emulation of the page table macros and routines. The Linux server does not make any sensible use of these page tables and they are mostly empty, but it allows more Linux code to compile and run unchanged.

VM Mappings

When Linux establishes a VM area, the Linux server has to use the Mach VM interfaces to perform the equivalent operation. We do that in the Linux "insert_vm_struct" and "remove_shared_vm_struct" routines.

Mapping a file is done by creating a memory object associated with the file and establishing the mapping with the "vm_map" Mach interface. Allocating zero-filled memory (for the "brk" system call for example) is done with the "vm_allocate" Mach interface. When removing a mapping, we use the "vm_deallocate" Mach interface.

This simple emulation has minimal impact on the original Linux code and covers the vast majority of the Linux VM operations. Unfortunately, we had to rewrite some Linux code to make it more Mach-friendly. For example, the "brk" system call shrinks a VM area by removing the old mapping and establishing a smaller one. This works in Linux because the page tables are not touched so the old memory is still there when the new mapping comes in place. Our emulation code discards the memory when removing the old mapping and cannot resuscitate it when establishing the new mapping. We just re-arranged the Linux code to avoid the “remove and replace” trick.

Memory Map

The “mem_map” array contains an entry for each physical memory page. Since it is widely used throughout the Linux code, we also chose to emulate this array and use the Linux page allocator. The server allocates a virtual memory area as big as the physical memory and uses this pool of pages when it needs memory, using the Linux “get_free_page” and “free_page” routines.

This is unnecessarily inefficient and restrictive, and we would like to get rid of this implementation in a future release. If the original Linux code did not use the mem_map array explicitly but hid it under macros or in-line routines, it would have given us freedom to implement whatever page allocation algorithm suits our architecture. This problem is an illustration of the advantages of modularity, which allows wider choices of implementation by reducing the interdependencies between system components.

External Inode Pager

As mentioned in the previous paragraph, mapped files are implemented by creating a memory object associated with the file and mapping this memory object. The Linux server then has to serve paging requests from the microkernel for this memory object. This is done by an external memory manager that we called the “inode pager”.

The inode pager is currently a single thread running in the Linux server task. It manages the relation between a Mach memory object and a Linux inode, and replies to microkernel paging requests.

When a page-in request comes in, the inode pager reads the required data from the disk and sends it back. For a page-out request, the microkernel sends the page inside the page-out message and the inode pager writes it back to disk. Of course, the microkernel only sends back dirty pages and silently discards clean ones, so the inode pager never has to write back text pages for example.

The inode pager is also responsible for flushing a memory object from the microkernel cache when needed, for example when a mapped binary is recompiled.

Dynamic Buffer Cache

The dynamic buffer cache was a decisive point when we chose Linux as the free UNIX to port on OSF MK. We had already experimented a dynamic buffer cache on the OSF/1 server with excellent results, but with some problems related to the OSF/1 file system design (it derived from BSD and is parallelized). We looked forward to making an easier experimentation with Linux, where everything was already in place.

The challenging part of a dynamic buffer cache for a Mach-based OS is that the buffer cache (in the Linux server) and the VM (in the microkernel) need to interact to let the system make the best use of the available memory.

Letting the buffer cache grow is the easy part: the Linux server manages only virtual memory and can therefore provide the buffer cache with more pages than there are in the physical memory. The tricky part is to get the buffer cache to shrink when the system is short on memory and before it starts paging.

Advisory Page Out

The microkernel doe not report memory shortage. When it really needs a page, it selects a physical page and sends a page-out request to the appropriate memory manager. The Linux server had no way to know that the system was short on memory before paging was already started.

OSF extended the external memory manager to offer “advisory page out”. That is, instead of un-mapping a page and sending it to the memory manager, the microkernel can now leave the page in place, send a “discard request” to the memory manager and let it take any appropriate action. The Linux server can then use the “try_to_free_page” routine and free a page other than the one selected by the microkernel. Of course, the microkernel cannot be made to rely on an external memory manager to eventually free a page. If the memory manager does not free a page in time, the microkernel will send the data to the default pager, a privileged and trusted memory manager.

On the Linux server side, the only major change required is to allocate the buffer cache pages from a separate memory object, the size of the physical memory, and backed by another external memory manager.

