```c
/*
 *  linux/kernel/panic.c
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 */

/*
 * This function is used through-out the kernel (including mm and fs)
 * to indicate a major problem.
 */
#include <linux/debug_locks.h>
#include <linux/interrupt.h>
#include <linux/kmsg_dump.h>
#include <linux/kallsyms.h>
#include <linux/notifier.h>
#include <linux/module.h>
#include <linux/random.h>
#include <linux/ftrace.h>
#include <linux/reboot.h>
#include <linux/delay.h>
#include <linux/kexec.h>
#include <linux/sched.h>
#include <linux/sysrq.h>
#include <linux/init.h>
#include <linux/nmi.h>

#define PANIC_TIMER_STEP 100
#define PANIC_BLINK_SPD 18

int panic_on_oops = CONFIG_PANIC_ON_OOPS_VALUE;
static unsigned long tainted_mask;
static int pause_on_oops;
static int pause_on_oops_flag;
static DEFINE_SPINLOCK(pause_on_oops_lock);
static bool crash_kexec_post_notifiers;
int panic_on_warn __read_mostly;

int panic_timeout = CONFIG_PANIC_TIMEOUT;
EXPORT_SYMBOL_GPL(panic_timeout);

ATOMIC_NOTIFIER_HEAD(panic_notifier_list);

EXPORT_SYMBOL(panic_notifier_list);

static long no_blink(int state)
{
        return 0;
}
```

```
/* Returns how long it waited in ms */
long (*panic_blink)(int state);
EXPORT_SYMBOL(panic_blink);

/*
 * Stop ourself in panic -- architecture code may override this
 */
void __weak panic_smp_self_stop(void)
{
        while (1)
                cpu_relax();
}

/**
 *      panic - halt the system
 *      @fmt: The text string to print
 *
 *      Display a message, then perform cleanups.
 *
 *      This function never returns.
 */
void panic(const char *fmt, ...)
{
        static DEFINE_SPINLOCK(panic_lock);
        static char buf[1024];
        va_list args;
        long i, i_next = 0;
        int state = 0;

        /*
         * Disable local interrupts. This will prevent panic_smp_self_stop
         * from deadlocking the first cpu that invokes the panic, since
         * there is nothing to prevent an interrupt handler (that runs
         * after the panic_lock is acquired) from invoking panic again.
         */
        local_irq_disable();

        /*
         * It's possible to come here directly from a panic-assertion and
         * not have preempt disabled. Some functions called from here want
         * preempt to be disabled. No point enabling it later though...
         *
         * Only one CPU is allowed to execute the panic code from here. For
         * multiple parallel invocations of panic, all other CPUs either
         * stop themself or will wait until they are stopped by the 1st CPU
         * with smp_send_stop().
         */
        if (!spin_trylock(&panic_lock))
                panic_smp_self_stop();
```

```
        console_verbose();
        bust_spinlocks(1);
        va_start(args, fmt);
        vsnprintf(buf, sizeof(buf), fmt, args);
        va_end(args);
        pr_emerg("Kernel panic - not syncing: %s\n", buf);
#ifdef CONFIG_DEBUG_BUGVERBOSE
        /*
         * Avoid nested stack-dumping if a panic occurs during oops processing
         */
        if (!test_taint(TAINT_DIE) && oops_in_progress <= 1)
                dump_stack();
#endif

        /*
         * If we have crashed and we have a crash kernel loaded let it handle
         * everything else.
         * If we want to run this after calling panic_notifiers, pass
         * the "crash_kexec_post_notifiers" option to the kernel.
         */
        if (!crash_kexec_post_notifiers)
                crash_kexec(NULL);

        /*
         * Note smp_send_stop is the usual smp shutdown function, which
         * unfortunately means it may not be hardened to work in a panic
         * situation.
         */
        smp_send_stop();

        /*
         * Run any panic handlers, including those that might need to
         * add information to the kmsg dump output.
