A Survey of Microkernel Operating Systems

(CSE 221 Final Project)

Kirill Levchenko
klevchen@cs.ucsd.edu

Introduction

In a traditional, monolithic operating system, the kernel provides all the services to the application programs. As the demands on the operating system grow, the kernel becomes larger and more complex. The microkernel approach solves the problem by providing a minimal set of services allowing traditional operating system services to be provided by special server processes executing in "user" space.

This separation of kernel and server allows protection within the operating system itself, which becomes more important as operating systems provide a greater variety of services. A server fault does not bring down the system, only the faulting server, which can be restarted immediately. The microkernel approach also allows different operating systems to coexist, running on top of the microkernel. Microsoft's Windows NT operating system [9], for example, supports both OS/2 and POSIX operating system interfaces. Thus, applications with different needs may use the operating system best suited to them. This modularity also allows the operating system to be easily distributed by locating the servers on different computing nodes, provided a sufficiently capable IPC mechanism.

We survey several microkernel operating systems—RC 4000 Nucleus, StarOS, CHORUS, SPIN, Exokernel, L4, and QNX—roughly in chronological order, compare their features and consider how the notion of a microkernel evolves in each.

RC 4000 Nucleus

The Nucleus for the RC 4000 system [3] was first described in 1970 by Per Brinch Hansen from A/S Regnecentralen. Brinch Hansen was concerned with "the fundamental aspects of control of an environment consisting of parallel, cooperating processes." The result was the system Nucleus, which dealt only with supporting the process abstraction and interprocess communication. It was recognized that "it is essential for the orderly growth of the system" that additional features be implemented outside the Nucleus by operating system processes.

Executing processes in the system have a parent-child relationship tracing back to the basic operating system at startup. A parent can start and stop child processes as well as create and remove them. This provides a mechanism for program swapping and loading to be implemented outside the Nucleus by operating system processes. A child process' address space is allocated from its parent, mirroring the control hierarchy. Process scheduling, however, occurs without regard for this hierarchy.

The Nucleus supports IPC in the form of asynchronous messages. A process performs a *send message* operation asynchronously to send a message. The message is queued until the receiving process performs the *wait message* operation. The *wait message* operation is

synchronous; the process will be blocked until a message is received. The receiver may then perform the (asynchronous) *send answer* operation which delivers the reply to the sender which must perform the (blocking) *wait answer* operation to receive it. Thus Nucleus IPC also supports interprocess synchronization in a manner that is safer than Dijkstra's semaphores.

StarOS

StarOS [7] is a multiprocessor operating system designed to run on the Cm* multiprocessor at Carnegie-Mellon University starting in 1977. The Cm* multiprocessor used for StarOS consists of 50 processors in clusters of 10 (interconnected by a local bus) connected together by an intercluster bus. Each processor has limited memory, however the system as a whole has sufficient memory for a full operating system and applications. Because of the limited memory available on each processor, the operating system is distributed across several processors, each processor running a nucleus and some subset of the modules making up the operating system. Each module is made up of several processes, although StarOS processes are smaller and simpler than traditional processes.

The nucleus provides support for typed objects, capabilities, and asynchronous messaging. It allows an object's data portion to be mapped to memory, allowing it to be accessed using the processor's normal memory access instructions. StarOS objects are very much like those in object-oriented languages: an object can only be manipulated by member functions. Object allocation in memory is performed by a special module called the object manager, of which there is one for every processor cluster.

StarOS uses the modular and distributed nature of the system to achieve fault tolerance. A failure of a processor or a process requires only that the process be restarted, potentially on a different processor. In the even of failure, a process is suspended and a capability to the process is sent in a message to a special bailout mailbox designated for the failing process. In this way, process fault handling is facilitated by the system.

Microkernel design in StarOS was motivated by two factors: Parallel machine design meant that the operating system could perform several services at once, motivating a modular decomposition of the services. The small memories of the Cm* processors required that operating services be implemented on more than one processor, if a rich set was to be provided. Almost as a side effect, the system also provided good fault-tolerance characteristics. A microkernel-style operating system was the logical choice.

