

Context switch internals

上下文切换内部

Asked 12 years, 8 months ago Modified 6 years, 6 months ago Viewed 42k times



I want to learn and fill gaps in my knowledge with the help of this question.

85



我想在这个问题的帮助下学习和填补我的知识空白。

So, a user is running a thread (kernel-level) and it now calls `yield` (a system call I presume). The scheduler must now save the context of the current thread in the TCB (which is stored in the kernel somewhere) and choose another thread to run and loads its context and jump to its `CS:EIP`. To narrow things down, I am working on Linux running on top of x86 architecture. Now, I want to get into the details:

因此，用户正在运行一个线程（内核级），它现在调用 `yield`（我推测是系统调用）。调度程序现在必须将当前线程的上下文保存在 TCB 中（存储在内核中的某个位置），然后选择另一个线程来运行并加载其上下文并跳转到其 `CS: EIP`。为了缩小范围，我正在研究运行在 x86 架构之上的 Linux。现在，我想深入了解细节：

So, first we have a system call:

所以，首先我们有一个 system 调用：

1) The wrapper function for `yield` will push the system call arguments onto the stack. Push the return address and raise an interrupt with the system call number pushed onto some register (say `EAX`).

1) `yield` 的包装函数会将系统调用参数推送到堆栈上。推送返回地址并引发中断，将系统调用号推送到某个寄存器（例如 `EAX`）。

2) The interrupt changes the CPU mode from user to kernel and jumps to the interrupt vector table and from there to the actual system call in the kernel.

2) 中断将 CPU 模式从用户更改为内核，并跳转到中断向量表，再从那里跳转到内核中的实际系统调用。

3) I guess the scheduler gets called now and now it must save the current state in the TCB. Here is my dilemma. Since, the scheduler will use the kernel stack and not the user stack for performing its operation (which means the `SS` and `SP` have to be changed) how does it store the state of the user without modifying any registers in the process. I have read on forums that there are special hardware instructions for saving state but then how does the scheduler get access to them and who runs these instructions and when?

3) 我猜调度器现在被调用，现在它必须将当前状态保存在 TCB 中。这是我的困境。由于调度器将使用 kernel 堆栈而不是 user 堆栈来执行其作（这意味着必须更改 `SS` 和 `SP`），因此它如何在修改进程中的任何 registers 的情况下存储用户的状态。我在论坛上读到有助于保存状态的特殊硬件说明，但是调度程序如何访问它们以及谁在何时运行这些指令？

4) The scheduler now stores the state into the TCB and loads another TCB.

4) 调度程序现在将状态存储到 TCB 中并加载另一个 TCB。

5) When the scheduler runs the original thread, the control gets back to the wrapper function which clears the stack and the thread resumes.

5) 当调度程序运行原始线程时，控件返回到包装函数，该函数清除堆栈，线程恢复。

Side questions: Does the scheduler run as a kernel-only thread (i.e. a thread which can run only kernel code)? Is there a separate kernel stack for each kernel-thread or each process?

附带问题：调度程序是否作为仅内核线程（即只能运行内核代码的线程）运行？每个内核线程或每个进程是否有单独的内核堆栈？

linux-kernel kernel scheduler context-switch

Share Improve this question Follow

edited Nov 25, 2018 at 22:01



Guy Avraham

3,712 3 41 54

asked Sep 27, 2012 at 21:22



Bruce

35.3k 77 183 266

3 Answers

Sorted by: Highest score (default) ⌵



131



+50



At a high level, there are two separate mechanisms to understand. The first is the kernel entry/exit mechanism: this switches a single running thread from running usermode code to running kernel code in the context of that thread, and back again. The second is the context switch mechanism itself, which switches in kernel mode from running in the context of one thread to another.