         */
        atomic_notifier_call_chain(&panic_notifier_list, 0, buf);

        kmsg_dump(KMSG_DUMP_PANIC);

        /*
         * If you doubt kdump always works fine in any situation,
         * "crash_kexec_post_notifiers" offers you a chance to run
         * panic_notifiers and dumping kmsg before kdump.
         * Note: since some panic_notifiers can make crashed kernel
         * more unstable, it can increase risks of the kdump failure too.
         */
        crash_kexec(NULL);

        bust_spinlocks(0);

        if (!panic_blink)
```

```
                panic_blink = no_blink;

        if (panic_timeout > 0) {
                /*
                 * Delay timeout seconds before rebooting the machine.
                 * We can't use the "normal" timers since we just panicked.
                 */
                pr_emerg("Rebooting in %d seconds..", panic_timeout);

                for (i = 0; i < panic_timeout * 1000; i += PANIC_TIMER_STEP) {
                        touch_nmi_watchdog();
                        if (i >= i_next) {
                                i += panic_blink(state ^= 1);
                                i_next = i + 3600 / PANIC_BLINK_SPD;
                        }
                        mdelay(PANIC_TIMER_STEP);
                }
        }
        if (panic_timeout != 0) {
                /*
                 * This will not be a clean reboot, with everything
                 * shutting down.  But if there is a chance of
                 * rebooting the system it will be rebooted.
                 */
                emergency_restart();
        }
#ifdef __sparc__
        {
                extern int stop_a_enabled;
                /* Make sure the user can actually press Stop-A (L1-A) */
                stop_a_enabled = 1;
                pr_emerg("Press Stop-A (L1-A) to return to the boot prom\n");
        }
#endif
#if defined(CONFIG_S390)
        {
                unsigned long caller;

                caller = (unsigned long)__builtin_return_address(0);
                disabled_wait(caller);
        }
#endif
        pr_emerg("---[ end Kernel panic - not syncing: %s\n", buf);
        local_irq_enable();
        for (i = 0; ; i += PANIC_TIMER_STEP) {
                touch_softlockup_watchdog();
                if (i >= i_next) {
                        i += panic_blink(state ^= 1);
                        i_next = i + 3600 / PANIC_BLINK_SPD;
                }
```

```
                mdelay(PANIC_TIMER_STEP);
        }
}

EXPORT_SYMBOL(panic);


struct tnt {
        u8      bit;
        char    true;
        char    false;
};

static const struct tnt tnts[] = {
        { TAINT_PROPRIETARY_MODULE,     'P', 'G' },
        { TAINT_FORCED_MODULE,          'F', ' ' },
        { TAINT_CPU_OUT_OF_SPEC,        'S', ' ' },
        { TAINT_FORCED_RMMOD,           'R', ' ' },
        { TAINT_MACHINE_CHECK,          'M', ' ' },
        { TAINT_BAD_PAGE,               'B', ' ' },
        { TAINT_USER,                   'U', ' ' },
        { TAINT_DIE,                    'D', ' ' },
        { TAINT_OVERRIDDEN_ACPI_TABLE,  'A', ' ' },
        { TAINT_WARN,                   'W', ' ' },
        { TAINT_CRAP,                   'C', ' ' },
        { TAINT_FIRMWARE_WORKAROUND,    'I', ' ' },
        { TAINT_OOT_MODULE,             'O', ' ' },
        { TAINT_UNSIGNED_MODULE,        'E', ' ' },
        { TAINT_SOFTLOCKUP,             'L', ' ' },
        { TAINT_LIVEPATCH,              'K', ' ' },
};

/**
 *      print_tainted - return a string to represent the kernel taint state.
 *
 *  'P' - Proprietary module has been loaded.
 *  'F' - Module has been forcibly loaded.
 *  'S' - SMP with CPUs not designed for SMP.
 *  'R' - User forced a module unload.
 *  'M' - System experienced a machine check exception.
 *  'B' - System has hit bad_page.
 *  'U' - Userspace-defined naughtiness.
 *  'D' - Kernel has oopsed before
 *  'A' - ACPI table overridden.
 *  'W' - Taint on warning.
 *  'C' - modules from drivers/staging are loaded.
