Lecture2 Basic Knowledge of Linux Kernel Programming

PEKING UNIVERSITY

Basic Knowledge of Linux Kernel Programming

- **Kernel Compiling Explanation**
- Kernel Debugging Techniques
- Module Programming
- AT&T Assembly Language
- Linux Booting Process
- Tools for Linux Kernel Source Code Reading
- Kernel & Git

PEKING UNIVERSITY

内核功能可以通过**Kconfig**进行配置。

- 内核源码树的目录下都有两个文档**Kconfig**和**Makefile**。
  - 分布到各目录的**Kconfig**构成了一个分布式的**内核配置数据库**，每个**Kconfig**分别描述了所属目录源文档相关的**内核配置菜单**。
  - 在内核配置**make menuconfig(**或**xconfig**等**)**时，从**Kconfig**中**读出**菜单，用户**选择**后**保存**到**.config**内核配置文档中。

简单的HelloWorld.c程序

- 修改相应目录下的Kconfig文件和Makefile文件
  - Kconfig文件

```
config  HELLOWORLD
        bool “helloworld”
        help
        .........
```

  - #Makefile文件

```
obj-$(CONFIG_HELLOWORLD)+=HelloWorld.o
```

最常见的是tristate和bool，分别对应于配置界面中[ ]和< >选项

tristate：可取y、n、m
bool：可取y、n
string：取值为字符串
hex：取值为十六进制数据
int：取值为十进制数据

Basic Knowledge of Linux Kernel Programming

- Kernel Compiling Explanation
- **Kernel Debugging Techniques**
- Module Programming
- AT&T Assembly Language
- Linux Booting Process
- Tools for Linux Kernel Source Code Reading
- Kernel & Git

Kernel Debugging Techniques

- 内核中的调试支持
- Printk
- kprobe
- strace
- Kernel Oops Messages
- The magic SysRq key
- 调试器及相关工具

内核中的调试支持

- 前提：掌握如何使用自己编译的内核
- 在“kernel hacking”菜单中
  - CONFIG_DEBUG_KERNEL
  - CONFIG_DEBUG_INFO
  - CONFIG_DEBUG_SLAB
  - CONFIG_DEBUG_PAGEALLOC
  - CONFIG_DEBUG_SPINLOCK
  - CONFIG_MAGIC_SYSRQ
- General setup
  - CONFIG_KALLSYMS/CONFIG_KALLSYMS_ALL

SysRq 魔法键

- 需要在配置内核时，选择**Magic SysRq Key**。然后在新内核启动后，使用如下命令激活**SysRq**功能。

  #echo “1” > /proc/sys/kernel/sysrq

- 当系统挂起时，可以使用：

  Alt+PrintScreen+[CommandKey]键

第三键及相应功能

- **p** 打印处理器消息.
- **t** 打印当前任务列表.
- **m** 打印内存信息.
- **b** **boot.** 立刻重启系统. 确认先同步和重新加载磁盘
- **k** 调用"安全注意键"(**SAK**) 功能. **SAK** 杀掉在当前控制台的所有运行的进程, 给你一个干净的终端.
- **s** 进行一个全部磁盘的紧急同步.
- **u** **umount.** 试图重新加载所有磁盘在只读模式. 这个操作, 常常在 **s** 之后马上调用, 可以节省大量的文件系统检查时间, 在系统处于严重麻烦时.
- **r** 关闭键盘原始模式; 用在一个崩溃的应用程序(例如 **X** 服务器)可能将你的键盘搞成一个奇怪的状态.

Printk

- 通过打印调试
  - **printk() 是调试内核代码时最常用的一种技术，和printf相似，用于内核打印消息。**
  - **For example**

```
printk(KERN_DEBUG “Here I am: %s:%d\n”,
__FUNCTION__, __LINE__);
```

  - **printk根据日志级别（loglevel）对消息进行分类，Loglevels （include/linux/printk.h中定义）**

```
#define KERN_EMERG    0/*紧急事件消息，系统崩溃之前提示，表示系统不可用*/
#define KERN_ALERT    1/*报告消息，表示必须立即采取措施*/
#define KERN_CRIT     2/*临界条件，通常涉及严重的硬件或软件操作失败*/
#define KERN_ERR      3/*错误条件，驱动程序常用KERN_ERR来报告硬件的错误*/
#define KERN_WARNING  4/*警告条件，对可能出现问题的情况进行警告*/
#define KERN_NOTICE   5/*正常但又重要的条件，用于提醒*/
#define KERN_INFO     6/*提示信息，如驱动程序启动时，打印硬件信息*/
#define KERN_DEBUG    7/*调试级别的消息*/
```

Cont.

- **Printk**
  - 未指定优先级的 **printk** 语句采用的默认级别是 **DEFAULT_MESSAGE_LOGLEVEL**（**kernel/printk.c** ）。
  - **console_loglevel** 定义了哪一个优先级的消息被输出到 **console**（日志输出级别小于**console_loglevel** ）

    console_loglevel 可以通过系统调用**sys_syslog**来修改
  - 也可以通过对文件**/proc/sys/kernel/printk**的访问来读取和修改

查看：

```
pkuss@ubuntu:~/linux-3.4.106$ cat /proc/sys/kernel/printk
4       4       1       7
pkuss@ubuntu:~/linux-3.4.106$
```

控制台日志级别

缺省消息日志级别

最小控制台日志级别

缺省控制台日志级别

修改：

```
root@ubuntu:~# echo 8 > /proc/sys/kernel/printk
```

小作业1：修改后cat /proc/sys/kernel/printk的输出是什么？

Kprobe

- 内核探测
  - Kprobe机制是内核提供的一种调试机制，它能够在不修改现有代码的基础上，灵活的跟踪内核函数的执行
  - Kprobe提供了三种形式的探测点

    1、kprobe

    2、jprobe　　使能kprobe：CONFIG_KPROBES

    3、kretprobe
  - 对于kprobe功能的实现主要利用了内核中的两个功能特性：异常（主要是int 3），单步执行（EFLAGS中的TF标志）。

    基本流程：

    1）备份被探测指令

    2）用异常指令替换被探测函数的指令，并将相关函数注册到异常处理函数链

    3）当运行到探测指令时，执行异常陷入（trap），调用kprobe的异常处理函数

    4）执行pre_handler钩子

    5）单步调试被探测函数的备份

    6）代码返回，执行原有指令，执行结束后触发单步异常

    7）清除单步标志，执行post_handler流程，并最终返回

Systemtap

- 基于**kprobe**
  - **krpobe** 需要用模块的方式编译并加载
- **Systemtap**通过脚本的方式自动生成劫持代码并自动加载和收集数据
  - 用户只需要编写脚本, 就可以完成调试并动态分析内核

strace

- **Using strace**
  - **strace命令是一个功能强大的工具，它可以显示程序所调用的所有系统调用。**

    它不仅可以显示调用，而且还能显示调用的参数，以符号方式显示返回值。
  - **# strace ls /dev**

```
root@ubuntu: ~
root@ubuntu:~#
root@ubuntu:~#
root@ubuntu:~# strace ls /dev
execve("/bin/ls", ["ls", "/dev"], [/* 24 vars */]) = 0
brk(0)                                  = 0xd35000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f6
519a1e000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=62622, ...}) = 0
mmap(NULL, 62622, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f6519a0e000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20T\0\0\0\0\0\0"..., 832
) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=121936, ...}) = 0
mmap(NULL, 2221680, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f
65195df000
mprotect(0x7f65195fc000, 2093056, PROT_NONE) = 0
mmap(0x7f65197fb000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYW
RITE, 3, 0x1c000) = 0x7f65197fb000
mmap(0x7f65197fd000, 1648, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONY
```

ftrace

- 功能最强大的调试、跟踪手段
- 需要开启相关内核配置选项
- 通过**debugfs**向用户空间提供访问接口
- 在**/sys/kernel/debug/trace**目录下提供了各种跟踪器和事件
- 利用**gcc**的**profile**特性在函数入口处添加了一段代码，**ftrace**重载这段代码实现**trace**功能

Kernel Oops Messages

- 很多错误都是**NULL**指针引用或使用其他不正确的指针数值。这些错误通常会导致一个**oops**消息。

- **oops**显示故障时的处理器状态，模块**CPU**寄存器内容，页描述符表的位置，以及其他信息。
  - For example:

```
ssize_t faulty_write (struct file *filp, const char __user
    *buf,size_t count, loff_t *pos)
{
/* make a simple fault by dereferencing a NULL
    pointer */
*(int *)0 = 0;
return 0;
}
```

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU:    0
EIP:    0060:[<d083a064>]    Not tainted
EFLAGS: 00010246   (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
eax: 00000000   ebx: 00000000   ecx: 00000000   edx: 00000000
esi: cf8b2460   edi: cf8b2480   ebp: 00000005   esp: c31c5f74
ds: 007b   es: 007b   ss: 0068
Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
       ffffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
       00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
 [<c0150558>] vfs_write+0xb8/0x130
 [<c0150682>] sys_write+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb

Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0
```

objdump反汇编，查看该函数相关位置的指令

- BUG（）
- BUG_ON()

可用ksymoops/CONFIG_KALLSYMS辅助消息查看

调试器和相关工具

- 上述几种内核调试方法：
  - 简单方便
  - Linux环境就可以使用，不需要太复杂的设置
  - 只能对内核进行监控，查询内核的状态；
  - 获得的信息很有限；
  - 做不到真正意义上的“调试”
    - 无法动态地修改系统的各种变量；
    - 无法在内核中进行断点设定、单步执行等操作。

调试器和相关工具——KDB

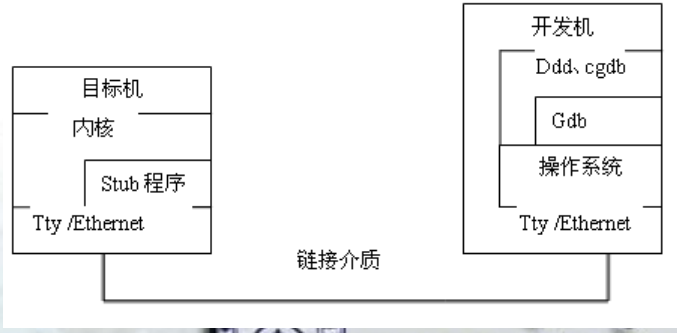
- **kdb是一个Linux系统的内核调试器，遵循GPL**
- **kdb support was added to the mainline Linux kernel in version 2.6.35.**
- 适合于调试内核空间的程序代码，原本是汇编级调试
  - **kdb支持包括x86(IA32)、IA64和MIPS在内的体系结构。**
- 目前已与**KGDB**合并，不再提供汇编级调试
- 使用方法：
  - **bp, go, bt, mds, mm ...**

```
[0]kdb> bt
ESP EIP Function (args)
0xcdbddf74 0xd087c5dc [scull]scull_read
0xcdbddf78 0xc0150718 vfs_read+0xb8
0xcdbddfa4 0xc01509c2 sys_read+0x42
0xcdbddfc4 0xc0103fcf syscall_call+0x7
[0]kdb>
```

调试器和相关工具——KGDB

- **kgdb 提供了一种使用 gdb调试 Linux 内核的机制，源码级调试**
- **kgdb was added to the mainline Linux kernel in version 2.6.26.**
- **使用KGDB可以象调试普通的应用程序那样，在内核中进行设置断点、检查变量值、单步跟踪程序运行等操作。**
- **使用KGDB调试时需要两台机器，一台作为开发机（Development Machine）,另一台作为目标机（Target Machine），两台机器之间通过串口或者以太网口相连。**

- 调试过程中，被调试的内核运行在目标机上，gdb调试器运行在开发机上。
- 目前，kgdb发布支持i386、x86_64、32-bit PPC、SPARC等几种体系结构的调试器。



http://www.kgdb.info

In order to support KDB, “KGDB” support must be turned on first (even if you aren’t using kgdb/gdb)：

- CONFIG_DEBUG_KERNEL=Y – includes debug information in the kernel compilation – required for basic kernel debugging support
- CONFIG_KGDB=Y – turn on basic kernel debug agent support
- CONFIG_KGDB_SERIAL_CONSOLE=Y – to share a serial console with kgdb.
  - Sysrq-g must be used to break in initially.
  - Selecting this will automatically set:
    - CONFIG_CONSOLE_POLL=Y
    - CONFIG_MAGIC_SYSRQ=Y – turn on MAGIC-SYSRQ key support
- CONFIG_KGDB_KDB=Y – actually turn on the KDB debugger feature

If your machine starts a serial console on ttyS0, you can bind kdb/kgdb to this serial console by writing the string “ttyS0″ to /sys/module/kgdboc/parameters/kgdboc. The kernel will respond with a message indicating that that driver is registered：

