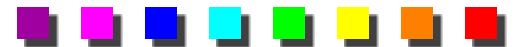


CS353 Linux Kernel

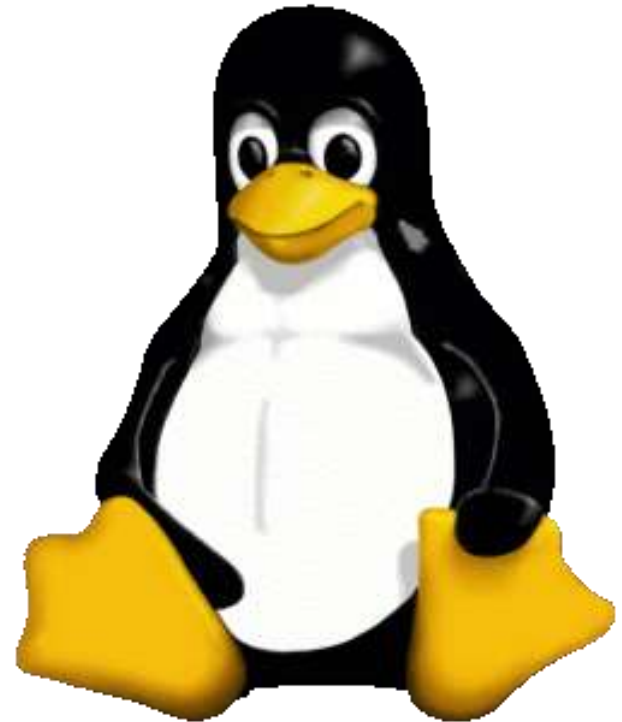
Chentao Wu 吴晨涛
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn



上海交通大学

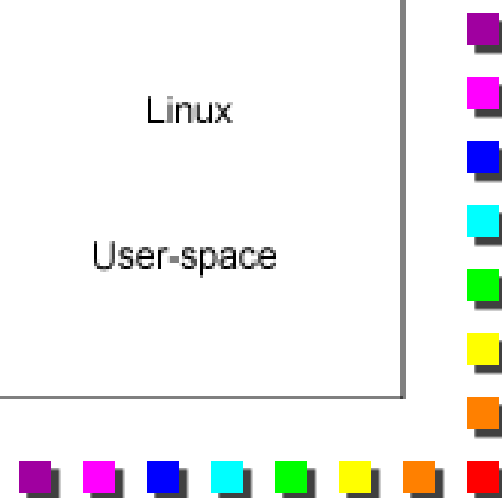
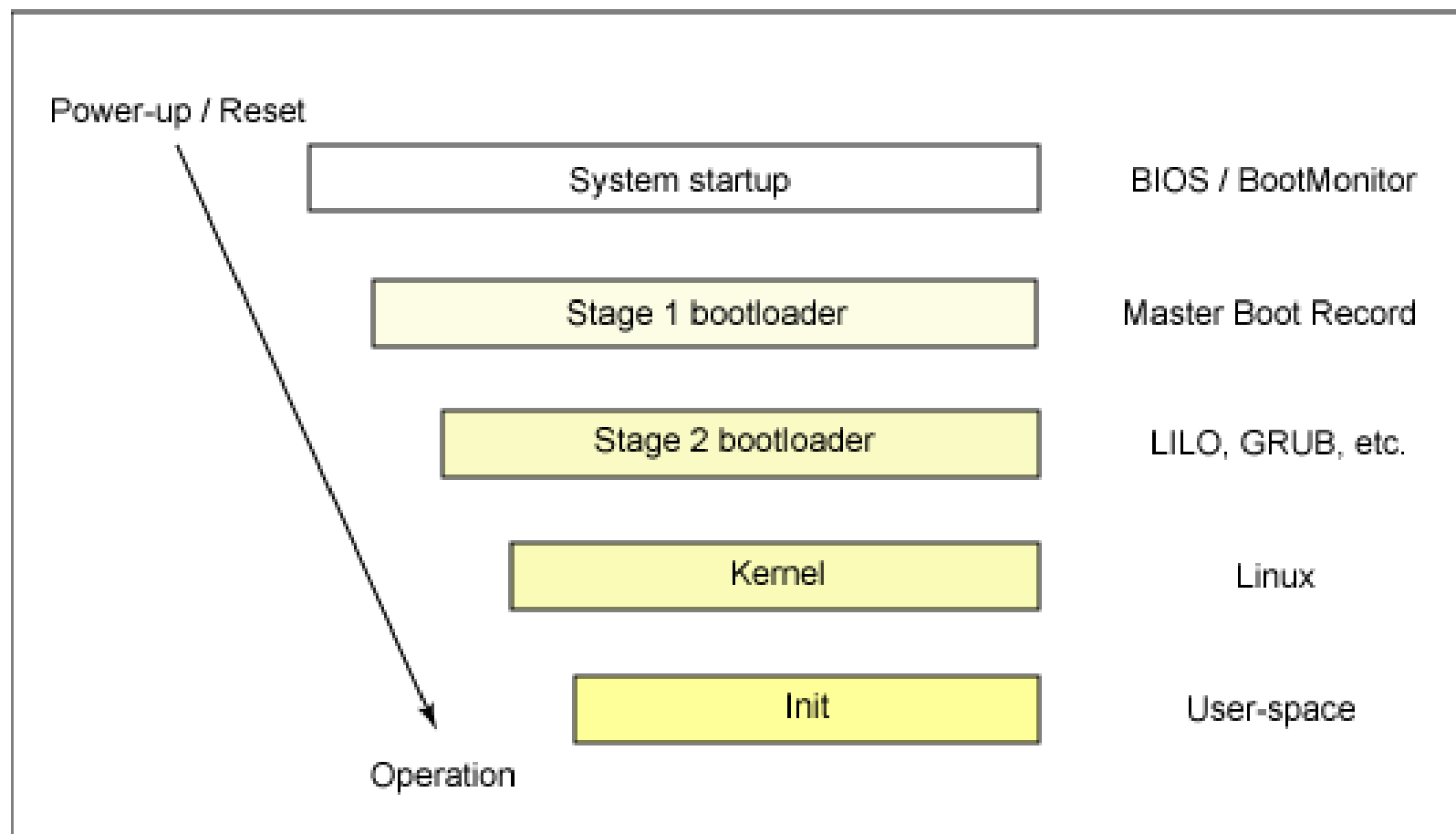
1. Linux Kernel Introduction (contd.) Booting

Chentao Wu
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn



上海交通大学

How Linux boot?

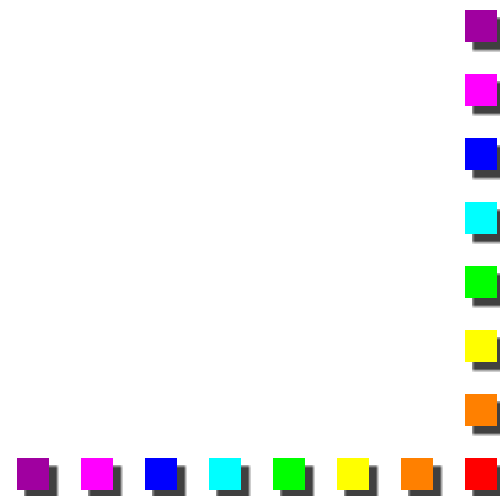


1. System Startup



How computer startup?

- Booting is a bootstrapping process that starts operating systems when the user turns on a computer system
- A boot sequence is the set of operations the computer performs when it is switched on that load an operating system



Booting sequence

1. Turn on
2. CPU jump to address of BIOS (0xFFFF0)
3. BIOS runs POST (Power-On Self Test)
4. Find bootable devices
5. Loads and execute boot sector from MBR
6. Load OS



BIOS (Basic Input/Output System)

- BIOS refers to the software code run by a computer when first powered on
- The primary function of BIOS is code program embedded on a chip that recognises and controls various devices that make up the computer.



