Linux 内 核 学 习 起 步

albcamus <albcamus@gmail.com>

Last Update：2007年12月06日

1，编译内核

========================

1.0 获取内核压缩

在http://www.kernel.org/pub/linux/kernel/v2.6/能看到一系列的文件，如ChangeLog-2.6.xx、linux-2.6.xx.tar.{gz|bz2}和patch-2.6.xx.{gz|bz2}，你应该下载linux-2.6.23.tar.bz2这样的压缩包。

1.1 找一个旧的.config作参考

安装内核开发包(以FC8为例，是kernel-devel-2.6.23.1-42.fc8.i686.rpm)，则/lib/modules/<version>/build目录下会有.config文件。

```
# cd linux-2.6.24-rc3
# cp /lib/modules/2.6.23.1-42.fc8/build/.config .
```

1.2 配置、编译和安装

配置：

```
# make menuconfig/gconfig/xconfig/oldconfig/defconfig/allyesconfig/allmodconfig
```

FYI： 一般用menuconfig比较多，如果你喜欢GUI风格，也可以选择基于Qt的xconfig或

基于Gtk+的gconfig。

FYI：可以用O=</path/to/build>指定编译生成的文件放在哪个目录. 举例来说，
如果我的内核源代码目录是/usr/src/linux-2.6.23，编译内核时指定了：

```
make O=/home/arc/build/linux-2.6.23
```

那么/lib/modules/2.6.23下的`source'和`build'这两个符号链接就分别指向了源代码和目标代码目录：

```
# ls -l /lib/modules/2.6.23/source
lrwxrwxrwx 1 root root 21 11-22 13:35 /lib/modules/2.6.23/source -
> /usr/src/linux-2.6.23

# ls -l /lib/modules/2.6.23/build
lrwxrwxrwx 1 root root 28 11-22 13:35 /lib/modules/2.6.23/build ->
/home/arc/build/linux-2.6.23
```

编译：

```
# make
```

FYI：　V=1　　/*verbose模式，把每一部执行的命令都打印出来。
*一个小技巧就是，把标准输出重定向到一个文件中，
*这样以后查找模块之间的依赖关系就很方便了
*/

C=1　　/*需要安装sparse程序，进行严格的静态C语法检查。
*一般开发者会通过它来预防BUG
*/

安装：

```
# make modules_install
# make install
```

FYI：正常情况下make install会根据你机器的配置为你定制initrd文件，并更新grub.conf文件中的内容。但如果重新引导时发现无法启动，注意手工mkinitrd。
例如：

```
# mkinitrd -v –preload libata.ko –with=ext3 /boot/initrd-2.6.24-rc3.img
2.6.24-rc3
```

其中—preload指定的模块会在/etc/modprobe.conf之前加载，而--with指定的模块会在这之后加载。

1.3 文档

```
# make htmldocs     //你也可以不用htmldocs指令HTML格式，而指定pdfdocs或
                    //psdocs

# make mandocs          //为kernel API生成man手册
# make installmandocs       /*将kernel API的手册页安装到man程序能找到的
                             *目录中， 这样就可以man copy_from_user了
                             */
```

FYI：执行make htmldocs/pdfdocs/psdocs之后，在O=指定的目录(如果没使用O=则是源代码目录)的Documentation/DocBook/下，会生成几份很重要的文档：

kernel api ：内核开发的API手册
usb ：USB host端的API手册
gadget ：USB device端的API手册
kernel locking ：内核加锁的HOWTO文档
kernel hacking ：内核开发的一些注意事项

FYI：内核源代码目录的Documentation目录：

kernel-parameter.txt ：内核参数，加在一个grub entry的kernel指令后面
filesystems/vfs.txt：Linux虚拟文件系统的深入介绍
memory-barriers ：关于barriers的文档
CodingStyle ：内核编码的规范
等等.