 *  'I' - Working around severe firmware bug.
 *  'O' - Out-of-tree module has been loaded.
 *  'E' - Unsigned module has been loaded.
 *  'L' - A soft lockup has previously occurred.
```

```
 *  'K' - Kernel has been live patched.
 *
 *      The string is overwritten by the next call to print_tainted().
 */
const char *print_tainted(void)
{
        static char buf[ARRAY_SIZE(tnts) + sizeof("Tainted: ")];

        if (tainted_mask) {
                char *s;
                int i;

                s = buf + sprintf(buf, "Tainted: ");
                for (i = 0; i < ARRAY_SIZE(tnts); i++) {
                        const struct tnt *t = &tnts[i];
                        *s++ = test_bit(t->bit, &tainted_mask) ?
                                        t->true : t->false;
                }
                *s = 0;
        } else
                snprintf(buf, sizeof(buf), "Not tainted");

        return buf;
}

int test_taint(unsigned flag)
{
        return test_bit(flag, &tainted_mask);
}
EXPORT_SYMBOL(test_taint);

unsigned long get_taint(void)
{
        return tainted_mask;
}

/**
 * add_taint: add a taint flag if not already set.
 * @flag: one of the TAINT_* constants.
 * @lockdep_ok: whether lock debugging is still OK.
 *
 * If something bad has gone wrong, you'll want @lockdebug_ok = false, but for
 * some noteworthtt-but-not-corrupting cases, it can be set to true.
 */
void add_taint(unsigned flag, enum lockdep_ok lockdep_ok)
{
        if (lockdep_ok == LOCKDEP_NOW_UNRELIABLE && __debug_locks_off())
                pr_warn("Disabling lock debugging due to kernel taint\n");

        set_bit(flag, &tainted_mask);
```

```
}
EXPORT_SYMBOL(add_taint);

static void spin_msec(int msecs)
{
        int i;

        for (i = 0; i < msecs; i++) {
                touch_nmi_watchdog();
                mdelay(1);
        }
}

/*
 * It just happens that oops_enter() and oops_exit() are identically
 * implemented...
 */
static void do_oops_enter_exit(void)
{
        unsigned long flags;
        static int spin_counter;

        if (!pause_on_oops)
                return;

        spin_lock_irqsave(&pause_on_oops_lock, flags);
        if (pause_on_oops_flag == 0) {
                /* This CPU may now print the oops message */
                pause_on_oops_flag = 1;
        } else {
                /* We need to stall this CPU */
                if (!spin_counter) {
                        /* This CPU gets to do the counting */
                        spin_counter = pause_on_oops;
                        do {
                                spin_unlock(&pause_on_oops_lock);
                                spin_msec(MSEC_PER_SEC);
                                spin_lock(&pause_on_oops_lock);
                        } while (--spin_counter);
                        pause_on_oops_flag = 0;
                } else {
                        /* This CPU waits for a different one */
                        while (spin_counter) {
                                spin_unlock(&pause_on_oops_lock);
                                spin_msec(1);
                                spin_lock(&pause_on_oops_lock);
                        }
                }
        }
        spin_unlock_irqrestore(&pause_on_oops_lock, flags);
```

```
}

/*
 * Return true if the calling CPU is allowed to print oops-related info.
 * This is a bit racy..
 */
int oops_may_print(void)
{
        return pause_on_oops_flag == 0;
}

/*
 * Called when the architecture enters its oops handler, before it prints
 * anything.  If this is the first CPU to oops, and it's oopsing the first
 * time then let it proceed.
 *
 * This is all enabled by the pause_on_oops kernel boot option.  We do all
 * this to ensure that oopses don't scroll off the screen.  It has the
 * side-effect of preventing later-oopsing CPUs from mucking up the display,
 * too.
 *
 * It turns out that the CPU which is allowed to print ends up pausing for
 * the right duration, whereas all the other CPUs pause for twice as long:
 * once in oops_enter(), once in oops_exit().
