第三章. 系统的启动和初始化(v0.1)

3.1 系统的启动过程

本节将通过 lab1 具体介绍一下 JOS 的启动过程，我们将讲述 BIOS 对系统的初始化、Boot Loader 程序的功能以及内核可执行文件装入内存的过程。通过本节的讲述读者将会了解到 PC 启动的一般原理。

1. 物理内存的分布

我们首先来分析一下 PC 开机以后的默认的物理内存的分配。PC 的物理内存空间会由硬件规定产生如下图所示的布局：

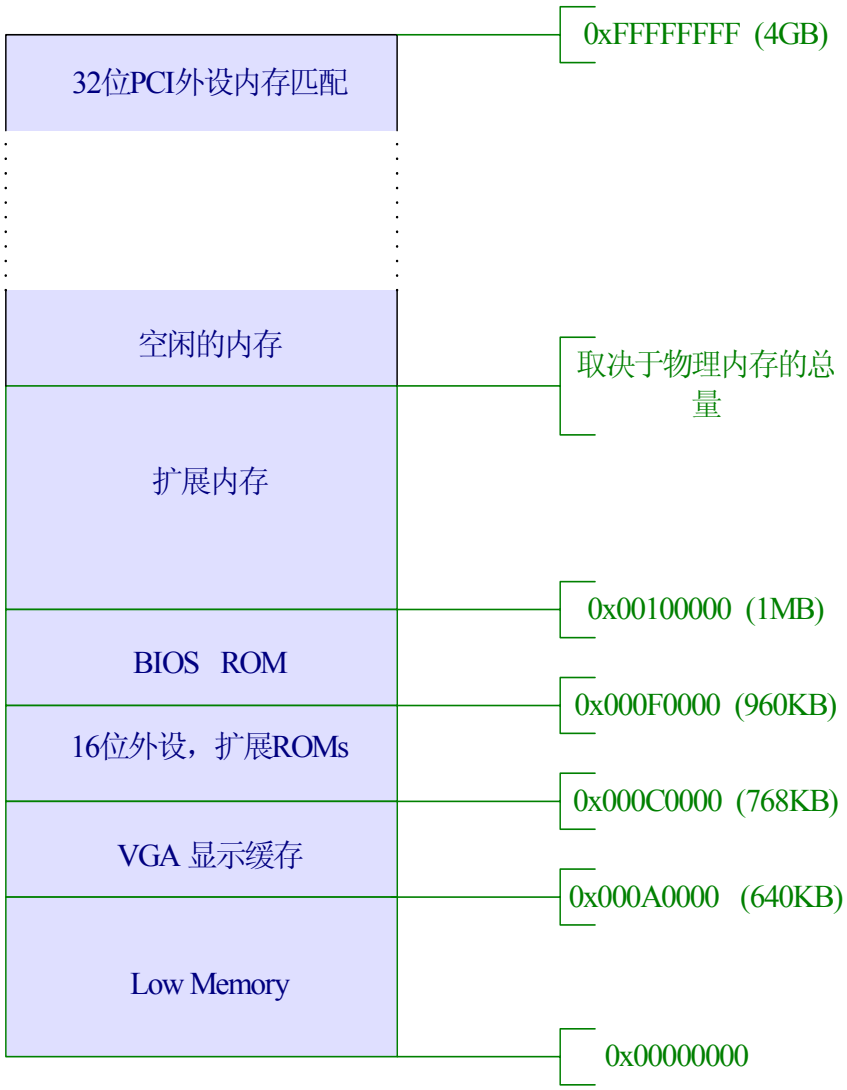


图 3-1 PC 默认物理内存布局

早期的 PC 是基于 16 位的 Intel 8088 的处理器的，因此只支持 1MB 的物理内存。早期的 PC 的物理内存是从 0x00000000 到 0x000FFFFF，而不是结束于 0xFFFFFFFF。如图 3-1 所示，物理内存的前 640KB 被标记为了“Low Memory”，这一块内存区域是早期 PC 唯一

可以使用的 RAM。事实上，非常早期的 PC 仅仅只能使用 16KB、32KB 或者 64KB 的 RAM。

从 0x000A0000 到 0x000FFFFF 的 384kB 的区域是被硬件保留着用于特殊通途的，比如像作为 VGA 的显示输出的缓存或者是被当作保存系统固化指令的非易失性存储器。这一部分内存区域中最重要的应该是保存在 0x000F0000 到 0x00100000 处占据 64KB 的基本输入输出系统(BIOS)。在早期的 PC 中，BIOS 是被存储在真正的只读存储器(ROM)中，但然而如今的 PC 将 BIOS 存储在可以更新的闪存中。BIOS 的作用是对系统进行初始化，比如像激活显卡、检查内存的总量。在进行完这些初始化后，BIOS 便将操作系统从一个合适的位置装载到内存，这些位置可以是软盘、硬盘、CD-ROM 或者是网络，在这之后，BIOS 便会将控制权交给操作系统。

当出现 80286 和 80386 处理器后，Intel 处理器终于打破了仅能访问 1MB 内存空间的限制，这两种处理器分别支持寻址 16MB 和 4GB 的内存空间。尽管如此，PC 架构还是保留了之前的物理内存低 1MB 空间的布局方式，这样做是为了保证和之前存在的软件相兼容，因此最新的 PC 会保留物理内存从 0x000A0000 到 0x00100000 的区域，这样便将系统可以使用的 RAM 分成了两个部分，一部分是低 640KB 的“Low Memory”，另一部分便是 1MB 以上部分的“扩展内存”。另外，32 位物理地址空间的最高部分往往被 BIOS 保留供 32 位的 PCI 外设所使用。

如今的 x86 处理器能够支持多于 4GB 的物理内存，于是 RAM 的范围能够扩展到超过 0xFFFFFFFF。在这种情况下，BIOS 需要保留 32 位物理地址空间的最高部分，这是为了将这个区域留给 32 位外设去匹配内存。在这里，由于设计的局限，我们实验中的 JOS 操作系统仅仅会使用 PC 物理内存的前 256MB，所以我们只需考虑 PC 只支持 32 位物理地址空间。

2. ROM BIOS

在 PC 启动的时候，首先会在实模式下运行 BIOS，读者应该还记得之前 1.2 节 Bochs 简介中所讲到的 Bochs 需要用一个镜像文件 BIOS-bochs-latest 来模拟真实的 BIOS，而在启动的时候这镜像文件的内容便会被装载到如图 3-1 所示的物理内存中 0x000F0000 到 0x00100000 的位置处。当我们刚启动 Bochs 时，我们会看到如下的画面：

```
========================================================================
                       Bochs x86 Emulator 2.0
                          December 21, 2002
========================================================================
00000000000i[     ] reading configuration from .bochsrc
Next at t=0
(0) [0x000ffff0] f000:fff0 (unk. ctxt): jmp e05b                  ; e968e0
<bochs:1> s
Next at t=1
(0) [0x000fe05b] f000:e05b (unk. ctxt): mov AL, 0f                ; b00f
<bochs:2> s
Next at t=2
(0) [0x000fe05d] f000:e05d (unk. ctxt): out 70, AL                ; e670
<bochs:3> s
Next at t=3
(0) [0x000fe05f] f000:e05f (unk. ctxt): in AL, 71                 ; e471
<bochs:4> s
Next at t=4
(0) [0x000fe061] f000:e061 (unk. ctxt): cmp AL, 00                ; 3c00
<bochs:5>
```

图 3-2 启动时 BIOS 的运行

可以看到，启动后执行的第一条指令是在内存 0x000FFFF0 处的“jmp e05b”，我们知道 BIOS 在内存中的上限是 0x00100000，于是在 0x000FFFF0 处执行第一条指令的话必然要跳

转这样才会有更多的 BIOS 指令可以执行。

为什么 Bochs 要以这种方式来启动呢？就是因为在刚开始的时候内存中没有任何其它的程序可以执行，于是将 CS 设置为 0xF000，将 IP 设置为 0xFFF0，物理地址为 0x000FFFF0，这样就保证了 BIOS 会在刚启动的时候得到控制权。在 BIOS 得到控制权后便会对系统进行一系列的初始化。当我们让 BIOS 继续执行便会看到如下的 Bochs 输出窗口的画面。

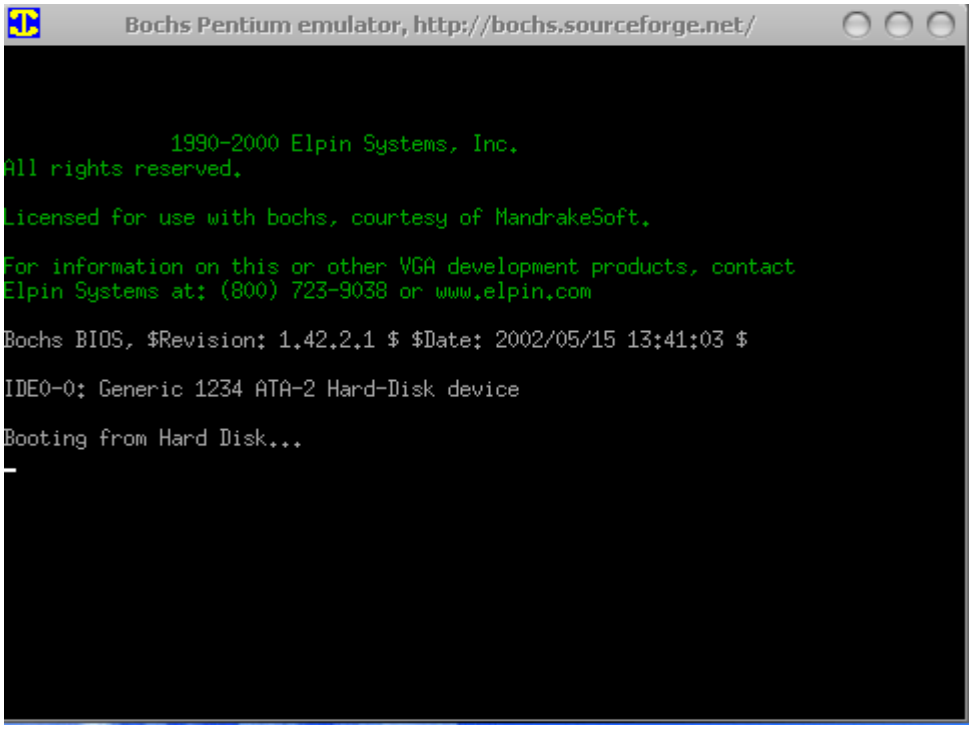


图 3-3 Bochs 执行输出

上图便是 BIOS 执行过程中的输出，可以看到“Booting from Hard Disk”，说明 BIOS 判断系统应该从硬盘启动，而这个时候 BIOS 会将一个称作 Boot Loader 的程序从硬盘读到内存的中并把控制权交给改程序。到这里 BIOS 的任务就算是完成了。

3. Boot Loader

我们已经知道 BIOS 在完成它的一系列初始化后便把控制权交给 Boot Loader 程序，在我们的 JOS 实验中，我们的 Boot Loader 程序会在编译成可执行代码后被放在模拟硬盘的第一个扇区。

```
-bash-2.05b$ gmake
+ as kern/entry.S
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 406 bytes (max 510)
+ mk obj/kern/bochs.img
-bash-2.05b$
```

图 3-4 Boot Loader 程序的编译链接

硬盘由于传统的原因被默认分割成了一个个大小为 512 字节的扇区，而扇区则是硬盘最小的读写单位，即每次对硬盘的读写操作只能够对一个或者多个扇区进行并且操作地址必须是 512 字节对齐的。如果说操作系统是从磁盘启动的话，则磁盘的第一个扇区就被称作“启动扇区”，因为 Boot Loader 的可执行程序就存放在这个扇区。在 JOS 实验中，当 BIOS 找到启动的磁盘后，便将 512 字节的启动扇区的内容装载到物理内存的 0x7c00 到 0x7dff 的位置，紧接着再执行一个跳转指令将 CS 设置为 0x0000，IP 设置为 0x7c00，这样便将控制权交给了 Boot Loader 程序。

在图 1-4 中可以看到 lab1 中的程序最终编译链接成了两个可执行文件 Boot 和 Kernel，其中 Kernel 是即将被 Boot Loader 程序装入内存的内核程序，而 Boot 便是 Boot Loader 本身的可执行程序，“boot block is 406 bytes (max 510)”这句话表示存放在第一个扇区的 Boot Loader 可执行程序的大小不能超过 510 个字节，由于磁盘的一个扇区的大小为 512 字节，这样便保证了 Boot Loader 仅仅只占据磁盘的第一个扇区。

另外我们要说的是在 PC 发展到很后来的时候才能够从 CD-ROM 启动，而 PC 架构师也重新考虑了 PC 的启动过程。然而从 CD-ROM 启动的过程略微有点复杂。CD-ROM 的一个扇区的大小不是 512 字节而是 2048 字节，并且 BIOS 也能够从 CD-ROM 装载更大的 Boot Loader 程序到内存。在本实验中由于规定是从硬盘启动，所以我们暂且不考虑从 CD-ROM 启动的问题。

下面我们就详细的讲述一下 Boot Loader 程序。在本实验中，Boot Loader 的源程序是由一个叫做的 boot.S 的 AT&T 汇编程序与一个叫做 main.c 的 C 程序组成的。这两部分分别完成两个不同的功能。其中 boot.S 主要是将处理器从实模式转换到 32 位的保护模式，这是因为只有在保护模式中我们才能访问到物理内存高于 1MB 的空间(保护模式我们之前在 1.1 节中有详细的讲解)。main.c 的主要作用是将内核的可执行代码从硬盘镜像中读入到内存中，具体的方式是运用 x86 专门的 I/O 指令，在这里我们只用了解它的原理，而对 I/O 指令本身我们不用做过多深入的了解。

下面我们首先来分段讲解一下 boot.S 源程序的具体意思，其中源程序中的英文注释在这里为了便于读者理解我们将其译为中文。

```
#include <inc/mmu.h>
.set PROT_MODE_CSEG, 0x8              # 代码段选择子
.set PROT_MODE_DSEG, 0x10             # 数据段选择子 r
.set CR0_PE_ON,          0x1          # 保护模式启动标识位

.globl start
start:
.code16                               # 16 位模式
cli                                   # 关中断
cld                                   # 关字符串操作自动增加

# 设置重要数据段寄存器(DS, ES, SS).
xorw    %ax,%ax                       # 将 ax 清零
movw    %ax,%ds                       # 初始化数据段寄存器
movw    %ax,%es                       # 初始化附加段寄存器
movw    %ax,%ss                       # 初始化堆栈段寄存器
```

