[SDN: 3](#_Toc14614)

[2014.1.15 3](#_Toc9528)

[2014.2.11 3](#_Toc17648)

[2014.2.16 3](#_Toc1951)

[Python: 4](#_Toc12158)

[2014.2.10 4](#_Toc10187)

[2014.2.11 8](#_Toc28691)

[2014.2.17 9](#_Toc16610)

[网络: 9](#_Toc23023)

[2014.2.10 9](#_Toc4874)

[大数据： 9](#_Toc9956)

[2014.2.10 9](#_Toc2569)

[2014.2.12 10](#_Toc12271)

[2014.2.13 10](#_Toc3569)

[2014.2.17 10](#_Toc3002)

[学习心得： 10](#_Toc30237)

[2014.2.10 10](#_Toc18990)

[Linux: 11](#_Toc6399)

[启动篇： 11](#_Toc32114)

[2014.2.21 11](#_Toc1945)

[2014.2.25 12](#_Toc1038)

[2014.2.27 12](#_Toc14751)

[问题 12](#_Toc25873)

[进程篇： 12](#_Toc22397)

[2014.3.12 12](#_Toc22114)

[问题： 14](#_Toc31207)

[内存篇： 15](#_Toc1706)

[2014.2.27 15](#_Toc12958)

[2014.3.4 15](#_Toc20342)

[2014.3.5 15](#_Toc4228)

[问题： 15](#_Toc22915)

[2014.3.9 16](#_Toc8512)

[2014.3.10 21](#_Toc19537)

[2014.3.14 24](#_Toc17653)

[2014.3.19 24](#_Toc11123)

[网络篇： 29](#_Toc3302)

[问题 29](#_Toc22326)

[Fs篇： 29](#_Toc25592)

[问题 29](#_Toc5006)

[Driver篇： 29](#_Toc30825)

[问题 29](#_Toc20337)

[虚拟化篇： 29](#_Toc22594)

[2014.2.16 29](#_Toc30725)

[算法篇： 30](#_Toc32328)

[问题 30](#_Toc15952)

[知识篇： 30](#_Toc15219)

[2014.2.11 30](#_Toc11079)

[2014.2.19 31](#_Toc2367)

[2014.2.24 32](#_Toc32111)

[2014.2.25 33](#_Toc29513)

[2014.2.27 34](#_Toc25708)

[2014.3.20 35](#_Toc31515)

[问题 36](#_Toc4664)

[其他 36](#_Toc12052)

# SDN:

### 2014.1.15

<http://net.it168.com/a2014/0114/1585/000001585978_all.shtml>

SDN的形态

1：Openflow技术趋势

2：NFV技术发展趋势

3：I2RS技术发展趋势

4：overlay技术发展趋势

### 2014.2.11

看了当前sdn的控制器的文章，考虑有空自己用mininet搭建一个环境，测试下几个开源控制器的性能，然后自己可以跟一个控制器项目看看

### 2014.2.16

虚拟化和云的区别：

1：规模

2：API接口

3：多租户

4：提供自服务平台

5：按需

6：弹性扩展，快速部署

HA(高可用)

控制节点HA

Mysql HA Galera

消息队列HA RabbitMQ

API接口HA Haproxy

虚拟机HA

分布式文件系统

依赖Heat

类vmware

监控

Ceilometer

虚拟机内存监控和Agent

Libvirt的内存监控不准确

配置管理工具：

Puppet

Chef

Saltstack（python开发）

ansible

# Python:

### 2014.2.10

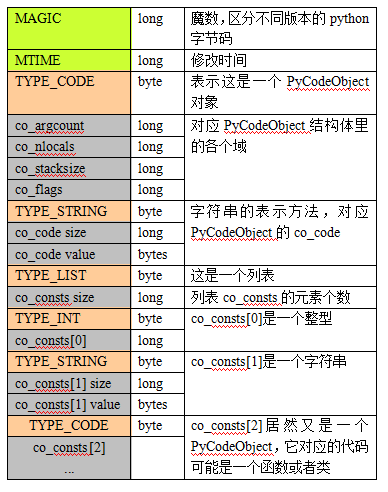
<http://blogread.cn/it/article/6606>

python程序的执行原理：

python是先把代码文件(.py)翻译成字节码，然后把自己码交给虚拟机，由虚拟机完成字节码的翻译。.pyc文件就是字节码。

字节码的产生时机：当import util时候会产生util.pyc会产生util的字节码

Python的字节码结构



如何看字节码：

Python提供了内置的compile可以编译python代码和查看PyCodeObject对象

例如：

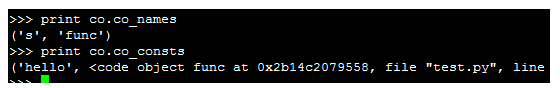
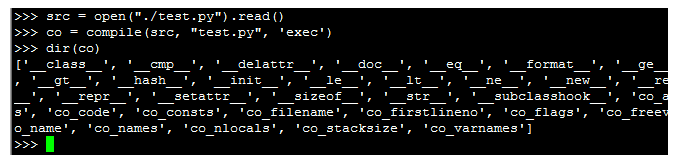
Test.py

S=’hello’

Def func():

Print S

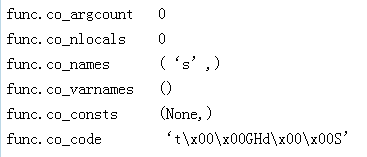
Func()



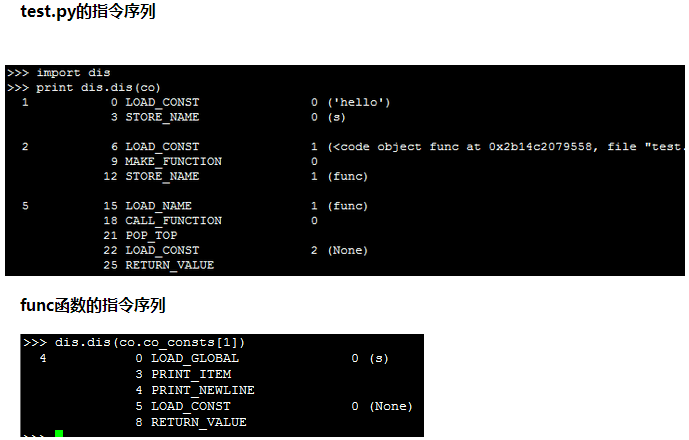
Dir列出的各个域都可以打印出来



Func=co.co\_consts[1]



Python内置的dis模块可以解析co\_code



第一列表示以下几个指令在py文件中的行号;

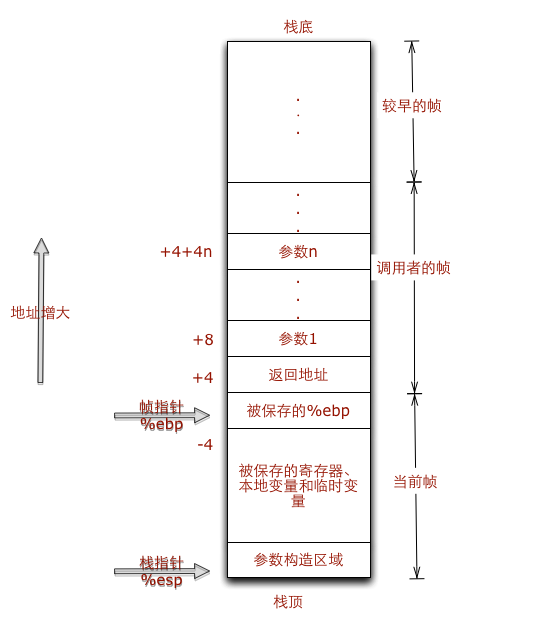
 第二列是该指令在指令序列co\_code里的偏移量;

 第三列是指令opcode的名称，分为有操作数和无操作数两种，opcode在指令序列中是一个字节的整数;

