On μ-Kernel Construction

Jochen Liedtke
GMD — German National Research Center for Information Technology *
jochen.liedtke@gmd.de

Abstract

From a software-technology point of view, the μ-kernel concept is superior to large integrated kernels. On the other hand, it is widely believed that (a) μ-kernel based systems are inherently inefficient and (b) they are not sufficiently flexible. Contradictory to this belief, we show and support by documentary evidence that inefficiency and inflexibility of current μ-kernels is not inherited from the basic idea but mostly from overloading the kernel and/or from improper implementation.

Based on functional reasons, we describe some concepts which must be implemented by a μ-kernel and illustrate their flexibility. Then, we analyze the performance critical points. We show what performance is achievable, that the efficiency is sufficient with respect to macro-kernels and why some published contradictory measurements are not evident. Furthermore, we describe some implementation techniques and illustrate why μ-kernels are inherently not portable, although they improve portability of the whole system.

*GMD SET–RS, 53754 Sankt Augustin, Germany

1 Rationale

μ-kernel based systems have been built long before the term itself was introduced, e.g. by Brinch Hansen [1970] and Wulf et al. [1974]. Traditionally, the word 'kernel' is used to denote the part of the operating system that is mandatory and common to all other software. The basic idea of the μ-kernel approach is to minimize this part, i.e. to implement outside the kernel whatever possible.

The software technological advantages of this approach are obvious:

(a) A clear μ-kernel interface enforces a more modular system structure.[1]
(b) Servers can use the mechanisms provided by the μ-kernel like any other user program. Server malfunction is as isolated as any other user program's malfunction.
(c) The system is more flexible and tailorable. Different strategies and APIs, implemented by different servers, can coexist in the system.

Although much effort has been invested in μ-kernel construction, the approach is not (yet) generally accepted. This is due to the fact that most existing μ-kernels do not perform sufficiently well. Lack of efficiency also heavily restricts flexibility, since important mechanisms and principles cannot be used in practice due to poor performance. In some cases, the μ-kernel interface has been weakened and special servers have been re-integrated into the kernel to regain efficiency.

It is widely believed that the mentioned inefficiency (and thus inflexibility) is inherent to the μ-kernel approach. Folklore holds that increased user-kernel mode

[1] Although many macro-kernels tend to be less modular, there are exceptions from this rule, e.g. Chorus [Rozier et al. 1988] and Peace [Schröder-Preikschat 1994].

and address-space switches are responsible. At a first glance, published performance measurements seem to support this view.

In fact, the cited performance studies measured the performance of a particular μ-kernel based system *without analyzing the reasons* which limit efficiency. We can only guess whether it is caused by the μ-kernel *approach*, by the *concepts* implemented by this particular μ-kernel or by the *implementation* of the μ-kernel. Since it is known that conventional IPC, one of the traditional μ-kernel bottlenecks, can be implemented an order of magnitude faster[2] than believed before, the question is still open. It might be possible that we are still not applying the appropriate construction techniques.

For the above reasons, we feel that a conceptual analysis is needed which derives μ-kernel concepts from pure functionality requirements (section 2) and that discusses achievable performance (section 4) and flexibility (section 3). Further sections discuss portability (section 5) and the chances of some new developments (section 6).

2 Some μ-Kernel Concepts

In this section, we reason about the minimal concepts or "primitives" that a μ-kernel should implement.[3] The determining criterion used is functionality, not performance. More precisely, a concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

We assume that the target system has to support interactive and/or not completely trustworthy applications, i.e. it has to deal with protection. We further assume that the hardware implements page-based virtual memory.

One inevitable requirement for such a system is that a programmer must be able to implement an arbitrary subsystem S in such a way that it cannot be disturbed or corrupted by other subsystems S'. This is the principle of independence: S can give guarantees independent of S'. The second requirement is that other subsystems must be able to rely on these guarantees. This is the principle of integrity: there must be a way for S_1 to address S_2 and to establish a communication channel which can neither be corrupted nor eavesdropped by S'.

Provided hardware and kernel are trustworthy, further security services, like those described by Gasser et al. [1989], can be implemented by servers. Their integrity can be ensured by system administration or by user-level boot servers. For illustration: a key server should deliver public-secret RSA key pairs on demand. It should guarantee that each pair has the desired RSA property and that each pair is delivered only once and only to the demander. The key server can only be realized if there are mechanisms which (a) protect its code and data, (b) ensure that nobody else reads or modifies the key and (c) enable the demander to check whether the key comes from the key server. Finding the key server can be done by means of a name server and checked by public key based authentication.

2.1 Address Spaces

At the hardware level, an *address space* is a mapping which associates each virtual page to a physical page frame or marks it 'non-accessible'. For the sake of simplicity, we omit access attributes like read-only and read/write. The mapping is implemented by TLB hardware and page tables.

The μ-kernel, the mandatory layer common to all subsystems, has to hide the hardware concept of address spaces, since otherwise, implementing protection would be impossible. The μ-kernel concept of address spaces must be tamed, but must permit the implementation of arbitrary protection (and non-protection) schemes on top of the μ-kernel. It should be simple and similar to the hardware concept.

The basic idea is to support recursive construction of address spaces outside the kernel. By magic, there is one address space σ_0 which essentially represents the physical memory and is controlled by the first subsystem S_0. At system start time, all other address spaces are empty. For constructing and maintaining further address spaces on top of σ_0, the μ-kernel provides three operations:

Grant. The owner 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

[2] Short user-to-user cross-address space IPC in L3 [Liedtke 1993] is 22 times faster than in Mach, both running on a 486. On the R2000, the specialized Exo-tlrpc [Engler et al. 1995] is 30 times faster than Mach's general RPC.

[3] Proving minimality, necessarity and completeness would be nice but is impossible, since there is no agreed-upon metric and all is Turing-equivalent.

space. The important restriction is that instead of physical page frames, the granter can only grant pages which are already accessible to itself.

Map. 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. Comparable to the granting case, the mapper can only map pages which itself already can access.

Flush. The owner of an address space can *flush* any of its pages. The flushed page remains accessible in the flusher's address space, but is removed from all other address spaces which had received the page directly or indirectly from the flusher. Although explicit consent of the affected address-space owners is not required, the operation is safe, since it is restricted to own pages. The users of these pages already agreed to accept a potential flushing, when they received the pages by mapping or granting.

Appendix A contains a more precise definition of address spaces and the above three operations.

Reasoning

The described address-space concept leaves memory management and paging outside the μ-kernel; only the grant, map and flush operations are retained inside the kernel. Mapping and flushing are required to implement memory managers and pagers on top of the μ-kernel.

The grant operation is required only in very special situations: consider a pager F which combines two underlying file systems (implemented as pagers f_1 and f_2, operating on top of the standard pager) into one unified file system (see figure 1). In this example, f_1 maps one of its pages to F which grants the received page to *user A*. By granting, the page disappears from F so that it is then available only in f_1 and *user A*; the resulting mappings are denoted by the thin line: the page is mapped in *user A*, f_1 and the standard pager. Flushing the page by the standard pager would affect f_1 and *user A*, flushing by f_1 only *user A*. F is not affected by either flush (and cannot flush itself), since it used the page only transiently. If F had used mapping instead of granting, it would have needed to replicate most of the bookkeeping which is already done in f_1 and f_2. Furthermore, granting avoids a potential address-space overflow of F.

