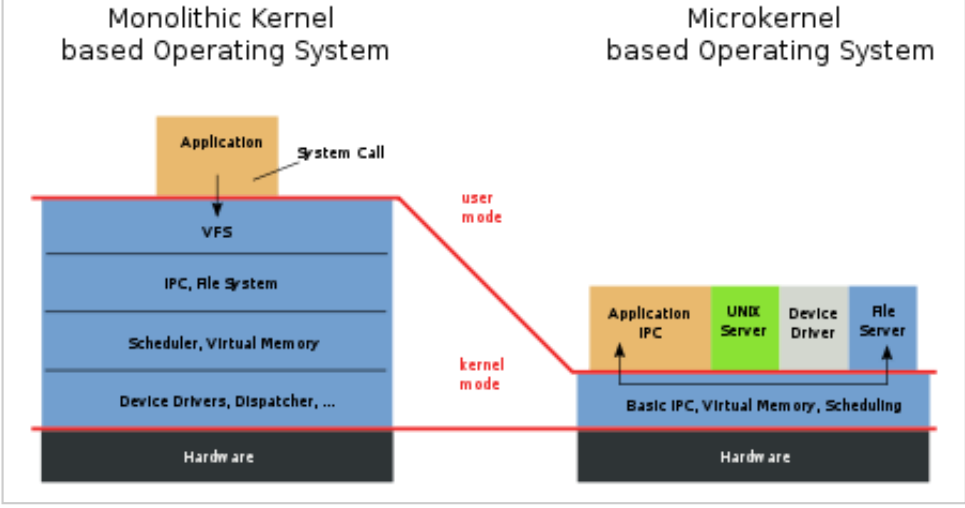
# Microkernel

From Wikipedia, the free encyclopedia

In computer science, a microkernel (also known as μ-kernel) is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system (OS). These mechanisms include low-level address space management, thread management, and inter-process communication (IPC).




Structure of monolithic and microkernel-based operating systems, respectively

If the hardware provides multiple rings or CPU modes, the microkernel may be the only software executing at the most privileged level, which is generally referred to as supervisor or kernel mode. Traditional operating system functions, such as device drivers, protocol stacks and file systems, are typically removed from the microkernel itself and are instead run in user space.[1]

In terms of the source code size, as a general rule microkernels tend to be smaller than monolithic kernels, usually sizing at under 10,000 lines of code. The MINIX 3 microkernel, for example, has fewer than 6,000 lines of code.[2]

## Contents

# History

Microkernels were developed in the 1980s as a response to changes in the computer world, and to several challenges adapting existing "mono-kernels" to these new systems. New device drivers, protocol stacks, file systems and other low-level systems were being developed all the time. This code was normally located in the monolithic kernel, and thus required considerable work and careful code management to work on. Microkernels were developed with the idea that all of these services would be implemented as user-space programs, like any other, allowing them to be worked on monolithically and started and stopped like any other program. This would not only allow these services to be more easily worked on, but also separated the kernel code to allow it to be finely tuned without worrying about unintended side effects. Moreover, it would allow entirely new operating systems to be "built up" on a common core, aiding OS research.

Microkernels were a very hot topic in the 1980s when the first usable local area networks were being introduced. The same mechanisms that allowed the kernel to be distributed into user space also allowed the system to be distributed across network links. The first microkernels, notably Mach, proved to have disappointing performance, but the inherent advantages appeared so great that it was a major line of research into the late 1990s. However, during this time the speed of computers grew greatly in relation to networking systems, and the disadvantages in performance came to overwhelm the advantages in development terms. Many attempts were made to adapt the existing systems to have better performance, but the overhead was always considerable and most of these efforts required the user-space programs to be moved back into the kernel. By 2000, most large-scale (Mach-like) efforts had ended, although OpenStep used an adapted Mach kernel called XNU, which is used in Darwin, an operating system serving as the open source part of both OS X and iOS.[3] As of 2012, the Mach-based GNU Hurd is also functional and included in testing versions of Arch Linux and Debian.