BIOS on board



BIOS on screen



上海交通大学

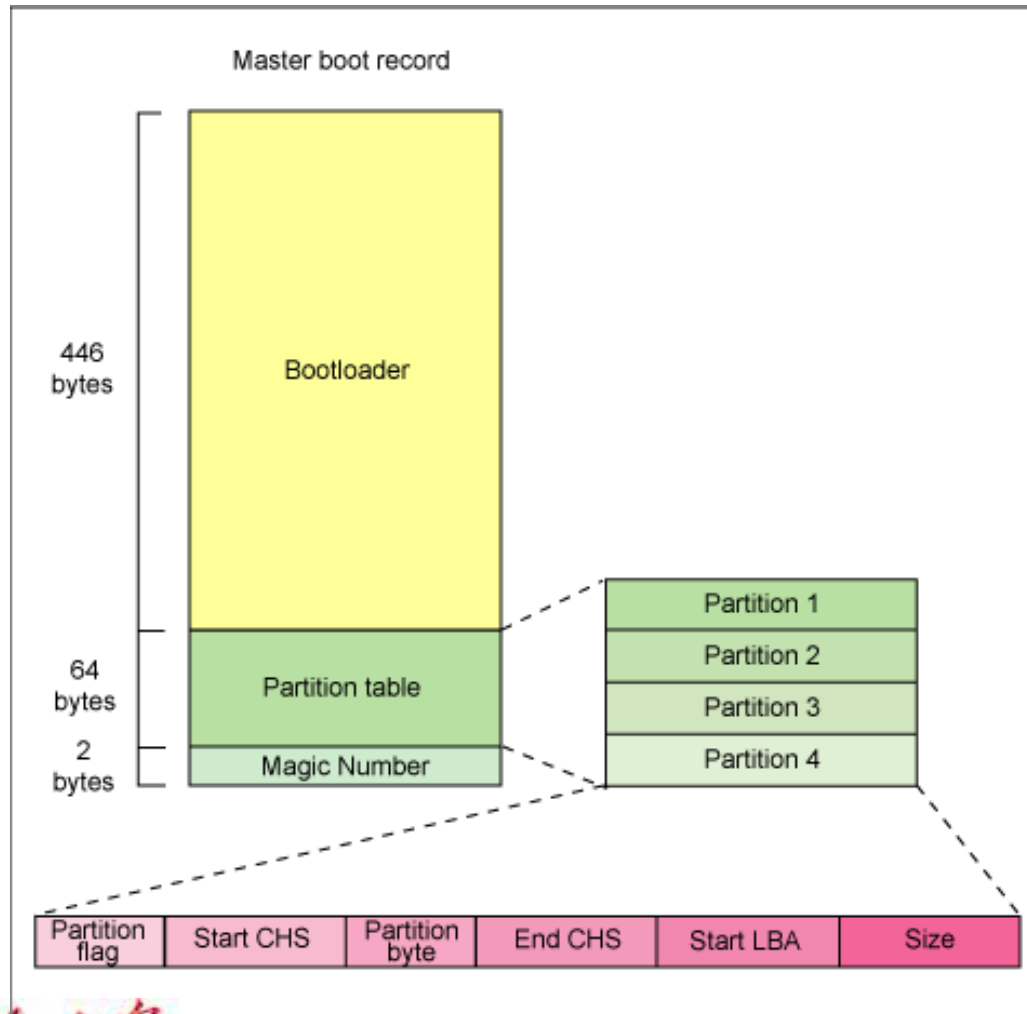
2. Bootloader

MBR (Master Boot Record)

- OS is booted from a hard disk, where the Master Boot Record (MBR) contains the primary boot loader
- The MBR is a 512-byte sector, located in the first sector on the disk (sector 1 of cylinder 0, head 0)
- After the MBR is loaded into RAM, the BIOS yields control to it.



MBR (Master Boot Record)



上海交通大学



MBR (Master Boot Record)

- The first 446 bytes are the primary boot loader, which contains both executable code and error message text
- The next sixty-four bytes are the partition table, which contains a record for each of four partitions
- The MBR ends with two bytes that are defined as the magic number (0xAA55). The magic number serves as a validation check of the MBR

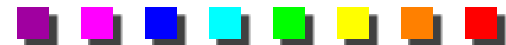


Extracting the MBR

- To see the contents of MBR, use this command:
- # dd if=/dev/hda of=mbr.bin bs=512 count=1
- # od -xa mbr.bin

**The dd command, which needs to be run from root, reads the first 512 bytes from /dev/hda (the first Integrated Drive Electronics, or IDE drive) and writes them to the mbr.bin file.

**The od command prints the binary file in hex and ASCII formats.



Boot loader

- Boot loader could be more aptly called the kernel loader. The task at this stage is to load the Linux kernel
- Optional, initial RAM disk
- GRUB and LILO are the most popular Linux boot loader.



Other boot loader (Several OS)

- bootman
- GRUB
- LILO
- NTLDR
- XOSL
- BootX
- loadlin
- Gujin
- Boot Camp
- Syslinux
- GAG



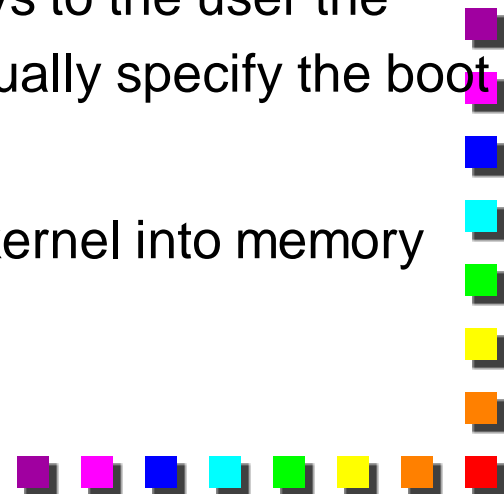
GRUB: GRand Unified Bootloader

- GRUB is an operating system independant boot loader
- A multiboot software packet from GNU
- Flexible command line interface
- File system access
- Support multiple executable format
- Support diskless system
- Download OS from network
- Etc.

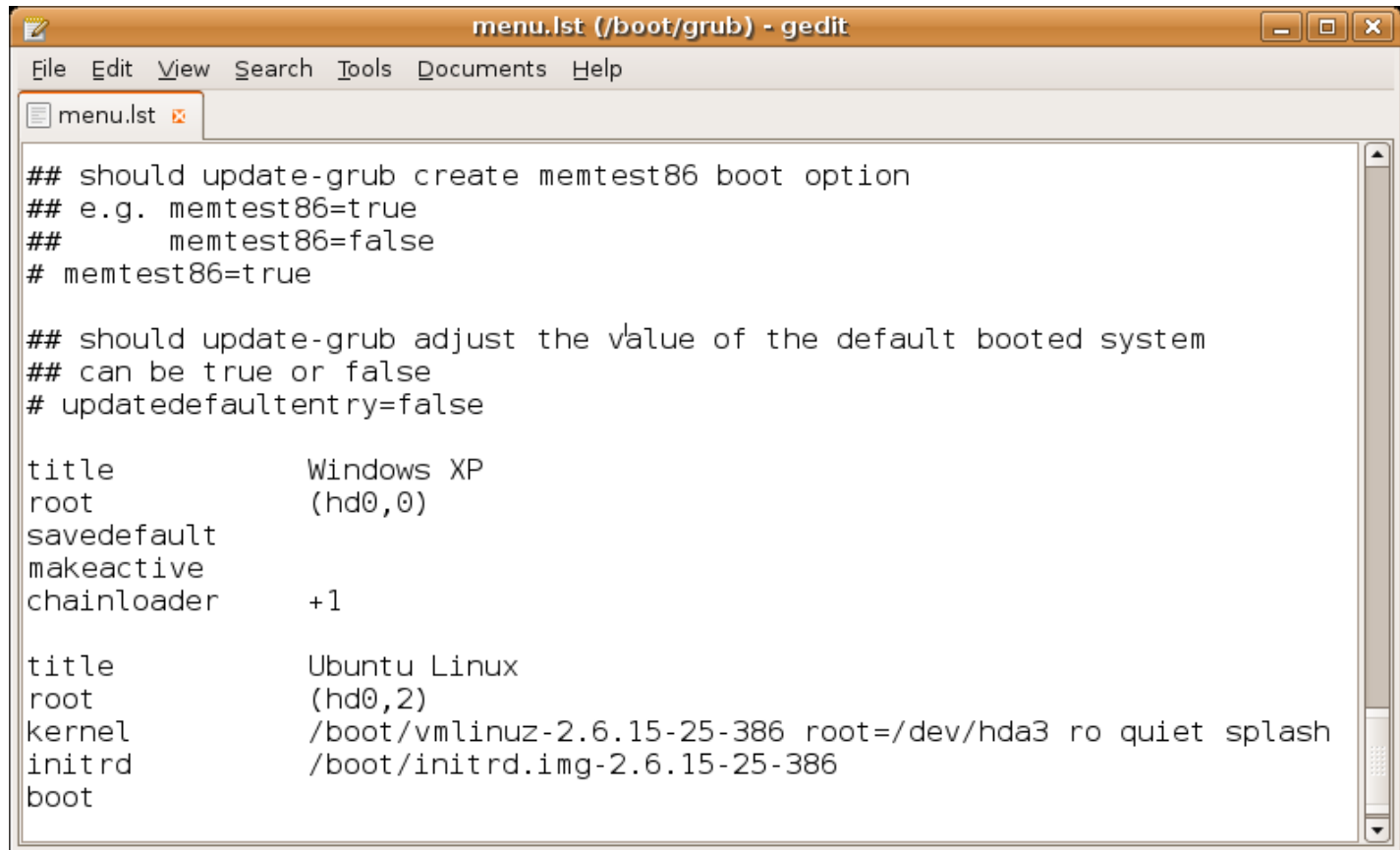


GRUB boot process

1. The BIOS finds a bootable device (hard disk) and transfers control to the master boot record
2. The MBR contains GRUB stage 1. Given the small size of the MBR, Stage 1 just load the next stage of GRUB
3. GRUB Stage 1.5 is located in the first 30 kilobytes of hard disk immediately following the MBR. Stage 1.5 loads Stage 2.
4. GRUB Stage 2 receives control, and displays to the user the GRUB boot menu (where the user can manually specify the boot parameters).
5. GRUB loads the user-selected (or default) kernel into memory and passes control on to the kernel.



