

Re: ten thousand small processes

Source: <http://unix.derkeiler.com/Mailing-Lists/FreeBSD/performance/2003-06/0090.html>

From: Terry Lambert (tlambert2_at_mindspring.com)

Date: 06/25/03

Date: Wed, 25 Jun 2003 02:26:02 -0700

To: "D. J. Bernstein" <djb@cr.jp.to>

"D. J. Bernstein" wrote:

- > *As I said, I don't particularly care about the text segment. I'm not*
- > *talking about ten thousand separate programs.*
- >
- > *Why does the memory manager keep the stack separate from data?*

Stack needs to be executable for the current signal trampoline code. FreeBSD might be able to adopt the OpenBSD "non-executable stack" code to let it do this, too, but that would really depend on the interaction with the various threads implementations, all of which have signal code that's likely to be unhappy with it.

- > *Suppose a*
- > *program has 1000 bytes of data+bss. You could organize VM as follows:*
- >
- > *0x7fffac18 0x7ffb000 0x80000000*
- > *<----- stack data+bss text, say 5 pages heap ----->*
- >
- > *As long as the stack doesn't chew up more than 3096 bytes and the heap*
- > *isn't used, there's just one page per process.*

The pages have to be on page boundaries. The absolute minimum number of pages you could have is 2, assuming you take your approach to putting the load address up high.

In general, however, realize that the total virtual address space consists of (KVA + UVA = 4G). The model you propose, of throwing the heap out at 0x80000000 implies that the KVA and UVA are separate.

While this is possible, in theory, in practice it's a bad idea, since the uiomove/copyin/copyout code depends on the ability to run in system space and map the user address at its expected location (in fact, it is mapped there by default on kernel entry, to avoid having to establish mappings for data copies that cross protection domains).

freebsd-performance: Re: ten thousand small processes

Breaking this would have some serious performance consequences, in terms of needing to create wired mapping for user pages in kernel space when copying data in and out — and having to do explicit address translations, which would significantly damage a number of aspects of performance.

So instead, the stack and heap grow towards each other, and the stack does not live in a data page.

In general, the use of `mmap()` that is causing your primary page-count problem is tunable, via the `mmap symlinks` flags hack, to get the total count down.

However, the counts is probably never going to go below 4 pages, if there is any heap memory in use at all, no matter what you do, and 5 or more, if you count page table entries, etc., against the process.

FWIW, most of these pages are phantoms — that is, they exist as page mappings with no backing pages, and backing pages are allocated on fault at reference time.

*> As for page tables: Instead of allocating space for a bunch of nearly
> identical page tables, why not overlap page tables, with the changes
> copied on a process switch?*

Each page table descriptor is capable of handling 4M of memory, either directly, as a 4M page mapping (not used for user processes, except for `mmap`'ed device memory with proper size and alignment, if it goes out of its way to establish the mapping). The typical mapping in a PTD pointing to a 4K page containing PTE's.

For the most part, overlap doesn't buy you anything in this case, because most programs are linked shared, which means they almost instantaneously have a delta on pages between the two processes, even if they were forked identically, as a result of the page (set) that contains the glue for the dynamic object references.

Even if you were talking a statically linked program, then you will need to worry about data pages that are copy-on-write written as soon as you start doing stack and heap accesses, so the page mappings are not shareable. The closest you will come to shareable page mappings is going to be `rfork()` (where they *are* shared, but are not COW, so it does you no good in your suggested case), or the case that the data space is very, very much larger than 4M, at which point it *may* make sense to make them COW.

I'm pretty sure this would not work at all for the 386, which does not support taking write faults on pages that are not marked writeable, when the processor is in system mode — this is a bug in the 386 page protection implementation, which has to be worked around for security

freebsd-performance: Re: ten thousand small processes

reasons by hacking the page mapping and taking a page not present fault, and then fixing it up based on apriori knowledge of the page mapping; failure to do this extra work would mean that someone could read or write arbitrary kernel addresses with arbitrary data, thus escalating priviledges by overwriting credentials.

Even throwing out the 386, I'm pretty sure that the fault on a page that contains page mappings is not defined for some circumstances, at least from my reading of the IA32 architecture manual. So to deal with this, you would probably need to move to explicit TLB shootdowns for most cases, and, again, you are screwed in the shared mappings department.