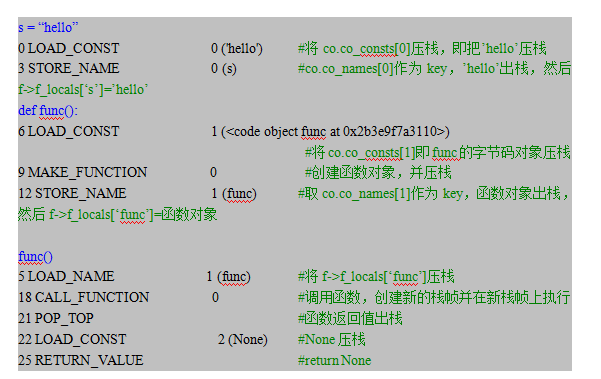
 第四列是操作数oparg，在指令序列中占两个字节，基本都是co\_consts或者co\_names的下标;

 第五列带括号的是操作数说明。

指令含义：与c语言类似：



各个co\_code的含义



### 2014.2.11

一

Csv(comma separated value): (<http://docs.python.org/2/library/csv.html>)就是一种处理csv格式的操作类

Codecs:字符的编码是按照某种在单字节字符和多字节字符之间进行转换的某种方法。从单字节到多字节叫做decodeing，从多字节到单字节叫做encoding；经常用到的是UTF-8和GB2312两种。Codecs模块中重要的函数式lookup，其他函数式：encoder，decoder，StreamReader，StreamWriter和StreamReaderWriter

StringIO：可以认为是内存中的文件系统

了解了generator object，默认有一个next方法。一个斐波那契数列的python实现



了解 os，sys,的操作方法

二

<http://bitworking.org/news/Why_so_many_Python_web_frameworks> python搭建简单web

三

<http://blog.csdn.net/jgood/article/details/5471626> 图片蜘蛛人

四：

<http://www.wsgjp.com.cn/?Redir=%2fDesktop.gspx> 管家婆

待学习python sqlit

### 2014.2.17

安装python2.6安装python pip和setuptools

wget <https://raw.github.com/pypa/pip/master/contrib/get-pip.py>

sh get-pip.py

在linux下配置出了一个简单的server网页

# 网络:

### 2014.2.10

<http://blogread.cn/it/article/6786>

TCP有4个拥塞控制机制：

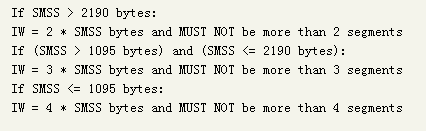
1：slow start, congestion avoidance, fast retransmit and fast recovery

Rwnd的大小取决于BDP，也就是带宽和时延的乘积

但是tcp使用16bit来记录窗口大小，最大也就64kb，超过64kb使用tcp\_window\_scaling机制具体参看<http://packetlife.net/blog/2010/aug/4/tcp-windows-and-window-scaling/>

Cwnd的大小取决于MSS(maximum segment size最大段大小)

min(4 \* MSS, max(2 \* MSS, 4380))



Google's advice ：《An Argument for Increasing TCP's Initial Congestion Window》

# 大数据：

### 2014.2.10

<http://blogread.cn/it/article/2455?f=wb2>

### 2014.2.12

<http://www.uml.org.cn/zjjs/201211065.asp> 分布式

### 2014.2.13

在分布是系统中三个核心：

Consistency/availability/partition tolerance

Consistency（一致性）：这个和数据库ACID的一致性类似，但这里关注的所有数据节点上的数据一致性和正确性，而数据库的ACID关注的是在在一个事务内，对数据的一些约束。

Availability（可用性）：关注的在某个结点的数据是否可用，可以认为某一个节点的系统是否可用，通信故障除外。

Partition Tolerance（分区可容忍性）：是否可以对数据进行分区。这是考虑到性能和可伸缩性。

### 2014.2.17

Google的十个核心技术可以分为四类：

分布式基础设施：GFS、Chubby（分布式锁）和protocol Buffer

分布式大规模数据处理：MapReduce和Sawzall

分布式数据库处理技术BigTable和数据库Sharding

Hadoop生态：

Ambari Zookeeper Hcatlog Oozie

(management) (Coordinstion) (tablemanagement) (Workflow Scheduler)

Hbase Pig(Data Flow) Hive(SQL)

(Big Table) MapReduce

HDFS

# 学习心得：

### 2014.2.10

<http://blogread.cn/it/article/4015?f=hot1> 学习的步骤

<http://chimera.labs.oreilly.com/books/1230000000545/index.html> 网络性能的书

# Linux:

## 启动篇：

### 2014.2.21

Linux启动的5个步骤：

1. 系统启动
2. Stage1：bootloader
3. Stage2：bootloader
4. Kernel
5. Init

启动顺序详细介绍：

1. 系统启动阶段：
2. 打开计算机
3. CPU跳到0xFFFF0处运行BIOS
4. BIOS找到一个可以启动的设备（例如：USB，硬盘，CD-rom）
5. 加载并执行MBR；（可用设备的头512字节）
6. stage1：bootloader：
7. bootloader运行512字节中的命令：

MBR的内容：前446字节：是bootloader

接下来的64字节是四个分区表

最后的8个字节是一个魔术字：0xaa55

具体参见：<http://www.wallcopper.com/os/1140.html>

/boot/grub/menu.conf

1. 执行446字节的内容，然后加载后面的30KB的内容
2. Grub接管控制，显示grub.boot菜单
3. Grub加载用户选择的kernel到内存中
4. Kernel的主要顺序

说明：kernel image的类型有两张：zImage：大小小于512KB，bzImage：大小大于512KB

1. Start() ./arch/i386/boot/head.S
2. Startup\_32() ./arch/i386/boot/compress/head.S

Decompress\_kernel() ./arch/i386/boot/compress/misc.c

1. startup\_32() ./arch/i386kernel/head.S
2. start\_kernel() ./init/main.c
3. cpu\_idle() ./init/main.c
4. 调用init脚本启动第一个进程
5. Init的作用：
6. 是kernel执行的第一个进程
7. 他是所有的进程的root
8. /etc/rc.d/rc.sysinit是第一个进程
9. /etc/inittab是第一个进程的配置文件
10. 在关机的时候，这个root进程会按照顺序关闭进程

阅读documentation/x86/boot.txt介绍了，引导的流程。

### 2014.2.25

Initramfs：

相关文章：<http://lwn.net/Articles/14448/>

<http://blog.linux.org.tw/~jserv/archives/001840.html>

<http://blog.linux.org.tw/~jserv/archives/001954.html>

<http://www.cnblogs.com/wwang/archive/2010/10/27/1862222.html>

文档：

Documentation/filesystems

我的理解：

Kernel在加载完成后，需要运行一个用户态的程序，但是用户的程序存在文件系统中，因此内核必须找到一个可以挂在的文件系统才可以完成系统的引导过程。还有一些驱动程序也是在文件系统中，所以如果想挂载文件系统，那么文件系统又需要底层设备，而设备又需要驱动，那么这就是死锁逻辑，所以在挂载真正系统前需要一些东西来引导。

### 2014.2.27

Initramfs相关的连接：

<http://www.cnblogs.com/KFC99/archive/2013/05/28/3104553.html>

<http://www.cnblogs.com/pied/archive/2013/01/29/2880718.html>

我的理解：内核生成的initramfs会调用init函数。

### 问题

1) MRB的512扇区的程序是怎么写入的？又是怎么编译生成的？

## 进程篇：

### 2014.3.12

介绍抢占和中断的文章：<http://www.cnblogs.com/hustcat/archive/2009/08/31/1557507.html>

介绍上下文的：<http://www.cnblogs.com/hustcat/articles/1505618.html>

当我们在shell中输入a.out的时候，shell先通过调用fork函数复制出一个子进程，然后用execve先删掉这个进程的用户区域（和shell长得一模一样），再将a.out和标准c库加载到存储器中，最后把PC指向hello的第一条语句，于是cpu就开始逐条执行指令。

用户态的程序：

0x40000000开始是动态库

程序起始：0x08048000

用户栈：0xbfffffff

可以看下：<http://blog.csdn.net/sunstars2009918/article/details/6000083>

中断的介绍：<https://www.ibm.com/developerworks/cn/linux/l-cn-linuxkernelint/>

中断可分为同步（synchronous）中断和异步（asynchronous）中断：

1. 同步中断是当指令执行时由 CPU 控制单元产生，之所以称为同步，是因为只有在一条指令执行完毕后 CPU 才会发出中断，而不是发生在代码指令执行期间，比如系统调用。

