Toward Real Microkernels

The inefficient, inflexible first generation inspired development of the vastly improved second generation, which may yet support a variety of operating systems.

THE microkernel story is full of good ideas and blind alleys. The story began with enthusiasm about the promised dramatic increase in flexibility, safety, and modularity. But over the years, enthusiasm changed to disappointment, because the first-generation microkernels were inefficient and inflexible. • Today, we observe radically new approaches to the microkernel idea that seek to avoid the old mistakes while overcoming the old constraints on flexibility and performance. The second-generation microkernels may be a basis for all types of operating systems, including timesharing, multimedia, and soft and hard real time.

The Kernel Vision

Traditionally, the word kernel denotes the mandatory part of the operating system common to all other software. The kernel can use all features of a processor (e.g., programming the memory management unit); software running in user mode cannot execute such safety-critical operations.

Most early operating systems were implemented by means of large monolithic kernels. Loosely speaking, the complete operating system—scheduling, file system, networking, device drivers, memory management, paging, and more—was packed into a single kernel.

In contrast, the microkernel approach involves minimizing the kernel and implementing servers outside the kernel. Ideally, the kernel implements only address spaces, interprocess communication (IPC), and basic scheduling. All servers—even device drivers—run in user mode and are treated exactly like any other application by the kernel. Since each server has its own address space, all these objects are protected from one another.

When the microkernel idea was introduced in the

Jochen Liedtke

late 1980s, the software technology advantages seemed obvious:

- Different application program interfaces (APIs), different file systems, and perhaps even different basic operating system strategies can coexist in one system. They are implemented as competing or cooperating servers.
- The system becomes more flexible and extensible. It can be more easily and effectively adapted to new hardware or new applications. Only selected servers need to be modified or added to the system. In particular, the impact of such modifications can be restricted to a subset of the system, so all other processes are not affected. Furthermore, modifications do not require building a new kernel; they can be made and tested online.
- All servers can use the mechanisms provided by the microkernel, such as multithreading and IPC.
- Server malfunction is as isolated as normal application malfunction.
- These advantages also hold for device drivers.
- A clean microkernel interface enforces a more modular system structure.
- A smaller kernel can be more easily maintained and should be less prone to error.
- Interdependencies between the various parts of the system can be restricted and reduced. In particular, the trusted computing base (TCB) comprises only the hardware, the microkernel, a disk driver, and perhaps a basic file system.[1] Other drivers and file and network systems need no longer be absolutely trustworthy.

Although these advantages seemed obvious, the first-generation microkernels could not substantiate them.

The First Generation

The microkernel idea met with efforts in the research community to build post-Unix operating systems. New hardware (e.g., multiprocessors, massively parallel systems), new application requirements (e.g., security, multimedia, and real-time distributed computing) and new programming methodologies (e.g., object orientation, multithreading, persistence) required novel operating-system concepts.

The corresponding objects and mechanisms—threads, address spaces, remote procedure calls (RPCs), message-based IPC, and group communication—were lower-level, more basic, and more general abstractions than the typical Unix primitives. In addition to the new mechanisms, providing an API compatible with Unix or another conventional operating system was a sine qua non; hence implementing Unix on top of the new systems was a natural consequence. Therefore, the microkernel idea became widely accepted by operating-system designers for two completely different reasons: (1) general flexibility and power and (2) the fact that microkernels offered a technique for preserving Unix compatibility while permitting development of novel operating systems.

Many academic projects took this path, including Amoeba [19], Choices [4], Ra [1], and V [7]; some even moved to commercial use, particularly Chorus [11], L3 [15], and Mach [10], which became the flagship of industrial microkernels.

