

《Linux 内核》实验手册

上海交通大学 计算机科学与工程系

审稿：邹南海 张衍民 朱轶

2009 年 6 月

目 录

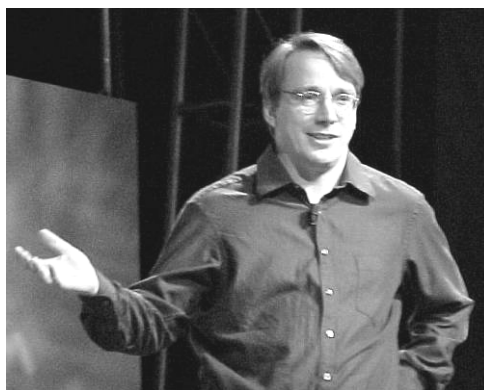
前言	1
§ 1 LINUX内核是什么?	1
§ 2 《LINUX内核》课程实验简介	1
模块编程	2
§ 1 前言: LINUX内核模块是什么?	2
§ 3 实验内容	2
§ 4 实验完成效果	2
§ 5 实验参考: 安装LINUX发行版	3
§ 6 实验参考: BASH(GNU BOURNE-AGAIN SHELL) 知识和常用命令	5
§ 7 实验参考: 下载最新版LINUX内核	6
§ 8 实验参考: 编译和安装新LINUX内核	7
§ 9 实验参考: 使用计算中心的计算机进行实验的注意事项	10
§ 10 实验参考: 内核模块的MAKEFILE样例	12
§ 11 实验参考: 一个简单的内核模块	13
§ 12 实验参考: PRINTK	14
§ 13 实验参考: MODULE_PARAM和MODULE_PARAM_ARRAY	14
§ 14 实验参考: CREATE_PROC_READ_ENTRY和CREATE_PROC_ENTRY	15
§ 15 附录: 同学提问	16
[1] typedef的语法	16
[2] 内核目录全路径中有空格导致不能make xconfig	17
[3] init.h module.h kernel.h 有什么作用?	18
[4] 在Debian 4.1.2 上启动新内核的办法	18
[5] 应用程序正在使用yum锁, 怎么办?	18
[6] module_param_string和module_param(charp)	19
[7] read_proc函数被执行两次的两种原因	19
进程管理与对称多处理器	21
§ 1 实验内容以及完成效果	21
作业一	21
作业二	22
§ 2 实验参考: 阅读内核代码	27
§ 3 实验参考: 将你对内核的改动做成补丁	30
§ 4 TGID_BASE_STUFF和TID_BASE_STUFF	31
§ 5 实验参考: SCHEDULE和SCHEDULE_TIMEOUT	31
§ 6 实验参考: 实现一个可写的PROC文件	32
§ 7 实验参考: __BUILTIN_RETURN_ADDRESS	32
§ 8 实验参考: LINUX内核代码段的起始和终止地址	32
§ 9 实验参考: 获得大于 4M的内存	33
§ 10 实验参考: 实现一个可读写的SEQ文件	34

§ 11	实验参考: GPROF (GNU PROFILER)	34
§ 12	实验参考: LINUX的PID=0 的进程和PID=1 的进程.....	34
§ 13	附录: 同学提问	35
[8]	switch_to	35
[9]	sched_info	36
[10]	proc_misc.c	37
[11]	INF ONE REG宏有什么区别.....	37
[12]	为什么switch_to宏有三个参数?	38
内存管理		41
§ 1	实验内容	41
§ 2	实验参考: 逻辑地址 (LOGICAL ADDRESS), 虚拟地址 (VIRTUAL ADDRESS), 物理地址 (PHYSICAL ADDRESS)	41
§ 3	实验参考: I386 的分段 段寄存器 段描述符 全局描述符表.....	42
§ 4	实验参考: /PROC/<PID>/MAPS 和 PMAP.....	42
§ 5	思考题	43
参考文献		44

前言

§ 1 Linux 内核是什么？

“Linux”或“Linux 内核”是用 C 语言和汇编语言写成的操作系统内核。Linux 提供了硬件抽象、磁盘及外部设备控制、文件系统控制、多任务等功能(而“Linux 发行版”表示建立在 Linux 之上的不同的操作系统)。Linux 由 L. Torvalds(右图为其照片)于 1991 年创造。



§ 2 《Linux 内核》课程实验简介

《Linux 内核》课程实验旨在培养同学自主实验能力，激发同学学习 Linux 的愿望和探求 Linux 原理的兴趣。同学们在十八个教学周内完成五个实验：模块编程；进程管理与对称多处理器；内存管理；文件系统 和 设备驱动程序。本实验手册包括实验内容和补充材料。学有余力的同学应当不拘泥于实验，主动研读手册中给出的资料、文献，主动思考回答问题。

模块编程

§ 1 前言：Linux 内核模块是什么？

模块是可以在 Linux 内核正在运行时，添加到内核中的代码。模块不是完整的可执行程序。模块中的代码是事件驱动的。模块必须有初始化函数(初始化函数用于为以后调用模块函数预先做准备，没有初始化函数的模块没有意义)和退出函数(退出函数负责释放资源和做清理工作，它在模块被卸载之前调用。没有退出函数的模块无法被卸载)。

模块运行在内核空间内，而应用程序运行在用户空间内。模块的代码执行时，处理器处于最高级别，处理器可以进行所有操作。

我们可以使用 `insmod` 程序将模块连接到内核，也可以用 `rmmod` 程序移除连接。

§ 3 实验内容

编译一新的 Linux 内核，并启动之。

编写一内核模块：加载和卸载此模块时能输出消息。消息用 `dmesg` 查看。

编写一内核模块：加载模块时可以指定一个整数参数。编写代码输出此参数。

编写一内核模块，创建一个 `/proc` 目录中的文件，并且读这个文件能读到数据。

§ 4 实验完成效果

本完成效果仅仅是一个例子。粗体字为命令，非粗体字为输出。

在全文中，如不提及，则执行的命令的操作系统均为 Ubuntu。

```
ls
1.ko 2.ko 3.ko

sudo insmod 1.ko

dmesg | tail -1
[16530.860331] Greeting from a linux kernel module.

lsmod | grep 1
1                2688  0
nls_iso8859_1    6528  1
nls_cp437        8192  1
```

```
.....

sudo rmmod 1

dmesg | tail -1

[16671.692236] Bye.

sudo insmod 2.ko int_param=1 string_param=hi array_param=1,2,3

dmesg | tail -3

[16688.903974] Param: int_param: 1;
[16688.903975]      string_param: hi;
[16688.903976]      array_param: 1, 2, 3,

sudo rmmod 2

ls /proc/Task1

ls: 无法访问/proc/Task1: 没有该文件或目录

sudo insmod 3.ko

ls /proc/Task1 -l

-r--r--r-- 1 root root 0 2009-02-22 14:45 /proc/Task1

cat /proc/Task1

Message from a linux kernel module ~.~

sudo rmmod 3

ls /proc/Task1

ls: 无法访问/proc/Task1: 没有该文件或目录
```

§ 5 实验参考：安装 Linux 发行版

为完成本实验，推荐装 Windows&Linux 双系统 / 多系统。如装双系统时遇到困难，也可在虚拟机中装 Linux 发行版。

- Linux 发行版推荐装 Ubuntu 或 Fedora。2009 年 2 月的最新版，分别通过以下链接下载

<ftp://ftp.sjtu.edu.cn/ubuntu-cd/8.10/ubuntu-8.10-desktop-i386.iso>;

<ftp://ftp.sjtu.edu.cn/fedora/linux/releases/10/Live/i686/F10-i686-Live.iso>。

若你的电脑因为硬件支持的原因，不能装最近版本，那么装稍旧版本 (Ubuntu 8.04 或 Fedora 9) 也可。

- 安装双系统的详细方法

见 http://wiki.debian.org.hk/w/Install_Ubuntu

http://wiki.debian.org.hk/w/Install_Fedora_Linux

- 在虚拟机中装 Linux 的方法

推荐装虚拟机 VirtualBox，可从 <http://www.virtualbox.org/wiki/Downloads> 下载之，其余安装

过程同上。

建议同学们为虚拟机分配至少 10.45G 的虚拟硬盘空间。硬盘空间不足的话，编译内核时会出 “ld: final link failed: No space left on device” 一类错误。

● Linux 装好后，还需要设置软件更新源，和装编译工具。

1. 用 Ubuntu 8.10 的同学推荐将/etc/apt/sources.list 内容改为：

```
deb http://ftp.sjtu.edu.cn/ubuntu/ intrepid main restricted universe multiverse
deb http://ftp.sjtu.edu.cn/ubuntu/ intrepid-backports restricted universe multiverse
deb http://ftp.sjtu.edu.cn/ubuntu/ intrepid-proposed main restricted universe multiverse
deb http://ftp.sjtu.edu.cn/ubuntu/ intrepid-security main restricted universe multiverse
deb http://ftp.sjtu.edu.cn/ubuntu/ intrepid-updates main restricted universe multiverse
deb-src http://ftp.sjtu.edu.cn/ubuntu/ intrepid main restricted universe multiverse
deb-src http://ftp.sjtu.edu.cn/ubuntu/ intrepid-backports main restricted universe multiverse
deb-src http://ftp.sjtu.edu.cn/ubuntu/ intrepid-proposed main restricted universe multiverse
deb-src http://ftp.sjtu.edu.cn/ubuntu/ intrepid-security main restricted universe multiverse
deb-src http://ftp.sjtu.edu.cn/ubuntu/ intrepid-updates main restricted universe multiverse
```

即使用我校的软件源，并安装 build-essential 软件包。

```
sudo apt-get update
```

```
sudo apt-get install build-essential -y
```

此外，编译内核需要安装 libncurses-dev libqt3-mt-dev 软件包。即

```
sudo apt-get install libncurses-dev libqt3-mt-dev -y
```

另外为生成 initramfs 镜像，需

```
sudo apt-get install initramfs-tools -y
```

2. 使用 Fedora 10 的同学建议添加上海交通大学软件源

```
su -c "rm /etc/yum.repos.d/* -f"
```

```
su -c "gedit /etc/yum.repos.d/sjtu.repo"
```

/etc/yum.repos.d/sjtu.repo 内容为

```
[Fedora-ftp.sjtu.edu.cn]
name=Fedora 10 - i386
baseurl=http://ftp.sjtu.edu.cn/fedora/linux/releases/10/Fedora/i386/os/
enabled=1
gpgcheck=0
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-Fedora file:///etc/pki/rpm-gpg/RPM-GPG-KEY
[Everything-ftp.sjtu.edu.cn]
name=Everything 10 - i386
baseurl=http://ftp.sjtu.edu.cn/fedora/linux/releases/10/Everything/i386/os/
enabled=1
gpgcheck=0
```

```
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-Fedora file:///etc/pki/rpm-gpg/RPM-GPG-KEY
[updates-ftp.sjtu.edu.cn]
name=Fedora updates
baseurl=http://ftp.sjtu.edu.cn/fedora/linux/updates/10/i386/
enabled=1
gpgcheck=0
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-Fedora file:///etc/pki/rpm-gpg/RPM-GPG-KEY
```

然后执行如下命令，安装必需的软件包

```
su -c "yum makecache"

su -c "yum install ncurses-devel qt3-devel libXi-devel gcc gcc-c++ make -y"
```

§ 6 实验参考：BASH (GNU Bourne-Again SHell) 知识和常用命令

- BASH 中 `~` 符号表示用户的主目录。root 用户的主目录为 `/root`。其他用户，如果有主目录的话，为 `/home/目录名/`。因此 root 用户执行 `ls ~` 等价于执行 `ls /root`
- `.` 表示当前目录 `..` 表示当前目录的上一级目录 `/` 表示根目录 根目录的上一级目录是根目录
- BASH 的通配符有
 - ? 匹配任意一个字符
 - * 匹配零个或任意多个字符
 - [a-z] 匹配所有小写字母
 - [124] 匹配 1 或 2 或 4
 - [!a] 匹配不是 a 的一个字符
 因此执行 `ls [abc]` 等价于执行 `ls a b c`
- 一个命令运行完后 BASH 才执行下一命令。若在命令最后加 `&` 符号，则该命令将后台运行，BASH 不等此命令结束，就开始执行下一命令。
- 用 `<` 和 `>` 进行输出和输入的重定向。如 `ls > a_file` 将 `ls` 命令的输出写入 `a_file` 文件。
- 在一个命令执行过程中，按 `Ctrl C` 停止它
- BASH 常用命令
 1. **cat** *concatenate files and print on the standard output*
语法: **cat** FILE 输出文件 FILE 的内容
 2. **ls** *list directory contents*
语法: **ls** [参数][更多参数][文件/文件夹][更多文件/文件夹] 中括号表示可选
参数:
 - a *do not ignore entries starting with .*
 - d *list directory entries instead of contents*
 - l *use a long listing format*
 例如:
 - `ls -la` 以长格式列出当前目录中所有文件/文件夹 (含隐藏文件/文件夹)
 - `ls /usr/local/bin/bash /etc/ /home/` 列出多个文件/文件夹。对于文件夹，列出其内容。
 3. **cp** *copy files and directories*
语法: **cp** SOURCE DEST
如果 SOURCE 是文件，那么:
 - 若 DEST 不存在，那么将 SOURCE 复制为 DEST
 - 若 DEST 是已经存在的文件，那么用 SOURCE 覆盖 DEST
 - 若 DEST 是已经存在的目录，那么将 SOURCE 复制为 DEST 目录内的一个文件