Avoiding Double Paging

Apart from making the best use of the available memory, a dynamic buffer cache also fixes a classic problem of Mach servers: double paging. The memory in the buffer cache comes from the disk and will eventually go back to the disk. There is some space reserved on disk for this data. Without a dedicated memory manager, buffer cache pages could be paged-out to the default pager's paging space when the microkernel decides to discard a buffer's memory. And when the Linux server decides to re-use this buffer (either to write it back to disk if it's dirty, or to overwrite it with another disk block), the page fault will cause the page to be read from the paging space. With a dynamic buffer cache, we get the opportunity to write the buffer back to disk (if dirty) or just discard it without any useless paging activity.

5. Linux Server on the PowerMac

Once the Linux server was robust on the Intel platform, and the Mach microkernel was ready on the PowerMac, the two needed to be wed. Since Linux has already been ported to some other PowerPC machines, we aimed to re-use as much of the code from the native Linux port as possible, with the goal of offering complete source and binary compatibility with this port of Linux on other PowerPC machines, in a similar way to that already done on Intel machines.

The major part of porting the Linux server was to adapt the necessary header files for the Mach server. This took approximately one week to do, and once this was done the server was able to start to boot on the PowerMac.

Being able to use the Linux server on PowerMac machines was not simply a question of porting the server, commands and libraries were also needed, together with a file-system from which Linux could boot. We added some code to the Mach kernel to recognize the disk label and partition tables on a Macintosh disk, and ported the mkfs tool from the Minix distribution to create an initial populated file-system. As for commands and libraries, we were able both to build them ourselves and also to recover commands and libraries from an early binary distribution of native Linux on the PowerPC. Below is a list of steps taken in porting the Linux Server to the PowerMac:

- porting the include files
- porting the server's processor-dependent state
- recognizing Mac disk labels and partitions
- creating a file-system
- booting the Linux server from the file-system
- running init (first server system call)
- paging file found
- running /etc/rc.S (first signal taken)
- single user # prompt
- multi-user
- networking
- self-hosting

The work to generate a minix file-system on a Mac disk was done in parallel to the port of the Linux server, using mach_perf as the server to boot from the file-system. Reaching the multi-user # prompt took less than three engineer-weeks from the start of the port to the PowerMac.

Differences with the native Linux

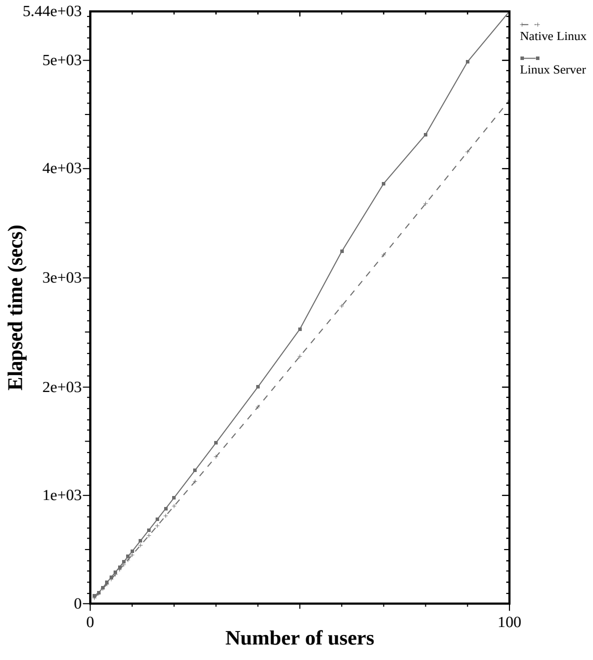
It is still a little early to give a complete listing of functional differences between native Linux on other PowerPC platforms and the Linux server on Power Macs, especially since both projects are under constant development. One current difference is that the Linux server offers the full 32-bit address space to user processes, rather than the restricted 31-bit address space offered by the native Linux.

6. Performance

We have not had time to make extensive performance measurements and analysis for the Linux server, but we ran a few benchmarks to get an idea of the critical areas. We used the NONAIM benchmark, derived from the AIM III benchmark, which measures the behavior of the system under an increasing workload, trying to simulate real life users behavior.

We also ran the Byte benchmarks which give finer results on the relative performance of more spe-

AIMIII 2.0 Elapsed time



cific parts of the system.

