

Re: Interrupt context...

Source: <http://linux.derkeiler.com/Newsgroups/comp.os.linux.development.system/2005-12/msg00031.html>

- *From:* Kasper Dupont <kasperd@xxxxxxxxxxxx>
 - *Date:* Thu, 01 Dec 2005 21:52:09 +0100
-

"arijit.p@xxxxxxxx" wrote:

- >
- > I have
- > gone through most of the posts on interrupt in usenet.

Wow! How many posts is that?

- >
- > When an interrupt comes, whatever the user is executing is saved in the
- > kernel stack and ISR is executed.

Correct.

- > Now my question is, what will be the
- > context of that interrupt?

That question is a bit vague.

- > What things will be setup to execute ISR?

The hardware will save at least the CPU flags and the program counter (also sometimes known as the instruction pointer). More may be saved depending on the architecture. When the CPU has saved what it wants to save, it switch privilege level if so required and jumps to the entry point for the interrupt handler. Here the kernel have assembler code to save all general purpose registers onto the stack before jumping to the C code doing the real work.

- > I mean will there be code, heap, stack allocated for this ISR?

Interrupt handlers are part of the kernel code. The heap concept doesn't really exist at this low level, but the handler does have access to the full kernel memory space.

Depending on architecture and kernel version, the handler may use either the kernel stack of the interrupted thread or a special interrupt stack. Either will be located in kernel memory.

Re: Interrupt context...

- > Or since
- > ISR is part of kernel, it will be executed on the kernel that was
- > loaded into RAM(meaning, that will use kernel code, heap, stack).

All of those will be in kernel memory.

- > If
- > so, then one more doubt – we have 4 GB AS for each process in linux and
- > within that 1Gb is for Kernel. So, for each process we setup a 1GB or
- > all process share a common 1GB kernel?

The same 1GB is shared by all threads on the system. And again the size of this depends on architecture and kernel version, but it is 1GB on the most common systems.

- > Is Kernel stack (8KB) is
- > allocated in that 1 GB?

It is allocated in kernel space, but the size depends on architecture and kernel version. It used to be 8KB but in 2.6 an option was introduced to reduce it to 4KB. There has been talk about removing the 8KB option and making it 4KB on future kernels. This applies at least to i386, I'm not so sure about other architectures. I think some architectures used to have 16KB stacks, but I'm not sure about that.

For smaller stacks to work it is important that interrupts execute on a separate stack. The way Linux was originally designed would require an interrupt stack to have the same size as the kernel stack for a thread, but that may have changed.

- > if yes, then does it puts a upper limit on
- > number of process that can be run on linux (since there is only < 1GB
- > on which we have to allocate 8K kernel stack for each process being
- > run)?

That is correct. But there are other reasons why the number of threads would be limited. The stack is not the only kernel memory that must be allocated for a thread. But it may be the largest.

- > what is the place holder (data structure) in 1GB kernel that
- > maintain all the Kernel stacks for all process?

I'm not sure I understand that question. But I think the answer is the struct `task_struct`. On some architectures this is located at the bottom of the stack, that way the stack pointer can be used to locate the `task_struct` of

Re: Interrupt context...

Re: Interrupt context...

the current thread. On each architecture there is a current macro, that will return a pointer to the task_struct of the current thread. These structs have pointers to each other and lots of other kernel data structures. And there are pointers to task_structs in a lot of places in the kernel data structures.

> How is the
> architecture? Assuming kernel is a process

....is an incorrect assumption.

> with code, heap and stack,
> so which part of that used to hold all the kernel stacks? How kernel
> finds out the exact kernel stack for the process which is scheduled
> now?

This is really the central trick in how a scheduler works. The scheduler saves most of the registers onto the stack, then it saves the stack pointer in the task_struct of the current thread and loads the stack pointer of the next thread to be executed. Then it simply proceeds restoring registers from the stack. It doesn't really have to find the stack, it just have to put the right value in the stack pointer register.

Since registers are saved both when a thread is interrupted (or enters kernel mode through a system call or an exception) and also by the scheduler, there will actually be two set of registers on the stack of each thread not currently executing on a CPU.

>
> Also, when a switch from user to ISR happens(or a context switch
> between another user process), is the previous process environment
> (code, heap, stack, I mean the pages allocated in the RAM for that
> process) destroyed?

No. The registers are saved on the stack. The memory allocations are not touched. However the scheduler will change the pointer to the top level page table if necessary. If an application creates multiple threads with shared address space, no change is required when switching between those. Also no change is required when switching to a kernel thread and back to the same process again. Since a TLB flush is required when switching page tables this does cost a significant amount of CPU time, for that reason the kernel will try to give threads CPU time in an order reducing the number of times it must change page tables.

Re: Interrupt context...

- > If not, then on what basis pages for new process
- > are loaded into RAM?
- >
- > Can I ignore all interrupts with cli assembly command?

No. First of all for security reasons that instruction only works in kernel mode. Besides on a multi CPU system other CPUs will still handle interrupts even if one have them disabled. It is possible to disable interrupts globally, but it is so expensive, that it should be done only in very rare cases. (Doing it when the system is booting or shutting down may be acceptable.)

- > Does all
- > interrupt comes through PIC? how about timer interrupt?

That really is architecture dependent. On original PC hardware the timer interrupt come through the PIC like most others. I think only interrupts produced by the CPU itself or the FPU could possibly bypass the PIC. And maybe you wouldn't even classify those as interrupts but rather as exceptions. Anyway on modern PCs with an APIC, it starts getting more complicated, and unfortunately I don't know all the details about that. Other architectures may be entirely different, but I guess most have some kind of interrupt controller.

- >
- > I am basically an C application programmer, I want to move to serious
- > network protocol development, is it a good idea to understand OS
- > (linux) concept before I start coding protocols?

You don't need to know all the details of the inner workings of the kernel just to implement network protocols. But of course knowing something about the kernel doesn't hurt.

What might be of interest is, that the first part of handling received packets happens in interrupt context. And for packets routed through the computer onto another interface, even the sending may happen before leaving interrupt context. (Same goes for low level replies generated by the kernel). So some of this may even work on a system that is more or less crashed, but it also does impose some restrictions on the kernel code handling this.

Packets generated by applications are OTOH initially handled in process context, except for packets that are queued or retransmitted.

Anyway most of this you don't need to understand as

Re: Interrupt context...

Re: Interrupt context...

long as you are only writing user mode code.

- > I felt understanding
- > lower level details would help me design and write better code for a
- > protocol. I know i have realised that late, but, at least I have a
- > start now. ;)

Actually I think knowing about latency, throughput, routing, and such stuff is more important than how a kernel works.

—

Kasper Dupont

Note to self: Don't try to allocate
256000 pages with GFP_KERNEL on x86.

.

- *Follow-Ups:*

- ◆ *Re: Interrupt context...*

- ◆ *From:* Nix

- *References:*

- ◆ *Interrupt context...*

- ◆ *From:* arijit.p@xxxxxxxxxx

- Prev by Date: *interrupts*
- Next by Date: *Re: interrupts*
- Previous by thread: *Re: Interrupt context...*
- Next by thread: *Re: Interrupt context...*
- Index(es):
 - ◆ *Date*
 - ◆ *Thread*