首先 boot 程序会进行初始化，先把代码段选择子与数据段选择子以及保护模式启动标志设置为了常量，然后关中断并且将 ds、es、ss 这些段寄存器全部清零。

```
seta20.1:
    inb     $0x64,%al               # 等待空闲的时候
    testb   $0x2,%al
    jnz     seta20.1
    movb    $0xd1,%al               # 将 0xd1 输出到第 0x64 号 I/O 端口
    outb    %al,$0x64

seta20.2:
    inb     $0x64,%al               # 等待空闲的时候
    testb   $0x2,%al
    jnz     seta20.2
    movb    $0xdf,%al               # 将 0xdf 输出到第 0x60 号 I/O 端口
    outb    %al,$0x60
```

这段代码的作用是打开 A20 地址线。在默认的情况下，第 20 根地址线一直为 0，这样做的目的是为了向下兼容早期的 PC。由于早期的 PC 仅仅只在实模式下进行寻址，这样所能理论上可以寻到的最大地址应该是 0xFFFF0+0xFFFF，这看上去超过了 1MB 的地址空间，然而因为早期的 PC 只有 20 根地址线，于是相当于最高位的进位时被忽略了，地址最终还是在 1MB 以内。所以当 PC 有了 32 根地址线并且能够在保护模式下寻址 4G 的地址空间后，为了向下兼容，在默认情况下将第 20 根地址线一直置 0，这样就可以让仅在实模式下运行的程序不会出现最高位的进位，相当于还是只有 20 根地址线再起作用。在这里，我们仅需要了解这段程序的大概意思，不需要具体了解每一行的作用。

```
    lgdt    gdtdesc
    movl    %cr0, %eax
    orl     $CR0_PE_ON, %eax
    movl    %eax, %cr0
    ljmp    $PROT_MODE_CSEG, $protcseg    # 跳转到下一条指令同时切换到 32 位的模式
```

这段代码的作用是将系统从实模式切换到保护模式。首先用“lgdt gdtdesc”这条指令将 GDT 表的首地址加载到 GDTR，然后将 cr0 寄存器的最低位置 1，标志着系统进入保护模式，最后用一个跳转指令让系统开始使用 32 位的寻址模式。可以看到最后一句长跳转指令实际上是在系统进入保护模式后执行的，于是在这里$PROT_MODE_CSEG，代表的是段选择子，从后面的 GDT 表中可以看到基地址是 0x0，而偏移地址是$protcseg，$protcseg 实际上代表的是接下来指令的链接地址，也就是可执行程序在内存中的虚拟地址，只是刚好在这里编译生成的可执行程序 boot 的加载地址与链接地址是一致的，于是$protcseg 就相当于指令在内存中世纪存放位置的物理地址，所以这个长跳转可以成功的跳转到下一条指令的位置。关于链接地址与加载地址的问题我们在后面会做详细的讨论。

```
    .code32                         # 32 模式
protcseg:
    movw    $PROT_MODE_DSEG, %ax
    movw    %ax, %ds
    movw    %ax, %es
```

```
movw    %ax, %fs
movw    %ax, %gs
movw    %ax, %ss                    # 初始化段寄存器
movl    $start, %esp                # 初始化堆栈指针
call bootmain                       # 调用 main.c 中的 bootmain 函数
spin:
  jmp spin
# GDT 表
.p2align 2                                  # GDT 表 4 字节对齐
gdt:
  SEG_NULL                  # 空表项
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # 代码段表项
  SEG(STA_W, 0x0, 0xffffffff)           # 数据段表项
gdtdesc:
  .word   0x17                              # gdt 表长度 - 1
  .long   gdt                               # gdt 表物理地址
```

在进入保护模式后，程序在重新对段寄存器进行了初始化并且赋值了堆栈指针后便调用 bootmain 函数。可以看到，在“call bootmain”之后便是一个无限循环的跳转指令，之所以是无限循环就是这个函数调用永远都不会有返回的可能性，这句程序仅仅只是让整个代码看起来有完整性。

之后的代码则是定义了 GDT 表。首先我们可以看到 GDT 表的存放位置是 4 字节对齐的，也就是说 GDT 表的物理首地址是 4 的倍数。然后我们可以看到 gdt 标识了 3 个 GDT 表项，在这里 boot.S 程序使用了 SEG_NULL 与 SEG(type,base,lim)这两个宏，所谓的宏就是与函数类似的东西，只不过在编译的时候函数被编译成相应的可执行代码，而宏则会被编译成对应的常量。在这里，这两个宏都定义于 lab1 中的 inc/mmu.h 的文件中。

其中 SEG_NULL 的定义为：

```
#define SEG_NULL                        \
    .word 0, 0;                         \
    .byte 0, 0, 0, 0
```

它的作用就是定义连续 8 个值为 0 的字节，这就表示一个空的 GDT 表项

而 SEG(type,base,lim)的定义为：

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

在这里，type 表示段属性，base 表示段基址，而 lim 则表示段长的界限，给出这三个参数就可以用这个宏来定义一个 GDT 表项。在这里段属性的参数也是一般式通过宏的形式给出的，下表给出了常用的一些宏，这些宏每个都代表表项中的一个 bit 位，同时也代表一种段的属性。

表 3-1 常用段属性的宏

宏	值	属性
STA_X	0x8	可执行的段
STA_E	0x4	向下扩展(该属性仅限于非可执行段)
STA_C	0x4	一致性的代码段(仅限于可执行段)

STA_W	0x2	可写(仅限于非可执行段)
STA_R	0x2	可读(仅限于可执行段)
STA_A	0x1	可访问的

4. 链接地址和加载地址

在上一部分对 boot.S 源程序的分析当中，我们可以看到在程序中有很多诸如 protcseg 和 gdt 这样的地址标识符，而且程序中也有很多地方直接引用了这些标识符，然而它们到底代表的是一个什么样的地址的值呢，和在内存中存放的位置的物理地址又有什么区别呢？我们接下来就详细的讨论一下这个问题。

在了解这个问题之前我们首先需要了解什么是链接地址以及什么是加载地址。链接地址实际上就是程序自己假设在内存中存放的位置，即编译器在编译的时候会认定程序将会连续的存放在从起始处的链接地址开始的内存空间，于是像 protcseg 这样的地址标识符就被编译成了那段代码开始处的链接地址。而加载地址则是可执行程序在物理内存中真正存放的位置，而在 JOS 实验中，Boot Loader 是被 BIOS 装载到内存的，而这里 BIOS 实际上规定 Boot Loader 是要存放在物理内存的 0x7c00 处，于是不论程序的链接地址怎么改变，它装载在内存中的位置都是不会变化的。

可是在这里就出现了一个问题，假如说程序的链接地址改变了会发生什么呢？下面我们就来讨论这个问题。lab1 中的 boot/Makefrag 文件的第 28 行实际上规定了 Boot Loader 的链接地址：(V)(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out $^

其中“0x7C00”便是规定的链接地址。在这里我们将它改成 0x7C10 然后重新编译后执行看看会发生什么情况。

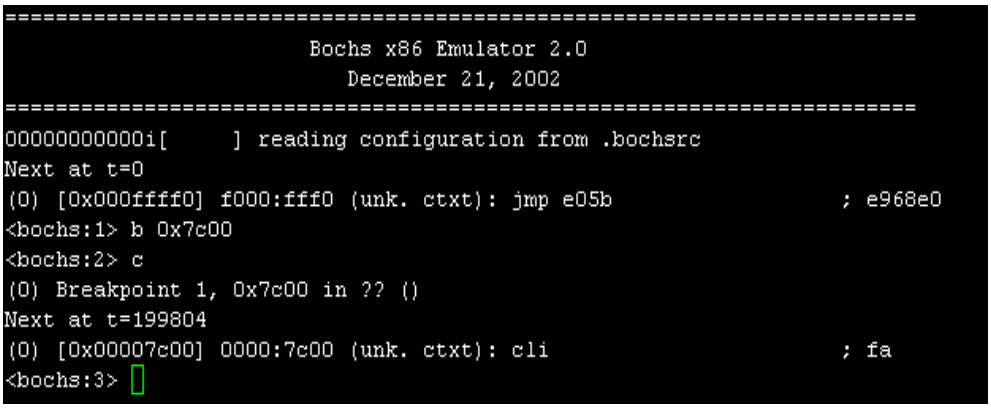


图 3-5 启动 Bochs 并在 0x7c00 处设置断点

首先我们先在物理内存的 0x7c00 处设置一个断点，因为我们知道 Boot Loader 一定会加载在这个位置，果然 0x7c00 处的第一条指令便是 boot.S 的第一条可执行指令。然后我们使用 Bochs 的单步执行命令“s”跟踪 Boot Loader 的执行，在执行了若干步后便出现了如下图所示的情况。

```
(0) [0x00007c2a] 0000:7c2a (unk. ctxt): mov CR0, EAX              ; 0f22c0
<bochs:22> s
Next at t=199824
(0) [0x00007c2d] 0000:7c2d (unk. ctxt): jmp 0008:7c42             ; ea427c0800
<bochs:23> s
========================================================================
Event type: PANIC
Device: [CPU  ]
Message: exception(): 3rd (13) exception with no resolution

A PANIC has occurred.  Do you want to:
  cont       - continue execution
  alwayscont - continue execution, and don't ask again.
               This affects only PANIC events from device [CPU  ]
  die        - stop execution now
  abort      - dump core
  debug      - continue and return to bochs debugger
Choose one of the actions above: [die]
```

图 3-6 Bochs 的出错画面

我们发现，当执行到位于内存中 0x7c2d 位置的的“jmp 0008：7c42”指令出现了错误。在这种情况下，我们可以查看一下编译的同时生成的反汇编文件 obj/boot/boot.asm，这样的文件会显示链接地址和相应的汇编指令。可以看到由于我们修改了链接地址，于是“jmp 0008：7c42”这条指令的链接地址实际上是 0x7c3d，所以它之后的由 protcseg 标识的指令的链接地址是 0x7c42，和它在物理内存中存放的位置是不一样的，于是跳转后当 CPU 发现无法识别下一条指令的时候便报错误。

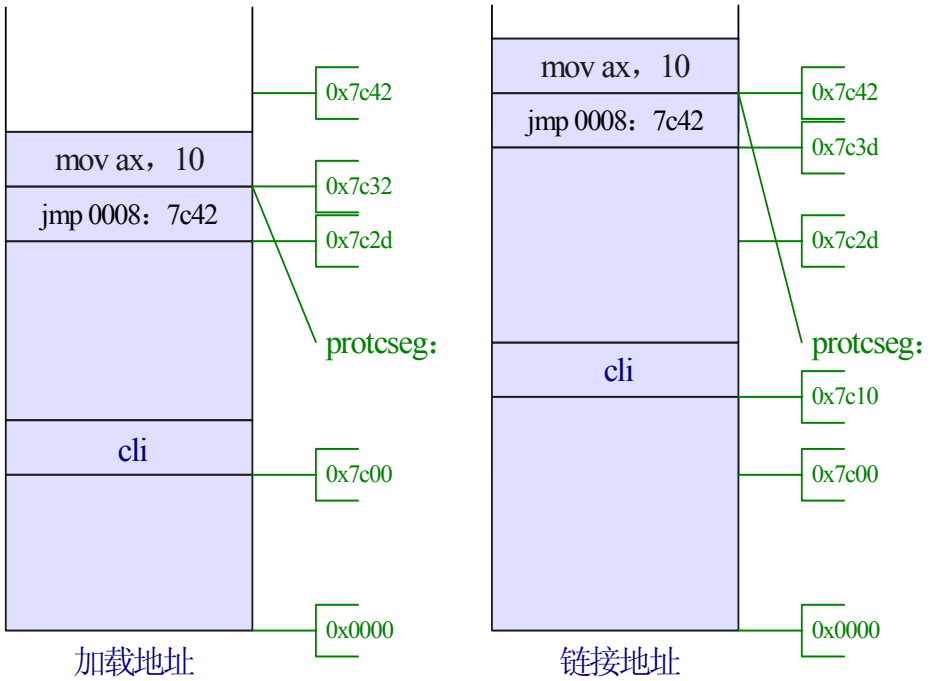


图 3-7 链接地址与加载地址

从上图中可以看出，在链接地址为准的情况下“jmp 0008：7c42”这句指令可以成功的跳转到“mov ax，10”这条指令处，然而当加载内存中以加载地址为准的时候，由于加载地址与链接地址的起始位置不同，于是实际上“mov ax，10”这条指令并不在 0x7c42 这个位置而是在 0x7c32 处，但是指令中的地址还是 0x7c42 没有变，这样程序就会跳转到一个错误的地方从而出错。

读者应该从上述的讲解中对链接地址和加载地址的区别有了一定得了解，在这里由于是 BIOS 规定 Boot Loader 必须加载在 0x7c00 这个地方，所以该程序的链接地址是固定的。而在今后讲解中我们会进一步了解到链接地址到加载地址的转换过程。

5. ELF 文件

从之前的讲解中我们知道 Boot Loader 除了将系统从实模式转换到保护模式之外还有一个重要的任务就是将内核的可执行程序加载到内存，在 JOS 操作系统实验中，这个可执行程序实际上就是一个 ELF 文件，所以要了解内核是如何装入内存的，我们首先需要了解一下 ELF 文件。

在这里，我们仅仅之需要简单的了解一下 ELF 文件的基本组成原理，以便之后能够很好的理解内核可执行文件以及其它的一些 ELF 文件加载到内存的过程。首先，ELF 文件可以分为这样几个部分：ELF 文件头、程序头表(program header table)、节头表(section header table)和文件内容。而其中文件内容部分又可以分为这样的几个节：.text 节、.rodata 节、.stab 节、.stabstr 节、.data 节、.bss 节、.comment 节。如果我们把 ELF 文件看做是一个连续顺序存放的数据块，则下图可以表明这样的一个文件的结构。

