CPU Bug 与 Linux Kernel

软件有 Bug,司空见惯,但 CPU 有 Bug,对 Programmer 而言确不多见.当自己写的程序发现错误后,我想任何理智的 Programmer 都会想,肯定是我错了,因为 CPU 是不会犯错的.这当然几乎是百分之一百正确的,但也只能说是"几乎",否则在 Linux Kernel Source 的相关体系结构的目录下为什么会有 bugs.h 这个文件呢?(如果是 x86 CPU 的话,请看 include/asm-i386/bugs.h)

CPU 是人设计的,人会犯错,自然城门失火,殃及池鱼,CPU 自然也有 bug.

读 Linux source,读着读着,难免会有泄气的感觉,因为涉及面实在太广,且代码量实在大,但不可否认,也确有"好玩"的时候,比如能发觉 CPU 也是有 Bug 的,而且也不少.有些 Kernel 能补救一下,有些对 kernel 而言,实在无能为力,只能报告 CPU 犯了错.

下面举 3 个 x86 架构的 CPU bug --- 2 个是 Intel 的,一个是 Cyrix 的,来看看 Linux kernel 是怎么处理的.

FDIV BUG --- Intel 奔腾处理器浮点除法的 bug

在 include/asm-i386/bugs.h 头文件的 check_fpu()函数中有下面一段代码片段

```
        static double __initdata x = 4195835.0;
        static double __initdata y = 3145727.0;
            ...
1           /* Test for the divl bug.. */
2           __asm__("fninit\n\t"
3           "fldl %1\n\t"
4           "fdivl %2\n\t"
5           "fmull %2\n\t"
6           "fldl %1\n\t"
7           "fsubp %%st,%%st(1)\n\t"
8           "fistpl %0\n\t"
9           "fwait\n\t"
10          "fninit"
11          : "=m" (*&boot_cpu_data.fdiv_bug)
12          : "m" (*&x), "m" (*&y));
13      if (boot_cpu_data.fdiv_bug)
14          printk("Hmm, FPU with FDIV bug.\n");
```

从代码的注释中就可看出这段代码在测试"divl bug".上面的 gcc 中的嵌入汇编是一段浮点运算指令.我来把它翻译成对应的 c 代码.

```
double x = 4195835.0;
double y = 3145727.0;

double temp = x / y * y - x;
int fdiv_bug;
fdiv_bug = (int)temp;
if(fdiv_bug != 0)
{
    printf("Hmm, FPU with FDIV bug.\n");
}
```

够简单了,看懂了吧!

原来在 Intel 奔腾处理器上当 4195835.0 / 3145727.0 时，CPU 得出的结论是 1.3337,但正确的结论应该是 1.3338(不信,那计算器算一下). 这样 x / y * y - x 原本应该等于在精度范围内的零(由于计算机只能表示有限位数，所以上面的结果实际上也是非零，但那已经超出 CPU 的 double 精度范围，即如此微小差异不是当前 CPU 的 double 浮点精度能表示的，比如一个 CPU 只能分辨 0.00001,那么对于 0.000005,对于 CPU 而言无法表示，即等于零，但由于 x / y 这一步的误差太大，造成 x / y * y - x 的最终结果的值是 Intel 奔腾处理器精度范围内可辨认的非零值. 由于碰上这个 bug 的机遇比你中全国体育彩票头奖的机会还要难，所以一开始，Intel 不承认，但在证据确凿的情况下，承认了该 bug,但认为这个 Bug 涉及面极小，没有采取什么措施. 但 Intel 没想到的是一帮屁计算机不懂的记者会大作文章，包括 CNN 的，把这个非常技术性，也确实很难碰到---有人甚至说要 one trillion 次才能碰到---的问题爆炒了一下，弄得 Intel 很被动，不得不宣布大规模回收有问题的 CPU，大大的破了一次财.

对这个浮点除法 Bug, Linux Kernel 如果在系统初始化阶段检测到该 bug,只是打印一行警告，没有什么措施. 我的理解是对这种 Bug, Kernel 没有 fix 的机会（kernel 也不是万能的，只有硬件给软件机会，软件才可能有所发挥. 下面的 Intel 的 f00f bug 就是典型）.