```
$ echo ttyS0 >/sys/module/kgdboc/parameters/kgdboc
kgdb: Registered I/O driver kgdboc.
```

调试器和相关工具——利用虚拟机调试

- VMware虚拟机下调试内核
  - 简介：在VMware上运行两个运行目标内核的虚拟机，同时使用KGDB来调试Linux内核。
- 使用J-LINK仿真器加vmware
  - 简介：使用J-LINK仿真器通过kgdb串口来调试内核。
- DDD + QEMU + busybox 来验证内核
  - 简介：ddd + qemu +busybox 这个工具套件可以建立一个运行目标内核的轻量级的虚拟机。并可以像调试普通应用程序一样对虚拟机中的内核代码进行单步调试。

Basic Knowledge of Linux Kernel Programming

- Kernel Compiling Explanation
- Kernel Debugging Techniques
- **Module Programming**
- AT&T Assembly Language
- Linux Booting Process
- Tools for Linux Kernel Source Code Reading
- Kernel & Git

PEKING UNIVERSITY

Module编程基础

A. Module基础

B. 传递参数

C. 导出符号表

D. 动手实践

A. Module基础

- 模块的全称是“动态可加载内核模块”（**Loadable Kernel Module，LKM**）
- 模块在内核空间运行
- 模块实际上是一种目标对象文件
  - ✓不能独立运行，但是其代码可以在运行时链接到系统中作为内核的一部分运行或从内核中取下，从而可以动态扩充内核的功能
- 这种目标代码通常由一组函数和数据结构组成

‘Superuser’ privileges

- Modifying a running kernel is ‘risky’
- Only authorized ‘system administrators’ are allowed to install kernel modules

内核模块的优点

- 使得内核更加紧凑和灵活
- 修改内核时，不必全部重新编译整个内核。系统如果需要使用新模块，只要编译相应的模块，然后使用insmod将模块装载即可
- 模块的目标代码一旦被链接到内核，它的作用域和静态链接的内核目标代码完全等价

内核模块的缺点

- 由于内核所占用的内存是不会被换出的，所以链接进内核的模块会给整个系统带来一定的性能和内存利用方面的损失；
- 装入内核的模块就成为内核的一部分，可以修改内核中的其他部分，因此，模块的使用不当会导致系统崩溃；
- 为了让内核模块能访问所有内核资源，内核必须维护符号表，并在装入和卸载模块时修改符号表；
- 模块会要求利用其它模块的功能，所以，内核要维护模块之间的依赖性.

内核模块与应用程序的区别

	C语言程序	模块
运行	用户空间	内核空间
入口	main()	module_init()指定;
出口	无	module_exit()指定;
编译	gcc –c	Makefile
链接	ld	insmod
运行	直接运行	insmod
调试	gdb	kdb, kgdb等

Linux编程风格

- Documentation/CodingStyle
- Scripts/checkpatch.pl 检查

相关命令

- **insmod <module.ko> [module parameters]**
  - **– Load the module**
- **rmmod**
  - **– Unload the module**
- **lsmod**
  - **– List all modules loaded into the kernel**
- **modprobe [-r] <module name>**
  - **– Load the module specified and modules it depends**

insmod

- 调用**insmod**程序将把需要插入的模块以目标代码的形式装载到内核中。

- 在装载的时候，**insmod**会自动运行我们在**module_init()**函数中定义的过程。
  - 注意，只有超级用户才能使用这个命令

- #insmod [path]modulename.ko

rmmod

- **rmmod将已经装载到内核的模块从内核中移出**
- **rmmod会自动运行在module_exit()函数中定义的过程**
- #rmmod　[path]modulename

lsmod

- **lsmod 显示当前系统中正在使用的模块信息**
- **实际上这个程序的功能就是读取/proc文件系统中的文件/proc/modules中的信息。**
  - **所以这个命令和cat /proc/modules等价。**
- #lsmod

模块依赖

- 一个模块A引用另一个模块B所导出的符号，我们就说模块B被模块A引用。

- 如果要装载模块A，必须先要装载模块B。否则，模块B所导出的那些符号的引用就不可能被链接到模块A中。

- 这种模块间的相互关系就叫做模块依赖。

modprobe

- 与**insmod**类似，用来装载模块到内核。

- 基本思想：
  - 考虑模块是否引用当前内核中不存在的符号
  - 如果有，搜索定义了这些符号的其他模块，同时将这些模块装载到内核。

最简单的例子

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
static int __init hello_init(void)
{
        printk(KERN_INFO "Hello world\n");
        return 0;
}
static void __exit hello_exit(void)
{
        printk(KERN_INFO "Goodbye world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

模块程序介绍

- **Init/exit**
  - 宏：**module_init/module_exit**
  - 声明模块初始化及清除函数所在的位置
  - 装载和卸载模块时，内核可以自动找到相应的函数

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

模块程序介绍

static int __init hello_init(void)
static void __exit hello_exit(void)

- static声明，因为这种函数在特定文件之外没有其它意义
- __init标记
  表明该函数只在初始化期间使用。
  模块装载后，将该函数占用的内存空间释放
- __exit标记
  该代码仅用于模块卸载。

编译模块

– Makefile文件

```
obj-m += hello.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

命令：#make

– Module includes more files

hello-objs += a.o b.o 或 hello-y += a.o b.o

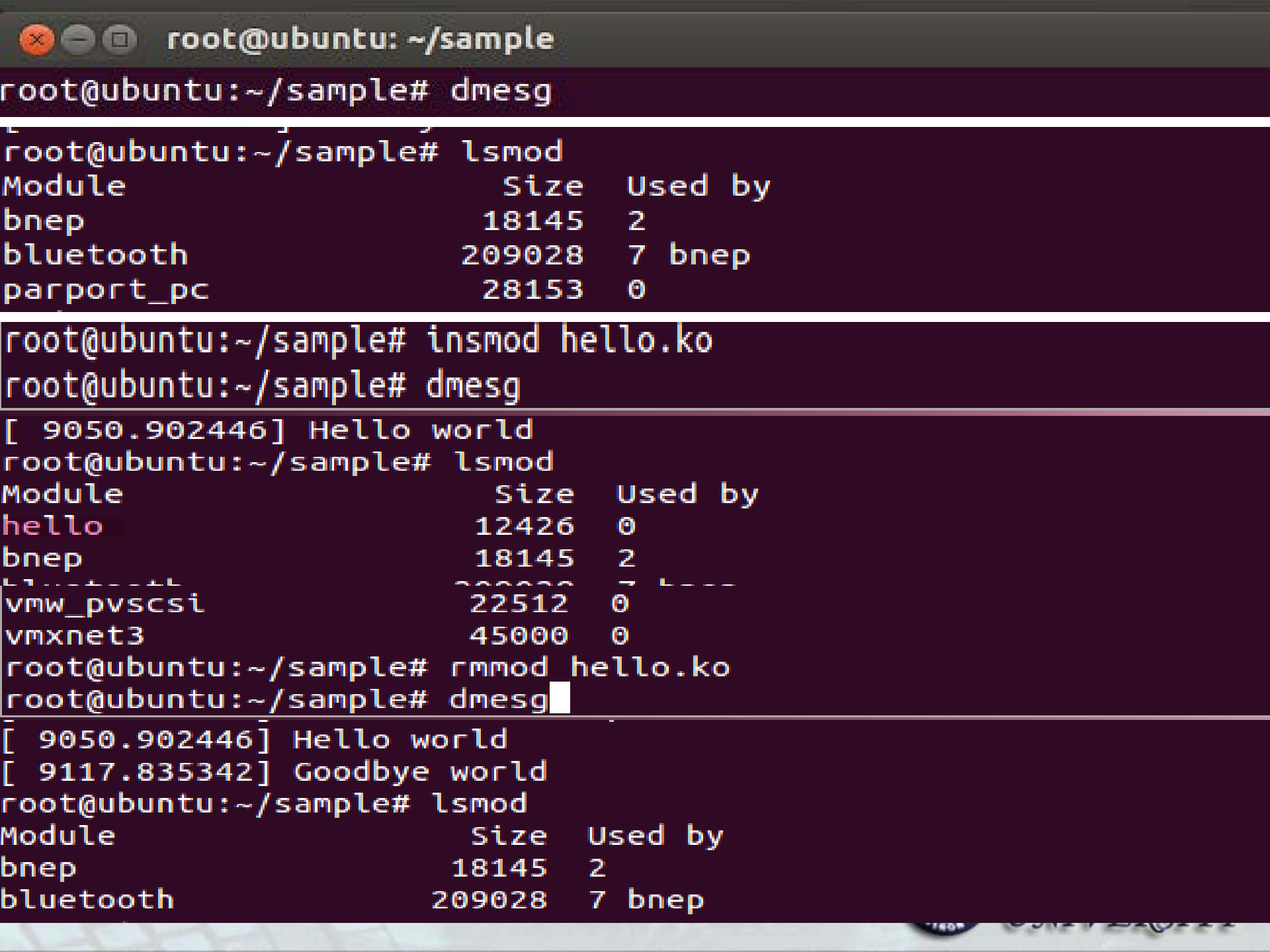
注意：all下一行需要有一个“Tab”键，不要写成空格或略去，不然系统无法识别，报错：nothing to be done for “all”。

```
root@ubuntu:~/sample# ls
hello.c  Makefile
root@ubuntu:~/sample# make
make -C /lib/modules/3.4.106/build M=/root/sample modules
make[1]: Entering directory `/home/songqing/linux-3.4.106'
  CC [M]  /root/sample/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/sample/hello.mod.o
  LD [M]  /root/sample/hello.ko
make[1]: Leaving directory `/home/songqing/linux-3.4.106'
root@ubuntu:~/sample# lsmod
Module                  Size  Used by
bnep                   18145  2
bluetooth             209028  7 bnep
parport_pc             28153  0
ppdev                  17031  0
snd_ens1371            25304  2
```

```
mptscsih               40290  1 mptspi
mptbase               101781  2 mptspi,mptscsih
vmw_pvscsi             22512  0
vmxnet3                45000  0
root@ubuntu:~/sample# ls
hello.c   hello.mod.c  hello.o   modules.order
hello.ko  hello.mod.o  Makefile  Module.symvers
root@ubuntu:~/sample#
```

装载和卸载模块

- 相关命令
  - lsmod
  - insmod hello.ko
  - rmmod hello.ko


root@ubuntu: ~/sample
root@ubuntu:~/sample# dmesg
root@ubuntu:~/sample# lsmod
Module Size Used by
bnep 18145 2
bluetooth 209028 7 bnep
parport_pc 28153 0
root@ubuntu:~/sample# insmod hello.ko
root@ubuntu:~/sample# dmesg
[9050.902446] Hello world
root@ubuntu:~/sample# lsmod
Module Size Used by
hello 12426 0
bnep 18145 2
vmw_pvscsi 22512 0
vmxnet3 45000 0
root@ubuntu:~/sample# rmmod hello.ko
root@ubuntu:~/sample# dmesg
[9050.902446] Hello world
[9117.835342] Goodbye world
root@ubuntu:~/sample# lsmod
Module Size Used by
bnep 18145 2
bluetooth 209028 7 bnep

B. 参数传递

- 有些模块需要传递一些参数
- 参数在模块加载时传递

  #insmod hello.ko test=2
- 参数需要使用**module_param**宏来声明

  module_param的参数：

  变量名称，类型以及访问许可掩码
- 支持的参数类型

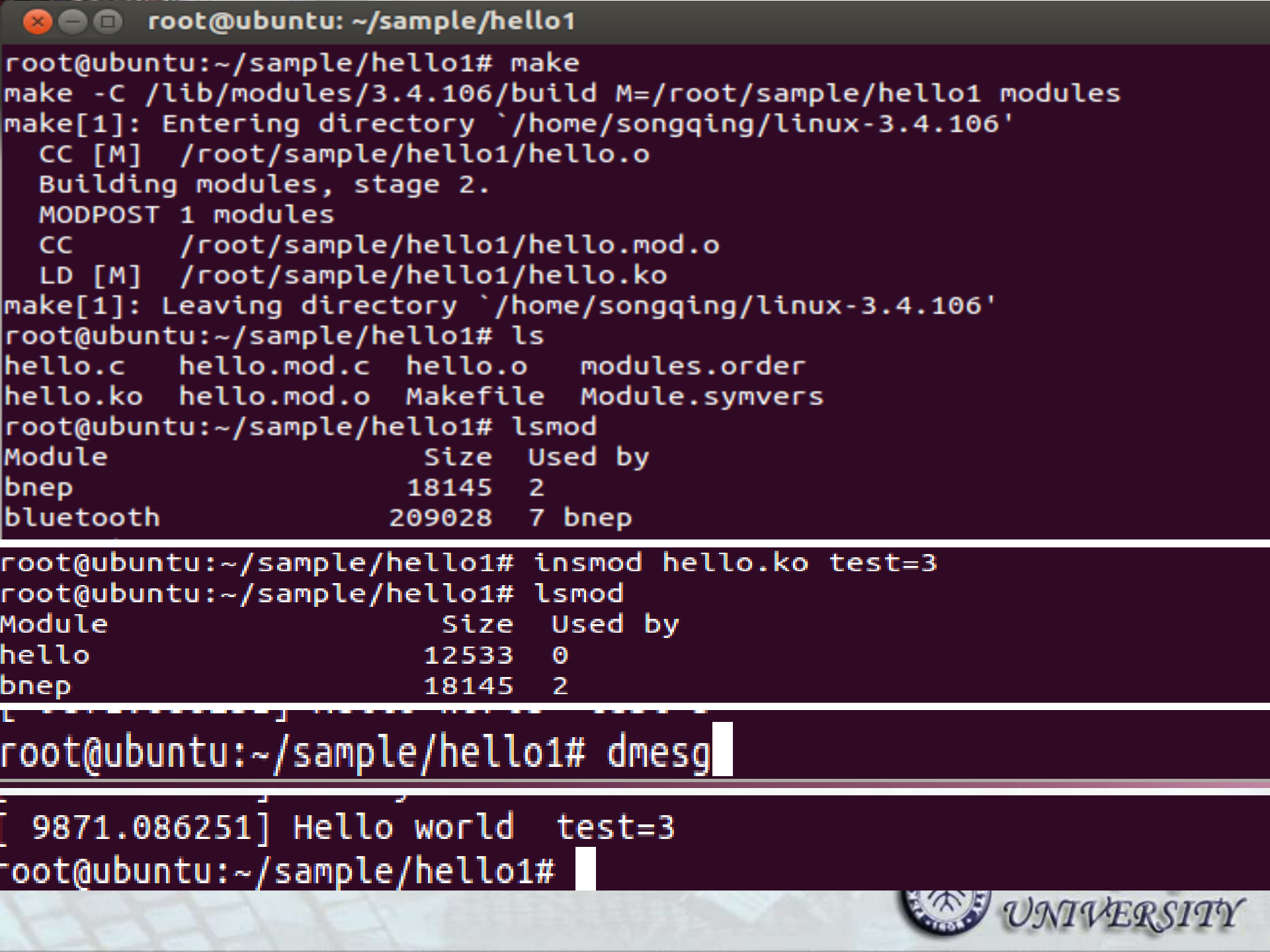
  Byte, short, ushort, int, uint, long, ulong, bool, charp

  Array (module_param_array(name, type, nump, perm))

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

static int test;
module_param(test, int, 0644);

static int __init hello_init(void)
{
    printk(KERN_INFO “Hello world  test=%d \n” , test);
    return 0;
}
static void __exit hello_exit(void)
{
    printk(KERN_INFO "Goodbye world\n");
}
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Test");
MODULE_AUTHOR("xxx");
module_init(hello_init);
module_exit(hello_exit);
```


root@ubuntu: ~/sample/hello1
root@ubuntu:~/sample/hello1# make
make -C /lib/modules/3.4.106/build M=/root/sample/hello1 modules
make[1]: Entering directory `/home/songqing/linux-3.4.106'
CC [M] /root/sample/hello1/hello.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/sample/hello1/hello.mod.o
LD [M] /root/sample/hello1/hello.ko
make[1]: Leaving directory `/home/songqing/linux-3.4.106'
root@ubuntu:~/sample/hello1# ls
hello.c hello.mod.c hello.o modules.order
hello.ko hello.mod.o Makefile Module.symvers
root@ubuntu:~/sample/hello1# lsmod
Module Size Used by
bnep 18145 2
bluetooth 209028 7 bnep
root@ubuntu:~/sample/hello1# insmod hello.ko test=3
root@ubuntu:~/sample/hello1# lsmod
Module Size Used by
hello 12533 0
bnep 18145 2
root@ubuntu:~/sample/hello1# dmesg
9871.086251] Hello world test=3
root@ubuntu:~/sample/hello1#

C. 导出符号表

- 如果一个模块需要向其他模块导出符号（方法或全局变量），需要使用：

  EXPORT_SYMBOL(name);

  EXPORT_SYMBOL_GPL(name);

  注意：符号必须在模块文件的全局部分导出，不能在函数部分导出。

  更多信息可参考 **<linux/module.h>**文件

- **Modules 仅可以使用由Kernel或者其他Modules导出的符号**
  不能使用Libc

- **/proc/kallsyms 可以显示所有导出的符号**