Example GRUB config file



```
## should update-grub create memtest86 boot option
## e.g. memtest86=true
##      memtest86=false
# memtest86=true

## should update-grub adjust the value of the default booted system
## can be true or false
# updatedefaultentry=false

title          Windows XP
root            (hd0,0)
savedefault
makeactive
chainloader     +1

title          Ubuntu Linux
root            (hd0,2)
kernel          /boot/vmlinuz-2.6.15-25-386 root=/dev/hda3 ro quiet splash
initrd          /boot/initrd.img-2.6.15-25-386
boot
```



LILO: Linux LOader

- Not depend on a specific file system
- Can boot from harddisk and floppy
- Up to 16 different images
- Must change LILO when kernel image file or config file is changed



3. Kernel



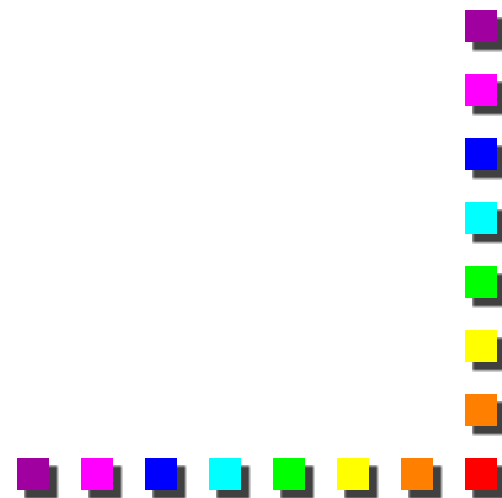
Kernel image

- The kernel is the central part in most computer operating systems because of its task, which is the management of the system's resources and the communication between hardware and software components
- Kernel is always store on memory until computer is tern off
- Kernel image is not an executable kernel, but a compress kernel image
- zImage size less than 512 KB
- bzImage size greater than 512 KB

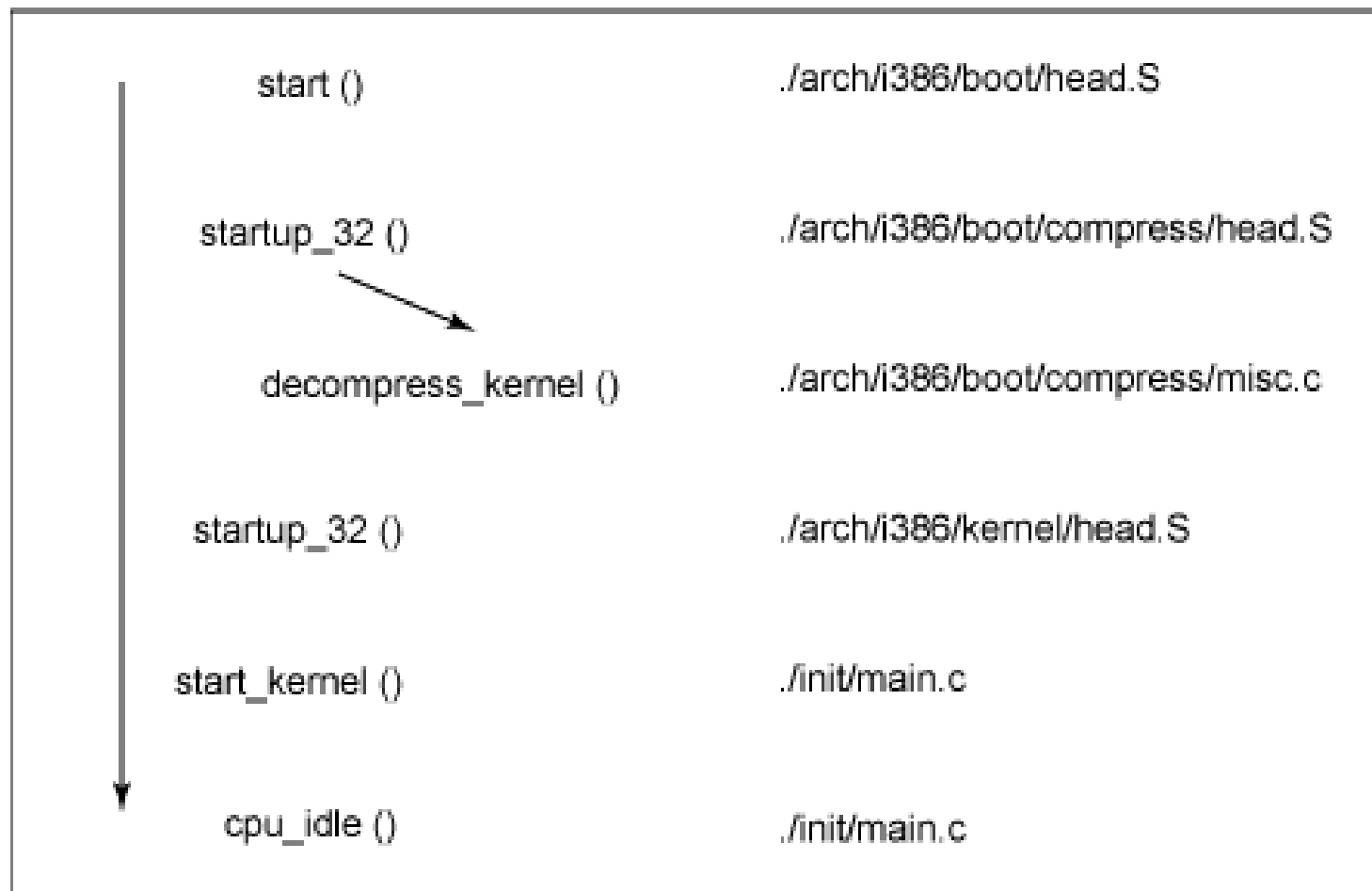


Task of kernel

- Process management
- Memory management
- Device management
- System call



Major functions flow for Linux kernel boot

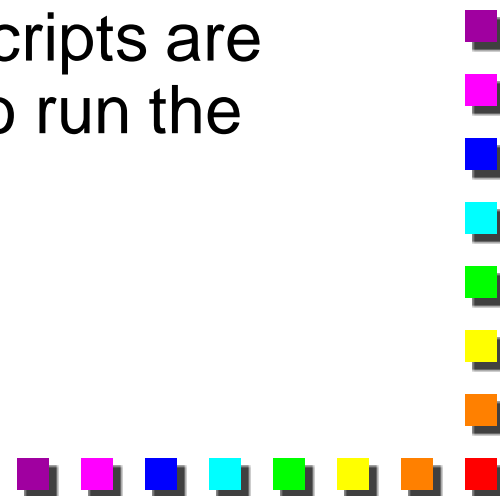


4. Init



Init process

- The first thing the kernel does is to execute init program
- Init is the root/parent of all processes executing on Linux
- The first processes that init starts is a script `/etc/rc.d/rc.sysinit`
- Based on the appropriate run-level, scripts are executed to start various processes to run the system and make it functional



The Linux Init Processes

- The init process is identified by process id "1"
- Init is responsible for starting system processes as defined in the /etc/inittab file
- Init typically will start multiple instances of "getty" which waits for console logins which spawn one's user shell process
- Upon shutdown, init controls the sequence and processes for shutdown



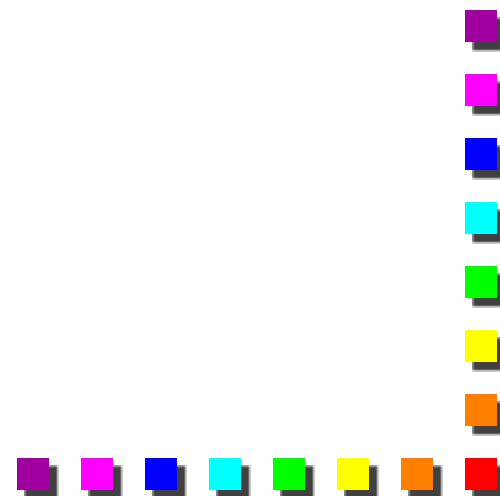
System processes

Process ID	Description
0	The Scheduler
1	The init process
2	kflushd
3	kupdate
4	kpiod
5	kswapd
6	mdrecoveryd



Inittab file

- The inittab file describes which processes are started at bootup and during normal operation
 - /etc/init.d/boot
 - /etc/init.d/rc
- The computer will be booted to the runlevel as defined by the initdefault directive in the /etc/inittab file
 - id:5:initdefault:



Runlevels

- A runlevel is a software configuration of the system which allows only a selected group of processes to exist
- The processes spawned by init for each of these runlevels are defined in the `/etc/inittab` file
- Init can be in one of eight runlevels: 0-6



Runlevels

Runlevel	Scripts Directory (Red Hat/Fedora Core)	State
0	/etc/rc.d/rc0.d/	shutdown/halt system
1	/etc/rc.d/rc1.d/	Single user mode
2	/etc/rc.d/rc2.d/	Multiuser with no network services exported
3	/etc/rc.d/rc3.d/	Default text/console only start. Full multiuser
4	/etc/rc.d/rc4.d/	Reserved for local use. Also X-windows (Slackware/BSD)
5	/etc/rc.d/rc5.d/	XDM X-windows GUI mode (Redhat/System V)
6	/etc/rc.d/rc6.d/	Reboot
s or S		Single user/Maintenance mode (Slackware)
M		Multiuser mode (Slackware)



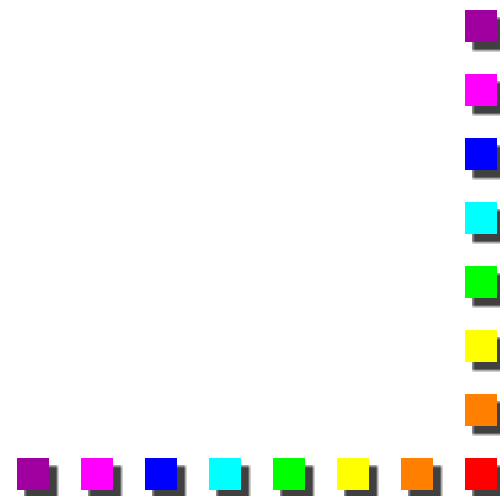
rc#.d files

- rc#.d files are the scripts for a given run level that run during boot and shutdown
- The scripts are found in the directory `/etc/rc.d/rc#.d/` where the symbol `#` represents the run level



