CACM 39(9), to appear Sep 1996
Do not cite! Do not copy!

Towards Real μ-Kernels

Jochen Liedtke *

The μ-Kernel story is full of good ideas and blind alleys. The story began with high initial enthusiasm about the promised dramatic increase in flexibility, safety and modularity. Over the years, this changed to disappointment because the first-generation μ-kernels turned out to be inefficient and inflexible.

At present, we are observing radically new approaches to the μ-kernel idea which try to avoid the past mistakes and to overcome the old flexibility and performance constraints. The second-generation μ-kernels might be a basis for all types of operating systems: timesharing, multi-media, soft and hard realtime.

The μ-Kernel Vision

Traditionally, the word 'kernel' is used to denote the part of the operating system that is mandatory and common to all other software. The kernel can use all features of a processor, e.g. program the MMU, while 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 was packed into one kernel: scheduling, file system, networking, device drivers, memory management, paging and more.

In contrast to this, the basic idea of the *μ-kernel* approach is to minimize the kernel and to implement whatever possible as *servers* outside of the kernel. Ideally, the kernel implements only address spaces, inter-process 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 against each other.

*GMD — German National Research Center for Information Technology, Schlo Birlinghoven, 53754 Sankt Augustin, Germany, e-mail: jochen.liedtke@gmd.de

When the idea was introduced, the software-technological advantages seemed manifold and obvious:

- Different application-program interfaces (APIs), different file systems, perhaps even different basic OS strategies can coexist in one system. They are implemented as competing or co-operating 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 that all other processes are not affected. Furthermore, modifications do not require building a new kernel; they can be made and tested on line.
- All servers can use the mechanisms provided by the μ-kernel, e.g. multi-threading and IPC.
- Server malfunction is as isolated as normal application malfunction.
- All this also holds for device drivers.
- A clean μ-kernel interface enforces a more modular system structure.
- A smaller kernel can be more easily maintained and should be less error-prone.
- Interdependencies between the various parts of the system can be restricted and reduced. (In particular, the trusted computing base[1] comprises only the hardware, the μ-kernel, a disk driver and perhaps a basic file system. Other drivers, file and network systems are no longer required to be absolutely trustworthy.)

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

[1] Trusted computing base (TCB) is the set of all components whose correct functionality is a necessary precondition for implementing security. Among others, hardware and kernel belong to the TCB.

The First Generation

The μ-kernel idea met with efforts to build post-Unix operating systems. New hardware (multi-processors, massively parallel systems), new application requirements (security, multi-media, real-time, distributed computing) and new programming methodologies (object orientation, multi-threading, persistence) required novel operating-system concepts.

The corresponding objects and mechanisms – threads, address spaces, remote procedure call (RPC), 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 to Unix or another conventional OS was a *conditio sine qua non*, hence implementing Unix on top of the new systems was a natural consequence. Therefore the μ-kernel idea became widely accepted by OS designers for two completely different reasons: the first was the expected general flexibility and power, the second was the fact that μ-kernels offered a technique for preserving Unix compatibility while nevertheless permitting novel operating systems to be build.

Many academic projects trod this path, e.g. Amoeba [19], Choices [4], Ra [1], V [7], some even moved to commercial use, in particular Chorus [11], L3 [15] and Mach [10], the latter becoming the industrial μ-kernel flagship.

Innovations

Mach's invention of the *external pager* [22] was the first conceptual breakthrough towards real μ-kernels. Its basic idea is that the kernel manages physical and virtual memory but forwards page faults to specific user-level tasks. These pagers implement the 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).

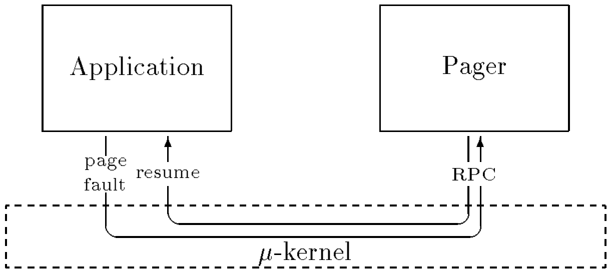

Figure 1: *Page Fault Processing.*

The 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 of the kernel.