如果 SOURCE 是目录，那么什么也不做。(用 -r 参数复制目录)

cp SOURCE1 SOURCE2 SOURCE3 DIRECTORY 将多个文件 (SOURCE1 SOURCE2 SOURCE3) 复制到 DIRECTORY 文件夹内

参数:

-r *copy directories recursively*

4. **mv** *rename or move files/directories*

语法: **mv** SOURCE DEST

如果 DEST 是目录，那么移动 SOURCE，做为 DEST 的子文件/子文件夹

如果 DEST 是文件，那么:

若 SOURCE 是目录，那么什么也不做

若 SOURCE 是文件，那么用 SOURCE 覆盖 DEST 再删去 SOURCE

5. **mkdir** *make directories*

参数:

-p *make parent directories as needed*

例如: **mkdir** -p dir1/dir2/dir3/dir4

6. **rm** *remove files or directories*

语法: **rm** 文件/文件夹 [更多文件/文件夹]

参数:

-r 指定 -r 参数时，rm 连同目录一并删除；不指定 -r 参数时不删除目录。

-i 删除每个文件/目录之前先询问是否删除

7. **pwd** *print name of current/working directory*

● 本实验用到的命令

8. **insmod** 文件名 [模块参数名=参数值] [...]

将模块代码连接到正在运行的 Linux 内核

9. **dmesg**

输出 Linux 内核环形缓冲区的内容。

10. **tail -3** 文件名

输出文件的最后三行。同理， **tail -10** 文件名 输出文件的最后十行

11. **grep** 表达式 文件名

输出文件中匹配表达式的行

12. **lsmod**

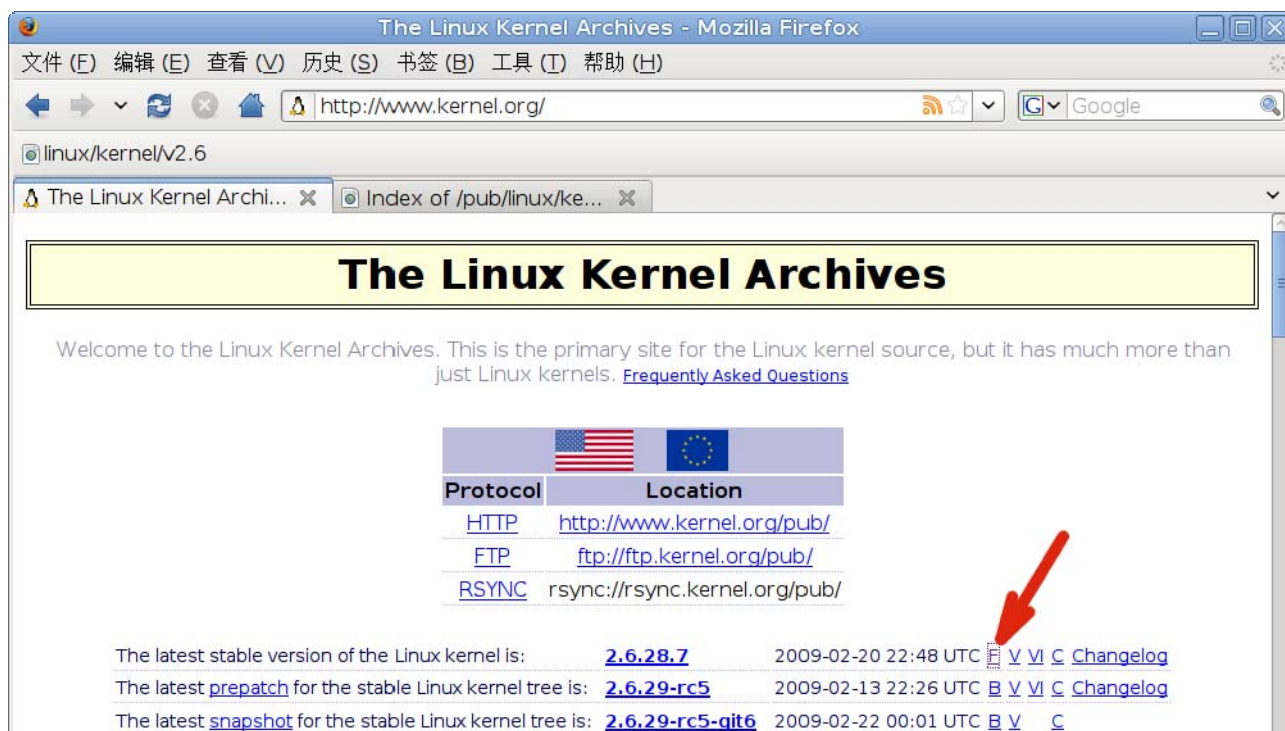
列出所有内核模块的名称、大小(单位为字节)、正在被哪些模块使用。

13. **rmmmod** 模块名

将模块于 Linux 内核之间的连接断开

§ 7 实验参考：下载最新版 Linux 内核

打开 <http://www.kernel.org>



点击箭头所指的“F”下载最新版 Linux 内核。

如果下载速度较慢，也可从位于中国大陆的镜像站点 www.cn.kernel.org 下载内核：

打开 <http://www.cn.kernel.org/pub/linux/kernel/v2.6/>，找到名为“LATEST-IS-2.6.xx.x”的文件。在 2009 年 2 月 22 日，此文件为 LATEST-IS-2.6.28.7。因此我们知道，内核最新版本为 2.6.28.7，对应文件为 linux-2.6.28.7.tar.bz2。我们下载

<http://www.cn.kernel.org/pub/linux/kernel/v2.6/linux-2.6.28.7.tar.bz2>

注：.bz2 文件，是一种使用 Burrows-Wheeler 压缩算法和 Huffman 编码的压缩文件。有兴趣研究 B-W 压缩算法的同学可以阅读以下文献：

1. I.Witten, R.Neal and J.Cleary, Arithmetic coding for data compression, Communications of the Association for Computing Machinery, 30 (6) 520-540, June 1987.
2. M.Burrows and D.Wheeler, A Block-sorting Lossless Data Compression Algorithm, 1994.
3. M.Nelson, Data Compression with the Burrows-Wheeler Transform, Dr. Dobbs' Journal, 1996.

注：-mm 补丁集合由 Andrew Morton(右图为其照片)维护。其中的补丁比官方的 Linux 内核代码更具有实验性。而一个补丁是否加入 Linux 内核是由 Andrew Morton 决定的。

有兴趣进一步了解 -mm 补丁集合的同学可参阅

1. <http://www.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.28-rc2/2.6.28-rc2-mm1/patch-list>
2. <http://www.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.28-rc2/2.6.28-rc2-mm1/broken-out/>



§ 8 实验参考：编译和安装新 Linux 内核

- 首先将内核压缩包。你可以在压缩包上点击鼠标右键，选择“解压缩到此处”。或者用 tar 命令，如

```
tar -jxvf linux-2.6.28.7.tar.bz2
```

注意：-v 选项表示列出解压出的文件。-v 可以不加。即 `tar -jxf linux-2.6.28.7.tar.bz2`

注意：linux-2.6.28.7/ 所在的全路径中不能有空格，否则不能 `make xconfig`。

以下两种全路径都是不对的：

`/home/TA/bad folder/linux-2.6.28.7/`

`/home/TA/linux kernel 2.6.28.7/`

- 然后使用以下命令，指定 2.6.28.7 内核使用电脑上原内核的配置，并启动图形界面配置程序。

```
cd linux-2.6.28.7/ (请根据实际情况做改变)
```

```
cp /boot/config-2.6.24-23-generic .config (请根据实际情况做改变)
```

```
make xconfig
```

注意：

1. 前两条命令不要照抄。第一条根据你下载的内核的版本而改变。
2. 第二条命令，要根据你电脑上的最高版本的 `/boot/config-2.6.xxx` 而变。例如你电脑上除了 `/boot/config-2.6.24-23-generic` 外还有 `/boot/config-2.6.27.6`，那么你执行

```
cp /boot/config-2.6.27.6 .config
```

3. `make xconfig` 命令启动一个基于 Qt GUI Library (Threaded runtime version) 的图形界面。你的电脑上必须装有 `libqt3-mt` 以及 `libqt3-mt-dev` 软件包，才能启动 `make xconfig`。在 Ubuntu 操作系统中，使用以下命令安装这些软件包。

```
sudo apt-get install libqt3-mt-dev -y
```

在 Fedora 操作系统中，使用以下命令安装这些软件包。

```
su -c "yum install qt3-devel libXi-devel -y"
```

在图形界面中，选择“File”菜单的“Save”，关闭图形界面。注意：如你使用的电脑是交大计算中心一楼的电脑，那么在关闭图形界面之前还要调整一些选项。（待补）

- 使用以下命令编译、安装内核、生成 initramfs 镜像。

使用 Ubuntu 的同学执行

```
make
```

```
sudo make modules_install
```

```
sudo make install
```

```
sudo mkinitramfs -o /boot/initrd.img-2.6.28.7 2.6.28.7
```

注意：最后一条命令不要照抄。要根据你下载的内核的版本而改变。若你电脑上没有 `mkinitramfs` 程序，则请安装 `initramfs-tools` 软件包。安装方法：

```
sudo apt-get install initramfs-tools
```

initramfs 镜像，即 `/boot/initrd.img-2.6.xx.x`，是用 `gzip` 算法压缩的 `cpio` 格式的文档。你的电脑启动时，Linux 内核将 `initramfs` 镜像解压缩并作为根文件系统。推荐对此感兴趣的同学阅读：

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

使用 Fedora 的同学执行

```
make  
  
su -c "make modules_install"  
  
su -c "make install"
```

- 使用 Ubuntu 的同学还需要修改 GRUB (GRand Unified Bootloader)。执行

```
sudo gedit /boot/grub/menu.lst
```

找到以 “title” 开始的行，按照原版 Linux 内核对应的内容，写上新版 Linux 内核。

例如使用 Ubuntu 的同学，原内核对应的行为：

```
title      Ubuntu 8.04.2, kernel 2.6.27.6  
  
root       (hd0,0)  
  
kernel     /boot/vmlinuz-2.6.27.6 root=UUID=7c503741-0611-4cd8-80ef-937aa6c202bb ro  
quiet splash locale=zh_CN    (注意, "kernel" 一行 和 本行 是同一行! 不要写错! )  
  
initrd     /boot/initrd.img-2.6.27.6  
  
quiet
```

则将这五行改为

```
title      Ubuntu 8.04.2, kernel 2.6.28.7  
  
root       (hd0,0)  
  
kernel     /boot/vmlinuz-2.6.28.7 root=UUID=7c503741-0611-4cd8-80ef-937aa6c202bb ro  
quiet splash locale=zh_CN  
  
initrd     /boot/initrd.img-2.6.28.7  
  
quiet  
  
title      Ubuntu 8.04.2, kernel 2.6.27.6  
  
root       (hd0,0)  
  
kernel     /boot/vmlinuz-2.6.27.6 root=UUID=7c503741-0611-4cd8-80ef-937aa6c202bb ro  
quiet splash locale=zh_CN  
  
initrd     /boot/initrd.img-2.6.27.6  
  
quiet
```

欲进一步了解GRUB，可阅读 <http://www.gnu.org/software/grub/manual/grub.html>

使用 Fedora 的同学不需要改 /boot/grub/menu.lst，不过 GRUB 默认启动旧版内核。欲令 GRUB 默认启动新版内核，请将 /boot/grub/menu.lst 中的 **default=1** 改为 **default=0**。

- 重新启动，在 GRUB 菜单中选择 kernel 2.6.28.7，即启动了新内核。也可以用以下命令查看运行中的内核版本

```
uname -r
```

- **注意：**不要删去 linux-2.6.28.7 目录(也就是编译内核用的目录)。因为 Linux 内核安装后，在 /lib/modules/内核版本号/ 中会创建 build 和 source 两个软链接。删去、重命名、移动位置等，都会破坏软链接，导致你不能完成作业。

