目录

内核崩溃重启动

内核通过 kdump 在崩溃时触发启动 kexec 装载的新内核，存储旧内存映像以便于调试，让系统在新内核上运行 ，从而避免了死机，增强了系统的稳定性。

用户使用工具 kexec 将新内核映像文件装载到用户内存缓冲区，然后，工具 kexec 会调用系统调用 sys_kexec_load 将缓存区的映像拷贝到内核动态内存页上。

一旦内核映像被装载后，系统就可以重新启动到它。用户可以使用命令 kexec -e 来开始重新启动到新内核。该命令调用系统调用 sys_reboot 通知内核执行重新启动。用户还可以使用命令 kexec -p 设置为系统崩溃时由内核重新启动到新内核。

在装载转储捕捉内核（新内核）后，如果系统发生崩溃（Kernel Panic），系统将重启动进入转储捕捉内核。重启动的触发点在函数 panic()、die()、die_nmi()和 sysrq 处理例程（按 ALT-SysRq-c 组合键）。

Linux 内核从内存固定地址运行，kexec 装载的新内核需要放置在当前系统内核正运行的位置。在 x86 系统上，内核位于物理地址 0x100000，虚拟地址为 0xc0000000（由宏 PAGE_OFFSET 定义），也就是说，新内核运行必须覆盖旧内核。新内核覆盖旧内核的工作分三个阶段完成：将新内核拷贝到旧内核中、将旧内核映像拷贝到其他内存中、覆盖旧内核，然后启动新内核。

内核映像描述结构

内核映像在内存中以段（与 ELF 的段对应）的形式分段存放，每个段在内存中的位置用结构 kexec_segment 描述，该结构列出如下（在 include/linux/kexec.h 中）：

```
struct kexec_segment {
    /*用户空间的缓存和大小*/
        void __user *buf;
        size_t bufsz;
     /*段的目标位置与大小*/
        unsigned long mem;      /* 用户空间把它看作类型(void *) */
        size_t memsz;
};
```

内核映像在内核中的信息用结构 kimage 描述，它包含各个映像段数组、映像条目的地址、控制代码页的页地址与开始地址、存放目的地的地址与页地址等信息。

结构 kimage 列出如下（在 include/linux/kexec.h 中）：

```
struct kimage {
        kimage_entry_t head;
```

```
        kimage_entry_t *entry;
        kimage_entry_t *last_entry;

        unsigned long destination;

        unsigned long start;
        struct page *control_code_page;
        struct page *swap_page;

        unsigned long nr_segments;
        struct kexec_segment segment[KEXEC_SEGMENT_MAX]; /*段数组*/

        struct list_head control_pages;
        struct list_head dest_pages;
        struct list_head unuseable_pages;

        /* 分配给崩溃内核的下一个控制页的地址*/
        unsigned long control_page;

        /* 指定特殊处理的标识*/
        unsigned int type : 1;
#define KEXEC_TYPE_DEFAULT 0
#define KEXEC_TYPE_CRASH   1
        unsigned int preserve_context : 1;
};
```

内核 kexec 接口函数说明如下：

```
extern void machine_kexec(struct kimage *image);   /*启动内核映像*/
extern int machine_kexec_prepare(struct kimage *image);  /*建立内核映像所需要的控制页*/
extern void machine_kexec_cleanup(struct kimage *image);
extern asmlinkage long sys_kexec_load(unsigned long entry,
                                      unsigned long nr_segments,
                                      struct kexec_segment __user *segments,
                                      unsigned long flags);     /*装载内核的系统调用*/
extern int kernel_kexec(void);   /*启动内核*/
```

重启动内核代码分析

当发出内核崩溃时，例如：当内核发生无法恢复的错误时，系统会调用函数 panic，该函数会调用函数 crash_kexec 重启动内核；当系统无法从硬件中断返回时，系统将调用函数 do_general_protection 处理错误，该函数调用函数 die 根据错误类型决定是否重启动内核。函数 die 的调用层次图如图 1 所示。

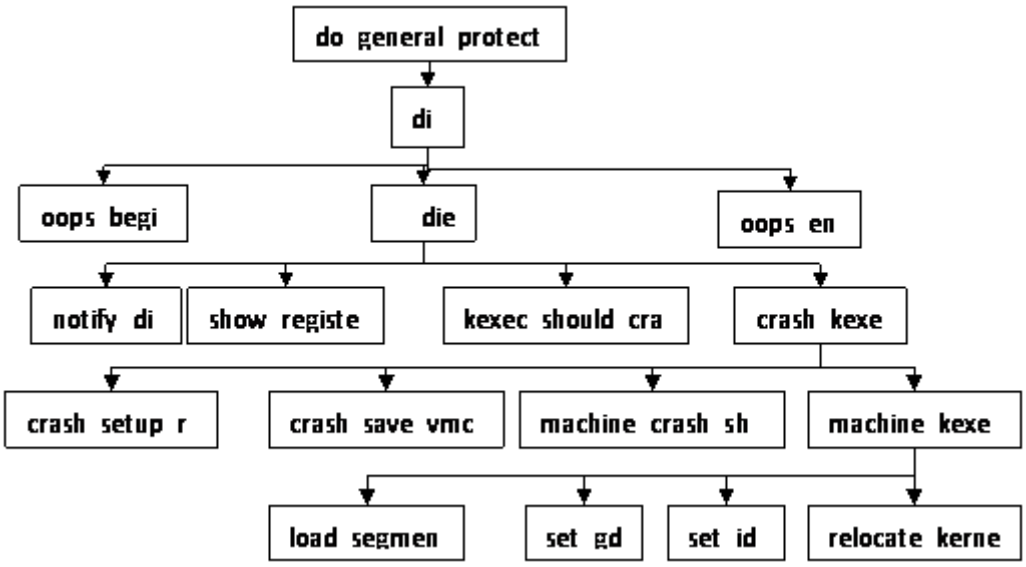

图 1 函数 die 的调用层次图

函数 die 进入 oops 处理，打印错误及现场信息，重启动转储捕捉内核，其列出如下（在 arch/x86/kernel/trap_64.c 中）：

```
void die(const char *str, struct pt_regs *regs, long err)
{
    /*进入 oops 处理时，调用此函数。加锁，存储中断标识，获取 CPU ID，让第一次出错 oops 的 CPU 处理错误，让其他 CPU 暂停*/
        unsigned long flags = oops_begin();

        if (!user_mode(regs))  /*如果寄存器是在保护模式或 8086 模式下设置，返回 1*/
                report_bug(regs->ip, regs);

        if (__die(str, regs, err))
                regs = NULL;
    /*退出 oops 处理时调用此函数，用来解锁、恢复中断标识、退出进程*/
        oops_end(flags, regs, SIGSEGV);
```

```
}
```

函数__die 发出通知 DIE_OOPS 给其他进程处理，判断是否应该让内核崩溃，如果应该崩溃，就调用函数 crash_kexec 启动转储捕捉内核。其列出如下：

```
int __kprobes __die(const char *str, struct pt_regs *regs, long err)
{
        …… /*省略了打印错误信息*/
     /*发出通知 DIE_OOPS，让其他进程处理此消息*/
        if (notify_die(DIE_OOPS, str, regs, err,
                        current->thread.trap_no, SIGSEGV) ==
NOTIFY_STOP)
                return 1;

        show_registers(regs);   /*打印寄存器的值*/
        add_taint(TAINT_DIE);  /*给污点变量 tainted 设置标识，表示内核出错状态*/
        /*打印执行记录小结，以免 oops 信息从屏幕上滚动不见了*/
        printk(KERN_ALERT "RIP ");
        printk_address(regs->ip, 1);
        printk(" RSP <%016lx>\n", regs->sp);
        if (kexec_should_crash(current))  /*检查当前进程是否应该崩溃*/
                crash_kexec(regs);        /*执行内核崩溃重启动转储捕捉内核*/
        return 0;
}
```

函数 kexec_should_crash 判断应该进行内核崩溃的情况：在中断中、pid 为 0、任务是初始化进程、在 oops 中发出崩溃，其列出如下（在 arch/x86/kernel/kexec.c 中）：

```
int kexec_should_crash(struct task_struct *p)
{
        if (in_interrupt() || !p->pid || is_global_init(p) ||
panic_on_oops)
                return 1;
        return 0;
```

```
}
```

函数 crash_kexec 加互斥锁，存储寄存器信息，启动新内核，该函数列出如下（在 arch/x86/kernel/kexec.c 中）：

```
void crash_kexec(struct pt_regs *regs)
{
        /*在这里 kexec_mutex 阻止运行在一个 CPU 上的 sys_kexec_load 替代崩溃后在其他 CPU 上正使用的崩溃内核。如果崩溃内核还没被定位在一个固定内存区域，xchg(&kexec_crash_image)足够用了*/
        if (mutex_trylock(&kexec_mutex)) {
                if (kexec_crash_image) {
                        struct pt_regs fixed_regs;
                        crash_setup_regs(&fixed_regs, regs);  /*将 regs 拷贝到 fixed_regs*、
                        crash_save_vmcoreinfo();   /*添加用于转储的提示信息*/
                        machine_crash_shutdown(&fixed_regs);
                        machine_kexec(kexec_crash_image);
                }
                mutex_unlock(&kexec_mutex);
        }
}
```

函数 machine_kexec 用数组变量 page_list 的格式（由控制页地址、页目录地址格式组成）对应映像头的格式，并将 page_list 传入汇编语言函数 relocate_kernel，便于提取 CPU 寄存器初始化数据。因为此函数在重启动新内核后不再返回，因此，不能在堆上分配内存，而采用栈的形式存放。

函数 machine_kexec 在准备后数据格式后，调用函数 relocate_kernel 定位和重启动新内核。

函数 relocate_kernel 是汇编语言编写的启动新内核之前的引导代码，它在解析页表后，将新内核拷贝替换内存中的旧内核映像，因为 x86 平台内核必须在固定地址启动。

函数 machine_kexec 列出如下（在 arch/x86/kernel/machine_kexec_64.c

中）：

```
#define PAGE_ALIGNED __attribute__ )
static u64 kexec_pgd[512] PAGE_ALIGNED;
static u64 kexec_pud0[512] PAGE_ALIGNED;
static u64 kexec_pmd0[512] PAGE_ALIGNED;
static u64 kexec_pte0[512] PAGE_ALIGNED;
static u64 kexec_pud1[512] PAGE_ALIGNED;
static u64 kexec_pmd1[512] PAGE_ALIGNED;
static u64 kexec_pte1[512] PAGE_ALIGNED;
void machine_kexec(struct kimage *image)
{
        unsigned long page_list[PAGES_NR];
        void *control_page;

        tracer_disable();  /*关闭跟踪*/

        /*关闭中断*/
        local_irq_disable();

    /*计算获取控制页地址*/
        control_page = page_address(image->control_code_page) +
PAGE_SIZE;
     /*将切换定位新内核的过渡代码函数 relocate_kernel 拷贝到控制页，该
函数用汇编语言编写*/
        memcpy(control_page, relocate_kernel, PAGE_SIZE);

        page_list[PA_CONTROL_PAGE] = virt_to_phys(control_page);
        page_list[VA_CONTROL_PAGE] = (unsigned long)relocate_kernel;
    /*用于定位页目录在映像中的位置*/
        page_list[PA_PGD] = virt_to_phys(&kexec_pgd);
        page_list[VA_PGD] = (unsigned long)kexec_pgd;
        page_list[PA_PUD_0] = virt_to_phys(&kexec_pud0);
        page_list[VA_PUD_0] = (unsigned long)kexec_pud0;
        page_list[PA_PMD_0] = virt_to_phys(&kexec_pmd0);
        page_list[VA_PMD_0] = (unsigned long)kexec_pmd0;
        page_list[PA_PTE_0] = virt_to_phys(&kexec_pte0);
        page_list[VA_PTE_0] = (unsigned long)kexec_pte0;
        page_list[PA_PUD_1] = virt_to_phys(&kexec_pud1);
        page_list[VA_PUD_1] = (unsigned long)kexec_pud1;
        page_list[PA_PMD_1] = virt_to_phys(&kexec_pmd1);
        page_list[VA_PMD_1] = (unsigned long)kexec_pmd1;
        page_list[PA_PTE_1] = virt_to_phys(&kexec_pte1);
```

```
        page_list[VA_PTE_1] = (unsigned long)kexec_pte1;

        page_list[PA_TABLE_PAGE] =
          (unsigned long)__pa(page_address(image-
>control_code_page));

        /* 段寄存器有可见和不可见部分，无论何时可见部分设置到指定的选
择子，不可见部分将从内存中的描述表中装载*/
        load_segments();  /*装载段*/
        /*设置 gdt 和 idt 为 0,现在 gdt 和 idt 是无效的，必须建立新的 idt
和 gdt 后再装载它们*/
        set_gdt(phys_to_virt(0),0);
        set_idt(phys_to_virt(0),0);

        /* 调用新的内核*/
        relocate_kernel((unsigned long)image->head, (unsigned
long)page_list,
                        image->start);
}
```