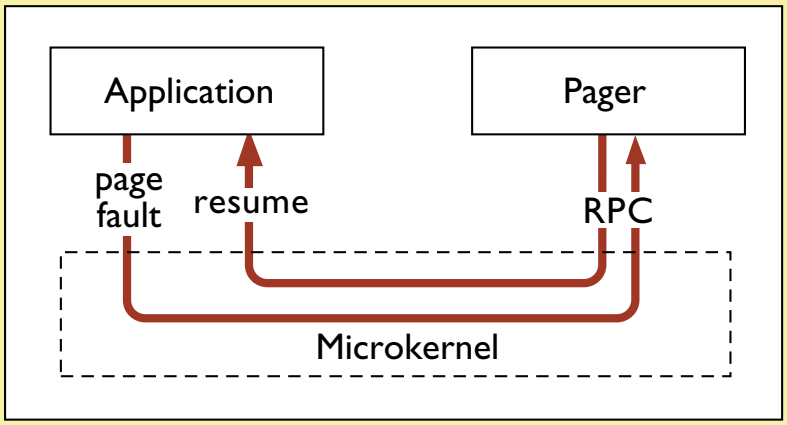

Figure 1. Page fault processing

Mach's external pager [22] was the first conceptual breakthrough toward real microkernels. The conceptual foundation of the external pager is that the kernel manages physical and virtual memory but forwards page faults to specific user-level tasks. These pagers implement mapping from virtual memory to backing store by writing back and loading page images. After a page fault, they usually return the appropriate page image to the kernel, which then establishes the virtual-to-physical memory mapping (see Figure 1). See the sidebar "Frequently Asked Questions on External Pagers."

This technique permits the mapping of files and databases into user address spaces without having to integrate the file/database systems into the kernel. Furthermore, different systems can be used simultaneously. Application-specific memory sharing and distributed shared memory can also be implemented by user-level servers outside the kernel.

The second conceptual step toward microkernels was the idea of handling hardware interrupts as IPC

[1]The TCB is the set of all components whose correct functionality is a precondition for security. Hardware and kernel both belong to the TCB.

Frequently Asked Questions on External Pagers

Is a pager required inside the microkernel? No.

How do user-level pagers affect system security? A pager can corrupt the data it maintains. So you rely on the correct functionality of the pager you use, whether it is kernel-integrated or user-level. However, different noninterfering user-level pagers can be used to increase security by, for example, holding sensitive data in a trustworthy and stable (standard) pager and less critical data in potentially less trustworthy pagers. Note that semantic dependencies are similar on all levels and are not operating-system specific; users rely as much on the correct functionality of a compiler as on that of a database system.

How expensive are user-level pagers? In principle, the overhead compared to that of an integrated pager is only one IPC and should be negligible. In practice, however, most first-generation microkernels use a complicated protocol with up to eight additional IPCs and implement IPC inefficiently. Overheads of 1,000 µs (Digital Equipment Corp. Alpha, 133 MHz) can occur. A microkernel of the second generation solves this performance problem by using only one additional IPC and making it fast (less than 10 µs).

messages [17] and including I/O ports in address spaces. The kernel captures the interrupt but does not handle it, instead generating a message for the user-level process currently associated with the interrupt. Thus, interrupt handling and device I/O are done completely outside of the kernel in the following way:

```
driver thread:
  do
     wait for (msg, sender) ;
     if sender = my hardware interrupt
       then read/write i/o ports ;
            reset hardware interrupt
       else . . .
     fi
  od .
```

In this approach, device drivers can be replaced, removed, or added dynamically—without linking a new kernel and rebooting the system. Drivers can thus be distributed to end users independent of the kernel. Furthermore, device drivers profit from using such microkernel mechanisms as multithreading, IPC, and address spaces. See the sidebar "Frequently Asked Questions on User-Level Device Drivers."

Disappointments

An appealing concept is only one side of the coin; the other is usefulness. For example, are the concepts flexible enough and the costs of that flexibility low enough for such real-world problems as multimedia, real time, and embedded systems?

With respect to efficiency, the communication facility is the most critical microkernel mechanism. Each invocation of an operating system or application service requires an RPC, generally consisting of two IPCs—the call and the return message. Therefore, microkernel architects spent much time optimizing the IPC mechanisms. Steady progress yielded up to twofold improvement in speed, but by 1991, the steps became less and less effective. Mach 3 stabilized at about 115 µs per IPC on a 486-DX50—comparable to most other microkernels. For example, a conventional Unix system call—roughly 20 µs on this hardware—has about 10 times less overhead than the Mach RPC. It seemed that 100 µs was the inherent cost of an IPC and that the concept had to be evaluated on this basis.

Since absolute time lacks meaning on its own, two more practical criteria are used for evaluation:

- Applications must not be degraded by the microkernel. This conservative criterion is a necessary precondition for practical acceptance.
- Microkernels must efficiently support new types of applications that cannot be implemented with good performance on conventional monolithic ker-

Frequently Asked Questions on User-Level Device Drivers

Can you really use a user-level disk driver for demand paging? Yes.