图1.1 menuconfig

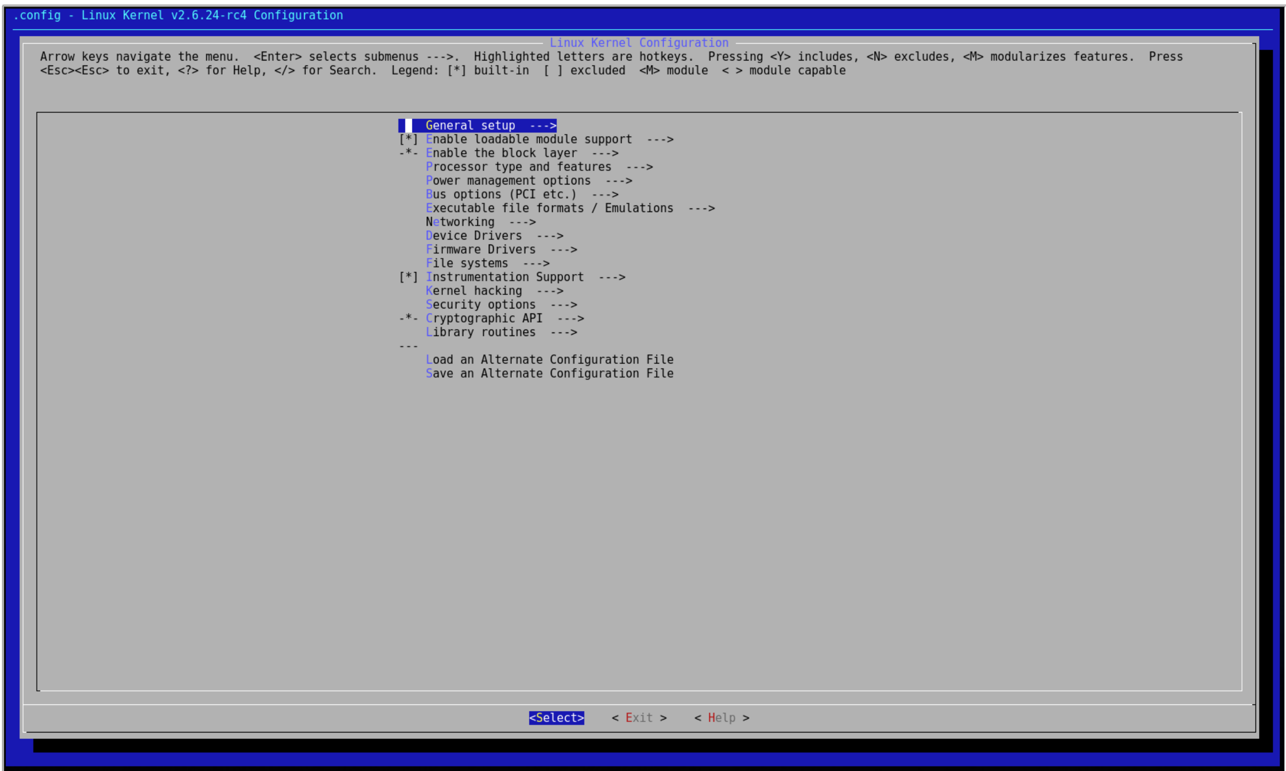


图 1.2 xconfig

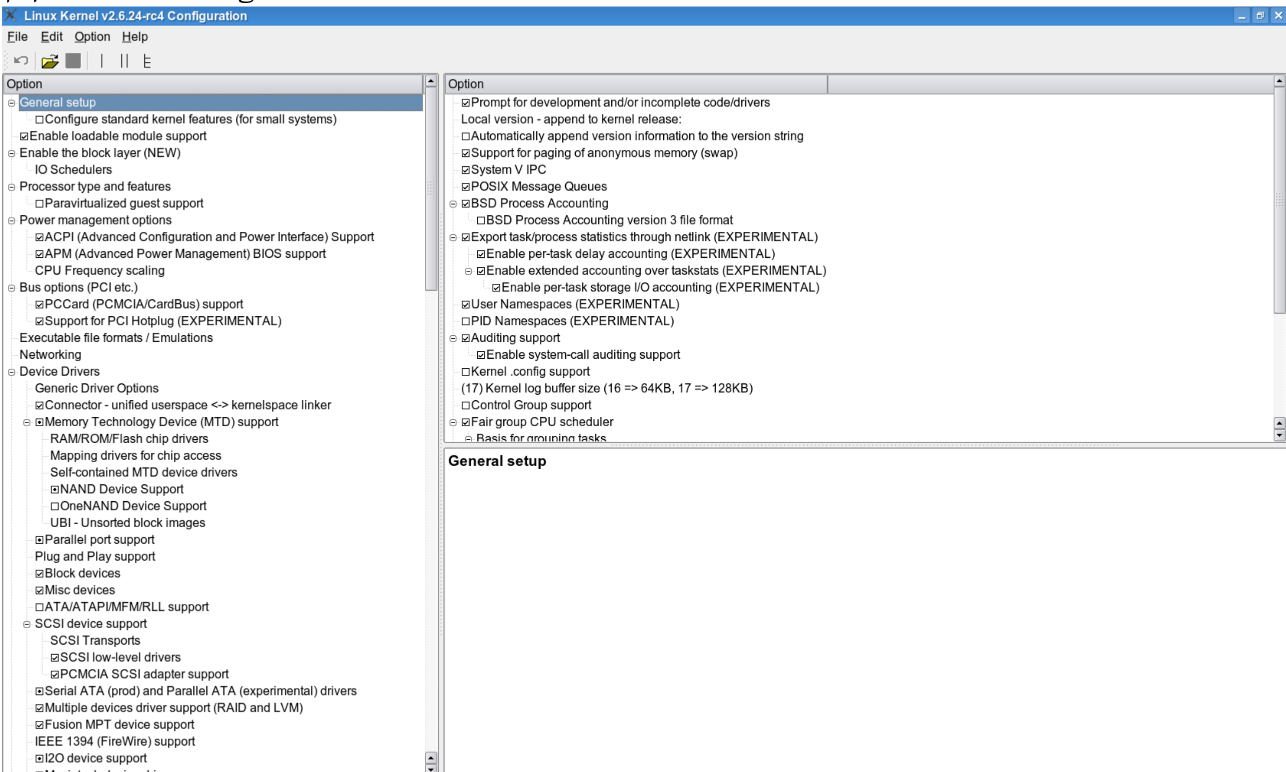
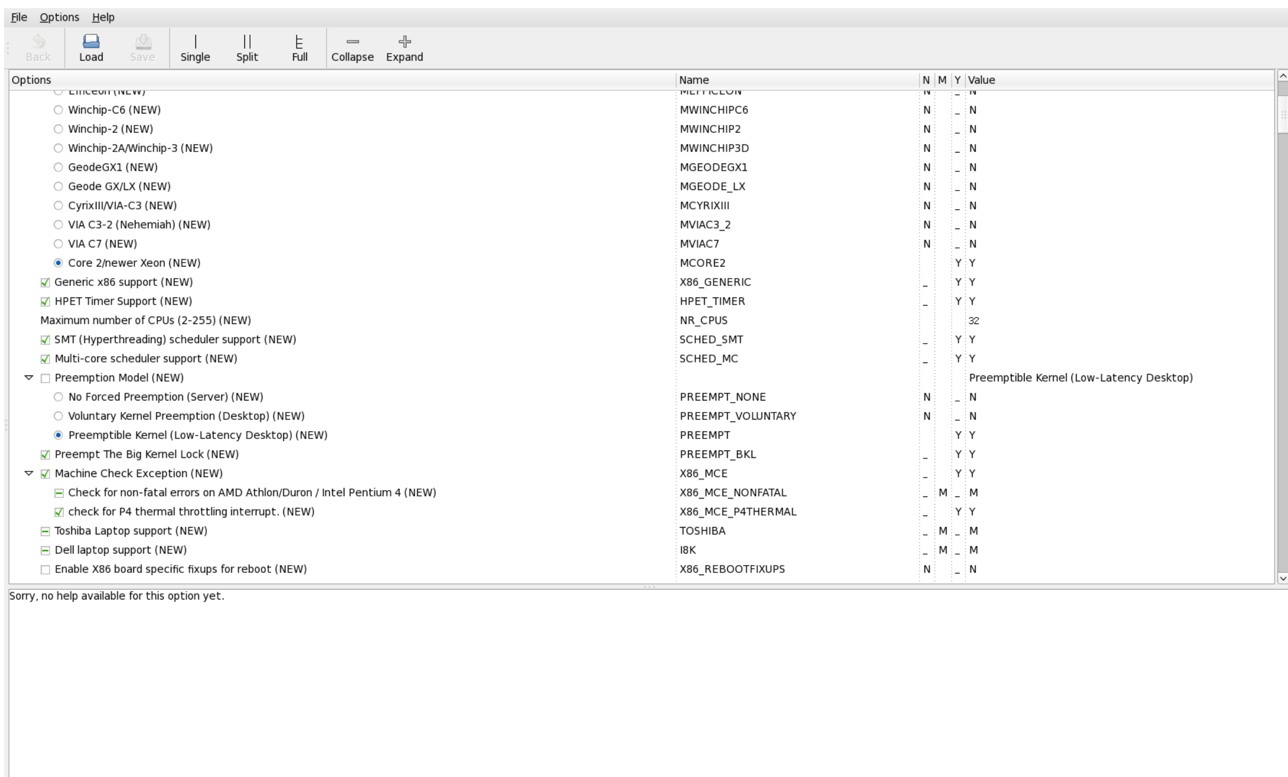


图 1.3 gconfig



2，浏览代码

=====================

2.1 vim + ctags + cscope

2.1.1 编译 ctags 需要的 tags 文件：

```
# make ARCH=x86 tags
```

FYI：ARCH=<arch>的意思是，除了指定的体系结构外，不索引其他体系结构相关的代码.

FYI：2.6.24 之前，用 i386 指代 32 位的 x86 处理器，用 x86_64 指代 64 位的 x86 处理器。 但在 2.6.24 中这 2 者即将合并，统称为 x86.

2.1.2 编译 cscope 程序需要的 cscope.out 文件：

```
# make ARCH=x86 cscope
```

2.1.3 在 Vim 中用 ctags 浏览

```
:ts do_fork    //tag select 效果等同于光标停在 do_fork 上，按 Ctrl-]。
               //如果想返回原来的位置，Ctrl-T
:tn            //tag next
:tp            //tag previous
:tf            //tag first
:tl            //tag last
```

FYI: 安装 Vim 的 taglist 插件，浏览更方便。 又，~/.vimrc 中控制 number:

```
 "added by albcamus
function! TlistWrapper2 ()
      Tlist
      if winnr('$') >= 2  " winnr() return the number of current window
                          " while winnr('$') returns the number of all windows
            set nonu
      else
            set nu
      endif
endfunction

command! -nargs=0 List call TlistWrapper2()
map <F1> :List<CR>:<CR>
```

则可以按 F1 键来调用 taglist。

2.1.4 在 Vim 中用 cscope 浏览

```
:cs add cscope.out     #添加一个 connection，浏览内核时经常发现 cscope 链接断开
                       可以调用:cs a cscope.out
:cs show               #显示所有 connections
:cs kill <#>           #杀死第#号 connection，第#号就是:cs show 显示出来的
:cs reset              #reset 所有 connections
:cs find c|d|e|f|g|i|s|t <name>
```

解释:

```
              c       查找 name 被哪些函数调用
```

d　　查找name调用了哪些函数
e　　egrep句型
f　　查找名为name的文件
g　　查找name的定义
i　　查找#include本name的文件
s　　查找name这个C符号
t　　查找何处对name赋值