F00F Bug --- Intel 奔腾处理器死锁

在 Intel 奔腾处理器下，无论是什么操作系统 --- Linux, Windows 9.X, Windows NT, DOS, FreeBSD 等等，你只要运行下面的代码，你就可以搞死 CPU.

```
char code[4] = {0xf0, 0x0f, 0xc7, 0xc8};

int main()
{
	void (*func)() = code;
	func();
	return	0;
}
```

这可是很严重的问题．因为要让 CPU 去执行那一行代码，真太容易了．比如你可以写个 ActiveX 控件包括那行代码,嵌在网页里，当有奔腾处理器的计算机浏览你的网页，从而下载你的 ActiveX 控件执行时，突然他发现他的机器没任何反应了－－－没有报错，没有 crash,甚至连那臭名昭著的 Blue Screen 都没有，就像科幻片里时间凝固的场景．

0xf00fc7c8 是什么呢?

在反汇编器里，它是这样一条指令 " lock cmpxchg8b %eax "

cmpxchg8b （Compare and Exchange 8 Bytes）的正常用法是带 1 个存储器操作数，例如 " cmpxchg8b (%ebx)",将(%ebx)中所指的 64 位内存操作数与 edx:eax 比较并相互交换。在 Intel 的 Instruction Manual 中是这么描述的

Compares the 64-bit value in EDX:EAX with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX. The destination operand is an 8-byte memory location. For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value.

举例

```
movl		$0x08002500, %esi
movl		$1,  %eax
mov		$0, %edx
xor		%ebx, %ebx
xor		%ecx, %ecx
cmpxhg8b	(%esi)
```

比较内存 0x08002500 的内容与%edx:%eax 中的值进行比较，如果内存 0x08002500 中的值为 1,则把它置为%ecx:%ebx 中的值；如果不相等，则把内存 0x08002500 中的值赋给%edx:%eax。

但跟据 Intel 指令的构成规律（指令的二进制 pattern），可以构造出象"cmpxhg8b %eax"这样的无意义的指令，CPU 在执行这条指令时会产生异常 6，这时 CPU 将从中断描述符表中启

动第 6 号中断向量---invalid opcode trap.但如果这条指令前面又加上 lock 前缀(指令码为 F0)，CPU 会错误的认为取中断向量是一个需要改写存储器的过程，于是就等待写操作的完成，这样就进入了死锁状态。

具体死锁原因要分析一下，在 Intel manual 上有对该指令的如下描述

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

最关键的是最后在括号中的那一句

The processor never produces a locked read without also producing a locked write.

也即是说 lock read 与 lock write 必须是成对出现的，在一个 lock read 以后必须跟着一个 lock write,就好像"lock read"是某个人进一间房子，然后锁上门，防止其他人在进入。只有里面的人通过"lock write"来开门出房间后，才能允许另外的人再次进入该房子。lock read 好比锁，lock write 好比锁的钥匙。

当执行 f00fc7c8 指令时，f00f bug 发生了。CPU 知道这是一条非法指令，所以它试图跳转到 invalid opcode 的处理代码。但由于前面的 lock 前缀，CPU 被搞糊涂了。当 CPU 发出总线 read 请求去读取 invalid opcode handler(在执行该代码以前，总得得到该代码吧)时，CPU 错误的发出了 LOCK#信号。而正象上面说的，LOCK#信号只有在修改内存的具有 Read-Modify-Write 的指令时才可以发出。在这种情况下，在获得 invalid opcode handler 地址时的两次连续内存读请求时 LOCK#信号保持着，但关键是并没有 write 请求，因为发生 trap 6 并不会引起改写 trap 6 handler 的地址。CPU 在等待一个永远不会到的期待，他将永远等待下去,**for ever**.

