

Re: Maybe we should stop "Paging Beth Stone" already...

Re: Maybe we should stop "Paging Beth Stone" already...

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2005-09/msg00888.html>

- *From:* "Richard Cooper" <spamandviruses@xxxxxxxxxxxxxxxxxxxx>
 - *Date:* Fri, 16 Sep 2005 20:04:34 GMT
-

On Fri, 16 Sep 2005 02:30:59 -0400, wolfgang kern <nowhere@xxxxxxxxxxxxxxxx> wrote:

```
| Yes, but the problem I was looking at was how do I get NASM to work  
| in my new OS without porting over GCC?
```

How about just rewrite the assemblers 'linker'-part?

Still leaves me with needing to compile NASM to run on my OS. I'd have to modify GCC so that it creates programs that work in my OS. I'll want to work on my OS while running my OS, so the assembler that it's written with has to run under it.

```
| You'd have to do 10,000 of them a second to take up 1% of  
| your CPU time.  
| If you do 1ms timeslices, that's 2000, which leaves another  
| 8000 for  
| an interrupt every 0.25ms.
```

I really do use IRQ0 driven 1mS time-slices.

Now I'd set the real time clock to one second intervals for real clock time, then I'd use the PIT in one shot mode, so that the scheduler simply sets it to go off at the time when it needs to switch tasks again. That way if you're running two tasks that both need the CPU a thousand times a second, it can program the PIT to a short 0.5 ms interval, but if you're only running a single program and everything else is currently blocked, then it doesn't have to set the PIT at all, since it won't need to pre-empt the program until either another IRQ occurs (like a network connection) or the program itself requests that something else run by writing to a IPC connection or something. That makes a lot more sense than the way things are usually done, using a single interval period and trying to make a compromise between scheduling latency and resource

Re: Maybe we should stop "Paging Beth Stone" already...

Re: Maybe we should stop "Paging Beth Stone" already...

consumption.

```
| If you don't use TSS, you've still got to swap all of those registers  
| manually, load a new CR3, and start making use of those segment  
| protections (call gates et. al.) since it's no longer possible to give  
| applications an address space that's totally isolated from the kernel.  
| So how much processor time can you really save while still maintaining  
| memory protection?
```

Save/restore/swap only what's needed ...

There isn't much in the TSS that you don't have to swap. There's the LDT Segment Selector, you probably wouldn't even use that, so you could not swap it, but that's all you can leave out.

You have to swap the program registers, because the programs are going to be using them. You have to swap CR3 if you want separate memory spaces. You have to swap the selectors, because it is possible to load them with anything and not cause a fault (the fault doesn't occur until you use them, and even if it's CS we're talking about, it's unlikely, but maybe possible to load CS with a bad value, and have an interrupt occur before the processor fetches the next instruction and causes a fault). The alternate stacks aren't used by the processor unless the task calls a different protection level, so they're not part of the TSS swap. As for the Previous Task Link, it's only one word, and one which you'll probably replace with a word of your own to tell you what you're doing.

So it's almost completely all necessary stuff in there, and I don't see how anyone could do it faster than the CPU does it itself without skimping out on their level of protection. I assume Intel knows that people want this to happen quickly, so I doubt any of those clock cycles are just wasted.

Now I'm not sure why that takes 314 clock cycles. The CPU has to do two of them at a time (save to one, load from the other), and so it's 157 each, for what's laid out as 20 dwords of information, which is almost 8 clock cycles each. I assume that the rest is lost in extra details involved in loading a new page table and then looking up the bases and limits of all of the selectors. Even with all of that, I can't account for where they are all going, but I'm sure that Intel puts as much effort into optimizing the TSS swap as it does into optimizing everything else, so if it could be done in 50 clock cycles, then I'm sure that's all the longer it would take.

```
I think 'protection' has become a very paranoid used item ..  
Total isolation may not be what we really want.
```

Re: Maybe we should stop "Paging Beth Stone" already...