The second conceptional step towards μ-kernels was the idea of handling hardware interrupts as IPC messages [17] (and including IO-ports in address spaces). The kernel captures the interrupt but does not handle it. Instead, it generates a message for the user-level process currently associated with this interrupt. Thus interrupt handling and device IO are done completely outside of the kernel:

```
driver thread:
   do
      wait for (msg, sender) ;
      if sender = my hardware interrupt
         then  read/write io 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 independently of the kernel. Furthermore, device drivers profit from using the μ-kernel mechanisms: multi-threading, IPC and address spaces.

Disappointments

An appealing concept is only one side of the coin, the other side is the practical usefulness. Are the costs of flexibility low enough and the concepts flexible enough, even for real-world problems like multi-media, real-time and embedded systems?

With respect to efficiency, the most critical μ-kernel mechanism is the communication facility. Each invocation of an OS or application service requires a remote procedure call (RPC) consisting in general of two IPCs – the 'call' and the 'return' message. Therefore, we μ-kernel architects spent much effort in optimizing step-by-step the IPC mechanisms. Steady progress led to an up to twofold improvement, but by mid 1991, the steps became less and less effective. Mach 3 stabilized at about 115 μs per IPC on a 486-DX50 which is comparable to most other μ-kernels. For illustration: a conventional Unix system call, roughly 20 μs on this hardware, has about ten times less overhead than the Mach RPC. It seemed

Frequently asked questions concerning external pagers:

1. *Is there a pager required inside the μ-kernel?*
   No.
2. *How do user-level pagers affect system security?*
   A pager can corrupt the data which it maintains. So you rely on the correct functionality of the pager you use, whether it is kernel-integrated or user-level. However, different non-interfering user-level pagers can be used to increase security: for example by holding sensitive data in a very trustworthy and stable (standard) pager and less critical data in potentially less trustworthy pagers. Note that similar semantic dependencies exist on all levels and are not OS-specific: one relies just as much on the correct functionality of a compiler or database system.
3. *How expensive are user-level pagers?*
   In principle, the overhead compared to an integrated pager is only one IPC and should be negligible. In practice however, most first generation μ-kernels use a very complicated protocol with up to 8 additional IPCs and have implemented IPC much too inefficiently. Overheads of 1000 μs (Alpha, 133 MHz) can occur. A μ-kernel of the second generation solves this performance problem by using only one additional IPC and making it fast (less than 10 μs).

Frequently asked questions concerning user-level device drivers:

1. *Can you really use a user-level disk driver for demand paging?*
   Yes.
2. *Can a user-level driver corrupt the system?*
   On the one hand, drivers are encapsulated in address spaces so that they can only access the memory and the IO-ports that are 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. So the risk depends on what hardware you make accessible to the driver.
3. *Do user-level drivers increase security?*
   Yes, because of encapsulation. For example, a mouse driver can do no more harm than an editor can.
4. *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 μ-kernels, user-level drivers perform as well as integrated ones in these systems. Less than 10 μs are required per interrupt or driver RPC.

that 100 μs was the inherent cost of IPC and that the concept had to be evaluated on this basis.

Since absolute time has no meaning on its own, two more practical criteria are used for evaluation: (a) existing applications must not be degraded by the μ-kernel; (b) μ-kernels must efficiently support new types of applications which cannot be implemented with good performance on conventional monolithic kernels. The conservative criterion (a) is a necessary precondition for practical acceptance. The progresssive criterion (b) must be satisfied for a real advance in technology.

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

Some applications performed as well under a μ-kernel as under a monolithic kernel, a few even slightly better. Unfortunately, other applications were substantially degraded. Chen *et al.* [6] compared applications under Ultrix and Mach on a DS 5000/200 and found peak degradations of up to 66% (see Figure 2). Condict *et al.* compared an 8-user AIM III

