Context Switching

From OSDev Wiki

In your average, memory-protected environment, a "context" is a virtual address space, the executable contained in it, its data etc.

A "context switch" occurs for a variety of reasons - because a kernel function has been called, the application has been preempted, or because it had yielded its time slice.

A context switch involves storing the old state and retrieving the new state. The actual information stored and retrieved may include EIP, the general registers, the segment registers, CR3 (and the paging structures), FPU/MMX registers, SSE registers and other things. Because a context switch can involve changing a large amount data it can be the one most costly operation in an operating system.

There are many ways of performing a context switch. The x86 CPU provides a way of doing it completely in hardware, but for performance and portability reasons most modern OS's do context switches in software.

Contents

Software Context Switching

Software context switching can be used on all CPUs, and can be used to save and reload only the state that needs to be changed. The basic idea is to provide a function that saves the current stack pointer (ESP) and reloads a new stack pointer (SS:ESP). When the function is called EIP would be stored on the old stack and a new EIP would be popped off the new stack when the function returns. Of course the operating system usually needs to change much more than just the stack and EIP.

Eflags, the general registers and any data segment registers should also be pushed on the old stack and popped off the new stack. If the paging structures need to be changed, CR3 will also need to be reloaded.

The FPU/MMX and SSE state could be saved and reloaded, but the CPU can also be tricked into generating an exception the first time that an FPU/MMX or SSE instruction is used by copying the hardware context switch mechanism (setting the TS flag in CR0).

Details

When the CPU changes to a higher privilege level (CPL 0 being the highest) it will load new values for SS and ESP from the Task State Segment (TSS). **If the operating system uses multiple privilege levels it must create and load a TSS**. An interrupt generated while the processor is in ring 3 will switch the stack to the resulting permission level stack entry in the TSS. During a software context switch the values for SS0:ESP0 (and possibly SS1:ESP1 or SS2:ESP2) will need to be set in the TSS. If the processor is operating in Long Mode, the stack selectors are no longer present and the RSP0-2 fields are used to provide the destination stack address.

If a context switch also entails a change in IO port permissions, a different TSS may be loaded for each Process. When running virtual 8086 tasks, the IO permission map in the TSS isn't checked to provide I/O port protection. IO protection can be implemented by setting the IO Permission Level to 0. This will generate a General Protection Fault when a process in ring 3 attempts to write to or read from an IO port. The GP fault handler can then check permissions and carry out the port IO on behalf of the user-mode code.

Other Possibilities

During a context switch the operating system can do additional work that isn't strictly part of the context switch. One common thing is calculating the amount of time the last thread/task/process used so that software (and the end user) can determine where all the CPU time is going. Another possibility would be dynamically changing thread/task/process priorities.

Performance Considerations

Translating a virtual address to a physical address is expensive. The processor must access the pages table structures, which usually have 3-4 levels. Thus, a single memory access actually requires 4-5 memory accesses.

To mitigate this issue, most modern processors cache virtual-to-physical translations in a translation lookaside buffer (TLB). The TLB is part of the MMU and is (mostly) transparent to the system developer and users.

When virtual memory is updated -- for instance, when one process's address space is replaced with another's during a software context switch -- the TLB suddenly contains "stale" translations that are no longer valid. These translations must be flushed for correct behavior. Writing to CR3 will flush the TLB. However, by writing to CR3, you also eliminate all translations for the kernel, in addition to the last user process. This is less than ideal, as the next few operations must wait for the slow virtual-to-physical translations.

Recent Intel and AMD processors sport a tagged TLB, which allow you to tag a given translation with a certain address space configuration. In this scheme TLB entries never get "stale", and thus there is no need to flush the TLB.

Hardware Context Switching

Some CPU's have a special mechanism to perform context switches in hardware. The following information gives details on 80x86 CPU's only.

The hardware context switching mechanism (called Hardware Task Switching in the CPU manuals) can be used to change all of the CPU's state except for the FPU/MMX and SSE state. To use the hardware mechanism you need to tell the CPU where to save the existing CPU state, and where to load the new

CPU state. The CPU state is always stored in a special data structure called a TSS (Task State Segment).

To trigger a context switch and tell the CPU where to load it's new state from the far version of CALL and JMP instructions are used. The offset given is ignored, and the segment is used to refer to a "TSS Descriptor" in the GDT. The TSS descriptor is used to specify the base address and limit of the TSS to be used to load the new CPU state from.

The CPU has a register called the "TR" (or Task Register) which tells which TSS will receive the old CPU state. When the TR register is loaded with an "LDTR" instruction the CPU looks at the GDT entry (specified with LDTR) and loads the visible part of TR with the GDT entry, and the hidden part with the base and limit of the GDT entry. When the CPU state is saved the hidden part of TR is used.