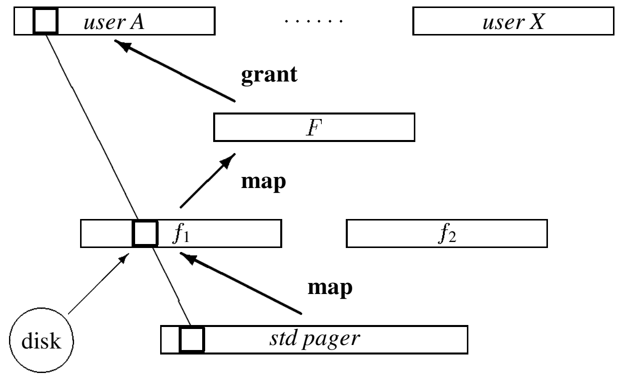

Figure 1: *A Granting Example.*

In general, granting is used when page mappings should be passed through a controlling subsystem without burdening the controller's address space by all pages mapped through it.

The model can easily be extended to access rights on pages. Mapping and granting copy the source page's access right or a subset of them, i.e., can restrict the access but not widen it. Special flushing operations may remove only specified access rights.

I/O

An address space is the natural abstraction for incorporating device ports. This is obvious for memory mapped I/O, but I/O ports can also be included. The granularity of control depends on the given processor. The 386 and its successors permit control per port (one very small page per port) but no mapping of port addresses (it enforces mappings with $v = v'$); the PowerPC uses pure memory mapped I/O, i.e., device ports can be controlled and mapped with 4K granularity.

Controlling I/O rights and device drivers is thus also done by memory managers and pagers on top of the μ-kernel.

2.2 Threads and IPC

A *thread* is an activity executing inside an address space. A thread τ is characterized by a set of registers, including at least an instruction pointer, a stack pointer and a state information. A thread's state also includes the address space $\sigma^{(\tau)}$ in which τ currently executes. This dynamic or static association to address spaces is the decisive reason for including the thread concept (or something equivalent) in the μ-kernel. To prevent corruption of address spaces, all changes to thread's address space ($\sigma^{(\tau)} := \sigma'$) must be controlled by the kernel. This implies that the μ-kernel includes the notion of some τ that represents the above mentioned activity. In some operating systems, there may be additional reasons for introducing threads as a basic abstraction, e.g. preemption. Note that choosing a concrete thread concept remains subject to further OS-specific design decisions.

Consequently, cross-address-space communication, also called inter-process communication (IPC), must be supported by the μ-kernel. The classical method is transferring messages between threads by the μ-kernel.

IPC always enforces a certain agreement between both parties of a communication: the sender decides to send information and determines its contents; the receiver determines whether it is willing to receive information and is free to interpret the received message. Therefore, IPC is not only the basic concept for communication between subsystems but also, together with address spaces, the foundation of independence.

Other forms of communication, remote procedure call (RPC) or controlled thread migration between address spaces, can be constructed from message-transfer based IPC.

Note that the *grant* and *map* operations (section 2.1) need IPC, since they require an agreement between granter/mapper and recipient of the mapping.

Supervising IPC

Architectures like those described by Yokote [1993] and Kühnhauser [1995] need not only supervise the memory of subjects but also their communication. This can be done by introducing either *communication channels* or *Clans* [Liedtke 1992] which allow supervision of IPC by user-defined servers. Such concepts are not discussed here, since they do not belong to the minimal set of concepts. We only remark that Clans do not burden the μ-kernel: their base cost is 2 cycles per IPC.

Interrupts

The natural abstraction for hardware interrupts is the IPC message. The hardware is regarded as a set of threads which have special thread ids and send empty messages (only consisting of the sender id) to associated software threads. A receiving thread concludes from the message source id, whether the message comes from a hardware interrupt and from which interrupt:

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

Transforming the interrupts into messages must be done by the kernel, but the μ-kernel is not involved in device-specific interrupt handling. In particular, it does not know anything about the interrupt semantics. On some processors, resetting the interrupt is a device specific action which can be handled by drivers at user level. The `iret`-instruction then is used solely for popping status information from the stack and/or switching back to user mode and can be hidden by the kernel. However, if a processor requires a privileged operation for releasing an interrupt, the kernel executes this action implicitly when the driver issues the next IPC operation.

2.3 Unique Identifiers

A μ-kernel must supply unique identifiers (uid) for something, either for threads or tasks or communication channels. Uids are required for reliable and efficient local communication. If S_1 wants to send a message to S_2, it needs to specify the destination S_2 (or some channel leading to S_2). Therefore, the μ-kernel must know which uid relates to S_2. On the other hand, the receiver S_2 wants to be sure that the message comes from S_1. Therefore the identifier must be unique, both in space and time.

In theory, cryptography could also be used. In practice, however, enciphering messages for local communication is far too expensive and the kernel must be trusted anyway. S_2 can also not rely on purely user-supplied capabilities, since S_1 or some other instance could duplicate and pass them to untrusted subsystems without control of S_2.

3 Flexibility

To illustrate the flexibility of the basic concepts, we sketch some applications which typically belong to the basic operating system but can easily be implemented on top of the μ-kernel. In this section, we show principal flexibility of a μ-kernel. Whether it is really as flexible in practice strongly depends on the achieved efficiency of the μ-kernel. The latter performance topic is discussed in section 4.

Memory Manager. A server managing the initial address space σ_0 is a classical main memory manager, but outside the μ-kernel. Memory managers can easily be stacked: M_0 maps or grants parts of the physical memory (σ_0) to σ_1, controlled by M_1, other parts to σ_2, controlled by M_2. Now we have two coexisting main memory managers.

Pager. A Pager may be integrated with a memory manager or use a memory managing server. Pagers use the μ-kernel's grant, map and flush primitives. The remaining interfaces, pager – client, pager – memory server and pager – device driver, are completely based on IPC and are user-level defined.

Pagers can be used to implement traditional paged virtual memory and file/database mapping into user address spaces as well as unpaged resident memory for device drivers and/or real time systems. Stacked pagers, i.e. multiple layers of pagers, can be used for combining access control with existing pagers or for combining various pagers (e.g. one per disk) into one composed object. User-supplied paging strategies [Lee et al. 1994; Cao et al. 1994] are handled at the user level and are in no way restricted by the μ-kernel. Stacked file systems [Khalidi and Nelson 1993] can be realized accordingly.

Multimedia Resource Allocation. Multimedia 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.

Note that resource allocators for multimedia and for timesharing can coexist. Managing allocation conflicts is part of the servers' jobs.

Device Driver. A device driver is a process which directly accesses hardware I/O ports mapped into its address space and receives messages from the hardware (interrupts) through the standard IPC mechanism. Device-specific memory, e.g. a screen, is handled by means of appropriate memory managers. Compared to other user-level processes, there is nothing special about a device driver. No device driver has to be integrated into the μ-kernel.[4]

Second Level Cache and TLB. Improving the hit rates of a secondary cache by means of page allocation or reallocation [Kessler and Hill 1992; Romer et al. 1994] can be implemented by means of a pager which applies some cache-dependent (hopefully conflict reducing) policy when allocating virtual pages in physical memory.

In theory, even a software TLB handler could be implemented like this. In practice, the first-level TLB handler will be implemented in the hardware or in the μ-kernel. However, a second-level TLB handler, e.g. handling misses of a hashed page table, might be implemented as a user-level server.

Remote Communication. 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 space is required, the communication server will also act as a special pager for the client. The μ-kernel is not involved.

Unix Server. Unix[5] system calls are implemented by IPC. The Unix server can act as a pager for its clients and also use memory sharing for communicating with its clients. The Unix server itself can be pageable or resident.