```
root@ubuntu: ~/sample
root@ubuntu:~/sample# cat /proc/kallsyms
ffffffffa0009000 d vmxnet3_driver_name  [vmxnet3]
ffffffffa0003a70 t vmxnet3_tq_destroy_all       [vmxnet3]
ffffffffa0004aa0 t vmxnet3_quiesce_dev  [vmxnet3]
ffffffffa0008580 r __mod_pci_device_table       [vmxnet3]
ffffffffa0004ee0 t vmxnet3_create_queues        [vmxnet3]
ffffffffa0006310 t vmxnet3_set_ethtool_ops      [vmxnet3]
ffffffffa0006250 t vmxnet3_set_features [vmxnet3]
ffffffffa0005460 t vmxnet3_force_close  [vmxnet3]
root@ubuntu:~/sample#
```

D. 2.6及后续版本内核中有关模块部分的改变

- 模块引用计数器
  - Linux2.4中在linux/module.h中定义了三个宏来维护实用计数：

    __MOD_INC_USE_COUNT　　当前模块计数加一
    __MOD_DEC_USE_COUNT　　当前模块计数减一
    __MOD_IN_USE　　计数非0时返回真
  - Linux2.6及后续版本内核中，模块引用计数器由系统自动维护
- 关于符号导出列表(list of exported symbols)
  - Linux2.4中会用EXPORT_NO_SYMBOLS宏，来表示不想导出任何变量或函数
  - Linux2.6及后续版本内核中这个宏也已经消失, 系统默认为不导出任何变量或函数

- 模块程序编译方法的改变
  - **Linux2.4**中命令为：
    gcc –Wall –DMODULE –D__KERNEL__ -DLINUX –c 源文件名**.c**

    其中：

    __KERNEL__：　即告诉头文件这些代码将在内核模式下运行。
    MODULE：　即告诉头文件要给出适当的内核模块的定义。
    LINUX：　并非必要，用在写一系列要在不止一个的操作系统上编译模块。
    -Wall：　显示所有**warning**信息。

  - **Linux2.6**及后续版本内核中必须写**Makefile**。通过**make**命令编译程序

- 用以加载的目标文件类型已经改变
  - **Linux2.4**中用以加载为模块的目标文件扩展名为**.o**
  - **Linux2.6**及后续版本内核中用以加载为模块的目标文件扩展名为**.ko**

Basic Knowledge of Linux Kernel Programming

- Kernel Compiling Explanation
- Kernel Debugging Techniques
- Module Programming
- **AT&T Assembly Language**
- Linux Booting Process
- Tools for Linux Kernel Source Code Reading
- Kernel & Git

PEKING UNIVERSITY

AT&T Assembly Language

- 与硬件相关部分的代码需要使用汇编语言
- 启动部分的代码有大小限制，
  - 精练的汇编可以缩小目标代码的**Size**。
- 需要被经常调用的代码，使用汇编提高性能

AT&T Assembly Language

- 源码中汇编语言出现在
  - .S 汇编文件
  - .c C语言文件：内联汇编

AT&T语法和Intel语法主要区别

- **Linux下汇编语言：AT&T语法(即GNU as 汇编语法)**
  - **AT&T语法起源于AT&T贝尔实验室，是在当时用于实现Unix系统的处理器操作码语法之上而形成的**
- **AT&T语法和Intel语法主要区别**
  - **AT&T使用$表示立即数，Intel不用，因此表示十进制2时，AT&T为$2，而Intel就是2**
  - **AT&T在寄存器前加%，比如eax寄存器表示为%eax**
  - **AT&T 处理操作数的顺序和Intel相反，比如，movl %eax, %ebx是将eax中的值传递给ebx，而Intel是这样的mov ebx, eax**
  - **AT&T在助记符的后面加上一个单独字符表示操作中数据的长度，比如movl $foo, %eax等同于Intel的mov eax, word ptr foo**
  - **长跳转和调用的格式不同，AT&T为ljmp $section, $offset，而Intel为jmp section:offset**

A. AT&T语法

1寄存器引用

- 引用寄存器要在寄存器号前加百分号%,如“movl %eax, %ebx”。

- 80386有如下寄存器：
  - 8个32-bit寄存器 %eax，%ebx，%ecx，%edx，%edi，%esi，%ebp，%esp；
  - 8个16-bit寄存器，它们事实上是上面8个32-bit寄存器的低16位：%ax，%bx，%cx，%dx，%di，%si，%bp，%sp；
  - 8个8-bit寄存器：%ah，%al，%bh，%bl，%ch，%cl，%dh，%dl。它们事实上是寄存器%ax，%bx，%cx，%dx的高8位和低8位；
  - 6个段寄存器：%cs(code)，%ds(data)，%ss(stack), %es，%fs，%gs；
  - 3个控制寄存器：%cr0，%cr2，%cr3；
  - 6个debug寄存器：%db0，%db1，%db2，%db3，%db6，%db7；
  - 2个测试寄存器：%tr6，%tr7；
  - 8个浮点寄存器栈：%st(0)，%st(1)，%st(2)，%st(3)，%st(4)，%st(5)，%st(6)，%st(7)。
- R…(64位)

2. 操作数顺序

操作数排列是从源（左）到目的（右）

如“**movl %eax（源）, %ebx（目的）**”

3. 立即数

使用立即数，要在数前面加符号**$**

如“**movl $0x04, %ebx**”

4. 符号常数

符号常数直接引用

value: .long 0x12a3f2de

movl value , %ebx

将常数0x12a3f2de装入寄存器ebx

引用符号地址在符号前加符号**$**

movl $value, % ebx

将符号value的地址装入寄存器ebx

5 操作数长度

操作数的长度用加在指令后的符号表示

- **b(byte, 8-bit), w(word, 16-bits), l(long, 32-bits)**

```
movb %al, %bl
movw %ax, %bx
movl %eax, %ebx
```

64-bit：movq %rax, %rbx

6 转移跳转指令

- 绝对转移和调用指令（jmp/call）
  - 操作数前要加上‘*’作为前缀
- 远程转移指令和调用指令的操作码
  - 在AT&T汇编格式中为“ljmp”和“lcall”
    - ljmp $section，$offset
    - lcall $section，$offset
  - 在Intel汇编格式中为“jmp far”和“call far”
    - jmp far section:offset
    - call far section:offset
  - 远程返回指令
    - lret

7 内存引用

- Intel语法的间接内存引用的格式为：
  section:[base+index*scale+disp]
- 而在AT&T语法中对应的形式为：
  section:disp (base,index,scale)
  - 其中，base和index是任意的32-bit base和index寄存器。
  - scale可以取值1，2，4，8。如果不指定scale值，则默认值为1。
  - section可以指定任意的段寄存器作为段前缀。
  - 计算方法：
    - base+index*scale+disp
  - 例如：
    call *sys_call_table(,%eax,4)
    这是entry_32.S中的一句，对应% eax*4 + sys_call_table, 在sys_call_table中找到相应系统调用的地址

B. GCC内联汇编

基本内联汇编的格式是：

```
__asm__ __volatile__("Instruction List");
```

1、__asm__

__asm__是GCC关键字asm的宏定义：

```
#define __asm__ asm
```

__asm__或asm用来声明一个内联汇编表达式，所以任何一个内联汇编表达式都必须以它开头

2、__volatile__

__volatile__ 是 GCC 关键字 volatile 的宏定义：

#define __volatile__ volatile

__volatile__ 或volatile是可选的

- 向GCC声明不要改写Instruction List
- 否则当使用了优化选项(-O)进行编译时，GCC将会根据自己的判断决定是否将这个内联汇编表达式中的指令优化

3、Instruction List

Instruction List是汇编指令序列
可以是空的，比如：

```
__asm__ __volatile__("");
__asm__ ("");
```

都是完全合法的内联汇编表达式

- 任意两个指令间要么被分号(；)分开，要么被放在两行（放在两行的方法既可以通过\n的方法来实现，也可以真正的放在两行）；
- 可以使用1对或多对引号，每1对引号里可以放任意条指令，所有的指令都要被放到引号中

- 在基本内联汇编中，“Instruction List”的书写的格式和直接在汇编文件中写非内联汇编相似：

```
__asm__(".align 2\n\t"
"movl %eax, %ebx\n\t"
"test %ebx, %ecx\n\t"
"jne error\n\t"
"sti\n\t"
"error: popl %edi\n\t"
"subl %ecx, %ebx");
```

上面例子的格式是Linux内联代码常用的格式，非常整齐。也建议大家使用这种格式来写内联汇编代码

- 带有C/C++表达式的内联汇编格式为：

  __asm__ __volatile__("Instruction List" : Output : Input : Modify);

- 从中我们可以看出它和基本内联汇编的不同之处在于：
  - 它多了3个部分(Output， Input，Clobber/Modify)。
  - 在括号中的4个部分通过冒号(:)分开。

  - **__asm__ (" " : : : "memory");**
  - **__asm__ ("mov %%eax, %%ebx" : "=b"(rv) : "a"(foo) : "eax", "ebx");**
  - **__asm__ __volatile__("lidt %0": "=m" (idt_descr));**
  - **__asm__("subl %2,%0\n\t"**
    "sbbl %3,%1"
    : "=a" (endlow), "=d" (endhigh)
    : "g" (startlow), "g" (starthigh), "0" (endlow), "1" (endhigh));

- **Output**

Output用来指定当前内联汇编语句的输出。我们看一看这个例子：

__asm__(“movl %%cr0, %0”: “=a” (cr0));

这个输出操作由两部分组成：

--括号括住的部分(cr0) → 是一个C/C++表达式, 其操作就等于C/C++的相等赋值

cr0 = output_value

--引号引住的部分“=a” →引号中的内容，被称作“操作约束”（Operation Constraint），该例中操作约束“=a”包含两个约束：等号(=)和字母a。

1.等号(=)说明括号中左值表达式cr0是一个 Write-Only的。每个输出操作数的限定字符串必须包含“=”表示他是一个输出操作数。

2.字母a是寄存器EAX / AX / AL的简写，说明cr0的值要从eax寄存器中获取，也就是说cr0 = eax这两部分都是每一个输出操作必不可少的。

最终这一点被转化成汇编指令就是

movl %cr0, %eax

movl %eax, address_of_cr0

- **Input**
  Input域的内容用来指定当前内联汇编语句的输入。我们看一看这个例子：

  __asm__(“movl %0, %%db7” : : “a” (cpu->db7));

  输入表达式也分为两部分：

  带括号的部分(cpu->db7)和带引号的部分 “a”。

  括号中的表达式cpu->db7是一个C/C++语言的表达式，

  引号号中的部分是约束部分，只能是默认的Read-Only的。

  约束中必须指定一个寄存器约束，例中的字母a表示当前输入变量cpu->db7要通过寄存器eax输入到当前内联汇编中。

- **Modify/Clobber**

__asm__ (“movl %0, %%ebx” : : “a”(foo) : “bx”);

寄存器%ebx出现在“Instruction List中”，并且被movl指令修改，但却未被任何Input/Output操作表达式指定，所以你需要在Modify域指定“bx”，以让GCC知道这一点。

如果有多个寄存器需要声明，在任意两个声明之间用逗号隔开

比如：
__asm__ ("movl %0, %%ebx; popl %%ecx" : : "a"(__foo) : "bx", "cx");

限制字符

- **r auto select a,b,c,d**
- **q auto select a,b,c,d**
- **m memory**
- **g r or m**
- **d %edx %ed %dl**
- **c %ecx %ec %cl**
- **b %ebx %eb %bl**
- **a %eax %ex %al**
- **D %edi %di**
- **S %esi %si**

一个内联汇编例子的详细解释

