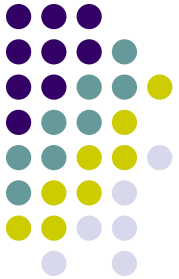
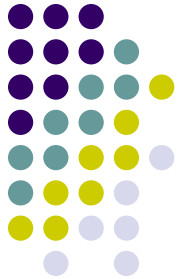


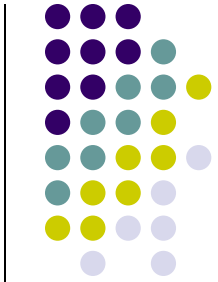
Lab1讲义

目录



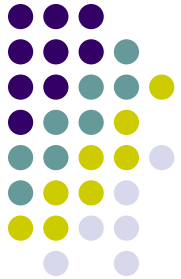
- Lab1代码树
- JOS的启动过程
- 实模式vs.保护模式
- ELF文件格式
- 显示输出
- JOS堆栈结构
- AT&T汇编和内联汇编

Lab1代码树



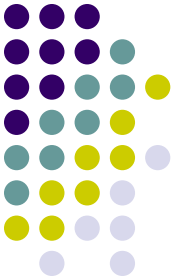
- lab1
  - boot/　引导扇区代码
  - inc/　头文件定义
  - kern/　内核代码
  - lib/　JOS的C库
  - user/　用户态程序
  - CODING　代码规范
  - GNUmakefile　这个Makefile最好读一下，代表整体组装的过程
  - conf/,mergedep.pl　相关文件
  - grade.sh　代码测试脚本

AT&T汇编



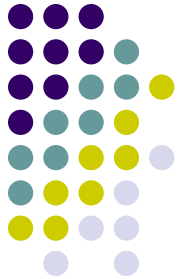
- 寄存器引用
- 立即数使用
- 操作数顺序
- 指令后缀
- 内存寻址

AT&T汇编语法(1)——寄存器引用



- 引用寄存器要在寄存器号前加百分号
  - %eax, %ebx
- 80386有如下寄存器
  - 8个32-bit寄存器
    - %eax，%ecx，%edx，%ebx，%esi，%edi，%esp，%ebp
  - 8个16-bit寄存器（ 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（%ax ~%dx的高8位和低8位）
    - %ah，%al，%bh，%bl，%ch，%cl，%dh，%dl
  - 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；

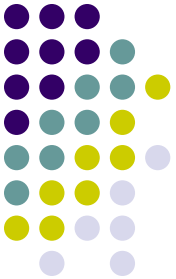
AT&T汇编语法(2)——立即数



- 使用立即数，要在数前面加符号$
  - $0x04
  - \lab1\boot\boot.S
    (1) testb$0x2,%al

    (2) .set CR0_PE_ON,0x1
        orl $CR0_PE_ON, %eax

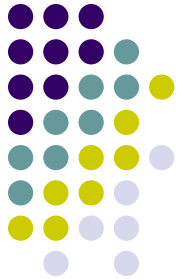
AT&T汇编语法(3)——操作数顺序



- 操作数顺序
  - ops source, target
  - 操作数顺序是AT&T语法的主要特征
- 示例
  - \lab1\boot\boot.S

AT&T	Intel
orl $0x1,%eax	**or eax,0x1**
movw %ax,%ds	**mov ds,ax**

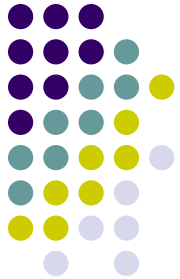
AT&T汇编语法(4)——指令后缀



- 指令后缀
  - **b代表byte(8bit)**
  - **w代表word(16bit)**
  - **l代表long(32bit)**
- 示例
  - **\lab1\boot\boot.S**

AT&T		Intel	
movw	**%ax,%ds**	**mov**	**ds,ax**
movb	**$0xdf,%al**	**mov**	**al,0xdf**
movl	**%eax, %cr0**	**mov**	**cr0,eax**

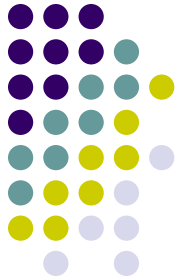
AT&T汇编语法(5)——内存寻址



- 寻址方式
  - **AT&T: displacement(base,index,scale)**
  - **Intel: [base+index*scale+displacement]**
- 示例
  - **-4(%ebp)**
    - **base = %ebp**
    - **displacement = -4**
    - **index,scale没有指定，则index为0**

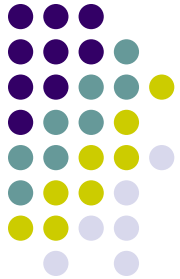
AT&T	Intel
movl -4(%ebp), %ecx	**mov ecx,dword ptr [ebp-4]**

AT&T汇编语法 (6)



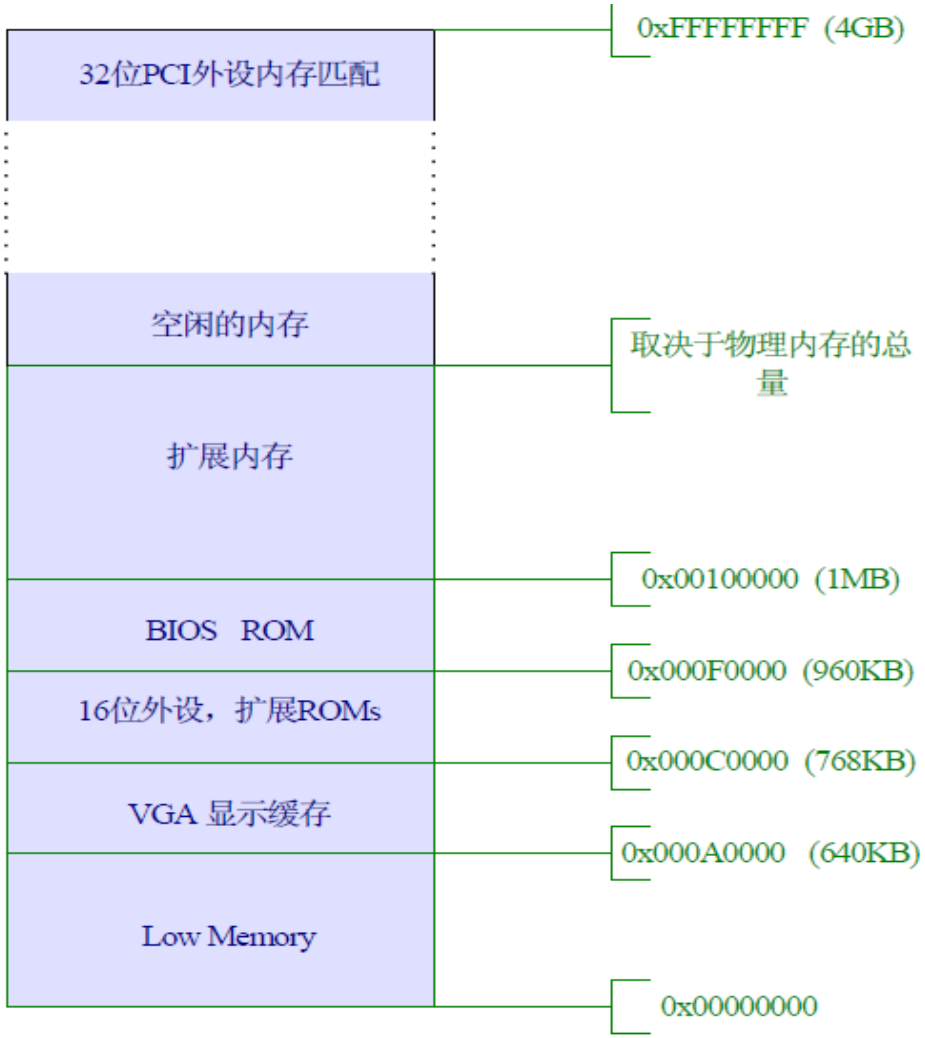
- 绝对转移与相对转移
  - 绝对转移指令直接获取目标地址，并跳转
    - **movl $do_pgfault, %eax**
    - **jmp *%eax;**
  - 相对转移指令先获取偏移量，再将偏移量加上“当前EIP”得到目标地址
    - **jmp .-100**
    - **jmp do_pgfault**
  - 汇编器会根据跳转范围自动生成相对跳转指令的偏移量

AT&T汇编语法 (7)