Conclusion. A small set of μ-kernel concepts lead to abstractions which stress flexibility, provided they perform well enough. The only thing which cannot be implemented on top of these abstractions is the processor architecture, registers, first-level caches and first-level TLBs.

[4] In general, there is no reason for integrating boot drivers into the kernel. The booter, e.g. located in ROM, simply loads a bit image into memory that contains the micro-kernel and perhaps some set of initial pagers and drivers (running at user mode and *not* linked but simply appended to the kernel). Afterwards, the boot drivers are no longer used.

[5] Unix is a registered trademark of UNIX System Laboratories.

4 Performance, Facts & Rumors

4.1 Switching Overhead

It is widely believed that switching between kernel and user mode, between address spaces and between threads is inherently expensive. Some measurements seem to support this belief.

4.1.1 Kernel–User Switches

Ousterhout [1990] measured the costs for executing the "null" kernel call *getpid*. Since the real getpid operation consists only of a few loads and stores, this method measures the basic costs of a kernel call. Normalized to a hypothetical machine with 10 MIPS rating (10× VAX 11/780 or roughly a 486 at 50 MHz), he showed that most machines need 20–30 μs per *getpid*, one required even 63 μs. Corroborating these results, we measured 18 μs per Mach[6] μ-kernel call *get_self_thread*. In fact, the measured kernel-call costs are high.

For analyzing the measured costs, our argument is based on a 486 (50 MHz) processor. We take an x86 processor, because kernel-user mode switches are extremely expensive on these processors. In contrast to the worst case processor, we use a best-case measurement for discussion, 18 μs for Mach on a 486/50.

The measured costs per kernel call are $18 \times 50 = 900$ cycles. The bare machine instruction for entering kernel mode costs 71 cycles, followed by an additional 36 cycles for returning to user mode. These two instructions switch between the user and kernel stack and push/pop flag register and instruction pointer. 107 cycles (about 2 μs) is therefore a lower bound on kernel–user mode switches. The remaining 800 or more cycles are pure *kernel overhead*. By this term, we denote all cycles which are solely due to the construction of the kernel, nevermind whether they are spent in executing instructions (800 cycles $\approx$ 500 instructions) or in cache and TLB misses (800 cycles $\approx$ 270 primary cache misses $\approx$ 90 TLB misses). We have to conclude that the measured kernels do a lot of work when entering and exiting the kernel. Note that this work by definition has no net effect.

Is an 800 cycle kernel overhead really necessary? The answer is *no*. Empirical proof: L3 [Liedtke 1993] has a minimal kernel overhead of 15 cycles. If the μ-kernel call is executed infrequently enough, it may increase by up to 57 additional cycles (3 TLB misses, 10 cache misses). The complete L3 kernel call costs are thus 123 to 180 cycles, mostly less than 3 μs.

The L3 μ-kernel is process oriented, uses a kernel stack per thread and supports persistent user processes (i.e. the kernel can be exchanged without affecting the remaining system, even if a process actually resides in kernel mode). Therefore, it should be possible for any other μ-kernel to achieve comparably low kernel call overhead on the same hardware.

Other processors may require a slightly higher overhead, but they offer substantially cheaper basic operations for entering and leaving kernel mode. >From an architectural point of view, calling the kernel from user mode is simply an indirect call, complemented by a stack switch and setting the internal 'kernel'-bit to permit privileged operations. Accordingly, returning from kernel mode is a normal return operation complemented by switching back to user stack and resetting the 'kernel'-bit. If the processor has different stack pointer registers for user and kernel stack, the stack switching costs can be hidden. Conceptually, entering and leaving kernel mode can perform exactly like a normal indirect call and return instruction (which do not rely on branch prediction). Ideally, this means 2+2=4 cycles on a 1-issue processor

Conclusion. Compared to the theoretical minimum, kernel–user mode switches are costly on some processors. Compared to existing kernels however, they can be improved 6 to 10 times by appropriate μ-kernel construction. Kernel–user mode switches are not a serious conceptual problem but an implementational one.

4.1.2 Address Space Switches

Folklore also considers address-space switches as costly. All measurements known to the author and related to this topic deal with combined thread and address-space switch costs. Therefore, in this section, we analyze only the architectural processor costs for pure address-space switching. The combined measurements are discussed together with thread switching.

Most modern processors use a physically indexed primary cache which is not affected by address-space switching. Switching the page table is usually very cheap: 1 to 10 cycles. The real costs are determined by the TLB architecture.

Some processors (e.g. Mips R4000) use tagged TLBs,

[6] Mach 3.0, NORMA MK 13

where each entry does not only contain the virtual page address but also the address-space id. Switching the address space is thus transparent to the TLB and costs no additional cycles. However, address-space switching may induce indirect costs, since shared pages occupy one TLB entry per address space. Provided that the μ-kernel (shared by all address spaces) has a small working set and that there are enough TLB entries, the problem should not be serious. However, we cannot support this empirically, since we do not know an appropriate μ-kernel running on such a processor.

Most current processors (e.g. 486, Pentium, PowerPC and Alpha) include untagged TLBs. An address-space switch thus requires a TLB flush. The real costs are determined by the TLB load operations which are required to re-establish the current working set later. If the working set consists of n pages, the TLB is fully-associative, has s entries and a TLB miss costs m cycles, at most min$(n, s) \times m$ cycles are required in total.

Apparently, larger untagged TLBs lead to a performance problem. For example, completely reloading the Pentium's data and code TLBs requires at least $(32 + 64) \times 9 = 864$ cycles. Therefore, intercepting a program every 100μs could imply an overhead of up to 9%. Although using the complete TLB is unrealistic[7], this worst-case calculation shows that switching page tables may become critical in some situations.

Fortunately, this is not a problem, since on Pentium and PowerPC, address-space switches can be handled differently. The PowerPC architecture includes segment registers which can be controlled by the μ-kernel and offer an additional address translation facility from the local 2^{32}-byte address space to a global 2^{52}-byte space. If we regard the global space as a set of one million local spaces, address-space switches can be implemented by reloading the segment registers instead of switching the page table. With 29 cycles for 3.5 GB or 12 cycles for 1 GB segment switching, the overhead is low compared to a no longer required TLB flush. In fact, we have a tagged TLB.

Things are not quite as easy on the Pentium or the 486. Since segments are mapped into a 2^{32}-byte space, mapping multiple user address spaces into one linear space must be handled dynamically and depends on the actually used sizes of the active user address spaces. The according implementation technique [Liedtke 1995] is transparent to the user and removes the potential performance bottleneck. Address space switch overhead then is 15 cycles on the Pentium and 39 cycles on 486.

For understanding that the restriction of a 2^{32}-byte global space is not crucial to performance, one has to mention that address spaces which are used only for very short periods and with small working sets are effectively very small in most cases, say 1 MB or less for a device driver. For example, we can multiplex one 3 GB user address space with 8 user spaces of 64 MB and additionally 128 user spaces of 1 MB. The trick is to share the smaller spaces with *all* large 3 GB spaces. Then any address-space switch to a medium or small space is always fast. Switching between two large address spaces is uncritical anyway, since switching between two large working sets implies TLB and cache miss costs, nevermind whether the two programs execute in the same or in different address spaces.

Table 1 shows the page table switch and segment switch overhead for several processors. For a TLB miss, the minimal and maximal cycles are given (provided that no referenced or modified bits need updating). In the case of 486, Pentium and PowerPC, this depends on whether the corresponding page table entry is found in the cache or not. As a minimal working set, we assume 4 pages. For the maximum case, we exclude 4 pages from the address-space overhead costs, because at most 4 pages are required by the μ-kernel and thus would as well occupy TLB entries when the address space would not be switched.