```
{
  int a=10, b;
  asm ("movl %1, %%eax;
  movl %%eax, %0;"
  :"=r"(b) /* output */
  :"r"(a) /* input */
  :"%eax"); /* clobbered register */
  }
```

Demo

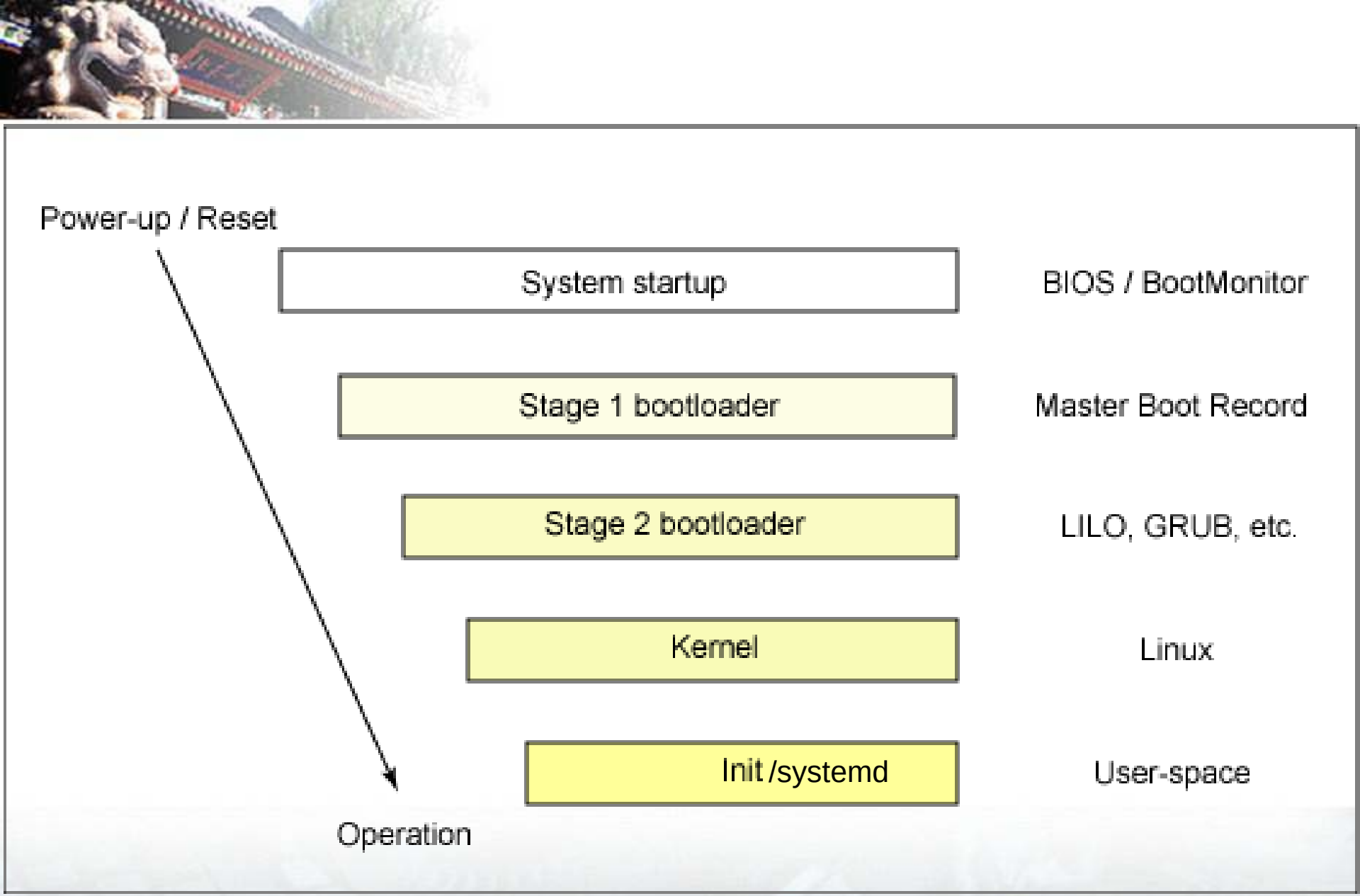
```
#include <stdio.h>
int main(void) {
   int foo=10, bar=15;
   __asm__ __volatile__ ("addl %%ebx,
   %%eax“
   : "=a"(foo) // ouput
   : “a”(foo), “b”(bar)); //input
   printf("foo+bar=%d\n", foo);
   return 0;
}
```

PEKING UNIVERSITY

Basic Knowledge of Linux Kernel Programming

- Kernel Compiling Explanation
- Kernel Debugging Techniques
- Module Programming
- AT&T Assembly Language
- **Linux Booting Process**
- Tools for Linux Kernel Source Code Reading
- Kernel & Git

PEKING UNIVERSITY



M. Tim Jones，Inside Linux boot process

Linux启动过程简要分析

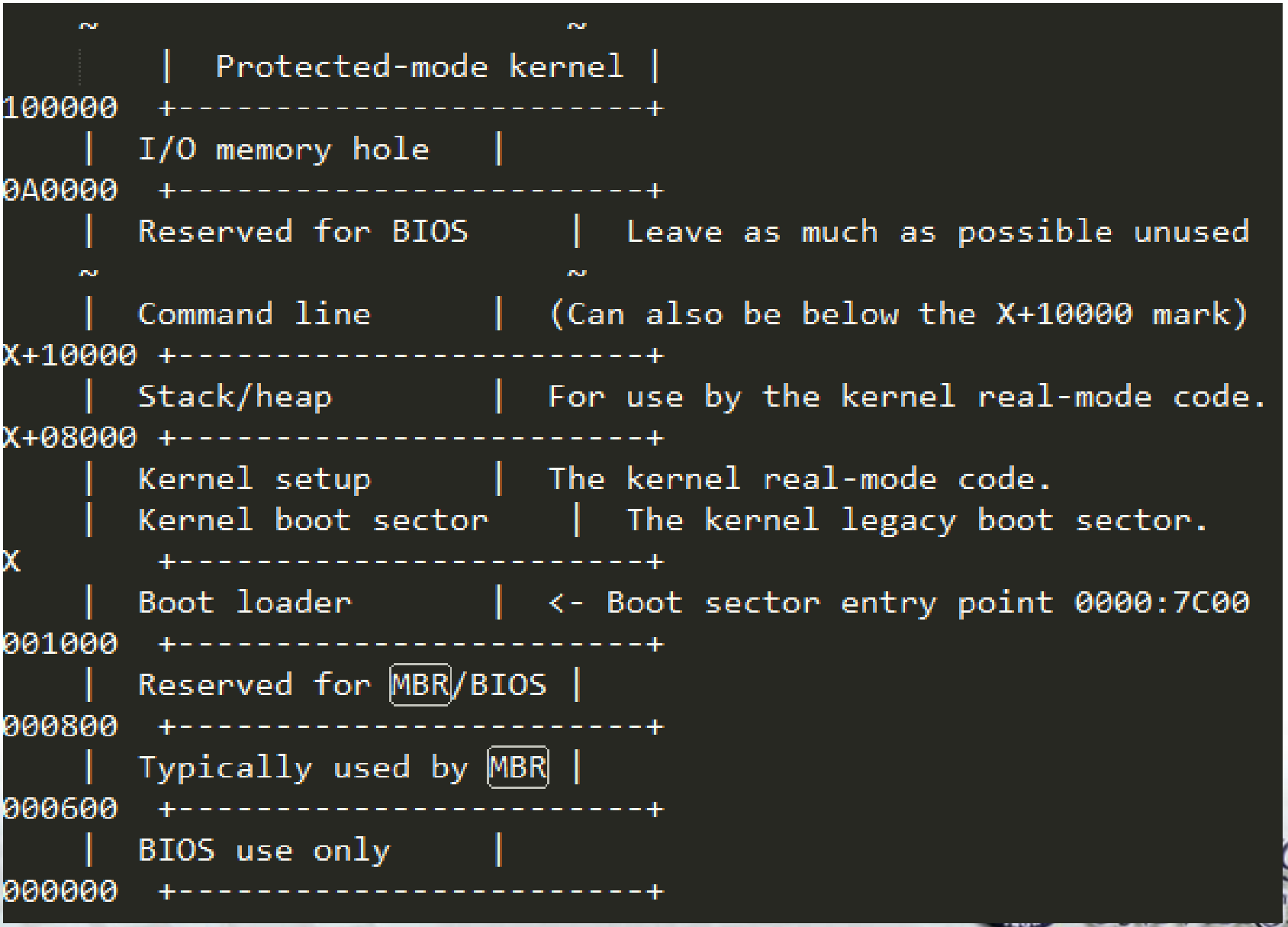
- 系统复位后，**CPU根据CS和IP的值执行FFFF0H处的指令。**
  - **CPU复位后，CS:IP的值为FFFF0H;**
  - **FFFF0H处的指令一般总是一个JMP指令，跳转地址通常是 BIOS 的入口地址。**
- **BIOS 进行开机自检，如检查内存，键盘等。**
- 产生BIOS中断INT13H，指向某个接入的可引导设备的第一个扇区（**MBR**）**并将这个扇区的内容装入内存0x7c00, 跳转到此地址，把控制权交给这段代码。**

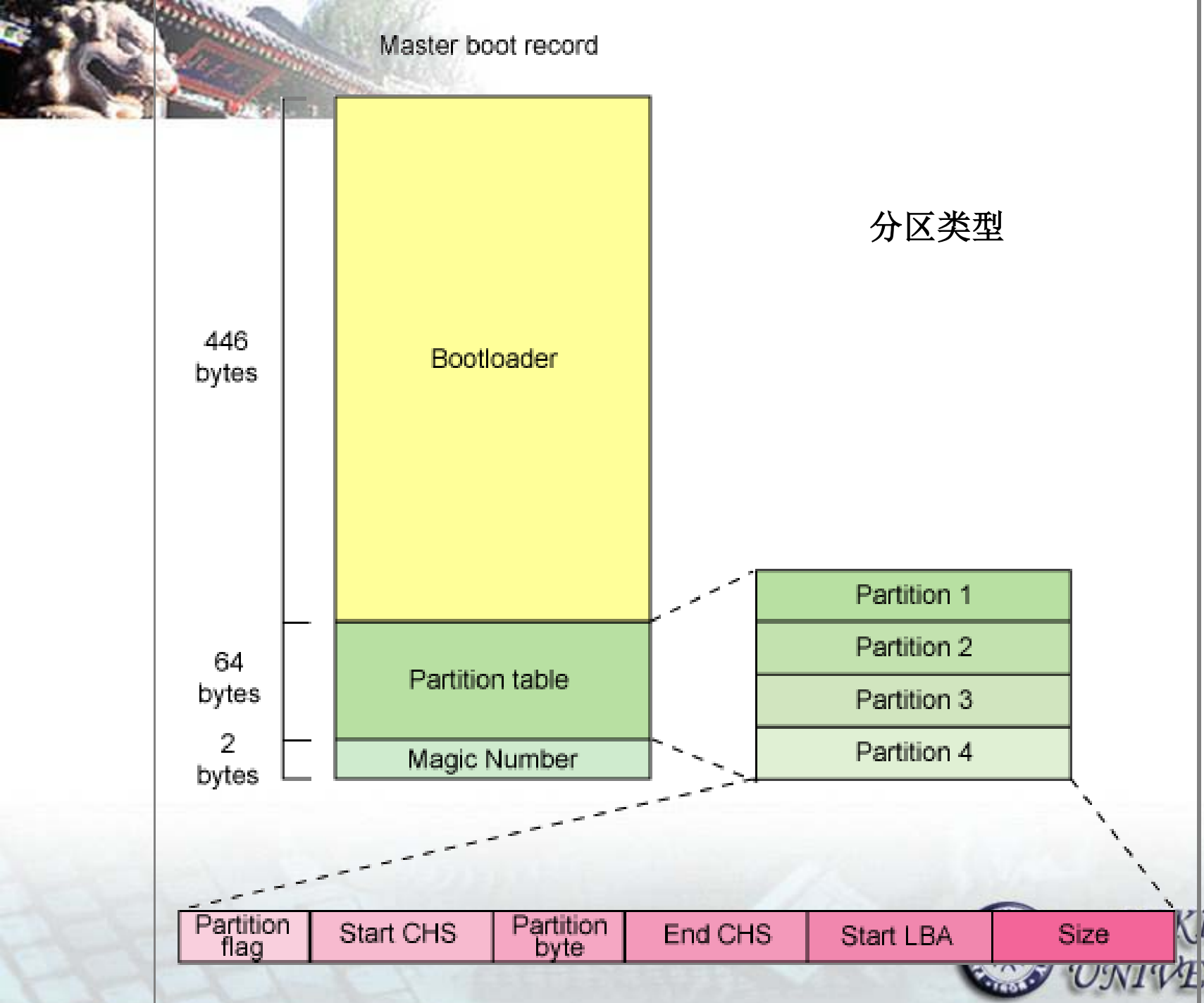
MBR

- 引导设备的第一扇区称为主引导记录（MBR， MASTER BOOT RECORD）

- MBR 的长度为512字节。
  - 第一部分为引导（PRE-BOOT）区，占了446个字节
  - 第二部分为分区表（PARTITION PABLE），共有66个字节，记录硬盘的分区信息。

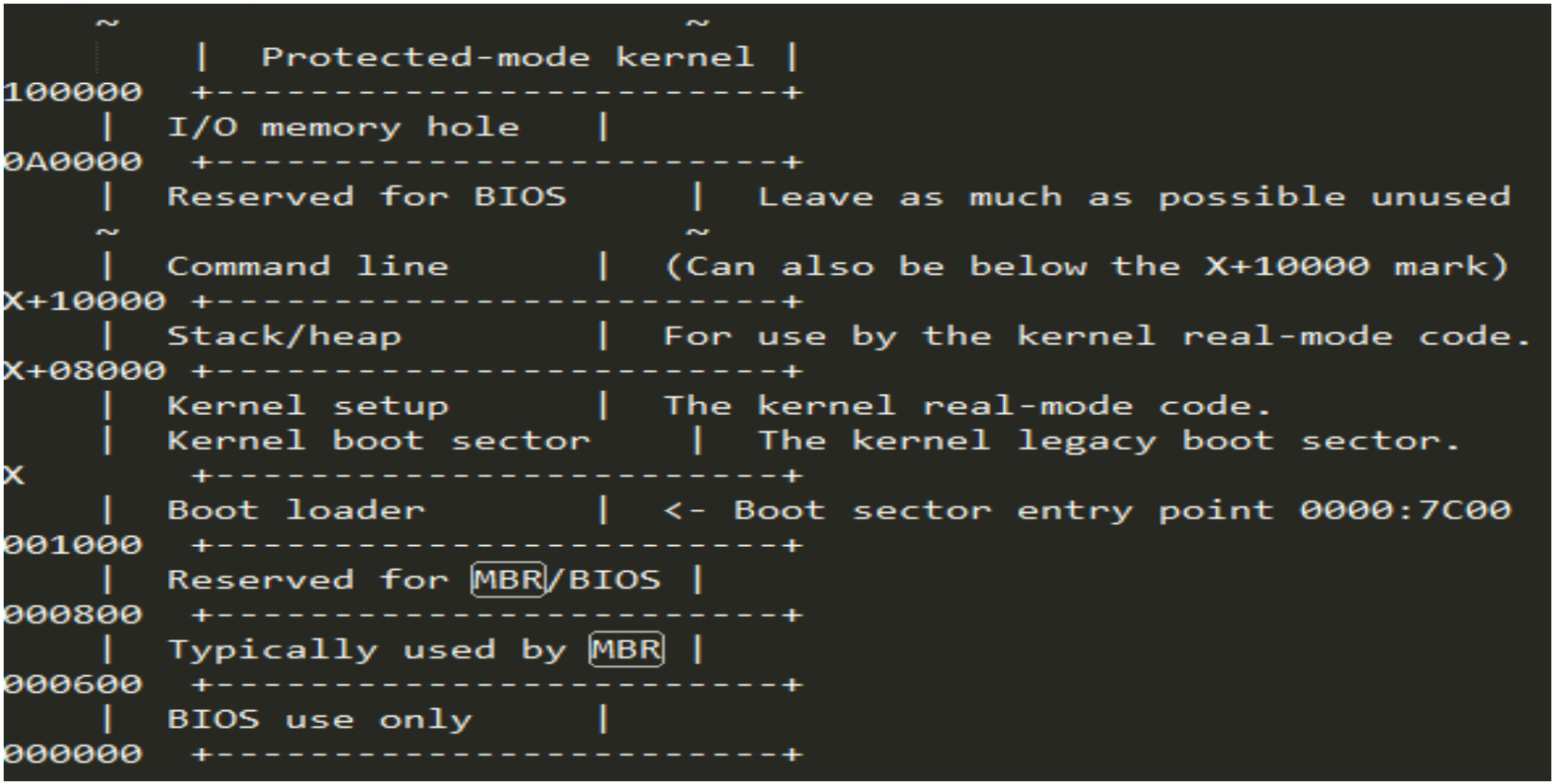
bzImage kernel内存分配





分区类型

bzImage kernel内存分配

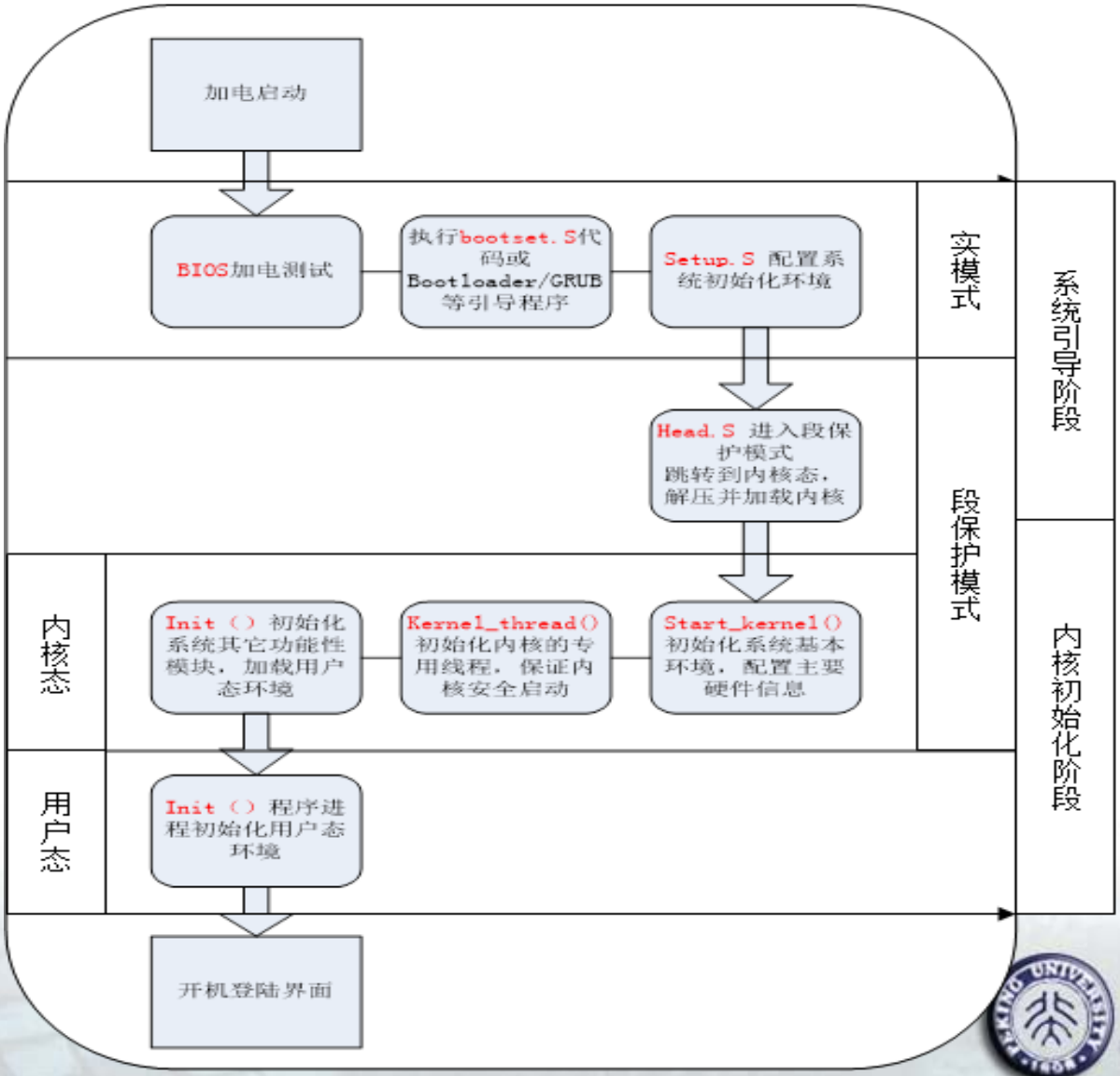


- **MBR向下移动到内存0x600，然后将活动分区的引导扇区加载到0x7c00**

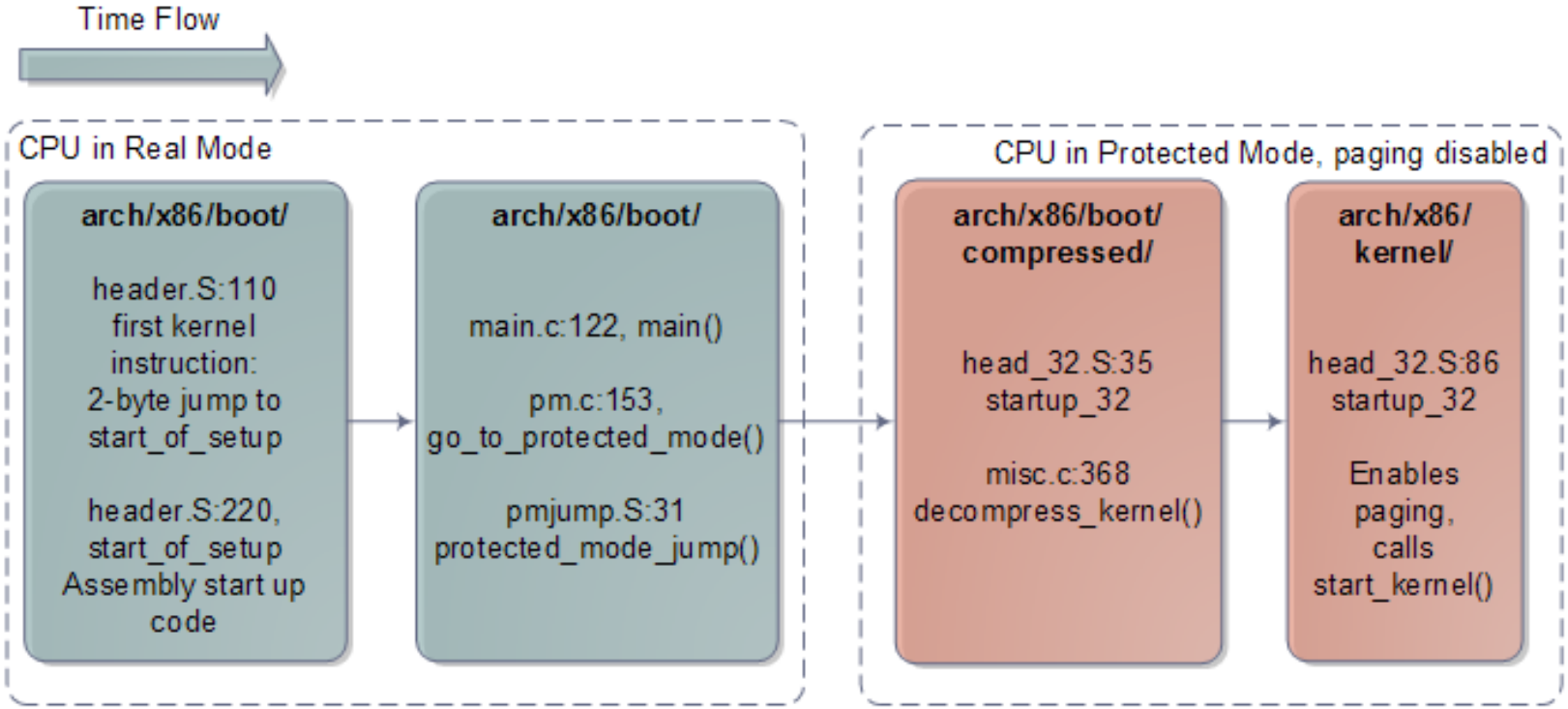
Grub主要引导流程

- 装载基本引导装载程序：**stage1（512字节）**
  - 其主要功能就是装载第二引导程序**(stage2)**
  - 主要因为在主引导扇区中没有足够的空间。
- 装载第二引导程序**(stage2)**
  - 第二引导程序可以装载一个特定的操作系统
  - 在**GRUB**中，这步是显示一个菜单或是输入命令
  - 由于**stage2**很大,所以它一般位于文件系统之中**(**通常是**boot**所在的根分区**)**
- 将机器的控制权转交给操作系统**(linux).** 操作系统接到控制权之后,开始**start_kernel**。

Linux Booting Process



Linux i386 内核引导流程



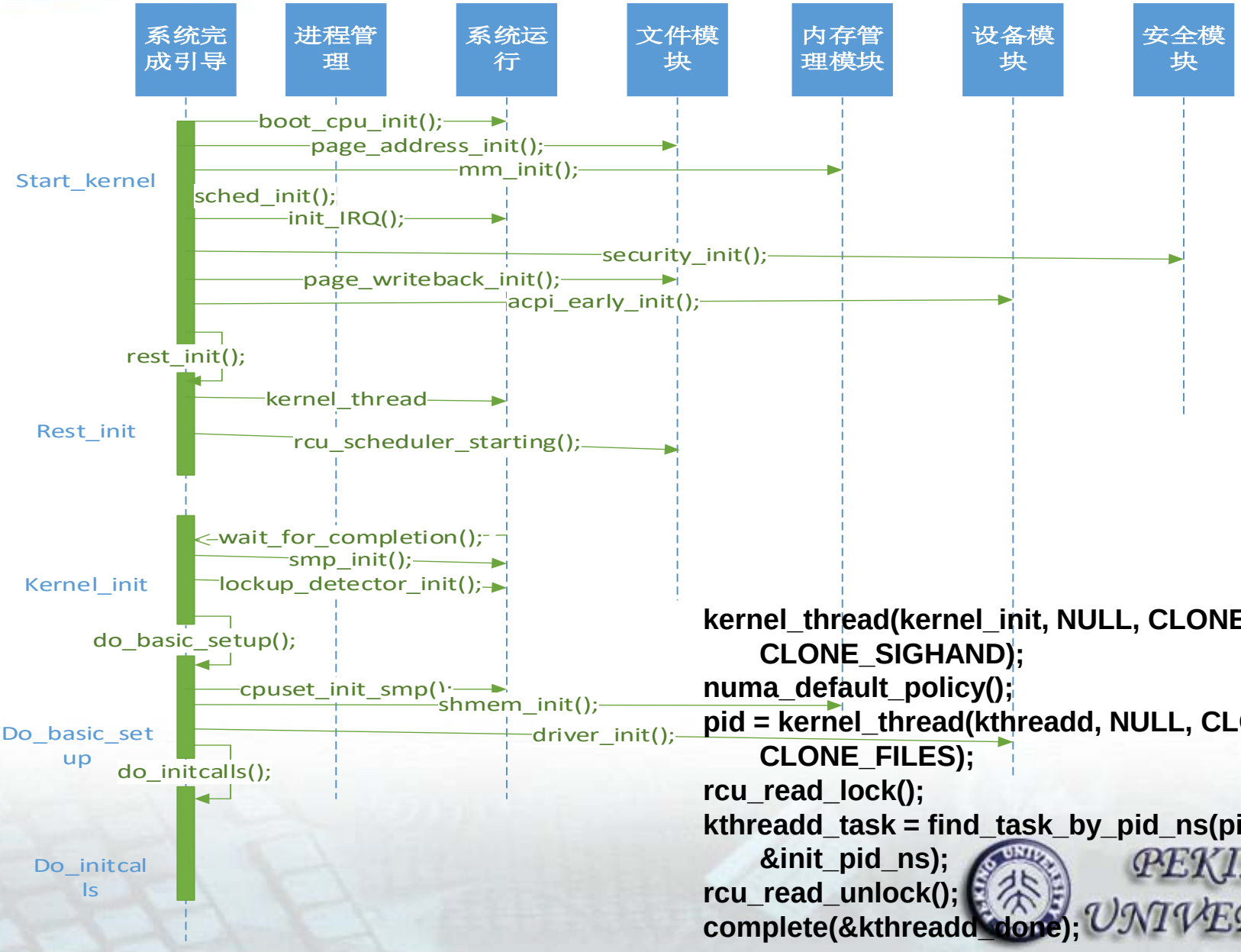
i386下启动过程中相关函数功能

- **start_of_setup (arch/x86/header.s)**
  - 重置所有的磁盘控制器。
  - 检查内核的签名(md5码)保证要载入的内核是正确的。
  - 清零 BSS段（Block Started by Symbol, 通常用来存放程序中未初始化的全局变量和静态变量)
  - 跳转到main() 函数执行剩下的初始化,启动内核.
- **main() （arch/x86/boot/main.c）**
  - 初始化early-boot console。
  - 初始化heap。
  - 检查内核是否支持本机的cpu。
  - 通过bios设置cpu的运行模式。
  - 检测本机内存型号大小。
  - 初始化键盘。
  - 设置显示模式。
  - 进入保护模式。

i386下启动过程中相关函数功能

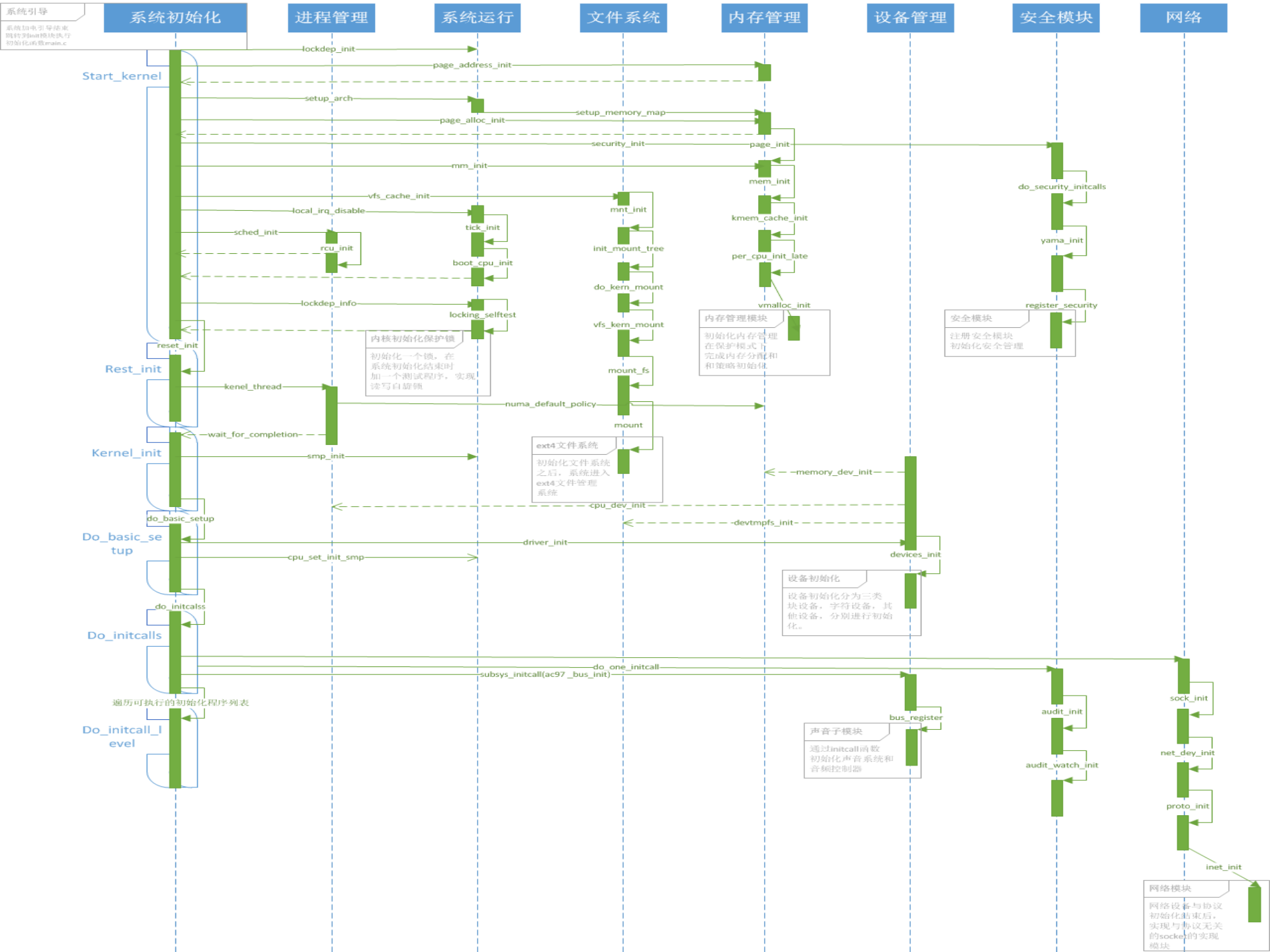
- **go_to_protected_mode**()((/arch/x86/boot/main.c)
  - 设置中断。
  - 设置idt。
  - 设置gdt
  - 跳转到head_32中
- head_32(/arch/x86/boot/head.s)
  - 跳转到start_kernel函数进行剩下的初始化。
- **start_kernel**()(arch/alpha/boot/main.c)
  - 载入内核镜像(vmlinux)
  - 驱动内核线程kernel_init 来进行子系统的初始化。

Linux 内核 x86 引导的主要函数流程