Linux 日志系统

日志是 Linux 安全体系的一个重要组成部分，日志信息提供了攻击发生的真实证据。由于攻击方式多种多样，一旦攻击攻破系统，系统通过日志文件应留下攻击信息。Linux 提供了网络、主机和用户级的日志信息。日志可记录如下内容：

记录系统和内核的运行信息；

记录每一次网络连接和它们的源 IP 地址、长度，有时还包括攻击者的用户名和使用的操作系统；

记录远程用户申请访问哪些文件；

记录用户可以控制哪些进程；

记录具体用户使用的每条命令。

Linux 日志系统概述

日志主要的功能是实时地监测系统状态，监测和追踪侵入者等。Linux 系统有 3

个主要的日志子系统：连接时间日志、进程统计日志和错误日志。

连接时间日志记录在/var/log/wtmp 和/var/run/utmp，用于跟踪谁在何时登录到系统。

进程统计日志（pacct 或 acct）给系统中的基本服务提供命令使用统计。

错误日志由 syslogd 执行，各种系统守护进程、用户程序通过 syslog 向文件/var/og/messages 记录值得注意的事件。

另外，应用程序可以创建日志，如：HTTP 和 FTP 等服务器常用单独的日志文件保持详细的日志信息。审计系统常保存安全审计信息在/var/log/audit.d 目录下的 audit.log 文件中。

常用的日志文件说明如下：

boot.log 记录了系统在引导过程中发生的事件。

cron 记录 crontab 守护进程 crond 所派生的子进程的动作。

¬maillog 记录了电子邮件的活动。

¬ acct/pacct 记录用户命令。¬

lastlog 记录最近几次成功登录的事件和最后一次不成功的登录。¬

¬ messages 从 syslog 中记录信息（有的链接到 syslog 文件）。¬

sulog 记录使用 su 命令的使用。¬

syslog 从 syslog 中记录信息（通常链接到 messages 文件）；¬

utmp 记录当前前登录用户的信息。¬

wtmp¬ 一个用户每次登录进入和退出时间的永久记录，wtmp 和 utmp 文件都是二进制文件，需要使用 who、w、users、last 和 ac 命令查看它们包含的信息。

¬

进程统计记录进程的活动。进程统计缺省下是关闭的，使用下面的命令设置打开进程统计：

```
# accton /var/log/pacct
```

关闭进程统计的命令列出如下：

```
# accton
```

可用 lastcomm 命令报告以前的执行文件在/var/log/pacct 记录的内容。

syslog 系统构架

内核和任何程序都可通过 syslog 系统记录事件消息，并将消息写到一个文件或设备中，还可以通过网络进行传送。syslog 系统构架如图 9。

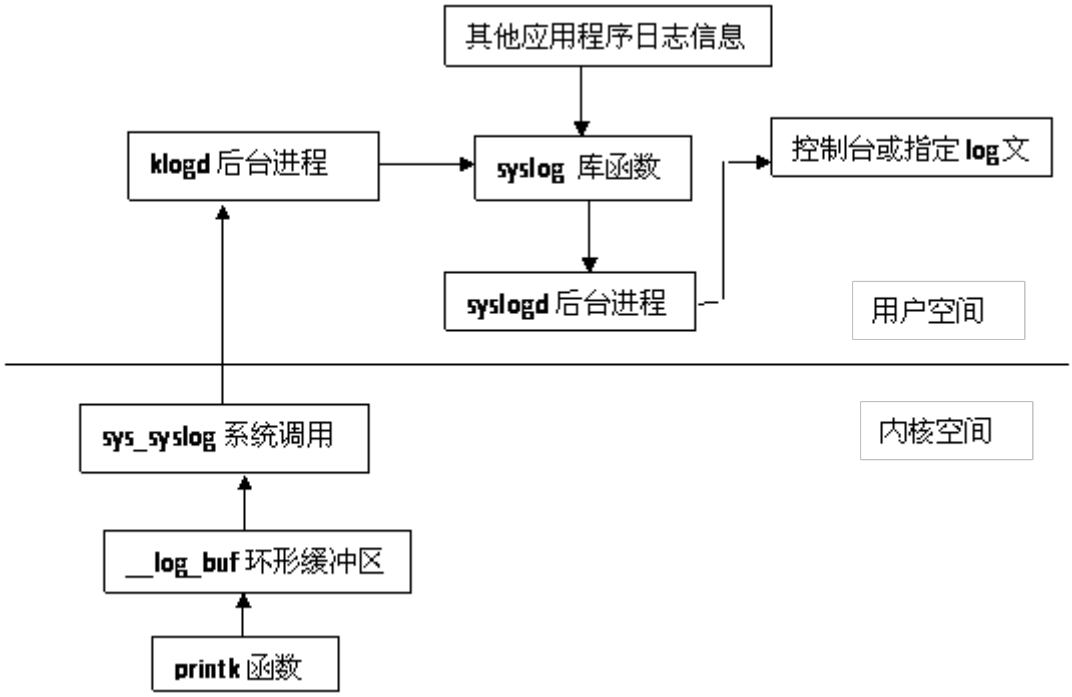

图 9 syslog 系统构架

图 9 中，klogd 后台进程监听和得到内核信息，并发送到 syslogd 后台进程。syslogd 监听和处理来自 syslog 库 API 的所有消息，并输出到控制台或指定文件中。

在内核，函数 printk 将消息写到一个长度为 LOG_BUF_LEN 字节的循环缓冲区中。如果循环缓冲区填满了，printk 就绕回缓冲区的开始处填写新数据，klogd 后台进程读取循环缓冲区中内核消息，并分发到 syslogd。

（1）系统调用 sys_syslog

头 sys/klog.h 提供了函数 syslog 或系统调用 sys_syslog 从内核的 printk 环形缓冲区读取消息，当环形缓冲区为空时，它会阻塞等待。

（2）后台进程 klogd

后台进程 klogd 调用 syslog 读取内核的消息。klogd 允许内核消息打印在系统终端上，优先级小于 7 的任何消息都可打印以终端上。优先级为 7 的消息是调试消息，不能出现在终端上，将由 syslogd 后台定向到文件中。

由内部错误条件引发的消息是非常重要的，开发者需要分析这些消息来判定出错的原因。错误消息以原始数据出现，需要通过符号表映射文件转换成可读的符号信息。程序编译时会产生符号表映射文件，它列出了重要变量和函数的地址位置。内核编译产生的符号映射文件为 system.map，后台进程 klogd 通过 system.map 文件将原始数据转成符号信息，然后，打印在系统终端或通过 syslog 库函数传递给 syslogd 后台。

如果没有运行 klogd，数据将保留在循环缓冲区中，直到某个进程读取或缓冲区溢出为止。

还可通过读取/proc/kmesg 文件获得调试信息。对/proc/kmesg 进行读操作时，日志缓冲区中被读取的数据就不再保留，而 syslog 系统调用却能随意地返回日志数据，并保留这些数据以便其他进程也能使用。

klogd 指定-f (file)选项，可将消息保存到某个特定的文件，还可修改 /etc/syslog.conf 来设置消息存放地点。

（3）syslog 库 API

syslog 库 API 定义在 syslog.h 文件中，它提供了函数 openlog、syslog 和 closelog 用于将应用程序中的日志消息写入日志系统。例如：应用程序 su 的打印日志消息函数列出如下：

```
#include <stdio.h>
#include "system.h"
# include <syslog.h>

static void log_su (struct passwd const *pw, bool successful)
{
  const char *new_user, *old_user, *tty;
```

```
  new_user = pw->pw_name;
  old_user = getlogin ();     //通过日志文件 utmp 得到登录的用户
  if (!old_user)  //如果没得到用户名，就通过 uid 获取
    {
      struct passwd *pwd = getpwuid (getuid ());
      old_user = (pwd ? pwd->pw_name : "");
    }
  tty = ttyname (STDERR_FILENO);     //通过标准描述符 STDERR_FILENO 得
到控制台名
  if (!tty)
    tty = "none";

  // base_name (program_name)表示用程序的基本名字作为消息标签头
  openlog (base_name (program_name), 0 , LOG_AUTH ); // LOG_AUTH 表示
安全或授权消息
  //打印消息
  syslog (LOG_NOTICE,     //消息优先级
          "%s(to %s) %s on %s",
          successful ? "" : "FAILED SU ",
          new_user,
          old_user, tty);
  closelog (); //关闭
}
```

（4）后台进程 syslogd

后台进程 syslogd 根据配置文件/etc/syslog.conf，将消息输出到指定存放地点。配置文件 syslog.conf 定义每类消息的存放地点。通常情况下，syslog 信息写到/var/adm 或/var/log 目录下的信息文件（messages.*）中。一个典型的 syslog 记录包括生成程序的名字和一个文本信息。

syslog.conf 文件每一行对每类消息提供一个选择域和一个动作域。由 tab 隔开：选择域指明消息的类型和优先级，动作域指明 syslogd 执行的动作。

例如，下面配置表示所有邮件消息记录到一个文件中：

```
#Log all the mail messages in one place

mail.* /var/log/maillog
```

下面配置表示将 alert 消息写到 root 和 tiger 的个人账号中：

#Root and Tiger get alert and higher messages

*.alert root,tiger

下面配置表示将内核消息记录到设置/dev/console 中：

#Log all kernel messages to the console

#Logging much else clutters up the screen

kern.* /dev/console

下面配置将 UUCP 和 news 设备产生的错误消息写入到指定日志文件中：

Save news errors of level crit and higher in a special file.

uucp,news.crit /var/log/spooler

printk 及控制台的日志级别

函数 printk 使用方法和 printf 相似，用于内核打印消息。printk 根据日志级别（loglevel）对消息进行分类。

日志级别用宏定义，日志级别宏展开为一个字符串，在编译时由预处理器将它和消息文本拼接成一个字符串，因此 printk 函数中日志级别宏和格式字符串间不能有逗号。

下面是两个 printk 的例子，一个用于打印调试信息；另一个用于打印临界条件信息。

printk(KERN_DEBUG "Here I am: %s:%i\n", _ _FILE_ _, _ _LINE_ _);

printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);

printk 的日志级别定义如下（在 linux26/includelinux/kernel.h 中）：

下面配置表示将 alert 消息写到 root 和 tiger 的个人账号中：

#Root and Tiger get alert and higher messages
*.alert root,tiger

下面配置表示将内核消息记录到设置/dev/console 中：

```
#Log all kernel messages to the console
#Logging much else clutters up the screen
kern.* /dev/console
```

