

# linux内核调试指南

转载 2016年05月14日 14:45:20

- 3171

大海里的鱼有很多，而我们需要的是鱼钩一只

一些前言

作者前言

知识从哪里来

为什么撰写本文档

为什么需要汇编级调试

\*\*\*第一部分：基础知识\*\*\*

总纲：内核世界的陷阱

源码阅读的陷阱

代码调试的陷阱

原理理解的陷阱

建立调试环境

发行版的选择和安装

为什么选debian

debian与ubuntu

从0安装debian

debian重要命令

中文环境设置

debian的键盘设置更改

英文Locale下使用中文输入法

pdf乱码的解决

建立编译环境

安装交叉编译工具

交叉编译工具下载网址

安装arm-linux-gnueabi-XXX 工具集

什么是EABI

[安装arm-elf-XXX 工具集](#)

[bin工具集的使用](#)

[arm-linux-gnueabi-gcc](#)

[arm-linux-gnueabi-gdb](#)

[qemu的使用](#)

[initrd.img的原理与制作](#)

[安装与使用](#)

[x86虚拟调试环境的建立](#)

[基于qemu和内核内置kgdb](#)

[基于qemu和qemu内置gdbstub](#)

[arm虚拟调试环境的建立](#)

[利用qemu](#)

[利用qemu安装debian linux](#)

[利用qemu安装能进行内核调试的系统](#)

[利用skyeye](#)

[skyeye虚拟机的内核调试](#)

[skyeye的安装与使用](#)

[快速试玩](#)

[快速配置能调试的环境](#)

[为s3c2410配置2.6.26内核](#)

[使用最新的skyeye](#)

[arm开发板调试环境的建立](#)

[基于串口](#)

[基于网口](#)

[gdb基础](#)

[基本命令](#)

[gdb之gui](#)

[gdb技巧](#)

[gdb宏](#)

[参考资料](#)

[gdb宏的使用](#)

实例

链表遍历类

功能增强类

汇编基础--X86篇

用户手册

AT&T汇编格式

内联汇编

汇编与C函数的相互调用

调用链形成和参数传递

寄存器的角色与保护

调用链的形成

栈帧结构与参数传递

完整的调用过程

调用链回溯的代码实现

C难点的汇编解释

例1

例2

例3

例4

优化级别的影响

优化选项

例子

汇编基础--ARM篇

用户手册

调用链形成和参数传递

壮观的标准

别名的烦恼

寄存器的角色与保护

条件执行

调用链的形成

栈帧结构与参数传递

完整的调用过程

调用链回溯的实现

源码浏览工具

调用图生成工具

find + grep

wine + SI

优缺点

安装wine

安装SI

SI的设置

SI的使用

global

Source-Navigator

vim + cscope/ctags

优缺点

安装cscope/ctags

命令选项

使用

建立索引

利用vim浏览源码

快捷键的使用

kscope

lxr

SI等与gdb的特点

调用链、调用树和调用图

理想调用链

函数指针调用

调用链的层次

非理想调用链

调用树与调用图

调用树的定义

调用树的作用

调用树的分类

调用树的显示

调用树的拼接

调用图

穿越盲区

穿越gdb的盲区

进程切换

中断异常

系统调用

穿越交叉索引工具的盲区

函数指针

查看函数的参数

工程方法

二叉断点

给调用指令下断点

绕过时钟中断的干扰

bug 与 OOPS

网站

\*\*\*第二部分：内核分析\*\*\*

调试相关子系统

kgdb源码分析

sysrq

oprofile

kprobes

驱动分析

载入模块符号

seq\_file.c的分析

module.c的分析

中断处理过程

s3c24xx内存初始化分析

虚拟地址空间

用户层的观察窗

交互，从内核层分析

理解设备模型

面向对象的实现

设备模型的分层

外围支持机制

sysfs

hotplug

文件系统

\*\*\*第三部分：其他工具\*\*\*

strace

ltrace

SystemTap

MEMWATCH

YAMD

Magic SysRq

附录：社区交流相关

补丁提交相关文档

补丁制作与提交示范

多补丁发送工具

git使用

附录：内核参考书籍文章

内核git库

书籍

子系统官方网站

参考文章

私人备忘

**大海里的鱼有很多，而我们需要的是鱼钩一只**

本文档由大家一起自由编写，修改和扩充，sniper负责维护。引用外来的文章要注明作者和来处。本文档所有命令都是在ubuntu/debian下的操作。选取的内核源码从文档开始编写时最新的内核版本-2.6.26开始，而且会随着linux的更新而不断更换新的版本。所以文档的内容可能前后不一致。相信大家有能力克服这个问题。

本文档的字符图示在linux环境下显示正常，在window下显示有细微的错乱。

本文档唯一的更新网址是：<http://wiki.zh-kernel.org/sniper> 转载请保留此网址。

有任何建议请发邮件：[s3c24xx@gmail.com](mailto:s3c24xx@gmail.com)

有任何问题请到邮件列表提问：<http://zh-kernel.org/mailman/listinfo/linux-kernel>

一些和内核调试分析有关的小工具放在：

<http://code.google.com/p/root-kit/>

## **一些前言**

### **作者前言**

一个人默默地敲打这篇文章也有段时间了。在这个过程中，没有收到任何的赞誉，也没接到任何的板砖，没有任何的反馈。就这么敲打着，修理着。但是本人从没怀疑这篇文档的价值，这是因为，本人就是这篇文档的亲身收益者。在这里把它“无私”奉献出来，乃是出于对于某类同道者锲而不舍孜孜以求的“德性”的认同和“同情”，你的痛苦我表示感同身受，你的迷茫我愿意一起分担。一定有人能从个文档受益，这便已让我知足。其实，写这个文档并非是件苦差，而是字字都是有感而发的，不吐不快的结果。这里的句句都是本人教训和经验的记录。

谈到调试器，世上存在两种截然不同的看法。其中一种，是超级解霸的作者，他认为“程序不是写出来的，好程序绝对是调试出来的”。对于这个观点，本人持着极不认同的态度。而第二种相反观点的人，便是linux之父linus了。他认为调试器只会“误人子弟”，只会导致人们迷于表象而不去真正理解源码本身。并以此为由，长期没把kgdb内置到内核中。对于调试器调试bug会引入错误的修正这个观点，我认为还是有点道理的。但是他以此为由而不把它集合到内核中，这个做法我就认为是毫无道理了。因为linus本人就说过：“我只使用GDB，而且我总是并不把它作为调试器来使用，只是将其作为一个可以用来分析程序的分解器来使用。”既然他可以这样做，为什么就认定他人使用gdb的目的就一定就是用来调试bug而不是另有所用呢？本人之所以这样说，这是因为本人正也是使用gdb主要是用来辅助分析内核代码而不是主要用来调试错误的。这也正就是本文的主题。

世上从不缺少解决问题的答案，缺少的是解决问题的方法。现在，linux的世界里已经不缺牛书了，将尽一千页一本的满载答案的砖头书接踵而来，但是渐渐地发现，看书看到后面就忘了前面，回到前面有忘了后面，甚至一个章节还没看完，那个子系统已经被完全重写了。慢慢地，就会怀疑“我是不是真的变老了？真的不行了？”但是我们从没想过：“凭什么我们就如此受制于人？他就能搞懂，而我就不行呢？”。其实，我们需要的是一种重其意而忘其形的根本之道，需要的是一种兵来将挡，火来水淹的通用解决方法。而绝不是淹没于牛人们的结论中。否则，遇到一个新的问题，就只能埋怨牛人的书还不够厚，以至于没把你需要的东西也包括进去了。牛人一定有一套牛方法，而他在书中不详说，我不认为是他故意“留一手”，而是认为这是对自身觉得习以为常的事物的一种疏忽。牛人的研究结果其实不是最重要的，他的研究方法和手段才是最重要的事情。而我，也渐渐地发现，调试器能带给我们很多有用的提示，使得我们能不断的寻找到思考的灵感和方向，也使得学习变得非常的有趣性和有目的性。我想，利用调试器辅助源码分析，是不是正是很多牛人正在做的而没有说出来的事情呢？无论答案如何，本人还是觉得，调试器是个好东西，不要轻易把它搁置在一旁。虽然很多高人也许已经是深安此道，甚至已经不需要它的提示了，但是它依然有益于我等功力尚浅的人。把这种经验和技巧记录下来，让需要这项技巧的人少化时间去摸索，这绝对不是一件坏事。

正是因为这个原因，随着文档慢慢地变大，也更加的觉得文档的题目起得有点不恰当了，题目起作“内核动态分析指南”更恰当点。文档的主旨是利用调试器动态分析内核，调试错误只是这个过程的副产品罢了。不过，这个新的名字实在是不够现在名字“刺眼”，所以也就没有启用它。



说了这么多的废话和出格的话，无非是有两个目的：这个文章慢慢的变得这么长了，如果没有半句的“人”话，没有半句的现实世界中的语句。那估计本人不是变成了机器人，阅读的人也会变成了机器人。顺便借这段文字交交朋友。另一个目的呢，是说不应拘束于工具，工具是死的，人是活的。如果某些工具确能带给我们某些有益的提示，我们就可以去尝试它，取起优点而舍其糟粕。

引用的原文：

Linus 谈调试器和内核如何发展：<http://www.bitcn.com/linux/kernel/200604/7493.html>

## 知识从哪里来

### 1. 永远不要忘记的三大帮助命令

- XXX -h(XXX -help)
- man -a XXX
- info XXX

### 2. 如何安装帮助文档

- \$ sudo synaptic 界面出来后，在“组别” -> “文档”选取你要的文档进行安装
- 或\$ apt-cache search Documentation | grep XXX 搜索需要的文档进行安装

### 3. 从软件/工具的官方网站阅读/下载文档

### 4. 从irc获取帮助 irc.freenode.net

### 5. 从邮件列表获取帮助 mailist <http://lkml.org/> <http://marc.info/>

### 6. 发行版社区文档或社区 <https://help.ubuntu.com/community/> <http://wiki.ubuntu.org.cn/>

### 7. 利用google搜索文档或阅读他人文章

### 8. 利用google搜索lkml

[http://www.google.cn/advanced\\_search?hl=zh-CN](http://www.google.cn/advanced_search?hl=zh-CN) 网域那里填上lkml.org

## 9. 获取内核文档

- 源码本身
- 源码中的注释
- 内核源码附带的文档 Documentation
- 相关的教科书
- 论文 免费论文引擎 <http://citeseerx.ist.psu.edu/>
- 内核子系统的官方网站
- 获取内核源码目录Documentation/DocBook/ 下已经编译好的书籍

找到最新版本的文档

```
$ apt-cache search linux-doc
```

安装最新的文档

```
$ sudo apt-get install linux-doc-2.6.24
```

阅读Documentation/DocBook/ 下已经编译好的书籍（html格式）

```
$ firefox /usr/share/doc/linux-doc-2.6.24/html/index.html
```

## 10. 买书

## 11. 书籍最后面的参考书目

## 12. 文章末尾的参考文章

## 13. 电子书搜索网站

emule: 只要知道书名, windows下用emule基本可以找到所有的英文版电子书。

但在linux不行, 可能是我的设置问题。

<http://rapidshare.com/index.html>

<http://www.netbks.com/>

## 为什么撰写本文档

todo:学习方法, 学习曲线, 参考书籍的特点和不足, 本文档的任务

内核学习曲线

### 1.只读书不看源码

参考书籍：Linux Kernel Development

## 2.参考源码读书(读书为主)

参考书籍：understanding the linux kernel

## 3.参考书读源码(看源码为主)

参考书籍：情景分析

## 4.只看源码不/少读书(提交补丁为主)

参考：lkml,main-tree, mm-tree

linux内核分析方法：

按分析的对象分：

- 1.代码： 分析的对象是源代码
- 2.数据： 分析的对象是内核运行时产生的数据

按观察对象的状态分：

- 1.静态： 观察的目标对象是静止不动的
- 2.动态： 观察的目标对象是动态变化的

所以综合地看，分析方法的种类有：

### 1.静态代码：

最原始的方式，阅读源代码

### 2.动态代码：

利用某些工具或手段，动态分析源代码。又分为

- a. 利用lxr, cscope, source insight等工具交叉索引源代码

b. 利用git,web-git通过阅读增量patch等形式观察源码的进化

c. 利用调试器跟随内核的运行动态观察内核正在运行的代码片段

### 3.静态数据：

观察的对象是内核在运行时产生或收集汇总出来的数据。又分为

a. 代码中printk语句打印出来的内核信息

b. 系统出错产生的oops,panic信息

c. 借助systemtap等类似工具提取的内核数据汇总

### 4.动态数据：

借助内核调试器实时观察内核不断产生的数据

可见内核调试器是最强大的内核分析工具，但它也不是“全功能”的工具。

1. 主要地，本文档聚焦于描述如何利用gdb对内核进行源码级别和汇编级别的观察和调试。

而这种调试的目的有两个：

- 确定bug产生的引入点。这部分内容放于本文档第一部分。
- 配合源码阅读工具(source insight,kscope等)，观察内核实时运行的状况，观察内核数据的产生和变化，以及观察各个函数的动态调用关系，从而以一种精确的动态的和验证性的方式来理解内核运作的原理。这部分内容放于本文档第二部分

前者是调试器应用的主要价值，而后者却是本文档的兴趣所在。

2. 因为需要观察用户层和内核层的交互，演示调试工具的全面功能等原因，本文档内容不完全局限于内核层。

3. 另外，为了提供内核调试知识的全面叙述，我们对其他调试工具，其他调试的问题比如检测内存泄露等内容，也会进行说明。此部分内容放于本文档的第三部分。

## 为什么需要汇编级调试

- 逆向工程的需要

例子1: NT 内核的进程调度分析笔记 <http://www.whitecell.org/list.php?id=11>

例子2: NT 下动态切换进程分析笔记 <http://www.whitecell.org/list.php?id=13>

在windows的世界里，内核源码和具体原理是不公开的。但很多牛人就凭一个破烂调试器阅读反汇编代码就能得到内部真相，可见调试器汇编级调试威力之大。但是在linux是源码公开的情况下，就没必要干那样的辛苦活了。但是因为以下原因，汇编级调试还是必要的。

- 汇编比C语言更低层

有时(比如代码优化)情况下，因为C代码经过了编译器的处理，调试器在c源码调试这个级别下给出的信息是无法理解的，甚至看起来是错误的。但是如果直接对调试器给出的反汇编代码进行分析，就不会受到那类问题的束缚。也就是说，进行汇编级别的调试能最大程度的利用调试器的功能。

- 汇编是C语义的解释

当你对某句C语言不是很理解时，看看编译器是怎么想的，是个很不错的办法。

- 能锻炼汇编源码的阅读能力

另一方面，内核中本来存在很多汇编源代码，进行汇编级调试也是锻炼阅读汇编源码能力的最有效方法。

当然，汇编级调试虽然强大，但代价也是很昂贵。和源码级调试相比，分析汇编代码花的时间要多上几十倍。所以，在源码公开的情况下，应该以源码级调试为主，特殊情况下才需要汇编级调试。

## \*\*\*第一部分：基础知识\*\*\*

## 总纲：内核世界的陷阱

也是阅读理解其他任何大型代码会遇到的问题。下面各节的内容都是围绕这些小项展开的。如果有的内容不知所云，先看后面内容，再回头看这里。

[先从其他地方复制过来，等待充实]

### 源码阅读的陷阱

源码不但是越来越大，更是越来越“刁”了。“刁”到了就是借助源码交叉索引工具也有它索引不到的地方。所以目前，即使是从源码阅读的角度而不是从调试的角度，只利用阅读工具不借助调试工具的话，源码都无法阅读。

源码“刁”到源码解析工具都无法解析的因素有：

1. 汇编源码包括内嵌汇编 可能无法被你的源码阅读工具所解析
2. 汇编代码和C代码之间的调用关系 无法被被源码阅读工具解析
3. 利用函数指针的函数调用 无法被被源码阅读工具解析
4. 宏“假函数” 可能无法被被源码阅读工具解析(SI不能解析，lxr能)

比如page\_buffers()。定义是：

```
#define page_buffers(page) \
({ \
    BUG_ON(!PagePrivate(page)); \
    ((struct buffer_head *)page_private(page)); \
})
```

5. 利用宏在编译时动态生成的函数体 无法被被源码阅读工具解析

比如fs/buffer.c中有一大批类似函数。比如buffer\_unwritten()

定义在buffer\_head.h

```
82 #define BUFFER_FNS(bit, name) \
..省略 \
91 static inline int buffer_##name(const struct buffer_head *bh) \
92 { \
93     return test_bit(BH_##bit, &(bh->b_state)); \
94 } \
..
```

130 `BUFFER_FNS(Unwritten, unwritten)`

这类函数一般是短小的内嵌函数，用gdb调试时都看不出来。只能靠字符搜索再加上一点机灵。

## 6. 函数/变量的某类c扩展属性标记， 可能导致该函数/变量无法被被源码阅读工具解析

比如`static struct vfsmount *bd_mnt __read_mostly;`中的`bd_mnt`

## 7. 其他语种的保留关键字，可能无法被你的源码阅读工具所解析

如默认配置的SI无法解析`struct class`，当然，这个问题和内核无关。

但是借助调试器，就能直接而轻易地解决上述源码解析工具难以解决的问题。

## 代码调试的陷阱

### 搭建调试环境

### gdb调试器的陷阱

#### 1. 宏“假函数”

#### 2. 内嵌函数

#### 3. 代码优化

#### 4. 汇编码

#### 5. 进程切换

#### 6. 中断处理

#### 7. 系统调用

## 原理理解的陷阱

### 0. 链接器脚本和make语法

下面这些杂七杂八的文件对内核整体原理的理解起着决定性的作用。

内核中的链接脚本

```
linux-2.6$ find ./ -name "*lds*"
```

内核中的重要宏文件

```
module_param* macros
```

```
include/linux/moduleparam.h
```

```
*__initcall Macros
```

```
include/linux/init.h
```

内核中的汇编文件

```
linux-2.6$ find ./ -name "*.S"
```

内核中的Makefile

```
linux-2.6$ find ./ -name "Makefile"
```

内核中的配置文件

```
linux-2.6$ find ./ -name "*config*"
```

1. C与汇编代码的相互调用
2. 各子系统间的接口互动
3. 内核的设计思想及其代码编写和运行形式

a) 基于对象的思想

例子：文件系统，设备模型

b) “发布—订阅”模型

例子：notification chain

## 建立调试环境

### 发行版的选择和安装

#### 为什么选debian

[如题] <http://www.debian.org/> <http://www.emdebian.org/>

为什么本人选择debian？ 因为：引用内容来之[www.debian.org](http://www.debian.org/)



“Debian 计划 是一个致力于创建一个自由操作系统的合作组织。...屁话省略...屁话..  
N多屁话之后: 当然, 人们真正需要的是应用软件, 也就是帮助他们完成工作的程序:  
从文档编辑, 到电子商务, 到游戏娱乐, 到软件开发。Debian 带来了超过 18733 个  
软件包 (为了能在您的机器上轻松的安装, 这些软件包都已经被编译包装为一种方便  
的格式) — 这些全部都是 自由 软件。”

原因终于看到了, 选择debian是因为本人比较懒, 比较笨。而debian正好迎合了我  
这种人的需求。

1. 它”带来了超过 18733 个 软件包”。18733这个数目非常不直观, 而且或许是N年  
前的数据了。我们可以到debian的ftp看看, 现在它可供安装的软件 and 工具达到了5个  
DVD的容量。难以想象, 在这5个DVD容量的工具库中, 还会找不到我所想要的东  
西。

2. debian有一个非常出名的安装包管理机制。你需要做的就是, 打开“立新得”软件,  
然后在一个小方框里写上你需要东西的相关信息, 然后再点点一个叫做“ 搜索”的小方  
块。接着, debian就会在它5个DVD大的工具库中寻找你想要的工具。在结果返回  
后, 选择好你的工具, 再点点一个叫做“应用”的小方块, 过一会, 就可以使用你的工  
具了。

再也没有了“缺少什么什么包”的烦人提示了, 一切都这么简单, 又这么强大。这, 正  
是我想要的。

## debian与ubuntu

[两者区别, 版本外号, 支持社区, source list等] 1. ubuntu的易用性比debian要好。  
尤其是中文支持, 还有ubuntu国内有活跃的社区。 2. 虽然ubuntu是基于debian的,  
apt 软件库也能获取到debian的软件, 但它毕竟是不同的系统环境, 理念不同, 对于  
一些偏门或太旧或太新的软件时, ubuntu往往不支持, 安装不了。比 如, gcc-3.4-ar  
m-linux-gnu这个包, 发行时间已久, ubuntu下安装不了, 但在debian下则可以。<http://www.ubuntu.com/community/ubuntustory/debian>

如不特别说明, 本文档所有命令都是在ubuntu Hardy Heron8.04版本 和debian testi  
ng版本下的操作。

## 从0安装debian

[如果想领教古典linux相对于windows的特色，请安装一次debian吧。尽管和以前比，已经很智能了。但安装了debian，选了中文环境，发现汉字都是歪歪倒倒的。而且没有汉字输入法，装了汉字输入法后，却用不了。不知道是我笨还是程序有bug。所以不得不用英文写下本烂文，怕把安装过程给忘了。需要翻译回中文]

How to install and configure a debian system from zero

### 1.install the system with one CD

Download CD iso file from debian official website, and burn it into a CD. Note that, we can just download the first CD iso but not DVDs or the whole series of CDs, because the first CD has already contained all the basis components of debian system and many other most common applications. We can use the first CD to install debian system, and then to install some other needed programs from it if needed. In this way, you can save much time spent on touching many inrelated things.

### 2.install application & tool from CD

you can install some common applications from the CD with the following command: `apt-get install expected-application`. Why can we do that without any more configuration? Why is it not need to has a ability to access internet? Well, Let's look at the file named `source.list` which identifying where to get software's package?? `deb cdrom:[Debian GNU/Linux testing _Lenny_ - Official Snapshot i386 CD Binary -1 20080605-15:01]/ lenny main` It means that system try to get something from your CD, so obviously that you can get some the most common but not all the tools available in debian official application repository.

### 3.try to access the internet

Thank to the first CD, we can do that easily. First, install the tool `ppp` contained in CD and its' configuration tool `pppoeconfig`. All these steps are described in file `ADSL (PPPOE) 接入指南.txt`

#### 4.search any useful information through the internet

now, we have built a base debian system, but it is too simple. I want to do some some thing, for example, to chat with some other people with pidgin, but it is not contained in the first CD, which just downloaded by you. And you may want to search some helps with google,etc. Just to do it, google is a most useful tool.

#### 5.search the internet updating source

I think you have get much thing through the google. But the most important thing is to get a available update source for your system, and change the source.list—that is /etc/apt/source.list. Now, I have got a good one, and it seems good. Don't forget to turn on the security entry in the orgion file source.list. That file looks like following after my updataion:

```
#deb cdrom:[Debian GNU/Linux testing _Lenny_ - Official Snapshot i386 CD Binary-1 :  
deb http://ftp.debian.org/debian/ lenny main contrib non-free  
deb http://security.debian.org/ lenny/updates main  
deb-src http://security.debian.org/ lenny/updates main
```



You should note that the internet address is debian office's, but It takes some while to get it. And my searching tool is google. :) Oh, we shoul run a command to update the new configuration to system before using it, don't ferget: apt-get update

#### 6.get help from IRC

Well, we have already been able to get some applications or tools from internet with command apte-get or wget,etc.. But I think the first thing to do is to get and install a very valuable tool named pidgin which can bring you into IRC world. Because Many experiance and kind person live in channel #debian of irc.freenode.net. You can get help from it very quickly. How to configure pidgin? Sorry, I don't like to answer such a problem , please just to google it or try it by yourself. I am not so kind as some guys living in IRC : )

#### 7.get and install synaptic

If you ever used ubuntu, you should agree that synaptic is good tool to update your system. It can save you much time of searching tools, typing commands, or managing the downloaded tools. But Unfortunately, such an important tool is not installed in the default system, and it is not contained in the first CD. So, we can just to get it with command “apt-get install synaptic”. After doing that successfully, I don’t want to type that command anymore. It’s so tedious to me.

8.get more tools with the help of synaptic

synaptic is my GOD in the linux world. Without it, I will become crazy. But now, I have owned it, so I can fly very freely in the internet sky. Just to search any tools and to update your system. And now, the CD used to install debian can be discarded, if you will never reinstall or rescue the system with it in future.

Now, the sun has rise up, and you have found the road to reback to civilization. Why? Just to ask your google and synaptic. :)

#### **debian重要命令**

[来源] 《APT and Dpkg 快速参考表》 [http://i18n.linux.net.cn/others/APT\\_and\\_Dpkg.php](http://i18n.linux.net.cn/others/APT_and_Dpkg.php)

Apt 不止是 apt-get

<http://www.erwinwang.com/node/10>

#### **中文环境设置**

##### **debian的键盘设置更改**

默认安装的debian，键盘的设置可能有问题。比如“|”打不出来。值得一提的是，这个设置甚至是和qemu的monitor模式相关联的。也就是说，qemu下有的字符也打不出来。如果有这个问题，按下面步骤设置

System→Preferences→Keyboard→Layouts

然后通过“Add”增加China,并设置它为默认，或者同时把其他的删除掉。

## 英文Locale下使用中文输入法

说明，中文环境比英文环境有很多缺点。比如编译时编译器的提示都给汉化了，有如，minicom的中文汉化界面是错乱的，而且minicom无法设置。本人一般是英文环境 + 中文输入法。先安装好好中文环境，系统中就有了中文输入法和其他一些和中文有关的东西。然后转到英文环境下，按照下面做法更改scim 的配置文件即可。

来自：<http://wiki.ubuntu.org.cn/index.php?title=%E8%8B%B1%E6%96%87Local%E4%B8%8B%E4%BD%BF%E7%94%A8%E4%B8%AD%E6%96%87%E8%BE%E9%93%E5%85%A5%E6%B3%95&variant=zh-cn>

编辑 /etc/gtk-2.0/gtk.immodules(如果存在的话) 或者 /usr/lib/gtk-2.0/2.10.0/immodule-files.d/libgtk2.0-0.immodules 文件，在xim 的 local 增加 en 也就是说：

```
"xim" "X Input Method" "gtk20" "/usr/share/locale" "ko:ja:th:zh"
```

改成：

```
"xim" "X Input Method" "gtk20" "/usr/share/locale" "en:ko:ja:th:zh"
```

注意，一定要重启一下机器。

## pdf乱码的解决

```
$sudo apt-get install xpdf-chinese-simplified xpdf-chinese-traditional poppler-data
```

<

>

参考：

<http://wiki.ubuntu.org.cn/PDF%E6%96%87%E6%A1%A3%E7%9A%84%E4%B9%B1%E7%A0%81%E9%97%AE%E9%A2%98>

## 建立编译环境

```
$ sudo apt-get install build-essential autoconf automake1.9 cvs subversion libncurses
```

<

>

其余的根据出错的提示，利用“立新得”搜索，然后进行安装。没有“立新得”界面程序的可以在终端下利用以下命令来搜索和安装。

```
$ sudo apt-get update
$ apt-cache search XXX
$ sudo apt-get install XXX
```

## 双硬盘系统切换设置的grub设置， 私人备忘用

```
title          Microsoft Windows XP Professional
root           (hd1,0)
savedefault
makeactive
map            (hd0) (hd1)
map            (hd1) (hd0)
chainloader    +1
```

## 安装交叉编译工具

### 交叉编译工具下载网址

下面是几个交叉编译工具下载网址，需要手动安装时，对比一下编译器的名称可以找到合适的下载地址。debian维护有自己的已经打包成.deb形式安装包，在debian软件库中。

[http://www.codesourcery.com/gnu\\_toolchains/arm/download.html](http://www.codesourcery.com/gnu_toolchains/arm/download.html)  
(据说是arm公司推荐的)

Download Sourcery G++ Lite Edition for ARM

Target OS	Download
EABI	Sourcery G++ Lite 2008q1-126 All versions...
uClinux	Sourcery G++ Lite 2008q1-152 All versions...
GNU/Linux	Sourcery G++ Lite 2008q1-126 All versions...
SymbianOS	Sourcery G++ Lite 2008q1-126 All versions...

到底是选EABI还是GNU/LINUX呢？应该是后者....

点GNU/LINUX的连接进去，可看到

Download	MD5 Checksum
IA32 GNU/Linux Installer	93eee13a08dd739811cd9b9b3e2b3212
IA32 Windows Installer	fac5b0cee1d9639c9f15e018e6d272ad

Documentation

Title	Format
Assembler (PDF)	PDF
Binary Utilities (PDF)	PDF
C Library (GLIBC) (PDF)	PDF
Compiler (PDF)	PDF
Debugger (PDF)	PDF

Getting Started Guide (PDF)	PDF
Linker (PDF)	PDF
Preprocessor (PDF)	PDF
Profiler (PDF)	PDF

## Advanced Packages

Expert users may prefer packages in these formats.

Download	MD5 Checksum
IA32 GNU/Linux TAR	4f11b0fa881864f220ab1bd84666108b
IA32 Windows TAR	ed6d25fd68301e728a1fba4cd5cb913f
Source TAR	2db28fb2aa80134e7d34d42b7039d866

名字标识不是很明显，进去看才知道。比如，IA32 GNU/Linux Installer对应的安装包名字叫arm-2008q1-126-arm-none-linux-gnueabi.bin  
为什么有个none? 迷茫中..

-----  
<http://ftp.snapgear.org:9981/pub/snapgear/tools/arm-linux/>

[DIR] Parent Directory	30-Sep-2003 15:44	-
[ ] arm-linux-tools-20031127.tar.gz	26-Nov-2007 16:56	141M
[ ] arm-linux-tools-20051123.tar.gz	24-Nov-2005 00:50	228M
[ ] arm-linux-tools-20061213.tar.gz	13-Dec-2006 13:31	230M
[ ] arm-linux-tools-20070808.tar.gz	30-Nov-2007 03:21	271M
[ ] binutils-2.16.tar.gz	16-Nov-2005 15:44	15.6M
[ ] binutils-2.17.tar.gz	06-Dec-2007 10:24	17.4M
[ ] build-arm-linux-3.4.4	02-Aug-2006 14:32	6k
[ ] build-arm-linux-4.2.1	30-Jul-2008 10:13	7k
[ ] elf2flt-20060707.tar.gz	17-Jan-2008 22:23	101k
[ ] elf2flt-20060708.tar.gz	30-Jul-2008 10:14	110k
[ ] gcc-3.4.4.tar.bz2	16-Nov-2005 15:39	26.3M
[ ] gcc-4.2.1.tar.bz2	06-Dec-2007 10:11	42.0M
[ ] genext2fs-1.3.tar.gz	03-Sep-2003 10:23	19k
[ ] glibc-2.3.3.tar.gz	16-Nov-2005 15:49	16.7M
[ ] glibc-2.3.6.tar.gz	06-Dec-2007 10:39	17.9M
[ ] glibc-linuxthreads-2.3.3.tar.gz	16-Nov-2005 15:49	303k
[ ] glibc-linuxthreads-2.3.6.tar.gz	06-Dec-2007 10:39	320k

-----  
<http://www.handhelds.org/download/projects/toolchain/>

[DIR] Parent Directory		-
[ ] README	28-Jul-2004 17:37	788
[DIR] archive/	28-Jul-2004 17:34	-
[ ] arm-linux-gcc-3.3.2.tar.bz2	03-Nov-2003 10:23	71M
[ ] arm-linux-gcc-3.4.1.tar.bz2	29-Jul-2004 14:01	41M
[DIR] beta/	28-Jul-2004 17:36	-
[ ] crosstool-0.27-gcc3.4.1.tar.gz	28-Jul-2004 17:21	2.0M
[ ] gcc-build-cross-3.3	31-Oct-2003 15:43	5.1K
[DIR] jacques/	24-Jul-2001 18:45	-
[ ] kernel-headers-sa-2.4.19-rmk6-pxa1-hh5.tar.gz	12-Mar-2003 17:42	4.7M
[DIR] monmotha/	13-Aug-2002 17:54	-
[DIR] osx/	14-Dec-2003 11:45	-
[DIR] pb/	22-Nov-2002 20:10	-
[DIR] source/	18-Mar-2004 16:12	-

-----

```

http://ftp.arm.linux.org.uk/pub/armlinux/toolchain/
[DIR] Parent Directory -
[ ] Oerlikon-DevKit-XScalev2.tar.gz 07-Feb-2003 22:30 3.7K
[ ] cross-2.95.3.tar.bz2 20-Jul-2001 21:12 35M
[ ] cross-3.0.tar.bz2 20-Jul-2001 22:27 39M
[ ] cross-3.2.tar.bz2 23-Aug-2002 11:04 81M
[ ] cross-3.2.tar.gz 23-Aug-2002 10:01 93M
[DIR] src-2.95.3/ 14-Jan-2002 17:52 -
[DIR] src-3.2/ 23-Aug-2002 10:53 -
-----
http://linux.omap.com/pub/toolchain/
[DIR] Parent Directory -
[ ] obsolete-gcc-3.3.2.t.> 15-May-2004 12:18 76M
-----
http://www.uclinux.org/pub/uClinux/arm-elf-tools/
To install the Linux binaries, login as root and run "sh ./XXX-elf-tools-20030314.s

m68k-elf-20030314/arm-elf-20030314
    Get the m68k binaries or the ARM binaries. The source is here.

m68k-elf-20020410/arm-elf-20011219
    Get the m68k binaries or the ARM binaries. The source is here.

m68k-elf-20020218/arm-elf-20011219
    Get the m68k binaries or the ARM binaries. The source is here.

m68k/arm-elf-20011219
    Get the m68k binaries or the ARM binaries. The source is here.

    You can also get Bernhard Kuhn's RPMs here.

m68k-elf-20010716
    Get the binaries here and the source from here.

m68k-elf-20010712
    Get the binaries here and the source from here.

m68k-elf-20010610
    Get the binaries here and the source from here.

m68k-elf-20010228
    The binaries are in two files, the compilers and the g++ headers. The source is
<

```

## 安装arm-linux-gnueabi-XXX 工具集

debian有自己维护的一套交叉编译工具集

[参考]<http://www.emdebian.org/tools/crosstools.html>

工具库: <http://www.emdebian.org/debian/pool/main/>



步骤：

## 1. 往/etc/apt/sources.list文件加入下面软件源

```
deb http://buildd.emdebian.org/debian/ unstable main
deb-src http://buildd.emdebian.org/debian/ unstable main
deb http://buildd.emdebian.org/debian/ testing main
deb-src http://buildd.emdebian.org/debian/ testing main
```

然后：

```
安装 emdebian-archive-keyring package
$ sudo apt-get install emdebian-archive-keyring
更新
$ sudo apt-get update
```

## 2. 安装交叉编译器

```
$ sudo apt-get install libc6-armel-cross libc6-dev-armel-cross binutils-arm-linux-  
< >
```

注意，在ubuntu8.04下，只能安装4.2版。把上面文字中的4.3全部换为4.2即可。

## 3. 安装交叉调试器

```
$sudo apt-get install gdb-arm-linux-gnueabi
```

注意：

a. 安装时使用名称：gdb-arm-linux-gnueabi，调用时使用命令名是：arm-linux-gnueabi-gdb

b. ubuntu下,arm-linux-gnueabi-gdb和gdb有冲突。

解决方法：

需要使用arm-linux-gnueabi-gdb时先卸载gdb,记下卸载gdb时与gdb一起被卸载的软件名，然后安装arm-linux- gnueabi-gdb。想换回gdb时，在反操作。apt-install remove arm-linux-gnueabi-gdb 然后 apt-get install gdb以及之前和gdb一起被卸载包。可以写个脚本自动完成这些操作。本人环境下的脚本是：

## 脚本1. install-armgdb.sh

```
#!/bin/sh
sudo apt-get remove gdb
sudo apt-get install gdb-arm-linux-gnueabi
```

## 脚本2. install-gdb.sh

```
#!/bin/sh
sudo apt-get remove gdb-arm-linux-gnueabi
sudo apt-get install appport appport-gtk appport-qt bug-buddy cgdb gdb python-appport ›
```

<  >

## 什么是EABI

答： 来自AAPCS

ABI： Application Binary Interface:

1) . The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the Linux ABI for the ARM Architecture.

2) . A particular aspect of the specifications to which independently produced relocatable files must conform in order to be statically linkable and executable. For example, the C++ ABI for the ARM Architecture, the Run-time ABI for the ARM Architecture, the C Library ABI for the ARM Architecture.

ARM-based ... based on the ARM architecture ...

EABI： An ABI suited to the needs of embedded (sometimes called free standing) applications.