在高级别上，有两种不同的机制需要理解。第一个是内核进入/退出机制：这会将单个正在运行的线程从运行用户模式代码切换到在该线程的上下文中运行内核代码，然后再切换回来。第二个是上下文切换机制本身，它在内核模式下从一个线程的上下文中运行切换到另一个线程。

So, when Thread A calls `sched_yield()` and is replaced by Thread B, what happens is:

因此，当线程 A 调用 `sched_yield ()` 并被线程 B 替换时，将发生以下情况：

- Thread A enters the kernel, changing from user mode to kernel mode;
线程 A 进入内核，从用户模式变为内核模式;
- Thread A in the kernel context-switches to Thread B in the kernel;
内核上下文中的线程 A 切换到内核中的线程 B;
- Thread B exits the kernel, changing from kernel mode back to user mode.
线程 B 退出内核，从内核模式切换回用户模式。

Each user thread has both a user-mode stack and a kernel-mode stack. When a thread enters the kernel, the current value of the user-mode stack (`SS:ESP`) and instruction pointer (`CS:EIP`) are saved to the thread's kernel-mode stack, and the CPU switches to the kernel-mode stack - with the `int $80` syscall mechanism, this is done by the CPU itself. The remaining register values and flags are then also saved to the kernel stack.

每个用户线程都有一个用户模式堆栈和一个内核模式堆栈。当线程进入内核时，用户模式堆栈（`SS: ESP`）和指令指针（`CS: EIP`）的当前值将保存到线程的内核模式堆栈中，CPU 切换到内核模式堆栈 - 使用 `int $80` syscall 机制，这是由 CPU 本身完成的。然后，剩余的 register 值和标志也会保存到 kernel 堆栈中。

When a thread returns from the kernel to user-mode, the register values and flags are popped from the kernel-mode stack, then the user-mode stack and instruction pointer values are restored from the saved values on the kernel-mode stack.

当线程从内核返回到用户模式时，将从内核模式堆栈中弹出寄存器值和标志，然后从内核模式堆栈上保存的值还原用户模式堆栈和指令指针值。

When a thread context-switches, it calls into the scheduler (the scheduler does not run as a separate thread - it always runs in the context of the current thread). The scheduler code selects a process to run next, and calls the `switch_to()` function. This function essentially just switches the kernel stacks - it saves the current value of the stack pointer into the TCB for the current thread (called `struct task_struct` in Linux), and loads a

previously-saved stack pointer from the TCB for the next thread. At this point it also saves and restores some other thread state that isn't usually used by the kernel - things like floating point/SSE registers. If the threads being switched don't share the same virtual memory space (ie. they're in different processes), the page tables are also switched.

当线程上下文切换时，它会调用调度程序（调度程序不作为单独的线程运行 - 它始终在当前线程的上下文中运行）。调度程序代码选择接下来要运行的进程，并调用 `switch_to ()` 函数。此函数实质上只是切换内核堆栈 - 它将堆栈指针的当前值保存到当前线程的 TCB 中（在 Linux 中称为 `struct task_struct`），并从 TCB 为下一个线程加载先前保存的堆栈指针。此时，它还会保存和恢复一些内核通常不使用的其他线程状态 - 比如浮点/SSE 寄存器。如果正在切换的线程不共享相同的虚拟内存空间（即它们位于不同的进程中），则页面也会切换。

So you can see that the core user-mode state of a thread isn't saved and restored at context-switch time - it's saved and restored to the thread's kernel stack when you enter and leave the kernel. The context-switch code doesn't have to worry about clobbering the user-mode register values - those are already safely saved away in the kernel stack by that point.

因此，你可以看到线程的核心用户模式状态不会在上下文切换时保存和还原 - 当你进入和离开内核时，它会保存并还原到线程的内核堆栈中。上下文切换代码不必担心破坏用户模式寄存器值 - 到那时这些值已经安全地保存在 kernel 堆栈中。

Share Improve this answer Follow

edited Jun 29, 2018 at 2:23

answered Oct 3, 2012 at 5:20



caf
240k ● 41 ● 340 ● 477

Great answer!! So, the scheduler uses the kernel stack of the thread it is switching from? Also, please provide some sources for your awesome knowledge.
很好的回答！！那么，调度程序使用它正在切换的线程的内核堆栈吗？另外，请提供一些来源来获取您的精彩知识。 – Bruce Oct 3, 2012 at 14:17

11 @Bruce: In my opinion the best source is the source - for example [the x86 switch_to routine](#). It helps to read it in conjunction with the platform documentation (eg. *Intel 64 and IA-32 Architectures Software Developer's Manual*, which is freely available from Intel).