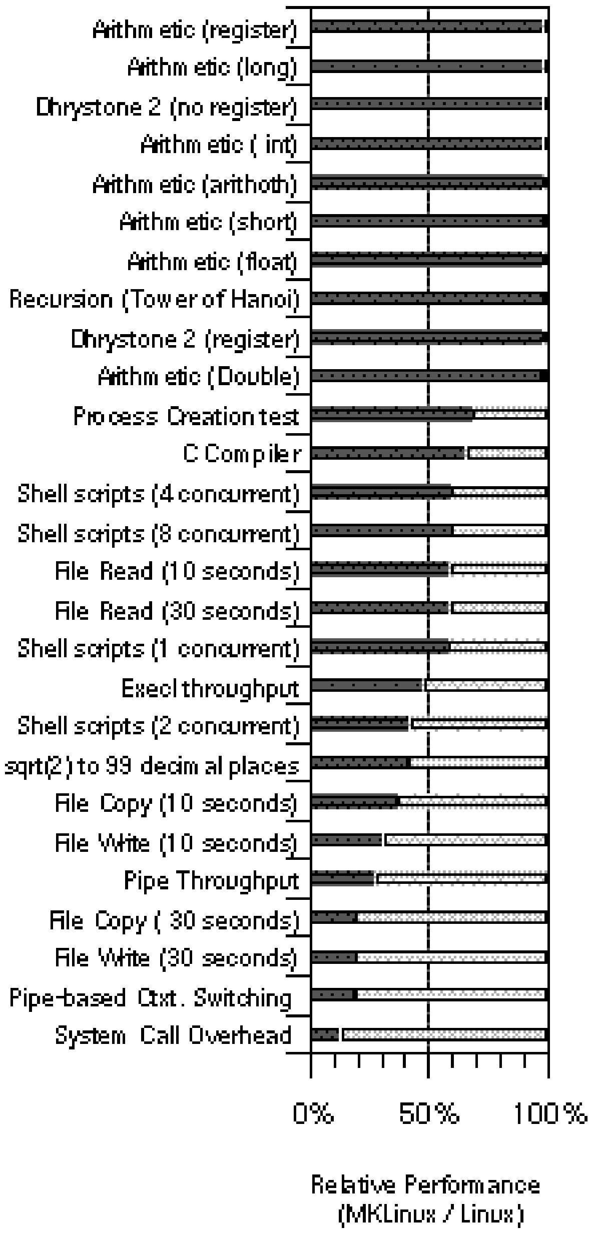
The benchmarks were run on a DEC PC450, with a 50MHz i486 and SCSI disks, running a Slackware3.0 ELF distribution. The benchmarks themselves were in a.out format and were run on a 1kB-block ext2 file-system.

We profiled the micro-kernel and Linux server during those benchmarks and performed quick optimizations in two areas:

• system call path,
• access to user memory.

These optimizations improved some benchmarks and we now have reasonable performance for AIMIII throughput when the system is not I/O bound. The Linux server reaches 93% of the AIMIII performance of the native Linux kernel.

Next, we started investigating disk I/O performance. The default file-system block size is 1 kilobyte on Linux. The Linux kernel is able to group disk requests into larger requests and doesn't suffer from the small block size. Neither the Linux server nor the micro-kernel perform such an optimization currently, and the penalty is made even worse by the extra overhead of Mach device interfaces. The result is that we read only one block per disk revolution.



Byte Benchmarks Results

Conclusion

Although the Byte results show there is still room for performance improvement, the AIM results are encouraging; even with the huge system call overhead penalty and the currently poor disk I/O performance, we are still within arm's reach of the native Linux kernel.

We plan to do more exhaustive performance measurements and analysis in the future and to extend our tests to multi-processor platforms.

7. Status and Future Work

Portability

It took approximately five engineer-months to convert Linux into a Linux server on the Intel x86 platform and two more engineer-week to port the Linux server to the PowerMac. Admittedly, this port was able to re-use some of the machine-dependent files from the Linux/PowerPC project, but it is a fact that the code that requires a lot of work when porting is not used by the Linux server. Bootstrap issues, device drivers, context switching, trap handling, and, last but not least, virtual memory management are all handled by the microkernel and what is left of these components in the Linux server is mostly machine independent.

What is left to port is:

• a few header files describing some VM constants, like page size and mask
• the system call and exception handling routine, which has to fill the Linux registers structure from the Mach thread state structure
• the "copy_thread" routine to initialize the state

of the new Mach thread during a fork system call
- the signal delivery routine which pushes the signal handler stack frame on the user stack.

Status on the Intel Platform

The two main missing pieces of functionality are virtual consoles, which seem necessary for X11 support, and the dynamic buffer cache support (the shrinking part). The performance is reasonable, although there is plenty of room for improvements.

