Evolution in Microkernel Design

Thorsten Scheuermann

COMP 242 Spring ’02

1 Introduction

A microkernel is an operating system kernel that provides only a small set core functionality which is required to execute with kernel privileges. In a microkernel−based operating system higher−level abstractions such as files or pipes are supposed to be implemented outside the ker−nel by using the offered kernel primitives. The goal of this paper is to give an overview of how microkernels developed since their introduction.

Section 2 summarizes two papers that give an overview of Mach, an early microkernel and show how a Unix system can be implemented on top of it. Section 3 covers Jochen Liedtke’s criticism of the first microkernel generation, presents possible ways to overcome their problems and a pa−per evaluating the real−world performance of a more recent microkernel. Section 4 concludes.

2 Mach

Mach is a microkernel developed at Carnegie Mellon University starting in the 80’s. It is one of the first of its kind. One of the most prominent current uses of Mach is as the foundation of Ap−ple’s Mac OS X operating system.

2.1 "Mach: A New Kernel Foundation For UNIX Development" [1]

This paper introduces Mach as a multiprocessor operating system kernel whose most notable features are a new virtual memory design and capability−based interprocess communication. The design of Mach as an extensible kernel is motivated by UNIX’s once clean and consistent inter−face to objects handled by the operating system through file descriptors and a small number of simple operations on them. Over time this consistency was weakened by adding more and more

mechanisms like System V IPC and BSD sockets. The philosophy of Mach is to provide "a small set of primitive functions" which can be used to build more complex services that are separated from the kernel. This modularization makes it easy to customize a system for different hardware. There are four basic abstractions provided by the Mach kernel: Tasks, threads, ports and mes−sages.

2.1.1 Tasks and Threads

A task consists of a paged virtual address space and provides an execution environment for threads. It also offers protected access to system resources like processors, port capabilities and virtual memory. Threads are being defined as "the basic unit of CPU utilization". All threads be−long to exactly one task and share access to all of it’s resources. In terms of switching between them, tasks require a high−overhead while threads generally have a low overhead.

2.1.2 Interprocess Communication Abstractions

In Mach a port represents a kernel−managed finite−length message queue that acts as an input port (one receiver, many senders). A message sent to a port is an arbitrary−sized collection of typed data objects. Ports are protected by capabilities which are obtained by receiving a message that contains capabilities to writing to a port. IPC can occur both synchronously or asynchro−nously. In the asynchronous case tasks are notified of the availability of a message through sig−nals. Sending of large messages between tasks is accomplished by mapping the sender’s memory containing the message into the receiver’s virtual address space with copy−on−write semantics. The paper is not clear about whether this happens for every message sent or only for messages exceeding a certain size limit.

The IPC facilities are used extensively throughout the system because all objects are represented by ports. Performing operations on a thread like suspend corresponds to sending an appropriate message to the port that represents that particular thread.

2.1.3 Microkernel Concepts

This first version of Mach does not emphasize the potential benefits of microkernels too much. However, an important microkernel concept that is implemented is memory paging in user space. It works by forwarding page fault events to a user−space paging server that is responsible for bringing the requested page into memory. Also, network−transparent IPC is implemented by having a user−level network server act as a proxy to tasks running on other machines on the net−work: Messages for remote tasks are sent to the network server which forwards the request to the network server on the destination machine. UNIX system calls and the file system are still im−plemented in kernel space which is an issue addressed by the second paper.

2.2 "Unix as an Application Program" [2]

This paper describes an effort to implement a classical Unix system that runs as a user task on top of the Mach microkernel, which is in line with the trend towards client/server computing: The Unix task acts as a server for traditional Unix facilities like files and sockets while the low−level hardware−dependent parts of a traditional Unix kernel are handled by Mach. Among the advantages of this system architecture are increased portability since the Unix server is mostly free of platform−specific code, network−transparency through the use of Mach IPC, and exten−sibility since it is conceivable to implement and test new Unix versions alongside the current Unix server.

A feature of Mach that is not mentioned in [1] is the ability to redirect system calls or traps so that they can be handled in user mode by the calling task. It is also stated in the paper that "the Mach kernel provides all low−level device support." IPC is used to transfer data from and to a device. It is unclear if the device drivers are part of the kernel or if they execute in user space.

2.2.1 Implementation

The implementation of the Unix server consists of two parts: The server itself and an emulation

library that transparently translates Unix system calls of user tasks into requests to the server. The server, which implements the major part of the Unix services provided, is running in a Mach task with multiple threads of control. Most server threads are used to handle incoming requests, but there are also some dedicated threads for actions that would be invoked through hardware interrupts in a traditional Unix kernel. Application programs communicate with the Unix server primarily through the Mach IPC facilities. A server thread is dispatched to handle each incoming message. Usually these messages translate directly into classical Unix system calls.

Files can be accessed through two interfaces: one uses IPC and one is memory−based. The IPC interface is provided to retain full network transparency, but it has limited performance in the case of the Unix server and the client application running on the same machine. For the memory−based interface the Unix server acts as a pager that maps the contents of a file into a memory region owned by the emulation library. The library then handles file accesses (read, write, lseek, etc.) of the application calling directly. This technique obviously does not work across the network but is much faster than the IPC−based interface.