对Linux内核来说，如果既有tags文件，又有cscope.out文件，则Vim中:set cst意味着使用cscope风格的^]，也就是:tag命令。于是：

如果想用ctags找到do_IRQ:
:ts do_IRQ
如果想用cscope找到do_IRQ:
:ta do_IRQ

FYI：可以在~/.vimrc中自定义命令，这样就不必每次都输入符号名：

```
nmap <C-\>g :cs find g <C-R>=expand("<cword>")<CR><CR>
nmap <C-\>c :cs find c <C-R>=expand("<cword>")<CR><CR>
nmap <C-\>s :cs find s <C-R>=expand("<cword>")<CR><CR>
nmap <C-\>t :cs find t <C-R>=expand("<cword>")<CR><CR>
nmap <C-\>e :cs find e <C-R>=expand("<cword>")<CR><CR>
nmap <C-\>f :cs find f <C-R>=expand("<cfile>")<CR><CR>
nmap <C-\>i :cs find i ^<C-R>=expand("<cfile>")<CR>$<CR>
nmap <C-\>d :cs find d <C-R>=expand("<cword>")<CR><CR>
```

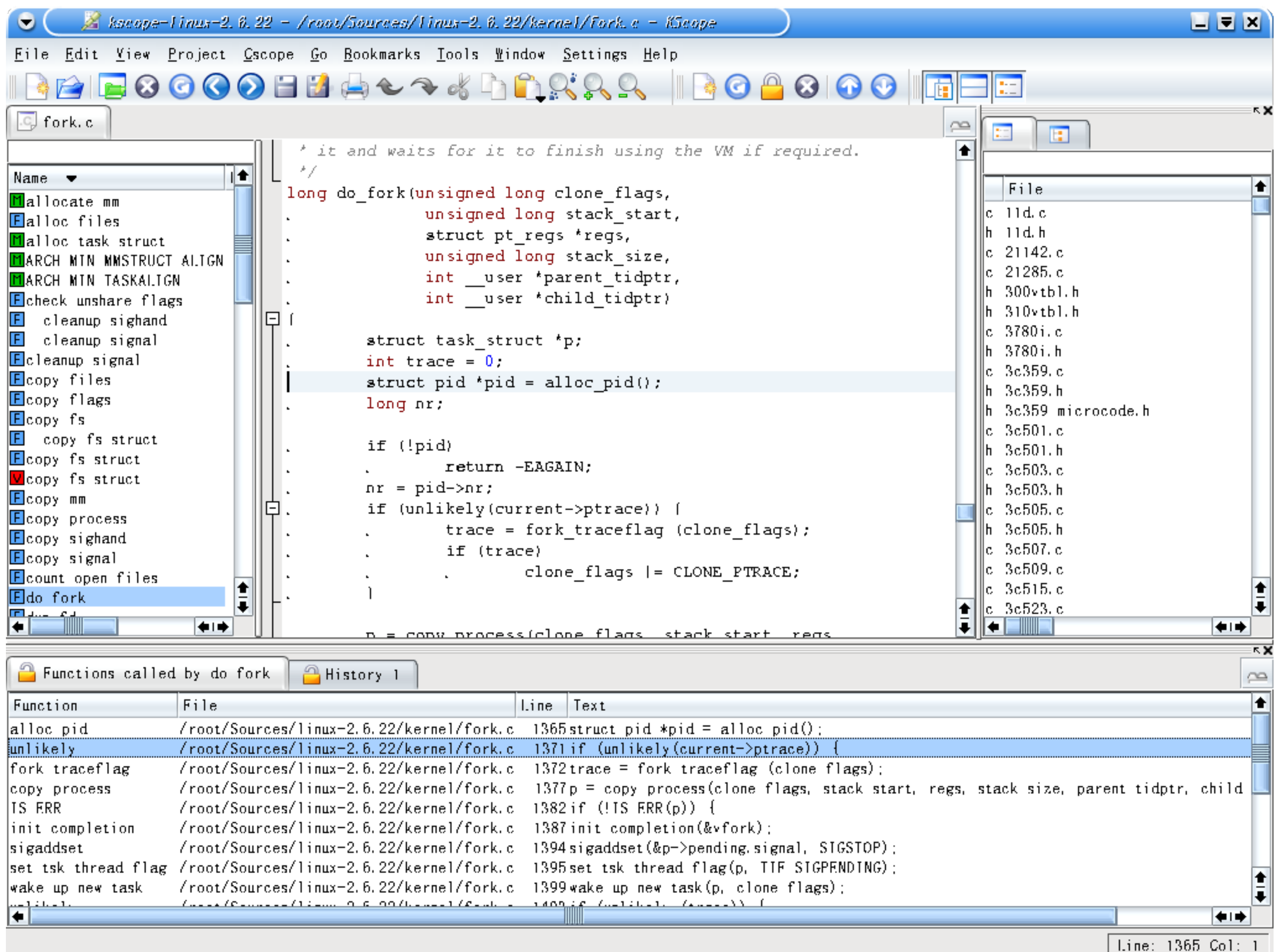
这样，当你在Vim中把光标挪到字符串do_fork上，按Ctrl-\c(先按Ctrl-\然后按c)，cscope就会列出所有调用了do_fork函数的地方。

```
        clear_local_APIC                    if (!apic_id_registered())
        disable_local_APIC                          BUG();
        lapic_shutdown
        verify_local_APIC                   /*
        sync_Arb_IDs                         * Intel recommends to set DFR, LDR and TPR before enabling
        init_bsp_APIC                        * an APIC.  See e.g. "AP-388 82489DX User's Manual" (Intel
        setup_local_APIC                     * document number 292116).  So here it goes...
        detect_init_APIC                     */
        init_apic_mappings                  init_apic_ldr();
        APIC_init_uniprocessor
        parse_lapic                         /*
        parse_nolapic                        * Set Task Priority to 'accept all'. We never change this
        parse_disable_lapic_tim              * later on.
        parse_lapic_timer_c2_ok              */
        apic_set_verbosity                  value = apic_read(APIC_TASKPRI);
        smp_spurious_interrupt              value &= ~APIC_TPRI_MASK;
        smp_error_interrupt                 apic_write_around(APIC_TASKPRI, value);
        apic_intr_init
        connect_bsp_APIC                    /*
        disconnect_bsp_APIC                  * After a crash, we no longer service the interrupts and a pending
        lapic_suspend                        * interrupt from previous kernel might still have ISR bit set.
        lapic_resume                         *
        apic_pm_activate                     * Most probably by now CPU has serviced that pending interrupt and
        init_lapic_sysfs                     * it might not have done the ack_APIC_irq() because it thought,
        apic_pm_activate                     * interrupt came from i8259 as ExtInt. LAPIC did not get EOI so it
                                             * does not clear the ISR bit and cpu thinks it has already serivced
-   genapic_32.h (/root/Sources              * the interrupt. Hence a vector might get locked. It was noticed
|-    macro                                  * for timer irq (vector 0x31). Issue an extra EOI to clear ISR.
        _ASM_GENAPIC_H                       */
        APICFUNC                            for (i = APIC_ISR_NR - 1; i >= 0; i--) {
        IPIFUNC                                     value = apic_read(APIC_ISR + i*0x10);
        IPIFUNC                                     for (j = 31; j >= 0; j--) {
        APIC_INIT                                           if (value & (1<<j))
                                                                    ack_APIC_irq();
|-    struct                                        }
        genapic                             }
 Tag_List     158,5      Bot ~/Sources/linux-2.6/arch/x86/kernel/apic_32.c                        852,20        56%
Cscope tag: setup_local_APIC
   #   line  filename / context / line
   1   1200  /root/Sources/linux-2.6/arch/x86/kernel/apic_32.c <<APIC_init_uniprocessor>>
             setup_local_APIC();
   2   1216  /root/Sources/linux-2.6/arch/x86/kernel/apic_64.c <<APIC_init_uniprocessor>>
             setup_local_APIC();
   3    281  /root/Sources/linux-2.6/arch/x86/kernel/smpboot_32.c <<smp_callin>>
             setup_local_APIC();
   4   1038  /root/Sources/linux-2.6/arch/x86/kernel/smpboot_32.c <<smp_boot_cpus>>
             setup_local_APIC();
   5   1049  /root/Sources/linux-2.6/arch/x86/kernel/smpboot_32.c <<smp_boot_cpus>>
             setup_local_APIC();
   6    214  /root/Sources/linux-2.6/arch/x86/kernel/smpboot_64.c <<smp_callin>>
             setup_local_APIC();
   7    886  /root/Sources/linux-2.6/arch/x86/kernel/smpboot_64.c <<smp_prepare_cpus>>
             setup_local_APIC();
   8     48  /root/Sources/linux-2.6/arch/x86/mach-visws/traps.c <<cobalt_init>>
             setup_local_APIC();
Choice number (<Enter> cancels):
```