Can a user-level driver corrupt the system? Drivers are encapsulated in address spaces, so they can access only the memory and the I/O ports granted to them. From this point of view, they can corrupt any component relying on their functionality, but not the whole system. However, drivers control hardware, and if this hardware permits system corruption (e.g., by physical memory access via DMA), no kernel can prevent the driver from corrupting the system. Risk depends on the hardware accessible to the driver.

Do user-level drivers increase security? Yes, because of encapsulation. For example, a mouse driver can do no more harm than an editor.

How expensive are user-level drivers? Most first-generation kernels (except L3) implement all time-critical drivers as kernel drivers. However, due to the fast communication facilities of second-generation microkernels, user-level drivers perform as well as integrated drivers in these systems. Less than 10 µs are required per interrupt or driver RPC.

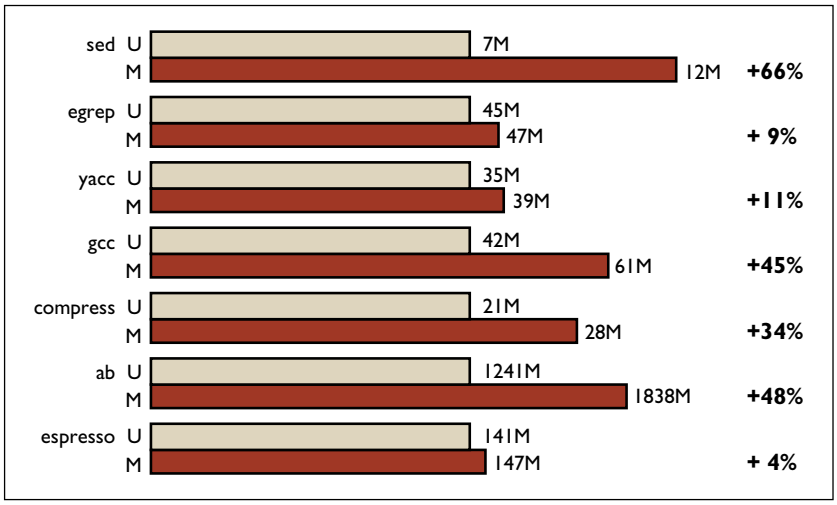

Figure 2. Non-idle cycles under Ultrix and Mach

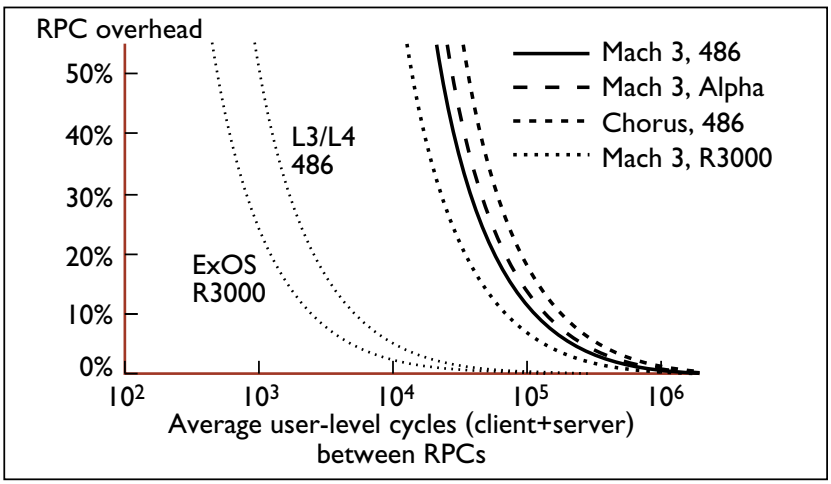

Figure 3. Relative RPC overhead

nels. This progressive criterion must be satisfied for a real advance in operating-system technology.

The conservative criterion can be evaluated by benchmarks running on the same hardware platform under a monolithic and a microkernel-based operating system. This method measures not only the primary (direct) IPC costs but also the secondary costs induced by structuring software with the client/server paradigm and by using the IPC mechanism.