@Bruce: 在我看来，最好的源是源 - 例如 [x86 switch_to 例程](#)。将其与平台文档（例如。*Intel 64 和 IA-32 架构软件开发人员手册*，可从 Intel 免费获得）。 – caf Oct 3, 2012 at 23:44

@caf Great answer! So the user-space registers are not saved in anywhere (except SS,CS,EIP,ESP), right? And where is TCB saved in kernel, on a heap?

@caf 很好的回答！所以用户空间寄存器不会保存在任何地方（除了 SS，CS，EIP，ESP），对吧？TCB 保存在内核中的什么位置，在堆上？ – WindChaser May 7, 2015 at 20:42

@WindChaser: You might have missed this part: *"The remaining register values and flags are then also saved to the kernel stack."*. The `task_struct` is dynamically allocated by the kernel (although the kernel doesn't really have a concept of "heap") and added to a global linked list of tasks.

@WindChaser: 您可能错过了这一部分：“剩余的寄存器值和标志随后也保存到内核堆栈中。`task_struct` 由内核动态分配（尽管内核实际上没有“heap”的概念），并添加到 task 的全局链表中。

– caf May 8, 2015 at 0:22

1 @Amnesiac: Not in this case - it is clear what the OP means because in point 2 they talk about transitioning from user to kernel mode.

@Amnesiac: 在这种情况下不是 - OP 的含义很清楚，因为在第 2 点中，他们谈到了从用户模式过渡到内核模式。 – caf Sep 21, 2015 at 1:25

▲

15

▼

🔖

🔄

What you missed during step 2 is that the stack gets switched from a thread's user-level stack (where you pushed args) to a thread's protected-level stack. The current context of the thread interrupted by the syscall is actually saved on this protected stack. Inside the ISR and just before entering the kernel, this protected-stack is again switched to *the* kernel stack you are talking about. Once inside the kernel, kernel functions such as scheduler's functions eventually use the kernel-stack. Later on, a thread gets elected by the scheduler and the system returns to the ISR, it switches back from the kernel stack to the newly elected (or the former if no higher priority thread is active) thread's protected-level stack, wich eventually contains the new thread context. Therefore the context is restored from this stack by code automatically (depending on the underlying architecture). Finally, a special instruction restores the latest touchy resgisters such as the stack pointer and the instruction pointer. Back in the userland...

在第 2 步中，您错过的是，堆栈从线程的用户级堆栈（您推送 args 的位置）切换到线程的受保护级别堆栈。被 syscall 中断的线程的当前上下文实际上保存在这个受保护的堆栈上。在 ISR 内部，就在进入内核之前，这个 protected-stack 再次切换到你正在谈论的内核堆栈。一旦进入内核，内核函数（如 scheduler 的函数）最终会使用 kernel-stack。稍后，线程由调度程序选择，系统返回到 ISR，它从内核堆栈切换回新选择的（如果没有更高优先级的线程处于活动状态，则切换回前者）线程的受保护级别堆栈，最终包含新的线程上下文。因此，代码会自动从此堆栈中恢复上下文（取决于底层架构）。最后，特殊指令会恢复最新的敏感注册表器，例如堆栈指针和指令指针。回到用户空间.....

To sum-up, a thread has (generally) two stacks, and the kernel itself has one. The kernel stack gets wiped at the end of each kernel entering. It's interesting to point out that since 2.6, the kernel itself gets threaded for some processing, therefore a kernel-thread has its own protected-level stack beside the general kernel-stack.

总而言之，一个线程（通常）有两个堆栈，内核本身有一个。内核堆栈在每个内核输入结束时被擦除。有趣的是，从 2.6 开始，内核本身被线程化以进行一些处理，因此内核线程在一般内核堆栈之外有自己的受保护级堆栈。

Some ressources: 一些资源：

- 3.3.3 *Performing the Process Switch* of **Understanding the Linux Kernel**, O'Reilly
3.3.3 执行理解 **Linux** 内核的进程切换（O'Reilly）
- 5.12.1 *Exception- or Interrupt-Handler Procedures* of the **Intel's manual 3A (sysprogramming)**. Chapter number may vary from edition to other, thus a lookup on "Stack Usage on Transfers to Interrupt and Exception-Handling Routines" should get you to the good one.
5.12.1 **Intel 手册 3A（sysprogramming）** 的异常或中断处理程序。章节编号可能因版本而异，因此查找“Stack Usage on Transfers to Interrupt and Exception-Handling Routines”应该能找到一个好的版本。