FYI：GNU Global 程序也不错。 对喜爱 Emacs 编辑器的朋友，还可以用 etags 和 xcscope 来替代。 喜欢 Source Insight 的朋友也可以常识一下 kscope，它的界面和 SI 很类似。

kscope 依赖于 KDE 的库，如果你用 gnome 作窗口管理器，也得安装了 KDE 才能安装 kscope。 使用 kscope 之前需要安装 ctags、cscope 和 dot 程序(dot 在 graphviz 包中)。

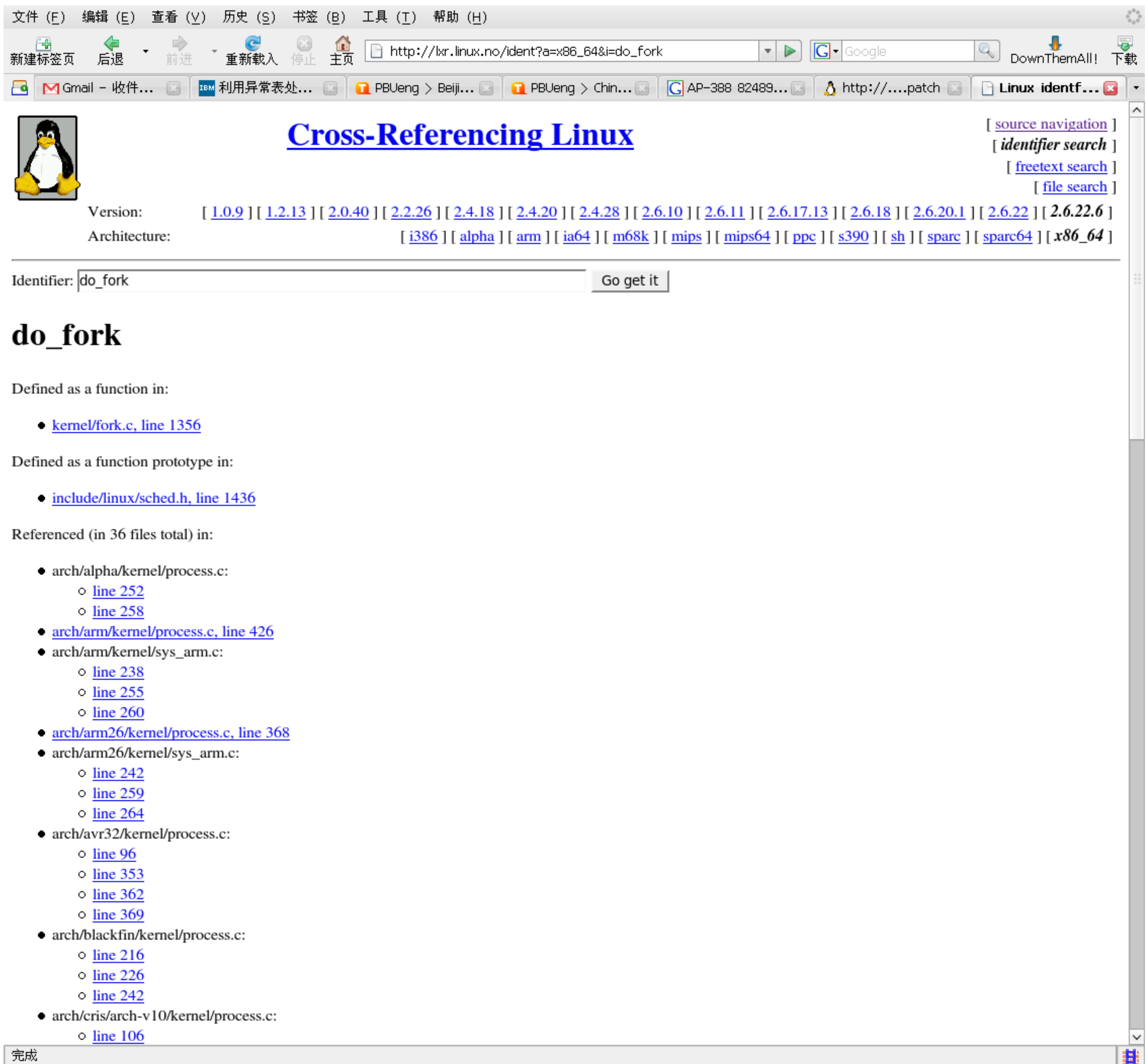
kscope 抓图：



2.2 lxr – Linux Cross Referrence

我一般都只在网上浏览:

http://lxr.linux.no/
http://users.sosdg.org/~qiyong/lxr/source/



3．内核相关的图书

==================================

入门推荐： LKD2 – 《Linux 内核设计与实现-第 2 版》

从入门开始，介绍了诸如中断、系统调用、虚拟文件系统、同步与互斥、内存管理、进程控制等方面，内容比较浅显易懂，是入门的好书。

优点： 适合入门 (个人认为，没有比 LKD2 更优秀的内核入门图书)
缺点： 内容不够深入，覆盖面不广。(对高手来说估计就像休闲材料)

3.1 ULK3 – 深入理解 Linux 内核-3rd

一本很全面的 Linux 内核原理书。 书中以 2.6.11 为示例版本，着重讲述了 x86 平台的 Linux 内核实现。

优点： 深入，全面
缺点： N/A
我觉得看完 ULK3 就是高手了:) 而且由于书中着重介绍了 X86 体系结构，也称得上半个 x86 专家了。

3.2 《Linux 内核源代码情景分析》

以 2.4.0 为例讲解，注重代码级别的剖析，对中断、内存管理、文件系统、SMP、PCI 和 USB、IPC 的讲解都是代码级别的深入细致。

优点： 深入
缺点： 针对的内核版本较旧，且每个「情景」都很长，不容易坚持读完。

FYI ： 新手不要从《情景分析》开始学习内核，这样只会增长你的学习周期。

3.3 LDD3 – Linux 设备驱动程序-3rd

LDD3 写的很精彩。 但如果要学会写具体的驱动程序，还是得参照硬件的 datasheet，读一个内核中现成的驱动程序。

FYI ：内核中自带的驱动程序 skeleton:

drivers/net/pci-skeleton.c 和 drivers/usb/usb-skeleton.c，分别是为 PCI/USB 驱动程序员提供的参考代码。

3.4 现代体系结构上的 Unix 系统 - 内核程序员的 SMP 和 Caching 技术

这本书着重讲解各种体系结构上的 Unix 实现注意事项，e.g. SMP 的同步与互斥、Cache 一致性问题。

优点：　作者知识面非常广，原理讲得很清楚。
缺点：　94 年的书，比较旧

3.5 Intel & AMD CPU 参考手册

最好带着问题有针对性的去读 Intel & AMD 的手册。

4，第一个模块：　Hello Kernel

===========================

4.1 代码

```
# cat Makefile

    ifneq ($(KERNELRELEASE),)
        obj-m := hello.o
    else
```

```
KBUILD := /lib/modules/`uname -r`/build

modules:
	make -C $(KBUILD) M=$(shell pwd) modules
clean:
	rm -f *.o *~ core .depend .*.cmd *.ko *.mod.c
	rm -rf .tmp_versions
endif
```

cat hello.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("The hello world demo module");
MODULE_AUTHOR("albcamus<albcamus@gmail.com>");

static int __init hello_init(void)
{
	printk("Hello Kernel!\n");

	return 0;
}

static void __exit hello_exit(void)
{
	printk("Bye!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

4.2 编译、加载和查看

编译:

make

加载

insmod ./hello.ko

查看:

dmesg |tail -1

Hello Kernel!