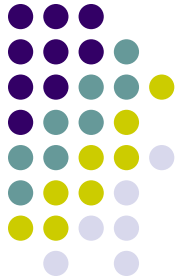
- 16位指令与32位指令
  - 助记符相同，但机器码不同
  - CPU工作在32位模式下时不能执行16位指令，反之亦然
  - .code16和.code32指示符表明以下代码按照16位还是32位汇编成机器码

PC开机后默认物理内存分布



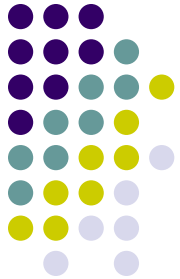
- 系统可以使用的内存
  - **0x00000-0xFFFFF** 早期计算机使用的内存
  - **0x00100000-0xFFFFFFFF** intel80386之后支持的扩展内存
- 系统保留的内存
  - **0xA0000-0xFFFFF** 为硬件保留
  - 通常**32**位地址空间的最高部分被保留给**32**位的**PCI**使用
- 设计局限所致，**JOS**系统只使用物理内存的前**256MB**

JOS启动过程



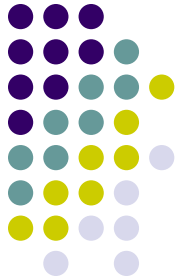
- PC加电启动
  - Bochs模拟硬件，可以通过修改Bochs的配置调整硬件
- BIOS运行
  - BIOS程序（BIOS-bochs-latest文件）被Bochs加载到0xF0000-0x100000
  - 处于实模式下，寻址空间1M
  - 检查和初始化硬件，加载引导扇区到0000:7c00 –0000:7dff

JOS启动过程



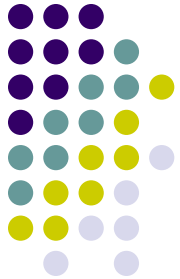
- 引导扇区（即BootLoader）执行
  - BIOS的最后一条指令是跳转到0x7c00处，把CPU控制权交给了BootLoader程序
  - 实模式转换为保护模式
  - 加载内核文件
- 内核开始执行
  - 初始化内核数据结构
  - 运行终端

引导扇区



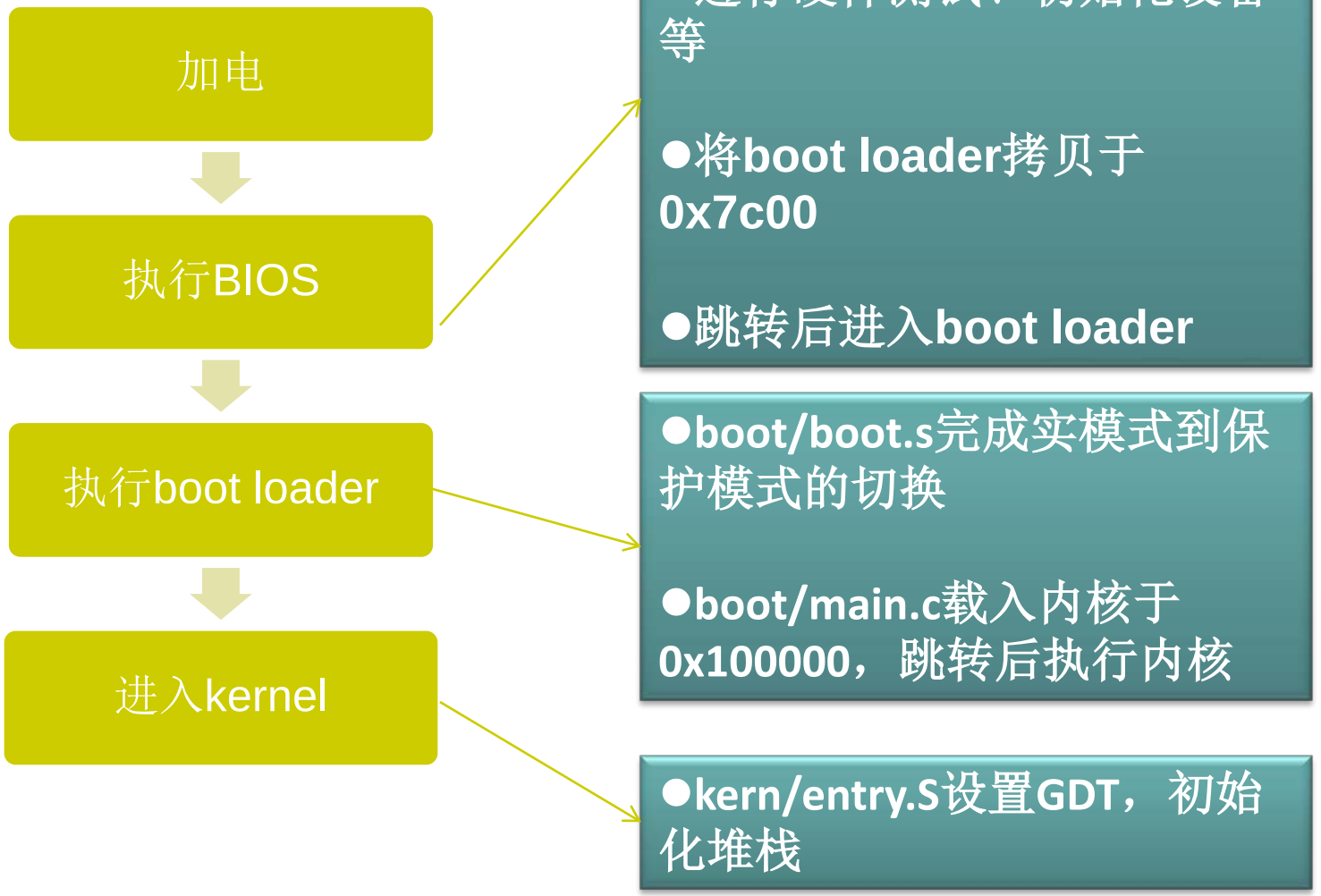
- 磁盘的第一个扇区是引导扇区
  - **JOS的引导扇区对应boot/代码，boot.S负责从实模式转化为保护模式，main.c负责加载内核**
- 引导扇区的大小为**512**字节
  - **JOS里编译后，bootloader小于510字节！**
  - **Boot/sign.pl负责补全bootloader**
  - 最后两个字节要求为**55AA**

光盘启动

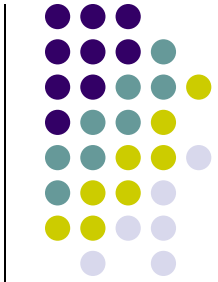


- 光盘启动
  - 扇区大小为**2048**字节
- 具体的规范
  - 文档**Bootable CD-ROM Format Specification**
  - 博客 http://blog.csdn.net/libeili/archive/2009/08/07/4422862.aspx

PC启动流程图

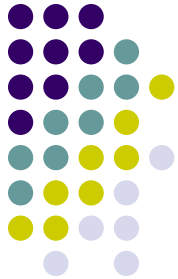

加电
执行BIOS
执行boot loader
进入kernel
●BIOS从0xffff0开始执行
●进行硬件测试、初始化设备等
●将boot loader拷贝于0x7c00
●跳转后进入boot loader
●boot/boot.s完成实模式到保护模式的切换
●boot/main.c载入内核于0x100000，跳转后执行内核
●kern/entry.S设置GDT，初始化堆栈

