

## Re: Allocating kernel memory

**Source:** <http://linux.derkeiler.com/Newsgroups/comp.os.linux.development.system/2004-05/0358.html>

---

**From:** Kasper Dupont (*kasperd\_at\_daimi.au.dk*)

**Date:** 05/16/04

Date: Sun, 16 May 2004 09:25:50 +0200

George Nelson wrote:

>

> *Kasper Dupont <kasperd@daimi.au.dk> wrote in message news:<40A6236B.4DCBF5EC@daimi.au.dk>...*

>>

>> *But who is to blame? Using 1% of your RAM for management data seems fair to me. And I know a lot of people complain about the 3GB for user space being too restrictive. And the cost of changing page tables on each user/kernel switch is too high.*

>>

>

> *I never complained about the real memory that was used but pointed out that a large portion of the kernel virtual address space is used up.*

Exactly my point. On a system with the virtual address space as large as the physical, Linux would have at least 25% of the physical memory mapped, and using 1% of that would still leave 24%

>

> *Also read section 3.7.3, Mixing 4KB and 4MB byte pages.*

Where do I find that? The link you posted earlier doesn't work anymore.

>

> *I'm afraid you are wrong. Paging was introduced primarily to resolve a memory fragmentation problem. I do not disagree with the other benefits that have been made possible from paging but its basic intent was to resolve the issue in early OS's of allocating space in a real address memory system. By treating memory in pages and having H/W assist to translate a virtual address to a physical address resolved the memory fragmentation problem which meant either a roll-out, roll-in of processes or a memory move when holes appeared in the physical memory space. It was quickly seen that paging allowed programs to have a larger virtual address space than available real*

- > *memory, the possibility of shared memory and a few additional safety*
- > *features (e.g not mapping the page at virtual address 0).*

But those are all things which happened before the first version of Linux was designed. And it doesn't really matter which came first, the point is all the advantages of paging are more important than the minor performance hit.

- >
- >> *The performance hit cause by paging shouldn't be*
- >> *much. If the page is in TLB the access should be*
- >> *as fast as it would have been if paging was not*
- >> *enabled. So TLB misses is the only cost.*
- >>
- >
- > *No. It still takes some time to perform the TLB lookup even though*
- > *that is significantly less than accessing the page table in memory.*

As long as it can be done within the same clockcycle it doesn't matter how long time it takes, it would still not affect the end result in any way. I don't have any measurements, but I don't believe the CPU designers would allow a TLB lookup to slow down the memory access. The memory bandwidth really should be the bottleneck.

- > *Disable paging at that will remove this. Every instruction and data*
- > *access will incur the TLB lookup penalty, small though it is.*

Actually AFAIK the TLB is between L1-cache and L2-cache, and with TLB in associative memory, the answer should come in the same clockcycle as it would have without.

- > *If paging does not invoke a*
- > *performance hit why is it that Cray supercomputers are real memory*
- > *systems?*

Maybe they can push the clock frequency a little higher that way. And they save space on the that can be used for more cache or something like that. They also have less work to do when designing the CPU.

Comparing a CPU without paging support and CPU with paging support is definitely not the same as comparing an OS with and without paging support running on a CPU that does have the support.

- > *All I have done is express surprise at the 1GB address space limit.*

The reasons for that design decision should have been made clear by now.

- >
- > *But 32-bit architectures will be around for a long time yet and real*
- > *memory will increase. Commercial users will want to get the best out*
- > *of their current investment and are more likely to add memory, disk to*
- > *an existing (32-bit system) than replace them.*

Sure, you can add more memory to your systems. But the problem really only applies to PAE systems, where you can add a lot more than 4GB of RAM. How many of them are there "out there"? Those 8GB systems I could find were using more than 50% of the zone 1 memory for stuff that could have been on zone 2. So you shouldn't experience any problem on an 16GB system either.

Deploying new systems with those problems today seems like a bad idea to me. If you need more than 4GB of RAM you should avoid PAE, AMD64 is a better design.

So we are only talking about systems with PAE and motherboards that support more than 16GB of RAM. I don't think those systems are so common, that they justify a significant design change. Besides there is a 4:4 split patch which you can use if you have one of those systems and are prepared to take the performance hit.

- > *And changing*
- > *architectures is only a temporary measure.*

Maybe. But problems would be like 20 years away if systems continue to grow at the current pace. And who knows what is going to happen the next 20 years?

- > *Now we are*
- > *at the stage where the opposite is the case, real memory systems*
- > *larger than the address space are economically viable.*

That has happened many times before. What this really show is people not wanting to replace an obsolete architecture. 64 bit systems should hav been the standard for many years.

When a 16 bit address space became too small, hardware with bankswitching was implemented as a workaround. When the 20 bit address space of 8086 became too small EMS did basically the same to work around the limit. When 32 bit address space became too small, PAE was introduced. All workarounds instead of a fix, i.e. a

better architecture.

And hopefully if/when we reach the limitations of 64 bit there will be implemented 128 bit systems instead of stupid workarounds like PAE, bankswitching, etc. But obviously I'm too optimistic.

