Linux Kernel GDB tracepoint module (KGTP)

Update in 2013-08-29

目录

关于本文

https://code.google.com/p/kgtp/wiki/HOWTOCN 是 HTML 格式的本文最后版本。

https://raw.github.com/teawater/kgtp/master/kgtpcn.pdf 是 PDF 格式的本文最后版本。

https://raw.github.com/teawater/kgtp/release/kgtpcn.pdf 是 PDF 格式的本文最后发布版本。

什么是 KGTP

KGTP 是一个 灵活，轻量级，实时 **Linux** (包括 **Android**) 调试器 和 跟踪器 。

使用 KGTP 不需要 在 Linux 内核上打 PATCH 或者重新编译，只要编译 KGTP 模块并 insmod 就可以。

其让 Linux 内核提供一个远程 GDB 调试接口，于是在本地或者远程的主机上的 GDB 可以在不需要停止内核的情况下用 GDB tracepoint 和其他一些功能 调试 和 跟踪 Linux 内核和应用程序。

即使板子上没有 GDB 而且其没有可用的远程接口，KGTP 也可以用离线调试的功能调试内核（见/sys/kernel/debug/gtpframe 和离线调试）。

KGTP 支持 **X86-32** ， **X86-64** ， **MIPS** 和 **ARM** 。

KGTP 在 Linux 内核 **2.6.18** 到 **upstream** 上都被测试过。

KGTP 新用户可以去看一下 Quickstart 。

请到 UPDATE 去看 KGTP 的更新信息。

需要帮助或者汇报问题

请把问题发到 https://github.com/teawater/kgtp/issues。
或者写信到 mailto:teawater@gmail.com?Subject=汇报一个 KGTP 的问题。
KGTP 小组将尽全力帮助你。

请到 https://code.google.com/p/kgtp/issues/list 访问旧问题列表。

GDB 调试普通程序和 KGTP 的区别表

这个表是给在使用过 GDB 调试程序的人准备的，他可以帮助你理解和记住 KGTP 的功能。

功能	**GDB** 调试普通程序	**GDB** 控制 **KGTP** 调试 **Linux** 内核
准备工作	系统里安装了 GDB。程序用 "-g"选项编译。	因为使用了一些 GDB 中的新功能，所以 KGTP 需要和 GDB 7.6 或者更新的版本。如果你的系统不提供这么新版本的 GDB，你可以到 http://code.google.com/p/gdbt/取得新版本 GDB。同时你可以在这里取得一步一步编译新版本 GDB 的介绍。 你还需要做一些 Linux 内核和 KGTP 的准备工作，请到 使用 KGTP 前的准备工作 取得如果做的介绍。
Attach	使用命令"gdb -p pid"或者 GDB 命令"attach pid"可以 attach 系统中的某个程序.	需要先 insmod gtp.ko，请看 [https://code.google.com/p/kgtp/wiki/HOWTOCN#执行]。 需要先 insmod gtp.ko，请看 https://code.google.com/p/kgtp/wiki/HOWTOCN#执行。 然后让 GDB 连接 KGTP,请看让 GDB 连接到 KGTP。 请 注意 GDB 连接到 KGTP 以后，Linux 内核不会停止。
Breakpoints	GDB 命令"b place_will_stop"，让程序在执行这个命令后执行，则程序将停止在设置这个断点的地方。	KGTP 不支持断点但是支持 tracepoint。Tracepoints 可以被看作一种特殊的断点。其可以设置在 Linux kernel 中的一些地方然后定义一些命令到它的 action 中。当 tracepoint 开始的时候，他们将会在内核执行到这些地方的时候执行这些命令。当 tracepoint 停止的时候，你可以像断点停止程序后你做的那样用 GDB 命令分析 tracepoint 得到的数据。 区别是断点会停止程序但是 KGTP 中的 tracepoint 不会。 请到 GDB tracepoint 看如何使用它。
读 Memory	GDB 停止程序后(也许不需要)，它可以用 GDB 命令"print"或者"x"等应用程序的内存。	你可以在 tracepoint 中设置特殊的 action 收集内存到 traceframe 中，在 tracepoint 停止后取得他们的值。collect expr1, expr2, ... 用 tfind 选择 trace 帧缓存里面的条目 或者你可以在内核或者应用程序执行的时候直接读他们的内存。在普通模式直接访问当前值

Step 和 continue	GDB 可以用命令"continue"继续程序的执行，用 CTRL-C 停止其。	KGTP 不会停止 Linux 内核，但是 tracepoint 可以开始和停止。启动和停止 tracepoint 或者用 while-stepping tracepoint 记录一定次数的 single-stepping 然后让 KGTP 切换到回放模式。这样其就支持执行和方向执行命令了。使用 while-stepping 让 Linux 内核做单步
Backtrace	GDB 可以用命令"backtrace"打印全部调用栈。	KGTP 也可以。如何 backtrace (stack dump)
Watchpoint	GDB 可以用 watchpoint 让程序在某些内存访问发生的时候停止。	KGTP 可以用 watch tracepoint 记录内存访问。如何用 watch tracepoint 控制硬件断点记录内存访问
调用函数	GDB 可以用命令"call function(xx,xx)"调用程序中的函数。	KGTP 可以用插件调用内核中的函数。如何增加用 C 写的插件

使用 KGTP 前的准备工作

Linux 内核

如果你的系统内核是自己编译的

要使用 KGTP，你需要打开下面这些内核选项：

General setup --->
[*] Kprobes

[*] Enable loadable module support --->

Kernel hacking --->
[*] Debug Filesystem
[*] Compile the kernel with debug info

如果你改了 Linux 内核 config 的任何项目，请重新编译你的内核。

如果是 **Android** 内核

默认的 Android Linux 内核 config 应该不支持 KGTP。要使用 KGTP，你需要打开下面这些内核选项：

[*] Enable loadable module support --->
General setup --->
[*] Prompt for development and/or incomplete code/drivers
[*] Kprobes
Kernel hacking --->
[*] Debug Filesystem
[*] Compile the kernel with debug info

如果你改了 Linux 内核 config 的任何项目，请重新编译你的内核。

如果你的系统内核是发行版自带的

你需要安装一些 Linux 内核软件包。

Ubuntu

安装 **Linux** 内核调试镜像的标准方法

1) 增加调试源到 Ubuntu 源列表。

- 在命令行按照下面的命令创建文件 /etc/apt/sources.list.d/ddebs.list：

  *echo "deb http://ddebs.ubuntu.com $(lsb_release -cs) main restricted universe multiverse" | *
  sudo tee -a /etc/apt/sources.list.d/ddebs.list
- 稳定版本（不能是 alpha 或者 betas）需要用命令行增加下面几行：

  echo "deb http://ddebs.ubuntu.com $(lsb_release -cs)-updates main restricted universe multiverse
  deb http://ddebs.ubuntu.com $(lsb_release -cs)-security main restricted universe multiverse
  *deb http://ddebs.ubuntu.com $(lsb_release -cs)-proposed main restricted universe multiverse" | *
  sudo tee -a /etc/apt/sources.list.d/ddebs.list
- 导入调试符号签名 key：

  sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 428D7C01
  运行：
  sudo apt-get update

2) 安装 Linux 内核调试镜像

sudo apt-get install linux-image-$(uname -r)-dbgsym

于是你可以在"/usr/lib/debug/boot/vmlinux-$(uname -r)"找到内核调试镜像。

请 注意 当内核更新的时候这一步 安装 **Linux** 内核调试镜像 需要再做一次。

安装 **Linux** 内核调试镜像的第二方法

如果用标准方法出现问题，请用下面这些命令安装 Linux 内核调试镜像。

wget http://ddebs.ubuntu.com/pool/main/l/linux/linux-image-$(uname -r)-dbgsym_$(dpkg -s linux-image-$(uname -r) | grep ^Version: | sed 's/Version: //')_$(uname -i | sed 's/x86_64/amd64/').ddeb
sudo dpkg -i linux-image-$(uname -r)-dbgsym_$(dpkg -s linux-image-$(uname -r) | grep ^Version: | sed 's/Version: //')_$(uname -i | sed 's/x86_64/amd64/').ddeb

请 注意 当内核更新的时候这个方法需要重新做一次。

安装内核头文件包

sudo apt-get install linux-headers-generic

安装内核源码

新方法

安装需要的软件包：

sudo apt-get install dpkg-dev

取得源码:

apt-get source linux-image-$(uname -r)

则在当前目录找到内核源码目录。

移动这个目录到"/build/buildd/"中。

老方法

- 安装源码包：

  sudo apt-get install linux-source

- 解压缩源码：

  sudo mkdir -p /build/buildd/
  sudo tar vxjf /usr/src/linux-source-$(uname -r | sed 's/-.//').tar.bz2 -C /build/buildd/*
  sudo rm -rf /build/buildd/linux-$(uname -r | sed 's/-.//')*
  sudo mv /build/buildd/linux-source-$(uname -r | sed 's/-.*//') /build/buildd/linux-$(uname -r | sed 's/-.//')*

请 注意 当内核更新的时候这一步 安装内核源码 需要再做一次。

Fedora

安装 **Linux** 内核调试镜像

使用下面的命令:

sudo debuginfo-install kernel

或者:

sudo yum --enablerepo=fedora-debuginfo install kernel-debuginfo

于是你可以在"/usr/lib/debug/lib/modules/$(uname -r)/vmlinux"找到内核调试镜像。

安装 **Linux** 内核开发包

sudo yum install kernel-devel-$(uname -r)

请 注意 在升级过内核包之后，你可能需要重新调用这个命令。

其他系统

需要安装 Linux 内核调试镜像和 Linux 内核源码包。

确定 Linux 内核调试镜像是正确的

因为 GDB 从 Linux 内核调试镜像里取得地址信息和调试信息，所以使用正确的 Linux 内核调试镜像是非常重要的。所以在使用 KGTP 前，请先做检查。

有 2 个方法进行检查，我建议 2 个方法都做一次来确保 Linux 内核调试镜像是正确的。

请 注意 如果你确定使用了正确的 Linux 内核调试镜像但是不能通过这个两个方法。请看 处理 Linux 内核调试镜像地址信息和 Linux 内核执行时不同的问题 。

当前 **Linux** 内核调试镜像在哪

在 UBUNTU 中，你可以在"/usr/lib/debug/boot/vmlinux-$(uname -r)"找到它。
在 Fedora 中，你可以在"/usr/lib/debug/lib/modules/$(uname -r)/vmlinux"找到它。
如果你自己编译的内核，你可以在内核编译目录找到 vmlinux 文件。

使用/proc/kallsyms

在运行着要 trace 的内核的系统上，用下面的命令取得 sys_read 和 sys_write 的地址：

```
sudo cat /proc/kallsyms | grep sys_read
ffffffff8117a520 T sys_read
sudo cat /proc/kallsyms | grep sys_write
ffffffff8117a5b0 T sys_write
```

于是我们就可以得到 sys_read 的地址是 0xffffffff8117a520，sys_write 的地址是 0xffffffff8117a5b0。

之后我们用 GDB 从 Linux 内核调试镜像中取得 sys_read 和 sys_write 的地址：

```
gdb ./vmlinux
(gdb) p sys_read
$1 = {long int (unsigned int, char *, size_t)} 0xffffffff8117a520
<sys_read>
(gdb) p sys_write
$2 = {long int (unsigned int, const char *, size_t)} 0xffffffff8117a5b0
<sys_write>
```

sys_read 和 sys_write 的地址一样，所以 Linux 内核调试镜像是正确的。

使用 **linux_banner**

```
sudo gdb ./vmlinux
(gdb) p linux_banner
$1 = "Linux version 3.4.0-rc4+ (teawater@teawater-Precision-
M4600) (gcc version 4.6.3 (GCC) ) #3 SMP Tue Apr 24 13:29:05 CST
2012\n"
```

linux_banner 是 Linux 内核调试镜像里的内核信息。

之后，根据 让 GDB 连接到 KGTP 里的方法连接到 KGTP 上并再次打印 linux_banner。

```
(gdb) target remote /sys/kernel/debug/gtp
Remote debugging using /sys/kernel/debug/gtp
0x0000000000000000 in irq_stack_union ()
(gdb) p linux_banner
$2 = "Linux version 3.4.0-rc4+ (teawater@teawater-Precision-
M4600) (gcc version 4.6.3 (GCC) ) #3 SMP Tue Apr 24 13:29:05 CST
2012\n"
```

这个 linux_banner 是 KGTP 正在 trace 的内核的内核信息，如果相同，则 Linux 内核调试镜像是正确的。

处理 **Linux** 内核调试镜像地址信息和 **Linux** 内核执行时不同的问题

在 X86_32 上，用 确定 Linux 内核调试镜像是正确的 介绍的方法发现 Linux 内核调试镜像地址信息和 Linux 内核执行时不同，而且确定使用的 Linux 内核调试镜像是正确的。

这个问题是因为：

Processor type and features --->
(0x1000000) Physical address where the kernel is loaded
(0x100000) Alignment value to which kernel should be aligned

这个两个参数的值不同。请 注意 "Physical address where the kernel is loaded" 有时不会在配置的时候显示，你可以通过搜索 "PHYSICAL_START" 取得它的值。