启动过程的实现



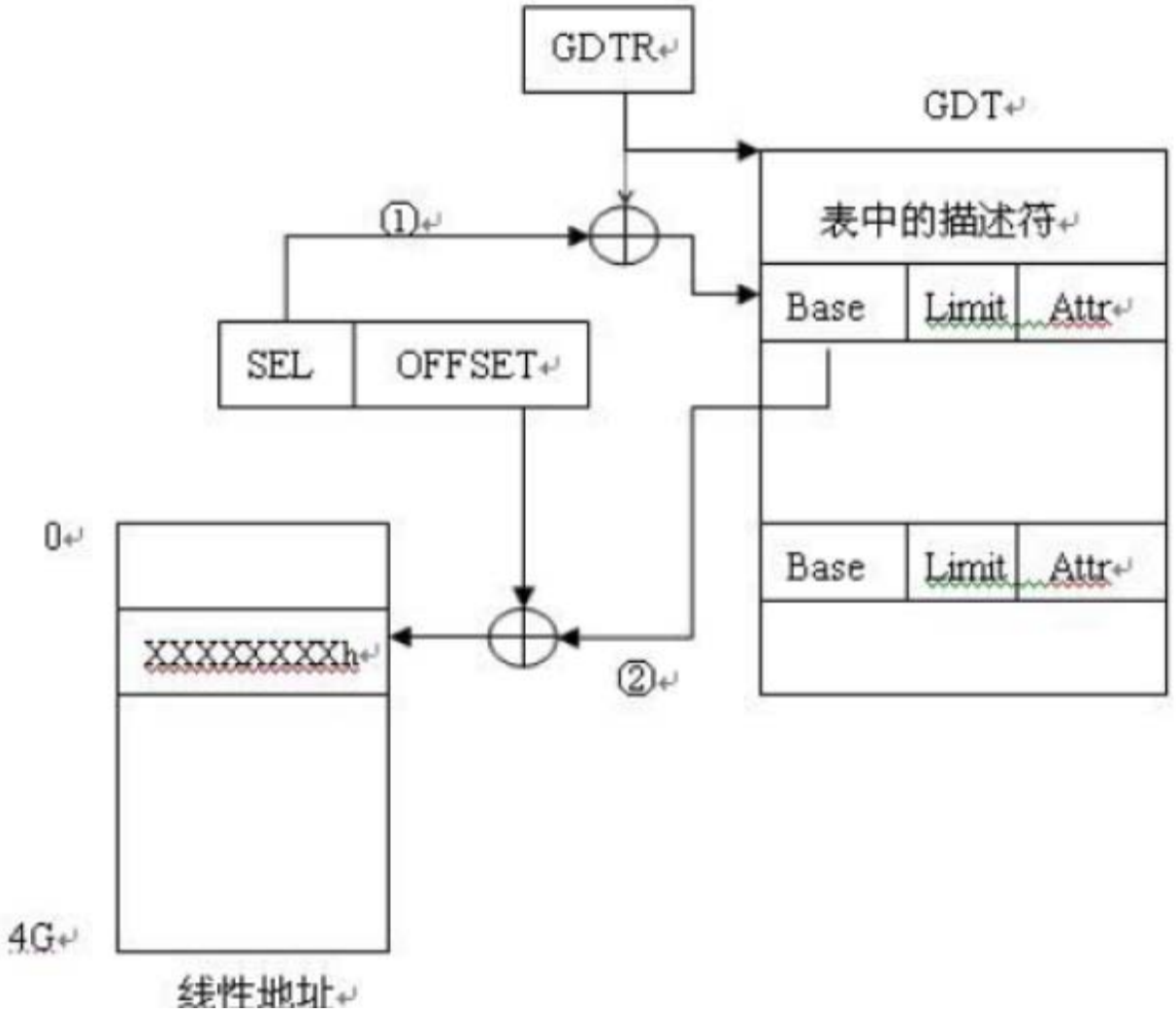
- **boot/**
  - **boot.S:初始化bootmain的C执行环境，从实模式转化到保护模式**
  - **main.c:装载内核，把CPU控制权交给内核代码**
  - **Makefrag: 引导文件的组织方式**
  - **sign.pl: 帮助修正引导文件（扩充为512字节等功能）**

实模式vs.保护模式

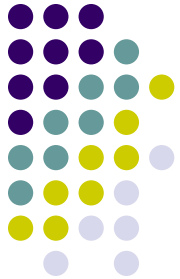


- 实模式
  - 16位寻址，1MB寻址空间
  - 实际物理地址20= （段寄存器16<< 4）+ 偏移地址16
  - 只能使用16位寄存器(ax, cx, dx, bx)
- 保护模式
  - 32位寻址空间，4GB寻址空间
  - 实际物理地址32= 段基址32+ 偏移地址32
  - 利用全局描述符表（GDT）保存段信息，寄存器GDTR保存GDT的物理地址，段寄存器保存段选择符
  - 可以使用32位的寄存器(eax, ecx, edx, ebx)
- 实模式转换到保护模式
  - 将控制寄存器cr0的PE(protection enable)位置1

保护模式下段式寻址

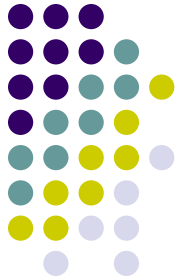

GDTR
GDT
表中的描述符
①
Base
Limit
Attr
SEL
OFFSET
0
Base
Limit
Attr
XXXXXXXXh
②
4G
线性地址

三种地址



- 虚拟地址
  - 进程角度看世界
- 线性地址
  - 虚拟地址在段式转换之后的地址
- 物理地址
  - 线性地址在页式转换之后的地址

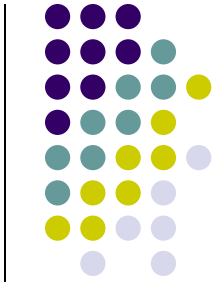
如何进入保护模式



- boot/boot.s

```
lgdt    gdtdesc      重新加载gdt表
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
ljmp    $PROT_MODE_CSEG, $protcseg
```

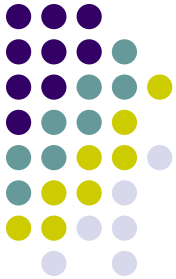
如何进入保护模式



- boot/boot.s

```
.p2align 2                                  4字节对齐

gdt:
  SEG_NULL                                          空段
  SEG(STA_X|STA_R, 0x0, 0xffffffff)     代码段
  SEG(STA_W, 0x0, 0xffffffff)               数据段

gdtdesc:
  .word   0x17                                gdt表的大小-1
  .long   gdt                               gdt表的内容
```

如何进入保护模式



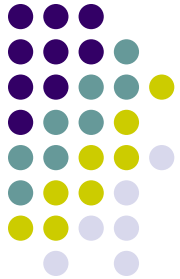
- GDT表项结构：

```
#define SEG_NULL                                                    \
    .word 0, 0;                                                     \
    .byte 0, 0, 0, 0
#define SEG(type,base,lim)                                          \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);              \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)),                 \
            (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

- GDT当前内容

段名	段基址	最大长度	权限
代码段	0x0	0xffffffff	STA_X|STA_R
数据段	0x0	0xffffffff	STA_W

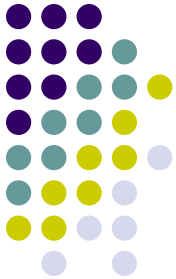
如何进入保护模式



- boot/boot.s

```
lgdt    gdtdesc      重新加载gdt表
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0          设置cr0寄存器，开启保护模式
ljmp    $PROT_MODE_CSEG, $protcseg
```

如何进入保护模式

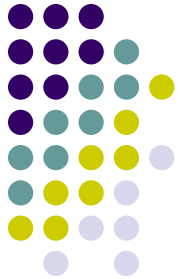


- boot/boot.s

```
lgdt    gdtdesc      重新加载gdt表
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0          设置cr0寄存器，开启保护模式
ljmp    $PROT_MODE_CSEG, $protcseg  进入32位模式