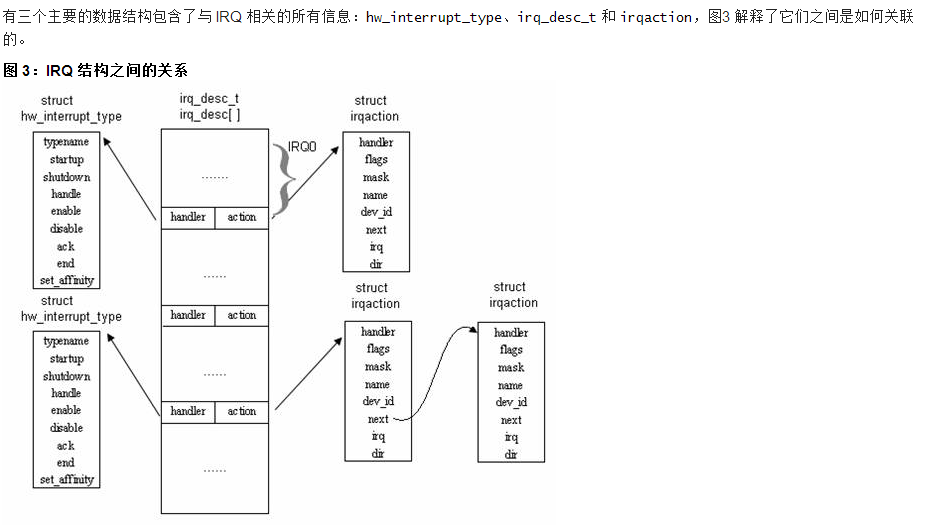
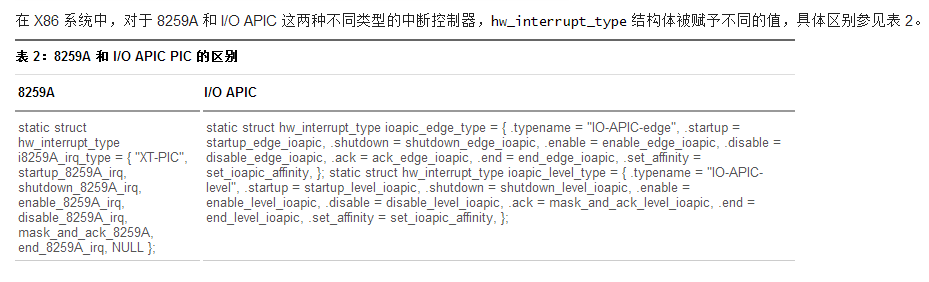
2. 异步中断是指由其他硬件设备依照 CPU 时钟信号随机产生，即意味着中断能够在指令之间发生，例如键盘中断。

根据 Intel 官方资料，同步中断称为异常（exception），异步中断被称为中断（interrupt）。

中断可分为可屏蔽中断（Maskable interrupt）和非屏蔽中断（Nomaskable interrupt）。异常可分为故障（fault）、陷阱（trap）、终止（abort）三类。

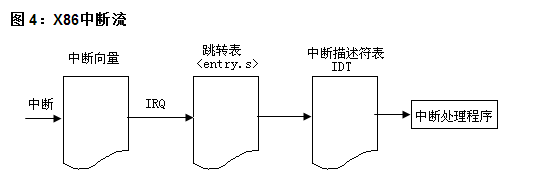


X86 体系结构的每个中断都被赋予一个唯一的编号或者向量（8 位无符号整数）。非屏蔽中断和异常向量是固定的，而可屏蔽中断向量可以通过对中断控制器的编程来改变。



对于每一个外设，要么以静态（声明为 static 类型的全局变量）或动态（调用 request\_irq 函数）的方式向 Linux 内核注册中断处理程序。不管以何种方式注册，都会声明或分配一块 irqaction 结构（其中 handler 指向中断服务程序），然后调用 setup\_irq() 函数，将irq\_desc\_t 和 irqaction 联系起来。

Irq\_entries\_start->common\_interrupt



<http://www.embeddedlinux.org.cn/html/yingjianqudong/201304/19-2554.html>

<https://www.ibm.com/developerworks/cn/linux/l-cn-linuxkernelint/>

<http://www.whitecell.org/list.php?id=44>

### 问题：

1. 在spin\_lock临界区中间发生了中断，内核会做什么？
2. 抢占和中断的关系是什么？在spin\_lock时候需要禁用抢占，而spin\_lock\_irq()需要禁用抢占也需要禁用中断？的场景

## 内存篇：

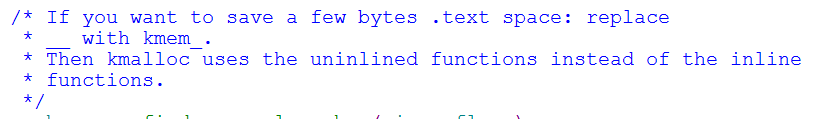
### 2014.2.27

用户态和内核态通信方式：

内存映射,netlink,notify，ioctl

### 2014.3.4

内核2.6.32中，在\_\_do\_kmalloc函数中有这么一句话他的意思是什么？



\_\_builtin\_constant\_p 用于判断一个值是否为编译时常数，如果参数size的值是常数，函数返回 1，否则返回 0。\_\_builtin\_constant\_p是GCC的内建函数

<http://blog.csdn.net/lights_joy/article/details/2732441> 介绍buddy的文章；

### 2014.3.5

物理内存分布：<http://blog.csdn.net/dlutbrucezhang/article/details/9058583>

malloc后系统都做了什么？<http://edsionte.com/techblog/archives/4174>

缓存：<http://edsionte.com/techblog/archives/4217>

Brk(<http://blog.csdn.net/u010246947/article/details/10472663)/mmap>

对虚拟内存的讲解：<http://forum.ubuntu.org.cn/viewtopic.php?t=276918>

Linux x86\_64中为什么没有高端内存的原因: <http://adam8157.info/blog/2012/07/linux-x86-64-vm>

我的理解：

内核是运行在虚拟空间中的：那么他的从PAGE\_OFFSET(0xc000000)到PAGE\_OFFSET+896M这个区间与0x0到896M一一对应，而PAGE\_OFFSET+896M到0xffffffff的内存与896M到PAGE\_OFFSET相映射。内核这么来管理内存；当分配0~896M的内存时候，就和内核的PAGE\_OFFSET+896M相对应，但是当分配896M到3G的内存时候，就需要利用PAGE\_OFFSET+896M到4G做映射了。

注意理解：brk/mmap/kmalloc/vmalloc

### 问题：

1. MMU完成的虚拟内存和物理内存的映射，那么kernel的职责是什么？
2. 中断上下文中使用什么来申请内存？
3. 3用户空间上下文和内核上下文怎么切换？用户空间上下文和中断怎么切换？内核上下文与中断怎么切换？
4. 信号(例如kill一个进程)在父进程和子进程/孙子进程之间是怎么传递的？
5. 对于内核中的宏 \_\_assmble\_\_等这些宏哪里找他们的定义位置
6. cache line对其的用处
7. CPU的cache line是如何和内存映射的，也就是

为什么slab的算法是的某些cache line长期无法用

而引入了coler来解决

1. slab分配器所用的元数据的内存是从哪里分配的？？
2. 总结一些/proc这个文件中的 都有哪些信息，作用是什么？
3. 存需要映射：

问题是 如果内存低于128M；

如果内存大于28M小于896M

如果内存大于896M小于3G。

如果内存大于3G小于4G

在这些情况下都是怎么映射的？？

sttic percpu ops

DECLARE\_PER\_CPU(type, name)

per-cp data at runtime

alloc\_percpu

### 2014.3.9

CentOS6.4内核升级, 2.6.\*版本升级 Kernel 3.10.\*

Kernel当前最新的稳定版为3.10.4，而CentOS6.4的内核一直还是2.6.x，内核版本2.6.28系列有一些内核BUG，最终决定升级linux系统内核版本。

在测试环境进行内核升级测试，步骤如下：

一：升级环境介绍：

内核升级测试环境介绍：

虚拟机软件：VMWare 9.0.0

测试系统： CentOS 6.4 x86\_64

确认升级前内核版本：

more /proc/version

Linux version 2.6.32-358.14.1.el6.x86\_64 (mockbuild@c6b10.bsys.dev.centos.org) (gcc version 4.4.7 20120313 (Red Hat 4.4.7-3) (GCC) ) #1 SMP Tue Jul 16 23:51:20 UTC 2013

uname -a

Linux master 2.6.32-358.14.1.el6.x86\_64 #1 SMP Tue Jul 16 23:51:20 UTC 2013 x86\_64 x86\_64 x86\_64 GNU/Linux