Some applications performed as well under a microkernel as under a monolithic kernel, and a few slightly better. Unfortunately, other applications were substantially degraded. Chen and Bershad [6] compared applications under Ultrix and Mach on a DECStation 5200/200 and found peak degradations of up to 66% on Mach (compared to Ultrix) (see Figure 2). Condict, et al [8] compared an eight-user AIM III benchmark on a 486-DX50 under a monolithic OSF/1 and a Mach-based OSF/1 and measured an average 50% degradation. The measurements corroborate that the degradation is essentially caused by IPC: At least 73% of the measured penalty is related to IPC or activities that are its direct consequence; 10% comes from multiprocessor provisions that could be dropped on this uniprocessor; and the remaining 17% is due to unspecified reasons.

Chen found that the performance differences are caused in part by a substantially higher cache-miss rate for the Mach-based system. This result could point to a principal weakness of server architectures when used with microkernels. However, the increased cache misses are caused by the Mach kernel, invoked for IPC, not by the higher modularity of the client/server architecture.

The measured microkernel penalty is too large to be ignored. From a practical point of view, the pure microkernel approach of the first generation was a failure.

As a consequence, both Chorus and Mach reintegrated the most critical servers and drivers into the kernel [3, 8]. Chorus's "supervisor actors" and Mach's "kernel-loaded tasks" run in kernel mode, can freely access kernel space, and can freely interact with each other. Gaining performance by saving user-kernel and address-space switches looked reasonable and seemed successful. However, while solving some performance problems, this workaround weakens the microkernel approach. If most drivers and servers are included in the kernel for performance reasons, the benefits—encapsulation, security, and flexibility—largely disappear.

Evaluation of the progressive criterion must be based on upcoming trends and applications. Object-orientation and distribution will cause increased cross-address-space interaction. As a consequence, RPC granularity will become finer; on average, clients and server will spend fewer cycles between successive RPCs.

Figure 3 shows the relative RPC overhead as a function of the average number of cycles spent by client and server between two successive RPCs. The overhead is given with respect to an ideal system where RPC is free. Total overhead means a program takes twice as much time for execution due to RPC as in the ideal system. It seems reasonable to assume that 10% is tolerated in most cases but that 50% is not. The 10% limit allows applications of first-generation microkernels to use one RPC every 100,000 user-level cycles; roughly 10,000 lines of client and server code must be executed before invoking the next

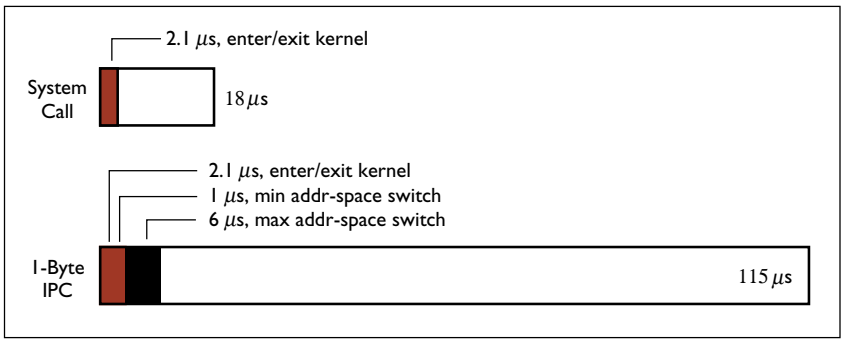

Figure 4. Hardware (black) vs. Mach (black and white) costs when used with a 486-DX50 CPU

RPC. The disappointing conclusion is that first-generation microkernels do not support fine-grained use of RPC. For comparison, Figure 3 also shows overheads for two second-generation microkernels. Under the 10% restriction, the second-generation microkernels permit roughly one RPC per 400 lines of executed user-level code.

Beside this performance-based inflexibility, another shortcoming became apparent over the years. The external-pager concept is in principle not sufficiently flexible. Its most important technical weakness is that main memory is still managed by the microkernel and can be controlled only rudimentarily by the external pager. However, multimedia file servers, real-time applications, frame buffer management, and some nonclassical applications require complete main-memory control.[2]

Conceptually, the weakness is the policy inside the microkernel. A "policy interface" permitting paramaterization and tuning of a built-in policy is convenient as long as that policy is suited for all applications. Its limitations are obvious as soon as a really novel policy is needed or a substantial modification is needed for a predefined policy.

The Second Generation

The deficiency analysis of the early microkernels identified user-kernel-mode switches, address-space switches, and memory penalties as primary sources of disappointing performance. Regarded superficially, this analysis was correct, because it was supported by detailed performance measurements.