.code32                     # Assemble for 32-bit mode
protcseg:
```

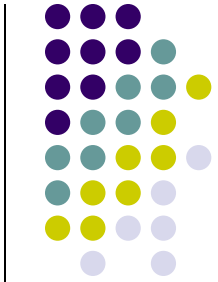
ljmp？

ljmp的秘密



- $PROT_MODE_CSEG 表示段选择子，被加载到cs中
- $protcseg 被加载到IP寄存器中
- ljmp的作用跳转到下一条语句
  - CS，IP寄存器会重新加载
  - 后面的代码都在32位保护模式下执行

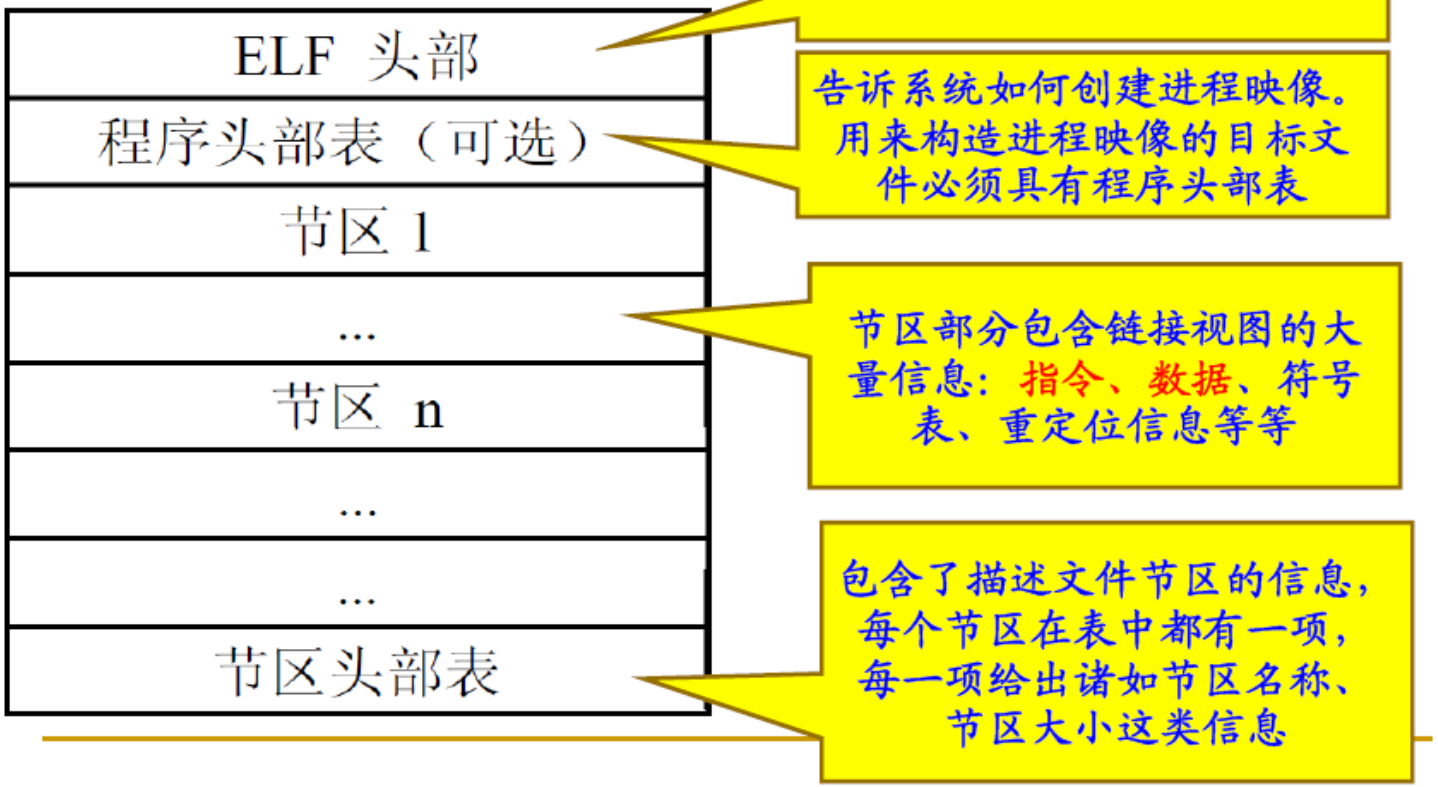
ELF文件格式



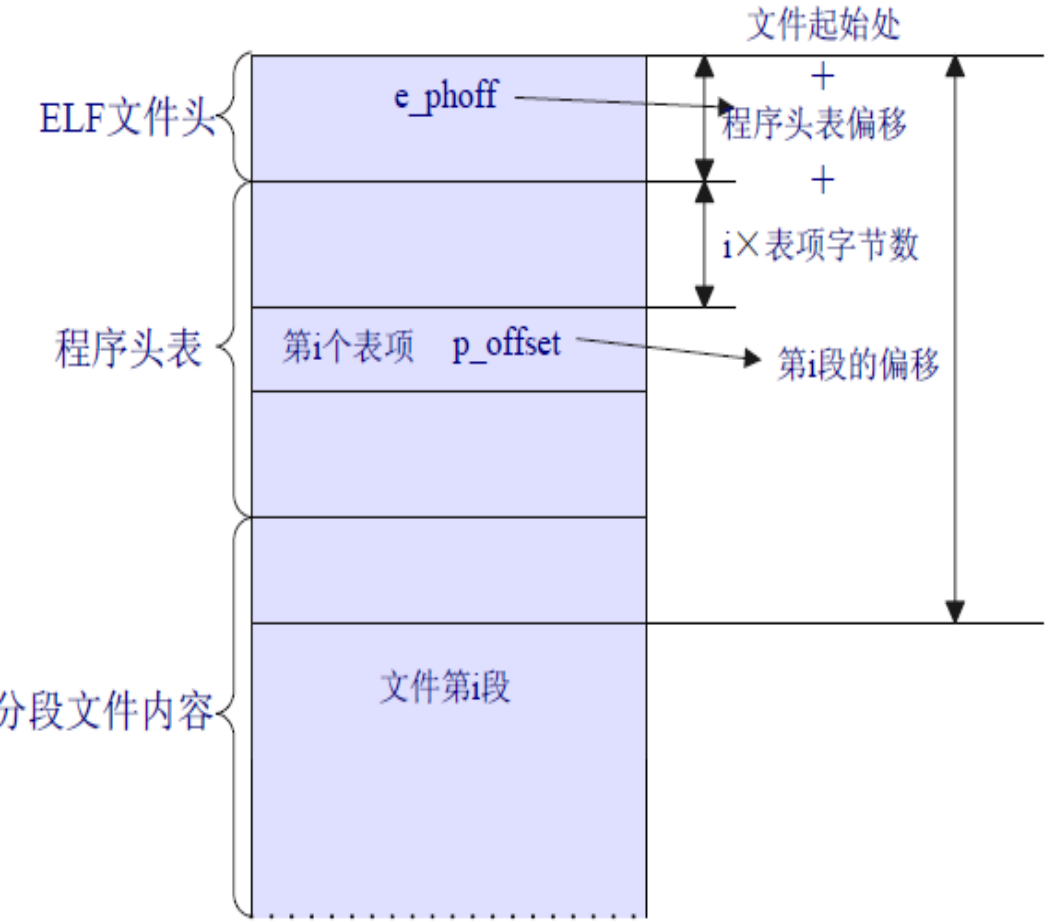
- ELF (Executable and Linking Format)
  - Boot loader需要将磁盘上的内核映像加载到内存中，内核映象是ELF格式文件
  - 类似于“字典”，由头部(header)和很多节区组成。

- 详细的格式定义参见：
  - the ELF specification：http://pdos.csail.mit.edu/6.828/2005/readings/elf.pdf

ELF文件结构


ELF 头部
程序头部表（可选）
节区 1
...
节区 n
...
...
节区头部表
描述整个文件的组织
告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表
节区部分包含链接视图的大量信息：指令、数据、符号表、重定位信息等等
包含了描述文件节区的信息，每个节区在表中都有一项，每一项给出诸如节区名称、节区大小这类信息

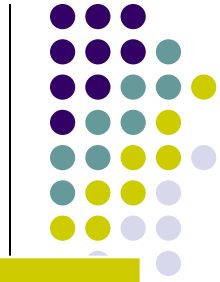
如何在ELF中找到某一段



重要的数据结构

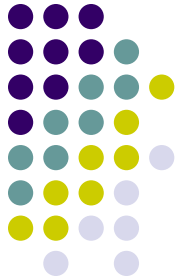
- ELF文件头：
  - struct Elf{}
- 程序头表相对于文件开头的偏移：
  - Elf->e_phoff
- 程序头表中段个数:
  - Elf->e_phnum
- 程序头表中的段:
  - struct Proghdr
- 段相对于文件开头的偏移:
  - Proghdr->p_offset

ELF文件头详细内容



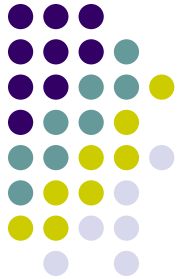
成员	含义
e_ident	Magic： 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 Class： ELF32 Data： 2's complement, little endian Version： 1 (current) OS/ABI： UNIX - System V ABI Version： 0
e_type	Type：REL (Relocatable file) ELF文件类型
e_machine	Machine：Intel 80386 ELF文件的CPU平台属性。相关常量以EM_开头
e_version	Version：0x1 ELF版本号。一般为常数 1
e_entry	Entry point address：0x0 入口地址，规定ELF程序的入口虚拟地址，操作系统在加载完该程序后从这个地址开始执行进程的指令。可重定位文件一般没有入口地址，则这个值为0
e_phoff	Start of program header
e_shoff	Start of section headers 段表在文件中的偏移

ELF文件头详细内容



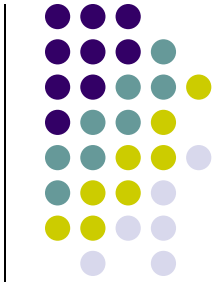
e_word	Flags：0x0 ELF标志位，用来标识一些ELF文件平台相关的属性。相关常量的格式一般为EF_machine_flag，machine为平台，flag为标志
e_ehsize	Size of this header 即ELF文件头本身的大小
e_phentsize	Size of program headers：0 (bytes)
e_phnum	Number of program headers：0
e_shentsize	Size of section headers 段表描述符的大小，这个一般等于sizeof(Elf32_Shdr)。
e_shnum	Number of section headers 段表描述符数量。这个值等于ELF文件中拥有的段的数量
e_shstrndx	Section header string table index 段表字符串表所在的段在段表中的下标。

链接地址vs.加载地址(1)



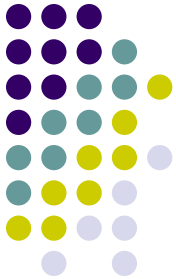
- 链接地址
  - 是虚拟地址
  - 代码中的绝对跳转地址和全局变量的地址都依赖于链接地址。当链接地址改变时，这些地址也会改变
  - 但是相对跳转不依赖于链接地址
- 加载地址
  - 程序被加载到的物理地址
- 关系
  - 链接地址需要和加载地址“保持一致”
    - 链接地址经过地址转换要等于物理地址

链接地址vs.加载地址(2)



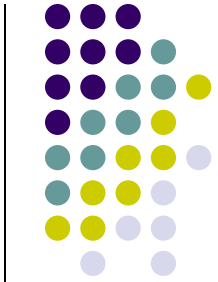
- 内核的加载地址
  - **0x100000处，参见boot/main.c中的代码**
- 内核的链接地址
  - **0xf0100000处，但是我们没有那么大的内存**
- 如何解决
  - **ELFHDR->e_entry & 0xFFFFFF**

链接地址vs.加载地址(3)



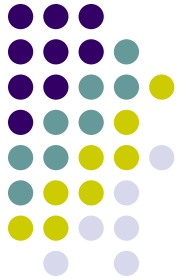
- Link和Load地址不一致会导致：
  - 直接跳转位置错误
  - 直接内存访问(只读数据区或bss等直接地址访问)错误
  - 堆和栈等的使用不受影响，但是可能会覆盖程序、数据区域

装载内核



- 内核加载
  - 内核被加载到**0x0010000(物理地址)**开始的内存中
  - 根据**Elf**文件格式，把内核镜像中的各个段都加载到指定的虚拟地址上
- 实现
  - 加载内核的代码 **boot/main.c**
  - **Elf**文件格式的定义 **inc/elf.h**

内核装入过程



#define SECTSIZE 512

一个扇区的大小

#define ELFHDR ((struct Elf *) 0x10000)

定义**ELF**文件头的位置，在内存的**0x10000**处。

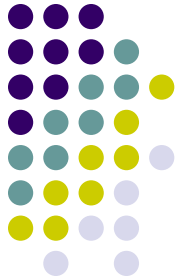
void readsect(void*, uint32_t);

读取磁盘上的一个扇区，扇区的偏移为参数

void readseg(uint32_t, uint32_t, uint32_t);

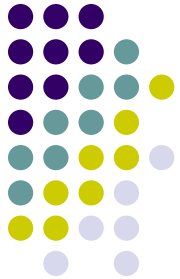
读取**ELF**文件中的一个段，第一个参数表示链接地址，转换后为加载地址，第二个参数为段的字节数，第三个参数为该段相对于文件头的偏移

内核装入过程