参考：

ABI/EABI/OABI <http://blog.csdn.net/hongjiujing/archive/2008/07/21/2686556.aspx>  
[X](#)

Re: 关于kernel ARM\_EABI <http://zh-kernel.org/pipermail/linux-kernel/2008-January/002793.html>

Why ARM's EABI matters <http://www.linuxdevices.com/articles/AT5920399313.html>

Why switch to EABI? [http://www.applieddata.net/forums/topic.asp?TOPIC\\_ID=2305](http://www.applieddata.net/forums/topic.asp?TOPIC_ID=2305)

ArmEabiPort <http://wiki.debian.org/ArmEabiPort>

## 安装arm-elf-XXX 工具集

注：arm-elf-XXX 工具集是用于uclinux的

1. 依据要求搜索下载相应的arm-elf-tools安装包。比如arm-elf-tools-20030315.sh
2. 安装：\$ ./arm-elf-tools-20030315.sh
3. 如果，该安装包年代过老，比如arm-elf-tools-20030315.sh，会出现下面的错误提示“tail: 无法打开“43”读取数据: 没有那个文件或目录。”。这时需要修改安装包源码。方法：vi arm-elf-tools-20030315.sh, 搜索tail, 在它后面加 -n .比如 把tail \${SKIP} \${SCRIPT} | gunzip | tar xvf -改成如下：tail -n \${SKIP} \${SCRIPT} | gunzip | tar xvf -
- 4.如何卸载已安装的arm-elf-tools? 答，重新安装一次，注意看终端提示。或直接vi arm-elf-tools-20030315.sh,看脚本的内容，

## bin工具集的使用

[该怎么称呼这类工具？待详述]

arm-elf-addr2line	arm-elf-elf2flt	arm-elf-gdb	arm-elf-objdump	arm-elf-ranlib
arm-elf-ar	arm-elf-flthdr	arm-elf-ld	arm-elf-protize	arm-elf-run
arm-elf-as	arm-elf-g++	arm-elf-ld.real	arm-elf-ranlib	
arm-elf-c++	arm-elf-gasp	arm-elf-nm	arm-elf-readelf	
arm-elf-c++filt	arm-elf-gcc	arm-elf-objcopy		
arm-linux-gnueabi-addr2line	arm-linux-gnueabi-g++	arm-linux-gnueabi-gprof		
arm-linux-gnueabi-ar	arm-linux-gnueabi-g++-4.2	arm-linux-gnueabi-ld		
arm-linux-gnueabi-as	arm-linux-gnueabi-gcc	arm-linux-gnueabi-nm		
arm-linux-gnueabi-c++filt	arm-linux-gnueabi-gcc-4.2	arm-linux-gnueabi-objcopy		
arm-linux-gnueabi-cpp	arm-linux-gnueabi-gdb	arm-linux-gnueabi-objdump		
arm-linux-gnueabi-cpp-4.2	arm-linux-gnueabi-gdbtui	arm-linux-gnueabi-ranlib		

如何获取这些工具的命令选项？看章节“知识从哪里来”一般是用命 `xxxxxx -help`就能得到简单的命令选项列表

下载arm-linux-gnueabi- 手册地址 [http://www.codesourcery.com/gnu\\_toolchains/arm/portal/release324](http://www.codesourcery.com/gnu_toolchains/arm/portal/release324)

然后搜索“arm”，便能找到处理器相关的特殊命令选项

### **arm-linux-gnueabi-gcc**

查看arm处理器相关的编译选项

```
$ vi arch/arm/Makefile
```

阅读Makefile文件，并联系源码根目录下的.config文件，便能知道arm-linux-gnueabi-gcc用了哪些编译选项。再到手册中 查找，便能知道这些选项是干什么用的，但手册中说的不是很详细。另外查找有用解释的方法的是，利用`make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig`，找到与命令选项有关联的CONFIG\_XXX的菜单项，看它的帮助说明.比如

```
$ vi arch/arm/Makefile
....
ifeq ($(CONFIG_AEABI),y)
CFLAGS_ABI      :=-mabi=aapcs-linux -mno-thumb-interwork
else
CFLAGS_ABI      :=$(call cc-option,-mapcs-32,-mabi=apcs-gnu) $(call cc-option,-mno-
endif
..
```

再查看CONFIG\_AEABI的帮助文档 `$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig` 找到CONFIG\_AEABI相关的菜单，看它的帮助文档，便能知道选项`-mabi=aapcs-linux -mno-thumb-interwork`的整体效果怎样的。

```
Use the ARM EABI to compile the kernel
CONFIG_AEABI:
This option allows for the kernel to be compiled using the latest
ARM ABI (aka EABI). This is only useful if you are using a user
space environment that is also compiled with EABI.
```

```
| Since there are major incompatibilities between the legacy ABI and  
| EABI, especially with regard to structure member alignment, this  
| option also changes the kernel syscall calling convention to  
| disambiguate both ABIs and allow for backward compatibility support  
| (selected with CONFIG_OABI_COMPAT).
```

```
| To use this you need GCC version 4.0.0 or later.
```

```
| Symbol: AEABI [=n]
```

```
| Prompt: Use the ARM EABI to compile the kernel
```

```
| Defined at arch/arm/Kconfig:554
```

```
| Location:
```

```
| -> Kernel Features
```

arm-linux-gnueabi-gcc的主要编译选项有如下几个。但是在编译内核时，这些选项是不需要手工去写的，而是通过make menuconfig生成包含了编译选项配置信息的.config文件。在make编译内核时，再利用Makefile文件中的规则结合.config文件提取出那些选项。

太多了，手册吧

## arm-linux-gnueabi-gdb

### 注意它的默认选项设置

```
$ arm-linux-gnueabi-gdb  
(gdb) show arm  
abi: The current ARM ABI is "auto" (currently "APCS").  
apcs32: Usage of ARM 32-bit mode is on.  
disassembler: The disassembly style is "std".  
fpu: The current ARM floating point model is "auto" (currently "fpa").  
(gdb)
```

但是，如果如果在命令后有参数vmlinux的话，它会自动识别出内核的abi，从而自动设置了gdb的abi。比如，在编译内核时，如果选了CONFIG\_AEABI，则gdb的提示如下

```
$ arm-linux-gnueabi-gdb vmlinux  
...  
(gdb) show arm  
abi: The current ARM ABI is "auto" (currently "AAPCS"). <--注意  
apcs32: Usage of ARM 32-bit mode is on.  
disassembler: The disassembly style is "std".  
fpu: The current ARM floating point model is "auto" (currently "softvfp").
```

## qemu的使用

### 参考手册

<http://bellard.org/qemu/user-doc.html>

<http://wiki.debian.org.tw/index.php/QEMU>

<http://www.h7.dion.ne.jp/~qemu-win/>

<http://bellard.org/qemu/>

### 邮件列表

<http://lists.gnu.org/archive/html/qemu-devel/>

### 参考文章

“QEMU安装使用全攻略” <http://forum.ubuntu.org.cn/viewtopic.php?p=248267&sid=f4e95025bdaf6a24a218315d03ad9933>

[补充命令] 引用自<http://bbs.chinaunix.net/viewthread.php?tid=779540>

安装过程中，要求换盘：

在qemu中按ctrl+alt+2切换到qemu monitor模式 输入?或help可以查看可用命令及使用说明。

(在其他版本的qemu中，运行qemu加载OS后，这个shell就会自动变成qemu monitor模式)

change device filename -- change a removable media

看来它就是用来换盘的了：change cdrom /rhel4/EL\_disc2.iso

切换回安装界面ctrl+alt+1

monitor下还有几个常用的命令：

savevm filename 将整个虚拟机当前状态保存起来

loadvm filename 恢复（最初我没用change换盘时，就是先savevm->重新运行qemu->loadvm）

sendkey keys 向VM中发送按键，例如你想在虚拟机里切换到另一个终端，按下了ctrl-alt-F2

不幸的是，切换的却是你的主系统，所以就需要用 sendkey了 sendkey ctrl-alt-f2

还有其他几个命令，自己看看啦。

经过N久终于装好了，现在可以启动试试：

[root@LFS distro]#qemu redhat.img -enable-audio -user-net -m 64

-user-net 相当于VMware的nat，主系统可以上，虚拟机就可以

-m 64 使用64M内存，缺省下使用128M

ctrl-alt-f 全屏

ctrl-alt 主机/虚拟机鼠标切换  
qemu还有一些其他参数，输入qemu可以查看其相关说明

## initrd.img的原理与制作

[扩展，原理，相关命令。下面的skyeye可能需要这部分知识]

“Linux2.6 内核的 Initrd 机制解析” <http://www.ibm.com/developerworks/cn/linux/l-k26initrd/>

“Introducing initramfs, a new model for initial RAM disks” <http://www.linuxdevices.com/articles/AT4017834659.html>

”深入理解 Linux 2.6 的 initramfs 机制 (上)” <http://blog.linux.org.tw/~jserv/archives/001954.html>

MKINITRAMFS <http://www.manpage.org/cgi-bin/man/man2html?8+mkinitramfs>

## 安装与使用

```
$ sudo apt-get install initramfs-tools  
$ mkinitramfs /lib/modules/2.6.26/ -o initrd.img-2.6.26
```

## x86虚拟调试环境的建立

### 参考

“debugging-linux-kernel-without-kgdb” <http://memyselfandtaco.blogspot.com/2008/06/debugging-linux-kernel-without-kgdb.html>

“使用 KGDB 调试 Linux 内核” [http://blog.chinaunix.net/u/8057/showart\\_1087126.html](http://blog.chinaunix.net/u/8057/showart_1087126.html)

“透過虛擬化技術體驗 kgdb (1)” <http://blog.linux.org.tw/~jserv/archives/002045.html>

## 基于qemu和内核内置kgdb

缺点：相对于下节的“基于qemu和qemu内置gdbstub”，这个方法配置麻烦。

优点：真机远程调试时只能使用内置kgdb这个方法。

[等待扩展, ...]

## 终极参考

“Using kgdb and the kgdb Internals” <http://www.kernel.org/pub/linux/kernel/people/jwessel/kgdb/index.html>

## 参考文章

“使用 KGDB 调试 Linux 内核” [http://blog.chinaunix.net/u/8057/showart\\_1087126.html](http://blog.chinaunix.net/u/8057/showart_1087126.html)

## 基于qemu和qemu内置gdbstub

- 参考文章

“Debugging Linux Kernel Without KGDB Patch (Qemu + GDB)” <http://memyselfandtaco.blogspot.com/2008/06/debugging-linux-kernel-without-kgdb.html>

- 优缺点

优点：相对上节，优点是操作简单，几乎不需要什么配置

缺点：真机的远程调试，就只能利用内核的内置kgdb了

说明：

如果长时间调试固定版本的内核，采取下面的把调试用内核安装的虚拟机内部就可以了。但是如果是要频繁地更换新内核或修改被调试内核，就需要采取把内核挂在虚拟机外部的形式。也就是用 -kernel 在虚拟机外面挂个内核，再利用-append 传递起内核启动参数等。[待研究]

[大概过了，待扩展...]

- 调试用内核的安装过程：

1. 利用qemu安装一个系统.



## 2. 在真机中配置并编译一个用于安装到虚拟系统中的新内核，注意配置时的选择

\* 配置和启动

### 1. 内核选项

同时，为了能在系统运行时中断系统并出发远程 gdb，必须打开内核 Magic Sys-Rq 键选项：[后记，

`CONFIG_MAGIC_SYSRQ=y`

打开内核符号调试：

`CONFIG_DEBUG_INFO=y`

<

>

## 3. 在真机下编译好虚拟机新内核的源码

4. 结束qemu，用以下命令在真机上挂载虚拟硬盘。然后把编译好的整个源码目录都拷贝到挂载好的虚拟硬盘上（真机上保留一份源码）。

```
$ sudo mount -o loop,offset=32256 debian.img /mnt
```

拷贝完后，在真机上卸载虚拟硬盘

```
$ sudo umount /mnt
```

5.启动虚拟机，进入旧系统，在新内核源码根目录下用以下命令给qemu的虚拟系统安装一个新的内核

拷贝模块

```
$ make modules_install
```

安装内核

```
$ make install
```

制作initrd.img

```
$ cd /boot
```

```
$ mkinitramfs /lib/modules/2.6.26/ -o initrd.img-2.6.26
```

检查/boot/grub/menu.lst 文件内容是否妥当

6.用以下命令重启虚拟系统，并选择进入新系统，确认新系统是否安装成功。

```
$ shutdown -r now
```

- 调试：

1. 在真机新内核源码目录下建立一个文件 .gdbinit 内容是

```
target remote localhost:1234
b start_kernel
#c
```

注意我把c注释掉是因为ddd和gdb有切换的需要。见“gdb技巧”

## 2. 用以下命令启动虚拟机

```
qemu -hda debian.img -cdrom ../debian-testing-i386-CD-1.iso -m 500 -S -s
```

## 3. 在真机新内核源码目录下运行

```
gdb ./vmlinux
```

### [实验记录]

实验过了，.config中不选择kgdb，利用qemu照样能调试。也不能调试start\_kernel以前的代码。比如head\_32.S中的代码。

```
CONFIG_HAVE_ARCH_KGDB=y
# CONFIG_KGDB is not set
```

但是不知CONFIG\_HAVE\_ARCH\_KGDB是在menuconfig菜单的哪里。想试试把这项去了qemu还能不能调试。

经测试，取消CONFIG\_HAVE\_ARCH\_KGDB后，qemu也能进行调试。情况不变。看来qemu能完全脱离内核中的kgdb就能调试内核。

### • 调试截图

步骤2:

```
XXX@ubuntu:/new/myqemu/debian-x86$ qemu -hda debian.img -cdrom ../debian-testing
```

步骤3:

由下图我们注意到：“基于qemu和qemu内置gdbstub”这个方法的调试，最早只能从函数 start\_kernel 内核在start\_kernel ()之前的初始化过程就无法观察了。这就是这个方法的最大缺点。但下节利用方法就可以从第一个机器指令开始进行。

```
XXX@ubuntu:/storage/myqemu/new/linux-2.6.26$ gdb ./vmlinux
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

There is NO WARRANTY, to the extent permitted by law. Type "show copying" and "show warranty" for details.  
This GDB was configured as "i486-linux-gnu"..  
0x0000ffff in ?? ()  
Breakpoint 1 at 0xc037f5ca: file init/main.c, line 535.  
(gdb) c  
Continuing.

Breakpoint 1, start\_kernel () at init/main.c:535  
535 {  
(gdb)

调试示意图:

给sys\_read下断点

(gdb) b sys\_read

Breakpoint 2 at 0xc017585e: file fs/read\_write.c, line 360.  
(gdb)

用快捷键 ctrl+x+2 打开tui,并按c继续运行,而后拦截到sys\_read

```
fs/read_write.c
354 {
355     struct file *file;
356     ssize_t ret = -EBADF;
357     int fput_needed;
358
359     file = fget_light(fd, &fput_needed);
B+> 360     if (file) {
361         loff_t pos = file_pos_read(file);
362         ret = vfs_read(file, buf, count, &pos);
363         file_pos_write(file, pos);
364         fput_light(file, fput_needed);
365     }
366
367     return ret;

0xc017585a <sys_read>      push    %ebp
0xc017585b <sys_read+1>     mov     %esp,%ebp
0xc017585d <sys_read+3>      push    %esi
B+> 0xc017585e <sys_read+4>     mov     $0xffffffff7,%esi
0xc0175863 <sys_read+9>      push    %ebx
0xc0175864 <sys_read+10>     sub     $0xc,%esp
0xc0175867 <sys_read+13>     mov     0x8(%ebp),%eax
0xc017586a <sys_read+16>     lea     -0xc(%ebp),%edx
0xc017586d <sys_read+19>     call   0xc0175f65 <fget_light>
0xc0175872 <sys_read+24>     test    %eax,%eax
0xc0175874 <sys_read+26>     mov     %eax,%ebx
0xc0175876 <sys_read+28>     je      0xc01758b1 <sys_read+87>
0xc0175878 <sys_read+30>     mov     0x24(%ebx),%edx
0xc017587b <sys_read+33>     mov     0x20(%eax),%eax
```

remote Thread 42000 In: sys\_read  
(gdb) c  
Continuing.

```
Breakpoint 2, sys_read (fd=3, buf=0xbfc781a4 "", count=512) at fs/read_write.c:360
(gdb)
```

## arm虚拟调试环境的建立

### 利用qemu

#### 利用qemu安装debian linux

目标:

本节在qemu虚拟机上安装一个基于arm的“桌面”系统，可以有X桌面，该虚拟系统能利用apt-get从debian的软件库下载数不完的用交 叉编译已经编译好的arm下的程序和工具。除了虚拟处理器是arm外，简直就是PC机。可以进行应用程序的本机(在虚拟机内进行)调试。但是，本人装的时候，如果选了安装桌面环境，内核就启动失败，好像是提示文件系统出错。[成功的麻烦把过程贴出来]

过程是:

Debian on an emulated ARM machine [http://www.aurel32.net/info/debian\\_arm\\_qemu.php](http://www.aurel32.net/info/debian_arm_qemu.php)

下面是过程的提炼步骤,方便查看。

#### 1.创建虚拟硬盘

```
$ qemu-img create -f qcow hda.img 40G
```

#### 2.下载必要文件

```
$ wget http://people.debian.org/~aurel32/arm-versatile/vmlinuz-2.6.18-6-versatile
$ wget http://people.debian.org/~aurel32/arm-versatile/initrd.img-2.6.18-6-versatile
$ wget http://ftp.de.debian.org/debian/dists/etch/main/installer-arm/current/images
```

<

>

#### 2.安装系统

```
qemu-system-arm -M versatilepb -kernel vmlinuz-2.6.18-6-versatile -initrd initrd.g
在安装过程中，为了节省时间，在这步choose a mirror of the debian archive
选http 回车；
debian archive mirror country 选taiwan 回车；
debian archive mirror 选ftp.tw.debian.org
```

<

>

安装好基本系统后，不要选择安装Desktop environment

安装完成后，它提示你把光盘拿掉并重启系统时，终止掉qemu。并用下一步的命令启动qemu。不要回车，

```
< >
```

### 3. 第一次启动系统

```
$ qemu-system-arm -M versatilepb -kernel vmlinuz-2.6.18-6-versatile -initrd initrd.
```

```
< >
```

### 4. 把旧的内核,initrd.img制作工具安装到虚拟机的系统内（操作在虚拟机内）

```
$ apt-get install initramfs-tools
```

```
$ wget http://people.debian.org/~aurel32/arm-versatile/linux-image-2.6.18-6-versati
```

```
$ su -c "dpkg -i linux-image-2.6.18-6-versatile_2.6.18.dfsg.1-18etch1+versatile_arm
```

```
< >
```

### 5.其他更多的玩法请看原文[http://www.aurel32.net/info/debian\\_arm\\_qemu.php](http://www.aurel32.net/info/debian_arm_qemu.php)

参考：

Debian ARM Linux on Qemu

<http://909ers.apl.washington.edu/~dushaw/ARM/#SYSTEM>

Running Linux for ARM processors under QEMU

[http://iomem.com/index.php?archives/2-Running-Linux-for-ARM-processors-under-QEMU.html&serendipity\[entrypage\]=2](http://iomem.com/index.php?archives/2-Running-Linux-for-ARM-processors-under-QEMU.html&serendipity[entrypage]=2)

Debian on an emulated ARM machine

[http://www.aurel32.net/info/debian\\_arm\\_qemu.php](http://www.aurel32.net/info/debian_arm_qemu.php)

**利用qemu安装能进行内核调试的系统**

[暂时没法子，期待扩展。下面这个例子可以，但没尝试。估计这个方法与下节的利用skyeeye的方法相比，没有优势。因为这个方法可能也是不能进行全程调试。但是下面网站的资料还是有一定参考价值的。]

使用qemu-jk2410做为学习环境：

<http://wiki.jk2410.org/wiki/%E4%BD%BF%E7%94%A8qemu-jk2410%E5%81%9A%E7%82%BA%E5%AD%B8%E7%BF%92%E7%92%B0%E5%A2%83>

另外：看看下面这个站点，

Firmware Linux: <http://landley.net/code/firmware/>

## **利用skyeye**

### **skyeye虚拟机的内核调试**

相对于利用qemu的方式，用skyeye虚拟机调试内核有个很重要的

优点是：

调试可以从第一条机器指令开始。这对研究系统启动过程提供了极大的便利。

### **skyeye的安装与使用**

该文非常好，好像没啥要扩充的

SkyEye硬件模拟平台，第二部分: 安装与使用

<http://www.ibm.com/developerworks/cn/linux/l-skyeye/part2/>

SkyEye User Manual <http://www.skyeye.org/wiki/UserManual>

<http://skyeye.wiki.sourceforge.net/>

参考文档：

Linux-2.6.20 on XXX platform

<http://skyeye.wiki.sourceforge.net/Linux>

uClinux-dist-20070130 on XXX platform

<http://skyeye.wiki.sourceforge.net/uClinux>

<http://www.linuxfans.org/bbs/thread-182101-1-1.html>

安装:

## 1. 安装主程序

在ubuntu系统能进行在线安装,但版本是v1.2,不是最新的

```
$sudo apt-get install skyeye
```

## 2. 测试套件

测试套件下载后解压开即可

地址: [http://sourceforge.net/project/showfiles.php?group\\_id=85554](http://sourceforge.net/project/showfiles.php?group_id=85554)

**快速试玩**

目的:

尽可能快的成功运行一个arm linux虚拟机。如果您化了很长时间也无法编译出一个能运行的内核,或写不出一个恰当的skyeye.conf时,在你的热情受到打击之前,我想这节是你急需的。

操作步骤:

1.依照上节说明安装好主程序,下载并解压好测试套件

2.进入测试套件的目录 skyeye-testsuite-1.2.5/linux/s3c2410/s3c2410x-2.6.14

可以看到有三个文件initrd.img skyeye.conf vmlinux

## 3.运行虚拟机

```
$skyeye -e vmlinux
```

注意下面的提示,说明平时要注意在启动命令前加上sudo

```
NOTICE: you should be root at first !!!  
NOTICE: you should inmod linux kernel net driver tun.o!!!  
NOTICE: if you don't make device node, you should do commands:  
NOTICE:      mkdir /dev/net; mknod /dev/net/tun c 10 200
```

NOTICE: now the net simulation function can not support!!!  
NOTICE: Please read SkyEye.README and try again!!!

4. 可以看到，一个2.6.14 版本的linux跑起来了，还带有一个lcd.

快速配置能调试的环境

参考：

[http://skyeye.wiki.sourceforge.net/linux\\_2\\_6\\_17\\_lubbock](http://skyeye.wiki.sourceforge.net/linux_2_6_17_lubbock)

环境条件：

1. ubuntu hardy 8.04
2. 安装了debian提供的交叉编译工具套件 arm-linux-gnueabi- (4.2版本)

目标：

这小节能得到基于pxa平台（类似s3c2410，也基于arm核心）的linux2.6.20内核的虚拟系统，具备调试功能。相比“基于qemu 和qemu内置gdbstub”该节，利用skyeye的调试有那节所没有的优点：调试时可以从内核运行的第一条指令开始 [这就是模拟硬件调试？] 。

参考手册：

XScale PXA250开发手册 <http://soft.laogu.com/download/intelpxa250.pdf>

ARMv5 体系结构参考手册 <http://www.arm.com/community/university/eulaarmarm.html>

操作步骤：

1. 下载linux-2.6.20 (由于交叉编译器太新，如果利用linux-2.6.17则编译不过)
2. 修改文件include/asm-arm/arch-pxa/memory.h 第18行

```
#define PHYS_OFFSET UL(0xa0000000)
为
#define PHYS_OFFSET UL(0xc0000000)
```



3. 下载内核配置选项，放置于linux-2.6.20源码的根目录下 [http://skyeye.wiki.sourceforge.net/space/showimage/skyeye\\_2.6.17\\_lubbock.config](http://skyeye.wiki.sourceforge.net/space/showimage/skyeye_2.6.17_lubbock.config)

这个下载好的配置文件已经帮我们做了的两件事

首先，在block device菜单下配置了ramdisk和initrd的支持

其次，把内核原来的启动参数改为

```
root=/dev/ram0 console=ttyS0 initrd=0xc0800000,0x00800000 rw mem=64M
```

4. 把下载到的skyeye\_2.6.17\_lubbock.config更名为.config

5. 编译内核

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

6. 创建文件 skyeye.conf，内容如下：

```
cpu: pxa25x
mach: pxa_lubbock
mem_bank: map=I, type=RW, addr=0x40000000, size=0x0c000000
mem_bank: map=M, type=RW, addr=0xc0000000, size=0x00800000
mem_bank: map=M, type=RW, addr=0xc0800000, size=0x00800000, file=./initrd.img
mem_bank: map=M, type=RW, addr=0xc1000000, size=0x00800000
mem_bank: map=M, type=RW, addr=0xc1800000, size=0x02800000
```

7. 从skyeye的测试套件中拷贝initrd.img到linux-2.6.20源码根目录下。该initrd.img的路径是：

```
skyeye-testsuite-1.2.5/linux/pxa/2.6.x/
```

8. 运行内核看看,在linux-2.6.20源码根目录下运行下面的命令。可以看到，内核成功运行

```
sudo skyeye -e vmlinux
```

调试：

1. 在linux-2.6.20源码根目录下运行命令：

```
sudo skyeye -d -e vmlinux
```

2. 在源码根目录下新开一个终端，并运行：

```
arm-linux-gnueabi-gdb ./vmlinux
```

gdb界面出来后

```
(gdb) target remote:12345
```

之后可以看到，下断点，查看汇编等一切调试功能和x86下都一样。

3. ddd下如何调用arm-linux-gnueabi-gdb ? 答

```
$ ddd --debugger arm-linux-gnueabi-gdb ./vmlinux
```

**为s3c2410配置2.6.26内核**

[启动过程中有若干错误提示，但内核能启动成功并运行。有待研究]

目标：

得到一个基于s3c2410cpu的2.6.26最新稳定内核的虚拟系统，能进行全程的内核调试，即调试能从第一条机器指令开始进行。

参考：

<http://skyeye.wiki.sourceforge.net/Linux>

<http://www.linuxfans.org/bbs/thread-182101-1-1.html>

环境条件：

1. ubuntu hardy 8.04

2. 安装了debian提供的交叉编译工具套件 arm-linux-gnueabi- (4.2版本)

操作步骤：

## 1.依据“安装交叉编译工具”这节，安装好交叉编译工具

## 2.修改源码

将include/asm-arm/arch-s3c2410/map.h里的

```
#define S3C2410_CS6 (0x30000000)
```

改为

```
#define S3C2410_CS6 (0xc0000000)
```

将include/asm-arm/arch-s3c2410/memory.h里的

```
#define PHYS_OFFSET UL(0x30000000)
```

改为

```
#define PHYS_OFFSET UL(0xc0000000)
```

## 3.把默认.config替换为s3c2410版本

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- s3c2410_defconfig
```

## 3.修改配置文件

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig
```

进入[Device Driver] ->[ Character Driver] -> [Serial Driver] 等菜单下，  
取消8250/16550 and compatible serial support的选择

## 4.修改内核启动命令

在Boot option --> Default kernel command string 里输入

```
mem=32M console=ttySAC0 root=/dev/ram initrd=0xc0800000,0x00800000 ramdisk_size=2048
```

<

>

## 5.编译

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

## 6.从skyeye的测试套件中拷贝相应的文件initrd.img和skyeye.conf到linux-2.6.26源码根目录下。这两个文件的位于

```
skyeye-testsuite-1.25/linux/s3c2410/s3c2410x-2.6.14/中
```

## 7.启动虚拟机



```
$ ddd --debugger arm-linux-gnueabi-gdb ./vmlinux
```

截图:

步骤2:

```
XXX@ubuntu:~/桌面/test/linux-2.6.26_s3c2410$ sudo skyye -d -e vmlinux
big_endian is false.
arch: arm
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x805f030
lcd_mod:1
dbct info: Note: DBCT not compiled in. This option will be ignored
uart_mod:0, desc_in:, desc_out:, converter:
SKYEE: use arm920t mmu ops
Loaded RAM ./initrd.img
start addr is set to 0xc0008000 by exec file.
debugmode= 1, filename = skyye.conf, server TCP port is 12345
-----
```

步骤3:

```
fqh@ubuntu:~/桌面/test/linux-2.6.26_s3c2410$ arm-linux-gnueabi-gdb vmlinux
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i486-linux-gnu --target=arm-linux-gnueabi"...
stext () at arch/arm/kernel/head.S:80
80      msr      cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE @ ensure svc m
Current language: auto; currently asm
(gdb) source extendinstr //载入辅助的gdb宏
-----
```

用快捷键 **ctrl+x+2** 打开tui模式后的图示，可看到调试是从第一条指令开始的。这对研究系统启动

```

└─arch/arm/kernel/head.S
> | 80      msr      cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE @ ensure s
| 81                                     @ and irqs disabled
| 82      mrc      p15, 0, r9, c0, c0      @ get processor id
| 83      bl      __lookup_processor_type  @ r5=procinfo r9=cpu
| 84      movs     r10, r5                  @ invalid processor
| 85      beq      __error_p                @ yes, error 'p'
| 86      bl      __lookup_machine_type    @ r5=machinfo
| 87      movs     r8, r5                  @ invalid machine (r
| 88      beq      __error_a                @ yes, error 'a'
| 89      bl      __vet_atags
| 90      bl      __create_page_tables

> | 0xc0008000 <stext>      msr      CPSR_c, #211      ; 0xd3
| 0xc0008004 <stext+4>      mrc      15, 0, r9, cr0, cr0, {0}
| 0xc0008008 <stext+8>      bl      0xc00082f8 <__lookup_processor_type>
| 0xc000800c <stext+12>      movs     r10, r5
| 0xc0008010 <stext+16>      beq      0xc0008190 <__error_p>
| 0xc0008014 <stext+20>      bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008018 <stext+24>      movs     r10, r5
| 0xc000801c <stext+28>      beq      0xc0008190 <__error_p>
| 0xc0008020 <stext+32>      bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008024 <stext+36>      movs     r10, r5
| 0xc0008028 <stext+40>      beq      0xc0008190 <__error_p>
| 0xc000802c <stext+44>      bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008030 <stext+48>      movs     r10, r5
| 0xc0008034 <stext+52>      beq      0xc0008190 <__error_p>
| 0xc0008038 <stext+56>      bl      0xc00082f8 <__lookup_processor_type>
| 0xc000803c <stext+60>      movs     r10, r5
| 0xc0008040 <stext+64>      beq      0xc0008190 <__error_p>
| 0xc0008044 <stext+68>      bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008048 <stext+72>      movs     r10, r5
| 0xc000804c <stext+76>      beq      0xc0008190 <__error_p>
| 0xc0008050 <stext+80>      bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008054 <stext+84>      movs     r10, r5
| 0xc0008058 <stext+88>      beq      0xc0008190 <__error_p>
| 0xc000805c <stext+92>      bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008060 <stext+96>      movs     r10, r5
| 0xc0008064 <stext+100>     beq      0xc0008190 <__error_p>
| 0xc0008068 <stext+104>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000806c <stext+108>     movs     r10, r5
| 0xc0008070 <stext+112>     beq      0xc0008190 <__error_p>
| 0xc0008074 <stext+116>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008078 <stext+120>     movs     r10, r5
| 0xc000807c <stext+124>     beq      0xc0008190 <__error_p>
| 0xc0008080 <stext+128>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008084 <stext+132>     movs     r10, r5
| 0xc0008088 <stext+136>     beq      0xc0008190 <__error_p>
| 0xc000808c <stext+140>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008090 <stext+144>     movs     r10, r5
| 0xc0008094 <stext+148>     beq      0xc0008190 <__error_p>
| 0xc0008098 <stext+152>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000809c <stext+156>     movs     r10, r5
| 0xc00080a0 <stext+160>     beq      0xc0008190 <__error_p>
| 0xc00080a4 <stext+164>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00080a8 <stext+168>     movs     r10, r5
| 0xc00080ac <stext+172>     beq      0xc0008190 <__error_p>
| 0xc00080b0 <stext+176>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00080b4 <stext+180>     movs     r10, r5
| 0xc00080b8 <stext+184>     beq      0xc0008190 <__error_p>
| 0xc00080bc <stext+188>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00080c0 <stext+192>     movs     r10, r5
| 0xc00080c4 <stext+196>     beq      0xc0008190 <__error_p>
| 0xc00080c8 <stext+200>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00080cc <stext+204>     movs     r10, r5
| 0xc00080d0 <stext+208>     beq      0xc0008190 <__error_p>
| 0xc00080d4 <stext+212>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00080d8 <stext+216>     movs     r10, r5
| 0xc00080dc <stext+220>     beq      0xc0008190 <__error_p>
| 0xc00080e0 <stext+224>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00080e4 <stext+228>     movs     r10, r5
| 0xc00080e8 <stext+232>     beq      0xc0008190 <__error_p>
| 0xc00080ec <stext+236>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00080f0 <stext+240>     movs     r10, r5
| 0xc00080f4 <stext+244>     beq      0xc0008190 <__error_p>
| 0xc00080f8 <stext+248>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00080fc <stext+252>     movs     r10, r5
| 0xc0008100 <stext+256>     beq      0xc0008190 <__error_p>
| 0xc0008104 <stext+260>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008108 <stext+264>     movs     r10, r5
| 0xc000810c <stext+268>     beq      0xc0008190 <__error_p>
| 0xc0008110 <stext+272>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008114 <stext+276>     movs     r10, r5
| 0xc0008118 <stext+280>     beq      0xc0008190 <__error_p>
| 0xc000811c <stext+284>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008120 <stext+288>     movs     r10, r5
| 0xc0008124 <stext+292>     beq      0xc0008190 <__error_p>
| 0xc0008128 <stext+296>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000812c <stext+300>     movs     r10, r5
| 0xc0008130 <stext+304>     beq      0xc0008190 <__error_p>
| 0xc0008134 <stext+308>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008138 <stext+312>     movs     r10, r5
| 0xc000813c <stext+316>     beq      0xc0008190 <__error_p>
| 0xc0008140 <stext+320>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008144 <stext+324>     movs     r10, r5
| 0xc0008148 <stext+328>     beq      0xc0008190 <__error_p>
| 0xc000814c <stext+332>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008150 <stext+336>     movs     r10, r5
| 0xc0008154 <stext+340>     beq      0xc0008190 <__error_p>
| 0xc0008158 <stext+344>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000815c <stext+348>     movs     r10, r5
| 0xc0008160 <stext+352>     beq      0xc0008190 <__error_p>
| 0xc0008164 <stext+356>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008168 <stext+360>     movs     r10, r5
| 0xc000816c <stext+364>     beq      0xc0008190 <__error_p>
| 0xc0008170 <stext+368>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008174 <stext+372>     movs     r10, r5
| 0xc0008178 <stext+376>     beq      0xc0008190 <__error_p>
| 0xc000817c <stext+380>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008180 <stext+384>     movs     r10, r5
| 0xc0008184 <stext+388>     beq      0xc0008190 <__error_p>
| 0xc0008188 <stext+392>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000818c <stext+396>     movs     r10, r5
| 0xc0008190 <stext+400>     beq      0xc0008190 <__error_p>
| 0xc0008194 <stext+404>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008198 <stext+408>     movs     r10, r5
| 0xc000819c <stext+412>     beq      0xc0008190 <__error_p>
| 0xc00081a0 <stext+416>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00081a4 <stext+420>     movs     r10, r5
| 0xc00081a8 <stext+424>     beq      0xc0008190 <__error_p>
| 0xc00081ac <stext+428>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00081b0 <stext+432>     movs     r10, r5
| 0xc00081b4 <stext+436>     beq      0xc0008190 <__error_p>
| 0xc00081b8 <stext+440>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00081bc <stext+444>     movs     r10, r5
| 0xc00081c0 <stext+448>     beq      0xc0008190 <__error_p>
| 0xc00081c4 <stext+452>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00081c8 <stext+456>     movs     r10, r5
| 0xc00081cc <stext+460>     beq      0xc0008190 <__error_p>
| 0xc00081d0 <stext+464>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00081d4 <stext+468>     movs     r10, r5
| 0xc00081d8 <stext+472>     beq      0xc0008190 <__error_p>
| 0xc00081dc <stext+476>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00081e0 <stext+480>     movs     r10, r5
| 0xc00081e4 <stext+484>     beq      0xc0008190 <__error_p>
| 0xc00081e8 <stext+488>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00081ec <stext+492>     movs     r10, r5
| 0xc00081f0 <stext+496>     beq      0xc0008190 <__error_p>
| 0xc00081f4 <stext+500>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00081f8 <stext+504>     movs     r10, r5
| 0xc00081fc <stext+508>     beq      0xc0008190 <__error_p>
| 0xc0008200 <stext+512>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008204 <stext+516>     movs     r10, r5
| 0xc0008208 <stext+520>     beq      0xc0008190 <__error_p>
| 0xc000820c <stext+524>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008210 <stext+528>     movs     r10, r5
| 0xc0008214 <stext+532>     beq      0xc0008190 <__error_p>
| 0xc0008218 <stext+536>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000821c <stext+540>     movs     r10, r5
| 0xc0008220 <stext+544>     beq      0xc0008190 <__error_p>
| 0xc0008224 <stext+548>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008228 <stext+552>     movs     r10, r5
| 0xc000822c <stext+556>     beq      0xc0008190 <__error_p>
| 0xc0008230 <stext+560>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008234 <stext+564>     movs     r10, r5
| 0xc0008238 <stext+568>     beq      0xc0008190 <__error_p>
| 0xc000823c <stext+572>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008240 <stext+576>     movs     r10, r5
| 0xc0008244 <stext+580>     beq      0xc0008190 <__error_p>
| 0xc0008248 <stext+584>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000824c <stext+588>     movs     r10, r5
| 0xc0008250 <stext+592>     beq      0xc0008190 <__error_p>
| 0xc0008254 <stext+596>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008258 <stext+600>     movs     r10, r5
| 0xc000825c <stext+604>     beq      0xc0008190 <__error_p>
| 0xc0008260 <stext+608>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008264 <stext+612>     movs     r10, r5
| 0xc0008268 <stext+616>     beq      0xc0008190 <__error_p>
| 0xc000826c <stext+620>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008270 <stext+624>     movs     r10, r5
| 0xc0008274 <stext+628>     beq      0xc0008190 <__error_p>
| 0xc0008278 <stext+632>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000827c <stext+636>     movs     r10, r5
| 0xc0008280 <stext+640>     beq      0xc0008190 <__error_p>
| 0xc0008284 <stext+644>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008288 <stext+648>     movs     r10, r5
| 0xc000828c <stext+652>     beq      0xc0008190 <__error_p>
| 0xc0008290 <stext+656>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008294 <stext+660>     movs     r10, r5
| 0xc0008298 <stext+664>     beq      0xc0008190 <__error_p>
| 0xc000829c <stext+668>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00082a0 <stext+672>     movs     r10, r5
| 0xc00082a4 <stext+676>     beq      0xc0008190 <__error_p>
| 0xc00082a8 <stext+680>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00082ac <stext+684>     movs     r10, r5
| 0xc00082b0 <stext+688>     beq      0xc0008190 <__error_p>
| 0xc00082b4 <stext+692>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00082b8 <stext+696>     movs     r10, r5
| 0xc00082bc <stext+700>     beq      0xc0008190 <__error_p>
| 0xc00082c0 <stext+704>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00082c4 <stext+708>     movs     r10, r5
| 0xc00082c8 <stext+712>     beq      0xc0008190 <__error_p>
| 0xc00082cc <stext+716>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00082d0 <stext+720>     movs     r10, r5
| 0xc00082d4 <stext+724>     beq      0xc0008190 <__error_p>
| 0xc00082d8 <stext+728>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00082dc <stext+732>     movs     r10, r5
| 0xc00082e0 <stext+736>     beq      0xc0008190 <__error_p>
| 0xc00082e4 <stext+740>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00082e8 <stext+744>     movs     r10, r5
| 0xc00082ec <stext+748>     beq      0xc0008190 <__error_p>
| 0xc00082f0 <stext+752>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00082f4 <stext+756>     movs     r10, r5
| 0xc00082f8 <stext+760>     beq      0xc0008190 <__error_p>
| 0xc0008300 <stext+768>     msr      CPSR_c, #211      ; 0xd3
| 0xc0008304 <stext+772>     mrc      15, 0, r9, cr0, cr0, {0}
| 0xc0008308 <stext+776>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000830c <stext+780>     movs     r10, r5
| 0xc0008310 <stext+784>     beq      0xc0008190 <__error_p>
| 0xc0008314 <stext+788>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008318 <stext+792>     movs     r10, r5
| 0xc000831c <stext+796>     beq      0xc0008190 <__error_p>
| 0xc0008320 <stext+800>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008324 <stext+804>     movs     r10, r5
| 0xc0008328 <stext+808>     beq      0xc0008190 <__error_p>
| 0xc000832c <stext+812>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008330 <stext+816>     movs     r10, r5
| 0xc0008334 <stext+820>     beq      0xc0008190 <__error_p>
| 0xc0008338 <stext+824>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000833c <stext+828>     movs     r10, r5
| 0xc0008340 <stext+832>     beq      0xc0008190 <__error_p>
| 0xc0008344 <stext+836>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008348 <stext+840>     movs     r10, r5
| 0xc000834c <stext+844>     beq      0xc0008190 <__error_p>
| 0xc0008350 <stext+848>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008354 <stext+852>     movs     r10, r5
| 0xc0008358 <stext+856>     beq      0xc0008190 <__error_p>
| 0xc000835c <stext+860>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008360 <stext+864>     movs     r10, r5
| 0xc0008364 <stext+868>     beq      0xc0008190 <__error_p>
| 0xc0008368 <stext+872>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000836c <stext+876>     movs     r10, r5
| 0xc0008370 <stext+880>     beq      0xc0008190 <__error_p>
| 0xc0008374 <stext+884>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008378 <stext+888>     movs     r10, r5
| 0xc000837c <stext+892>     beq      0xc0008190 <__error_p>
| 0xc0008380 <stext+896>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008384 <stext+900>     movs     r10, r5
| 0xc0008388 <stext+904>     beq      0xc0008190 <__error_p>
| 0xc000838c <stext+908>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008390 <stext+912>     movs     r10, r5
| 0xc0008394 <stext+916>     beq      0xc0008190 <__error_p>
| 0xc0008398 <stext+920>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000839c <stext+924>     movs     r10, r5
| 0xc00083a0 <stext+928>     beq      0xc0008190 <__error_p>
| 0xc00083a4 <stext+932>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00083a8 <stext+936>     movs     r10, r5
| 0xc00083ac <stext+940>     beq      0xc0008190 <__error_p>
| 0xc00083b0 <stext+944>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00083b4 <stext+948>     movs     r10, r5
| 0xc00083b8 <stext+952>     beq      0xc0008190 <__error_p>
| 0xc00083bc <stext+956>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00083c0 <stext+960>     movs     r10, r5
| 0xc00083c4 <stext+964>     beq      0xc0008190 <__error_p>
| 0xc00083c8 <stext+968>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00083cc <stext+972>     movs     r10, r5
| 0xc00083d0 <stext+976>     beq      0xc0008190 <__error_p>
| 0xc00083d4 <stext+980>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00083d8 <stext+984>     movs     r10, r5
| 0xc00083dc <stext+988>     beq      0xc0008190 <__error_p>
| 0xc00083e0 <stext+992>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00083e4 <stext+996>     movs     r10, r5
| 0xc00083e8 <stext+1000>     beq      0xc0008190 <__error_p>
| 0xc00083ec <stext+1004>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc00083f0 <stext+1008>     movs     r10, r5
| 0xc00083f4 <stext+1012>     beq      0xc0008190 <__error_p>
| 0xc00083f8 <stext+1016>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008400 <stext+1024>     msr      CPSR_c, #211      ; 0xd3
| 0xc0008404 <stext+1028>     mrc      15, 0, r9, cr0, cr0, {0}
| 0xc0008408 <stext+1032>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000840c <stext+1036>     movs     r10, r5
| 0xc0008410 <stext+1040>     beq      0xc0008190 <__error_p>
| 0xc0008414 <stext+1044>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008418 <stext+1048>     movs     r10, r5
| 0xc000841c <stext+1052>     beq      0xc0008190 <__error_p>
| 0xc0008420 <stext+1056>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008424 <stext+1060>     movs     r10, r5
| 0xc0008428 <stext+1064>     beq      0xc0008190 <__error_p>
| 0xc000842c <stext+1068>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008430 <stext+1072>     movs     r10, r5
| 0xc0008434 <stext+1076>     beq      0xc0008190 <__error_p>
| 0xc0008438 <stext+1080>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc000843c <stext+1084>     movs     r10, r5
| 0xc0008440 <stext+1088>     beq      0xc0008190 <__error_p>
| 0xc0008444 <stext+1092>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008448 <stext+1096>     movs     r10, r5
| 0xc000844c <stext+1100>     beq      0xc0008190 <__error_p>
| 0xc0008450 <stext+1104>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008454 <stext+1108>     movs     r10, r5
| 0xc0008458 <stext+1112>     beq      0xc0008190 <__error_p>
| 0xc000845c <stext+1116>     bl      0xc00082f8 <__lookup_processor_type>
| 0xc0008460 <stext+1120>     movs
```