Although major work on microkernels had largely ended, experimenters continued development. It has since been shown that many of the performance problems of earlier designs were not a fundamental requirement of the concept, but instead due to the designer's desire to use single-purpose systems to implement as many of these services as possible. Using a more pragmatic approach to the problem, including assembly code and relying on the processor to enforce concepts normally supported in software led to a new series of microkernels with dramatically improved performance.

Microkernels are closely related to exokernels.[4] They also have much in common with hypervisors,[5] but the latter make no claim to minimality and are specialized to supporting virtual machines; indeed, the L4 microkernel frequently finds use in a hypervisor capacity.

# Introduction

Early operating system kernels were rather small, partly because computer memory was limited. As the capability of computers grew, the number of devices the kernel had to control also grew. Through the early history of Unix, kernels were generally small, even though those kernels contained various device drivers and file system implementations. When address spaces increased from 16 to 32 bits, kernel design was no longer cramped by the hardware architecture, and kernels began to grow.

The Berkeley Software Distribution (BSD) of Unix began the era of big kernels. In addition to operating a basic system consisting of the CPU, disks and printers, BSD started adding additional file systems, a complete TCP/IP networking system, and a number of "virtual" devices that allowed the existing programs to work invisibly over the network. This growth continued for many years, resulting in kernels with millions of lines of source code. As a result of this growth, kernels were more prone to bugs and became increasingly difficult to maintain.

The microkernel was designed to address the increasing growth of kernels and the difficulties that came with them. In theory, the microkernel design allows for easier management of code due to its division into user space services. This also allows for increased security and stability resulting from the reduced amount of code running in kernel mode. For example, if a networking service crashed due to buffer overflow, only the networking service's memory would be corrupted, leaving the rest of the system still functional.

## Inter-process communication

Inter-process communication (IPC) is any mechanism which allows separate processes to communicate with each other, usually by sending messages. Shared memory is strictly speaking also an inter-process communication mechanism, but the abbreviation IPC usually only refers to message passing, and it is the latter that is particularly relevant to microkernels. IPC allows the operating system to be built from a number of small programs called servers, which are used by other programs on the system, invoked via IPC. Most or all support for peripheral hardware is handled in this fashion, with servers for device drivers, network protocol stacks, file systems, graphics, etc.

IPC can be synchronous or asynchronous. Asynchronous IPC is analogous to network communication: the sender dispatches a message and continues executing. The receiver checks (polls) for the availability of the message by attempting a receive, or is alerted to it via some notification mechanism. Asynchronous IPC requires that the kernel maintains buffers and queues for messages, and deals with buffer overflows; it also requires double copying of messages (sender to kernel and kernel to receiver). In synchronous IPC, the first party (sender or receiver) blocks until the other party is ready to perform the IPC. It does not require buffering or multiple copies, but the implicit rendezvous can make programming tricky. Most programmers prefer asynchronous send and synchronous receive.

First-generation microkernels typically supported synchronous as well as asynchronous IPC, and suffered from poor IPC performance. Jochen Liedtke identified the design and implementation of the IPC mechanisms as the underlying reason for this poor

performance. In his L4 microkernel he pioneered methods that lowered IPC costs by an order of magnitude.[6] These include an IPC system call that supports a send as well as a receive operation, making all IPC synchronous, and passing as much data as possible in registers. Furthermore, Liedtke introduced the concept of the direct process switch, where during an IPC execution an (incomplete) context switch is performed from the sender directly to the receiver. If, as in L4, part or all of the message is passed in registers, this transfers the in-register part of the message without any copying at all. Furthermore, the overhead of invoking the scheduler is avoided; this is especially beneficial in the common case where IPC is used in an RPC-type fashion by a client invoking a server. Another optimization, called lazy scheduling, avoids traversing scheduling queues during IPC by leaving threads that block during IPC in the ready queue. Once the scheduler is invoked, it moves such threads to the appropriate waiting queue. As in many cases a thread gets unblocked before the next scheduler invocation, this approach saves significant work. Similar approaches have since been adopted by QNX and MINIX 3.