```
void
bootmain(void)
{
    ……
    readseg((uint32_t) ELFHDR, SECTSIZE*8, 0); 读取第一个页

    if (ELFHDR->e_magic != ELF_MAGIC)      判断魔数
        goto bad;

    //ph表示ELF段表首地址
    ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);

    eph = ph + ELFHDR->e_phnum; //eph表示ELF段表的末地址

    for (; ph < eph; ph++)    //循环读取每个段
        readseg(ph->p_va, ph->p_memsz, ph->p_offset);
```

内核装入过程



跳转到ELF程序入口执行，不会返回

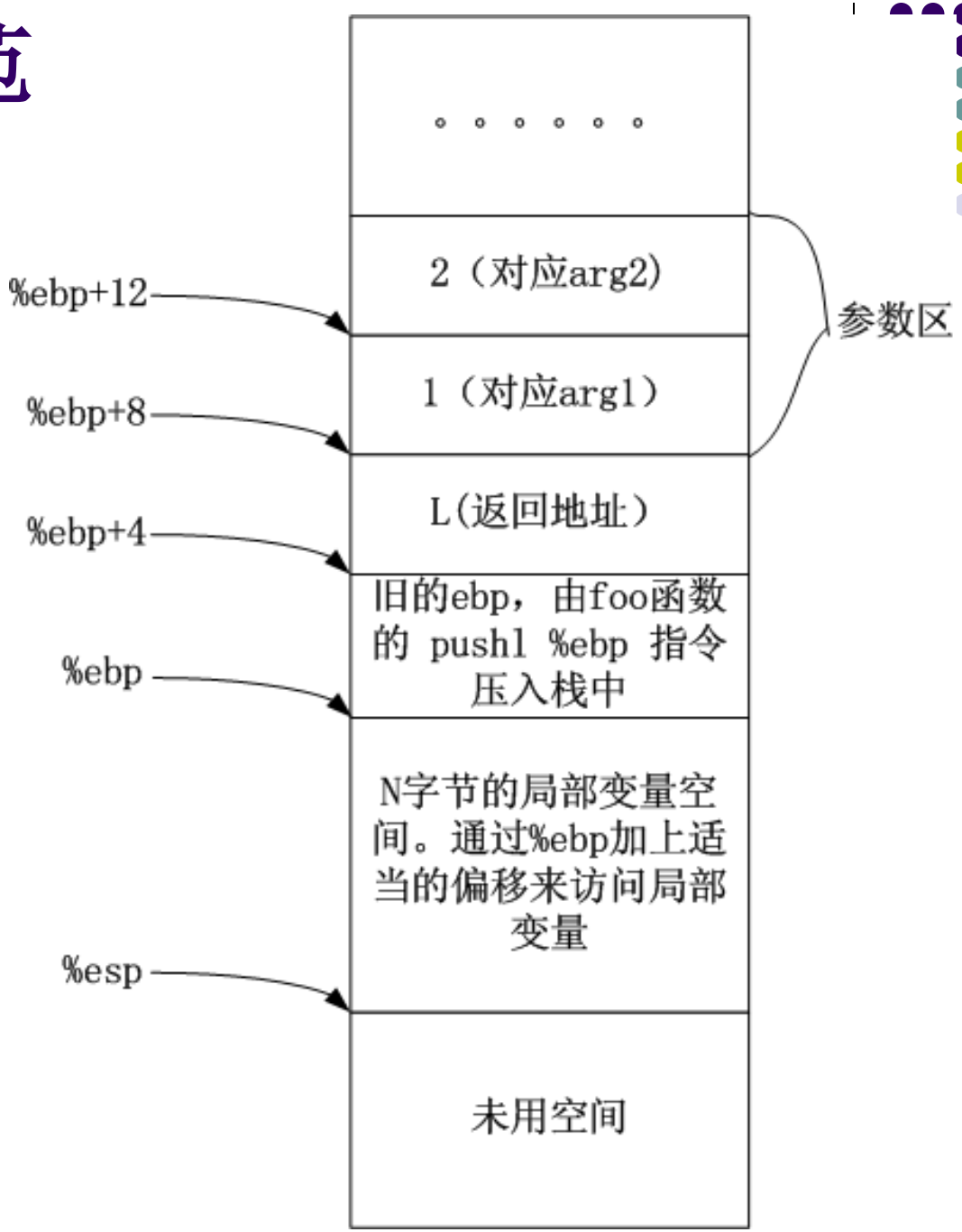
```
((void (*)(void)) (ELFHDR->e_entry & 0xFFFFFF))();
```