```

0xc0008018 <stext+24>    movs    r8, r5
0xc000801c <stext+28>    beq     0xc00081e8 <__error_a>
0xc0008020 <stext+32>    bl      0xc00083a0 <__vet_atags>
0xc0008024 <stext+36>    bl      0xc0008078 <__create_page_tables>
0xc0008028 <stext+40>    ldr     sp, [pc, #240] ; 0xc0008120 <__switch_data>

```

remote Thread 42000 In: stext

(gdb) b sys\_read //下断点

Breakpoint 1 at 0xc008cc4c: file fs/read\_write.c, line 354.

(gdb) c

调试示意图

效果可能与你机器上看到的不一樣。这个例子中，每个gdb单步指令都会自动显示backtrace。这是因为

```

include/asm/thread_info.h
91      */
92      static inline struct thread_info *current_thread_info(void) __attribute__((always_inline))
93
94      static inline struct thread_info *current_thread_info(void)
95      {
96          register unsigned long sp asm ("sp");
97      >   return (struct thread_info *) (sp & ~(THREAD_SIZE - 1));
98      }
99
100     /* thread information allocation */
101     #ifdef CONFIG_DEBUG_STACK_USAGE
102     #define alloc_thread_info(tsk) \
103         ((struct thread_info *) __get_free_pages(GFP_KERNEL | __GFP_ZERO,
104           THREAD_SIZE_ORDER))

```

```

0xc008d480 <fget_light>      mov     r12, sp
0xc008d484 <fget_light+4>      push    {r11, r12, lr, pc}
0xc008d488 <fget_light+8>      sub     r11, r12, #4 ; 0x4
0xc008d48c <fget_light+12>     bic     r3, sp, #8128 ; 0x1fc0
9> 0xc008d490 <fget_light+16>    bic     r3, r3, #63 ; 0x3f
0xc008d494 <fget_light+20>    ldr     r3, [r3, #12]
0xc008d498 <fget_light+24>    mov     r12, #0 ; 0x0
0xc008d49c <fget_light+28>    ldr     r2, [r3, #560]
0xc008d4a0 <fget_light+32>    str     r12, [r1]
0xc008d4a4 <fget_light+36>    ldr     r3, [r2]
0xc008d4a8 <fget_light+40>    cmp     r3, #1 ; 0x1
0xc008d4ac <fget_light+44>    bne     0xc008d4d0 <fget_light+80>
0xc008d4b0 <fget_light+48>    ldr     r2, [r2, #4]
0xc008d4b4 <fget_light+52>    ldr     r3, [r2]

```

remote Thread 42000 In: fget\_light

Program received signal SIGHUP, Hangup.

0xc008d490 in fget\_light (fd=1, fput\_needed=0xc1c17ed4) at include/asm/thread\_info.h:103

```

#0  0xc008d490 in fget_light (fd=1, fput_needed=0xc1c17ed4) at include/asm/thread_info.h:103
#1  0xc008cc5c in sys_read (fd=1, buf=0xc1196800 "", count=512) at fs/read_write.c:354
#2  0xc000ac7c in rd_load_image (from=0xc02b43bc "/initrd.image") at init/do_mounts.c:121
#3  0xc000bba8 in initrd_load () at init/do_mounts.c:121

```

```
#4 0xc00094c0 in prepare_namespace () at init/do_mounts.c:384
#5 0xc0008a9c in kernel_init (unused=<value optimized out>) at init/main.c:878
#6 0xc0048484 in sys_waitid (which=<value optimized out>, upid=-1044283692, info=
address 0x4
) at kernel/exit.c:1689
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb)
```

## 使用最新的skyeye

### 1. 新版本的改进

在ubuntu下利用在线安装命令所安装的skyeye是旧的版本，新版本修正了旧版本的一些小问题。比如，旧版本在调试时会出现下面一些烦人的小提示。

```
Can't send signals to this remote system.  SIGHUP not sent.
```

```
Program received signal SIGHUP, Hangup.
```

但是，两个版本并不是完全兼容的，主要是skyeye.conf的处理上。不过，幸好这些都是很容易解决的问题。

### 2. 新版本的安装

[http://sourceforge.net/project/showfiles.php?group\\_id=85554](http://sourceforge.net/project/showfiles.php?group_id=85554)

到上面的网站下载最新版本，目前是skyeye-1.2.6\_rc1。解压后用下面命令编译就可以了

```
$. /configure
$ make STATIC=1
```

然后把在源码根目录下生成的skyeye拷到内核目录下运行即可。这样系统中的老版本skyeye还照样可以使用。

```
sudo ./skyeye -d -e vmlinux
```

### 3. 新老版本的兼容问题

主要是skyeye.conf的格式识别上。老版本要求load\_address,load\_address\_mask不能写在 skyeye.conf文件内部，只能用-l选项指定。如果运行老版本时提示skyeye.conf出错，你就得去查查那里，并手动修改处理一下即可。

## arm开发板调试环境的建立

### 基于串口

为qq2440平台移植2.6.26或更新内核，并建立kgdb调试环境

进行中...

[移植中的一些零碎的笔记]

### 1.内核版本

使用linus的git，但是已知2.6.25中arm已经支持kgdb了。

```
XXX@ubuntu:/storage/linus-git/linux-2.6$ git-describe  
v2.6.27-rc9-2-g85ba94b
```

### 2.

arm体系的默认配置文件在  
arch/arm/configs

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- s3c2410_defconfig
```

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig
```

选取以下选项

```
CONFIG_DEBUG_INFO=y
```

```
CONFIG_KGDB=y
```

```
CONFIG_KGDB_SERIAL_CONSOLE=y
```

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

移植环境

windows: 硬盘安装的真实系统(XP)

ubuntu: 运行在windows下的vmware虚拟机中

qq2440开发板: 真实开发板, IP是192.168.1.230

第一天:(完成)

熟悉开发板, PC机, 虚拟机的网络互连

理解内核启动过程



开发板与PC机(XP)PING不通的原因有

1. PC机开着防火
2. PC机上的VMWARE的网络设置有问题（先卸载确认）
3. 安全类软件造成，比如卡巴司机(先卸载，不行重装系统)

ubuntu的网络配置分两种情况，一种是平时上网用的，一种是和开发板通讯用的。

平时使用虚拟机ubuntu上网的配置：

连接方式选出NAT: used to share the host's IP address

虚拟系统启动后,桌面右上角的

wired connection->properties->configuration选automatic configuration(DHCP)

开发板挂载ubuntu虚拟系统中的nfs

1.虚拟机本身的网络设置不用动

2.虚拟系统如ubuntu的网卡设置改为桥接

edit virtual machine settings->virtual machine setting->hardware->ethernet  
->bridged:connected directly to the physical network

3.虚拟系统启动后,桌面右上角的manual network configuration要改.

点左键->network settings->wired connection->properties:enable roaming mode不选,  
connection settings

configuration:static IP address

IP address:192.168.1.111 与PC机IP, 开发板IP同个网段

subnet mask:255.255.255.0

gateway address:空

PC机网络信息：

Ethernet adapter 本地连接：

```
Connection-specific DNS Suffix  . :  
IP Address. . . . . : 192.168.1.100  
Subnet Mask . . . . . : 255.255.255.0  
Default Gateway . . . . . :
```

开发板的网络信息：

[root@none) /]# ifconfig

```
eth0      Link encap:Ethernet  HWaddr 08:00:3E:26:0A:5B  
          inet addr:192.168.1.230  Bcast:192.168.1.255  Mask:255.255.255.0  
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1  
          RX packets:1011 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:610 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:1000  
          RX bytes:111858 (109.2 KiB)  TX bytes:57276 (55.9 KiB)  
          Interrupt:53 Base address:0x300
```

windows打开ubuntu中的samba共享目录的方法

\\192.168.1.111

ubuntu中nfs服务的安装和启用

```
$ sudo apt-get install nfs-common
$ sudo apt-get install nfs-kernel-server
```

```
$ sudo vi /etc/exports
/new/root_nfs *(rw, sync)
```

```
$ sudo /etc/init.d/nfs-kernel-server start
```

#### 4. 检查

```
$ showmount -e localhost
开发板挂载nfs成功后可看到显示结果是
All mount points on localhost:
192.168.1.230:/new/root_nfs
```

开发板挂载ubuntu中的nfs  
(此时运行的文件系统还是在开发板上)

```
mount -t nfs -o nolock 192.168.1.111:/new/root_nfs /tmp/fuck
```

192.168.1.111: ubuntu的IP  
/tmp/fuck: 开发板中的挂载点

```
[root@(none) /]# mount -t nfs -o nolock 192.168.1.111:/new/root_nfs /tmp/fuck
[root@(none) /]# cd /tmp/fuck/
[root@(none) fuck]# ls
bin          lib          proc         usr
dev          linuxrc     sbin         var
etc          mnt         shanghaitan.mp3  www
home        opt         tmp
-----
```

通过nfs启动开发板  
(挂载的文件系统是在ubuntu虚拟系统上)

下面文字来自于: Embedded Linux Primer: A Practical, Real-World Approach

```
ip=192.168.1.139:192.168.1.1:192.168.1.1:255.255.255.0:coyote1:eth0:off
ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<PROTO>
```

Here, client-ip is the target's IP address; server-ip is the address of the NFS server.

本人的实际操作命令参数

```
param set linux_cmd_line "console=ttySAC0 root=/dev/nfs nfsroot=192.168.1.111:/n
```

注意把编辑器的换行功能去掉后，再复制上面的命令。

192.168.1.130是开发板的IP，系统启动后，用ifconfig就会显示这个IP地址。可以随意设置，当然

130:192.168.1.111: nfs的server,也就是ubuntu的IP

按住空格键重启开发板,出现:

```
+-----+
| S3C2440A USB Downloader ver R0.03 2004 Jan |
+-----+
USB: IN_ENDPOINT:1 OUT_ENDPOINT:3
FORMAT: <ADDR(DATA):4>+<SIZE(n+10):4>+<DATA:n>+<CS:2>
NOTE: Power off/on or press the reset button for 1 sec
      in order to get a valid USB device address.

NAND device: Manufacture ID: 0xec, Chip ID: 0x76 (Samsung K9D1208V0M)
Found saved vivi parameters.
Press Return to start the LINUX/Wince now, any other key for vivi
type "help" for help.
Supervivi> menu
```

```
##### FriendlyARM BIOS for 2440 #####
```

```
[x] bon part 0 320k 2368k
[v] Download vivi
[k] Download linux kernel
[y] Download root_yaffs image
[c] Download root_cramfs image
[n] Download Nboot
[e] Download Eboot
[i] Download WinCE NK.nb0
[w] Download WinCE NK.bin
[d] Download & Run
[f] Format the nand flash
[p] Partition for Linux
[b] Boot the system
[s] Set the boot parameters
[t] Print the TOC struct of wince
[q] Goto shell of vivi
Enter your selection: s //<--
```

```
##### Parameter Menu #####
```

```
[r] Reset parameter table to default table
[s] Set parameter
[v] View the parameter table
[w] Write the parameter table to flash memeory
[q] Quit
Enter your selection: s //<--
```

Enter the parameter's name(mach\_type, media\_type, linux\_cmd\_line, etc): linux\_cm

Enter the parameter's value(if the value contains space, enclose it with "\"): "

Change linux command line to "console=ttySAC0 root=/dev/nfs nfsroot=192.168.1.11:

##### Parameter Menu #####

[r] Reset parameter table to default table  
[s] Set parameter  
[v] View the parameter table  
[w] Write the parameter table to flash memory  
[q] Quit

Enter your selection: w //<--

Found block size = 0x0000c000

Erasing... ... done

Writing... ... done

Written 49152 bytes

Saved vivi private data

第二天: (完成)

文件系统制作

理解系统启动过程

先实验在skyeye下能不能成功, 学习一下文件系统的制作。而后再下载到开发板实验

```
dd if=/dev/zero of=./test.image bs=1k count=8192
```

块大小单位: 1k, 8120块, 8M

```
mke2fs ./test.image
```

格式化

```
mkdir fuckroot
```

```
tar -xzvf root_mini.tgz
```

```
sudo mount -o loop test.image ./fuckroot/
```

```
cp -r root_mini/* fuckroot/
```

```
sudo umount fuckroot/
```

可以将文件系统映像压缩后再使用:

```
gzip -v9 test.image > test.image.gz
```

本人这个文件系统解压后的大小是6.4M, 制作成8M大的test.image, 压缩成test.image.gz后只有2M, 但是利用skyeye启动时, 解压花的时间比较长。

命令行中的ramdisk\_size太小, 修改.

```
mem=32M console=ttySAC0 root=/dev/ram initrd=0xc0800000,0x00800000 ramdisk_size=8192
```

```
ramdisk_size=N
```

This parameter tells the RAM disk driver to set up RAM disks of N k size.

问题, 文件系统没创建console设备节点:

```
RAMDISK: Loading 8192KiB [1 disk] into ram disk... done.
```

```
VFS: Mounted root (ext2 filesystem).
```

```
Freeing init memory: 132K
```

```
Warning: unable to open an initial console.
```

创建rootfs过程中, 在/dev目录下手动创建如下节点:

```
mknod -m 660 null c 1 3
```

```
mknod -m 660 console c 5 1
```

```

VFS: Mounted root (ext2 filesystem).
Freeing init memory: 132K
hwclock: Could not access RTC: No such file or directory
mknod: /dev/pts/0: No such file or directory
mount: Mounting none on /tmp failed: Invalid argument
mount: Mounting none on /var failed: Invalid argument
/etc/init.d/rcS: /etc/init.d/rcS: 44: cannot create /dev/vc/0: Directory nonexist
/etc/init.d/rcS: /etc/init.d/rcS: 45: cannot create /dev/vc/0: Directory nonexist
/etc/rc.d/init.d/httpd: /etc/rc.d/init.d/httpd: 16: /sbin/boa: not found
/etc/init.d/rcS: /etc/init.d/rcS: 48: cannot create /dev/vc/0: Directory nonexist
/etc/init.d/rcS: /etc/init.d/rcS: 49: cannot create /dev/vc/0: Directory nonexist
/etc/rc.d/init.d/leds: /etc/rc.d/init.d/leds: 16: /etc/init.d/rcS: /etc/init.d/rcS: 49: cannot create /dev/vc/0: Directory nonexist
/etc/init.d/rcS: /etc/init.d/rcS: 53: cannot create /dev/vc/0: Directory nonexist

/sbin/led-player: not found
SIOCSIFADDR: No such device
SIOCGIFFLAGS: No such device
/etc/init.d/rcS: /etc/init.d/rcS: 59: /sbin/madplay: not found

Please press Enter to activate this console.
-sh: can't access tty; job control turned off
id: unknown uid 0
[@FriendlyARM /]# ls
bin          home         lost+found   sbin         var
dev          lib          mnt         tmp          www
etc          linuxrc     proc        usr
[@FriendlyARM /dev]# ls
console  dsp      fb0      mixer    null     sda1    tty1    video0

```

还有一堆提示，但总算系统能跑了。

现在我的心头大患是udev的问题，因为2.6.26内核中没有devfs了。但有下面这篇文章参考udev轻松上路  
<http://www.linuxforum.net/forum/showflat.php?Cat=&Board=embedded&Number=628054&page=1>

第三天：(完成)

移植内核2.6.27-rc9到qq2440开发板，实现基本功能，能挂载板上文件系统。

步骤：

- 1.使用vivi修改mach\_type参数
- 2.修改时钟频率
- 3.修改源码正确分区
- 4.禁止nand的ECC校验

分述：

问题1.表现

```
Uncompressing Linux.....
```

```
Error: unrecognized/unsupported machine ID (r1 = 0x0000030e).
```

Available machine support:

ID (hex)	NAME
00000001	EMDK2410

```

0000015b      IPAQ-H1940
0000039f      Acer-N35
00000290      Acer-N30
0000014b      Simtec-BAST
000002a8      Nex Vision - Otom 1.1
00000400      AML_M5900
000001db      Thorcom-VR1000
00000454      QT2410
000003fe      SMDK2413
000003f1      SMDK2412
00000377      S3C2413
00000474      VSTMS
000002de      Simtec-Anubis
0000034a      Simtec-OSIRIS
00000250      IPAQ-RX3715
0000016a      SMDK2440
000002a9      NexVision - Nexcoder 2440
0000043c      SMDK2443

```

Please check your kernel config and/or bootloader.

解决方法:

##### Parameter Menu #####

[r] Reset parameter table to default table

[s] Set parameter

[v] View the parameter table

[w] Write the parameter table to flash memory

[q] Quit

Enter your selection: s

Enter the parameter's name(mach\_type, media\_type, linux\_cmd\_line, etc): mach\_type

Enter the parameter's value(if the value contains space, enclose it with "): 362

Change 'mach\_type' value. 0x0000030e(782) to 0x0000016a(362)

问题2.表现

Uncompressing Linux.....

8?'·{e#???;?·7'0??3G?#?G'?乱码

解决方法:

```

static void __init smdk2440_map_io(void)
{
    s3c24xx_init_io(smdk2440_iodesc, ARRAY_SIZE(smdk2440_iodesc));
    s3c24xx_init_clocks(12000000);//修改处, 原为16934400
    s3c24xx_init_uarts(smdk2440_uartcfgs, ARRAY_SIZE(smdk2440_uartcfgs));
}

```

问题3.表现

VFS: Cannot open root device "mtdblock2" or unknown-block(31,2)

Please append a correct "root=" boot option; here are the available partitions:

```

1f00      16 mtdblock0 (driver?)
1f01      2048 mtdblock1 (driver?)
1f02      4096 mtdblock2 (driver?)
1f03      2048 mtdblock3 (driver?)
1f04      4096 mtdblock4 (driver?)
1f05     10240 mtdblock5 (driver?)
1f06     24576 mtdblock6 (driver?)
1f07     16384 mtdblock7 (driver?)

```

Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(31,2)

依据nand分区修改源码:

```
static struct mtd_partition smdk_default_nand_part[] = {
    [0] = {
        .name    = "vivi",
        .size    = 0x00030000,
        .offset  = 0,
    },

    [1] = {
        .name    = "kernel",
        .offset  = 0x00050000,
        .size    = 0x00200000,
    },

    [2] = {
        .name    = "root",
        .offset  = 0x00250000,
        .size    = 0x03dac000,
    },
};
```

问题4. 表现

Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(31,2)  
导致上面panic的原因是没有禁止Flash ECC校验

解决:

```
s3c2410_nand_init_chip()
```

```
..
```

```
    if (set->disable_ecc)
```

```
        chip->ecc.mode = NAND_ECC_NONE;
```

```
    chip->ecc.mode = NAND_ECC_NONE; //<- 在函数最后加上
```

启动信息:

Copy linux kernel from 0x00050000 to 0x30008000, size = 0x00200000 ... done

zImage magic = 0x016f2818

Setup linux parameters at 0x30000100

linux command line is: "noinitrd root=/dev/mtdblock2 init=/linuxrc console=ttySAC0

MACH\_TYPE = 362

NOW, Booting Linux.....

Uncompressing Linux.....

Linux version 2.6.27-rc9 (fqh@ubuntu-sniper) (gcc version 4.2.4 (Debian 4.2.4-3))

CPU: ARM920T [41129200] revision 0 (ARMv4T), cr=c0007177

Machine: SMDK2440

ATAG\_INITRD is deprecated; please update your bootloader.

Memory policy: ECC disabled, Data cache writeback

CPU S3C2440A (id 0x32440001)

S3C244X: core 405.000 MHz, memory 101.250 MHz, peripheral 50.625 MHz

S3C24XX Clocks, (c) 2004 Simtec Electronics

CLOCK: Slow mode (1.500 MHz), fast, MPLL on, UPLL on

CPU0: D VIVT write-back cache

CPU0: I cache: 16384 bytes, associativity 64, 32 byte lines, 8 sets

CPU0: D cache: 16384 bytes, associativity 64, 32 byte lines, 8 sets

Built 1 zonelists in Zone order, mobility grouping on. Total pages: 16256

Kernel command line: noinitrd root=/dev/mtdblock2 init=/linuxrc console=ttySAC0

irq: clearing pending ext status 00000200

irq: clearing subpending status 00000002

BTB hash table entries: 256 (order: 8, 1024 bytes)

```
timer tcon=00000000, tcnt a4ca, tcfg 00000200,00000000, usec 00001e57
Console: colour dummy device 80x30
console [ttySAC0] enabled
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 64MB = 64MB total
Memory: 61140KB available (3224K code, 335K data, 144K init)
Calibrating delay loop... 201.93 BogoMIPS (lpj=504832)
Mount-cache hash table entries: 512
CPU: Testing write buffer coherency: ok
net_namespace: 440 bytes
NET: Registered protocol family 16
S3C2410 Power Management, (c) 2004 Simtec Electronics
S3C2440: Initialising architecture
S3C2440: IRQ Support
S3C24XX DMA Driver, (c) 2003-2004,2006 Simtec Electronics
DMA channel 0 at c4800000, irq 33
DMA channel 1 at c4800040, irq 34
DMA channel 2 at c4800080, irq 35
DMA channel 3 at c48000c0, irq 36
S3C244X: Clock Support, DVS off
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
NET: Registered protocol family 2
IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
TCP established hash table entries: 2048 (order: 2, 16384 bytes)
TCP bind hash table entries: 2048 (order: 1, 8192 bytes)
TCP: Hash tables configured (established 2048 bind 2048)
TCP reno registered
NET: Registered protocol family 1
NetWinder Floating Point Emulator V0.97 (extended precision)
JFFS2 version 2.2. (NAND) (SUMMARY) © 2001-2006 Red Hat, Inc.
msgmni has been set to 119
io scheduler noop registered
io scheduler anticipatory registered (default)
io scheduler deadline registered
io scheduler cfq registered
Console: switching to colour frame buffer device 30x40
fb0: s3c2410fb frame buffer device
lp: driver loaded but no devices found
ppdev: user-space parallel port driver
Serial: 8250/16550 driver4 ports, IRQ sharing enabled
s3c2440-uart.0: s3c2410_serial0 at MMIO 0x50000000 (irq = 70) is a S3C2440
s3c2440-uart.1: s3c2410_serial1 at MMIO 0x50004000 (irq = 73) is a S3C2440
s3c2440-uart.2: s3c2410_serial2 at MMIO 0x50008000 (irq = 76) is a S3C2440
brd: module loaded
loop: module loaded
dm9000 Ethernet Driver, V1.31
Uniform Multi-Platform E-IDE driver
Driver 'sd' needs updating - please use bus_type methods
S3C24XX NAND Driver, (c) 2004 Simtec Electronics
s3c2440-nand s3c2440-nand: Tacls=3, 29ns Twrph0=7 69ns, Twrph1=3 29ns
```



```
NAND device: Manufacturer ID: 0xec, Chip ID: 0x76 (Samsung NAND 64MiB 3,3V 8-bit)
NAND_ECC_NONE selected by board driver. This is not recommended !!
Scanning device for bad blocks
Bad eraseblock 562 at 0x008c8000
Bad eraseblock 566 at 0x008d8000
Creating 3 MTD partitions on "NAND 64MiB 3,3V 8-bit":
0x00000000-0x00030000 : "vivi"
0x00050000-0x00250000 : "kernel"
0x00250000-0x03ffc000 : "root"
usbmon: debugfs is not available
s3c2410-ohci s3c2410-ohci: S3C24XX OHCI
s3c2410-ohci s3c2410-ohci: new USB bus registered, assigned bus number 1
s3c2410-ohci s3c2410-ohci: irq 42, io mem 0x49000000
usb usb1: configuration #1 chosen from 1 choice
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 2 ports detected
usbcore: registered new interface driver libusual
usbcore: registered new interface driver usbserial
usbserial: USB Serial support registered for generic
usbcore: registered new interface driver usbserial_generic
usbserial: USB Serial Driver core
usbserial: USB Serial support registered for FTDI USB Serial Device
usbcore: registered new interface driver ftdi_sio
ftdi_sio: v1.4.3:USB FTDI Serial Converters Driver
usbserial: USB Serial support registered for pl2303
usbcore: registered new interface driver pl2303
pl2303: Prolific PL2303 USB to serial adaptor driver
mice: PS/2 mouse device common for all mice
S3C24XX RTC, (c) 2004,2006 Simtec Electronics
s3c2440-i2c s3c2440-i2c: slave address 0x10
s3c2440-i2c s3c2440-i2c: bus frequency set to 98 KHz
s3c2440-i2c s3c2440-i2c: i2c-0: S3C I2C adapter
S3C2410 Watchdog Timer, (c) 2004 Simtec Electronics
s3c2410-wdt s3c2410-wdt: watchdog inactive, reset disabled, irq enabled
TCP cubic registered
NET: Registered protocol family 17
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
VFS: Mounted root (cramfs filesystem) readonly.
Freeing init memory: 144K
hwclock: Could not access RTC: No such file or directory
mknod: /dev/pts/0: Read-only file system
ln: /dev/video0: Read-only file system
ln: /dev/fb0: Read-only file system
ln: /dev/tty1: Read-only file system
ln: /dev/dsp: Read-only file system
ln: /dev/mixer: Read-only file system
ln: /dev/sda1: Read-only file system
/etc/init.d/rcS: /etc/init.d/rcS: 44: cannot create /dev/vc/0: Read-only file system
/etc/init.d/rcS: /etc/init.d/rcS: 45: cannot create /dev/vc/0: Read-only file system
/etc/rc.d/init.d/httpd: /etc/rc.d/init.d/httpd: 16: /sbin/boa: not found
/etc/init.d/rcS: /etc/init.d/rcS: 48: cannot create /dev/vc/0: Read-only file system
/etc/init.d/rcS: /etc/init.d/rcS: 49: cannot create /dev/vc/0: Read-only file system
```

```
/etc/rc.d/init.d/leds: /etc/rc.d/init.d/leds: 16: /sbin/led-player: not found
/etc/init.d/rcS: /etc/init.d/rcS: 52: cannot create /dev/vc/0: Read-only file system
/etc/init.d/rcS: /etc/init.d/rcS: 53: cannot create /dev/vc/0: Read-only file system
```

```
SIOCSIFADDR: No such device
```

```
SIOCGIFFLAGS: No such device
```

```
/etc/init.d/rcS: /etc/init.d/rcS: 59: /sbin/madplay: not found
```

Please press Enter to activate this console.

```
-sh: can't access tty; job control turned off
```

```
id: unknown uid 0
```

```
[@FriendlyARM /]# uname -a
```

```
Linux FriendlyARM 2.6.27-rc9 #8 Sat Oct 11 03:17:21 CST 2008 armv4tl unknown
```

```
[@FriendlyARM /]#
```

第四天:

实现XP下虚拟机中的ubuntu利用gdb通过串口调试开发板上的2.6.27-rc9内核问题, 开发板只有一个串口, 给gdb占用了, 怎么操作开发板?

第五天:

实现硬盘安装的ubuntu系统利用gdb通过串口调试开发板上的内核。

第六天:

移植cs8900a网卡驱动。

实现开发板从硬盘ubuntu的nfs启动。

实现硬盘安装的ubuntu系统利用gdb通过网口调试开发板上的内核。

参考:

ubuntu8.04+skyyeye1.2.4搭建linux2.6.24+s3c2410的模拟arm-linux开发环境

<http://www.google.cn/search?complete=1&hl=zh-CN&newwindow=1&client=firefox-a&rls=org.mozilla:en-US:official>

[http://blog.chinaunix.net/u2/72751/showart\\_1130655.html](http://blog.chinaunix.net/u2/72751/showart_1130655.html)

<http://www.akae.cn/bbs/redirect.php?tid=6929&goto=lastpost>

<

>

## 基于网口

## gdb基础

### 基本命令

推荐这篇, 内容很全: gdb 使用手册 <http://blog.chinaunix.net/u/11240/showart.php?id=340632>

终极参考: Debugging with GDB [http://sourceware.org/gdb/current/onlinedocs/gdb.html#SEC\\_Top](http://sourceware.org/gdb/current/onlinedocs/gdb.html#SEC_Top)

## **gdb之gui**

网址：

cgdb:<http://cgdb.sourceforge.net/>

kgdb:<http://www.kdbg.org/screenshot.php>

ddd:<http://www.gnu.org/software/ddd/>

insight:<http://sourceware.org/insight/>

这些工具在ubuntu下都有编译好的.deb安装包，利用“立新得”就直接搜索然后在线安装。

这篇短文是我的浅陋之见，我接触这些gui的时间也不久。错误难免。虚拟机：qemu

内核内置kgdb

developer machine: 运行gdb

除了只用命令行gdb外，还可以用gdb的gui,有

1.cgdb 缺点：界面简陋，自动化程度低，只是把terminal分为两部分，上面部分显示源码，下面打命令。由于没有显示反汇编的窗体，不适合要求使用到 stepi命令的场合。优点：运行快，锻炼手指头。最大的优点是，它有完美的代码着色功能。其他几款调试器中都没有。

2.ddd: 缺点：与kdbg相比，界面凌乱。优点：代码显示效果比kdbg好，c和反汇编代码分开在两个窗口。可以随时暂停程序的运行。data windows 这个功能非常强大灵活。提示 ddd -tty 2>/dev/null ./vmlinux ; remote target localhost:1234

3. kdbg: 缺点：功能比ddd弱。字体太小，c和反汇编代码交错显示，反汇编代码折叠隐藏在C代码之间，要显示反汇编代码要手动展开，不可忍受。太过界面化，居然找不到是在哪里手动打gdb命令。致命缺点是，内核跑起来后，如果没有断点拦截，就没法把内核的运行暂停下来，kdbg成了没事佬，源码窗口的显示不更新。另一个

致命缺点是，如果没有源码只有二进制文件，虽然可以下断点，但无法显示反汇编代码，没意义。据说kdbg是用来调试kde程序的，实际上也能调试内核。优点：窗口可以整合到一块，稳定。有变化的寄存器会显示红色。提示 `kdbg -r localhost:1234 ./vmlinux`

4. insight: 和ddd都是基于TCL/TK，比较相似。优点：源码显示功能最强，可以选择C和反汇编代码分开和交叉显示。可以选择反汇编代码使用intel还是 at&t格式。可以列出当前有哪些源文件，当前文件有哪些函数。变化的寄存器有改变颜色的功能，ddd则没有。缺点：和ddd一样，小窗口无法整合到窗口中，但比ddd差的是，主窗口最大化后小窗口无法保持置顶。相对ddd的大劣势是没有一个强大的data windows。感觉界面比ddd强大，但灵活性比ddd差点。对于调试内核来说，还有一个和kdbg相同的大缺点，内核只能通过断点暂停运行，而ddd下还可以用ctrl+c暂停内核。另外它有个SB错误，显示backtrace的窗口,标题居然是stack. 提示: `insight ./vmlinux`

5. xgdb: 古董级别。没事干的时候可以玩玩

6. 其实，gdb自带了一个基于curses的gui。启动方式是gdbtui xxx; 或者在gdb启动之后用命令layout启动gui。很好用，可以至多同时显示三个分窗口。要是代码有着色功能就好了。

针对内核调试的总结：

1. kdbg不适合调试内核

3. 如果想复习gdb强大的命令，选cgdb或纯gdb。

4. 如果想学习汇编，insight是不二选择。

5 如果倾向于把调试器当作浏览器使用，作为source insight等工具的辅助工具，在内核运行中拦截函数，分析函数的调用关系，不需要反汇编的话，则cgdb是不错的选择。(source insight等源码分析工具有个共同的缺点，因为体系和内核配置不同，一个函数有很多的定义，借助调试器可以在内核运行的时候找出实际调用的那个)

6.insight和ddd很接近，各有千秋。但如果侧重于追溯数据结构体间的联系，ddd更好一点，因为它有data window，它的强项是数据和数据结构关系分析并用图像方式

显示出来 (What is DDD? Data Display Debugger)。如果侧重于分析汇编指令是怎么在cpu中跑的, 推荐用insight, 因为它汇编代码显示功能更细致。

7.可惜目前在ubuntu8.04下,ddd+qemu组合用来调试驱动时有bug: 驱动函数被拦截时如果正在qemu的系统下操作, 鼠标就会冻结在qemu的屏幕中。其实调试单个驱动, 用gdb就足够了。ddd等gui一般用来调试理解内核原理。

## **gdb技巧**

另外有用的命令 ptype, whatis

- - - -

更多相关技巧:


### 1. 获取struct page结构的大小

```
(gdb) p mem_map
$80 = (struct page *) 0xc1000000
(gdb) p mem_map+1
$81 = (struct page *) 0xc1000020
(gdb) p/x 0xc1000020 - 0xc1000000
$82 = 0x20
```

### 2.

打印前从指针mem\_map所指起的5个page结构体

```
(gdb) p *mem_map@5
$83 = {{flags = 1024, _count = {counter = 1}, _mapcount = {counter = -1}, {inuse =
```



用ddd的图形显示命令是 (gdb) graph display \*mem\_map@5

参考 p \*array@len

@的左边是数组的首地址的值, 也就是变量array所指向的内容, 右边则是数据的长度, 其保存在变量len中

### 3.

每运行一次stepi/next等命令后显示下一步要将要运行的反汇编指令

```
(gdb) display/i $pc
6: x/i $pc
0xc0144fb6 <init_cgroup_root+22>:    mov    %esp,%ebp
(gdb) stepi
6: x/i $pc
0xc0144fb8 <init_cgroup_root+24>:    mov    %edx,0x44(%eax)
```

提示：display的管理：

undisplay delete display disable display enable display info display

#### 4.使结构体的显示更漂亮

```
(gdb) show print pretty
Prettyprinting of structures is on.
(gdb) set print pretty off
(gdb) p *init_task->group_info
$12 = {ngroups = 0, usage = {counter = 14}, small_block = {0 <repeats 32 times>}, r
(gdb) set print pretty on
(gdb) p *init_task->group_info
$13 = {
  ngroups = 0,
  usage = {
    counter = 14
  },
  small_block = {0 <repeats 32 times>},
  nblocks = 0,
  blocks = 0xc0355530
}
< >
```

(注：6.7.条来自[http://techcenter.dicder.com/2006/0906/content\\_173.html](http://techcenter.dicder.com/2006/0906/content_173.html))

#### 5. 使用自定义命令。

```
(gdb) define nid

Type commands for definition of "nid".

End with a line saying just "end".

>ni

>disassemble $pc $pc+16
```

>end

6. 纯gdb的多窗口显示 GUI调试器可以同时打开多个小窗口，分别显示寄存器、汇编和源代码等。在gdb里也可以做到，但同时最多只能显示两个窗口，试了一下也很方便的。基本命令如下：

a) `layout src` 仅显示源代码窗口。

b) `layout asm` 仅显示汇编代码窗口。

c) `layout split` 显示源代码和汇编代码窗口。

d) `layout regs` 显示寄存器和源代码窗口，或者寄存器和汇编代码窗口。

e) `layout next` 和 `layout prev` 切换窗口。

f) ctrl + L 刷新屏幕。

g) `C-x 1` 单窗口模式。

h) `C-x 2` 双窗口模式。

i) `C-x a` 回到传统模式。

7. 字符gdb中，如何在每执行一次next命令后都自动显示backtrace的内容 这个问题实际是如何一次执行多条命令。用自定义命令解决

```
(gdb) define nbt
Type commands for definition of "nbt".
End with a line saying just "end".
>next
>bt
>end
(gdb) nbt
#0  early_cpu_init () at arch/x86/kernel/cpu/common.c:626
#1  0xc0384ca9 in setup_arch (cmdline_p=0xc0379fe8)
    at arch/x86/kernel/setup_32.c:765
#2  0xc037f62e in start_kernel () at init/main.c:564
#3  0xc037f008 in i386_start_kernel () at arch/x86/kernel/head32.c:13
#4  0x00000000 in ?? ()
(gdb)
```

## 8. gdb在TUI模式下如何把光标焦点转移到command窗口，以便能用上下箭头键能快速翻出历史指令？

实际是转换“active”窗口。

`C-x o:` `ctrl+x`,接着放开这两个键，然后在按`o`（不需要+`ctrl`）

关于TUI更多信息：

[http://sourceware.org/gdb/current/onlinedocs/gdb\\_23.html#SEC236](http://sourceware.org/gdb/current/onlinedocs/gdb_23.html#SEC236)

还有组合键

`C-x C-a`

`C-x a`

`C-x A` 退出TUI模式

`C-x 1` 只用一个窗口

`C-x 2` 用两个窗口，按多次会有不同两个窗口的组合形式

`C-x o active` 窗口转移

`C-x s` 进入和退出TUI SingleKey 模式

注：`C-x o`多次使用相当于依次执行以下命令

`focus src` 转移焦点到源码窗口。

`focus asm`

`focus regs`

`focus cmd`

TUI模式还有以下专用命令

`info win`

`layout next`

`layout prev`

`layout src`

`layout asm`

`layout split`

`layout regs`

`focus next`

`refresh`

`tui reg float`

`tui reg general`

`tui reg next`

`tui reg system`

`update`

`winheight name +count`

`winheight name -count`

`tabset nchars`

## 9. 如何在子函数调用和退出时都暂停运行 `watch $ebp`

## 10. 如何获取结构体中特定域的相对偏移量，比如`struct stak_struct`中`lock_depth`的相对偏移量？



```
(gdb) p/x &(*(struct task_struct *)0).lock_depth
$7 = 0x14
```

11. 如何能够交换使用ddd与gdb，也就是说使用ddd调试时，想换回使用纯gdb，同时保证启用gdb后保证“调试上下文”没有任何变化？

只要.gdbinit 文件没包含 c, next..等等能驱动gdb继续调试的命令就可以。

12. 如何通过函数名确定所在的源文件

```
(gdb) info line vfs_mkdir
Line 2131 of "fs/namei.c" starts at address 0xc017c048 <vfs_mkdir> and ends at 0xc017c048
< | >
```

13. 由汇编指令地址确定该指令所对应源码的所在行(注：一行c语言一般对应几行汇编指令)

info line \*xxxxxxx (xxx是汇编指令地址)

14. 如何快速定位函数中某句C语句对应汇编指令的开始地址。比如以下 [内容太大，准备移到其他位置]

```
2130     int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
2131     {
2132     .....
2145         DQUOT_INIT(dir);
2146         error = dir->i_op->mkdir(dir, dentry, mode);//<-我们想确定这句语句的
2147         if (!error)
2148             fsnotify_mkdir(dir, dentry);
2149         return error;
2150     }
```

首先，通过函数名查询对应的源文件

```
(gdb) info line vfs_mkdir
Line 2131 of "fs/namei.c" starts at address 0xc017c048 <vfs_mkdir> and ends at 0xc017c048
< | >
```

然后，利用info line 源文件:目标语句的行数 就能查询到

```
(gdb) info line fs/namei.c:2146
Line 2146 of "fs/namei.c" starts at address 0xc017c0ee <vfs_mkdir+166> and ends at
```

## 验证一下

```
(gdb) disass 0xc017c0ee
Dump of assembler code for function vfs_mkdir:
0xc017c048 <vfs_mkdir+0>:      push    %ebp
.....
0xc017c0e4 <vfs_mkdir+156>:    mov     0x24(%eax),%ecx
0xc017c0e7 <vfs_mkdir+159>:    or      $0xffffffff,%edx
0xc017c0ea <vfs_mkdir+162>:    mov     %esi,%eax
0xc017c0ec <vfs_mkdir+164>:    call    *(%ecx)
0xc017c0ee <vfs_mkdir+166>:    mov     0x98(%esi),%ebx //
0xc017c0f4 <vfs_mkdir+172>:    mov     %edi,%edx      //参数 dentry -> %edx
0xc017c0f6 <vfs_mkdir+174>:    mov     %esi,%eax      //参数dir -> %eax
0xc017c0f8 <vfs_mkdir+176>:    mov     -0x10(%ebp),%ecx //参数mode -> %ecx
0xc017c0fb <vfs_mkdir+179>:    call    *0x14(%ebx)    //dir->i_op->mkdir(dir, der
0xc017c0fe <vfs_mkdir+182>:    test    %eax,%eax      //判断返回值(error = dir->i_
0xc017c100 <vfs_mkdir+184>:    mov     %eax,%ebx      //保存返回值
0xc017c102 <vfs_mkdir+186>:    jne     0xc017c15d <vfs_mkdir+277> //如果返回值 != 0
0xc017c104 <vfs_mkdir+188>:    testb   $0x4,0x11c(%esi) //内联函数fsnotifi
//static inline void inode_dir_notify(struct inode *inoc
//{
//    if (inode->i_dnotify_mask & (event)) <-注意这里判断什

0xc017c10b <vfs_mkdir+195>:    je      0xc017c119 <vfs_mkdir+209>
.....
0xc017c15d <vfs_mkdir+277>:    lea     -0xc(%ebp),%esp
0xc017c160 <vfs_mkdir+280>:    mov     %ebx,%eax
```

我们通过mkdir参数个数，及testb 指令基本判定我们的猜测没错。也就是说vfs\_mkdir函数中dir→i\_op→mkdir的实际调用是在0xc017c0fb <vfs\_mkdir+179>: call \*0x14(%ebx)

## 15. 下断点的形式

1. b 函数名
2. b \*指令地址
3. b 源码:行数