CHORUS

CHORUS [11] development started at INRIA, France in 1980. The system described here is the third version, developed in 1987. CHORUS is a distributed system providing a medium-sized microkernel, called the nucleus, and a set of system servers, which provide standard operating system functionality; one such set of servers provides Unix system services. The nucleus provides interrupt handling, scheduling, VM management, and IPC. The unit of execution is a thread, contained within an actor, which is the unit of resource ownership (what we would call today a process). An actor encapsulates an address space, a set of resources, and a set of threads. One type of resource is a port, which (to the owner) is a queue buffering incoming

messages and (to a sender) the address of a mailbox. A port can only be owned by one actor, however ports may migrate or be collected into port groups, which allow the actual port receiving the message to change. Thus asynchronous message-passing is one type of IPC supported by CHORUS, the other is RPC. While message passing offers a best-effort service model, RPC guarantees exactly-once delivery. If the target of the RPC goes down, the caller is notified. A third type of IPC is available to threads in the same actor, and that is communication via their shared memory.

Each port has a 128-bit unique identifier (UI), which acts as a capability to the port. Having the UI allows sending to the port. Controlling who is allowed to send to the port rests on the difficulty of guessing the UI of the port. Capabilities to other resources consist of the port UI of the server managing the resource and a key, which the server owning the objects interprets. Thus access control and enforcement is done by each server rather than the nucleus.

Information in CHORUS is stored in segments. Segments are managed by special actors called mappers which provide backing store and access control for the segments. A segment is identified by the UI of the mapper and a segment identifier local to the mapper. It may be mapped into areas of an actor's address space called regions. Any access to a region is automatically translated by the nucleus into an access to the segment. The nucleus VM manager does paging, invoking the appropriate mapper in the even of a page fault.

Interrupts, traps, and exceptions are handled by actors as well. When the nucleus receives an interrupt, it reports it one at a time to a list of actors registered for the interrupt until some actor handles it.

CHORUS is a distributed system in the sense that IPC is network-transparent, so a system server need not reside on the same network node (site in CHORUS) as the client; however each site must have a nucleus. In this sense CHORUS is like StarOS. Like StarOS, CHORUS also has capabilities, however CHORUS capabilities are not controlled by the system but by the server containing the object. CHORUS IPC is also simpler than StarOS: CHORUS messages are an untyped sequence of bytes, it is up to the receiver to interpret the information.

Although CHORUS allowed services to be distributed on the network, the network latency would not permit as much service distribution as StarOS. The intended benefit of microkernel design in CHORUS was support for multiple simultaneous operating systems that could provide a variety of services each tailored to an application's needs.

SPIN

The SPIN [2] operating system was proposed at the University of Washington in 1994. It recognizes the ability of microkernels to provide application-specific operating system services, arguing, however, that the existing model of an application communicating with a user-level server is inefficient because of the overhead of crossing from the application to the kernel, the kernel to the server, the server back to the kernel, and finally back to the application. This problem was addressed in Mach, an earlier microkernel, by moving the server into kernel space, eliminating half the overhead [10]. However this approach sacrifices protection within the operating system, one of the desirable features of microkernels, for the sake of performance competitive with monolithic systems.

SPIN takes a two-pronged approach to reducing the number of context-switches required

in an external operation. The first part, application-level libraries, places some of the server code inside the application so that simple requests to the server may be handled without switching contexts. The second part, spindles, places code inside the kernel, allowing kernel-level events to be handled efficiently without switching outside the kernel. These spindles are installed by the application into the kernel, where they register to receive certain kernel events, such as interrupts, or message deliveries. Because they operate at the kernel level, they can implement some services more efficiently than a server communicating via standard IPC. Spindles can also perform a wider range of services that would be possible outside the kernel.

In order to assure protection, spindles must be written in a special language and compiled by the kernel (or a trusted server). During compilation, the spindle is statically checked for security. Additional dynamic type and value checking must be done at run-time. The authors posit that modern compiler technology will allow compiled spindles to perform as efficiently as native kernel code.

SPIN raises some interesting questions: How should spindles interact (without interfering) within the kernel? Is the additional complexity of spindles acceptable to application programmers? SPIN does highlight, however, the inherent tension in microkernel systems between the benefits of protection and modularity, and the efficiency of such loose component coupling.