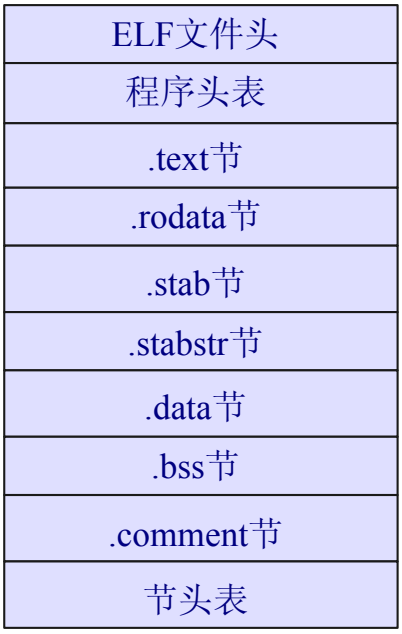


图 3-8 ELF 文件结构

从图 3-8 中可以看出 ELF 文件中需要读到内存的部分都集中在文件的中间，下面我们首先就介绍一下中间的这几个节的具体含义：

.text 节：可执行指令的部分。

.rodata 节：只读全局变量部分。

.stab 节：符号表部分，这一部分的功能是程序报错时可以提供错误信息，具体的在第三章我们会详细的介绍。

.stabstr 节：符号表字符串部分，具体的也会在第三章做详细的介绍。

.data 节：可读可写的全局变量部分。

.bss 节：未初始化的全局变量部分，这一部分不会在磁盘有存储空间，因为这些变量并没有被初始化，因此全部默认为 0，于是在将这节装入到内存的时候程序需要为其分配相应大小的初始值为 0 的内存空间。

.comment 节：注释部分，这一部分不会被加载到内存。

除了这些文件的具体内容以外，ELF 文件头、程序头表和节头表都是用来让程序来识

别 ELF 文件并且找到具体数据的位置。

ELF 文件头的数据结构如下所示：

```
struct Elf {
    uint32_t e_magic;   // 标识文件是否是 ELF 文件
    uint8_t e_elf[12];   // 魔数和相关信息
    uint16_t e_type;     // 文件类型
    uint16_t e_machine;// 针对体系结构
    uint32_t e_version; // 版本信息
    uint32_t e_entry;    // Entry point  程序入口点
    uint32_t e_phoff;    // 程序头表偏移量
    uint32_t e_shoff;    // 节头表偏移量
    uint32_t e_flags;    // 处理器特定标志
    uint16_t e_ehsize;    // 文件头长度
    uint16_t e_phentsize;// 程序头部长度
    uint16_t e_phnum;    // 程序头部个数
    uint16_t e_shentsize;// 节头部长度
    uint16_t e_shnum;    // 节头部个数
    uint16_t e_shstrndx; // 节头部字符索引
};
```

ELF 文件头比较重要的几个结构体成员是 e_entry、e_phoff、e_phnum、e_shoff、e_shnum。其中 e_entry 是可执行程序的入口地址，即从内存的这个闻之开始执行，在这里入口地址是虚拟地址，也就是链接地址；e_phoff 和 e_phnum 可以用来找到所有的程序头表项，e_phoff 是程序头表的第一项相对于 ELF 文件的开始位置的偏移，而 e_phnum 则是表项的个数；同理 e_ shoff 和 e_ shnum 可以用来找到所有的节头表项。

程序头表实际上是将文件的内容分成了好几个段，而每个表项就代表了一个段，这里的段是不同于之前节的概念，有可能就是同时几个节包含在同一个段里。程序头表项的数据结构如下所示：

```
struct Proghdr {
    uint32_t p_type;     // 段类型
    uint32_t p_offset;   // 段位置相对于文件开始处的偏移量
    uint32_t p_va;       // 段在内存中地址(虚拟地址)
    uint32_t p_pa;       // 段的物理地址
    uint32_t p_filesz;   // 段在文件中的长度
    uint32_t p_memsz; // 段在内存中的长度
    uint32_t p_flags;    // 段标志
    uint32_t p_align;    // 段在内存中的对齐标志
};
```

这里比较重要的几个成员是 p_offset、p_va、p_filesz 和 p_memsz。其中通过 p_offset 可以找到该段在磁盘中的位置，通过 p_va 可以知道应该把这个段放到内存的那个位置，而之所以需要 p_filesz 和 p_memsz 这两个长度是因为像之前所讲的.bss 这种节在硬盘没有存储空间而在内存中程序需要为其分配空间。

下面我们通过一个图来看看用 ELF 文件头与程序头表项如何找到文件的第 i 段。

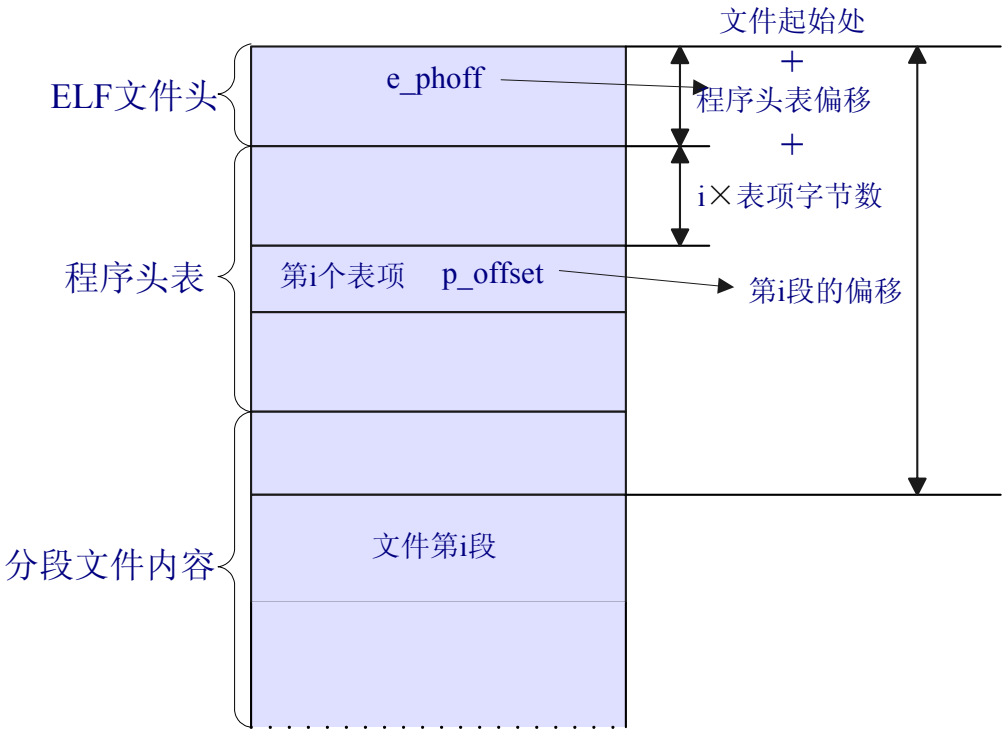
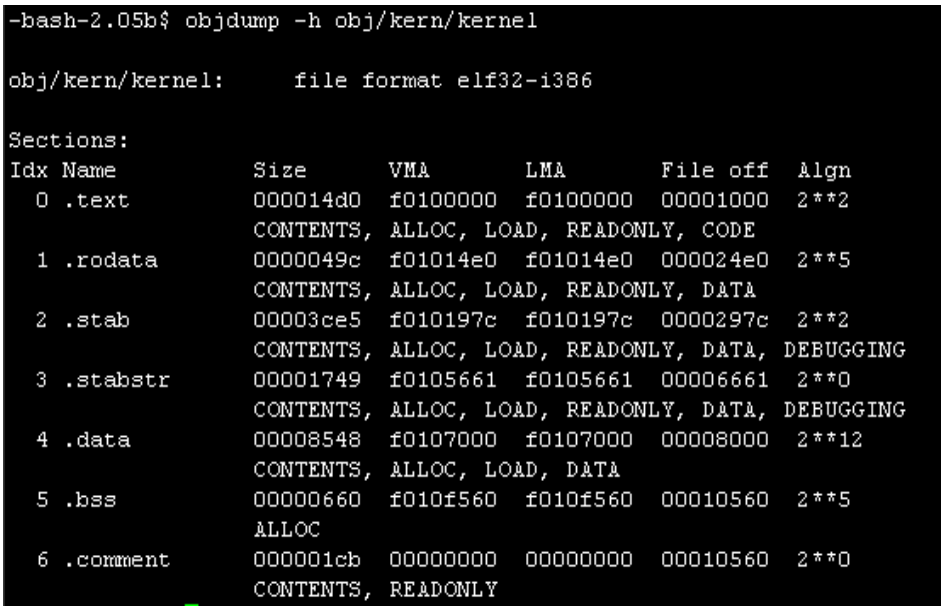


图 3-9 找到文件第 i 段的过程

而另一个节头表的功能则是让程序能够找到特定的某一节，其中节头表项的数据结构如下所示：

```
struct Secthdr {
    uint32_t sh_name;     // 节名称
    uint32_t sh_type;     // 节类型
    uint32_t sh_flags;    // 节标志
    uint32_t sh_addr;     // 节在内存中的虚拟地址
    uint32_t sh_offset;   // 相对于文件首部的偏移
    uint32_t sh_size;     // 节大小(字节数)
    uint32_t sh_link;     // 与其它节的关系
    uint32_t sh_info;     // 其它信息
    uint32_t sh_addralign; // 字节对齐标志
    uint32_t sh_entsize;  // 表项大小
};
```

而通过 ELF 文件头与节头表找到文件的某一节的方式和之前所说的找到某一段的方式是类似的。而在试验中，我们可以用“objdump –h 可执行文件”这样的命令来查看 ELF 文件的每个节的信息。



```
-bash-2.05b$ objdump -h obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000014d0  f0100000  f0100000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       0000049c  f01014e0  f01014e0  000024e0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003ce5  f010197c  f010197c  0000297c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA, DEBUGGING
  3 .stabstr      00001749  f0105661  f0105661  00006661  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA, DEBUGGING
  4 .data         00008548  f0107000  f0107000  00008000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000660  f010f560  f010f560  00010560  2**5
                  ALLOC
  6 .comment      000001cb  00000000  00000000  00010560  2**0
                  CONTENTS, READONLY
```

图 3-10 使用命令查看 kernel 可执行文件节信息

在上图中，我们使用“objdump –h obj/kern/kernel”命令查看 kernel 文件，可以看到 kernel 可执行文件是由我们之前所说的 7 个节所组成的，其中我们还可以发现.bss 节与.comment 节在文件中的偏移是一样的，这就说明.bss 在硬盘中式不占用空间的，仅仅只是记载了它的长度。

6. 内核的装入过程

在对 ELF 文件有了基本认识后，我们再来研究一下 Boot Loader 具体是如何将 kernel 的可执行 ELF 文件加载到内存中的。我们在之前讲过 lab1 中的 boot/main.c 源程序的功能就是从硬盘读取 kernel，下面我们就分块来分析一下这个 C 程序。

```
#include <inc/x86.h>
#include <inc/elf.h>
#define SECTSIZE 512
#define ELFHDR          ((struct Elf *) 0x10000) // 定义一个指向内存中 ELF 文件头存放位置的结构体指针。
void readsect(void*, uint32_t);
void readseg(uint32_t, uint32_t, uint32_t);
```

readsect(void*, uint32_t)是程序需要使用的一个读取磁盘上一个扇区的函数，而 readseg(uint32_t, uint32_t, uint32_t)则是读取 ELF 文件中一段的一个函数。在这里还定义了 ELF 文件头应该存放在内存的 0x10000 处。

```
void bootmain(void)
{
    struct Proghdr *ph, *eph;
    readseg((uint32_t) ELFHDR, SECTSIZE*8, 0); // 将文件的前 4KB 读入内存
    if (ELFHDR->e_magic != ELF_MAGIC) // 判断该文件是否为 ELF 文件
        goto bad;
    ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff); // 将指针指向程序头表的首地址
    eph = ph + ELFHDR->e_phnum; // 明确文件段的个数
    for (; ph < eph; ph++)
        readseg(ph->p_va, ph->p_memsz, ph->p_offset); // 用 readseg 函数依次将文件的每一段读入内存中相应的位置
    ((void (*)(void)) (ELFHDR->e_entry & 0xFFFFFF))(); // 在将内核加载到内存中后转移到内核入口地址处执行，并且不会再返回
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
    while (1)
}
```

可以看到 SECTSIZE*8 代表 4KB，这就是说在一开始，程序就将文件的前 4KB 读入内存，这其中包括 ELF 文件头以及程序头表，根据我们之前所讲解的可以得知这样就可以找到文件的每一段。于是紧接着程序利用 readseg 函数将内核文件的每一段依次读入内存然后转到内核入口地址执行。实际上我们可以通过“objdump –f 可执行文件”命令来查看 ELF 文件的入口地址。

```
-bash-2.05b$ objdump -f obj/kern/kernel

obj/kern/kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0xf010000c
```

图 3-11 用命令查看程序入口地址

如上图所示，可以看到 kernel 的入口地址是 0xf010000c，然而在程序跳转的时候，却将 0xf010000c 与 0xFFFFFF 相与以后的值作为入口地址，这是因为 kernel 程序的链接地址是 0xf0100000，而由于实际的物理内存没有那么大，在 0xf0100000 地址处并没有物理内存，所以真正的加载地址实际上是 0x100000，相应的入口地址也就是 0x10000c，即是 0xf010000c 与 0xFFFFFF 相与以后的值。

```
void readseg(uint32_t va, uint32_t count, uint32_t offset)
{
    uint32_t end_va;
    va &= 0xFFFFFF;       // 将链接地址转换成加载地址
    end_va = va + count;   // 找到内存中加载地址的最末端
    va &= ~(SECTSIZE - 1); // 由于硬盘中的每一扇区加载到内存的时候都需要512 字节对齐，于是在这里把起始加载地址向下对齐到 512 字节的倍数的地址处
    offset = (offset / SECTSIZE) + 1; // 将在硬盘中的偏移由字节数转换成扇区数，由于内核可执行程序是从磁盘的第二个扇区开始存储的，所以需要加 1
    while (va < end_va) {
        readsect((uint8_t*) va, offset);
        va += SECTSIZE;
        offset++;   // 用 readsect 函数一个扇区一个扇区的读取文件的这一段
    }
}
```

readseg 函数是用来从硬盘读取我文件的一段，它有三个参数 va、count、offset，其中 va 代表这一段的链接地址，count 代表这一段在内存中所占得字节数，offset 代表这一段在文件中的相对于文件首的偏移。

