μ-Kernels Must And Can Be Small

Jochen Liedtke
IBM T. J. Watson Research Center *
GMD — German National Research Center for Information Technology †
jochen@watson.ibm.com

Abstract

For a general acceptance, μ-kernels must be fast and not burden applications. For fulfilling these conditions, cache architectures require μ-kernels to be small. The L4 μ-kernel shows that smallness can be achieved.

1. μ-kernels must be small

This is not obvious. Most first-generation μ-kernels were large; typically they need 300 Kbyte of code and 140 system calls. Some of their architects argued that 'μ' in this context means 'lower level' and not 'small size'. Demanding smallness radically differs from this approach. It could (and in fact it does) change μ-kernel technology dramatically.

Why should μ-kernels be as small as possible? (We avoid the term "minimal" because of its mathematical implications.) The reasons are *performance, flexibility* and perhaps *correctness*.

1.1. "A non-small μ-kernel is not fast."

The most relevant performance costs of a μ-kernel result from its cache consumption. If a frequently invoked kernel operation accesses a substantial part of the primary cache ("floods the cache"), the user is punished twice. First, the kernel operation itself is degraded by cache misses, since it must displace user code and data. Second, the user program has to pay for additional cache misses, since it must re-establish its cache working set. From Chen's [1993] measurements for a Mach benchmark on a DS 5000/200, we can calculate that on average 20% of the system cache misses are caused by user-kernel competition. We expect a substantially higher number for more up-to-date hardware and faster (but non-small) kernels.[1]

On modern processors, the mentioned cache misses might consume up to 5 (five!) times as many cycles as the kernel code required for execution in the ideal case.

What happens if the kernel working set is reduced, say from 75% to 15% of the cache size?

- The small kernel needs only 1/5 of the instructions. Since μ-kernel operations usually execute very few loops, we can expect a corresponding 4 to 5 times speed improvement.
- There is a good chance that competition between user working-set and kernel working-set is substantially reduced. Ideally, frequently invoking a μ-kernel operation does neither cost kernel-level nor subsequent user-level cache misses.

Cache eating

A somehow strange effect occurs if the μ-kernel working set is substantially larger than the cache: enlarging the cache does not really improve the μ-kernel's performance; in the worst case, it might even degrade it since the cache-reestablishing costs increase.

For a rough understanding of these effects, we look at a rather naive model: the cache is highly associative and the application always fills it completely between successive μ-kernel operations. Assume that filling the complete cache costs c cycles and that the μ-kernel operation requires a working set twice as large as the cache. Then $2c$ cycles are required for loading μ-kernel

*30 Saw Mill River Road, Hawthorne, NY 10532, USA
†GMD SET–RS, 53754 Sankt Augustin, Germany

[1] The DS 5000/200 had a 64-K direct-mapped cache with 4-byte lines. In spite of the large primary cache, cache misses degraded on average each system instruction by 0.86 cycles. The corresponding average penalty per user instruction was only 0.15 cycles. Therefore, we can conclude that kernel cache working sets were relatively large and user cache working sets relatively small. This will change dramatically on a modern processor with typically a 2- or 4-way set-associative primary cache of only 16 K. Furthermore, the larger cache lines (typically 32 bytes) will increase competition.

code and data; re-establishing the application's working set needs c cycles additionally. Now we double the cache size. The μ-kernel cache-fill costs remain $2c$ cycles. However, re-establishing the application's working set may now cost up to $2c$ cycles. In total, doubling the cache size can increase the μ-kernel-operation net cost from $3c$ to $4c$ cycles. If the larger cache improves the pure application's speed, the μ-kernel-operation cost *relative to the application* increase even more than 25%.

If a system uses a fine-grained client-server architecture and makes heavy use of micro-kernel operations, the mentioned effects might effectively neutralize the cache enlargement for the application.

Server interaction

A popular counterargument against keeping kernels small is: The server's cache consumption has to be taken into consideration as well. What does it matter whether the μ-kernel or the server floods the cache? Why not integrate the server in the kernel? The user will pay the same.