```
(gdb) b fs/namei.c:2146
```

```
Breakpoint 9 at 0xc017c0ee: file fs/namei.c, line 2146.
```

## 16. 陷入循环语句后，想自动运行到循环语句结束：

u

## 17. 重复当前的gdb指令

按enter键即可

### **gdb宏**

本小节意义：为了方便把调试内容复制出来，而又需要一定的功能，本人经常使用的工具是gdb的tui。所以gdb宏的使用更是成了不可缺少的辅助手段。比如extendinstr宏，能实时显示调用链的情况，相当于实现了ddd的backtrace分窗口。其他宏的作用就不说了。

### **参考资料**

kgdb官方的gdb宏 <http://kgdb.linsyssoft.com/downloads.htm>

“Fun with strace and the GDB Debugger” <http://www.ibm.com/developerworks/aix/library/au-unix-strace.html>

“GNU Project Debugger: More fun with GDB” <http://www.ibm.com/developerworks/aix/library/au-gdb.html>

“14.3.4. Useful Kernel gdb Macros” from “Embedded Linux Primer” <http://book.opensourceproject.org.cn/embedded/embeddedprime/>

### **gdb宏的使用**

假设要使用下节的lsmod，该gdb宏能列举内核中的模块。在内核源码目录下建立一个新文件lsmod,内容见下节。

装载宏

```
(gdb) source lsmod
```

查看说明

```
(gdb) help lsmod
```

```
list module struct's address, text address and their module name
```

使用

```
(gdb) lsmod
```

```
(gdb) lsmod
```

Address	text	Module
0xE014DDA0	0xE014D000	nls_iso8859_1

```

0xE0169AE0      0xE0164000      isofs
0xE014BA20      0xE0148000      zlib_inflate
0xE0161FE0      0xE0152000      udf
.....
0xE0012DE0      0xE000B000      processor
0xE0008EA0      0xE0008000      fan
0xE00223E0      0xE0020000      thermal_sys
----end----
(gdb)

(gdb)

```

我们查看一下processor模块结构体的内容

```

(gdb) p *(struct module *)0xE0012DE0
$10 = {
  state = MODULE_STATE_LIVE,
  list = {
    next = 0xe0008ea4,
    prev = 0xe0018984
  },
  name = "processor", '\0' <repeats 50 times>,
  mkobj = {
    kobj = {
      name = 0xd5910ba0 "processor",
      kref = {
        refcount = {
          counter = 3
        }
      },
      entry = {
        next = 0xe00189d0,
        ...
        ...

```

为了方便查看该结构中指针域所指向的结构体，可在ddd下用以下命令打开数据图形然后展开查看  
(gdb) graph display \*(struct module \*)0xE0012DE0

## 实例

给出的例子都在2.6.26内核上测试通过。

## 链表遍历类

宏名: lsmod(有小bug,饭后再看)

作用: 列举内核模块的名称及对应模块结构体的地址，以及text段的地址 [todo，导出.bss,.data地址]

```

define lsmod
    printf "Address\t\ttext\t\tModule\n"

```

```

set $m=(struct list_head *)&modules
set $done=0
#获取结构体内特定域的相对偏移, 见"gdb技巧"
set $offset=&(*(struct module *)0).list
while ( !$done )
    set $mp=(struct module *)((char *)$m->next - (char *)$offset)
    printf "0x%X\t0x%X\t%s\n", $mp, $mp->module_core,$mp->name
    if ( $mp->list->next == &modules)
        set $done=1
    end
    set $m=$m->next
end
printf "----end----\n"

end

document lsmod
list module struct's address, text address and their module name
end

```

效果如下

```

(gdb) lsmod
Address      text      Module
0xE014DDA0   0xE014D000 nls_iso8859_1
0xE0169AE0   0xE0164000 isofs
0xE014BA20   0xE0148000 zlib_inflate
0xE0161FE0   0xE0152000 udf
.....
0xE001BEA0   0xE001A000 8390
0xE017EEC0   0xE016C000 ide_core
0xE0018980   0xE0015000 thermal
0xE0012DE0   0xE000B000 processor
0xE0008EA0   0xE0008000 fan
0xE00223E0   0xE0020000 thermal_sys
----end----

```

宏名: psusr,pskern

作用: 列举所有task的结构地址, 状态, PID, PPID, comm。

psusr,只列举用户层可见的进程; pskern, 列举内核层可见的所有进程。

```

define __show_state
    if ($arg0->state == 0)
        printf "running\t\t"
    else
        if ($arg0->state == 1)
            printf "sleeping\t"

```

```

        else
        if ($arg0->state == 2)
            printf "disksleep\t"
        else
            if ($arg0->state == 4)
                printf "zombie\t"
            else
                if ($arg0->state == 8)
                    printf "stopped\t"
                else
                    if ($arg0->state == 16)
                        printf "wpaging\t"
                    else
                        printf "%d\t\t",
                    end
                end
            end
        end
    end
end
end
end
end
document __show_state
internal macro, don't call it by hand
end

define psusr
    printf "address\t\tstate\t\tuid\tpid\tppid\tcomm\n"
    set $init_t = &init_task
    set $tasks_off=((size_t)&((struct task_struct *)0)->tasks)
    set $next_t=((char *)($init_t->tasks).next) - $tasks_off

    while ($next_t != $init_t)
        set $next_t=(struct task_struct *)$next_t
        printf "0x%08X\t", $next_t
        __show_state $next_t
        printf "%d\t%d\t%d\t%s\n", \
            $next_t->uid, $next_t->pid, \
            $next_t->parent->pid, $next_t->comm
        set $next_t=(char *)($next_t->tasks.next) - $tasks_off
    end

    printf "address\t\tstate\t\tuid\tpid\tppid\tcomm\n"
    printf "----end----\n"

end
document psusr
print information for all tasks, but not including thread members.
This command looks like "ps -aux" in userspace.
end

define pskern
    printf "address\t\tstate\t\tuid\tpid\tppid\tcomm\n"

```

```

set $init_t = &init_task
printf "0x%08X\t", $init_t
__show_state $init_t
printf "%d\t%d\t%d\t%s\n", \
    $init_t->uid, $init_t->pid, \
    $init_t->parent->pid, $init_t->comm

set $tasks_off=((size_t)&((struct task_struct *)0)->tasks)
set $thread_off=((size_t)&((struct task_struct *)0)->thread_group.next)
set $next_t=(((char *)($init_t->tasks).next) - $tasks_off)

while ($next_t != $init_t)
    set $next_t=(struct task_struct *)$next_t

    printf "0x%08X\t", $next_t
    __show_state $next_t
    printf "%d\t%d\t%d\t%s\n", \
        $next_t->uid, $next_t->pid, \
        $next_t->parent->pid, $next_t->comm

    set $next_th=(((char *)$next_t->thread_group.next) - $thread_off)

    while ($next_th != $next_t)
        set $next_th=(struct task_struct *)$next_th

        printf "0x%08X\t", $next_th
        __show_state $next_th
        printf "%d\t%d\t%d\t%s\n", \
            $next_th->uid, $next_th->pid, \
            $next_th->parent->pid, $next_th->comm

        set $next_th=(((char *)$next_th->thread_group.next) - $thread_off)
    end

    set $next_t=(char *)($next_t->tasks.next) - $tasks_off
end

printf "address\t\tstate\t\tuid\t\tpid\t\tppid\t\tcomm\n"
printf "-----end-----\n"

```

end

document pskern

**print** infor **for** all tasks viewed **in** kernel, including all thread members  
and swapper(PID==0).

end

效果如下

(gdb) source ps

(gdb) psusr

address	state	uid	pid	ppid	comm
0xDC43F8A0	sleeping	0	1	0	init





```

0xDC486600    pipefs
0xDC486000    anon_inodefs
0xD58C5A00    tmpfs
0xD58C5200    inotifyfs
0xD8C09800    devpts
0xD8C09600    hugetlbfs
0xD8C09400    mqueue
0xD590E000    tmpfs
0xD59E4C00    hda1
0xD5908A00    tmpfs
0xD7753200    tmpfs
0xDBD66400    hdc
----end----

```

## 功能增强类

宏名： eih, lih, ooi

作用： 克服时钟中断干扰与中断无关的目标代码的调试(X86下适用)，解释请看“工程方法”

说明： 使用gdb或ddd时，进入中断后用finish命令的话常常是要么无法返回被中断的原指令处后停住，而是继续运行，要么是会进入到另一个时钟中断中；但是好像在insight下没这个问题。使用这个gdb宏可以解决该问题。

```

define eih
b common_interrupt
b native_iret
end

document eih
eih: early interrupt hacking, break common_interrupt and native_iret
end

define lih
b apic_timer_interrupt
b irq_return
end

document lih
lih: late interrupt hacking, break apic_timer_interrupt and irq_return
end

define ooi
c
stepi
end

```

```
document ooi
ooi: out of interrupt, return to the instruction interrupted by interrupt handler
end
```

宏名： extendinstr

作用： 扩展指令集。配合gdb自带的tui使用，能代替ddd等界面工具的部分功能。

说明： 指令开头:s→step,si→stepi,n→next,ni→nexti,中间bt→bt,末尾i→info args & info local

```
define inar
printf "-----args start----\n"
info args
end

define inlo
printf "-----local start----\n"
info local
end

define btl
printf "-----\n"
bt
end

define sibt
stepi
btl
end

define sbt
step
btl
end

define nibt
nexti
btl
end

define nbt
next
btl
end

define sibti
inar
```

```
inlo
stepi
btl
end

define sbti
inar
inlo
step
btl
end

define nibti
inar
inlo
nexti
btl
end

define nbti
inar
inlo
next
btl
end
```

## 效果

宏名： quick

作用： 超级快捷键。gdb的快捷键并没用用尽所有的按键。我们可以利用空余的按键定义自己的命令。方便起见，我只是利用自定义命令简单的实现该该功能，而不是自定义快捷键。可以根据自己偏好来定义。

说明： 这个宏是配合前面的宏ooi和宏extendinstr使用的。这样，如果调试时进入了时钟中断，按a + enter就可以瞬间返回；q + enter->sibt; z + enter->finish。

```
define a
ooi
end

define q
sibt
end

define z
```



```

        if ($arg0->state == 8)
            printf "stopped\t"
        else
            if ($arg0->state == 16)
                printf "wpaging\t"
            else
                printf "%d\t\t",
            end
        end
    end
end
end
end
end
end
document __show_state
internal macro, don't call it by hand
end

define psusr
    printf "address\t\tstate\t\tuid\tpid\tppid\tcomm\n"
    set $init_t = &init_task
    set $tasks_off=((size_t)&((struct task_struct *)0)->tasks)
    set $next_t=((char *)($init_t->tasks).next) - $tasks_off

    while ($next_t != $init_t)
        set $next_t=(struct task_struct *)$next_t
        printf "0x%08X\t", $next_t
        __show_state $next_t
        printf "%d\t%d\t%d\t%s\n", \
            $next_t->uid, $next_t->pid, \
            $next_t->parent->pid, $next_t->comm
        set $next_t=(char *)($next_t->tasks.next) - $tasks_off
    end

    printf "address\t\tstate\t\tuid\tpid\tppid\tcomm\n"
    printf "----end----\n"

end
document psusr
print information for all tasks, but not including thread members.
This command looks like "ps -aux" in userspace.
end

define pskern
    printf "address\t\tstate\t\tuid\tpid\tppid\tcomm\n"
    set $init_t = &init_task
    printf "0x%08X\t", $init_t
    __show_state $init_t
    printf "%d\t%d\t%d\t%s\n", \
        $init_t->uid, $init_t->pid, \
        $init_t->parent->pid, $init_t->comm

```

```

set $tasks_off=((size_t)&((struct task_struct *)0)->tasks)
set $thread_off=((size_t)&((struct task_struct *)0)->thread_group.next)
set $next_t=(((char *)($init_t->tasks).next) - $tasks_off)

while ($next_t != $init_t)
    set $next_t=(struct task_struct *)$next_t

    printf "0x%08X\t", $next_t
    __show_state $next_t
    printf "%d\t%d\t%d\t%s\n", \
        $next_t->uid, $next_t->pid, \
        $next_t->parent->pid, $next_t->comm

    set $next_th=(((char *)$next_t->thread_group.next) - $thread_off)

    while ($next_th != $next_t)
        set $next_th=(struct task_struct *)$next_th

        printf "0x%08X\t", $next_th
        __show_state $next_th
        printf "%d\t%d\t%d\t%s\n", \
            $next_th->uid, $next_th->pid, \
            $next_th->parent->pid, $next_th->comm

        set $next_th=(((char *)$next_th->thread_group.next) - $thread_off)
    end

    set $next_t=(char *)($next_t->tasks.next) - $tasks_off
end

printf "address\t\tstate\t\tuid\t\tpid\t\tppid\t\tcomm\n"
printf "----end----\n"

```

```

end
document pskern
print infor for all tasks viewed in kernel, including all thread members
and swapper(PID==0).
end

```

```

define __prinfo_nobp
    printf "\npid %d; addr:0x%08x; comm %s:\n", \
        $arg0.pid, $arg0, $arg0.comm
    printf "=====\n"
    set var $stackp = $arg0.thread.sp
    set var $stack_top = ($stackp & ~4095) + 4096

    while ($stackp < $stack_top)
        if (*( $stackp) > _stext && *( $stackp) < _sinittext)
            info symbol *( $stackp)
        end
        set $stackp += 4
    end
end

```

```

document __prinfo_nobp
internal macro, don't call it by hand.
end

define bttnobp
    set $tasks_off=((size_t)&((struct task_struct *)0)->tasks)
    set $thread_off=((size_t)&((struct task_struct *)0)->thread_group.next)
    set $init_t=&init_task
    set $next_t=(((char *)($init_t->tasks).next) - $tasks_off)

    while ($next_t != $init_t)
        set $next_t=(struct task_struct *)$next_t
        __prinfo_nobp $next_t
        set $next_th=(((char *)$next_t->thread_group.next) - $thread_off)
        while ($next_th != $next_t)
            set $next_th=(struct task_struct *)$next_th
            __prinfo_nobp $next_th
            set $next_th=(((char *)$next_th->thread_group.next) - $thread_off)
        end
        set $next_t=(char *)($next_t->tasks.next) - $tasks_off
    end
end
document bttnobp
    dump all thread stack traces on a kernel compiled with !CONFIG_FRAME_POINTER
end

define __prinfo
    printf "\npid %d; addr:0x%08x; comm %s:\n", \
        $arg0.pid, $arg0, $arg0.comm
    printf "=====\n"
    set var $stackp = $arg0.thread.sp
    set var $stack_top = ($stackp & ~4095) + 4096
    set var $stack_bot = ($stackp & ~4095)

    set $stackp = *($stackp)
    while (($stackp < $stack_top) && ($stackp > $stack_bot))
        set var $addr = *($stackp + 4)
        info symbol $addr
        set $stackp = *($stackp)
    end
end
document __prinfo
internal macro, don't call it by hand.
end

define btt
    set $tasks_off=((size_t)&((struct task_struct *)0)->tasks)
    set $thread_off=((size_t)&((struct task_struct *)0)->thread_group.next)
    set $init_t=&init_task
    set $next_t=(((char *)($init_t->tasks).next) - $tasks_off)

```

```

while ($next_t != $init_t)
    set $next_t=(struct task_struct *)$next_t
    __prinfo $next_t
    set $next_th=(((char *)$next_t->thread_group.next) - $thread_off)
    while ($next_th != $next_t)
        set $next_th=(struct task_struct *)$next_th
        __prinfo $next_th
        set $next_th=(((char *)$next_th->thread_group.next) - $thread_off)
    end
    set $next_t=(char *)($next_t->tasks.next) - $tasks_off
end
end
document btt
    dump all thread stack traces on a kernel compiled with CONFIG_FRAME_POINTER
end

define btpid
    set var $pid = $arg0
    set $tasks_off=((size_t)&((struct task_struct *)0)->tasks)
    set $thread_off=((size_t)&((struct task_struct *)0)->thread_group)
    set $init_t=&init_task
    set $next_t=(((char *)($init_t->tasks).next) - $tasks_off)
    set var $pid_task = 0

    while ($next_t != $init_t)
        set $next_t=(struct task_struct *)$next_t

        if ($next_t.pid == $pid)
            set $pid_task = $next_t
        end

        set $next_th=(((char *)$next_t->thread_group.next) - $thread_off)
        while ($next_th != $next_t)
            set $next_th=(struct task_struct *)$next_th
            if ($next_th.pid == $pid)
                set $pid_task = $next_th
            end
            set $next_th=(((char *)$next_th->thread_group.next) - $thread_off)
        end
        set $next_t=(char *)($next_t->tasks.next) - $tasks_off
    end

    __prinfo $pid_task

end
document btpid
    backtrace of pid
end

define trapinfo
    set var $pid = $arg0
    set $tasks_off=((size_t)&((struct task_struct *)0)->tasks)
    set $thread_off=((size_t)&((struct task_struct *)0)->thread_group.next)

```



```

set $init_t=&init_task
set $next_t=(((char *)($init_t->tasks).next) - $tasks_off)
set var $pid_task = 0

while ($next_t != $init_t)
    set $next_t=(struct task_struct *)$next_t

    if ($next_t.pid == $pid)
        set $pid_task = $next_t
    end

    set $next_th=(((char *)$next_t->thread_group.next) - $thread_off)
    while ($next_th != $next_t)
        set $next_th=(struct task_struct *)$next_th
        if ($next_th.pid == $pid)
            set $pid_task = $next_th
        end
        set $next_th=(((char *)$next_th->thread_group.next) - $thread_off)
    end
    set $next_t=(char *)($next_t->tasks.next) - $tasks_off
end

printf "Trapno %ld, cr2 0x%lx, error_code %ld\n", $pid_task.thread.trap_no,
    $pid_task.thread.cr2, $pid_task.thread.error_code

end
document trapinfo
    Run info threads and lookup pid of thread #1
    'trapinfo <pid>' will tell you by which trap & possibly
    address the kernel panicked.
end

define dmesg
    set $i = 0
    set $end_idx = (log_end - 1) & (log_buf_len - 1)

    while ($i < logged_chars)
        set $idx = (log_end - 1 - logged_chars + $i) & (log_buf_len - 1)

        if ($idx + 100 <= $end_idx) || \
            ($end_idx <= $idx && $idx + 100 < log_buf_len)
            printf "%.100s", &log_buf[$idx]
            set $i = $i + 100
        else
            printf "%c", log_buf[$idx]
            set $i = $i + 1
        end
    end
end
document dmesg
    print the kernel ring buffer
end

```

宏名: vmap, lsvmaps, lsmode, lsmodesects, lsallmodesects

说明: 没测试, 待更新

来源 <http://jeanmarc.saffroy.free.fr/kdump2gdb/>

```
# Copyright Jean-Marc Saffroy <saffroy@gmail.com> 2006
# This program is free software, distributed under the terms of the
# GNU General Public License version 2.

# a few useful(?) macros for x86-64 VMM hacks

# useful constants
set $PAGE_SIZE = (1<<12)
set $__PHYSICAL_MASK = (1 << 46)-1
set $PTE_MASK = ~($PAGE_SIZE-1) & $__PHYSICAL_MASK
set $__PAGE_OFFSET = 0xfffff81000000000
set $_PAGE_PSE = 0x80

define vmap
    set $addr = (long)$arg0
    # index in each of the 4 levels of page directories
    set $pgd = $addr >> 39 & (1<<9)-1
    set $pud = $addr >> 30 & (1<<9)-1
    set $pmd = $addr >> 21 & (1<<9)-1
    set $pte = $addr >> 12 & (1<<9)-1
    # offset in page
    set $off = $addr & (1<<12)-1
    #printf "%03x %03x %03x %03x %03x\n", $pgd, $pud, $pmd, $pte, $off

    set $pgd_off = (pgd_t *) &init_level4_pgt + $pgd
    #printf "pgd_off: %lx pgd: %lx\n", $pgd_off, (long)$pgd_off->pgd
    set $pgd_page = ((long)$pgd_off->pgd & $PTE_MASK) + $__PAGE_OFFSET
    #printf "pgd_page: %lx\n", $pgd_page

    set $pud_off = ((pud_t *) $pgd_page) + $pud
    #printf "pud_off: %lx pud: %lx\n", $pud_off, (long)$pud_off->pud
    set $pud_page = ((long)$pud_off->pud & $PTE_MASK) + $__PAGE_OFFSET
    #printf "pud_page: %lx\n", $pud_page

    set $pmd_off = ((pmd_t *) $pud_page) + $pmd
    #printf "pmd_off: %lx pmd: %lx\n", $pmd_off, (long)$pmd_off->pmd
    set $pmd_page = ((long)$pmd_off->pmd & $PTE_MASK) + $__PAGE_OFFSET
    #printf "pmd_page: %lx\n", $pmd_page

    if ((long)$pmd_off->pmd & $_PAGE_PSE) != 0
        #printf "PSE page! "
        set $paddr = $pmd_page + ($addr & (1<<21)-1)
    else
        set $pte_off = ((pte_t *) $pmd_page) + $pte
        #printf "pte_off: %lx pte: %lx\n", $pte_off, (long)$pte_off->pte
```

```

        set $pte_page = ((long)$pte_off->pte & $PTE_MASK) + $__PAGE_OFFSET
        #printf "pte_page: %lx\n", $pte_page
        set $paddr = $pte_page + $off
    end
    #printf "remapped physical addr: %lx\n", $paddr
    printf "%lx -> %lx\n", $addr, $paddr
end

document vmap
Usage: vmap <address>
Convert a kernel remapped virtual address to an identity-mapped address.
end

define lsvmaps
    set $map = vmlist
    set $gcount = 0
    while $map != 0
        if $map->pages != 0
            set $vaddr = (long)$map->addr
            set $count = (long)$map->size / $PAGE_SIZE
            set $gcount = $gcount + $count - 1
            while $count > 1
                vmap $vaddr
                set $vaddr = $vaddr + $PAGE_SIZE
                set $count = $count - 1
            end
        end
        set $map = $map->next
    end
    printf "page count: %d\n", $gcount
end

document lsvmaps
List all kernel remapped pages (vmalloc regions) and corresponding identity-mapped
end

define lsmod
    set $mod = modules.next
    printf "struct module      size  name\n"
    while $mod != &modules
        set $m = (struct module *)((char*)$mod-(char*)&((struct module*)0)->list)
        printf "0x%lx % 8d %s\n", $m, $m->core_size, $m->name
        set $mod = $mod->next
    end
end

document lsmod
List loaded kernel modules.
end

define lsmodsects
    set $mod = (struct module *)$arg0
    printf "add-symbol-file %s.ko 0x%lx ", $mod->name, $mod->sect_attrs->addr
    set $i = 1

```

```

        while $mod->sect_attrs->grp->attrs[$i] != 0
            printf "-s %s ", (char*)$mod->sect_attrs->attrs[$i].name
            printf "0x%lx ", $mod->sect_attrs->attrs[$i].address
            set $i = $i + 1
        end
        printf "\n"
    end

document lsmmodsects

Usage: lsmmodsects <address of struct module>
Prints "add-symbol-file..." command to load sections of the given module.
end

define lsallmodsects
    set $mdl = modules.next
    while $mdl != &modules
        set $m = (struct module *)((char*)$mdl-(char*)&((struct module*)0)->list)
        lsmmodsects $m
        set $mdl = $mdl->next
    end
end

document lsallmodsects
Calls lsmmodsects on all modules.
end

```

## 汇编基础--X86篇

注意：某些内容不具备普遍性。比如给出的反汇编代码，在不同的优化等级下是不同的。但是在熟悉了典型的函数调用链反汇编代码，对于有变化的其他形式也就不难理解了。

### 用户手册

Intel® 64 and IA-32 Architectures Software Developer's Manuals

<http://www.intel.com/products/processor/manuals/index.htm>

### AT&T汇编格式

#### 参考

“AT&T汇编语言与GCC内嵌汇编简介” [http://blog.chinaunix.net/u2/73528/showart\\_1110874.html](http://blog.chinaunix.net/u2/73528/showart_1110874.html)

[杂类文章]

“Linux Assembly and Disassembly an Introduction” <http://www.milw0rm.com/papers/47>

## 内联汇编

GCC-Inline-Assembly-HOWTO <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>

## 汇编与C函数的相互调用

### 调用链形成和参数传递

参考文章 [多如牛毛]

“Guide: Function Calling Conventions” <http://www.delorie.com/djgpp/doc/ug/asm/calling.html>

“Intel x86 Function-call Conventions - Assembly View” <http://www.unixwiz.net/techtips/win32-callconv-asm.html>

“C Function Call Conventions and the Stack” <http://www.cs.umbc.edu/~chang/cs313.s02/stack.shtml>

“The C Calling Convention and the 8086: Using the Stack Frame” <http://www.et.byu.edu/groups/ece425web/stable/labs/StackFrame.html>

“C Function Calling Convention” <http://adamw-dev.blogspot.com/2007/05/c-function-calling-convention.html>

“C函数调用在GNU汇编中的实现” <http://www.unixresources.net/linux/clf/cpu/archiver/00/00/59/75/597564.html>

“函数调用的几个概念：\_stdcall, \_cdecl...” [http://blog.chinaunix.net/u2/67530/showart\\_601750.html](http://blog.chinaunix.net/u2/67530/showart_601750.html)

“Calling conventions(调用规则)” [http://www.bobd.cn/itschool/Program/delphi/200612/itschool\\_12084.html](http://www.bobd.cn/itschool/Program/delphi/200612/itschool_12084.html)

[扩展，简要说明原理。并用实例解析]

x86终极参考

CHAPTER 6 PROCEDURE CALLS, INTERRUPTS, AND EXCEPTIONS of

IA-32 Intel\_ Architecture Software Developer's Manual Volume 1\_ Basic Architecture.pdf <http://download.intel.com/design/processor/manuals/253665.pdf>

### 寄存器的角色与保护

- 寄存器的角色

#### 1. %esp: 栈指针

指向栈的顶端，也就是指向栈的最后一个正在使用的元素。%esp的值隐式地受到几个机器指令的影响，比如push,pop,call,ret等。

#### 2. %ebp: 基址指针

指向当前栈的基地址，有时也称为“帧指针”。与%esp不同的是，它必须显式地进行操作才能改变值。

#### 3. %eip: 指令指针

保存着下一个被执行机器指令的地址。当CPU执行call指令时，%eip的值自动被保存到栈中。还有，任何一个“jump”跳转指令都会直接地改变%eip

- 两条规则

1. gcc要求在函数调用的前后，寄存器%ebx, %esi, %edi, %ebp, %esp,%ds, %es, %ss的值保持不变。所以被调用函数如果需要修改这些寄存器的值，被调用函数必须负责对它们进行保护。 [后三个??]

2. gcc规定在函数调用的前后，寄存器%eax,%edx,%ecx的值可以改变。所以调用函数如果需要防止子函数破坏这三个寄存器的值，调用者必须在函数调用前自己负责保护它们。

我们注意到，是保护，不一定是保存。如果确认没用到某寄存器，那么该寄存器就不需要一定要有一个先保存到栈而后再恢复原值的过程。

这两条规则实际是定义了对系统资源使用的权限和义务。

第一条规则，是银行和借贷者的关系。有人向银行借了几千万，结果赌博全输光了。还钱的期限到了，银行的行长对借贷者说“没事，你回家吧。几千万而已，我拿我工资给你垫上”。我想这样的事决不会发生，行长一个电话110过去，借贷者一天后就把钱还清了。所以，这里，调用函数是银行行长，子函数是借贷者。

第二条规则，则是老爸和儿子的关系了。儿子对老爸说“老爸，解我100去买球鞋，我明天还你”。结果，第二天，老爸没钱吃饭了，问儿子“还钱”。儿子说“昨晚逛街碰到一个美女，请了一顿，把钱化光了”。老爸没法子，总不能把儿子绳以正法吧。怪只能怪自己事前没防这招咯。所以，这里，调用函数是老爸，子函数是儿子你。

#### • 返回值

1. Integers (of any size up to 32 bits) and pointers are returned in the %eax register.
2. Floating point values are returned in the 387 top-of-stack register, st(0).
3. Return values of type long long int are returned in %edx:%eax (the most significant 64 bits in %edx).
4. Returning a structure is complicated and rarely useful; try to avoid it. (Note that returning a structure is not supported by gcc 2.95.4 and earlier.)
5. If your function returns void (e.g. no value), the contents of these registers are undefined.

### 调用链的形成

#### • 应用层实例解析

我们回头看看“寄存器的角色”这一小节，很快就能明白调用链的形成的本质。

调用链包含两方面的内容

1.返回地址的保存与恢复

2.旧栈帧的保存与恢复

因为在普通的调用形式中(call调用), 返回地址的保存与恢复是由处理器机制本身保证的, 不需人工维护。调用指令call的执行自动将call指令之下的指令地址压入栈中, 被调用函数返回时, ret指令的执行会重新将返回地址从栈弹出传送到pc中。要求下面分析旧栈帧的保存与恢复。

旧栈帧的保存与恢复, 无非就是要解决两大问题:

1. 建立新栈帧 这一步很简单, 栈帧无非有两个头, 底端和顶端。%esp指向栈的顶端, 而%esp是不需要手工维护的, 随着push,pop等指令, 它自己就在改变自己。那么又怎么建立栈帧的底端呢? 我们知道, 栈底(也就是基址)是由%ebp指定的, 在一个栈帧的整个生命周期里, %ebp的值都不变, 也就是说, 赋个合适的值给它就完事。怎么赋值就是问题所在了。我们知道, %esp指向栈中最后一个被使用的元素。所以, 当我们正在使用(我们认为的)第一个元素时, 把%esp的值赋给%ebp,%ebp不就是指向栈的基址了吗?

2. 保护旧栈帧的信息 同样的问题, 保护旧栈帧的信息, 就是保存旧栈帧指向底端和顶端的指针值, 也就是旧%ebp,%esbp的值。当函数调用指令刚执行完, 马上就要保护作案现场了。首先, push %ebp,这句就把旧栈帧的基地址保存在栈的顶端。此时, %esp指向的内存地址中, 就放着旧栈帧的基地址的值。但是还不够啊, %esp是个不可靠的东西, 它经常在变化, 必须把这个地址放到一个不会隐式变化的寄存器中。于是选择了%ebp。mov %esp %ebp.这样, %ebp指向的内存地址中, 就放着旧栈帧的基地址的值。这就解放了%esp, 可以用%esp来动态指向新栈帧的顶端了。按照定义, %ebp所指向的地址是新栈帧的底端, 也就是新栈帧的第一个元素, 也就是说新栈帧第一个元素的值是旧栈帧基址。

但是注意, %ebp指向的地址再加4bytes的地址上, 存放的是被调用函数的返回地址。在执行call指令时, call指令后面的那个指令的地址(也就是被调用函数的返回地址)被自动隐式地放到了栈中。

当子函数返回时, 再按照上面文字进行逆操作, 就能恢复旧栈帧的信息。

```
#include <stdio.h>

void func()
{}

void funb()
{
```



```

    func();
}

```

```

void funa()
{
    funb();
}

```

```

int main()
{
    funa();
}

```

```

-----
08048344 <func>:
#include <stdio.h>

```

```

void func()
{}
8048344:    55                push    %ebp
8048345:    89 e5             mov     %esp,%ebp
8048347:    5d                pop     %ebp
8048348:    c3                ret

```

```

08048349 <funb>:

```

```

void funb()
{
8048349:    55                push    %ebp
804834a:    89 e5             mov     %esp,%ebp
    func();
804834c:    e8 f3 ff ff ff   call    8048344 <func>
}

```

```

8048351:    5d                pop     %ebp
8048352:    c3                ret

```

```

08048353 <funa>:

```

```

void funa()
{
8048353:    55                push    %ebp
8048354:    89 e5             mov     %esp,%ebp
    funb();
8048356:    e8 ee ff ff ff   call    8048349 <funb>
}
804835b:    5d                pop     %ebp
804835c:    c3                ret

```

```

0804835d <main>:

```

```

int main()

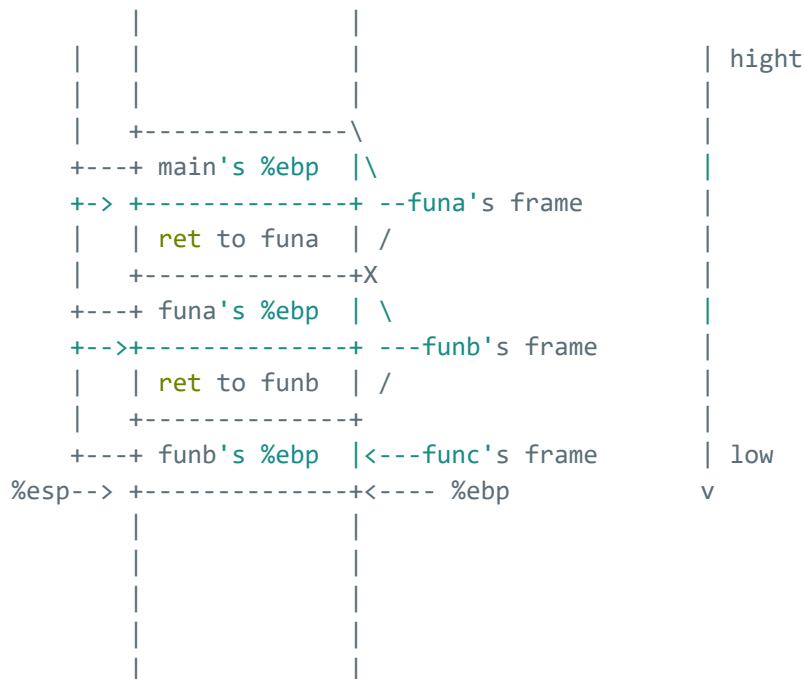
```

```

{
804835d:      8d 4c 24 04      lea    0x4(%esp),%ecx
8048361:      83 e4 f0          and    $0xffffffff0,%esp
8048364:      ff 71 fc          pushl  -0x4(%ecx)
8048367:      55              push   %ebp
8048368:      89 e5            mov    %esp,%ebp
804836a:      51              push   %ecx
      funa();
804836b:      e8 e3 ff ff      call   8048353 <funa>
}
8048370:      59              pop     %ecx
8048371:      5d              pop     %ebp
8048372:      8d 61 fc          lea    -0x4(%ecx),%esp
8048375:      c3              ret
8048376:      90              nop
8048377:      90              nop
8048378:      90              nop
8048379:      90              nop
804837a:      90              nop
804837b:      90              nop
804837c:      90              nop
804837d:      90              nop
804837e:      90              nop
804837f:      90              nop

```

func被调用后内存如下



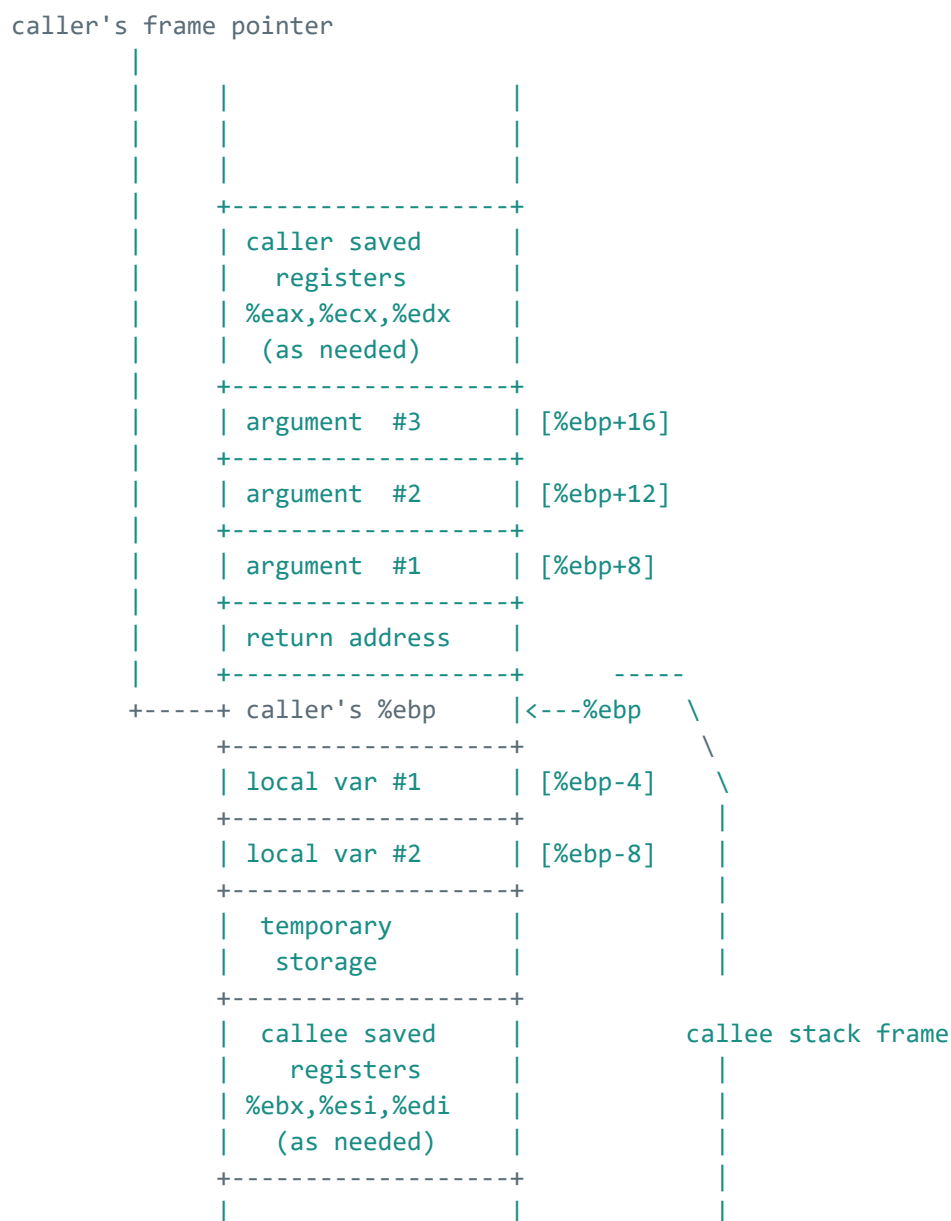
## • 内核层实例解析

## 栈帧结构与参数传递

- 栈元素引用的就近原则

为了说明就近原则，我们先看看典型和全面的栈帧是怎样的。函数caller调用子函数callee所形成的栈帧。

1. 从被调用的子函数callee来看，获取caller的传递的实参，以及建立自身本地变量时，因为内存地址都靠近栈帧的基址，所以这两种引用都是利用%ebp加上偏移量的形式。
2. 相反，主函数在调用子函数前，在为子函数准备实参时，因为实参位于栈帧末端，所以对实参的引用都是利用%esp加上偏移量的形式(没画出来)





caller:调用者 callee:被调用者

## 完整的调用过程

函数caller调用子函数callee，这是应用层的普通函数调用过程。如果是远调用，跨态调用要考虑的东西更多。但这个例子已经充分展示了调用过程的繁复部分。

- 函数调用前调用者的动作

1.%eax,%edx,%ecx入栈（可选）

2.子函数的参数入栈

- 函数调用 call callee

call机器指令，原子性自动地完成了两种任务.

1.%eip入栈, 保存了callee函数的返回地址

2.callee的函数地址传递到%eip.

所以下一指令就从callee函数的第一指令开始运行。控制权转移给callee

- 函数调用后被调用者的动作

1.保存caller栈帧基址 push %ebp

2.建立callee栈帧基址 mov %esp, %ebp

3.分配本地变量和临时存储的空间 sub \$XXX, %esp

4.本地变量赋值

5.%ebx,%esi,%edi入栈(可选)

- 调用返回前被调用者的动作

1. %ebx, %esi, %edi 还原(出栈, 可选)

2. 释放本地变量和临时存储的栈空间 `mov %ebp, %esp`

3. 还原 caller 栈帧的基址 `pop %ebp`

或者 2.3. 步用一条元语指令完成 `leave`

4. 调用返回 `ret`

该指令把存放于栈的返回地址取出(出栈), 存放到 %eip 中。下一指令就从 `call callee` 指令的下一指令开始运行。控制权返回给 caller

- 调用返回后调用者的动作

1. 释放存放 callee 参数的栈空间 `add $XXX, %esp`

2. 转移 %eax 的值 (子函数的返回值, 可选)

3. 还原 %eax, %edx, %ecx (出栈, 可选)

- 应用层实例解析

应用层参数的传入: 用户层参数的传递是利用栈来完成的。函数右边的参数先入栈, 位于栈的高地址。反之, 函数左边的参数后入栈, 位于栈的低地址。

例子请看 “C 难点的汇编解释”

- 内核层实例解析

内核层参数的传入: 混合使用寄存器和栈来传递参数。当参数个数不多于3个时, 参数从左到右依次传递到 %eax, %edx, %ecx. 当参数个数多于3时, 从第4个起的其余参数通过栈传递。同样, 函数右边的参数先入栈, 位于栈的高地址。反之, 函数左边的参数后入栈, 位于栈的低地址。

- 系统调用实例解析

系统调用的参数传递: [以后再看]

## C库函数