卸载:

rmmod hello

FYI : 接下来做什么? 在 Hello world 之后，有很多可以练习的小模块可以做，例如截获系统调用表(这个最好只拿来玩)，写一个 dummy 字符设备驱动

程序，添加一个系统调用，等等。

（TBD：如果有时间，就讲 intercept-execve 模块）

5，内核相关的网络资源

==

http://www.kernel.org

- git trees
- bugzilla
- LKML（和其他的 mailing list）

FYI：两个新闻组 comp.os.linux.development.system（讨论 linux 内核程序设计），和 fa.linux.kernel(LKML 的 USENET 版)
新闻组可以用 evolution/thunderbird 等订阅，也可以在 http://groups.google.com 访问。

http://kerneltrap.org

Linux 和 BSD 内核的技术新闻。 如果没时间跟踪 LKML，那么经常浏览 kerneltrap 是个好主意。

http://lwn.net

Linux weekly news

http://www.linuxforum.net

从1999年开始，积累了大量的高质量内核技术文档资源。

http://linux.chinaunix.net

积累壮大中。 有很多高手活跃，如zx_wing、puppylove、daemeon等。

http://www.ibm.com/developerworks/cn/views/linux/articles.jsp?view_by=search&search_by=%E5%86%85%E6%A0%B8

ibm developers works网站有很多关于内核的文档，大都写得通俗易懂。

http://zh-kernel.org/mailman/listinfo/linux-kernel

由Li Yang @ freescale维护，活跃着很多在Kernel开发领域活跃的华人，如Herbert Xu, Mingming Cao, Bryan Wu, Shaohua Li, Yanmin Zhang等都有参与，此外还有各个社区的高手如CLF的wheelz、daemeon、zh1lg等。

该社区有翻译的内核文档，还有Intel OTC的Linux ACPI项目。

6，其他的知识

=========================

在学习Linux内核时，我们经常发现很多内核之外的知识要掌握，否则就会成为

学习之路上的绊脚石。 大体上来说，各个领域都有它的特定知识需要掌握，例如要学习Linux内核的网络协议栈实现，就不可能不学习TCP/IP； 要学习Linux是如何管理物理内存，就不可能不学习某个CPU的内存管理机制。 等等。

这里只说一点： GCC的C扩展 和 inline ASM。

GCC的C扩展：

$ info gcc 'c e'

INLINE ASM(针对i386)：

《情景分析》第一章讲解inline ASM比较透彻。 另外有两篇很好的文档：
http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

http://groups.google.com/group/muc.lists.linux-kernel/browse_thread/thread/ca8860dd41ddd42b

针对x86-64的inline汇编和i386区别不大。

7，内核调试

====================

7.1 printk

朴素的printk永远是最受欢迎的。

缺点： 有时BUG导致直接reboot了，printk的内容甚至都看不到。 不过没关系，我们可以用console=/dev/ttyS0把console的输出定向到串口，在另一端接收；也可以用netconsole(详见Documentation/networking/netconsole.txt)

优点： 随时随地可用。

FYI： 使用 GCC 或 C99 的可变参数宏包裹 kprintk

```
C99 方式：
  =======
  #define MYDEBUG
  #ifdef MYDEBUG
          #define  dbg_print(fmt, ...) \
                  printk("%s-%d: "fmt, __FILE__, __LINE__,
##__VA_ARGS__);
  #else
          #define dbg_print(fmt, ...)  do {} while (0)
  #endif

  GCC 方式：
  =======
  #define MYDEBUG
  #ifdef MYDEBUG
          #define dbg_print(fmt, args...) \
                  printk("%s-%d: "fmt, __FILE__, __LINE__,
##args);
  #else
          #define dbg_print(fmt, args...) do {} while(0)
  #endif
```

然后就可以使用：

```
dbg_print("This is a debug messge, i = %d, and string is %d\n", i, str);
```

7.2 kdb

由 SGI 公司开发的一系列补丁。
http://oss.sgi.com/projects/kdb/

优点： 更新很快，基本能和官方内核同步。 稳定。
缺点： 只能做到汇编级别的源代码单步，不能做到 C 代码单步。

7.3 kgdb

linsys 公司开发的一系列补丁。 Linux 内核维护者 Andrew Morton 就使用它。

优点： C 源代码级别的单步。

缺点： BUG 较多，需要串口，两台机器。 免费版本支持的内核版本很少。
FYI ：这个问题可以规避：
a)，VMWare 中可以指定虚拟机的串口为 FIFO 文件，从而两台虚

拟机实现"串口"互联;

b), 较新的 kgdb 已经支持 kgdboe - Over Ethernet. 只要以太网卡驱动程序支持 NETPOLL 即可。

FYI : kgdb 即将被合并到官方内核中, 不过最早也得是 2.6.25.

FYI : 比 linsys 和 sourceforge 要全的一个地方:

http://www.kernel.org/pub/linux/kernel/people/agk/patches/2.6/

另 Andrew Morton 的-mm tree 中自带 kgdb:

http://www.eu.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/

7.4 kprobes/systemtap

kprobes 自 2.6.9 加入到官方内核中。
systemtap 是由 redhat 和 Intel 等公司开发的调试工具, 类似于 Solaris 的 DTrace。 它基于 kprobes。

kprobes
优点: 几乎内核中的任何地方都可以插入监测点. 而且可以使用它来截获那些不导出的符号(只要在 kallsyms 中有)
缺点: N/A

systemtap:

下载:

```
git clone git://sources.redhat.com/git/systemtap.git
```

优点: 用户态, 功能强大
缺点: 需要学习一门新的语言(stap)

文档:

主页:
http://sourceware.org/systemtap/documentation.html

使用 systemtap 调试内核:

http://www.ibm.com/developerworks/cn/linux/l-systemtap/

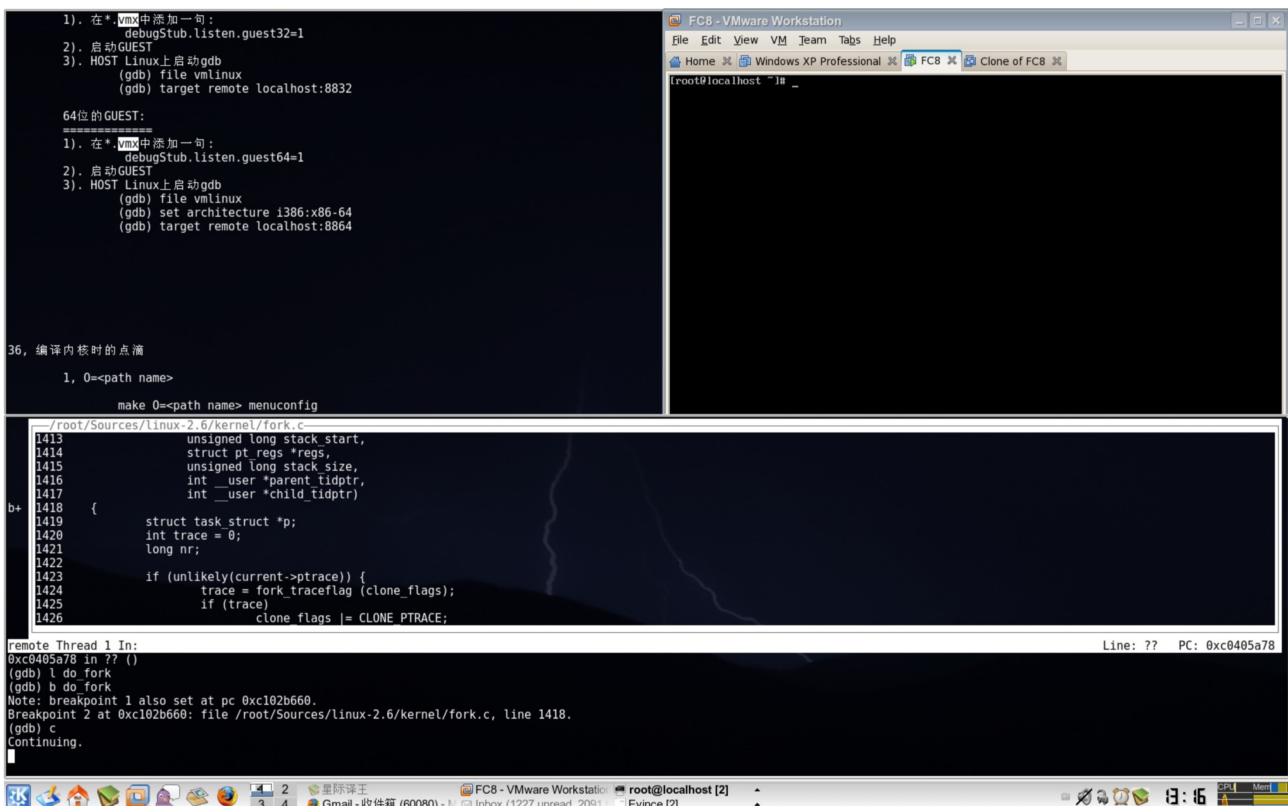
Linux 下的一个全新的性能测量和调试诊断工具 systemtap，第 3 部分：
http://www.ibm.com/developerworks/cn/linux/l-cn-systemtap3/

7.5 VMWare workstation 6.x

VMWare workstation 在 VCPU 中实现了 gdb stub

优点： 简单(不需要串口、不需要多台机器)，C 源代码单步.
缺点： 并不针对 Linux 系统，不能针对线程栈做分析。 学习意义大于工程意义。

文档：
http://blog.chinaunix.net/u/548/showart.php?id=282575



图： WS6.0

8，现代硬件： ARCH & DRIVER

==

8.1 现代X86 CPU

Intel 64 and IA 32 CPU Manuals:
 http://www.intel.com/products/processor/manuals/index.htm

AMD CPU Manuals:
 http://developer.amd.com/devguides.jsp

一些Linux支持(或需要更好地支持)的现代X86 CPU的特性:

多处理器: SMP，同步和互斥，APIC

内存管理: PAE，PSE，MTRR/PAT，Cache/TLB，x86-64，NUMA

系统调用: int 0x80，sysenter/sysexit，syscall/sysret

电源管理: ACPI
 -> 电源管理
 -> MPS
 -> PCI IRQ Routing
 -> PNP
 -> E820

扩展指令: MMX，SSE/SSE2/SSE3/SSE4，3DNow!

安 全 : NX

虚拟机 : VMX(Intel VT)，SVM(AMD-V)

8.2 外围设备

外围设备资料，建议下载 Intel ICH8 南桥芯片的 datasheet：
http://www.intel.com/design/chipsets/datashts/313056.htm

南桥芯片的 datasheet 会囊括 host controllers 手册。 当然，具体到某种设备，例如 USB 或 SATA，还需要看相关的规范。

9，80386 的分段机制

=======================================

1. 80386 为段映射提供的硬件

- GDT 表，LDT 表，TSS，都在内存中。(Linux 的实现是：这 3 个东西都是每个 CPU 有一个)
- 每个 CPU 中都有一个 gdtr 指向 GDT 表，一个 ldtr 指向 LDT 表，一个 tr 指向 TSS。
- 每个 CPU 有 6 个段寄存器：CS，DS，SS，ES，FS，GS。它们都是 16-bit 的。
  其中 CS、DS 分别指向代码和数据段，SS 指向堆栈段，都是特殊的；而 ES、FS、GS 可以用作普通目的，指向哪个 data segment 都成。

  这些段寄存器，叫作"Segment Selector"。亦即：由它的值决定一个地址(16 位的 segment selector 加上 32 位的线性地址)采用 GDT 或 LDT 中的哪个 entry — 每个 entry 都代表一个"Segment"，所以一个 entry 就是一个"Segment Descriptor"。
- 一个 segment selector 是这样的：

```
15________________3_|__2_|__1____0|
 |_______INDEX______|_TI_|__RPL___|
```

RPL:　Request Privilege Level。

CS寄存器的[1:0]就叫CPL，current privilege level。

TI:　Table Indicator。TI==0时，表示使用GDT表；TI==1时，使用LDT表。

INDEX:　在GDT或LDT表中的位置。

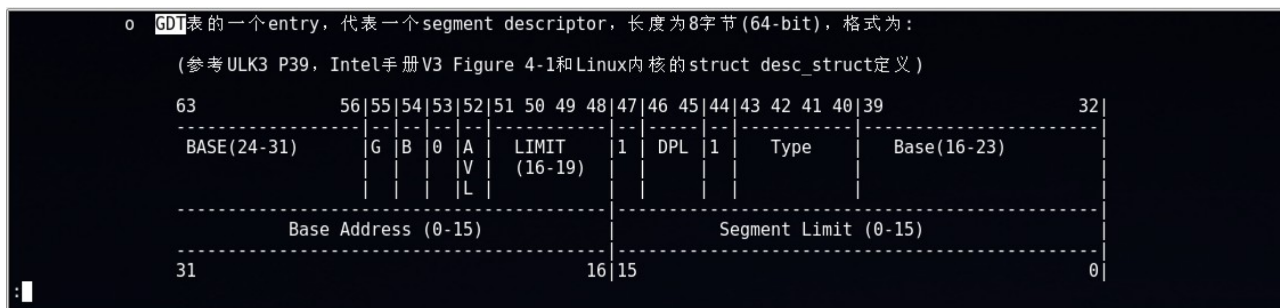
例如，TI==0,使用GDT表。INDEX为2，gdtr==0x00020000，而GDT表每个entry的大小为8字节(见下条)，于是会找到：

*0x00020000 + (2 * 8) == 0x00020010*

这个地址就是GDT表中相应的segment descriptor在内存中的地址。

o GDT表的一个entry，代表一个segment descriptor，长度为8字节(64-bit)，格式为：

(参考ULK3 P39，Intel手册V3 Figure 4-1和Linux内核的struct desc_struct定义)



```
     o  GDT表的一个entry，代表一个segment descriptor，长度为8字节(64-bit)，格式为：

        (参考ULK3 P39，Intel手册V3 Figure 4-1和Linux内核的struct desc_struct定义)

        63              56|55|54|53|52|51 50 49 48|47|46 45|44|43 42 41 40|39                  32|
        ------------------|--|--|--|--|-----------|--|-----|--|-----------|----------------------|
         BASE(24-31)      |G |B |0 |A |   LIMIT   |1 | DPL |1 |   Type    |    Base(16-23)       |
                          |  |  |  |V |  (16-19)  |  |     |  |           |                      |
                          |  |  |  |L |           |  |     |  |           |                      |
        ------------------------------------------|------------------------------------------------|
                     Base Address (0-15)          |             Segment Limit (0-15)               |
        ------------------------------------------|------------------------------------------------|
        31                                      16|15                                             0|
:
```