上面是跟据 Intel 的 manual 分析的，但毕竟不是电子专业出身，还是没有感性任知。要是能在逻辑分析仪上看一下当 CPU 发出这条命令后的波形与相关数据就容易理解多了。

这实在是一个很严重的问题，Intel 马上给出了 2 个用软件来修补的解决方案。Linux 选择了一种，并实现了它。

修补对内核的改动很少，主要在两个部分

1。系统初始化时，当在编译内核时加入对 bug 的 fix 时，要做一些特殊处理。由于该 bug 不能象 fdiv bug 那样动态的去检测---一检测就锁死 CPU,所以只能由用户通过编译内核静态的加入。所谓特殊处理就是使得执行这条指令时首先发出的是 page fault exception, 而不是 invalid opcode exception.

2。 在 page fault exception handler 中加入对该 bug 的辨认，如果是由此 bug 引起的 page fault,

则再转调用 invalid opcode handler.

第一部分的代码如下（我把散在多个文件中的相关修改集中到一块）

```
#ifndef CONFIG_X86_F00F_WORKS_OK
void __init trap_init_f00f_bug(void)
{
	/*
	* "idt" is magic - it overlaps the idt_descr
	* variable so that updating idt will automatically
	* update the idt descriptor..
	*/
	__set_fixmap(FIX_F00F, __pa(&idt_table), PAGE_KERNEL_RO);
	idt = (struct desc_struct *)__fix_to_virt(FIX_F00F);

	__asm__ __volatile__("lidt %0": "=m" (idt_descr));
}
#endif



void __set_fixmap (enum fixed_addresses idx, unsigned long phys, pgprot_t flags)
{
	unsigned long address = __fix_to_virt(idx);

	if (idx >= __end_of_fixed_addresses) {
		printk("Invalid __set_fixmap\n");
		return;
	}
	set_pte_phys(address, phys, flags);
}

#define __fix_to_virt(x) (FIXADDR_TOP - ((x) << PAGE_SHIFT))

#define FIXADDR_TOP	(0xffffe000UL)

enum fixed_addresses {
#ifdef CONFIG_X86_LOCAL_APIC
	FIX_APIC_BASE, /* local (CPU) APIC) -- required for SMP or not */
#endif
#ifdef CONFIG_X86_IO_APIC
	FIX_IO_APIC_BASE_0,
	FIX_IO_APIC_BASE_END = FIX_IO_APIC_BASE_0 + MAX_IO_APICS-1,
#endif
```

```
41      #ifdef CONFIG_X86_VISWS_APIC
42          FIX_CO_CPU,     /* Cobalt timer */
43          FIX_CO_APIC,    /* Cobalt APIC Redirection Table */
44          FIX_LI_PCIA,    /* Lithium PCI Bridge A */
45          FIX_LI_PCIB,    /* Lithium PCI Bridge B */
46      #endif
47      #ifndef CONFIG_X86_F00F_WORKS_OK
48          FIX_F00F,
49      #endif
50      #ifdef CONFIG_HIGHMEM
51          FIX_KMAP_BEGIN,    /* reserved pte's for temporary kernel mappings */
52          FIX_KMAP_END = FIX_KMAP_BEGIN+(KM_TYPE_NR*NR_CPUS)-1,
53      #endif
54          __end_of_permanent_fixed_addresses,
55          /* temporary boot-time mappings, used before ioremap() is functional */
56      #define NR_FIX_BTMAPS   16
57          FIX_BTMAP_END = __end_of_permanent_fixed_addresses,
58          FIX_BTMAP_BEGIN = FIX_BTMAP_END + NR_FIX_BTMAPS - 1,
59          __end_of_fixed_addresses
60      };
61
```

我来解释一下

9 __set_fixmap(FIX_F00F, __pa(&idt_table), PAGE_KERNEL_RO);