升级前grup文件信息：

[more /boot/grub/grub.conf

# grub.conf generated by anaconda

#

# Note that you do not have to rerun grub after making changes to this file

# NOTICE: You have a /boot partition. This means that

# all kernel and initrd paths are relative to /boot/, eg.

# root (hd0,0)

# kernel /vmlinuz-version ro root=/dev/mapper/vg\_slave2-lv\_root

# initrd /initrd-[generic-]version.img

#boot=/dev/sda

default=0

timeout=5

splashimage=(hd0,0)/grub/splash.xpm.gz

hiddenmenu

title CentOS (2.6.32-220.el6.x86\_64)

root (hd0,0)

kernel /vmlinuz-2.6.32-220.el6.x86\_64 ro root=/dev/mapper/vg\_slave2-lv\_root nomodeset rd\_NO\_LUKS rd\_NO\_MD quiet rhgb crashkernel=auto LANG=zh\_CN.UTF-8 rd\_LVM\_LV=vg\_slave2/lv\_root K

EYBOARDTYPE=pc KEYTABLE=us rd\_LVM\_LV=vg\_slave2/lv\_swap rd\_NO\_DM

initrd /initramfs-2.6.32-220.el6.x86\_64.img

二：安装系统所需要的编译工具：

yum install wget gcc gc bc gd make perl ncurses-devel xz -y

如果执行上面的安装命令后，在编译过程中提示缺少依赖软件包请执行下面的软件安装命令

yum install gcc gcc-c++ autoconf libjpeg libjpeg-devel libpng libpng-devel freetype freetype-devel libxml2 libxml2-devel zlib zlib-devel glibc glibc-devel glib2 glib2-devel bzip2 bzip2-devel zip unzip ncurses ncurses-devel curl curl-devel e2fsprogs e2fsprogs-devel krb5-devel libidn libidn-devel openssl openssh openssl-devel nss\_ldap openldap openldap-devel openldap-clients openldap-servers libxslt-devel libevent-devel ntp libtool-ltdl bison libtool vim-enhanced python wget lsof iptraf strace lrzsz kernel-devel kernel-headers pam-devel Tcl/Tk cmake ncurses-devel bison setuptool popt-devel rsynx openssh system-config-network-tui gcc gc bc gd make perl ncurses-devel xz -y

三：下载当前最稳定的Kernel并解压

cd /usr/src

wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.10.4.tar.xz

xz -d linux-3.10.4.tar.xz

tar -xvf linux-3.10.4.tar.xz

cd linux-3.10.4

四：清理系统内的环境变量及依赖

make mrproper #清除环境变量（清除配置文件）

make dep #清除依赖

cp /boot/config-\* /usr/src/kernels/ #为了方便编译配置，将/boot下的配置文件复制到当前目录下的/usr/src/kernels/文件中

五：准备开始啦

make menuconfig出现General setup菜单，根据自己的需要对Kernel进行增、减。完后按Tab键跳到下面菜单中的 < Exit > 上，回车，系统问

Do you wish to save your new configuration ?to continue.

< Yes >< No >

菜单默认在< Yes >上，按回车确认保存。

make clean #确保所有东西均保持最新状态

make bzImage #生成内核文件

make modules #编译模块（注：此处需要很长时间，请耐心等待）

make modules\_install #安装模块

make install #安装，此处报了一个vmware的错，详见最后故障排查

如果以上步骤都顺利执行完成，那么恭喜你内核升级已基本完成。

六：修改系统启动菜单并重启服务器，使服务器在下次启动使用新的内核。

[root@master ~]# cat /boot/grub/grub.conf

# grub.conf generated by anaconda

#

# Note that you do not have to rerun grub after making changes to this file

# NOTICE: You have a /boot partition. This means that

# all kernel and initrd paths are relative to /boot/, eg.

# root (hd0,0)

# kernel /vmlinuz-version ro root=/dev/mapper/vg\_master-lv\_root

# initrd /initrd-[generic-]version.img

#boot=/dev/sda

default=0 #原本是1修改成 0

timeout=5

splashimage=(hd0,0)/grub/splash.xpm.gz

hiddenmenu

title CentOS (3.10.4)

root (hd0,0)

kernel /vmlinuz-3.10.4 ro root=/dev/mapper/vg\_master-lv\_root nomodeset rd\_LVM\_LV=vg\_master/lv\_swap KEYBOARDTYPE=pc KEYTABLE=us rd\_NO\_MD rd\_NO\_LUKS quiet rhgb crashkernel=auto LANG=en\_US.UTF-8 rd\_LVM\_LV=vg\_master/lv\_root rd\_NO\_DM

initrd /initramfs-3.10.4.img

title CentOS (2.6.32-358.14.1.el6.x86\_64)

root (hd0,0)

kernel /vmlinuz-2.6.32-358.14.1.el6.x86\_64 ro root=/dev/mapper/vg\_master-lv\_root nomodeset rd\_LVM\_LV=vg\_master/lv\_swap KEYBOARDTYPE=pc KEYTABLE=us rd\_NO\_MD rd\_NO\_LUKS quiet rhgb crashkernel=auto LANG=en\_US.UTF-8 rd\_LVM\_LV=vg\_master/lv\_root rd\_NO\_DM

initrd /initramfs-2.6.32-358.14.1.el6.x86\_64.img

此时需要重启以便启动新的内核，

init 6

七：确认内核版本升级成功：

[root@master ~]# uname -a

Linux master 3.10.4 #内核版本已升级

[root@master ~]# cat /proc/version

Linux version 3.10.4 (root@master) #内核版本已升级

八：故障排查：

在执行 make install 内核升级完成后报错如下：

ERROR: modinfo: could not find module vmware\_balloon

找不到 vmware\_balloon 模块，

原因有2：

首先，确认你的 .config 文件里面有 CONFIG\_VMWARE\_BALLOON=m 这一行，或者你可以用 make menuconfig 进去内核编译菜单，选中 Device Drivers -> MISC devices -> VMware Balloon Driver 为 M 或者 \*

其次，而这个模块在后面的版本中，已经更名为 vmw\_balloon，所以 可以用下面的命令来解决：

cd /lib/modules/3.x.xx/kernel/drivers/misc #将版本号改成你自己的

ln -s vmw\_balloon.ko vmware\_balloon.ko #建立软连接

#回头找一台正式服务器试一下，应该不会报这个错误。

问题：kthread\_run(threadfn, data, namefmt, …) 可以用来创建一个内核线程，

它的参数含义如下：

/\*\*

\* kthread\_run - create and wake a thread.

\* @threadfn: the function to run until signal\_pending(current).

\* @data: data ptr for @threadfn.

\* @namefmt: printf-style name for the thread.

kthread\_stop() 用来销毁一个内核线程。

它们的用法可以参考: kernel/rcutorture.c

内存分配与释放可以用：kmalloc() / kfree() (linux/slab.h)

请设计一个模块：

1）设计一个多读者多写者的可动态扩展的(不会丢包)的ring buffer。

2）跑多个线程来测试你的ring buffer是否正确。

3）可以用参数来指定跑的读者和写者的数量。

### 2014.3.10

DEFINE\_PER\_CPU详解：

static DEFINE\_PER\_CPU(struct runqueue, runqueues);

首先，在arch/i386/kernel/vmlinux.lds中有 

/\* will be freed after init \*/

. = ALIGN(4096); /\* Init code and data \*/

\_\_init\_begin = .;

/\* 此处省略若干行:) \*/

. = ALIGN(32);

\_\_per\_cpu\_start = .;

.data.percpu : { \*(.data.percpu) }

\_\_per\_cpu\_end = .;

. = ALIGN(4096);

\_\_init\_end = .;

/\* freed after init ends here \*/

这说明\_\_per\_cpu\_start和\_\_per\_cpu\_end标识.data.percpu这个section的开头和结尾   
并且，整个.data.percpu这个section都在\_\_init\_begin和\_\_init\_end之间，   
也就是说，该section所占内存会在系统启动后释放(free)掉   
  