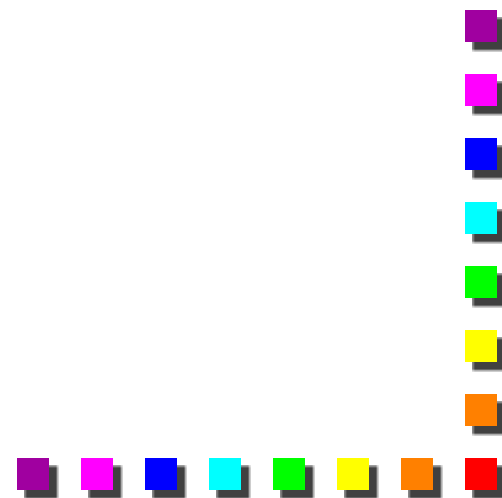
init.d

- Deamon is a background process
- init.d is a directory that admin can start/stop individual demons by changing on it
 - /etc/rc.d/init.d/ (Red Hat/Fedora)
 - /etc/init.d/ (S.u.s.e.)
 - /etc/init.d/ (Debian)



Start/stop daemon

- Admin can issuing the command and either the start, stop, status, restart or reload option
- i.e. to stop the web server:
 - `cd /etc/rc.d/init.d/`
 - (or `/etc/init.d/` for S.u.s.e. and Debian)
 - `httpd stop`



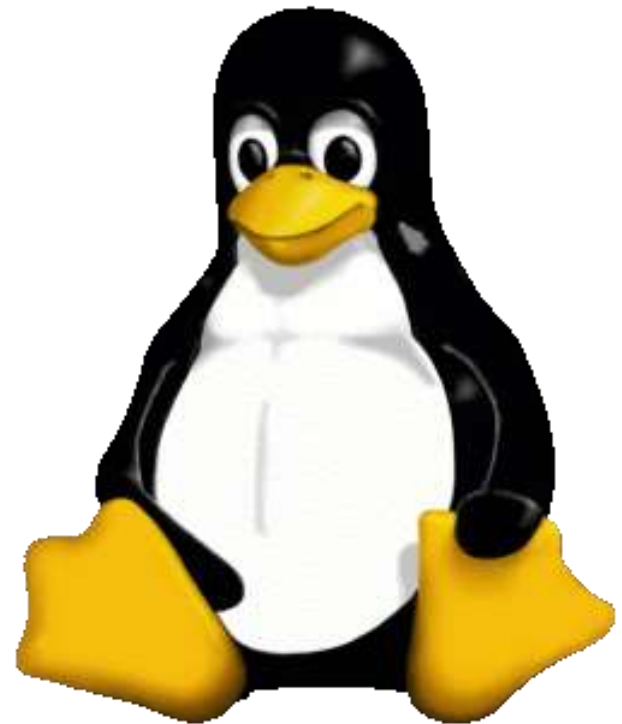
Read Source Code

- Source Insight 3.5
 - Download source code from <http://www.kernel.org>
- Web site:
 - <http://lxr.oss.org.cn/>
 - <http://lxr.free-electrons.com>



Linux系统启动 源代码分析

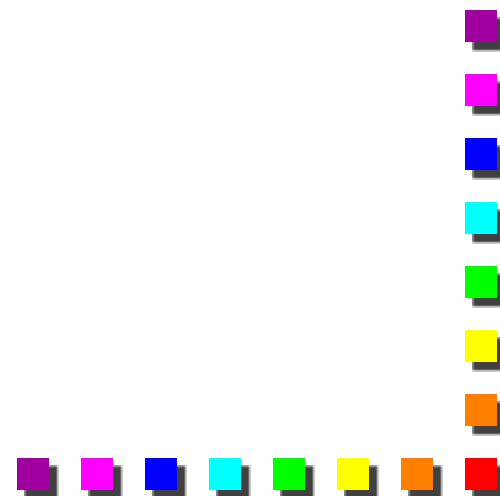
Chentao Wu
Assistant Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn



上海交通大学

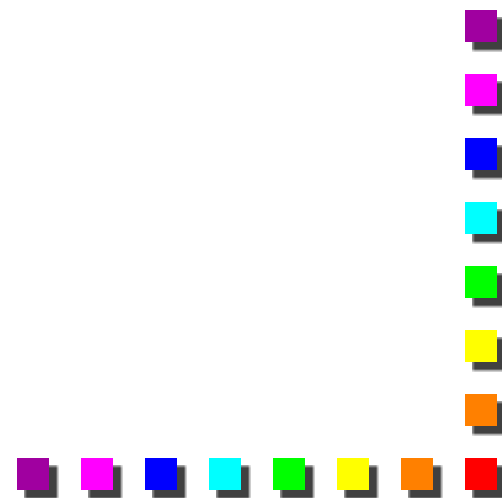
代码类型

- Linux源代码中的C语言代码
- Linux源代码中的汇编语言代码：两种
 - 完全的汇编代码，以.s作为文件名后缀
 - 嵌入在c程序中的汇编代码。



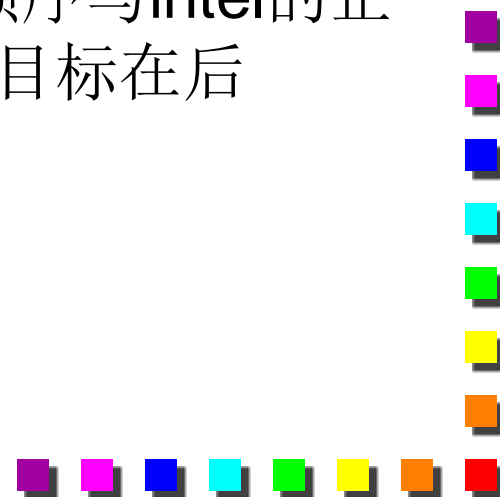
C语言代码

- Linux主体是用GNU的C语言编写
- 从c++中吸收了“inline”和“const”
- 支持“属性描述符”（attribute）
- 增加了新的基本数据类型“long long int”用于支持64位cpu



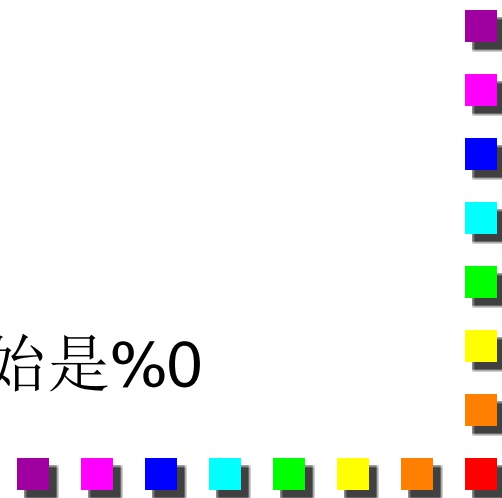
汇编语言代码

- 与一般的386汇编语言采用intel定义不同，它采用的是AT&T定义的格式。主要差别如下：
 - Intel中多使用大写字母，而这里大多使用小写字母
 - 寄存器名前面要加“%”作为前缀，
 - 指令的源操作数与目标操作数的顺序与intel的正好相反。AT&T格式中，源在前，目标在后



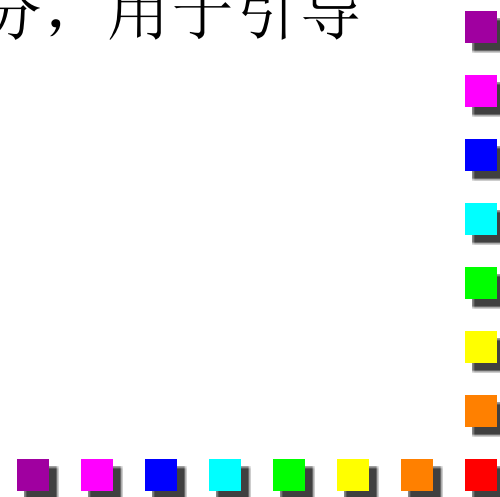
汇编语言代码

- 访问内存的指令的操作数大小（即宽度）由操作码名称的最后一个字母决定，用作操作码后缀的字母有b（8位），w（16位），l（32位），e.g movb
- 直接操作数要加“\$”作为前缀，intel中不用
- 基本格式
 - asm(“汇编语句”
:输出寄存器
:输入寄存器
:会被修改的寄存器);
- 输出和输入寄存器统一按顺序编号，起始是%0



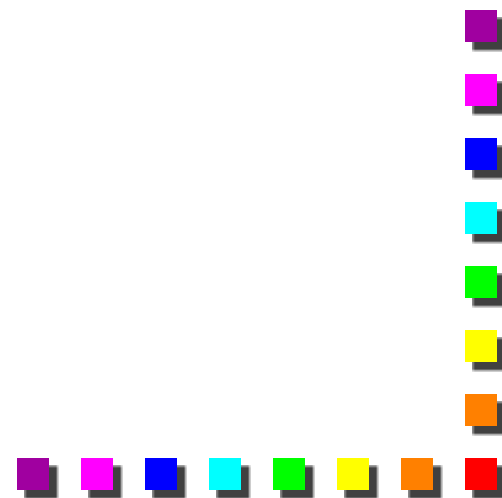
基础知识

- Linux的启动是指从系统加电到控制台显示登录提示为止的运行阶段：
 - 主要相关的代码是在arch/i386/boot中：
 - bootsect.S,这是linux引导扇区的源代码
 - setup.S这是辅助程序的一部分
 - video.S这是辅助程序的另外一部分，用于引导过程中的屏幕显示