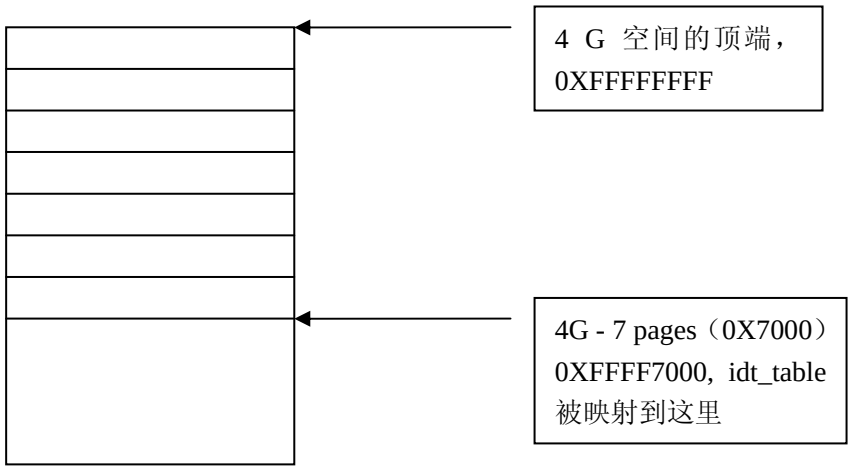
该行代码把 idt_table (中断向量表)映射到 4 G virtual address 的顶端某页，具体第几页由枚举常量 FIX_F00F 的值决定---这里我假设为 7。__pa(&idt_table)是取得 idt_table 的 physical address,也就是当前 idt_table 的虚拟地址减去 3G 的常量。同时把新映射到的 virtual address（0xffff7000）置为 Read Only (PAGE_KERNEL_RO) --- 通过 26 行代码

26 set_pte_phys(address, phys, flags);

通过修改页表来实现物理 phys 地址到虚拟 address 地址的映射。

12 __asm__ __volatile__("lidt %0": "=m" (idt_descr));

重新载入 idt table.也就是如果发生中断，exception 等，都将从新的 idt table 中去获得 handler.



上面做的这一切都是为了使得当 CPU 执行 0Xf00fc7c8 的指令时，在触发 invalid opcode exception 以前先触发 page fault exception(缺页异常)。

为什么会触发 page fault exception 呢？
关键是上面 PAGE_KERNEL_RO 的页属性。即新映射的 idt table 是只读的，而在上面分析中知道由于在指令前加了 lock 前缀，CPU 会误认为取 invalid opcode handler 的地址是一个需要改写内存的动作，但这时的 invalid opcode handler 所在的页属性又是只读的，违反规则，产生 Page Fault (缺页异常并非只属于“缺页”)。