This is a red herring. The μ-kernel idea is to separate function (the servers) and basic security structure (the kernel). Any user is willing to pay for the function (e.g. copying a file) but not for things like address spaces or IPC which have no direct net effect for the application. The required flexibility and extensibility by adding and modifying servers is widely accepted, *provided* that the structural costs, i.e. the costs of the μ-kernel, are negligible. Ideally an application linked together with a server should behave like a client/server pair of tasks. If a specific system runs too many servers and too many clients simultaneously, this is the standard problem of a too small machine. If all systems are burdened by too many servers (which even do not do the right things for you), even if you run only a small application, you use a monolithic kernel.

A second counterargument is: μ-kernels need more copying, in particular between drivers and applications. This is not true if the address spaces are constructed properly.

Second-level caches

Modern high-performance processors use three-level cache systems. A small and very fast primary on-chip cache (typically 16 K), a fast second-level near-chip cache (typically 128 K) and a large third-level off-chip cache. Because of its limited size, the above-mentioned primary-cache-related arguments apply as well to the second-level cache. However, we have not only to take into consideration the working set of *one* frequently-used μ-kernel operation but the combined working set of *all* operations, except the extraordinarily infrequent ones.

Code and data

Theoretically, a large μ-kernel could have a small instruction-cache working set. However, in practice, this never happens. Our experience taught us that the only chance to get small code working sets is to construct a small kernel.

However, a μ-kernel's data-cache working set much more depends on the structure than on the total size of the kernel data. It is essential to minimize global kernel data, e.g. to avoid system-wide hash tables. Data per task, per thread or per page are less critical, since they burden the cache only if the corresponding objects are manipulated.

1.2. "A non-small μ-kernel is inflexible."

All that is wired in the kernel cannot be modified by higher levels. Since μ-kernels are in a way the most general software (used by any application and potentially *even by any OS*), generality, flexibility and adaptibility is vital for them. We know two strategies for solving the problem:

1. *Many alternate policies in the μ-kernel!*
   The method does not work, because (a) competing policies contradict each other, (b) the number of useful policies is too huge, (c) the set of policies required for current and future applications and OSs is unknown and (d) policy integration enlarges the μ-kernel and thus costs performance.

2. *Only basic mechanisms in the μ-kernel, no policies!*
   If the mechanisms are general and powerful enough, they should permit to implement any reasonable policy. For example, Exokernel [Engler et al. 1995] and L4 [Liedtke 1995] presently explore this strategy.

Making the μ-kernel extensible like in the Spin OS [Bershad et al. 1995] belongs to the second category. There is no real *conceptional* difference between extending the kernel code by a user-written handler and extending the system at user-level by a new server. In both cases, the new software runs on top of an abstract machine, the "μ-kernel". Whether the new code

runs in kernel mode or in user mode is a technical detail. It might have consequences in performance (although the Spin results are discouraging) but not in functionality.

For the second strategy, we must look for basic mechanisms which are *general* and *efficient*. Although it cannot be proven formally, most engineers strongly believe that such basic mechanisms can be found only if (a) the underlying concepts are simple and orthogonal, (b) operate at the lowest possible level and (c) are sufficiently abstract (independent of concrete hardware).

1.3. "Even a small μ-kernel is incorrect."

Undoubtedly, correctness is a reason to keep software small. In the μ-kernel context, however, the argument is probably not as strong as most people believe:

1. The μ-kernel has to tame the hardware's parallism (due to external interrupts even on a uniprocessor) and the software's concurrency. These inherently hard problems are relatively independent of the kernel size.

2. Small μ-kernels require better integration of hardware architecture and kernel architecture. Correspondingly, they enable and require more non-simple optimizations. Both integration and optimization are difficult and hence improve the probability of including bugs.

From our experience, the chance that a small μ-kernel becomes "sufficiently correct" over time is greater than in the case of a large kernel. However, the correlation between size and errors is sublinear.

2. μ-kernels can be small

2.1. Abstractions: 3

The L4 μ-kernel [Liedtke 1996] is based on two basic concepts, *threads* and *address spaces*:

- A Thread is an independent flow of control inside an address space. Threads are identified by unique identfiers and communicate via IPC. The μ-kernel offers *preemption RPC* to implement user-level schedulers, user-level threads, optimistic fast synchronization etc. Whenever a preemption occurs, the kernel generates an RPC to a user-specified preempter (if the user specified one). Like a pager handles space faults, a preempter handles time faults.