因为有   
#define DEFINE\_PER\_CPU(type, name)   
\_\_attribute\_\_((\_\_section\_\_(".data.percpu"))) \_\_typeof\_\_(type) per\_cpu\_\_##name   
  
所以   
static DEFINE\_PER\_CPU(struct runqueue, runqueues);   
会扩展成   
\_\_attribute\_\_((\_\_section\_\_(".data.percpu"))) \_\_typeof\_\_(struct runqueue)   
per\_cpu\_\_runqueues;   
也就是在.data.percpu这个section中定义了一个变量per\_cpu\_\_runqueues，   
其类型是struct runqueue。事实上，这里所谓的变量per\_cpu\_\_runqueues，   
其实就是一个偏移量，标识该变量的地址。   
  
--------------------   
其次，系统启动后，在start\_kernel()中会调用如下函数 

unsigned long \_\_per\_cpu\_offset[NR\_CPUS];

static void \_\_init setup\_per\_cpu\_areas(void)

{

unsigned long size, i;

char \*ptr;

/\* Created by linker magic \*/

extern char \_\_per\_cpu\_start[], \_\_per\_cpu\_end[];

/\* Copy section for each CPU (we discard the original) \*/  
 size = ALIGN(\_\_per\_cpu\_end - \_\_per\_cpu\_start, SMP\_CACHE\_BYTES);  
#ifdef CONFIG\_MODULES  
 if (size < PERCPU\_ENOUGH\_ROOM)  
 size = PERCPU\_ENOUGH\_ROOM;  
#endif

ptr = alloc\_bootmem(size \* NR\_CPUS);

for (i = 0; i < NR\_CPUS; i++, ptr += size) {

\_\_per\_cpu\_offset[i] = ptr - \_\_per\_cpu\_start;

memcpy(ptr, \_\_per\_cpu\_start, \_\_per\_cpu\_end - \_\_per\_cpu\_start);

}

}

在该函数中，为每个CPU分配一段专有数据区，并将.data.percpu中的数据拷贝到其中，   
每个CPU各有一份。由于数据从\_\_per\_cpu\_start处转移到各CPU自己的专有数据区中了，   
因此存取其中的变量就不能再用原先的值了，比如存取per\_cpu\_\_runqueues   
就不能再用per\_cpu\_\_runqueues了，需要做一个偏移量的调整，   
即需要加上各CPU自己的专有数据区首地址相对于\_\_per\_cpu\_start的偏移量。   
在这里也就是\_\_per\_cpu\_offset[i]，其中CPU i的专有数据区相对于   
\_\_per\_cpu\_start的偏移量为\_\_per\_cpu\_offset[i]。   
这样，就可以方便地计算专有数据区中各变量的新地址，比如对于per\_cpu\_runqueues，   
其新地址即变成per\_cpu\_runqueues+\_\_per\_cpu\_offset[i]。   
  
经过这样的处理，.data.percpu这个section在系统初始化后就可以释放了。   
  
--------------------   
再看如何存取per cpu的变量 

/\* This macro obfuscates arithmetic on a variable address so that gcc

shouldn't recognize the original var, and make assumptions about it \*/

#define RELOC\_HIDE(ptr, off)

({ unsigned long \_\_ptr;

\_\_asm\_\_ ("" : "=g"(\_\_ptr) : "0"(ptr));

(typeof(ptr)) (\_\_ptr + (off)); })

/\* var is in discarded region: offset to particular copy we want \*/

#define per\_cpu(var, cpu) (\*RELOC\_HIDE(&per\_cpu\_\_##var, \_\_per\_cpu\_offset[cpu]))

#define \_\_get\_cpu\_var(var) per\_cpu(var, smp\_processor\_id())

#define get\_cpu\_var(var) (\*({ preempt\_disable(); &\_\_get\_cpu\_var(var); }))

对于\_\_get\_cpu\_var(runqueues)，将等效地扩展为   
\_\_per\_cpu\_offset[smp\_processor\_id()] + per\_cpu\_\_runqueues   
并且是一个lvalue，也就是说可以进行赋值操作。   
这正好是上述per\_cpu\_\_runqueues变量在对应CPU的专有数据区中的新地址。   
  
由于不同的per cpu变量有不同的偏移量，并且不同的CPU其专有数据区首地址不同，   
因此，通过\_\_get\_cpu\_var()便访问到了不同的变量。

### 2014.3.14

Buddy算法可以看的函数：rmqueue\_bulk和free\_page\_bulk。

set\_page\_order来设置页的阶数，最后每个阶 通过lru管理；

### 2014.3.19

内存屏障：<http://ifeve.com/linux-memory-barriers/#abstract-menory-access-model>

1：不能假设cpu对无关联的load和store指令会按给定的顺序发出

2：必须要假定重叠的内存访问可能会被合并或丢弃。

为什么需要内存屏障：因为1）编译器优化/ 2）缓存优化 /3） CPU乱序执行

CPU对指令如下保证：

* 1. On any given CPU,dependent memory access will be issued in order,with respect to itself.
  2. Overlapping loads and stores within a particular CPU will appear to ordered within that CPU.
  3. It must not be assumed that independent loads and stores will be issued in the order given.
  4. It must be assumed that overlapping memory accesses may be merged or discarded.

内存屏障的作用：

1. 写（store）内存屏障。在写屏障之前的STORE操作将先于所有在写屏障之后的STORE操作
2. 写数据依赖屏障。两条Load指令，第二条Load指令依赖于第一条Load指令的结果，则数据依赖屏障保障第二条指令的目标地址将被更新。
3. 读（load）内存屏障。读屏障包含数据依赖屏障的功能，并且保证所有出现在屏障之前的Load操作都先于所有出现在屏障之后的Load操作被系统中的其他组件所感知。
4. 通用内存屏障，通用内存屏障保证所有出现在屏障之前的Load和Store操作都将先于所有出现在屏障之后的Load和Store操作被系统中的其他组件所感知
5. Lock操作，他的作用相当于一个单向渗透屏障。他保证所有出现在Lock之后的内存操作都将在Lock操作被系统中其他组件所感知自后才发生。出现在Lock之前的内存操作可能在Lock完成之后才发生。Lock操作总是跟Unlock操作配对出现。
6. Unlock操作。他保证所有出现在Unlock之前的内存操作都将在Unlock操作被系统中其他组件所感知之前发生。
7. 1

Gcc 4.2以上提供了一下内置的原子指令：

type \_\_sync\_fetch\_and\_add (type \*ptr, type value);

type \_\_sync\_fetch\_and\_sub (type \*ptr, type value);

type \_\_sync\_fetch\_and\_or (type \*ptr, type value);

type \_\_sync\_fetch\_and\_and (type \*ptr, type value);

type \_\_sync\_fetch\_and\_xor (type \*ptr, type value);

type \_\_sync\_fetch\_and\_nand (type \*ptr, type value);

type \_\_sync\_add\_and\_fetch (type \*ptr, type value);

type \_\_sync\_sub\_and\_fetch (type \*ptr, type value);

type \_\_sync\_or\_and\_fetch (type \*ptr, type value);

type \_\_sync\_and\_and\_fetch (type \*ptr, type value);

type \_\_sync\_xor\_and\_fetch (type \*ptr, type value);

type \_\_sync\_nand\_and\_fetch (type \*ptr, type value);

<http://hi.baidu.com/yaosheng001/item/0add477c3bbfc73a6cc37cfe>

内核中定义的内存屏障原语有：  
    
  #define barrier() \_\_asm\_\_ \_\_volatile\_\_("": : :"memory")  
  #define mb() alternative("lock; addl $0,0(%%esp)", "mfence", X86\_FEATURE\_XMM2)  
  #define rmb() alternative("lock; addl $0,0(%%esp)", "lfence", X86\_FEATURE\_XMM2)

 #ifdef CONFIG\_SMP  
  #define smp\_mb() mb()  
  #define smp\_rmb() rmb()  
  #define smp\_wmb() wmb()  
  #define smp\_read\_barrier\_depends() read\_barrier\_depends()  
  #define set\_mb(var, value) do { (void) xchg(&var, value); } while (0)  
 #else  
  #define smp\_mb() barrier()  
  #define smp\_rmb() barrier()  
  #define smp\_wmb() barrier()  
  #define smp\_read\_barrier\_depends() do { } while(0)  
  #define set\_mb(var, value) do { var = value; barrier(); } while (0)  
 #endif