下面配置将 UUCP 和 news 设备产生的错误消息写入到指定日志文件中：

```
# Save news errors of level crit and higher in a special file.
uucp,news.crit /var/log/spooler
```

1.3 printk 及控制台的日志级别

函数 printk 使用方法和 printf 相似，用于内核打印消息。printk 根据日志级别（loglevel）对消息进行分类。
日志级别用宏定义，日志级别宏展开为一个字符串，在编译时由预处理器将它和消息文本拼接成一个字符串，因此 printk 函数中日志级别宏和格式字符串间不能有逗号。
下面是两个 printk 的例子，一个用于打印调试信息；另一个用于打印临界条件信息。

```
printk(KERN_DEBUG "Here I am: %s:%i\n", _ _FILE_ _, _ _LINE_ _);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

printk 的日志级别定义如下（在 linux26/includelinux/kernel.h 中）：

```
#define KERN_EMERG      "<0>"   /*紧急事件消息，系统崩溃之前提示，表示系统不可用*/
#define KERN_ALERT      "<1>"   /*报告消息，表示必须立即采取措施*/
#define KERN_CRIT       "<2>"   /*临界条件，通常涉及严重的硬件或软件操作失败*/
#define KERN_ERR        "<3>"   /*错误条件，驱动程序常用 KERN_ERR 来报告硬件的错误*/
#define KERN_WARNING    "<4>"   /* 警告条件，对可能出现问题的情况进行警告*/
#define KERN_NOTICE     "<5>"   /* 正常但又重要的条件，用于提醒。常用于与安全相关的消息*/
#define KERN_INFO       "<6>"   /* 提示信息，如驱动程序启动时，打印硬件信息*/
#define KERN_DEBUG      "<7>"   ¬/*调试级别的消息*/

extern int console_printk[];
```

```
#define console_loglevel   (console_printk[0])
#define default_message_loglevel   (console_printk[1])
#define minimum_console_loglevel   (console_printk[2])
#define default_console_loglevel   (console_printk[3])
```

日志级别范围从 0 到 7，没有指定日志级别的 printk 语句默认采用的级别是 DEFAULT_MESSAGE_LOGLEVEL，其定义列出如下（在 linux26/kernel/printk.c 中）：

```
/*没有定义日志级别的 printk 使用下面的缺省级别*/
#define DEFAULT_MESSAGE_LOGLEVEL 4     /* KERN_WARNING 警告条件*/
```

内核可把消息打印到当前控制台上，可以指定控制台为字符模式的终端或打印机等。默认情况下，"控制台"就是当前地虚拟终端。

为了更好地控制不同级别的信息显示在控制台上，内核设置了控制台的日志级别 console_loglevel。printk 的日志级别作用是打印一定级别的消息，与之类似，控制台只显示一定级别的消息。

当日志级别小于 console_loglevel 时，消息才能显示出来。控制台相应的日志级别定义如下：

```
/* 显示比这个级别更重发的消息*/
#define MINIMUM_CONSOLE_LOGLEVEL   1      /*可以使用的最小日志级别*/
#define DEFAULT_CONSOLE_LOGLEVEL   7      /*比 KERN_DEBUG 更重要的消息都被打印*/

int console_printk[4] = {
        DEFAULT_CONSOLE_LOGLEVEL,       /*控制台日志级别，优先级高于该值的消息将在控制台显示*/
    /*缺省消息日志级别，printk 没定义优先级时，打印这个优先级以上的消息*/
        DEFAULT_MESSAGE_LOGLEVEL,
    /*最小控制台日志级别，控制台日志级别可被设置的最小值（最高优先级）*/
        MINIMUM_CONSOLE_LOGLEVEL,
        DEFAULT_CONSOLE_LOGLEVEL,       /* 缺省控制台日志级别*/
```

```
};
```

如果系统运行了 klogd 和 syslogd，则无论 console_loglevel 为何值，内核消息都将追加到/var/log/messages 中。如果 klogd 没有运行，消息不会传递到用户空间，只能查看/proc/kmsg 了。

变量 console_loglevel 的初始值是 DEFAULT_CONSOLE_LOGLEVEL，可以通过 sys_syslog 系统调用进行修改。调用 klogd 时可以指定-c 开关选项来修改这个变量。如果要修改它的当前值，必须先杀掉 klogd，再加-c 选项重新启动它。

通过读写/proc/sys/kernel/printk 文件可读取和修改控制台的日志级别。查看这个文件的方法如下：

#cat /proc/sys/kernel/printk

6 4 1 7

上面显示的 4 个数据分别对应控制台日志级别、缺省的消息日志级别、最低的控制台日志级别和缺省的控制台日志级别。

可用下面的命令设置当前日志级别：

echo 8 > /proc/sys/kernel/printk

printk 打印消息机制

在内核中，函数 printk 将消息打印到环形缓冲区__log_buf 中，并将消息传给控制台进行显示。控制台驱动程序根据控制台的日志级别显示日志消息。

应用程序通过系统调用 sys_syslog 管理环形缓冲区__log_buf，它可以读取数据、清除缓冲区、设置日志级别、开/关控制台等。

当系统调用 sys_syslog 从环形缓冲区__log_buf 读取数据时，如果缓冲区没有数据，系统调用 sys_syslog 所在进程将被加入到等待队列 log_wait 中进行等待。当 printk 打印数据到缓冲区后，将唤醒系统调用 sys_syslog 所在进程从缓冲区中读取数据。等待队列 log_wait 定义如下：

DECLARE_WAIT_QUEUE_HEAD(log_wait);//等待队列 log_wait

环形缓冲区__log_buf 在使用之前就是已定义好的全局变量，缓冲区的长度为 1

<< CONFIG_LOG_BUF_SHIFT。变量 CONFIG_LOG_BUF_SHIFT 在内核编译时由配置文件定义，对于 i386 平台，其值定义如下（在 linux26/arch/i386/defconfig 中）：

CONFIG_LOG_BUF_SHIFT=18

内核编译时，编译器根据配置文件的设置，产生如下宏定义：

#define CONFIG_LOG_BUF_SHIFT 18

环形缓冲区__log_buf 定义如下（在 linux26/kernel/printk.c 中）：

```
#define __LOG_BUF_LEN  (1 << CONFIG_LOG_BUF_SHIFT)  //定义环形缓冲区的长度，i386 平台为
static char __log_buf[__LOG_BUF_LEN];  //printk 的环形缓冲区
static char *log_buf = __log_buf;
static int log_buf_len = __LOG_BUF_LEN;
/*互斥锁 logbuf_lock 保护 log_buf、log_start、log_end、con_start 和 logged_chars */
static DEFINE_SPINLOCK(logbuf_lock);
```

通过宏定义 LOG_BUF，缓冲区__log_buf 具备了环形缓冲区的操作行为。宏定义 LOG_BUF 得到缓冲区指定位置序号的字符，位置序号超过缓冲区长度时，通过与长度掩码 LOG_BUF_MASK 进行逻辑与操作，位置序号循环回到环形缓冲区中的位置。

宏定义 LOG_BUF 及位置序号掩码 LOG_BUF_MASK 的定义列出如下：

```
#define LOG_BUF_MASK     (log_buf_len-1)
#define LOG_BUF(idx)   (log_buf[(idx) & LOG_BUF_MASK])
```

为了指明环形缓冲区__log_buf 字符读取位置，定义了下面的位置变量：

```
static unsigned long log_start;        /*系统调用 syslog 读取的下一个字符*/
static unsigned long con_start;        /*送向控制台的下一个字符*/
static unsigned long log_end;  /*最近已写字符序号加 1 */
```

```
static unsigned long logged_chars; /*自从上一次 read+clear 操作以来产生的字符数*/
```

内核任何地方调用可以调用函数 printk 打印调试、安全、提示和错误消息。函数 printk 尝试得到控制台信号量（console_sem），如果得到，就将信息输出到环形缓冲区__log_buf 中，然后函数 release_console_sem()在释放信号量之前把环形缓冲区中的消息送到控制台，调用控制台驱动程序显示打印的信息。如果没得到信号量，就只将信息输出到环形缓冲区后返回。函数 printk 的调用层次如图 19。

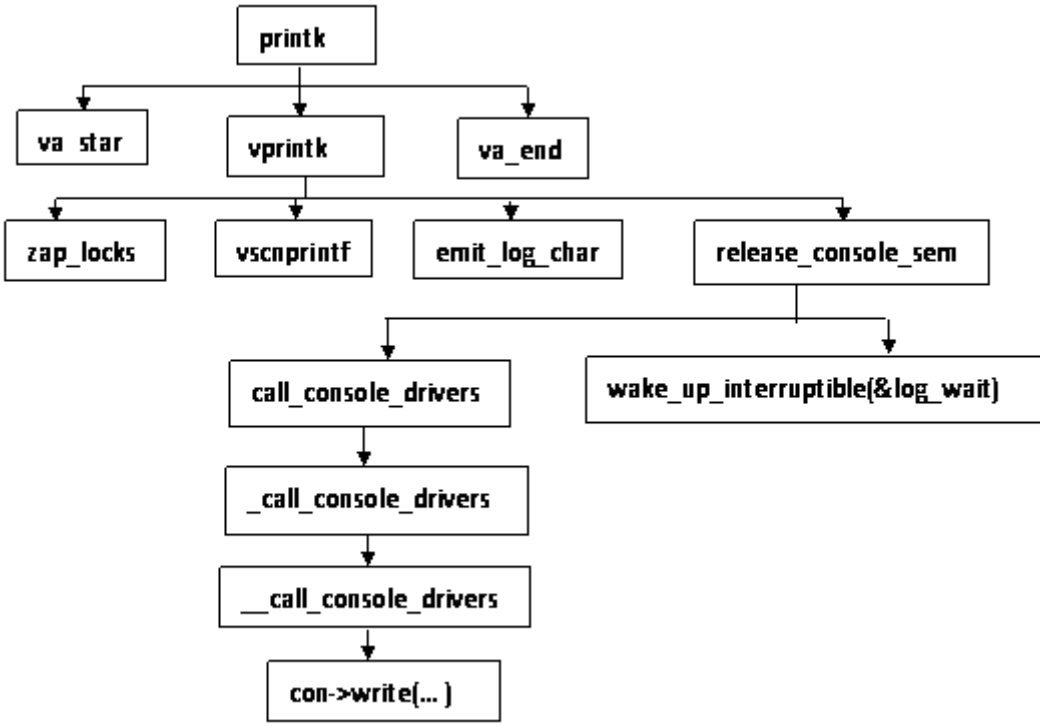

图 19 函数 printk 的调用层次图

函数 printk 列出如下（在 linux26/kernel/printk.c 中）：

```
asmlinkage int printk(const char *fmt, ...)
{
        va_list args;
        int r;

        va_start(args, fmt);
```

```
        r = vprintk(fmt, args);
        va_end(args);

        return r;
}

asmlinkage int vprintk(const char *fmt, va_list args)
{
        unsigned long flags;
        int printed_len;
        char *p;
        static char printk_buf[1024];
        static int log_level_unknown = 1;

        preempt_disable();  //关闭内核抢占
        if (unlikely(oops_in_progress) && printk_cpu ==
smp_processor_id())
                /*如果在 printk 运行时，这个 CPU 发生崩溃，确信不能死
锁，10 秒 1 次初始化锁 logbuf_lock 和 console_sem，留时间给控制台打印完
全的 oops 信息*/
                zap_locks();

        local_irq_save(flags);  //存储本地中断标识
        lockdep_off();
        spin_lock(&logbuf_lock);
        printk_cpu = smp_processor_id();

        /*将输出信息发送到临时缓冲区 printk_buf */
        printed_len = vscnprintf(printk_buf, sizeof(printk_buf), fmt,
args);