你可以通过修改 "Alignment value to which kernel should be aligned" 的值和 "Physical address where the kernel is loaded" 来处理这个问题。

这个问题不影响 X86_64。

取得 KGTP

通过 http 下载 KGTP

请到 https://github.com/teawater/kgtp 或者 UPDATE 去下载源码包。

通过 git 下载 KGTP

下面的命令将让你取得 KGTP 的最新版本：

```
git clone https://github.com/teawater/kgtp.git
```

下面的命令将让你取得 KGTP 最后的发布版本：

```
git clone https://github.com/teawater/kgtp.git
git checkout release -b release
```

镜像

https://code.csdn.net/teawater/kgtp

https://git.oschina.net/teawater/kgtp

https://www.gitshell.com/teawater/kgtp/

配置 KGTP

下面这部分是在 KGTP Makefile 里的配置。用这个配置，KGTP 将自动和当前系统的内核一起编译。

```
KERNELDIR := /lib/modules/`uname -r`/build
CROSS_COMPILE :=
```

KERELDIR 设置了你要一起编译的内核，默认情况下，KGTP 会和当前的内核一起编译。

请 注意 这个目录应该是内核编译目录或者 linux-headers 目录，而不是内核源码目录。内核编译目录只有在编译成功后才能使用。

CROSS_COMPILE 设置编译 KGTP 的编译器前缀名。留空则使用默认编译器。

ARCH 是体系结构。

或者你可以通过修改 KGTP 目录里的 Makefile 选择你要和哪个内核一起编译以及你用什么编译器编译 KGTP。

例如：

```
KERNELDIR := /home/teawater/kernel/bamd64
CROSS_COMPILE :=x86_64-glibc_std-
ARCH := x86_64
```

KERNELDIR 设置为 /home/teawater/kernel/bamd64。 Compiler 设置为 x86_64-glibc_std-gcc。

编译 KGTP

普通编译

```
cd kgtp/
make
```

在一些编译环境中(例如 Android)将出现一些编译应用程序 getmod 或者 getframe 的错误。请忽略这些错误并使用目录中的 gtp.ko。

如果你在 Fedora 上得到出错信息"/usr/bin/ld: cannot find -lc"，请用下面的命令处理其。

```
sudo yum install glibc-static
```

用一些特殊选项编译 KGTP

大部分时候，KGTP 可以自动选择正确的参数和和各种版本的 Linux 内核一起编译。
但是如果你想配置一些特殊选项，可以按照下面的介绍来做：

用这个选项，KGTP 将不自动选择任何编译选项。

make AUTO=0

用这个选项，KGTP 将使用简单 frame 替代 KGTP ring buffer。
简单 frame 不支持 gtpframe_pipe，它现在只用来调试 KGTP。

make AUTO=0 FRAME_SIMPLE=1

用这个选项，$clock 将返回 rdtsc 的值而不是 local_clock。

make AUTO=0 CLOCK_CYCLE=1

用这个选项，KGTP 可以用 procfs 替代 debugfs。

make AUTO=0 USE_PROC=1

这些选线可以一起使用，例如：

make AUTO=0 FRAME_SIMPLE=1 CLOCK_CYCLE=1

安装和卸载 KGTP

因为 KGTP 可以直接在编译目录里 insmod，所以不编译后不安装也可以直接使用（见 如何让 GDB 连接 KGTP）。但是如果需要也可以将其安装到系统中。 安装：

```
cd kgtp/
sudo make install
```

卸载：

```
cd kgtp/
sudo make uninstall
```

和 DKMS 一起使用 KGTP

如果你需要的话，你还可以让 DKMS 来使用 KGTP。

下面的命令将拷贝 KGTP 的文件到 DKMS 需要的目录中。

```
cd kgtp/
sudo make dkms
```

于是你可以用 DKMS 命令控制 KGTP。请到
http://linux.dell.com/dkms/manpage.html 去看如何使用 DKMS。

使用 KGTP Linux 内核 patch

大多数时候，你不需要 KGTP patch，因为 KGTP 以一个 LKM 的形式编译安装更为方便。但是为了帮助人们集成 KGTP 到他们自己的内核树，KGTP 也提供了 patch. 在 KGTP 目录中：

- **gtp_3.7_to_upstream.patch** 是给 Linux kernel 从 3.7 到 upstream 的 patch。
- **gtp_3.0_to_3.6.patch** 是给 Linux kernel 从 3.0 到 3.6 的 patch。
- **gtp_2.6.39.patch** 是给 Linux kernel 2.6.39 的 patch。
- **gtp_2.6.33_to_2.6.38.patch** 是给 Linux kernel 从 2.6.33 到 2.6.38 的 patch。
- **gtp_2.6.20_to_2.6.32.patch** 是给 Linux kernel 从 2.6.20 到 2.6.32 的 patch。
- **gtp_older_to_2.6.19.patch** 是给 Linux kernel 2.6.19 以及更早版本的 patch。

安装可以和 KGTP 一起使用的 GDB

早于 7.6 版本的 GDB 的 tracepoint 功能有一些 bug，而且还有一些功能做的不是很好。

所以如果你的 GDB 小于 7.6 请到 https://code.google.com/p/gdbt/ 去安装可以和 KGTP 一起使用的 GDB。这里提供 UBUBTU, CentOS, Fedora, Mandriva, RHEL, SLE, openSUSE 源。其他系统还可以下载静态编译版本。

如果你有 GDB 的问题，请根据这里的信息需要帮助或者汇报问题。

如何让 GDB 连接 KGTP

要使用 KGTP 的功能需要先让 GDB 连接到 KGTP 上。

普通 Linux

安装 KGTP 模块

如果你已经安装了 KGTP 在你的系统中，你可以：

```
sudo modprobe gtp
```

或者你可以直接使用 KGTP 目录里的文件：

```
cd kgtp/
sudo insmod gtp.ko
```

处理找不到"/sys/kernel/debug/gtp"的问题

如果你有这个问题，请先确定你的内核 config 打开了"Debug Filesystem"。如果你的系统内核是自己编译的

如果它以及被打开了，请用下面命令 mount sysfs。

sudo mount -t sysfs none /sys/

也许你可能会得到一些错误例如"sysfs is already mounted on /sys"，请忽略他们。

请用下面命令 mount debugfs。

mount -t debugfs none /sys/kernel/debug/

然后你就找到"/sys/kernel/debug/gtp"。

让 GDB 连接到 KGTP

请 注意 让 GDB 打开正确的 vmlinux 文件非常重要。请到 确定 Linux 内核调试镜像是正确的 去看下如何做。

GDB 在本地主机上

sudo gdb ./vmlinux
(gdb) target remote /sys/kernel/debug/gtp
Remote debugging using /sys/kernel/debug/gtp
0x0000000000000000 in ?? ()

然后你就可以用 GDB 命令调试和跟踪 Linux 内核了。

如果 GDB 在远程主机上

用 nc 把 KGTP 接口映射到端口 1024 上。

```
sudo su
nc -l 1234 </sys/kernel/debug/gtp >/sys/kernel/debug/gtp
#(nc -l -p 1234 </sys/kernel/debug/gtp >/sys/kernel/debug/gtp 给老版本的 nc)
```

之后，nc 会在那里等待连接。

让 GDB 连接 1234 端口。

```
gdb-release ./vmlinux
(gdb) target remote xxx.xxx.xxx.xxx:1234
```

然后你就可以用 GDB 命令调试和跟踪 Linux 内核了。

Android

这个视频介绍了使用 GDB 连接 Android 上 KGTP 的过程，可访问 http://www.tudou.com/programs/view/qCumSPhByFI/ 或者 http://youtu.be/9YMpAvsl37I 进行观看。

安装 KGTP 模块

第一步 确定 ADB 已经连接到 Android 上。

第二步 拷贝 KGTP 模块到 Android 上。

sudo adb push gtp.ko /

目录 "/" 可能是只读的。你可以选择其他目录或者用命令"sudo adb shell mount -o rw,remount /"把这个目录 remount 为可写。

第三步 安装 KGTP 模块。

adb shell insmod /gtp.ko

处理找不到"/sys/kernel/debug/gtp"的问题

如果你有这个问题，请先确定你的内核 config 打开了"Debug Filesystem"。如果你的系统内核是自己编译的

如果它以及被打开了，请用下面命令 mount sysfs。

sudo adb shell mount -t sysfs none /sys/

也许你可能会得到一些错误例如"Device or resource busy"，请忽略他们。

请用下面命令 mount debugfs。

sudo adb shell mount -t debugfs none /sys/kernel/debug/

然后你就找到"/sys/kernel/debug/gtp"。

GDB 连接 KGTP

用 nc 将 KGTP 接口映射到 1024 端口上。

```
adb forward tcp:1234 tcp:1234
adb shell "nc -l -p 1234 </sys/kernel/debug/gtp
>/sys/kernel/debug/gtp"
#(adb shell "nc -l 1234 </sys/kernel/debug/gtp
>/sys/kernel/debug/gtp" 给新版本的 nc)
```

之后，nc 会在那里等待连接。

让 GDB 连接 1234 端口。

```
gdb-release ./vmlinux
(gdb) target remote :1234
```

然后你就可以用 GDB 命令调试和跟踪 Linux 内核了。

增加模块的符号信息到 GDB

有时你需要添加一个 Linux 内核模块的符号信息到 GDB 好调试其。

手动增加符号信息不太容易，所以 KGTP 包里包含了 GDB Python 脚本"getmod.py"和程序"getmod"可以帮到你。

如何使用 getmod

"getmod" 是用 C 写的所以你可以把它用在任何地方即使是一个嵌入式环境。

例如：

```
#下面的命令将把 Linux 内核模块信息以 GDB 命令的格式保存到文
件/tmp/mi。
sudo getmod >/tmp/mi
#在 GDB 那边：
(gdb) source /tmp/mi
add symbol table from file "/lib/modules/2.6.39-
rc5+/kernel/fs/nls/nls_iso8859-1.ko" at
        .text_addr = 0xf80de000
        .note.gnu.build-id_addr = 0xf80de088
        .exit.text_addr = 0xf80de074
        .init.text_addr = 0xf8118000
        .rodata.str1.1_addr = 0xf80de0ac
        .rodata_addr = 0xf80de0c0
        __mcount_loc_addr = 0xf80de9c0
        .data_addr = 0xf80de9e0
        .gnu.linkonce.this_module_addr = 0xf80dea00
#这条 GDB 命令后，所有 Linux 内核模块信息都被装载进 GDB 了。
#After this GDB command, all the Linux Kernel module info is loaded
into GDB.
```

如果你使用远程调试或者离线调试，你可以需要修改基本目录。下面是一个例子：

```
#/home/teawater/kernel/b26 是 GDB 所在主机上内核模块所在的路径
sudo ./getmod -r /home/teawater/kernel/b26 >~/tmp/mi
```

如何使用 **getmod.py**

请 注意 https://code.google.com/p/gdbt/下载的静态编译 GDB 不能使用 getmod.py。

在使用 getmod.py 前连接到 KGTP。

```
(gdb) source ~/kgtp/getmod.py
```

于是这个脚本将自动装载 Linux 内核模块到 GDB 中。

如何使用 GDB 控制 KGTP 跟踪和调试 Linux 内核

在普通模式直接访问当前值

在 GDB 连到 KGTP 上以后，如果没有用 GDB 命令"tfind"选择一条 trace 帧缓存里面的条目，GDB 就处于 普通模式。于是你可以直接访问内存(Linux 内核或者用户程序)的值和 trace 状态变量的值。

如果你选择了一个 trace 帧条目，可以用 GDB 命令"tfind -1"返回到普通模式。请到在普通模式直接访问当前值取得 GDB 命令"tfind"的更多信息。

例如你可以用下面的命令访问"jiffies_64"：

```
(gdb) p jiffies_64
```

或者你可以用下面的命令访问"static LIST_HEAD(modules)"的第一条记录：

```
(gdb) p *((struct module *)((char *)modules->next - ((size_t)
&(((struct module *)0)->list))))
```

或者你可以访问"DEFINE_PER_CPU(struct device *, mce_device);"CPU0 的数据：

```
p *(struct device *)(__per_cpu_offset[0]+(uint64_t)(&mce_device))
```

如果想在用一个 GDB 命令显示多个变量，请使用下面的例子：

```
(gdb) printf "%4d %4d %4d %4d %4d %4d %18d %lu\n", this_rq-
>cpu, this_rq->nr_running, this_rq->nr_uninterruptible, nr_active,
calc_load_tasks->counter, this_rq->calc_load_active, delta, this_rq-
>calc_load_update
2 1 0 0 0 0 673538312 717077240
```

用户程序的内存

KGTP 可以不同停止应用层程序的情况下直接读取其内存，例如：

```
#连接 KGTP(这里和前面介绍的连接方法不同)
(gdb) target extended-remote /sys/kernel/debug/gtp
#增加一个新的 inferior 用来分析应用程序的信息。
(gdb) add-inferior
Added inferior 2
#切换到这个 inferior
(gdb) inferior 2
[Switching to inferior 2 [<null>] (<noexec>)]
#转载这个程序的符号文件
(gdb) file ~/kernel/svn/bak/a.out
Reading symbols from /home/teawater/kernel/svn/bak/a.out...done.
#attach 到这个进程上(这不会停止这个程序)。
(gdb) attach 10039
Attaching to program: /home/teawater/kernel/svn/bak/a.out, Remote
target
Reading symbols from /lib/x86_64-linux-gnu/libc.so.6...(no debugging
symbols found)...done.
Loaded symbols for /lib/x86_64-linux-gnu/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging
symbols found)...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x0000000000000000 in ?? ()
#于是可以读这个程序的内存
(gdb) p c
$3 = -1222013392
(gdb) p c
$4 = -590910451
```