当我们使用 Bochs 调试命令查看内存的中的 ELF 文件头的相应信息后，我们发现 kernel 可执行文件实际上是分为两段，第一段在文件中的偏移是 0x1000，而在内存中占据的字节数是 0x6daa，链接地址 0xf0100000；而第二段在文件中的偏移是 0x8000，而在内存中占据的字节数是 0x8bc0，链接地址 0xf0107000；

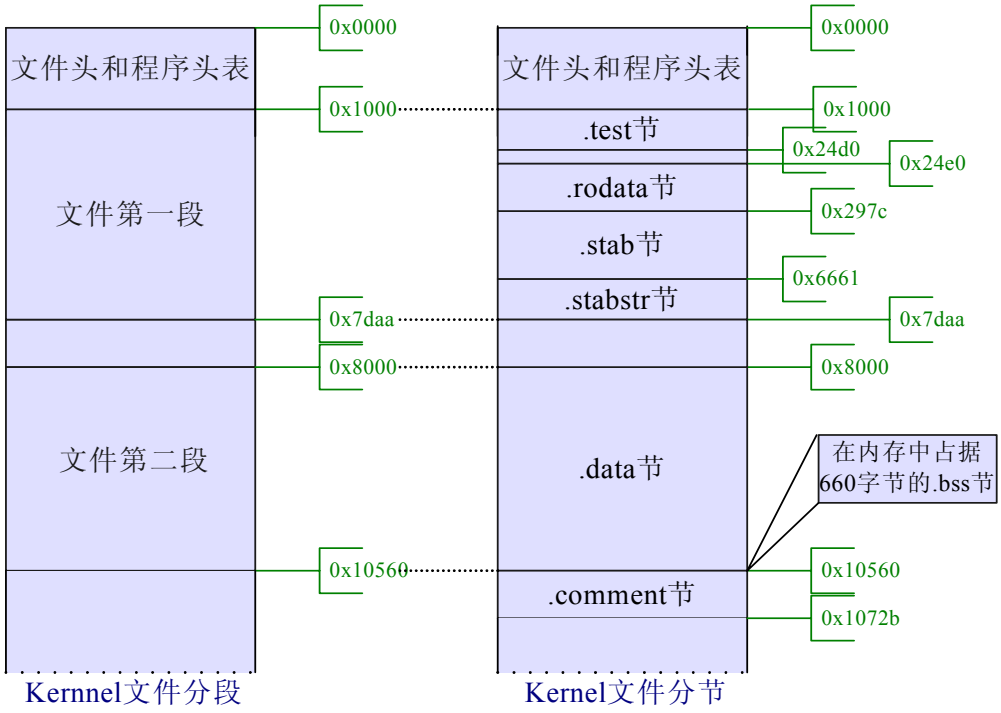


图 3-12 kernel 的分段和分节

在上图中可以看到 kernel 可执行文件的第一段包含了 ELF 文件的.test 节、.rodata 节、.stab

节以及.stabstr 节，而文件的第二段包含了.data 节以及在硬盘上不占用空间但在内存中占据 660 字节的.bss 节，在这里程序头表的第二项会用 p_filesz 成员变量标注该段在文件占用的字节数并且同时用 p_memsz 标注在内存中占用的字节数，这样 Boot Loader 便会在从硬盘读入第二段的同时为.bss 节在内存中分配空间。另外我们可以看到，.comment 节没有被包含在任意一段中，这表明它没有被装入内存。

```
void waitdisk(void)
{
    while ((inb(0x1F7) & 0xC0) != 0x40) //  循环直到硬盘准备好
}
```

waitdisk 函数的作用是等待直到硬盘准备好可以让程序读取数据。

```
void readsect(void *dst, uint32_t offset)
{
    waitdisk();   //  等待直到硬盘准备好
    outb(0x1F2, 1);
    outb(0x1F3, offset);
    outb(0x1F4, offset >> 8);
    outb(0x1F5, offset >> 16);
    outb(0x1F6, (offset >> 24) | 0xE0);
    outb(0x1F7, 0x20);
    waitdisk();   //  等待直到硬盘准备好
    insl(0x1F0, dst, SECTSIZE/4); //  读取一个扇区的数据
}
```

readsect 函数的作用是从硬盘读取一个扇区的数据，函数参数 offset 代表硬盘的第几个扇区，而 dst 代表这 512 字节的数据应该存放在内存中的哪个位置，这里具体指的是物理地址。对于这个函数，我们只需要明白它大概的功能就行了而不用深究它每一句的代码的含义。

3.2 显示输出

我们在刚接触 C 语言的时候应该都用过 printf()这个函数，用这个函数我们可以非常容易的在屏幕上输出字符串。在通常情况下，我们都会将这个函数视为现成的，可以直接使用的，可是在 JOS 试验中，所有底层操作都需要操作系统的代码来实现，于是在实现这个简单的操作系统的同时我们也重新将 printf()函数的功能实现了一遍。当然，与显示输出有关的绝大多数代码在实验中已经写好了，所以在这里，我们需要做的是了解一下显示输出的功能是如何实现的。

1. 关于 cprintf()函数功能的实现

实际上，在 JOS 实验中，我们并没有 printf()这个函数，取而代之的则是 cprintf()这个函数，接下来我们需要研究的就是 cprintf()的功能是如何实现的。

在 lab1 中，与实现显示输出相关的文件有 3 个，它们分别是 kern/printf.c、kern/console.c 以及 lib/printfmt.c。首先，我们来看看 kern/printf.c，我们所研究的 cprintf()函数就是在这个文件中定义的。

我们首先看看 kern/printf.c 中 cprintf()函数的原型。

```
int    cprintf(const char *fmt, ...)
{
       va_list ap;
       int cnt;
       va_start(ap, fmt);
       cnt = vcprintf(fmt, ap);
       va_end(ap);
       return cnt;
}
```

可以看到，这个函数的参数的个数以及类型都是不确定的，我们应该还记得刚开始学习 C 语言的时候，我们会用诸如“printf(“%d %c”,a,b)”这样形式的语句打印数字、字符或者是字符串，而在这里实际上 cprintf 也是一样的道理，第一个参数 fmt 代表的是显示字符串的指针，显示字符串就是像我们在 printf 函数中使用的诸如“%d %c”这样的东西，而 fmt 之后的参数的值就是用来在显示的时候替代%d、%c 之类的东西。在这里 cprintf 函数的参数除了第一个 fmt 是确定的，后面的参数都是不确定的，可以是常数、整形变量、浮点变量、字符、字符串指针。具体的 cprintf 函数的使用格式我们会在之后介绍 cprintfmt 函数时做详细的介绍。

我们再来看看函数体，函数首先定义了一个 va_list 的变量 ap，va_list 是在 C 语言中解决变参问题的一组宏，在这里，我们对这个宏只需要一下几点：

1) 用 va_list 可以定义一个 va_list 型的变量，这个变量是指向参数的指针。
2) 用 va_start 宏可以初始化一个 va_list 变量，这个宏有两个参数，第一个是 va_list 变量本身，第二个是可变的参数的前一个参数，是一个固定的参数。
3) 用 va_arg 宏可以返回可变的参数，这个宏也有两个参数，第一个是 va_list 变量，即指向参数的指针，第二个是我们要返回的参数的类型。
4) 最后我们还可以用 va_end 宏结束可变参数的获取。

在 cprintf 函数中，我们可以看到在定义了 va_list 变量 ap 后马上就用 va_start 宏对其进行了初始化，用 va_start 宏进行初始化的时候第二个参数是 fmt，也就是 cprintf 函数的第一个参数，即可变参数之前的固定参数，于是这时 ap 便指向了后面的可变参数。如图 2-1 所示，函数的参数实际上都是存放在内存的堆栈中的，而且参数会按照先后顺序依次存放，靠前的参数会存放在较低的地址处，其中每个参数会根据其类型被分配相应大小的空间。于是 ap 在这个时候便指向了可变参数 1 的存放地址，这样我们就可以用 va_arg 宏依次读取之后的可变参数。在对参数指针进行了初始化后，程序接着调用了 vcprintf 函数，在得到 vcprintf 函数的返回值后，最后便使用 va_end 宏结束了对可变参数的获取。

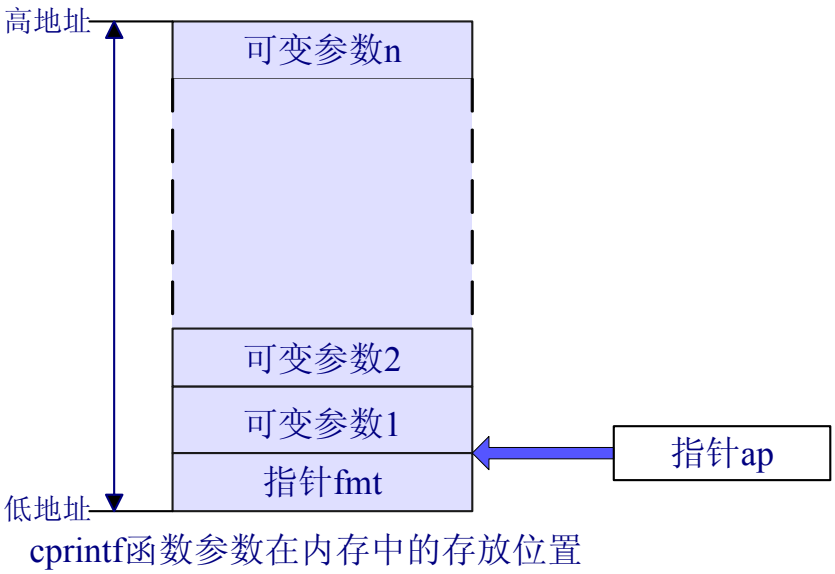


图 3-13 cprintf 函数参数与指针 ap 的关系

接着我们就来看看 cprintf 函数所调用的 vcprintf 函数。

```
int    vcprintf(const char *fmt, va_list ap)
{
       int cnt = 0;
       vprintfmt((void*)putch, &cnt, fmt, ap);
       return cnt;
}
```

这个函数有两个参数 fmt 和 ap，其中 fmt 是之前所说的显示字符串的指针，而 ap 当然就是之前 cprintf 函数的可变参数指针。vcprintf 在函数体中又调用了 vprintfmt 函数，而整形变量 cnt 则是函数的返回值。在这里，我们可以看到 vprintfmt 函数的第一个参数实际上是一个函数指针，putch 函数被当做成了一个参数，putch 函数的功能是输出一个字符在屏幕上，函数体如下所示：

```
static void putch(int ch, int *cnt)
{
       cputchar(ch);
       *cnt++;
}
```

putch 函数有两个参数，其中整形变量 ch 代表的是要输出的字符，因为 int 的变量是 32 位的，而一个字符的 ASCII 码只需要有 8 位，所以实际上 32 位整形变量的低八位代表的是字符的 ASCII 码，而第 8 位到 15 位代表的是输出字符的格式，因此 int 变量的高 16 位实际上没有用的；而 cnt 指针指向一个整形变量，这个整形变量每当用 putch 函数输出一个字符后就加 1。图 3-14 表示的是用 putch 函数输出一个字符的函数调用过程图。

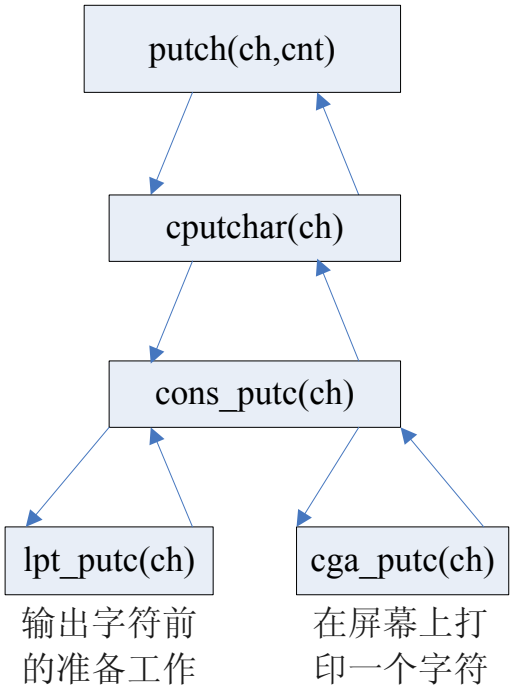


图 3-14 用 putch 函数打印一个字符的函数调用过程

在屏幕上打印一个字符的功能主要是由上图中的 lpt_putc 函数和 cga_putc 函数实现的，其中对于 lpt_putc 函数我们并不需要做过多的了解，只需要知道它是用来进行一些硬件初始化的工作即可。而 cga_putc 则是我们所需要了解的一个函数，这个函数有一个 int 类型的参数，没有返回值。下面我们就分段来解析一下这个函数。

```
void    cga_putc(int c)
{
      //如果没有设置字符格式，则默认为白色背景上的黑色字体
      if (!(c & ~0xFF))
            c |= 0x0700;
```

首先看看在函数开头处的这一条条件判断语句，我们在之前有讲过，从 putch 函数传递过来的整形参数 c 的低 8 位是字符的 ASCII 码，而 8 到 15 位则是字符输出的格式，于是在这里函数首先判断字符的格式有没有事先设定，如果没有，即 8 到 15 位皆为 0，则系统将会将这个字符的设定为默认格式。下图表示的是 8 到 15 位是如何确定字符的格式的：