        /*拷贝 printk_buf 数据到循环缓冲区，如果调用者没提供合适的日志
级别，插入默认值*/
        for (p = printk_buf; *p; p++) {
                if (log_level_unknown) {
                        /* log_level_unknown signals the start of a
new line */
                    if (printk_time) {
                            int loglev_char;
                            char tbuf[50], *tp;
                            unsigned tlen;
                            unsigned long long t;
                            unsigned long nanosec_rem;
```

```
                                /*在时间输出之前强制日输出志级别*/
                                if (p[0] == '<' && p[1] >='0' &&
                                   p[1] <= '7' && p[2] == '>') {
                                        loglev_char = p[1];      //获
取日志级别字符
                                        p += 3;
                                        printed_len -= 3;
                                } else {
                                        loglev_char =
default_message_loglevel
                                                + '0';
                                }
                                t = printk_clock();    //返回当前时
钟，以 ns 为单位
                                nanosec_rem = do_div(t, 1000000000);
                                tlen = sprintf(tbuf,
                                                "<%c>[%5lu.%06lu] ",
                                                loglev_char,
                                                (unsigned long)t,
                                                nanosec_rem/1000);
//写入格式化后的日志级别和时间

                                for (tp = tbuf; tp < tbuf + tlen;
tp++)
                                        emit_log_char(*tp);   //输出日
志级别和时间字符到循环缓冲区
                                printed_len += tlen;
                        } else {
                                if (p[0] != '<' || p[1] < '0' ||
                                   p[1] > '7' || p[2] != '>') {
                                        emit_log_char('<');

        emit_log_char(default_message_loglevel
                                                + '0');   //输出字符到
循环缓冲区
                                        emit_log_char('>');
                                        printed_len += 3;
                                }
                        }
                        log_level_unknown = 0;
                        if (!*p)
                                break;
                }
                emit_log_char(*p);    //输出其他 printk_buf 数据到循环
```

缓冲区

```
                if (*p == '\n')
                        log_level_unknown = 1;
        }

        if (!down_trylock(&console_sem)) {
                /*拥有控制台驱动程序，降低 spinlock 并让
release_console_sem()打印字符 */
                console_locked = 1;
                printk_cpu = UINT_MAX;
                spin_unlock(&logbuf_lock);

                /*如果 CPU 准备好，控制台就输出字符。函数 cpu_online 检
测 CPU 是否在线，函数 have_callable_console()检测是否有注册的控制台大
启动时就可以使用*/
                if (cpu_online(smp_processor_id()) ||
have_callable_console()) {
                        console_may_schedule = 0;
                        release_console_sem();
                } else {
                        /*释放锁避免刷新缓冲区*/
                        console_locked = 0;
                        up(&console_sem);
                }
                lockdep_on();
                local_irq_restore(flags); //恢复本地中断标识
        } else {
                /*如果其他进程拥有这个驱动程序，本线程降低 spinlock，
允许信号量持有者运行并调用控制台驱动程序输出字符*/
                printk_cpu = UINT_MAX;
                spin_unlock(&logbuf_lock);
                lockdep_on();
                local_irq_restore(flags); //恢复本地中断标识
        }

        preempt_enable();  //开启抢占机制
        return printed_len;
}
```

函数 release_console_sem()给控制台系统开锁，释放控制台系统及驱动程序调用者持有的信号量。持有信号量时，表示 printk 已在缓冲区存有数据。函数 release_console_sem()在释放信号量之前将这些数据送给控制台显示。如果后

台进程 klogd 在等待环形缓冲区装上数据，它唤醒 klogd 进程。

函数 release_console_sem 列出如下（在 linux26/kernel/printk.c 中）：

```
void release_console_sem(void)
{
        unsigned long flags;
        unsigned long _con_start, _log_end;
        unsigned long wake_klogd = 0;

        for ( ; ; ) {
                spin_lock_irqsave(&logbuf_lock, flags);
                wake_klogd |= log_start - log_end;
                if (con_start == log_end)
                        break;                  /* 没有需要打印的数据*/
                _con_start = con_start;
                _log_end = log_end;
                con_start = log_end;            /* Flush */
                spin_unlock_irqrestore(&logbuf_lock, flags);
          //调用控制台 driver 的 write 函数写到控制台
                call_console_drivers(_con_start, _log_end);
        }
        console_locked = 0;
        console_may_schedule = 0;
        up(&console_sem);
        spin_unlock_irqrestore(&logbuf_lock, flags);
        if (wake_klogd && !oops_in_progress &&
waitqueue_active(&log_wait))
                wake_up_interruptible(&log_wait);//唤醒在等待队列上的
进程
}
```

函数_call_console_drivers 将缓冲区中从 start 到 end - 1 的数据输出到控制台进行显示。在输出数据到控制台之前，它检查消息的日志级别。只有日志级别小于控制台日志级别 console_loglevel 的消息，才能交给控制台驱动程序进行显示。

函数_call_console_drivers 列出如下：

```
static void _call_console_drivers(unsigned long start,
```

```
                                unsigned long end, int msg_log_level)
{
    //日志级别小于控制台日志级别的消息才能输向控制台
        if ((msg_log_level < console_loglevel || ignore_loglevel) &&
                        console_drivers && start != end) {
                if ((start & LOG_BUF_MASK) > (end & LOG_BUF_MASK)) {
                        /* 调用控制台驱动程序的写操作函数 */
                        __call_console_drivers(start & LOG_BUF_MASK,
log_buf_len);
                        __call_console_drivers(0, end & LOG_BUF_MASK);
                } else {
                        __call_console_drivers(start, end);
                }
        }
}
```

函数__call_console_drivers 调用控制台驱动程序的写操作函数显示消息。其列出如下：

```
static void __call_console_drivers(unsigned long start, unsigned long
end)
{
        struct console *con;

        for (con = console_drivers; con; con = con->next) {
                if ((con->flags & CON_ENABLED) && con->write &&
                                (cpu_online(smp_processor_id()) ||
                                (con->flags & CON_ANYTIME)))
                        con->write(con, &LOG_BUF(start), end - start);
//调用驱动程序的写操作函数
        }
}
```

系统调用 sys_syslog

系统调用 sys_syslog 根据参数 type 的命令执行相应的操作。参数 type 定义的命令列出如下：

0 -- 关闭日志，当前没实现。

1 -- 打开日志，当前没实现。

2 -- 从环形缓冲区读取日志消息。

3 --读取保留在环形缓冲区的所有消息。

4 --读取并清除保留在环形缓冲区的所有消息。

5 -- 清除环形缓冲区。

6 -- 关闭 printk 到控制台的打印。

7 -- 开启 printk 到控制台的打印。

8 -- 设置打印到控制台的消息的日志级别。

9 -- 返回日志缓冲区中没读取的字符数。

10 -- 返回日志缓冲区的大小。

sys_syslog 函数列出如下（在 linux26/kernel/printk.c 中）：

```
asmlinkage long sys_syslog(int type, char __user * buf, int len)
{
        return do_syslog(type, buf, len);
}

int do_syslog(int type, char __user *buf, int len)
{
        unsigned long i, j, limit, count;
        int do_clear = 0;
        char c;
        int error = 0;

        error = security_syslog(type);  //检查是否调用这个函数的权限
        if (error)
                return error;

        switch (type) {
        case 0:         /* 关闭日志 */
                break;
```

```
        case 1:         /* 打开日志*/
                break;
        case 2:         /*读取日志信息*/
                error = -EINVAL;
                if (!buf || len < 0)
                        goto out;
                error = 0;
                if (!len)
                        goto out;
                if (!access_ok(VERIFY_WRITE, buf, len)) {  //验证是否
有写的权限
                        error = -EFAULT;
                        goto out;
                }
        //当 log_start - log_end 为 0 时，表示环形缓冲区无数据可读，把
当前进程放入等待队列 log_wait
                error = wait_event_interruptible(log_wait,
(log_start - log_end));
                if (error)
                        goto out;
                i = 0;
                spin_lock_irq(&logbuf_lock);
                while (!error && (log_start != log_end) && i < len) {
                        c = LOG_BUF(log_start);  //从环形缓冲区得到读
取位置 log_start
                        log_start++;
                        spin_unlock_irq(&logbuf_lock);
                        error = __put_user(c,buf);  //将 c 地址的字符传
递给用户空间的 buf 中
                        buf++;
                        i++;
                        cond_resched();  //条件调度，让其他进程有运行
时间
                        spin_lock_irq(&logbuf_lock);
                }
                spin_unlock_irq(&logbuf_lock);
                if (!error)
                        error = i;
                break;
        case 4:         /* 读/清除上一次内核消息*/
                do_clear = 1;
                /* FALL THRU */
        case 3:         /*读取上一次内核消息*/
                error = -EINVAL;
```

```
                if (!buf || len < 0)
                        goto out;
                error = 0;
                if (!len)  //读取长度为 0
                        goto out;
                if (!access_ok(VERIFY_WRITE, buf, len)) {  //验证有写
权限
                        error = -EFAULT;
                        goto out;
                }
                count = len;
                if (count > log_buf_len)
                        count = log_buf_len;
                spin_lock_irq(&logbuf_lock);
                if (count > logged_chars)  // logged_chars 是上次读/清
除以来产生的日志字符数
                        count = logged_chars;
                if (do_clear)
                        logged_chars = 0;
                limit = log_end;
                /* __put_user() 可以睡眠，当__put_user 睡眠时，
printk()可能覆盖写正拷贝到用户空间的消息，因此，这些消息被反方向拷
贝，将 buf 覆盖部分的数据重写到 buf 起始位置*/
                for (i = 0; i < count && !error; i++) {  //读取 count
个字符
                        j = limit-1-i;
                        if (j + log_buf_len < log_end)
                                break;
                        c = LOG_BUF(j);     //从环形缓冲区得到读取位置
j
                        spin_unlock_irq(&logbuf_lock);
                //将 c 位置的字符传递给用户空间的 buf 中，如果发生错误，
将发生错误的 c 位置给 error
                        error = __put_user(c,&buf[count-1-i]);
                        cond_resched();
                        spin_lock_irq(&logbuf_lock);
                }
                spin_unlock_irq(&logbuf_lock);

                if (error)
                        break;
                error = i;
                if (i != count) {  //表示__put_user 没有拷贝完成
                        int offset = count-error;
```