Surprisingly, a deeper analysis shows that the three points—user-kernel-mode switches, address-space switches, and memory penalties—are not the real problems; the hardware-inherited costs of mode and address-space switching are only 3%–7% of the measured costs (see Figure 4). A detailed discussion can be found in [16].

The situation was strange. On the one hand, we knew the kernels could run at least 10 times faster; on the other, after optimizing microkernels for years, we no longer saw new significant optimization possibilities. This contradiction suggested the efficiency problem was caused by the basic architecture of these kernels.

Indeed, most early microkernels evolved step by step from monolithic kernels, remaining rich in concepts and large in code size. For example, Mach 3 offers approximately 140 system calls and needs more than 300 Kbytes of code. Reducing a large monolithic kernel may not lead to a real microkernel.

Radical New Designs

A new radical approach, designing a microkernel architecture from scratch, seemed promising and necessary. Exokernel [9] and L4 [16], discussed here, both concentrate on a minimal and clean new architecture and support highly extensible operating systems.

- **Exokernel.** Exokernel, developed at MIT in 1994-95, is a small, hardware-dependent microkernel based on the idea that abstractions are costly and restrict flexibility [9]. The microkernel should multiplex hardware primitives in a secure way. The current exokernel, which is tailored to the Mips architecture and gets excellent performance for kernel primitives, is based on the philosophy that a kernel should provide no abstractions but only a minimal set of primitives (although the Exokernel includes device drivers). Consequently, the Exokernel interface is architecture dependent, dedicated to software-controlled translation lookalike buffers (TLBs). The basic communication primitive is the protected control transfer that crosses address spaces but does not transfer arguments. A lightweight RPC based on this primitive takes 10 μs on a Mips R3000 processor, while a Mach RPC needs 95 μs. Unanswered is the question of whether the right abstractions perform better and lead to better-structured and more efficient applications than Exokernel's primitives do.
- **L4.** Developed at GMD in 1995, L4 is based on the theses that efficiency and flexibility require a minimal set of general microkernel abstractions and that microkernels are processor dependent. In [16], we show that even such compatible processors as the 486 and the Pentium need different microkernel implementations (with the same API)—not only different coding but different algorithms and data structures. Like optimizing code generators, microkernels are inherently not portable, although they improve the portability of a whole system. L4 supplies three abstractions—

[2]Wiring specific pages in memory is not enough. To control second-level cache usage, DMA and management of memory areas with specific hardware-defined semantics, you need complete allocation control. The same control is needed for deallocation to enable application-specific checkpointing and swapping.

address spaces (described in the next section), threads, and IPC—implements only seven system calls, and needs only 12 Kbytes of code. Across-address-space IPC on a 486-DX50 takes 5 µs for an 8-byte argument and 18 µs for 512 bytes. The corresponding Mach numbers are 115 µs (8 bytes) and 172 µs (512 bytes). With 2 x 5 µs, the basic L4-RPC is twice as fast as a conventional Unix system call. It remains unknown whether L4's abstractions, despite being substantially more flexible than the abstractions of the first generation, are flexible and powerful enough for all types of operating systems.

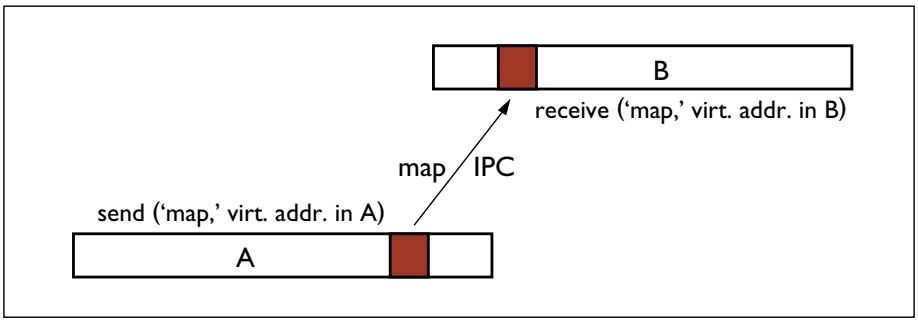

Figure 5. Recursively constructed address spaces

Both approaches seem to overcome the performance problem. Exokernel's and L4's communication is up to 20 times faster than that of first-generation IPC.