§ 9 实验参考：使用计算中心的计算机进行实验的注意事项

同学们请穿鞋套进入计算中心的机房。一楼值班处有售鞋套，0.5 元一副。

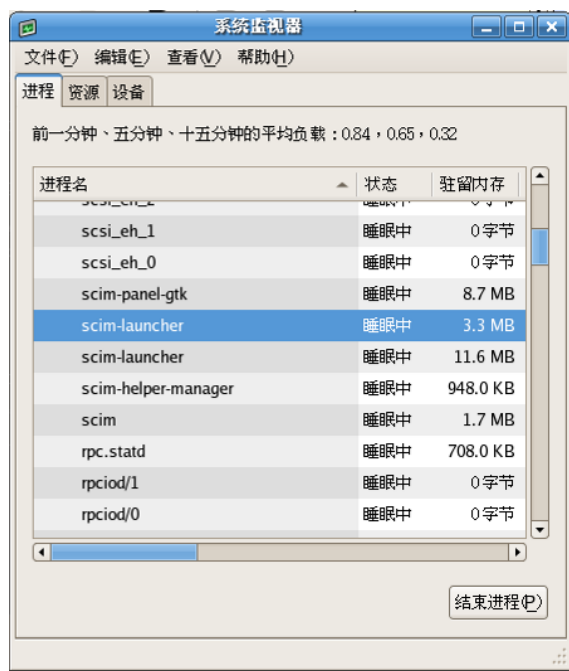
计算中心的计算机上 Linux 发行版为 Fedora Core 5。在教室墙壁上写有用户名和密码。请向老师询问 root 密码。

计算中心预装了 gcc、make 3.8.0。因此无需 yum install 即可使用 make xconfig。

计算中心 gcc 为 4.1.0，用此版本编译内核时有警告：

#warning gcc-4.1.0 is known to miscompile the kernel. A different compiler version is recommended. 本应当使用最新版gcc来编译内核，不过由于我校已不提供Fedora Core 5的软件仓库，因此无法安装新版gcc。在实验时，忽略此警告。同学们使用自己电脑编译内核时，务必使用新版gcc。

计算机中心的 scim 输入法是旧版的，有 BUG。因此你经常会不能输入任何字符。解法是，打开“桌面”菜单，选择“管理”，选择“系统监视器”，在“进程”一栏内找到两个名为 scim-launcher 的进程，在内存占用少的那个 scim-launcher 进程上点击鼠标右键，选择“杀死进程”。然后就能输入字符了。需要再次输入中文时，打开一个终端，执行 scim。：)

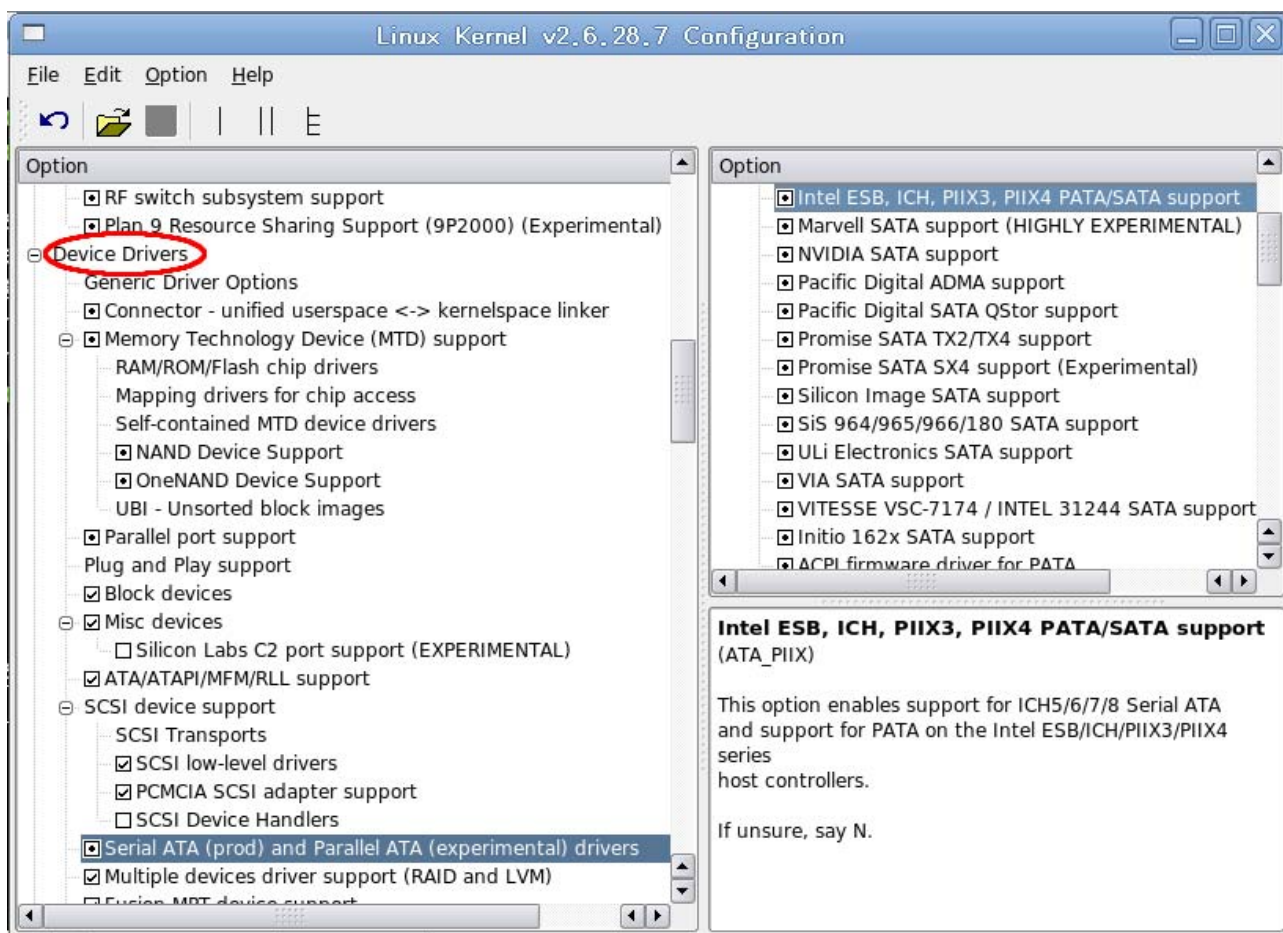


计算中心原 Linux 内核为 2.6.15 版，它的配置文件与最新版内核配置文件有较大不同。新版内核使用 2.6.15 的 /boot/config 将不能挂载硬盘。我们需要将 ata_piix 模块加入内核。

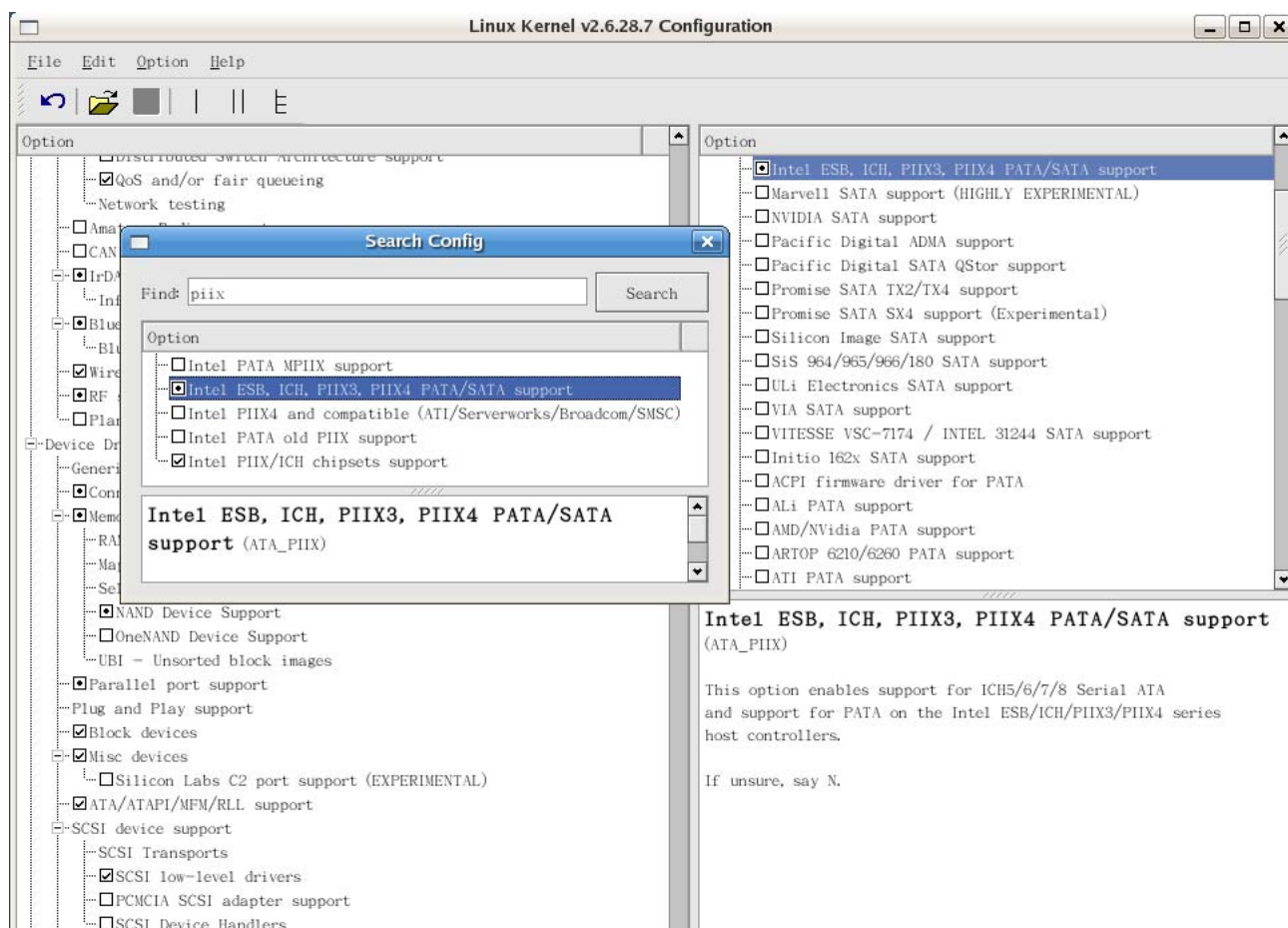
如下图所示，执行 make xconfig 后，

先在左侧树形图中选上“Device Drivers”大类(红色椭圆)下的“Serial ATA (prod) and Parallel ATA

(experimental) drivers (ATA)”项(左侧被选中的项)。



再选择Edit菜单的Find，输入piix，点Search。



对于 2.6.28.7 版内核，应当搜到以下五项：

Intel PATA MPIIX support

Intel ESB, ICH, PIIX3, PIIX4 PATA/SATA support (ATA_PIIX)

Intel PIIX4 and compatible (ATI/Serverworks/Broadcom/SMSC) (I2C_PIIX4)

Intel PATA old PIIX support (PATA_OLDPIIX)

Intel PIIX/ICH chipsets support (BLK_DEV_PIIX)

如图所示，把 Intel ESB, ICH, PIIX3, PIIX4 PATA/SATA support (ATA_PIIX) 打上点。之后选“File”“Save”保存配置，关闭图形界面，参考上文“使用以下命令编译、安装内核、生成initramfs镜像”所述继续进行实验。

另外，由于我校从 www.kernel.org 下载 Linux 内核需要至少十分钟。为节约时间，在计算中心做实验前，应事先下载好 Linux 内核，用 U 盘等带到计算中心。

§ 10 实验参考：内核模块的 Makefile 样例

假设你的模块代码写在 homework1.c 中，那么推荐你创建一个名为 Makefile 的文件，内容如下

```
obj-m := homework1.o

KDIR := /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)
```

```
all:

    make -C $(KDIR) M=$(PWD) modules

clean:

    rm *.o *.ko *.mod.c Module.symvers modules.order -f
```

注意：make 和 rm 开头的两行，开头的空白是一个制表符（Tab），不是四个/八个空格！

将 Makefile 与 homework1.c 置于相同文件夹内，执行

```
make
```

将编译你的模块，产生 homework1.ko homework1.mod.c homework1.mod.o homework1.o Module.symvers 等文件。

而执行

```
make clean
```

将只保留 Makefile 与 homework1.c，删去其它文件。

要使用一个 Makefile 编译多个文件，如 homework1.c homework2.c homework3.c，则可将第一行改为

```
obj-m := homework1.o homework2.o homework3.o
```