```
//错误处理
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
    while (1)
        /* do nothing */;
}
```

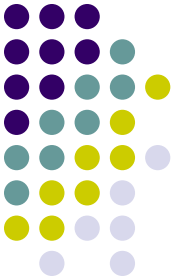
GCC调用规范

- 基于堆栈的子程序结构

foo(arg1,arg2)

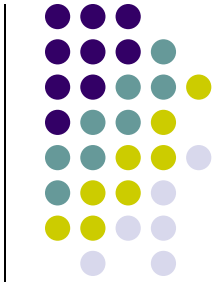


GCC调用规范(2)



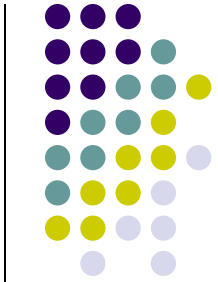
- 堆栈对齐
  - IA-32中堆栈是4字节对齐，IA-64中可以是8字节对齐
  - 压入堆栈的数据必须对齐，byte数据在压入堆栈之前必须扩展为四字节
- 堆栈生长方向
  - IA-32中堆栈向下生长，压入数据堆栈指针减少，弹出数据堆栈指针增加

GCC调用规范(3)



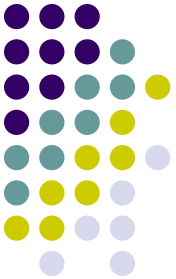
- 堆栈指针
  - IA-32中堆栈指针指向栈顶第一个可用地址
- C语言函数结构
  - 活动记录
    - 函数参数、返回地址、局部变量
  - 变量上下文环境
    - C语言两层函数结构，不是全局变量，便是局部变量
    - 全局变量在数据段，局部变量位于堆栈

GCC调用规范(4)



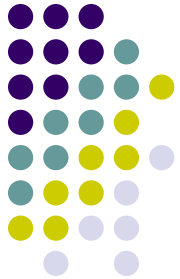
- 参数传递方式
  - 参数传递顺序
    - C从右向左压入参数
  - 清除堆栈上传递的参数
    - C由caller清除参数
  - C方式的优点
    - 可变参数列表容易实现
    - 汇编语言编写C函数
      - 简化了汇编语言子程序的复杂性，考虑需要使用可变参数的汇编子程序

GCC调用规范(5)



- 返回值传递方式
  - 简单类型，指针类型，**%eax**寄存器或者**%eax**和**%edx**
  - 结构体类型，函数原型中返回类型作为形式参数列表的一部分；**struct my f(int k)**将转化为**void f(struct my *p, int k);** 实际调用时将返回对象的赋值目标地址作为参数

调用过程分析——调用者



(1) 参数倒序压栈

```
pushl $2
pushl $1
```

(2) 调用子函数

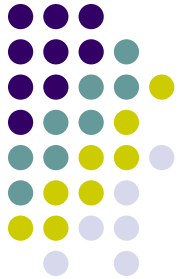
```
call foo
```

(3) 子函数返回，清理堆栈

```
addl $8, %esp
```

; 把%esp加8：把为foo准备的两个参数从栈中清除

调用过程分析——被调用者



(1) 保存调用者ebp，设置ebp新值

```
foo:
    pushl %ebp
    movl %esp, %ebp
```

(2) 准备局部变量空间

```
    subl $N, %esp;
```
准备N字节的局部变量空间

(3) 子函数执行主体

函数返回前要将返回值传入eax

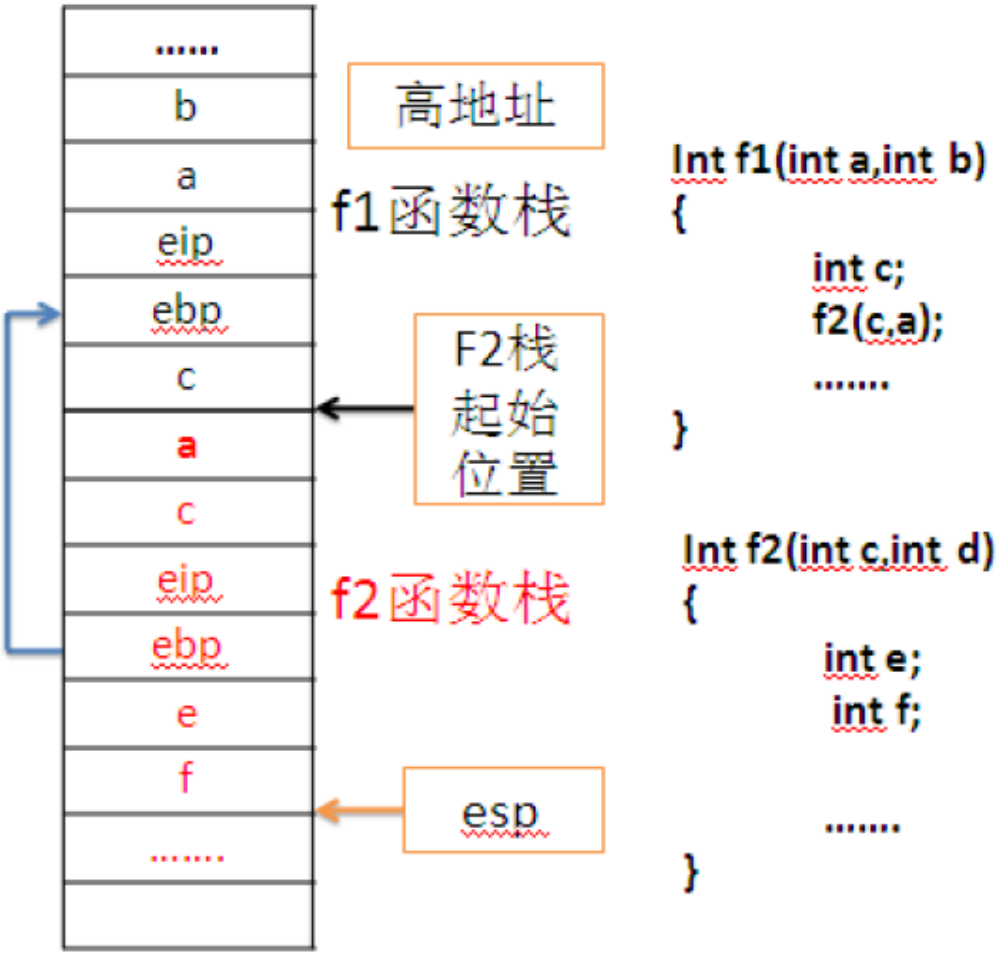
(4) 清除子函数堆栈并恢复ebp

```
    movl %ebp, %esp
    popl %ebp
```

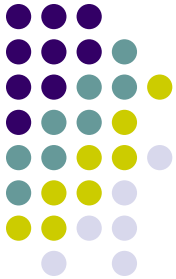
(5) 子函数返回