Some nonmicrokernel systems try to reduce communication costs by avoiding IPC. As with Chorus and Mach, Solaris and Linux support kernel-loadable modules. The Spin system [2] extends the Synthesis [20] idea and uses a kernel-integrated compiler to generate safe code inside the kernel space. Communicating with servers of this kind requires fewer address-space switches. The reduced IPC costs of second-generation microkernels might make this technique obsolete or even disqualify it, since kernel compilers impose overhead on the kernel. However, the question of which is superior—kernel-compiler technology or a pure microkernel approach—is open as long as there is no sound implementation integrating a kernel-compiler with a second-level microkernel.

More Flexibility

Performance-related constraints seem to be disappearing. The problem of first-generation microkernels was the limitation of the external-pager concept hardwiring a policy inside the kernel. This limitation was largely removed by L4's address-space concept, which provides a pure mechanism interface. Instead of offering a policy, the kernel's role is confined to offering the basic mechanisms required to implement the appropriate policies. These basic mechanisms permit implementation of various protection schemes and even of physical memory management on top of the microkernel.

The idea is to support the recursive construction of address spaces outside the kernel (see Figure 5). An initial address space represents the physical memory and is controlled by the first address-space server. At system start time, all other address spaces are empty. For construction and maintenance of further address spaces on top of the initial space, the microkernel provides three operations: grant, map, and demap.

The owner[3] of an address space can *grant* any of its pages to another space, provided the recipient agrees. The granted page is removed from the granter's address space and is included in the grantee's address space. The important restriction is that instead of physical page frames, the granter can grant only those pages already accessible to itself. The owner of an address space can also *map* any of its pages into another address space, if the recipient agrees. Afterward, the page can be accessed in both address spaces. In contrast to granting, in mapping, the page is not removed from the mapper's address space. As in the granting case, the mapper can map pages to which it already has access.

The owner of an address space can *demap* any of its pages. The demapped page remains accessible in the demapper's address space but is removed from all other address spaces that received the page directly or indirectly from the demapper. Although explicit consent of the address-space owners is not required, the operation is safe, since it is restricted to owned pages. The users of these pages already agreed to accept a potential demapping when they received the pages by mapping or granting. See the sidebar "Frequently Asked Questions on Memory Servers."

Since mapping a page requires consent between mapper and mappee, as does granting, it is implemented by IPC. In Figure 6, the mapper A sends a map message to the mappee B specifying by an appropriate receive operation that it is willing to accept a mapping and determines the virtual address of the page inside its own address space.

The address-space concept leaves memory management and paging outside the microkernel; only the grant, map, and demap operations are retained inside the kernel. Mapping and demapping are required to implement memory managers and pagers on top of the microkernel. Granting is used only in special situations to avoid double bookkeeping and address-space overflow. For a more detailed description, see [16].

In contrast to the external-pager concept, the kernel confines itself to mechanisms. Policies are left

[3]"Owner" means the thread or threads that execute inside the address space.

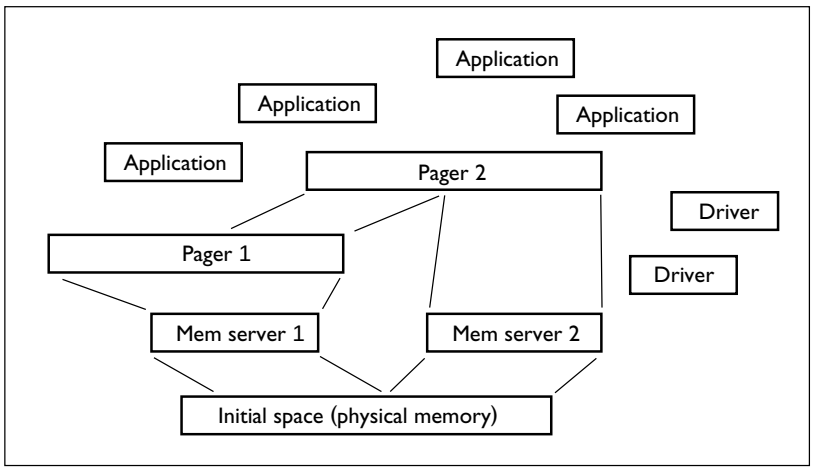

Figure 6. A maps page by IPC to B

completely to user-level servers. To illustrate, we sketch some low-level services that can be implemented by address-space servers based on the mechanisms. A server managing the initial address space is a classic main memory manager, though outside the microkernel. Memory managers can easily be stacked; the initial memory server maps or grants parts of the physical memory to memory server 1 and memory server 2. Now we have two coexisting main-memory managers.

