Improved Address-Space Switching on Pentium Processors by Transparently Multiplexing User Address Spaces

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

GMD Technical Report No. 933

November 1995

[*] GMD SET-RS, Schloß Birlinghoven, 53754 Sankt Augustin, Germany

Abstract

Address-space switch requires a TLB flush on many processors. With increasing TLB size, the secondary costs of address-space switching due to TLB refill can thus increase substantially. For the Pentium processor, we describe an optimization to avoid TLB flush in many cases. The method is transparent to the user and thus requires no extension of the μ-kernel interface.

Contents

1 Rationale 7

2 The x86 Memory Model 8

3 Address-Space Switch on 486 9

4 Address-Space Switch on Pentium 11

4.1 The IPC Path 12

4.2 Message Transfer To/From Small Spaces 15

4.3 Dynamic Association of Small Address-Spaces 15

5 First Results 15

1 Rationale

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, 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[1], 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

[1] 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.

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	TLB miss	Page Table	Segment
	entries	cycles	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*

2 The x86 Memory Model

The reader should be familiar with the segmentation model and the paging mechanisms of the x86-processor family. A detailed description can be found in the reference manuals of the 486 [Intel Corp. 1990] and Pentium processor [Intel Corp. 1993]. Segmentation and paging does not differ significantly on both processors.

3 Address-Space Switch on 486

L4 (as L3 [Liedtke 1993]) favours a flat memory model. On this processor, the virtual address space has a size of 4 GB. It consists of 3.5 GB of user space and 0.5 GB of kernel space, see Figure 1. Besides others, the kernel area contains the kernel code and the physical memory needed for page tables. Parts of the kernel area are shared between all address spaces. An address-space switch thus has no effect on these parts.

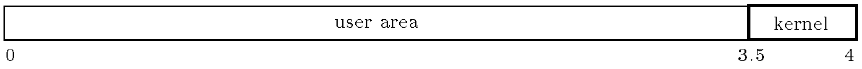

Figure 1: *486 Address Space.*

The user-accessible segment *flat space* describes the complete address space. User access to the kernel part is prohibited by means of page-level protection. Therefore, both kernel and user can use the same segment for data access.

Inter-process communication (IPC) is by far the most relevant reason for address-space switches. Note that hardware interrupts are also handled as IPC. Therefore, an interrupt usually also causes an address space switch through the IPC path. From the performance point of view, we have only to examine address-space switch by IPC.

The IPC path, respectively the parts which are relevant with respect to address-space switch, look like follows:

```
ipc system call (dest task) :
    if ds $\neq$ flat space then ds := flat space fi ;
      ⋮
    switch thread ;
    if dest task $\neq$ source task
        then  current space := space [dest task] ;
              flush TLB ;
    fi .
```

```
ipc_system_call:                                      cycles
              ⋮
        mov   ebx,ds                                   3
        cmp   ebx,flat_space                           1
        jnz   ipc_load_ds                              1
              ⋮
        mov   edi,[edx+proot_ptr]                      1

        switch_thread

        IFNZ [ebp+proot_ptr],edi                       3
             mov   edi,[edi]                           1
             mov   cr3,edi                             4 + 9n
        FI

        iretd
```

The first 3 instructions check, whether the segment register ds is loaded with the segment *flat_space* which describes the flat address space. Loading a segment register is relatively expensive on this processor; it costs 9 cycles if the segment descriptor is found in the data cache, 12 cycles in the case of a cache miss. Checking a segment register is twice as fast (5 cycles) as loading it.

Each thread control block (tcb) has a variable *proot_ptr* which holds a pointer into the *task_proot* array. This array holds per task the current page directory address, i.e. the physical address of respective task's page table root. At the end of the IPC path, the processor's current page table root register (cr3) is loaded with the new page table root, iff source (ebp) and destination tcb (edx) belong to different tasks, i.e. if their proot_ptrs point to different task_proot entries.

The macro *switch_thread* switches the stack pointer from the source thread's to the destination thread's kernel stack. Thus the concluding instruction 'iretd' returns to the receiving thread, popping its user-level stack and instruction pointer from its kernel stack.

In total, $14 + 9n$ cycles are required for address-space handling, where n is the number of TLB misses caused by the TLB flush due to reloading cr3. The minimal value for n is 4 (see [Liedtke 1993, pp. 188]), the maximum is 32. So the switch costs vary from 50 to 302 cycles, provided all TLB loads are handled without cache miss.