In a client-server system, most communication is essentially synchronous, even if using asynchronous primitives, as the typical operation is a client invoking a server and then waiting for a reply. As it also lends itself to more efficient implementation, modern microkernels generally follow L4's lead and only provide a synchronous IPC primitive. Asynchronous IPC can be implemented on top by using helper threads. However, versions of L4 deployed in commercial products have found it necessary to add an asynchronous notification mechanism to better support asynchronous communication. This signal-like mechanism does not carry data and therefore does not require buffering by the kernel.

As synchronous IPC blocks the first party until the other is ready, unrestricted use could easily lead to deadlocks. Furthermore, a client could easily mount a denial-of-service attack on a server by sending a request and never attempting to receive the reply. Therefore, synchronous IPC must provide a means to prevent indefinite blocking. Many microkernels provide timeouts on IPC calls, which limit the blocking time. In practice, choosing sensible timeout values is difficult, and systems almost inevitably use infinite timeouts for clients and zero timeouts for servers. As a consequence, the trend is towards not providing arbitrary timeouts, but only a flag which indicates that the IPC should fail immediately if the partner is not ready. This approach effectively provides a choice of the two timeout values of zero and infinity. Recent versions of L4 and MINIX have gone down this path (older versions of L4 used timeouts, as does QNX).

## Servers

Microkernel servers are essentially daemon programs like any others, except that the kernel grants some of them privileges to interact with parts of physical memory that are otherwise off limits to most programs. This allows some servers, particularly device drivers, to interact directly with hardware.

A basic set of servers for a general-purpose microkernel includes file system servers, device driver servers, networking servers, display servers, and user interface device servers. This set of servers (drawn from QNX) provides roughly the set of services offered by a Unix monolithic kernel. The necessary servers are started at system startup and provide services, such as file, network, and device access, to ordinary application programs. With such servers running in the environment of a user application, server development is similar to ordinary application development, rather than the build-and-boot process needed for kernel development.

Additionally, many "crashes" can be corrected by simply stopping and restarting the server. However, part of the system state is lost with the failing server, hence this approach requires applications to cope with failure. A good example is a server responsible for TCP/IP connections: If this server is restarted, applications will experience a "lost" connection, a normal occurrence in a networked system. For other services, failure is less expected and may require changes to application code. For QNX, restart capability is offered as the QNX High Availability Toolkit.[7]

To make all servers restartable, some microkernels have concentrated on adding various database-like methods such as transactions, replication and checkpointing to preserve essential state across single server restarts. An example is ChorusOS, which was made for high-availability applications in the telecommunications world. Chorus included features to allow any "properly written" server to be restarted at any time, with clients using those servers being paused while the server brought itself back into its original state. However, such kernel features are incompatible with the minimality principle, and are thus not provided in modern microkernels, which instead rely on appropriate user-level protocols.

## Device drivers

Device drivers frequently perform direct memory access (DMA), and therefore can write to arbitrary locations of physical memory, including various kernel data structures. Such drivers must therefore be trusted. It is a common misconception that this means that they must be part of the kernel. In fact, a driver is not inherently more or less trustworthy by being part of the kernel.

While running a device driver in user space does not necessarily reduce the damage a misbehaving driver can cause, in practice it is beneficial for system stability in the presence of buggy (rather than malicious) drivers: memory-access violations by the driver code itself (as opposed to the device) may still be caught by the memory-management hardware. Furthermore, many devices are not DMA-capable, their drivers can be made untrusted by running them in user space. Recently, an increasing number of computers feature IOMMUs, many of which can be used to restrict a device's access to physical memory.[8] (IBM mainframes have had IO MMUs since the IBM System/360 Model 67 and System/370.) This also allows user-mode drivers to become untrusted.