```
kernel_thread(kernel_init, NULL, CLONE_FS |
        CLONE_SIGHAND);
numa_default_policy();
pid = kernel_thread(kthreadd, NULL, CLONE_FS |
        CLONE_FILES);
rcu_read_lock();
kthreadd_task = find_task_by_pid_ns(pid,
        &init_pid_ns);
rcu_read_unlock();
complete(&kthreadd_done);
```

PEKING UNIVERSITY


系统引导
系统加电引导结束
跳转到init模块执行
初始化函数main.c
系统初始化
进程管理
系统运行
文件系统
内存管理
设备管理
安全模块
网络
lockdep_init
page_address_init
Start_kernel
setup_arch
setup_memory_map
page_alloc_init
security_init
page_init
mm_init
mem_init
do_security_initcalls
vfs_cache_init
mnt_init
local_irq_disable
tick_init
kmem_cache_init
yama_init
sched_init
rcu_init
init_mount_tree
per_cpu_init_late
boot_cpu_init
do_kern_mount
register_security
lockdep_info
vmalloc_init
locking_selftest
vfs_kern_mount
内存管理模块
初始化内存管理
在保护模式下
完成内存分配和
和策略初始化
安全模块
注册安全模块
初始化安全管理
内核初始化保护锁
初始化一个锁，在
系统初始化结束时
加一个测试程序，实现
读写自旋锁
reset_init
Rest_init
mount_fs
kenel_thread
numa_default_policy
mount
wait_for_completion
Kernel_init
smp_init
ext4文件系统
初始化文件系统
之后，系统进入
ext4文件管理
系统
memory_dev_init
cpu_dev_init
devtmpfs_init
do_basic_setup
Do_basic_setup
driver_init
devices_init
cpu_set_init_smp
设备初始化
设备初始化分为三类
块设备，字符设备，其
他设备，分别进行初始
化。
do_initcalls
Do_initcalls
do_one_initcall
subsys_initcall(ac97 _bus_init)
遍历可执行的初始化程序列表
sock_init
bus_register
audit_init
net_dev_init
Do_initcall_level
声音子模块
通过initcall函数
初始化声音系统和
音频控制器
audit_watch_init
proto_init
inet_init
网络模块
网络设备与协议
初始化结束后，
实现与协议无关
的socket的实现
模块

start_kernel

- lockdep_init
- lock_kernel
- tick_init
- boot_cpu_init
- page_address_init
- printk
- setup_arch
- setup_per_cpu_areas
- build_all_zonelists
- page_alloc_init
- vfs_caches_init_early
- trap_init
- mm_init
- sched_init
- init_IRQ
- init_timers
- softirq_init
- time_init
- profile_init
- console_init
- vfs_caches_init

kernel_init

- lock_kernel
- smp_init
- do_basic_setup
- init_post

PEKING UNIVERSITY

- 初始化同步与互斥环境
- 启动大内核锁
- 注册时钟事件监听器
- 激活第一个CPU
- 初始化地址散列表
- 打印版本信息
- 执行setup_arch()函数设置与体系结构相关的环境
- 设置每CPU环境
- 初始化物理内存管理区列表
- 利用early_res分配内存
- 虚拟文件系统早期初始化（dentry 和 inode的hashtable初始化）
- 初始化异常服务

- 启用伙伴算法
- 初始化slab分配器
- 初始化非连续内存区
- 初始化调度程序
- 设置APIC中断服务
- 初始化本地软时钟
- 软中断初始化
- 初始化定时器中断
- 安装根文件系统

- 激活所有cpu
- 外设驱动初始化
- 启动shell环境

System V init

- User space init (init process)
  - Read /etc/inittab
  - Execute init scripts (/etc/rc*.d, …) per current run level
    - Init script will mount fs/start daemons/start x …
  - Start gettty, open console
  - Type user name/password, login check them, and start shell
- Run level
  - 1 – single user mode
  - 3 – multiple user mode
  - 5 – X mode
  - 6 - reboot