	TLB entries	TLB miss cycles	Page Table switch cycles	Segment switch cycles
486	32	9...13	36...364	39
Pentium	96	9...13	36...1196	15
PowerPC 601	256	?	?	29
Alpha 21064	40	20...50[a]	80...1800	n/a
Mips R4000	48	20...50[a]	0[b]	n/a

[a] Alpha and Mips TLB misses are handled by software.
[b] R4000 has a tagged TLB.

Table 1: *Address Space Switch Overhead*

Conclusion. Properly constructed address-space switches are not very expensive, less than 50 cycles on modern processors. On a 100 MHz processor, the

[7] Both TLBs are 4-way set-associative. Working sets which are not compact in the virtual address space, usually imply some conflicts so that only about half of the TLB entries are used simultaneously. Furthermore, a working set of 64 data pages will most likely lead to cache thrashing: in best case, the cache supports 4×32 bytes per page. Since the cache is only 2-way set-associative, probably only 1 or 2 cache entries can be used per page in practice.

inherited costs of address-space switches can be ignored roughly up to 100,000 switches per second. Special optimizations, like executing dedicated servers in kernel space, are superfluous. Expensive context switching in some existing μ-kernels is due to implementation and not caused by inherent problems with the concept.

4.1.3 Thread Switches and IPC

Ousterhout [1990] also measured context switching in some Unix systems by echoing one byte back and forth through pipes between two processes. Again normalized to a 10 Mips machine, most results are between 400 and 800 μs per ping-pong, one was 1450 μs. All existing μ-kernels are at least 2 times faster, but it is proved by construction that 10 μs, i.e. a 40 to 80 times faster RPC is achievable. Table 2 gives the costs of echoing one byte by a round trip RPC, i.e. two IPC operations.[8]

System	CPU, MHz	RPC time (round trip)	cycles/IPC (oneway)
full IPC semantics			
L3	486, 50	10 μs	250
QNX	486, 33	76 μs	1254
Mach	R2000, 16.7	190 μs	1584
SRC RPC	CVAX, 12.5	464 μs	2900
Mach	486, 50	230 μs	5750
Amoeba	68020, 15	800 μs	6000
Spin	Alpha 21064, 133	102 μs	6783
Mach	Alpha 21064, 133	104 μs	6916
restricted IPC semantics			
Exo-tlrpc	R2000, 16.7	6 μs	53
Spring	SparcV8, 40	11 μs	220
DP-Mach	486, 66	16 μs	528
LRPC	CVAX, 12.5	157 μs	981

Table 2: *1-byte-RPC performance*

All times are user to user, cross-address space.They include system call, argument copy, stack and address space switch costs. Exokernel, Spring and L3 show that communication can be implemented pretty fast and that the costs are heavily influenced by the processor architecture: Spring on Sparc has to deal with register windows, whereas L3 is burdened by the fact that a 486 trap is 100 cycles more expensive than a Sparc trap.

The effect of using segment based address-space switch on Pentium is shown in figure 2. One long running application with a stable working set (2 to 64 data pages) executes a short RPC to a server with a small working set (2 pages). After the RPC, the application re-accesses all its pages. Measurement is done by 100,000 repetitions and comparing each run against running the application (100,000 time accessing all pages) without RPC. The given times are round trip RPC times, user to user, plus the required time for re-establishing the application's working set.

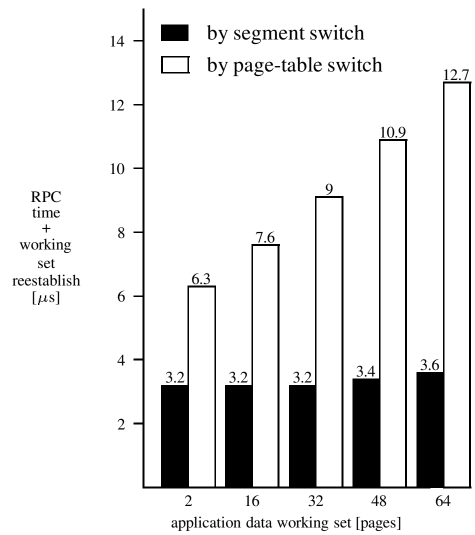

Figure 2: *Segmented Versus Standard Address-Space Switch in L4 on Pentium, 90 MHz.*

Conclusion. IPC can be implemented fast enough to handle also hardware interrupts by this mechanism.

4.2 Memory Effects

Chen and Bershad [1993] compared the memory system behaviour of Ultrix, a large monolithic Unix system, with that of the Mach μ-kernel which was complemented with a Unix server. They measured memory cycle overhead per instruction (MCPI) and found that programs running under Mach + Unix server had a substantially higher MCPI than running the same programs under Ultrix. For some programs, the differences were up to 0.25 cycles per instruction, averaged over the total program (user + system). Similar memory system degradation of Mach versus Ultrix is noticed by others [Nagle et al. 1994]. The crucial point is whether

[8] The respective data is taken from [Liedtke 1993; Hildebrand 1992; Schroeder and Burroughs 1989; Draves et al. 1991; van Renesse et al. 1988; Liedtke 1993; Bershad et al. 1995; Engler et al. 1995; Hamilton and Kougiouris 1993; Bryce and Muller 1995; Bershad et al. 1989].

this problem is due to the way that Mach is constructed, or whether it is caused by the μ-kernel approach.

Chen and Bershad [1993, p. 125] state: "This suggests that microkernel optimizations focussing exclusively on IPC [...], without considering other sources of system overhead such as MCPI, will have a limited impact on overall system performance." Although one might suppose a principal impact of OS architecture, the mentioned paper exclusively presents facts "as is" about a specific implementation without analyzing the reasons for memory system degradation.

Careful analysis of the results is thus required. According to the original paper, we comprise under 'system' either all Ultrix activities or the joined activities of the Mach μ-kernel, Unix emulation library and Unix server. The Ultrix case is denoted by U, the Mach case by M. We restrict our analysis to the samples that show a significant MCPI difference for both systems: *sed, egrep, yacc, gcc, compress, espresso* and the andrew benchmark *ab*.

In figure 3, we present the results of Chen's figure 2-1 in a slightly reordered way. We have colored MCPI

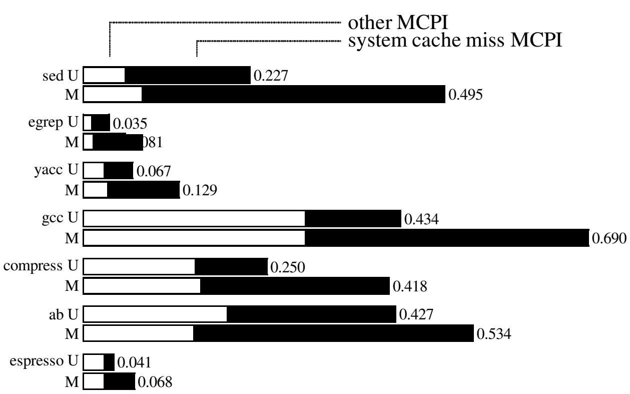

Figure 3: *Baseline MCPI for Ultrix and Mach.*

black that are due to system i-cache or d-cache misses. The white bars comprise all other causes, system write buffer stalls, system uncached reads, user i-cache and d-cache misses and user write buffer stalls. It is easy to see that the white bars do not differ significantly between Ultrix and Mach; the average difference is 0.00, the standard deviation is 0.02 MCPI.