§ 11 实验参考：一个简单的内核模块

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

static int __init hello_init(void){
    printk("<3>Greeting from a linux kernel module.\n");
    return 0;
}

static void __exit hello_exit(void){
    printk("<3>Bye.\n");
}

module_init(hello_init);
```



```
module_exit(hello_exit);
```

```
MODULE_LICENSE("GPL");
```

MODULE_AUTHOR、MODULE_DESCRIPTION、MODULE_VERSION、MODULE_PARM_DESC 等所写的信息，用

```
modinfo homework1.ko
```

可以看到。

§ 12 实验参考：printk

printk() 函数用于打印一条内核消息。其用法与标准 C 语言中的 printf 函数相同，不过 printk 的消息的前三个字符可以表示消息级别(当且仅当第一个字符为<，第三个字符为>，第二个字符为 0~7)。

例如：

```
printk( "<3>A Kernel Message. %s\n", "Hello world. ");
```

在<linux/kernel.h>中有如下定义：

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

“<0>”表示最重要，“<7>”表示最不重要。

若消息前三个字符不满足第一节末尾括号中所述的条件，则 printk 在输出消息之前，先输出三个字符：< default_message_loglevel 和 >。

default_message_loglevel 可通过

```
cat /proc/sys/kernel/printk
```

得到。

```
4      4      1      7
```

这四个数依次是内核代码中的 console_loglevel, default_message_loglevel, minimum_console_level 和 default_console_loglevel。

根据默认的 Linux 内核配置，printk 输出消息之前会输出时间，时间的单位为秒，表示电脑最近一次启动到 printk 时的时间。

§ 13 实验参考: module_param 和 module_param_array

● module_param

1. 语法: `module_param(name, type, perm)`
2. 其中, `name` 为模块参数的名称。
3. `type` 为 `byte`(相当于 `unsigned char`), `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp`(相当于 `char *`, 不超过 1024 字节的字符串), `bool`(相当于 `int`), `invbool`(相当于 `int`)。
若参数为 `bool` 类型, 那么设置参数值为 'y', 'Y', '1' 表示 `true`, 'n', 'N', '0' 表示 `false`。
若参数为 `invbool` 类型, 那么 'y', 'Y', '1' 表示 `false`, 'n', 'N', '0' 表示 `true`。
4. `perm` 表示权限。若 `perm` 不为 0, 则模块装载后, 会在 `/sys/module/模块名/parameters/` 目录中产生对应于每个模块参数的文件。`perm` 即为此文件的权限。
`perm` 的取值可参考 `linux-2.6.xx.x/include/linux/stat.h` 中的定义

```
#define S_IRUSR 00400    文件所有者可读
#define S_IWUSR 00200    文件所有者可写
#define S_IXUSR 00100    文件所有者可执行
#define S_IRGRP 00040    与文件所有者同组的用户可读
#define S_IWGRP 00020
#define S_IXGRP 00010
#define S_IROTH 00004    与文件所有者不同组的用户可读
#define S_IWOTH 00002
#define S_IXOTH 00001
```

在 C 语言中, 将以上权限用 | 操作符连接以得到你想设置的权限。:)

5. 举例:

```
static char * whom=" world" ;

static int howmany=1;

module_param(howmany, int, S_IRUGO);

module_param(whom, charp, S_IRUGO);
```

为模块定义了两个参数。

● module_param_array

1. 语法: `module_param_array(name, type, num, perm)`
2. 其中, `name`, `type` 意义同 `module_param` 中所述。`num` 是整型指针 (`int *`), 模块装载成功后, 数组元素个数会被存于 `*num`。

§ 14 实验参考: create_proc_read_entry 和 create_proc_entry

欲创建 /proc 文件系统文件, 可以使用 create_proc_read_entry 或者 create_proc_entry。
create_proc_read_entry 函数原型为

```
struct proc_dir_entry *create_proc_read_entry(const char *name,  
mode_t mode, struct proc_dir_entry *base, read_proc_t *read_proc, void *data);
```

其中 read_proc_t 的定义是

```
typedef int (read_proc_t)(char *page, char **start, off_t off, int count,  
int *eof, void *data);
```

create_proc_entry 函数原型为

```
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode,  
struct proc_dir_entry *parent)
```

你可以改动 proc_dir_entry 来实现特殊效果, 例如

```
struct proc_dir_entry *base = 0;  
base = create_proc_entry("R", 0444, 0);  
base->read_proc = read_proc;  
base->owner = THIS_MODULE; //有了这一行, proc_dir_entry 在使用时就不能卸载模块了  
base->size = 100; //让 /proc 文件大小非零 =D
```

倪立群发给我代码以说明 proc_dir_entry 的使用。谢谢!

另外, 可用 proc_mkdir 创建目录、用 proc_symlink 创建符号链接。函数原型为:

```
struct proc_dir_entry *proc_mkdir(const char *name,  
struct proc_dir_entry *parent);  
struct proc_dir_entry *proc_symlink(const char *name,  
struct proc_dir_entry *parent, const char *dest);
```

§ 15 附录: 同学提问

[1] typedef 的语法

助教, 以下代码麻烦解释一下, 谢谢!

```
typedef int Myfunc(const char *, const struct stat *, int);
```

```
static Myfunc myfunc;

static int myftw(char *, Myfunc *);
```

回答:

```
typedef int Myfunc(const char *, const struct stat *, int);
```

定义了一种名为Myfunc的类型，用这种类型定义的都是函数。

```
static Myfunc myfunc; 相当于 static int myfunc(const char *, const struct stat *, int);
```

但是这样写是不合语法的: `static Myfunc myfunc{}`

```
static int myftw(char *, Myfunc *); 相当于 static int myftw(char *, int (*)(const char *, const struct
stat *, int));
```

请参考以下例子，注意example3函数声明中参数的写法，并思考：example2和example3的参数是相同类型的吗？

```
#include <stdio.h>

typedef int Myfunc(int); //定义了一种名为 Myfunc 的类型，用这种类型定义的都是函数。

typedef int (*point_to_Myfunc)(int); //定义了一种名为 point_to_Myfunc 的类型，用这种类型
定义的都是函数指针。

static Myfunc example; //函数声明，等价于 static int example(int);

static int example(int a){ //函数定义
}

static void example2(Myfunc * f){
}

static void example3(int (*)(int));

static void example3(int (*point_to_Myfunc)(int)){
}

int main(){
    point_to_Myfunc f = &example; //相当于 point_to_Myfunc f = example;

    example2(f); example3(f);

    return 0;
}
```

[2] 内核目录全路径中有空格导致不能 make xconfig

助教，make xconfig 有以下错误，请问是怎么回事？

```
make xconfig

Makefile:303: /home/me/Linux: No such file or directory
```

```
Makefile:303: Kernel/linux-2.6.28.7/scripts/Kbuild.include: No such file or directory
Makefile:439: /home/me/Linux: No such file or directory
Makefile:439: Kernel/linux-2.6.28.7/arch/x86/Makefile: No such file or directory
make: *** No rule to make target `Kernel/linux-2.6.28.7/arch/x86/Makefile'. Stop.
```

回答:

linux-2.6.28.7/ 所在的全路径中不能有空格。将 `/home/me/Linux Kernel/linux-2.6.28.7/` 更名为 `/home/me/Linux_Kernel/linux-2.6.28.7/` 后, `make xconfig` 就不会出错了。

[3]init.h module.h kernel.h 有什么作用?

助教,

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

分别有什么作用?

回答:

linux/init.h 定义了 `__init`、`__exit`、`module_init`、`module_exit` 宏。

linux/module.h 定义了 `MODULE_LICENSE` 宏。

linux/kernel.h 里有很多常用函数的内核 API 的函数原型。如定义了 `KERN_DEBUG` 宏。

本手册举的模块例子, 不包含 `linux/init.h` 和 `linux/kernel.h` 也能编译通过。

[4]在 Debian 4.1.2 上启动新内核的办法

助教, 这是在 Debian 4.1.2 (Lenny) 上启动新内核的办法:

启动新内核时出现以下错误并死机。

```
Begin: Mounting root file system. Begin:Running /scripts/local-top.....Done
Begin: Waiting for root file system, [3.896125] clock tsc unstable
```

将 `/boot/grub/menu.lst` 中的内核启动项由 `/dev/hda5` 修改为 `/dev/sda5` 后, 启动正常

```
title      Debian GNU/Linux, kernel 2.6.28.7
root       (hd0,4)
kernel     /boot/vmlinuz-2.6.28.7 root=/dev/sda5 ro
initrd     /boot/initrd.img-2.6.28.7
```

参见 http://www.debian.org/releases/lenny/i386/release-notes/ch-upgrading.zh_TW.html#how-to-recover (翁文川 提供)

[5]应用程序正在使用 yum 锁，怎么办？

助教，我执行 `su -c "yum install gcc -y"` 时出错

锁已被用于 `/var/run/yum.pid`: 一个 PID 为 3287 的应用程序正在使用 yum 锁；等待其退出
怎么办？

回答：执行 `su -c "/etc/init.d/yum-updatesd stop"` 或者 `su -c "rm /var/run/yum.pid -f" =)`

[6]module_param_string 和 module_param(charp)

助教，

```
static char S[10] = "String";

static char *PS = 0;

//module_param(S, charp, 0444); //不行

module_param(PS, charp, 0444); //行
```

为什么？

回答：

内核这样处理 charp 类型的模块参数：

执行 `sudo insmod param.ko PS=some_string` 后，内核分配一块内存保存 "some_string" 然后将 PS 的值改为此内存地址。

注意：内核分配的内存块为 "some_string" 的长度+4 字节，所以建议同学们，改写 *PS 时不要超出内存块。
欲将上述数组 S 作为模块参数，可使用

```
static char S[100]="String";

module_param_string(S, S, 100, 0444);
```

第一个 S 是参数名 (insmod 用的)，第二个 S 是代码中的数组的名字，100 是数组大小 (单位是字符)

[7]read_proc 函数被执行两次的两种原因

助教，为什么我用 `cat /proc/homework3_proc_file`，`read_proc` 函数被执行了两次？

```
int read_proc(char *page, char **start, off_t offset, int count, int *eof, void *data){
    int len = 0;
    len += sprintf(page+len, "Twice\n");
    *eof = 1;
    return len;
}
```

回答：

在 `fs/proc/generic.c` 的 `proc_file_read` 函数中，第一个 while 循环调用模块的 `read_proc` 函数。while 循环的条件是 `"(nbytes > 0) && !eof"`。这说明 *eof 取非零值即令模块中的 `read_proc` 函数不再被调用。

“read_proc 函数执行两次”是因为“cat /proc/homework3_proc_file”做了两次 read 系统调用。在 T. Granlund 和 R. Stallman 编写的 coreutils-6.10 里，cat 是这样实现的：
(参见 src/cat.c 和 lib/safe-read.c。 <http://www.gnu.org/software/coreutils/>)

```
for (;;) {  
    ... ..  
    n_read = safe_read (input_desc, inbuf, insize);  
    if (n_read == SAFE_READ_ERROR) {  
        ... .. (结束循环)  
    }  
    if (n_read == 0) {  
        ... .. (结束循环)  
    }  
    ... .. (不结束循环)  
}
```

safe_read 只做一次 read(2) 系统调用。(此函数这样取名，可能是因为它解决了 Tru64 5.1 的一个 BUG =)
据代码分析，cat 会做两次 read(2) 系统调用。

用 fread(buffer, 1, 2000, homework3_proc_file) 会调用 read_proc 两次。而 read(homework3_proc_file, buffer, 2000) 只调用 read_proc 一次。这是因为 glibc 的 fread 包装了 read 系统调用，读指定数量的字符或者读到 eof 才结束。(朱轶)

梁健怡 翁文川 同学将他们的作业程序发给我了以重现“执行两次”问题。梁健怡也写了读/proc 的程序。谢谢！

进程管理与对称多处理器

§ 1 实验内容以及完成效果

作业一

记录进程被调度到 CPU 上执行的总次数。在 `task_struct` 结构中增加一个名为 `ctx` 的变量。每次进程被调度到 CPU 上执行时，增加 `ctx` 的值。建立 `/proc/进程号/ctx` 文件，读此文件能读到 `ctx` 的值。

● 详细说明(张衍民):

1. `ctx` 添加为 `task_struct` 的第一项或者为最后一项，都是正确的。汇编代码使用 `task_struct` 的成员时，成员的地址偏移量是在编译时动态生成的。为了提高效率，`ctx` 应放在与调度相关的成员附近。
2. `ctx` 的初始值是 0。进程通过“clone”系统调用产生的进程/线程的 `ctx` 是 0。
3. 在 `/proc/[number]/ctx` 输出单个进程的调度次数比较好，输出线程组的调度次数总和也可以，但那样占用较多系统处理时间。
4. 也可以生成 `/proc/[number]/task/[number]/ctx` 文件，通过它输出线程的调度次数。
5. 对第一个进程，不用单独处理 `ctx` 的初始化。

完成效果

将以下代码写入文件 `block.c`