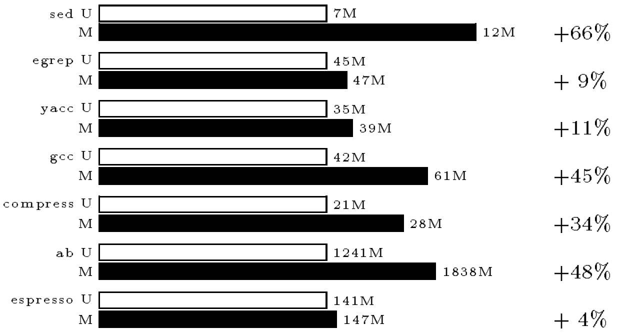

Figure 2: *Non-Idle Cycles under Ultrix and Mach.*

benchmark on a 486-DX50 under a monolithic OSF/1 versus a Mach-based OSF/1 and measured an average 1.5-fold 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 which are direct consequences thereof; 10% comes from multiprocessor provisions which could be dropped on this uniprocessor, and the remaining 17% is due to unspecified miscellaneous reasons. Chen found that the performance differences are partially caused by a substantially higher cache-miss rate for the Mach-based system. This result could point to a principal weakness of server architectures on top of μ-kernels. However, the increased cache misses are caused by the Mach kernel, invoked for IPC, and not by the higher modularity of the client/server architecture.

The measured μ-kernel penalty is too large to be ignored. From a practical point of view, the pure μ-kernel approach of the first generation failed.

As a consequence, both Chorus and Mach decided to re-integrate the most critical servers and drivers in the kernel [3; 8]. Chorus' "supervisor actors" as well as 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 to be successful. However, while solving some performance problems, this workaround weakens the μ-kernel approach. If most drivers and servers must be included in the kernel for performance reasons, the mentioned benefits – encapsulation, security, flexibility – largely disappear.

Evaluating the progressive criterion must be based on upcoming trends and applications. Due to object-orientation and distribution, cross-address-space interaction will increase. As a consequence, RPC granularity will become finer: on average, clients and server will spend less cycles between successive RPCs.

Figure 3 shows the relative RPC overhead, as a function of the average number of cycles spent by

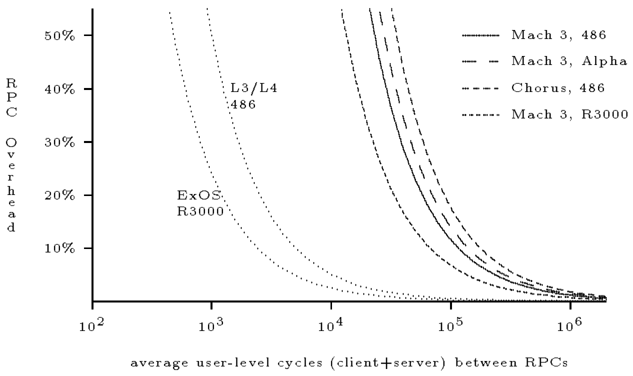

Figure 3: *Relative RPC Overhead.*

client and server between two successive RPCs. The overhead is given with respect to an ideal system where RPC is for free. 100% overhead means that a program takes twice as much time due to RPC as in the ideal system. It seems reasonable to assume that 10% will be tolerated in most cases but not 50%. The 10% limit allows applications of first-generation μ-kernels 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 RPC. The disappointing conclusion is that first-generation μ-kernels do not support fine-grained use of RPC. For comparison, the figure also shows overheads for two second-generation μ-kernels. Under the 10%-restriction, they permit roughly one RPC per 400 lines of executed user-level code.

Besides this performance-based inflexibility, a further shortcoming became apparent over the years. The external-pager concept is *in principle* not sufficiently flexible. Its most important technical weak point is that the main memory is still managed by the μ-kernel and can only rudimentarily be controlled by the external pager. However, multi-media file servers, real-time applications, frame buffers management and some non-classical applications require complete main-memory control.[2]

Conceptually, the weak point is the policy remaining inside the μ-kernel. A "policy interface" which permits paramaterization and tuning of a built-in policy is convenient as long as that policy is well-suited for all applications. Its limitations become obvious as soon as a really novel policy is needed or a substantial modification to a predefined policy.

The Second Generation

The deficiency analysis of the early μ-kernels identified user-kernel-mode switches, address-space switches and memory penalties as being primary reasons for disappointing performance. Regarded superficially, this analysis was obviously right, since it was substantiated by measurements.

Surprisingly, a deeper analysis shows that the three points mentioned cannot be the real reasons: the hardware-inherited costs of mode and address-space switching are only 3% to 7% of the measured costs (see Figure 4). A detailed discussion can be found in [16].

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

Indeed, most early μ-kernels have evolved step-by-step from monolithic kernels, remaining rich in concepts and quite large in code size. Mach 3 offers approximately 140 system calls and needs more than 300 Kbytes of code.

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

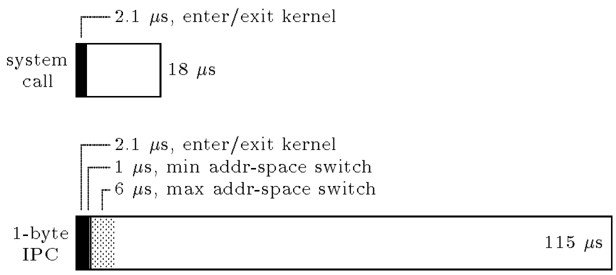

Figure 4: *Hardware (black) Versus Mach (black+white) Costs, 486-DX50.*

Perhaps reducing a large monolithic kernel does not lead to a real μ-kernel?