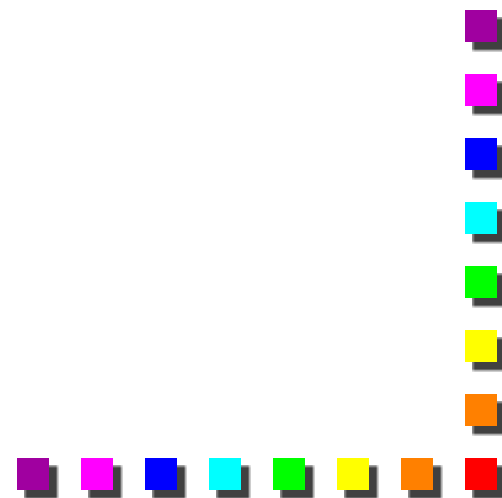
基础知识

- 另外，子目录**compressed**中还有两个源代码文件 **head.S**, **misc.c**。用于内核映象的解压缩。也属于辅助程序一部分。
- 经过编译，汇编和连接后就形成三个部分：引导扇区的映象**bootsect**，辅助程序**setup**和内核映象本身。
- 大小不超过**508KB**的内核引导映象称为小映象 **zImage**；否则称为大内核**bzImage**



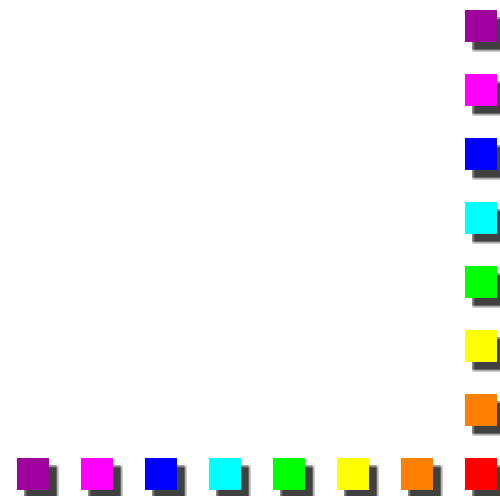
系统引导

- 加电开机后，intel cpu在实模式下工作，只能使用低端的640kb（即0xA0000以下）的内存空间(why?)
- 由ROM BIOS或者lilo将启动盘的第一扇区（引导扇区）的内容装入起始地址为0x7c00的内存空间，然后跳转到0x7c00开始执行引导扇区的代码
- 该引导扇区内的代码就是bootset.S汇编后生成的二进制代码



系统引导

- 该段代码(`bootset.S`)将自身转移到`0x90000`处，然后跳转到那里继续执行，并通过**bios**提供的“`int 0x13`”调用从磁盘上读入**setup**和内核的映像，然后跳转到**setup**的代码中，为执行内核映像做准备
- 对部分代码的解释如下所示：



系统引导—bootsect.S中的部分代码

```
movw $BOOTSEG, %ax
movw %ax, %ds          # %ds = BOOTSEG, 将ds段寄存器设为0x7c00
movw $INITSEG, %ax
movw %ax, %es          # %ax = %es = INITSEG, 将es段寄存器设为0x9000
movw $256, %cx         # 移动计数值=256
subw %si, %si          # 源地址ds:si=0x07c0:0x0000
subw %di, %di          # 目标地es:di=0x9000:0x0000
cld                   # 清方向标志位
rep                   # 重复执行直到cx=0
movsw                 # 移动1个字
ljmp $INITSEG, $go     # 间接跳转, INITSEG指出跳转到的段地址
```

...

```
go:  movw    $0x4000-12, %di          # 0x4000 is an arbitrary value >=
                                         # length of bootsect + length of
                                         # setup + room for stack;
                                         # 12 is disk parm size.

      movw    %ax, %ds              # 将ds,ss都置成移动后代码所在的段处0x9000
      movw    %ax, %ss
      movw    %di, %sp              # 设置堆栈put stack at INITSEG:0x4000-12.
```

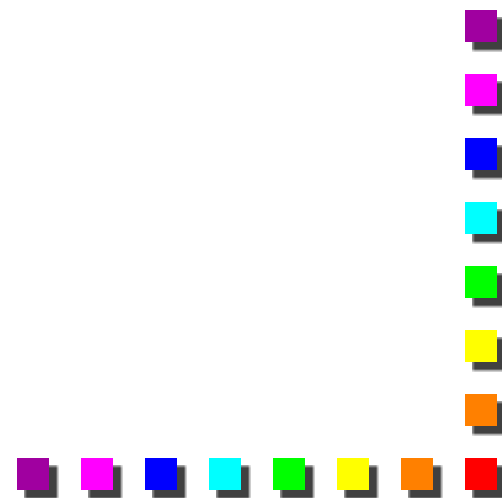


上海交通大学



系统引导—bootsect.S中部分代码的解释

- 这段代码将启动扇区代码由0x7C00移至0x90000处。Linux将地址为0x90000的代码段称为INITSEG。然后跳转到go标志，准备一块堆栈，栈底位于\$INITSEG:0x4000-12



系统引导—bootsect.S中部分代码

load_setup:

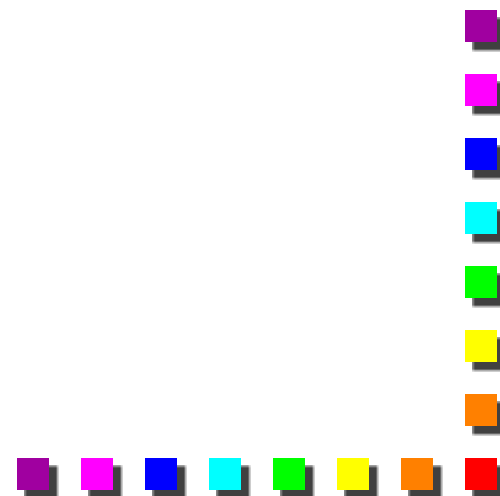
```
xorb  %ah,%ah  #ah=0x02  读磁盘扇区到内存； al=需读出的扇区数， ...
xorb  %dl,%dl
int    $0x13
xorw   %dx,%dx
movb   $0x02,%cl
movw   $0x0200,%bx
movb   $0x02,%ah
movb   setup_sects,%al
int    $0x13
jnc    ok_load_setup
```

```
pushw %ax
call   print_nl
movw   %sp,%bp
call   print_hex
popw   %ax
jmp    load_setup
```

ok_load_setup:

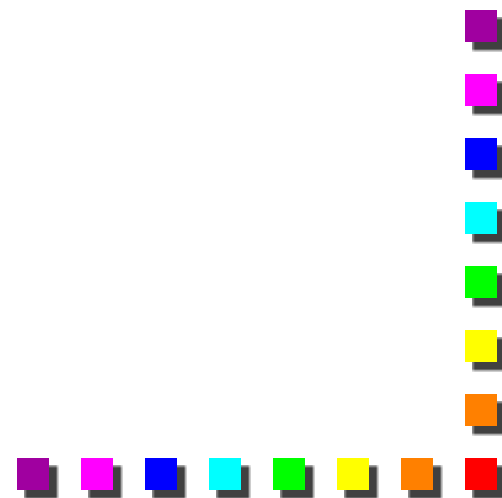


上海交通大学



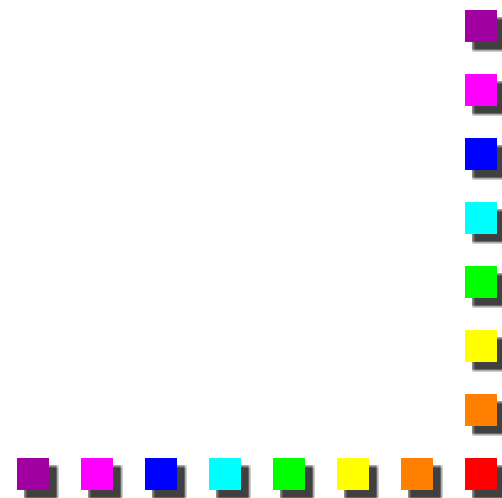
系统引导—bootsect.S中部分代码的解释

- 该段代码利用BIOS中提供的读磁盘调用“int 0x13”从磁盘将setup.S装入到9000: 0200（linux中称之为SETUPPSEG段），即紧跟在bootsect.S之后，共四个扇区
- 如果载入失败，则不断尝试循环。除非某次尝试成功，否则只有等待系统重启



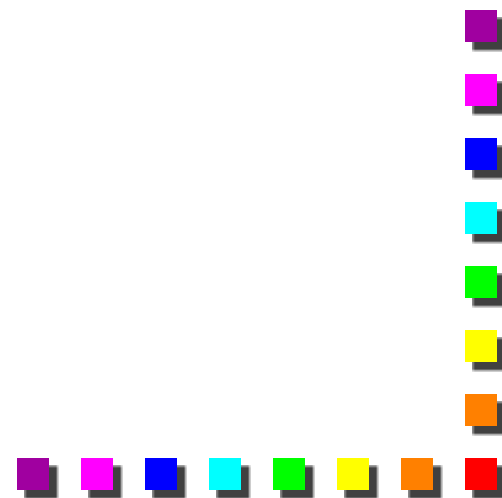
通过LILO来进行引导

- LILO (linux loader) 也存储在启动扇区中, 用以让用户选择上电后使用何种操作系统
- LILO在系统安装阶段建立关于核心代码占用硬盘数据块的位置的对照表。启动时LILO将利用这张表引导BIOS装入指定的操作系统