We conclude that the differences in memory system behaviour are essentially caused by increased system cache misses for Mach. They could be conflict misses (the measured system used direct mapped caches) or capacity misses. A large fraction of conflict misses would suggest a potential problem due to OS *structure*.

Chen and Bershad measured cache conflicts by comparing the direct mapped to a simulated 2-way cache.[9] They found that system self-interference is more important than user/system interference, but the data also show that the ratio of conflict to capacity misses in Mach is *lower* than in Ultrix. Table 4 shows the conflict (black) and capacity (white) system cache misses both in an absolute scale (left) and as ratio (right).

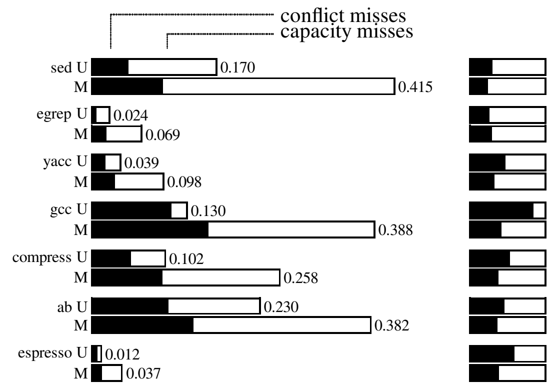

Figure 4: *MCPI Caused by Cache Misses.*

>From this we can deduce that the increased cache misses are caused by higher cache consumption of the system (Mach + emulation library + Unix server), not by conflicts which are inherent to the system's structure.

The next task is to find the component which is responsible for the higher cache consumption. We assume that the used Unix single server behaves comparably to the corresponding part of the Ultrix kernel. This is supported by the fact that the samples spent even fewer instructions in Mach's Unix server than in the corresponding Ultrix routines. We also exclude Mach's emulation library, since Chen and Bershad report that only 3% or less of system overhead is caused by it.

What remains is Mach itself, including trap handling, IPC and memory management, which therefore must induce nearly all of the additional cache misses.

Therefore, the mentioned measurements suggest that memory system degradation is caused solely by high cache consumption of the μ-kernel. Or in other words: drastically reducing the cache working set of a μ-kernel will solve the problem.

Since a μ-kernel is basically a set of procedures which are invoked by user-level threads or hardware, a high

[9]Although this method does not determine all conflict misses as defined by Hill and Smith [1989], it can be used as a first-level approximation.

cache consumption can only[10] be explained by a large number of very frequently used μ-kernel operations or by high cache working sets of a few frequently used operations. According to section 2, the first case has to be considered as a conceptual mistake. Large cache working sets are also not an inherent feature of μ-kernels. For example, L3 requires less than 1 K for short IPC. (Recall: voluminous communication can be made by dynamic or static mapping so that the cache is not flooded by copying very long messages.)

Mogul and Borg [1991] reported an increase in cache misses after preemptively-scheduled context switches between applications with large working sets. This depends mostly on the application load and the requirement for interleaved execution (timesharing). The type of kernel is almost irrelevant. We showed (section 4.1.2 and 4.1.3) that μ-kernel context switches are not expensive in the sense that there is not much difference between executing application + servers in one or in multiple address spaces.

Conclusion. The hypothesis that μ-kernel architectures inherently lead to memory system degradation is not substantiated. On the contrary, the quoted measurements support the hypothesis that properly constructed μ-kernels will automatically avoid the memory system degradation measured for Mach.

5 Non-Portability

Older μ-kernels were built machine-independently on top of a small hardware-dependent layer. This approach has strong advantages from the software technological point of view: programmers did not need to know very much about processors and the resulting μ-kernels could easily be ported to new machines. Unfortunately, this approach prevented these μ-kernels from achieving the necessary performance and thus flexibility.

In retrospective, we should not be surprised, since building a μ-kernel on top of abstract hardware has serious implications:

- Such a μ-kernel cannot take advantage of specific hardware.
- It cannot take precautions to circumvent or avoid performance problems of specific hardware.
- The additional layer per se costs performance.

μ-kernels form the lowest layer of operating systems beyond the hardware. Therefore, we should accept that they are as hardware dependent as optimizing code generators. We have learned that not only the coding but

- even the algorithms used inside a μ-kernel and its internal concepts are extremely processor dependent.

5.1 Compatible Processors

For illustration, we briefly describe how a μ-kernel has to be *conceptually* modified even when "ported" from 486 to Pentium, i.e. to a compatible processor.

Although the Pentium processor is binary compatible to the 486, there are some differences in the internal hardware architecture (see table 3) which influence the internal μ-kernel architecture:

		486		Pentium	
TLB	entries, ways	32 (u)	4×	32 (i) + 64 (d)	4×
Cache	size, ways	8K (u)	4×	8K (i) + 8K (d)	2×
	line, write	16B	through	32B	back
fast instructions		1 cycle		0.5–1 cycle	
segment register		9 cycles		3 cycles	
trap		107 cycles		69 cycles	

Table 3: *486 / Pentium Differences*

User-address-space implementation. As mentioned in section 4.1.2, a Pentium μ-kernel should use segment registers for implementing user address spaces so that each 2^{32}-byte hardware address space shares all small and one large user address space. Recall that this can be implemented transparently to the user.

Ford [1993] proposed a similar technique for the 486, and table 1 also suggests it for the 486. Nevertheless, the conventional hardware-address-space switch is preferrable on this processor. Expensive segment register loads and additional instructions at various places in the kernel sum to roughly 130 cycles required in addition. Now look at the relevant situation: an address-space switch from a large space to a small one and back to the

[10] We do not believe that the Mach kernel flushes the cache explicitly. The measured system was a uniprocessor with physically tagged caches. The hardware does not even require explicit cache flushes for DMA.

large. Assuming cache hits, the costs of the segment register model would be $(130+39) \times 2 = 338$ cycles, whereas the conventional address-space model would require $28 \times 9 + 36 = 288$ cycles in the theoretical case of 100% TLB use, $14 \times 9 + 36 = 162$ cycles for the more probable case that the large address space uses only 50% of the TLB and only 72 cycles in the best case. In total, the conventional method wins.

On the Pentium however, the segment register method pays. The reasons are several: (a) Segment register loads are faster. (b) Fast instructions are cheaper, whereas the overhead by trap and TLB misses remain nearly constant. (c) Conflict cache misses (which, relative to instruction execution, are anyway more expensive) are more likely because of reduced associativity. Avoiding TLB misses thus also reduces cache conflicts. (d) Due to the three times larger TLB, the flush costs can increase substantially. As a result, on Pentium, the segment register method always pays (see figure 2).

As a consequence, we have to implement an additional user-address-space multiplexer, we have to modify address-space switch routines, handling of user supplied addresses, thread control blocks, task control blocks, the IPC implementation and the address-space structure as seen by the kernel. In total, the mentioned changes affect algorithms in about half of all μ-kernel modules.

IPC implementation. Due to reduced associativity, the Pentium caches tend to exhibit increased conflict misses. One simple way to improve cache behaviour during IPC is by restructuring the thread control block data such that it profits from the doubled cache line size. This can be adopted to the 486 kernel, since it has no effect on 486 and can be implemented transparently to the user.

In the 486 kernel, thread control blocks (including kernel stacks) were page aligned. IPC always accesses 2 control blocks and kernel stacks simultaneously. The cache hardware maps the according data of both control blocks to identical cache addresses. Due to its 4-way associativity, this problem could be ignored on the 486. However, Pentium's data cache is only 2-way set-associative. A nice optimization is to align thread control blocks no longer on 4K but on 1K boundaries. (1K is the lower bound due to internal reasons.) Then there is a 75% chance that two randomly selected control blocks do not compete in the cache.