这部分的代码如下（arch/i386/mm/fault.c）

```
/*
 * This routine handles page faults.  It determines the address,
 * and the problem, and then passes it off to one of the appropriate
 * routines.
 *
 * error_code:
 *	bit 0 == 0 means no page found, 1 means protection fault
 *	bit 1 == 0 means read, 1 means write
 *	bit 2 == 0 means kernel, 1 means user-mode
 */
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
	struct task_struct *tsk;
	struct mm_struct *mm;
	struct vm_area_struct * vma;
	unsigned long address;
```

```
unsigned long page;
unsigned long fixup;
int write;
siginfo_t info;

/* get the address */
__asm__("movl %%cr2,%0":"=r" (address));

/* It's safe to allow irq's after cr2 has been saved */
if (regs->eflags & X86_EFLAGS_IF)
        local_irq_enable();

tsk = current;

/*
 * We fault-in kernel-space virtual memory on-demand. The
 * 'reference' page table is init_mm.pgd.
 *
 * NOTE! We MUST NOT take any locks for this case. We may
 * be in an interrupt or a critical region, and should
 * only copy the information from the master page table,
 * nothing more.
 *
 * This verifies that the fault happens in kernel space
 * (error_code & 4) == 0, and that the fault was not a
 * protection error (error_code & 1) == 0.
 */
if (address >= TASK_SIZE && !(error_code & 5))
        goto vmalloc_fault;

mm = tsk->mm;
info.si_code = SEGV_MAPERR;

/*
 * If we're in an interrupt or have no user
 * context, we must not take the fault..
 */
if (in_interrupt() || !mm)
        goto no_context;

down_read(&mm->mmap_sem);

vma = find_vma(mm, address);
```

```
	if (!vma)
		goto bad_area;
	if (vma->vm_start <= address)
		goto good_area;
	if (!(vma->vm_flags & VM_GROWSDOWN))
		goto bad_area;
	if (error_code & 4) {
		/*
		 * accessing the stack below %esp is always a bug.
		 * The "+ 32" is there due to some instructions (like
		 * pusha) doing post-decrement on the stack and that
		 * doesn't show up until later..
		 */
		if (address + 32 < regs->esp)
			goto bad_area;
	}
	if (expand_stack(vma, address))
		goto bad_area;
/*
 * Ok, we have a good vm_area for this memory access, so
 * we can handle it..
 */
good_area:
	info.si_code = SEGV_ACCERR;
	write = 0;
	switch (error_code & 3) {
		default:	/* 3: write, present */
#ifdef TEST_VERIFY_AREA
			if (regs->cs == KERNEL_CS)
				printk("WP fault at %08lx\n", regs->eip);
#endif
			/* fall through */
		case 2:		/* write, not present */
			if (!(vma->vm_flags & VM_WRITE))
				goto bad_area;
			write++;
			break;
		case 1:		/* read, present */
			goto bad_area;
		case 0:		/* read, not present */
			if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
				goto bad_area;
	}
```

```
 survive:
        /*
         * If for any reason at all we couldn't handle the fault,
         * make sure we exit gracefully rather than endlessly redo
         * the fault.
         */
        switch (handle_mm_fault(mm, vma, address, write)) {
        case 1:
                tsk->min_flt++;
                break;
        case 2:
                tsk->maj_flt++;
                break;
        case 0:
                goto do_sigbus;
        default:
                goto out_of_memory;
        }

        /*
         * Did it hit the DOS screen memory VA from vm86 mode?
         */
        if (regs->eflags & VM_MASK) {
                unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
                if (bit < 32)
                        tsk->thread.screen_bitmap |= 1 << bit;
        }
        up_read(&mm->mmap_sem);
        return;

/*
 * Something tried to access memory that isn't in our memory map..
 * Fix it, but check if it's kernel or user first..
 */
bad_area:
        up_read(&mm->mmap_sem);

        /* User mode accesses just cause a SIGSEGV */
        if (error_code & 4) {
                tsk->thread.cr2 = address;
                tsk->thread.error_code = error_code;
                tsk->thread.trap_no = 14;
```

```
            info.si_signo = SIGSEGV;
            info.si_errno = 0;
            /* info.si_code has been set above */
            info.si_addr = (void *)address;
            force_sig_info(SIGSEGV, &info, tsk);
            return;
        }

        /*
         * Pentium F0 0F C7 C8 bug workaround.
         */
        if (boot_cpu_data.f00f_bug) {
            unsigned long nr;

            nr = (address - idt) >> 3;

            if (nr == 6) {
                do_invalid_op(regs, 0);
                return;
            }
        }

no_context:
        /* Are we prepared to handle this kernel fault?    */
        if ((fixup = search_exception_table(regs->eip)) != 0) {
            regs->eip = fixup;
            return;
        }

/*
 * Oops. The kernel tried to access some bad page. We'll have to
 * terminate things with extreme prejudice.
 */

        bust_spinlocks(1);

        if (address < PAGE_SIZE)
            printk(KERN_ALERT "Unable to handle kernel NULL pointer dereference");
        else
            printk(KERN_ALERT "Unable to handle kernel paging request");
        printk(" at virtual address %08lx\n",address);
        printk(" printing eip:\n");
        printk("%08lx\n", regs->eip);
```

```
        asm("movl %%cr3,%0":"=r" (page));
        page = ((unsigned long *) __va(page))[address >> 22];
        printk(KERN_ALERT "*pde = %08lx\n", page);
        if (page & 1) {
                page &= PAGE_MASK;
                address &= 0x003ff000;
                page = ((unsigned long *) __va(page))[address >> PAGE_SHIFT];
                printk(KERN_ALERT "*pte = %08lx\n", page);
        }
        die("Oops", regs, error_code);
        bust_spinlocks(0);
        do_exit(SIGKILL);

/*
 * We ran out of memory, or some other thing happened to us that made
 * us unable to handle the page fault gracefully.
 */
out_of_memory:
        if (tsk->pid == 1) {
                yield();
                goto survive;
        }
        up_read(&mm->mmap_sem);
        printk("VM: killing process %s\n", tsk->comm);
        if (error_code & 4)
                do_exit(SIGKILL);
        goto no_context;

do_sigbus:
        up_read(&mm->mmap_sem);

        /*
         * Send a sigbus, regardless of whether we were in kernel
         * or user mode.
         */
        tsk->thread.cr2 = address;
        tsk->thread.error_code = error_code;
        tsk->thread.trap_no = 14;
        info.si_signo = SIGBUS;
        info.si_errno = 0;
        info.si_code = BUS_ADRERR;
        info.si_addr = (void *)address;
        force_sig_info(SIGBUS, &info, tsk);
```

```
        /* Kernel mode? Handle exceptions or die */
        if (!(error_code & 4))
                goto no_context;
        return;

vmalloc_fault:
        {
                /*
                 * Synchronize this task's top level page-table
                 * with the 'reference' page table.
                 *
                 * Do _not_ use "tsk" here. We might be inside
                 * an interrupt in the middle of a task switch..
                 */
                int offset = __pgd_offset(address);
                pgd_t *pgd, *pgd_k;
                pmd_t *pmd, *pmd_k;
                pte_t *pte_k;

                asm("movl %%cr3,%0":"=r" (pgd));
                pgd = offset + (pgd_t *)__va(pgd);
                pgd_k = init_mm.pgd + offset;

                if (!pgd_present(*pgd_k))
                        goto no_context;
                set_pgd(pgd, *pgd_k);

                pmd = pmd_offset(pgd, address);
                pmd_k = pmd_offset(pgd_k, address);
                if (!pmd_present(*pmd_k))
                        goto no_context;
                set_pmd(pmd, *pmd_k);

                pte_k = pte_offset(pmd_k, address);
                if (!pte_present(*pte_k))
                        goto no_context;
                return;
        }
}
```

