

[PATCH 9/9] traps: x86: finalize unification of traps.c

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2008-10/msg01257.html>

- *From:* Alexander van Heukelum <heukelum@xxxxxxxxxxx>
 - *Date:* Fri, 3 Oct 2008 22:00:40 +0200
-

traps_32.c and traps_64.c are now equal. Move one to traps.c,
delete the other one and change the Makefile

Signed-off-by: Alexander van Heukelum <heukelum@xxxxxxxxxxx>

```
----
arch/x86/kernel/Makefile | 2 +-
arch/x86/kernel/traps.c | 1075 ++++++++++++++++++++++++++++++++++++++
arch/x86/kernel/traps_32.c | 1075 -----
arch/x86/kernel/traps_64.c | 1075 -----
4 files changed, 1076 insertions(+), 2151 deletions(-)
create mode 100644 arch/x86/kernel/traps.c
delete mode 100644 arch/x86/kernel/traps_32.c
delete mode 100644 arch/x86/kernel/traps_64.c
```

```
diff --git a/arch/x86/kernel/Makefile b/arch/x86/kernel/Makefile
index 2a5f58f..1e428e0 100644
--- a/arch/x86/kernel/Makefile
+++ b/arch/x86/kernel/Makefile
@@ -24,7 +24,7 @@ CFLAGS_tsc.o := $(nostackp)
CFLAGS_paravirt.o := $(nostackp)
```

```
obj-y := process_$(BITS).o signal_$(BITS).o entry_$(BITS).o
-obj-y += traps_$(BITS).o irq_$(BITS).o dumpstack_$(BITS).o
+obj-y += traps.o irq_$(BITS).o dumpstack_$(BITS).o
obj-y += time_$(BITS).o ioport.o ldt.o
obj-y += setup.o i8259.o irqinit_$(BITS).o setup_percpu.o
obj-$(CONFIG_X86_VISWS) += visws_quirks.o
diff --git a/arch/x86/kernel/traps.c b/arch/x86/kernel/traps.c
new file mode 100644
index 0000000..54e08d2
--- /dev/null
+++ b/arch/x86/kernel/traps.c
@@ -0,0 +1,1075 @@
+/*
+ * Copyright (C) 1991, 1992 Linus Torvalds
+ * Copyright (C) 2000, 2001, 2002 Andi Kleen, SuSE Labs
+ *
+ * Pentium III FXSR, SSE support
```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
+ * Gareth Hughes <gareth@xxxxxxxxxxxx>, May 2000
+ */
+
+/*
+ * Handle hardware traps and faults.
+ */
+#include <linux/interrupt.h>
+#include <linux/kallsyms.h>
+#include <linux/spinlock.h>
+#include <linux/kprobes.h>
+#include <linux/uaccess.h>
+#include <linux/utsname.h>
+#include <linux/kdebug.h>
+#include <linux/kernel.h>
+#include <linux/module.h>
+#include <linux/ptrace.h>
+#include <linux/string.h>
+#include <linux/unwind.h>
+#include <linux/delay.h>
+#include <linux/errno.h>
+#include <linux/kexec.h>
+#include <linux/sched.h>
+#include <linux/timer.h>
+#include <linux/init.h>
+#include <linux/bug.h>
+#include <linux/nmi.h>
+#include <linux/mm.h>
+#include <linux/smp.h>
+#include <linux/io.h>
+
+#ifdef CONFIG_EISA
+#include <linux/ioport.h>
+#include <linux/eisa.h>
+#endif
+
+#ifdef CONFIG_MCA
+#include <linux/mca.h>
+#endif
+
+#if defined(CONFIG_EDAC)
+#include <linux/edac.h>
+#endif
+
+#include <asm/stacktrace.h>
+#include <asm/processor.h>
+#include <asm/kmemcheck.h>
+#include <asm/debugreg.h>
+#include <asm/atomic.h>
+#include <asm/system.h>
+#include <asm/unwind.h>
+#include <asm/traps.h>
```

```

#include <asm/desc.h>
#include <asm/i387.h>
+
#include <mach_traps.h>
+
#ifdef CONFIG_X86_64
#include <asm/pgalloc.h>
#include <asm/proto.h>
#include <asm/pda.h>
#else
#include <asm/processor-flags.h>
#include <asm/arch_hooks.h>
#include <asm/nmi.h>
#include <asm/smp.h>
#include <asm/io.h>
+
#include "cpu/mcheck/mce.h"
+
DECLARE_BITMAP(used_vectors, NR_VECTORS);
EXPORT_SYMBOL_GPL(used_vectors);
+
asmlinkage int system_call(void);
+
/* Do we ignore FPU interrupts ? */
char ignore_fpu_irq;
+
/*
 * The IDT has to be page-aligned to simplify the Pentium
 * F0 0F bug workaround.. We have a special link segment
 * for this.
 */
gate_desc idt_table[256]
+ __attribute__((__section__(".data.idt"))) = { { { { 0, 0 } } }, };
#endif
+
static int ignore_nmis;
+
static inline void conditional_sti(struct pt_regs *regs)
+{
+ if (regs->flags & X86_EFLAGS_IF)
+ local_irq_enable();
+}
+
static inline void preempt_conditional_sti(struct pt_regs *regs)
+{
+ inc_preempt_count();
+ if (regs->flags & X86_EFLAGS_IF)
+ local_irq_enable();
+}
+
static inline void preempt_conditional_cli(struct pt_regs *regs)

```

```

+{
+ if (regs->flags & X86_EFLAGS_IF)
+ local_irq_disable();
+ dec_preempt_count();
+}
+
+#ifdef CONFIG_X86_32
+static inline void
+die_if_kernel(const char *str, struct pt_regs *regs, long err)
+{
+ if (!user_mode_vm(regs))
+ die(str, regs, err);
+}
+
+/*
+ * Perform the lazy TSS's I/O bitmap copy. If the TSS has an
+ * invalid offset set (the LAZY one) and the faulting thread has
+ * a valid I/O bitmap pointer, we copy the I/O bitmap in the TSS,
+ * we set the offset field correctly and return 1.
+ */
+static int lazy_iobitmap_copy(void)
+{
+ struct thread_struct *thread;
+ struct tss_struct *tss;
+ int cpu;
+
+ cpu = get_cpu();
+ tss = &per_cpu(init_tss, cpu);
+ thread = &current->thread;
+
+ if (tss->x86_tss.io_bitmap_base == INVALID_IO_BITMAP_OFFSET_LAZY &&
+ thread->io_bitmap_ptr) {
+ memcpy(tss->io_bitmap, thread->io_bitmap_ptr,
+ thread->io_bitmap_max);
+ /*
+ * If the previously set map was extending to higher ports
+ * than the current one, pad extra space with 0xff (no access).
+ */
+ if (thread->io_bitmap_max < tss->io_bitmap_max) {
+ memset((char *) tss->io_bitmap +
+ thread->io_bitmap_max, 0xff,
+ tss->io_bitmap_max - thread->io_bitmap_max);
+ }
+ tss->io_bitmap_max = thread->io_bitmap_max;
+ tss->x86_tss.io_bitmap_base = IO_BITMAP_OFFSET;
+ tss->io_bitmap_owner = thread;
+ put_cpu();
+
+ return 1;
+ }
+ put_cpu();

```

```

+
+ return 0;
+}
+#endif
+
+static void __kprobes
+do_trap(int trapnr, int signr, char *str, struct pt_regs *regs,
+ long error_code, signinfo_t *info)
+{
+ struct task_struct *tsk = current;
+
+ #ifdef CONFIG_X86_32
+ if (regs->flags & X86_VM_MASK) {
+ /*
+ * traps 0, 1, 3, 4, and 5 should be forwarded to vm86.
+ * On nmi (interrupt 2), do_trap should not be called.
+ */
+ if (trapnr < 6)
+ goto vm86_trap;
+ goto trap_signal;
+ }
+ #endif
+
+ if (!user_mode(regs))
+ goto kernel_trap;
+
+ #ifdef CONFIG_X86_32
+ trap_signal:
+ #endif
+ /*
+ * We want error_code and trap_no set for userspace faults and
+ * kernelspace faults which result in die(), but not
+ * kernelspace faults which are fixed up. die() gives the
+ * process no chance to handle the signal and notice the
+ * kernel fault information, so that won't result in polluting
+ * the information about previously queued, but not yet
+ * delivered, faults. See also do_general_protection below.
+ */
+ tsk->thread.error_code = error_code;
+ tsk->thread.trap_no = trapnr;
+
+ #ifdef CONFIG_X86_64
+ if (show_unhandled_signals && unhandled_signal(tsk, signr) &&
+ printk_ratelimit()) {
+ printk(KERN_INFO
+ "%s[%d] trap %s ip:%lx sp:%lx error:%lx",
+ tsk->comm, tsk->pid, str,
+ regs->ip, regs->sp, error_code);
+ print_vma_addr(" in ", regs->ip);
+ printk("\n");
+ }
+ }

```