Surprisingly, this affects the internal bit-structure of unique thread identifiers supplied by the μ-kernel (see [Liedtke 1993] for details). Therefore, the new kernel cannot simply replace the old one, since (persistent) user programs already hold uids which would become invalid.

5.2 Incompatible Processors

Processors of competing families differ in instruction set, register architecture, exception handling, cache/TLB architecture, protection and memory model. Especially the latter ones radically influence μ-kernel structure. There are systems with

- multi-level page tables,
- hashed page tables,
- (no) reference bits,
- (no) page protection,
- strange page protection[11],
- single/multiple page sizes,
- 2^{32}-, 2^{43}-, 2^{52}- and 2^{64}-byte address spaces,
- flat and segmented address spaces,
- various segment models,
- tagged/untagged TLBs,
- virtually/physically tagged caches.

The differences are orders of magnitude higher than between 486 and Pentium. We have to expect that a new processor requires a new μ-kernel design.

For illustration, we compare two different kernels on two different processors: the Exokernel [Engler et al. 1995] running on an R2000 and L3 running on a 486. Although this is similar to comparing apples with oranges, a careful analysis of the performance differences helps understanding the performance-determining factors and weighting the differences in processor architecture. Finally, this results in different μ-kernel architectures.

We compare Exokernel's protected control transfer (PCT) with L3's IPC. Exo-PCT on the R2000 requires about 35 cycles, whereas L3 takes 250 cycles on a 486 processor for an 8-byte message transfer. If this difference cannot be explained by different functionality and/or average processor performance, there must be an anomaly relevant to μ-kernel design.

Exo-PCT is a "substrate for implementing efficient IPC mechanisms. [It] changes the program counter to

[11] e.g. the 386 ignores write protection in kernel mode, the PowerPC supports read only in kernel mode but this implies that the page is seen in user mode as well.

an agreed-upon value in the callee, donates the current time-slice to the callee's processor environment, and installs required elements of the callee's processor context." L3-IPC is used for secure communication between potentially untrusted partners; it therefore additionally checks the communication permission (whether the partner is willing to receive a message from the sender and whether no clan borderline is crossed), synchronizes both threads, supports error recovery by send and receive timeouts, and permits complex messages to reduce marshaling costs and IPC frequency. From our experience, extending Exo-PCT accordingly should require no more than 30 additional cycles. (Note that using PCT for a trusted LRPC already costs an additional 18 cycles, see table 2.) Therefore, we assume that a hypothetical L3-equivalent "Exo-IPC" would cost about 65 cycles on the R2000. Finally, we must take into consideration that the cycles of both processors are not equivalent as far as most-frequently-executed instructions are concerned. Based on SpecInts, roughly 1.4 486-cycles appear to do as much work as one R2000-cycle; comparing the five instructions most relevant in this context (2-op-alu, 3-op-alu, load, branch taken and not taken) gives 1.6 for well-optimized code. Thus we estimate that the Exo-IPC would cost up to approx. 100 486-cycles being definitely less than L3's 250 cycles.

This substantial difference in timing indicates an *isolated difference* between both processor architectures that strongly influences IPC (and perhaps other μ-kernel mechanisms), but not average programs.

In fact, the 486 processor imposes a high penalty on entering/exiting the kernel and requires a TLB flush per IPC due to its untagged TLB. This costs at least $107 + 49 = 156$ cycles. On the other hand, the R2000 has a tagged TLB, i.e. avoids the TLB flush, and needs less than 20 cycles for entering and exiting the kernel. From the above example, we learn two lessons:

- For well-engineered μ-kernels on different processor architectures, in particular with different memory systems, we should expect isolated timing differences that are not related to overall processor performance.
- Different architectures require processor-specific optimization techniques that even affect the global μ-kernel structure.

To understand the second point, recall that the mandatory 486-TLB flush requires minimization of the number of subsequent TLB misses. The relevant techniques [Liedtke 1993, pp. 179,182–183] are mostly based on proper address space construction: concentrating processor-internal tables and heavily used kernel data in one page (there is no unmapped memory on then 486), implementing control blocks and kernel stacks as virtual objects, lazy scheduling. In toto, these techniques save 11 TLB misses, i.e. at least 99 cycles on the 486 and are thus inevitable.

Due to its unmapped memory facility and tagged TLB, the mentioned constraint disappears on the R2000. Consequently, the internal structure (address space structure, page fault handling, perhaps control block access and scheduling) of a corresponding kernel can substantially differ from a 486-kernel. If other factors also imply implementing control blocks as physical objects, even the uids will differ between the R2000 (*no* $\times$ *pointer size* $+x$) and 486 kernel (*no* $\times$ *control block size* $+x$).

Conclusion. μ-kernels form the link between a minimal "μ"-set of abstractions and the bare processor. The performance demands are comparable to those of earlier microprogramming. As a consequence, μ-kernels are inherently not portable. Instead, they are the processor dependent basis for portable operating systems.

6 Synthesis, Spin, DP-Mach, Panda, Cache and Exokernel

Synthesis. Henry Massalin's Synthesis operating system [Pu et al. 1988] is another example of a high performing (and non-portable) kernel. Its distinguishing feature was a kernel-integrated "compiler" which generated kernel code at runtime. For example, when issuing a read pipe system call, the Synthesis kernel generated specialized code for reading out of *this pipe* and modified the respective invocation. This technique was highly successful on the 68030. However (a good example for non-portability), it would most probably no longer pay on modern processors, because (a) code inflation will degrade cache performance and (b) frequent generation of small code chunks pollutes the instruction cache.

Spin. Spin [Bershad et al. 1994; Bershad et al. 1995] is a new development which tries to extend the Synthesis idea: user-supplied algorithms are translated by a kernel compiler and added to the kernel, i.e. the user may write new system calls. By controlling branches and memory references, the compiler ensures that the newly-generated code does not violate kernel or user integrity. This approach reduces kernel–user mode switches and sometimes address space switches. Spin is based on Mach and may thus inherit many of its inefficiencies which makes it difficult to evaluate performance results. Rescaling them to an efficient μ-kernel with fast kernel–user mode switches and fast IPC is needed. The most crucial problem, however, is the estimation of how an optimized μ-kernel architecture and the requirements coming from a kernel compiler interfere with each other. Kernel architecture and performance might be e.g. affected by the requirement for larger kernel stacks. (A pure μ-kernel needs only a few hundred bytes per kernel stack.) Furthermore, the costs of safety-guaranteeing code have to be related to μ-kernel overhead and to optimal user-level code.

The first published results [Bershad et al. 1995] cannot answer these questions: On an Alpha 21064, 133 MHz, a Spin system call needs nearly twice as many cycles (1600, 12μs) as the already expensive Mach system call (900, 7μs). The application measurements show that Mach can be substantially improved by using a kernel compiler; however, it remains open whether this technique can reach or outperform a pure μ-kernel approach like that described here. For example, a simple user-level page-fault handler (1100 μs under Mach) executes in 17 μs under Spin. However, we must take into consideration that in a traditional μ-kernel, the kernel is invoked and left only twice: page fault (enter), message to pager (exit), reply map message (enter+exit). The Spin technique can save only one system call which on this processor *should* cost less than 1 μs i.e. with 12 μs the actual Spin overhead is far beyond the ideal traditional overhead of 1+1 μs.