- Address spaces are recursively constructed by user-level servers, also called pagers. Basic mechanisms are map, grant and unmap of *fpages*. An fpage (or flexpage) is a logical page of size 2^n, ranging from one physical page up to a complete address space.

Threads and address spaces are complemented by the *Clan & Chief* concept. A Clan is a set of tasks[2] headed by a Chief task. Inside the Clan, all messages are transferred directly. Messages crossing Clan borderlines are redirected by the μ-kernel to the corresponding Chief. Clans are not required for normal security but can be used to implement e.g. local reference monitors, multi-level-security policies and distributed systems. Since chiefs are user-level tasks, the clan concept allows sophisticated and user-definable checks as well as active control.

2.2. System calls: 7

ipc is the basic system call for inter-process communication and synchronization. All communication is synchronous and unbuffered: a message is transferred from the sender to the recipient if and only if the recipient has invoked a corresponding ipc operation. The sender blocks until this happens or a sender-specified timeout occurs.

Ipc can be used to copy data as well as to *map* or *grant* fpages from the sender's to the recipient's address space.

id_nearest delivers either the invoker's own id or the nearest chief towards the specified destination.

fpage_unmap unmaps the specified *fpage* from all address spaces into which the invoker mapped it directly or indirectly.

thread_switch releases the processor so that either a specified or an arbitrary thread can be executed.

thread_schedule Tasks acting as schedulers can define *priority*, *timeslice length* and an *external preempter* for all threads that currently run at a priority less or equal to the invoking task's *maximum controlled priority*.

lthread_ex_regs reads and writes (exchanges) some register values (e.g. instruction pointer and stack pointer) of another thread belonging to the same

[2] We use the term 'task' to denote an address space in conjunction with with its threads.

task. The system call serves to implement signalling, user-level threads and even thread creation and deletion. Conceptually, creating a task includes creating all of its threads. All except the first one initially run an idle loop. Of course, the kernel does neither allocate control blocks nor time slices etc. to them. Setting stack pointer and instruction pointer of such a thread then really generates the thread.

task_new deletes and creates (exchanges) a task. Tasks can be *active* or *inactive*. An active task consists of an address space and threads executing (or waiting) in this space. An inactive task is empty. It occupies no resources, has no address space and no threads. Loosely speaking, inactive tasks are not really existing but represent only the right to create an active task. Any task, active or inactive, belongs to a Clan. Only the Clan's Chief can modify the task by this system call. Besides deletion (active $\rightarrow$ inactive), creation (inactive $\rightarrow$ active) and replacement (active $\rightarrow$ active), the chief can also transfer an inactive task to another clan (inactive $\rightarrow$ inactive).

2.3. Code size: 12 K

The L4/486 μ-kernel needs slightly less than 12 K of code. This value does not cover the optional kernel debugger (10 K) and the initialization code (4 K) whose memory is released after initialization and made available at user level.

A short IPC operation needs approximately 10% of the 8 K primary cache.

Global kernel data (except page tables and the mapping database) are 4 to 12 K, depending on the number of allocated tasks. Besides some small tables required by the processor (IDT, GDT), it uses basically one (486) or two (Pentium) words per allocated task which points to the corresponding address-space root. The mapping database consists of one tree per physical page frame reflecting the effective mappings (by map, unmap and grant operations) of each frame. Since the nodes of the trees are more or less randomly distributed (it is a heap), the corresponding table has the bad cache properties of system-global data. However, the mapping database is only accessed when the mapping really changes which usually occurs infrequently. The frequent operations, IPC and address-space switch, do not access the mapping database.

References

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. 1995. Extensibility, safety and performance in the Spin operating system. In *15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain Resort, CO, pp. 267–284.

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

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

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

Liedtke, J. 1996. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021 (Sept.), GMD — German National Research Center for Information Technology, Sankt Augustin. also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, Sep 1996.