代码比较长(如果你对现代操作系统是如何实现虚拟内存感兴趣的话，那实在应该看看这个函数)，但真正与该 bug 有关的就寥寥数行

```
/* get the address */
__asm__("movl %%cr2,%0":"=r" (address));
```

首先获得发生异常的地址 --- 从 CR2 register 获得。

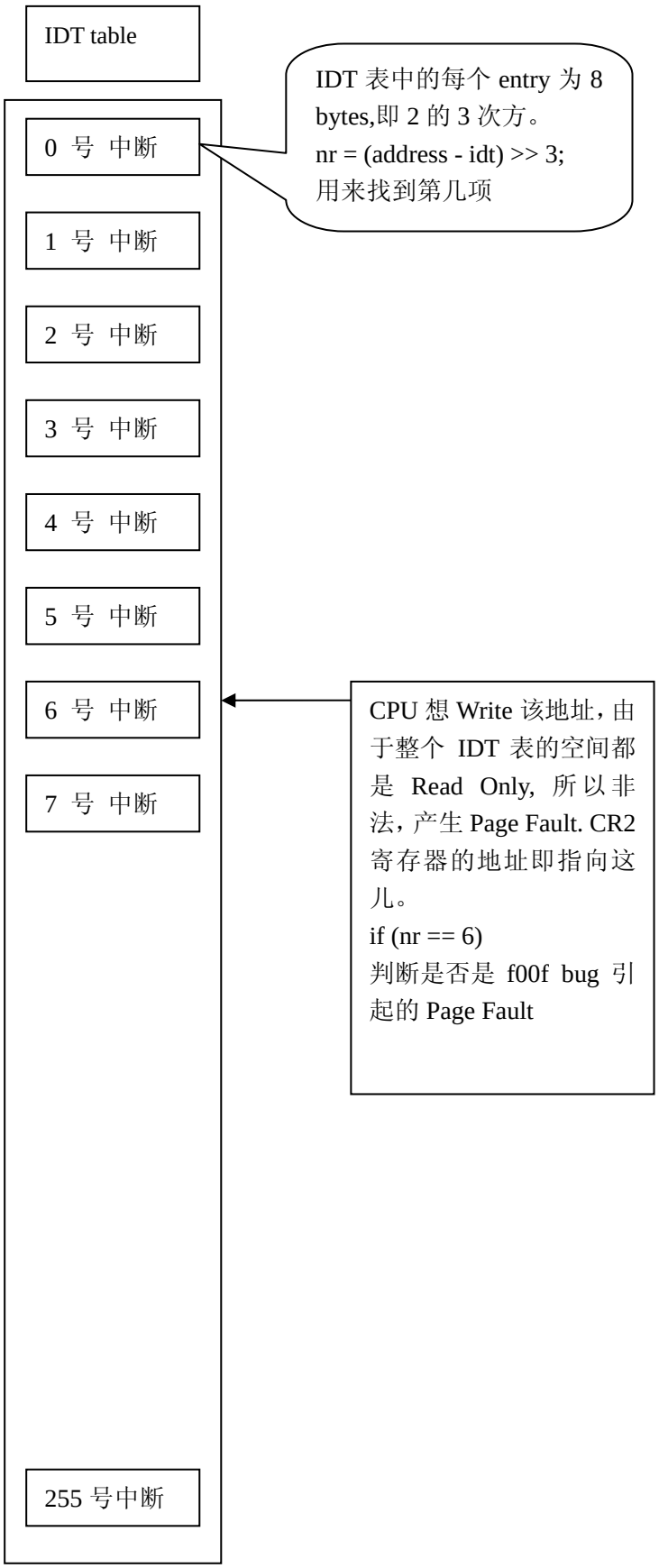
```
/*
 * Pentium F0 0F C7 C8 bug workaround.
 */
if (boot_cpu_data.f00f_bug) {
	unsigned long nr;

	nr = (address - idt) >> 3;

	if (nr == 6) {
		do_invalid_op(regs, 0);
		return;
	}
}
```