```

+ #endif
+
+ if (info)
+ force_sig_info(signr, info, tsk);
+ else
+ force_sig(signr, tsk);
+ return;
+
+ kernel_trap:
+ if (!fixup_exception(regs)) {
+ tsk->thread.error_code = error_code;
+ tsk->thread.trap_no = trapnr;
+ die(str, regs, error_code);
+ }
+ return;
+
+ #ifdef CONFIG_X86_32
+ vm86_trap:
+ if (handle_vm86_trap((struct kernel_vm86_regs *) regs,
+ error_code, trapnr))
+ goto trap_signal;
+ return;
+ #endif
+ }
+
+ #define DO_ERROR(trapnr, signr, str, name) \
+ dotraplinkage void do_###name(struct pt_regs *regs, long error_code) \
+ { \
+ if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) \
+ == NOTIFY_STOP) \
+ return; \
+ conditional_sti(regs); \
+ do_trap(trapnr, signr, str, regs, error_code, NULL); \
+ }
+
+ #define DO_ERROR_INFO(trapnr, signr, str, name, sicode, siaddr) \
+ dotraplinkage void do_###name(struct pt_regs *regs, long error_code) \
+ { \
+ siginfo_t info; \
+ info.si_signo = signr; \
+ info.si_errno = 0; \
+ info.si_code = sicode; \
+ info.si_addr = (void __user *)siaddr; \
+ if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) \
+ == NOTIFY_STOP) \
+ return; \
+ conditional_sti(regs); \
+ do_trap(trapnr, signr, str, regs, error_code, &info); \
+ }
+
+ DO_ERROR_INFO(0, SIGFPE, "divide error", divide_error, FPE_INTDIV, regs->ip)

```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
+DO_ERROR(4, SIGSEGV, "overflow", overflow)
+DO_ERROR(5, SIGSEGV, "bounds", bounds)
+DO_ERROR_INFO(6, SIGILL, "invalid opcode", invalid_op, ILL_ILLOPN, regs->ip)
+DO_ERROR(9, SIGFPE, "coprocessor segment overrun", coprocessor_segment_overrun)
+DO_ERROR(10, SIGSEGV, "invalid TSS", invalid_TSS)
+DO_ERROR(11, SIGBUS, "segment not present", segment_not_present)
+#ifdef CONFIG_X86_32
+DO_ERROR(12, SIGBUS, "stack segment", stack_segment)
+#endif
+DO_ERROR_INFO(17, SIGBUS, "alignment check", alignment_check, BUS_ADRALN, 0)
+
+#ifdef CONFIG_X86_64
+/* Runs on IST stack */
+dotraplinkage void do_stack_segment(struct pt_regs *regs, long error_code)
+{
+ if (notify_die(DIE_TRAP, "stack segment", regs, error_code,
+ 12, SIGBUS) == NOTIFY_STOP)
+ return;
+ preempt_conditional_sti(regs);
+ do_trap(12, SIGBUS, "stack segment", regs, error_code, NULL);
+ preempt_conditional_cli(regs);
+}
+
+dotraplinkage void do_double_fault(struct pt_regs *regs, long error_code)
+{
+ static const char str[] = "double fault";
+ struct task_struct *tsk = current;
+
+ /* Return not checked because double check cannot be ignored */
+ notify_die(DIE_TRAP, str, regs, error_code, 8, SIGSEGV);
+
+ tsk->thread.error_code = error_code;
+ tsk->thread.trap_no = 8;
+
+ /* This is always a kernel trap and never fixable (and thus must
+ never return). */
+ for (;;)
+ die(str, regs, error_code);
+}
+#endif
+
+dotraplinkage void __kprobes
+do_general_protection(struct pt_regs *regs, long error_code)
+{
+ struct task_struct *tsk;
+
+ conditional_sti(regs);
+
+#ifdef CONFIG_X86_32
+ if (lazy_iobitmap_copy()) {
+ /* restart the faulting instruction */
```

```

+ return;
+ }
+
+ if (regs->flags & X86_VM_MASK)
+ goto gp_in_vm86;
+ #endif
+
+ tsk = current;
+ if (!user_mode(regs))
+ goto gp_in_kernel;
+
+ tsk->thread.error_code = error_code;
+ tsk->thread.trap_no = 13;
+
+ if (show_unhandled_signals && unhandled_signal(tsk, SIGSEGV) &&
+ printk_ratelimit()) {
+ printk(KERN_INFO
+ "%s[%d] general protection ip:%lx sp:%lx error:%lx",
+ tsk->comm, task_pid_nr(tsk),
+ regs->ip, regs->sp, error_code);
+ print_vma_addr(" in ", regs->ip);
+ printk("\n");
+ }
+
+ force_sig(SIGSEGV, tsk);
+ return;
+
+ #ifdef CONFIG_X86_32
+ gp_in_vm86:
+ local_irq_enable();
+ handle_vm86_fault((struct kernel_vm86_regs *) regs, error_code);
+ return;
+ #endif
+
+ gp_in_kernel:
+ if (fixup_exception(regs))
+ return;
+
+ tsk->thread.error_code = error_code;
+ tsk->thread.trap_no = 13;
+ if (notify_die(DIE_GPF, "general protection fault", regs,
+ error_code, 13, SIGSEGV) == NOTIFY_STOP)
+ return;
+ die("general protection fault", regs, error_code);
+ }
+
+ static notrace __kprobes void
+ mem_parity_error(unsigned char reason, struct pt_regs *regs)
+ {
+ printk(KERN_EMERG
+ "Uhhuh. NMI received for unknown reason %02x on CPU %d.\n",

```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
+ reason, smp_processor_id());
+
+ printk(KERN_EMERG
+ "You have some hardware problem, likely on the PCI bus.\n");
+
+ #if defined(CONFIG_EDAC)
+ if (edac_handler_set()) {
+ edac_atomic_assert_error();
+ return;
+ }
+ #endif
+
+ if (panic_on_unrecovered_nmi)
+ panic("NMI: Not continuing");
+
+ printk(KERN_EMERG "Dazed and confused, but trying to continue\n");
+
+ /* Clear and disable the memory parity error line. */
+ reason = (reason & 0xf) | 4;
+ outb(reason, 0x61);
+ }
+
+ static notrace __kprobes void
+ io_check_error(unsigned char reason, struct pt_regs *regs)
+ {
+ unsigned long i;
+
+ printk(KERN_EMERG "NMI: IOCK error (debug interrupt?)\n");
+ show_registers(regs);
+
+ /* Re-enable the IOCK line, wait for a few seconds */
+ reason = (reason & 0xf) | 8;
+ outb(reason, 0x61);
+
+ i = 2000;
+ while (--i)
+ udelay(1000);
+
+ reason &= ~8;
+ outb(reason, 0x61);
+ }
+
+ static notrace __kprobes void
+ unknown_nmi_error(unsigned char reason, struct pt_regs *regs)
+ {
+ if (notify_die(DIE_NMIUNKNOWN, "nmi", regs, reason, 2, SIGINT) ==
+ NOTIFY_STOP)
+ return;
+ #ifdef CONFIG_MCA
+ /*
+ * Might actually be able to figure out what the guilty party
```

```

+ * is:
+ */
+ if (MCA_bus) {
+ mca_handle_nmi();
+ return;
+ }
+ #endif
+ printk(KERN_EMERG
+ "Uhhuh. NMI received for unknown reason %02x on CPU %d.\n",
+ reason, smp_processor_id());
+
+ printk(KERN_EMERG "Do you have a strange power saving mode enabled?\n");
+ if (panic_on_unrecovered_nmi)
+ panic("NMI: Not continuing");
+
+ printk(KERN_EMERG "Dazed and confused, but trying to continue\n");
+ }
+
+ #ifdef CONFIG_X86_32
+ static DEFINE_SPINLOCK(nmi_print_lock);
+
+ void notrace __kprobes die_nmi(char *str, struct pt_regs *regs, int do_panic)
+ {
+ if (notify_die(DIE_NMIWATCHDOG, str, regs, 0, 2, SIGINT) == NOTIFY_STOP)
+ return;
+
+ spin_lock(&nmi_print_lock);
+ /*
+ * We are in trouble anyway, lets at least try
+ * to get a message out:
+ */
+ bust_spinlocks(1);
+ printk(KERN_EMERG "%s", str);
+ printk(" on CPU%d, ip %08lx, registers:\n",
+ smp_processor_id(), regs->ip);
+ show_registers(regs);
+ if (do_panic)
+ panic("Non maskable interrupt");
+ console_silent();
+ spin_unlock(&nmi_print_lock);
+ bust_spinlocks(0);
+
+ /*
+ * If we are in kernel we are probably nested up pretty bad
+ * and might aswell get out now while we still can:
+ */
+ if (!user_mode_vm(regs)) {
+ current->thread.trap_no = 2;
+ crash_kexec(regs);
+ }
+
+

```