```
                        /* 拷贝期间缓冲区溢出，纠正用户空间缓冲区*/
                        for (i = 0; i < error; i++) {
                                if (__get_user(c,&buf[i+offset]) ||
                                    __put_user(c,&buf[i])) {  //将覆盖
部分的数据重写到 buf 起始位置
                                        error = -EFAULT;
                                        break;
                                }
                                cond_resched();
                        }
                }
                break;
        case 5:         /* 清除环形缓冲区*/
                logged_chars = 0;
                break;
        case 6:         /*关闭向控制台输出消息*/
                console_loglevel = minimum_console_loglevel;
                break;
        case 7:         /*开启向控制台输出消息*/
                console_loglevel = default_console_loglevel;
                break;
        case 8:         /* 设置打印到控制台的日志级别*/
                error = -EINVAL;
                if (len < 1 || len > 8)
                        goto out;
                if (len < minimum_console_loglevel)
                        len = minimum_console_loglevel;
                console_loglevel = len;
                error = 0;
                break;
        case 9:         /* 得到日志消息所占缓冲区的大小*/
                error = log_end - log_start;
                break;
        case 10:        /*返回环形缓冲区大小*/
                error = log_buf_len;
                break;
        default:
                error = -EINVAL;
                break;
        }
out:
        return error;
}
```

内核探测 kprobe

kprobe（内核探测，kernel probe）是一个动态地收集调试和性能信息的工具，内核探测分为 kprobe, jprobe 和 kretprobe（也称 return probe，返回探测）三种。kprobe 可插入内核中任何指令处；jprobe 插入内核函数入口，方便于访问函数的参数；return probe 用于探测指定函数的返回值。

kprobe 的构架如图 1 所示，在 x86 构架上，kprobe 利用例外处理机制，它修改了标准断点、调试和一些其他的例外处理例程。大多数探测的处理在断点上下文和调试例外处理例程中完成。调试例外处理例程组成了 kprobe 构架依赖层（Achitecture Dependent Layer）。kprobe 非构架依赖层（Architecture Independent Layer）是 kprobe 管理器，用于注册和注销探测函数。用户在内核模块提供由 kprobe 管理器注册的探测处理例程。

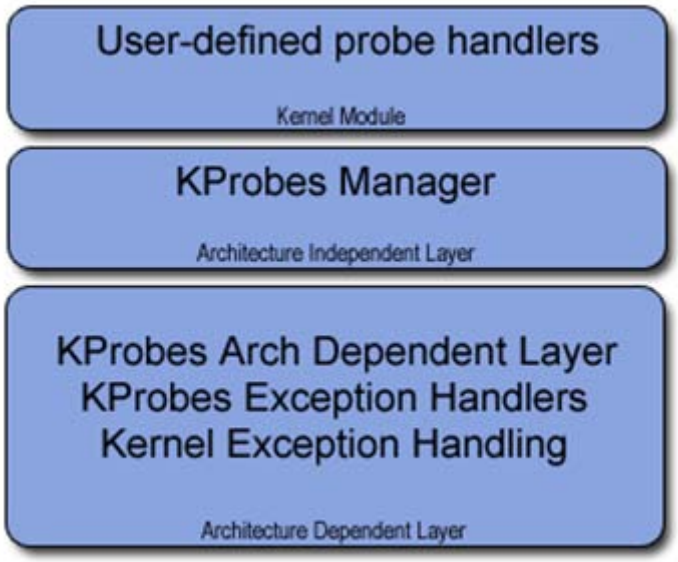

图 1 kprobe 构架示意图

内核探测工作机制及对象结构

（1）kprobe 工作机制

kprobe 工作机制示意图如图 1 所示。处理探测的步骤依赖于构架，当注册一个 kprobe 处理例程函数后，kprobe 备份被探测的指令，并将断点指令(在 i386 和 x86_64 的 int3 指令)替代被探测指令的第一个（或几个）字节。当 CPU 执行到

探测点时，它执行断点指令 int3，产生一个"陷阱"（trap），CPU 的寄存器值被保存。指令 int3 被执行产生陷阱，引起控制到达陷阱处理例程 do_int3。该例程通过一个中断门调用，因此，控制到达时应关闭中断。

陷阱处理例程 do_int3 通过 notifier_call_chain 机制（内核的异步通知机制）将控制传递给 kprobe 注册的处理例程。kprobe 检查探测点是否注册了探测处理例程，如果未设置，返回 0，如果设置了探测处理例程函数，就调用它。

kprobe 执行与 kprobe 相关的预处理例程"pre_handler"，将结构 kprobe 实例和存储的寄存器值的地址传递给处理例程。

接着，kprobe 单步执行已探测指令的拷贝。在单步执行完指令后，如果有后处理例程"post_handler"，kprobe 将执行它。以后，处理器将继续执行探测点后的指令。

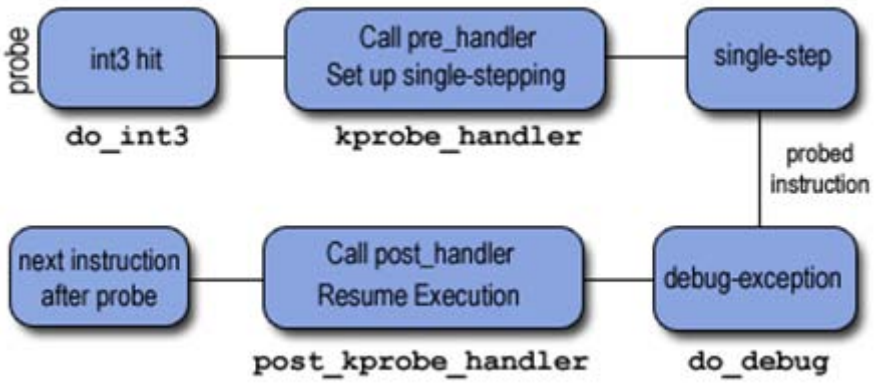

图 1 kprobe 工作机制示意图

结构 kprobe 描述了内核探测点的指令、地址、探测例程等信息，是内核探测的对象结构。其列出如下（在 include/linux/kprobe.h 中）：

```
struct kprobe {
        struct hlist_node hlist;    /*哈希链表*/

        /* 多个处理例程支持的 kprobe 链表 */
        struct list_head list;
        /*引用计数指示相应内核模块被引用了多少次*/
        unsigned int mod_refcounted;
        /*探测临时失败的次数*/
        unsigned long nmissed;
        /*探测点的位置*/
```

```
        kprobe_opcode_t *addr;
    /*注册函数指定探测点的符号名或地址偏移，如果同时指定两个，探测会返回-EINVAL 错误*/
        /*允许用户指示探测点的符号名*/
        const char *symbol_name;
        /*探测点的符号地址偏移，用于计算探测点*/
        unsigned int offset;
        /*在 addr 被执行前调用的预处理例程*/
        kprobe_pre_handler_t pre_handler;
        /*在 addr 被执行完成后调用的后处理例程*/
        kprobe_post_handler_t post_handler;
        /*如果执行 addr 过程中引起了一个失败错误（如：页错误），就调用此函数。如果此函数处理了错误，它返回 1，否则，内核将可以看见错误*/
        kprobe_fault_handler_t fault_handler;
        /*如果在探测处理例程中有断点陷阱发生，调用此函数。如果此函数处理了断点，它返回 1，否则，内核将可以看见错误*/
        kprobe_break_handler_t break_handler;
        /*存储的程序操作代码（该代码被断点替换）*/
        kprobe_opcode_t opcode;
        /* 最初指令的拷贝，与处理器构架相关*/
        struct arch_specific_insn ainsn;
};
```

（2）jprobe（函数入口探测）工作机制

jprobe 通过在函数入口点的 kprobe 机制实现。它利用简单的镜像原理允许对探测函数的参数进行无缝地访问。jprobe 处理例程应与被探测的函数有同样的原型（同样的参数列表和返回值类型），并总是通过调用 kprobe 函数 jprobe_return 进行结束。

当运行到探测点时，kprobe 备份已存储的寄存器值和栈的通用部分。接着，kprobe 将存储的指令指针指到 jprobe 的处理例程，该例程与被探测的函数有同样的寄存器和栈内核。当处理例程运行完后，它调用函数 jprobe_return，再次产生陷阱，恢复最初的栈内核和处理器状态，并切换到被探测的函数中运行。

注意：被探测函数的参数可通过栈或寄存器传递。jprobe 在这两种情况都能工作，只要 jprobe 处理例程函数原型与被探测函数完全一样。因为 jprobe 同时备份了栈和寄存器。

结构 jprobe 是函数入口内核探测的对象结构，它从基类结构 kprobe 继承，描

述了被探测的函数入口地址。其列出如下（在 include/linux/kprobe.h 中）：

```
struct jprobe {
        struct kprobe kp;
        void *entry;    /*探测处理代码将跳入的入口地址*/
};
```

jprobe 的工作机制示意图如图 2 所示。jprobe 必须传递控制到另一个与被探测函数原型相同的函数，并接着将控制给回被探测的函数，状态与 jprobe 被执行前一样。jprobe 使用了 kprobe 的机制，不同的是 jprobe 定义自己的 pre-handler 为 setjmp_pre_handler()，并用另一个例程 break_handler 定义为函数 longjmp_break_handler。jprobe 通过三步实现，分别说明如下：

第一步，当探测点运行时，控制到达基类 kprobe 的处理例程 kprobe_handler()，该例程注册时设置为 setjmp_pre_handler()。函数 setjmp_pre_handler 在切换指令指针 eip 到用户定义的处理例程函数之前存储栈和寄存器内容。接着，函数 setjmp_pre_handler 返回 1，告诉 kprobe_handler()简单地返回，而不是建立像 kprobe 一样的单步调试。在返回时，控制到达用户定义的函数（User Jprobe Handler），访问被探测函数的参数。在用户定义的函数执行完后，它调用函数 jprobe_return()替换正常的返回。

第二步，函数 jprobe_return 截去当前的栈帧，并产生断点，通过 do_int3()传输控制到 kprobe_handler()。kprobe_handler()发现已产生的断点地址（即在 jprobe_handler()中的 int3 指令地址）还没有一个注册的 probe，但 kprobe 在当前 CPU 是激活的。它假定断点一定是由 jprobe 产生的。因此，调用 current_kprobe 以前存储的例程 break_handler（即函数 longjmp_break_handler）。例程 break_handler 恢复栈和寄存器内容，这些内容是在控制传输到用户定义的函数并返回之前被存储的。

第三步 kprobe_handler()建立指令的单步调试，剩余的操作序列与 kprobe 一样。

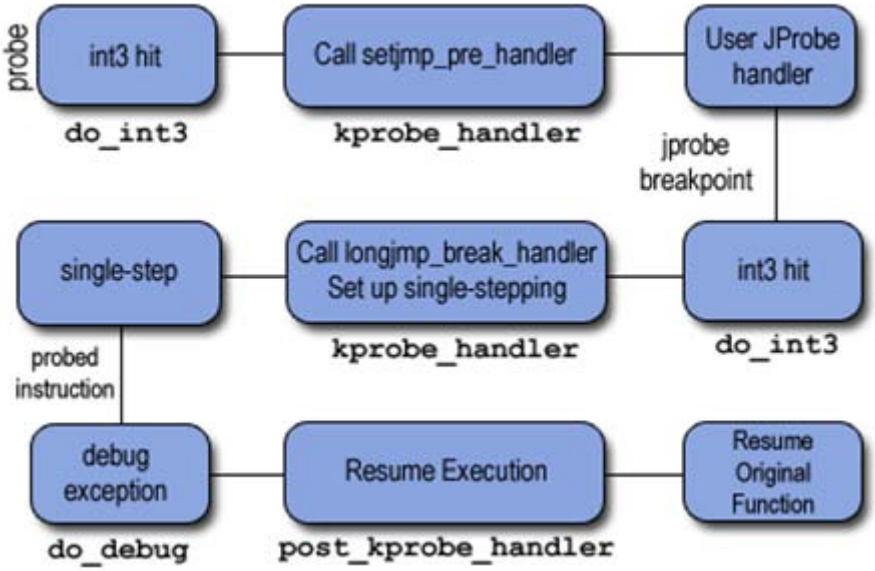

图 2 jprobe 的工作机制示意图

（3）kretprobe（函数返回探测）工作机制

当内核调用函数 register_kretprobe 时，kprobe 在函数的入口点建立 kprobe。当内核程序调用被探测的函数时，探测点被运行，kprobe 存储函数返回地址的拷贝，并用一个"trampoline"替换返回地址。"trampoline"是一个任意的代码块，通常可以是一个 nop 指令。在启动时，kprobe 在 trampoline 中注册了一个 kprobe。

当被探测函数执行到它的返回点时，探测点将控制传递给 trampoline，kprobe 开始运行。

kprobe 的 trampoline 处理例程调用用户定义的与 kretprobe 相关的返回处理例程。在返回处理例程执行完后，设置指令指针到已存储的返回地址，恢复到原来函数的返回地址进行正常执行。

函数返回探测对象用结构 kretprobe 描述，该结构从基类结构 kprobe 继承，描述了处理例程、可被同时激活的被探测函数的最大实例数等信息。其列出如下：

```
struct kretprobe {
        struct kprobe kp;
        kretprobe_handler_t handler;     /*探测处理例程*/
        kretprobe_handler_t entry_handler;
        int maxactive;       /*可被同时激活的被探测函数的最大实例数*/
```