```
ssize_t read(int fd, void *buf, size_t count);
```

000b6a30 <\_\_read>:

```

b6a30:    65 83 3d 0c 00 00 00    cmpl    $0x0,%gs:0xc
b6a37:    00
b6a38:    75 1d                    jne     b6a57 <__read+0x27>
b6a3a:    53                      push    %ebx
b6a3b:    8b 54 24 10             mov     0x10(%esp),%edx    //count
b6a3f:    8b 4c 24 0c             mov     0xc(%esp),%ecx    //buf
b6a43:    8b 5c 24 08             mov     0x8(%esp),%ebx    //fd
b6a47:    b8 03 00 00 00          mov     $0x3,%eax        //系统调用号
b6a4c:    cd 80                  int     $0x80
b6a4e:    5b                      pop     %ebx
b6a4f:    3d 01 f0 ff ff          cmp     $0xffffffff01,%eax
b6a54:    73 2d                    jae     b6a83 <__read+0x53>
b6a56:    c3                      ret
b6a57:    e8 14 ae 01 00          call    d1870 <pthread_exit+0x110>
b6a5c:    50                      push    %eax
b6a5d:    53                      push    %ebx
b6a5e:    8b 54 24 14             mov     0x14(%esp),%edx
b6a62:    8b 4c 24 10             mov     0x10(%esp),%ecx
b6a66:    8b 5c 24 0c             mov     0xc(%esp),%ebx
b6a6a:    b8 03 00 00 00          mov     $0x3,%eax
b6a6f:    cd 80                  int     $0x80
b6a71:    5b                      pop     %ebx
b6a72:    87 04 24                xchg    %eax, (%esp)
b6a75:    e8 c6 ad 01 00          call    d1840 <pthread_exit+0xe0>
b6a7a:    58                      pop     %eax
b6a7b:    3d 01 f0 ff ff          cmp     $0xffffffff01,%eax
b6a80:    73 01                    jae     b6a83 <__read+0x53>
b6a82:    c3                      ret
b6a83:    e8 8e 5a 04 00          call    fc516 <__frame_state_for+0xb96>
b6a88:    81 c1 6c e5 07 00       add     $0x7e56c,%ecx
b6a8e:    8b 89 e0 ff ff ff       mov     -0x20(%ecx),%ecx
b6a94:    31 d2                    xor     %edx,%edx
b6a96:    29 c2                    sub     %eax,%edx
b6a98:    65 03 0d 00 00 00 00    add     %gs:0x0,%ecx
b6a9f:    89 11                    mov     %edx, (%ecx)
b6aa1:    83 c8 ff                or      $0xffffffff,%eax
b6aa4:    eb dc                    jmp     b6a82 <__read+0x52>
b6aa6:    90                      nop

```

调用号#define \_\_NR\_read 3

(gdb) disass sys\_read

Dump of assembler code for function sys\_read:

```

0xc017585a <sys_read+0>:    push    %ebp
0xc017585b <sys_read+1>:    mov     %esp,%ebp
0xc017585d <sys_read+3>:    push    %esi
0xc017585e <sys_read+4>:    mov     $0xffffffff7,%esi
0xc0175863 <sys_read+9>:    push    %ebx
0xc0175864 <sys_read+10>:   sub     $0xc,%esp

```

```

0xc0175867 <sys_read+13>:    mov     0x8(%ebp),%eax
0xc017586a <sys_read+16>:    lea     -0xc(%ebp),%edx
0xc017586d <sys_read+19>:    call   0xc0175f65 <fget_light>
0xc0175872 <sys_read+24>:    test    %eax,%eax
0xc0175874 <sys_read+26>:    mov     %eax,%ebx
0xc0175876 <sys_read+28>:    je      0xc01758b1 <sys_read+87>
0xc0175878 <sys_read+30>:    mov     0x24(%ebx),%edx
0xc017587b <sys_read+33>:    mov     0x20(%eax),%eax
0xc017587e <sys_read+36>:    mov     0x10(%ebp),%ecx
0xc0175881 <sys_read+39>:    mov     %edx,-0x10(%ebp)
0xc0175884 <sys_read+42>:    mov     0xc(%ebp),%edx
0xc0175887 <sys_read+45>:    mov     %eax,-0x14(%ebp)
0xc017588a <sys_read+48>:    lea     -0x14(%ebp),%eax
0xc017588d <sys_read+51>:    push    %eax
0xc017588e <sys_read+52>:    mov     %ebx,%eax
0xc0175890 <sys_read+54>:    call   0xc01753c1 <vfs_read>
0xc0175895 <sys_read+59>:    mov     -0x10(%ebp),%edx
0xc0175898 <sys_read+62>:    mov     %eax,%esi
0xc017589a <sys_read+64>:    mov     -0x14(%ebp),%eax
0xc017589d <sys_read+67>:    mov     %edx,0x24(%ebx)
0xc01758a0 <sys_read+70>:    mov     %eax,0x20(%ebx)
0xc01758a3 <sys_read+73>:    cmpl    $0x0,-0xc(%ebp)
0xc01758a7 <sys_read+77>:    pop     %eax
0xc01758a8 <sys_read+78>:    je      0xc01758b1 <sys_read+87>
0xc01758aa <sys_read+80>:    mov     %ebx,%eax
0xc01758ac <sys_read+82>:    call   0xc0175eae <fput>
0xc01758b1 <sys_read+87>:    lea     -0x8(%ebp),%esp
0xc01758b4 <sys_read+90>:    mov     %esi,%eax
0xc01758b6 <sys_read+92>:    pop     %ebx
0xc01758b7 <sys_read+93>:    pop     %esi
0xc01758b8 <sys_read+94>:    pop     %ebp
0xc01758b9 <sys_read+95>:    ret

```

End of assembler dump.

(gdb) list fget\_light

```

313      * holds a refcnt to that file. That check has to be done at fget() only
314      * and a flag is returned to be passed to the corresponding fput_light()
315      * There must not be a cloning between an fget_light/fput_light pair.
316      */
317      struct file *fget_light(unsigned int fd, int *fput_needed)

```

来自2.6.11

```

378 #define __syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4, \
379     type5,arg5,type6,arg6) \
380 type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5,type6 arg6
381 { \
382 long __res; \
383 __asm__ volatile ("push %%ebp ; movl %%eax,%%ebp ; movl %1,%%eax ; int $0x80
384     : "=a" (__res) \
385     : "i" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
386     "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)), \
387     "0" ((long)(arg6))); \
388 __syscall_return(type,__res); \

```

## 调用链回溯的代码实现

内核中(x86)对调用链的回溯的代码实现在文件dumpstack\_32.c文件中。主要函数是dump\_trace和print\_context\_stack.

待解释

## C难点的汇编解释

### 例1

if ... else if

这个例子有人看来也许是非常非常地简单，但就这个例子，有的人还真给我考”倒”了。他的回话是“还真没见过这样子的代码”。但是，这样的代码在内核中比比皆是,比如后面附上的函数代码 do\_path\_lookup。如果对if ... else if 理解有偏差，对内核代码的逻辑理解根本就是差以千里。

```
#include <stdio.h>

int main()
{
    int i = 1;
    int j = 2;

    if (i == 1)
        printf("i,ok\n");
    else if (j == 2)
        printf("j,ok\n");

    return 0;
}
```

这个例子，有人会疑问为什么”j,ok”没打印出来。现在我们分析下它的汇编代码

```
08048374 <main>:
8048374:      8d 4c 24 04          lea     0x4(%esp),%ecx
8048378:      83 e4 f0            and     $0xffffffff0,%esp
804837b:      ff 71 fc            pushl   -0x4(%ecx)
804837e:      55                  push    %ebp
804837f:      89 e5                mov     %esp,%ebp          //以上汇编码保护
8048381:      51                  push    %ecx              //%ecx入栈保护
8048382:      83 ec 14            sub     $0x14,%esp        //建立本地变量
8048385:      c7 45 f8 01 00 00 00 movl     $0x1,-0x8(%ebp)   //变量i赋值，记
```



8048393:	83 7d f8 01	cmpl	\$0x1, -0x8(%ebp)	//i和1比较
8048397:	75 0e	jne	80483a7 <main+0x33>	//如果i-1不等0, i
8048399:	c7 04 24 90 84 04 08	movl	\$0x8048490, (%esp)	//printf函数第一
				//记得子函数的实参
80483a0:	e8 2f ff ff ff	call	80482d4 <puts@plt>	//调用printf
80483a5:	eb 12	jmp	80483b9 <main+0x45>	//printf返回后, i
80483a7:	83 7d f4 02	cmpl	\$0x2, -0xc(%ebp)	
80483ab:	75 0c	jne	80483b9 <main+0x45>	
80483ad:	c7 04 24 95 84 04 08	movl	\$0x8048495, (%esp)	
80483b4:	e8 1b ff ff ff	call	80482d4 <puts@plt>	
80483b9:	b8 00 00 00 00	mov	\$0x0, %eax	;//%eax赋值0, %eax
80483be:	83 c4 14	add	\$0x14, %esp	;//撤销新栈帧的本址
80483c1:	59	pop	%ecx	;//恢复保存的旧%ecx
80483c2:	5d	pop	%ebp	;//以下汇编码都是堆栈
80483c3:	8d 61 fc	lea	-0x4(%ecx), %esp	
80483c6:	c3	ret		

经过上面的汇编代码分析，可见c代码块

```
else if (j == 2)
    printf("j,ok\n");
```

对应的汇编代码是：

80483a7:	83 7d f4 02	cmpl	\$0x2, -0xc(%ebp)
80483ab:	75 0c	jne	80483b9 <main+0x45>
80483ad:	c7 04 24 95 84 04 08	movl	\$0x8048495, (%esp)
80483b4:	e8 1b ff ff ff	call	80482d4 <puts@plt>

上面的代码指令根本就没有机会运行。

结论，一个if ... else if ..else..

```
if (判断语句1)
    代码块1
else if (判断语句2)
    代码块2;
else if ....

..
else 代码块N;
```

语句块1,2..N的运行机会是一种互斥的关系。当然它们的“机会优先级”是不一样的。语句块1,2..N只有一个有被运行的机会，如果没有else甚至可能没有一个语句块能被运行。

## 内核代码实例

```
static int do_path_lookup(int dfd, const char *name,
                        unsigned int flags, struct nameidata *nd)
{
    int retval = 0;
    int fput_needed;
    struct file *file;
    struct fs_struct *fs = current->fs;

    nd->last_type = LAST_ROOT; /* if there are only slashes... */
    nd->flags = flags;
    nd->depth = 0;

    if (*name=='/') {
        read_lock(&fs->lock);
        if (fs->altroot.dentry && !(nd->flags & LOOKUP_NOALT)) {
            nd->path = fs->altroot;
            path_get(&fs->altroot);
            read_unlock(&fs->lock);
            if (__emul_lookup_dentry(name, nd))
                goto out; /* found in altroot */
            read_lock(&fs->lock);
        }
        nd->path = fs->root;
        path_get(&fs->root);
        read_unlock(&fs->lock);
    } else if (dfd == AT_FDCWD) {
        read_lock(&fs->lock);
        nd->path = fs->pwd;
        path_get(&fs->pwd);
        read_unlock(&fs->lock);
    } else {
        struct dentry *dentry;

        file = fget_light(dfd, &fput_needed);
        retval = -EBADF;
        if (!file)
            goto out_fail;

        dentry = file->f_path.dentry;

        retval = -ENOTDIR;
        if (!S_ISDIR(dentry->d_inode->i_mode))

            goto fput_fail;

        retval = file_permission(file, MAY_EXEC);
        if (retval)
```

```

        goto fput_fail;

        nd->path = file->f_path;
        path_get(&file->f_path);

        fput_light(file, fput_needed);
    }

    retval = path_walk(name, nd);
out:
    if (unlikely(!retval && !audit_dummy_context() && nd->path.dentry &&
                nd->path.dentry->d_inode))
        audit_inode(name, nd->path.dentry);
out_fail:
    return retval;

fput_fail:
    fput_light(file, fput_needed);
    goto out_fail;
}

```

## 例2

短路逻辑算法。

这样的例子在内核代码中也是非常地多，一般用在短的函数或宏中。

```

#include <stdio.h>

int main()
{
    int a = 1;
    int b = 2;

    if (a || ++b)
        printf("%d\n", b);

    return 0;
}

```

这个例子，有人会疑问为什么b的值没有变化，还是为2。现在我们分析下它的汇编代码

```

08048374 <main>:
8048374:      8d 4c 24 04          lea     0x4(%esp),%ecx
8048378:      83 e4 f0             and     $0xffffffff0,%esp
804837b:      ff 71 fc             pushl   -0x4(%ecx)
804837e:      55                   push    %ebp
804837f:      89 e5                 mov     %esp,%ebp
//以上汇编代码

```

8048381:	51	push	%ecx	//%ecx入栈保护
8048382:	83 ec 24	sub	\$0x24,%esp	//创建本地变量和
8048385:	c7 45 f8 01 00 00 00	movl	\$0x1,-0x8(%ebp)	//变量a赋值,记
804838c:	c7 45 f4 02 00 00 00	movl	\$0x2,-0xc(%ebp)	//变量b赋值
8048393:	83 7d f8 00	cmpl	\$0x0,-0x8(%ebp)	//变量a和0比较,
8048397:	75 0a	jne	80483a3 <main+0x2f>	//a-0如果不等0,
				//已经知道a==1,
8048399:	83 45 f4 01	addl	\$0x1,-0xc(%ebp)	
804839d:	83 7d f4 00	cmpl	\$0x0,-0xc(%ebp)	
80483a1:	74 13	je	80483b6 <main+0x42>	
80483a3:	8b 45 f4	mov	-0xc(%ebp),%eax	//把变量b的值放
80483a6:	89 44 24 04	mov	%eax,0x4(%esp)	//接着把它作为pr
				//记得子函数的实
80483aa:	c7 04 24 90 84 04 08	movl	\$0x8048490,(%esp)	//printf函数第一
				//是保存到栈的,
80483b1:	e8 22 ff ff ff	call	80482d8 <printf@plt>	//调用printf函数
80483b6:	b8 00 00 00 00	mov	\$0x0,%eax	;//%eax赋值0,%ea
80483bb:	83 c4 24	add	\$0x24,%esp	//撤销新栈帧的本
80483be:	59	pop	%ecx	//恢复保存的旧%e
80483bf:	5d	pop	%ebp	//以下汇编码都是
80483c0:	8d 61 fc	lea	-0x4(%ecx),%esp	
80483c3:	c3	ret		

分析可见C语句 if (a || ++b)中的++b对应的汇编码是

8048399:	83 45 f4 01	addl	\$0x1,-0xc(%ebp)
804839d:	83 7d f4 00	cmpl	\$0x0,-0xc(%ebp)
80483a1:	74 13	je	80483b6 <main+0x42>

可是因为a==1,表达式a已经为真, ++b这个语句, 也就是上面的汇编码, 根本就没运行。所以变量b的值没有自增, 还是保持为2。

## 结论

表达式 a, b

a || b: 如果a为真,b就不管;如果运行到b,a必已是假

a && b: 如果a为假,b就不管;如果运行到b,a必已是真

## 内核代码实例

```
static struct char_device_struct *
__register_chrdev_region(unsigned int major, unsigned int baseminor,
                        int minorct, const char *name)
{
    .....

    i = major_to_index(major);
```

```

        for (cp = &chrdevs[i]; *cp; cp = &(*cp)->next)
            if ((*cp)->major > major ||
                ((*cp)->major == major &&
                 ((*cp)->baseminor >= baseminor) ||
                 ((*cp)->baseminor + (*cp)->minorct > baseminor))))
                break;

.....
}

```

### 例3

#### 自增自减

自增自减，以及增减的前后问题。这类代码在内核数不胜数。理解稍有偏差，就会产生“边界问题”，或者在条件判断时理解出错。

```

#include <stdio.h>

int main()
{
    int i = -1;
    if (!i++) {
        printf("inner: %d\n", i);
    }
    printf("outer: %d\n", i);

    return 0;
}

```

#### 汇编代码

08048374 <main>:

8048374:	8d 4c 24 04	lea	0x4(%esp),%ecx
8048378:	83 e4 f0	and	\$0xffffffff0,%esp
804837b:	ff 71 fc	pushl	-0x4(%ecx)
804837e:	55	push	%ebp
804837f:	89 e5	mov	%esp,%ebp
8048381:	51	push	%ecx
8048382:	83 ec 24	sub	\$0x24,%esp
8048385:	c7 45 f8 ff ff ff ff	movl	\$0xffffffff,-0x8(%ebp)
804838c:	83 45 f8 01	addl	\$0x1,-0x8(%ebp)
8048390:	83 7d f8 01	cmpl	\$0x1,-0x8(%ebp)
8048394:	75 13	jne	80483a9 <main+0x35>
8048396:	8b 45 f8	mov	-0x8(%ebp),%eax
8048399:	89 44 24 04	mov	%eax,0x4(%esp)

```

804839d:    c7 04 24 90 84 04 08    movl    $0x8048490, (%esp)
80483a4:    e8 2f ff ff ff          call    80482d8 <printf@plt>
80483a9:    8b 45 f8                mov     -0x8(%ebp), %eax
80483ac:    89 44 24 04             mov     %eax, 0x4(%esp)
80483b0:    c7 04 24 9b 84 04 08    movl    $0x804849b, (%esp)
80483b7:    e8 1c ff ff ff          call    80482d8 <printf@plt>
80483bc:    b8 00 00 00 00          mov     $0x0, %eax
80483c1:    83 c4 24                add     $0x24, %esp
80483c4:    59                      pop     %ecx
80483c5:    5d                      pop     %ebp
80483c6:    8d 61 fc                lea     -0x4(%ecx), %esp
80483c9:    c3                      ret
80483ca:    90                      nop

```

## 内核代码实例

```

int platform_add_devices(struct platform_device **devs, int num)
{
    int i, ret = 0;

    for (i = 0; i < num; i++) {
        ret = platform_device_register(devs[i]);
        if (ret) {
            while (--i >= 0) /*没错，devs[i]没注册成功的话，从devs[i-1]开始
                             platform_device_unregister(devs[i]);
                             break;
            }
        }

        return ret;
    }
}

```

### 例4

## 函数指针

### 解释在“穿越交叉索引工具的盲区”→函数指针

```

#include <stdio.h>

int main()
{
    int myfunc(int a, int b)
    {
        int c = a + b;
        printf("%d\n", c);
        return 0;
    }
}

```

```

    int (*funa)(int, int) = myfunc;
    int (*funb)(int, int) = &myfunc;
    int (*func)(int, int) = (int (*)(int, int))myfunc;
    int (*fund)(int, int) = (int (*)(int, int))(&myfunc);

    myfunc(1, 2);
    funa(3, 4);
    funb(5, 6);
    func(7, 8);
    fund(9, 10);

    return 0;
}

```

编译:

```
$ gcc -g -Wall fuk.c //注意, 没有任何警告
```

```

int main()
{
8048374:      8d 4c 24 04          lea    0x4(%esp),%ecx
....省略
    int (*funa)(int, int) = myfunc;
8048385:      c7 45 f8 13 84 04 08  movl    $0x8048413,-0x8(%ebp)
    int (*funb)(int, int) = &myfunc;
804838c:      c7 45 f4 13 84 04 08  movl    $0x8048413,-0xc(%ebp)
    int (*func)(int, int) = (int (*)(int, int))myfunc;
8048393:      c7 45 f0 13 84 04 08  movl    $0x8048413,-0x10(%ebp)
    int (*fund)(int, int) = (int (*)(int, int))(&myfunc);
804839a:      c7 45 ec 13 84 04 08  movl    $0x8048413,-0x14(%ebp)

    myfunc(1, 2);
...省略
    funa(3, 4);
80483b5:      c7 44 24 04 04 00 00  movl    $0x4,0x4(%esp)
80483bc:      00
80483bd:      c7 04 24 03 00 00 00  movl    $0x3, (%esp)
80483c4:      8b 45 f8              mov     -0x8(%ebp),%eax
80483c7:      ff d0                call    *%eax
    funb(5, 6);
....省略, funb, func, fund汇编码和funa完全相同

    return 0;
8048405:      b8 00 00 00 00        mov     $0x0,%eax
}
804840a:      83 c4 24              add     $0x24,%esp
...省略

```

08048413 <myfunc.1933>:

```
#include <stdio.h>
```

```

int main()
{
    int myfunc(int a, int b)
    {
8048413:      55                    push    %ebp

```

```
.....省略
}
```

```
xxx@ubuntu:~/dt/test$ gdb a.out
```

```
GNU gdb 6.8-debian
```

```
...
```

```
(gdb) list
```

```
1      #include <stdio.h>
```

```
.....
```

```
17          funa(3, 4);
```

```
....
```

```
20
```

```
(gdb) b 17
```

```
(gdb) r
```

```
Starting program: /home/xxx/桌面/test/a.out
```

```
Breakpoint 1, main () at fuck.c:17
```

```
17          funa(3, 4);
```

```
(gdb) display/i $pc
```

```
1: x/i $pc
```

```
0x80483b5 <main+65>:    movl    $0x4,0x4(%esp)
```

```
(gdb) stepi
```

```
0x080483bd    17          funa(3, 4);
```

```
1: x/i $pc
```

```
0x80483bd <main+73>:    movl    $0x3,(%esp)
```

```
(gdb)
```

```
0x080483c4    17          funa(3, 4);
```

```
1: x/i $pc
```

```
0x80483c4 <main+80>:    mov     -0x8(%ebp),%eax
```

```
(gdb)
```

```
0x080483c7    17          funa(3, 4);
```

```
1: x/i $pc
```

```
0x80483c7 <main+83>:    call   *%eax
```

```
(gdb) p/x $eax
```

```
$4 = 0x8048413
```

```
(gdb) info line *0x8048413
```

```
Line 6 of "fuck.c" starts at address 0x8048413 <myfunc> and ends at 0x8048419 <myfu
```

```
(gdb)
```

```
< | >
```

## 其他例子

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    static const char *operstates[] = {
```

```
        "unknown",
```

```
        "notpresent", /* currently unused */
```

```
        "down",
```

```
        "lowerlayerdown",
```

```
        "testing", /* currently unused */
```



```
"dormant",  
"up"  
};
```

```
enum {  
IF_OPER_UNKNOWN,  
IF_OPER_NOTPRESENT,  
IF_OPER_DOWN,  
IF_OPER_LOWERLAYERDOWN,  
IF_OPER_TESTING,  
IF_OPER_DORMANT,  
IF_OPER_UP,  
};
```

```
enum {  
OPER_UNKNOWN = 1,  
OPER_NOTPRESENT,  
OPER_DOWN,  
OPER_LOWERLAYERDOWN,  
OPER_TESTING,  
OPER_DORMANT,  
OPER_UP,  
};
```

```
enum {  
UNKNOWN,  
NOTPRESENT = 6,  
DOWN,  
LOWERLAYERDOWN,  
TESTING,  
DORMANT,  
UP,  
};
```

```
printf("%d\n", sizeof(operstates));  
printf("%d\n", sizeof(operstates[0]));
```

```
printf("%d\n", IF_OPER_DOWN);
```

```
printf("%d\n", OPER_DOWN);
```

```
printf("%d\n", UNKNOWN);  
printf("%d\n", DOWN);
```

```
}
```

## 优化级别的影响

这部分内容有点偏题，没必要这么钻牛角尖。但是为了说明“调试用的代码和实际运行的代码是不一样”的这个事实以及因为代码优化导致的“非理想状态”的调用链问题(见“内核初窥”)，有必要用观察一个实例，以便有个直观的印象。

首先应该知道，有没有指定调试选项-g(-debug)，在相同优先级下生成的代码都是一样的。差别只是，指定-g后，多生成了一个调试表。

## 优化选项

下面文字来自“ARM 系列应用技术完全手册”

使用-Onum选择编译器的优化级别。优化级别分别有

- -O0: 除一些简单的代码编号外，关闭所有优化，该选项可提供最直接的优化信息。
- -O1: 关闭严重影响调试效果的优化功能。使用该编译选项，编译器会移除程序中未使用到的内联函数和静态函数。如果于-debug(也就是-g)一起使用，该选项可以在较好的代码密度下，给出最佳调试视图。
- -O2: 生成充分优化代码。如果与-debug一起使用，调试效果可能不令人满意，因为对目标代码到源代码的映射可能因为代码优化而发生变化。如果不生成调试表，这是默认优化级别。
- -O3: 最高优化级别。使用该优化级别，使生成的代码在时间和空间上寻求平衡。

## 例子

```
#include <stdio.h>

int add(int a, int b)
{
    return (a + b);
}

void funa()
{
    int a = 3 + 4;
    int b;
    printf("%d\n", a);
    b = add(5,6);
    printf("%d\n", b);
}
```

```
int main()
{
    int m = 1 + 2;
    printf("%d\n", m);
    funa();
}
```

```
$ gcc -g -O0 src.c (或者不指定优化选项: gcc -g src.c, 编译出的机器码一样)
$ objdump -d a.out
```

得到一个结论: 如果指定了-g而没指定优化等级, 那么默认优化等级是最低的-O0

```
08048374 <add>:
8048374:    55                push    %ebp
8048375:    89 e5             mov     %esp,%ebp
8048377:    8b 45 0c          mov     0xc(%ebp),%eax
804837a:    03 45 08          add     0x8(%ebp),%eax
804837d:    5d                pop     %ebp
804837e:    c3                ret

0804837f <funa>:
804837f:    55                push    %ebp
8048380:    89 e5             mov     %esp,%ebp                //保存旧栈
8048382:    83 ec 18          sub     $0x18,%esp              //分配栈帧
8048385:    c7 45 fc 07 00 00 00 movl    $0x7,-0x4(%ebp)         //注意编译
                                //a放到临
804838c:    8b 45 fc          mov     -0x4(%ebp),%eax          //a放到临
804838f:    89 44 24 04       mov     %eax,0x4(%esp)           //接着作为
8048393:    c7 04 24 d0 84 04 08 movl    $0x80484d0,(%esp)        //printf
804839a:    e8 39 ff ff ff    call    80482d8 <printf@plt>    //printf
804839f:    c7 44 24 04 06 00 00 movl    $0x6,0x4(%esp)          //add(5,0
80483a6:    00
80483a7:    c7 04 24 05 00 00 00 movl    $0x5,(%esp)              //add(5,0
80483ae:    e8 c1 ff ff ff    call    8048374 <add>           //调用add
80483b3:    89 45 f8          mov     %eax,-0x8(%ebp)          //-0x8(%
80483b6:    8b 45 f8          mov     -0x8(%ebp),%eax          //b放到临
80483b9:    89 44 24 04       mov     %eax,0x4(%esp)           //接着作为
80483bd:    c7 04 24 d0 84 04 08 movl    $0x80484d0,(%esp)        //printf
80483c4:    e8 0f ff ff ff    call    80482d8 <printf@plt>    //printf
80483c9:    c9                leave   %ebp                     //撤销新栈
80483ca:    c3                ret                                //funa返回

080483cb <main>:
80483cb:    8d 4c 24 04       lea     0x4(%esp),%ecx
80483cf:    83 e4 f0          and     $0xffffffff0,%esp
80483d2:    ff 71 fc          pushl   -0x4(%ecx)
80483d5:    55                push    %ebp
80483d6:    89 e5             mov     %esp,%ebp
80483d8:    51                push    %ecx
80483d9:    83 ec 24          sub     $0x24,%esp
80483dc:    c7 45 f8 03 00 00 00 movl    $0x3,-0x8(%ebp)
80483e3:    8b 45 f8          mov     -0x8(%ebp),%eax
80483e6:    89 44 24 04       mov     %eax,0x4(%esp)
```

```

80483ea:    c7 04 24 d0 84 04 08    movl    $0x80484d0, (%esp)
80483f1:    e8 e2 fe ff ff         call    80482d8 <printf@plt>
80483f6:    e8 84 ff ff ff         call    804837f <funa>
80483fb:    83 c4 24               add     $0x24,%esp
80483fe:    59                     pop     %ecx
80483ff:    5d                     pop     %ebp
8048400:    8d 61 fc               lea     -0x4(%ecx),%esp
8048403:    c3                     ret

```

```

$ gcc -g -O1 src.c
$ objdump -d a.out

```

08048374 <add>:

```

8048374:    55                     push    %ebp
8048375:    89 e5                 mov     %esp,%ebp
8048377:    8b 45 0c              mov     0xc(%ebp),%eax
804837a:    03 45 08              add     0x8(%ebp),%eax
804837d:    5d                     pop     %ebp
804837e:    c3                     ret

```

0804837f <funa>:

//funa与-  
//代码量少

```

804837f:    55                     push    %ebp
8048380:    89 e5                 mov     %esp,%ebp
8048382:    83 ec 08              sub     $0x8,%esp
8048385:    c7 44 24 04 07 00 00 movl    $0x7,0x4(%esp)

```

//分配栈帧  
//printf  
//注意，与

```

804838c:    00
804838d:    c7 04 24 c0 84 04 08 movl    $0x80484c0, (%esp)
8048394:    e8 3f ff ff ff         call    80482d8 <printf@plt>
8048399:    c7 44 24 04 06 00 00 movl    $0x6,0x4(%esp)
80483a0:    00
80483a1:    c7 04 24 05 00 00 00 movl    $0x5, (%esp)
80483a8:    e8 c7 ff ff ff         call    8048374 <add>
80483ad:    89 44 24 04             mov     %eax,0x4(%esp)

```

//printf  
//add(5,0)  
//add的返回  
//注意，与

```

80483b1:    c7 04 24 c0 84 04 08 movl    $0x80484c0, (%esp)
80483b8:    e8 1b ff ff ff         call    80482d8 <printf@plt>
80483bd:    c9                     leave   %eax
80483be:    c3                     ret

```

//printf

080483bf <main>:

```

80483bf:    8d 4c 24 04           lea     0x4(%esp),%ecx
80483c3:    83 e4 f0              and     $0xffffffff0,%esp
80483c6:    ff 71 fc              pushl   -0x4(%ecx)
80483c9:    55                     push    %ebp
80483ca:    89 e5                 mov     %esp,%ebp
80483cc:    51                     push    %ecx
80483cd:    83 ec 14              sub     $0x14,%esp
80483d0:    c7 44 24 04 03 00 00 movl    $0x3,0x4(%esp)
80483d7:    00
80483d8:    c7 04 24 c0 84 04 08 movl    $0x80484c0, (%esp)
80483df:    e8 f4 ff ff ff         call    80482d8 <printf@plt>

```

```

80483e4:    e8 96 ff ff ff      call    804837f <funa>
80483e9:    83 c4 14            add     $0x14,%esp
80483ec:    59                  pop     %ecx
80483ed:    5d                  pop     %ebp
80483ee:    8d 61 fc            lea     -0x4(%ecx),%esp
80483f1:    c3                  ret

```

```

$ gcc -g -O2 src.c
$ objdump -d a.out

```

我们应该知道，如果没有指定-g和优化选项，那么默认的优化等级就是-O2

```

08048380 <add>:
8048380:    55                  push    %ebp
8048381:    89 e5              mov     %esp,%ebp
8048383:    8b 45 0c           mov     0xc(%ebp),%eax
8048386:    03 45 08           add     0x8(%ebp),%eax
8048389:    5d                  pop     %ebp
804838a:    c3                  ret
804838b:    90                  nop
804838c:    8d 74 26 00       lea     0x0(%esi),%esi

08048390 <funa>:
8048390:    55                  push    %ebp
8048391:    89 e5              mov     %esp,%ebp
8048393:    83 ec 08           sub     $0x8,%esp
8048396:    c7 44 24 04 07 00 00 movl    $0x7,0x4(%esp)
804839d:    00
804839e:    c7 04 24 d0 84 04 08 movl    $0x80484d0,(%esp)
80483a5:    e8 2e ff ff ff     call    80482d8 <printf@plt>
80483aa:    c7 44 24 04 06 00 00 movl    $0x6,0x4(%esp)
80483b1:    00
80483b2:    c7 04 24 05 00 00 00 movl    $0x5,(%esp)
80483b9:    e8 c2 ff ff ff     call    8048380 <add>
80483be:    c7 04 24 d0 84 04 08 movl    $0x80484d0,(%esp) //第二个参数
80483c5:    89 44 24 04         mov     %eax,0x4(%esp) //第一个参数
//但是入栈
//printf

80483c9:    e8 0a ff ff ff     call    80482d8 <printf@plt>
80483ce:    c9                  leave
80483cf:    c3                  ret

080483d0 <main>:
80483d0:    8d 4c 24 04       lea     0x4(%esp),%ecx
80483d4:    83 e4 f0          and     $0xffffffff0,%esp
80483d7:    ff 71 fc          pushl   -0x4(%ecx)
80483da:    55                  push    %ebp
80483db:    89 e5              mov     %esp,%ebp
80483dd:    51                  push    %ecx
80483de:    83 ec 14          sub     $0x14,%esp
80483e1:    c7 44 24 04 03 00 00 movl    $0x3,0x4(%esp)
80483e8:    00
80483e9:    c7 04 24 d0 84 04 08 movl    $0x80484d0,(%esp)
80483f0:    e8 e3 ff ff ff     call    80482d8 <printf@plt>

```

```

80483f5:    e8 96 ff ff ff    call    8048390 <funa>
80483fa:    83 c4 14          add     $0x14,%esp
80483fd:    59               pop     %ecx
80483fe:    5d               pop     %ebp
80483ff:    8d 61 fc          lea     -0x4(%ecx),%esp

```

```

8048402:    c3              ret

```

```

$ gcc -g -O3 src.c
$ objdump -d a.out

```

```

048380 <add>:

```

```

8048380:    55              push    %ebp
8048381:    89 e5           mov     %esp,%ebp
8048383:    8b 45 0c        mov     0xc(%ebp),%eax
8048386:    03 45 08        add     0x8(%ebp),%eax
8048389:    5d              pop     %ebp
804838a:    c3              ret
804838b:    90              nop
804838c:    8d 74 26 00     lea     0x0(%esi),%esi

```

```

08048390 <funa>:

```

//与-O2相

```

8048390:    55              push    %ebp
8048391:    89 e5           mov     %esp,%ebp
8048393:    83 ec 08        sub     $0x8,%esp
8048396:    c7 44 24 04 07 00 00 movl    $0x7,0x4(%esp)
804839d:    00
804839e:    c7 04 24 e0 84 04 08 movl    $0x80484e0, (%esp)
80483a5:    e8 2e ff ff ff    call    80482d8 <printf@plt>
80483aa:    c7 44 24 04 0b 00 00 movl    $0xb,0x4(%esp)

```

//注意，与  
//之前应该  
//太简单，  
//编译器重

```

80483b1:    00
80483b2:    c7 04 24 e0 84 04 08 movl    $0x80484e0, (%esp)
80483b9:    e8 1a ff ff ff    call    80482d8 <printf@plt>
80483be:    c9              leave   %eax
80483bf:    c3              ret

```

//printf

```

080483c0 <main>:

```

```

80483c0:    8d 4c 24 04     lea     0x4(%esp),%ecx
80483c4:    83 e4 f0        and     $0xfffffffff0,%esp
80483c7:    ff 71 fc        pushl   -0x4(%ecx)
80483ca:    55              push    %ebp
80483cb:    89 e5           mov     %esp,%ebp
80483cd:    51              push    %ecx
80483ce:    83 ec 14        sub     $0x14,%esp
80483d1:    c7 44 24 04 03 00 00 movl    $0x3,0x4(%esp)
80483d8:    00
80483d9:    c7 04 24 e0 84 04 08 movl    $0x80484e0, (%esp)
80483e0:    e8 f3 ff ff ff    call    80482d8 <printf@plt>

```

```

80483e5:    c7 44 24 04 07 00 00    movl    $0x7,0x4(%esp)
80483ec:    00
80483ed:    c7 04 24 e0 84 04 08    movl    $0x80484e0,(%esp)
80483f4:    e8 df fe ff ff          call    80482d8 <printf@plt>
80483f9:    c7 44 24 04 0b 00 00    movl    $0xb,0x4(%esp)
8048400:    00
8048401:    c7 04 24 e0 84 04 08    movl    $0x80484e0,(%esp)
8048408:    e8 cb fe ff ff          call    80482d8 <printf@plt>
804840d:    83 c4 14                add     $0x14,%esp
8048410:    59                      pop     %ecx
8048411:    5d                      pop     %ebp
8048412:    8d 61 fc                lea     -0x4(%ecx),%esp
8048415:    c3                      ret

```

## 汇编基础--ARM篇

说明：

1. 部分内容和X86的重复，重复部分请参考X86的内容。
2. 某些内容不具备普遍性。比如给出的反汇编代码，在不同的优化等级下是不同的。但是在熟悉了典型的函数调用链反汇编代码，对于有变化的其他形式也就不难理解了。

### 用户手册

ARM7TDMI Technical Reference Manual

ARM920T Technical Reference Manual

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.home/index.html>

指令速查 [http://www.arm.com/pdfs/QRC0001H\\_rvct\\_v2.1\\_arm.pdf](http://www.arm.com/pdfs/QRC0001H_rvct_v2.1_arm.pdf)

### 调用链形成和参数传递

注意：arm体系过程调用的文字说明部分，都是依据AAPCS标准。

### 壮观的标准

参考：

## AAPCS

### Procedure Call Standard for the ARM Architecture

[http://infocenter.arm.com/help/topic/com.arm.doc.ihl0042b/IHL0042B\\_aapcs.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ihl0042b/IHL0042B_aapcs.pdf)

终于在“ARM Procedure Call Standard”中找到了答案

PCS	Procedure Call Standard.
AAPCS	Procedure Call Standard for the ARM Architecture (this standard).
APCS	ARM Procedure Call Standard (obsolete).
TPCS	Thumb Procedure Call Standard (obsolete).
ATPCS	ARM-Thumb Procedure Call Standard (precursor to this standard).
PIC, PID	Position-independent code, position-independent data.

下面的标准已过时

## APCS

ARM Procedure Call Standard <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0041c/BGBGFIDA.html>

Using the ARM Procedure Call Standard <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0040d/Chdbceig.html>

APCS 简介[http://www.bsdmap.com/UNIX\\_html/ARM/apcsintro.html#01](http://www.bsdmap.com/UNIX_html/ARM/apcsintro.html#01)

## TPCS

Thumb Procedure Call Standard <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0041c/BCEEAHAF.html>

Using the Thumb Procedure Call Standard <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0040d/Cihdbchi.html>

## ATPCS

About the ARM-Thumb Procedure Call Standard <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/Bcfcieh.html>



## 别名的烦恼

arm体系的函数调用标准换了好几个版本，对寄存器的别名也是不一样。不同的调试器，或者它在不同的选项下，对同一个寄存器可能就有多种称呼。又或者你在调试器下看到的名称和书籍上的不一样。所以，又必要知道这些寄存器各自都有哪些别名。

我们运行下命令

```
$ arm-linux-gnueabi-objdump --help
....省略
The following ARM specific disassembler options are supported for use with
the -M switch:
  reg-names-special-atpcs  Select special register names used in the ATPCS
  reg-names-atpcs          Select register names used in the ATPCS
  reg-names-apcs           Select register names used in the APCS
  reg-names-std            Select register names used in ARM's ISA documentation
  reg-names-gcc            Select register names used by GCC
  reg-names-raw            Select raw register names
  force-thumb              Assume all insns are Thumb insns
  no-force-thumb           Examine preceeding label to determine an insn's type
```

我们下载它的源码打开看看

```
$ sudo apt-get source binutils-arm-linux-gnueabi
```

完成后，在下载目录下多了几个东东，其中有一个文件夹binutils-2.18.1~cvs20080103，这是debian对官方binutils进行过修改的源码。在里面搜索文件arm-dis.c，该文件中有以下这个数组。

就是不同标准下各个寄存器的不同别名。

```
static const arm_regname regnames[] =
{
  { "raw", "Select raw register names",
    { "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7", "r8", "r9", "r10", "r11",
  { "gcc", "Select register names used by GCC",
    { "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7", "r8", "r9", "s1", "fp",
  { "std", "Select register names used in ARM's ISA documentation",
    { "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7", "r8", "r9", "r10", "r11",
  { "apcs", "Select register names used in the APCS",
    { "a1", "a2", "a3", "a4", "v1", "v2", "v3", "v4", "v5", "v6", "s1", "fp",
  { "atpcs", "Select register names used in the ATPCS",
    { "a1", "a2", "a3", "a4", "v1", "v2", "v3", "v4", "v5", "v6", "v7", "v8",
  { "special-atpcs", "Select special register names used in the ATPCS",
    { "a1", "a2", "a3", "a4", "v1", "v2", "v3", "WR", "v5", "SB", "SL", "FP",
```

```
};
```

但是可以看到，该列表并没有包含AAPCS标准，AAPCS标准对 r9 又引入了一个别名 TR，这样AAPCS下，别名，是依赖于不同平台的选择。

[扩展，简要说明原理。并用实例解析]

寄存器的角色与保护

- 寄存器的角色(AAPCS标准)

寄存器	可选寄存器名	特殊寄存器名	在函数调用中的角色
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6/SB/TR	Platform register. The meaning of this register is defined by the platform standard
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

前四个寄存器r0-r3 (a1-a4)用于传递参数给子函数或从函数中返回结果值。他们也可用于在一个函数中保存寄存器的值（但是，一般只用在子函数调用中）。

寄存器r12 (IP) 可在函数以及该函数调用的任何子函数中被链接器用作临时寄存器。它也可以在函数调用中用于保存寄存器的值。

寄存器r9的角色是平台相关的。虚拟系统可能赋予该寄存器任何角色，因此必须说明它的用法。比如，在位置无关数据模型中它可以指定为static base(SB)，或者在带有本地线程存储的环境中指定它为thread register(TR)。该寄存器的使用可能要求在所有调用过程前后，它保存的值必须不变。在一个不需要这样特殊寄存器的虚拟平台上，r9可以指定为新增的callee-saved variable register,v6.