New Radical Designs

A new radical (from Latin *radicalis*: "from the root") approach, designing a μ-kernel architecture from scratch, seemed promising and necessary. We discuss Exokernel [9] and L4 [16] which both concentrate on a minimal and clean new architecture and support highly extensible operating systems.

Exokernel developed at MIT is a small and hardware-dependent μ-kernel. It is based on the thesis that

- Abstractions are costly and restrict flexibility. The μ-kernel should only multiplex hardware primitives in a secure way.

The current exokernel is tailored to the Mips architecture and gets excellent performance for kernel primitives. It is based on the philosophy that a kernel should *not* provide abstractions but only a minimal set of primitives (although the Exokernel includes device drivers). Consequently, the Exokernel interface is architecture dependent, in particular dedicated to software-controlled TLBs. The basic communication primitive is the protected control transfer which crosses address spaces but does not transfer arguments. A lightweight remote procedure call based on this primitives takes 10 μs on an R3000 while Mach RPC needs 95 μs.

The open question: might the right abstractions perform better and lead to better structured and more efficient applications than Exokernel's primitives?

L4 has been developed at GMD. It is based on the theses that

- Efficiency and flexibility require a minimal set of general μ-kernel abstractions.
- μ-kernels are processor dependent.

In [16] we show that even compatible processors like 486 and Pentium need different μ-kernel implementations (with the same API), not only different coding but different algorithms and data structures. Like optimizing code generators, μ-kernels are inherently not portable, although they improve the portability of the whole system. L4 supplies three abstractions: address spaces (described in the next section), threads and IPC. It implements only 10 system calls and needs 20 Kbytes of code. Cross-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 2×5 μs, the basic L4-RPC is twice as fast as a conventional Unix system call.

The open question: although L4's abstractions are substantially more flexible than the abstractions of the first generation, we still do not know whether they are flexible and powerful enough for all types of operating systems.

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

Some non-μ-kernel systems are trying 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 less address-space switches. The strongly-reduced IPC costs of second-generation μ-kernels might make this technique obsolete or even disqualify it, since kernel-compilers impose a certain overhead on the kernel. However, the question remains open as long as there is no sound implementation integrating a kernel-compiler with a second-level μ-kernel.

Increasing Flexibility

Performance-related constraints seem to be disappearing. A further problem of first-generation μ-kernels was the earlier mentioned principal limitation of the external-pager concept wiring a policy inside the kernel. This has been largely removed by L4's address-space concept which provides a pure "mechanism interface". Instead of offering a policy, the kernel confines itself to offering the basic mechanisms required to implement the appropriate policies. This permits the implementation of various protection schemes and even physical memory management on top of the μ-kernel.

The basic idea is to support the recursive construction of address spaces outside of the kernel (Figure 6). By magic, there is an initial address space which essentially represents the physical memory and is controlled by the first address-space server. At system start time, all other address spaces are empty. For constructing and maintaining further address spaces on top of the initial space, the μ-kernel 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 included into the grantee's address space. The important restriction is that instead of physical page frames, the granter can only grant pages which are already accessible to itself. The owner of an address space can *map* any of its pages into another address space, provided the recipient agrees. Afterwards, the page can be accessed in both address spaces. In contrast to granting, the page is not removed from the mapper's address space. As in the granting case, the mapper can only map pages which it already has access to. 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 which have received the page directly or indirectly from the demapper. Although explicit consent of the affected 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.