在这个例子中，我使用了一个多 inferior 命令。请到 http://sourceware.org/gdb/current/onlinedocs/gdb/Inferiors-and-Programs.html 去取得其更多相关信息。

trace 状态变量

你可以使用和访问内存一样的命令访问 TSV。 请到 如果使用 trace 状态变量 取得更多 TSV 的信息。

GDB tracepoint

tracepoint 就是 GDB 定义一些地址和一些动作。在 tracepoint 启动之后，当 Linux 内核执行到那些地址的时候，KGTP 将执行这些动作(它们中的有些会收集数据并存入 tracepoint 帧缓冲)并把它们发给调试目标(KGTP)。而后，Linux 内核将继续执行。

KGTP 提供了一些接口可以让 GDB 或者其他程序取出 tracepoint 帧缓冲的数据做分析。

关于这些接口，文档前面已经介绍了"/sys/kernel/debug/gtp"，将在后面介绍"/sys/kernel/debug/gtpframe" 和 "/sys/kernel/debug/gtpframe_pipe"。

GDB tracepoint 文档在 http://sourceware.org/gdb/current/onlinedocs/gdb/Tracepoints.html。

设置 tracepint

trace 命令非常类似 break 命令，它的参数可以是文件行，函数名或者一个地址。trace 将定义一个或者多个地址定义一个 tracepoint，KGTP 将在这个点做一些动作。

这是一些使用 trace 命令的例子：

(gdb) trace foo.c:121 // 一个文件和行号

(gdb) trace +2 // 2 行以后

(gdb) trace my_function // 函数的第一行

*(gdb) trace *my_function // 函数的第一个地址*

*(gdb) trace *0x2117c4 // 一个地址*

这个函数确实存在但是设置 **tracepoint** 到上面会失败如何处理

GCC 为了提高程序执行效率会 inline 一些 static 函数。因为目标文件没有 inline 函数的符号，所以你不能设置 tracepoint 在函数名上。

你可以用"trace 文件:行号"在其上设置断点。

如何设置条件 **tracepoint**

http://sourceware.org/gdb/current/onlinedocs/gdb/Tracepoint-Conditions.html

和 breakpoint 一样，我们可以设置 tracepoint 的触发条件。而且因为条件检查是在 KGTP 执行的，所以速度比 breakpoint 的条件检查快很多。

例如：

(gdb) trace handle_irq if (irq == 47)

tracepoint 1 的动作将只在 irq 是 47 的时候才被执行。

你还可以用 GDB 命令"condition"设置 tracepoint 的触发条件。GDB 命令"condition N COND"将设置 tracepoint N 只有条件 COND 为真的时候执行。

例如：

(gdb) trace handle_irq
(gdb) condition 1 (irq == 47)

GDB 命令"info tracepoint"将显示 tracepoint 的 ID。

$bpnum 的值是最后一个 GDB tracepoint 的 ID，所以你可以不取得 tracepoint 的 ID 就用 condition 来设置最后设置的 tracepoint 的条件，例如：

(gdb) trace handle_irq
(gdb) condition $bpnum (irq == 47)

如何处理错误 "Unsupported operator (null) (52) in expression."

如果你使用关于字符串的条件 tracepoint，你在调用"tstart"的时候可能得到这个出错信息。

你可以转化 char 为 int 来处理这个问题，例如：

```
(gdb) p/x 'A'
$4 = 0x41
(gdb) condition 1 (buf[0] == 0x41)
```

actions [num]

这个命令将设置一组 action 当 tracepoint num 触发的时候执行。如果没有设置 num 则将设置 action 到最近创建的 tracepoint 上(因此你可以定义一个 tracepoint 然后直接输入 actions 而不需要参数)。然后就要在后面输入 action，最后以 end 为结束。到目前为止，支持的 action 有 collect，teval 和 while-stepping。

collect expr1, expr2, ...

当 tracepoint 触发的时候，收集表达式的值。这个命令可接受用逗号分割的一组列表，这些列表除了可以是全局，局部或者本地变量，还可以是下面的这些参数：

$regs 收集全部寄存器。
$args 收集函数参数。
$locals 收集全部局部变量。

请 注意 collect 一个指针(collect ptr)将只能 collect 这个指针的地址. 在指针前面增加一个 * 将会让 action collect 指针指向的数据(collect *ptr)。

teval expr1, expr2, ...

当 tracepoint 触发的时候，执行指定的表达式。这个命令可接受用逗号分割的一组列表。表达式的结果将被删除，所以最主要的作用是把值设置到 trace 状态变量中 (see 普通 trace 状态变量)，而不用想 collect 一样把这些值存到 trace 帧中。

while-stepping n

请到 使用 while-stepping 让 Linux 内核做单步 去看如何使用它。

启动和停止 **tracepoint**

tracepoint 只有在用下面的 GDB 命令启动后才可以执行 action：

(gdb) tstart

它可以用下面的命令停止：

(gdb) tstop

Enable 和 disable tracepoint

和 breakpoint 一样，tracepoint 可以使用 GDB 命令 "enable" 和 "disable"。但是请注意 它们只在 tracepoint 停止的时候有效。

用 **tfind** 选择 **trace** 帧缓存里面的条目

tracepoint 停止的时候，GDB 命令"tfind"可以用来选择 trace 帧缓存里面的条目。

当 GDB 在"tfind"模式的时候，其只能显示 tracepoint action collect 的存在于这个条目中的数据。所以 GDB 将输出一些错误信息如果想打印没有 collect 的数据例如函数的参数。这不是 bug，不用担心。

如果想选择下一个条目，可以再次使用命令"tfind"。还可以用"tfind 条目 ID"去选择某个条目。

要回到普通模式(在普通模式直接访问当前值)，请使用 GDB 命令"tfind -1"。 请到 http://sourceware.org/gdb/current/onlinedocs/gdb/tfind.html 取得它的详细信息。

如何处理错误 "No such file or directory." 或者 "没有那个文件或目录."

当 GDB 不能找到 Linux 内核源码的时候，其就会显示这个错误信息。 例如：

```
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  vfs_read (file=0xffff8801b6c3a500, buf=0x3f588b8 <Address
0x3f588b8 out of bounds>, count=8192,
    pos=0xffff8801eee49f48) at /build/buildd/linux-
3.2.0/fs/read_write.c:365
365     /build/buildd/linux-3.2.0/fs/read_write.c: 没有那个文件或目录.
```

你可以用 GDB 命令 "set substitute-path" 处理它。前面这个例子 Linux 内核源码在"/build/buildd/test/linux-3.2.0/"但是 vmlinux 让 GDB 在"/build/buildd/linux-3.2.0/"找内核远啊，你可以处理他们：

```
(gdb) set substitute-path /build/buildd/linux-3.2.0/
/build/buildd/test/linux-3.2.0/
(gdb) tfind
Found trace frame 1, tracepoint 1
#0  vfs_read (file=0xffff8801c36e6400, buf=0x7fff51a8f110
<Address 0x7fff51a8f110 out of bounds>, count=16,
    pos=0xffff8801761dff48) at /build/buildd/linux-
3.2.0/fs/read_write.c:365
365     {
```

GDB 还提供其他的命令处理源码问题，请到 http://sourceware.org/gdb/current/onlinedocs/gdb/Source-Path.html 取得他们的介绍。

保存 trace 帧信息到一个文件中

/sys/kernel/debug/gtpframe 是一个当 KGTP 停止时的 tfine 格式（GDB 可以读取它）的接口。

请 注意 有些"cp"不能很好的处理这个问题，可以用"cat /sys/kernel/debug/gtpframe > ./gtpframe"拷贝它。

你可以在需要的时候打开文件 gtpframe:

```
(gdb) target tfile ./gtpframe
Tracepoint 1 at 0xffffffff8114f3dc: file /home/teawater/kernel/linux-
2.6/fs/readdir.c, line 24.
Created tracepoint 1 for target's tracepoint 1 at 0xffffffff8114f3c0.
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  vfs_readdir (file=0xffff880036e8f300, filler=0xffffffff8114f240
<filldir>, buf=0xffff880001e5bf38)
    at /home/teawater/kernel/linux-2.6/fs/readdir.c:24
24      {
```

显示和存储 **tracepoint**

你可以用 GDB 命令"info tracepoints"显示所有的 tracepoint。

你可以用 GDB 命令"save tracepoints filename"保存所有的设置 tracepoint 的命令到文件 filename 里。于是你可以在之后用 GDB 命令"source filename"设置重新这些 tracepoint。

删除 **tracepoint**

GDB 命令"delete id"将删除 tracepoint id。如果"delete"没有参数，则删除所有 tracepoint。

用 tracepoint 从内核中某点取得寄存器信息

下面是记录内核调用函数"vfs_readdir"时的寄存器信息的例子：

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) trace vfs_readdir
Tracepoint 1 at 0xc01a1ac0: file
/home/teawater/kernel/linux-2.6/fs/readdir.c, line 23.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect $reg
>end
(gdb) tstart
(gdb) shell ls
(gdb) tstop
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  0xc01a1ac1 in vfs_readdir (file=0xc5528d00, filler=0xc01a1900
<filldir64>,
   buf=0xc0d09f90) at /home/teawater/kernel/linux-
2.6/fs/readdir.c:23
23      /home/teawater/kernel/linux-2.6/fs/readdir.c: No such file or
directory.
       in /home/teawater/kernel/linux-2.6/fs/readdir.c
(gdb) info reg
eax            0xc5528d00       -984445696
ecx            0xc0d09f90       -1060069488
edx            0xc01a1900       -1072031488
ebx            0xfffffff7       -9
esp            0xc0d09f8c       0xc0d09f8c
ebp            0x0      0x0
esi            0x8061480        134616192
edi            0xc5528d00       -984445696
eip            0xc01a1ac1       0xc01a1ac1 <vfs_readdir+1>
eflags         0x286    [ PF SF IF ]
cs             0x60     96
ss             0x8061480        134616192
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
(gdb) tfind
Found trace frame 1, tracepoint 1
0xc01a1ac1      23      in /home/teawater/kernel/linux-2.6/fs/readdir.c
(gdb) info reg
eax            0xc5528d00       -984445696
```

```
ecx            0xc0d09f90       -1060069488
edx            0xc01a1900       -1072031488
ebx            0xfffffff7       -9
esp            0xc0d09f8c       0xc0d09f8c
ebp            0x0      0x0
esi            0x8061480        134616192
edi            0xc5528d00       -984445696
eip            0xc01a1ac1       0xc01a1ac1 <vfs_readdir+1>
eflags         0x286    [ PF SF IF ]
cs             0x60     96
ss             0x8061480        134616192
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

用 tracepoint 从内核中某点取得变量的值

下面是记录内核调用函数"vfs_readdir"时"jiffies_64"的值的例子：

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) trace vfs_readdir
Tracepoint 1 at 0xc01ed740: file /home/teawater/kernel/linux-
2.6/fs/readdir.c, line 24.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect jiffies_64
>collect file->f_path.dentry->d_iname
>end
(gdb) tstart
(gdb) shell ls
arch    drivers   include  kernel    mm            Module.symvers
security  System.map  virt
block   firmware  init     lib       modules.builtin  net           sound     t
vmlinux
crypto  fs        ipc      Makefile  modules.order    scripts       source
usr       vmlinux.o
(gdb) tstop
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  0xc01ed741 in vfs_readdir (file=0xf4063000, filler=0xc01ed580
<filldir64>, buf=0xd6dfdf90)
    at /home/teawater/kernel/linux-2.6/fs/readdir.c:24
24      {
(gdb) p jiffies_64
$1 = 4297248706
(gdb) p file->f_path.dentry->d_iname
$1 = "b26", '\000' <repeats 28 times>
```

显示当前这一条 **trace** 缓存里存储的所有信息

在用"tfind"选择好一个条目后，你可以用"tdump"。

```
(gdb) tdump
Data collected at tracepoint 1, trace frame 0:
$cr = void
file->f_path.dentry->d_iname =
"gtp\000.google.chrome.g05ZYO\000\235\337\000\000\000\000\200\
067k\364\200\067", <incomplete sequence \364>
jiffies_64 = 4319751455
```

取得 **tracepoint** 的状态

请用 GDB 命令"tstatus"。

设置 trace 缓存为循环缓存

http://sourceware.org/gdb/current/onlinedocs/gdb/Starting-and-Stopping-Trace-Experiments.html

帧缓存默认情况下不是循环缓存。当缓存满了的时候，tracepoint 将停止。

下面的命令将设置 trace 缓存为循环缓存，当缓存满了的时候，其将自动删除最早的数据并继续 trace。

(gdb) set circular-trace-buffer on

GDB 断开的时候不要停止 tracepoint

http://sourceware.org/gdb/current/onlinedocs/gdb/Starting-and-Stopping-Trace-Experiments.html

默认情况下，当 GDB 断开 KGTP 的时候将自动停止 tracepoint 并删除 trace 帧。

下面的命令将打开 KGTP disconnect-trace。在设置之后，当 GDB 断开 KGTP 的时候，KGTP 将不停止 tracepoint。GDB 重新连到 KGTP 的时候，其可以继续控制 KGTP。

(gdb) set disconnected-tracing on

kprobes-optimization 和 tracepoint 的执行速度

因为 tracepoint 是和 Linux 内核一起执行，所以它的速度将影响到系统执行的速度。

KGTP tracepoint 是基于 Linux 内核 kprobe。因为普通 kprobe 是基于断点指令，所以它的速度不是很快。

但是如果你的 arch 是 X86_64 或者 X86_32 而且内核配置没有打开"Preemptible Kernel" (PREEMPT)，kprobe 的速度将被 kprobes-optimization (CONFIG_OPTPROBES)提高很多。

可以用下面的命令来确认：

```
sysctl -A | grep kprobe
debug.kprobes-optimization = 1
```

这个的意思就是你的系统支持 kprobes-optimization。

请 注意 一些 KGTP 的功能会导致 tracepoint 只能使用普通 kprobe 即使系统支持 kprobes-optimization。文档将在介绍这些功能的时候增加提醒，如果你很介意 tracepoint 的速度就请避免使用这些功能。

如何使用 **trace** 状态变量

http://sourceware.org/gdb/current/onlinedocs/gdb/Trace-State-Variables.html

trace 状态变量简称 TSV。

TSV 可以在 tracepoint action 和 condition 中被访问，并且可以直接被 GDB 命令访问。

请 注意 GDB 7.2.1 和更晚的版本直接访问 trace 状态变量，比他们老的 GDB 只能通过命令"info tvariables"取得 trace 状态变量的值。

普通 trace 状态变量

定义 trace 状态变量$c.