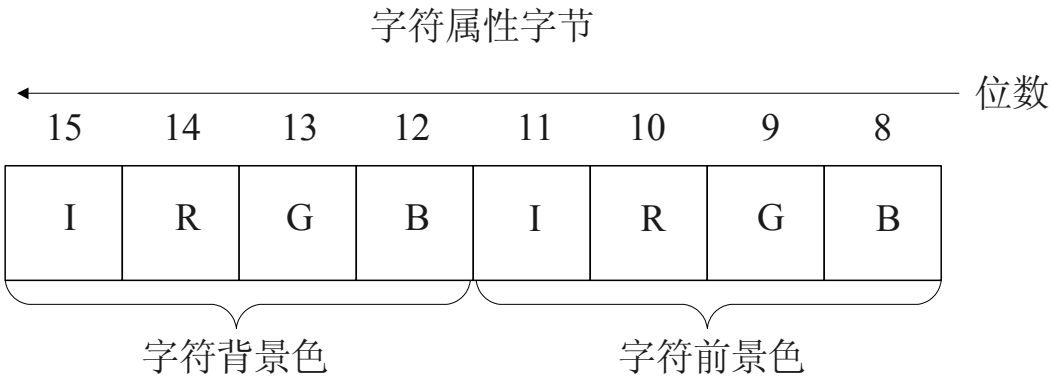


图 3-15 字符显示属性字节

可以看到高 4 位决定了字符的背景色，可以有 16 种颜色，同样低四位则决定了字符本身的颜色，在这里，R 代表红色的色素，G 代表绿色的色素，B 代表蓝色的色素，这三个色素的组合就可以组成 8 种不同的颜色，而 I 则表示颜色是否是高亮的，于是这样便可以有 16 种颜色。在确定了字符的显示属性后，程序便开始判断如何进行输出：

```
switch (c & 0xff) {
    case '\b':    // 表示退格
        if (crt_pos > 0) {
            crt_pos--;
            crt_buf[crt_pos] = (c & ~0xff) | ' ';
        }
        break;
    case '\n':    // 表示换行
        crt_pos += CRT_COLS;
    case '\r':    // 表示光标退到这一行的开头处
        crt_pos -= (crt_pos % CRT_COLS);
        break;
    case '\t':    // 光标向前移动 5 格
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
        break;
    default:
        crt_buf[crt_pos++] = c;        // 往屏幕上打印一个字符
        break;
    }
```

在这里，crt_buf 是一个指向 16 位无符号整形数的静态指针，它实际上指向的是内存中物理地址为 0xb8000 的位置，我们在 2.1 节中有讲到物理内存的 0xa0000 到 0xc0000 这 128KB 的空间是留给 VGA 显示缓存的，实际上在我们的试验中从 0xb8000 这个位置开始的一部分内存空间便是可以直接与显示屏相关联的显存，图 2-4 详细的讲述了内存空间是如何和显示屏相关联的。

在本实验中，显示屏规定为 25 行，每行可以输出 80 个字符，由于每个字符实际上占据显存中的两个字节，于是物理内存中从 0xb8000 到 0xb8fa0 之间的内容都会以字符的形式在屏幕上显示出来。crt_pos 是一个静态的 16 位无符号整形变量，如果把 crt_buf 指向的内存空间看做 16 位整形数的数组，则 crt_pos 则是数组的下标，在这里它实际上表示的是光标的位置，而 CRT_COLS 则是一个常量，表示一行可以输出的字符数，即为 25。

于是当字符为’\b’时，表示要退格，就把 crt_pos 减 1 表示光标向后退一格，并且将光标当前指向位置对应的显存中的两个字节的值置为(c & ~0xff) | ' '，即把原来的字符替换为了一个空格，这样便完成了退格的操作。

当字符为’\n’时，表示换行，则把 crt_pos 加上 25，即将光标的位置换到下一行相同的位置。

当字符为'\r'时，表示光标退到这一行的开头之处，于是将 crt_pos 减去(crt_pos % CRT_COLS)。

当字符为'\t'时，表示光标向前移动 5 格，所以就递归的调用 cons_putc 函数连续打印 5 个空格。

当字符不是以上的这些特殊字符时，程序便将其直接写入显存中，并且将光标的位置加 1，值得注意的是在这里 crt_buf[crt_pos++]表示的是内存中 16 位的空间，然而整形变量 c 是 32 位的，于是在写入内存的时候只取 c 的低 16 位。

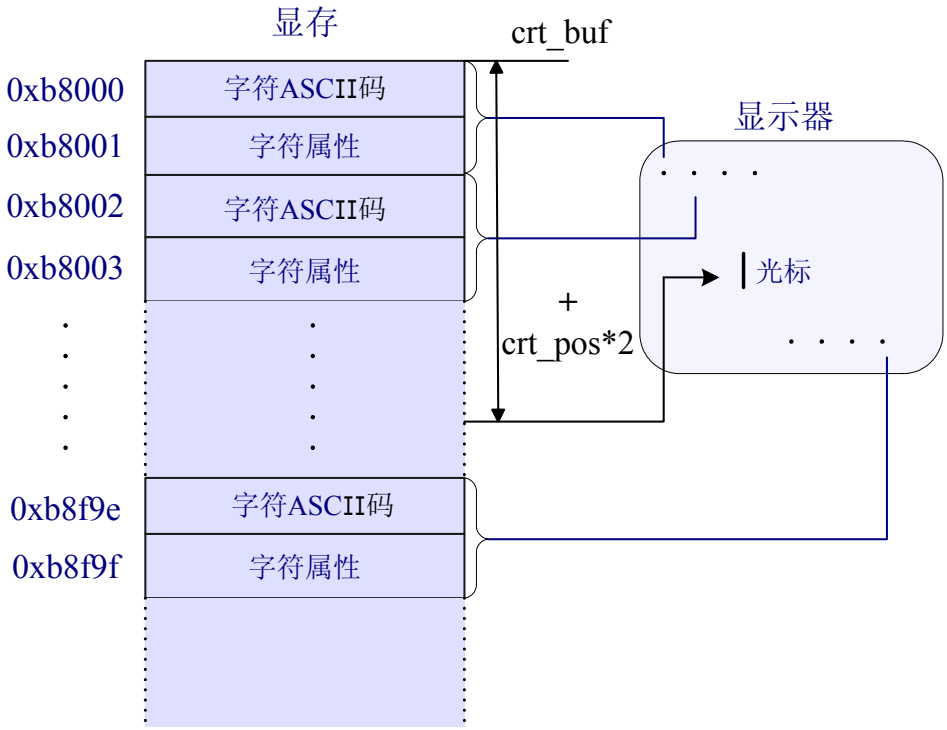


图 3-16 显存与显示屏的对应关系

```
if (crt_pos >= CRT_SIZE) {
        int i;
        memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) *
sizeof(uint16_t));// 执行滚屏操作
        for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
            crt_buf[i] = 0x0700 | ' '; // 将最后一行全部输出为空格
        crt_pos -= CRT_COLS; // 重订光标的位置
    }

    outb(addr_6845, 14);
    outb(addr_6845 + 1, crt_pos >> 8);
    outb(addr_6845, 15);
    outb(addr_6845 + 1, crt_pos);
}
```

在这之后函数便要考虑显存溢出的问题，即在物理地址超过 0xb8fa0 的内存部分中存储字符数据，此时实际上显示屏就无法显示超过的部分，这个时候通常下显示屏都会滚屏好让

最新输出的字符能够显示出来。

在这里，我们可以看到程序在每输出一个字符后都先判断 crt_pos 是否大于或等于 CRT_SIZE，而 CRT_SIZE 实际上就是一个屏幕可以输出的字符数，即 80*25。当等于 CRT_SIZE 时，说明此时已经满屏，当大于 CRT_SIZE 时，说明有字符没有显示出来。当满足这两种情况的中的一种时，程序所做的处理是将屏幕上第二行到最后一行的的字符数据复制到第一行到倒数第二行去，然后将屏幕的最后一行输出为空格。这是因为由于每次输出一个字符后都会进行判断，所以只有在输出'\n'或者'\t'时才会有可能造成光标位置超出屏幕范围的情况，而这个时候应该输出的最后一行的字符一定都是空格。在滚屏后最后要做的便是将光标的位置设置成当前正确的位置，由于屏幕向上滚了一行，于是光标也向前移动一行。

cga_putc 函数最后连续引用了 outb 函数，在这里我们只需要了解这是用来对硬件进行一些操作就够了。

在了解了 cga_putc 函数的原理后，我们大致应该就清楚了是如何用 putch 函数将一个字符输出到屏幕上的，在了解了 putch 函数后，我们便可以开始更深入的了解 cprintf 函数是如何实现的。cprintf 函数所调用的 vcprintf 函数在执行过程中调用了 lib/printfmt.c 中的 vprintfmt 函数，而这个函数便是字符串输出功能的主要实现部分。由于函数主体部分比较长，在这里我们就不列出具体代码，下图便是这个函数的程序流程图：

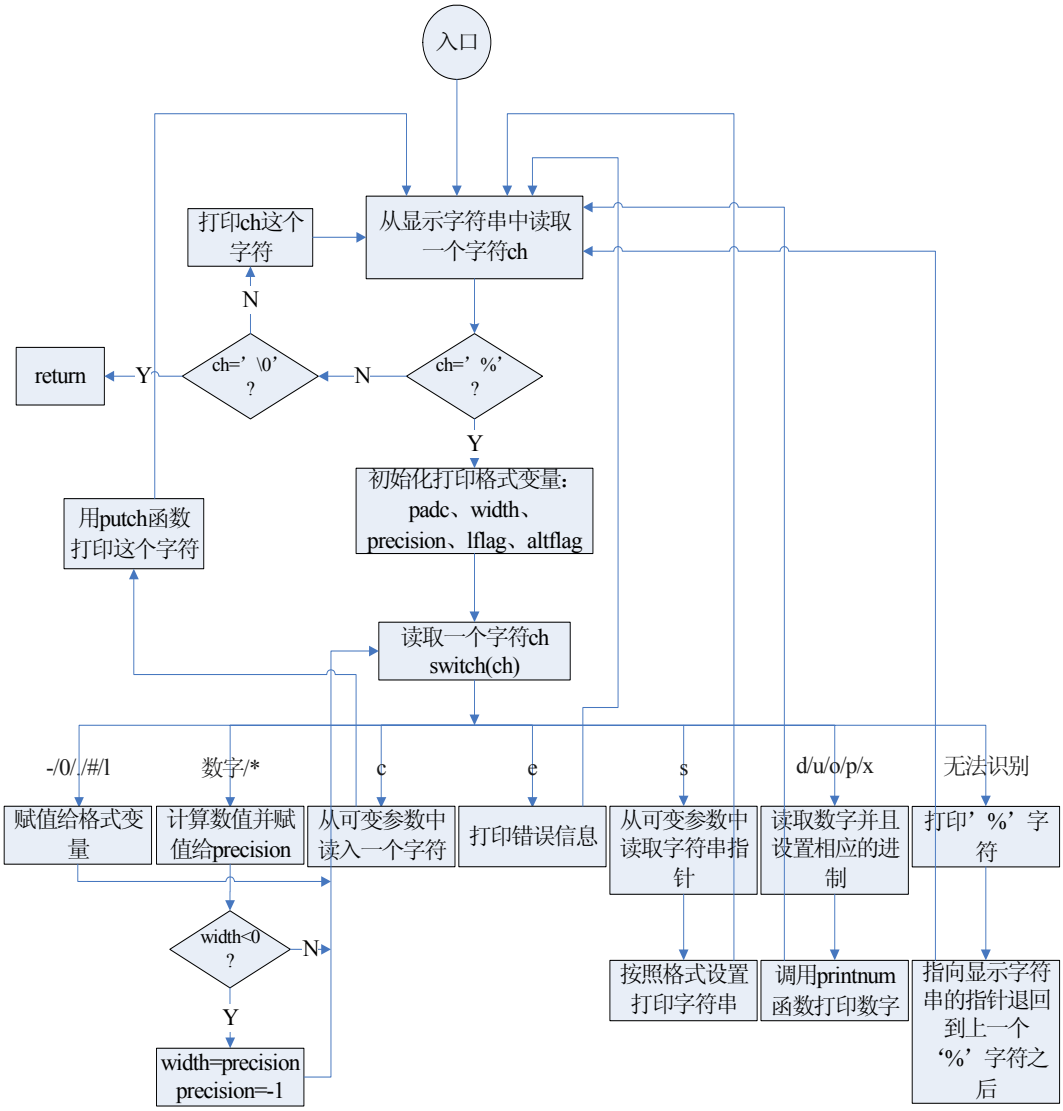


图 3-17 vprintfmt 函数的程序流程图

这里图 3-17 只是表达了一个大致的程序执行流程，而实际上有很多细节并没有显现出来。首先就是关于这 5 个格式变量：padc、width、precision、lflag、altflag。padc 代表的是填充字符，在初始化的时候 padc 变量会被初始化为空格符，而当程序在显示字符串的’%’字符后读到’ -’或者’0’的字符时便会将’ -’或者’0’赋值给 padc。width 代表的是打印的一个字符串或者一个数字在屏幕上所占的宽度，而 precision 则特指一个字符串在屏幕上应显示的长度，当 precision 大于字符串本身长度时相当于 precision 就等于字符串长度。于是在显示字符串的时候 precision 小于 width 的部分则由之前所说的填充字符 padc 来补充，如果 width 小于 precision 则字符串的宽度就等于 precision，而 precision 得默认值-1 代表显示长度为字符串本来的长度。当打印字符串的时候，padc=’-’代表着字符串需要左对齐，右边补空格，padc=’ ’代表字符串右对齐，而左边由空格补齐，padc=’0’代表字符串右对齐，左边由 0 补齐。在我们这个实验中当输出数字时会一律的右对齐，左边补 padc，数字显示长度为数字本身的长度。lfag 变量则是专门在输出数字的时候起作用，在我们这个实验中为了简单起见实际上是不支持输出浮点数的，于是 vprintfmt 函数只能够支持输出整形数，，输出整形数时，当 lflag=0 时，表示将参数当做 int 型的来输出，当 lflag=1 时，表示当做 long 型的来输出，而当 lflag=2 时表示当做 long long 型的来输出。最后，altflag 变量表示当 altflag=1 时函数若输出乱码则用’?’代替。

我们再来看看 vprintfmt 函数打印一个字符串具体是如何实现的：

```
case 's':
            if ((p = va_arg(ap, char *)) == NULL)
                p = "(null)";   // 当字符串指针为空时，将它指向"(null)"字符串
            if (width > 0 && padc != '-')
                for (width -= strnlen(p, precision); width > 0; width--)
                    putch(padc, putdat); // 字符串右对齐，左边补相应数量的空格或者 0
            for (; (ch = *p++) != '\0' && (precision < 0 || --precision >= 0); width--)
                if (altflag && (ch < ' ' || ch > '~'))
                    putch('?', putdat);
                else
                    putch(ch, putdat); // 打印相应长度的字符串
            for (; width > 0; width--)
                putch(' ', putdat); // 当字符串是左对齐的时候打印相应数量的空格
            break;
```