```
#include <stdio.h>

int main() {
    while(1) getchar();

    return 0;
}
```

编译成名为 `block` 的程序

```
gcc block.c -o block -Wall
```

打开一个“终端”，执行

```
./block
```

打开另一个“终端”，执行

```
ps -e | grep block

51 ?          00:00:00 kblockd/0 (这两行是第一行命令的输出)
7711 pts/0    00:00:00 block
```


执行

```
cd /proc/7711  
  
cat ctx  
  
5855
```

回到上一个终端，敲 a，回车；
回到第二个终端，执行

```
cat ctx  
  
5856
```

实验结束，执行

```
kill -9 7711 (此命令不要照抄！要根据 block 进程的 PID 做改变)
```

作业二

记录哪个函数调用了 `schedule` 和 `schedule_timeout`，记录分别调用了 `schedule` 和 `schedule_timeout` 多少次。

下文中称调用 `schedule` 和 `schedule_timeout` 的函数为 `caller`，称 `schedule` 和 `schedule_timeout` 为 `callee`。

建立 `/proc/ctxsw` 和 `/proc/ctxsw_status` 文件。

读 `/proc/ctxsw` 能读到 `caller`、`callee` 的信息，信息包含很多行，每行格式为

```
caller_ip  callee_ip  count  
(十六进制) (十六进制) (十进制)
```

这里 `caller_ip`，`callee_ip` 通过 Intel 80x86 CPU 的 IP(指令指针)寄存器的值来确定。等于 IP 寄存器的值，减去内核代码段的起始地址，再除以 4(同学可将 4 改为别的数)。

读 `/proc/ctxsw_status` 能读到

```
Not investigating.
```

或者

```
Investigating all processes.
```

或者

```
Investigating process <进程号>.
```

`/proc/ctxsw` 文件可以写。

- 写入字符串 “r”，可清空 `caller`、`callee`、调用次数记录。
- 写入字符串 “1”，开始记录，记录所有进程。
- 写入字符串 “1 pid=整数 i”，开始记录，只记录 PID(进程号)等于整数 i 的进程。
- 写入字符串 “0”，停止记录。记录不要清空。

本作业只记录大于等于 `_stext`, 小于 `_etext` 的 `caller_ip(_stext` 和 `_etext` 的含义, 请参考下文“Linux 内核代码段的起始和终止地址”一节)。不记录模块的函数直接调 `schedule` 的 `callerip`, 因为模块(.ko) 连接到内核时, 它的各函数的地址是不固定的。

由于 `callee` 只有两个: 即 `schedule` 和 `schedule_timeout`。所以可以用两个数组 `A1` 和 `A2` (数组元素为 `callerip` 和 `count`), `A1` 记录 `schedule` 的 `caller`, `A2` 记录 `schedule_timeout` 的 `caller`。但是考虑扩展性, 如果我们要记录更多的函数, 那么需要添加数组 `A3`、`A4`... 因此用一个简单的 Hash 表完成作业较好, Hash 表的 Key 是 `callerip` 和 `calleeip`, 值是调用次数。

完成效果

将以下代码写入 `transform.c`

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/gmon_out.h>

#define GMON_SIZE 8*1024*1024

int main(int argc, char* argv[]) {

    char *gmon_out, *buf;

    FILE *in, *out;

    gmon_out = (char*) malloc(GMON_SIZE);

    /* gmon_hdr */

    *(struct gmon_hdr*)gmon_out = (struct gmon_hdr){

        .cookie=GMON_MAGIC,

        .version={GMON_VERSION, 0, 0, 0},

        .spare={0},

    };

    buf = gmon_out + sizeof(struct gmon_hdr);

    /* gmon_hist_hdr */

    *buf++ = GMON_TAG_TIME_HIST;

    /* base pc address of sample buffer = 0 */
```

```
*(char**)buf = 0;

buf += sizeof(char *);

/* max pc address of sampled buffer = ~0 */
*(char**)buf = (char *)-1;

buf += sizeof(char *);

/* size of sample buffer = max */
*(int *) buf = 0;

buf += 4;

/* profiling clock rate */
*(int *) buf = 1024;

buf += 4;

/* phys. dim. */
strncpy(buf, "seconds", 15);

buf += 15;

/* 's' for "seconds" */
*buf++='s' ;

in = fopen(argv[1], "r");

int num_record = 0;

char *from_pc, *self_pc;

int count;

while (fscanf(in, "%p %p %d\n", &from_pc, &self_pc, &count) == 3) {

    num_record ++;

    *buf++ = GMON_TAG_CG_ARC;

    memcpy(buf, &from_pc, sizeof(char*)); buf += sizeof(char*);

    memcpy(buf, &self_pc, sizeof(char*)); buf += sizeof(char*);

    memcpy(buf, &count, 4); buf += 4;

}

out = fopen(argv[2], "w");

fwrite(gmon_out, buf-gmon_out, 1, out);
```

```
printf("%d records transformed.\n", num_record);  
  
return 0;  
  
}
```

编译成名为 transform 的程序

```
gcc transform.c -o transform -Wall
```

执行(粗体字是命令, 非粗体字是输出)

```
ls /proc/ctxsw*  
  
/proc/ctxsw /proc/ctxsw_status  
  
cat /proc/ctxsw_status  
  
Not investigating.  
  
./block & (这是作业一中编译的程序)  
  
[1] 8161  
  
[1]+ Stopped ./block  
  
echo "1 pid=8161" > /proc/ctxsw  
  
cat /proc/ctxsw_status  
  
Investigating process 8161.  
  
fg  
  
./block  
  
a (按键盘上的 A 键, 并回车)  
  
^Z (按 Ctrl Z )  
  
[1]+ Stopped ./block  
  
bg  
  
[1]+ ./block &  
  
echo "0" > /proc/ctxsw  
  
cat /proc/ctxsw_status  
  
Not investigating.  
  
cat /proc/ctxsw  
  
ffffffff80247420 ffffffff80499928 2
```

```

ffffffff803b146c ffffffff8049a364 2
ffffffff8049a3f8 ffffffff80499928 2

./transform /proc/ctxsw ./output
3 records transformed.

gprof linux-2.6.28.7/vmlinux ./output

... .. (省略了部分输出)

%   cumulative   self              self   total
time   seconds  seconds    calls  Ts/call  Ts/call  name
0.00    0.00    0.00        4    0.00    0.00  thread_return
0.00    0.00    0.00        2    0.00    0.00  schedule_timeout

... ..

Call graph (explanation follows)

granularity: each sample hit covers 0 byte(s) no time propagated

index % time    self  children    called    name
-----
          0.00   0.00      2/4      do_signal_stop [4071]
          0.00   0.00      2/4      schedule_timeout [2]
[1]      0.0    0.00   0.00      4        thread_return [1]
-----
          0.00   0.00      2/2      n_tty_read [7705]
[2]      0.0    0.00   0.00      2        schedule_timeout [2]
          0.00   0.00      2/4      thread_return [1]
-----
... ..

Index by function name

[2] schedule_timeout    [1] thread_return

echo "r">/proc/ctxsw

```

```
cat /proc/ctxsw
```

(没有输出)

§ 2 实验参考：阅读内核代码

推荐两种阅读内核代码的方式：1. KScope 2. VIM & Ctags。

- KScope (<http://kscope.sourceforge.net/>)

使用 Ubuntu 的同学这样安装 KScope：

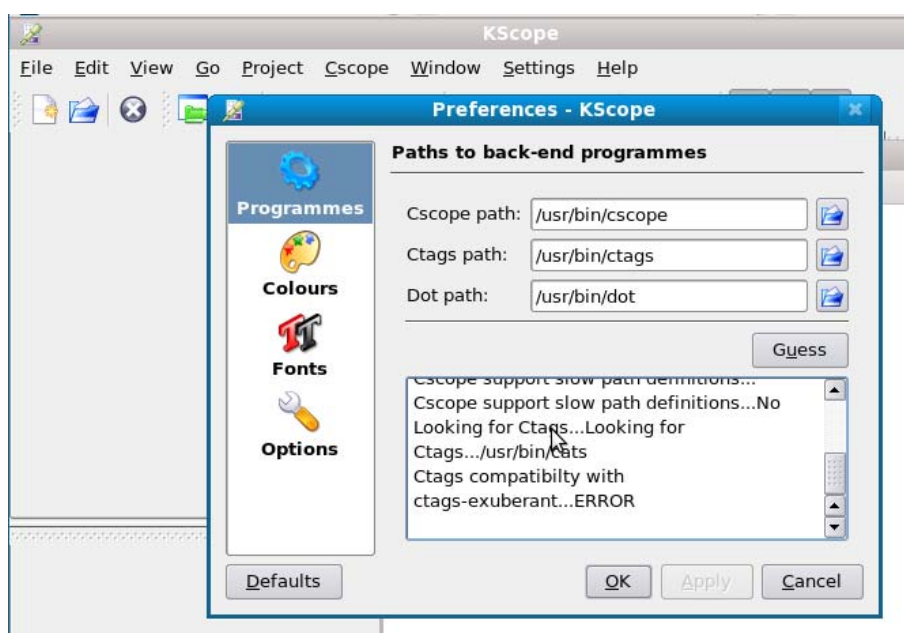
```
sudo apt-get install kscope -y
```

使用 Fedora 的同学这样安装：

```
su -c "yum install kscope -y"
```

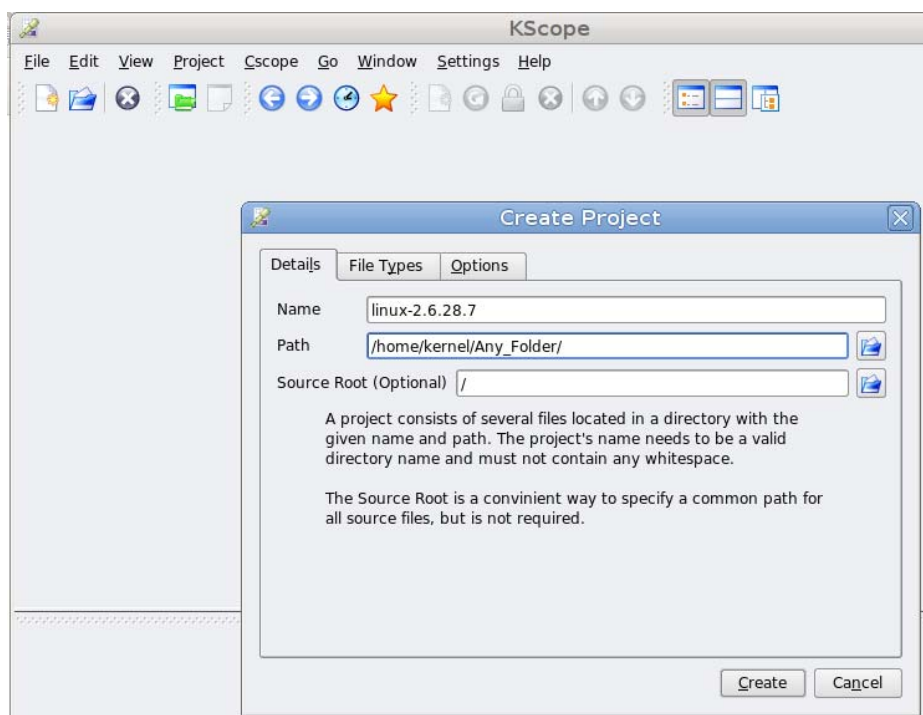
装好后，打开一个终端，执行 kscope。

如果出现这个窗口，请不要点“Guess”（因为有BUG），而是按照图示填写。



点击“Project”菜单的“New Project...”