User-mode drivers actually predate microkernels. The Michigan Terminal System (MTS), in 1967, supported user space drivers (including its file system support), the first operating system to be designed with that capability.[9] Historically, drivers were less of a problem, as the number of devices was small and trusted anyway, so having them in the kernel simplified the design and avoided potential performance problems. This led to the traditional driver-in-the-kernel style of Unix,[10] Linux, and Windows before Windows XP. With the proliferation of various kinds of peripherals, the amount of driver code escalated and in modern operating systems dominates the kernel in code size.

## Essential components and minimality

As a microkernel must allow building arbitrary operating system services on top, it must provide some core functionality. At a minimum, this includes:

- some mechanisms for dealing with address spaces, required for managing memory protection
- some execution abstraction to manage CPU allocation, typically threads or scheduler activations
- inter-process communication, required to invoke servers running in their own address spaces

This minimal design was pioneered by Brinch Hansen's Nucleus and the hypervisor of IBM's VM. It has since been formalised in Liedtke's minimality principle:

> A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality.[11]

Everything else can be done in a usermode program, although device drivers implemented as user programs may on some processor architectures require special privileges to access I/O hardware.

Related to the minimality principle, and equally important for microkernel design, is the separation of mechanism and policy, it is what enables the construction of arbitrary systems on top of a minimal kernel. Any policy built into the kernel cannot be overwritten at user level and therefore limits the generality of the microkernel.[4] Policy implemented in user-level servers can be changed by replacing the servers (or letting the application choose between competing servers offering similar services).

For efficiency, most microkernels contain schedulers and manage timers, in violation of the minimality principle and the principle of policy-mechanism separation.

Start up (booting) of a microkernel-based system requires device drivers, which are not part of the kernel. Typically this means that they are packaged with the kernel in the boot image, and the kernel supports a bootstrap protocol that defines how the

drivers are located and started; this is the traditional bootstrap procedure of L4 microkernels. Some microkernels simplify this by placing some key drivers inside the kernel (in violation of the minimality principle), LynxOS and the original Minix are examples. Some even include a file system in the kernel to simplify booting. A microkernel-based system may boot via multiboot compatible boot loader. Such systems usually load statically-linked servers to make an initial bootstrap or mount an OS image to continue bootstrapping.

A key component of a microkernel is a good IPC system and virtual-memory-manager design that allows implementing page-fault handling and swapping in usermode servers in a safe way. Since all services are performed by usermode programs, efficient means of communication between programs are essential, far more so than in monolithic kernels. The design of the IPC system makes or breaks a microkernel. To be effective, the IPC system must not only have low overhead, but also interact well with CPU scheduling.

## Performance

On most mainstream processors, obtaining a service is inherently more expensive in a microkernel-based system than a monolithic system.[4] In the monolithic system, the service is obtained by a single system call, which requires two mode switches (changes of the processor's ring or CPU mode). In the microkernel-based system, the service is obtained by sending an IPC message to a server, and obtaining the result in another IPC message from the server. This requires a context switch if the drivers are implemented as processes, or a function call if they are implemented as procedures. In addition, passing actual data to the server and back may incur extra copying overhead, while in a monolithic system the kernel can directly access the data in the client's buffers.

Performance is therefore a potential issue in microkernel systems. Indeed, the experience of first-generation microkernels such as Mach and ChorusOS showed that systems based on them performed very poorly.[12] However, Jochen Liedtke showed that Mach's performance problems were the result of poor design and implementation, specifically Mach's excessive page cache footprint.[11] Liedtke demonstrated with his own L4 microkernel that through careful design and implementation, and especially by following the minimality principle, IPC costs could be reduced by more than an order of magnitude compared to Mach. L4's IPC performance is still unbeaten across a range of architectures.[13][14][15]

While these results demonstrate that the poor performance of systems based on first-generation microkernels is not representative for second-generation kernels such as L4, this constitutes no proof that microkernel-based systems can be built with good performance. It has been shown that a monolithic Linux server ported to L4 exhibits only a few percent overhead over native Linux.[16] However, such a single-server system exhibits few, if any, of the advantages microkernels are supposed to provide by structuring operating system functionality into separate servers.