我们看到当程序识别了显示字符串中’%’后的’s’字符后便从可变参数中读入字符串指针，若指针为空，则让它指向一个"(null)"字符串。然后再判断输出是左对齐还是右对齐，若 padc=’-’，表示是左对齐，否则是右对齐。确认是右对齐的话，按照我们之前所讲的，用 width 减去字符串实际显示长度便得到需要在左边补空格或 0 的个数。在这之后程序便开始打印字符串本身，可以看到若 precision 大于 0 则显示长度等于 precision 与字符串长度之间的最小值，precision 小于 0 则显示长度等于字符串本身的长度。最后程序判断如果字符串是左对齐的话则在右侧剩余空间补充空格。

而在打印数字的时候则会用到 printnum 这个函数，该函数的主体如下所示。该函数的参数 num 代表需要打印出来的整形数，base 代表整形数的进制，其它的参数和 vprintfmt 函数中代表同样的意思。在这里，当 num 超过 1 位时函数递归的调用自己本身，这样便可以

先打印高位的数字，当 num 只有 1 位时，程序便首先按照右对齐的格式在左侧打印填充字符，然后打印这个数字。

可以看到无论是打印字符串还是数字，都是将它们分解成一个个单个的字符然后用 putch 函数一个一个的打印出来。所以 putch 函数可谓是显示输出的基本函数。

```
static void printnum(void (*putch)(int, void*), void *putdat, unsigned long long num,
unsigned base, int width, int padc)
{
    if (num >= base) {
        printnum(putch, putdat, num / base, base, width - 1, padc); //  先打印高位
    } else {
        while (--width > 0)
            putch(padc, putdat); //  按照右对齐的格式在左侧补齐填充字符
    }
    putch("0123456789abcdef"[num % base], putdat); //  打印 1 位数字
}
```

在我们 JOS 试验中，由于简化的原因，所以最终 cprintf 函数实现的功能实际上与我们刚开始接触 C 语言时学到的 printf 函数是有所不同的，即一些 printf 所具备的功能 cprintf 不能实现，但是尽管如此，cprintf 函数在 JOS 实验中还是能够满足我们打印的需要，而且使用格式与 printf 函数基本相同，具体格式读者可以查阅 C 语言中有关 printf 函数的相关资料，在此我们就不在多说。读者在这里需要掌握的是 JOS 中 cprintf 打印数字或者打印字符串大致的实现过程，对于具体的细节则无需深究。

2. 关于 cprintf 函数的示例

由于在前面我们讲述了 cprintf 函数的实现原理，所以在这里，我们首先将通过一个例子来帮助读者熟悉一下具体的流程：

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

在这里"x %d, y %x, z %d\n"便是显示字符串，整形变量 x、y、z 便是可变参数。在初始化了 va_list 变量 ap 后，ap 便指向了 x 所存放的位置，之后便可以通过 va_arg 宏依次得到变量 x、y、z。当函数将指向"x %d, y %x, z %d\n"字符串的指针 fmt 和指向可变参数 x、y、z 的指针 ap 传到函数 vprintfmt 后，vprintfmt 函数便可以根据显示字符串的内容打印出相应格式的内容。可以看到%d 代表以有符号十进制整数的形式打印出来，%x 代表以无符号 16 进制整数的形式打印出来。于是执行结果如下图所示：

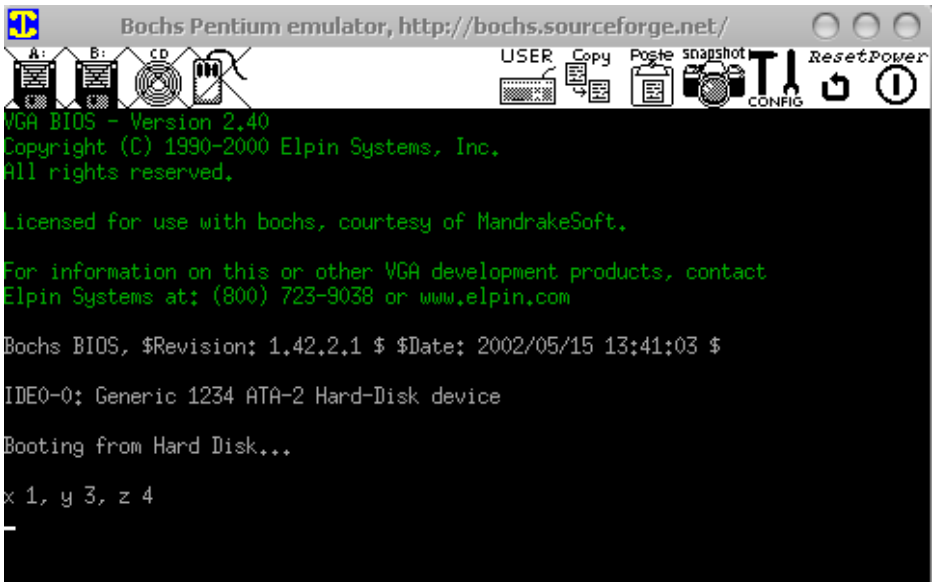


图 3-18 cprintf 函数打印示例图 1

如果把 x、y、z 都变成负数，则情况会有所不同：

int x = -1, y = -3, z = -4;

cprintf("x %d, y %x, z %d\n", x, y, z);

执行结果如下图所示：

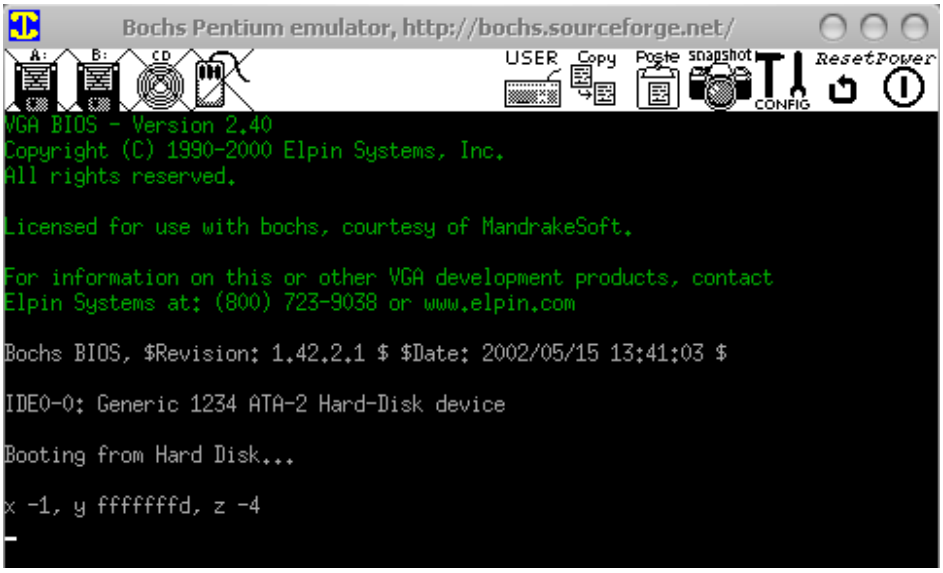


图 3-19 cprintf 函数打印示例图 2

可以看到，x 打印出来是-1，z 打印出来是-4，然而 y 打印出来却不是-3，而是 fffffffd，这是因为在显示字符串中%x 表示无符号的 16 进制数，而-3 在内存中是以补码的形式存储的，于是在内存中-3 实际上是 fffffffd，所以将它看做是一个无符号数时，打印出来的结果便是 fffffffd。值得注意的是，在这里，我们都是在内核启动后执行的第一个函数中添加这样的 cprintf 语句，于是在启动信息打印出来后便打印出了 cprintf 函数的运行结果。

接下来我们再来看一个例子：

unsigned int i = 0x00646c72;

cprintf("H%x Wo%s", 57616, &i);

这段代码的运行结果如下图所示：

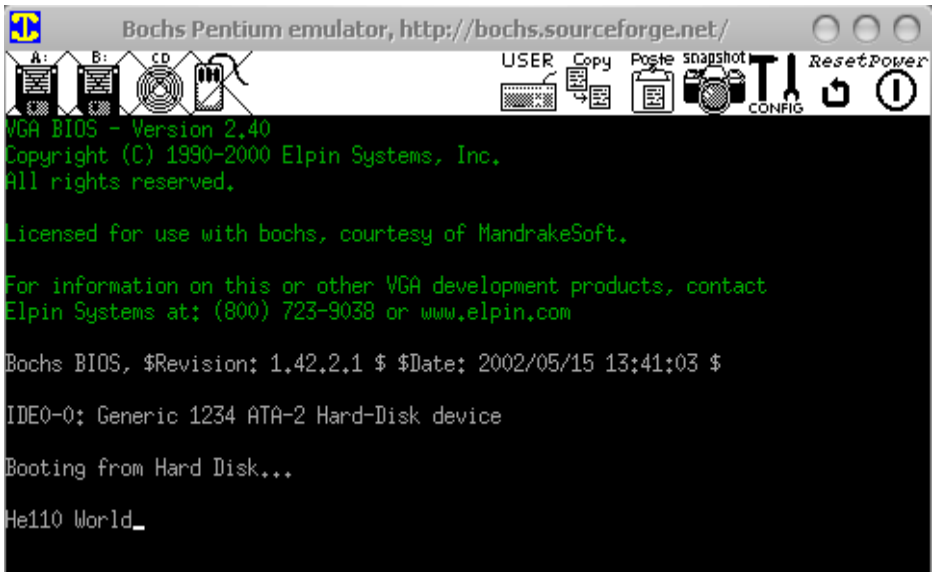


图 3-20 cprintf 函数打印示例图 3

读者可能会对这个结果感到奇怪，为什么会打印出“Hello World”呢？仔细观察，才发现“Hello”实际上是由“H”与数字 e110 组成的，这就是因为十进制数 57616 用 16 进制数来表示便是 e110。而无符号整形数在这里则表示了一个字符串“rld”，如下图所示，由于无符号整形数是占 4 个字节，而低位数字是存在低地址处。若将这四个字节看做一个字符串，则每个字节代表的就是一个字符的 ASCⅡ码，所以低位的 0x72 代表的是字符‘r’，而最高位的 0x00 代表就是空字符，即标识字符串的结束。于是字符串与“Wo”组成了“World”，所以最终在屏幕上输出了“He110 World”。

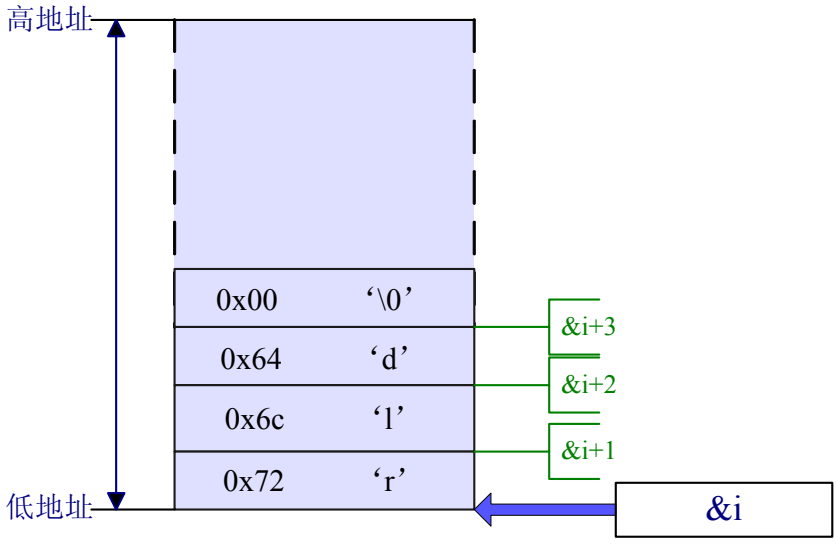


图 3-21 无符号整形数 0x00646c72 所表示的字符串

最后我们来看一个例子：

cprintf("x=%d y=%d", 3);

可以看到在句 cprintf 函数的调用中，出现了两个“%d”，意味着需要两个整形的可变参数，然而函数的可变参数却只有 3 一个，那么这样最后的输出结果会是什么呢？经过试验后，我们发现结果如下图所示：

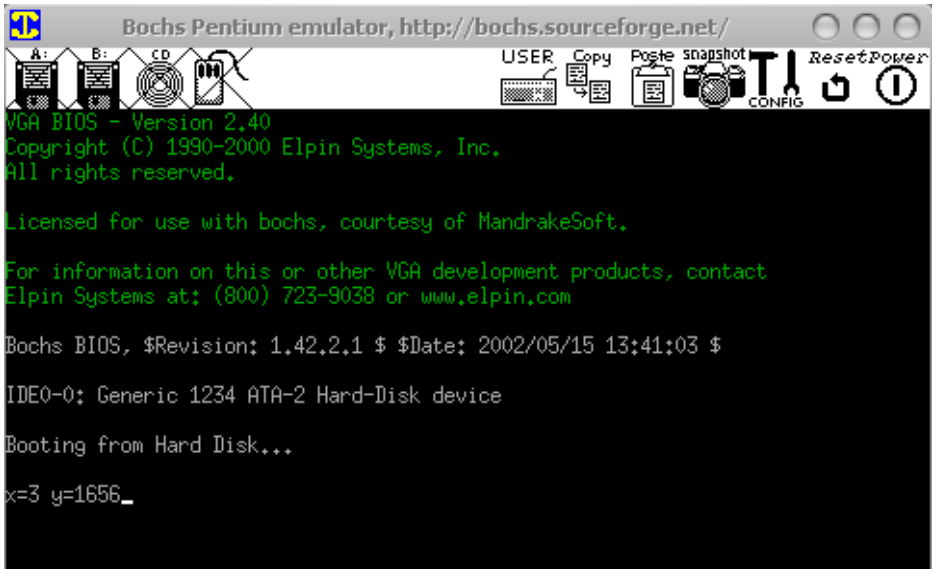


图 3-22 cprintf 函数打印示例图 4

可以看到，显示出来 y=1656，是个随机的数字，这是因为可变参数只有一个，而可变参数指针指向的是这一个参数存放的位置，当函数试图去在内存中寻找不存在的第二个参数的时候便会在内存中存放第一个参数之后的位置中去取，而这个位置存放的内容我们无法确定，因此打印出来的便是一个随机的数字。