```
    ret
```

函数调用时的堆栈情况

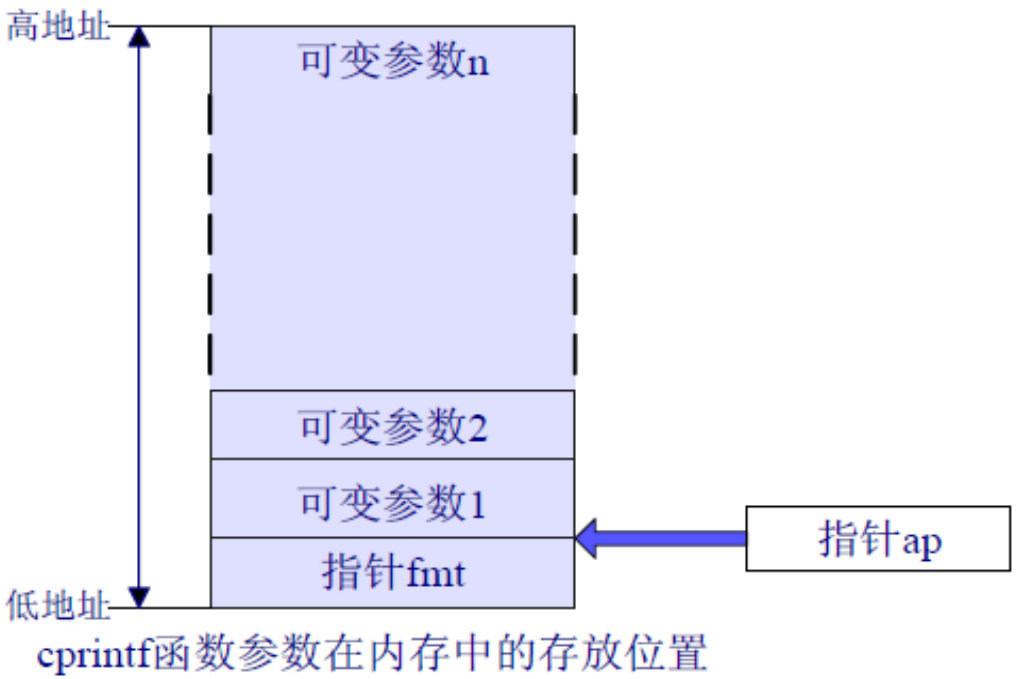

.....
b
a
eip
ebp
c
a
c
eip
ebp
e
f
.....
高地址
f1函数栈
F2栈
起始
位置
f2函数栈
esp
Int f1(int a,int b)
{
int c;
f2(c,a);
.....
}
Int f2(int c,int d)
{
int e;
int f;
.....
}

显示输出



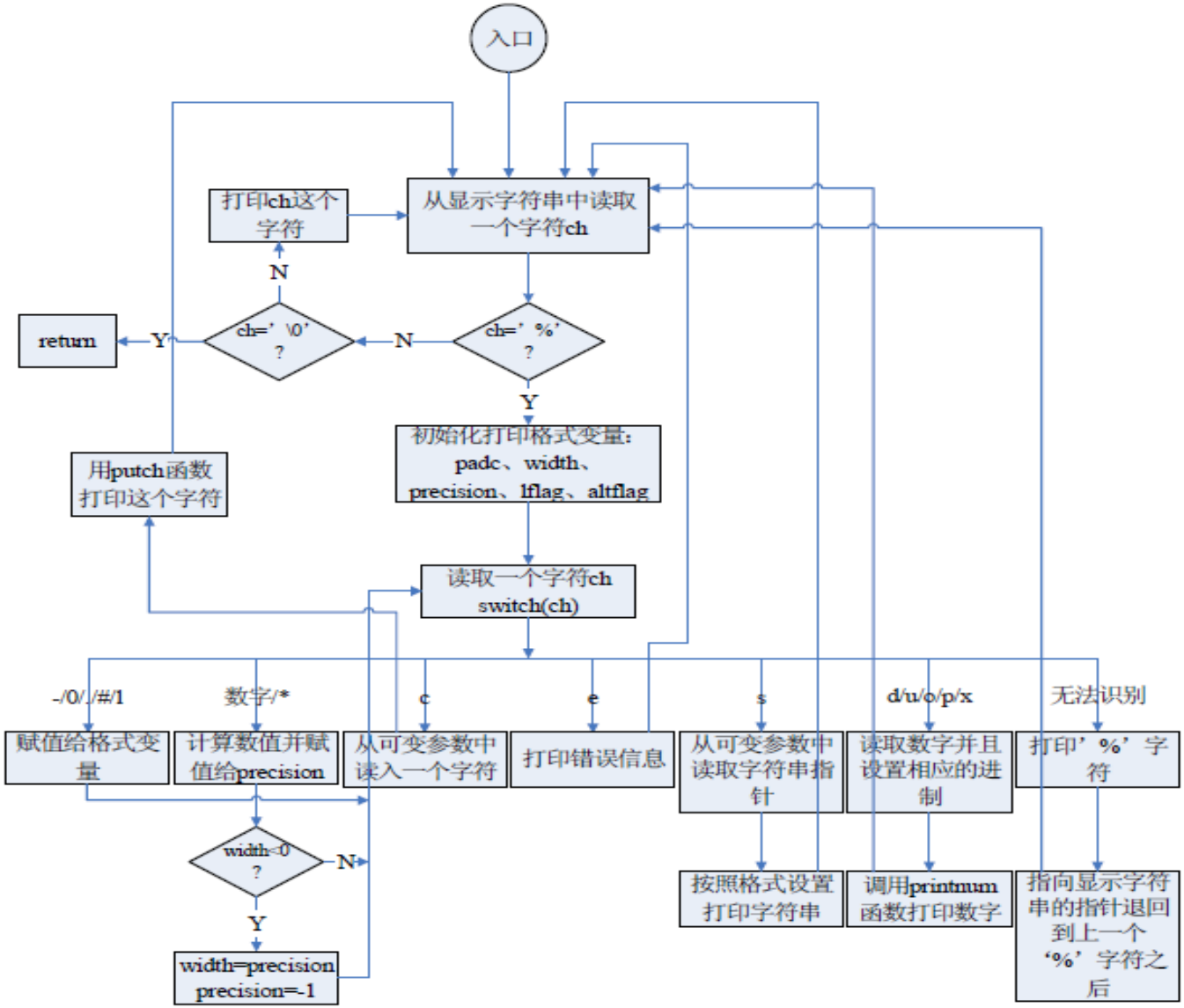
- JOS中实现了自己的printf版本：cprintf
  - 用户态函数声明inc/stdio.h，实现lib/*，
  - 内核态的函数使用的输出函数 kern/printf.c
- cprintf函数声明
  - void cprintf(const char *fmt, ...)
  - 与printf用法相同，支持可变参数

cprintf如何实现可变参数



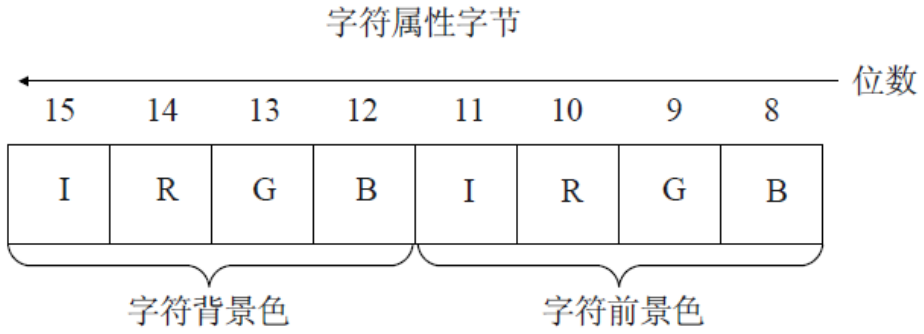
cprintf函数参数在内存中的存放位置

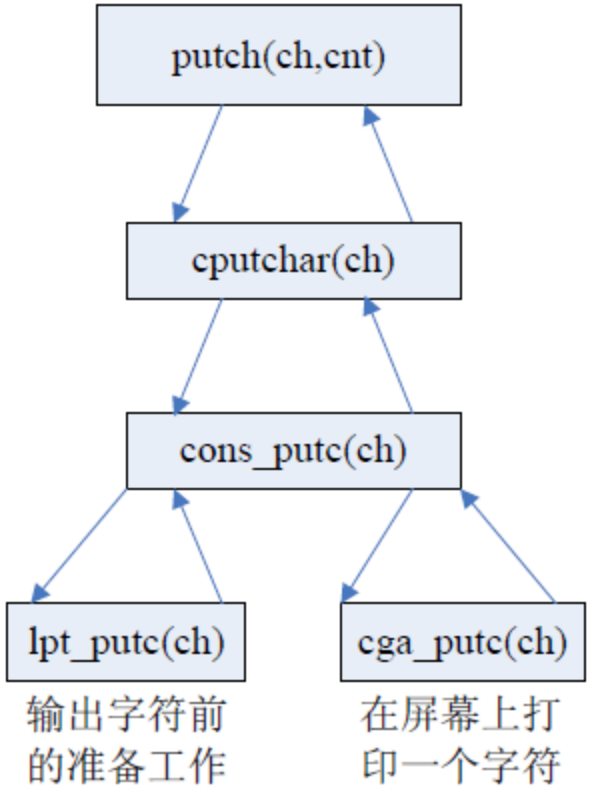
vprintfmt实现可变参数



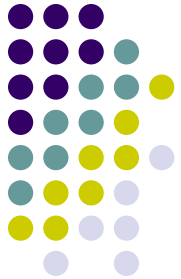
putch负责显示字符

- putch负责显示单个字符
- cga_putc负责设置显示的格式，在屏幕输出
  - ch的低6位表示字符编码
  - ch的高8位表示字符属性
- 字符属性



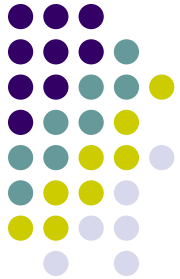


JOS堆栈结构



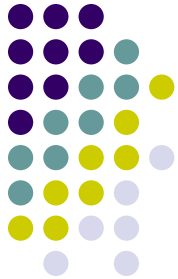
- JOS堆栈
  - 从高地址向低地址增长
  - 栈顶 esp
  - 进栈 esp-4，出栈 esp+4

进入内核时的堆栈情况



- 代码在**kern/entry.s**中
- 设置**ebp**寄存器，基址指针为**0x0**
  - **movl $0x0,%ebp**
- 设置栈顶
  - **movl $(bootstacktop),%esp**

进入内核时的堆栈情况



- 在 **kern/entry.S**中定义

.p2align ***PGSHIFT*** 对齐**4KB**

.globl ***bootstack***

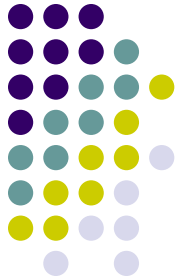
bootstack: 栈顶

.space ***KSTKSIZE*** 栈大小为**32KB**

.globl ***bootstacktop***

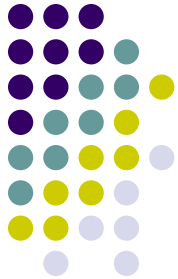
bootstacktop: 栈底

内联汇编



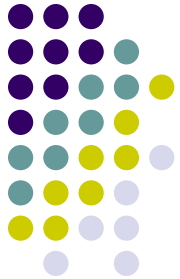
- **GCC内联汇编**
  - 基本格式
    - **asm(“statements”)**
    - **“asm”也可以由“__asm__”来代替**
    - **在“asm”后面有时也会加上“__volatile__”表示编译器不要优化代码，后面的指令保留原样**
  - **举例：__asm__ __volatile__("hlt")**

内联汇编



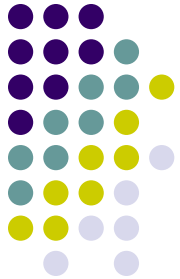
- 多行的格式
  - 如果有很多行汇编，则每一行后要加上“**\n\t**”
  - **gcc** 在处理汇编时，是要把**asm(…)**的内容“打印”到汇编文件中，所以格式控制字符是必要的
- 举例
  - **__asm__("movl $1, %eax\n\t**
  - **"movl $4, %ebx\n\t"**
  - **"int $ 0x80");**

内联汇编



- 扩展的内联汇编格式
  - __asm__(
    汇编语句模板:
    输出部分:
    输入部分:
    破坏描述部分);
  - 共四个部分：汇编语句模板，输出部分，输入部分，破坏描述部分，各部分使用“:”格开，汇编语句模板必不可少，其他三部分可选，如果使用了后面的部分，而前面部分为空，也需要用“:”格开，相应部分内容为空

内联汇编

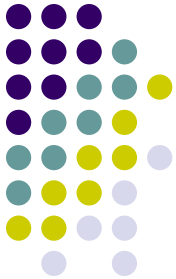


- 举例