在 Path 栏，填一个已经建立好的空的目录（不要填写内核的目录/home/kernel/linux-2.6.28.7）。此目录将用于保存 cscope.proj 等文件，它们包含与项目相关的信息。



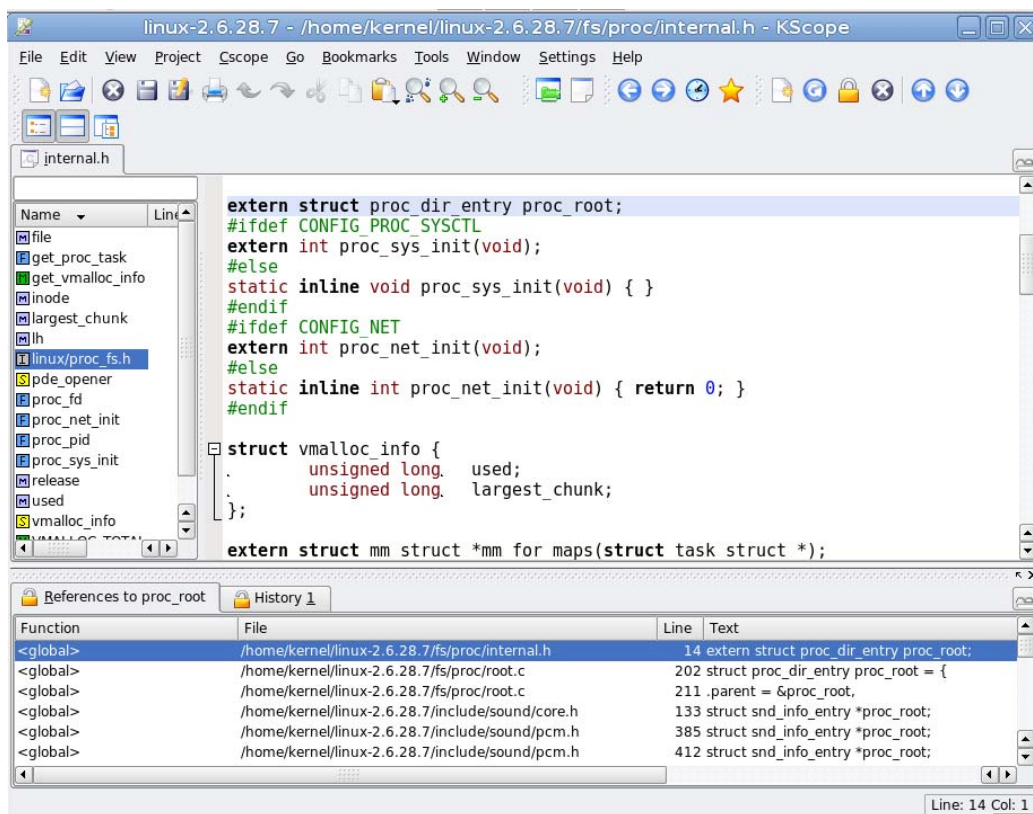
File Types里加上 .S(汇编语言的代码)。“cc”也可以加。不过 2.6.28.7 内核里只有一个 .cc 文件。Options 选择 Kernel project(-k)。点击 “Create” 按钮。

在接下来出现的 Project Files 对话框中，点 Tree... 选择内核代码所在的目录，如选择 /home/kernel/linux-2.6.28.7。

注意：/home/kernel/linux-2.6.28.7 全路径中不能有中文。

等待 “KScope - Please wait while KScope builds the database” 对话框自动关闭后，就可以开始阅读代码了。

例如，点 “Cscope” 菜单的 “References” 项，输入 proc_root，就能查到所有引用。



- VIM(Vi Improved <http://www.vim.org/>) & Ctags(<http://ctags.sourceforge.net/>)
关于 VIM 的用法, 由于篇幅限制, 这里不详细叙述了。以下资料很有帮助:
 - VIM Cheat Sheet <http://www.viemu.com/vi-vim-cheat-sheet.gif>
 - VIM Commands <http://fprintf.net/vimCheatSheet.html>
 - VIM Quick Reference Card
<http://www.digilife.be/quickreferences/QRC/VIM%20Quick%20Reference%20Card.pdf>

使用 Ubuntu 的同学这样安装 VIM 和 Ctags:

```
sudo apt-get install vim exuberant-ctags -y
```

使用 Fedora 的同学这样安装:

```
su -c "yum install vim-X11 ctags -y"
```

装好后, 打开终端, 执行

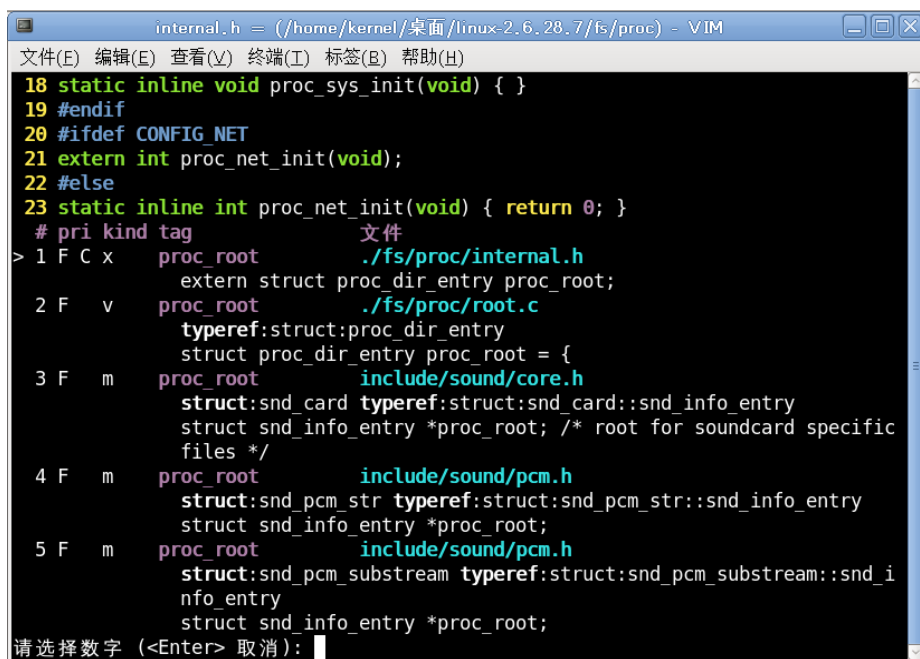
```
cd /home/kernel/linux-2.6.28.7 (这条命令不要照抄, 按照你自己电脑上内核的目录做更改)
make tags (或者 ctags -R *)
```

这样就生成了名为 `tags` 的文件。有了这个文件, 若要立即打开 `proc_root` 的定义, 可执行

```
vim -t proc_root
```

`proc_root` 共有 5 处定义。

输入:ts 并回车, 可以选择跳转到哪一个定义处。



```

internal.h = (/home/kernel/桌面/linux-2.6.28.7/fs/proc) - VIM
文件(E) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
18 static inline void proc_sys_init(void) { }
19 #endif
20 #ifdef CONFIG_NET
21 extern int proc_net_init(void);
22 #else
23 static inline int proc_net_init(void) { return 0; }
24 #endif
25 #define proc_root \
26     extern struct proc_dir_entry proc_root;
27     struct proc_dir_entry proc_root = {
28         .name = "proc",
29         .small = "proc",
30         .proc_fops = &proc_fops,
31         .data = 0,
32     };
33     #include <sound/core.h>
34     struct snd_card typeref:struct:snd_card::snd_info_entry
35     struct snd_info_entry *proc_root; /* root for soundcard specific
36     files */
37     #include <sound/pcm.h>
38     struct snd_pcm_str typeref:struct:snd_pcm_str::snd_info_entry
39     struct snd_info_entry *proc_root;
40     #include <sound/pcm.h>
41     struct snd_pcm_substream typeref:struct:snd_pcm_substream::snd_i
42     nfo_entry
43     struct snd_info_entry *proc_root;
44
45 请选择数字 (<Enter> 取消):

```

另外，将光标移动到变量名/函数名上，按 **Ctrl]**，效果和执行命令 **vim -t 名** 相同。

阅读完 `proc_root` 的定义后，按 **Ctrl+T** 将返回原来的位置。

执行 **:ts /proc_root** 可列出所有名称中包含“`proc_root`”的变量/函数。

§ 3 实验参考：将你对内核的改动做成补丁

在输出成补丁前，我们需要先删去配置、删去所有编译产生的文件等。这不宜直接在当前内核目录中操作，否则下一次编译内核时需要重新配置和编译大量源文件。推荐同学们将内核目录复制出一副本，并用副本做补丁。

假设内核目录为 `/home/kernel/linux-2.6.28.7`。执行命令

```

cd /home/kernel

cp linux-2.6.28.7 linux-2.6.28.7.myhomework -r

cd linux-2.6.28.7.myhomework

make mrproper

```

将删去配置、所有编译产生的文件。

再将 `linux-2.6.28.7.tar.bz2` 解压到 `/home/kernel/linux-2.6.28.7.origin`

执行

```
diff -Nrup linux-2.6.28.7.origin linux-2.6.28.7.myhomework > myhomework
```

就生成了一个名为 `myhomework` 的补丁。

注意：`linux-2.6.28.7.origin linux-2.6.28.7.myhomework` 不要写错顺序！

补充：如果同学的硬盘上没有多余的空间来复制一份 `linux-2.6.28.7`，或者不想花时间复制、`make mrproper`，那么将 `linux-2.6.28.7.tar.bz2` 解压为 `linux-2.6.28.7.myhomework`，将你在 `linux-2.6.28.7/` 中改过的文件逐个复制到 `linux-2.6.28.7.myhomework`，再用 `diff` 生成补丁。

请不要将内核目录复制到 NTFS(New Technology File System)或者 FAT(File Allocation Table)分区上再用 diff 生成补丁。因为 NTFS 或者 FAT 都不能建立软链接。

diff 的各参数的含义是:

- -N 将不存在的文件看作是已经存在的空文件。如果不加这一选项, 当 linux-2.6.28.7.origin 里没有 some_file.c 而 linux-2.6.28.7.myhomework 里有 some_file.c 时, diff 不会输出 some_file.c 的内容。
- -r 递归地输出差异。如果不加 -r, diff 只输出 linux-2.6.28.7.origin linux-2.6.28.7.myhomework 里的首层文件的差异, 不输出 arch block 等子目录里的文件的差异。
- -u 除了输出差异外, 还输出差异所在行的前 3 行和后 3 行代码。
- -p 输出差异是在哪个 C 语言函数里。

注意: 我们不加 -a 参数。当 diff 发现两个二进制文件有差异时, 不会输出差异。加 -a 参数时, diff 将二进制文件也当作文本文件。

§ 4 tgid_base_stuff 和 tid_base_stuff

Linux 中, 线程是轻量级进程。POSIX(Portable Operating System Interface of UniX)规定, 同一进程克隆出的线程的“pid”是相同的, 因此在 Linux 中, 用 task_struct 的 tgid 表示线程的 ID, 领头线程的 tgid 等于 pid, 而其他的线程的 tgid 等于领头线程的 tgid, pid 各不相同。相应地, getpid(2) 系统调用返回 tgid。gettid(2) 系统调用返回 pid。

线程组反映在 /proc/[number]/task 中。/proc/[number]/task 中的每个子目录对应 tgid 为 number 的每个线程。tid_base_stuff 数组用于生成 /proc/[number]/task /[tid]。而 tgid_base_stuff 数组用于生成 /proc/[number]

同学们可以使用以下函数/宏枚举所有进程(参见 include/linux/sched.h)

```
next_task(p)
```

```
do_each_thread
```

```
for_each_process(p)
```

枚举同一组的线程:

```
struct task_struct *next_thread(const struct task_struct *p)
```

```
while_each_thread
```

§ 5 实验参考: schedule 和 schedule_timeout

- schedule 函数(见 kernel/sched.c)负责选择一个进程并将 CPU 分配给该进程, 使该进程替换当前进程。一个进程因为不能立刻获得必需的资源时, schedule 会被调用, 以阻塞此进程。

- schedule_timeout(见 kernel/timer.c)的函数原型为

```
signed long __sched schedule_timeout(signed long timeout);
```

它令当前进程立即睡眠, 经过 timeout 个 jiffies 后再唤醒此进程。如果进程为 TASK_INTERRUPTIBLE 状态, 进程可以提前被一个信号唤醒。根据内核配置的不同, jiffies 等于 1/100, 1/250, 1/300 或者 1/1000 秒。HZ 个 jiffies 表示一秒。HZ 是内核代码中的宏。

- Linux 2.6.x 系列的 O(1)的“完全公平调度算法”由 C. Kolivas 提出



原型，由 I. Monar 维护(右上图为 Kolivas，右下图为 Monar)。

● 欲进一步研究此算法的同学，可阅读内核目录中的 Documentation/scheduler/sched-design-CFS.txt 及相关文献。

§ 6 实验参考：实现一个可写的 proc 文件

请同学参阅实验一的实验参考：create_proc_entry。为函数返回的 proc_dir_entry*指定 write_proc，可以实现可写的 proc 文件。

write_proc 的函数原型为

```
write_proc_t * write_proc;

typedef int (write_proc_t)(struct file *file, const char __user *buffer,
                           unsigned long count, void *data);
```

参数 buffer 由用户程序给出的，建议使用 copy_from_user 将 buffer[] 的内容复制到另一个数组中。

```
unsigned long copy_from_user(void * to, const void __user * from, unsigned long n);
```

此函数试着复制 n 个字节。返回有多少字节没能复制。返回 0 表示复制成功。不能成功复制时，例如复制了 m 字节，有 k 字节不能复制，则将 $to[m+1, \dots, m+k]$ 填充为 ‘\0’。

使用 copy_from_user 而非直接读 buffer[]，是因为 Linux 的内存是分段的。同学们可阅读

The Linux Kernel Module Programming Guide

http://www.linuxtopia.org/online_books/Linux_Kernel_Module_Programming_Guide/x773.html

§ 7 实验参考：__builtin_return_address

这是 GCC (GNU Compiler Collection) 的内置函数。它的原型是

```
void * __builtin_return_address (unsigned int level)
```

__builtin_return_address(0) 返回 “当前函数结束后接着应该执行哪个地址的指令”。

__builtin_return_address(1) 返回 “调用当前函数的函数，结束后接着应该执行哪个地址的指令”。依此类推。

注意：__builtin_return_address(level>=1) 仅用于调试。内核编译通常不将 level>=1 的返回地址放到栈内，所以同学们在本实验中，不要使用 level>=1，否则可能引起 OOPS。

想一想：如何用 __builtin_return_address 取得 callerip 和 calleeip?