4 Address-Space Switch on Pentium

On the Pentium processor, the virtual address space has also a size of 4 GB. It consists of 3 GB of conventional user space, 0.5 GB for small user spaces and 0.5 GB of kernel space, see Figure 2. Tasks requiring only a small address space do not get the *flat user space* but an appropriate *small user space* inside the *small* region of the address space. The trick is to share the *small* region between all address spaces. Then switching from a large to a small user address space never requires a true address-space (page-table) switch and avoids flushing the TLB. The same holds when switching from a small to another small user space and when switching back from a small user space to the previously active large space.

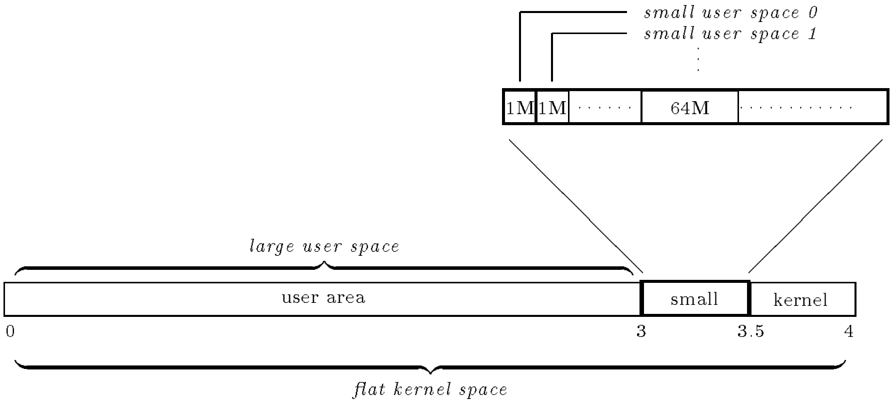

Figure 2: *Pentium Address Space.*

- Switching between large space is done conventionally by switching to a new page table tree. The only user accessible segment, *flat user space*, then describes the large user space region, its base is 0 its size 3M.

- Switching to a small user space is done by modifying the segment descriptor of *flat user space*; base and size are set to the corresponding small-user-space values.

Note that this is transparent to the user, since in all cases the same segment, *'flat user space'*, is used. Changing its base and size is more or less invisible[2] to user-level threads.

[2] There is a user-level instruction on this processor which delivers the size of a segment. By means of this, a user thread *can* find out whether it actually has a small or large segment.

4.1 The IPC Path

Now the IPC path, respectively the parts which are relevant with respect to address-space switch, look like follows:

```
ipc system call (dest task) :
    ds := flat kernel space ;
        ⋮
    switch thread ;
    if has large space (dest task)
        then  if dest task ≠ previous large task
                  then  current space := space [dest task] ;
                        flush TLB ;
                        previous large task := dest task
              fi ;
              gdt [flat user space].base := 0 ;
              gdt [flat user space].size := 3M
        else
              gdt [flat user space].base := small base [dest task] ;
              gdt [flat user space].size := small size [dest task]
    fi ;
    ds := flat user space ;
    es := flat user space ;
    fs := flat user space ;
    gs := flat user space .
```

Since the pages of both the large and the small region must be accessible by user threads (but running in different tasks), we can no longer rely on page-level protection to prevent a thread from accessing data outside its user space. Instead, we must use a user-accessible *flat-user-space* segment and a kernel-only-accessible segement, *flat kernel space*.

When entering the kernel, a segment register must be loaded with *flat-kernel-space*. When leaving the kernel, the user segment must be reloaded. Since in general, the base address and the size of the flat user segment have changed due to a logical address-space switch, all size and base values in all segment registers must be updated. This is done by reloading gs, fs, es, ds explicitly with the flat-user-space segment. The two segment registers cs and ss are automatically updated by the 'iretd'instruction. The corresponding machine instructions are:

```
ipc_system_call:                              cycles
            ⋮
       mov  ebx,flat_kernel_space                 1
       mov  ds,ebx                                3
            ⋮
       mov  edi,[edx+proot_ptr]                   1

       switch_thread

       mov  eax,[edi]                             1
       IFB  eax,size physical_ptab_memory         3
            IFNZ [current_cr3],eax                  4
                 mov   [current_cr3],eax               1
                 mov   cr3,eax                        21 + 9n
            FI
            mov   eax,0x00CB00FF [a]                4
       FI
       mov  [gpt_flat_user_descr+1],ah            1
       mov  ah,0xF3 [b]                           0
       mov  [gpt_flat_user_descr+4],eax           1

       mov  eax,flat_user_space                   0
       mov  ds,eax                                3
       mov  es,eax                                3
       mov  fs,eax                                3
       mov  gs,eax                                3

       iretd
```

[a] This value ensures that the gdt descriptor is set to a base address of 0 and a size of 3M, see [Intel Corp. 1993].