We will address both the functionality and performance issues as soon as possible.

Status on the PowerMac Platform

At the time we write this paper, the PowerMac has just started booting Linux with a minimal file-system. It reaches the "login" prompt and is able to run most of the commands we have. We have a minimal set of drivers; we use a serial line as console and can access the SCSI disks. No floppy, CD-ROM, graphics or network drivers are available for the moment, but work is in progress on these.

Linux Code Base

The Linux server is currently based on the 1.2.13 Linux kernel version. It can be seen as a port of the Linux kernel to new architectures: osfmach3_i386 and osfmach3_ppc.

Because we have always taken care to minimize the changes to Linux code, it is fairly easy to upgrade to new Linux kernel releases. We will provide a Linux server based on the latest 1.3 kernel (or maybe 1.4) as soon as possible, but our current priority is to complete the ports to the PowerMac.

Linux Device Drivers

A team at Columbia University has developed a framework to include Linux device drivers into Mach with virtually no changes. This work was done on Mach4, but should be fairly easy to adapt to OSF MK. This would greatly improve the supported hardware list of the microkernel for the Intel x86 platform.

Development Environment

The OSF uses its own development environment, called ODE (OSF Development Environment). ODE is available for free from the OSF, and is currently required for building OSF MK. Although this still requires more work, OSF MK should compile for the x86 and PowerMac platforms from an Intel x86 system running Linux (server or monolithic), using a GCC compiler.

The Linux server is built using the same method and tools as the regular Linux kernel.

Being isolated from the hardware by the microkernel, the Linux server can share the machine with other operating system servers. In fact, it was developed as a regular OSF/1 process, started from a shell and debugged with a Mach-aware version of GDB (which can handle multi-threaded applications). This is a very powerful way to debug the system. There is no need to reboot the machine before each test and it provides full user mode debugging possibilities; it is possible to debug the Linux server with GDB from its very first instruction.

Although Linux does not support multi-threaded tasks (at least not in the way we would like it to), we were able to start a Linux server from another Linux server. And more generally, one could run whatever any desired system personalities in parallel on a single machine.

The microkernel can be debugged using the powerful (although afflicted with a weird syntax) kernel debugger on the Intel x86 platform, or using a remote GDB for the PowerMac.

Availability

The OSF MK microkernel and the ODE development environment are protected by the OSF Free Copyright, allowing free sources and binaries usage, copying and distribution under certain conditions, including some restrictions on commercial use. The Linux server is protected by the GNU General Public License.

They are both freely available from the OSF Open Software Mall (**http://www.osf.org/mall**).

8. Related Work

BSD-Lite Server

Johannes Helander, Jukka Virtanen and others in Finland developed a BSD-Lite server based on the BSD4.4 sources. This server has an architecture similar to the BSD4.3 UX server from CMU: it uses an emulation library mapped into the user task's address space and uses similar code to manage its threads (CMU c-threads).

They partially ported their server to OSF MK in March 1995, but their server architecture did not allow them to take advantage of OSF MK's performance improvements. The Lites server may be ported to the latest OSF MK free release, sometime in 1996.

We could have used their work as a basis for our free UNIX server, but, because we do not use emulation libraries and have rather different server architecture designs, we preferred to start from scratch. We also wanted to demonstrate that a non-BSD UNIX could be implemented on Mach. OSF/1 and BSD-4.4 have similar VM implementations, derived from Mach's VM, making it fairly straightforward to emulate with Mach interfaces.

GNU HURD

The GNU HURD has a very innovative and interesting design and might well be the first complete multi-server OS to tun on top of Mach. Several multi-server projects have been started at CMU and OSF, but they never quite reached product-quality, due to design flaws and a relative lack of interest for such complex software.

The GNU HURD is not yet available and we wanted to offer a development environment to our members and the research community as early as possible, so we produced yet another single server.

9. Conclusions

We have demonstrated that OSF MK could support a Linux personality server with reasonable performance and that the combination could be painlessly ported to a new hardware platform: the PowerMac.

We are now able to offer a completely free and unencumbered development environment based on the microkernel. We hope that the research community will find this environment attractive for their microkernel related projects.

Acknowledgments

We would like to thank Brett Halle, Michael Burg and others at Apple Computer, Inc. for the valuable PowerMac expertise they have offered during the port of the microkernel.

Thanks to Philippe Bernadat for helping us in measuring and analyzing the system's performance. Philippe Bernadat also helped with microkernel enhancements for the Linux server support on the Intel x86.