 1). smp\_xxx()和xxx()的区别  
    
  为了给其它CPU也提供相关的barrier宏。 例如x86的rmb()是用了lfence指令，但其它CPU不能用这个指令。

 2). 关于barrier()宏，jkl大师是这么说的：

  CPU越过内存屏障后，将刷新自己对存储器的缓冲状态。这条语句实际上不生成任何代码，但可使gcc在  
  barrier()之后刷新寄存器对变量的分配。  
   
     也就是说，barrier()宏只约束gcc编译器，不约束运行时的CPU行为。 举例：  
    
  1 int a = 5, b = 6;  
  2 barrier();  
  3 a = b;  
   
     在line 3，GCC不会用存放b的寄存器给a赋值，而是invalidate b的Cache line，重新读内存中的b值，赋值给a。

 3). mb() vs. rmb() vs. wmb()

     rmb()不允许读操作穿过内存屏障；wmb()不允许写操作穿过屏障；而mb()二者都不允许。

     看IA32上wmb()的定义：  
     #ifdef CONFIG\_X86\_OOSTORE  
      #define wmb() alternative("lock;addl $0,0(%%esp)", "sfence", X86\_FEATURE\_XMM);  
     #else  
      #define wmb() \_\_asm\_\_ \_\_volatile\_\_ ("": : :"memory");  
     #endif

     Intel和AMD都没有在IA32 CPU中实现乱续写(Out-Of-Order Store)，所以wmb()定义为空操作，不约束CPU行为；但  
     有些IA32 CPU厂商实现了OOO Store，所以就有了使用sfence的那个wmb()实现。

 4). 内存屏障的体系结构语义  
    
    4.1) 只有一个主体(CPU或DMA控制器)访问内存时，无论如何也不需要barrier；但如果有两个或更多主体访问内存，且  
  其中有一个在观测另一个，就需要barrier了。

    4.2) IA32 CPU调用有lock前缀的指令，或者如xchg这样的指令，会导致其它的CPU也触发一定的动作来同步自己的Cache。  
  CPU的#lock引脚链接到北桥芯片(North Bridge)的#lock引脚，当带lock前缀的执行执行时，北桥芯片会拉起#lock  
  电平，从而锁住总线，直到该指令执行完毕再放开。  而总线加锁会自动invalidate所有CPU对 \_该指令设计的内存\_  
  的Cache，因此barrier就能保证所有CPU的Cache一致性。

    4.3) 接着解释。  
         lock前缀(或cpuid、xchg等指令)使得本CPU的Cache写入了内存，该写入动作也会引起别的CPU invalidate其Cache。  
  IA32在每个CPU内部实现了Snoopying(BUS-Watching)技术，监视着总线上是否发生了写内存操作(由某个CPU或DMA控  
  制器发出的)，只要发生了，就invalidate相关的Cache line。 因此，只要lock前缀导致本CPU写内存，就必将导致  
  所有CPU去invalidate其相关的Cache line。

  两个地方可能除外：  
   -> 如果采用write-through策略，则根本不存在缓存一致性问题(Linux对全部内存采用write-back策略);  
   -> TLB也是Cache，但它的一致性(至少在IA32上)不能通过Snoopying技术解决，而是要发送  
      INVALIDATE\_TLB\_VECTOR这个IPI给其它的CPU。  
   
   4.4) 进一步解释，MESI协议  
          
        M: Modified，已修改  
        E: Exclusive，排他  
        S: Shared，共享  
        I: Invalid，无效

        IA32 的CPU实现了MESI协议来保证Cache coherence。 CPU的总线监测单元，始终监视着总线上所有的内存写操作，  
        以便随时调整自己的Cache状态。

             -> Modified。 本CPU写，则直接写到Cache，不产生总线事物；其它CPU写，则不涉及本CPU的Cache，其它CPU  
                    读，则本CPU需要把Cache line中的数据提供给它，而不是让它去读内存。  
        
      -> Exclusive。只有本CPU有该内存的Cache，而且和内存一致。 本CPU的写操作会导致转到Modified状态。

      -> Shared。   多个CPU都对该内存有Cache，而且内容一致。任何一个CPU写自己的这个Cache都必须通知其它  
                    的CPU。

      -> Invalid。  一旦Cache line进入这个状态，CPU读数据就必须发出总线事物，从内存读。

  5) 考虑到DMA  
     
  5.1). Wirte through策略。 这种情形比较简单。

        -> 本CPU写内存，是write through的，因此无论什么时候DMA读内存，读到的都是正确数据。  
        -> DMA写内存，如果DMA要写的内存被本CPU缓存了，那么必须Invalidate这个Cache line。下次CPU读它，就  
           直接从内存读。  
    
  5.2). Write back策略。 这种情形相当复杂。  
     
        -> DMA读内存。被本CPU总线监视单元发现，而且本地Cache中有Modified数据，本CPU就截获DMA的内存读操作，  
           把自己Cache Line中的数据返回给它。

        -> DMA写内存。而且所写的位置在本CPU的Cache中，这又分两种情况：  
          a@ Cache Line状态未被CPU修改过(即cache和内存一致)，那么invalidate该cache line。  
     b@ Cache Line状态已经被修改过，又分2种情况：  
       
     <1> DMA写操作会替换CPU Cache line所对应的整行内存数据，那么DMA写，CPU则invalidate  
         自己的Cache Line。  
     <2> DMA写操作只替换Cache Line对应的内存数据的一部分，那么CPU必须捕获DMA写操作的新  
         数据(即DMA想把它写入内存的)，用来更新Cache Line的相关部分。

问题：

\_\_ASSEMBLY\_\_/ \_\_CHECKER\_\_/\_\_GNUC\_\_ 这些宏定义的位置在哪里？

## 网络篇：

### 问题

## Fs篇：

### 问题

## Driver篇：

### 问题

## 虚拟化篇：

### 2014.2.16

半虚拟化：

在guest的kernel中将所有的敏感操作全部替换掉（需要修改guest kernel的代码）

Pv-ops in linux kernel

Hypercall in xen

Int on 32bit

Syscall on 64bit

这种pv-ops的形式的缺点是：需要有guest kernel的代码并修改。但是windows没有源代码所以没法使用。Windows公司向linux kernel提交了一个patch，使得windows可以跑在PV模式下面。使得跑windows虚拟化方案。

硬件虚拟化：

引入新的CPU模式用来运行guest

Guest mode vs host mode

Non-root mode vs root mode on x86

Vmxon指令，vmxof指令可以在guest mode和host mode之间转换

硬件虚拟化+PV（部分虚拟化）

1：减少vm-exit

2：更快的IO

3：精确的time-keeping

4：搞笑的spinlock

5：接口

Virtio

MSR扩展

其他的虚拟化：

VT-d（DMA、IRQ remapping）

Pass-through （sr-iov）

Ive-migation

虚拟化的书籍：1）intel 系统虚拟化 2）kvm的vtr

轻量级虚拟化对互联网公司更有利。Container 基于cgroup的docker

Vmware Exsi

Fedora使用虚拟化：

<http://fedoraproject.org/wiki/Getting_started_with_virtualization/zh-cn>

## 算法篇：

Hlist中的prev只想前一个next的指针，这样直接用\*(node->pprev)=node就可以直接修改或者访问前驱的next的节点了

## 问题

## 知识篇：

1：安装linux（LFS和组合）

2：linux的使用（shell等等）

3：linux的启动步骤

Grub，init进程

4：一个ko的书写

### 2014.2.11

内核的学习方法：

1. 全面了解内核框架
2. 找一个感兴趣了研究
3. 融入内核社区
4. 坚持

Vim+cscope+ctags

使用Readme /Kconfig/Makefile是将关注的代码范围缩小

git blame可以比较git中的差异

如果你看不懂kernel代码了，不妨用git blame看看它为什么要做这些的修改。  
很多问题可以在patch的changelog中找到

<http://www.cnblogs.com/biyeymyhjob/archive/2012/07/20/2601655.html>

### 2014.2.19

编译linux内核：

1. make defconfig
2. make –jn
3. make module
4. make module\_install
5. make install