- > *I see no reason*
- > *why this trend will change and the similar problems will arise at some*
- > *point in the future for 64-bit systems.*

Nobody knows. There are people suggesting, that the current trend cannot continue for another 20 years.

- >
- > *A simpler, more elegant kernel perhaps?*

We had that. But it was limited to 1GB of physical RAM. (And 2GB of physical RAM with a 2:2 split). The demand for more RAM lead to the current design. Actually I recall a comparison between Linux and Windows which was done for Microsoft many years ago. It was at a time when 32MB of RAM was a luxury. Microsoft decided the comparison were to be done on a computer with 4GB of RAM because they knew Linux would not use all of it.

With PAE the simple elegant design is not possible. There is no way you can put 64GB of RAM into a 4GB address space. A new design was required. And since we had a design that would allow the kernel address space to be smaller than the physical RAM, the 1GB extra that was in some cases taken for the kernel could be reclaimed for user space.

- > *With a 1GB address space on a*
- > *64GB memory system more than likely used as a server, the demands*
- > *placed on kernel memory resources for memory will be great (large*
- > *number of user processes each with their kernel resident data*
- > *structures, internal structures to handle memory managment, I/O,*
- > *buffer caches and so on).*

I just took a look on how much memory was used for those purposes on two 8GB systems. The memory reserved for kernel code and the mem\_map array is about 128MB. In addition the structures you are talking about are all allocated from the slabs, and are typically the only slab users with any significant amount of allocations. The slabs on those two systems used respectively 135 and 270 MB of RAM. So while they might not scale to a 64GB physical RAM system, there is certainly no problem with having only one fourth of your physical RAM in address space, as will be the case on a better architecture.

BTW the kernel data required for a process is AFAIK about 16KB all included, and work is being done on reducing that. Recent kernels reduced the stack from 8KB to 4KB. That means 1000 processes (which is a pretty high number) would still only require 16MB of RAM.

> *This in turn will cause the kernel to spend  
> more time managing memory reducing performance*

I don't see why.

> *and limiting the scalability of the system*

Not if you compile the kernel for a reasonable architecture.

> *Also, if there is no use for a  
> kernel with a larger address space then why are there kernel versions  
> with a 2:2 split*

That was before high memory was introduced. At that time the kernel could not use the memory at all if it did not fit into the kernel address space. So a system with 1GB of physical RAM could not use all of the physical RAM unless you used the 2:2 split.

With the introduction of high memory the 2:2 split was removed because you no longer needed a large kernel address space.

> *and I believe there is also a patch for a 4GB kernel  
> address space?*

That is because on a PAE system with 64GB of physical RAM the 1GB kernel address space is really too small. That means the 4:4 patch is really a workaround for a lousy architecture.

> *I guess I am not alone.*

You have 2GB of physical RAM. There is one hell of a difference between 2GB and 64GB. Nobody in their right minds would use the 4:4 patch for such a small system.

>  
> > *The existing solution sucks and only helps when any user  
> > process is transferring data in/out of the kernel  
> >  
> > That actually happens a lot. But no, you are not even*

- > > *right about that. Every kernel/user switch would be*
- > > *slowed down by a need to switch page tables. That means*
- > > *you'd have to pay the price on every single exception,*
- > > *system call, interrupt, and so on.*
- > >
- >
- > *How long does it take to load a segment selector?*

No you would need to switch page tables, that is the expensive part.

- > *One segment selector selects*
- > *the kernel page table the other selects the user process page table.*
- > *Two separate 4GB address spaces, little overhead, no TLB flushing and*

Are you talking about the same architecture as the 4:4 split is written for. According to this posting from the kernel mailing list, the price is real:  
<http://www.google.com/groups?selm=1tXIA-6SM-1%40gated-at.bofh.it>

If you know about features in the architecture that the kernel developers does not know about, I think you should tell them about it. Post your ideas to the kernel mailing list and get praised (or hammered down because you don't know what you are talking about).

- >
- > > > *- as soon as a task*
- > > > *switch takes place the TLB's are flushed anyway.*
- > >
- > > *Right, but they are rare. The kernel does a great work*
- > > *to make them as rare as by any means possible.*
- > >
- >
- > *Only every time a process is scheduled to run. Not sure what the*
- > *process time quantum is under Linux (about 1/5th of a second I think).*
- > *So at least every quantum interval there is likely to be a TLB flush*
- > *in a multi-process environment (since this will cause a process switch*
- > *and a change in page tables for the new process).*

And 1/5th of a second is rare compared to the thousands of interrupts per second. And tens of thousands (if not millions) of system calls per second.

In a multiprocessor system with fewer CPU bound processes than CPUs, it should be possible for those to run for multiple seconds without a TLB flush.

comp.os.linux.development.system: Re: Allocating kernel memory

You can switch from a user mode process to a kernel thread and back again without a TLB flush. And between use processes (threads) sharing their entire address space.

--

Kasper Dupont -- der bruger for meget tid paa usenet.  
For sending spam use abuse@mk.lir.dk and kasperd@mk.lir.dk  
I'd rather be a hammer than a nail.