```

+ do_exit(SIGSEGV);
+ }
+ #endif
+
+ static notrace __kprobes void default_do_nmi(struct pt_regs *regs)
+ {
+   unsigned char reason = 0;
+   int cpu;
+
+   cpu = smp_processor_id();
+
+   /* Only the BSP gets external NMIs from the system. */
+   if (!cpu)
+     reason = get_nmi_reason();
+
+   if (!(reason & 0xc0)) {
+     if (notify_die(DIE_NMI_IPI, "nmi_ipi", regs, reason, 2, SIGINT)
+         == NOTIFY_STOP)
+       return;
+     #ifdef CONFIG_X86_LOCAL_APIC
+     /*
+      * Ok, so this is none of the documented NMI sources,
+      * so it must be the NMI watchdog.
+      */
+     if (nmi_watchdog_tick(regs, reason))
+       return;
+     if (!do_nmi_callback(regs, cpu))
+       unknown_nmi_error(reason, regs);
+   #else
+     unknown_nmi_error(reason, regs);
+   #endif
+
+   return;
+ }
+ if (notify_die(DIE_NMI, "nmi", regs, reason, 2, SIGINT) == NOTIFY_STOP)
+   return;
+
+ /* AK: following checks seem to be broken on modern chipsets. FIXME */
+ if (reason & 0x80)
+   mem_parity_error(reason, regs);
+ if (reason & 0x40)
+   io_check_error(reason, regs);
+ #ifdef CONFIG_X86_32
+ /*
+  * Reassert NMI in case it became active meanwhile
+  * as it's edge-triggered:
+  */
+   reassert_nmi();
+ #endif
+ }
+

```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
+dotraplinkage notrace __kprobes void
+do_nmi(struct pt_regs *regs, long error_code)
+{
+ nmi_enter();
+
+#ifdef CONFIG_X86_32
+ { int cpu; cpu = smp_processor_id(); ++nmi_count(cpu); }
+#else
+ add_pda(__nmi_count, 1);
+#endif
+
+ if (!ignore_nmis)
+ default_do_nmi(regs);
+
+ nmi_exit();
+}
+
+void stop_nmi(void)
+{
+ acpi_nmi_disable();
+ ignore_nmis++;
+}
+
+void restart_nmi(void)
+{
+ ignore_nmis--;
+ acpi_nmi_enable();
+}
+
+/* May run on IST stack. */
+dotraplinkage void __kprobes do_int3(struct pt_regs *regs, long error_code)
+{
+#ifdef CONFIG_KPROBES
+ if (notify_die(DIE_INT3, "int3", regs, error_code, 3, SIGTRAP)
+ == NOTIFY_STOP)
+ return;
+#else
+ if (notify_die(DIE_TRAP, "int3", regs, error_code, 3, SIGTRAP)
+ == NOTIFY_STOP)
+ return;
+#endif
+
+ preempt_conditional_sti(regs);
+ do_trap(3, SIGTRAP, "int3", regs, error_code, NULL);
+ preempt_conditional_cli(regs);
+}
+
+#ifdef CONFIG_X86_64
+/* Help handler running on IST stack to switch back to user stack
+ for scheduling or signal handling. The actual stack switch is done in
+ entry.S */
```

```

+asmlinkage __kprobes struct pt_regs *sync_regs(struct pt_regs *eregs)
+{
+ struct pt_regs *regs = eregs;
+ /* Did already sync */
+ if (eregs == (struct pt_regs *)eregs->sp)
+ ;
+ /* Exception from user space */
+ else if (user_mode(eregs))
+ regs = task_pt_regs(current);
+ /* Exception from kernel and interrupts are enabled. Move to
+ kernel process stack. */
+ else if (eregs->flags & X86_EFLAGS_IF)
+ regs = (struct pt_regs *) (eregs->sp - sizeof(struct pt_regs));
+ if (eregs != regs)
+ *regs = *eregs;
+ return regs;
+}
+#endif
+
+/*
+ * Our handling of the processor debug registers is non-trivial.
+ * We do not clear them on entry and exit from the kernel. Therefore
+ * it is possible to get a watchpoint trap here from inside the kernel.
+ * However, the code in ./ptrace.c has ensured that the user can
+ * only set watchpoints on userspace addresses. Therefore the in-kernel
+ * watchpoint trap can only occur in code which is reading/writing
+ * from user space. Such code must not hold kernel locks (since it
+ * can equally take a page fault), therefore it is safe to call
+ * force_sig_info even though that claims and releases locks.
+ *
+ * Code in ./signal.c ensures that the debug control register
+ * is restored before we deliver any signal, and therefore that
+ * user code runs with the correct debug control register even though
+ * we clear it here.
+ *
+ * Being careful here means that we don't have to be as careful in a
+ * lot of more complicated places (task switching can be a bit lazy
+ * about restoring all the debug state, and ptrace doesn't have to
+ * find every occurrence of the TF bit that could be saved away even
+ * by user code)
+ *
+ * May run on IST stack.
+ */
+dotraplinkage void __kprobes do_debug(struct pt_regs *regs, long error_code)
+{
+ struct task_struct *tsk = current;
+ unsigned long condition;
+ int si_code;
+
+ get_debugreg(condition, 6);
+

```

```

+ /* Catch kmemcheck conditions first of all! */
+ if (condition & DR_STEP && kmemcheck_trap(regs))
+ return;
+
+ /*
+ * The processor cleared BTF, so don't mark that we need it set.
+ */
+ clear_tsk_thread_flag(tsk, TIF_DEBUGCTLMSR);
+ tsk->thread.debugctlmsr = 0;
+
+ if (notify_die(DIE_DEBUG, "debug", regs, condition, error_code,
+ SIGTRAP) == NOTIFY_STOP)
+ return;
+
+ /* It's safe to allow irq's after DR6 has been saved */
+ preempt_conditional_sti(regs);
+
+ /* Mask out spurious debug traps due to lazy DR7 setting */
+ if (condition & (DR_TRAP0|DR_TRAP1|DR_TRAP2|DR_TRAP3)) {
+ if (!tsk->thread.debugreg7)
+ goto clear_dr7;
+ }
+
+ #ifdef CONFIG_X86_32
+ if (regs->flags & X86_VM_MASK)
+ goto debug_vm86;
+ #endif
+
+ /* Save debug status register where ptrace can see it */
+ tsk->thread.debugreg6 = condition;
+
+ /*
+ * Single-stepping through TF: make sure we ignore any events in
+ * kernel space (but re-enable TF when returning to user mode).
+ */
+ if (condition & DR_STEP) {
+ if (!user_mode(regs))
+ goto clear_TF_reenable;
+ }
+
+ si_code = get_si_code(condition);
+ /* Ok, finally something we can handle */
+ send_sigtrap(tsk, regs, error_code, si_code);
+
+ /*
+ * Disable additional traps. They'll be re-enabled when
+ * the signal is delivered.
+ */
+ clear_dr7:
+ set_debugreg(0, 7);
+ preempt_conditional_cli(regs);

```

```

+ return;
+
+#ifdef CONFIG_X86_32
+debug_vm86:
+ handle_vm86_trap((struct kernel_vm86_regs *) regs, error_code, 1);
+ preempt_conditional_cli(regs);
+ return;
+#endif
+
+clear_TF_reenable:
+ set_tsk_thread_flag(tsk, TIF_SINGLESTEP);
+ regs->flags &= ~X86_EFLAGS_TF;
+ preempt_conditional_cli(regs);
+ return;
+}
+
+#ifdef CONFIG_X86_64
+static int kernel_math_error(struct pt_regs *regs, const char *str, int trapnr)
+{
+ if (fixup_exception(regs))
+ return 1;
+
+ notify_die(DIE_GPF, str, regs, 0, trapnr, SIGFPE);
+ /* Illegal floating point operation in the kernel */
+ current->thread.trap_no = trapnr;
+ die(str, regs, 0);
+ return 0;
+}
+#endif
+
+/*
+ * Note that we play around with the 'TS' bit in an attempt to get
+ * the correct behaviour even in the presence of the asynchronous
+ * IRQ13 behaviour
+ */
+void math_error(void __user *ip)
+{
+ struct task_struct *task;
+ siginfo_t info;
+ unsigned short cwd, swd;
+
+ /*
+ * Save the info for the exception handler and clear the error.
+ */
+ task = current;
+ save_init_fpu(task);
+ task->thread.trap_no = 16;
+ task->thread.error_code = 0;
+ info.si_signo = SIGFPE;
+ info.si_errno = 0;
+ info.si_code = __SI_FAULT;