第一个ko

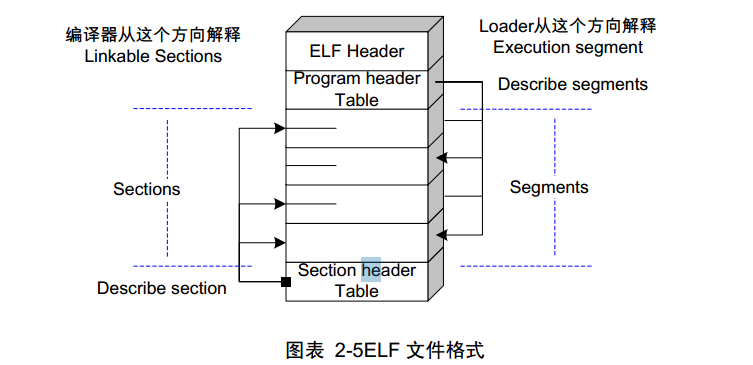
1. makefile的规则
2. 代码的框架

do\_initcalls函数：

1. include/init.h里面

ELF文件的格式：

1. relocatbale:由编译器和汇编器生成，由linker处理它
2. Executable:所有的重定位和符号解析都完成了，只有共享库在运行啥时候才能解析
3. Share object：包含linker需要的符号信息和运行时刻所需的代码



Ld –verbose :用户态的ld script

Arch/i386/kernel/vmlinux.lds.S:内核态的ld script

### 2014.2.24

进程描述的结构体

struct task\_struct

struct pid

struct upid

struct thread\_info

1)process creation

态线程处理函数：clone fork vfork exec\*

内核态的线程：内核态使用的线性地址大于PAGE\_OFFSE.

普通进程用的是4G的线性空间

/boot/config-`uname -r`.x86\_64 这个是内核编译时候的配置文件

kernel\_thread

问题：

内核在什么时机会自己创建进程/线程（中断，用户调用）

内核创建的线程/进程的方式与用户态创建线程进程分别是什么？区别是什么？

kthreadd、migration（任务前一的线程），

ksoftirqd（软中断线程），watchdog（开门狗线程）

内核的线程和用户的线程生成的区别：

了解如下函数的执行过程

do\_fork()

copy\_process

选择一个pid

初始化 vfork修改CLONE\_VFORK标记为1

wake\_up\_new\_task

CLONE\_VFORK\_SET?-----wait\_for\_completion

copy\_process的过程

check flages

dump\_task\_

fork\_init

#define current get\_current()

execve()->sys\_execve()

使用系统调用来执行一个可执行程序？？

do\_execve

打开可执行文件 open\_exec()

bprm\_init

sched

sched\_exec()

prepare\_binprm()

copy环境变量

search\_binary\_handler()

process termination

exit()->do\_exit

exit() and \_exit()

无论用户态还是内核态：那么多的系统调用，我们通过什么好的方式来获得我们需要的系统调用呢？

Linux中文件系统的mode：

-：表示文件

d:表示目录

b：表示块儿设备

c：表示字符设备

l：表示连接

s：表示套接字：用来给不同机器之间不同进程间消息交互的。

在centos中，yum install crash 如果要使用这个工具需要包含kerenl-debuginfo

Yum install kernel-debuginfo

中断上下文与进程上下文的区别：进程上下文从用户发起，进入系统调用，或者没有进入内核，就在用户态执行。而中断上下文是在当前的进程上下文中进行的，借用当前上下文的栈，直接执行的。

### 2014.2.25

Kdump,perf, make menuconfig下面的kernel hacking里找有内存方面的

Intel的x86开发手册

<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

2.6.32内核中，排队自旋锁

自旋锁在初始化的时候会吧自己初始化为0

Spin\_lock:会首相把自己的slock付给本的变量，并把slock+1，然后不停的比较本地变量的高8bit和低8bit（高8bit表示我的顺序，低8bit表示当前的顺序），同时还要不停的将那个实际的slock的低8bit复制到我的本地变量来。

Spin\_unlock:的作用是，直接讲低8bit进行加1处理。

所以核心思想就是：高8bit存储的是我自己是第几位，低8bit存储的是当前轮到第几位了。如果这两个相等了，则就是轮到我了

### 2014.2.27

把linux安装到U盘中:

1:把U盘分区分两个，一个可以供在win下使用（），另一个用来安装Linux系统

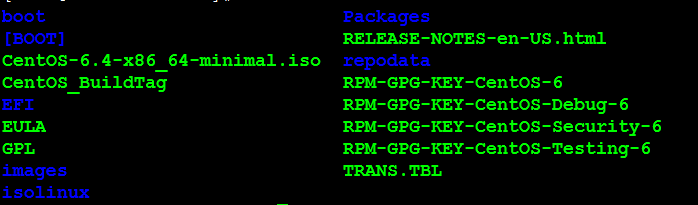
在linux系统中，接入U盘，用fdisk命令，将U盘分成2个区

* 1. 在linux系统中插入U盘后，一般会是/dev/sdb会是你的第二个盘设备（如果你之前只有一个盘的话）
  2. Fdisk –l可以查看到你的u盘具体是那个，我的是/dev/sdb
  3. Fdisk /dev/sdb 用fdisk对u盘分区，我们打算建立2个分区，1个分区用于win；1个分区用于安装linux。输入fdisk /dev/sdb后，就进入了fdisk的命令行用d删除所有分区，n建立2个主分区，win可以小一些;这样会出来2个分区，/dev/sdb1和/dev/sdb2
  4. 格式化u盘，mkfs.vfat /dev/sdb1,第二个分区不用格式化，在安装的时候可以格式化

2:把grub安装到u盘中

* 1. 安装grub到u盘，mkdir /mnt/sdb1 ;mount /dev/sdb1 /mnt/sdb1; grub-install –root-directory=/mnt/sdb1 /dev/sdb;这样子你把电脑改为U盘启动现在就可以进入grub界面了

3:把linux iso文件和解压后的iso的所有文件放入win下可以使用的目录中；（我是在windows下操作的，就把你的linux.iso文件放入win目录，以及把linux.iso解压后的文件也放入到目录下；这两个操作都放到根目录下）如下图所示



注：在安装grub时候会成圣一些boot文件，解压iso后也会有boot文件，不要用解压后的文件覆盖了，grub-install的文件，这些用grub-install生成的，其他就完全copy就行了

4:把pc机改为U盘启动，然后安装linux到U盘。

1. 启动后，进入grub，这时候在grub中输入 root(hd0,0)
2. Setup (hd0)
3. Setup (hd0,0)
4. Setup (hd0,1)
5. Kernel /isolinux/vmlinuz
6. Initrd /isolinux/initrd.img
7. Boot
8. 下面就可以安装了，在这里都可以用默认配置，唯一就是在选择安装的目录时候选择/dev/sda2,这个就是刚才咱们建立的分区，但是没有初始化的分区
9. 安装完成后重启就行了。

Which demo

rpm -qf \sbin\demo

rpm -qli demo

一些命令：nm/ldd/objdump/od

<http://adam8157.info/> 里面有一篇讲解kprob调试内核的!!