```
#include <stdio.h>
int main() {
    int a = 10, b;
    __asm__("movl %1, %%eax\n\t"
            "movl %%eax, %0\n\t"
            :"=r"(b) /* output */
            :"r"(a) /* input */
            :"%eax" /* clobbered register */ );
    printf("Result: %d, %d\n", a, b);
    return 0;
}
```

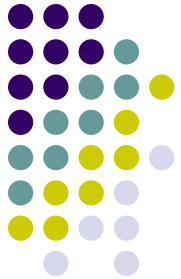
- 这个程序将变量a的值赋给b

内联汇编



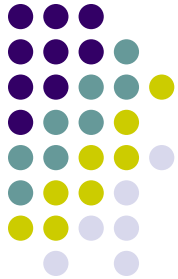
- **"r"(a)**指示编译器分配一个寄存器保存变量**a**的值，作为汇编指令的输入
- **“=r”(b)**指示编译器分配一个寄存器保存变量**b**的值，作为汇编指令的输出
- 指令中的**%1**（按照约束条件的顺序，**b**对应**%0**，**a**对应**1%**），至于**%1**究竟代表哪个寄存器则由编译器自己决定。汇编指令首先把**%1**所代表的寄存器的值传给**eax**（为了和**%1**这种占位符区分，**eax**前面要求加两个**%**号），然后把**eax**的值再传给**%0**所代表的寄存器
- 在执行这两条指令的过程中，寄存器**eax**的值被改变了，所以把**"%eax"**写在第四部分，告诉编译器在执行这条**__asm__**语句时**eax**要被改写，所以在此期间不要用**eax**保存其它值

内联汇编



- 参阅内联汇编手册
  - Brennan's的内联汇编指南（Inline Assembly with DJGPP）
  - http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

Part 1 PC Bootstraps



- Exercise1：X86汇编基础

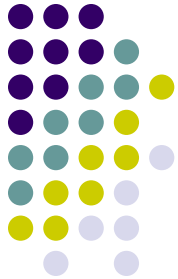
  阅读《PC Assembly Language》这本书（以下部分可以忽略：第一章1.3.5之后部分，第5、6章，7.2）

  阅读Brennan’s Guide中的Syntax部分

Bochs模拟器—使用

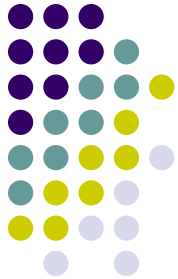
- Exercise 2：熟悉bochs基本命令的使用
  - 从0xf000:0xfff0开始单步跟踪BIOS的执行
  - 在初始化位置0x7c00设置实地址断点，测试断点正常
  - 从0x7c00开始跟踪代码运行，将单步跟踪反汇编得到的代码与boot/boot.S和obj/boot/boot.asm进行比较
  - 在obj目录下自己找一个内核中的代码位置，设置断点并进行测试

BIOS启动过程



- Exercise 3：查看BIOS中前5~6条命令的内容，参考Phil Storrs I/O Ports Description大致了解这些命令的作用

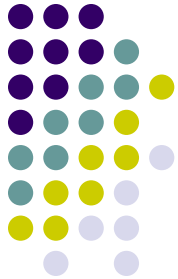
Part 2 Boot Loader



Exercise 4：在boot扇区被加载到的0x7c00地址处设置断点，在bochs中跟踪代码的执行，并将其与反汇编后的文件对照

跟踪boot/main.c中的read_sector()，将c文件中的语句与汇编语句对应起来；找出汇编文件中与cmain函数中读取剩余扇区循环对应的第一条语句以及最后一条语句，并跟踪剩下的语句

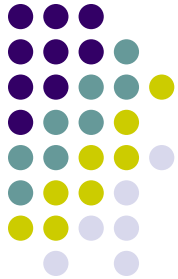
Boot Loader



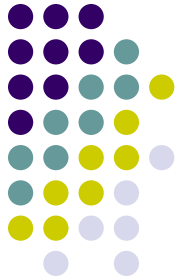
Exercise 5：阅读Lions注释的第三章“Reading C Programs”，特别注意其中关于指针使用的例子。（为了避免以后不必要的麻烦，不要跳过这一个练习）

Exercise 6：在BIOS进入boot loader以及boot loader进入内核的两个地方设置断点；查看当时0x00100000地址开始的8个word的内容；为什么不一样？第二个断点处的内容是什么？

Boot Loader



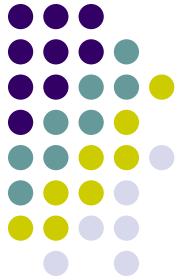
Exercise 7：再次跟踪boot loader，预测由于链接地址设置错误而产生问题的第一条指令；在boot/Makefrag中修改该链接地址来验证你的想法；最后不要忘了改回正确的链接地址



Part 3 The Kernel

- Exercise 8：内核布局
  - 使用 Bochs来跟踪JOS kernel，找到新的段式地址映射起作用的地方；使用bochs查看 GDT表的值，猜测在虚实地址转换发生错误的时候，第一条会出错的指令地址；修改GDT表的相关值来验证你的想法；最后恢复正确的值

终端字符打印

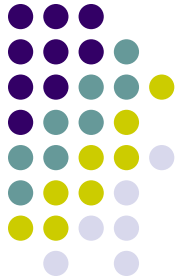


- Exercise 9：补充刻意漏掉的代码，用"%o"打印出8进制数字.
- 并确保能回答出实习要求文档中的6个问题

堆栈

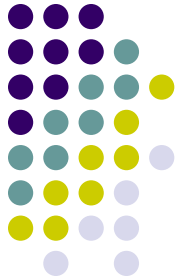
- Exercise 10：了解内核在哪里初始化它的堆栈，并知道堆栈被定位到内存的什么地方；以及内核如何为它的堆栈预留空间；以及在这片预留的空间中，哪一端是堆栈初始化后的栈顶(由栈顶指针来指向)?

堆栈



- Exercise 11：熟悉GCC的调用规范，在obj/kern/kernel.asm中找到test_backtrace函数的地址，用 Bochs在那里设置断点；然后可以了解在内核启动后每次该函数被调用所发生的情况
- Exercise 12: 按照文档描述的格式要求实现backtrace 函数，输出采用文档所述的标准形式

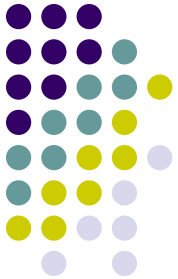
参考文献



- ELF文件格式定义参见：
  - the ELF specification：http://pdos.csail.mit.edu/6.828/2005/readings/elf.pdf
- 内联汇编格式
  - Brennan's的内联汇编指南

    http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

参考文献



- 光盘启动
  - 文档Bootable CD-ROM Format Specification
  - 博客 http://blog.csdn.net/libeili/archive/2009/08/07/4422862.aspx