>From our experience, we expect a notable gain if a kernel compiler eliminates nested IPC redirection, e.g. when using deep hierarchies of Clans or Custodians [Härtig et al. 1993]. Efficient integration of the kernel compiler technique and appropriate μ-kernel design might be a promising research direction.

Utah-Mach. Ford and Lepreau [1994] changed Mach IPC semantics to migrating RPC which is based on thread migration between address spaces, similar to the Clouds model [Bernabeu-Auban et al. 1988]. Substantial performance gain was achieved, a factor of 3 to 4.

DP-Mach. DP-Mach [Bryce and Muller 1995] implements multiple domains of protection within one user address space and offers a protected inter-domain call. The performance results (see table 2) are encouraging. However, although this inter-domain call is highly specialized, it is twice as slow as achievable by a general RPC mechanism. In fact, an inter-domain call needs two kernel calls and two address-space switches. A general RPC requires two additional thread switches and argument transfers[12]. Apparently, the kernel call and address-space switch costs dominate. Bryce and Muller presented an interesting optimization for small inter-domain calls: when switching back from a very small domain, the TLB is only selectively flushed. Although the effects are rather limited on their host machine (a 486 with only 32 TLB entries), it might become more relevant on processors with larger TLBs. To analyze whether kernel enrichment by inter-domain calls pays, we need e.g. a Pentium implementation and then compare it with a general RPC based on segment switching.

Panda. The Panda system's [Assenmacher et al. 1993] μ-kernel is a further example of a small kernel which delegates as much as possible to user space. Besides its two basic concepts *protection domain* and *virtual processor*, the Panda kernel handles only interrupts and exceptions.

Cache-Kernel. The Cache-kernel [Cheriton and Duda 1994] is also a small and hardware-dependent μ-kernel. In contrast to the Exokernel, it relies on a small fixed (non extensible) virtual machine. It caches kernels, threads, address spaces and mappings. The term 'caching' refers to the fact that the μ-kernel never handles the complete set of e.g. all address spaces, but only a dynamically selected subset. It was hoped that this technique would lead to a smaller μ-kernel interface and also to less μ-kernel code, since it no longer has to deal with special but infrequent cases. In fact, this could be done as well on top of a pure μ-kernel by means of according pagers. (Kernel data structures, e.g.

[12] Sometimes, the argument transfer can be omitted. For implementing inter-domain calls, a pager can be used which shares the address spaces of caller and callee such that the trusted callee can access the parameters in the caller space. E.g. LRPC [Bershad et al. 1989] and NetWare [Major et al. 1994] use a similar technique.

thread control blocks, could be held in virtual memory in the same way as other data.)

Exokernel. In contrast to Spin, the Exokernel [Engler et al. 1994; Engler et al. 1995] is a small and hardware-dependent μ-kernel. In accordance with our processor-dependency thesis, the exokernel is tailored to the R2000 and gets excellent performance values for its primitives. In contrast to our approach, it is based on the philosophy that a kernel should *not* provide abstractions but only a minimal set of primitives. Consequently, the Exokernel interface is archtecture dependent, in particular dedicated to software-controlled TLBs. A further difference to our driver-less μ-kernel approach is that Exokernel appears to partially integrate device drivers, in particular for disks, networks and frame buffers.

We believe that dropping the abstractional approach could only be justified by substantial performance gains. Whether these can be achieved remains open (see discussion in section 5.2) until we have well-engineered exo- and abstractional μ-kernels on the same hardware platform. It might then turn out that the right abstractions are even more efficient than securely multiplexing hardware primitives or, on the other hand, that abstractions are too inflexible. We should try to decide these questions by constructing comparable μ-kernels on at least two reference platforms. Such a co-construction will probably also lead to new insights for both approaches.

7 Conclusions

A μ-kernel can provide higher layers with a minimal set of appropriate abstractions that are flexible enough to allow implementation of arbitrary operating systems and allow exploitation of a wide range of hardware. The presented mechanisms (*address space* with map, flush and grant operation, *threads* with IPC and *unique identifiers*) form such a basis. Multi-level-security systems may additionally need *clans* or a similar reference monitor concept. Choosing the right abstractions is crucial for both flexibility and performance. Some existing μ-kernels chose inappropriate abstractions, or too many or too specialized and inflexible ones.

Similar to optimizing code generators, μ-kernels must be constructed per processor and are inherently not portable. Basic implementation decisions, most algorithms and data structures inside a μ-kernel are processor dependent. Their design must be guided by performance prediction and analysis. Besides inappropriate basic abstractions, the most frequent mistakes come from insufficient understanding of the combined hardware-software system or inefficient implementation.

The presented design shows that it is possible to achieve well performing μ-kernels through processor-specific implementations of processor-independent abstractions.

Availability

The source code of the L4 μ-kernel, a successor of the L3 μ-kernel, is available for examination and experimentation through the web:

http://borneo.gmd.de/RS/L4.

Acknowledgements

Many thanks to Hermann H"artig for discussion and Rich Uhlig for proofreading and stylistic help. Further thanks for reviewing remarks to Dejan Milojicic, some anonymous referees and Sacha Krakowiak for shepherding.

A Address Spaces

An Abstract Model of Address Spaces

We describe address spaces as mappings. $\sigma_0 : V \rightarrow R \cup \{\phi\}$ is the initial address space, where V is the set of virtual pages, R the set of available physical (real) pages and ϕ the nilpage which cannot be accessed. Further address spaces are defined recursively as mappings $\sigma : V \rightarrow (\Sigma \times V) \cup \{\phi\}$, where Σ is the set of address spaces. It is convenient to regard each mapping as a one column table which contains $\sigma(v)$ for all $v \in V$ and can be indexed by v. We denote the elements of this table by σ_v.

All modifications of address spaces are based on the replacement operation: we write $\sigma_v \leftarrow x$ to describe a change of σ at v, precisely:

$$\text{flush}\,(\sigma, v) \quad ; \quad \sigma_v := x \quad .$$

A page potentially mapped at v in σ is flushed, and the new value x is copied into σ_v. This operation is internal to the μ-kernel. We use it only for describing the three exported operations.

A subsystem S with address space σ can *grant* any of its pages v to a subsystem S' with address space σ' provided S' agrees:

$$\sigma'_{v'} \leftarrow \sigma_v \quad , \quad \sigma_v \leftarrow \phi \quad .$$

Note that S determines which of its pages should be granted, whereas S' determines at which virtual address the granted page should be mapped in σ'. The granted page is transferred to σ' and removed from σ.

A subsystem S with address space σ can *map* any of its pages v to a subsystem S' with address space σ' provided S' agrees:

$$\sigma'_{v'} \leftarrow (\sigma, v) \quad .$$

In contrast to grant, the mapped page remains in the mapper's space σ and *a link to the page in the mapper's address space* (σ, v) *is stored in the receiving address space* σ', instead of transferring the existing link from σ_v to $\sigma'_{v'}$. This operation permits to construct address spaces recursively, i.e. new spaces based on existing ones.

Flushing, the reverse operation, can be executed without explicit agreement of the mappees, since they agreed implicitly when accepting the prior map operation. S can *flush* any of its pages:

$$\forall_{\sigma'_{v'} = (\sigma,v)} : \sigma'_{v'} \leftarrow \phi \quad .$$

Note that $\leftarrow$ and *flush* are defined recursively. Flushing recursively affects also all mappings which are indirectly derived from σ_v.

No cycles can be established by these three operations, since $\leftarrow$ flushes the destination prior to copying.