Hope this help! 希望这有帮助！

Share Improve this answer Follow

edited Oct 3, 2012 at 8:40

answered Sep 28, 2012 at 13:33



Benny
4,331 ● 1 ● 27 ● 29

1 Actually I am more confused than before :-). Can you provide a reference for your answer. That might help.

其实我比以前更困惑了：-)。您能否为您的答案提供參考。这可能会有所帮助。 – Bruce Sep 28, 2012 at 22:55

When you say "...the stack gets switched from a thread's user-level stack (where you pushed args) to a thread's protected-level stack. The current context of the thread interrupted by the syscall is actually saved on this protected stack.", how does it switch the stack pointer to point to the protected-level stack while at the same time save the original stack pointer (and all the registers) onto said stack?

当您说"...堆栈从线程的用户级堆栈（推送 args 的位置）切换到线程的受保护级别堆栈。被系统调用中断的线程的当前上下文实际上保存在这个受保护的堆栈上”，它如何切换堆栈指针以指向受保护的级别堆栈，同时将原始堆栈指针（和所有寄存器）保存到所述堆栈上？

– mclaassen Aug 13, 2014 at 13:40

@mclaassen Good question; this is arch-dependent. Usually there are 2 stack pointers managed internally. On ARM there are 2 stack pointer registers ('normal' and 'interrupt' sp's: *psp* and *misp* in the doc). On Intel the previous SP is pushed on the *Ring0* stack, thus restored from there.

@mclaassen 好问题;这是依赖于 Arch 的。通常有 2 个堆栈指针在内部管理。在 ARM 上有 2 个堆栈指针寄存器（文档中的 'normal' 和 'interrupt' sp: *psp* 和 *misp*）。在 Intel 上，以前的 SP 被推送到 *Ring0* 堆栈上，从而从那里恢复。


– Benny Aug 13, 2014 at 14:42


▲

11

▼

Kernel itself have no stack at all. The same is true for the process. It also have no stack. Threads are only system citizens which are considered as execution units. Due to this only threads can be scheduled and only threads have stacks. But there is one point which kernel mode code exploits heavily - every moment of time system works in the context of the currently active thread. Due to this kernel itself can reuse the stack of the currently active stack. Note that only one of them can execute at the same moment of time either kernel code or user code. Due to this when kernel is invoked it just reuse thread stack and perform a cleanup before returning control back to the interrupted activities in the thread. The

 same mechanism works for interrupt handlers. The same mechanism is exploited by signal handlers.



内核本身根本没有堆栈。这个过程也是如此。它也没有堆栈。线程只是被视为执行单元的系统公民。因此，只能调度线程，并且只有线程具有堆栈。但是内核模式代码严重利用了一点 - 系统的每一刻都在当前活动线程的上下文中工作。因此，内核本身可以重用当前活动堆栈的堆栈。请注意，它们中只有一个可以在同一时刻执行内核代码或用户代码。因此，当调用内核时，它只需重用线程堆栈并执行清理，然后再将控制权返回给线程中被中断的活动。相同的机制也适用于中断处理程序。信号处理程序也利用了相同的机制。

In its turn thread stack is divided into two isolated parts, one of which called user stack (because it is used when thread executes in user mode), and second one is called kernel stack (because it is used when thread executes in kernel mode). Once thread crosses the border between user and kernel mode, CPU automatically switches it from one stack to another. Both stack are tracked by kernel and CPU differently. For the kernel stack, CPU permanently keeps in mind pointer to the top of the kernel stack of the thread. It is easy, because this address is constant for the thread. Each time when thread enters the kernel it found empty kernel stack and each time when it returns to the user mode it cleans kernel stack. In the same time CPU doesn't keep in mind pointer to the top of the user stack, when thread runs in the kernel mode. Instead during entering to the kernel, CPU creates special "interrupt" stack frame on the top of the kernel stack and stores the value of the user mode stack pointer in that frame. When thread exits the kernel, CPU restores the value of ESP from previously created "interrupt" stack frame, immediately before its cleanup. (on legacy x86 the pair of instructions int/iret handle enter and exit from kernel mode)