A number of commercial multi-server systems exist, in particular the real-time systems QNX and Integrity. No comprehensive comparison of performance relative to monolithic systems has been published for those multiserver systems. Furthermore, performance does not seem to be the overriding concern for those commercial systems, which instead emphasize reliably quick interrupt handling response times (QNX) and simplicity for the sake of robustness. An attempt to build a high-performance multiserver operating system was the IBM Sawmill Linux project.[17] However, this project was never completed.

It has been shown in the meantime that user-level device drivers can come close to the performance of in-kernel drivers even for such high-throughput, high-interrupt devices as Gigabit Ethernet.[18] This seems to imply that high-performance multi-server systems are possible.

## Security

The security benefits of microkernels have been frequently discussed.[19][20] In the context of security the minimality principle of microkernels is, some have argued, a direct consequence of the principle of least privilege, according to which all code should have only the privileges needed to provide required functionality. Minimality requires that a system's trusted computing base (TCB) should be kept minimal. As the kernel (the code that executes in the privileged mode of the hardware) has unvetted access to any data and can thus violate its integrity or confidentiality, the kernel is always part of the TCB. Minimizing it is natural in a security-driven design.

Consequently, microkernel designs have been used for systems designed for high-security applications, including KeyKOS, EROS and military systems. In fact common criteria (CC) at the highest assurance level (Evaluation Assurance Level (EAL) 7) has an explicit requirement that the target of evaluation be "simple", an acknowledgment of the practical impossibility of establishing true trustworthiness for a complex system. Unfortunately, again, the term "simple" is misleading and ill-defined. At least the Department of Defense Trusted Computer System Evaluation Criteria introduced somewhat more precise verbiage at the B3/A1 classes, to wit, "The TCB shall [implement] complete, conceptually simple protection mechanisms with precisely defined semantics. Significant system engineering shall be directed toward minimizing the complexity of the TCB, as well as excluding from the TCB those modules that are not protection-critical."

## Third generation

Recent work on microkernels has been focusing on formal specifications of the kernel API, and formal proofs of the API's security properties and implementation correctness. The first example of this is a mathematical proof of the confinement mechanisms in EROS, based on a simplified model of the EROS API.[21] More recently, a comprehensive set of machine-checked proofs has been performed of the properties of the protection model of seL4, a version of L4.[22]

This has led to what is referred to as third-generation microkernels,[23] characterised by a security-oriented API with resource access controlled by capabilities, virtualization as a first-class concern, novel approaches to kernel resource management,[24] and a design goal of suitability for formal analysis, besides the usual goal of high performance. Examples are Coyotos, seL4, Nova,[25][26] and Fiasco.OC.[25][27]

In the case of seL4, complete formal verification of the implementation has been achieved,[23] i.e. a mathematical proof that the kernel's implementation is consistent with its formal specification. This provides a guarantee that the properties proved about the API actually hold for the real kernel, a degree of assurance which goes beyond even CC EAL7. It was followed by proofs of security-enforcement properties of the API, and a proof demonstrating that the executable binary code is a correct translation of the C implementation, taking the compiler out of the TCB. Taken together, these proofs establish an end-to-end proof of security properties of the kernel.[28]

## Nanokernel

The term nanokernel or picokernel historically referred to:

- A kernel where the total amount of kernel code, i.e. code executing in the privileged mode of the hardware, is very small. The term picokernel was sometimes used to further emphasize small size. The term nanokernel was coined by Jonathan S. Shapiro in the paper The KeyKOS NanoKernel Architecture (http://www.cis.upenn.edu/~KeyKOS/NanoKernel/NanoKernel.html). It was a sardonic response to Mach, which claimed to be a microkernel while Shapiro considered it monolithic, essentially unstructured, and slower than the systems it sought to replace. Subsequent reuse of and response to the term, including the picokernel coinage, suggest that the point was largely missed. Both nanokernel and picokernel have subsequently come to have the same meaning expressed by the term microkernel.
- A virtualization layer underneath an operating system, which is more correctly referred to as a hypervisor.
- A hardware abstraction layer that forms the lowest-level part of a kernel, sometimes used to provide real-time functionality to normal OS's, like Adeos.