Implementing the Model

At a first glance, deriving the phyical address of page v in address space σ seems to be rather complicated and expensive:

$$\sigma(v) = \begin{cases} \sigma'(v') & \text{if } \sigma_v = (\sigma', v') \\ r & \text{if } \sigma_v = r \\ \phi & \text{if } \sigma_v = \phi \end{cases}$$

Fortunately, a recursive evaluation of $\sigma(v)$ is never required. The three basic operations guarantee that the physical address of a virtual page will never change, except by flushing. For implementation, we therefore complement each σ by an additional table P, where P_v corresponds to σ_v and holds either the physical address of v or ϕ. Mapping and granting then include

$$P'_{v'} \;:=\; P_v$$

and each replacement $\sigma_v \leftarrow \phi$ invoked by a flush operation includes

$$P_v \;:=\; \phi \quad .$$

P_v can always be used instead of evaluating $\sigma(v)$. In fact, P is equivalent to a hardware page table. μ-kernel address spaces can be implemented straightforward by means of the hardware-address-translation facilities.

The recommended implementation of σ is to use one mapping tree per physical page frame which describes all actual mappings of the frame. Each node contains (P, v), where v is the according virtual page in the address space which is implemented by the page table P.

Assume that a grant-, map- or flush-operation deals with a page v in address space σ to which the page table P is associated. In a first step, the operation selects the according tree by P_v, the physical page. In the next step, it selects the node of the tree that contains (P, v). (This selection can be done by parsing the tree or in a single step, if P_v is extended by a link to the node.) Granting then simply replaces the values stored in the node and map creates a new child node for storing (P', v'). Flush lets the selected node unaffected but parses and erases the complete subtree, where $P'_v \;:=\; \phi$ is executed for each node (P', v') in the subtree.

References

Assenmacher, H., Breitbach, T., Buhler, P., Hübsch, V., and Schwarz, R. 1993. The Panda system architecture – a pico-kernel approach. In *4th Workshop on Future Trends of Distributed Computing Systems*, Lisboa, Portugal, pp. 470–476.

Bernabeu-Auban, J. M., Hutto, P. W., and Khalidi, Y. A. 1988. The architecture of the Ra kernel. Tech. Rep. GIT-ICS-87/35 (Jan.), Georgia Institute of Technology, Atlanta, GA.

Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. 1989. Lightweight remote procedure call. In *12th ACM Symposium on Operating System Principles (SOSP)*, Lichfield Park, AR, pp. 102–113.

Bershad, B. N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., and Sirer, E. G. 1994. Spin – an extensible microkernel for application-specific operating system services. In *6th SIGOPS European Workshop*, Schloß Dagstuhl, Germany, pp. 68–71.

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. xx–xx.

Brinch Hansen, P. 1970. The nucleus of a multiprogramming system. *Commun. ACM 13,* 4 (April), 238–241.

Bryce, C. and Muller, G. 1995. Matching micro-kernels to modern applications using fine-grained memory protection. In *IEEE Symposium on Parallel Distributed Systems*, San Antonio, TX.

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

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.

Cheriton, D. R. and Duda, K. J. 1994. A caching model of operating system kernel functionality. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, pp. 179–194.

Digital Equipment Corp. 1992. *DECChip 21064-AA Risc Microprocessor Data Sheet*. Digital Equipment Corp.

Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. 1991. Using continuations to implement thread management and communication in operating systems. In *13th ACM Symposium on Operating System Principles (SOSP)*, Pacific Grove, CA, pp. 122–136.

Engler, D., Kaashoek, M. F., and O'Toole, J. 1994. The operating system kernel as a secure programmable machine. In *6th SIGOPS European Workshop*, Schloß Dagstuhl, Germany, pp. 62–67.

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. xx–xx.

Ford, B. 1993. private communication.

Ford, B. and Lepreau, J. 1994. Evolving Mach 3.0 to a migrating thread model. In *Usenix Winter Conference*, CA, pp. 97–114.

Gasser, M., Goldstein, A., Kaufmann, C., and Lampson, B. 1989. The Digital distributed system security architecture. In *12th National Computer Security Conference (NIST/NCSC)*, Baltimore, pp. 305–319.

Hamilton, G. and Kougiouris, P. 1993. The Spring nucleus: A microkernel for objects. In *Summer Usenix Conference*, Cincinnati, OH, pp. 147–160.

Härtig, H., Kowalski, O., and Kühnhauser, W. 1993. The Birlix security architecture. *Journal of Computer Security 2,* 1, 5–21.

Hildebrand, D. 1992. An architectural overview of QNX. In *1st Usenix Workshop on Micro-kernels and Other Kernel Architectures*, Seattle, WA, pp. 113–126.

Hill, M. D. and Smith, A. J. 1989. Evaluating associativity in CPU caches. *IEEE Transactions on Computers 38,* 12 (Dec.), 1612–1630.

Intel Corp. 1990. *i486 Microprocessor Programmer's Reference Manual*. Intel Corp.

Intel Corp. 1993. *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*. Intel Corp.

Kane, G. and Heinrich, J. 1992. *MIPS Risc Architecture*. Prentice Hall.

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

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

Kühnhauser, W. E. 1995. A paradigm for user-defined security policies. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, Bad Neuenahr, Germany.

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

Liedtke, J. 1992. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, Kiel, pp. 294–305. Springer.

Liedtke, J. 1993. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, Asheville, NC, pp. 175–188.

Liedtke, J. 1995. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Arbeitspapiere der GMD No. 933 (Sept.), GMD — German National Research Center for Information Technology, Sankt Augustin.

Major, D., Minshall, G., and Powell, K. 1994. An overview of the NetWare operating system. In *Winter Usenix Conference*, San Francisco, CA.

Mogul, J. C. and Borg, A. 1991. The effect of context switches on cache performance. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Santa Clara, CA, pp. 75–84.

Motorola Inc. 1993. *PowerPC 601 RISC Microprocessor User's Manual*. Motorola Inc.

Nagle, D., Uhlig, R., Mudge, T., and Sechrest, S. 1994. Optimal allocation of on-chip memory for multiple-API operating systems. In *21th Annual International Symposium on Computer Architecture (ISCA)*, Chicago, IL, pp. 358–369.

Ousterhout, J. K. 1990. Why aren't operating systems getting faster as fast as hardware? In *Usenix Summer Conference*, Anaheim, CA, pp. 247–256.

Pu, C., Massalin, H., and Ioannidis, J. 1988. The Synthesis kernel. *Computing Systems 1,* 1 (Jan.), 11–32.

Romer, T. H., Lee, D. L., Bershad, B. N., and Chen, B. 1994. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, pp. 255–266.

Rozier, M., Abrossimov, A., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., and Neuhauser, W. 1988. Chorus distributed operating system. *Computing Systems 1,* 4, 305–370.

Schröder-Preikschat, W. 1994. *The Logical Design of Parallel Operating Systems*. Prentice Hall.

Schroeder, M. D. and Burroughs, M. 1989. Performance of the Firefly RPC. In *12th ACM Symposium on Operating System Principles (SOSP)*, Lichfield Park, AR, pp. 83–90.

van Renesse, R., van Staveren, H., and Tanenbaum, A. S. 1988. Performance of the world's fastest distributed operating system. *Operating Systems Review 22,* 4 (Oct.), 25–34.

Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. 1974. Hydra: The kernel of a multiprocessing operating system. *Commun. ACM 17,* 6 (July), 337–345.

Yokote, Y. 1993. Kernel-structuring for object-oriented operating systems: The Apertos approach. In *International Symposium on Object Technologies for Advanced Software*. Springer.