3.3 JOS 内核

通过 3.1 节的讲解我们得知了内核可执行文件是如何被加载到内存中的，而在 Boot Loader 完成了它的工作之后便会用((void (*)(void)) (ELFHDR->e_entry & 0xFFFFFF))()这样一句代码(见 boot/main.c 文件)执行指令的跳转，由于在 JOS 中规定内核可执行文件是存放在内存中从 0x100000 开始的一段区域内，于是在跳转的时候只取入口地址的低 24 位，这样就可以避免跳转到物理内存中并不存在高地址处。然而由于内核的链接地址是 0xf0100000，这样在内核代码中势必有很多的指针的值是大于 0xf0100000 的，那么系统是如何让这些指针在寻址的时候能够找到物理内存中正确的位置的呢？

从 1.1 节保护模式简介中我们了解到在保护模式下物理地址=GDT 表项中段基址+偏移地址，而实际上指针的值通常就是偏移地址，那么段基址是多少呢？当系统进入保护模式后我们可以在 Bochs 中使用“info gdt”这样的命令来查看 GDT 表的内容，而我们应该还记得在 Boot Loader 刚开始执行的时候是首先将系统从实模式转换到保护模式，于是我们首先通过设置断点和使用“info gdt”命令来查看一下这个时候的 GDT 表项。

```
<bochs:69>
Next at t=199871
(0) [0x00007c53] 0008:00007c53 (unk. ctxt): mov ES, AX                ; 8ec0
<bochs:70> info gdt
Global Descriptor Table (0x00007c68):
GDT[0x00]=??? descriptor hi=00000000, lo=00000000
GDT[0x01]=Code segment, linearaddr=00000000, len=fffff * 4Kbytes, Execute/Read, 32-bit addrs
GDT[0x02]=Data segment, linearaddr=00000000, len=fffff * 4Kbytes, Read/Write, Accessed
You can list individual interrupts with 'info gdt NUM'.
<bochs:71>
```

图 3-23 Boot Loader 执行时的 GDT 表项

如图 3-23 所示在这个时候 GDT 的表项有两个是有效的，其中一个是标识代码段的，另

一个是标识数据段的，可以看到 linearaddr 这一项都是 0，即代表段基址是 0。于是在这个时候偏移地址实际上就等于物理地址。我们再来看看在进入内核后 GDT 表有没有什么变化。

```
Next at t=23405926
(0) [0x0010080c] 0008:f010080c (unk. ctxt): jmp f010090c              ; ebfe
<bochs:2> info gdt
Global Descriptor Table (0x0010f000):
GDT[0x00]=??? descriptor hi=00000000, lo=00000000
GDT[0x01]=Code segment, linearaddr=10000000, len=fffff * 4Kbytes, Execute/Read, 32-bit addrs
GDT[0x02]=Data segment, linearaddr=10000000, len=fffff * 4Kbytes, Read/Write, Accessed
You can list individual interrupts with 'info gdt NUM'.
<bochs:3>
```

图 3-24 进入内核后的 GDT 表项

可以看到这个时候显示 linearaddr=10000000，由于在内核执行过程中偏移地址都是大于 0xf0100000 的，于是在加上 0x10000000 后便造成了高位的进位，但是由于地址最多就只有 32 位，于是实际上高位的进位就被舍去了，这样一来，最后物理地址便会是内存的低位，在 0x100000 附近的区域，而不会寻址到 0xf0100000 这样的不存在的高位物理地址空间。

在弄清楚了这个问题后我们便开始深入 JOS 内核做进一步的了解。

1. 部分内核代码解析

我们首先来看看 kern/entry.S 文件，因为在内核被加载到内存后系统便立即跳转开始执行 kern/entry.S 的代码，这个文件中的程序也就相当于内核的入口程序，下面我们就来详细分析一下这段代码中的关键部分。这里我们只列出了部分核心的代码，读者可以自己去阅读 kern/entry.S 文件来了解其它的部分。

```
    lgdt    RELOC(mygdtdesc)          # 加载新的 GDT 表
    movl    $DATA_SEL, %eax           # 数据段选择子
    movw    %ax,%ds                   # 初始化数据段选择子
    movw    %ax,%es                   #  初始化附加段选择子
    movw    %ax,%ss                   #  初始化堆栈段选择子
    ljmp    $CODE_SEL,$relocated          # 通过跳转重新加载 cs 寄存器
relocated:
    movl    $0x0,%ebp                 # 将 ebp 寄存器清零
    movl    $(bootstacktop),%esp      # 设置堆栈指针
    call    i386_init                 # 调用 kern/init.c 中的函数
spin:   jmp  spin                     #无限循环跳转，正常情况下永远不会执行这条语句
```

入口程序主要完成这几个功能。首先它加载了新的 GDT 表，由之前所讲的我们可以得知进入内核后新的 GDT 表的表项中的代码段和数据段的段基址都是 0x10000000。紧接着入口程序初始化了几个常用的段寄存器 ds、es、ss 以及与堆栈相关的寄存器 ebp 和 esp，esp 通常指向的是栈顶，在这里 bootstacktop 代表的是内核中一个临时的堆栈的顶部，细节我们可以从下面的这段代码中的得知，这段代码也是在 kern/entry.S 中。

```
    .p2align   PGSHIFT          # 强制 4K 字节对齐
    .globl          bootstack
bootstack:
    .space          KSTKSIZE
    .globl          bootstacktop
bootstacktop:
```

可以看到在这里定义了两个全局变量 bootstack 和 bootstacktop，bootstack 标识了内存中的一个位置，表示从这里开始的 KSTKSIZE 个字节的区域都是属于这个临时堆栈的(KSTKSIZE 在 inc/memlayout.h 中定义为 32k)，而 bootstacktop 则指向的是这段区域后的第一个字节，由于刚开始的时候堆栈是空的，所以栈顶便是 bootstacktop 所指向的位置，于是程序便将 bootstacktop 的值赋给了 esp 寄存器。关于堆栈我们在之后会做更详细的讨论。

在初始化堆栈指针后，程序调用了 i386_init 函数，这个函数是在 kern/init.c 中，它会对内核进行一系列的初始化，接下来我们会详细的讨论这个函数。

```
void i386_init(void)
{
    extern char edata[], end[];
    memset(edata, 0, end - edata); // 初始化 bss 节
    cons_init();   // 初始化控制台
    cprintf("6828 decimal is %o octal!\n", 6828); // 测试打印
    test_backtrace(5);   // 测试函数
    while (1)
        monitor(NULL);
}
```

首先我们可以看到两个外部字符数组变量 edata 和 end，其中 edata 表示的是 bss 节在内存中开始的位置，而 end 则是表示内核可执行程序在内存中结束的位置。由 2.1 节中对 ELF 文件的讲解我们可以知道 bss 节是文件在内存中的最后一部分，于是 edata 与 end 之间的部分便是 bss 节的部分，我们又知道 bss 节的内容是未初始化的变量，而这些变量是默认为零的，所以在一开始的时候程序要用 memset(edata, 0, end - edata)这句代码将这些变量都置为零。

在初始化 bss 节后，接下来程序调用了 cons_init 函数，这个函数的原型是在 kern/console.c 中，这个函数的功能包括一系列的系统初始化，有显存的初始化、键盘的初始化之类的，在这里我们不需要对此有太详细的了解。

接着程序调用了 cprintf 函数用 8 进制的形式打印一个 10 进制的数，当我们刚开始做 lab1 的实验的时候 2.2 节中所讲的 vprintfmt 函数中的关于打印 8 进制数的部分还并没有实现，所以这个时候会打印出如下的结果：

6828 decimal is XXX octal!

test_backtrace 函数是通过堆栈来对函数调用进行回溯，这在后面讲堆栈的时候我们会对其进行详细的解析。

最后程序无限循环的调用了 monitor 函数，这个函数的原型在 kern/monitor.c 中，它的功能是提示用户输入命令与操作系统进行交互。

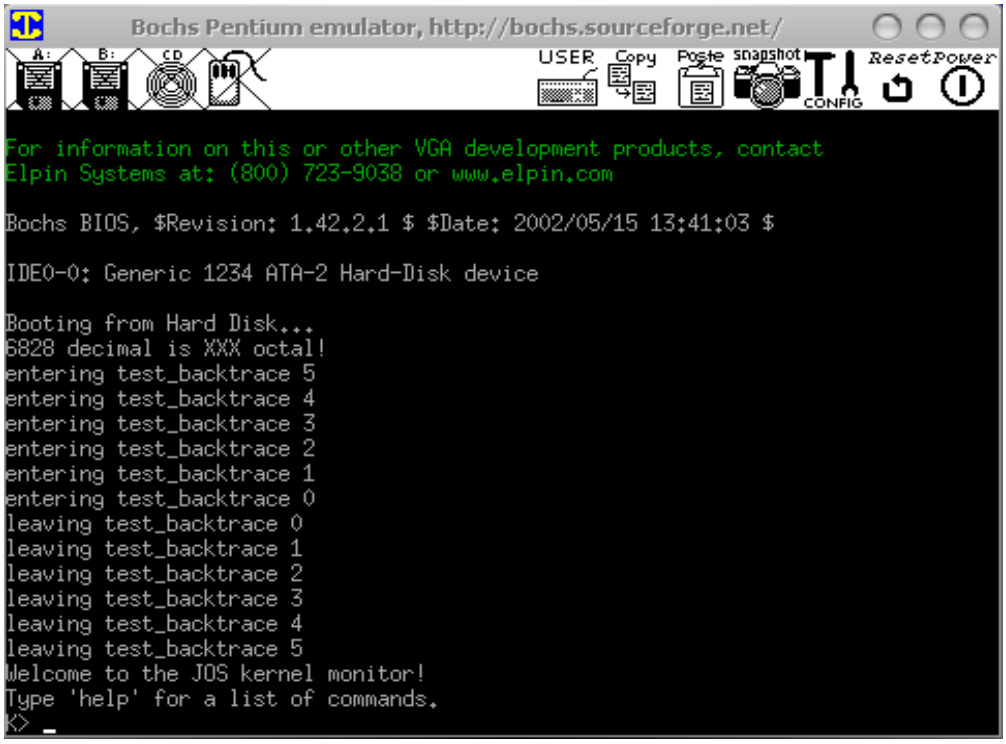


图 3-25

如图 3-25 所示，JOS 启动后便会打印上面这些内容，在出现“k>”后用户便可以输入命令，目前可以使用的命令只有两个，分别是“help”与“kerninfo”。下面我们来讲讲用户命令的实现原理。

用户命令是由 kern/monitor.c 中的 monitor 函数实现的，在讲命令式如何实现之前我们要先讲一下定义在 kern/monitor.c 中的结构体 Command，下面是定义的原型：

```
struct Command {
    const char *name;   // 命令名字
    const char *desc;   // 命令作用
    int (*func)(int argc, char** argv, struct Trapframe* tf);   // 命令相关函数指针
};
```

在这里 name 成员变量便是用户需要输入的命令名字，比如像“help”。成员变量 desc 是指向一个字符串的指针，这个字符串显示了这个命令的作用。当我们使用 help 命令时便会显示每个命令的作用，如下图所示：

图 3-26

可以看到这个时候我们仅有的两个命令“help”与“kerninfo”，其中“help”的作用是显示所有可用的命令，而“kerninfo”则是显示内核的信息。另外第三个成员变量命令相关函数指针则是指向处理这个命令的函数。

实际上，我们需要在 kern/monitor.c 中为每一个用户能够使用的命令定义一个 Command 结构体变量，就像如下的代码所示：

```
static struct Command commands[] = {
    { "help", "Display this list of commands", mon_help },
```

```
    { "kerninfo", "Display information about the kernel", mon_kerninfo },
};
```

所有的命令构成了一个 Command 结构体数组，当我们需要增加某个命令时便可以在这个数组中添加相应的数组项。

接下来我们便来讲一下具体 monitor 函数是如何响应用户输入的一个命令的，下面便是函数的原型：

```
void monitor(struct Trapframe *tf)
{
    char *buf;
    cprintf("Welcome to the JOS kernel monitor!\n");
    cprintf("Type 'help' for a list of commands.\n"); //  打印命令行信息
    while (1) {
        buf = readline("K> "); //  等待用户输入命令
        if (buf != NULL)
            if (runcmd(buf, tf) < 0) //  处理命令
                break;
    }
}
```

在这里，readline 函数的功能是等待用户输入一个命令字符串，当用户敲“回车”键时便代表命令输入结束，指针 buf 指向的便是这个输入的字符串存放的位置，接着程序便开始处理命令。runcmd 是专门用来处理命令字符串的一个函数，它有两个参数，第一个 buf 是指向命令字符串的指针，第二个参数 tf 在这里我们暂时不做讨论。

runcmd 函数也是定义在 kern/monitor.c 中的，接下来我们分几个部分来讨论这个函数。

```
static int runcmd(char *buf, struct Trapframe *tf)
{
    int argc;
    char *argv[MAXARGS];
    int i;
    argc = 0;
    argv[argc] = 0;
```

首先函数初始化了几个变量，argv 是一个指针数组，每个数组项指向一个字符串。因为一个命令分为命令名字以及命令参数，所以一个命令字符串一般可以分为好几个子字符串，于是每个指针数组项就指向一个子字符串。argc 代表命令参数的个数，MAXARGS-2 代表一个命令容许的参数的最多的个数。

```
while (1) {
        while (*buf && strchr(WHITESPACE, *buf))
            *buf++ = 0; // 把所有空格字符都置为空字符
        if (*buf == 0)
            break;   // 命令结束
        if (argc == MAXARGS-1) {
            cprintf("Too many arguments (max %d)\n", MAXARGS);
            return 0; // 参数个数超过最大个数的限制
        }
        argv[argc++] = buf; // 指向相应的字符串
        while (*buf && !strchr(WHITESPACE, *buf))
            buf++; // 跳过非空格的字符
    }
    argv[argc] = 0;
```