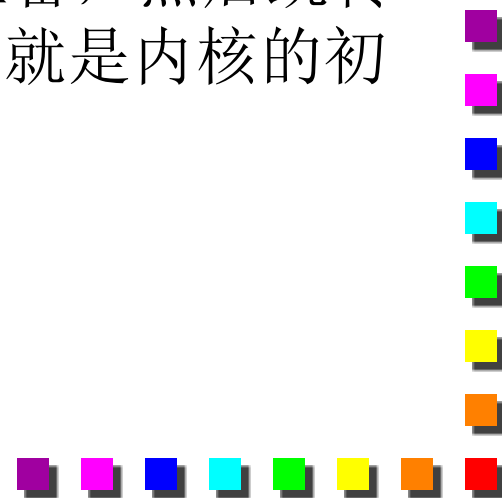
通过LILO来进行引导

- LILO将用户在启动时输入的命令和参数存储在empty_zero_page(0x5000)的后半页，供arch/i386/kernel/setup.c文件的setup_arch()函数使用
- LILO完成任务后，跳转至setup.S程序，转入实模式下的系统初始化



实模式下的系统初始化

- `setup.S`连同内核映像由`bootsect.S`装入。`setup.S`从BIOS获取计算机系统的参数，放到内存参数区，仍在实模式下运行
- Cpu在`setup`的执行过程中转入32位保护模式的段式寻址方式
- 辅助程序`setup`为内核映像的执行做好准备，然后跳转到`0x100000`开始内核本身的执行，此后就是内核的初始化过程



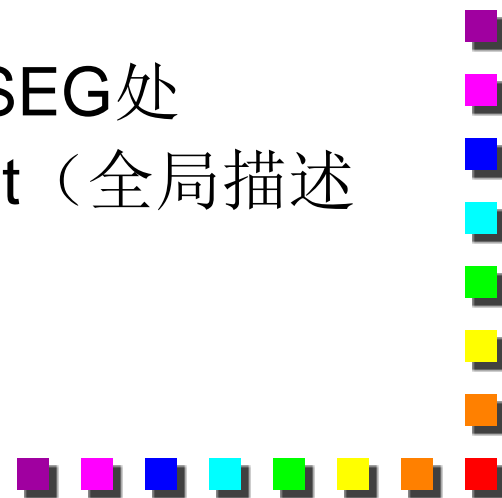
实模式下的系统初始化—setup.S

■ 版本检查和参数设置

- 检查签名“55AA5A5A”，此签名位于setup.S代码段的末尾，判断安装程序是否完全安装进来
- 判断核心（kernel）是否为BIG_KERNEL
- 设置参数

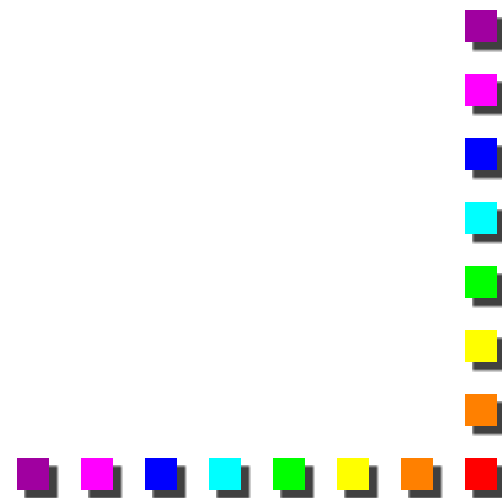
■ 为进入保护模式做准备，主要包括

- 关中断
- 检查自身（setup.S）是否在SETUPSEG处
- 置idt（中断描述符表）为空，设置gdt（全局描述符表）
- 真正进入保护模式



保护模式下的系统初始化

- 保护模式下的核心初始化模块从0x10000开始执行，负责检查数据区，idt表，页表和寄存器的初始化，同时进行一些必要的状态检查，最后转入start_kernel()模块。如果核心系统是压缩存放的，则先执行解压缩。保护模式下的初始化主要包括：



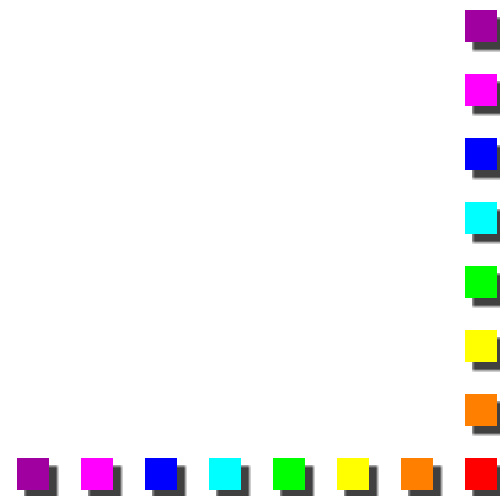
保护模式下的系统初始化

- 初始化寄存器和数据区
- 核心代码解压缩
- 页表初始化
- 初始化idt, gdt和ldt
- 启动核心

下面分项介绍

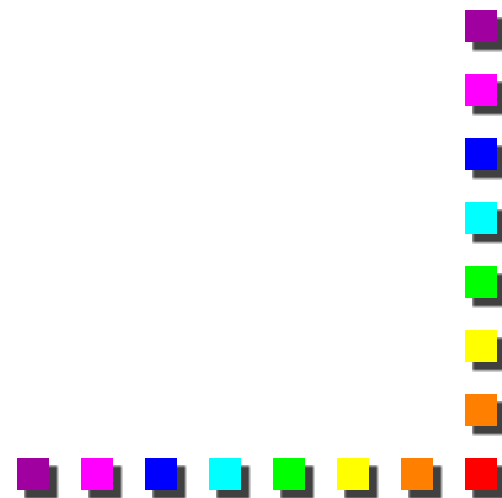


上海交通大学



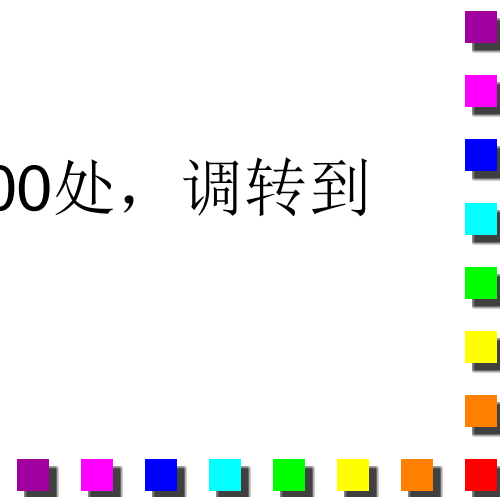
初始化寄存器和数据区

- Arch/i386/boot/compressed目录下的head.S是段保护模式的汇编程序，先设置堆栈，然后调用同目录下的misc.c文件的decompress_kernel()函数解压缩
 - 设置堆栈与寄存器
 - 检查A20线是否有效
 - 数据区BSS全部清零
 - 转入核心代码解压缩过程



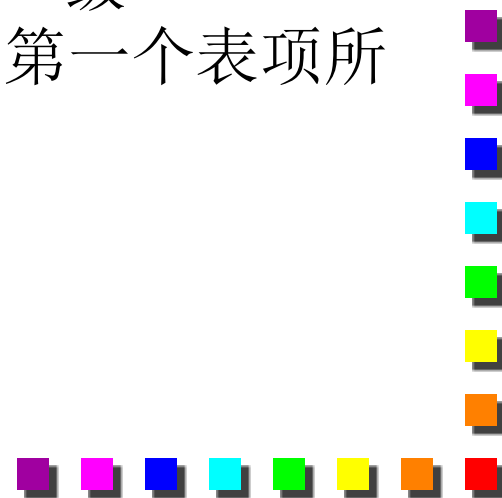
核心代码解压缩

- 调用mics.c中的decompress_kernel开始解压缩。解压缩的步骤为：
 - 设置output_buffer
 - Makecrc: 建立一张CRC（校验）表（lib/inflate.c）
 - 调用gunzip()解压缩，同时比较CRC表，如果不一致说明解压出错。
 - 检查kernel的大小
 - 解除压缩以后的内核映像放在0x10000处，调转到此处执行。



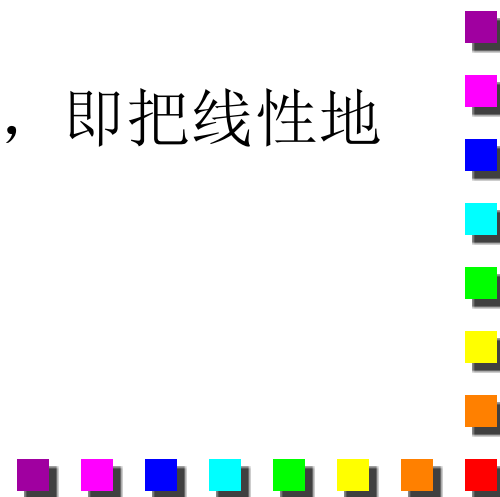
页表初始化

- 进行解压缩后，核心系统的入口就是arch/i386/kernel目录下的head.S。系统先初始化寄存器和数据区，然后执行以下步骤：
 - 将ds, es, fs, gs寄存器初始化为_KERNEL_DS的值
 - 进行两级页表的部分初始化。其中第一级swapper_pg_dir是页目录，页目录的第一个表项所指的二级页表称为pg0