There is also at least one case where the term nanokernel is used to refer not to a small kernel, but one that supports a nanosecond clock resolution.[29]

## See also

- Kernel (computer science)
  - Exokernel, a research kernel architecture with a more minimalist approach to kernel technology.
  - Hybrid kernel

    - Monolithic kernel
    - Loadable kernel module
- Trusted computing base
- Tanenbaum–Torvalds debate

# References


1. Jorrit N. Herder (2005-02-23). "Toward a True Microkernel Operating System" (http://www.minix3.org/theses/herder-true-microkernel.pdf) (PDF). minix3.org. Retrieved 2015-06-22.
2. "The MINIX 3 Operating System" (http://www.webcitation.org/64sAstWXM). minix3.org. Archived from the original (http://www.minix3.org/) on 2012-01-20.
3. "Porting UNIX/Linux Applications to Mac OS X" (http://developer.apple.com/library/mac/#documentation/Porting/Conceptual/PortingUnix/glossary/glossary.html#//apple_ref/doc/uid/TP40002859-TPXREF101). Apple. Retrieved 26 April 2011.
4. Liedtke, Jochen (September 1996). "Towards Real Microkernels". Communications of the ACM 39 (9): 70–77. doi:10.1145/234215.234473 (https://dx.doi.org/10.1145%2F234215.234473).
5. Heiser, Gernot; Uhlig, Volkmar; LeVasseur, Joshua (January 2006). "Are Virtual-Machine Monitors Microkernels Done Right?" (http://os.ibds.kit.edu/65_747.php) (PDF). ACM SIGOPS Operating Systems Review (ACM) 40 (1): 95–99. doi:10.1145/1113361.1113363 (https://dx.doi.org/10.1145%2F1113361.1113363).
6. Liedtke, Jochen (December 1993). "Improving IPC by kernel design". 14th ACM Symposium on Operating System Principles. Asheville, NC, USA. pp. 175–88. CiteSeerX: 10.1.1.40.1293 (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.1293).
7. http://www.qnx.com/download/download/8107/QNX_High_Availability_Toolkit.pdf QNX High Availability Toolkit
8. Wong, William (2007-04-27). "I/O, I/O, It's Off to Virtual Work We Go" (http://www.elecdesign.com/Articles/Index.cfm?AD=1&ArticleID=15350). Electronic Design. Retrieved 2009-06-08.
9. Alexander, Michael T. (1971). "Organization and Features of the Michigan Terminal System". Proceedings of the November 16–18, 1971, fall joint computer conference 40: 589–591. doi:10.1145/1478873.1478951 (https://dx.doi.org/10.1145%2F1478873.1478951).
10. Lions, John (1977-08-01). Lions' Commentary on UNIX 6th Edition, with Source Code. Peer-To-Peer Communications. ISBN 978-1-57398-013-5.
11. Liedtke, Jochen (December 1995). "On μ-Kernel Construction". 15th ACM symposium on Operating Systems Principles. pp. 237–250. doi:10.1145/224056.224075 (https://dx.doi.org/10.1145%2F224056.224075).
12. Chen, Bradley; Bershad, Brian (December 1993). "The Impact of Operating System Structure on Memory System Performance". 14th ACM Symposium on Operating System Principles. Asheville, NC, USA. pp. 120–33. doi:10.1145/168619.168629 (https://dx.doi.org/10.1145%2F168619.168629).
13. Liedtke, Jochen; Elphinstone, Kevin; Schönberg, Sebastian; Härtig, Hermann; Heiser, Gernot; Islam, Nayeem; Jaeger, Trent (May 1997). "Achieved IPC performance (still the foundation for extensibility)" (http://ieeexplore.ieee.org/xpl/RecentCon.jsp?punumber=4643). 6th Workshop on Hot Topics in Operating Systems. Cape Cod, MA, USA: IEEE. pp. 28–31.
14. Gray, Charles; Chapman, Matthew; Chubb, Peter; Mosberger-Tang, David; Heiser, Gernot (April 2005). "Itanium—a system implementor's tale" (http://www.usenix.org/publications/library/proceedings/usenix05/tech/general/gray.html). USENIX Annual Technical Conference. Annaheim, CA, USA. pp. 264–278.
15. van Schaik, Carl; Heiser, Gernot (January 2007). "High-performance microkernels and virtualisation on ARM and segmented architectures" (http://ertos.nicta.com.au/publications). 1st International Workshop on Microkernels for Embedded Systems. Sydney, Australia: NICTA. pp. 11–21. Retrieved 2007-04-01.