```
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if
    you do not have networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
```

/etc/inittab

id:5:initdefault:

PEKING UNIVERSITY

GTK+ 和 Qt

- 跨平台开源UI工具包
- GNOME、LXDE等基于GTK+开发
- KDE基于Qt开发

BIOS→UEFI：EFI

EFI (Extensible Firmware Interface)，和BIOS一样，主要在启动过程中完成硬件初始化。

- **模块化，C语言风格的参数堆栈传递方式，动态链接的形式构建的系统**
- **与BIOS中断的方式不同，EFI通过加载EFI驱动的方式识别系统硬件并完成初始化**
- **EFI驱动由专用于EFI的虚拟机器指令EFI字节码编写成，需要在EFI驱动运行环境DXE下解释运行，这样EFI既可以实现通配，又提供了良好的兼容。**
- **摒弃16位实模式，因此可以在任何内存地址存放任何信息**
- **功能上完全等同于一个轻量化的OS**
  - **只是一个硬件和操作系统间的接口；**
  - **不提供中断访问机制，使用轮询的方式检查并解释硬件，较OS下的驱动执行效率较低**
  - **只有简单的存储器管理机制，在段保护模式下只将存储器分段，所有程序都可以存取任何一段位置，不提供真实的保护服务。**

BIOS→UEFI： GPT

GPT：EFI引入的GUID磁盘分区系统

- 传统MBR磁盘只能存在4个主分区，逻辑分区也是有数量的，太多的逻辑分区会严重影响系统启动;GPT支持任意多的分区
- MBR硬盘分区最大仅支持2T容量，GPT每个分区大小原则上是无限制的
- GPT的分区类型由GUID表唯一指定，其中的EFI系统分区可以被EFI存取，用来存取部分驱动和应用程序，这原则上会使EFI系统分区变得不安全，但是一般这里放置的都是些“边缘”数据

BIOS→UEFI：UEFI

UEFI：英特尔发布的**EFI 1.1**，之后的**2.0**由于众多公司加入，**EFI**就属于了**Unified EFI Form**国际组织，改称为**UEFI**。与**EFI**相比，**UEFI**主要有以下改进：

- **UEFI**具有完整的图形驱动功能：**EFI**原则上加入了图形驱动，但为了保证和**BIOS**的良好过渡，多数还是一种类**DOS**界面，只支持**PS/2**键盘操作（极少数支持鼠标操作），不支持**USB**键盘和鼠标。**UEFI**拥有完整的图形驱动，支持**PS/2**和**USB**键盘以及鼠标
- **UEFI**具有安全启动（**Secure Boot**），也叫做固件验证。开启安全启动后，主板会根据**TPM**芯片（或者**CPU**内置的**TPM**）记录的硬件签名对各硬件判断，只有符合认证的硬件驱动才会被加载。

BIOS→UEFI：UEFI组成部分

无论EFI还是UEFI，都必须要有预加载环境、驱动执行环境、驱动程序等必要部分组成:

- 预加载：一般地，预加载环境和驱动执行环境是存储在UEFI（UEFI BIOS）芯片中的。打开电源后，UEFI预加载环境首先开始执行，负责CPU和内存（是全部容量）的初始化工作。
- 驱动执行环境（DXE）：CPU和内存初始化成功后，驱动执行环境（DXE）载入
- UEFI枚举搜索各个硬件的UEFI驱动并相继加载，完成硬件初始化工作，相比BIOS的读中断加载速度会快的多
- 硬件驱动全部加载完毕后，启动操作系统

System V init → upstart

- **2006： Scott James Remnant (Ubuntu6.10)**
- 名词：
  - **Event**：状态改变信息
  - **Job**：事件+命令 （事件被触发，init执行对应的命令）
- 事件驱动
  - 只包含按需启动的脚本
  - 启动或终止服务
    - 运行级别改变
    - 事件发生（文件系统加载、打印机安装.....）
    - 任务启动、关闭、改变时
- 与system V init兼容

System V init → systemd

- **2010: Lennart Poettering**
- 系统服务间的依赖关系——实现初始化时服务的并行启动
- 与**sysV**兼容
  - 解析**sysV init**脚本，转化依赖关系，提供并发性
- 基于**socket**和**Dbus**
- **Linux only**
  - 不兼容**POSIX**，不接受相关补丁

Basic Knowledge of Linux Kernel Programming

- Kernel Compiling Explanation
- Kernel Debugging Techniques
- Module Programming
- AT&T Assembly Language
- Linux Booting Process
- **Tools for Linux Kernel Source Code Reading**
- Kernel & Git

PEKING UNIVERSITY

阅读源代码的工具

- vim或emacs编辑器，配合grep、egrep等文本搜索工具

用vim查看main.c

amos@ubuntu: ~/linux-5.3/init

File Edit View Search Terminal Help

```
extern void time_init(void);
/* Default late time init is NULL. archs can override this later. */
void (*__initdata late_time_init)(void);

/* Untouched command line saved by arch-specific code. */
char __initdata boot_command_line[COMMAND_LINE_SIZE];
/* Untouched saved command line (eg. for /proc) */
char *saved_command_line;
/* Command line for parameter parsing */
static char *static_command_line;
/* Command line for per-initcall parameter parsing */
static char *initcall_command_line;

static char *execute_command;
static char *ramdisk_execute_command;

/*
 * Used to generate warnings if static_key manipulation functions are used
 * before jump_label_init is called.
 */
bool static_key_initialized __read_mostly;
EXPORT_SYMBOL_GPL(static_key_initialized);
```

阅读源代码的工具

find pathname -options [-print -exec -ok…]

例 #find -type f -print | xargs grep -i execute_command

```
amos@ubuntu:~/linux-5.3/init$ find -type f -print | xargs grep -i execute_command
Binary file ./main.o matches
./main.c:static char *execute_command;
./main.c:static char *ramdisk_execute_command;
./main.c:       execute_command = str;
./main.c:       ramdisk_execute_command = str;
./main.c:       if (ramdisk_execute_command) {
./main.c:               ret = run_init_process(ramdisk_execute_command);
./main.c:                       ramdisk_execute_command, ret);
./main.c:       if (execute_command) {
./main.c:               ret = run_init_process(execute_command);
./main.c:                       execute_command, ret);
./main.c:       if (!ramdisk_execute_command)
./main.c:               ramdisk_execute_command = "/init";
./main.c:                       ramdisk_execute_command, 0) != 0) {
./main.c:               ramdisk_execute_command = NULL;
Binary file ./.main.c.swp matches
amos@ubuntu:~/linux-5.3/init$
```

例：在main.c中查找execute_command

amos@ubuntu: ~/linux-5.3/init

File Edit View Search Terminal Help

```
void (*__initdata late_time_init)(void);

/* Untouched command line saved by arch-specific code. */
char __initdata boot_command_line[COMMAND_LINE_SIZE];
/* Untouched saved command line (eg. for /proc) */
char *saved_command_line;
/* Command line for parameter parsing */
static char *static_command_line;
/* Command line for per-initcall parameter parsing */
static char *initcall_command_line;
static char *execute_command;
static char *ramdisk_execute_command;

/*
 * Used to generate warnings if static_key manipulation functions are used
 * before jump_label_init is called.
 */
bool static_key_initialized __read_mostly;
EXPORT_SYMBOL_GPL(static_key_initialized);

/*
 * If set, this is an indication to the drivers that reset the underlying
 * device before going ahead with the initialization otherwise driver might
:grep execute_command main.c
```

结果列表

```
main.c:142:static char *execute_command;
main.c:143:static char *ramdisk_execute_command;
main.c:338:     execute_command = str;
main.c:355:     ramdisk_execute_command = str;
main.c:1128:    if (ramdisk_execute_command) {
main.c:1129:            ret = run_init_process(ramdisk_execute_command);
main.c:1133:                  ramdisk_execute_command, ret);
main.c:1142:    if (execute_command) {
main.c:1143:            ret = run_init_process(execute_command);
main.c:1147:                execute_command, ret);
main.c:1205:    if (!ramdisk_execute_command)
main.c:1206:            ramdisk_execute_command = "/init";
main.c:1209:                  ramdisk_execute_command, 0) != 0) {
main.c:1210:            ramdisk_execute_command = NULL;