Exokernel

The idea of a minimal microkernel—exokernel [4]—comes from M.I.T.; it was first introduced in 1995. The goal of an exokernel is to provide the most minimal interface to the hardware that would allow secure resource sharing between user-level applications. Here, applications are really full operating systems implemented on top of the exokernel. Thus, rather than emulating hardware resources, an exokernel *exports* them, without attempting to support an abstraction. The result of this minimalist goal is a thin layer above the hardware that provides applications with secure binding to resources, a protocol for resource revocation, and a means to transfer control.

A secure binding allows the application to use a system resource, such as a page of memory or a TLB entry. An application's secure binding is checked only once, at bind time, so that subsequent accesses to the resource can be granted efficiently. In the case of a memory page, this is done by the hardware. Another way to provide these semantics is through application code, such as a packet filter, which is checked once when it is presented by the application. This is similar to SPIN's spindles, though spindles are more of an optimization to the system rather the implementation of it. Just as resources can be bound to an application, they can be revoked as well. Applications may be asked to return certain resources to the system, which they must do if possible. If a resource is not returned to the system when asked for, the exokernel may take it "by force," notifying the application. Further access to the resource will cause a security violation.

Finally, the exokernel provides a protected control transfer, which transfers control to a designated point in another process, yields the time slice to the process, and sets up to context. The control transfer does not affect application-visible registers, which can be used to pass data between processes. The result is a very efficient and extensible base for applications

implementing IPC.

A full operating system, ExOS was implemented on top of the Aegis exokernel and compared to Ultrix, a highly-optimized monolithic kernel. Benchmarks show ExOS on Aegis to be significantly faster at most operations. In many cases, this is a result of (legitimately) optimizing for special cases, which Ultrix fails to do because of its generality. Although ExOS does not have the full feature set of Ultrix, there is still some value in the results which prove that a microkernel-based approach to operating systems achieves the performance standard of a monolithic operating system. In the case of exokernels, performance is achieved by an application of the end-to-end principle to operating systems, pushing as much work up to the application level as possible.

L4

L4 [1] appeared in 1995 from the German National Research Center for Information Technology. Like Exokernel, it was a "second-generation" microkernel aimed at redeeming the microkernel cause. Also like Exokernel, it attacked the performance problem by simplifying the interface presented by the microkernel. However the aim of L4 was less iconoclastic: the microkernel supported to notion of threads and addresses spaces, with IPC between address spaces. A thread is an activity executing within an address space; the relationship is analogous to threads in actors in CHORUS. Threads are scheduled by the microkernel based on priority. L4 provides three operations on address spaces: grant, which transfers a part of one address space to another, map, which shares a part of one address space with another, and unmap, which revokes a sharing. Of course, the process receiving the address space must accept it. Page remapping forms one of the three types of IPC messages supported by L4. The other two are a register-based messages and a data buffer. The former places the parameters in registers while the latter copies data to the destination's address space. L4 IPC is synchronous, which simplifies the implementation and provides a means of process synchronization; it does, however, introduce a potentially undesirable dependency between processes.

To test the performance of L4, the Linux operating system was ported to run on top of L4 [5]. The result, L^{4}Linux, outperforms MkLinux (a Mach microkernel-based version of Linux), but performs slightly worse than native Linux. The latter should not be surprising: Linux was ported to L4 with few modifications, achieving full binary compatibility. The results are promising for a POSIX system built especially for L4, taking full advantage of its features.

QNX

QNX [6] is a commercial microkernel real-time operating system advertised by its creator as a "microkernel done right." It was created in 1981; the version described here is the present-day version 4.0. The microkernel provides interprocess communication, process scheduling, interrupt dispatching, and network communication. The last service may seem out of place: most minimal microkernels would implement network services as a separate server. Indeed, there is a Network Manager which provides such services, however it also attaches to the microkernel, allowing it to implement network-transparent IPC. This permits network transparency at all levels of the system, so that an application may use a server located on

another network node.

QNX IPC is synchronous and message-based. A processes that sends a message blocks until the receiver receives the message and sends a reply. Though blocking can limit the capability of the IPC system, the author argues that it is simple and fast, allowing more complex IPC to be implemented on top of this model.