16. Härtig, Hermann; Hohmuth, Michael; Liedtke, Jochen; Schönberg, Sebastian (October 1997). "The performance of μ-kernel-based systems" (http://portal.acm.org/citation.cfm?id=266660&dl=ACM&coll=&CFID=15151515&CFTOKEN=6184618). Proceedings of the sixteenth ACM symposium on Operating systems principles: 66 – 77. doi:10.1145/268998.266660 (https://dx.doi.org/10.1145%2F268998.266660). ISBN 0-89791-916-5.
17. Gefflaut, Alain; Jaeger, Trent; Park, Yoonho; Liedtke, Jochen; Elphinstone, Kevin J.; Uhlig, Volkmar; Tidswell, Jonathon E.; Deller, Luke et al. (2000). "The Sawmill multiserver approach". 9th ACM SIGOPS European Worshop. Kolding, Denmark. pp. 109 – 114.
18. Leslie, Ben; Chubb, Peter; FitzRoy-Dale, Nicholas; Götz, Stefan; Gray, Charles; Macpherson, Luke; Potts, Daniel; Shen, Yueting; Elphinstone, Kevin; Heiser, Gernot (September 2005). "User-level device drivers: achieved performance". Journal of Computer Science and Technology 20 (5): 654 – 664. doi:10.1007/s11390-005-0654-4 (https://dx.doi.org/10.1007%2Fs11390-005-0654-4).
19. Tanenbaum, Andrew S.. "Tanenbaum-Torvalds debate, part II" (http://www.cs.vu.nl/~ast/reliable-os/).
20. Tanenbaum, A., Herder, J. and Bos, H. (May 2006).
21. Shapiro, Jonathan S.; Weber, Samuel. "Verifying the EROS Confinement Mechanism" (http://www.eros-os.org/papers/oakland2000.ps). IEEE Conference on Security and Privacy.
22. Elkaduwe, Dhammika; Klein, Gerwin; Elphinstone, Kevin (2007). Verified Protection Model of the seL4 Microkernel (http://ertos.org/publications/papers/Elkaduwe_GE_07.abstract). submitted for publication.
23. Klein, Gerwin; Elphinstone, Kevin; Heiser, Gernot; Andronick, June; Cock, David; Derrin, Philip; Elkaduwe, Dhammika; Engelhardt, Kai; Kolanski, Rafal; Norrish, Michael; Sewell, Thomas; Tuch, Harvey; Winwood, Simon (October 2009). "seL4: Formal verification of an OS kernel" (http://www.sigops.org/sosp/sosp09/papers/klein-sosp09.pdf) (PDF). 22nd ACM Symposium on Operating System Principles. Big Sky, MT, USA.
24. Elkaduwe, Dhammika; Derrin, Philip; Elphinstone, Kevin (April 2008). "Kernel design for isolation and assurance of physical memory" (http://ertos.nicta.com.au/publications/papers/Elkaduwe_DE_08.abstract). 1st Workshop on Isolation and Integration in Embedded Systems. Glasgow, UK. doi:10.1145/1435458 (https://dx.doi.org/10.1145%2F1435458).
25. "TUD Home: Operating Systems: Research: Microkernel & Hypervisor" (http://www.inf.tu-dresden.de/index.php?node_id=2697). Faculty of Computer Science. Technische Universität Dresden. 12 August 2010. Retrieved 5 November 2011.
26. Steinberg, Udo; Kauer, Bernhard (April 2010). "NOVA: A Microhypervisor-Based Secure Virtualization Architecture" (http://doi.acm.org/10.1145/1755913.1755935). Eurosys 2010. Paris, France. pp. 209 – 222.
27. Lackorzynski, Adam; Warg, Alexander (March 2009). "Taming Subsystems - Capabilities as Universal Resource Access Control in L4" (http://portal.acm.org/citation.cfm?id=1519135&dl=ACM). IIES'09: Second Workshop on Isolation and Integration in Embedded Systems. Nuremberg, Germany.
28. Klein, Gerwin; Andronick, June; Elphinstone, Kevin; Murray, Toby; Sewell, Thomas; Kolanski, Rafal; Heiser, Gernot (February 2014). "Comprehensive Formal Verification of an OS Microkernel". ACM Transactions on Computer Systems 32 (1): 2:1 – 2:70. doi:10.1145/2560537 (https://dx.doi.org/10.1145%2F2560537).
29. http://www.eecis.udel.edu/~mills/database/papers/nano/nano2.pdf