Press ENTER or type command to continue
```

光标定位到第一个结果处

amos@ubuntu: ~/linux-5.3/init

File Edit View Search Terminal Help

```
/* Untouched command line saved by arch-specific code. */
char __initdata boot_command_line[COMMAND_LINE_SIZE];
/* Untouched saved command line (eg. for /proc) */
char *saved_command_line;
/* Command line for parameter parsing */
static char *static_command_line;
/* Command line for per-initcall parameter parsing */
static char *initcall_command_line;
static char *execute_command;
static char *ramdisk_execute_command;

/*
 * Used to generate warnings if static_key manipulation functions are used
 * before jump_label_init is called.
 */
bool static_key_initialized __read_mostly;
EXPORT_SYMBOL_GPL(static_key_initialized);

/*
 * If set, this is an indication to the drivers that reset the underlying
 * device before going ahead with the initialization otherwise driver might
 * rely on the BIOS and skip the reset operation.
(1 of 14): static char *execute_command;                    142,1         10%
```

cn/cp命令光标移动到上/下一个

amos@ubuntu: ~/linux-5.3/init

File Edit View Search Terminal Help

```
/* Untouched command line saved by arch-specific code. */
char __initdata boot_command_line[COMMAND_LINE_SIZE];
/* Untouched saved command line (eg. for /proc) */
char *saved_command_line;
/* Command line for parameter parsing */
static char *static_command_line;
/* Command line for per-initcall parameter parsing */
static char *initcall_command_line;
static char *execute_command;
static char *ramdisk_execute_command;

/*
 * Used to generate warnings if static_key manipulation functions are used
 * before jump_label_init is called.
 */
bool static_key_initialized __read_mostly;
EXPORT_SYMBOL_GPL(static_key_initialized);