```
        int nmissed;        /*被探测函数的返回被忽略的次数，由于
maxactive 太小引起*/
        size_t data_size;
        struct hlist_head free_instances;
        spinlock_t lock;
};
```

被探测函数的返回地址保存在类型为 kretprobe_instance 的对象中。在调用函数 register_kretprobe 之前，用户设置结构 kretprobe 的域 maxactive，指定可同时探测指定函数的实例个数。函数 register_kretprobe 预分配 kretprobe_instance 对象的个数。该对象结构 kretprobe_instance 列出如下：

```
struct kretprobe_instance {
        struct hlist_node hlist;
        struct kretprobe *rp;       /*指向返回探测对象*/
        kprobe_opcode_t *ret_addr; /*被探测函数的返回地址*/
        struct task_struct *task;     /*被探测函数所在的任务*/
        char data[0];             /*每个实例的私有数据*/
};
```

例如：如果函数是非递归的，并且被调用时持有自旋锁，则 maxactive 为 1 就足够了；如果被探测函数是非递归的，并且从不会放弃 CPU（如：通过信号量或抢占机制占据 CPU），那么， maxactive 为 NR_CPUS 就足够了；如果 maxactive<= 0, 它会被设置到缺省值。如果编译内核时配置 CONFIG_PREEMPT，打开内核抢占功能，那么 maxactive 的值为 max(10, 2*NR_CPUS)，否则，缺省值为 NR_CPUS。

如果 maxactive 设置得太小，将仅会丢失一些探测，不会影响其他方面。当注册了 kretprobe 机制时，结构 kretprobe 的域 nmissed 被设置到 0,并且每次进入探测函数却没有可用的 kretprobe_instance 对象用于建立 kretprobe 机制时，域 nmissed 就加 1。

Kretprobes 还提供了可选的用户指定处理例程，运行在函数入口点。该例程通过设置结构 kretprobe 的域 entry_handler 指定。无论何时 kretprobe 放置 kprobe 在函数的入口点处，如果用户定义的 entry_handler 的存在，它将被触

发运行。

如果 entry_handler 返回 0（表示执行成功）,那么，一个相应的返回例程可确保在函数返回处被调用。如果返回非 0（表示执行出错），kprobe 将留下原始的返回地址，kretprobe 对特定的函数实例没有其他更多的影响。

可用单个 kretprobe_instance 对象匹配多个入口和返回例程调用。用户还可以给每个返回实例指定私有数据，作为每个 kretprobe_instance 对象的部分。当在用户入口和返回处理例程之间共离私有数据时，此方法很有用。每个私有数据对象的大小可在 kretprobe 注册时，通过设置结构 kretprobe 的域 data_size 指定。该数据可通过每个 kretprobe_instance 对象的域 data 进行访问。

在进入被探测函数但没有可用的 kretprobe_instance 对象时，还将增加 nmissed 的计数值，程序将跳过用户的 entry_handler 调用运行。

kprobe 的接口函数

kprobe 为每一类型的探测点提供了注册和卸载函数。分别说明如下：

-

函数 register_kprobe

函数 register_kprobe 注册一个 kprobe 类型的探测点，参数 kp 指向探测对象。该函数调用成功时返回 0，否则返回负错误码。其函数原型列出如下（在 include/linux/kprobes.sh 中）：

```
int register_kprobe(struct kprobe *kp);
```

该注册函数会在 kp->addr 地址处注册一个 kprobes 类型的探测点，当执行到该探测点时，将调用函数 kp->pre_handler，执行完被探测函数后，将调用 kp->post_handler。如果在执行 kp->pre_handler 或 kp->post_handler 时或在单步跟踪被探测函数期间发生错误，将调用 kp->fault_handler。

探测点预处理函数 pre_handler 的原型列出如下：

```
int pre_handler(struct kprobe *p, struct pt_regs *regs);
```

用户必须按该原型参数格式定义自己的 pre_handler，参数 p 指向探测对象，

参数 regs 指向在探测点处保存的寄存器内容。

探测点后处理函数 post_handler 的原型列出如下：

```
void post_handler(struct kprobe *p, struct pt_regs *regs, unsigned long flags);
```

错误处理函数 fault_handler 的原型列出如下：

```
int fault_handler(struct kprobe *p, struct pt_regs *regs, int trapnr);
```

参数 trapnr 是与错误处理相关的依赖于处理器架构的陷阱号，例如：i386 的保护错误是 13，页失效错误是 14。如果成功地处理了异常，它应返回 1。

-

函数 register_jprobe

函数 register_jprobe 用于注册 jprobes 类型的探测点，其原型列出如下：

```
int register_jprobe(struct jprobe *jp);
```

该注册函数在 jp->kp.addr 注册一个 jprobes 类型的探测点，该地址必须是被探测函数的第一条指令的地址，当内核运行到该探测点时，jp->entry 指定的处理函数会被执行，该处理函数的参数列表和返回类型应与被探测函数完全相同，并且它在返回前必须调用函数 jprobe_return。如果注册成功，该函数返回 0，否则返回负的错误码。

-

函数 register_kretprobe

函数 register_kretprobe 用于注册类型为 kretprobes 的探测点，其原型列出如下：

```
int register_kretprobe(struct kretprobe *rp);
```

该注册函数在地址 rp->kp.addr 注册了一个 kretprobe 类型的探测对象，当被探测函数返回时，rp->handler 被调用。如果调用成功，它返回 0，否则返回负错误码。

kretprobe 处理函数的原型列出如下：

```
int kretprobe_handler(struct kretprobe_instance *ri, struct pt_regs *regs);
```

参数 regs 指向保存的寄存器，ri 指向结构 kretprobe_instance 对象，该对象根据注册时用户指定的 maxactive 值分配，含有被探测函数实例的返回地址。

-

函数 unregister_*probe

每一个注册函数都有相应的卸载函数，下面三个注销函数与分与三个类型的注册函数相对应：

```
void unregister_kprobe(struct kprobe *kp);

void unregister_jprobe(struct jprobe *jp);

void unregister_kretprobe(struct kretprobe *rp);
```

调试跟踪系统调用 ptrace

通过系统调用 ptrace（），一个进程可以动态地读/写另一个进程的内存和寄存器，包括其指令空间、数据空间、堆栈以及所有的寄存器。GNU 的调试工具 gdb 就是一个典型应用的例子。

调试寄存器

当程序访问到断点的线性地址且满足特定条件时，就跳转到异常处理程序。80386 可支持同时设置四个断点条件，编程人员可在程序中的四个位置设置条件，使其转向异常处理程序。这四个断点的每一个断点，都可以是如下三种不同类型的任何一种：

只在指令地址与断点地址一致时，断点有效。
数据写入地址与断点地址一致时，断点有效。
数据读出地址或数据写入地址与断点地址一致时，断点有效。

在 80386 中有编号为 DR0 至 DR7 的 8 个寄存器，只能在 0 级特权级进行访问。

其中四个用于断点，两个用于控制，另两个保留。这 8 个寄存器还可设置 DR6 及 DR7 中的 BD 位和 GD 位来完成更高一级的保护，即在 0 级也不能操作。调试寄存器列出如下图：

调试寄存器

这些寄存器的功能如下：

DR0—DR3 寄存器 DR0—DR3 对应与四个断点条件相联系的线性地址（断点条件则在 DR7 中）。

DR4—DR5 保留。

DR6 是调试状态寄存器。当一个调试异常产生时，设置 DR6 的相应位来指示调试异常发生的原因。

DR7 是调试控制寄存器。分别对应四个断点寄存器的控制位，对断点的启用及断点类型的选择进行控制。所有断点寄存器的保护也在此寄存器中规定。

DR6 各位的功能

B0—B3 当断点线性地址寄存器规定的条件被检测到时,将对应的 B0—B3 位置 1。

BD 如下一条指令要对八个调试寄存器之一进行读或写时，则在指令的边界 BD 位置 1。在一条指令内，每当即将读写调试寄存器时，也 BD 位置 1。BD 位置 1 与 DR7 中 GD 位启用与否无关。

BS 如果单步异常发生时，BS 位被置 1。单步条件由 EFLAGS 寄存器中的 TF 位启用。如果程序由于单步条件进入调试处理程序，则 BS 位被置 1。与 DR6 中的其它位不同的是，BS 位只在单步陷阱实际发生时才置位，而不是检测到单步条件就置位。

BT BT 位对任务切换导致 TSS 中的调试陷阱位被启用而造成的调试异常，指示其原因。对这一条件，在 DR7 中没有启用位。

DR6 中的各个标志位，在处理机的各种清除操作中不受影响，因此，调试异常处理程序在运行以前，应清除 DR6，以避免下一次检测到异常条件时，受到原来的 DR6 中状态位的影响。

DR7 各位的功能

LEN LEN 为一个两位的字段，用以指示断点的长度。每一断点寄存器对应一个这样的字段，所以共有四个这样的字段分别对应四个断点寄存器。LEN 的四种译码状态对应的断点长度如下

LEN	说明
0 0	断点为一字节
0 1	断点为两字节
1 0	保留
1 1	断点为四字节

这里，如果断点是多字节长度，则必须按对应多字节边界进行对齐。如果对应断点是一个指令地址，则 LEN 必须为 00

RWE RWE 也是两位的字段，用以指示引起断点异常的访问类型。共有四个 RWE 字段分别对应四个断点寄存器，RWE 的四种译码状态对应的访问类型如下

RWE	说明
0 0	指令
0 1	数据写
1 0	保留
1 1	数据读和写

GE/LE GE/LE 为分别指示准确的全局/局部数据断点。如果 GE 或 LE 被置位，则处理器将放慢执行速度，使得数据断点准确地把产生断点的指令报告出来。如果这些位没有置位，则处理器在执行数据写的指令接近执行结束稍前一点报告

断点条件。

L0—L3/G0—G3 L0—L3 及 G0—G3 位分别为四个断点寄存器的局部及全局启用信号。如果有任一个局部或全局启用位被置位，则由对应断点寄存器 DRi 规定的断点被启用。

GD GD 位启用调试寄存器保护条件。注意，处理程序在每次转入调试异常处理程序入口处清除 GD 位，从而使处理程序可以不受限制地访问调试寄存器。

前述的各个 L 位（即 LE，L0—L3）是有关任务的局部位，使调试条件只在特定的任务启用。而各个 G 位（即 GD，G0—G3）是全局的，调试条件对系统中的所有任务皆有效。在每次任务切换时，处理器都要清除 L 位。

TSS 中的调度陷阱

每当通过 TSS 发生任务切换时，TSS 中的 T 位使调试处理程序被调用，这就为调试程序管理某些任务的活动提供了一种方便的方法。DR6 中的 BT 位指示对该位的检测，DR7 中对该位没有特别的启用位。

> 如果调试处理程序是通过任务门使用的，则不能设置对应 TSS 的调试陷阱位。否则，将发生调试处理程序的无限循环

INT3

断点指令的第一个字节用 INT3 指令替代，程序执行到断点处遇到断点指令，并进入 INT3 处理程序。一般情况下，INT3 指令在代码中执行断点，保留断点寄存器用于数据断点。代码中可以插入任意数量的 INT3 指令，而断点寄存器提供最多四个断点。

在某些情况下没法使用 INT3 而只能使用断点寄存器。如：

.由 ROM 提供的代码中无法能插入 INT3 指令。

.由于使用了 INT3，原来的程序代码被修改，使执行此代码的其它任务也被中断。

.INT3 不能执行数据断点。

程序的单步执行

将程序的一步一步地执行，对程序调试者来说，可方便地观察和分析操作数

据、操作指令及操作结果，发现程序是在哪一步发生错误。80386 的单步功能通过陷阱来实现。单步陷阱在 EFLAGS 寄存器中的 TF 位置位时启用。在一条指令开始执行时，如果有 TF=1，则在指令执行的末尾产生调试异常，并进入调试处理程序。在这里，"指令开始执行时，TF=1"这一条件是重要的。有此条件的限制，使 TF 位置位 1 的指令不会产生单步陷阱。每次产生单步陷阱之后，在进入调试处理程序之前要将 TF 位清除。此外，在处理中断或异常时，也清除 TF 位。

> 如果外部中断与单步中断同时发生，则单步中断被优先处理，并清除 TF。在调试处理程序第一条指令执行之前，如仍有悬挂的中断请求，则响应并处理中断。因此，中断处理是在没有单步启用的情况下完成的。如果希望在中断处理程序中使用单步功能。则需先把中断处理程序的第一条指令设置为断点，当程序运行到断点处停下来之后，再启用单步功能。

ptrace（）的系统调用

ptrace 系统调用对应的处理函数为 sys_ptrace（）。其参数 pid 为操作对象的进程号，参数 request 是具体的操作，在 include/linux/ptrace.h 中有这些操作码的宏定义。sys_ptrace 函数分析如下（在 arch/i386/kernel/ptrace.c 中）：