通常，寄存器r4-r8, r10 和 r11 (v1-v5, v7 和 v8)用于保存函数的本地变量。这些寄存器中，只有v1-v4能被整个thumb指令集一致地使用，但是AAPCS并没有规定Thumb代码只能使用这些寄存器。

子函数必须保护寄存器r4-r8, r10, r11 和 SP(还有r9,如果在函数调用过程中r6被指定为v6的话)的值。

在所有的函数调用标准中，寄存器r12-r15都扮演特殊的角色。依据这些角色，它们被标注为IP, SP, LR 和 PC。

### 寄存器CPSR的属性（省）

- 寄存器保护规则

子函数必须保护寄存器r4-r8, r10, r11 和 SP(还有r9,如果在函数调用过程中r6被指定为v6的话)的值。子函数调用

- 子函数调用

ARM 和 Thumb 指令集都有一个函数调用指令元语，BL,它执行branch-with-link 操作。BL的执行效果是把紧跟程序计数器的下一个值 - - 也就是返回地址 - - 传送到链接寄存器(LR)，然后把目标地址传送到程序寄存器(PC)中。如果 BL指令是在Thumb状态下执行的，链接寄存器的Bit 0就设置为1;如果是在ARM状态下执行的，则设置为0。执行的结果是，把控制权转给目标地址，并把存放在LR中的返回地址作为附加的参数传递给了被调用的函数。

当返回地址装载到PC时，控制就返回给跟随BL后面的指令。

子函数调用可以由具有下面效果的任何指令序列完成：

```
LR[31:1] ← 返回地址
LR[0] ← 返回地址的代码类型 (0 ARM, 1 Thumb)
PC ← 子函数地址
...
```

返回地址：

例如，在ARM状态中，调用由r4指定了地址的子函数

```
do:
    MOV LR, PC
    BX  r4
    ...
```

注意，相同的指令序列在Thumb状态中将不能工作，因为设置LR的指令并没有拷贝Thumb 状态标志位到LR[0]中。

在ARM V5架构中，ARM 和 Thumb指令集都提供了BLX指令，它将调用由一个寄存器指定了地址的子函数，并正确地设置返回地址为程序计数器的下一个值。

### 条件执行

操作码 [31:28]	助记符扩展	解释	用于执行的标志位状态
0000	EQ	相等/等于0	Z置位
0001	NE	不等	Z清0
0010	CS/HS	进位/无符号数高于或等于	C置位
0011	CC/LO	无进位/无符号数小于	C清0
0100	MI	负数	N置位
0101	PL	正数或0	N清0
0110	VS	溢出	V置位
0111	VC	未溢出	V清0
1000	HI	无符号数高于	C置位, Z清0
1001	LS	无符号数小于或等于	C清0,Z置位
1010	GE	有符号数大于或等于	N等于V
1011	LT	有符号数小于	N不等于V
1100	GT	有符号数大于	Z清0且N等于V
1101	LE	有符号数小于或等于	Z置位且N不等于V
1110	AL	总是	任何状态
1111	NV	从不(未使用)	无

### 调用链的形成

注意对比ARM和X86在调用链形成的类似和区别之处。

区别，首先在寄存器的名称和角色的差异。

1. X86中寄存器%eip指向的是下一个将要执行的指令。在ARM中也有个类似别名的寄存器ip。但这个寄存器ip的作用并不是指向的是下一个将要执行的指令。在ARM中，寄存器pc才是起着X86中寄存器%eip的角色，也就是包含下一个将要执行指令的地址。而ARM中的ip寄存器，作用比较自由，类似干杂工的人，一般用于临时寄存器。[扩展，引用权威手册的话]

2. X86中，返回地址是直接保存在栈中的。但是ARM不一样了，它寄存器比X86多得多，财大气粗，所以，返回地址保存在了专用的寄存器lr(link register)中。但是，不要以为把返回地址放到专用的寄存器中会省事，其实反而多事了。因为，在调用函数刚执行完调用语句之时，lr保存的是子函数的返回地址，而指令控制权转移到了子函数后，子函数照样可能调用自己的子函数，依次需要使用lr。所以自然也就有了lr的值的保存与恢复的问题，解决方法还是要靠压栈解决。（参考下面的内容）

3. 我们知道，描述栈帧就是描述栈帧的基址和顶端地址。在X86中，用专用的寄存器%ebp保存栈基址，也就是base pointer；%esp保存栈顶端地址，也就是stack pointer。在ARM中，也有专用的寄存器保存栈顶端地址，就是SP(stack pointer的简称)。但是，在保存栈基址这方面，依据最新的AAPCS标准，ARM就很吝啬了，没有一个保存栈基址的专用寄存器。又不过呢，在APCS和ATPCS标准中，有fp寄存器用于保存帧指针(frame pointer，也就是X86的base pointer)。在现在的编译器，可以看到，还是依照惯例把fp用于保存帧指针。既然如此，当然也有个入栈保存恢复的问题。

调用链包含两方面的内容，和X86类似

### 1.返回地址的保存与恢复

由调用函数在执行调用指令时把子函数的返回地址传送进连接寄存器lr中，指令控制权转交给子函数后，再由子函数负责把上层函数的lr(也就是子函数的返回地址)保存到栈中。然后子函数在返回前的最后时刻，再负责把lr的保存值从栈弹回到lr中，从而恢复了上层函数的lr。这时还没完事，子函数在执行返回指令时，由返回指令把lr的值传送到寄存器pc(Program Counter)，从而导致接下来的指令是从子函数的返回地址开始运行。这样，指令控制权就返回给了调用函数。

我们应当注意到，ARM中调用指令也是多种多样的。有b,bl,bx,bxl。如果调用指令是不带连接的指令，比如b,bx,这时就要人工给lr赋值。不过为了简便，我不再区分这两类指令，而把实现跳转和连接以及可能的换态这些功能的整个指令序列为“调用指令”，相关区别参考指令手册。在ARM中，返回指令和调用指令都是同一套的。而X86，调用用call,返回用ret。

### 2.旧栈帧的保存与恢复

对比X86栈帧的保存与恢复的方式，ARM的更加简单直接。就是直接把上一栈帧的帧指针（frame pointer，也就是栈帧基地址）以及栈顶端指针sp(stack pointer)压入栈中。子函数返回时，在执行返回指令之前的最后关头才从栈弹出fp和sp的值，从而恢复旧栈帧。这个过程真的没有遗漏了吗？我们看下，上面的步骤保证了调用函数的栈帧不被破坏，但是子函数自己的栈帧却没有建立起来呢。首先是帧指针需要人赋值。这个情形和X86非常相似。子函数在使用 栈帧之前，把上层函数的栈顶端指针sp赋给一个临时寄存器ip，然后在旧fp的值被压栈保存之后，把ip的值减去4，再赋给帧指针寄存器fp，此时，fp 就指向了新栈帧的基址。这是因为，新栈帧基址刚好位于旧栈帧栈顶之下，地址低了4字节。其次，子函数栈帧的栈顶指针sp也是要考虑的，根据压栈指令的不同，sp可能不需要人工维护，也可能需要人工维护 [有疑问...????] 。

我们还注意到，在X86中，子函数的栈帧的底端（也就是%ebp所指的内存位置）存放着上一层栈帧的基址指针(旧%ebp)的值，一层层下去，这样就形成回溯的链条。那么，在ARM之下，也是靠子函数的栈帧的底端提供回溯的能力的吗？当然不是。实际上子函数的栈帧的基址位置存放的是什么呢，这无所谓 的。

[疑问??? 如果旧fp保存在新栈帧中的位置不是固定的，那么调试器是如何做到栈帧回溯的呢？]

根据AAPCS标准的规定，子函数必须保护寄存器r4-r8, r10, r11 和 SP(还有r9,如果在函数调用过程中r6被指定为v6的话)的值。注意，它用的字眼是“保护”，而不是“保存”。

#### • 应用层实例解析

```
#include <stdio.h>

void func()
{

}

void funb()
{
    func();
}

void funa()
{
    funb();
}
```

```

int main()
{
    funa();
}
-----
000083b0 <func>:
#include <stdio.h>

void func()
{}

83b0:      e1a0c00d      mov     ip, sp
83b4:      e92dd800      push   {fp, ip, lr, pc}
83b8:      e24cb004      sub     fp, ip, #4      ; 0x4
83bc:      e24bd00c      sub     sp, fp, #12     ; 0xc
83c0:      e89d6800      ldm     sp, {fp, sp, lr}

83c4:      e12fff1e      bx      lr

000083c8 <funb>:

void funb()
{
83c8:      e1a0c00d      mov     ip, sp
83cc:      e92dd800      push   {fp, ip, lr, pc}
83d0:      e24cb004      sub     fp, ip, #4      ; 0x4
    func();
83d4:      ebfffff5      bl      83b0 <func>
}

83d8:      e24bd00c      sub     sp, fp, #12     ; 0xc
83dc:      e89d6800      ldm     sp, {fp, sp, lr}
83e0:      e12fff1e      bx      lr

000083e4 <funa>:

void funa()
{

83e4:      e1a0c00d      mov     ip, sp
83e8:      e92dd800      push   {fp, ip, lr, pc}
83ec:      e24cb004      sub     fp, ip, #4      ; 0x4
    funb();
83f0:      ebfffff4      bl      83c8 <funb>
}

83f4:      e24bd00c      sub     sp, fp, #12     ; 0xc
83f8:      e89d6800      ldm     sp, {fp, sp, lr}
83fc:      e12fff1e      bx      lr
00008400 <main>:

int main()
{
8400:      e1a0c00d      mov     ip, sp

```

```

8404:      e92dd800      push    {fp, ip, lr, pc}
8408:      e24cb004      sub     fp, ip, #4      ; 0x4
      funa();
840c:      ebfffff4      bl      83e4 <funa>
}
8410:      e24bd00c      sub     sp, fp, #12     ; 0xc
8414:      e89d6800      ldm     sp, {fp, sp, lr}
8418:      e12fff1e      bx      lr

```

- 内核层实例解析

## 栈帧结构与参数传递

[1.栈:栈对齐, 栈限制。2.参数传递: variadic函数, nonvariadic函数。3.结果的返回  
4.互交代码(ARM-Thumb interworking)]

## 栈帧示意图



## 完整的调用过程

函数caller调用子函数callee, 这是应用层的普通函数调用过程。如果是远调用, 跨态调用要考虑的东西更多。但这个例子已经充分展示了调用过程的繁复部分。

- 函数调用前调用者的动作



- 函数调用 call callee
- 函数调用后被调用者的动作
- 调用返回前被调用者的动作
- 调用返回后调用者的动作
- 应用层实例解析
- 内核层实例解析

## 调用链回溯的实现

arm体系对调用链的回溯的代码实现主要在

arch/arm/kernel/traps.c 和arch/arm/lib/backtrace.S.其中核心函数是backtrace.S中的\_\_

待解释

```
---/*
 * linux/arch/arm/lib/backtrace.S
 *
 * Copyright (C) 1995, 1996 Russell King
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 *
 * 27/03/03 Ian Molton Clean up CONFIG_CPU
 *
 */
#include <linux/linkage.h>
#include <asm/assembler.h>
        .text

@ fp is 0 or stack frame

#define frame    r4
#define sv_fp    r5
#define sv_pc    r6
#define mask     r7
#define offset   r8

ENTRY(__backtrace)
        mov     r1, #0x10
        mov     r0, fp

ENTRY(c_backtrace)
```

```

#if !defined(CONFIG_FRAME_POINTER) || !defined(CONFIG_PRINTK)
        mov     pc, lr
ENDPROC(__backtrace)
ENDPROC(c_backtrace)
#else
        stmfd   sp!, {r4 - r8, lr}      @ Save an extra register so we ha
        movs    frame, r0               @ if frame pointer is zero
        beq     no_frame                @ we have no stack frames

        tst     r1, #0x10               @ 26 or 32-bit mode?
        moveq   mask, #0xfc000003      @ mask for 26-bit
        movne   mask, #0               @ mask for 32-bit

1:      stmfd   sp!, {pc}               @ calculate offset of PC stored
        ldr     r0, [sp], #4           @ by stmfd for this CPU
        adr     r1, 1b
        sub     offset, r0, r1

/*
 * Stack frame layout:
 *      optionally saved caller registers (r4 - r10)
 *      saved fp
 *      saved sp
 *      saved lr
 *      frame => saved pc
 *      optionally saved arguments (r0 - r3)
 *      saved sp => <next word>
 *
 * Functions start with the following code sequence:
 *      mov     ip, sp
 *      stmfd   sp!, {r0 - r3} (optional)
 *      corrected pc => stmfd sp!, {..., fp, ip, lr, pc}
 */
for_each_frame: tst     frame, mask      @ Check for address exceptions
                bne     no_frame

1001:      ldr     sv_pc, [frame, #0]     @ get saved pc
1002:      ldr     sv_fp, [frame, #-12]   @ get saved fp

        sub     sv_pc, sv_pc, offset    @ Correct PC for prefetching
        bic     sv_pc, sv_pc, mask      @ mask PC/LR for the mode

1003:      ldr     r2, [sv_pc, #-4]       @ if stmfd sp!, {args} exists,
        ldr     r3, .Ldsi+4             @ adjust saved 'pc' back one
        teq     r3, r2, lsr #10         @ instruction
        subne   r0, sv_pc, #4           @ allow for mov
        subeq   r0, sv_pc, #8           @ allow for mov + stmia

        ldr     r1, [frame, #-4]        @ get saved lr
        mov     r2, frame
        bic     r1, r1, mask            @ mask PC/LR for the mode
        bl      dump_backtrace_entry

        ldr     r1, [frame, #-4]        @ if stmfd sp!, {args} exists

```

```

        ldr    r3, .Ldsi+4
        teq    r3, r1, lsr #10
        ldreq  r0, [frame, #-8]      @ get sp
        subeq  r0, r0, #4           @ point at the last arg
        bleq   .Ldumpstm           @ dump saved registers

1004:    ldr    r1, [sv_pc, #0]        @ if stmfd sp!, {..., fp, ip, lr,
        ldr    r3, .Ldsi           @ instruction exists,
        teq    r3, r1, lsr #10
        subeq  r0, frame, #16
        bleq   .Ldumpstm           @ dump saved registers

        teq    sv_fp, #0           @ zero saved fp means
        beq    no_frame            @ no further frames

        cmp    sv_fp, frame        @ next frame must be
        mov    frame, sv_fp        @ above the current frame
        bhi    for_each_frame

1006:    adr    r0, .Lbad
        mov    r1, frame
        bl     printk
no_frame:    ldmfd  sp!, {r4 - r8, pc}
ENDPROC(__backtrace)
ENDPROC(c_backtrace)

.section __ex_table,"a"
.align 3
.long 1001b, 1006b
.long 1002b, 1006b
.long 1003b, 1006b
.long 1004b, 1006b
.previous

#define instr r4
#define reg    r5
#define stack r6

.Ldumpstm:    stmfd  sp!, {instr, reg, stack, r7, lr}
        mov    stack, r0
        mov    instr, r1
        mov    reg, #10
        mov    r7, #0
1:    mov    r3, #1
        tst    instr, r3, lsl reg
        beq    2f
        add    r7, r7, #1
        teq    r7, #6
        moveq  r7, #1
        moveq  r1, #'\n'
        movne  r1, #' '
        ldr    r3, [stack], #-4
        mov    r2, reg
        adr    r0, 1f

```

```

2:          bl      printk
          subs     reg, reg, #1
          bpl      1b
          teq      r7, #0
          adrne    r0, .Lcr
          blne     printk
          ldmdfd   sp!, {instr, reg, stack, r7, pc}

.Lfp:      .asciz   "%cr%d:%08x"
.Lcr:      .asciz   "\n"
.Lbad:     .asciz   "Backtrace aborted due to bad frame pointer <%p>\n"
          .align
.Ldsi:     .word    0xe92dd800 >> 10      @ stmfd sp!, {... fp, ip, lr, pc}
          .word    0xe92d0000 >> 10      @ stmfd sp!, {}

#endif

```

## 源码浏览工具

本节意义： 内核源码的代码量越来越大，不借助源码交叉索引工具根本是无法阅读了。一定要熟练灵活掌握此类工具的使用

### 调用图生成工具

#### 1.CodeViz

官网：

<http://www.csn.ul.ie/~mel/projects/codeviz/>

安装使用：

CodeViz —— 一款分析C\_C++源代码中函数调用关系的调用图生成工具.pdf

<http://linux.chinaunix.net/bbs/thread-1031921-1-1.html>

用CodeViz产生函数调用图

<http://barry-popy.blog.sohu.com/31629163.html>

分析函数调用关系图(call graph)的几种方法

<http://blog.csdn.net/Solstice/archive/2005/09/24/488865.aspx>

用CodeViz绘制函数调用关系图(call graph)

<http://blog.csdn.net/Solstice/archive/2005/09/22/486788.aspx>

2.ncc

## **find + grep**

对于源码的阅读工具，一般是选取后面提到的某种源码索引工具，再和find以及grep“高低搭配”一起来使用。

1.命令选项

2.正则表达式

Regular Expression HOWTO: <http://www.amk.ca/python/howto/regex/>

正则表达式之道: [http://net.pku.edu.cn/~yhf/tao\\_regexps\\_zh.html](http://net.pku.edu.cn/~yhf/tao_regexps_zh.html)

## **wine + SI**

wine + source insight

### **优缺点**

优点： SI的特点是有图形界面，操作和浏览特别方便快捷。特别是它的“函数调用树”的图形显示功能，以及分窗口自动显示函数，变量等定义的功能。

缺点： 不能解析汇编源文件。

### **安装wine**

在ubuntu/debian下用以下命令就能在线安装wine

```
$ sudo apt-get install wine
```

安装好后，就能看到wine的快捷菜单被添加到了任务栏的“应用程序”中。

## 安装SI

wine安装好后，就可以像在windows一样去安装使用SI了。安装完成后，SI的快捷菜单被添加到“应用程序”→“wine”→“programs”→“source insight3”中。以后用快捷菜单就能启动SI

## SI的设置

字体，颜色就不说了。现在加入 无名小卒 大侠发现的一个有用设置。

preferences→ display→ trim long path names with ellipses. 去掉该选项的选择。这样就能直接在上下两个分窗口的标题栏上看到一个源文件的全路径。如果不去掉的话，对于长路径它会用...的形式来表示路径的一部分。

## SI的使用

可以乱点乱试一下，它能提供很多的功能。其中一些经常要到的功能有 查找符号；函数调用的函数，被调用的函数；以及调用关系的多层展开显示；字符串搜索等。

## global

[待玩] <http://www.gnu.org/software/global/>

## Source-Navigator

[待玩] <http://sourcnav.sourceforge.net/>

安装：

在ubuntu下可以在线安装

```
$ sudo apt-get install sourcnav
```

运行：

```
$ snavigator
```

## vim + cscope/ctags

参考：

cscope的官方教程 “The Vim/Cscope tutorial”：

[http://cscope.sourceforge.net/cscope\\_vim\\_tutorial.html](http://cscope.sourceforge.net/cscope_vim_tutorial.html)

对应的中文翻译：<http://www.gracecode.com/Archive/Display/316>

<http://www.lupaworld.com/?uid-151392-action-viewspace-itemid-106656>

<http://dev.21tx.com/2007/02/21/10252.html>

### 优缺点

优点： 本人感觉在终端下看源码比较舒服。

缺点： 没有一个实时显示函数/变量定义的分窗口。也不能直接显示“调用树”，但有其他小工具可以实现该功能。也许vim高手能解决这些问题。

### 安装cscope/ctags

ubuntu/debian下用以下命令就能在线安装

```
$ sudo apt-get install cscope ctags
```

### 命令选项

在终端下可以用 man info -help等形式查看cscope/ctags的手册

在vim下查看手册的方式是

```
:help cscope  
和  
:help ctags
```

### 1. 以下是cscope建立索引文件用到的一些选项

-R： 在生成索引文件时，搜索子目录树中的代码

- b: 只生成索引文件，不进入cscope的界面
- q: 生成cscope.in.out和cscope.po.out文件，加快cscope的索引速度
- k: 在生成索引文件时，不搜索/usr/include目录
- i: 如果保存文件列表的文件名不是cscope.files时，需要加此选项告诉cscope到哪儿去找源文件列表
- I dir: 在-I选项指出的目录中查找头文件
- u: 扫描所有文件，重新生成交叉索引文件
- C: 在搜索时忽略大小写
- P path: 在以相对路径表示的文件前加上的path，这样，你不用切换到你数据库文件所在的目录也可

## 2. 在vim下利用:cscope find <关键字> 命令的选项有

- s: 查找C语言符号，即查找函数名、宏、枚举值等出现的地方
- g: 查找函数、宏、枚举等定义的位置，类似ctags所提供的功能
- d: 查找本函数调用的函数
- c: 查找调用本函数的函数
- t: 查找指定的字符串
- e: 查找egrep模式，相当于egrep功能，但查找速度快多了
- f: 查找并打开文件，类似vim的find功能
- i: 查找包含本文件的文

### 使用

#### 建立索引

#### [可能要修改]

用以下命令先产生一个文件列表，然后让cscope为这个列表中的每个文件都生成索引。在这里，我们只关注.h, .c, .S文件，所以只对他们进行索引。可以根据自己需求进行更改。接着我们用-bq选项利用cscope生成索引。选项意义见上节。同时也生成ctags索引。

```
#!/bin/sh
find . -name "*.h" -o -name "*.c" -o -name "*.S" > cscope.files
```



```
cscope -bkq -i cscope.files
ctags -R
```

### 利用vim浏览源码

切换到内核源码的目录上，运行vim，然后在vim下导入索引

```
$vim
:cscope add cscope.out
```

然后就可以在vim下调用“:cscope find <关键字>”来查找函数的定义，函数调用的函数以及被调用函数等

“:cscope find <关键字>”可以缩写为“:cs f <关键字>”

比如以下命令用来查找sys\_read的定义

```
:cs f g sys_read
```

“cs f”的其他命令选项请看上节

### 快捷键的使用

```
ctrl + t : 退回
ctrl + ] : 进入光标处的变量/函数的定义处
```

## kscope

kscope是cscope的图形前端工具。在ubuntu下可以在线安装。它的界面上和操作上与source insight都比较类似。但是目前它对cpu的占用很大，不是很好。但是它和cscope相比，有一个很大的优点是：可以图形显示“函数调用树”，甚至这个功能比SI还强大。

```
$sudo apt-get install kscope
```

## lxr

### 1. 优缺点

优点：本身好像没什么特别的优点。但是有专门提供这种服务的网站，上面有很多不同系统的不同版本源码

缺点：在本机上配置运行的话，配置麻烦。如果是浏览lxr站点的方式，速度比较慢。

2. lxr官方：<http://lxr.linux.no/>

特点是可以浏览历史上linux所有版本的源码，可以看到它的演化过程。

3. 其他系统的源码 <http://fxr.watson.org/>

估计超一流的内核开发人员，可能会经常访问此类站点。因为他需要借鉴其他系统的设计思想。

## SI等与gdb的特点

在源码阅读的功能上：

1. SI等适合“面读”，也就是读一个代码段，并且提供更舒适的阅读辅助手段。SI适合分析函数全面的逻辑。

2. gdb适合“线读”，也就是以追踪调用链的方式深入阅读，并且提供了数据分析的调试功能。适合分析特定情况下的函数逻辑表现。

## 调用链、调用树和调用图

为了能使用调试器，必须理解函数调用链在调试器级别的表现形式。但是，因为存在内嵌函数和代码优化等原因，调试器的表现形式和源码浏览器下的表现形式是不一样的。它们两者的信息显示可能存在“错位”的现象。本节的目的就是为了磨合调试器和交叉索引工具之间的代沟。

为了简化问题的描述，在实际分析前，先将知识点分解介绍一下。

### 理想调用链

下面我给出一个处于“理想状态”的经典backtrace(backtrace的意思是“回溯”，依照它的作用来说，也就是本人说的调用链)。所谓“理想 状态的”的backtrace是指，可以利用内核源码交叉索引工具，依据gdb给出的这个backtrace，从frame 0开始一级级往后最追溯，能够一直追溯到最前面的frame N，而且追溯的过程中，没有出现多出来的连接frameN和frame(N-1)的“过渡”frame。

注意其中的两个条件：1.能够 2.不多出。但是，在现实的世界里，往往没这么美好。源码浏览工具往往要么“不能”，要么“多出”。造成前者的原因在于源码浏览工具的局限性，造成后者的是内嵌函数以及代码优化。详细情况可看下节的分析。

追溯的方法对于source insight来说就是：打开“relation window”→选中要被追溯的函数→右键→选“view relation”→选“referenced by functions”,这样就能显示出调用了被选函数的函数来。

我们拿下面这个“理想状态”的backtrace分析一下

```
(gdb) bt
#0  kref_init (kref=0xdc40abe4) at lib/kref.c:33
#1  0xc01de8be in kobject_init_internal (kobj=0xdc40abe0) at lib/kobject.c:149
#2  0xc01de928 in kobject_init (kobj=0xdc40abe0, ktype=0xc035b9dc) at lib/kobject.c:149
#3  0xc01de972 in kobject_create () at lib/kobject.c:619
#4  0xc01def53 in kobject_create_and_add (name=0xdc40abe4 "", parent=0xc035b9dc) at lib/kobject.c:619
#5  0xc0393b04 in mnt_init () at fs/namespace.c:2333
#6  0xc039382b in vfs_caches_init (mempages=108676) at fs/dcache.c:2212
#7  0xc037f868 in start_kernel () at init/main.c:666
#8  0xc037f008 in i386_start_kernel () at arch/x86/kernel/head32.c:13
#9  0x00000000 in ?? ()
```

理想状态下的backtrace各个域的含义是(注意，在非理想状态的backtrace中，这些含义往往对不上号)

#frameN的编号 frame(N-1)的返回地址(注：fram0没有这项) in frameN所处的函数(该函数的参数。

我们看下

```
#0  kref_init (kref=0xdc40abe4) at lib/kref.c:33
```

它说明frame0时，kref\_init正要运行。传入的参数是0xdc40abe4。函数kref\_init从源文件lib/kref.c第33行开始。在gdb下调用shell来查看源文件

```
(gdb) shell vi lib/kref.c
```

vi 出来后打命令:set nu可看到

```

31 */
32 void kref_init(struct kref *kref)
33 {
34     kref_set(kref, 1);
35 }
36

```

我们再看看frame0这一瞬间是不是“kref\_init正要运行”。应该知道，“正要运行”和“正要被调用”是两个不同的概念。前者来说，到了下一个指令，代码的控制权就会交给了被调用的函数；而后者，到了下一个指令，代码的控制权还在调用者手里，

```

(gdb) f 0
#0 kref_init (kref=0xdc40abe4) at lib/kref.c:33
33     {
(gdb) info registers
....
edi            0x0      0
eip            0xc01df520 0xc01df520 <kref_init> //<-注意eip是下一个将要运行的指令
eflags        0x282    [ SF IF ]
....
(gdb) disass kref_init
Dump of assembler code for function kref_init:
0xc01df520 <kref_init+0>:    push    %ebp //对比上面，eip指向这里
0xc01df521 <kref_init+1>:    mov     %esp,%ebp
...
0xc01df52f <kref_init+15>:    ret
End of assembler dump.
(gdb)

```

可见，kobject\_init\_internal的调用指令call已经执行完毕，到了frame0时，下一个指令“将要运行”函数kref\_init。

再看看

```

#1 0xc01de8be in kobject_init_internal (kobj=0xdc40abe0) at lib/kobject.c:149

```

frameN与frame(N-1)之间是调用的关系，前者调用了后者。也就是说，frame1的kobject\_init\_internal调用frame0的kref\_init，并且kref\_init函数返回后，将返回到地址0xc01de8be继续执行。0xc01de8be就在kobject\_init\_internal的体内，函数kobject\_init\_internal中调用kref\_init的C语句位于lib/kobject.c的149行。

查看一下kobject\_init\_internal的反汇编码

```
(gdb) disass kobject_init_internal
Dump of assembler code for function kobject_init_internal:
0xc01de8ac <kobject_init_internal+0>:    push    %ebp
0xc01de8ad <kobject_init_internal+1>:    test    %eax,%eax
0xc01de8af <kobject_init_internal+3>:    mov     %esp,%ebp
0xc01de8b1 <kobject_init_internal+5>:    push    %ebx
0xc01de8b2 <kobject_init_internal+6>:    mov     %eax,%ebx
0xc01de8b4 <kobject_init_internal+8>:    je      0xc01de8d3 <kobject_init_internal+39>
0xc01de8b6 <kobject_init_internal+10>:   lea     0x4(%eax),%eax
0xc01de8b9 <kobject_init_internal+13>:   call    0xc01df520 <kref_init>
0xc01de8be <kobject_init_internal+18>:   lea     0x8(%ebx),%eax //注意这个地址0xc01de8be
0xc01de8c1 <kobject_init_internal+21>:   mov     %eax,0x8(%ebx)
```

再看看lib/kobject.c, 看看最后的那个行数的意义

```
145 static void kobject_init_internal(struct kobject *kobj)
146 {
147     if (!kobj)
148         return;
149     kref_init(&kobj->kref); //注意kobject_init_internal调用子函数kref_init
150     INIT_LIST_HEAD(&kobj->entry);
151     kobj->state_in_sysfs = 0;
152     kobj->state_add_uevent_sent = 0;
153     kobj->state_remove_uevent_sent = 0;
154     kobj->state_initialized = 1;
155 }
```

在验证一下

```
#2 0xc01de928 in kobject_init (kobj=0xdc40abe0, ktype=0xc035b9dc) at lib/kobject.c:145
```

看看kobject\_init的反汇编码

```
(gdb) disass kobject_init
Dump of assembler code for function kobject_init:
0xc01de8f3 <kobject_init+0>:    push    %ebp
.....
0xc01de923 <kobject_init+48>:    call    0xc01de8ac <kobject_init_internal>
0xc01de928 <kobject_init+53>:    mov     %esi,0x18(%ebx) //注意这个地址0xc01de928
.....
0xc01de94b <kobject_init+88>:    pop     %ebp
0xc01de94c <kobject_init+89>:    ret
End of assembler dump.
```

看看看看lib/kobject.c, 看看最后的那个行数的意义

```
263 void kobject_init(struct kobject *kobj, struct kobj_type *ktype)
264 {
265     char *err_str;
266     .....
282     kobject_init_internal(kobj);    /////注意kobject_init调用子函数kobject_in
283     kobj->ktype = ktype;
284     .....
287     printk(KERN_ERR "kobject (%p): %s\n", kobj, err_str);
288     dump_stack
```

通过这两个例子, 可见最初的猜想是正确的。

## 函数指针调用

本小节意义: 在利用SI等工具查看函数调用链时, 遇到的一个最多的问题是函数指针的调用。所以把该小节内容移到这里来, 为下小节的叙述作铺垫。SI等交叉索引工具不能在父函数内部解析出这种调用关系。

我们经常碰到这种情况: 如果内核中函数A是通过函数指针调用函数B, 那么源码交叉索引工具(如source insight, kscope等)就无法通过函数B的名称回溯到上层函数A。这是因为在函数A内部对函数B的调用并不是通过函数B的名称, 而是利用指向函数B代码块的指针 (函数指针)。

要想解决这个问题, 方法有两种:

### 1. 利用字符串搜索功能:

搜索函数指针的变量名。如果已经知道的是子函数, 想找出通过指针调用它的所有上层父函数: 利用子函数的函数名进行搜索, 就能找到所有相应的函数指针 变量赋值的语句。然后搜索该函数指针变量就能得到所有可能调用该函数的上层父函数。相反, 如果是已经知道父函数, 想知道该父函数体内的一个函数指针可能会 调用哪些子函数, 可以搜索该函数指针变量(一般在该变量名前加个点号“.”), 这样可以搜索出所有给该函数指针变量赋值的语句, 从而找出所有可能的子函数。

当然, 既然是字符串搜索, 搜索结果中会夹带其他没用的信息, 这需要进行进一步的筛选。这个方法能搜索出依赖某函数指针变量的所有调用关系。

## 2. 利用调试工具：

在目标函数处下断点。调试器会实时拦截该函数的调用，然后用bt命令就能看到整个调用链。

这个方法得到的只是一个特定的具体调用关系。可能还有其他很多的潜在调用路径。

然而，我们研究的目标并不满足于知道调用链。下面我们观察函数究竟是怎样利用函数指针调用子函数的。[\[待整理\]](#)

```

2130 int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
2131 {
2132     int error = may_create(dir, dentry, NULL);
2133
2134     if (error)
2135         return error;
2136
2137     if (!dir->i_op || !dir->i_op->mkdir)
2138         return -EPERM;
2139
2140     mode &= (S_IRWXUGO|S_ISVTX);
2141     error = security_inode_mkdir(dir, dentry, mode);
2142     if (error)
2143         return error;
2144
2145     DQUOT_INIT(dir);
2146     error = dir->i_op->mkdir(dir, dentry, mode);
2147     if (!error)
2148         fsnotify_mkdir(dir, dentry);
2149     return error;
2150 }

```

## 对源码文件下断点

```
(gdb) b fs/namei.c:2146
```

```
Breakpoint 9 at 0xc017c0ee: file fs/namei.c, line 2146.
```

问题一：

动态分析call \*0x14(%ebx)是怎么回事，函数指针

Register group: general				
eax	0xdc20b0a8	-601837400	ecx	0x1e0...
edx	0xdb9526c0	-610982208	ebx	0xe01...
esp	0xd8c5bf1c	0xd8c5bf1c	ebp	0xd8c...
esi	0xdc20b0a8	-601837400	edi	0xdb9...
eip	0xc017c0fb	0xc017c0fb <vfs_mkdir+179>	eflags	0x200...
cs	0x60	96	ss	0x68

ds	0x7b	123	es	0x
fs	0xd8	216	gs	0x

```

0xc017c0ea <vfs_mkdir+162>    mov     %esi,%eax
0xc017c0ec <vfs_mkdir+164>    call    *(%ecx)
B+ 0xc017c0ee <vfs_mkdir+166>    mov     0x98(%esi),%ebx
0xc017c0f4 <vfs_mkdir+172>    mov     %edi,%edx
0xc017c0f6 <vfs_mkdir+174>    mov     %esi,%eax
0xc017c0f8 <vfs_mkdir+176>    mov     -0x10(%ebp),%ecx
> 0xc017c0fb <vfs_mkdir+179>    call    *0x14(%ebx)
0xc017c0fe <vfs_mkdir+182>    test    %eax,%eax
0xc017c100 <vfs_mkdir+184>    mov     %eax,%ebx
0xc017c102 <vfs_mkdir+186>    jne     0xc017c15d <vfs_mkdir+277>
0xc017c104 <vfs_mkdir+188>    testb   $0x4,0x11c(%esi)
0xc017c10b <vfs_mkdir+195>    je      0xc017c119 <vfs_mkdir+209>
0xc017c10d <vfs_mkdir+197>    mov     $0x4,%edx
0xc017c112 <vfs_mkdir+202>    mov     %esi,%eax

```

remote Thread 42000 In: vfs\_mkdir

```

i_state = 1,
dirtied_when = 0,
i_flags = 0,
i_writecount = {
    counter = 0
},
i_security = 0x0,
i_private = 0x0
}

```

(gdb) p/x \$ebx

\$20 = 0xe01c87d4

(gdb) p/x \$ebx+0x14

\$21 = 0xe01c87e8

(gdb) p &sfs\_dir\_inode\_ops

\$13 = (struct inode\_operations \*) 0xe01c87d4

(gdb) p/x \*(int \*)0xe01c87d4@10

\$18 = {0xe01c75b1, 0xe01c7677, 0xc018d3f0, 0xc018cc91, 0xe01c75dd, 0xe01c75c0, 0}

(gdb) disass sfs\_mkdir

Dump of assembler code for function sfs\_mkdir:

```

0xe01c75c0 <sfs_mkdir+0>:    push    %ebp                //<-
0xe01c75c1 <sfs_mkdir+1>:    or      $0x40,%ch
0xe01c75c4 <sfs_mkdir+4>:    mov     %esp,%ebp
0xe01c75c6 <sfs_mkdir+6>:    push    %ebx
0xe01c75c7 <sfs_mkdir+7>:    mov     %eax,%ebx

```



```

0xe01c75c9 <sfs_mkdir+9>:      push    $0x0
0xe01c75cb <sfs_mkdir+11>:     call    0xe01c7510 <sfs_mknod>
0xe01c75d0 <sfs_mkdir+16>:     pop     %edx
0xe01c75d1 <sfs_mkdir+17>:     test    %eax,%eax
0xe01c75d3 <sfs_mkdir+19>:     jne     0xe01c75d8 <sfs_mkdir+24>
0xe01c75d5 <sfs_mkdir+21>:     incl    0x28(%ebx)
0xe01c75d8 <sfs_mkdir+24>:     mov     -0x4(%ebp),%ebx
0xe01c75db <sfs_mkdir+27>:     leave
0xe01c75dc <sfs_mkdir+28>:     ret

```

End of assembler dump.

(gdb) p/x \*0xe01c87e8

\$9 = 0xe01c75c0 // <-sfs\_mkdir的地址

(gdb)

```

struct inode_operations {
    int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
    struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct inode *,struct dentry *,const char *);
    int (*mkdir) (struct inode *,struct dentry *,int);
    .....
};

```

```

struct inode_operations sfs_dir_inode_ops = {
...
    .mkdir          = sfs_mkdir,
...
};

```

---

```

0xc017c0fb <vfs_mkdir+179>      call    *0x14(%ebx) 为什么要加 * ?

```

```

call *0x14(%ebx) ==
push %eip
mov 0x14(%ebx) %eip

```

注意call与mov指令语义的区别

mov 0x14(%ebx) %eax; 把存放在地址0x14(%ebx)中的32位数据拷贝到%eax  
mov %eax 0x14(%ebx); 把%eax的值拷贝到地址0x14(%ebx)指向的内存中  
call 0x14(%ebx) : 结果是跳到地址0x14(%ebx)继续执行(当然对于本例来说, 该地址指向的并非代码)  
call \*0x14(%ebx) : 取出存放在地址0x14(%ebx)中的32位数据, 把该数据作为目标地址, 跳到该地址继续执行

mov \$0xe01c75c9 %eax ; 0xe01c75c9被认为是立即数, 前面有\$。没有mov 0xe01c75c9 %eax这种形式。  
call 0xe01c75c9 ; 0xe01c75c9被认为是地址。没有call \$0xe01c75c9这种形式。  
注意, 也没有call %eax等形式(假设%eax放着目标地址)。需用 call \*%eax, 同样, \*%eax表示从%eax指向的地址中取出数据

	-		--
	-		--
	-		--



```

static int sfs_mkdir(..)
0xe01c75c0 <sfs_mkdir+0>:
0xe01c75c1 <sfs_mkdir+1>:
0xe01c75c4 <sfs_mkdir+4>:
0xe01c75c6 <sfs_mkdir+6>:
0xe01c75c7 <sfs_mkdir+7>:
0xe01c75c9 <sfs_mkdir+9>:
0xe01c75cb <sfs_mkdir+11>:
0xe01c75d0 <sfs_mkdir+16>:
0xe01c75d1 <sfs_mkdir+17>:
0xe01c75d3 <sfs_mkdir+19>:

0xe01c75d5 <sfs_mkdir+21>:
0xe01c75d8 <sfs_mkdir+24>:
0xe01c75db <sfs_mkdir+27>:
0xe01c75dc <sfs_mkdir+28>:

```

address

问题二:

下面的`dir->i_op->mkdir()`, 为什么不是`dir.i_op.mkdir.` 和 `->` 有什么区别

一般得, 有一个结构体变量`a`, 其中`a`有一个域`b`, 想取得`b`的值, 一般用`a.b`;  
而如果`a`是一个指向结构体的指针变量, 取域`b`的值一般用`a->b`.

```

static int sfs_mkdir(struct inode * dir, struct dentry * dentry, int mode)
{
....
}

```

```

2130 int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
2131 {

```

```

2146         error = dir->i_op->mkdir(dir, dentry, mode);
...
2150 }

```

```

struct inode {
...
const struct inode_operations *i_op;
...
};

struct inode_operations {
...
    int (*mkdir) (struct inode *, struct dentry *, int);
...
};

```

dir: 取得(struct inode \*)dir

dir->i\_op: 取得(const struct inode\_operations \*)i\_op

dir->i\_op->mkdir: 取得(int (\*)(struct inode \*, struct dentry \*, int))mkdir

dir->i\_op->mkdir(dir, dentry, mode)也就是 函数指针变量名(参数...)