Since mapping (or granting) a page requires consent between mapper and mappee, it is realized by IPC. In Figure 5 the mapper A sends a 'map' message to the mappee B which specifies 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.

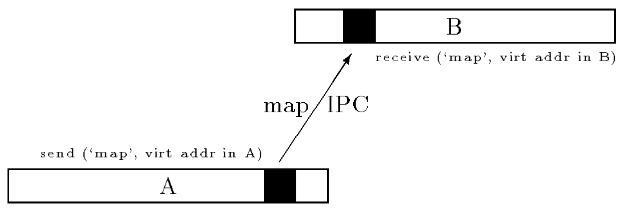

Figure 5: *A maps page by IPC to B.*

The described address-space concept leaves memory management and paging outside of the μ-kernel; only the grant, map and demap operations are retained inside the kernel. Mapping and demapping are

[3] 'Owner' means the thread or the threads which execute inside the address space.

Frequently asked questions concerning memory servers:

1. *Is a kernel without main memory management this still a μ-kernel, or is it a sub-μ-kernel?* 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 μ-kernel since it is a direct result of applying the μ-kernel paradigm of making the kernel minimal. The insight that μ-kernels have to be much smaller than in the first generation does not justify a new sub-μ term. The underlying paradigm has not changed.
2. *Since the kernel is not usable without a memory server, why not include it in the μ-kernel?* First: Systems can offer alternative coexisting memory servers. Examples are timesharing (paging), real-time, multi-media and file caching. Second: 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 for disk caching.

required to implement memory managers and pagers on top of the μ-kernel. Granting is only used in special situations to avoid double bookkeeping and address-space overflow. (For a more detailed description see [16].)

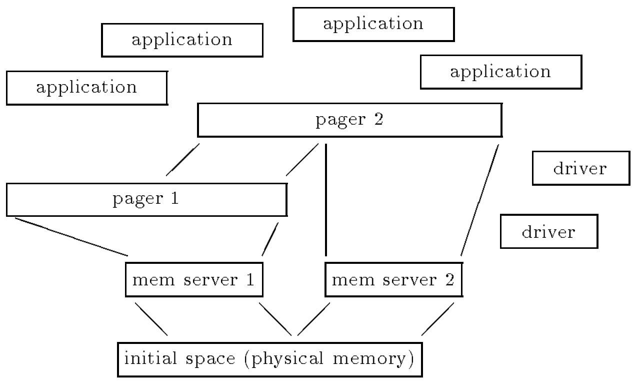

Figure 6: *Recursively Constructed Address Spaces.*

In contrast to the external-pager concept, the kernel confines itself to mechanisms. Policies are completely left to user-level servers. For illustration, we sketch some low-level services which can be implemented by *address-space servers* based on the mentioned mechanisms:

A server managing the initial address space is a classical main memory manager, though outside of the μ-kernel. Memory managers can easily be stacked: the initial memory server maps or grants parts of the physical memory to memory server 1 and 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 μ-kernel'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, real-time or multimedia systems. User-supplied paging strategies [14; 5] are handled at the user level and are in no way restricted by the μ-kernel. Stacked address spaces like in Grasshopper [18] and stacked file systems [13] can be realized in the same fashion.

Multi-media and other real-time applications require memory resources to be allocated in a way that allows predictable execution times. The above mentioned user-level memory managers and pagers permit e.g. fixed allocation of physical memory for specific data or locking data in memory for a given time. Multi-media and timesharing resource allocators can coexist if the servers cooperate.

Memory-based devices like 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 by means of a pager which applies cache-dependent policies for allocating virtual pages in physical memory.

Remote IPC is implemented by communication servers which translate 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 will also act as a special pager for the client. In contrast to the first generation, there is no packet filter inside the μ-kernel.

Unix system calls are implemented by IPC. The

Unix server can act as a pager for its clients and can also share memory for communication with its clients. The Unix server itself can be pageable or resident.

In Practice?

Although academic experiments of porting applications and OS personalities to second-generation μ-kernels look promising, we should recall that we dealt only with the *known* problems of μ-kernels. There are still no real-life practical experiences with second-generation μ-kernels. Although we are optimistic, it is still research and new hard problems might arise.

Summary

Most older μ-kernels evolved from monolithic kernels and did not achieve sufficient flexibility and performance. Although theoretically an advantageous concept, the μ-kernel approach was therefore not widely accepted in practice. However, a new generation of μ-kernel architectures shows promising results: performance and flexibility have improved an order of magnitude. Whether Exokernel's non-abstractions, Spin's kernel compiler, L4's address-space concept or a synthesis of them is the best way forward, is still open to debate. In each case, we expect to see efficient and flexible operating systems based on second-generation μ-kernels.