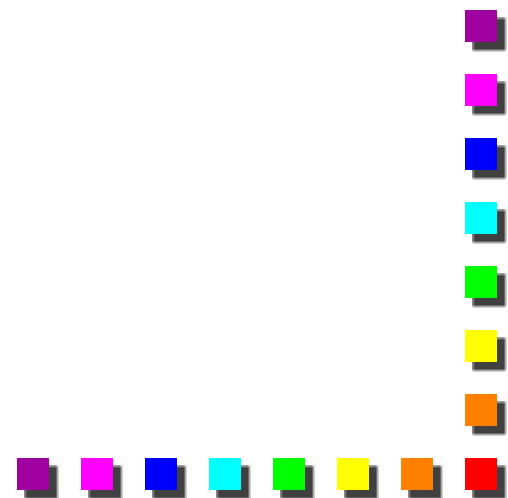
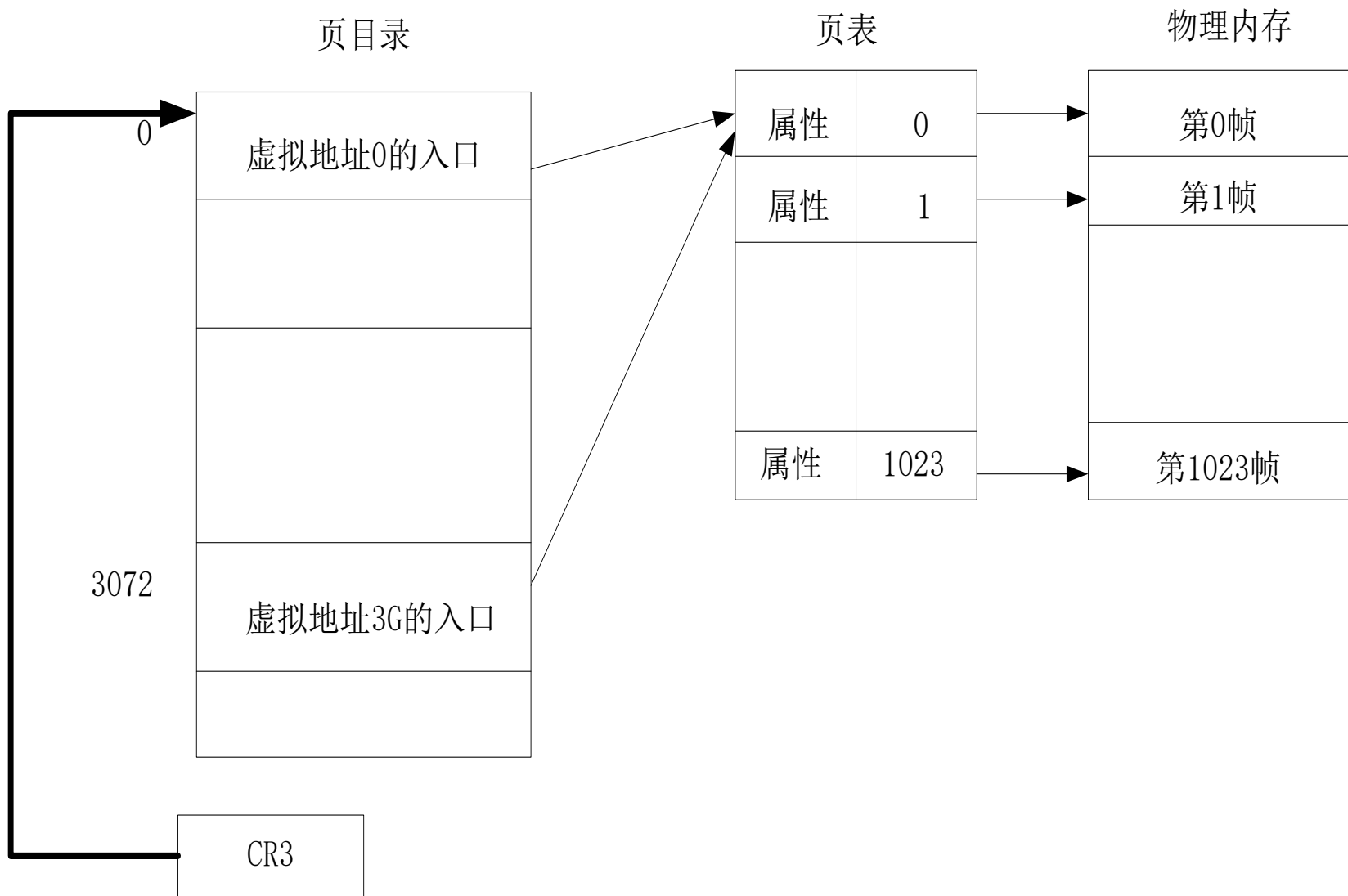
页表初始化

- 清空BSS区（未初始化数据区）
- 跳转到setup_idt处对idt表进行初始化
- 复制bootup参数到empty_zero_page
- 检查cpu类型
- 下面详细解释一下如何进行两级页表的初始化：
 - 先把swapper_pg_dir清零
 - Pg0登记在页目录的第0项和第768项，即把线性地址0和3G都指向pg0



- 初始化二级页表pg0和pg1
 - CPU控制寄存器的初始化：使CR3指向swapper_page_dir，将CR0的PG位置位(CPU的paging功能启动位)。CPU的页管理功能便生效
- 两级页表初始化的图示如下：



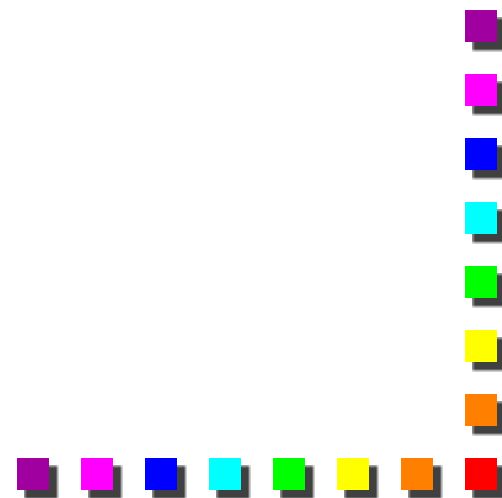


初始化idt, gdt, ldt (三步)

■ 1、初始化gdt

Gdt表项数=2个内核态段+两个用户态段+4个空闲表项+4个APM段+ $2 \times \text{NRTASK}$ 个用于LDT和TSS描述的段。

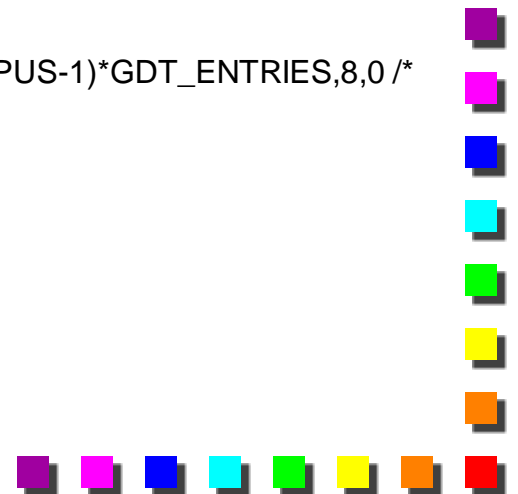
Gdt的初始化的代码如下页图示：



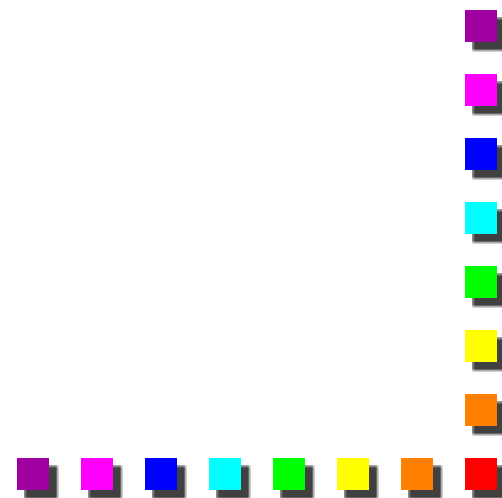
ENTRY(cpu_gdt_table)		/* Segments used for calling PnP BIOS */	
.quad 0x0000000000000000	/*	.quad 0x00c09a0000000000	/*
NULL descriptor */		0x80 32-bit code */	
.quad 0x0000000000000000	/*	.quad 0x00809a0000000000	/*
0x0b reserved */		0x88 16-bit code */	
.quad 0x0000000000000000	/*	.quad 0x0080920000000000	/*
0x13 reserved */		0x90 16-bit data */	
.quad 0x0000000000000000	/*	.quad 0x0080920000000000	/*
0x1b reserved */		0x98 16-bit data */	
.quad 0x00cffa000000ffff /* 0x23 user		.quad 0x0080920000000000	/*
4GB code at 0x00000000 */		0xa0 16-bit data */	
.quad 0x00cff2000000ffff /* 0x2b user		/*	
4GB data at 0x00000000 */		* The APM segments have byte granularity	
.quad 0x0000000000000000	/*	and their bases	
0x33 TLS entry 1 */		* and limits are set at run time.	
.quad 0x0000000000000000	/*	*/	
0x3b TLS entry 2 */		.quad 0x00409a0000000000	/*
.quad 0x0000000000000000	/*	0xa8 APM CS code */	
0x43 TLS entry 3 */		.quad 0x00009a0000000000	/*
.quad 0x0000000000000000	/*	0xb0 APM CS 16 code (16 bit) */	
0x4b reserved */		.quad 0x0040920000000000	/*
.quad 0x0000000000000000	/*	0xb8 APM DS data */	
0x53 reserved */		#if CONFIG_SMP	
.quad 0x0000000000000000	/*	.fill (NR_CPUS-1)*GDT_ENTRIES,8,0 /*	
0x5b reserved */		other CPU's GDT */	
.quad 0x00cf9a000000ffff /* 0x60 kernel		#endif	
4GB code at 0x00000000 */			
.quad 0x00cf92000000ffff /* 0x68 kernel			
4GB data at 0x00000000 */			
.quad 0x0000000000000000	/*		
0x70 TSS descriptor */			
.quad 0x0000000000000000	/*		
0x78 LDT descriptor */			