 */
void oops_enter(void)
{
        tracing_off();
        /* can't trust the integrity of the kernel anymore: */
        debug_locks_off();
        do_oops_enter_exit();
}

/*
 * 64-bit random ID for oopses:
 */
static u64 oops_id;

static int init_oops_id(void)
{
        if (!oops_id)
                get_random_bytes(&oops_id, sizeof(oops_id));
        else
                oops_id++;

        return 0;
}
late_initcall(init_oops_id);

void print_oops_end_marker(void)
```

```
{
        init_oops_id();
        pr_warn("---[ end trace %016llx ]---\n", (unsigned long long)oops_id);
}

/*
 * Called when the architecture exits its oops handler, after printing
 * everything.
 */
void oops_exit(void)
{
        do_oops_enter_exit();
        print_oops_end_marker();
        kmsg_dump(KMSG_DUMP_OOPS);
}

#ifdef WANT_WARN_ON_SLOWPATH
struct slowpath_args {
        const char *fmt;
        va_list args;
};

static void warn_slowpath_common(const char *file, int line, void *caller,
                                 unsigned taint, struct slowpath_args *args)
{
        disable_trace_on_warning();

        pr_warn("------------[ cut here ]------------\n");
        pr_warn("WARNING: CPU: %d PID: %d at %s:%d %pS()\n",
                raw_smp_processor_id(), current->pid, file, line, caller);

        if (args)
                vprintk(args->fmt, args->args);

        if (panic_on_warn) {
                /*
                 * This thread may hit another WARN() in the panic path.
                 * Resetting this prevents additional WARN() from panicking the
                 * system on this thread.  Other threads are blocked by the
                 * panic_mutex in panic().
                 */
                panic_on_warn = 0;
                panic("panic_on_warn set ...\n");
        }

        print_modules();
        dump_stack();
        print_oops_end_marker();
        /* Just a warning, don't kill lockdep. */
        add_taint(taint, LOCKDEP_STILL_OK);
```

```
}

void warn_slowpath_fmt(const char *file, int line, const char *fmt, ...)
{
        struct slowpath_args args;

        args.fmt = fmt;
        va_start(args.args, fmt);
        warn_slowpath_common(file, line, __builtin_return_address(0),
                             TAINT_WARN, &args);
        va_end(args.args);
}
EXPORT_SYMBOL(warn_slowpath_fmt);

void warn_slowpath_fmt_taint(const char *file, int line,
                             unsigned taint, const char *fmt, ...)
{
        struct slowpath_args args;

        args.fmt = fmt;
        va_start(args.args, fmt);
        warn_slowpath_common(file, line, __builtin_return_address(0),
                             taint, &args);
        va_end(args.args);
}
EXPORT_SYMBOL(warn_slowpath_fmt_taint);

void warn_slowpath_null(const char *file, int line)
{
        warn_slowpath_common(file, line, __builtin_return_address(0),
                             TAINT_WARN, NULL);
}
EXPORT_SYMBOL(warn_slowpath_null);
#endif

#ifdef CONFIG_CC_STACKPROTECTOR

/*
 * Called when gcc's -fstack-protector feature is used, and
 * gcc detects corruption of the on-stack canary value
 */
__visible void __stack_chk_fail(void)
{
        panic("stack-protector: Kernel stack is corrupted in: %p\n",
                __builtin_return_address(0));
}
EXPORT_SYMBOL(__stack_chk_fail);

#endif
```

```
core_param(panic, panic_timeout, int, 0644);
core_param(pause_on_oops, pause_on_oops, int, 0644);
core_param(panic_on_warn, panic_on_warn, int, 0644);

static int __init setup_crash_kexec_post_notifiers(char *s)
{
        crash_kexec_post_notifiers = true;
        return 0;
}
early_param("crash_kexec_post_notifiers", setup_crash_kexec_post_notifiers);

static int __init oops_setup(char *s)
{
        if (!s)
                return -EINVAL;
        if (!strcmp(s, "panic"))
                panic_on_oops = 1;
        return 0;
}
early_param("oops", oops_setup);
```