```

```

+ info.si_addr = ip;
+ /*
+ * (~cwd & swd) will mask out exceptions that are not set to unmasked
+ * status. 0x3f is the exception bits in these regs, 0x200 is the
+ * C1 reg you need in case of a stack fault, 0x040 is the stack
+ * fault bit. We should only be taking one exception at a time,
+ * so if this combination doesn't produce any single exception,
+ * then we have a bad program that isn't synchronizing its FPU usage
+ * and it will suffer the consequences since we won't be able to
+ * fully reproduce the context of the exception
+ */
+ cwd = get_fpu_cwd(task);
+ swd = get_fpu_swd(task);
+ switch (swd & ~cwd & 0x3f) {
+ case 0x000: /* No unmasked exception */
+ #ifdef CONFIG_X86_32
+ return;
+ #endif
+ default: /* Multiple exceptions */
+ break;
+ case 0x001: /* Invalid Op */
+ /*
+ * swd & 0x240 == 0x040: Stack Underflow
+ * swd & 0x240 == 0x240: Stack Overflow
+ * User must clear the SF bit (0x40) if set
+ */
+ info.si_code = FPE_FLTINV;
+ break;
+ case 0x002: /* Denormalize */
+ case 0x010: /* Underflow */
+ info.si_code = FPE_FLTUND;
+ break;
+ case 0x004: /* Zero Divide */
+ info.si_code = FPE_FLTDIV;
+ break;
+ case 0x008: /* Overflow */
+ info.si_code = FPE_FLTOVF;
+ break;
+ case 0x020: /* Precision */
+ info.si_code = FPE_FLTRES;
+ break;
+ }
+ force_sig_info(SIGFPE, &info, task);
+ }
+
+ dotraplinkage void do_coprocessor_error(struct pt_regs *regs, long error_code)
+ {
+ conditional_sti(regs);
+
+ #ifdef CONFIG_X86_32
+ ignore_fpu_irq = 1;

```

```

+ #else
+ if (!user_mode(regs) &&
+ kernel_math_error(regs, "kernel x87 math error", 16))
+ return;
+ #endif
+
+ math_error((void __user *)regs->ip);
+ }
+
+ static void simd_math_error(void __user *ip)
+ {
+ struct task_struct *task;
+ siginfo_t info;
+ unsigned short mxcsr;
+
+ /*
+ * Save the info for the exception handler and clear the error.
+ */
+ task = current;
+ save_init_fpu(task);
+ task->thread.trap_no = 19;
+ task->thread.error_code = 0;
+ info.si_signo = SIGFPE;
+ info.si_errno = 0;
+ info.si_code = __SI_FAULT;
+ info.si_addr = ip;
+ /*
+ * The SIMD FPU exceptions are handled a little differently, as there
+ * is only a single status/control register. Thus, to determine which
+ * unmasked exception was caught we must mask the exception mask bits
+ * at 0x1f80, and then use these to mask the exception bits at 0x3f.
+ */
+ mxcsr = get_fpu_mxcsr(task);
+ switch (~((mxcsr & 0x1f80) >> 7) & (mxcsr & 0x3f)) {
+ case 0x000:
+ default:
+ break;
+ case 0x001: /* Invalid Op */
+ info.si_code = FPE_FLTINV;
+ break;
+ case 0x002: /* Denormalize */
+ case 0x010: /* Underflow */
+ info.si_code = FPE_FLTUND;
+ break;
+ case 0x004: /* Zero Divide */
+ info.si_code = FPE_FLTDIV;
+ break;
+ case 0x008: /* Overflow */
+ info.si_code = FPE_FLTOVF;
+ break;
+ case 0x020: /* Precision */

```

```

+ info.si_code = FPE_FLTRES;
+ break;
+ }
+ force_sig_info(SIGFPE, &info, task);
+}
+
+dotraplinkage void
+do_simd_coprocessor_error(struct pt_regs *regs, long error_code)
+{
+ conditional_sti(regs);
+
+ #ifdef CONFIG_X86_32
+ if (cpu_has_xmm) {
+ /* Handle SIMD FPU exceptions on PIII+ processors. */
+ ignore_fpu_irq = 1;
+ simd_math_error((void __user *)regs->ip);
+ return;
+ }
+ /*
+ * Handle strange cache flush from user space exception
+ * in all other cases. This is undocumented behaviour.
+ */
+ if (regs->flags & X86_VM_MASK) {
+ handle_vm86_fault((struct kernel_vm86_regs *)regs, error_code);
+ return;
+ }
+ current->thread.trap_no = 19;
+ current->thread.error_code = error_code;
+ die_if_kernel("cache flush denied", regs, error_code);
+ force_sig(SIGSEGV, current);
+ #else
+ if (!user_mode(regs) &&
+ kernel_math_error(regs, "kernel simd math error", 19))
+ return;
+ simd_math_error((void __user *)regs->ip);
+ #endif
+}
+
+dotraplinkage void
+do_spurious_interrupt_bug(struct pt_regs *regs, long error_code)
+{
+ conditional_sti(regs);
+ #if 0
+ /* No need to warn about this any longer. */
+ printk(KERN_INFO "Ignoring P6 Local APIC Spurious Interrupt Bug...\n");
+ #endif
+}
+
+ #ifdef CONFIG_X86_32
+ unsigned long patch_esprefix_desc(unsigned long uesp, unsigned long kesp)
+ {

```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
+ struct desc_struct *gdt = get_cpu_gdt_table(smp_processor_id());
+ unsigned long base = (kesp - uesp) & -THREAD_SIZE;
+ unsigned long new_ksesp = kesp - base;
+ unsigned long lim_pages = (new_ksesp | (THREAD_SIZE - 1)) >> PAGE_SHIFT;
+ __u64 desc = *(__u64 *)&gdt[GDT_ENTRY_ESPFIX_SS];
+
+ /* Set up base for espfix segment */
+ desc &= 0x00f0ff0000000000ULL;
+ desc |= (((__u64)base) << 16) & 0x000000ffffff0000ULL |
+ (((__u64)base) << 32) & 0xff00000000000000ULL |
+ (((__u64)lim_pages) << 32) & 0x000f000000000000ULL |
+ (lim_pages & 0xffff);
+ *(__u64 *)&gdt[GDT_ENTRY_ESPFIX_SS] = desc;
+
+ return new_ksesp;
+}
+
+#else
+asmlinkage void __attribute__((weak)) smp_thermal_interrupt(void)
+{
+}
+
+asmlinkage void __attribute__((weak)) mce_threshold_interrupt(void)
+{
+}
+
+#endif
+
+/*
+ * 'math_state_restore()' saves the current math information in the
+ * old math state array, and gets the new ones from the current task
+ *
+ * Careful.. There are problems with IBM-designed IRQ13 behaviour.
+ * Don't touch unless you *really* know how it works.
+ *
+ * Must be called with kernel preemption disabled (in this case,
+ * local interrupts are disabled at the call-site in entry.S).
+ */
+asmlinkage void math_state_restore(void)
+{
+ struct thread_info *thread = current_thread_info();
+ struct task_struct *tsk = thread->task;
+
+ if (!tsk_used_math(tsk)) {
+ local_irq_enable();
+ /*
+ * does a slab alloc which can sleep
+ */
+ if (init_fpu(tsk)) {
+ /*
+ * ran out of memory!
+ */
+ do_group_exit(SIGKILL);

```

```

+ return;
+ }
+ local_irq_disable();
+ }
+
+ clts(); /* Allow maths ops (or we recurse) */
+#ifdef CONFIG_X86_32
+ restore_fpu(tsk);
+#else
+ /*
+ * Paranoid restore. send a SIGSEGV if we fail to restore the state.
+ */
+ if (unlikely(restore_fpu_checking(tsk))) {
+ stts();
+ force_sig(SIGSEGV, tsk);
+ return;
+ }
+#endif
+ thread->status |= TS_USEDGPU; /* So we fnsave on switch_to() */
+ tsk->fpu_counter++;
+ }
+EXPORT_SYMBOL_GPL(math_state_restore);
+
+#ifndef CONFIG_MATH_EMULATION
+asmlinkage void math_emulate(long arg)
+{
+ printk(KERN_EMERG
+ "math-emulation not enabled and no coprocessor found.\n");
+ printk(KERN_EMERG "killing %s.\n", current->comm);
+ force_sig(SIGFPE, current);
+ schedule();
+ }
+#endif /* CONFIG_MATH_EMULATION */
+
+dotraplinkage void __kprobes
+do_device_not_available(struct pt_regs *regs, long error)
+{
+#ifdef CONFIG_X86_32
+ if (read_cr0() & X86_CR0_EM) {
+ conditional_sti(regs);
+ math_emulate(0);
+ } else {
+ math_state_restore(); /* interrupts still off */
+ conditional_sti(regs);
+ }
+#else
+ math_state_restore();
+#endif
+ }
+
+#ifdef CONFIG_X86_32

```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
+#ifdef CONFIG_X86_MCE
+dotraplinkage void __kprobes do_machine_check(struct pt_regs *regs, long error)
+{
+ conditional_sti(regs);
+ machine_check_vector(regs, error);
+}
+#endif
+
+dotraplinkage void do_iret_error(struct pt_regs *regs, long error_code)
+{
+ sinfo_t info;
+ local_irq_enable();
+
+ info.si_signo = SIGILL;
+ info.si_errno = 0;
+ info.si_code = ILL_BADSTK;
+ info.si_addr = 0;
+ if (notify_die(DIE_TRAP, "iret exception",
+ regs, error_code, 32, SIGILL) == NOTIFY_STOP)
+ return;
+ do_trap(32, SIGILL, "iret exception", regs, error_code, &info);
+}
+#endif
+
+void __init trap_init(void)
+{
+#ifdef CONFIG_X86_32
+ int i;
+#endif
+
+#ifdef CONFIG_EISA
+ void __iomem *p = early_ioremap(0x0FFFD9, 4);
+
+ if (readl(p) == 'E' + ('T'<<8) + ('S'<<16) + ('A'<<24))
+ EISA_bus = 1;
+ early_iounmap(p, 4);
+#endif
+
+ set_intr_gate(0, &divide_error);
+ set_intr_gate_ist(1, &debug, DEBUG_STACK);
+ set_intr_gate_ist(2, &nmi, NMI_STACK);
+ /* int3 can be called from all */
+ set_system_intr_gate_ist(3, &int3, DEBUG_STACK);
+ /* int4 can be called from all */
+ set_system_intr_gate(4, &overflow);
+ set_intr_gate(5, &bounds);
+ set_intr_gate(6, &invalid_op);
+ set_intr_gate(7, &device_not_available);
+#ifdef CONFIG_X86_32
+ set_task_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);
+#else
```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
+ set_intr_gate_ist(8, &double_fault, DOUBLEFAULT_STACK);
+ #endif
+ set_intr_gate(9, &coprocessor_segment_overrun);
+ set_intr_gate(10, &invalid_TSS);
+ set_intr_gate(11, &segment_not_present);
+ set_intr_gate_ist(12, &stack_segment, STACKFAULT_STACK);
+ set_intr_gate(13, &general_protection);
+ set_intr_gate(14, &page_fault);
+ set_intr_gate(15, &spurious_interrupt_bug);
+ set_intr_gate(16, &coprocessor_error);
+ set_intr_gate(17, &alignment_check);
+ #ifdef CONFIG_X86_MCE
+ set_intr_gate_ist(18, &machine_check, MCE_STACK);
+ #endif
+ set_intr_gate(19, &simd_coprocessor_error);
+
+ #ifdef CONFIG_IA32_EMULATION
+ set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
+ #endif
+
+ #ifdef CONFIG_X86_32
+ if (cpu_has_fxsr) {
+ printk(KERN_INFO "Enabling fast FPU save and restore... ");
+ set_in_cr4(X86_CR4_OSFXSR);
+ printk("done.\n");
+ }
+ if (cpu_has_xmm) {
+ printk(KERN_INFO
+ "Enabling unmasked SIMD FPU exception support... ");
+ set_in_cr4(X86_CR4_OSXMMEXCPT);
+ printk("done.\n");
+ }
+
+ set_system_trap_gate(SYSCALL_VECTOR, &system_call);
+
+ /* Reserve all the builtin and the syscall vector: */
+ for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
+ set_bit(i, used_vectors);
+
+ set_bit(SYSCALL_VECTOR, used_vectors);
+ #endif
+ /*
+ * Should be a barrier for any external CPU state:
+ */
+ cpu_init();
+
+ #ifdef CONFIG_X86_32
+ trap_init_hook();
+ #endif
+ }
diff --git a/arch/x86/kernel/traps_32.c b/arch/x86/kernel/traps_32.c
```