### 2014.3.20

<http://www.cnblogs.com/fnng/archive/2012/01/07/2315685.html>

初次建立git hub的使用

**Global setup:**  
 Set up git  
  git config --global user.name "Your Name"  
  git config --global user.email defnngj@gmail.com  
        
**Next steps:**  
  mkdir hibernaet-demo2  
  cd hibernaet-demo2  
  git init  
  touch README  
  git add README  
  git commit -m 'first commit'  
  git remote add origin git@github.com:defnngj/hibernaet-demo2.git  
  git push -u origin master

**Existing Git Repo?**  
  cd existing\_git\_repo  
  git remote add origin git@github.com:defnngj/hibernaet-demo2.git  
  git push -u origin master

**Importing a Subversion Repo?**  
  Click here   
When you're done:  
  Continue

讲解kernel的make的Makefile文件：<http://www.groad.net/bbs/thread-3844-1-1.html>

内核配置选项：<http://works.jinbuguo.com/kernel/longterm-3_10-options.html>

Make localmodconfig的讲解http://www.lijingquan.net/linux-kernel-localconfig.html

Gcc编译时候会有四个步骤：

1：预编译 gcc -E //生成预编译代码

2：生成汇编 gcc -S //生成汇编代码

3：编译： gcc -c //编译成二进制文件

4：连接 gcc //连接生成可执行文件

S

### 2014.4.3

<http://lenky.info/archives/2012/05/28/1688> 讲解了编译内核时候的几个有用的命令：

升级[Linux内核](http://lenky.info/archives/tag/linux%e5%86%85%e6%a0%b8)的操作已经变得很简单，基本的几个命令即可搞定：[make](http://lenky.info/archives/tag/make) menuconfig、make、make modules、make modules\_install、make install。除了这几个常用选项，make命令还有另外的几个选项对于我们也是十分有帮助，在内核源码树目录下执行make help可以看到这些选项，不过下面还是介绍几个我认为比较有用的选项：  
1，make V=1，这个选项能显示出编译过程中的详细信息，即是verbose编译模式，在之前的《[显示make编译信息](http://lenky.info/2012/04/18/%E6%98%BE%E7%A4%BAmake%E7%BC%96%E8%AF%91%E4%BF%A1%E6%81%AF/)》文章里曾介绍过修改makefile来达到这个目的，但make V=1更简单方便，而make V=0或直接make（V=0是默认值）则是quiet编译模式，只会显示出简单的编译信息。

|  |  |
| --- | --- |
| 1 | [root@localhost linux-2.6.30]# make V=1 |

2，make dir/：指定编译目标，编译指定的某个目录：

|  |  |
| --- | --- |
| 1 | [root@localhost linux-2.6.30]# make drivers/scsi/ |

3，make dir/file.ko：指定编译目标，编译指定的某个模块：

|  |  |
| --- | --- |
| 1 | [root@localhost linux-2.6.30]# make drivers/net/igb/igb.ko |

4，make dir/file.[ois]：指定编译目标，编译指定的某个文件为目标文件、头文件展开文件、汇编文件：

|  |  |
| --- | --- |
| 1  2  3 | [root@localhost linux-2.6.30]# make drivers/net/igb/igb\_main.o  [root@localhost linux-2.6.30]# make drivers/net/igb/igb\_main.i  [root@localhost linux-2.6.30]# make drivers/net/igb/igb\_main.s |

5，make checkstack：函数栈静态检查  
6，make includecheck：头文件重复静态检查  
最后看下命令make help执行的显示信息：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91 | [root@localhost linux-2.6.30]# make help  Cleaning targets:    clean       - Remove most generated files but keep the config and                      enough build support to build external modules    mrproper    - Remove all generated files + config + various backup files    distclean   - mrproper + remove editor backup and patch files    Configuration targets:    config      - Update current config utilising a line-oriented program    menuconfig      - Update current config utilising a menu based program    xconfig     - Update current config utilising a QT based front-end    gconfig     - Update current config utilising a GTK based front-end    oldconfig   - Update current config utilising a provided .config as base    silentoldconfig - Same as oldconfig, but quietly    randconfig      - New config with random answer to all options    defconfig   - New config with default answer to all options    allmodconfig    - New config selecting modules when possible    allyesconfig    - New config where all options are accepted with yes    allnoconfig     - New config where all options are answered with no    Other generic targets:    all         - Build all targets marked with [\*]  \* vmlinux     - Build the bare kernel  \* modules     - Build all modules    modules\_install - Install all modules to INSTALL\_MOD\_PATH (default: /)    firmware\_install- Install all firmware to INSTALL\_FW\_PATH                      (default: $(INSTALL\_MOD\_PATH)/lib/firmware)    dir/            - Build all files in dir and below    dir/file.[ois]  - Build specified target only    dir/file.ko     - Build module including final link    modules\_prepare - Set up for building external modules    tags/TAGS   - Generate tags file for editors    cscope      - Generate cscope index    kernelrelease   - Output the release version string    kernelversion   - Output the version stored in Makefile    headers\_install - Install sanitised kernel headers to INSTALL\_HDR\_PATH                      (default: /usr/src/linux-2.6.30/usr)    Static analysers    checkstack      - Generate a list of stack hogs    namespacecheck  - Name space analysis on compiled kernel    versioncheck    - Sanity check on version.h usage    includecheck    - Check for duplicate included header files    export\_report   - List the usages of all exported symbols    headers\_check   - Sanity check on exported headers    headerdep       - Detect inclusion cycles in headers    Kernel packaging:    rpm-pkg         - Build both source and binary RPM kernel packages    binrpm-pkg      - Build only the binary kernel package    deb-pkg         - Build the kernel as an deb package    tar-pkg         - Build the kernel as an uncompressed tarball    targz-pkg       - Build the kernel as a gzip compressed tarball    tarbz2-pkg      - Build the kernel as a bzip2 compressed tarball    Documentation targets:   Linux kernel internal documentation in different formats:    htmldocs        - HTML    pdfdocs         - PDF    psdocs          - Postscript    xmldocs         - XML DocBook    mandocs         - man pages    installmandocs  - install man pages generated by mandocs    cleandocs       - clean all generated DocBook files    Architecture specific targets (x86):  \* bzImage      - Compressed kernel image (arch/x86/boot/bzImage)    install      - Install kernel using                    (your) ~/bin/installkernel or                    (distribution) /sbin/installkernel or                    install to $(INSTALL\_PATH) and run lilo    fdimage      - Create 1.4MB boot floppy image (arch/x86/boot/fdimage)    fdimage144   - Create 1.4MB boot floppy image (arch/x86/boot/fdimage)    fdimage288   - Create 2.8MB boot floppy image (arch/x86/boot/fdimage)    isoimage     - Create a boot CD-ROM image (arch/x86/boot/image.iso)                    bzdisk/fdimage\*/isoimage also accept:                    FDARGS="..."  arguments for the booted kernel                    FDINITRD=file initrd for the booted kernel      i386\_defconfig           - Build for i386    x86\_64\_defconfig         - Build for x86\_64      make V=0|1 [targets] 0 => quiet build (default), 1 => verbose build    make V=2   [targets] 2 => give reason for rebuild of target    make O=dir [targets] Locate all output files in "dir", including .config    make C=1   [targets] Check all c source with $CHECK (sparse by default)    make C=2   [targets] Force check of all c source with $CHECK    Execute "make" or "make all" to build all targets marked with [\*]  For further info see the ./README file  [root@localhost linux-2.6.30]# |

### 问题

# 其他

大神的进google经历<http://www.cnblogs.com/figure9/archive/2013/01/09/2853649.html>

<http://blog.csdn.net/Hackbuteer1/article/list/1>

四城，大成网，齐家

四城的建材团购一般价格给力

地板：必美

吊顶：泰力达

<http://www.17zhuang.com/thread-1245138806-1-1.html>

<http://blog.csdn.net/sabalol/article/details/1645512>

[凌码信息技术（上海）有限公司](http://search.51job.com/list/co,c,133338,000000,10,1.html)

3月8号惠万家活动地点：红星美凯龙双楠店负1层中庭

3月9号德格橱柜专场：红星美凯龙双楠店9楼A区901-906 （二环路西一段6号 注意！不是佳灵路

3月15号联邦吊顶专场：锣锅巷122号云龙大厦1509

3月16号丹菲儿衣柜专场：老南门大桥锦里东路10号（省政协对面

3月22号瑞士卢森地板专场：南门富森美家居建材MALL一楼中庭

装修材料：

卫生间：

1. 吊顶；2）五金件；3）当水槽；4）淋雨隔断；5）马桶；6）地漏‘7）化妆台；8）淋雨/浴霸；9）瓷砖

厨房：

1. 橱柜；2）吊柜3）吊柜4）油烟机；5）灶具；6）洗手台；7）瓷砖；8）洗拖把的池子

卧室：

1. 地板；2）衣柜；3）空调；4）床；5）包阳台；

客厅：

1. 瓷砖；2）电视柜；3）沙发；4）书桌（细化都有什么桌子【餐桌、书桌、电视柜前面的桌子】）；5）储物柜；6）桌子；7）吊顶

入户花园:

1）鞋柜；2）窗帘；

其他物品：

灯具；板凳；门槛石；开关面板；

<http://blog.csdn.net/snowtiger999/article/details/8988873>

<http://linux.chinaitlab.com/administer/953925.html>

<http://linux.chinaitlab.com/kernel/821107.html>