```
(gdb) tvariable $c
```

trace 状态变量 $c 被创建并初始化 0。 下面的 action 将使用$c 记录内核里发生了多少次 IRQ。

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) trace handle_irq
(gdb) actions
Enter actions for tracepoint 3, one per line.
End with a line saying just "end".
>collect $c     #Save current value of $c to the trace frame buffer.
>teval $c=$c+1  #Increase the $c.
>end
```

你还可以将某个变量的值传到状态变量里，但是别忘记转化这个值为"uint64_t"。

```
>teval $c=(uint64_t)a
```

你可以取得$c 的值当 trace 在运行或者停止的时候。

```
(gdb) tstart
(gdb) info tvariables
$c              0          31554
(gdb) p $c
$5 = 33652
(gdb) tstop
(gdb) p $c
$9 = 105559
```

当使用 tfind 的时候，你可以分析 trace frame buffer。如果 trace 状态变量被收集了，你可以把它取出来。

```
(gdb) tstop
(gdb) tfind
(gdb) info tvariables
$c              0          0
(gdb) p $c
$6 = 0
(gdb) tfind 100
(gdb) p $c
$7 = 100
```

如果需要的时候，访问 trace 状态变量的 tracepoint action 将自动加锁，所以其可以很好的处理 trace 状态变量的竞态条件问题。

下面这个例子即使在一个多 CPU 的环境也可以正常使用。

```
>teval $c=$c+1
```

Per_cpu trace 状态变量

Per_cpu trace 状态变量是一种特殊的普通 trace 状态变量。 当一个 tracepoint action 访问到其的时候，其将自动访问这个 CPU 的 Per_cpu trace 状态变量。

它有两个优点：

1. 访问 Per_cpu trace 状态变量的 tracepoint actions 不存在竞态条件问题，所以其不需要对 trace 状态变量加锁。所以其在多核的机器上速度更快。

2. 写针对记录某个 CPU 的 tracepoint actions 比普通 trace 状态变量更容易。

如何定义

Per_cpu trace 状态变量有两种类型：

本地 CPU 变量

"per_cpu_"+string

或者

"p_"+string

例如：

(gdb) tvariable $p_count

在 tracepoint action 中访问这个 trace 状态变量的时候，其将返回这个变量在这个 action 运行的 CPU 上的值。

CPU id 变量

"per_cpu_"+string+CPU_id

或者

"p_"+string+CPU_id

例如：

```
(gdb) tvariable $p_count0
(gdb) tvariable $p_count1
(gdb) tvariable $p_count2
(gdb) tvariable $p_count3
```

在 tracepoint action 或者 GDB 命令行中访问这个变量的时候，其将返回这个变量在 CPU CPI_id 上的值。

下面这个例子可以自动这个这台主机上的每个 CPU 定义 CPU id 变量。(请 注意 用这些命令之前需要让 GDB 连上 KGTP。)

```
(gdb) set $tmp=0
(gdb) while $tmp<$cpu_number
 >eval "tvariable $p_count%d",$tmp
 >set $tmp=$tmp+1
 >end
```

例子 1

这个例子定义了一个记录每个 CPU 调用多少次 vfs_read 的 tracepoint。

```
tvariable $p_count
set $tmp=0
while $tmp<$cpu_number
  eval "tvariable $p_count%d",$tmp
  set $tmp=$tmp+1
  end
trace vfs_read
actions
  teval $p_count=$p_count+1
  end
```

于是你可以在"tstart"后显示每个 CPU 调用了多少次 vfs_read：

```
(gdb) p $p_count0
$3 = 44802
(gdb) p $p_count1
$4 = 55272
(gdb) p $p_count2
$5 = 102085
(gdb) p $p_count3
```

例子 2

这个例子记录了每个 CPU 上关闭 IRQ 时间最长的函数的 stack dump。

```
set pagination off

tvariable $bt=1024
tvariable $p_count
tvariable $p_cc
set $tmp=0
while $tmp<$cpu_number
eval "tvariable $p_cc%d",$tmp
set $tmp=$tmp+1
end

tvariable $ignore_error=1

trace arch_local_irq_disable
  commands
    teval $p_count=$clock
  end
trace arch_local_irq_enable if ($p_count && $p_cc < $clock -
$p_count)
  commands
    teval $p_cc = $clock - $p_count
    collect $bt
    collect $p_cc
    teval $p_count=0
  end

enable
set pagination on
```

特殊 trace 状态变量 $current_task，$current_task_pid，$current_thread_info，$cpu_id，$dump_stack，$printk_level，$printk_format，$printk_tmp，$clock，$hardirq_count，$softirq_count 和 $irq_count

KGTP 特殊 trace 状态变量$current_task，$current_thread_info，$cpu_id 和 $clock 可以很容易的访问各种特殊的值，当你用 GDB 连到 KGTP 后就可以访问到他们。你可以在 tracepoint 条件和 actions 里使用他们。

在 tracepoint 条件和 actions 里访问$current_task 可以取得 get_current()的返回值。

在 tracepoint 条件和 actions 里访问$current_task_pid 可以取得 get_current()->pid 的值。

在 tracepoint 条件和 actions 里访问$current_thread_info 可以取得 current_thread_info()的返回值。

在 tracepoint 条件和 actions 里访问$cpu_id 可以取得 smp_processor_id()的返回值。

在 tracepoint 条件和 actions 里访问$clock 可以取得 local_clock()的返回值，也就是取得纳秒为单位的时间戳。

$rdtsc 只在体系结构是 X86 或者 X86_64 的时候访问的到，任何时候访问它可以取得用指令 RDTSC 取得的 TSC 的值。

在 tracepoint 条件和 actions 里访问$hardirq_count 可以取得 hardirq_count()的返回值。

在 tracepoint 条件和 actions 里访问$softirq_count 可以取得 softirq_count()的返回值。

在 tracepoint 条件和 actions 里访问$irq_count 可以取得 irq_count()的返回值。

KGTP 还有一些特殊 trace 状态变量$dump_stack，$printk_level，$printk_format 和 $printk_tmp。他们可以用来直接显示值，请看如何让 tracepoint 直接输出信息。

下面是一个用$c 记录进程 16663 调用多少次 vfs_read 并收集 thread_info 结构的例子：

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) trace vfs_read if (((struct task_struct *)$current_task)->pid ==
16663)
(gdb) tvariable $c
(gdb) actions
Enter actions for tracepoint 4, one per line.
End with a line saying just "end".
>teval $c=$c+1
>collect (*(struct thread_info *)$current_thread_info)
>end
(gdb) tstart
```

```
(gdb) info tvariables
Name          Initial     Current
$c            0           184
$current_task   0           <unknown>
$current_thread_info 0          <unknown>
$cpu_id       0           <unknown>
(gdb) tstop
(gdb) tfind
(gdb) p *(struct thread_info *)$current_thread_info
$10 = {task = 0xf0ac6580, exec_domain = 0xc07b1400, flags = 0,
status = 0, cpu = 1, preempt_count = 2, addr_limit = {
   seg = 4294967295}, restart_block = {fn = 0xc0159fb0
<do_no_restart_syscall>, {{arg0 = 138300720, arg1 = 11,
      arg2 = 1, arg3 = 78}, futex = {uaddr = 0x83e4d30, val = 11,
flags = 1, bitset = 78, time = 977063750,
      uaddr2 = 0x0}, nanosleep = {index = 138300720, rmtp = 0xb,
expires = 335007449089}, poll = {
      ufds = 0x83e4d30, nfds = 11, has_timeout = 1, tv_sec = 78,
tv_nsec = 977063750}}},
  sysenter_return = 0xb77ce424, previous_esp = 0, supervisor_stack
= 0xef340044 "", uaccess_err = 0}
```

这是一个记录每个 CPU 调用了多少次 sys_read()的例子。

```
(gdb) tvariable $c0
(gdb) tvariable $c1
(gdb) trace sys_read
(gdb) condition $bpnum ($cpu_id == 0)
(gdb) actions
>teval $c0=$c0+1
>end
(gdb) trace sys_read
(gdb) condition $bpnum ($cpu_id == 1)
(gdb) actions
>teval $c1=$c1+1
>end
(gdb) info tvariables
Name          Initial     Current
$current_task   0           <unknown>
$cpu_id       0           <unknown>
$c0           0           3255
$c1           0           1904
```

sys_read() 在 CPU0 上被执行了 3255 次，CPU1 上执行了 1904 次。请 注意 这个例子只是为了显示如何使用$cpu_id，实际上用 per_cpu trace 状态变量写更好。

特殊 trace 状态变量 $self_trace

$self_trace 和前面介绍的特殊 trace 状态变量不同，它是用来控制 tracepoint 的行为的。

默认情况下，tracepoint 被触发后，如果 current_task 是 KGTP 自己的进程（GDB，netcat，getframe 或者其他访问 KGTP 接口的进程）的时候，其将不执行任何 actions。

如果你想让 tracepoint actions 和任何 task 的时候都执行，请包含一个包含一个访问到 $self_trace 的命令到 actions 中，也就是说增加下面的命令到 actions 中：

```
>teval $self_trace=0
```

用$kret trace 函数的结尾

有时，因为内核是用优化编译的，所以在函数结尾设置 tracepoint 有时很困难。这时你可以用$kret 帮助你。

$kret 是一个类似$self_trace 的特殊 trace 状态变量。当你在 tracepoint action 里设置它的值的时候，这个 tracepoint 将用 kretprobe 而不是 kprobe 注册。于是其就可以 trace 一个函数的结尾。

请 注意 这个 tracepoint 必须用 **"function_name"** 的格式设置在函数的第一个地址上。

下面的部分是一个例子：

```
#"*(function_name)" format can make certain that GDB send the
first address of function to KGTP.
(gdb) trace *vfs_read
(gdb) actions
>teval $kret=0
#Following part you can set commands that you want.
```

用 $ignore_error 和 $last_errno 忽略 tstart 的错误

当 KGTP 在 tstart 取得错误，这个命令将失败。

但有时我们需要忽略这个错误信息并让 KGTP 工作。例如：如果你在 inline 函数 spin_lock 设置 tracepoint，这个 tracepoint 将被设置到很多地址上，有一些地址不能设置 kprobe，于是它就会让 tstart 出错。这时你就可以用"$ignore_error"忽略这些错误。

最后一个错误信息将存在"$last_errno"中。

(gdb) tvariable $ignore_error=1

这个命令将打开忽略。

(gdb) tvariable $ignore_error=0

这个命令将关闭忽略。

使用 $cooked_clock 和 $cooked_rdtsc 取得不包含 KGTP 运行时间的时间信息

访问这两个 trace 状态变量可以取得不包含 KGTP 运行时间的时间信息，于是我们可以取得一段代码更真实的执行时间即使这个 tracepoint 的 action 比较复杂。

使用 $xtime_sec 和 $xtime_nsec 取得 timespec

访问 trace 状态变量将返回用 getnstimeofday 取得的 timespec 时间信息。

$xtime_sec 将返回 timespec 秒的部分。

$xtime_nsec 将返回 timespec 纳秒的部分。

如何 backtrace (stack dump)

每次你的程序做一个函数调用的时候， 这次调用的信息就会生成。这些信息包括调用函数的地址，调用参数，局部变量的值。这些信息被存储在我们称为栈帧的地方，栈帧是从调用栈中分配而来。

通过$bt 收集栈并用 GDB 命令 backtrace 进行分析

因为这个方法更快（因为在 trace 的时候只收集）而且可以分析出大部分的调用栈中的信息（前面介绍的栈信息都可以分析出来），所以时间你使用这个方法做栈分析。

首先我们需要在 tracepoint action 中增加命令收集栈。

~~GDB 收集栈的通常命令是：在 x86_32, 下面的命令将收集 512 字节的栈内容。~~

~~>collect *(unsigned char *)$esp@512~~

~~在 x86_64, 下面的命令将收集 512 字节的栈内容。~~

~~>collect *(unsigned char *)$rsp@512~~

~~在 MIPS 或者 ARM, 下面的命令将收集 512 字节的栈内容。~~

~~>collect *(unsigned char *)$sp@512~~

~~这些命令很难记，而且不同的体系结构需要不同的命令。~~

KGTP 有一个特殊 trace 状态变量$bt。如果 tracepoint action 访问到它，KGTP 将自动收集$bt 长度（默认值是 512）的栈。下面这个 action 将收集 512 字节的栈内存：