```
deleted file mode 100644
index 54e08d2..0000000
--- a/arch/x86/kernel/traps_32.c
+++ /dev/null
@@ -1,1075 +0,0 @@
-/*
- * Copyright (C) 1991, 1992 Linus Torvalds
- * Copyright (C) 2000, 2001, 2002 Andi Kleen, SuSE Labs
- *
- * Pentium III FXSR, SSE support
- * Gareth Hughes <gareth@xxxxxxxxxxxx>, May 2000
- */
-
-/*
- * Handle hardware traps and faults.
- */
-#include <linux/interrupt.h>
-#include <linux/kallsyms.h>
-#include <linux/spinlock.h>
-#include <linux/kprobes.h>
-#include <linux/uaccess.h>
-#include <linux/utsname.h>
-#include <linux/kdebug.h>
-#include <linux/kernel.h>
-#include <linux/module.h>
-#include <linux/ptrace.h>
-#include <linux/string.h>
-#include <linux/unwind.h>
-#include <linux/delay.h>
-#include <linux/errno.h>
-#include <linux/kexec.h>
-#include <linux/sched.h>
-#include <linux/timer.h>
-#include <linux/init.h>
-#include <linux/bug.h>
-#include <linux/nmi.h>
-#include <linux/mm.h>
-#include <linux/smp.h>
-#include <linux/io.h>
-
-#ifdef CONFIG_EISA
-#include <linux/ioport.h>
-#include <linux/eisa.h>
-#endif
-
-#ifdef CONFIG_MCA
-#include <linux/mca.h>
-#endif
-
-#if defined(CONFIG_EDAC)
-#include <linux/edac.h>
```

```

-#endif
-
-#include <asm/stacktrace.h>
-#include <asm/processor.h>
-#include <asm/kmemcheck.h>
-#include <asm/debugreg.h>
-#include <asm/atomic.h>
-#include <asm/system.h>
-#include <asm/unwind.h>
-#include <asm/traps.h>
-#include <asm/desc.h>
-#include <asm/i387.h>
-
-#include <mach_traps.h>
-
-#ifdef CONFIG_X86_64
-#include <asm/pgalloc.h>
-#include <asm/proto.h>
-#include <asm/pda.h>
-#else
-#include <asm/processor-flags.h>
-#include <asm/arch_hooks.h>
-#include <asm/nmi.h>
-#include <asm/smp.h>
-#include <asm/io.h>
-
-#include "cpu/mcheck/mce.h"
-
-DECLARE_BITMAP(used_vectors, NR_VECTORS);
-EXPORT_SYMBOL_GPL(used_vectors);
-
-asmlinkage int system_call(void);
-
-/* Do we ignore FPU interrupts ? */
-char ignore_fpu_irq;
-
-/*
- * The IDT has to be page-aligned to simplify the Pentium
- * F0 0F bug workaround.. We have a special link segment
- * for this.
- */
-gate_desc idt_table[256]
-__attribute__((__section__(".data.idt"))) = { { { { 0, 0 } } }, };
-#endif
-
-static int ignore_nmis;
-
-static inline void conditional_sti(struct pt_regs *regs)
-{
- if (regs->flags & X86_EFLAGS_IF)
- local_irq_enable();

```