A pager may be integrated with a memory manager or use a memory-managing server. Pagers use the microkernel's grant, map, and demap primitives. The remaining interfaces, pager-client, pager-memory server, and pager-device driver, are completely based on IPC and are defined outside the kernel. Pagers can be used to implement traditional paged virtual memory and file/database mapping, as well as unpaged resident memory for device drivers and real-time or multimedia systems. User-supplied paging strategies [5, 14] are handled at the user level and are in no way restricted by the microkernel. Stacked address spaces, like those in Grasshopper [18], and stacked file systems [13] can be realized in the same fashion.

Multimedia and other real-time applications require allocation of memory resources in a way that allows predictable execution times. For example, user-level memory managers and pagers permit fixed allocation of physical memory for specific data or for locking data in memory for a given time. Multimedia and timesharing resource allocators can coexist if the servers cooperate.

Such memory-based devices as bitmap displays are realized by a memory manager holding the screen memory in its address space.

Improving the hit rates of a secondary cache by means of page allocation or reallocation [12, 21] can be implemented through a pager that applies cache-dependent policies for allocating virtual pages in physical memory.

Remote IPC is implemented by communication servers translating local messages to external communication protocols and vice versa. The communication hardware is accessed by device drivers. If special sharing of communication buffers and user address spaces is required, the communication server also acts as a special pager for the client. In contrast to the first generation, there is no packet filter inside the microkernel.

Unix system calls are implemented by IPC. The Unix server can act as a pager for its clients and can share memory for communication with its clients. The Unix server itself is pageable or resident.

Frequently Asked Questions on Memory Servers

Is a kernel without main-memory management still a microkernel or is it a submicrokernel? It is a kernel because it is mandatory to all other levels. There is no alternative kernel, although alternative memory servers may coexist. It is a microkernel because it is a direct result of applying the microkernel paradigm of making the kernel minimal. The insight that microkernels have to be much smaller than in the first generation does not justify a new "submicro" term. The underlying paradigm is the same.

Since the kernel is not usable without a memory server, why not include it in the microkernel? Two reasons:

- Operating systems can offer coexisting alternative memory servers. Examples are timesharing (paging), real-time, multimedia, and file caching techniques.
- Memory servers may be as machine (not processor) dependent as device drivers.

Specialized servers are required for controlling second-level caches and device-specific memories. If a machine has fast and slow (uncached) main-memory regions, a corresponding memory server might ensure that only fast memory is used for paging and slow memory is reserved for disk caching.

Conclusion

Although academic experiments in porting applications and operating system personalities to second-generation microkernels look promising, we have covered only the known problems of microkernels. There is no real-life practical experience with second-

generation microkernels to draw on. Although we are optimistic, second-generation microkernel design is still research, and new problems can arise.

Most older microkernels evolved from monolithic kernels and did not achieve sufficient flexibility and performance. Although theoretically advantageous, the microkernel approach was never widely accepted in practice. However, a new generation of microkernel architectures shows promising results, and performance and flexibility have improved by an order of magnitude. Still debatable is whether Exokernel's nonabstractions, Spin's kernel compiler, L4's address-space concept, or a synthesis of these approaches is the best way forward. In each case, we expect efficient and flexible operating systems based on second-generation microkernels to be developed.

The microkernel approach was the first software architecture to be examined in detail from the performance point of view. We learned that applying the performance criterion to such a complex system is not trivial. Naive, uninterpreted measurements are sometimes misleading. Although early microkernel measurements suggested reducing the frequency of user-to-user IPC, the real problem was the structure and implementation of the kernels. To avoid such misinterpretations in the future, we should always try to understand why we get the measured results. As in physics, computer science should regard measurements as experiments used to validate or repudiate a theory.