```
>collect $bt
```

如果你想改变$bt 的值，你可以在"tstart"使用下面这个 GDB 命令：

```
(gdb) tvariable $bt=1024
```

下面的部分是一个收集栈并用 GDB 进行分析的例子：

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) trace vfs_readdir
Tracepoint 1 at 0xffffffff8118c300: file
/home/teawater/kernel2/linux/fs/readdir.c, line 24.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect $bt
>end
(gdb) tstart
(gdb) shell ls
1     crypto    fs          include  kernel    mm
Module.symvers  security  System.map  vmlinux
arch   drivers   hotcode.html   init    lib      modules.builtin  net
sound     usr         vmlinux.o
block  firmware  hotcode.html~  ipc     Makefile  modules.order
scripts        source    virt
(gdb) tstop
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  vfs_readdir (file=0xffff8800c5556d00, filler=0xffffffff8118c4b0
```

```
<filldir>, buf=0xffff880108709f40)
    at /home/teawater/kernel2/linux/fs/readdir.c:24
24      {
(gdb) bt
#0  vfs_readdir (file=0xffff8800c5556d00, filler=0xffffffff8118c4b0
<filldir>, buf=0xffff880108709f40)
    at /home/teawater/kernel2/linux/fs/readdir.c:24
#1  0xffffffff8118c689 in sys_getdents (fd=<optimized out>,
dirent=0x1398c58, count=32768) at
/home/teawater/kernel2/linux/fs/readdir.c:214
#2  <signal handler called>
#3  0x00007f00253848a5 in ?? ()
#4  0x00003efd32cddfc9 in ?? ()
#5  0x00002c15b7d04101 in ?? ()
#6  0x000019c0c5704bf1 in ?? ()
#7  0x0000000900000000 in ?? ()
#8  0x000009988cc8d269 in ?? ()
#9  0x000009988cc9b8d1 in ?? ()
#10 0x0000000000000000 in ?? ()
(gdb) up
#1  0xffffffff8118c689 in sys_getdents (fd=<optimized out>,
dirent=0x1398c58, count=32768) at
/home/teawater/kernel2/linux/fs/readdir.c:214
214             error = vfs_readdir(file, filldir, &buf);
(gdb) p buf
$1 = {current_dir = 0x1398c58, previous = 0x0, count = 32768,
error = 0}
(gdb) p error
$3 = -9
(gdb) frame 0
#0  vfs_readdir (file=0xffff8800c5556d00, filler=0xffffffff8118c4b0
<filldir>, buf=0xffff880108709f40)
    at /home/teawater/kernel2/linux/fs/readdir.c:24
24      {
```

从这个例子，我们可以看到一些分析调用栈的 GDB 命令：

- **bt** 是 GDB 命令 backtrace 的别名，这个命令将打印 stack 中的信息：每一行是一个调用栈。
- **up n** 是向上移动 n 个帧。如果 n 是正数，则向外到更高的帧，一直到这个栈最大的一行。n 的默认值是 1。
- **down n** 是向下移动 n 个帧。如果 n 是正数，则向内到更低的帧，一直到最新创建的那个栈帧。n 的默认值是 1。

  is move n frames down the stack. For positive numbers n, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. n defaults to one. 你可以把 down 缩写为 do。

- **frame n** 是选择帧 n。帧 0 是最近创建的帧，帧 1 调用这个帧的帧。所以最高的帧是 main。

你还可以看到当你用 up，down 或者 frame 来选择调用栈帧的时候，你可以输出不同帧的参数和局部变量。

要取得更多关于如何使用 GDB 分析调用栈的信息，请到 http://sourceware.org/gdb/current/onlinedocs/gdb/Stack.html。

用$_ret来取得当前函数的调用函数的栈

如果你只想取得当前函数的调用函数的栈，可以用$_ret。

请 注意 使用$_ret 的 tracepoint 不能设置在函数的第一个地址上。

例如：

```
(gdb) list vfs_read
360     }
361
362     EXPORT_SYMBOL(do_sync_read);
363
364     ssize_t vfs_read(struct file *file, char __user *buf, size_t count,
loff_t *pos)
365     {
366             ssize_t ret;
367
368             if (!(file->f_mode & FMODE_READ))
369                     return -EBADF;
(gdb) trace 368
Tracepoint 2 at 0xffffffff8117a244: file
/home/teawater/kernel2/linux/fs/read_write.c, line 368.
(gdb) actions
Enter actions for tracepoint 2, one per line.
End with a line saying just "end".
>collect $_ret
>end
(gdb) tstart
(gdb) tstop
(gdb) tfind
Found trace frame 0, tracepoint 2
#0  vfs_read (file=0xffff880141c46000, buf=0x359bda0 <Address
0x359bda0 out of bounds>, count=8192, pos=0xffff88012fa49f48)
    at /home/teawater/kernel2/linux/fs/read_write.c:368
368             if (!(file->f_mode & FMODE_READ))
(gdb) bt
#0  vfs_read (file=0xffff880141c46000, buf=0x359bda0 <Address
0x359bda0 out of bounds>, count=8192, pos=0xffff88012fa49f48)
    at /home/teawater/kernel2/linux/fs/read_write.c:368
#1  0xffffffff8117a3ea in sys_read (fd=<optimized out>,
buf=<unavailable>, count=<unavailable>)
    at /home/teawater/kernel2/linux/fs/read_write.c:469
Backtrace stopped: not enough registers or memory available to
unwind further
(gdb) up
#1  0xffffffff8117a3ea in sys_read (fd=<optimized out>,
buf=<unavailable>, count=<unavailable>)
    at /home/teawater/kernel2/linux/fs/read_write.c:469
```

```
469                 ret = vfs_read(file, buf, count, &pos);
(gdb) p ret
$2 = -9
```

我们可以看到调用 vfs_read 的函数是 sys_read，函数 sys_read 的局部变量 ret 的值是-9。

用 $dump_stack 输出栈分析到 printk 里

因为这个方法需要在 trace 的时候分析栈并调用 printk，所以它比较慢，不安全，不清晰也不能访问调用栈中的很多内容，所以我建议你上一部分介绍的方法

KGTP 有一个特殊的 trace 状态变量$dump_stack，收集这个变量可以令 GDB 调用栈分析并用 printk 输出。 下面是一个让内核输出 vfs_readdir 栈分析的例子：

```
target remote /sys/kernel/debug/gtp
trace vfs_readdir
  commands
    collect $dump_stack
  end
```

于是你的内核就会 printk 这样的信息：

```
[22779.208064] gtp 1:Pid: 441, comm: python Not tainted 2.6.39-
rc3+ #46
[22779.208068] Call Trace:
[22779.208072]  [<fe653cca>] gtp_get_var+0x4a/0xa0 [gtp]
[22779.208076]  [<fe653d79>] gtp_collect_var+0x59/0xa0 [gtp]
[22779.208080]  [<fe655974>] gtp_action_x+0x1bb4/0x1dc0 [gtp]
[22779.208084]  [<c05b6408>] ? _raw_spin_unlock+0x18/0x40
[22779.208088]  [<c023f152>] ? __find_get_block_slow+0xd2/0x160
[22779.208091]  [<c01a8c56>] ? delayacct_end+0x96/0xb0
[22779.208100]  [<c023f404>] ? __find_get_block+0x84/0x1d0
[22779.208103]  [<c05b6408>] ? _raw_spin_unlock+0x18/0x40
[22779.208106]  [<c02e0838>] ? find_revoke_record+0xa8/0xc0
[22779.208109]  [<c02e0c45>] ?
jbd2_journal_cancel_revoke+0xd5/0xe0
[22779.208112]  [<c02db51f>] ?
__jbd2_journal_temp_unlink_buffer+0x2f/0x110
[22779.208115]  [<fe655c4c>] gtp_kp_pre_handler+0xcc/0x1c0
[gtp]
[22779.208118]  [<c05b8a88>]
kprobe_exceptions_notify+0x3d8/0x440
[22779.208121]  [<c05b7d54>] ?
hw_breakpoint_exceptions_notify+0x14/0x180
[22779.208124]  [<c05b95eb>] ? sub_preempt_count+0x7b/0xb0
[22779.208126]  [<c0227ac5>] ? vfs_readdir+0x15/0xb0
[22779.208128]  [<c0227ac4>] ? vfs_readdir+0x14/0xb0
[22779.208131]  [<c05b9743>] notifier_call_chain+0x43/0x60
[22779.208134]  [<c05b9798>]
__atomic_notifier_call_chain+0x38/0x50
[22779.208137]  [<c05b97cf>] atomic_notifier_call_chain+0x1f/0x30
[22779.208140]  [<c05b980d>] notify_die+0x2d/0x30
[22779.208142]  [<c05b71c5>] do_int3+0x35/0xa0
```

如何让 tracepoint 直接输出信息

在前面的章节，你可以看到如果想取得 Linux 内核的信息，你需要用 tracepoint "collect" action 来保存信息到 tracepoint 帧中并用 GDB tfind 命来来分析这些数据帧。

但是有时我们希望直接取得这些数据，所以 KGTP 提供了一种直接取得这些数据的方法。

切换 collect 为直接输出数据

KGTP 有特殊 trace 状态变量$printk_level，$printk_format 和 $printk_tmp 支持这个功能。

$printk_level，如果这个值是 8（这是默认值），"collect" action 将是普通行为也就是保存数据到 tracepoint 帧中。

如果值是 0-7，"collect" 将以这个数字为 printk 级别输出信息，这些级别是：

0 KERN_EMERG system is unusable
1 KERN_ALERT action must be taken immediately
2 KERN_CRIT critical conditions
3 KERN_ERR error conditions
4 KERN_WARNING warning conditions
5 KERN_NOTICE normal but significant condition
6 KERN_INFO informational
7 KERN_DEBUG debug-level messages

$printk_format，collect printk 将按照这里设置的格式进行输出。 这些格式是：

0 这是默认值。
如果 *collect* 的长度是 *1*，*2*，*4*，*8* 则其将输出一个无符号十进制数。
如果不是，则其将输出十六进制字串。
1 输出值是有符号十进制数。
2 输出值是无符号十进制数。
3 输出值是无符号十六进制数。
4 输出值是字符串。
5 输出值是十六进制字串。

如果要输出一个全局变量，需要将其先设置到$printk_tmp 中。

下面是一个显示调用 vfs_readdir 时的计数，pid，jiffies_64 和文件名的例子：

```
(gdb) target remote /sys/kernel/debug/gtp
(gdb) tvariable $c
(gdb) trace vfs_readdir
(gdb) actions
>teval $printk_level=0
>collect $c=$c+1
>collect ((struct task_struct *)$current_task)->pid
>collect $printk_tmp=jiffies_64
```

```
>teval $printk_format=4
>collect file->f_path.dentry->d_iname
>end
```

于是内核将 printk 这些信息：

```
gtp 1:$c=$c+1=41
gtp 1:((struct task_struct *)$current_task)->pid=12085
gtp 1:$printk_tmp=jiffies_64=4322021438
gtp 1:file->f_path.dentry->d_iname=b26
gtp 1:$c=$c+1=42
gtp 1:((struct task_struct *)$current_task)->pid=12085
gtp 1:$printk_tmp=jiffies_64=4322021438
gtp 1:file->f_path.dentry->d_iname=b26
```

"gtp 1" 的意思是数据是 tracepoint 1 输出的。

如何用 **watch tracepoint** 控制硬件断点记录内存访问

Watch tracepoint 可以通过设置一些特殊的 trace 状态变量设置硬件断点来记录内存访问。

请 注意 watch tracepoint 现在只有 X86 和 X86_64 支持。而且因为 Linux 2.6.26 和更老版本有一些 IPI 的问题，只有 Linux 2.6.27 和更新版本上可以正常使用动态 watch tracepoint。

watch tracepoint 的 trace 状态变量

名称	普通 **tracepoint** 写	普通 **tracepoint** 读	静态 **static tracepoint** 写	静态 **static tracepoint** 读	动态 **static tracepoint** 写	动态 **static tracepoint** 写
$watch_static	不支持	不支持	如果"teval $watch_static=1"则这个 tracepoint 是静态 watch tracepoint。	不支持	如果"teval $watch_static=0"则这个 tracepoint 是动态 watch tracepoint。	不支持
$watch_set_id	当这个 tracepoint 要设置一个动态 watch tracepoint 的时候，设置动态 watch tracepoint 的 ID 到 $watch_set_id 来标明你要设置哪个动态 watch tracepoint。	不支持	不支持	不支持	不支持	不支持
$watch_set_addr	当这个 tracepoint 要设置一个动态 watch tracepoint 的时候，设置动态 watch tracepoint 的地址到 $watch_set_addr 来标明你要设置哪个动态 watch tracepoint。	不支持	不支持	不支持	不支持	不支持