```

-}
-
-static inline void preempt_conditional_sti(struct pt_regs *regs)
-{
- inc_preempt_count();
- if (regs->flags & X86_EFLAGS_IF)
- local_irq_enable();
-}
-
-static inline void preempt_conditional_cli(struct pt_regs *regs)
-{
- if (regs->flags & X86_EFLAGS_IF)
- local_irq_disable();
- dec_preempt_count();
-}
-
-#ifdef CONFIG_X86_32
-static inline void
-die_if_kernel(const char *str, struct pt_regs *regs, long err)
-{
- if (!user_mode_vm(regs))
- die(str, regs, err);
-}
-
-/*
- * Perform the lazy TSS's I/O bitmap copy. If the TSS has an
- * invalid offset set (the LAZY one) and the faulting thread has
- * a valid I/O bitmap pointer, we copy the I/O bitmap in the TSS,
- * we set the offset field correctly and return 1.
- */
-static int lazy_iobitmap_copy(void)
-{
- struct thread_struct *thread;
- struct tss_struct *tss;
- int cpu;
-
- cpu = get_cpu();
- tss = &per_cpu(init_tss, cpu);
- thread = &current->thread;
-
- if (tss->x86_tss.io_bitmap_base == INVALID_IO_BITMAP_OFFSET_LAZY &&
- thread->io_bitmap_ptr) {
- memcpy(tss->io_bitmap, thread->io_bitmap_ptr,
- thread->io_bitmap_max);
- /*
- * If the previously set map was extending to higher ports
- * than the current one, pad extra space with 0xff (no access).
- */
- if (thread->io_bitmap_max < tss->io_bitmap_max) {
- memset((char *) tss->io_bitmap +
- thread->io_bitmap_max, 0xff,

```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
- tss->io_bitmap_max = thread->io_bitmap_max);
- }
- tss->io_bitmap_max = thread->io_bitmap_max;
- tss->x86_tss.io_bitmap_base = IO_BITMAP_OFFSET;
- tss->io_bitmap_owner = thread;
- put_cpu();
-
- return 1;
- }
- put_cpu();
-
- return 0;
-}
-#endif
-
-static void __kprobes
-do_trap(int trapnr, int signr, char *str, struct pt_regs *regs,
- long error_code, siginfo_t *info)
-{
- struct task_struct *tsk = current;
-
-#ifdef CONFIG_X86_32
- if (regs->flags & X86_VM_MASK) {
- /*
- * traps 0, 1, 3, 4, and 5 should be forwarded to vm86.
- * On nmi (interrupt 2), do_trap should not be called.
- */
- if (trapnr < 6)
- goto vm86_trap;
- goto trap_signal;
- }
-#endif
-
- if (!user_mode(regs))
- goto kernel_trap;
-
-#ifdef CONFIG_X86_32
-trap_signal:
-#endif
- /*
- * We want error_code and trap_no set for userspace faults and
- * kernelspace faults which result in die(), but not
- * kernelspace faults which are fixed up. die() gives the
- * process no chance to handle the signal and notice the
- * kernel fault information, so that won't result in polluting
- * the information about previously queued, but not yet
- * delivered, faults. See also do_general_protection below.
- */
- tsk->thread.error_code = error_code;
- tsk->thread.trap_no = trapnr;
-
-}
```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
-#ifdef CONFIG_X86_64
- if (show_unhandled_signals && unhandled_signal(tsk, signr) &&
- printk_ratelimit()) {
- printk(KERN_INFO
- "%s[%d] trap %s ip:%lx sp:%lx error:%lx",
- tsk->comm, tsk->pid, str,
- regs->ip, regs->sp, error_code);
- print_vma_addr(" in ", regs->ip);
- printk("\n");
- }
-#endif
-
- if (info)
- force_sig_info(signr, info, tsk);
- else
- force_sig(signr, tsk);
- return;
-
-#kernel_trap:
- if (!fixup_exception(regs)) {
- tsk->thread.error_code = error_code;
- tsk->thread.trap_no = trapnr;
- die(str, regs, error_code);
- }
- return;
-
-#ifdef CONFIG_X86_32
-#vm86_trap:
- if (handle_vm86_trap((struct kernel_vm86_regs *) regs,
- error_code, trapnr))
- goto trap_signal;
- return;
-#endif
-}
-
-#define DO_ERROR(trapnr, signr, str, name) \
-#dotraplinkage void do_###name(struct pt_regs *regs, long error_code) \
-{ \
- if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) \
- == NOTIFY_STOP) \
- return; \
- conditional_sti(regs); \
- do_trap(trapnr, signr, str, regs, error_code, NULL); \
-}
-
-#define DO_ERROR_INFO(trapnr, signr, str, name, sicode, siaddr) \
-#dotraplinkage void do_###name(struct pt_regs *regs, long error_code) \
-{ \
- siginfo_t info; \
- info.si_signo = signr; \
- info.si_errno = 0; \
```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
- info.si_code = sicode; \
- info.si_addr = (void __user *)siaddr; \
- if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) \
- == NOTIFY_STOP) \
- return; \
- conditional_sti(regs); \
- do_trap(trapnr, signr, str, regs, error_code, &info); \
-}
-
-DO_ERROR_INFO(0, SIGFPE, "divide error", divide_error, FPE_INTDIV, regs->ip)
-DO_ERROR(4, SIGSEGV, "overflow", overflow)
-DO_ERROR(5, SIGSEGV, "bounds", bounds)
-DO_ERROR_INFO(6, SIGILL, "invalid opcode", invalid_op, ILL_ILLOPN, regs->ip)
-DO_ERROR(9, SIGFPE, "coprocessor segment overrun", coprocessor_segment_overrun)
-DO_ERROR(10, SIGSEGV, "invalid TSS", invalid_TSS)
-DO_ERROR(11, SIGBUS, "segment not present", segment_not_present)
-#ifdef CONFIG_X86_32
-DO_ERROR(12, SIGBUS, "stack segment", stack_segment)
-#endif
-DO_ERROR_INFO(17, SIGBUS, "alignment check", alignment_check, BUS_ADRALN, 0)
-
-#ifdef CONFIG_X86_64
-/* Runs on IST stack */
-dotraplinkage void do_stack_segment(struct pt_regs *regs, long error_code)
-{
- if (notify_die(DIE_TRAP, "stack segment", regs, error_code,
- 12, SIGBUS) == NOTIFY_STOP)
- return;
- preempt_conditional_sti(regs);
- do_trap(12, SIGBUS, "stack segment", regs, error_code, NULL);
- preempt_conditional_cli(regs);
-}
-
-dotraplinkage void do_double_fault(struct pt_regs *regs, long error_code)
-{
- static const char str[] = "double fault";
- struct task_struct *tsk = current;
-
- /* Return not checked because double check cannot be ignored */
- notify_die(DIE_TRAP, str, regs, error_code, 8, SIGSEGV);
-
- tsk->thread.error_code = error_code;
- tsk->thread.trap_no = 8;
-
- /* This is always a kernel trap and never fixable (and thus must
- never return). */
- for (;;)
- die(str, regs, error_code);
-}
-#endif
-
```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
-dotraplinkage void __kprobes
-do_general_protection(struct pt_regs *regs, long error_code)
-{
- struct task_struct *tsk;
-
- conditional_sti(regs);
-
-#ifdef CONFIG_X86_32
- if (lazy_iobitmap_copy()) {
- /* restart the faulting instruction */
- return;
- }
-
- if (regs->flags & X86_VM_MASK)
- goto gp_in_vm86;
-#endif
-
- tsk = current;
- if (!user_mode(regs))
- goto gp_in_kernel;
-
- tsk->thread.error_code = error_code;
- tsk->thread.trap_no = 13;
-
- if (show_unhandled_signals && unhandled_signal(tsk, SIGSEGV) &&
- printk_ratelimit()) {
- printk(KERN_INFO
- "%s[%d] general protection ip:%lx sp:%lx error:%lx",
- tsk->comm, task_pid_nr(tsk),
- regs->ip, regs->sp, error_code);
- print_vma_addr(" in ", regs->ip);
- printk("\n");
- }
-
- force_sig(SIGSEGV, tsk);
- return;
-
-#ifdef CONFIG_X86_32
-gp_in_vm86:
- local_irq_enable();
- handle_vm86_fault((struct kernel_vm86_regs *) regs, error_code);
- return;
-#endif
-
-gp_in_kernel:
- if (fixup_exception(regs))
- return;
-
- tsk->thread.error_code = error_code;
- tsk->thread.trap_no = 13;
- if (notify_die(DIE_GPF, "general protection fault", regs,
```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
- error_code, 13, SIGSEGV) == NOTIFY_STOP)
- return;
- die("general protection fault", regs, error_code);
-}
-
-static notrace __kprobes void
-mem_parity_error(unsigned char reason, struct pt_regs *regs)
-{
- printk(KERN_EMERG
- "Uhhuh. NMI received for unknown reason %02x on CPU %d.\n",
- reason, smp_processor_id());
-
- printk(KERN_EMERG
- "You have some hardware problem, likely on the PCI bus.\n");
-
-#if defined(CONFIG_EDAC)
- if (edac_handler_set()) {
- edac_atomic_assert_error();
- return;
- }
-#endif
-
- if (panic_on_unrecovered_nmi)
- panic("NMI: Not continuing");
-
- printk(KERN_EMERG "Dazed and confused, but trying to continue\n");
-
- /* Clear and disable the memory parity error line. */
- reason = (reason & 0xf) | 4;
- outb(reason, 0x61);
-}
-
-static notrace __kprobes void
-io_check_error(unsigned char reason, struct pt_regs *regs)
-{
- unsigned long i;
-
- printk(KERN_EMERG "NMI: IOCK error (debug interrupt?)\n");
- show_registers(regs);
-
- /* Re-enable the IOCK line, wait for a few seconds */
- reason = (reason & 0xf) | 8;
- outb(reason, 0x61);
-
- i = 2000;
- while (--i)
- udelay(1000);
-
- reason &= ~8;
- outb(reason, 0x61);
-}
```

```

-
- static notrace __kprobes void
- unknown_nmi_error(unsigned char reason, struct pt_regs *regs)
- {
- if (notify_die(DIE_NMIUNKNOWN, "nmi", regs, reason, 2, SIGINT) ==
- NOTIFY_STOP)
- return;
- #ifdef CONFIG_MCA
- /*
- * Might actually be able to figure out what the guilty party
- * is:
- */
- if (MCA_bus) {
- mca_handle_nmi();
- return;
- }
- #endif
- printk(KERN_EMERG
- "Uhhuh. NMI received for unknown reason %02x on CPU %d.\n",
- reason, smp_processor_id());
-
- printk(KERN_EMERG "Do you have a strange power saving mode enabled?\n");
- if (panic_on_unrecovered_nmi)
- panic("NMI: Not continuing");
-
- printk(KERN_EMERG "Dazed and confused, but trying to continue\n");
- }
-
- #ifdef CONFIG_X86_32
- static DEFINE_SPINLOCK(nmi_print_lock);
-
- void notrace __kprobes die_nmi(char *str, struct pt_regs *regs, int do_panic)
- {
- if (notify_die(DIE_NMIWATCHDOG, str, regs, 0, 2, SIGINT) == NOTIFY_STOP)
- return;
-
- spin_lock(&nmi_print_lock);
- /*
- * We are in trouble anyway, lets at least try
- * to get a message out:
- */
- bust_spinlocks(1);
- printk(KERN_EMERG "%s", str);
- printk(" on CPU%d, ip %08lx, registers:\n",
- smp_processor_id(), regs->ip);
- show_registers(regs);
- if (do_panic)
- panic("Non maskable interrupt");
- console_silent();
- spin_unlock(&nmi_print_lock);
- bust_spinlocks(0);