Table 48-1

A	Accessed
AVL	Available to System programmer
B	Big
C	Conforming
D	Default
DPL	Descriptor Privilige Level
E	Expansion Direction
G	Granularity
R	Readable
LIMIT	Segment Limit
W	Writable
P	Present

Linux内核对这个entry格式的定义：

```
struct desc_struct {
        u16 limit0;
        u16 base0;
        unsigned base1 : 8, type : 4, s : 1, dpl : 2, p : 1;
```

```
        unsigned limit : 4, avl : 1, l : 1, d : 1, g : 1, base2 : 8;
    } __attribute__((packed));
```

注意对 Code 段，来说，limit 由 Limit 和 G 决定； 对 Data 段和 Stack 段来说，Limit 还依赖于 B 和 E 位。但 Linux 根本不管 B 和 E 位。

其中代码段描述符、数据段描述符都可以放在 GDT 或 LDT 中作为 entries，但 TSS 段描述符只能放在 GDT 表中作一个 entry，LDT 描述符只能放在 GDT 表中作一个 entry。

- 对应 cs/ds/ss/es/fs/gs 这 6 个 segement selectors，可以当前指定 6 个 segment descriptors(在 gdt 表或 ldt 表中)。 为此，80x86 CPU 提供了 6 个软件不可见的 64-bit 寄存器，存储这 6 个当前的 segment descriptors。 这样，例如 cs 和 ds 已确定，那么 CPU 就把相应的 descriptor 加载到不可见寄存器中，从此不再访问内存中的 GDT/LDT 表读 descriptors，直到相应的 cs/ds 等 selector 被改变。

2. Linux 对段映射的实现。

ULK3 说，2.6 Linux 只有在 x86 处理器需要时，才在段映射一事上敷衍敷衍它。

- 作为 GDT 表中的一个 entry，__KERNEL_CS 的定义：

```
#define GDT_ENTRY_KERNEL_CS (GDT_ENTRY_KERNEL_BASE + 0)
#define __KERNEL_CS (GDT_ENTRY_KERNEL_CS * 8)
```

  解释：

  Linux 把内核代码段作为 GDT 表的第 12 项，所以其 INDEX 就是 12。 INDEX 在 CS 的高 13 位，低 3 位的 TI==0，因为用的是 GDT；RPL==0，因为是 RING0 权限。 这样，__KERNEL_CS 就是 12 << 3，亦即 12 * 8。

- 而__USER_CS：

```
#define GDT_ENTRY_DEFAULT_USER_CS        14
#define __USER_CS (GDT_ENTRY_DEFAULT_USER_CS * 8 + 3)
```

  解释：
  Linux 把用户代码段作为 GDT 表的第 14 项，所以其 INDEX 是 14。 INDEX 在 CS 的高 13 位，所以左移 3 位，亦即乘上 8。 至于低 3 位，因为用的是 GDT，所以 TI==0；又因为用户态，所以 RPL==3。所以__USER_CS 就是 14 << 3 + 3，亦即 14 * 8 + 3。

- __KERNEL_DS 是 GDT 表中第 13 项，__USER_DS 是 GDT 表中第 15 项。

好吧，我们看看Linux定义的这4个segments，其base address和limit的设置。

```
GDT[12] == kernel cs
GDT[13] == kernel ds
GDT[14] == user cs
GDT[15] == user ds
```

在GDT表中这4个entries的base address都是0x00000000，其limit都是0xfffff。 每个etnry的G标志被设为1。(注意，如果一个entry的G设为0，表示limit的单位是byte，那么limit == 0xfffff就指定了1MB的空间；如果G被设置为1， 则limit的单位是4096字节，于是limit==0xfffff就表示 (2^20)*(2^12) == 4GB 的空间)

Linux就是这么敷衍80386的segmenting unit的： 4个段，每个都是flat的4GB大小。分段，什么都不做。 逻辑地址.offset == 线性地址。(注意，intel的"逻辑地址"定义是48bit的，其中16bit是segment selector)

arch/x86/kernel/cpu/common.c :

```
DEFINE_PER_CPU(struct gdt_page, gdt_page) = { .gdt = {
        [GDT_ENTRY_KERNEL_CS] = { 0x0000ffff, 0x00cf9a00 },
        [GDT_ENTRY_KERNEL_DS] = { 0x0000ffff, 0x00cf9200 },
        [GDT_ENTRY_DEFAULT_USER_CS] = { 0x0000ffff, 0x00cffa00 },
        [GDT_ENTRY_DEFAULT_USER_DS] = { 0x0000ffff, 0x00cff200 },

        {其他的忽略}
} };
EXPORT_PER_CPU_SYMBOL_GPL(gdt_page);
```

我们来看看，__KERNEL_CS的GDT entry赋值为{ 0x0000ffff, 0x00cf9a00 }是什么意思。

__KERNEL_CS在GDT表中的entry值:

对照着Table 48-1，知道base address为0x00000000，limit为0xfffff，Gruanularity为1 - 亦即4KBytes为单位。

0000 0000 1100 1111 1001 1010 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111

o UP Linux中只有一个GDT表，但SMP中，每个CPU都有自己的GDT表。(include/asm-x86/desc_32.h)

```
struct gdt_page
{
	struct desc_struct gdt[GDT_ENTRIES];
} __attribute__((aligned(PAGE_SIZE)));

DECLARE_PER_CPU(struct gdt_page, gdt_page);
```

每个 CPU 都有一个 GDT 表，变量名就叫做 gdt_page，是个 struct gdt_page 结构，其中域 gdt[GDT_ENTRIES]就是 GDT 表，每个 entry 都是一个 struct desc_struct 结构体变量。

o Linux 下基本不用 LDT。
所以只定义了一个默认的 LDT，放在 default_ldt 数组中。它包含 5 项，但只有两个常用：一个兼容 iBCS 操作系统的可执行文件，一个兼容 Solaris/x86 的可执行文件。

Wine 需要修改 ldt，于是 Linux 提供了 modify_ldt(2)系统调用。

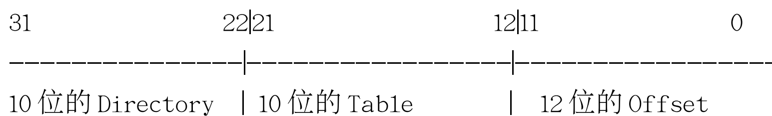
10. IA32 Linux 的 2-level 页面映射

===

参考： INTEL 手册 V3：Ch3, Section 3.7： Page Translation using 32-bit Physical Addressing

PART I：线性地址的划分

2-level：32 位的线性地址被划分为 3 部分：



高 10 位 ： 又叫 Page Directory, PD
中间 10 位 ： Page Directory, PT
低 12 位 ： 页内 offset

每个进程都有自己的页表，用以找到其 PD 的物理地址，称为 PGD：Page Global Directory。

每个进程都有自己的页表，其 PGD 的物理地址放在 CR3 寄存器中，见 schedule > context_switch > switch_mm > load_cr3(next->pgd).