```
asmlinkage int sys_ptrace(long request, long pid, long addr, long
data)
{
        struct task_struct *child;
        struct user * dummy = NULL;
        int i, ret;
        unsigned long __user *datap = (unsigned long __user *)data;

        lock_kernel();
        ret = -EPERM;
        if (request == PTRACE_TRACEME) {
                //是否已被 traced
                if (current->ptrace & PT_PTRACED)
                        goto out;
                ret = security_ptrace(current->parent, current);
                if (ret)
                        goto out;
                //设置进程标识中的 ptrace 位
                current->ptrace |= PT_PTRACED;
```

```
                ret = 0;
                goto out;
        }
        ret = -ESRCH;
        read_lock(&tasklist_lock);
        child = find_task_by_pid(pid); // 查找 task 结构
        if (child)
                get_task_struct(child);
        read_unlock(&tasklist_lock);
        if (!child)
                goto out;

        ret = -EPERM;
        if (pid == 1)           // init 进程不能调试
                goto out_tsk;

        if (request == PTRACE_ATTACH) {
                ret = ptrace_attach(child);
                goto out_tsk;
        }
        ret = ptrace_check_attach(child, request == PTRACE_KILL);
        if (ret < 0)
                goto out_tsk;

        switch (request) {
        /* when I and D space are separate, these will need to be
fixed. */
        case PTRACE_PEEKTEXT: // 从进程的 text 段中读取一个字，在地址
addr
        case PTRACE_PEEKDATA: {// 从进程的 data 段中读取一个字
                unsigned long tmp;
                int copied;

                copied = access_process_vm(child, addr, &tmp,
sizeof(tmp), 0);
                ret = -EIO;
                if (copied != sizeof(tmp))
                        break;
                ret = put_user(tmp, datap);
                break;
        }
      //在 USER 区域 addr 位置读取一个字
        case PTRACE_PEEKUSR: {
                unsigned long tmp;
```

```
            ……
              if(addr < FRAME_SIZE*sizeof(long))
                      tmp = getreg(child, addr);
              …
              ret = put_user(tmp, datap);
              break;
       }
       /* when I and D space are separate, this will have to be
fixed. */
       case PTRACE_POKETEXT: //向进程的 text 段 addr 地址写一个字
       case PTRACE_POKEDATA: //向进程的 data 段 addr 地址写一个字
              ret = 0;
              if (access_process_vm(child, addr, &data,
sizeof(data), 1)
                 == sizeof(data))
                      break;
              ret = -EIO;
              break;

       case PTRACE_POKEUSR: //向 USER 区域 addr 位置写一个字
              ret = -EIO;
              …
              if (addr < FRAME_SIZE*sizeof(long)) {
                      ret = putreg(child, addr, data);
                      break;
              }
              ……
              break;

     case PTRACE_SYSCALL: ////继续执行直到下一个系统调用返回
       case PTRACE_CONT: { //在信号后重新开始，继续运行一个进程
              long tmp;

              ret = -EIO;
              if ((unsigned long) data > _NSIG)
                      break;
              if (request == PTRACE_SYSCALL) {
                      set_tsk_thread_flag(child, TIF_SYSCALL_TRACE);
              }
              else {
                      clear_tsk_thread_flag(child,
TIF_SYSCALL_TRACE);
              }
              clear_tsk_thread_flag(child, TIF_SINGLESTEP);
```

```
                child->exit_code = data;
                //确信单步执行位没被设置
                tmp = get_stack_long(child, EFL_OFFSET) & ~TRAP_FLAG;
                put_stack_long(child, EFL_OFFSET,tmp);
                wake_up_process(child);
                ret = 0;
                break;
        }
        //迫使子进程退出，发出 sigkill 信号
        case PTRACE_KILL: {//杀死一个进程
                long tmp;

                ret = 0;
                if (child->state == TASK_ZOMBIE)      /* already dead
*/
                        break;
                child->exit_code = SIGKILL;
                clear_tsk_thread_flag(child, TIF_SINGLESTEP);
                //确信单步执行位没被设置
                tmp = get_stack_long(child, EFL_OFFSET) & ~TRAP_FLAG;
                put_stack_long(child, EFL_OFFSET, tmp);
                wake_up_process(child);
                break;
        }
      case PTRACE_SINGLESTEP: {  //单步运行一个进程，设置陷阱标识
                ……
                tmp = get_stack_long(child, EFL_OFFSET) | TRAP_FLAG;
                put_stack_long(child, EFL_OFFSET, tmp);
                ……
                break;
        }
        case PTRACE_DETACH: //结束对 PTRACE_ATTACH 设置的进程的调试
                ret = ptrace_detach(child, data);
                break;

        case PTRACE_GETREGS: { //从子进程中得到所在寄存器的值
                if (!access_ok(VERIFY_WRITE, datap,
FRAME_SIZE*sizeof(long))) {
                        ret = -EIO;
                        break;
                }
                for ( i = 0; i < FRAME_SIZE*sizeof(long); i +=
sizeof(long) ) {
                        __put_user(getreg(child, i), datap);
```

```
                        datap++;
                }
                ret = 0;
                break;
        }
        case PTRACE_SETREGS: { //设置所有子进程寄存器的值
                unsigned long tmp;
                if (!access_ok(VERIFY_READ, datap,
FRAME_SIZE*sizeof(long))) {
                        ret = -EIO;
                        break;
                }
                for ( i = 0; i < FRAME_SIZE*sizeof(long); i +=
sizeof(long) ) {
                        __get_user(tmp, datap);
                        putreg(child, i, tmp);
                        datap++;
                }
                ret = 0;
                break;
        }
        case PTRACE_GETFPREGS: { //得到子进程 FPU 状态
                ……
                get_fpregs((struct user_i387_struct __user *)data,
child);
                break;
        }

        case PTRACE_SETFPREGS: { //设置子进程 FPU 状态
                ……
                set_fpregs(child, (struct user_i387_struct __user
*)data);
                ret = 0;
                break;
        }

        case PTRACE_GETFPXREGS: { //得到子进程扩展 FPU 状态
                ……
                ret = get_fpxregs((struct user_fxsr_struct __user
*)data, child);
                break;
        }

        case PTRACE_SETFPXREGS: { //设置子进程扩展 FPU 状态
```

```
                ……
                ret = set_fpxregs(child, (struct user_fxsr_struct
__user *)data);
                break;
        }

        case PTRACE_GET_THREAD_AREA:
                ret = ptrace_get_thread_area(child, addr,
                                             (struct user_desc __user *)
data);
                break;

        case PTRACE_SET_THREAD_AREA:
                ret = ptrace_set_thread_area(child, addr,
                                             (struct user_desc __user *)
data);
                break;

        default:
                ret = ptrace_request(child, request, addr, data);
                break;
        }
out_tsk:
        put_task_struct(child);
out:
        unlock_kernel();
        return ret;
}
```

---

sys_ptrace 函数调用了一些具体操作函数，下面就这些具体操作函数进行分析。

函数 get_stack_long()和 put_stack_long()为对子进程核心堆栈的操作。这两个函数列出如下：

```
static inline int get_stack_long(struct task_struct *task, int
offset)
{
        unsigned char *stack;
        stack = (unsigned char *)task->tss.esp0;      //  获得 ESP0 寄
存器值
```

```
        stack += offset;                                        //  加偏
移量
        return (*((int *)stack));
}

static inline int put_stack_long(struct task_struct *task, int
offset,
        unsigned long data)
{
        unsigned char * stack;
        stack = (unsigned char *) task->tss.esp0;     //  获得 ESP0 寄
存器值
        stack += offset;
        *(unsigned long *) stack = data;
        return 0;
}
```

---

getreg() putreg()两个函数，完成了对被调试子进程的寄存器读写功能。函数中参数 regno，表示寄存器的序号。这两个函数列出如下：

```
static int putreg(struct task_struct *child,
        unsigned long regno, unsigned long value)
{
        switch (regno >> 2) {
                case TEST_EAX:          //不能读写 EAX
                        return -EIO;
                case FS:
                        if (value && (value & 3) != 3)
                                return -EIO;            //  优先级不为
3，出错
                        child->tss.fs = value; //通过 TSS 写寄存器 fs
                        return 0;
                case GS:
                        if (value && (value & 3) != 3)
                                return -EIO;
                        child->tss.gs = value; //  通过 TSS 写寄存器 gs
                        return 0;
                case DS:
                case ES:
                        if (value && (value & 3) != 3)
                                return -EIO;
```

```
                        value &= 0xffff;                // ds es 为 16 位
                        break;
                case SS:
                case CS:
                        if ((value & 3) != 3)
                                return -EIO;
                        value &= 0xffff;
                        break;
                case EFL:
                        value &= FLAG_MASK;     /*  EFLAG 访问权限设定
*/
                        value |= get_stack_long(child, EFL_OFFSET) &
~FLAG_MASK;
        }
        if (regno > GS*4)
                regno -= 2*4;                   /*  修正偏移量  */
        put_stack_long(child, regno - sizeof(struct pt_regs), value);
        return 0;
}

static unsigned long getreg(struct task_struct *child,
        unsigned long regno)
{
        unsigned long retval = ~0UL;
        switch (regno >> 2) {
                case FS:
                        retval = child->tss.fs;         // 通过 TSS 读寄
存器 gs
                        break;
                case GS:
                        retval = child->tss.gs;         //  通过 TSS 读
寄存器 fs
                        break;
                case DS:
                case ES:
                case SS:
                case CS:
                        retval = 0xffff;
                default:
                        if (regno > GS*4)               // 修正偏移量
                                regno -= 2*4;
                        regno = regno - sizeof(struct pt_regs);
                        retval &= get_stack_long(child, regno);
        }
```

```
        return retval;
}
```

函数 write_long()和 read_long()访问进程空间的内存，它是通过调用 Linux 的分页管理机制完成的。它从进程的 task 结构中读出进程内存的描述 mm 结构，并依次按页目录、中间页目录、页表的顺序查找到物理页，再进行读写操作。函数 put_long()和 get_long()对一个页内数据进行读写操作。函数 write_long()和 read_long()分别调用 put_long()和 get_long()函数完成跨页之间的读写操作。

这里只列出了 get_long 函数，其它函数有类似的操作。