```

```

-
- /*
- * If we are in kernel we are probably nested up pretty bad
- * and might aswell get out now while we still can:
- */
- if (!user_mode_vm(regs)) {
-     current->thread.trap_no = 2;
-     crash_kexec(regs);
- }
-
- do_exit(SIGSEGV);
-}
-#endif
-
-static notrace __kprobes void default_do_nmi(struct pt_regs *regs)
-{
-     unsigned char reason = 0;
-     int cpu;
-
-     cpu = smp_processor_id();
-
-     /* Only the BSP gets external NMIs from the system. */
-     if (!cpu)
-         reason = get_nmi_reason();
-
-     if (!(reason & 0xc0)) {
-         if (notify_die(DIE_NMI_IPI, "nmi_ipi", regs, reason, 2, SIGINT)
-             == NOTIFY_STOP)
-             return;
-#ifdef CONFIG_X86_LOCAL_APIC
-         /*
-          * Ok, so this is none of the documented NMI sources,
-          * so it must be the NMI watchdog.
-          */
-         if (nmi_watchdog_tick(regs, reason))
-             return;
-         if (!do_nmi_callback(regs, cpu))
-             unknown_nmi_error(reason, regs);
-#else
-         unknown_nmi_error(reason, regs);
-#endif
-     }
-
-     return;
- }
- if (notify_die(DIE_NMI, "nmi", regs, reason, 2, SIGINT) == NOTIFY_STOP)
-     return;
-
-     /* AK: following checks seem to be broken on modern chipsets. FIXME */
-     if (reason & 0x80)
-         mem_parity_error(reason, regs);
-     if (reason & 0x40)

```

```

- io_check_error(reason, regs);
-#ifdef CONFIG_X86_32
- /*
- * Reassert NMI in case it became active meanwhile
- * as it's edge-triggered:
- */
- reassert_nmi();
-#endif
-}
-
-dotraplinkage notrace __kprobes void
-do_nmi(struct pt_regs *regs, long error_code)
-{
- nmi_enter();
-
-#ifdef CONFIG_X86_32
- { int cpu; cpu = smp_processor_id(); ++nmi_count(cpu); }
-#else
- add_pda(__nmi_count, 1);
-#endif
-
- if (!ignore_nmis)
- default_do_nmi(regs);
-
- nmi_exit();
-}
-
-void stop_nmi(void)
-{
- acpi_nmi_disable();
- ignore_nmis++;
-}
-
-void restart_nmi(void)
-{
- ignore_nmis--;
- acpi_nmi_enable();
-}
-
-/* May run on IST stack. */
-dotraplinkage void __kprobes do_int3(struct pt_regs *regs, long error_code)
-{
-#ifdef CONFIG_KPROBES
- if (notify_die(DIE_INT3, "int3", regs, error_code, 3, SIGTRAP)
- == NOTIFY_STOP)
- return;
-#else
- if (notify_die(DIE_TRAP, "int3", regs, error_code, 3, SIGTRAP)
- == NOTIFY_STOP)
- return;
-#endif
-}

```

```

-
- preempt_conditional_sti(regs);
- do_trap(3, SIGTRAP, "int3", regs, error_code, NULL);
- preempt_conditional_cli(regs);
- }
-
-#ifdef CONFIG_X86_64
-/* Help handler running on IST stack to switch back to user stack
- for scheduling or signal handling. The actual stack switch is done in
- entry.S */
-asmlinkage __kprobes struct pt_regs *sync_regs(struct pt_regs *eregs)
-{
- struct pt_regs *regs = eregs;
- /* Did already sync */
- if (eregs == (struct pt_regs *)eregs->sp)
- ;
- /* Exception from user space */
- else if (user_mode(eregs))
- regs = task_pt_regs(current);
- /* Exception from kernel and interrupts are enabled. Move to
- kernel process stack. */
- else if (eregs->flags & X86_EFLAGS_IF)
- regs = (struct pt_regs *) (eregs->sp - sizeof(struct pt_regs));
- if (eregs != regs)
- *regs = *eregs;
- return regs;
-}
-#endif
-
-/*
- * Our handling of the processor debug registers is non-trivial.
- * We do not clear them on entry and exit from the kernel. Therefore
- * it is possible to get a watchpoint trap here from inside the kernel.
- * However, the code in ./ptrace.c has ensured that the user can
- * only set watchpoints on userspace addresses. Therefore the in-kernel
- * watchpoint trap can only occur in code which is reading/writing
- * from user space. Such code must not hold kernel locks (since it
- * can equally take a page fault), therefore it is safe to call
- * force_sig_info even though that claims and releases locks.
- *
- * Code in ./signal.c ensures that the debug control register
- * is restored before we deliver any signal, and therefore that
- * user code runs with the correct debug control register even though
- * we clear it here.
- *
- * Being careful here means that we don't have to be as careful in a
- * lot of more complicated places (task switching can be a bit lazy
- * about restoring all the debug state, and ptrace doesn't have to
- * find every occurrence of the TF bit that could be saved away even
- * by user code)
- *

```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
- * May run on IST stack.
- */
-dotraplinkage void __kprobes do_debug(struct pt_regs *regs, long error_code)
-{
- struct task_struct *tsk = current;
- unsigned long condition;
- int si_code;
-
- get_debugreg(condition, 6);
-
- /* Catch kmemcheck conditions first of all! */
- if (condition & DR_STEP && kmemcheck_trap(regs))
- return;
-
- /*
- * The processor cleared BTF, so don't mark that we need it set.
- */
- clear_tsk_thread_flag(tsk, TIF_DEBUGCTLMSR);
- tsk->thread.debugctlmsr = 0;
-
- if (notify_die(DIE_DEBUG, "debug", regs, condition, error_code,
- SIGTRAP) == NOTIFY_STOP)
- return;
-
- /* It's safe to allow irq's after DR6 has been saved */
- preempt_conditional_sti(regs);
-
- /* Mask out spurious debug traps due to lazy DR7 setting */
- if (condition & (DR_TRAP0|DR_TRAP1|DR_TRAP2|DR_TRAP3)) {
- if (!tsk->thread.debugreg7)
- goto clear_dr7;
- }
-
-#ifdef CONFIG_X86_32
- if (regs->flags & X86_VM_MASK)
- goto debug_vm86;
-#endif
-
- /* Save debug status register where ptrace can see it */
- tsk->thread.debugreg6 = condition;
-
- /*
- * Single-stepping through TF: make sure we ignore any events in
- * kernel space (but re-enable TF when returning to user mode).
- */
- if (condition & DR_STEP) {
- if (!user_mode(regs))
- goto clear_TF_reenable;
- }
-
- si_code = get_si_code(condition);
```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
- /* Ok, finally something we can handle */
- send_sigtrap(tsk, regs, error_code, si_code);
-
- /*
- * Disable additional traps. They'll be re-enabled when
- * the signal is delivered.
- */
-clear_dr7:
- set_debugreg(0, 7);
- preempt_conditional_cli(regs);
- return;
-
-#ifdef CONFIG_X86_32
-debug_vm86:
- handle_vm86_trap((struct kernel_vm86_regs *) regs, error_code, 1);
- preempt_conditional_cli(regs);
- return;
-#endif
-
-clear_TF_reenable:
- set_tsk_thread_flag(tsk, TIF_SINGLESTEP);
- regs->flags &= ~X86_EFLAGS_TF;
- preempt_conditional_cli(regs);
- return;
-}
-
-#ifdef CONFIG_X86_64
-static int kernel_math_error(struct pt_regs *regs, const char *str, int trapnr)
-{
- if (fixup_exception(regs))
- return 1;
-
- notify_die(DIE_GPF, str, regs, 0, trapnr, SIGFPE);
- /* Illegal floating point operation in the kernel */
- current->thread.trap_no = trapnr;
- die(str, regs, 0);
- return 0;
-}
-#endif
-
-/*
- * Note that we play around with the 'TS' bit in an attempt to get
- * the correct behaviour even in the presence of the asynchronous
- * IRQ13 behaviour
- */
-void math_error(void __user *ip)
-{
- struct task_struct *task;
- siginfo_t info;
- unsigned short cwd, swd;
-}
```

```

- /*
- * Save the info for the exception handler and clear the error.
- */
- task = current;
- save_init_fpu(task);
- task->thread.trap_no = 16;
- task->thread.error_code = 0;
- info.si_signo = SIGFPE;
- info.si_errno = 0;
- info.si_code = __SI_FAULT;
- info.si_addr = ip;
- /*
- * (~cwd & swd) will mask out exceptions that are not set to unmasked
- * status. 0x3f is the exception bits in these regs, 0x200 is the
- * C1 reg you need in case of a stack fault, 0x040 is the stack
- * fault bit. We should only be taking one exception at a time,
- * so if this combination doesn't produce any single exception,
- * then we have a bad program that isn't synchronizing its FPU usage
- * and it will suffer the consequences since we won't be able to
- * fully reproduce the context of the exception
- */
- cwd = get_fpu_cwd(task);
- swd = get_fpu_swd(task);
- switch (swd & ~cwd & 0x3f) {
- case 0x000: /* No unmasked exception */
- #ifdef CONFIG_X86_32
- return;
- #endif
- default: /* Multiple exceptions */
- break;
- case 0x001: /* Invalid Op */
- /*
- * swd & 0x240 == 0x040: Stack Underflow
- * swd & 0x240 == 0x240: Stack Overflow
- * User must clear the SF bit (0x40) if set
- */
- info.si_code = FPE_FLTINV;
- break;
- case 0x002: /* Denormalize */
- case 0x010: /* Underflow */
- info.si_code = FPE_FLTUND;
- break;
- case 0x004: /* Zero Divide */
- info.si_code = FPE_FLTDIV;
- break;
- case 0x008: /* Overflow */
- info.si_code = FPE_FLTOVF;
- break;
- case 0x020: /* Precision */
- info.si_code = FPE_FLTRES;
- break;