反过来，线程堆栈分为两个独立的部分，其中一个称为用户堆栈（因为它在线程以用户模式执行时使用），另一个称为内核堆栈（因为它在线程以内核模式执行时使用）。一旦线程越过用户模式和内核模式之间的边界，CPU 就会自动将其从一个堆栈切换到另一个堆栈。内核和 CPU 对这两个堆栈的跟踪方式不同。对于内核堆栈，CPU 始终牢记指向线程内核堆栈顶部的指针。这很容易，因为此地址对于线程是常量。每次 thread 进入内核时，它都会发现空的内核堆栈，每次返回用户模式时，它都会清理内核堆栈。同时，当线程在内核模式下运行时，CPU 不会记住指向用户堆栈顶部的指针。相反，在进入内核期间，CPU 会在内核堆栈的顶部创建特殊的“interrupt”堆栈帧，并将用户模式堆栈指针的值存储在该帧中。当线程退出内核时，CPU 会在清理之前从先前创建的“interrupt”堆栈帧中恢复 ESP 的值。（在旧版 x86 上，一对指令 int/iret 句柄进入和退出内核模式）

During entering to the kernel mode, immediately after CPU will have created "interrupt" stack frame, kernel pushes content of the rest of CPU registers to the kernel stack. Note that is saves values only for those registers, which can be used by kernel code. For example kernel doesn't save content of SSE registers just because it will never touch them. Similarly just before asking CPU to return control back to the user mode, kernel pops previously saved content back to the registers.

在进入内核模式期间，在 CPU 创建“interrupt”堆栈帧后，内核会立即将其余 CPU 寄存器的内容推送到内核堆栈。请注意，is 仅保存那些 registers 的值，这些 registers 可以由 kernel code 使用。例如，kernel 不会保存 SSE 寄存器的内容，因为它永远不会触及它们。同样，就在要求 CPU 将控制权返回给 user mode 之前，kernel 将之前保存的内容弹出回 registers。

Note that in such systems as Windows and Linux there is a notion of system thread (frequently called kernel thread, I know it is confusing). System threads a kind of special threads, because they execute only in kernel mode and due to this have no user part of the stack. Kernel employs them for auxiliary housekeeping tasks.

请注意，在 Windows 和 Linux 等系统中，有一个系统线程的概念（通常称为内核线程，我知道它令人困惑）。系统线程 一种特殊的线程，因为它们仅在内核模式下执行，因此没有堆栈的用户部分。Kernel 使用它们来完成辅助内务处理任务。

Thread switch is performed only in kernel mode. That mean that both threads outgoing and incoming run in kernel mode, both uses their own kernel stacks, and both have kernel stacks have "interrupt" frames with pointers to the top of the user stacks. Key point of the thread switch is a switch between kernel stacks of threads, as simple as:

线程切换仅在内核模式下执行。这意味着传出线程和传入线程都在内核模式下运行，都使用自己的内核堆栈，并且都有内核堆栈具有“中断”帧，这些帧带有指向用户堆栈顶部的指针。线程切换的关键点是线程的内核堆栈之间的切换，简单如下：

```
pushad; // save context of outgoing thread on the top of the kernel stack of outgoing thread
; here kernel uses kernel stack of outgoing thread
mov [TCB_of_outgoing_thread], ESP;
mov ESP, [TCB_of_incoming_thread]
; here kernel uses kernel stack of incoming thread
popad; // save context of incoming thread from the top of the kernel stack of incoming thread
```

Note that there is only one function in the kernel that performs thread switch. Due to this each time when kernel has stacks switched it can find a context of incoming thread on the top of the stack. Just because every time before stack switch kernel pushes context of outgoing thread to its stack.