QNX also provides a POSIX server, providing compatibility to existing Unix code, an essential feature in a commercial system. Another essential feature, of course, is performance, in which QNX claims to surpass even monolithic kernels, delivering performance limited only by the hardware. More importantly, QNX exists as a testament to the commercial viability of microkernel systems.

Conclusion

We started with Brinch Hansen's RC 4000 Nucleus, which was motivated by the protection offered by a microkernel approach as well as the appeal of an extensible system. Its influence cannot be underestimated: twenty-five years later, the modern L4 microkernel provides the same hierarchical address spaces and message-based IPC, although L4 IPC is simpler than that of Nucleus.

The StarOS microkernel approach was motivated by a different set of requirements, mainly those of a distributed environment with limited memory. Because each node could not, by necessity, contain an entire operating system, StarOS implemented most of its services outside the nucleus, which provided only the basic services. StarOS introduced the notion of small, light processes executing within the same address space in order to service requests more quickly. StarOS also realized some of the reliability inherent in microkernels.

CHORUS improved on some of the concepts of StarOS, clarifying the separation of address space and execution within the address space (actors and threads). Recognizing the limitation of providing a single type of IPC, it provided both RPC and asynchronous messages. It also cleanly pushed capabilities outside the kernel, making them entirely operating system dependent. Its goal was to provide a flexible framework on top of which distributed operating system could be built.

Unfortunately microkernels were severely underperforming with respect to their monolithic counterparts, forcing the issue of performance to the forefront of microkernel design. SPIN's approach is based on the recognition that the cost of crossing protection domains is amplified by the microkernel philosophy of service segregation. The solution was to allow operating system code both within the application (via libraries) and the kernel (via spindles).

However the real microkernel ideology [8] calls for increased modularity and service separation,which fundamentally requires lowering the cost of context switches rather than reducing the number of them. Both Exokernel and L4 follow this approach, though from slightly different angles. Both provide a very minimal set of services, however Exokernel simply exports the hardware in a secure way, categorically refusing to provide any "added value;" L4, on the other hand, permits some minimal amount of service abstraction. These microkernels also no longer attempt to provide hardware-independence, freeing them to optimize more vigorously.

The viability of the microkernel design has been shown both experimentally, in the

L4Linux system, and commercially in the QNX operating system, which uses microkernel design to provide real-time guarantees. It is safe to say that microkernels are not a passing trend, but a real alternative to monolithic systems.

References

[1] A. Au, G. Heiser, *L4 User Manual, Version 1.14*, University of New South Wales School of Computer Science and Engineering, 1999.

[2] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Prdyak, S. Savage, E. Sirer, "SPIN — An Extensible Microkernel for Application-specific Operating System Services," *University of Washington Dept. of Computer Science and Engineering Technical Report 94-03-03*, 1994.

[3] P. Brinch Hansen, "The Nucleus of a Multiprogramming System," *Communications of the ACM Vol. 13(4)*, 1970.

[4] D. Engler, M. Kaashoek, J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Symposium on Operating Systems Principles*, 1995.

[5] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, J. Wolter, "The Performance of µ-Kernel-Based Systems," *16th ACM Symposium on Operating Systems Principles (SOSP '97)*, 1997.

[6] D. Hildebrand, An Architectural Overview of QNX, `http://www.qnx.com/literature/whitepapers/archoverview.html`, 2001.

[7] A. Jones, R. Chanseler, I. Durham, K. Schwans, S. Vegdahl, "StarOS, a Multiprocessor Operating System for the Support of Task Forces," *Proc. 7th Symposium on Operating Systems Principles*, 1979.

[8] J. Liedtke, "Towards Real µ-Kernels," *Communications of the ACM Vol. 39(9)* , 1996.

[9] Microsoft, *Microsoft Windows NT, the Foundation: Design Goals, System Architecture*, `http://www.microsoft.com/ntserver/ProductInfo/Casestudies/NT/nt_Foundation.asp`, 2001.

[10] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, R. Sanzi, "Mach: A Foundation for Open Systems," *Proceedings of the Second Workshop on Workstation Operating Systems (WWOS2)*, 1989.

[11] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "CHORUS Distributed Operating Systems," *Computing Systems Vol. 1 (4)*, 1998.