PART II：MMU 分页单元寻找物理地址的过程

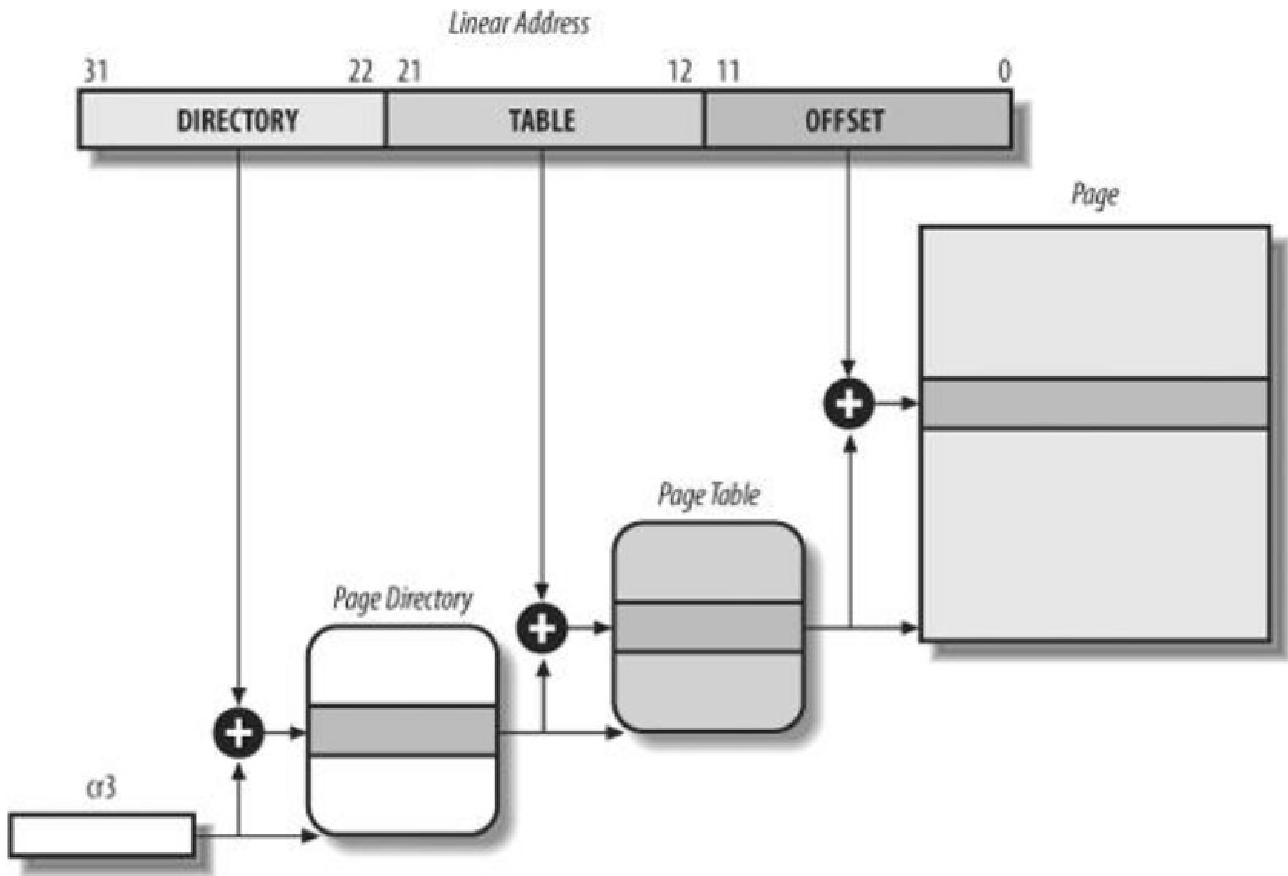
---

1) MMU 根据 CR3 中的 pgd 值， 和要访问的那个线性地址的高 10bit(就是在 PD 中的 offset)，在 PD 中找到相应的的 entry —— 一个 PD 的 entry，是一个 32 位的物理地址。 其中高 20bit 是 Page Table 的 Base Address.

2) MMU 根据 Page Table 的 Base Address，和要访问的那个线性地址的中间 10bit([12:21])，在 PT 中找到相应的 entry —— 一个 PT 的 entry(就是传说中的 PTE)，是一个 32 位的物理地址。 其中高 20bit 是 Page 的 Base Address -- 找到 Page 的 base address 就找到了页面的物理地址。

3) MMU 根据 Page 的物理地址(which 低 12 位都是零，因为页首地址是 4096 对齐的)，和要访问的那个线性地址的低 12bit，找到该地址在该页面中的位置 — 这就是传说中的「物理地址」，和那个要访问的线性地址对应的。

线性地址=>物理地址(来自 ULK3):



PART III：标志

----

* PD和PT中的entries，其格式是一样的。 只不过：
* PD的entry约束的是一个包含PT的页面，而PT的entry约束的是一个包含有效数据的页面。

BIT	Flag	说明
0	P, Present	为1，则Page在RAM中；　为0，则Page不在RAM中。 如果CPU试图访问某线性地址，结果在中途发现PDE或PTE的P标志为0，则会把该线性地址写入CR2寄存器，然后触发Page Fault。 CPU并不改写PDE或PTE的flags，而是让OS来做这些: 1). 把页面从交换设备上swap到RAM中 2). 为该物理页面建立起映射，也就是修改相应的PDE和PTE，并且将P设置为1。 其他的flags如Dirty和Accessed，页可以在此时设置。 3). Invalidate这个PTE在TLB Cache中的条目 4). 从Page-Fault handler中返回 我觉得，CPU遭遇PDE.P==0和PTE.P==0的情况还不一样。 后者说明页面不存在，而前者说明连页表都不存在.
1	R/W, Read/Write	为0，则只读；　为1，则可读写。 指定页面，或一组页面的读写属性。(啊，为什么会是「一组页面」? Intel手册说，PTE约束一个页面，而PDE约束了一个Page Table，从而可以约束更多页面 -- Bull shit！) 当CR0.WP == 0 (CPU复位时默认就这样)，且CPU处于Supervisor模式，则任何页面都是可以读/写的。 当CPU处于User模式，则Supervisor模式的页面既不可读也不可写，即使是User模式的页面，也要受这里的读写属性来约束。 从PPro开始，CPU支持copy-on-write. 亦即: a) CR0.WP == 1时，如果某User模式的页面只读，那么Supervisor模式的CPU去尝试写它，会发生page fault。 b) CR0.WP == 1时，如果某Supervisor模式的页面只读，那么不管CPU处于什么模式，都不能写它
2	U/S, User/Supervisor	为0,表示页面是Supervisor态的；　为1,表示页面是User态的。
3	PWT, Write-Through	为0，则write-back；　为1,则write-through. 如果CR0.CD == 1，则CPU会忽略该bit。
4	PCD, Cache Disabled	为1，则该页面不被Cache；　为0，则页面被Cache 当为MMIO的页面建立映射时，应该置位该比特以禁止CPU Cache它。
5	A, Accessed	为1,表示页面被访问过了；　为0,表示没被访问过。 一般来说，一个页面刚被load进内存时，该位置0。 什么时间它被访问了，CPU就会把该位置1.
6	D, Dirty	***Note，PDE中这一bit不起作用。只有PTE中的该bit起作用*** 为1,表示页面Dirty；　为0,表示页面Clean。
7 @ PDE (4K pages)	PS, Page Size	为0表示4K大小；　为1表示4M大小。(如果为1,且PAE被激活，则页面大小为2M)
7 @ PTE (4K Pages) 和12@PDE(4M Pages)	PAT, Page Attribute Table Index	PAT在Pentium III中引入。 与PCD和PWT标志一起使用，来在PAT表中索引到某个entry。 对于不支持PAT的CPU，该位应当永远置0。 FIXME: PAT有空研究一下
8	G, Global Page	该位在PPro CPU中引入。 为1，则这是一个Global Page；　为0则不是GP。 注意，当CR4.PGE==1时才有效。 Global Page的意思是:　它是全局的，因此当进程切换时(或CR3被写时)，不需要invalidate该页的映射在TLB中的entry。 FIXME：我觉得Linux的内核态内存就是这样。 各个进程的CR3所指向的页表都有一部分是指向了内核内存的页表，但是所有进程都使用想通的内核内存页表，因此不需要在进程切换时改变什么。
[9:11]	Avail, Available for system programmer's use	N/A

Thank you!

Any Questions?