2.2.2 Optimizations

The first version of the Unix server used only IPC for communication with the user programs. The performance of that version left room for improvement so a number of optimizations were made.

Due to the system’s extensive use of the IPC facilities their performance was optimized both through tweaking thread scheduling and by reducing the number of messages to the Unix server by handling some Unix system calls directly in the emulation library. File access was optimized by adding the memory−based interface as described above. Finally the C thread library used in the Unix server to coordinate all server threads had to be optimized. The original version mapped each C thread directly to a Mach kernel thread. The high number of kernel threads lim−

ited performance so the thread library was modified to have a number of C threads share one kernel thread.

2.2.3 Performance

Some benchmark data is presented in which the Unix server approach compares favorably to SunOS 4.0 and Mach 2.5 (which handles Unix system calls inside the kernel). Mach 2.5 comes off as being a little faster than the Unix server in the given benchmarks but both Mach−based systems beat SunOS in a compiler benchmark (SunOS: 49 sec, Mach 2.5: 28.5 sec, Mach w/Unix Server, 28.4 sec). It has to be kept in mind though that the performance of the Unix server was in part gained through giving up on a consistent IPC−based interface to the server.

3 Second−Generation Microkernels

3.1 "Towards Real Microkernels" [3]

Jochen Liedtke’s article gives a good overview of microkernel design issues, critiques early mi−crokernel development efforts and describes more recent approaches which might be successful in overcoming the identified problems. To Liedtke, the microkernel idea is very compelling, since such an architecture offers many advantages: The system becomes more modular and thus also more flexible and extensible, failing servers do not take necessarily the whole system down, it would be possible to implement different operating system strategies, e.g. several memory managers, in different servers running concurrently, and a small kernel should be easier to maintain and to keep bug−free.

However, first−generation microkernels of which Mach was the most successful one were un−able to realize these advantages. The author credits Mach’s external pager concept as the "first conceptual breakthrough toward real microkernels". Another important step was the handling of hardware interrupts as IPC messages that are generated by the kernel and processed by a user−level device driver. The problem with early microkernel boils down to limited performance and

the resulting compromises made to compensate for that. The most critical microkernel facility in terms of efficiency is IPC. On a 486−DX50 a regular Unix system call (~20us) has about 10 times less overhead than Mach RPC (~230us), where an RPC call consists of two IPC messages. These results are from 1991 and at that time 100us seemed to be the inherent cost of an IPC message. Liedkte claims that this cost makes early microkernels inefficient for current uses and cites a paper which shows that Mach has a peak performance degradation of up to 66% with certain Unix applications when compared to Ultrix (on a DECStation 5200/200). Moreover, first−generation microkernels are also believed to be unsuitable as the foundation of future applications since trends like object−orientation and distribution will cause a steady increase in the frequency of RPC calls.

There is also another weakness besides performance: The external pager concept turned out to be too inflexible because main memory is still managed by the kernel itself and the pager has only very limited control. Liedtke gives multimedia file servers, real−time applications and frame buffer management as example applications that require complete control over main memory.

The problem here is that the kernel implements a policy that only is convenient as long as it fits all applications, but hinders implementation of systems that require a novel, different policy.

Early microkernels evolved gradually from large monolithic kernels. Even though they reduced functionality they were still generally large and implemented a lot of concepts. Mach 3 for ex−ample has over 300 Kbytes of code and its API consists of approximately 140 system calls. Newer microkernels that try to overcome the weaknesses identified in their predecessors are build from the ground up with an emphasis on minimality and a clean and simple architecture.

Two examples of this approach are Exokernel and L4.

Exokernel is a hardware−dependent microkernel that was developed at MIT in 1994−95. It is tied to the Mips architecture and provides no abstractions but only a small set of primitives to

multiplex the hardware in a safe way. An RPC call based on the kernel's primitives takes only 10us on a Mips R3000 compared to 95us for a Mach RPC call.

L4 was developed at GMD in 1995. It is based on the premise that a microkernel should provide a minimal set of general abstractions and that it is inherently processor−dependent. Even com−patible processors like a 486 and a Pentium need different implementations of the same API for optimum performance. The only abstractions that L4 provides are address spaces, IPC and threads. It offers just 7 system calls with a code size of 12 Kbytes. IPC with a message size of 8 bytes takes 5us on a 486−DX50, with 512 bytes it takes 18us. The numbers for Mach IPC on the same machine are 115us and 172us.

Both Exokernel and L4 provide solutions to overcoming the performance problems. L4 also ad−dresses the problem of the external pager being too inflexible by providing a pure mechanism interface to virtual address spaces that allows memory management policies to be implemented outside the kernel. L4 allows the "recursive construction of address spaces outside the kernel" using three operations: *Grant* allows the owner of a page to map it to another process' address space and remove it from it's own. *Map* is similar to *grant* except for the fact that the memory page in question remains in the owner's address space which shares the page between the owner and the receiver. *Demap* allows the owner to revoke a previously mapped page from another ad−dress space.