We also want to acknowledge the work of the team who has been porting native Linux to the PowerPC and especially Joseph Brothers, Daniel Puertas and Gary Thomas.

References

[1] Mike Acetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian and Michael Young. "Mach: A New Kernel Foundation For UNIX Development". *USENIX Proceedings, Summer 1986, pp. 93-112, Atlanta*.

[2] Paul Roy, David Black, Paulo Guedes, John Lo Verso, Durriya Netterwala, Faramarz Rabii, OSF RI, and Michael Barnett, Bradford Kemp, Michael Leibensperger, Chris Peak, Roman Zajcew, Locus Computing Corporation. "An OSF/1 UNIX for Massively Parallel Multicomputers". *OSF RI Collected Papers Vol. 2, 1993*.

[3] Bryan Ford, Jay Lepreau. "Evolving Mach 3.0 to use Migrating Threads". *UUCS-93-022*.

[4] Rozier, M. et al. "CHORUS Distributed Operating Systems". *Computing Systems 1(4), December, 1988*.

[5] Dan Swartzendruber. "A preemptible MACH kernel". *OSF RI Collected Papers Vol. 3, 1994.*

[6] Franco Travostino. "MACH3 Locking Protocol.". *OSF RI Collected Papers Vol. 2., 1993*.

[7] Uresh Vahalia, "UNIX Internals: The New Frontiers". *Prentice-Hall*.

[8] Ed Burke, Michael Condict, David Mitchell, Franklin Reynolds, Peter Watkins, Bill Willcox. "RPC Design for Real-Time MACH". *OSF RI Collected Papers Vol. 3, 1994*.

[9] Franco Travostino and Franklin Reynolds. "An O-O Communication Subsystem for Real-time Distributed Mach". *1994 IEEE Proceedings of the Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*.

[10] N.C. Hutchinson and L.L.Peterson. "The x-kernel: an Architecture for Implementing Network Protocols". *IEEE Trans. on Software Eng., vol. 17, no. 1, pp 64-76, Jan. 1991*.

[11] Philippe Bernadat, Christian Bruel, James Loveluck Eamonn McManus and Jose Rogado. "A Performant Microkernel based OS for the HP PA-RISC". *OSF RI Collected Papers Vol. 4, 1995*.

[12] David Black and Philippe Bernadat. "Configurable Kernel Project Overview". *OSF RI Collected Papers Vol. 4, 1995*.

[13] Bill Bryant, Steve Sears, David Black and Alan Langerman. "An Introduction to Mach 3.0's XMM system". *OSF RI Collected Papers Vol. 2, 1994*.

[14] Joseph Caradonna. "The Event Trace Analysis Package Design Specifications". OSF RI Collected Papers Vol.4, 1995.

[15] Robert Haydt, Joseph Caradonna and Franklin Reynodls. "Mach Scheduling Framework", OSF RI Collected Papers Vol.3, 1994.

[16] Keith Loepere et al. "MK++ Kernel High Level Design". 1995.

[17] Simon Patience. "Redirecting System Calls in Mach3.0: An alternative to the emulator". OSF RI Collected Papers Vol.1, 1993.

[18] David Golub, Randall Dean, Alessandro Forin Richard Rashid, “Unix as an Application Program”, *Usenix Conference Proceedings*, Summer 1990

[19] Philippe Bernadat. “Microkernel benchmarking techniques”. Slides. OSF RI Symposium ‘93.

The OSF RI Collected Papers are accessible on the WWW at **http://www.osf.org/os/os.coll.papers/**.

François Barbou des Places received the MSc. in Computer Science from the University of Paris XI, Orsay, France in 1987 and a *Diplôme d’Etudes Approfondies* from the University of Grenoble, France in 1989. He also graduated from the Ecole Nationale Supérieure d’Informatique et de Mathématiques Appliquées in Grenoble in 1989. He is currently a Research Engineer at the OSF Research Institute in Grenoble.

Nick Stephen received a first class honours BSc. in Computer Science from the University of Southampton, England in 1990. He is currently a Research Engineer at the OSF Research Institute in Grenoble, France. His research interests include microkernel operating systems and distributed systems.

Franklin Reynolds has been involved in the computer industry for over twenty years. For the last six years he has been employed at the Open Software Foundation’s Research Institute in Cambridge, MA. His current research interests include real-time and highly available distributed systems and active networks.