if (boot_cpu_data.f00f_bug) {

检查本 CPU 是否有该 bug,如果有，则检查该异常地址是否是 invalid opcode handler 在 idt table 中的地址，如果是则转调用 do_invalid_op（）。


IDT table
0 号 中断
1 号 中断
2 号 中断
3 号 中断
4 号 中断
5 号 中断
6 号 中断
7 号 中断
255 号中断
IDT 表中的每个 entry 为 8 bytes,即 2 的 3 次方。
nr = (address - idt) >> 3;
用来找到第几项
CPU 想 Write 该地址，由于整个 IDT 表的空间都是 Read Only, 所以非法，产生 Page Fault. CR2 寄存器的地址即指向这儿。
if (nr == 6)
判断是否是 f00f bug 引起的 Page Fault

Cyrix coma Bug --- Cryix 死锁

在 Cyrix 6x86CPU 执行下面的程序,会使整个计算机锁死.

```
static unsigned char c[4] = {0x36, 0x78, 0x38, 0x36}; //其实该数组中的值是
                                                      //什么倒是无所谓的

int main()
{
      __asm__("movl $c, %ebx\n\t"
      "1:xchgl (%ebx), %eax\n\t"
      "movl %eax, %edx\n\t"
      "jmp 1b\n\t"
      );

      return      0;
}
```

不要去追究上面代码是什么意思(本来这代码就没什么意思),只是说当碰到上面类似的指令时,CPU 会被锁死。可别忘了,上面的代码是在用户态执行的,而现代操作系统的一大要求就是用户态程序可以乱来,那是你的事,你做事,你负责,不应该影响到其他应用程序,更何况是操作系统本身。而上面的代码竟然可以搞死 OS,这绝对是什么地方出了错!

```
1          "1:xchgl (%ebx), %eax\n\t"
2          "movl %eax, %edx\n\t"
3          "jmp 1b\n\t"
```

寥寥 3 行代码,构成一个极短的无限循环,按我们常规的 x86 汇编知识,这段代码会使得该代码所在的应用进入无限循环,在 Unix OS 下的话只能通过 kill 来把它杀掉,但怎么会产生锁死 CPU 的后果呢?