```
static unsigned long get_long(struct task_struct * tsk,
        struct vm_area_struct * vma, unsigned long addr)
{
        pgd_t * pgdir;
        pmd_t * pgmiddle;
        pte_t * pgtable;
        unsigned long page;

repeat:
        pgdir = pgd_offset(vma->vm_mm, addr);// 查找页目录
        if (pgd_none(*pgdir)) {
                handle_mm_fault(tsk, vma, addr, 0);//缺页处理
                goto repeat;
        }
        …
        pgmiddle = pmd_offset(pgdir, addr);    // 查找中间页目录
        if (pmd_none(*pgmiddle)) {
                handle_mm_fault(tsk, vma, addr, 0);    // 缺页处理
                goto repeat;
        }
        …
        pgtable = pte_offset(pgmiddle, addr);          //  查找页表
        if (!pte_present(*pgtable)) {
                handle_mm_fault(tsk, vma, addr, 0);    //  缺页处理
                goto repeat;
        }
        page = pte_page(*pgtable);
        if (MAP_NR(page) >= max_mapnr)
                return 0;                                       // 越界
```

```
出错
        page += addr & ~PAGE_MASK;
        return *(unsigned long *) page;
}
```

系统调用跟踪

系统调用跟踪是被调试进程在进入系统调用或完成系统调用时中止进程，设置断点。调试器通过调用 ptrace(PTRACE_SYSCALL)使进程继续运行，直到系统调用开始或结束。在 ptrace(PTRACE_SYSCALL)处理中，设置了进程标志 PF_TRACESYS。在系统调用时如果判断进程标志设置了 PF_TRACESYS 则调用函数 do_syscall_trace。代码如下：

syscall_trace 函数在/linux/arch/i386/ptrace.c 中定义。

syscall_trace 函数完成了系统调用中断的功能，其流程如下：

```
/* notification of system call entry/exit
 * - triggered by current->work.syscall_trace
 */
__attribute__((regparm(3)))
void do_syscall_trace(struct pt_regs *regs, int entryexit)
{
        if (unlikely(current->audit_context)) {
                if (!entryexit)
                        audit_syscall_entry(current, regs->orig_eax,
                                            regs->ebx, regs->ecx,
                                            regs->edx, regs->esi);
                else
                        audit_syscall_exit(current, regs->eax);
        }

        if (!test_thread_flag(TIF_SYSCALL_TRACE) &&
            !test_thread_flag(TIF_SINGLESTEP))
                return;
        if (!(current->ptrace & PT_PTRACED))
                return;
        /* the 0x80 provides a way for the tracing parent to
distinguish
           between a syscall stop and SIGTRAP delivery */
        ptrace_notify(SIGTRAP | ((current->ptrace & PT_TRACESYSGOOD)
&&
                                 !test_thread_flag(TIF_SINGLESTEP) ?
```

```
0x80 : 0));

        /*
         * this isn't the same as continuing with a signal, but it
will do
         * for normal use.  strace only continues with a signal if
the
         * stopping signal is not SIGTRAP.  -brl
         */
        if (current->exit_code) {
                send_sig(current->exit_code, current, 1);
                current->exit_code = 0;
        }
}
```

调试陷阱处理

λ 调试陷阱处理（异常 1 处理），完成单步执行和断点中断处理。在程序设置断点（指令和数据断点）、单步执行、TSS 调试陷阱情况下，在 i386 中引起调试异常，调试异常的编号为 1。当程序调试时用户进程一些寄存器没有初始化或设置可能引起非正常的调试异常。进程在调试状态下。对于一些非正常的调试异常做一些清理工作。调试异常的调试工作由 \linux\arch\i386\kernel\traps.c 中的函数 do_debug 完成。do_debug 函数处理中对于正常的调试异常产生 SIGTRAP 信号。

do_debug 函数列出如下：

```
asmlinkage void do_debug(struct pt_regs * regs, long error_code)
{
        unsigned int condition;
        struct task_struct *tsk = current;
        siginfo_t info;

        __asm__ __volatile__("movl %%db6,%0" : "=r" (condition)); //
读取 DR6 值

        if (notify_die(DIE_DEBUG, "debug", regs, condition,
error_code,
                                        SIGTRAP) == NOTIFY_STOP)
                return;
```

```
        //在DR6存储后，允许中断是安全的
        if (regs->eflags & X86_EFLAGS_IF)
                local_irq_enable();

        /* Mask out spurious debug traps due to lazy DR7 setting */
        if (condition & (DR_TRAP0|DR_TRAP1|DR_TRAP2|DR_TRAP3)) {
                if (!tsk->thread.debugreg[7]) //是否TSS中DR7为0
                        goto clear_dr7;
        }

        if (regs->eflags & VM_MASK) //判断是否是虚拟8086方式
                goto debug_vm86;

        //保存debug状态寄存器在ptrace能得到它的地方
        tsk->thread.debugreg[6] = condition;

        /* Mask out spurious TF errors due to lazy TF clearing */
        if (condition & DR_STEP) {//是否为单步异常
                /*
                 * The TF error should be masked out only if the
current
                 * process is not traced and if the TRAP flag has
been set
                 * previously by a tracing process (condition
detected by
                 * the PT_DTRACE flag); remember that the i386 TRAP
flag
                 * can be modified by the process itself in user
mode,
                 * allowing programs to debug themselves without the
ptrace()
                 * interface.
                 */
                if ((regs->xcs & 3) == 0) //是否为核心引起异常
                        goto clear_TF_reenable;
                if ((tsk->ptrace & (PT_DTRACE|PT_PTRACED)) ==
PT_DTRACE)
                        goto clear_TF;
        }

        /* Ok, finally something we can handle */
        tsk->thread.trap_no = 1;
        tsk->thread.error_code = error_code;
        info.si_signo = SIGTRAP;
```

```
        info.si_errno = 0;
        info.si_code = TRAP_BRKPT;

        /* If this is a kernel mode trap, save the user PC on entry
to
         * the kernel, that's what the debugger can make sense of.
         */
        info.si_addr = ((regs->xcs & 3) == 0) ? (void __user *)tsk-
>thread.eip
                                              : (void __user *)regs-
>eip;
        force_sig_info(SIGTRAP, &info, tsk); // 产生 SIGTRAP 信号

        /* Disable additional traps. They'll be re-enabled when
         * the signal is delivered.
         */
clear_dr7:
        __asm__("movl %0,%%db7"
                : /* no output */
                : "r" (0));
        return;

debug_vm86: // 转向虚拟 8086 陷阱处理
        handle_vm86_trap((struct kernel_vm86_regs *) regs,
error_code, 1);
        return;

clear_TF_reenable:
        set_tsk_thread_flag(tsk, TIF_SINGLESTEP);
clear_TF: //清除 TF 标志
        regs->eflags &= ~TF_MASK;
        return;
}
```

调试器运行方法

调试器对一个程序进行调试，需要装入代码，创建相应的进程，并使其进入调试状态。在 GDB 调试器中命令 run 是通过下面方法运行调试运行程序的：

```
int pid;
  pid = fork ();//创建子进程
```

```
  if (pid < 0)
    perror_with_name ("fork");
  if (pid == 0)
{//子进程
  //进程标志 PF_PTRACED 置位，子进程进入调试状态
      ptrace (PTRACE_TRACEME, 0, 0, 0);
      //系统装入可执行文件代码，装入完毕后，则会向自身发送 SIGTRAP 信
号
      //      程序名，  参数
      execv (program, allargs);
      fprintf (stderr, "Cannot exec %s: %s.\n", program,
               errno < sys_nerr ? sys_errlist[errno] : "unknown
error");
      fflush (stderr);
      _exit (0177);
}
//在处理 SIGTRAP 信号时，中止子进程执行，并通知调试器（父进程）让其运
行,
//父进程则从 wait 调用中返回
//至此，要调试的程序作为子进程已经调入，并中止在程序的第一条指令上
       wait(pid);
   //使子进程继续执行
ptrace (PTRACE_CONT, pid, 0, 0);
//等待子进程中断或退出
wait(pid);
```

- 调试已经运行的进程

调试器还能对已经运行的进程进行调试。通过调用 ptrace 的 PTRACE_ATTACH 功能可以实现。具体做法如下：

```
ptrace(PTRACE_ATTACH, pid, 0,0)

wait(pid);
```

首先调用 ptrace(PTRACE_ATTACH, pid, 0,0)。ptrace 为完成对这个进程的调试设置，首先设置进程标志置 PF_PTRACED。再将它设置为调试器的子进程，最后向它发信号 SIGSTOP 中止它的运行，使它进入调试状态。

调试器在调用 ptrace 后需调用 wait，等待要调试的进程进入 STOP 状态。

在 GDB 调试器中命令 attach 则是通过上述方法调试现有的进程的。

- 退出程序调试

对于使用 ptrace 的 PTRACE_ATTACH 功能调试的程序，希望放弃调试，使其继续执行时可以使用 ptrace 的 PTRACE_DETACH 功能。

调用了 ptrace(PTRACE_DETACH，pid, 0,0)，终止调试一个子进程。此处理与 PTRACE_ATTACH 处理相反。在此做了一些清理操作：清除 PF_TRACESYS 和 PF_PTRACED 进程标志，清除 TF 标志，父进程指针还原。最后唤醒此进程，让其继续执行

在 GDB 调试器中命令 detach 则是通过上述方法调试现有的进程的。

- 终止程序调试

对被调试的进程，不想再调试时，可以调用 ptrace 的 PTRACE_KILL 功能杀死被调试的进程。

调用了 ptrace(PTRACE_KILL，pid, 0,0)，把子进程继续的信号设置为 SIGKILL，然后唤醒子进程，由于子进程是在 do_signal 处理中进入 stop 的，所以它将继续处理 SIGKILL 部分的代码，从而使子进程终止

在 GDB 调试器中命令 kill 则是通过上述方法调试现有的进程的。

- 设置断点

80386 提供了两种方式在调试器中设置断点，INT3 和利用调试寄存器。如果使用 INT3 方式设置断点，则调试器通过 ptrace 的 PTRACE_POKETEXT 功能在断点处插入 INT3 单字节指令。当进程运行到断点时（INT3 处），则系统进入异常 3 的处理。

若使用调试寄存器，则调试器通过调用 ptrace(PTRACE_POKEUSR，pid，0，data)在 DR0-DR3 寄存器设置与四个断点条件的每一个相联系的线性地址在 DR7 中设置断点条件。被跟踪进程运行到断点处时，CPU 产生异常 1，从而转至函数 do_debug 处理。由于子进程在调试状态下属于正常调试异常，所以 do_debug 函数处理中产生 SIGTRAP 信号，为处理这个信号，进入 do_signal，使被调试进程停止，并通知调试器（父进程），此时得到子进程终止原因为 SIGTRAP。

在有些情况之下，要求调试器调试某进程时，当进程收到某一信号的时候中断进程运行。如：被调试进程在某处运算错误，进程会接收到 SIGFPE 信号，在正

常运行状况下，会 Coredump，而调试的情况下则希望在产生错误代码处停止运行，可以让用户调试错误原因。

对于已经被调试的进程（PF_PTRACED 标志置位），当受到任何信号（SIGKILL 除外）会中止其运行，并通知调试器（父进程）。

- 单步执行

单步执行也是一种使进程中止的情况。当用户调用 ptrace 的 PTRACE_SINGLESTEP 功能时，ptrace 处理中，将用户态标志寄存器 EFLAG 中 TF 标志为置位，并让进程继续运行。当进程回到用户态运行了一条指令后，CPU 产生异常 1，从而转至函数 do_debug 处理。由于子进程在调试状态下属于正常调试异常，所以 do_debug 函数处理中产生 SIGTRAP 信号，为处理这个信号，进入 do_signal，使被调试进程停止，并通知调试器（父进程），此时得到子进程终止原因为 SIGTRAP。

- 系统调用进程跟踪

对程序的调试，有时希望对系统调用进程跟踪。当程序进行系统调用时中断其运行。ptrace 提供 PTRACE_SYSCALL 功能完成此功能。在 ptrace 调用中设置了进程标志 PF_TRACESYS，表示进程对系统调用进行跟踪，并继续执行进程（具体分析见 ptrace 函数分析）。直到进程调用系统调用时，则中止其运行，并通知调试器（父进程