§ 8 实验参考：Linux 内核代码段的起始和终止地址

在 include/asm-generic/sections.h 文件中有：

```
extern char _stext[], _etext[];
```

(_stext 和 _etext 的定义，在 arch/x86/kernel/vmlinux_32.lds.S 和 vmlinux_64.lds.S)

_stext 是内核的代码段的开始地址，_etext 是内核的代码段之后的第一个地址。



它们的值可以通过读 `/proc/kallsyms` 得到。

`kallsyms` 文件的内容是内核的所有导出的变量/函数。第一列是变量/函数的地址，第二列是类型，第三列是名称(用 `EXPORT_SYMBOL` 和 `EXPORT_SYMBOL_GPL` 宏可导出变量/函数)。

类型参见以下命令的输出

```
man nm
```

注意，`_stext`、`_etext` 的类型是 `T`，表示它们位于代码段。因此对它们来说，第一列是值。如

```
ffffffff80209000 T _stext
ffffffff804a250d T _etext
```

表示 `_stext=ffffffff80209000`，`_etext=ffffffff804a250d`。

(而 `_text=_stext+sizeof(.text.head)`)

`.text.head` 段大部分是用汇编语言写的启动代码 张衍民)

请同学们阅读理解以下程序，并思考和解释：

此程序为什么有这样的输出？

```
#include <stdio.h>

extern int _etext, _edata, _end;

int x;

int f(void);

int main() {

    printf("_etext=%p _edata=%p _end=%p\n", &_etext, &_edata, &_end);

    printf("main=%p f=%p &x=%p\n", main, f, &x);

    return 0;

}

int f(void) { }
```

输出

```
_etext=0x4005f2 _edata=0x600948 _end=0x600950
main=0x400498 f=0x4004fd &x=0x60094c
```

§ 9 实验参考：获得大于 4M 的内存

- 本实验中同学们需要分配大块内存。这样的分配容易失败，因为随着时间的流逝，Linux 内存趋于碎片化，难于找到连续的大块内存；而且 `alloc_pages` 又有 1024 个页的限制。我们使用以下函数

```
#include <linux/bootmem.h>
```

```
void *alloc_bootmem(unsigned long size); size 是字节数
```

- bootmem 是指 bootmem 内存分配器。它仅用在内核启动阶段，仅为内核分配、保留内存，它在 buddy 内存分配系统初始化前被销毁。

bootmem 内存分配器初始化由 init_bootmem_node 完成。函数调用关系是

```
start_kernel(init/main.c)->setup_arch(arch/x86/kernel/setup.c)
->initmem_init(arch/x86/mm/init_32.c)->setup_bootmem_allocator(arch/x86/mm/init_32.c)
->init_bootmem_node(mm/bootmem.c)
```

bootmem 内存分配器销毁由 free_all_bootmem 完成。函数调用关系是

```
start_kernel(init/main.c)->mem_init(arch/x86/mm/init_32.c)
->free_all_bootmem(mm/bootmem.c)
```

同学们要注意，要在 init_bootmem_node 调用后、free_all_bootmem 调用前使用 alloc_bootmem。

- 有兴趣进一步研究 bootmem 的同学可参考以下文献
1. A.Nayani, M.Gorman and R.Castro, Memory Management in Linux, 2002.
 2. 陈莉君，深入分析Linux内核源代码，
<http://www.kerneltravel.net/kernel-book/深入分析Linux内核源码.html> .
 3. D.Bovet and M.Cesati, Understanding the Linux Kernel, 3rd ed., Chapter 8.

§ 10 实验参考：实现一个可读写的 seq 文件

- 本实验中，/proc/ctxsw 文件内容较多，如果使用实验一的办法做作业，那么每次调用 read_proc 只能输出不超过 4K 字节的信息。将/proc/ctxsw 实现为一个可读写的 seq 文件很简便。
 - 请同学们阅读以下文献：
1. fs/proc/generic.c 代码中的注释 “How to be a proc read function”
 2. Driver porting: The seq_file interface, <http://lwn.net/Articles/22355/> .
 3. R.Dunlap, Linux kernel seq_file HOWTO,
http://www.xenotime.net/linux/doc/seq_file_howto.txt .

§ 11 实验参考：gprof (GNU PROFILER)

请同学们阅读以下文献，了解 gprof (GNU PROFILER) 的原理和使用方法

1. S.Graham, P.Kessler and M.McKusick, An Execution Profiler for Modular Programs.
2. S.Graham, P.Kessler and M.McKusick, gprof: A Call Graph Execution Profiler.
3. J.Fenlason and R.Stallman, GNU gprof,
http://gnu.huihoo.org/gprof-2.9.1/html_node/gprof_toc.html

§ 12 实验参考：Linux 的 pid=0 的进程和 pid=1 的进程

Linux 内核启动后即以 pid=0 的进程 (进程 0) 的身份运行。进程 0 不是由 clone 系统调用产生的，它的 task_struct 为

```
struct task_struct init_task = INIT_TASK(init_task); (arch/x86/kernel/init_task.c)
```

相应地有如下静态变量

```
static struct fs_struct init_fs = INIT_FS;

static struct signal_struct init_signals = INIT_SIGNALS(init_signals);

static struct sighand_struct init_sighand = INIT_SIGHAND(init_sighand);

struct mm_struct init_mm = INIT_MM(init_mm); (以上四个在 arch/x86/kernel/init_task.c)

struct files_struct init_files; (fs/file.c)
```

进程 0 用 clone 系统调用产生 pid=1 的进程(进程 1)，然后进程 0 调用 void cpu_idle(void)，进入无限循环。

进程 1 由 rest_init 函数产生。其代码为 kernel_init 函数。进程 1 通过系统调用，执行文件系统中的 init 程序。

欲进一步了解进程 1 的同学，可阅读以下文献：

1. Linux: Linux Init Process and PC Boot Procedure,
<http://www.yolinux.com/TUTORIALS/LinuxTutorialInitProcess.html>
2. Linux Init and System Initialization
http://www.comptechdoc.org/os/linux/startupman/linux_suinit.html
3. Linux startup process – Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Linux_startup_process

§ 13 附录：同学提问

[8]switch_to

梁健怡发现了这一现象：把“next->ctx++;”写在 schedule() 的“context_switch(rq, prev, next);
/*unlock the rq*/”后，新内核启动就死机。

回答：

context_switch 中的“switch_to(prev, next, prev);”被宏展开为：

```
do {
    unsigned long ebx, ecx, edx, esi, edi;

    asm volatile("pushfl\n\t"        /* save  flags */
                 "pushl %%ebp\n\t"   /* save  EBP  */
                 "movl %%esp, %[prev_sp]\n\t" /* save  ESP  */
                 "movl %[next_sp], %%esp\n\t" /* restore ESP */
                 "movl $1f, %[prev_ip]\n\t" /* save  EIP  */
```

```

    "pushl %[next_ip]\n\t" /* restore EIP */
    "jmp __switch_to\n\t" /* regparm call */
    "1:\n\t"
    "popl %%ebp\n\t" /* restore EBP */
    "popfl\n\t" /* restore flags */
    /* output parameters */
    : [prev_sp] "=m" (prev->thread.sp),
      [prev_ip] "=m" (prev->thread.ip),
      "=a" (last),
      /* clobbered output registers: */
      "=b" (ebx), "=c" (ecx), "=d" (edx), "=S" (esi), "=D" (edi)
      /* input parameters: */
    : [next_sp] "m" (next->thread.sp), [next_ip] "m" (next->thread.ip),
      /* regparm parameters for __switch_to(): */
      [prev] "a" (prev), [next] "d" (next)
    : /* reloaded segment registers */"memory");
} while (0);

```

sp 寄存器的值是栈顶，“movl %[next_sp],%%esp”指令改变 sp 的值，“popl %%ebp”改变了 bp 的值，无论是通过 bp 加偏移引用 next、还是通过 sp 加偏移引用 next，得到的 next 都不是__switch_to 调用之前的 next 了。因此写 next->ctx 导致死机。

请同学们思考：

1. 把“prev->ctx++;”写在 schedule() 的“context_switch(...);”后，能否正常访问？
2. 为什么 switch_to 宏有三个参数而不是两个？

这两个问题的答案在本节末尾。同学们在自己思考前不要阅读答案。

[9] sched_info

助教，在 kernel/sched_stats.h 的 sched_info_arrive 函数中有

```
t->sched_info.pcount++; (t 是 struct task_struct *)
```

这个 pcount 是否是进程被调度到 CPU 上运行的次数？

回答：

是的。参见 kernel/sched_stats.h 的 sched_info_switch 函数。配置内核时选择了“Kernel hacking” -> “Kernel debugging” -> “Collect scheduler statistics” 或选择了“General setup” -> “Export task/process statistics through netlink” -> “Enable per-task delay accounting”，sched_info_arrive 才被执行。

[10]proc_misc.c

助教，proc_misc.c 不见了！在哪生成的/proc/[number]/中的文件？

回答：

Linux 2.6.28.7 内核的 proc 文件系统结构，相比以前版本，有较大变化。由 tgid_base_stuff 数组生成文件/目录的操作，见 proc_pident_readdir 和 proc_pident_lookup 函数。

[11]INF ONE REG 宏有什么区别

助教，fs/proc/base.c 中的 INF、ONE、REG 宏有什么区别？

回答：

阅读并理解以下代码

```
#define INF(NAME, MODE, OTYPE) \
    NOD(NAME, (S_IFREG|(MODE)), \
    NULL, &proc_info_file_operations, \
    { .proc_read = &proc_##OTYPE } )

static const struct file_operations proc_info_file_operations = {
    .read      = proc_info_read, }; (回忆实验一里的 proc_read_entry)

#define ONE(NAME, MODE, OTYPE) \
    NOD(NAME, (S_IFREG|(MODE)), \
    NULL, &proc_single_file_operations, \
    { .proc_show = &proc_##OTYPE } )

static const struct file_operations proc_single_file_operations = {
    .open      = proc_single_open,
    .read      = seq_read,
    .llseek    = seq_lseek,
    .release    = single_release,
}; (回忆本实验中的 seq 文件)
```



```

#define REG(NAME, MODE, OTYPE) \
    NOD(NAME, (S_IFREG|(MODE)), NULL, \
    &proc_##OTYPE##_operations, {})
```

(REG 宏用于自定义 file_operations, 例如)

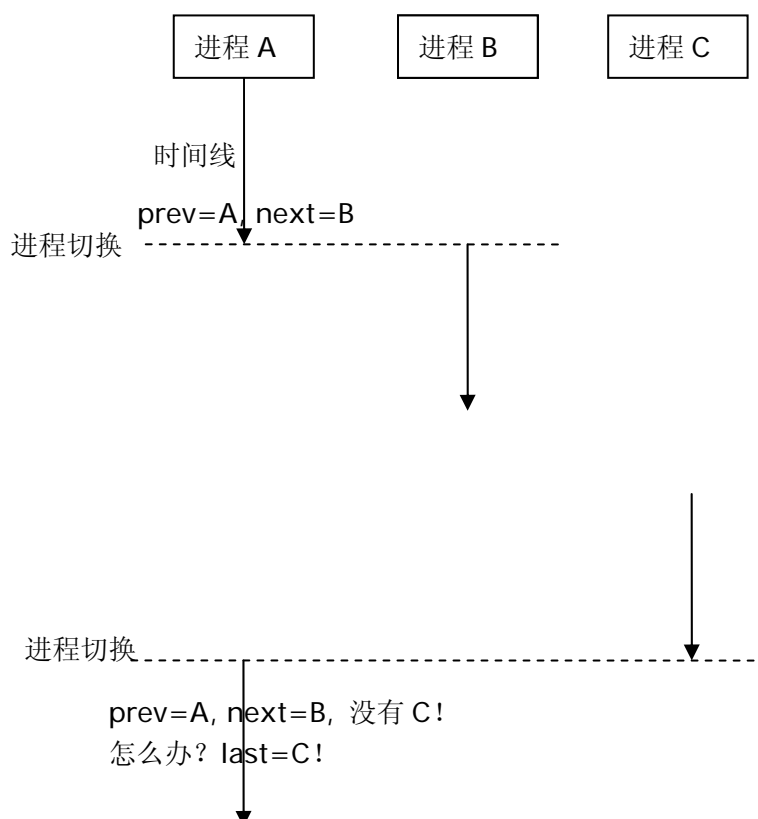
```

REG("current", S_IRUGO|S_IWUGO, pid_attr),

static const struct file_operations proc_pid_attr_operations = {
    .read      = proc_pid_attr_read,
    .write     = proc_pid_attr_write,
};
```