```

```

- }
- force_sig_info(SIGFPE, &info, task);
-}
-
-dotraplinkage void do_coprocessor_error(struct pt_regs *regs, long error_code)
-{
- conditional_sti(regs);
-
-#ifdef CONFIG_X86_32
- ignore_fpu_irq = 1;
-#else
- if (!user_mode(regs) &&
- kernel_math_error(regs, "kernel x87 math error", 16))
- return;
-#endif
-
- math_error((void __user *)regs->ip);
-}
-
-static void simd_math_error(void __user *ip)
-{
- struct task_struct *task;
- siginfo_t info;
- unsigned short mxcsr;
-
- /*
- * Save the info for the exception handler and clear the error.
- */
- task = current;
- save_init_fpu(task);
- task->thread.trap_no = 19;
- task->thread.error_code = 0;
- info.si_signo = SIGFPE;
- info.si_errno = 0;
- info.si_code = __SI_FAULT;
- info.si_addr = ip;
- /*
- * The SIMD FPU exceptions are handled a little differently, as there
- * is only a single status/control register. Thus, to determine which
- * unmasked exception was caught we must mask the exception mask bits
- * at 0x1f80, and then use these to mask the exception bits at 0x3f.
- */
- mxcsr = get_fpu_mxcsr(task);
- switch (~((mxcsr & 0x1f80) >> 7) & (mxcsr & 0x3f)) {
- case 0x000:
- default:
- break;
- case 0x001: /* Invalid Op */
- info.si_code = FPE_FLTINV;
- break;
- case 0x002: /* Denormalize */

```

```

- case 0x010: /* Underflow */
- info.si_code = FPE_FLTUND;
- break;
- case 0x004: /* Zero Divide */
- info.si_code = FPE_FLTDIV;
- break;
- case 0x008: /* Overflow */
- info.si_code = FPE_FLTOVF;
- break;
- case 0x020: /* Precision */
- info.si_code = FPE_FLTRES;
- break;
- }
- force_sig_info(SIGFPE, &info, task);
-}
-
-dotraplinkage void
-do_simd_coprocessor_error(struct pt_regs *regs, long error_code)
-{
- conditional_sti(regs);
-
-#ifdef CONFIG_X86_32
- if (cpu_has_xmm) {
- /* Handle SIMD FPU exceptions on PIII+ processors. */
- ignore_fpu_irq = 1;
- simd_math_error((void __user *)regs->ip);
- return;
- }
- /*
- * Handle strange cache flush from user space exception
- * in all other cases. This is undocumented behaviour.
- */
- if (regs->flags & X86_VM_MASK) {
- handle_vm86_fault((struct kernel_vm86_regs *)regs, error_code);
- return;
- }
- current->thread.trap_no = 19;
- current->thread.error_code = error_code;
- die_if_kernel("cache flush denied", regs, error_code);
- force_sig(SIGSEGV, current);
-#else
- if (!user_mode(regs) &&
- kernel_math_error(regs, "kernel simd math error", 19))
- return;
- simd_math_error((void __user *)regs->ip);
-#endif
-}
-
-dotraplinkage void
-do_spurious_interrupt_bug(struct pt_regs *regs, long error_code)
-{

```

```

- conditional_sti(regs);
-#if 0
- /* No need to warn about this any longer. */
- printk(KERN_INFO "Ignoring P6 Local APIC Spurious Interrupt Bug...\n");
-#endif
-}
-
-#ifdef CONFIG_X86_32
-unsigned long patch_espfix_desc(unsigned long uesp, unsigned long kesp)
-{
- struct desc_struct *gdt = get_cpu_gdt_table(smp_processor_id());
- unsigned long base = (kesp - uesp) & -THREAD_SIZE;
- unsigned long new_ksesp = kesp - base;
- unsigned long lim_pages = (new_ksesp | (THREAD_SIZE - 1)) >> PAGE_SHIFT;
- __u64 desc = *(__u64 *)&gdt[GDT_ENTRY_ESPFIX_SS];
-
- /* Set up base for espfix segment */
- desc &= 0x00f0ff0000000000ULL;
- desc |= (((__u64)base) << 16) & 0x000000ffffff0000ULL |
- (((__u64)base) << 32) & 0xff00000000000000ULL |
- (((__u64)lim_pages) << 32) & 0x000f000000000000ULL |
- (lim_pages & 0xffff);
- *(__u64 *)&gdt[GDT_ENTRY_ESPFIX_SS] = desc;
-
- return new_ksesp;
-}
-#else
-asm__linkage void __attribute__((weak)) smp_thermal_interrupt(void)
-{
-}
-
-asm__linkage void __attribute__((weak)) mce_threshold_interrupt(void)
-{
-}
-#endif
-
-/*
- * 'math_state_restore()' saves the current math information in the
- * old math state array, and gets the new ones from the current task
- *
- * Careful.. There are problems with IBM-designed IRQ13 behaviour.
- * Don't touch unless you *really* know how it works.
- *
- * Must be called with kernel preemption disabled (in this case,
- * local interrupts are disabled at the call-site in entry.S).
- */
-asm__linkage void math_state_restore(void)
-{
- struct thread_info *thread = current_thread_info();
- struct task_struct *tsk = thread->task;
-}

```

```

- if (!tsk_used_math(tsk)) {
- local_irq_enable();
- /*
- * does a slab alloc which can sleep
- */
- if (init_fpu(tsk)) {
- /*
- * ran out of memory!
- */
- do_group_exit(SIGKILL);
- return;
- }
- local_irq_disable();
- }
-
- clts(); /* Allow maths ops (or we recurse) */
-#ifdef CONFIG_X86_32
- restore_fpu(tsk);
-#else
- /*
- * Paranoid restore. send a SIGSEGV if we fail to restore the state.
- */
- if (unlikely(restore_fpu_checking(tsk))) {
- stts();
- force_sig(SIGSEGV, tsk);
- return;
- }
-#endif
- thread->status |= TS_USED_FPU; /* So we fnsave on switch_to() */
- tsk->fpu_counter++;
- }
- EXPORT_SYMBOL_GPL(math_state_restore);
-
-#ifndef CONFIG_MATH_EMULATION
- asmlinkage void math_emulate(long arg)
- {
- printk(KERN_EMERG
- "math-emulation not enabled and no coprocessor found.\n");
- printk(KERN_EMERG "killing %s.\n", current->comm);
- force_sig(SIGFPE, current);
- schedule();
- }
-#endif /* CONFIG_MATH_EMULATION */
-
- dotraplinkage void __kprobes
- do_device_not_available(struct pt_regs *regs, long error)
- {
-#ifdef CONFIG_X86_32
- if (read_cr0() & X86_CR0_EM) {
- conditional_sti(regs);
- math_emulate(0);

```

```

- } else {
- math_state_restore(); /* interrupts still off */
- conditional_sti(regs);
- }
-#else
- math_state_restore();
-#endif
-}
-
-#ifndef CONFIG_X86_32
-#ifndef CONFIG_X86_MCE
-dotraplinkage void __kprobes do_machine_check(struct pt_regs *regs, long error)
-{
- conditional_sti(regs);
- machine_check_vector(regs, error);
-}
-#endif
-
-dotraplinkage void do_iret_error(struct pt_regs *regs, long error_code)
-{
- siginfo_t info;
- local_irq_enable();
-
- info.si_signo = SIGILL;
- info.si_errno = 0;
- info.si_code = ILL_BADSTK;
- info.si_addr = 0;
- if (notify_die(DIE_TRAP, "iret exception",
- regs, error_code, 32, SIGILL) == NOTIFY_STOP)
- return;
- do_trap(32, SIGILL, "iret exception", regs, error_code, &info);
-}
-#endif
-
-void __init trap_init(void)
-{
-#ifndef CONFIG_X86_32
- int i;
-#endif
-
-#ifndef CONFIG_EISA
- void __iomem *p = early_ioremap(0x0FFFD9, 4);
-
- if (readl(p) == 'E' + ('T'<<8) + ('S'<<16) + ('A'<<24))
- EISA_bus = 1;
- early_iounmap(p, 4);
-#endif
-
- set_intr_gate(0, &divide_error);
- set_intr_gate_ist(1, &debug, DEBUG_STACK);
- set_intr_gate_ist(2, &nmi, NMI_STACK);

```

[PATCH 9/9] traps: x86: finalize unification of traps.c

```
- /* int3 can be called from all */  
- set_system_intr_gate_ist(3, &int3, DEBUG_STACK);  
- /* int4 can be called from all */  
- set_system_intr_gate(4, &overflow);  
- set_intr_gate(5, &bounds);  
- set_intr_gate(6, &invalid_op);  
- set_intr_gate(7, &device_not_available);  
-#ifdef CONFIG_X86_32  
- set_task_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);  
-#else  
- set_intr_gate_ist(8, &
```