$watch_type	当这个 tracepoint 要设置一个动态 watch tracepoint 的时候，设置 watch 类型到 $watch_type。 0 是执行。 1 是写。 2 是读或者写。	取得这个 tracepoint 设置到 $watch_type 里的值。	设置 watch tracepoint 的类型。	取得这个 watch tracepoint 的类型。	设置 watch tracepoint 的默认类型。	取得这个 watch tracepoint 在实际执行中的类型。
$watch_size	当这个 tracepoint 要设置一个动态 watch tracepoint 的时候，设置 watch 长度到 $watch_size。 长度是 1, 2, 4, 8。	取得这个 tracepoint 设置到 $watch_size 里的值。	设置 watch tracepoint 的长度。	取得这个 watch tracepoint 的长度。	设置 watch tracepoint 的默认长度。	取得这个 watch tracepoint 在实际执行中的长度。
$watch_start	设置地址到动态 watch tracepoint($watch_set_addr 或者 $watch_set_id 设置)中并让其开始工作。	取得这次开始的返回值。（其可能会失败因为 X86 只有 4 个硬件断点） 取得 0 则成功，小于 0 则是错误 ID。	不支持	不支持	不支持	不支持
$watch_stop	设置地址到 $watch_stop 将让一个 watch 这个地址的动态 watch tracepoint 停止。	取得这次停止的返回值。	不支持	不支持	不支持	不支持
$watch_trace_num	不支持	不支持	不支持	不支持	不支持	设置这个动态 watch tracepoint 的 tracepoint 的

						号码。
$watch_trace_addr	不支持	不支持	不支持	不支持	不支持	设置这个动态 watch tracepoint 的 tracepoint 的地址。
$watch_addr	不支持	不支持	不支持	这个 watch tracepoint 监视的地址。	不支持	这个 watch tracepoint 监视的地址。
$watch_val	不支持	不支持	不支持	这个 watch tracepoint 监视的内存的当前值。	不支持	这个 watch tracepoint 监视的内存的当前值。
$watch_prev_val	不支持	不支持	不支持	这个 watch tracepoint 监视的内存的修改前值。	不支持	这个 watch tracepoint 监视的内存的修改前值。
$watch_count	不支持	不支持	不支持	不支持	不支持	这个 watch tracepoint 会话的一个特殊计数 ID。

静态 watch tracepoint

当你要监视全局变量或者可以取得地址的变量的值的时候，你可以使用静态 watch tracepoint。下面是一个监视 jiffies_64 写的例子:

```
#静态 watch tracepoint 从 tracepoint 的地址中取得要监视的地址
trace *&jiffies_64
  actions
    #Set this watch tracepoint to static
    teval $watch_static=1
    #Watch memory write
    teval $watch_type=1
    teval $watch_size=8
    collect $watch_val
    collect $watch_prev_val
    collect $bt
  end
```

动态 watch tracepoint

当你要监视局部变量或者只能在函数中取得地址的变量的值的时候，你可以使用动态 watch tracepoint。下面是一个监视函数 function get_empty_filp 中 f->f_posf->f_op 写的例子:

```
trace *1
  commands
    teval $watch_static=0
    teval $watch_type=1
    teval $watch_size=8
    collect $bt
    collect $watch_addr
    collect $watch_val
    collect $watch_prev_val
  end
```

定义了一个动态 watch tracepoint。地址"1"并不是其要监视的地址。其将帮助 tracepoint 来找到这个动态 watch tracepoint。

```
list get_empty_filp
trace 133
  commands
    teval $watch_set_addr=1
    teval $watch_size=4
    teval $watch_start=&(f->f_pos)
    teval $watch_size=8
    teval $watch_start=&(f->f_op)
  end
```

在函数 get_empty_filp 中定义一个普通 tracepoint，其将开始监视 f->f_pos 和 f->f_op。

```
trace file_sb_list_del
  commands
    teval $watch_stop=&(file->f_pos)
    teval $watch_stop=&(file->f_op)
  end
```

在函数 file_sb_list_del 中定义一个普通 tracepoint，其将停止监视 file->f_pos 和 file->f_op。

使用 **while-stepping** 让 **Linux** 内核做单步

请 注意 while-stepping 现在只有 X86 和 X86_64 支持。

介绍使用 while-stepping 的视频 http://www.codepark.us/a/12。

如何使用 **while-stepping**

while-stepping 是一种可以包含 actions 的特殊 tracepoint action。

当一个 actions 中包含了“while-stepping n”的 tracepoint 执行的时候，其将做 n 次单步并执行 while-stepping 的 actions。例如：

```
trace vfs_read
#因为单步会影响系统速度，所以最好用 passcount 或者 condition 限制 tracepoint 的执行次数。
passcount 1
  commands
    collect $bt
    collect $step_count
    #做 2000 次单步。
    while-stepping 2000
      #下面这部分是"while-stepping 2000"的 actions。
      #因为单步可能会执行到其他函数，所以最好不要访问局部变量。
      collect $bt
      collect $step_count
    end
  end
```

请 注意 tracepoint 在执行单步的时候会关闭当前 CPU 的中断。 在 actions 中访问 **$step_count** 将得到从 1 开始的这步的计数。

读 while-stepping 的 traceframe

不同 step 的数据将会被记录到不同的 traceframe 中，你可以用 tfind (用 tfind 选择 trace 帧缓存里面的条目) 选择他们。

或者你可以将 KGTP 切换到回放模式，这样 GDB 可以用执行和反向执行命令选择一个 while-stepping tracepoint 的 traceframe。例如：

用 tfind 选择一个 while-stepping 的 traceframe。

```
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  vfs_read (file=0xffff8801f7bd4c00, buf=0x7fff74e4edb0
<Address 0x7fff74e4edb0 out of bounds>, count=16,
    pos=0xffff8801f4b45f48) at /build/buildd/linux-
3.2.0/fs/read_write.c:365
365     {
```

下面的命令将切换 KGTP 到回放模式。

```
(gdb) monitor replay
(gdb) tfind -1
No longer looking at any trace frame
#0  vfs_read (file=0xffff8801f7bd4c00, buf=0x7fff74e4edb0
<Address 0x7fff74e4edb0 out of bounds>, count=16,
    pos=0xffff8801f4b45f48) at /build/buildd/linux-
3.2.0/fs/read_write.c:365
365     {
```

于是可以使用执行命令。

```
(gdb) n
368             if (!(file->f_mode & FMODE_READ))
(gdb) p file->f_mode
$5 = 3
```

设置断点 (只在回放模式下有效，不会影响到 Linux 内核执行)。

```
(gdb) b 375
Breakpoint 2 at 0xffffffff81179b75: file /build/buildd/linux-
3.2.0/fs/read_write.c, line 375.
(gdb) c
Continuing.

Breakpoint 2, vfs_read (file=0xffff8801f7bd4c00,
buf=0x7fff74e4edb0 <Address 0x7fff74e4edb0 out of bounds>,
count=16,
    pos=0xffff8801f4b45f48) at /build/buildd/linux-
```

```
3.2.0/fs/read_write.c:375
375             ret = rw_verify_area(READ, file, pos, count);
(gdb) s
rw_verify_area (read_write=0, file=0xffff8801f7bd4c00,
ppos=0xffff8801f4b45f48, count=16)
    at /build/buildd/linux-3.2.0/fs/read_write.c:300
300             inode = file->f_path.dentry->d_inode;
```

使用反向执行命令。

```
(gdb) rs

Breakpoint 2, vfs_read (file=0xffff8801f7bd4c00,
buf=0x7fff74e4edb0 <Address 0x7fff74e4edb0 out of bounds>,
count=16,
    pos=0xffff8801f4b45f48) at /build/buildd/linux-
3.2.0/fs/read_write.c:375
375             ret = rw_verify_area(READ, file, pos, count);
(gdb) rn
372             if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
```

GDB 命令 tstart，tstop，tfind 或者 quit 可以自动关闭回放模式。

如何显示被优化掉的变量值

有时 GDB 会这样输出信息：

> *inode has been optimized out of existence.*
> *res has been optimized out of existence.*

这是因为 inode 和 res 的值被优化掉了。内核用-O2 编译的所以你有时会碰到这个问题。

有两个方法处理这个问题：

升级你的 GCC

VTA branch http://gcc.gnu.org/wiki/Var_Tracking_Assignments 已经整合进 GCC 4.5，其可以帮助生成之前被标记为"optimized out"的值的调试信息。

通过分析汇编代码取得访问被优化掉变量的方法

即使升级了 GCC，你可能还会遇到问题。主要原因是数据在寄存器中但是 GCC 没有把信息放到调试信息中。所以 GDB 只能显示这个变量被又优化掉了。

但你可以通过分析汇编代码取得这个变量在哪并在 tracepoint actions 中访问其。

下面是一个在函数 get_empty_filp 中寻找变量"f"并在 tracepoint actions 中使用其的例子：

我们想 collect 变量 f 的值，但是其已经被优化掉了。

```
(gdb) list get_empty_filp
...
...
...
137             INIT_LIST_HEAD(&f->f_u.fu_list);
138             atomic_long_set(&f->f_count, 1);
139             rwlock_init(&f->f_owner.lock);
140             spin_lock_init(&f->f_lock);
141             eventpoll_init_file(f);
(gdb)
142             /* f->f_version: 0 */
143             return f;
(gdb) trace 143
Tracepoint 1 at 0xffffffff8119b30e: file fs/file_table.c, line 143.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect f
`f' is optimized away and cannot be collected.
```

现在用"disassemble /m"命令取得和"f"有关的汇编代码和源码并分析他们。

```
(gdb) disassemble /m get_empty_filp
...
...
...
125             f = kmem_cache_zalloc(filp_cachep, GFP_KERNEL);
126             if (unlikely(!f))
  0xffffffff8119b28c <+92>:    test   %rax,%rax
  0xffffffff8119b292 <+98>:    je     0xffffffff8119b362
<get_empty_filp+306>

127                     return ERR_PTR(-ENOMEM);
  0xffffffff8119b362 <+306>:   mov    $0xfffffffffffffff4,%rax
  0xffffffff8119b369 <+313>:   jmp    0xffffffff8119b311
<get_empty_filp+225>
```

因为"+98"到"+132"的代码因为属于 inline 函数所以没有在这里显示，但是你可以

用"disassemble get_empty_filp"取得他们。

```
  0xffffffff8119b287 <+87>:    callq  0xffffffff81181cb0
<kmem_cache_alloc>
  0xffffffff8119b28c <+92>:    test   %rax,%rax
  0xffffffff8119b28f <+95>:    mov    %rax,%rbx
  0xffffffff8119b292 <+98>:    je     0xffffffff8119b362
<get_empty_filp+306>
  0xffffffff8119b298 <+104>:   mov    0xb4d406(%rip),%edx        #
0xffffffff81ce86a4 <percpu_counter_batch>
  0xffffffff8119b29e <+110>:   mov    $0x1,%esi
  0xffffffff8119b2a3 <+115>:   mov    $0xffffffff81c05340,%rdi
---Type <return> to continue, or q <return> to quit---
  0xffffffff8119b2aa <+122>:   callq  0xffffffff8130dd20
<__percpu_counter_add>
```

根据汇编代码你可以看到 kmem_cache_alloc 的返回值在$rax 中，其的值被设置到了$rbx 中。

看起来$rbx 有"f"的值，让我们看其他的汇编代码。

```
128
129             percpu_counter_inc(&nr_files);
130             f->f_cred = get_cred(cred);
  0xffffffff8119b2b4 <+132>:   mov    %r12,0x70(%rbx)
```

设置一个值到 f 的元素中。汇编代码是设置$r12 的值到以$rbx 为基础地址的内存中。其让$rbx 看起来是"f"。

```
131             error = security_file_alloc(f);
  0xffffffff8119b2b8 <+136>:   mov    %rbx,%rdi
  0xffffffff8119b2bb <+139>:   callq  0xffffffff8128ee30
<security_file_alloc>

132             if (unlikely(error)) {
  0xffffffff8119b2c0 <+144>:   test   %eax,%eax
  0xffffffff8119b2c2 <+146>:   jne    0xffffffff8119b36b
<get_empty_filp+315>
---Type <return> to continue, or q <return> to quit---

133                     file_free(f);
134                     return ERR_PTR(error);
  0xffffffff8119b393 <+355>:   movslq -0x14(%rbp),%rax
  0xffffffff8119b397 <+359>:   jmpq   0xffffffff8119b311
<get_empty_filp+225>

135             }
136
137             INIT_LIST_HEAD(&f->f_u.fu_list);
```

```
138             atomic_long_set(&f->f_count, 1);
139             rwlock_init(&f->f_owner.lock);
  0xffffffff8119b2e4 <+180>:   movl   $0x100000,0x50(%rbx)

140             spin_lock_init(&f->f_lock);
  0xffffffff8119b2c8 <+152>:   xor    %eax,%eax
  0xffffffff8119b2d1 <+161>:   mov    %ax,0x30(%rbx)

141             eventpoll_init_file(f);
142             /* f->f_version: 0 */
143             return f;
  0xffffffff8119b30e <+222>:   mov    %rbx,%rax
```