请注意，内核中只有一个函数执行线程切换。因此，每次内核切换堆栈时，它都可以在堆栈顶部找到传入线程的上下文。只是因为每次在堆栈切换之前，内核都会将传出线程的上下文推送到它的堆栈中。

Note also that every time after stack switch and before returning back to the user mode, kernel reloads the mind of CPU by new value of the top of kernel stack. Making this it assures that when new active thread will try to enter kernel in future it will be switched by CPU to its own kernel stack.

另请注意，每次在堆栈切换之后和返回用户模式之前，内核都会通过内核堆栈顶部的新值重新加载 CPU。这样做可以确保将来新的活动线程尝试进入内核时，它将由 CPU 切换到自己的内核堆栈。

Note also that not all registers are saved on the stack during thread switch, some registers like FPU/MMX/SSE are saved in specially dedicated area in TCB of outgoing thread. Kernel employs different strategy here for two reasons. First of all not every thread in the system uses them. Pushing their content to and and popping it from the stack for every thread is inefficient. And second one there are special instructions for "fast" saving and loading of their content. And these instructions doesn't use stack.


另请注意，在线程切换期间，并非所有寄存器都保存在堆栈上，一些寄存器（如 FPU/MMX/SSE）保存在传出线程的 TCB 中的专用区域。Kernel 在这里采用不同的策略有两个原因。首先，并非系统中的每个线程都使用它们。为每个线程将他们的内容推送到堆栈并从堆栈中弹出它效率低下。第二个是“快速”保存和加载其内容的特殊说明。并且这些指令不使用 stack。

Note also that in fact kernel part of the thread stack has fixed size and is allocated as part of TCB. (true for Linux and I believe for Windows too)

另请注意，实际上线程堆栈的内核部分具有固定大小，并作为 TCB 的一部分进行分配。（适用于 Linux，我相信 Windows 也是如此）

Share Improve this answer Follow

answered Oct 25, 2016 at 11:59

 ZarathustrA
3,680 ● 34 ● 28

Can you please clarify, where does then other part of User Stack is stored (since interrupt frame is only for Stack Pointer) during Thread parking i.e. WAIT?

您能否澄清一下，在线程停放期间，即 WAIT、用户堆栈的其他部分存储在哪里（因为中断帧仅用于堆栈指针）？ – uptoyou Sep 19, 2019 at 20:46

Thread switch performs in the kernel mode. Thus to make it, the thread should enter into kernel mode. However, each time when thread switches from user mode into kernel mode, the kernel saves the state of CPU registers on the kernel part of thread stack and restores them when switches back into user mode.

线程切换在内核模式下执行。因此，要做到这一点，线程应该进入内核模式。但是，每次线程从用户模式切换到内核模式时，内核都会将 CPU 寄存器的状态保存在线程堆栈的内核部分，并在切换回用户模式时恢复它们。

– ZarathustrA Sep 23, 2019 at 14:46

yep, thanks, you mentioned it. However, there are also local variables, function arguments, function return pointer, which i guess are located on the User Stack. If so, when Thread switches to kernel mode, where does those User variables stored ? I mean those, that locates in RAM memory, that haven't reached the CPU registers yet.

是的，谢谢，你提到了。但是，还有局部变量、函数参数、函数返回指针，我猜它们位于用户堆栈上。如果是这样，当 Thread 切换到内核模式时，这些 User 变量存储在哪里？我的意思是那些位于 RAM 内存中的、尚未到达 CPU 寄存器的。

– uptoyou Sep 23, 2019 at 21:07

They are stored in user mode part of the thread stack, which is stored in user memory. When you switch into kernel mode, kernel switches to kernel part of the thread stack and do not uses user part, thus all data in the user part of the stack is preserved in the same state.

它们存储在线程堆栈的用户模式部分，线程堆栈存储在用户内存中。切换到内核模式时，内核会切换到线程堆栈的内核部分，而不使用用户部分，因此堆栈的用户部分中的所有数据都保持相同的状态。

– ZarathustrA Sep 24, 2019 at 0:31

Start asking to get answers

开始提问以获得答案

Find the answer to your question by asking.
通过提问找到您问题的答案。

Ask question

Explore related questions

探索相关问题

- linux-kernel linux 内核 kernel 内核 scheduler 调度
- context-switch 上下文切换

See similar questions with these tags.
请参阅带有这些标签的类似问题。