Lessons

The μ-kernel 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 μ-kernel measurements suggested reducing the frequency of user-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 is done in physics, computer science should regard measurements as experiments that are used to validate or falsify a theory.*

Although steady evolution is a powerful methodology, sometimes a radically new approach is needed. Most problems of the first-generation μ-kernels were caused by their step-by-step development. The μ-kernels designed from scratch gave completely different results which could not be extrapolated at all from previous experiences.

References

[1] Bernabeu-Auban, J. M., Hutto, P. W., and Khalidi, Y. A.. The architecture of the Ra kernel. Technical Report GIT-ICS-87/35, Georgia Institute of Technology, Atlanta, GA, January 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 15th ACM Symposium on Operating System Principles (SOSP), pages 267–284, Copper Mountain Resort, CO, December 1995.

[3] Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D., and Rozier, M.. A new look at microkernel-based Unix operating systems. Technical Report CS/TR-91-7, Chorus systèmes, Paris, 1991.

[4] Campbell, R., Islam, N., Madany, P., and Raila, D. Designing and implementing Choices: an object-oriented system in C++. Communications of the ACM, 36(9):117–126, September 1993.

[5] Cao, P., Felten, E. W., and Li, K.. Implementation and performance of application-controlled file caching. In 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 165–178, Monterey, CA, November 1994.

[6] Chen, J. B. and Bershad, B. N. The impact of operating system structure on memory system performance. In 14th ACM Symposium on Operating System Principles (SOSP), pages 120–133, Asheville, NC, December 1993.

[7] Cheriton, D. R., Whitehead, G. R., and Sznyter, E. W. Binary emulation of Unix using the V kernel. In Usenix Summer Conference, pages 73–86, Anaheim, CA, June 1990.

[8] Condict, M., Bolinger D., McManus, E., Mitchell, D., and Lewontin, S. Microkernel modularity with integrated kernel performance. Technical report, OSF Research Institute, Cambridge, MA, April 1994.

[9] Engler, D., Kaashoek, M. F., and O'Toole, J. Exokernel, an operating system architecture for application-level resource management. In 15th ACM Symposium on Operating System Principles (SOSP), pages 251–266, Copper Mountain Resort, CO, December 1995.

[10] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an application program. In Usenix Summer Conference, pages 87–96, Anaheim, CA, June 1990.

[11] Guillemont, M. The chorus distributed operating system: Design and implementation. In ACM International Symposium on Local Computer Networks, pages 207–223, Firenze, April 1982.

[12] Kessler, R. and Hill, M. D. Page placement algorithms for large real-indexed caches. ACM Transactions on Computer Systems, 10(4):11–22, November 1992.

[13] Khalidi, Y. A. and Nelson, M. N. Extensible file systems in Spring. In 14th ACM Symposium on Operating System Principles (SOSP), pages 1–14, Asheville, NC, December 1993.

[14] Lee, C. H., Chen, M. C., and Chang, R. C. HiPEC: high performance external virtual memory caching. In 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 153–164, Monterey, CA, November 1994.

[15] Liedtke, J. A persistent system in real use – experiences of the first 13 years –. In 3rd International Workshop on Object Orientation in Operating Systems (IWOOOS), pages 2–11, Asheville, NC, December 1993.

[16] Liedtke, J. On μ-kernel construction. In 15th ACM Symposium on Operating System Principles (SOSP), pages 237–250, Copper Mountain Resort, CO, December 1995.

[17] Liedtke, J., Bartling, U., Beyer, U., Heinrichs, D., Ruland, R., and Szalay, G. Two years of experience with a μ-kernel based OS. Operating Systems Review, 25(2):51–62, April 1991.

[18] Lindström, A., Rosenberg, J., and Dearle, A. The grand unified theory of address spaces. In 5th Workshop on Hot Topics in Operating Systems (HotOS-V), pages xx–xx, Orcas Island, WA, May 1995.

[19] Mullender, A. J. The amoeba distributed operating system: Selected papers 1984–1987. Technical Report Tract. No. 41, CWI, Amsterdam, 1987.

[20] Pu, C., Massalin, H., and Ioannidis, J. The Synthesis kernel. Computing Systems, 1(1):11–32, January 1988.

[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 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 255–266, Monterey, CA, November 1994.

[22] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In 11th ACM Symposium on Operating System Principles (SOSP), Austin, TE, November 1987.