[b] These three instructions update the gdt descriptor of the flat-user-space segment. For understanding its semantic, look at the description of data-segment descriptors in the Pentium reference manual [Intel Corp. 1993].

Table 2 shows the address-space-switch costs, depending on the type of switch. Switches to a small space and "back" to the previous large space are cheap. Pseudo switches to the same address space are also optimized so that thread switching within an address space need no special optimization. Table 3 shows the switch costs for the typical RPCs.

type	address-space switch $S \rightarrow D$	previous large space	[cycles]	
LL	large → large	$\neq D$	$95 \ldots 914$	$(50 + 9n)$
LP	large → large	$= D$	28	
LS	large → small		23	
SS	small → small		23	
SP	small → large	$= D$	28	
SL	small → large	$\neq D$	$95 \ldots 914$	$(50 + 9n)$

Table 2: *Address-Space Switch Costs.*

type	Client ↔ Server		[cycles]
LS + SP	large ↔ small		51
SS + SS	small ↔ small		46
SL + LS	small ↔ large	server ≠ previous	118...937
SP + LS		server = previous	51
LL + LL	large ↔ large	inter-task	190...1828
LP + LP		intra-task	56

Table 3: *Typical RPCs, Address-Space Switch Costs.*

4.2 Message Transfer To/From Small Spaces

The general method to copy messages cross address spaces (without additionally copying them into kernel space) is described in [Liedtke 1993]. This method is based on temporary mapping in the kernel area. Small user spaces offer an even better solution.

As Figure 3 shows, a message can be copied into or out of a small user space by the kernel without any additional mapping, because in kernel space, both user spaces occupy different address regions and are simultaneously accessible. (Recall that all small user spaces are shared between all large address spaces.) Of course, the same method can be used between two small user spaces.

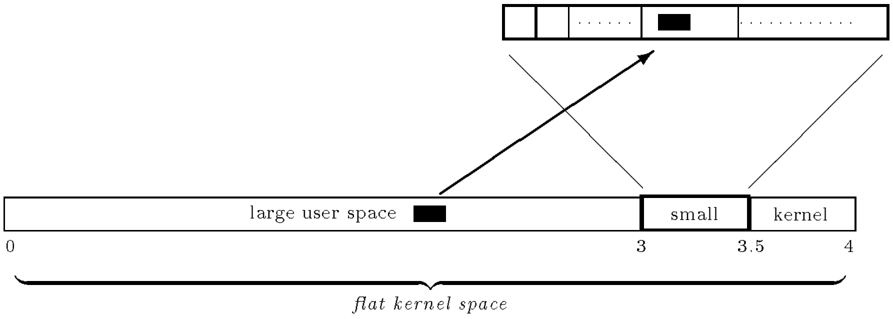

Figure 3: *Message Transfer To/From Small User Space.*

4.3 Dynamic Association of Small Address-Spaces

The system dynamically decides whether a user address space is implemented as a large or a small user space. If a task has no objects mapped to the upper part of its address space and an appropriate free small user space is available, the small space is associated to the task. When the task tries to map objects outside this small area, its user space is automatically converted to a large one. If the system ran out of small user spaces, it tries to convert sleeping "small" tasks to large ones.

These operations are transparent to user-level.

5 First Results

The effect of using segment based address-space switch on Pentium is shown in figure 4. 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.

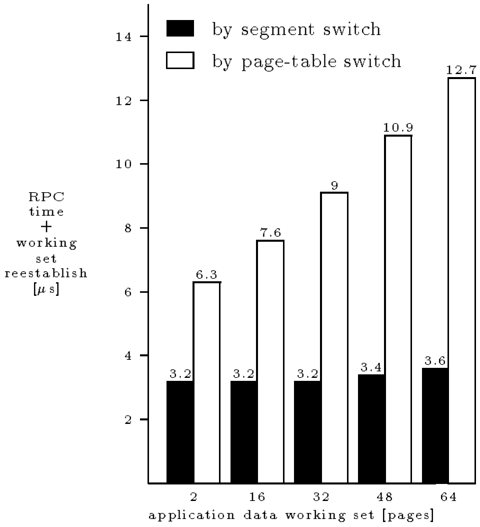

Figure 4: *Segmented Versus Standard Address-Space Switch in L4 on Pentium, 90 MHz.*

References

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

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.

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

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