I don't know about you, but I like it when I write a program and it crashes, and I don't have to reboot my computer because it overwrote part of the OS.

Now if you're writing an embedded system they you may well not want protection, but for a general use system, where my computer is busy doing other things of varying importance while at the same time I'm writing programs that are crashing left and right, I want some protection in there. It's a nice thing to have.

The OS itself (the instance which allow user-code execution) must be able to access all user memory at every time.

As it can. The OS has every page of memory mapped into it's page tables, so it can see everything. When it switches to an application task via a TSS, a new page table is loaded which only contains pages belonging to the application.

Why waste memory with a new set of page tables for every task?

Well, in Linux, each program gets it's own address space, so the program always loads at about 0x80000000, and the top of the stack is always around 0xC0000000. This lets any application use up to a gigabyte of stack before Linux is forced to tell it that it's gone too far. (Linux may want to kill it sooner, but at least it's not forced to kill it because there's simply no more address space left between the stack and the program.)

Naturally not every process can use a 1GB stack because there isn't enough RAM for that, but when you start up a process you don't know and even the process may not know how much stack it's going to need. So the safe thing to do is to leave a lot of address space in there so that the stack can potentially grow to an enormous size, but only allocate memory to it when it's actually needed.

If you run every application in the same page table, then you've got to share that address space between applications. So even if you're only supporting 1024 simultaneous applications, that still divides 4GB of address space down to 4MB. 4MB isn't very much address space.

You also can't just start out giving each program's stack a small amount of address space, because you can't just move the program's stack whenever you want. Even if you adjust ESP for the program, you don't know how many pointers it's stored in one place or another to some data that's on it's stack, so if you move the stack to a new part of the address space where there's more room for it to grow, you break the application.

Re: Maybe we should stop "Paging Beth Stone" already...

And it may often improve performance if two user-threads can communicate by shared memory rather than detour over the OS.

That's why OSs support shared memory. You tell the OS you want shared memory, and it puts the same page of physical memory into the page tables of each process.

This is also how shared libraries are done. You tell the OS to map an executable somewhere into memory (read-only of course), and no matter how many processes load the shared library, only one copy of it exists in memory. Part of every process' pages tables point to that same piece of memory.

An added benefit is that each application can have this shared memory at a different address, so you don't have to choose an address that all applications that want access to it have free, and in fact usually each application is allowed to tell the OS at what address it wants the shared memory to be.

Do we really need to lock out I/O instructions from user space
?

At least by default. No reason you can't enable them with the proper permissions. But if a program crashes and starts executing random data, it could theoretically do anything. Say the code is a filesystem driver, so it's got buffered data from the disk in it's memory space. One of the files buffered there is an IDE driver. Then it screws up it's stack and returns to an address that puts it right in front of the port access code in that IDE driver, and it proceeds to write a random sector to the disk. Even though you've done the right thing and made the filesystem driver separate from the IDE driver, in this case it wasn't enough because the port access was still there.

Sure, when all the offered protection is required (which I doubt), then TSS-switching may be more effective than discrete switching.

Like I've said, I wouldn't want my OS to not have all of that protection. For an embedded system I wouldn't use protection, but for a desktop OS, I want it.

| Even if we all agreed on the design of the OS, I doubt we would all
| agree on the assembler to use.

Re: Maybe we should stop "Paging Beth Stone" already...

Now this would be a lesser problem, I'd say use all which support the new API :)

Do you mean like none of them, and instead just write a new one?

Using every assembler would definately be a mess, but maybe, just maybe, everyone could agree on the design of a new assembler. I don't know, though. Betov seems to be quite against using semicolons to seperate instructions, and I'm quite against using pipe characters, so I don't think that would get too far. A few more things like that ("dest, src" vs. "src, dest") and you've got everyone wanting a different assembler.

On the other hand, all I can think of is '|' vs. ';' and "dest, src" vs. "src, dest", so maybe a signifigant number of us could agree on something.

.