上海交通大学



- 2、设定idt寄存器为idt_descr变量的当前值，指向idt表（共256项），但目前不允许中断（尚未设置中断门）
- 3、在新的页管理方式下，重新设置堆栈，段选择寄存器，描述符寄存器。



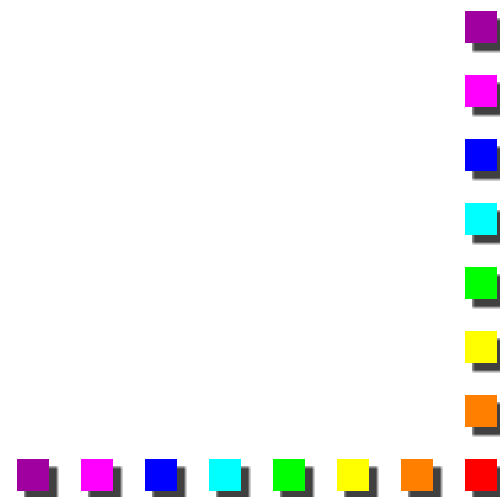
启动核心

- 前面对CPU进行了初始化，并启动了保护模式
- 现在的任务是初始化内核的核心数据结构，这些数据结构主要涉及：
 - 中断管理
 - 进程管理
 - 内存管理
 - 设备管理
- 各种数据结构纷繁复杂，需要对各部分进行分析



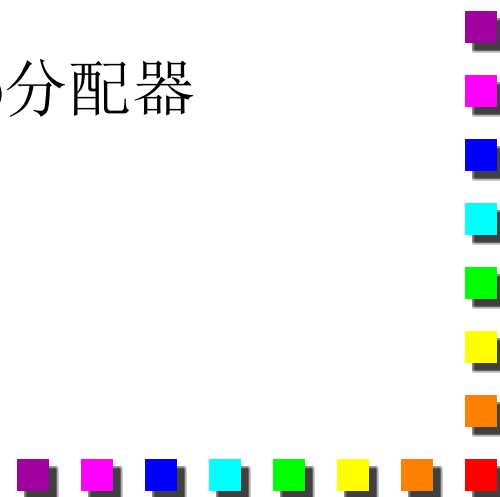
启动核心

- 进入保护模式后，系统从`start_kernel`处开始执行，`Start_kernel()`函数变成0号进程，不再返回
- `Start_kernel`显示版本信息，调用`setup_arch()` (`arch/i386/kernel/setup.c`):初始化核心的数据结构
- 最后，调用`kernel_thread()`创建`init`进程，进行系统配置
- 该部分的代码在`init/main.c`中



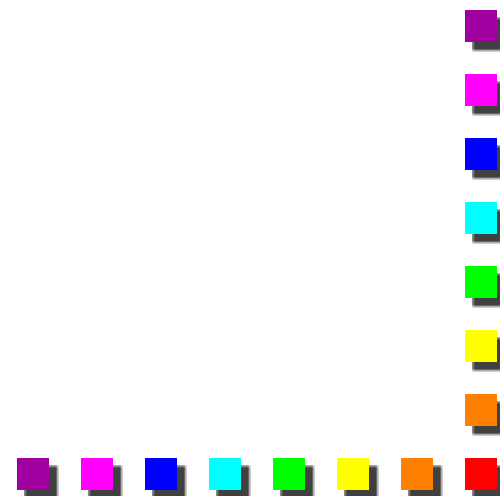
启动核心

- 核心数据结构的初始化
 - 调用paging_init()初始化页表
 - 调用mem_init()初始化页描述符
 - 调用trap_init()和init_IRQ()完成IDT最后的初始化工作
 - 调用k_mem_cache_init()和kmem_cache_sizes_init()初始化slab分配器



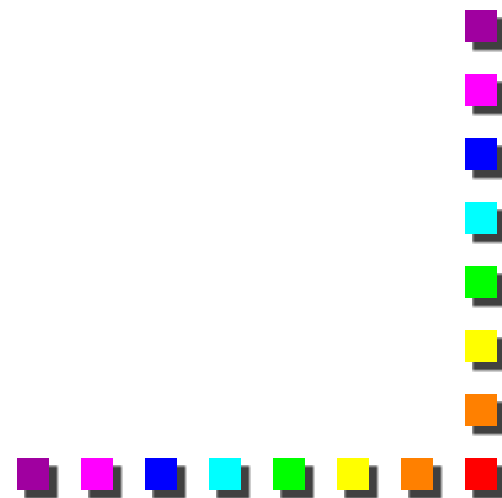
启动核心

- 调用`time_init()`初始化系统日期和时间
- 调用`kernel_thread`为进程1创建内核线程
- 父进程创建`init`子进程之后，返回执行`cpu_idle`



Init进程和系统配置

- Init进程（1号进程）首先创建一些后台进程来维护系统，然后进行系统配置，执行shell编写的初始化程序。然后转入用户态运行
- Init进程的执行流程如下



```

static int init(void * unused)
{
    lock_kernel();

    smp_init();
#if CONFIG_SMP
    migration_init();
#endif
    do_basic_setup();

    prepare_namespace();

    /*
     * Ok, we have completed the initial
    bootup, and
     * we're essentially up and running.
    Get rid of the
     * initmem segments and start the
    user-mode stuff..
     */
    free_initmem();
    unlock_kernel();

    if (open("/dev/console", O_RDWR, 0)
    < 0)
        printk("Warning: unable
to open an initial

```

```

console.\n");

    (void) dup(0);
    (void) dup(0);

    /*
     * We try each of these until one
    succeeds.
     *
     * The Bourne shell can be used
    instead of init if we are
     * trying to recover a really broken
    machine.
     */

    if (execute_command)

        execve(execute_command,argv_init,
envp_init);
    ;
        execve("/sbin/init",argv_init,envp_init)
        execve("/etc/init",argv_init,envp_init);
        execve("/bin/init",argv_init,envp_init);
        execve("/bin/sh",argv_init,envp_init);
        panic("No init found. Try passing
init= option to kernel.");
    }

```

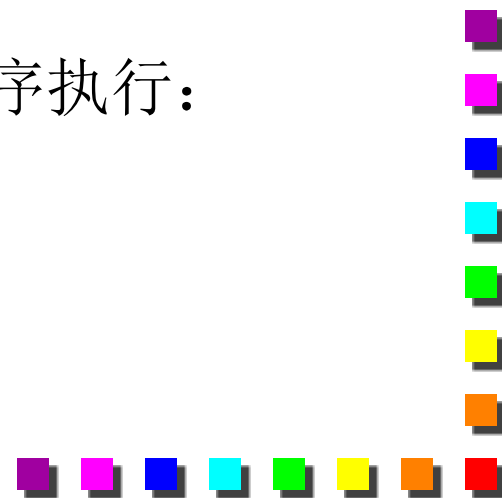


上海交通大学



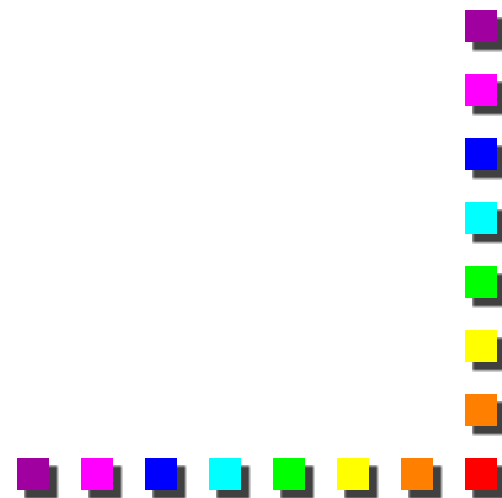
对init进程的解释

- 首先调用函数do_basic_setup()做系统初始化的工作（这之前系统只启动了cpu，内存和一些进程管理方面的工作）
- 调用free_initmem()函数，将初始化过程中使用的范围在_init_begin和_init_end之间的页面释放给空闲页面链表
- 打开一个控制台设备
- 如果存在指定命令就执行，否则，按顺序执行：



Init进程

- 如果存在 “/sbin/init”文件，则跳转去执行 “/sbin/init”
- 如果存在 “/etc/init”文件，则跳转去执行 “/etc/init”
- 如果存在 “/bin/init”文件，则跳转去执行 “/bin/init”
- 如果存在 “/bin/sh”文件，则跳转去执行 “/bin/sh”



思考题

- 在i386中，内核可执行代码在内存中的首地址是否可随意选择？为什么？
- 主引导扇区位于硬盘什么位置？如果一个硬盘的主引导扇区有故障，此硬盘是否还可以使用？
- 在没有LILO的情况下，系统是怎样引导的
- 进入保护模式为什么要打开A20地址线？
- Linux内核在实模式下的初始化完成哪些功能？
- 进程0和init进程的主要任务是什么？