在检查了其他汇编代码后，你可以确定$rbx 就是"f"。

于是你可以在 tracepoint actions 中通过访问$rbx 而访问"f"，例如：

```
(gdb) trace 143
Tracepoint 1 at 0xffffffff8119b30e: file fs/file_table.c, line 143.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
#collect f
>collect $rbx
#collect *f
>collect *((struct file *)$rbx)
#collect f->f_op
>collect ((struct file *)$rbx)->f_op
>end
```

如何取得函数指针指向的函数

如果函数指针没有被优化掉

你可以直接 collect 这个指针，例如：

```
377             count = ret;
378             if (file->f_op->read)
379                     ret = file->f_op->read(file, buf, count, pos);
(gdb)
(gdb) trace 379
Tracepoint 1 at 0xffffffff81173ba5: file
/home/teawater/kernel/linux/fs/read_write.c, line 379.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect file->f_op->read
>end
(gdb) tstart
(gdb) tstop
(gdb) tfind
(gdb) p file->f_op->read
$5 = (ssize_t (*)(struct file *, char *, size_t, loff_t *))
0xffffffff81173190 <do_sync_read>
#于是就知道 file->f_op->read 指向 do_sync_read。
```

如果函数指针被优化掉了

可以用 tracepoint step 处理这个问题，例如：

```
#找到调用指针的指令
(gdb) disassemble /rm vfs_read
379                     ret = file->f_op->read(file, buf, count, pos);
  0xffffffff81173ba5 <+181>:   48 89 da       mov    %rbx,%rdx
  0xffffffff81173ba8 <+184>:   4c 89 e9       mov    %r13,%rcx
  0xffffffff81173bab <+187>:   4c 89 e6       mov    %r12,%rsi
  0xffffffff81173bae <+190>:   4c 89 f7       mov    %r14,%rdi
  0xffffffff81173bb1 <+193>:   ff d0   callq  *%rax
  0xffffffff81173bb3 <+195>:   48 89 c3       mov    %rax,%rbx
(gdb) trace *0xffffffff81173bb1
Tracepoint 1 at 0xffffffff81173bb1: file
/home/teawater/kernel/linux/fs/read_write.c, line 379.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>while-stepping 1
 >collect $reg
 >end
>end
(gdb) tstart
(gdb) tstop
(gdb) tfind
#0  tty_read (file=0xffff88006ca74900, buf=0xb6b7dc <Address
0xb6b7dc out of bounds>, count=8176,
    ppos=0xffff88006e197f48) at
/home/teawater/kernel/linux/drivers/tty/tty_io.c:960
960     {
#于是就知道 file->f_op->read 指向 tty_read。
```

请 注意 while-stepping 将让 tracepoint 不能使用 kprobes-optimization。

/sys/kernel/debug/gtpframe 和离线调试

/sys/kernel/debug/gtpframe 是一个当 KGTP 停止时的 tfine 格式（GDB 可以读取它）的接口。

在运行 GDB 的主机上：

改变 "target remote XXXX" 为：

```
(gdb) target remote | perl ./getgtprsp.pl
```

之后像平时一样设置 tracepoint：

```
(gdb) trace vfs_readdir
Tracepoint 1 at 0xffffffff8114f3c0: file /home/teawater/kernel/linux-
2.6/fs/readdir.c, line 24.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
#If your GDB support tracepoint "printf" (see "Howto use tracepoint
printf"), use it to show the value directly is better.
>collect $reg
>end
(gdb) tstart
(gdb) stop
(gdb) quit
```

于是你可以在当前目录找到文件 gtpstart 和 gtpstop，把他们拷贝到你想调试的主机上。

在被调试主机上，先拷贝 KGTP 目录中的程序"putgtprsp"和"gtp.ko"到这台机器上。insmod gtp.ko 之后：

启动 tracepoint：

```
./putgtprsp ./gtpstart
```

停止 tracepoint：

```
./putgtprsp ./gtpstop
```

可以按照如何让 tracepoint 直接输出信息直接在板子上显示信息。

如果要保存 trace 帧之后再分析，你可以拷贝文件"/sys/kernel/debug/gtpframe"到有 GDB 的主机上。

请 注意 有些"cp"不能很好的处理这个问题，可以用"cat /sys/kernel/debug/gtpframe

> ./gtpframe"拷贝它。

在运行 GDB 的主机上：

```
(gdb) target tfile ./gtpframe
Tracepoint 1 at 0xffffffff8114f3dc: file /home/teawater/kernel/linux-
2.6/fs/readdir.c, line 24.
Created tracepoint 1 for target's tracepoint 1 at 0xffffffff8114f3c0.
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  vfs_readdir (file=0xffff880036e8f300, filler=0xffffffff8114f240
<filldir>, buf=0xffff880001e5bf38)
    at /home/teawater/kernel/linux-2.6/fs/readdir.c:24
24      {
```

请 注意 如果你想在使用离线调试后从远程主机上的 GDB 连接 KGTP，你需要在调用"nc"之前"rmmod gtp"和"insmod gtp.ko"。

如何使用 /sys/kernel/debug/gtpframe_pipe

这个接口提供和"gtpframe"同样的数据，但是可以在 KGTP tracepoint 运行的时候也可以使用。在数据读出之后，其将自动从 trace 帧里删除类似 ftrace "trace_pipe"。

用 GDB 读帧信息

```
#连接到接口上
(gdb) target tfile /sys/kernel/debug/gtpframe_pipe
#取得一个 trace 帧条目
(gdb) tfind 0
Found trace frame 0, tracepoint 1
#取得下一个
(gdb) tfind
Target failed to find requested trace frame.
(gdb) tfind 0
Found trace frame 0, tracepoint 1
```

这个方法和 python 一起分析内核比较好，add-ons/hotcode.py 就是这样的例子。

用 **cat** 读帧信息

sudo cat /sys/kernel/debug/gtpframe_pipe > g

于是所有帧信息都被存入了文件"g"。

用 **getframe** 读帧信息

KGTP 包含一个"getframe"可以用来帮助取得 trace 帧。

下面这里是它的帮助：

getframe -h
Get the trace frame of KGTP and save them in current
directory with tfile format.
Usage: ./getframe [option]

-g n Set the minimum free size limit to n G.
When free size of current disk is smaller than n G,
./getframe will exit (-q) or wait some seconds (-w).
The default value of it is 2 G.

-q Quit when current disk is smaller than
minimum free size limit (-g).

-w n Wait n seconds when current disk is smaller
than minimum free size limit (-g).

-e n Set the entry number of each tfile to n.
The default value of it is 1000.

-h Display this information.

使用 $pipe_trace

为了锁安全，KGTP 默认将自动忽略读/sys/kernel/debug/gtpframe_pipe 的任务。

如果你真希望 trace 这个任务而且确定这是安全的，你可以使用"tstart"之前使用下面的命令：

```
(gdb) tvariable $pipe_trace=1
```

于是 KGTP 将不再忽略读/sys/kernel/debug/gtpframe_pipe 的任务。

和用户层程序一起使用 KGTP

直接读用户层程序的内存

KGTP 可以不同停止应用层程序的情况下直接读取其内存，请到用户程序的内存取得如何做。

在 tracepoint 收集用户层程序的栈信息(可用来做 backtrace)

$current 是一个特殊 trace 状态变量。当一个 tracepoint 的 action 访问其的时候，tracepoint 将收集当前 task 的寄存器和内存值而不是内核中的值。

一般来说，tracepoint 通过 **task_pt_regs** 取得寄存器的值。于是在 tracepoint actions 中 collect **$current** 将让 tracepoint 访问当前 task。例如：

```
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect $current
>collect $bt
>end
```

此外，针对一些参数中包含指向当前 TASK 寄存器指针的特殊函数(例如：X86 的 do_IRQ 函数)，tracepoint 需要从函数的参数中取得寄存器信息。则设置指针到 $current 将让 tracepoint 得到其。例如：

```
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>teval $current=(uint64_t)regs
>collect $bt
>end
```

$current_task_user 是一个特殊 trace 状态变量。当 current task 在 user 模式的时候，其的值为真。

用这两个 trace 状态变量，就可以用 KGTP 收集用户层程序的栈信息(可用来做 backtrace)。

下面这个例子显示如何从用户层到 Linux 内核层做 backtrace(stack dump)：

```
#连接 KGTP(和上一节介绍的方法相同)
(gdb) target extended-remote /sys/kernel/debug/gtp
#设置一个收集进程 18776 的用户栈的 tracepoint。
(gdb) trace vfs_read
Tracepoint 1 at 0xffffffff8117a3d0: file
/home/teawater/kernel/linux/fs/read_write.c, line 365.
(gdb) condition 1 ($current_task_user && $current_task_pid ==
18776)
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect $current
>collect $bt
>end
#Setup a tracepoint that collect kernel space stack of task 18776.
```

```
#设置一个收集进程 18776 的内核栈的 tracepoint。
(gdb) trace vfs_read
Note: breakpoint 1 also set at pc 0xffffffff8117a3d0.
Tracepoint 2 at 0xffffffff8117a3d0: file
/home/teawater/kernel/linux/fs/read_write.c, line 365.
(gdb) condition 2 ($current_task_user && $current_task_pid ==
18776)
(gdb) actions
Enter actions for tracepoint 2, one per line.
End with a line saying just "end".
>collect $bt
>end
(gdb) tstart
(gdb) tstop
#下面这部分和上一节相同，增加一个新的 inferior 用来分析应用程序的信息。
(gdb) add-inferior
Added inferior 2
(gdb) inferior 2
[Switching to inferior 2 [<null>] (<noexec>)]
(gdb) file gdb
Reading symbols from /usr/local/bin/gdb...done.
(gdb) attach 18776
#tracepoint 1 收集了用户层的栈信息。
(gdb) tfind
Found trace frame 0, tracepoint 1
#0  0x00007f77331d7d0f in __read_nocancel () from /lib/x86_64-
linux-gnu/libpthread.so.0
#这是程序 18776 用户层的 backtrace。
(gdb) bt
#0  0x00007f77331d7d0f in __read_nocancel () from /lib/x86_64-
linux-gnu/libpthread.so.0
#1  0x000000000078e145 in rl_callback_read_char () at
../../src/readline/callback.c:201
#2  0x0000000000069de79 in rl_callback_read_char_wrapper
(client_data=<optimized out>) at ../../src/gdb/event-top.c:169
#3  0x0000000000069ccf8 in process_event () at ../../src/gdb/event-
loop.c:401
#4  process_event () at ../../src/gdb/event-loop.c:351
#5  0x0000000000069d448 in gdb_do_one_event () at
../../src/gdb/event-loop.c:465
#6  0x0000000000069d5d5 in start_event_loop () at
../../src/gdb/event-loop.c:490
#7  0x0000000000697083 in captured_command_loop
(data=<optimized out>) at ../../src/gdb/main.c:226
#8  0x000000000695d8b in catch_errors (func=0x697070
<captured_command_loop>, func_args=0x0, errstring=0x14df99e
"",
    mask=6) at ../../src/gdb/exceptions.c:546
```

#9 0x00000000006979e6 in captured_main (data=<optimized out>) at ../../src/gdb/main.c:1001
#10 0x0000000000695d8b in catch_errors (func=0x697360 <captured_main>,
*   func@entry=<error reading variable: PC not available>, func_args=0x7fff08afd5b0,*
*   func_args@entry=<error reading variable: PC not available>, errstring=<unavailable>,*
*   errstring@entry=<error reading variable: PC not available>, mask=<unavailable>,*
*   mask@entry=<error reading variable: PC not available>) at ../../src/gdb/exceptions.c:546*
#11 <unavailable> in ?? ()
Backtrace stopped: not enough registers or memory available to unwind further
#tracepoint 2 收集了内核空间的栈，所以要切换回 *inferior 1* 装载内核调试信息。
(gdb) tfind
Found trace frame 1, tracepoint 2
#0 0xffffffff8117a3d0 in ?? ()
(gdb) inferior 1
[Switching to inferior 1 [Remote target] (/home/teawater/kernel/b/vmlinux)]
[Switching to thread 1 (Remote target)]
#0 vfs_read (file=0xffff88021a559500, buf=0x7fff08afd31f <Address 0x7fff08afd31f out of bounds>, count=1,
*   pos=0xffff8800c47e1f48) at /home/teawater/kernel/linux/fs/read_write.c:365*
365   {
#这是内核栈的 *backtrace*。
(gdb) bt
#0 vfs_read (file=0xffff88021a559500, buf=0x7fff08afd31f <Address 0x7fff08afd31f out of bounds>, count=1,
*   pos=0xffff8800c47e1f48) at /home/teawater/kernel/linux/fs/read_write.c:365*
#1 0xffffffff8117a59a in sys_read (fd=<optimized out>, buf=0x7fff08afd31f <Address 0x7fff08afd31f out of bounds>,
*   count=1) at /home/teawater/kernel/linux/fs/read_write.c:469*
#2 <signal handler called>
#3 0x00007f77331d7d10 in ?? ()
#4 0x0000000000000000 in ?? ()