The author admits that at the point of writing there has been no practical experience on whether second−generation microkernels can substantiate the potential benefits of microkernel architec−tures and that it is possible for new problems to arise when trying to use them in larger systems. However, the outlined performance enhancements in recent microkernel designs leave room for optimism.

3.2 "The Performance of Micro−Kernel−Based Systems" [4]

This last paper describes an effort to run Linux on top of the L4 microkernel in order to explore the suitability and performance of this second−generation microkernel for real−world applica−tions. Different benchmarks are used to compare L4Linux against native Linux and two different versions of MkLinux, a Linux implementation that runs on top of Mach 3.0.

Running Linux on top of L4 was accomplished using essentially the same approach as in [2]. To get an indication of whether L4 provides enough performance and whether it's interface is gen−eral and flexible enough, the developers restricted themselves from modifying L4 to support the Linux implementation on top of it. The Linux kernel runs as a server in an L4 task besides the user processes which also each execute in their own L4 task. L4 maps the whole physical mem−ory into the Linux server which acts as a memory manager for the user processes. Interrupts are forwarded to the Linux server through L4 IPC messages. Linux system calls are handled through IPC. L4Linux provides an emulation library that replaces the standard C library and invokes system calls through IPC messages.

Several benchmarks were run to evaluate the performance of L4Linux in comparison to an un−modified Linux system and two versions of MkLinux which run Linux on top of the Mach ker−nel. One MkLinux version runs Linux as a user process just like L4Linux does while in the other version the Linux server is co−located in the kernel. The benchmarks give insight into the ques−tions of what performance penalty the L4 foundation introduces compared to native Linux, whether the performance of the microkernel that Linux is running on matters and by how much co−locating the Linux server inside the kernel improves performance. All benchmarks are run on a 133MHz Pentium PC. The system call overhead was measured through the getpid() system call which is the shortest one in Linux. On native Linux a getpid() invocation takes 1.68us, on L4Linux 3.94us, on MkLinux in−kernel 15.41us and on MkLinux in user mode 110.60us. A number of other results of isolated operating system features as measured by the lmbench and

hbench benchmarks show similar results. The authors were surprised by the fact that the performance penalties of co−locating Linux in the Mach microkernel are still relatively high compared to running Linux on top of L4, but also admit that the L4Linux penalties are not as low as was hoped.

Another benchmark that simulated the workload of a multiuser system was used to see how per−formance is affected in a more realistic setting. In this benchmark on average over all system loads L4Linux achieves 91.7% of the performance of native Linux, MkLinux in−kernel 71% and MkLinux user 51%.

The benchmark results indicate that performance of the underlying microkernels matters for overall system performance and that co−location cannot solve the performance problem of an inefficient microkernel.

4 Conclusions

The microkernel approach to operating system construction does promise some compelling ad−vantages over classic monolithic kernels. The most important ones seem to be the increased modularity which brings the added benefit of flexibility in easily configuring the operating sys−tem for different classes of machines from handheld devices to large servers by choosing which memory managers, file systems, etc. to run. Current versions of Linux already offer this flexi−bility to some extent but configuration has to be done by compiling a custom kernel containing only the desired features. Even with a custom Linux kernel the individual components still all run in kernel space so that just one flawed component can compromise the stability of the whole system. User−space servers or device drivers on top of a microkernel are more secure in this re−gard because they are protected from each other through the memory−protection mechanisms provided by the hardware and the microkernel.

[4] gives a clear indication that Mach and other first−generation microkernels do not provide a

feasible foundation for future operating systems due to the considerable performance overhead introduced by these systems. The much more modest overhead introduced by recent microker−nels is easily offset by continuously increasing hardware speeds and seems to be a small price to pay for an extensible platform that facilitates future operating system development.

In the short term the most attractive market for microkernels appear to be embedded devices. Their hardware constraints (small memory, limited computing power) make small efficient mi−crokernels an excellent match. Moreover, since embedded devices are generally freed from the burden of backwards−compatibility it is much more reasonable to create a specialized embedded system around a microkernel than it is on a desktop PC. The QNX microkernel [5], which has the added benefit of being a real−time kernel has had some success in this market. Mainstream desktop operating systems do not have a history of being very innovative and instead rely on proven concepts, so microkernels will probably not make their way into that segment anytime soon, if at all. A notable exception is Mac OS X which is based on a BSD Unix foundation using Mach 3.0. However, considering the performance evaluations in the surveyed papers, especially [4], it is unclear what the actual benefits of using Mach are for this system.

I believe it will be very interesting to see more systems making use of microkernel technology in the future.

5 References

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

[2] Golub, D., Dean, R. Forin, A. and Rashid, R. Unix as an Application Program. In Usenix 1990 Summer Conference, June, 1990

[3] Liedtke, J. Towards Real Microkernels. In Communications of the ACM, 39(9):70−77, September 1996

[4] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., Wolter, J. The Performance of Micro−Kernel−Based Systems. In 16th ACM Symposium on Operating System Principles (SOSP), October 1997

[5] QNX Website, http://www.qnx.com