[12] 为什么 switch_to 宏有三个参数？

因为在进程切换中，涉及三个进程（不是两个！）。如下图所示：



这是汇编程序的全注释

```
do {
```

```

unsigned long ebx, ecx, edx, esi, edi;

asm volatile("pushfl\n\t"           把标志寄存器入栈

asm 相当于__asm__; volatile 相当于__volatile__;

volatile 令 GCC 不将这段汇编代码移到别处

    "pushl %%ebp\n\t"           把 EBP(Base Pointer) 寄存器入栈

EBP 寄存器用作基地址, 和偏移量组合来访问堆栈

    "movl %%esp, %[prev_sp]\n\t"   注意, 是将 esp 的值赋给 prev_sp!

    "movl %[next_sp], %%esp\n\t"

    "movl $1f, %[prev_ip]\n\t"   1f 表示向下找, 第一个标号 1: 的地址

    "pushl %[next_ip]\n\t"

    "jmp __switch_to\n\t"       pushl 和 jmp 是个函数调用

__switch_to 的返回值存在 eax 寄存器中

    "1:\n\t"

    "popl %%ebp\n\t"

    "popfl\n\t"               出栈, 出栈的值赋给标志寄存器

这段汇编代码执行完后, 将 eax 的值写入 last, 将 ebx, ecx, edx, esi, edi 的值写入局部变量
ebx, ecx, edx, esi, edi

    : [prev_sp] "=m" (prev->thread.sp), prev->thread 是 struct thread_struct

thread.sp 是 unsigned long

    [prev_ip] "=m" (prev->thread.ip),

thread.ip 是 unsigned long

    "=a" (last), "=b" (ebx), "=c" (ecx), "=d" (edx), "=S" (esi), "=D" (edi)

    : [next_sp] "m" (next->thread.sp), [next_ip] "m" (next->thread.ip),

操作数 next_sp 和 next_ip 在内存中

    [prev] "a" (prev), [next] "d" (next)

这段汇编代码执行前, eax 寄存器的值等于 prev, edx 寄存器的值等于 next

    : "memory");

这段汇编代码执行完毕后, 内存中的内容已经改变

} while (0);

```

第二个问题的答案就显而易见了。第一个问题的回答是, 可以正常访问。

- 欲进一步学习 AT&T 汇编语言的同学，推荐阅读：
R.Blum, Professional Assembly Language, Wiley Publishing, Inc., 2005.
- 欲进一步了解 GCC 内联汇编的语法的同学，可阅读以下文献：
 1. GCC-Inline-Assembly-HOWTO,
<http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>
 2. Extended Asm - Using the GNU Compiler Collection,
<http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>

内存管理

§ 1 实验内容

- 写一个名为 mtest 的模块。模块创建文件 /proc/mtest。/proc/mtest 可写。
- 写入 “listvma” 则模块用 printk 输出当前进程的所有 Virtual Memory Area (VMA)，格式是每行对应一个 VMA，每行格式为

开始地址 结束地址 权限

例如

0x10000 0x20000 rwx
0x30000 0x40000 r—

- 写入 “findpage 虚拟地址” 则模块通过查页表和地址转换，用 printk 输出虚拟地址对应的物理地址。如果物理地址不存在则输出 “translation not found”。
- 写入 “writeval 虚拟地址 无符号长整型值” 则模块改动当前进程对应于虚拟地址的内存的值。注意：模块代码必须检查参数是否合法，模块不可以写内核使用的内存，即不可以写 identity mapping address。
- 同学实现一个用户程序。程序中有一个整型变量，比方说名字是 v，初始值是 0。程序启动后即等待输入命令。输入命令 “write 整数 i”，程序写 /proc/mtest，从而将 v 的值改成 i。输入命令 “print”，程序用 printf(“%d\n”, v) 输出 v 的值。
- 注意：GNU Lib C 完全缓冲文件流。同学调用用 fprintf、fwrite 等函数后，数据没有立即写入 /proc/mtest。应在 fprintf 之后调用 fflush。（邹南海）

同学们可阅读http://www.pixelbeat.org/programming/stdio_buffering/

<http://www.gnu.org/software/hello/manual/libc/Flushing-Buffers.html#Flushing-Buffers>

§ 2 实验参考：逻辑地址(Logical Address)，虚拟地址(Virtual Address)，物理地址(Physical Address)

逻辑地址是程序的指令中使用的地址。一个逻辑地址包括一个段和一个偏移量；虚拟地址是一个无符号整数，从 CPU 核发往内存管理单元(Memory Management Unit)的是虚拟地址，注意，CPU 核和 MMU 都在 CPU 内；物理地址是 CPU 发往内存总线的地址。

同学可进一步阅读以下文献

1. Physical Address, http://en.wikipedia.org/wiki/Physical_address
2. Logical Address, http://en.wikipedia.org/wiki/Logical_address
3. Virtual Address Space, http://en.wikipedia.org/wiki/Virtual_address_space
4. Memory Management Unit, http://en.wikipedia.org/wiki/Memory_management_unit

§ 3 实验参考: i386 的分段 段寄存器 段描述符 全局描述符表

i386 架构的“分段内存模型 (Segmented memory model)”中, 的一个程序可以有多个独立的地址空间, 每一个地址空间称为一个“段”, 用“16 比特段选择符 + 32 比特偏移量”表示地址。程序的代码、数据、栈在不同的段中, 这样, 随着程序栈的增长, 栈不会覆盖代码、数据。

i386 架构的 CS, DS, SS, ES, FS 和 GS 是段寄存器, 分别保存 16 比特的段选择符。CS 表示代码段, SS 表示栈段, DS, ES, FS, GS 表示数据段。

以下文献中有详细叙述, 请同学阅读。

1. Memory Addressing,
Understanding Linux Kernel, Chapter 2.
2. Segmented Addressing,
IA-32 Intel Architecture Software Developer's Manual, Vol. 1, Section 1.3.5
3. Memory Organization,
IA-32 Intel Architecture Software Developer's Manual, Vol. 1, Section 3.3
4. Segment Registers,
IA-32 Intel Architecture Software Developer's Manual, Vol. 1, Section 3.4.2
5. Operand Addressing,
IA-32 Intel Architecture Software Developer's Manual, Vol. 1, Section 3.7
6. D. Sedory, The Segment: Offset Addressing Scheme,
<http://mirror.href.com/thestarman/asm/debug/Segments.html>
7. x86 memory segmentation,
http://en.wikipedia.org/wiki/Segment_register

§ 4 实验参考: /proc/<pid>/maps 和 pmap

- 可以通过读/proc/<pid>/maps 文件得知程序的 VMA 情况。
此文件的实现请参阅内核代码 fs/proc/task_mmu.c 的 show_map_vma() 函数。
该文件的每一行对应一个 VMA, 每一行包括:

<code>vm_start vm_end flags offset_in_file major minor ino path</code>
--

vm_start 和 vm_end 是 VMA 的起始地址和 (结束地址+1)。

flags 包含四个字符, 如“rw-p”, 其中 r 表示可读, w 表示可写, x 表示可执行, s 表示可以共享, p 表示不可以共享, - 是占位符, 当不可读/不可写/不可执行时用 - 占位。

offset_in_file 表示此 VMA 的开始地址是文件开始的多少字节。

major 和 minor 是文件所在设备的主设备号和次设备号。

ino 是文件的 inode 的编号。

path 是文件的路径。

- 也可以使用 pmap 命令。语法为 `pmap -d 进程号` 或者 `pmap -x 进程号`

§ 5 思考题

1. 阅读 Understanding The Linux Kernel 的第二章，回答：为什么 i386 架构的 Linux Kernel 的线性地址最大值是 4G-1？从 CPU 的角度、从页表的角度解释。
2. 计算 i386 架构上，一个进程的页表的体积的最小可能值，和最大可能值，单位为字节。可用内存为 4G。
3. 页表项中有一个叫“Read/Write”的比特，它为 1 则表示此页可读写，为 0 则表示此页只读。那么，如何判断一个页是否有可执行权限？
4. 分别复述 常规分页、扩展分页 (Extended Paging)、和 Physical Address Extension，由线性地址求物理地址的过程。
5. 分别解释高速缓存的直接映射、N 路组关联和全关联。

参考文献

- [1] R. Blum, Professional Assembly Language, Wiley Publishing, Inc., 2005.
- [2] D. Bovet and M. Cesati, Understanding the Linux Kernel, Third Edition, O'Reilly, 2005.
- [3] J. Corbet, A. Rubini and G. Kroah-Hartman, Linux Device Drivers, Third Edition, China Electric Power Press, 2005.
- [4] Driver porting: The seq_file interface, <http://lwn.net/Articles/22355/> .
- [5] R. Dunlap, Linux kernel seq_file HOWTO,
http://www.xenotime.net/linux/doc/seq_file_howto.txt .
- [6] Extended Asm - Using the GNU Compiler Collection,
<http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>
- [7] J. Fenlason and R. Stallman, GNU gprof,
http://gnu.huihoo.org/gprof-2.9.1/html_node/gprof_toc.html
- [8] GCC-Inline-Assembly-HOWTO,
<http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>
- [9] S. Graham, P. Kessler and M. McKusick, An Execution Profiler for Modular Programs, Software - Practice and Experience, Vol. 13, pp. 671-685, 1983.
- [10] S. Graham, P. Kessler and M. McKusick, gprof: A Call Graph Execution Profiler, Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No 6, pp. 120-126, June 1982.
- [11] T. Granlund and R. Stallman, Coreutils - GNU core utilities,
<http://www.gnu.org/software/coreutils/>
- [12] HOWTO do Linux kernel development - take 3,
<http://permlink.gmane.org/gmane.linux.kernel/349656>
- [13] Kernel Trap, <http://kerneltrap.org/man/linux/9> .
- [14] R. Landley, Introducing initramfs, a new model for initial RAM disks,
<http://www.linuxdevices.com/articles/AT4017834659.html>
- [15] R. Love, Linux Kernel Development, Second Edition, China Machine Press, 2006.
- [16] Linux Init and System Initialization,
http://www.comptechdoc.org/os/linux/startupman/linux_suinit.html .
- [17] The Linux Kernel Module Programming Guide
http://www.linuxtopia.org/online_books/Linux_Kernel_Module_Programming_Guide/x773.html
- [18] The Linux Kernel: The Book, <http://kernelbook.sourceforge.net/> .
- [19] Linux kernel - Wikipedia, http://en.wikipedia.org/wiki/Linux_kernel .
- [20] Linux: Linux Init Process and PC Boot Procedure,
<http://www.yolinux.com/TUTORIALS/LinuxTutorialInitProcess.html> .
- [21] Linux startup process - Wikipedia, the free encyclopedia,
http://en.wikipedia.org/wiki/Linux_startup_process .
- [22] W. Mauerer, Professional Linux Kernel Architecture, Wiley Publishing Inc., 2008.
- [23] A. Nayani, M. Gorman and R. Castro, Memory Management in Linux, 2002.
- [24] proc(5): process info pseudo-filesystem, <http://linux.die.net/man/5/proc> .

- [25] C.Rodriguez, G.Fischer and S.Smolski, The Linux Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures, Prentice Hall PTR, 2005.
<http://book.opensourceproject.org.cn/kernel/kernelpri/> .
- [26] D.Rusling, Linux Kernel, Chinese translation,
<http://man.chinaunix.net/tech/lyceum/linuxK/tlk.html>
- [27] P.Salzman, M.Burian and O.Pomerantz, The Linux Kernel Module Programming Guide,
http://www.linuxtopia.org/online_books/Linux_Kernel_Module_Programming_Guide/, 2005.
- [28] W.Stevens and S.Rago, Advanced Programming in the UNIX Environment, Second Edition, POSTS & TELECOM Press, 2006.
- [29] Tanenbaum - Torvalds debate, http://en.wikipedia.org/wiki/Tanenbaum-Torvalds_debate .
- [30] Using the GNU Compiler Collection (GCC), <http://gcc.gnu.org/onlinedocs/gcc/> .
- [31] 安装 Ubuntu/Kubuntu/Xubuntu - OSWikiHK, http://wiki.debian.org.hk/w/Install_Ubuntu
- [32] 安装 Fedora - OSWikiHK, http://wiki.debian.org.hk/w/Install_Fedora_Linux
- [33] 陈莉君, 深入分析 Linux 内核源代码,
<http://www.kerneltravel.net/kernel-book/深入分析Linux内核源码.html> .
- [34] 毛德操, 胡希明, Linux 内核源代码情景分析(上、下), 浙江大学出版社, 2001.