如何使用 **add-ons/hotcode.py**

这个脚本可以通过记录并分析中断处理时候的取得的 PC 值从而得到 Linux kernel 或者用户层程序的热点代码。

请到 http://code.google.com/p/kgtp/wiki/hotcode 去看如何使用它。

如何增加用 C 写的插件

KGTP 支持用 C 写的插件，插件将被编译成 LKM。 KGTP support plugin that write in C. The plugin will be built as LKM

API

```
#include "gtp.h"
```

这是插件需要的包含 API 的头文件。

```
extern int gtp_plugin_mod_register(struct module *mod);
extern int gtp_plugin_mod_unregister(struct module *mod);
```

这两个函数注册和注销插件模块。这样 KGTP 就可以在访问插件模块资源的时候增加其的引用计数了。

```
extern struct gtp_var *gtp_plugin_var_add(char *name, int64_t val,
                                          struct gtp_var_hooks *hooks);
```

这个函数会增加特殊 trace 状态变量到 KGTP。

- **name** 特殊 trace 状态变量的名字.
- **val** 特殊 trace 状态变量的初始值.
- **hooks** 函数指针。如果这个功能不支持，函数指针就设置为 NULL。
- 返回值 成功返回 gtp_var 指针。失败则返回用 IS_ERR 和 PTR_ERR 可以处理的错误码。

```
struct gtp_var_hooks {
        int     (*gdb_set_val)(struct gtp_trace_s *unused, struct gtp_var
*var,
                               int64_t val);
        int     (*gdb_get_val)(struct gtp_trace_s *unused, struct gtp_var
*var,
                               int64_t *val);
        int     (*agent_set_val)(struct gtp_trace_s *gts, struct gtp_var
*var,
                                 int64_t val);
        int     (*agent_get_val)(struct gtp_trace_s *gts, struct gtp_var
*var,
                                 int64_t *val);
};
```

- **gdb_set_val** 在 GDB 设置 TSV 值的时候调用，请 注意 TSV 只能被 GDB 命令"tvariable $xxx=1"设置而且只有在 GDB 命令"tstart"的时候才会被发到 KGTP。
  - **unused** 是无用的，只用来让这个指针可以和 agent_set_val 共享函数。
  - **var** 是指向 gtp_var 的指针，于是当多个 TSV 共享一个函数的时候，这个值可以用来判定哪个 TSV 被访问了。
  - **val** 是 GDB 设置来的值。

    - 返回值 错误返回-1，正确返回 0。
- **gdb_get_val** 在 GDB 取 TSV 值的时候被调用。请 注意 取 TSV 值和设置 TSV 不同，设置任何时候都会直接从 KGTP 里取，并且取值的 GDB 命令和访问一个普通的 GDB 内部变量一样。例如："p $xxx"。
    - **unused** 和 gdb_set_val 作用相同。
    - **var** 和 gdb_set_val 作用相同。
    - **val** 用来返回值的指针。
    - 返回值 和 gdb_set_val 作用相同。
- **agent_set_val** 在 tracepoint action(teval_expr1,_expr2,_...)设置 TSV 的时候调用。
    - **gts** 是指向 tracepoint 会话结构的指针。
    - **var** 和 gdb_set_val 作用相同。
    - **val** action 设置的值。
    - 返回值 和 gdb_set_val 作用相同。
- **agent_get_val** will be called when tracepoint action(collect expr1, expr2, ...或者 teval expr1, expr2, ...) get the TSV.
    - **gts** 和 agent_set_val 作用相同。
    - **var** 和 gdb_set_val 作用相同。
    - **val** 和 gdb_get_val 作用相同。
    - 返回值 和 gdb_set_val 作用相同。

*extern int gtp_plugin_var_del(struct gtp_var *var);*
当 rmmod 插件模块的时候，用这个函数删除 gtp_plugin_var_add 增加的 TSV。

例子

KGTP 目录里的 plugin_example.c 是 KGTP plugin 的例子，可以用"make P=1"直接编译其。其将增加四个 TSV 到 KGTP 中。

- **$test1** 什么也不支持。
- **$test2** 支持被 GDB 或者 tracepoint action 读写。
- **$test3** 只支持 tracepoint action 写，当设置一个值到里面的时候，其将找到这个值对应的符号并打印出来。例如 "teval $test3=(int64_t)$rip"。
- **$test4** 只支持 tracepoint action 写，当设置值的时候其将打印当前 tracepoint 的地址的符号。

如何使用

- 安装 KGTP 模块 如何让 GDB 连接 KGTP
- insmod plugin_example.ko
- 让 GDB 连上 KGTP 并使用其。
- 断开 GDB. 如果 GDB 断开的时候不要停止 tracepoint 中的选项设置为打开，则设置其为关闭。
- rmmod plugin_example.ko

请 注意 KGTP 支持加入多个插件。

如何使用性能计数器

性能计数器是大部分现代 CPU 都有的特殊硬件寄存器。这些寄存器对一些硬件事件进行计数：例如指令执行数量，cachemisses 数量，分支预测失败数，而且这些计数不会让应用程序或者内核变慢。其还可以设置到达一定的值的时候发生中断，这些就可以用来分析在某 CPU 上执行程序的性能。

Linux 性能计数器子系统 perf event 可以用来取得性能计数器的值。你可以用 KGTP perf event trace 状态变量访问这些值。

请读内核目录里的 tools/perf/design.txt 文件取得 perf event 的更多信息。

定义一个 perf event trace 状态变量

访问一个性能计数器需要定义下面的 trace 状态变量：

"pe_cpu_"+tv_name	定义性能计数器的 *CPU ID*。
"pe_type_"+tv_name	定义性能计数器的类型。
"pe_config_"+tv_name	定义性能计数器的配置。
"pe_en_"+tv_name	定义性能计数器的启动开关。默认情况下性能计数器是关闭的。
"pe_val_"+tv_name	访问这个变量能取得性能计数器的值。

定义一个 per_cpu perf event trace 状态变量

定义一个 per_cpu perf event trace 状态变量和 Per_cpu trace 状态变量一样。

```
"p_pe_"+perf_event type+string+CPU_id
```

请 注意 如果定义一个 per_cpu perf event trace 状态变量，就不需要在定义 cpu id("pe_cpu")因为 KGTP 已经取得了 CPU 的 ID。

perf event 的类型和配置

类型可以是：

0 PERF_TYPE_HARDWARE
1 PERF_TYPE_SOFTWARE
2 PERF_TYPE_TRACEPOINT
3 PERF_TYPE_HW_CACHE
4 PERF_TYPE_RAW
5 PERF_TYPE_BREAKPOINT

如果类型是 0(PERF_TYPE_HARDWARE)，配置可以是：

0 PERF_COUNT_HW_CPU_CYCLES
1 PERF_COUNT_HW_INSTRUCTIONS
2 PERF_COUNT_HW_CACHE_REFERENCES
3 PERF_COUNT_HW_CACHE_MISSES
4 PERF_COUNT_HW_BRANCH_INSTRUCTIONS
5 PERF_COUNT_HW_BRANCH_MISSES
6 PERF_COUNT_HW_BUS_CYCLES
7 PERF_COUNT_HW_STALLED_CYCLES_FRONTEND
8 PERF_COUNT_HW_STALLED_CYCLES_BACKEND

如果类型是 3(PERF_TYPE_HW_CACHE)，配置要分为 3 部分： 第一部分是 cache id，其在设置进配置的时候需要 << 0：

0 PERF_COUNT_HW_CACHE_L1D
1 PERF_COUNT_HW_CACHE_L1I
2 PERF_COUNT_HW_CACHE_LL
3 PERF_COUNT_HW_CACHE_DTLB
4 PERF_COUNT_HW_CACHE_ITLB
5 PERF_COUNT_HW_CACHE_BPU

第二部分是 cache op id，其在设置进配置的时候需要 << 8：

0 PERF_COUNT_HW_CACHE_OP_READ
1 PERF_COUNT_HW_CACHE_OP_WRITE
2 PERF_COUNT_HW_CACHE_OP_PREFETCH

第三部分是 cache op result id，其在设置进配置的时候需要 << 16：

0 PERF_COUNT_HW_CACHE_RESULT_ACCESS
1 PERF_COUNT_HW_CACHE_RESULT_MISS

如果你想取得 PERF_COUNT_HW_CACHE_L1I(1), PERF_COUNT_HW_CACHE_OP_WRITE(1) and PERF_COUNT_HW_CACHE_RESULT_MISS(1)你需要使用：

(gdb) tvariable $pe_config_cache=1 | (1 << 8) | (1 << 16)

内核目录中的 tools/perf/design.txt 是关于 perf event 的类型和配置。

用$p_pe_en打开和关闭一个CPU上所有的perf event

我认为取得一段代码的性能计数器信息比较好的办法是在函数开头打开计数器在函数结束的时候关闭计数器。你可以用"pe_en"设置他们，但是如果你有多个perf event trace状态变量的时候，这样会让tracepoint action很大。$p_pe_en就是处理这种问题的。 你可以打开所有perf event trace状态变量在当前CPU上用下面的action：

```
>teval $p_pe_en=1
```

设置$p_pe_en为0来关闭他们。

```
>teval $p_pe_en=0
```

用来帮助设置和取得 perf event trace 状态变量的 GDB 脚本

下面这个 GDB 脚本定义了 2 个命令 dpe 和 spe 来帮助定义和显示 perf event trace 状态变量。

你可以把他们存在~/.gdbinit 或者你自己的 tracepoint 脚本中。于是你就可以在 GDB 中直接使用这 2 个命令。

```
define dpe
  if ($argc < 2)
    printf "Usage: dpe pe_type pe_config [enable]\n"
  end
  if ($argc >= 2)
    eval "tvariable $p_pe_val_%d%d_c",$arg0, $arg1
    eval "tvariable $p_pe_en_%d%d_c",$arg0, $arg1
    set $tmp=0
    while $tmp<$cpu_number
      eval "tvariable $p_pe_type_%d%d_c%d=%d",$arg0, $arg1, $tmp,
$arg0
      eval "tvariable $p_pe_config_%d%d_c%d=%d",$arg0, $arg1,
$tmp, $arg1
      eval "tvariable $p_pe_val_%d%d_c%d=0",$arg0, $arg1, $tmp
      if ($argc >= 3)
        eval "tvariable $p_pe_en_%d%d_c%d=%d",$arg0, $arg1, $tmp,
$arg2
      end
      set $tmp=$tmp+1
    end
  end
end

document dpe
Usage: dpe pe_type pe_config [enable]
end

define spe
  if ($argc != 2 && $argc != 3)
    printf "Usage: spe pe_type pe_config [cpu_id]\n"
  end
  if ($argc == 2)
    set $tmp=0
    while $tmp<$cpu_number
      eval "printf \"$p_pe_val_%%d%%d_c%%d=%%ld\\n\",$arg0,
$arg1, $tmp, $p_pe_val_%d%d_c%d", $arg0, $arg1, $tmp
      set $tmp=$tmp+1
    end
```

```
  end
  if ($argc == 3)
    eval "printf \"$p_pe_val_%%d%%d_c%%d=%%ld\\n\",$arg0, $arg1,
$tmp, $p_pe_val_%d%d_c%d", $arg0, $arg1, $arg2
  end
end

document spe
Usage: spe pe_type pe_config [cpu_id]
end
```

下面是一个取得函数 tcp_v4_rcv 性能计数器的例子：

```
#连接 KGTP
(gdb) target remote /sys/kernel/debug/gtp
#定义 3 个 perf event trace 状态变量
PERF_COUNT_HW_CPU_CYCLES，PERF_COUNT_HW_CACHE_MIS
SES 和 PERF_COUNT_HW_BRANCH_MISSES。
(gdb) dpe 0 0
(gdb) dpe 0 3
(gdb) dpe 0 5
#在函数开头打开这个 CPU 的性能寄存器
(gdb) trace tcp_v4_rcv
(gdb) action
>teval $p_pe_en=1
>end
#$kret 让我们可以处理到函数 tcp_v4_rcv 的结尾：
(gdb) trace *(tcp_v4_rcv)
(gdb) action
>teval $kret=0
#关闭这个 CPU 上的所有性能计数器
>teval $p_pe_en=0
#访问这些 perf event trace 状态变量将取得他们的值
>collect $p_pe_val_00_0
>collect $p_pe_val_03_0
>collect $p_pe_val_05_0
#设置这些 perf event trace 状态变量为 0
>teval $p_pe_val_00_0=0
>teval $p_pe_val_03_0=0
>teval $p_pe_val_05_0=0
>end
tstart
#等一会让每个 CPU 收一些 TCP 包
(gdb) tstop
(gdb) tfind
(gdb) spe 0 0 $cpu_id
$p_pe_val_00_2=12676
(gdb) spe 0 3 $cpu_id
```

$p_pe_val_03_2=7
(gdb) spe 0 5 $cpu_id
$p_pe_val_05_2=97