/*
 * If set, this is an indication to the drivers that reset the underlying
 * device before going ahead with the initialization otherwise driver might
 * rely on the BIOS and skip the reset operation.
(2 of 14): static char *ramdisk_execute_command;                143,0-1       10%
```

阅读源代码的工具

Linux:vim或emacs编辑器，配合cscope、ctags、etags等交叉索引工具

- Vim+Ctags：在源码目录下执行
  #make tags
  #vim –t function_variable_name
- cscope安装：
  #sudo apt-get install cscope

阅读源代码的工具

Vim+Ctags用法：

:ta x

跳转到符号x的定义处，如果有多个符号，直接跳转到第一处

:ts x

列出符号x的定义

:tj x

可以看做上面两个命令的合并，如果只找到一个符号定义，那么直接跳转到符号定义处，如果有多个，则让用户自行选择。

Ctrl+]

跳转到当前光标下符号的定义处，和ta类似。

Ctrl+t

跳转到上一个符号定义处，和上面的配合基本上就能自由跳转了。

tn, :tp是在符号的多个定义之间跳转

阅读源代码的工具

vim+cscope:

除了能查找函数定义，还能查找函数在哪里被调用过等信息， 在一定程度上弥补ctags的不足。

- 命令：

：cs find s ---- 查找C语言符号，即查找函数名、宏、枚举值等出现的 地方

：cs find g ---- 查找函数、宏、枚举等定义的位置，类似ctags所提供的 功能

：cs find d ---- 查找本函数调用的函数：cs find c ---- 查找调用本函数的 函数

：cs find t: ---- 查找指定的字符串

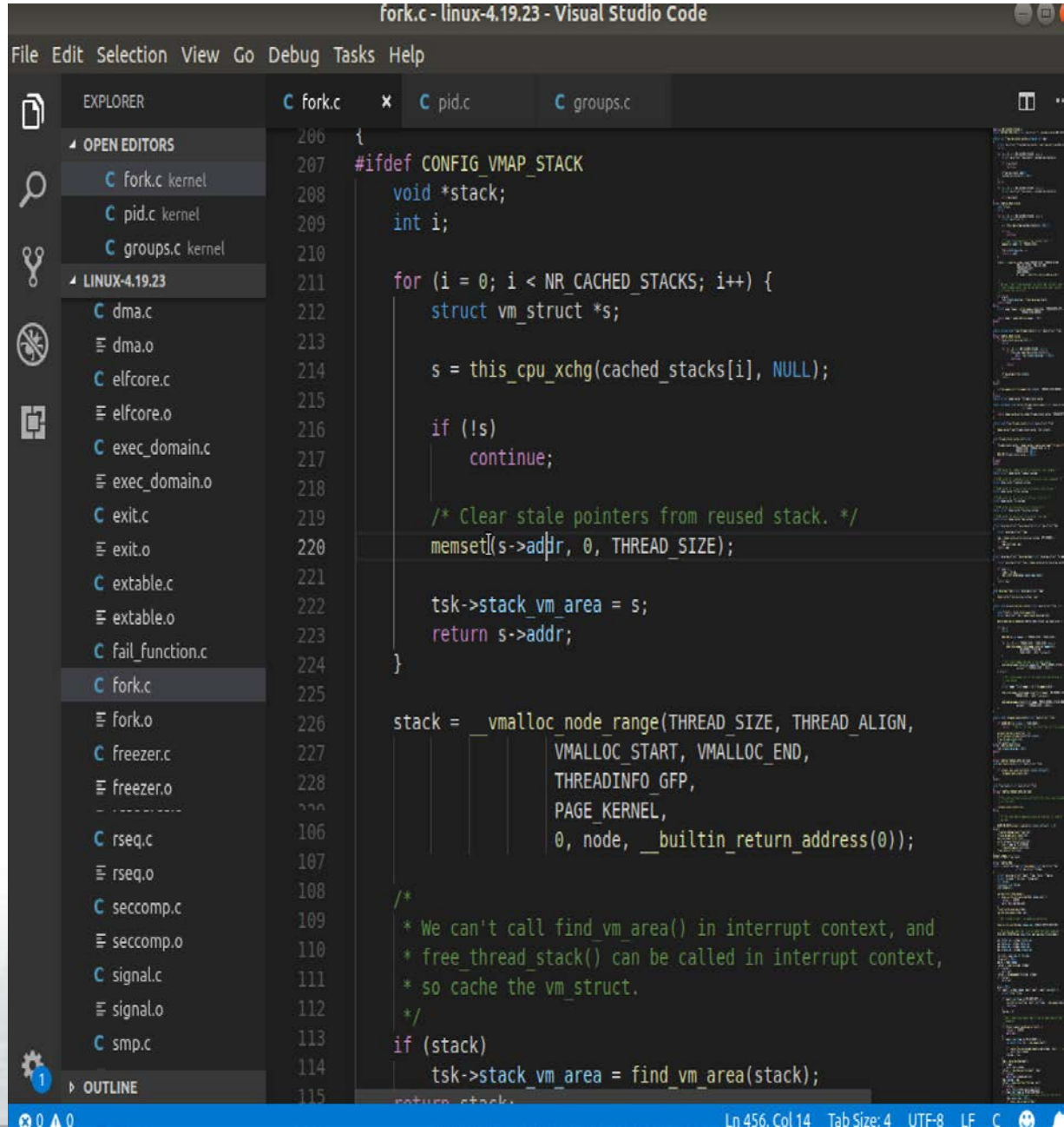
：cs find e ---- 查找egrep模式，相当于egrep功能，但查找速度快多了

：cs find f ---- 查找并打开文件，类似vim的find功能

阅读源代码的工具——IDE

➢ **VScode**

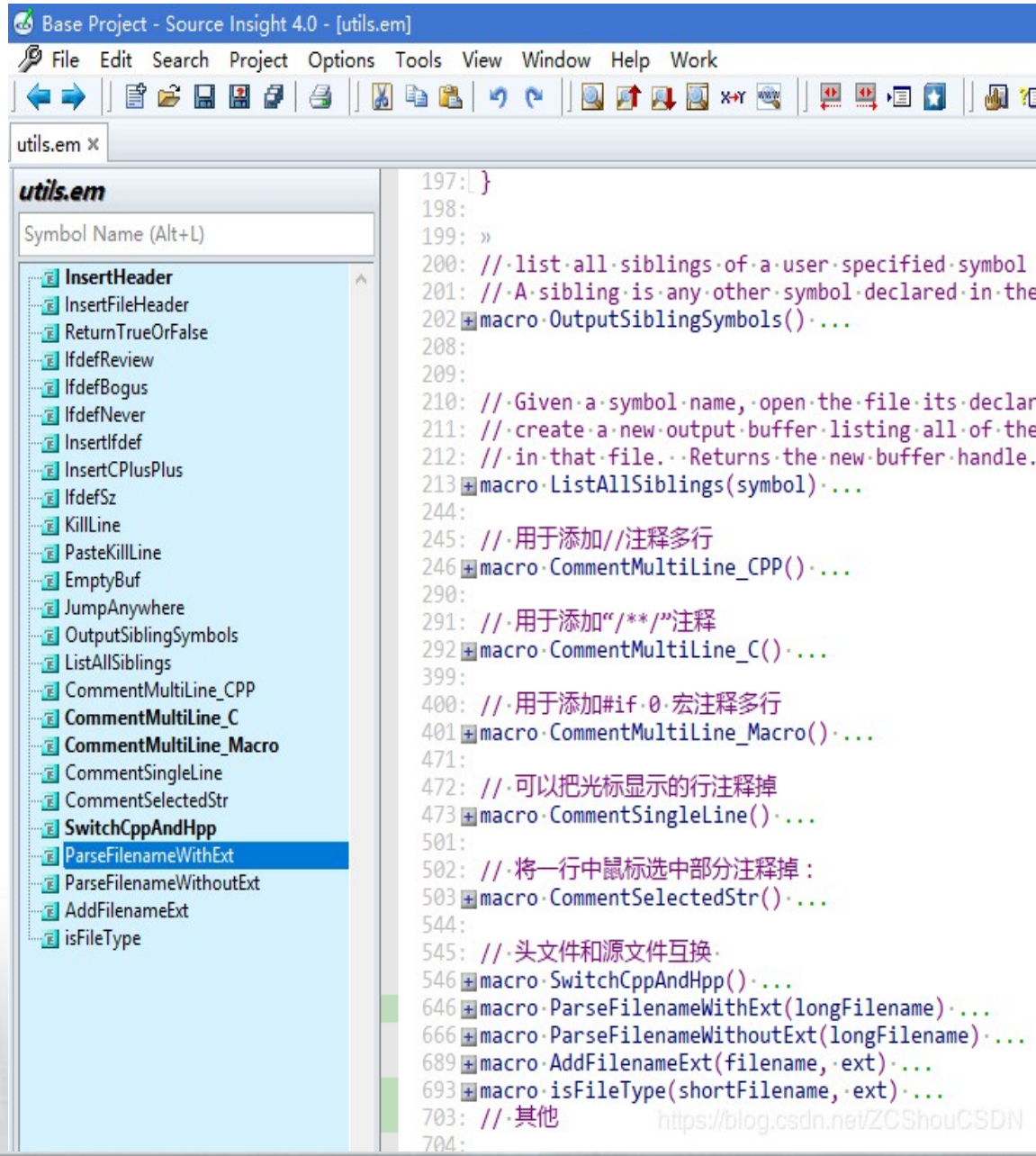
- **vscode是一个轻量且强大的代码编辑器，支持Windows，OS X和Linux。但有集成的markdown即时预览，是一个轻量且好用的用来阅读linux源代码的IDE**
- **vscode的缺陷是无法自动完成补全**



阅读源代码的工具

- **Windows:Source Insight**

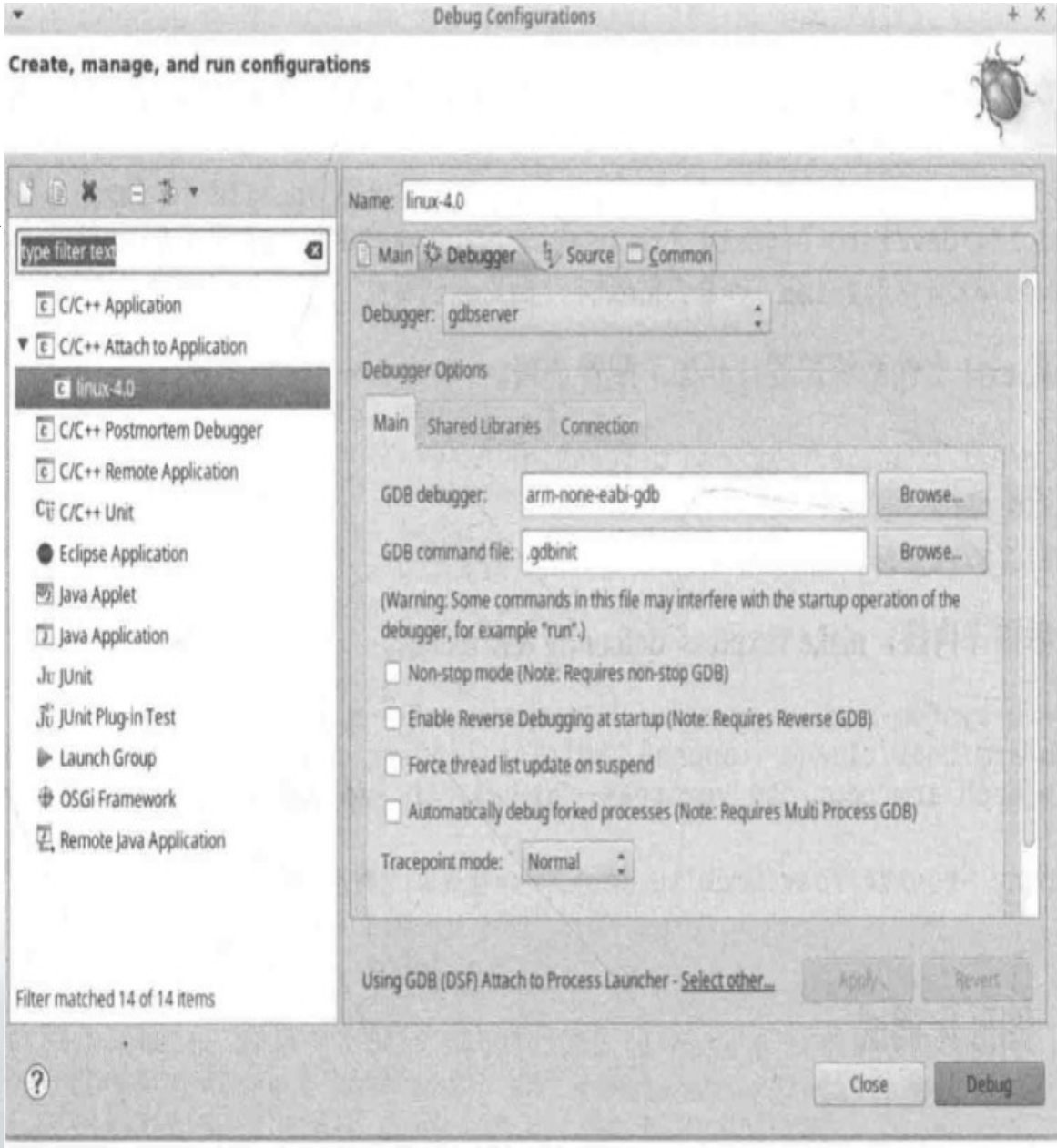
- source insight + sublime
  或者
  给sublime 集成 cscope + ctags插件
- source insight存在以下问题:
  A. 不支持unicode编码
  B. 不能跨平台



阅读源代码的工具——IDE

➢**Eclipse+CDT**

- **Eclipse** 是著名的跨平台开源集成开发环境**(IDE),**最初主要用于**JAVA**语言开发,目前可以支持**C/C++**、**Python** 等多种语言
- **Eclipse**安装**cdt**插件之后，可以完成代码阅读以及调试，缺点是软件较大



阅读源代码的工具——在线阅读

➢ 以网页的形式通过浏览器在线阅读**Linux**内核源代码。

1. **http://lxr.linux.no/**
2. **https://elixir.bootlin.com**
3. **http://lxr.freeelectrons.com/**
4. **https://lxr.missinglinkelectronics.com/+trees**
5. **http://oldlinux.org/Linux.old/kernel/Historic/**

LXR | **Welcome to lxr.linux.no -- the Linux Cross Reference** | Prefs

Welcome to lxr.linux.no

LXR (formerly "the Linux Cross Referencer") is a software toolset for indexing and presenting source code repositories. LXR was initially targeted at the Linux source code, but has proved usable for a wide range of software projects. lxr.linux.no is currently running an experimental fork of the LXR software.

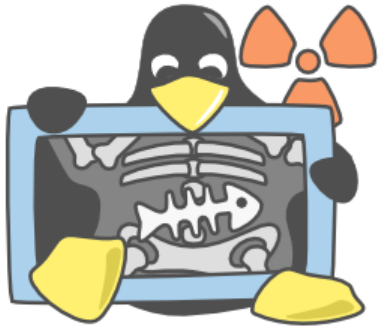
Browse the code

These are the browsable source code repositores at lxr.linux.no:

- Linux 2.6.11 and later
- Linux 2.5.0 - 2.6.11
- Linux 0.01 - 2.4.31
- SYSLINUX
- coreboot (current tree)
- coreboot v2 (formerly LinuxBIOS v2)
- coreboot v3 (formerly LinuxBIOS v3)
- Mac OS X Darwin/xnu
- Perl

LXR Source code

For the interested, the source code is available as a git repository at `git://lxr.linux.no/git/lxrng.git`. Not all of the functionality present in mainline LXR is available in this version, and the documentation is unfortunately rather sparse. Don't hesitate to contact `lxr@linux.no`



What's new

2013-08-01

The freetext search index for the Linux kernel is currently in the process of being rebuilt, as it has suffered some form of corruption. The freetext search functionality will be absent or spotty while this happens, but at least the server errors that some of you have reported lately should be gone now.

2010-05-07

A badly timed hardware malfunction brought lxr.linux.no offline for several days. Thanks to the folks at Redpill Linpro for bringing the

阅读源代码的工具

- **Windows---Source Insight / VS code**
- **Linux---Source Navigator**
- **vim或emacs编辑器，配合cscope、ctags、etags等交叉索引工具。**

  Vim+Ctags

  源码目录下：#make tags

  #vim –t fun
- **vim或emacs编辑器，配合grep、egrep等文本搜索工具。**
- **Vscode**
- **Eclipse+CDT**
- **以网页的形式通过浏览器浏览**

  可从http://lxr.linux.no/下载该工具也可以直接访问http://lxr.linux.no/在线阅读Linux内核源代码。
- **slickedit、clion、vim+ycm**

Basic Knowledge of Linux Kernel Programming

- Kernel Compiling Explanation
- Kernel Debugging Techniques
- Module Programming
- AT&T Assembly Language
- Linux Booting Process
- Tools for Linux Kernel Source Code Reading
- **Kernel & Git**

PEKING UNIVERSITY

Linus & Git & github

- **Linux社区以Linus为主基于BitKeeper开发的**
- **2005年诞生，最先进的分布式版本控制系统**

Git的目标如下：

- 速度
- 设计简单
- 强烈支持非线性开发（数千个并行分支）
- 完全分布式
- 能够高效处理**Linux**内核等大型项目（速度和数据大小）

易于使用，运行速度惊人，在大型项目中非常高效，并且具有用于非线性开发的出色分支系统。

GitHub：面向开源及私有软件项目的托管平台，只支持**git** 作为唯一的版本库格式进行托管。

PEKING UNIVERSITY

Git安装&初始化

- 安装
  - **Ubuntu系统中，git可以使用下面的命令安装**

    >> sudo apt-get install git
  - **Win/Mac系统可以下载安装包进行安装**
- 安装后可以配置一下

  >> git config --global user.name your_name

  >> git config --global user.email your_email
- 本地的**Git**仓库和远程仓库之间的传输可以通过**SSH**加密，因此我们还需要生成**SSH**密钥

  >> ssh-keygen -t rsa -C your_email

  然后将生成的公钥上传至远程仓库配置中(公钥保存在id_rsa.pub文件中)：登录GitHub→“Account settings”→“SSH Keys”→“Add SSH Key”（粘贴id_rsa.pub内容）→“Add Key”

Git Repository

- 创建版本库：Git可以管理版本库中的所有文件，即Git可以跟踪每个文件的修改、删除，以便将来可以查看历史，或者“还原”。

  >> git init

  *当前目录下多了.git（隐藏）目录，Git用来跟踪管理版本库

- 把文件添加到仓库：

  >> git add your_file

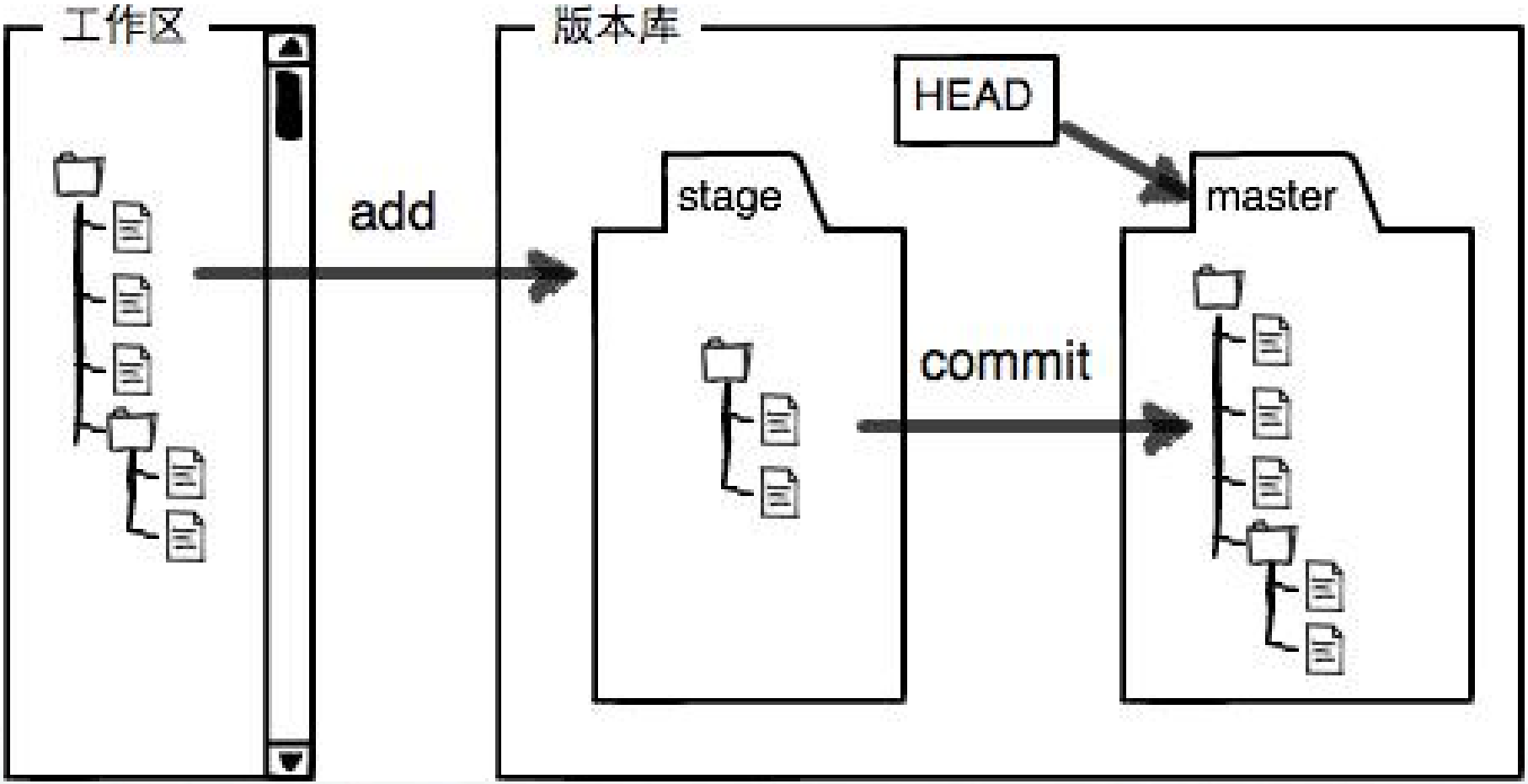
  *git rm命令从暂存库中删除文件

- 提交更改(在本地)

  >> git commit -m “message”

  *HEAD表示当前版本，也就是最新的提交；
  *commit id是一个SHA-1摘要值

本地架构



From
https://www.liaoxuefeng.com/wiki/896043488029600

Branch

每次提交，Git都将其串成一条时间线，这条时间线就是一个分支。在Git里，至少有一个分支，叫主分支master，HEAD指向它。

- 创建新的分支并切换到新分支：

  >> git branch -b dev

  加入-b参数相当于两条命令

  >> git branch dev ***>> git checkout dev***

  *目前引入了**switch**命令后相当于* ***>>git switch -c dev***

- 查看当前所有分支

  >> git branch

- 把dev分支合并到当前分支

  >> git merge dev

- 删除分支

  >> git branch -d dev

远程推送

- 多人协作等环境下，需要把远程仓库代码**fork**到自己的仓库，并从自己的仓库克隆代码到本地

  ***>> git clone** 仓库地址*

  *如果这一步不成功，可能是**SSH**公钥没有正确配置

  这时**Git**自动把本地的**master**分支和远程的**master**分支对应起来，远程仓库的默认名称是**origin**。
- 查看远程的信息**(-v**：查看详细信息）

  >> git remote -v
- 将在本地某分支提交的**commit**推送到远程**master**分支

  >> git push origin master
- 拉取远程**master**分支的更改并合并到本地**master**分支**(**如果是当前分支可以忽略冒号**)**

  >> git pull origin master:master

  这步相当于拉取**+**合并：

  >> git fetch origin master

  >> git merge FETCH_HEAD

  FETCH_HEAD：一个指针，指向目前从远程仓库拉取的末端版本

不同的协议地址

以Git项目本身进行版本库克隆为例：

- **Git协议（智能协议）：**
  - **git clone git://git.kernel.org/pub/scm/git/git.git**
- **HTTP(S)哑协议：**
  - **git clone http://www.kernel.org/pub/scm/git/git.git**
- **HTTP(S)智能协议：**
  - **使用Git 1.6.6或更高版本访问。**
  - **git clone https://github.com/git/git.git**

多人协作

通常工作模式：

1. 用**git push origin <branch-name>**推送自己的修改
2. 如果推送失败，则因为远程分支比你的本地更新，需要先用**git pull**试图合并；
3. 如果合并有冲突，则解决冲突，并在本地提交；
4. 没有冲突或者解决掉冲突后，再用

   >>git push origin <branch-name>

   推送就能成功

********如果**git pull**提示**no tracking information**，则说明本地分支和远程分支的链接关系没有创建，用命令

>>git branch --set-upstream-to <branch-name> origin/<branch-name>

Tag

kernel的主线git代码库只有一个master分支，稳定版git代码库也只有少数linux-x.x.y分支，而tag下却分出了许多版本号

Tag是什么：Git的标签是版本库的快照，是指向某个commit的指针，一个tag对应一个commit。因为commit号是一串数字，打上标签就容易理解和记忆了

- 打标签

  >> git tag "标签"

- 查看所有标签

  >> git tag

- 如果不是当前**commit**，而是给历史**commit**打标签，首先需要找到历史**commit id**

  >> git log --pretty=oneline --abbrev-commit

  >> git tag "标签" "commit id"

PEKING UNIVERSITY

Tag vs Branch

- **tag也可以checkout，可以切换到所有时间线的任意一个tag**

  >> git checkout “标签”

  在Linus Torvalds‘ tree中，可以利用tag切换版本

  >> git checkout v5.0

  branch的checkout是切换到不同的时间线。

随着commit不断提交，时间线在变化，branch代表的版本也在变，而tag对应的是固定的commit。

- **对于主线来说，不同版本创建多个branches是不必要的，tree的发展是不断的，一旦某个版本发布之后，就成为历史，它的内容是不会变化的，因此linus坚持用tag，而不是创建多个可变的branches。**
- **对于stable-kernel，里边有一些以”.y”结尾的分支，如linux-5.0.y，这是由于在linux-stable tree里有LTS的内核版本，这些版本在发布之后，还需要对其进行维护，比如把新版本的bug fix或新功能合并进已发行的版本中去，也就是大家所熟悉的更新。此时单靠tag就不够用了。**

其它相关

- **git status：仓库当前的状态**

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   readme.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

其它相关

1. 对比两个历史commit
   - git diff “tag1” “tag2”
2. 对master分支创建压缩包
   - git archive master | bzip2 > master.tar.bz2
3. git log：显示从最近到最远的提交日志
4. git reset：版本回退
   - git reset --hard commit_id
5. git checkout：暂存区内容替换掉工作区内容
   - git checkout -- file

作业2

- 模块编程实验报告

自学内容

- 操作系统的安装
- Linux基本命令使用
- Linux程序开发方法
- VI编辑器的使用
- **Makefile**的书写方法
- 其它**Linux**使用方法、技巧等

推荐参考书：

《Linux系统管理技术手册》

《鸟叔的私房菜》系列

《Linux内核技术手册》http://www.kroah.com/lkn/

PEKING UNIVERSITY