A step further with Hardware Switches ...

In addition to the CALL and JMP instructions, a context switch can be triggered by a using a Task-Gate Descriptor. Unlike TSS Descriptors, task-gate descriptors can be in the GDT, LDT or IDT. Normally, task-gate descriptors are used in the IDT, so that an exception (or IRQ) can cause a context switch, which is the only way of handling a double fault exception with complete reliability.

The design of the basic hardware mechanism is limited by the number of usable entries in the GDT because TSS descriptors can be in the GDT only (theoretical limit is 8190 tasks). However, it is possible to avoid this restriction by dynamically changing TSS descriptor/s, by setting the TSS descriptor's base before each context switch. Care must be taken when using this approach when task-gate descriptors in the IDT are also used (the TSS descriptors referred to by each task-gate descriptor would have to be constant). Also context switches can't be initiated with a CALL instruction, because the CPU saves the GDT entry to use for the return in the TSS's "backlink" field.

If the FPU/MMX and SSE state also needs to be changed during a context switch there are a few options. The data could be explicitly saved by any code that causes a context switch, or the CPU can generate an exception the first time an FPU/MMX or SSE instruction is used. With the second option, the exception handlers would save the old FPU/MMX/SSE state and reload the new state. This option may prevent this data from being changed when it's not necessary (for e.g. when no tasks or only one task is using them), but fails to work correctly in a multiprocessor environment without additional synchronization which may be more expensive than using the first option.

Performance Considerations

Because the hardware mechanism saves almost all of the CPU state it can be slower than is necessary. For example, when the CPU loads new segment registers it does all of the access and permission checks that are involved. As most modern operating systems don't use segmentation loading the segment registers during context switches may be not be required, so for performance reasons these operating systems tend not to use the hardware context switching mechanism. Due to it not being used as much CPU manufacturers don't optimize CPUs for this method anymore (AFAIK). In addition the new 64 bit CPU's do not support hardware context switches when in 64 bit/long mode.

However, there was an interesting post on OSNews by Aage in July 2004, quantifying the amount of unavoidable hardware overhead involved in a context switch. It appears that the hardware overhead in a context switch on a modern P4 processor dwarfs the overhead involved in saving/loading registers (995ns of HW overhead vs 67ns to save/load registers). From this, it would appear that any performance gains from switching to software task switching would be minimal, amounting to no more than a few percentage points. However, Brendan points out in this post (http://forum.osdev.org/viewtopic.php?p=117933#p117933) that this is *horrendously wrong* and explains why.

There is actually quite little you can do in software to improve the overhead of context switches. Most of the overhead is hardware related. Sure you can tweak the code that stores/restores registers, performs scheduling, and stuff, but in the grand scheme of things hardware overhead dominates (I'll substantiate that below). Using the x86 as an example architecture:

Assuming the context switch is initiated by an interrupt, the overhead of switching from user-level to kernel-level on a (2.8 GHz) P4 is 1348 cycles, on a (200 MHz) P2 227 cycles. Why the big cycle difference? It seems like the P4 flushes its micro-op cache as part of handling an interrupt (go to arstechnica.com for some details on the micro-op cache). Counting actual time, the P4 takes 481 ns and the P2 1335 ns.

The return from kernel-level to user-level will cost you 923 cycles (330 ns) on a P4, 180 cycles (900 ns) on a P2.

The overhead of storing / restoring registers (not counting any TLB overhead / excluding cost of FPU register store / restore) is 188 cycles on the P4 (67 ns), 79 cycles on the P2 (395 ns).

A context switch also includes the overhead of switching address spaces (if we're switching between processes, not threads). The minimal cost of switching between two address spaces (counting a minimal TLB reload of 1 code page, 1 data page, and 1 stack page) is 516 cycles on a P4 (184 ns) and 177 cycles on a P3 (885 ns).

So the equation is (for a P4):

811 ns (HW) + 184 ns (HW: address space switch) + 67 ns (register store / restore) + ?? (scheduling overhead) = cost of context switch.

That'll leave you with 995 ns of HW overhead. You can spend as much as 2598 cycles in the scheduler before SW overhead dominates.

So, measured in actual time the cost of context switches is declining (P2: 3120 ns vs. P4: 995 ns - 3:1). But looking at CPU clock speed differences (P2: 200 MHz vs P4: 2800 MHz - 1:14), one can only conclude that the cost of context switches is rising.

And yes, I used some home-grown software to perform these measurements.

—Aage, *OSNews*

Retrieved from "http://wiki.osdev.org/index.php?title=Context_Switching&oldid=16908"
Category: Processes and Threads

- This page was last modified on 15 October 2014, at 11:54.

- This page has been accessed 91,375 times.