Although steady evolution is a powerful methodology, sometimes a radically new approach is needed. Most problems of the first-generation microkernels were caused by their step-by-step development. The microkernels designed from scratch gave completely different results that could not have been extrapolated from previous experience. ◻

References

1. Bernabeu-Auban, J.M., Hutto, P.W., and Khalidi, Y.A. The architecture of the Ra kernel. Tech. Rep. GIT-ICS-87/35, Georgia Institute of Technology, Atlanta, 1988.
2. Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. Extensibility, safety, and performance in the Spin operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)* (Copper Mountain Resort, Colo, Dec. 1995). ACM Press, 1995, pp. 267–284.
3. Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D., and Rozier, M. A new look at microkernel-based Unix operating systems. Tech. Rep. CS/TR-91-7, Chorus systèmes, Paris, France, 1991.
4. Campbell, R., Islam, N., Madany, P., and Raila, D. Designing and implementing Choices: An object-oriented system in C++. *Commun. ACM 36*, 9 (Sept. 1993), 117–126.
5. Cao, P., Felten, E.W., and Li, K. Implementation and performance of application-controlled file caching. In *Proceedings of the 1st Usenix Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, Calif., Nov. 1994). ACM Press, New York, 1994, pp. 165–178.
6. Chen, J.B. and Bershad, B.N. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)* (Asheville, N.C., Dec. 1993). ACM Press, 1993, pp. 120–133.
7. Cheriton, D.R., Whitehead, G.R., and Sznyter, E.W. Binary emulation of Unix using the V kernel. In *Proceedings of the Usenix Summer Conference* (Anaheim, Calif., June 1990), pp. 73–86.
8. Condict, M., Bolinger D., McManus, E., Mitchell, D., and Lewontin, S. Microkernel modularity with integrated kernel performance. Tech. Rep., OSF Research Institute, Cambridge, Mass, 1994.
9. Engler, D., Kaashoek, M.F., and O'Toole, J. Exokernel, an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)* (Copper Mountain Resort, Colo., Dec. 1995) ACM Press, 1995, pp. 251–266.
10. Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an application program. In *Proceedings of the Usenix Summer Conference* (Anaheim, Calif., June 1990). Usenix Association, 1990, pp. 87–96.
11. Guillemont, M. The Chorus distributed operating system: Design and implementation. In *Proceedings of the ACM International Symposium on Local Computer Networks* (Firenze, Italy, Apr. 1982) ACM Press, 1982, pp. 207–223.
12. Kessler, R., and Hill, M.D. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 11–22.
13. Khalidi, Y.A., and Nelson, M.N. Extensible file systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)* (Asheville, N.C., Dec. 1993). ACM Press, New York, 1993, pp. 1–14.
14. Lee, C.H., Chen, M.C., and Chang, R.C. HiPEC: High performance external virtual memory caching. In *Proceedings of the 1st Usenix Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, Calif., Nov. 1994). ACM Press, New York, 1994, pp. 153–164.
15. Liedtke, J. A persistent system in real use—experiences of the first 13 years. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems (IWOOOS)* (Asheville, N.C., Dec. 1993) IEEE Computer Society Press, Washington, 1993, pp. 2–11.
16. Liedtke, J. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)* (Copper Mountain Resort, Colo., Dec. 1995). ACM Press, New York, 1995, pp. 237-250.
17. Liedtke, J., Bartling, U., Beyer, U., Heinrichs, D., Ruland, R., and Szalay, G. Two years of experience with a microkernel based operating system. *Oper. Syst. Rev. 25*, 2 (Apr. 1991), 51–62.
18. Lindstroem, A., Rosenberg, J., and Dearle, A. The grand unified theory of address spaces. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)* (Orcas Island, Wash., May 1995).
19. Mullender, A.J. The Amoeba distributed operating system: Selected papers 1984–1987. Tech. Rep. Tract. 41, CWI, Amsterdam, 1987.
20. Pu, C., Massalin, H., and Ioannidis, J. The Synthesis kernel. *Comput. Syst. 1*, 1 (Jan. 1988), 11–32.
21. Romer, T.H., Lee, D.L., Bershad, B.N., and Chen, B. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proceedings of the 1st Usenix Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, Calif., Nov. 1994). ACM Press, New York, 1994, pp. 255–266.
22. Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating System Principles (SOSP)* (Austin, Tex., Nov. 1987). ACM Press, New York, 1987.

JOCHEN LIEDTKE is a senior researcher in the German National Research Center for Information Technology (GMD) and at IBM's T.J. Watson Research Laboratory. He can be reached at jochen.liedtke@gmd.de