让我们来分析一下

1 "1:xchgl (%ebx), %eax\n\t"

这行代码是把%ebx 所指向的 4 个 bytes 与%eax 中的值进行互换。这是一个即使在多 CPU 情况下也是安全的原子操作。

2 "movl %eax, %edx\n\t"

只是寄存器间的赋值,打死也不会有什么错

3 "jmp 1b\n\t"

它只是跳转到第一行去,而第一行的地址肯定是合法的。

第一行固然是原子操作,不能被打断,但在第一行与第二行之间,第二行与第三行之间及第三行跳转到第一行之间,CPU 都可以接受中断,timer 中断应该可以被打入,从而剥夺该应用的 CPU 时间,让其他应用有机会运行。

我们的分析是对的,但这都基于一个前提,该 CPU 是经典的 CISC 架构的 CPU,即指令的执行完全是串行的,执行完第一行,然后执行第二行,再执行第三行, ... 很遗憾,现在几乎找不到所谓"经典的 CISC 架构的 CPU"了,象那些经典的 RISC CPU,如 MIPS 等就不说了,即使是 CISC CPU 的典型 Intel, Cyrix, AMD 等都已经融入 RISC 设计。也就是大家肯定听说过(报纸上,广告里)的,如 pipelining(流水线),banch predicting(分支预测)等,但不一定很明白其意思,或者即使明白其意思,但不一定估计到它的影响所及。

而该 BUG 就是深受如 Pipelining 与 Banch Predicting 之影响。如果没有这两者,那上面的代码一点问题也没有.让我们用 RISC 的眼光来分析一下上面三行代码:

在执行 xchgl (%ebx), %eax 时的中后期过程(一条指令的执行很粗略的分可以分为取指令,解码指令,执行指令,这是非常非常粗的划分,很不科学,如果你想了解细分过程,应当去看某些 RISC CPU 的手册),CPU 实际上也依此启动了下面的两条指令的执行过程(Pipelining),而 jmp 1b 是启动了 Banch Predicting---其实根本不需要 predicting,因为这是绝对跳转。原本 xchgl (%ebx), %eax 指令是原子操作,即在该指令的执行期是不可打断的,类似于在该指令的整个执行其间,伴随有 cli (clear interrupt flag)的效果。但关键是由于 Pipelining 和 Banch Predicting 造成的并行性,对 xchgl (%ebx), %eax 指令的不可打断性超过了本行指令的范围。它把 1,2,3 行代码全部覆盖,使得这 3 行代码成为一个整体,不可被中断。而可悲的是,该整体又是一个不会转出来的无限循环。结果就是 CPU 一直在那儿执行这三行指令,拒绝一切中断,包括 timer 中断。

对该 Bug,Cyrix 给出的解决方案是对 Cyrix CPU 的 CCR1 寄存器(Cyrix CPU 特有的寄存器,其它 x86 CPU 没有)的位 0x10 置位,实际上就是串行化执行 xchg 指令,不要 Pipelining 了.无奈之举.

Linux 对待该 Bug,有点象对待 Intel 的 FDIV Bug,只是在检测到是 Cyrix 6x86 CPU 的情况下,对用户发出警告而已。

在 arch/i386/kernel/setup.c 中的 init_cyrix () 函数中,有如下代码片断

```
/* 6x86's contain this bug */
c->coma_bug = 1;
```

然后再 show_cpuinfo () 函数中打印出警告

```
seq_printf(m, "fdiv_bug\t: %s\n"
              "hlt_bug\t\t: %s\n"
              "f00f_bug\t: %s\n"
              "coma_bug\t: %s\n"
              "fpu\t\t: %s\n"
              "fpu_exception\t: %s\n"
              "cpuid level\t: %d\n"
              "wp\t\t: %s\n"
              "flags\t\t:",
               c->fdiv_bug ? "yes" : "no",
               c->hlt_works_ok ? "no" : "yes",
               c->f00f_bug ? "yes" : "no",
               c->coma_bug ? "yes" : "no",
               c->hard_math ? "yes" : "no",
               fpu_exception ? "yes" : "no",
               c->cpuid_level,
               c->wp_works_ok ? "yes" : "no");
```

如果你的 CPU 是 Cyrix 6x86 的话,你可以通过 dmesg 命令看到此警告.

连我们一向信赖的 CPU 都有 Bug,我们软件开发人员也廖可自慰： " 何况软件呢 " !^_^

附录：

本文摘录的 Linux Kernel 代码基于 2.4.20 内核。参考 Intel 奔腾处理器手册。

2007-3-8， 23 : 00

Walter Zhou
zhoulong@sh163.net