Kernel Stacks

Kernel stacks on x86-64 bit

Most of the text from Keith Owens, hacked by AK

x86_64 page size (PAGE_SIZE) is 4K.

Like all other architectures, x86_64 has a kernel stack for every active thread. These thread stacks are THREAD_SIZE (2*PAGE_SIZE) big. These stacks contain useful data as long as a thread is alive or a zombie. While the thread is in user space the kernel stack is empty except for the thread_info structure at the bottom.

In addition to the per thread stacks, there are specialized stacks associated with each CPU. These stacks are only used while the kernel is in control on that CPU; when a CPU returns to user space the specialized stacks contain no useful data. The main CPU stacks are:

- Interrupt stack. IRQ_STACK_SIZE

  Used for external hardware interrupts. If this is the first external hardware interrupt (i.e. not a nested hardware interrupt) then the kernel switches from the current task to the interrupt stack. Like the split thread and interrupt stacks on i386, this gives more room for kernel interrupt processing without having to increase the size of every per thread stack.

  The interrupt stack is also used when processing a softirq.

Switching to the kernel interrupt stack is done by software based on a per CPU interrupt nest counter. This is needed because x86-64 "IST" hardware stacks cannot nest without races.

x86_64 also has a feature which is not available on i386, the ability to automatically switch to a new stack for designated events such as double fault or NMI, which makes it easier to handle these unusual events on x86_64. This feature is called the Interrupt Stack Table (IST). There can be up to 7 IST entries per CPU. The IST code is an index into the Task State Segment (TSS). The IST entries in the TSS point to dedicated stacks; each stack can be a different size.

An IST is selected by a non-zero value in the IST field of an interrupt-gate descriptor. When an interrupt occurs and the hardware loads such a descriptor, the hardware automatically sets the new stack pointer based on the IST value, then invokes the interrupt handler. If the interrupt came from user mode, then the interrupt handler prologue will switch back to the per-thread stack. If software wants to allow nested IST interrupts then the handler must adjust the IST values on entry to and exit from the interrupt handler. (This is occasionally done, e.g. for debug exceptions.)

Events with different IST codes (i.e. with different stacks) can be nested. For example, a debug interrupt can safely be interrupted by an NMI. arch/x86_64/kernel/entry.S::paranoidentry adjusts the stack pointers on entry to and exit from all IST events, in theory allowing IST events with the same code to be nested. However in most cases, the stack size allocated to an IST assumes no nesting for the same code. If that assumption is ever broken then the stacks will become corrupt.

The currently assigned IST stacks are:

- ESTACK_DF. EXCEPTION_STKSZ (PAGE_SIZE).

  Used for interrupt 8 - Double Fault Exception (#DF).

  Invoked when handling one exception causes another exception. Happens when the kernel is very confused (e.g. kernel stack pointer corrupt). Using a separate stack allows the kernel to recover from it well enough in many cases to still output an oops.

- ESTACK_NMI. EXCEPTION_STKSZ (PAGE_SIZE).

  Used for non-maskable interrupts (NMI).

  NMI can be delivered at any time, including when the kernel is in the middle of switching stacks. Using IST for NMI events avoids making assumptions about the previous state of the kernel stack.

- ESTACK_DB. EXCEPTION_STKSZ (PAGE_SIZE).

  Used for hardware debug interrupts (interrupt 1) and for software debug interrupts (INT3).

  When debugging a kernel, debug interrupts (both hardware and software) can occur at any time. Using IST for these interrupts avoids making assumptions about the previous state of the kernel stack.

  To handle nested #DB correctly there exist two instances of DB stacks. On #DB entry the IST stackpointer for #DB is switched to the second instance so a nested #DB starts from a clean stack. The nested #DB switches the IST stackpointer to a guard hole to catch triple nesting.

- ESTACK_MCE. EXCEPTION_STKSZ (PAGE_SIZE).

  Used for interrupt 18 - Machine Check Exception (#MC).

  MCE can be delivered at any time, including when the kernel is in the middle of switching stacks. Using IST for MCE events avoids making assumptions about the previous state of the kernel stack.

For more details see the Intel IA32 or AMD AMD64 architecture manuals.

Printing backtraces on x86

The question about the '?' preceding function names in an x86 stacktrace keeps popping up, here's an indepth explanation. It helps if the reader stares at print_context_stack() and the whole machinery in and around arch/x86/kernel/dumpstack.c.

Adapted from Ingo's mail, Message-ID: <20150521101614.GA10889@gmail.com>:

We always scan the full kernel stack for return addresses stored on the kernel stack(s) [1], from stack top to stack bottom, and print out anything that 'looks like' a kernel text address.

If it fits into the frame pointer chain, we print it without a question mark, knowing that it's part of the real backtrace.

If the address does not fit into our expected frame pointer chain we still print it, but we print a '?'. It can mean two things:

- either the address is not part of the call chain: it's just stale values on the kernel stack, from earlier function calls. This is the common case.
- or it is part of the call chain, but the frame pointer was not set up properly within the function, so we don't recognize it.

This way we will always print out the real call chain (plus a few more entries), regardless of whether the frame pointer was set up correctly or not - but in most cases we'll get the call chain right as well. The entries printed are strictly in stack order, so you can deduce more information from that as well.

The most important property of this method is that we _never_ lose information: we always strive to print _all_ addresses on the stack(s) that look like kernel text addresses, so if debug information is wrong, we still print out the real call chain as well - just with more question marks than ideal.

[1] For things like IRQ and IST stacks, we also scan those stacks, in the right order, and try to cross from one stack into another reconstructing the call chain. This works most of the time.