函数指针是一个指针，它向目标函数的代码块的第一个指令。

函数名的值等于该函数第一条指令的地址。

```

(gdb) p sfs_mkdir
$20 = {int (struct inode *, struct dentry *, int)} 0xe01c75c0 <sfs_mkdir>
(gdb) p &sfs_mkdir
$21 = (int (*)(struct inode *, struct dentry *, int)) 0xe01c75c0 <sfs_mkdir>
(gdb) p dir->i_op->mkdir
$18 = (int (*)(struct inode *, struct dentry *, int)) 0xe01c75c0 <sfs_mkdir>
前者指明变量名/函数名的类型，后者是它的值

```

```

struct inode_operations sfs_dir_inode_ops = {
...
    .mkdir = sfs_mkdir,
...
};

```

函数的两种调用形式： 函数指针变量名(参数...) 函数名(参数...)

严格地说，从C语言的形式看来，前者通过函数指针变量名调用函数，后者通过函数名调用，是不同的。但从汇编级代码看来，都是转化为指令call 函数地址。是一样的。

引入了函数指针变量后，这个变量就可以动态地赋值，从而指向不同的函数体，实现某些特殊的功能。

我们再看下函数指针的赋值.mkdir = sfs\_mkdir,

严格地说，mkdir和sfs\_mkdir是类型不同的东西，但在编译时自动经过了类型转换。所以下面这些类型

```

.mkdir = sfs_mkdir,
.mkdir = &sfs_mkdir,
.mkdir = (int (*)(struct inode *, struct dentry *, int))sfs_mkdir,
.mkdir = (int (*)(struct inode *, struct dentry *, int))(&sfs_mkdir),

```

假设有函数指针 `a`，要调用该指针指向的函数，有意思的是，又有两形式：

`a(参数);`

或者

`(*a)(参数);`

硬要理解的话，可以这样想：`a`之指向一个函数体的指针，按么`(*a)`自然就是得到了那个函数本身了。`(*a)(参数)`这个形式就变成了普通的函数调用，这比直接利用函数指针的变量名进行调用更直观，或者有人错误地认为本来就应该这样调用才正确。其实，这些形式上的东西是给人看的，到了汇编级别的指令都是一样的。

linux使用第一种形式，而openbsd使用下面的第二种。linux不管在函数指针的赋值还是函数的调用都是采取最简洁的形式。

举个openbsd下的例子，可见它的形式不是很美观。

```
int
sobind(struct socket *so, struct mbuf *nam, struct proc *p)
{
    int s = splsoftnet();
    int error;

    error = (*so->so_proto->pr_usrreq)(so, PRU_BIND, NULL, nam, NULL, p);
    splx(s);
    return (error);
}
```

其中结构体：

```
struct socket {
...
    struct protosw *so_proto;      /* protocol handle */
...
}

struct protosw {
...
    int (*pr_usrreq)(struct socket *, int, struct mbuf *,
                     struct mbuf *, struct mbuf *, struct proc *);
...}
```

可见，下面这句

```
(*so->so_proto->pr_usrreq)(so, PRU_BIND, NULL, nam, NULL, p);
```

最后变成

```
(*pr_usrreq)(so, PRU_BIND, NULL, nam, NULL, p);
```

---

例子

```
#include <stdio.h>
```

```
int main()
```

```

int myfunc(int a)
{
    printf("%d\n", a);
    return 0;
}
int (*funa)(int) = myfunc;
int (*funb)(int) = &myfunc;
int (*func)(int) = (int (*)(int))myfunc;
int (*fund)(int) = (int (*)(int))(&myfunc);

myfunc(1);
funa(2);
funb(3);
func(4);
fund(5);

(*funa)(2);
(*funb)(3);
(*func)(4);
(*fund)(5);

return 0;
}

```

那几个函数调用的代码部分的反汇编码如下，可见在汇编级别，是一样的指令。

```

08048374 <main>:
8048374: 8d 4c 24 04      lea    0x4(%esp),%ecx
8048378: 83 e4 f0        and    $0xffffffff0,%esp
804837b: ff 71 fc        pushl  -0x4(%ecx)
804837e: 55             push   %ebp
804837f: 89 e5          mov    %esp,%ebp
8048381: 51             push   %ecx
8048382: 83 ec 14        sub    $0x14,%esp
8048385: c7 45 f8 1b 84 04 08 movl    $0x804841b,-0x8(%ebp)
804838c: c7 45 f4 1b 84 04 08 movl    $0x804841b,-0xc(%ebp)
8048393: c7 45 f0 1b 84 04 08 movl    $0x804841b,-0x10(%ebp)
804839a: c7 45 ec 1b 84 04 08 movl    $0x804841b,-0x14(%ebp)
80483a1: c7 04 24 01 00 00 00 movl    $0x1, (%esp)
80483a8: e8 6e 00 00 00  call   804841b <myfunc.1932>
80483ad: c7 04 24 02 00 00 00 movl    $0x2, (%esp)
80483b4: 8b 45 f8        mov    -0x8(%ebp),%eax
80483b7: ff d0          call   *%eax
80483b9: c7 04 24 03 00 00 00 movl    $0x3, (%esp)
80483c0: 8b 45 f4        mov    -0xc(%ebp),%eax
80483c3: ff d0          call   *%eax
80483c5: c7 04 24 04 00 00 00 movl    $0x4, (%esp)
80483cc: 8b 45 f0        mov    -0x10(%ebp),%eax
80483cf: ff d0          call   *%eax
80483d1: c7 04 24 05 00 00 00 movl    $0x5, (%esp)
80483d8: 8b 45 ec        mov    -0x14(%ebp),%eax
80483db: ff d0          call   *%eax
80483dd: c7 04 24 02 00 00 00 movl    $0x2, (%esp)

```

80483e4:	8b 45 f8	mov	-0x8(%ebp),%eax
80483e7:	ff d0	call	*%eax
80483e9:	c7 04 24 03 00 00 00	movl	\$0x3, (%esp)
80483f0:	8b 45 f4	mov	-0xc(%ebp),%eax
80483f3:	ff d0	call	*%eax
80483f5:	c7 04 24 04 00 00 00	movl	\$0x4, (%esp)
80483fc:	8b 45 f0	mov	-0x10(%ebp),%eax
80483ff:	ff d0	call	*%eax
8048401:	c7 04 24 05 00 00 00	movl	\$0x5, (%esp)
8048408:	8b 45 ec	mov	-0x14(%ebp),%eax
804840b:	ff d0	call	*%eax

## 调用链的层次

1. 人观念层次
2. 交叉解析器层次
2. c调用层次
3. 编译器(机器码静态)层次
4. 运行时(机器码动态)层次，也叫调试器层次

很明显，前面所讲的“理想状态”的backtrace就是指在交叉解析器层次下和在调试器层次下的表现相同的调用链。

## 非理想调用链

任务：

从一个断点开始，从后向前推导，分析出ramfs注册函数的调用过程。同时，观察调试器的优点和局限性。

ramfs文件系统的注册函数是register\_filesystem(&ramfs\_fs\_type)。为了更快定位，在上层函数init\_ramfs\_fs下断点。而后在gdb下得到的调用链是

```
(gdb) bt
#0  register_filesystem (fs=0xc03595cc) at fs/filesystems.c:68
#1  0xc0394594 in init_ramfs_fs () at fs/ramfs/inode.c:213
#2  0xc037f473 in kernel_init (unused=<value optimized out>) at init/main.c:708
#3  0xc010463f in kernel_thread_helper () at arch/x86/kernel/entry_32.S:1013
```

我们注意到：

1. 这个backtrace包含的函数只有4个，实际上并非如此。经过分析，它实际上（用C的观点看）调用链如下所示，这是为什么呢？

start\_kernel→rest\_init→kernel\_thread→kernel\_thread\_helper→call %ebx (即call kernel\_init)→do\_basic\_setup→do\_initcalls→do\_one\_initcall→result = fn() (即call init\_ramfs\_fs)→register\_filesystem

2. backtrace回溯到kernel\_thread\_helper后就再没下文了。又是什么使得调试器变成了瞎子，无法看得再远了呢？

欲见其详，且听下回分解

[下面准备材料]

kernel\_init对do\_basic\_setup的调用被优化成内联函数

do\_basic\_setup对do\_initcalls的调用被优化成内联函数

do\_initcalls对do\_one\_initcall的调用被优化成内联函数

有三层的非内联函数都被被优化成内联函数，整个代码被优化的乱七八糟。

```
838 static int __init kernel_init(void * unused)
839 {
840     .....
864     cpuset_init_smp();
865
866     do_basic_setup();
867
868     .....
887     return 0;
888 }
```

```
static void __init do_basic_setup(void)
{
    /* drivers will send hotplug events */
    init_workqueues();
    usermodehelper_init();
    driver_init();
    init_irq_proc();
    do_initcalls();
}
```

```

}

741 static void __init do_initcalls(void)
742 {
743     initcall_t *call;
744
745     for (call = __initcall_start; call < __initcall_end; call++)
746         do_one_initcall(*call);
747
748     /* Make sure there is no pending stuff from the initcall sequence */
749     flush_scheduled_work();
750 }

```

```

static void __init do_one_initcall(initcall_t fn)
{
    int count = preempt_count();
    ktime_t t0, t1, delta;
    char msgbuf[64];
    int result;

    if (initcall_debug) {
        print_fn_descriptor_symbol("calling %s\n", fn);
        t0 = ktime_get();
    }

    result = fn();

    if (initcall_debug) {
        ....
    }
}

```

```

static inline void print_fn_descriptor_symbol(const char *fmt, void *addr)
{
    #if defined(CONFIG_IA64) || defined(CONFIG_PPC64)
        addr = *(void **)addr;
    #endif

    print_symbol(fmt, (unsigned long)addr);
}

```

(gdb) disass kernel\_init

Dump of assembler code for function kernel\_init:

```

0xc037f349 <kernel_init+0>:    push    %ebp
0xc037f34a <kernel_init+1>:    mov     %esp,%ebp
0xc037f34c <kernel_init+3>:    push    %edi
0xc037f34d <kernel_init+4>:    push    %esi
.....
0xc037f413 <kernel_init+202>:   call    0xc0391454 <cpuset_init_smp>
0xc037f418 <kernel_init+207>:   call    0xc0390081 <init_workqueues> //<-do_basi
0xc037f41d <kernel_init+212>:   call    0xc039004e <usermodehelper_init>
0xc037f422 <kernel_init+217>:   call    0xc039b7d1 <driver_init>
0xc037f427 <kernel_init+222>:   call    0xc0153e18 <init_irq_proc>
0xc037f42c <kernel_init+227>:   movl    $0xc03aa470,-0x5c(%ebp) //do_initcalls被
0xc037f433 <kernel_init+234>:   pop     %eax

```



```

0xc037f434 <kernel_init+235>: pop    %edx
0xc037f435 <kernel_init+236>: jmp    0xc037f559 <kernel_init+528>
0xc037f43a <kernel_init+241>: mov    -0x5c(%ebp),%eax //do_one_initcall被优化成内
0xc037f43d <kernel_init+244>: mov    (%eax),%eax
0xc037f43f <kernel_init+246>: mov    %eax,-0x58(%ebp)
0xc037f442 <kernel_init+249>: mov    %esp,%eax
0xc037f444 <kernel_init+251>: and    $0xfffffe00,%eax
0xc037f449 <kernel_init+256>: mov    0x14(%eax),%eax
0xc037f44c <kernel_init+259>: cmpl   $0x0,0xc03a1820
0xc037f453 <kernel_init+266>: mov    %eax,-0x54(%ebp)
0xc037f456 <kernel_init+269>: je     0xc037f470 <kernel_init+295>
0xc037f458 <kernel_init+271>: mov    -0x58(%ebp),%edx //内联函数print_fn_descrip
0xc037f45b <kernel_init+274>: mov    $0xc030d1be,%eax
0xc037f460 <kernel_init+279>: call   0xc013f598 <__print_symbol> //内联函数print_fi
0xc037f465 <kernel_init+284>: call   0xc013352f <ktime_get>
0xc037f46a <kernel_init+289>: mov    %eax,-0x64(%ebp)
0xc037f46d <kernel_init+292>: mov    %edx,-0x60(%ebp)
0xc037f470 <kernel_init+295>: call   *-0x58(%ebp) //do_one_initcall中的调用语句re

.....
0xc037f553 <kernel_init+522>: pop    %edi
0xc037f554 <kernel_init+523>: pop    %eax
0xc037f555 <kernel_init+524>: addl   $0x4,-0x5c(%ebp)
0xc037f559 <kernel_init+528>: cmpl   $0xc03aa804,-0x5c(%ebp) //
0xc037f560 <kernel_init+535>: jb     0xc037f43a <kernel_init+241> //

```

## 如何在汇编码中定位内联（或被优化掉的非内联）函数

1. 利用前后相关函数的提示
2. 函数的前戏码定位函数的开始
3. 注意跳转语句
4. 利用调试器辅助定位(见gdb技巧)

## 调用树与调用图

[待充实]

### 调用树的定义

一个复杂的函数调用一定是调用了多个子函数，同时这些子函数又会调用若干“孙”函数，这样依次调用并依次返回到最初的父函数后，就形成了树状的调用关系，我们称之为“调用树”。

## 调用树的作用

函数调用树是比函数调用链更为复杂的观察对象。如果能够显示调用树，就可以对调用的整个过程有个直观的了解。

## 调用树的分类

函数调用树有两类：

### 1. 抽象调用树

也叫虚拟调用树。比如在源码中，父函数调用了子函数a, b, c。那么对这三个函数的调用逻辑都考虑进去，这就是“抽象调用”。抽象调用树能全面的描述了父函数的逻辑和代码开发员的意图。但是，在实际的环境中，这三个函数未必就全部会调用到。把在实际的具体情况下未调用的“潜在”调用关系去掉后，剩下的调用树就称为“具体调用树”。明显，具体调用树不能全面显示代码开发员的意图，只是放映具体环境下函数的调用关系。

### 2. 具体调用树

也叫实时调用树。解释见上。

## 调用树的显示

### 1. 抽象调用树的显示

借助source insight等工具可以图形显示抽象调用树。

### 1. 具体调用树的显示

据本人的了解，目前gdb没有一个类似“bt”那样的能显示函数调用树的命令，但是借助gdb宏也许能够实现显示调用树的功能，这有待研究。不过，目前已经有个现成的调试工具可以显示调用树，它就是 systemtap。

效果如下：

```
[...]  
0 klogd(1391):->sys_read  
14 klogd(1391): ->fget_light
```

```
22 klogd(1391): <-fget_light
27 klogd(1391): ->vfs_read
35 klogd(1391): ->rw_verify_area
43 klogd(1391): <-rw_verify_area
49 klogd(1391): ->kmsg_read
 0 sendmail(1696):->sys_read
17 sendmail(1696): ->fget_light
26 sendmail(1696): <-fget_light
34 sendmail(1696): ->vfs_read
44 sendmail(1696): ->rw_verify_area
52 sendmail(1696): <-rw_verify_area
58 sendmail(1696): ->proc_file_read
70 sendmail(1696): ->loadavg_read_proc
84 sendmail(1696): ->proc_calc_metrics
92 sendmail(1696): <-proc_calc_metrics
95 sendmail(1696): <-loadavg_read_proc
101 sendmail(1696): <-proc_file_read
106 sendmail(1696): ->dnotify_parent
115 sendmail(1696): <-dnotify_parent
119 sendmail(1696): ->inotify_dentry_parent_queue_event
127 sendmail(1696): <-inotify_dentry_parent_queue_event
133 sendmail(1696): ->inotify_inode_queue_event
141 sendmail(1696): <-inotify_inode_queue_event
146 sendmail(1696): <-vfs_read
151 sendmail(1696):<-sys_read
[...]
```

见于

[http://sourceware.org/systemtap/wiki/WSCallGraph?highlight=1\)](http://sourceware.org/systemtap/wiki/WSCallGraph?highlight=1)

### 调用树的拼接

对于一个更刁的函数调用来说，利用工具显示的抽象调用树和具体调用调用树可能是不完整的。比如，对于抽象调用树来说，它的显示工具是source insight。但是如果这个函数对某个子函数或在更下层的函数对下下层的函数调用是通过函数指针来调用的，那么source insight显示的调用树中就会漏掉通过函数指针调用的子函数，以及以子函数为根的子调用树。这是因为函数指针变量的赋值是发生在代码动态运行时的。source insight无法利用静态的源码就捕捉到未来才出现东西，甚至它也无法在形式上解析出“那里存在一个利用函数指针的调用”。这就要通过阅读源码来找出这种调用关系。同时，可以利用调试器实时找出具体情况下是通过那个函数指针调用了哪个特定的下层函数。这样就能把漏掉的子调用树拼接到父调用树中。

可见，这些内容又回归到了调用链的内容。具体看前面。

## 调用图

各函数间的像蜘蛛网一样的调用关系的图形表示就是调用图了，显然它比调用树更复杂。

## 穿越盲区

本节意义：经过上面章节的叙述，利用源码交叉索引工具 + 调试器已经能解决大部分问题，但是因为调试器和交叉索引工具的各自局限性，依然会存在一些问题。本节尝试如何联合交叉索引工具以及调试器再加上人脑来解决各自的缺点。

[观察积累中，待扩展]

### 穿越gdb的盲区

进程切换

中断异常

系统调用

### 穿越交叉索引工具的盲区

函数指针

该小节内容移到了：调用链的状态→函数指针调用

### 查看函数的参数

我们知道，一个函数的计算结果并不都是通过它的返回值返回的，有时会通过函数的参数返回真正感兴趣的数据。看内核源码的时候，如果调用链过长，涉及内容和数据结构过多的话，往往是看到最后都记不住函数的参数哪些是已经“初始化的”。

这也是交叉索引工具无法克服的先天弱点。它能动态索引源码，却无法动态查看数据。此时，可以利用gdb给目标函数下断点，而后可以用命令info args查看参数，另外命令info local可查看本地变量。当然在ddd下查看效果会更好。

内容简单，不展开了。

## 工程方法

## 二叉断点

实例“什么/proc下无法创建目录？”

### 给调用指令下断点

如果对目标函数下断点后，受到很多骚扰，那么就转为在上层函数内对目标函数的调用指令处下断点。如果你已经进入了上层函数，对调用指令下断点，是更为精确的断点方法。

### 绕过时钟中断的干扰

有时我们调试的程序与中断无关的，但是由于时钟中断的异步到来，在调试过程中经常会自动进入时钟中断处理例程中，这严重干扰了我们的工作。用下面的方法可绕过时钟中断的干扰。

注：

使用GDB与QEMU调试内核时的问题分析：<http://www.chinaitlab.com/linux/kernel/356774.html>

关于qemu在单步指令时进入时钟中断的问题，上面给出链接给出了比较“深入”地探讨。这个问题涉及虚拟机本身，有人说是虚拟机相对于真机的固有缺陷，似乎很深邃，我没那个能力也没那个时间研究。但是我们应该知道，如果问题足够的复杂，以至于解决它要花费太高的代价，那么绕过这个问题是个更明智的解决方法。

### 解决方法(手工)

#### 1. 内核启动早期

事先下两个断点

```
b common_interrupt
b native_iret
```

### 自定义返回命令

```
(gdb) define ooi
Type commands for definition of "ooi".
```

```
End with a line saying just "end".
>c
>stepi
>end
```

一旦时钟中断产生，就会拦截在中断处理的通用入口common\_interrupt，然后运行返回指令，就会“回到”被时钟中断打断的原指令处

ooi

## 2. 内核启动完毕

事先下两个断点

```
b apic_timer_interrupt
b irq_return
```

一旦时钟中断产生，就会拦截在中断处理例程apic\_timer\_interrupt，然后运行返回指令，就会“回到”被时钟中断打断的原指令处

ooi

分析记录，待整理

提示，分析异常和中断的处理过程比分析C代码更直观，因为源码本身是汇编码。

```
arch/x86/kernel/entry_32.S
614      SAVE_ALL
615      TRACE_IRQS_OFF
616      movl %esp,%eax
617      call do_IRQ
> 618      jmp ret_from_intr
619      ENDPROC(common_interrupt)
620      CFI_ENDPROC
621
622      #define BUILD_INTERRUPT(name, nr) \
623      ENTRY(name) \
624      RING0_INT_FRAME; \
625      pushl $~(nr); \
626      CFI_ADJUST_CFA_OFFSET 4; \
627      SAVE_ALL; \

0xc01043e1 <common_interrupt+17>      mov     %edx,%ds
0xc01043e3 <common_interrupt+19>      mov     %edx,%es
0xc01043e5 <common_interrupt+21>      mov     $0xd8,%edx
0xc01043ea <common_interrupt+26>      mov     %edx,%fs
0xc01043ec <common_interrupt+28>      mov     %esp,%eax
0xc01043ee <common_interrupt+30>      call    0xc0106151 <do_IRQ>
> 0xc01043f3 <common_interrupt+35>      imn     0xc01038dc <ret_from_exception>
```

0xc01043f8	<reschedule_interrupt>	push	\$0xfffffffff03
0xc01043fd	<reschedule_interrupt+5>	cld	
0xc01043fe	<reschedule_interrupt+6>	push	%fs
0xc0104400	<reschedule_interrupt+8>	push	%es
0xc0104401	<reschedule_interrupt+9>	push	%ds
0xc0104402	<reschedule_interrupt+10>	push	%eax
0xc0104403	<reschedule_interrupt+11>	push	%ebp

remote Thread 42000 In: common\_interrupt

(gdb)

(gdb)

(gdb) bt

```
#0 common_interrupt () at arch/x86/kernel/entry_32.S:618
#1 0x00000292 in ?? ()
#2 0xc01880db in alloc_vfsmnt (name=0xc031dcf3 "rootfs") at include/linux/slab.h:110
#3 0xc0176919 in vfs_kern_mount (type=0xc0359678, flags=0, name=0xc031dcf3 "rootfs") at fs/vfs.c:110
#4 0xc0176a2f in do_kern_mount (fstype=0xc031dcf3 "rootfs", flags=0, name=0xc031dcf3 "rootfs") at fs/vfs.c:110
#5 0xc0393b33 in mnt_init () at fs/namespace.c:2285
#6 0xc039382b in vfs_caches_init (mempages=108676) at fs/dcache.c:2212
#7 0xc037f868 in start_kernel () at init/main.c:666
#8 0xc037f008 in i386_start_kernel () at arch/x86/kernel/head32.c:13
#9 0x00000000 in ?? ()
```

(gdb) disass

(gdb)

----

arch/x86/kernel/entry_32.S			
401	cmpl	\$((SEGMENT_LDT << 8)   USER_RPL), %eax	
402	CFI_REMEMBER_STATE		
403	je	ldt_ss	# returning to user-space with
404	restore_nocheck:		
405	TRACE_IRQS_IRET		
406	restore_nocheck_notrace:		
407	RESTORE_REGS		
408	addl	\$4, %esp	# skip orig_eax/error_code
409	CFI_ADJUST_CFA_OFFSET	-4	
410	irq_return:		
> 411	INTERRUPT_RETURN		
412	.section	.fixup, "ax"	
413	ENTRY(iret_exc)		
414	pushl	\$0	# no error code

0xc0103a61	<restore_nocheck_notrace>	pop	%ebx
0xc0103a62	<restore_nocheck_notrace+1>	pop	%ecx
0xc0103a63	<restore_nocheck_notrace+2>	pop	%edx
0xc0103a64	<restore_nocheck_notrace+3>	pop	%esi
0xc0103a65	<restore_nocheck_notrace+4>	pop	%edi
0xc0103a66	<restore_nocheck_notrace+5>	pop	%ebp
0xc0103a67	<restore_nocheck_notrace+6>	pop	%eax
0xc0103a68	<restore_nocheck_notrace+7>	pop	%ds
0xc0103a69	<restore_nocheck_notrace+8>	pop	%es
0xc0103a6a	<restore_nocheck_notrace+9>	pop	%fs
0xc0103a6c	<restore_nocheck_notrace+11>	add	\$0x4,%esp
> 0xc0103a6f	<irq_return>	jmp	*%cs:0xc0353b54
0xc0103a70	<ldt_ss>	ldl	0xc03a70(%eax), %eax

0xc0103a7b <ldt\_ss+5>

jne 0xc0103a61 <restore\_nocheck\_no

remote Thread 42000 In: irq\_return

(gdb) stepi

0xc0103a64 in restore\_nocheck\_notrace () at arch/x86/kernel/entry\_32.S:407

0xc0103a65 in restore\_nocheck\_notrace () at arch/x86/kernel/entry\_32.S:407

0xc0103a66 in restore\_nocheck\_notrace () at arch/x86/kernel/entry\_32.S:407

Watchpoint 3: \$ebp

Old value = (void \*) 0xc0378000

New value = (void \*) 0xc0379f4c

0xc0103a67 in restore\_nocheck\_notrace () at arch/x86/kernel/entry\_32.S:407

0xc0103a68 in restore\_nocheck\_notrace () at arch/x86/kernel/entry\_32.S:407

0xc0103a69 in restore\_nocheck\_notrace () at arch/x86/kernel/entry\_32.S:407

0xc0103a6a in restore\_nocheck\_notrace () at arch/x86/kernel/entry\_32.S:407

restore\_nocheck\_notrace () at arch/x86/kernel/entry\_32.S:408

irq\_return () at arch/x86/kernel/entry\_32.S:411

(gdb)

-----

arch/x86/kernel/entry\_32.S

864 jmp irq\_return

865 CFI\_ENDPROC

866 KPROBE\_END(nmi)

867

868 #ifdef CONFIG\_PARAVIRT

869 ENTRY(native\_iret)

> 870 iret

871 .section \_\_ex\_table,"a"

872 .align 4

873 .long native\_iret, iret\_exc

874 .previous

875 END(native\_iret)

876

877 ENTRY(native\_irq\_enable\_syscall\_ret)

> 0xc01045a8 <native\_iret>

iret

0xc01045a9

lea 0x0(%esi),%esi

0xc01045ac <native\_irq\_enable\_syscall\_ret>

sti

0xc01045ad <native\_irq\_enable\_syscall\_ret+1>

sysexit

0xc01045af

nop

0xc01045b0 <overflow>

push \$0x0

0xc01045b2 <overflow+2>

push \$0xc0105030

0xc01045b7 <overflow+7>

jmp 0xc02bfb80 <page\_fault

0xc01045bc <bounds>

push \$0x0

0xc01045be <bounds+2>

push \$0xc0104fe3

0xc01045c3 <bounds+7>

jmp 0xc02bfb80 <page\_fault

0xc01045c8 <invalid\_op>

push \$0x0

0xc01045ca <invalid\_op+2>

push \$0xc0104f6d

0xc01045cf <invalid\_op+7>

jmp 0xc02bfb80 <page\_fault

remote Thread 42000 In: native\_iret

#9 0x00000000 in ?? ()

(gdb) stepi

native\_iret () at arch/x86/kernel/entry\_32.S:870



$\wedge$ 

```
(gdb)
```

mm/slab.c



```
remote Thread 42000 In: kmem cache alloc
```

```
#7  0xc037f008 in i386_start_kernel () at arch/x86/kernel/head32.c:13
#8  0x00000000 in ?? ()
```

```
(gdb) bt
```

```
#4  0xc0555055 in init_init () at fs/namespace.c:2285
#5  0xc030382b in ufs_cache_init (messages=108676) at fs/ufs/cach.c:3313
```

```
#6 0xc037f868 in start_kernel () at init/main.c:666
#7 0xc037f008 in i386_start_kernel () at arch/x86/kernel/head32.c:13
#8 0x00000000 in ?? ()
(gdb) list
(gdb) disass
(gdb)
```

## bug 与 OOPS

[主要研究定位bug的技巧，找出是哪条指令引发了panic似乎很容易。但要找出错误产生的源头似乎是门艺术了]

经过上面章节的叙述，本小节问题的解决已不成问题了。不再展开叙述。可以参考下面链接。

### 参考手册

“Using kgdb and the kgdb Internals” <http://www.kernel.org/pub/linux/kernel/people/jwessel/kgdb/index.html>

kgdb官网 <http://kgdb.linsyssoft.com/>

### 参考书籍 (freebsd)

“Debugging Kernel Problems” <http://www.google.cn/search?q=Debugging+Kernel+Problems&ie=utf-8&oe=utf-8&aq=t&rls=com.ubuntu:zh-CN:unofficial&client=firefox-a>

“Chapter 10 Kernel Debugging” [http://www.freebsd.org/doc/en\\_US.ISO8859-1/books/developers-handbook/kerneldebug.html](http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/kerneldebug.html)

### 参考书籍(linux)

Chapter 14. Kernel Debugging Techniques of “Embedded Linux Primer: A Practical, Real-World Approach”

<http://book.opensourceproject.org.cn/embedded/embeddedprime/>

### 参考文章

“掌握 Linux 调试技术” <http://www.ibm.com/developerworks/cn/linux/sdk/l-debug/index.html>

“定位Oops的具体代码行” [http://blog.chinaunix.net/u/12592/showart\\_1092733.html](http://blog.chinaunix.net/u/12592/showart_1092733.html)

“跟踪内核 oops” <http://wiki.zh-kernel.org/doc/oops-tracing.txt>

“例解Linux Kernel Debug” [http://blog.chinaunix.net/u/2108/showart\\_164703.html](http://blog.chinaunix.net/u/2108/showart_164703.html)

“kernel debug的一些小手段” [http://blog.chinaunix.net/u/12592/showart\\_499502.html](http://blog.chinaunix.net/u/12592/showart_499502.html)

“Kernel Debugging Techniques” <http://www.linuxjournal.com/article/9252>

[参考文章] 有的已过时，而且深度不够。

## 网站

<http://bugzilla.kernel.org/>

<http://www.kerneloops.org/>

<http://www.lkml.org/> 搜索bug

## \*\*\*第二部分：内核分析\*\*\*

这部分的内容侧重于内核原理分析，其中涉及gdb调试器的内容不是很多，但它起的作用很关键，主要用于观察内核数据的生成及变化，在对源码理解有困惑时用于验证自己的猜想。另外，gdb一个很重要的功能是，拦截通过函数指针调用的函数，从而追溯整个调用链，交叉索引工具无法做到这点。

另外，调试内核时，利用gdb的“list 函数名”命令看到的C代码都是当前处理器当前配置下内核实际运行的函数版本：“disass 函数名”看到的都是处理器实际运行时的机器代码，也就是说define语句和inline函数已经被编译器处理了，而且编译器也完成了优化。所以，gdb 本身就是一种不可替代的源码浏览工具，它能筛选掉出实际运行的函数版本，又能呈现出实际运行的机器码。

## 调试相关子系统

### kgdb源码分析

gdb远程串口协议

[http://sourceware.org/gdb/current/onlinedocs/gdb\\_34.html#SEC706](http://sourceware.org/gdb/current/onlinedocs/gdb_34.html#SEC706)

[http://www.huihoo.org/mirrors/pub/embed/document/debugger/ew\\_GDB\\_RSP.pdf](http://www.huihoo.org/mirrors/pub/embed/document/debugger/ew_GDB_RSP.pdf)

Jason Wessel的linux-2.6-kgdb.git

<http://git.kernel.org/?p=linux/kernel/git/jwessel/linux-2.6-kgdb.git;a=summary>

gdb调试模式

```
(gdb) set debug serial 1
(gdb) set debug remote 1
```

**sysrq**

**oprofile**

**kprobes**

## 驱动分析

[分析一个简单的驱动，观察函数调用流程。重点观察驱动与驱动模型，以及和系统内核的交互过程。比如，中断的整个生命周期。]

参考：

“Debugging kernel modules” <http://lwn.net/Articles/90913/>

“Linux 系统内核的调试” <http://www.ibm.com/developerworks/cn/linux/l-kdb/>

“Linux 可加载内核模块剖析” <http://www.ibm.com/developerworks/cn/linux/l-lkm/>

“使用 KGDB 调试 Linux 内核” [http://blog.chinaunix.net/u/8057/showart\\_1087126.html](http://blog.chinaunix.net/u/8057/showart_1087126.html)

“使用 /proc 文件系统来访问 Linux 内核的内容” <http://www.ibm.com/developerworks/s/cn/linux/l-proc.html>

如何查找出当前系统所安装模块驱动对应的源码，从而对其做些修改等实验？

提示：

1. lsmod 列出模块名
2. modinfo 模块名， 查看模块信息
3. 模块名， 模块信息中的别名， 模块的参数说明文字都可结合source insight查找该模块的源码文件；模块信息中的模块路径也可用来定位对应源码的路径以及相关的kconfig文件，从而获取更多相关信息。一般源码文件的名称就是模块名或在模块名的基础上加上某些后缀，用模块名的方法查找不出时再利用其他信息查找。
4. 如果利用以上方法还找不到源文件，或者一个模块对应着几个源文件，可使用最后的必杀绝招。比如lsmod后得到一个sr\_mod。我们用modinfo sr\_mod的得到它的已编译文件的路径是 /lib/modules/2.6.24-19-generic/kernel/drivers/scsi/sr\_mod.ko；把它拷贝出来，并用命令objdump -d sr\_mod.ko 查看它的机器码，就可以知道它使用了哪些函数，利用这些函数名就可以结合source insight搜索出源码了。

## 载入模块符号

首先，在虚拟系统上装入目标模块foo，然后到/sys/module/foo/sections/下查看目标模块的section偏移地址信息。

实例

```
debian:/sys/module/smplesfs/sections# cat .text .data .bss
0xe01c7000
0xe01c864c
0xe01c8b20
```

然后，到真机的gdb下用add-symbol-file命令装载目标模块的符号信息 格式如下

```
add-symbol-file /path/to/module 0xe01c7000 \ # .text
-s .data 0xe01c864c \
-s .bss 0xe01c8b20
```

## 实例

```
(gdb) add-symbol-file test/day11/samplefs.ko 0xe01c7000 -s .data 0xe01c864c -s .bss 0xe01c8b20
add symbol table from file "test/day11/samplefs.ko" at
      .text_addr = 0xe01c7000
      .data_addr = 0xe01c864c
      .bss_addr = 0xe01c8b20
(y or n) y
Reading symbols from /storage/myqemu/new/linux-2.6.26/test/day11/samplefs.ko...done
(gdb)
```

<  >

然后，余下的对模块的调试就类似对内核的调试了。

## seq\_file.c的分析

## module.c的分析

## 中断处理过程

## s3c24xx内存初始化分析

[从这节开始，侧重于利用kgdb和source insight理解内核原理] [网上好像没这个内容。只看源码的话，因为source insight不能解析汇编源文件，在汇编源码中定位到初始化的源头好像很难，利用调试器很容易做到这点]

## 虚拟地址空间

## 用户层的观察窗

[待充实]

3G~4G虚拟地址空间的用途。（来自于qemu虚拟机的dmesg启动信息,500m物理内存）

```
<4>Zone PFN ranges:
<4>  DMA                0 ->      4096
<4>  Normal            4096 ->    127984
<4>  HighMem          127984 ->    127984
```

```
<6>virtual kernel memory layout:
```

```

<4>    fixmap   : 0xffff4c000 - 0xfffff000    ( 716 kB)
<4>    pkmap    : 0xff800000 - 0xffc00000    (4096 kB)
<4>    vmalloc   : 0xe0000000 - 0xff7fe000    ( 503 MB)
<4>    lowmem    : 0xc0000000 - 0xdf3f0000    ( 499 MB)
<4>    .init     : 0xc037f000 - 0xc03bb000    ( 240 kB)
<4>    .data     : 0xc02c0875 - 0xc03773ac    ( 730 kB)
<4>    .text     : 0xc0100000 - 0xc02c0875    (1794 kB)

```

3G~4G虚拟地址空间的用途。（来自于qemu虚拟机的dmesg启动信息,897m物理内存）

<4>Zone PFN ranges:

```

<4>  DMA          0 ->      4096
<4>  Normal       4096 ->   229376
<4>  HighMem     229376 ->   229616

```

<6>virtual kernel memory layout:

```

<4>    fixmap   : 0xffff4c000 - 0xfffff000    ( 716 kB)
<4>    pkmap    : 0xff800000 - 0xffc00000    (4096 kB)
<4>    vmalloc   : 0xf8800000 - 0xff7fe000    ( 111 MB)
<4>    lowmem    : 0xc0000000 - 0xf8000000    ( 896 MB)
<4>    .init     : 0xc037f000 - 0xc03bb000    ( 240 kB)
<4>    .data     : 0xc02c0875 - 0xc03773ac    ( 730 kB)
<4>    .text     : 0xc0100000 - 0xc02c0875    (1794 kB)

```

3G~4G虚拟地址空间的用途。（来自真机的dmesg启动信息,3G物理内存）

[ 0.000000] Zone PFN ranges:

```

[ 0.000000]  DMA          0 ->      4096
[ 0.000000]  Normal       4096 ->   229376
[ 0.000000]  HighMem     229376 ->   786416

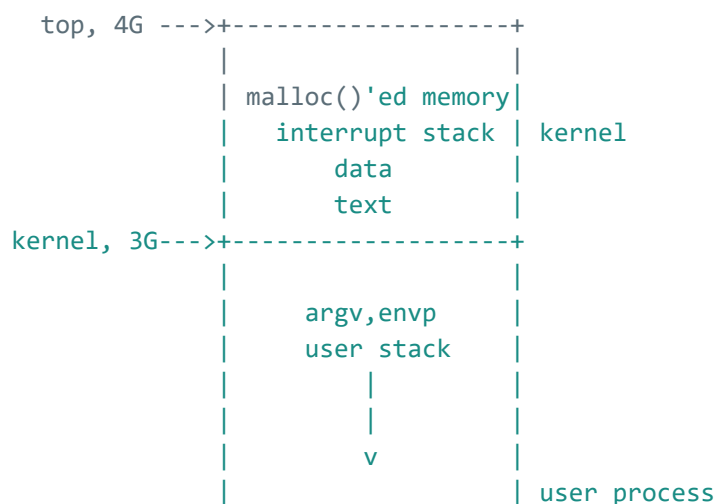
```

[ 33.262853] virtual kernel memory layout:

```

[ 33.262854]    fixmap   : 0xffff4b000 - 0xfffff000    ( 720 kB)
[ 33.262855]    pkmap    : 0xff800000 - 0xffc00000    (4096 kB)
[ 33.262856]    vmalloc   : 0xf8800000 - 0xff7fe000    ( 111 MB)
[ 33.262857]    lowmem    : 0xc0000000 - 0xf8000000    ( 896 MB)
[ 33.262858]    .init     : 0xc0421000 - 0xc047d000    ( 368 kB)
[ 33.262859]    .data     : 0xc03204c4 - 0xc041bdc4    (1006 kB)
[ 33.262861]    .text     : 0xc0100000 - 0xc03204c4    (2177 kB)

```





Layout of virtual address space

我们验证一下用户空间的内容(上图的下部分) [未完, 待续] 引用于<http://linux.chinaunix.net/bbs/viewthread.php?tid=978491>

查看进程的虚拟地址空间是如何使用的。

该文件有6列, 分别为:

地址: 库在进程里地址范围

权限: 虚拟内存的权限, r=读, w=写, x=, s=共享, p=私有;

偏移量: 库在进程里地址范围

设备: 映像文件的主设备号和次设备号;

节点: 映像文件的节点号;

路径: 映像文件的路径

每项都与一个vm\_area\_struct结构成员对应,

-----

```
struct vm_area_struct {
    struct mm_struct * vm_mm;           /* The address space we belong to. */
    unsigned long vm_start;             /* Our start address within vm_mm. */
    unsigned long vm_end;               /* The first byte after our end address
                                         within vm_mm. */

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;              /* Access permissions of this VMA. */
    unsigned long vm_flags;             /* Flags, listed below. */

    struct rb_node vm_rb;

    /*
     * For areas with an address space and backing store,
     * linkage into the address_space->i_mmap prio tree, or
     * linkage to the list of like vmas hanging off its node, or
     * linkage of vma in the address_space->i_mmap_nonlinear list.
     */
    union {
        struct {
            struct list_head list;
            void *parent; /* aligns with prio_tree_node parent */
            struct vm_area_struct *head;
        } vm_set;

        struct raw_prio_tree_node prio_tree_node;
    } shared;
};
```



```

/*
 * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
 * list, after a COW of one of the file pages. A MAP_SHARED vma
 * can only be in the i_mmap tree. An anonymous MAP_PRIVATE, stack
 * or brk vma (with NULL file) can only be in an anon_vma list.
 */
struct list_head anon_vma_node; /* Serialized by anon_vma->lock */
struct anon_vma *anon_vma;      /* Serialized by page_table_lock */

/* Function pointers to deal with this struct. */
struct vm_operations_struct * vm_ops;

/* Information about our backing store: */
unsigned long vm_pgoff;          /* Offset (within vm_file) in PAGE_SIZE
                                units, *not* PAGE_CACHE_SIZE */
struct file * vm_file;          /* File we map to (can be NULL). */
void * vm_private_data;         /* was vm_pte (shared mem) */
unsigned long vm_truncate_count; /* truncate_count or restart_addr */

#ifdef CONFIG_MMU
    atomic_t vm_usage;          /* refcount (VMAs shared if !MMU) */
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *vm_policy; /* NUMA policy for the VMA */
#endif