For the most part, this won't work anyway, since most x86 UNIX systems don't use a separate GDT entry per process, but instead have one for the recursive mapping, one for the kernel, one for text, and one for data. Occasionally, they also support one per VM86() instance.

Changing this would have the negative effect of limiting the total number of processes you could run simultaneously (see the Linux mailing list archives for why they switched away from using the TSS in order to do context switching: all TSS descriptors must reside in the GDT, which has only 8192 entries).

It's not worth accepting this limit, in order to obtain page table sharing for a minority of tasks, since you seem to be interested in small tasks which could not benefit from doing this anyway.

I really reccomend:

Protected Mode Software Architecture
Tim Shanley
MindShare, Inc.
Addison-Wesley Publishing Company
ISBN: 0-201-55447-X

Specifically:

Chapter 8: Code Segments
Chapter 9: Data and Stack Segments
Chapter 11: Mechanics of a Task Switch
Chapter 13: Virtual Paging

> *As for 39 pages of VM, mostly stack: Can the system actually allocate
> 390000 pages of VM? I'm only mildly concerned with the memory-management
> time; what bothers me is the loss of valuable address space. I hope that
> this 128-kilobyte stack carelessness doesn't reflect a general policy of
> dishonest VM allocation ("overcommitment"); I need to be able to
> preallocate memory with proper error detection, so that I can guarantee
> the success of subsequent operations.*

Re: ten thousand small processes

freebsd–performance: Re: ten thousand small processes

It is, in fact, memory overcommit. It would take some work to disable this. BSD systems have used memory overcommit ever since they adopted the Mach VM architecture.

> *As for malloc()'s careless use of memory: Is it really asking so much
> that a single malloc(1) not be expanded by a factor of 16384?*

You need to ask malloc() for this behaviour explicitly; it's not implicit in the default system configuration.

The primary reason for this is that most programmers who have been trained in the last two decades believe that "All the world's a VAX"; e.g.: "memory is free" and "Intel Byte order is right and network byte order is wrong", and other untruths.

Given this reality, most programs these days are what a lot of us who were trained up when 32K was an **amazing** amount of memory would politely call "bloated pigs". It makes sense, in that case, to be aggressive in allocation of memory, in the expectation that programs will tend to be memory–unconscious enough that they will tend to use everything you preallocate on their behalf.

[... default malloc behaviour complaints that are controllable
via flags settings ...]

> *(Quite a few of my programs simulate this effect by checking for space
> in a bss array, typically 2K. But setting aside the right amount of
> space would mean compiling, inspecting the brk alignment, and
> recompiling. I also feel bad chewing up space on systems where malloc()
> actually knows what it's doing.)*

I defy you to name one modern UNIX variant that doesn't overcommit memory. I doubt there is a malloc() existant on these systems that "actually knows what its doing".

BTW: I would class your suggestion of COW page table sharing as a type of memory overcommit, as well.

If you can find a counterexample, the logical thing to do is to port it and carry it around with your code, if you need that type of behaviour, since most libc malloc() implementations are defined to use weak symbols, to permit you to override their malloc/free implementation completely. So long as your implementation conforms to POSIX, everything else (strsave(3), et. al.) should "just work" with the replacement.

> *As for the safety of writing code that makes malloc() fail horribly:
> After the Solaris treatment of BSD sockets, and the ``look, Ma, I can
> make an only–slightly–broken imitation of poll() using select()!''
> epidemic, I don't trust OS distributors to reserve syscall names for
> actual syscalls. I encounter more than enough portability problems*

Re: ten thousand small processes

freebsd-performance: Re: ten thousand small processes

> *without going out of my way to look for them.*

Malloc isn't, and never has been, a system call on POSIX/UNIX systems. It's **always** been in section 3, for long as it's been an accepted part of the OS. Only things whose manual pages are in section 2 are system calls, and that's not a guarantee that they will stay there: basically, it's legal to implement all of POSIX on top of asynchronous system calls with a wait primitive, and provide all of the "expected" POSIX semantics in nothing more than a user space library.

-- Terry

freebsd-performance@freebsd.org mailing list

<http://lists.freebsd.org/mailman/listinfo/freebsd-performance>

To unsubscribe, send any mail to "freebsd-performance-unsubscribe@freebsd.org"