# Further reading

- scientific articles about microkernels (http://citeseer.csail.mit.edu/cs?q=microkernel) (on CiteSeer), including:
  - Dan Hildebrand (1992). "An Architectural Overview of QNX". Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures: 113 – 126. ISBN 1-880446-42-1. – the basic QNX reference.
  - Tanenbaum, A., Herder, J. and Bos, H. (May 2006). "Can We Make Operating

Systems Reliable and Secure?" (http://www.computer.org/portal/site/computer/menuitem.eb7d70008ce52e4b0ef1bd108bcd45f3/index.jsp?&pName=computer_level1&path=computer/homepage/0506&file=cover1.xml&xsl=article.xsl). Computer 39 (5): 44–51. doi:10.1109/MC.2006.156 (https://dx.doi.org/10.1109%2FMC.2006.156). -the basic reliable reference.
    - Black, D.L., Golub, D.B., Julin, D.P., Rashid, R.F., Draves, R.P., Dean, R.W., Forin, A., Barrera, J., Tokuda, H., Malan, G., and Bohman, D. (March 1992). "Microkernel Operating System Architecture and Mach". J. of Information Processing 14 (4). - the basic Mach reference.
- MicroKernel page (http://c2.com/cgi/wiki?MicroKernel) from the Portland Pattern Repository
- The Tanenbaum–Torvalds debate
    - The Tanenbaum-Torvalds Debate, 1992.01.29 (http://www.oreilly.com/catalog/opensources/book/appa.html)
    - Tanenbaum, A. S. "Can We Make Operating Systems Reliable and Secure? (http://www.computer.org/portal/site/computer/menuitem.5d61c1d591162e4b0ef1bd108bcd45f3/index.jsp?&pName=computer_level1_article&TheCat=1005&path=computer/homepage/0506&file=cover1.xml&xsl=article.xsl&)".
    - Torvalds, L. Linus Torvalds about the microkernels again, 2006.05.09 (http://www.realworldtech.com/forums/index.cfm?action=detail&id=66630&threadid=66595&roomid=11)
    - Shapiro, J. "Debunking Linus's Latest (http://www.coyotos.org/docs/misc/linus-rebuttal.html)".
    - Tanenbaum, A. S. "Tanenbaum-Torvalds Debate: Part II (http://www.cs.vu.nl/~ast/reliable-os/)".



Categories: Microkernels

---