```

[todo 换个简单的程序]

```
$ ps -aux | grep firefox
```

Warning: bad ps **syntax**, perhaps a bogus '-'? See <http://procps.sf.net/faq.html>

```
fqh      8230  4.7  2.5 205872 80024 ?        Tl   14:54   0:19 /usr/lib/firefox
fqh      8313  0.0  0.0  3220   764 pts/1    R+   15:01   0:00 grep firefox
```

```
(gdb) attach 8230
```

```
...
```

```
.....
```

```
Loaded symbols for /usr/lib/libflashsupport.so
```

```
Reading symbols from /usr/lib/libpulse.so.0...(no debugging symbols found)...done
```

```
Loaded symbols for /usr/lib/libpulse.so.0
```

```
Reading symbols from /lib/libcap.so.1...(no debugging symbols found)...done.
```

```
Loaded symbols for /lib/libcap.so.1
```

```
(no debugging symbols found)
```

```
0xb7f24410 in __kernel_vsyscall ()
```

```
(gdb) bt
```

```
#0 0xb7f24410 in __kernel_vsyscall ()
```

```
#1 0xb7d46c07 in poll () from /lib/tls/i686/cmov/libc.so.6
```

```
#2 0xb6b4e1c6 in ?? () from /usr/lib/libglib-2.0.so.0
```

```
#3 0xb6b4e74e in g_main_context_iteration () from /usr/lib/libglib-2.0.so.0
```

```
#4 0xb77ba87c in ?? () from /usr/lib/xulrunner-1.9.0.1/libxul.so
```

```
#5 0xb77cf624 in ?? () from /usr/lib/xulrunner-1.9.0.1/libxul.so
```

```
#6 0xb77cf36f in ?? () from /usr/lib/xulrunner-1.9.0.1/libxul.so
```

```

#7  0xb787ecd6 in ?? () from /usr/lib/xulrunner-1.9.0.1/libxul.so
#8  0xb784e31f in ?? () from /usr/lib/xulrunner-1.9.0.1/libxul.so
#9  0xb77cf75e in ?? () from /usr/lib/xulrunner-1.9.0.1/libxul.so
#10 0xb765f122 in ?? () from /usr/lib/xulrunner-1.9.0.1/libxul.so
#11 0xb70b3a88 in XRE_main () from /usr/lib/xulrunner-1.9.0.1/libxul.so
#12 0x08049033 in ?? ()
#13 0xb7c90450 in __libc_start_main () from /lib/tls/i686/cmov/libc.so.6
#14 0x08048cc1 in ?? ()
(gdb)

```

```

$ cat /proc/8230/maps
08048000-0804f000 r-xp 00000000 08:01 7022914 /usr/lib/firefox-3.0.1/firefox <
0804f000-08050000 rw-p 00006000 08:01 7022914 /usr/lib/firefox-3.0.1/firefox <
08050000-0abd4000 rw-p 08050000 00:00 0 [heap] <
ae060000-ae063000 r-xp 00000000 08:01 6941098 /usr/lib/libflashsupport.so <
ae063000-ae064000 rw-p 00002000 08:01 6941098 /usr/lib/libflashsupport.so <
.....
..
b7f20000-b7f21000 rw-p 00001000 08:01 6942869 /usr/lib/libplds4.so.0d
b7f21000-b7f22000 r--p 00000000 08:01 6966184 /usr/lib/locale/zh_CN.utf8/LC_IDENTIFI
b7f22000-b7f24000 rw-p b7f22000 00:00 0
b7f24000-b7f25000 r-xp b7f24000 00:00 0 [vdso]
b7f25000-b7f3f000 r-xp 00000000 08:01 2326545 /lib/ld-2.7.so
b7f3f000-b7f41000 rw-p 00019000 08:01 2326545 /lib/ld-2.7.so
bfbcd000-bfc0a000 rw-p bffc3000 00:00 0 [stack] <
[
$
< >

```

## 交互，从内核层分析

[扩展]

## 理解设备模型

[结合source insight分析一个内核子系统的原理。源码分析工具虽好，但却是个死的东西，不能实时观察数据的生成和变化。如果在内核运行的时候，搭配调试器来分析，这个过程一定很形象和有趣]

## 面向对象的实现

## 设备模型的分层

## 外围支持机制

sysfs

hotplug

## 文件系统

参考书籍：

UNIX Filesystems Evolution, Design, and Implementation.pdf :

<http://www.google.cn/search?q=UNIX+Filesystems+Evolution%2C+Design%2C+and+Implementation&ie=utf-8&oe=utf-8&aq=t&rls=com.ubuntu:zh-CN:unofficial&client=firefox-a>

站点：

Ext4 (and Ext2/Ext3) Wiki: [http://ext4.wiki.kernel.org/index.php/Main\\_Page](http://ext4.wiki.kernel.org/index.php/Main_Page)

Ext4 Development project: <http://www.bullopensource.org/ext4/>

ext2-devel maillist archive: <http://sourceforge.net/mailarchive/forum.php?forum=ext2-devel>

参考文章：

“Linux Filesystems in 21 days 45 minutes” <http://us1.samba.org/samba/ftp/cifs-cvs/ols2006-fs-tutorial-smf.pdf>

## \*\*\*第三部分：其他工具\*\*\*

### strace

- 作用: strace能拦截和记录应用程序发起的系统调用和它收到的信号。主要用于观察应用层和内核层的交互。
- 命令选项: 查看, `$strace -help` 或 `$man strace` 或 `$info strace`
- 实例

### ltrace

- 作用: ltrace用于监控程序发起的库函数调用以及程序收到的信号。

## SystemTap

- 动态收集Linux内核信息和性能数据
- 官方 <http://sourceware.org/systemtap/>
- 参考文章

<http://www.ibm.com/developerworks/cn/linux/l-cn-systemtap3/index.html>

<http://www.ibm.com/developerworks/cn/linux/l-systemtap/index.html>

<http://sourceware.org/systemtap/tutorial/>

<http://sourceware.org/systemtap/wiki>

ubuntu下的配置安装: <http://sourceware.org/systemtap/wiki/SystemtapOnUbuntu>

## MEMWATCH

- 作用: 跟踪程序中的内存泄漏和错误

## YAMD

- 作用: 查找 C 和 C++ 中动态的、与内存分配有关的问题

## Magic SysRq

内核文档 sysrq.txt

<http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/sysrq.txt;h=10a0263ebb3f01e832c7827cc75d7fe54b341a6f;hb=f8d56f1771e4867acc461146764b4feeb5245669>

linux内核测试指南 相关章节

<http://wiki.zh-kernel.org/#%E6%96%87%E7%AB%A0>

## 附录：社区交流相关

## 补丁提交相关文档

如何参与 Linux 内核开发

<http://wiki.zh-kernel.org/doc/howto>

Linux内核代码风格

<http://wiki.zh-kernel.org/doc/codingstyle>

Linux内核开发邮件客户端资料

<http://wiki.zh-kernel.org/doc/email-clients.txt>

Linux内核补丁提交注意事项

<http://wiki.zh-kernel.org/doc/linux%E5%86%85%E6%A0%B8%E8%A1%A5%E4%B8%81%E6%8F%90%E4%BA%A4%E6%B3%A8%E6%84%8F%E4%BA%8B%E9%A1%B9>

基于git的Gentoo中文文档开发流程

[http://www.gentoo-cn.org/doc/zh\\_cn/git-howto.xml](http://www.gentoo-cn.org/doc/zh_cn/git-howto.xml)

mutt配置使用

<http://hi.baidu.com/springtty/blog/item/e6b25ddbb52f51ddb7fd4805.html>

<http://www.kongove.cn/blog/?p=149>

<http://www.kongove.cn/blog/?p=229>

<http://www.kongove.cn/blog/?p=225>

## 补丁制作与提交示范

说明：对于初学者，又是发单个补丁的话，本人推荐使用邮件客户端claws。它的优点是有线索成组的功能和草稿上有字数的标尺。本人感觉claws比 Sylpheed要快上几十倍。claws本来是Sylpheed的实验版，后来独立出来了。内核社区中使用的邮件

客户端大多数是mutt（并不是专指发补丁）。发补丁的所用工具也多种多样，有用各种邮件客户端的(mutt,claws,kmail...)，有使用git-send-email的，还有使用quilt的，真是打开眼界。

对于本人，发补丁用的工具目前在内核社区中是独一无二的，因为该工具是我自己写的：) 凡是用它发出的邮件，在邮件头会有个标志，比如：User-Agent: sniper-patch-carrier/1.06。无论单个还是多个补丁，都用它来发送，它的特点是够用简单又强大。源码在下小节中。

但下面的例子，还是使用claws发补丁的举例。

## 补丁的任务

mm/oom\_kill.c:badness() 函数

```
/**
 * badness - calculate a numeric value for how bad this task has been
 * @p: task struct of which task we should calculate
 * @uptime: current uptime in seconds
 * @mem: target memory controller//<-Li Zefan大侠上次提交了一个补丁，去掉了badness()的
 *                                     但是忘了删除该参数的说明了。现在的任务是提交补丁把它
 *
 * ...省略
 */
```

```
unsigned long badness(struct task_struct *p, unsigned long uptime)
{
```

<  >

## 过程

### 1. 进入git树

```
XXX@ubuntu:~$ cd /storage/linus-git/linux-2.6/
XXX@ubuntu:/storage/linus-git/linux-2.6$
```

### 2. 更新git树

```
XXX@ubuntu:/storage/linus-git/linux-2.6$ git-pull
Already up-to-date.
```

### 3. 修改目标源码

```
XXX@ubuntu:/storage/linus-git/linux-2.6$ vi mm/oom_kill.c
删除掉那个参数说明后结束vi,返回到shell下
```

### 4. 制作补丁

```
XXX@ubuntu:/storage/linus-git/linux-2.6$ git-diff > ../oom_kill.patch
```

## 5. 还原git树

```
XXX@ubuntu:/storage/linus-git/linux-2.6$ patch -p1 < ../oom_kill.patch -R
patching file mm/oom_kill.c
```

还原的方法或者采用下面方式

```
XXX@ubuntu:/storage/linus-git/linux-2.6$ git-gui
界面出来后，点击Branch->reset
```

## 6. 点击compose新建邮件

### 7. 点击insert file 载入补丁文件

```
diff --git a/mm/oom_kill.c b/mm/oom_kill.c
index 64e5b4b..460f90e 100644
--- a/mm/oom_kill.c
+++ b/mm/oom_kill.c
@@ -38,7 +38,6 @@ static DEFINE_SPINLOCK(zone_scan_mutex);
 * badness - calculate a numeric value for how bad this task has been
 * @p: task struct of which task we should calculate
 * @uptime: current uptime in seconds
- * @mem: target memory controller
 *
 * The formula used is relatively simple and documented inline in the
 * function. The main rationale is that we want to select a good task
```

## 8. 补全其他信息，比如 标题，to，cc等，还有信件内容

本例是：

标题起为：[PATCH]mm/oom\_kill.c: cleanup kerneldoc of badness()  
//为讨好Randy.Dunlap，特意写了字眼kerneldoc，因为他是主管内核文档的

```
to "Randy.Dunlap" <rdunlap@xenotime.net>
//收件人是谁得根据补丁的性质查看内核源码中的MAINTAINERS文件，
//难以确认是谁时，可以到linus-git的web-git下参看你修改文件的历史记录，看别人是发给谁的
cc linux-kernel@vger.kernel.org, //这个一定要有
linux-mm@kvack.org //修改文件所在的子系统的邮件列表，当收件人写错时，
//子系统的头目们可能会注意到和接受你的补丁
```

邮件本身处理后变成下面的格式

Paramter @mem has been removed since v2.6.26, now delete its comment. //补丁的作用

```
Signed-off-by: your-name <your-address@gmail.com> //你的签收
--- //三个'- '，表示下面的内容是补丁了。应用补丁的工具会根据这个标志提取补丁
diff --git a/mm/oom_kill.c b/mm/oom_kill.c
index 64e5b4b..460f90e 100644
--- a/mm/oom_kill.c
+++ b/mm/oom_kill.c
@@ -38,7 +38,6 @@ static DEFINE_SPINLOCK(zone_scan_mutex);
 * badness - calculate a numeric value for how bad this task has been
 * @p: task struct of which task we should calculate
 * @uptime: current uptime in seconds
- * @mem: target memory controller
 *
 * The formula used is relatively simple and documented inline in the
 * function. The main rationale is that we want to select a good task
```

## 9. 然后发信

## 10. 等待回复

Randy Dunlap果然勤快，两个小时不到就收到了他的信件

Acked-by: Randy Dunlap <rdunlap@xenotime.net>

Thanks.

说明：

Acked-by表示他认为补丁正确，但并不自己接收。维护者回复applied才算是接受了。

当然如果得到比较有威望的人acked-by，被接受的可能性就大大的提高了。--Li Yang大牛

看来，我把补丁的维护人搞错了，因为Randy Dunlap并没领我的情：(

注意，这个不能急。有的补丁或许是因为太微小，对方都没回复你，其实他已经收录了你的补丁。到子系统树向linus的git树合并时就会看到你的补丁(通常会在LKML中有个集体通告，说明这批补丁中包含了哪些内容。)。等一个多星期无妨。

后记：过了三四天后，收到了一封信如下。这是mm树的邮件系统发来的。可见Randy Dunlap把这个补丁提交给mm树了。在mm树经过验证补丁正确后就会再汇合到linus的主线树中。这是补丁接受的一种方式。当然，我遇到的情况有，子系统维护人回复你applied to xx(树)，然后该负责人就要求linus merge他的树，这样就收不到mm树的通知信。甚至有时子系统维护人接受补丁了都不吭一声，然后补丁又是直接merge到主线树中。补丁接受的流程大概就这样了。

From: akpm@linux-foundation.org

To: mm-commits@vger.kernel.org

Cc: qhfeng.kernel@gmail.com, rdunlap@xenotime.net

Subject: + mm-oom\_killc-fix-badness-kerneldoc.patch added to -mm tree

Date: Thu, 30 Oct 2008 14:47:53 -0700

The patch titled

mm/oom\_kill.c: fix badness() kerneldoc

has been added to the -mm tree. Its filename is

mm-oom\_killc-fix-badness-kerneldoc.patch

....省略

The current -mm tree may be found at <http://userweb.kernel.org/~akpm/mmotm/>

-----  
Subject: mm/oom\_kill.c: fix badness() kerneldoc

From: Qinghuang Feng <qhfeng.kernel@gmail.com>

Paramter @mem has been removed since v2.6.26, now delete its comment.

Signed-off-by: Qinghuang Feng <qhfeng.kernel@gmail.com>

Acked-by: Randy Dunlap <rdunlap@xenotime.net>

Signed-off-by: Andrew Morton <akpm@linux-foundation.org>

---

...补丁内容省略

—

Patches currently in -mm which might be from qhfeng.kernel@gmail.com are



```
origin.patch
mm-oom_killc-fix-badness-kernel-doc.patch
linux-next.patch
```

## 多补丁发送工具

有很多此类工具，其中功能最强的是git-send-email.但本人还是自己动手丰衣足食，参照其他工具写了一个。perl学得不久，很没perl的味道. 这个工具的优点是用perl自带的邮件引擎通过smtp.gmail.com（目前只支持gmail，其他没实验过）发送邮件，而不是通过 sendmail,mutt等第三方工具发送，省去了那些工具麻烦的设置过程。当然，最主要还是因为，自己写的东西可以随时改进，好玩。

### 使用

把下面的主程序复制到一个文本文件中，并把它命名为:sniperpatchcarrier.pl. 并写好另一个控制文件control。发送补丁时使用下面命令即可

```
perl sniperpatchcarrier.pl control
```

### 控制文件的内容如下

```
SMTP: 你的SMTP服务器地址
USER: 你所用SMTP对应邮箱的用户名
PWD: 你所用SMTP对应邮箱的用户密码
From: 你的名字 <你的邮件地址>
To: 目的地 <目标邮箱地址>
Cc: 其他人 <邮件地址>
Cc: 其他人 <邮件地址>
Subject: [PATCH 1/n] 第一个补丁的标题
File: 第一个补丁文件的路径
Subject: [PATCH n/n] 第n个补丁的标题
File: 第n个补丁文件的路径
```

说明,第一项（比如SMTP等)大小写都可，项目之间（比如To:和邮件地址）有无空格均可。邮件地址的用户名可有可无，没写明用户名时<> 符号可有可无。但为防止意外，还是建议照着下面的模版修改使用。注意，目前只支持gmail。要想支持其他邮箱，简单修改源码即可。

```
SMTP: smtp.gmail.com
USER: your-account@gmail.com
PWD: your-password
From: your-naem <your-account@gmail.com>
```

```
To: linux-kernel@vger.nonexist.org
Cc: Randy Dunlap <rdunlap@nonexit.net>
Cc: Jeff Garzik <jeff@nonexist.org>
Cc: linux-ide@vger.nonexist.org
Bcc:
Subject: [PATCH 1/2] drivers/ata/pata_radisys.c: cleanup kernel-doc
File: radisys_set_dmamode.patch
Subject: [PATCH 2/2] drivers/ata/libata-core.c: cleanup kernel-doc
File: ata_qc_new_init.patch
```

## 主程序

```
#!/usr/bin/perl

use Email::Send;
use Email::Simple::Creator; # or other Email::

sub usage
{
    print <<EOT;
Usage: sniperpatchcarrier controlfile

Sample control file:
# this is a comment
SMTP: smtp.gmail.com
USER: your-smtp-account-username
PWD: your-smtp-account-password
From: Joe Blow <jb\@example.com>
To: Harry Hacker<hhacker\@another.example.com>
Cc: Lurker One <lurker1\@yet_another.example.com>
Cc: lurker2\@a_third.example.com
Bcc: blindrecipient\@secret_dropoff.example.com
Subject: [PATCH 1/2] Short sweet descriptive phrase
File: path-of-message-text-for-first-patch
Subject: [PATCH 2/2] Another short sweet phrase
File: path-of-message-text-for-second-patch
```

Above sends out two email messages, with specified Subject lines, and contents from corresponding Files.

Each "File" line sends a message, using the latest values for the other keywords set so far in file. The To, Cc and Bcc lists accumulate.

First word on each line of control file is a keyword. It can be any mix of upper/lower case, with optional trailing colon. The keyword "Subject" can be abbreviated as "Subj".

Try testing by first sending patch set only to one or more of your own email addresses.

The following documents explain how to submit patches to the

Linux kernel:

- 1) Documentation/SubmittingPatches, a file in the kernel source  
<http://lxr.linux.no/source/Documentation/SubmittingPatches>
- 2) Documentation/CodingStyle  
<http://lxr.linux.no/source/Documentation/CodingStyle>
- 3) Andrew Morton's "The Perfect Patch", available at:  
<http://www.zip.com.au/~akpm/linux/patches/stuff/tpp.txt>
- 4) Jeff Garzik's "Linux kernel patch submission format", at:  
<http://linux.yyz.us/patch-format.html>
- 5) Greg Kroah-Hartman's "How to piss off a kernel subsystem maintainer"  
<http://www.kroah.com/log/2005/03/31/>
- 6) Linus's email on the canonical patch format:  
<http://lkml.org/lkml/2005/4/7/183>

Linus describes the canonical patch format:

That canonical format is:

Subject: [PATCH 001/123] [<area>:] <explanation>

together with the first line of the body being a

From: Original Author <origa@email.com>

followed by an empty line and then the body of the explanation.

After the body of the explanation comes the "Signed-off-by:" lines, and then a simple "---" line, and below that comes the diffstat of the patch and then the patch itself.

EOT

```
    exit(1);
}

$VERSION = "1.06";

$smtpserver="";
$smtpuser="";
$smtppwd="";

$fromaddr="";
@subjects = (); #note: don't to use ""!
@patches = ();
$toaddrs = "";
$ccaddrs = "";
$bccaddrs = "";
$refid = "";

$patch_count = 0;
$patch_content = "";

$control_file = "";
```

```

sub parse_control
{
    #0. verify the integrity of control file
    if (open(INTEGRITY, $control_file)) {
        my $integrity = 1;
        my %integrity_panic = (
            "smtp" => "smtp:",
            "user" => "user:",
            "pwd" => "pwd:",
            "subj" => "subject:",
            "file" => "file:",
        );
        my $ctrl_file;
        print "\n>checking the integrity of control file...\n\n";
        while (<INTEGRITY>) {
            $ctrl_file .= $_;
        }

        unless ($ctrl_file =~ /smtp\s*\:/i) {
            $integrity = 0;
        } else {$integrity_panic{"smtp"} = 0;}

        unless ($ctrl_file =~ /user\s*\:/i) {
            $integrity = 0;
        } else {$integrity_panic{"user"} = 0;}

        unless ($ctrl_file =~ /pwd\s*\:/i) {
            $integrity = 0;
        } else {$integrity_panic{"pwd"} = 0;}

        unless ($ctrl_file =~ /subj(?:ect)?\s*\:/i) {
            $integrity = 0;
        } else {$integrity_panic{"subj"} = 0;}

        unless ($ctrl_file =~ /file\s*\:/i) {
            $integrity = 0;
        } else {$integrity_panic{"file"} = 0;}

        if ($integrity == 0) {
            print "ERROR: control file is not integrate. \n";
            print "You must add the following item(s) to it!\n\n";
            my $key;
            my $value;
            while (($key, $value) = each %integrity_panic) {
                #print $key.">".$value."\n";
                if ($value){
                    print $value."\n";
                }
            }
            exit(1);
        } else {print "ok!\n";}

        close(INTEGRITY);
    }
}

```

```

#<-open
} else {
    print "can't open control file!\n";
    exit(1);
}

#1. parse control file
if (open(CTRLFILE, $control_file)) {
    print "\n>reading the control file\n";
    while (<CTRLFILE>) {
        print;
        #FIXME: why the last \s* doesn't work as what I want?
        #so I fix it in the following. see stuff under close(CTR
        if (m{smtp\s*:\s*(.*)\s*}i) {
            $smtpserver = $1;
        } elsif (m{user\s*:\s*(.*)\s*}i) {
            $smtpuser = $1;
        } elsif (m{pwd\s*:\s*(.*)\s*}i) {
            $smtppwd = $1;
        } elsif (m{from\s*:\s*(.*)\s*}i) {
            $fromaddr = $1 ;
        } elsif (m{to\s*:\s*(.*)\s*}i) {
            $toaddrs .= $1.", ";
        } elsif (m{bcc\b\s*:\s*(.*)\s*}i) {
            $ccaddrs .= $1.", ";
        } elsif (m{bcc\s*:\s*(.*)\s*}i) {
            $bccaddrs .= $1.", ";
        } elsif (m{subj(?:ect)?\s*:\s*(.*)\s*}i) {
            push @subjects, $1;
        } elsif (m{file\s*:\s*(.*)\s*}i) {
            push @patches, $1;
        }
    }

    close(CTRLFILE);

    print "\nafter fixed\n";
    #now we fix anything MAY introduce the bug. Pretty strictly!
    print "\n>smtp config:\n";
    $smtpserver =~ s/\s+$///;
    print $smtpserver."\n";
    $smtpuser =~ s/\s+$///;
    print $smtpuser."\n";
    $smtppwd =~ s/\s+$///;
    print $smtppwd."\n";

    print "\n>mail address:\n";
    #we assume that "," is invalid for a mail-address itself.
    $fromaddr =~ s/(\s*\,\s*)$//;
    print "from:". $fromaddr. "\n";
    $toaddrs =~ s/(\s+,\,)/\,/g;      #for user input
    $toaddrs =~ s/(\,\s*)$//;        #for action of srcipt itself
    print "to:". $toaddrs. "\n";

```

```

$ccaddrs =~ s/\s+\\,\\/g;
$ccaddrs =~ s/(\\,\\s*)$//;
print "cc:". $ccaddrs. "\n";
$bccaddrs =~ s/(\\s+\\,)/\\,\\/g;
$bccaddrs =~ s/(\\,\\s*)$//;
print "bcc:". $bccaddrs. "\n";

#subjects don't need to fix.
#defer the fix of patch file name to #2

print "\nall mail address is ok?\nIf NO problem, press \"c\" to continue\n";
chomp (my $answer = <STDIN>);
unless ($answer =~ /^c$/) {
    print "aborted by user.\n";
    exit(1);
}

#2 check configuration
print ">>checking the relation between subject and patch file...\n\n";
if (scalar(@subjects) == scalar(@patches)) {
    $patch_count = scalar(@subjects);
    my $check_i;
    for ($check_i = 0; $check_i < $patch_count; $check_i++) {
        #the deferred fix
        $patches[$check_i] =~ s/\s+$//;
        my $show_i = $check_i + 1;
        print "$show_i:". $subjects[$check_i]. "\n";
        print "$show_i:". $patches[$check_i]. "\n\n";
    }
    print "All relations are ok?\nIf NO problem, press \"c\" to continue\n";
    chomp (my $answer = <STDIN>);
    unless ($answer =~ /^c$/) {
        print "aborted by user.\n";
        exit(1);
    }
} else {
    print "<< ERROR! the count of subjects isn't equal to the patches\n";
    exit(1);
}

#//<-open
} else {
    print "can't open control file!\n";
    exit(1);
}

#3 check the patches format
my $patch_i = undef;
$check_patch_global_success = 1;
print "\n>>checking patch format...\n\n";
for ($patch_i = 0; $patch_i < $patch_count; $patch_i++) {
    my $show_patch_i = $patch_i + 1;
    print "patch $show_patch_i: ";

```

```

#patch 0 is not the real patch, don't check it.
if ($subjects[$patch_i] =~ /\[s*PATCH\s*0+\\//i) {
    print "found [PATCH 0/*], ignore it...\n\n";
    next;
}

if (open(PATCH_CHECK, $patches[$patch_i])) {

    my $success = 1;
    $line = undef;
    while (<PATCH_CHECK>) {
        $line .= $_;
    }
    my $sign_match = undef;
    if ($line =~ m{(\bSigned-off-by:\s+(?:.+?)\s+<(?:.+?)@.
        $sign_match = $1;
        #print $sign_match;
    } else {
        $success = 0;
        print "<< ERROR: no sign or sign format error.\n";
        print "Please obey the following example strictly";
        print "Signed-off-by: your-name <your-mail@gmail.com>\n";
    }

    if ($success == 1){
        unless ($line =~ m{\b$sign_match\-\-\-\s*\n}) {
            $success = 0;
            print "<< ERROR: no patch separator \"--";
            print "please add \"---\" in a new line ";
        }
    }

    if ($success == 1) {
        print "ok\n";
    } else {
        print "ERROR\n";
        $check_patch_global_success = 0;
    }

    print "\n";
    close(PATCH_CHECK);
} else {
    print "<< FATAL error: can't open \"$patches[$patch_i]\"";
    exit(1);
}

#<-for
}

#TODO: add a summarisation of bad patch?
if ($check_patch_global_success == 0) {
    print "Abort, please check your patch format.\n";
    print "The following is a good example:\n\n";

```

```
        print <<GOOD_PATCH;
Paramter \@mem has been removed since v2.6.26, now delete its comment.
```

```
Signed-off-by: your-name <your-name@gmail.com>
```

```
---
```

```
diff --git a/mm/oom_kill.c b/mm/oom_kill.c
```

```
index 64e5b4b..460f90e 100644
```

```
--- a/mm/oom_kill.c
```

```
+++ b/mm/oom_kill.c
```

```
...(ignore the diff content)
```

```
GOOD_PATCH
```

```
        print "\n";
        exit(1);
```

```
        #<-if ($check_patch...
    } else {
        print "ok, all patches format are right.\n\n"
    }
}
```

```
}
```

```
sub create_mailer
```

```
{
    $mailer = Email::Send->new( {
        mailer => 'SMTP::TLS',
        mailer_args => [
            Host => $smtpserver,
            Port => 587,
            User => $smtpuser,
            Password => $smtppwd,
            #Hello => 'fayland.org',
        ]
    } );
}
```

```
sub send_onemail
```

```
{
    my ($index) = @_ ;

    my $email = Email::Simple->create(
        header => [
            From    => $fromaddr,
            To      => $toaddrs,
            Cc      => $ccaddrs,
            Bcc     => $bccaddrs,
            Subject => $subjects[$index],
        ],
        body => $patch_content,
    );
    $email->header_set( 'User-Agent' => "sniper-patch-carrier/$VERSION" );

    print " sending email: $subjects[$index] ...\n";
}
```



```

        eval { $mailer->send($email) };
        if (!$@) {
            print ">>success!\n";
        } else { die ">>Error sending email: $@"}
    }
}

```

```

sub read_onemail
{
    my ($index) = @_ ;
    my $file = $patches[$index];
    print " reading file: $file ...\n";
    $patch_content = undef;
    if (open(FILE, $file)) {
        while (<FILE>) {
            $patch_content .= $_;
        }
        close(FILE);
    } else {
        print "can't open patch!";
        exit(1);
    }
}

```

```

sub send_mails
{
    print ">>now sending all patches...\n\n";
    print " file count: $patch_count\n";
    my $i ;
    for ($i = 0; $i < $patch_count; $i++) {
        my $show_i = $i + 1;
        print "\n$show_i:\n";
        read_onemail($i);
        send_onemail($i);
    }
    print "\nGood! over\n\n";
}

```

```

sub parse_argv
{
    unless ($control_file = shift @ARGV) {
        usage;
    }
}

```

```

parse_argv;
parse_control;
create_mailer;
send_mails;

```

## git使用

Git 中文教程

<http://www.linuxsir.org/main/doc/git/gittutorcn.htm>

git使用小结

<http://wangcong.org/blog/?p=307>

学习 Git

<http://www.zeuux.org/science/learning-git.cn.html>

## 附录：内核参考书籍文章

### 内核git库

内核git库：

<http://git.kernel.org/?p=linux/kernel/git>

linus-git

<http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary>

net

<http://git.kernel.org/?p=linux/kernel/git/davem/net-2.6.git;a=summary>

net-next

<http://git.kernel.org/?p=linux/kernel/git/davem/net-next-2.6.git;a=summary>

linux-next

<http://git.kernel.org/?p=linux/kernel/git/sfr/linux-next.git;a=summary>

免费git库<http://repo.or.cz/>

## 书籍

### 综合类：

“understanding the linux kernel”

”linux kernel development“

“linux源代码情景分析”

“Embedded.Linux.Primer.A.Practical.Real.World.Approach.”

“The\_Linux\_Kernel\_Primer\_A\_Top\_Down\_Approach\_For\_x86\_and\_PowerPC\_Architectures”

### 子系统类：

#### 文件系统：

“UNIX Filesystems Evolution, Design, and Implementation”

“File System Forensic Analysis”

“Windows NT File System Internals”

#### 内存管理：

“Understanding The Linux Virtual Memory Manager”

#### 网络系统：

“The Linux® Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel”

“Understanding.Linux.Network.Internals”

The Linux TCP/IP Stack: Networking for Embedded Systems (2.6.0-test10)

#### 网络协议

<http://zh.wikipedia.org/wiki/%E7%BD%91%E7%BB%9C%E4%BC%A0%E8%BE%93%E5%8D%8F%E8%AE%AE>

<http://www.ietf.org/>

通过编号查看 <http://www.ietf.org/rfc.html>

通过名称查询 <http://www.rfc-editor.org/rfcsearch.html>

Where and how to get new RFCs

```
to: rfc-info@isi.edu
subject getting rfcs
help:ways_to_get_rfcs
```

TCP/IP详解中文版

<http://220.113.41.171/cgi-bin/parker/search?String=TCP+IP+%E8%AF%A6%E8%A7%A3>

TCP/IP详解 所用源码

4.4BSD-Lite.tar.gz

<http://www.rcub.bg.ac.yu/~ggajic/pub/4.4BSD/>

大侠文章

[http://blog.chinaunix.net/u2/64681/article\\_86575.html](http://blog.chinaunix.net/u2/64681/article_86575.html)

网络编程

Unix Network Programming Vol 1

Unix Network Programming Vol 2

Network\_Programming\_With\_Perl

Linux Socket Programming By Example

网络教材

Computer Networks, Fourth Edition

驱动开发:

“linux device drivers”

“Essential.Linux.Device.Drivers”

源码本身及附带文档

其他操作系统的设计与实现:

The Design and Implementation of the 4.4BSD Operating System

The Design and Implementation of the FreeBSD Operating System

Solaris Internals Solaris 10 and OpenSolaris Kernel Architecture

## **子系统官方网站**

\* MM

<http://linux-mm.org/>

\* USB

<http://www.linux-usb.org/>

<http://www.usb.org/home>

\* MTD

<http://www.linux-mtd.infradead.org/>

\* ARM

<http://www.arm.linux.org.uk/>

\* uclinux

<http://www.uclinux.org/>

\* NET:

<http://www.linuxfoundation.org/en/Net>

wireless

<http://linuxwireless.org/>

IPsec

<http://www.ipsec-howto.org/>

netfilter

<http://netfilter.org/>

Linux Advanced Routing & Traffic Control

<http://lartc.org/>

Frame Diverter

<http://diverter.sourceforge.net/>

网络工具源码

iputils

```
$ apt-get source iputils
```

net-tools

```
$ apt-get source net-tools
```

Iproute2

<http://www.linuxfoundation.org/en/Net:Iproute2>

\* FS:

List of file systems

[http://en.wikipedia.org/wiki/List\\_of\\_file\\_systems](http://en.wikipedia.org/wiki/List_of_file_systems)

ext4

<http://sourceforge.net/mailarchive/forum.php?forum=ext2-devel>

<http://www.bullopensource.org/ext4/>

[http://ext4.wiki.kernel.org/index.php/Main\\_Page](http://ext4.wiki.kernel.org/index.php/Main_Page)

btrfs

[http://btrfs.wiki.kernel.org/index.php/Project\\_ideas](http://btrfs.wiki.kernel.org/index.php/Project_ideas)

coda

<http://www.coda.cs.cmu.edu/>

nfs

[http://wiki.linux-nfs.org/wiki/index.php/Main\\_Page](http://wiki.linux-nfs.org/wiki/index.php/Main_Page)

yaffs

<http://www.yaffs.net/>

jffs

<http://sourceware.org/jffs2/jffs2-html/>

logfs

<http://www.logfs.org/logfs/>

ubifs

<http://www.linux-mtd.infradead.org/doc/ubifs.html>

\* 其他:

U-Boot

<http://www.denx.de/wiki/U-Boot>

<http://sourceforge.net/projects/u-boot/>

udev

<http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html>

busybox

<http://www.busybox.net/>

skyeye

<http://www.skyeye.org/index.shtml>

## 参考文章

IBM-Linux 相关专题 <http://www.ibm.com/developerworks/cn/linux/> “Debugging Kernel Modules with User Mode Linux”

<http://www.linuxjournal.com/article/5749>

“Debugging Memory on Linux” <http://www.linuxjournal.com/article/4681>

“DDD—Data Display Debugger” <http://www.linuxjournal.com/article/2315>

“Linux 系统内核的调试” <http://www.ibm.com/developerworks/cn/linux/l-kdb/>

System Dump和Core Dump的区别 <http://hi.baidu.com/iruler/blog/item/c203de3522ff398ea61e122c.html>



<http://www.linuxjournal.com/user/800887/track>

<http://www.linuxjournal.com/> <http://www.ibm.com/developerworks/cn/linux/l-devm-apper/index.html>

read 系统调用剖析 <http://www.ibm.com/developerworks/cn/linux/l-cn-read/index.html>

[http://blog.chinaunix.net/u/4206/showart\\_501237.html](http://blog.chinaunix.net/u/4206/showart_501237.html)

<http://hi.baidu.com/linux%5Fkernel/blog/category/pci%C9%E8%B1%B8%C7%FD%B6%AF>

[http://wiki.jk2410.org/wiki/Main\\_Page](http://wiki.jk2410.org/wiki/Main_Page)

<http://www.ibm.com/developerworks/cn/linux/l-cn-clocks/index.html>

利用Vmware5.5.1 和 kgdb调试 x86平台的kernel

[http://blog.chinaunix.net/u/22617/showart\\_338509.html](http://blog.chinaunix.net/u/22617/showart_338509.html)

Welcome to Linux From Scratch

<http://www.linuxfromscratch.org/>

Unreliable Guide To Locking

<http://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/index.html>

How do I printk <type> correctly?

<http://lkml.org/lkml/2008/10/23/132>

<http://www.ibm.com/developerworks/cn/linux/l-linux-networking-stack/>

KernelJanitors/TODO

<http://kernelnewbies.org/KernelJanitors/TODO>

sparse主页

<http://www.kernel.org/pub/linux/kernel/people/josh/sparse/>

<http://linux.bkbits.net:8080/linux-2.6/search/?PAGE=search&EXPR=sparse&SEARCH=ChangeSet+comments>

Coccinelle - a Framework for Linux Device Driver Evolution

<http://www.emn.fr/x-info/coccinelle/>

linux论文 <http://www.linuxsymposium.org>

[www.linuxsymposium.org/2006/linuxsymposium\\_procv2.pdf](http://www.linuxsymposium.org/2006/linuxsymposium_procv2.pdf)

[www.linuxsymposium.org/2006/linuxsymposium\\_procv1.pdf](http://www.linuxsymposium.org/2006/linuxsymposium_procv1.pdf)

understanding the linux kernel 在线文档

<http://www.linux-security.cn/ebooks/ulk3-html/>

Data Structures and Algorithms with Object-Oriented Design Patterns in C++/Java/C#/Python/Ruby/Lua/Perl/PHP

<http://www.brpreiss.com/books/opus4/>

<ftp://ftp.akaedu.org/./1.html>

<ftp://ftp.freebsd.org/>

<http://bsd.org/>

<http://www.micrium.com/>

<http://v1.moblin.org/index.php>

<http://moblin.org/>

<http://www.linuxdriver.cn/>

# Integrating Flexible Support for Security Policies into the Linux Operating System

<http://www.nsa.gov/selinux/papers/slinux/slinux.html>

## 私人备忘

cpan设置

Going to `read /home/fqh/.cpan/sources/modules/02packages.details.txt.gz`

Warning: Your `/home/fqh/.cpan/sources/modules/02packages.details.txt.gz` does not contain the list of modules.  
是选取站点不可用造成的。

<http://tech.foolpig.com/2008/10/22/cpan-error-modulelist/>

1. 删除掉.cpan

2. `perl -MCPAN -e shell`

或1,2步骤换为`conf init`命令

3. 选了africa下的三个站点

4. cpan设置完后, `reload index`即可

5. 列举模块`m`

6. 查询 `d /模块/`

---

`__attribute__((context(x,0,1)))` means "you need not hold x before, but you will hold one more of x after".

`__attribute__((context(x,1,0)))` means "you must already hold x, and you will no longer hold x after".

`__attribute__((context(x,1,1)))` means "you must already hold x, and you will continue to hold x".

<

>

## sendpatchset 的地址

[http://google.com/codesearch?hl=en&q=show:UIY7Kd7jXdU:OMvU\\_Vh8FvE:EkVrWPVcX9w&sa=N&ct=rd&cs\\_p=http://www.speakeasy.net&cs\\_f=~pj99/cgi/sendpatchset](http://google.com/codesearch?hl=en&q=show:UIY7Kd7jXdU:OMvU_Vh8FvE:EkVrWPVcX9w&sa=N&ct=rd&cs_p=http://www.speakeasy.net&cs_f=~pj99/cgi/sendpatchset)

[http://pleac.sourceforge.net/pleac\\_perl/index.html](http://pleac.sourceforge.net/pleac_perl/index.html)

[http://search.cpan.org/~fayland/Email-Send-SMTP-TLS-0.02/lib/Email/Send/SMTP/TLS.pm#\\_top](http://search.cpan.org/~fayland/Email-Send-SMTP-TLS-0.02/lib/Email/Send/SMTP/TLS.pm#_top)

[http://www.61dh.com/blog/2008/10/perl\\_27.html](http://www.61dh.com/blog/2008/10/perl_27.html)

[http://blog.chinaunix.net/u2/77776/showart\\_1227451.html](http://blog.chinaunix.net/u2/77776/showart_1227451.html)

剑桥辞典 <http://dictionary.cambridge.org/>

<http://www.merriam-webster.com/>

<http://dictionary.reference.com/>

<http://www.thefreedictionary.com/>

[http://people.freebsd.org/~murray/bsd\\_flier.html](http://people.freebsd.org/~murray/bsd_flier.html)

<http://s3c24xx.wiki.zoho.com/>

<http://bobzhang.wiki.zoho.com/>

<http://code.google.com/p/root-kit/>

免费git库<http://repo.or.cz/>