这段的功能是让指针指向了每个子字符串并且把命令字符串中的空格都换成了空字符，因为用户在输入命令的时候，命令名和参数之间，参数和参数之间都是由空格相隔开的，这样处理后每个子字符串的结尾便都是一个空字符，便可以方便之后读取这个字符串，就如下图所示：

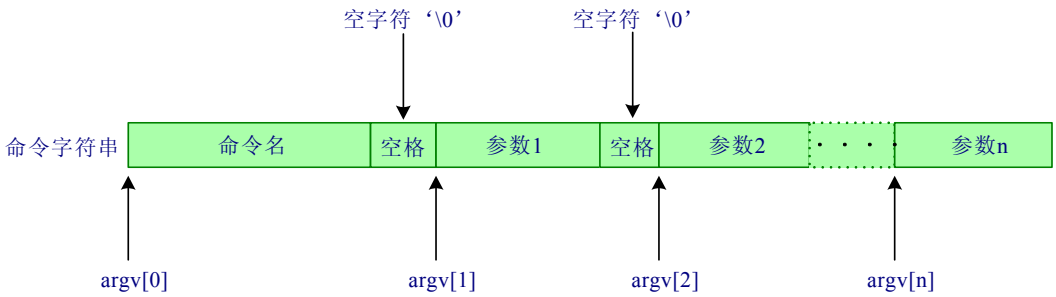


图 3-27 命令字符串的读取

从上图可以看出在这段代码执行完后 argv 指针数组以及命令字符串的状态。此时变量 argc 的值等于 n+1，也就是子字符串的个数，有了 argv 与 argc 这两个参数后，程序便可以开始处理该命令。

```
if (argc == 0)
        return 0;     // 当没有命令时返回
    for (i = 0; i < NCOMMANDS; i++) {
        if (strcmp(argv[0], commands[i].name) == 0)
            return commands[i].func(argc, argv, tf);
    }// 在所有可以执行的命令中寻找与输入的命令名相同的命令，并将 argc 与 argv 当做命令函数的参数。
    cprintf("Unknown command '%s'\n", argv[0]);
    return 0; // 无法识别的命令名
}
```

这段程序的作用就是用命令函数处理相应的命令，比如说假如用户输入了“help”命令，则 kern/monitor.c 中的 mon_help 函数便会用来处理这个命令。

2. 堆栈解析

在数据结构中我们都应该学过堆栈这个概念，先进后出是堆栈的特点，在我们常规的理解下，栈顶应该是在内存的高地址处，而栈底是在低地址处，然而实际上在内存中却恰恰相反，从下图中我们就可以看出来。

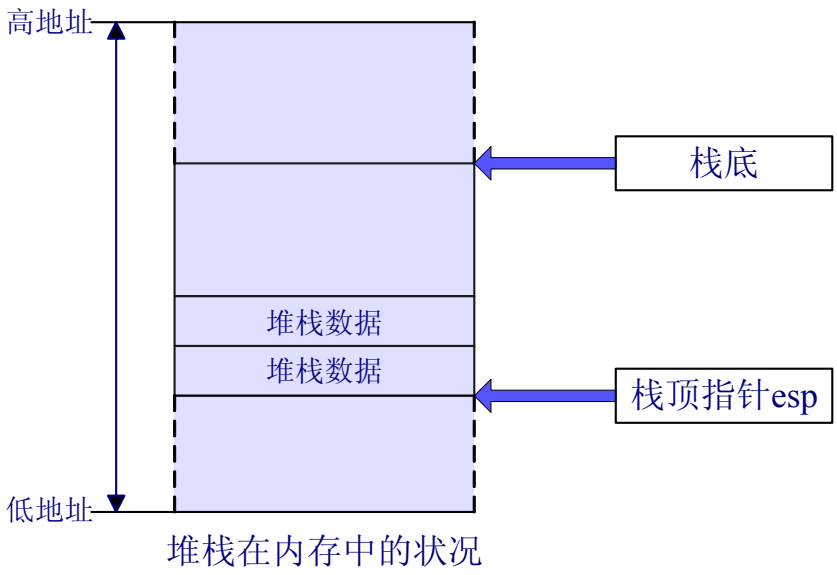


图 3-28 堆栈原理

在这里，栈底是固定的，栈顶是可以变化的，下面我们来看看出栈和进栈是什么样子的状况。

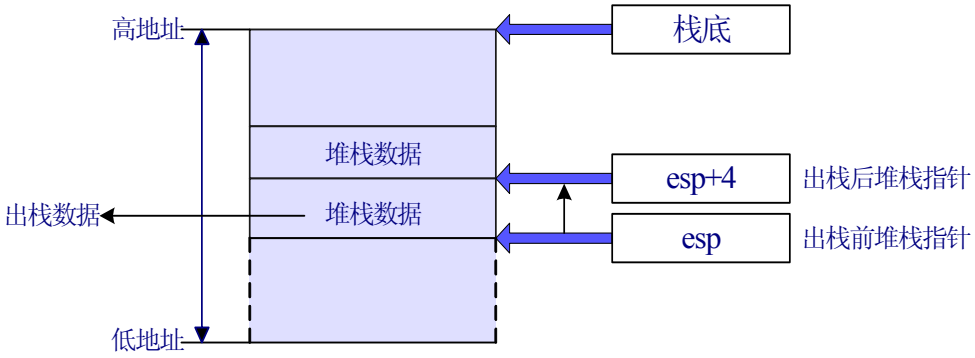


图 3-29 堆栈出栈一个字的数据的状况

假设我们需要堆栈出栈一个字的数据，即 4 个字节，这个时候我们需要当前 esp 指向位置开始的 4 个字节读出来，并且在这之后把 esp 加 4。

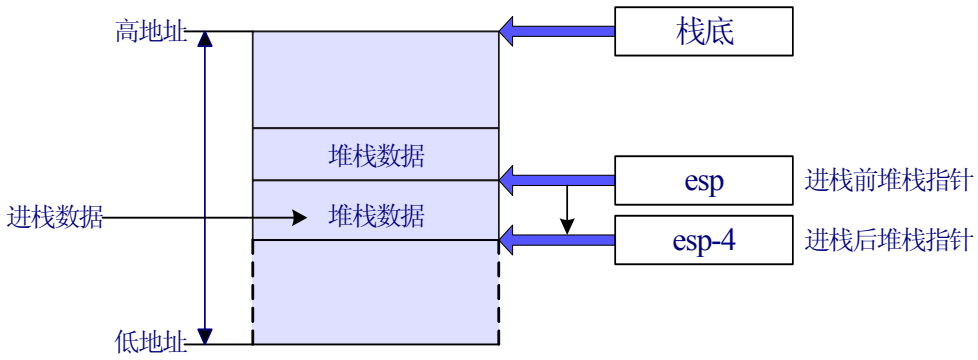


图 3-30 堆栈进栈一个字的数据的状况

进栈则是一样的道理，当需要进栈一个字的数据时，将 esp 减 4，并把这个字存放在 esp 指向位置开始的 4 个字节处。

在讲完堆栈的原理后我们来看看 JOS 操作系统的堆栈，在之前的讲解中我们可以了解到在刚进入内核的时候程序定义了一个暂时的堆栈，这个堆栈的大小为 32k，而且刚开始的时候堆栈为空，栈顶指针 esp 是指向栈底。这个堆栈的栈底的虚拟地址实际上是大于 0xf0000000 的，然后在第三章中我们会了解到无论是内核的堆栈还是用户的堆栈的虚拟地址都应该是小于 0xf0000000 的。

在内核初始化函数 i386_init 中调用了一个 test_backtrace 函数，这个函数的作用是与测试系统的堆栈有关，下面我们就来研究一下这个函数的作用。

```
void test_backtrace(int x)
{
    cprintf("entering test_backtrace %d\n", x);
    if (x > 0)
        test_backtrace(x-1);
    else
        mon_backtrace(0, 0, 0);
    cprintf("leaving test_backtrace %d\n", x);
}
```

可以看到，这个函数运用了递归的调用，test_backtrace 函数自己不断的调用自己，其中嵌套的层次是由参数 x 决定的，最终在嵌套的最里层的函数调用了 mon_backtrace 函数，而在这里这个 mon_backtrace 函数是由读者自己去完成的，它的作用就是通过系统的堆栈区追溯函数一层一层的调用，在完成这个函数之前我们首先来了解一下程序是如何利用通过堆栈来完成函数的调用与返回的。

首先我们了解一下于这个过程相关的几个关键的寄存器。eip 存储当前执行指令的下一条指令在内存中的偏移地址，esp 存储指向栈顶的指针，而 ebp 则是存储指向当前函数需要使用的参数的指针。在程序中，如果需要调用一个函数，首先会将函数需要的参数进栈，然后将 eip 中的一个字进栈，也就是下一条指令在内存中的位置，这样在函数调用结束后便可以通过堆栈中的 eip 值返回调用函数的程序。而在一进入调用函数的时候，第一件事便是将 ebp 进栈，然后将当前的 esp 的值赋给 ebp，这样此时 ebp 便指向了堆栈中存储 ebp、eip 和函数参数的地方，所以 ebp 通常都是指向当前函数所需要的参数，相当于每个函数都有自己的一个 ebp，所以当一个函数在内部调用另一个函数的时候，被调用函数执行时的 ebp 的值

指向调用它的函数的 ebp 值存放的位置。

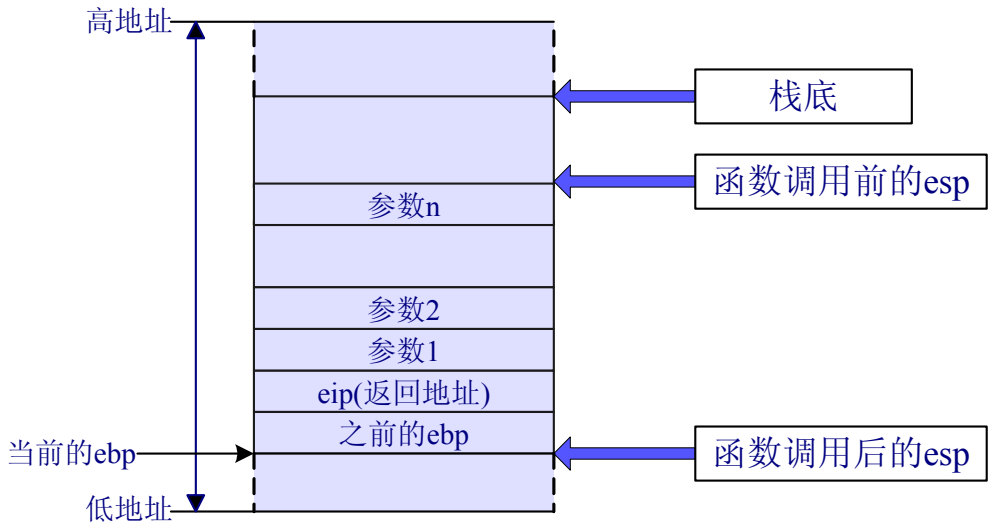


图 3-31 调用函数时堆栈的变化

从上图中我们可以清楚的看到各个重要数据在内存中的位置，既然每个执行的函数都有自己的一个 ebp 值，而这个 ebp 作为指针又是指向调用该函数的函数的 ebp 值存放的位置，所有当 test_backtrace 函数不断的自己调用自己形成几层的嵌套的时候我们便可以通过 ebp 来进行函数调用的回溯,这也就是 mon_backtrace 函数所要干的事情。

有了上面所讲的基础，下面我们就来具体研究一下 mon_backtrace 函数。在 lab1 中这个函数是需要我们自己去完成的，我们首先来看看实验要求我们 mon_backtrace 函数完成什么样的输出：

Stack backtrace:

ebp f0109e58 eip f0100a62 args 00000001 f0109e80 f0109e98 f0100ed2 00000031

ebp f0109ed8 eip f01000d6 args 00000000 00000000 f0100058 f0109f28 00000061

可以看出在回溯的过程中我们需要打印出每个函数的 ebp 值、程序返回地址以及它的参数。我们知道 mon_backtrace 函数是在 test_backtrace 函数中调用的，当调用 mon_backtrace 时，实际上 test_backtrace 已经自己调用自己，自身嵌套了 6 次，那么 mon_backtrace 函数此时是嵌套在最里面的，这个时候理所当然 mon_backtrace 在执行的时候会有自己的 ebp，只要我们得到了这个 ebp 值，我们就可以按照之前所讲的对外层嵌套的函数进行回溯了。

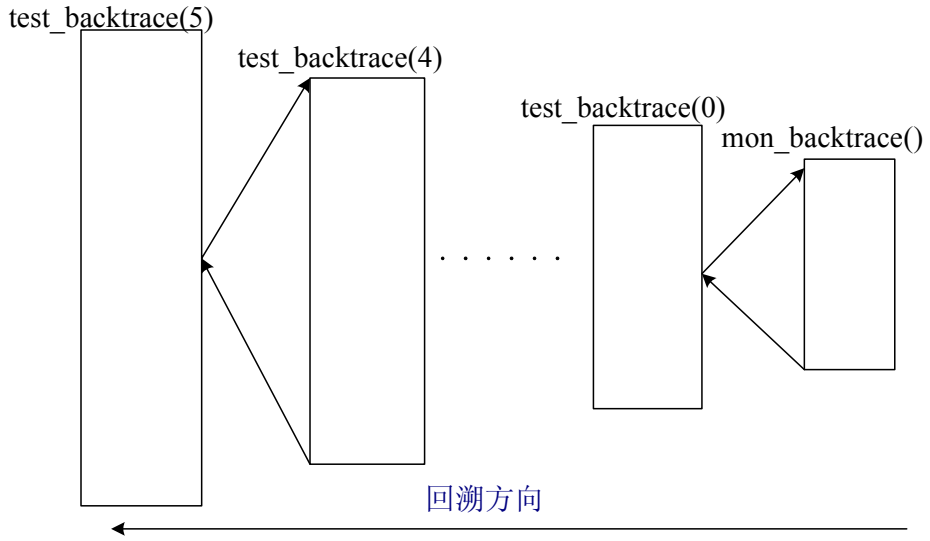


图 3-32 函数的嵌套

为了在程序中能得到 ebp 的值，我们需要使用 read_ebp 这个函数，这个函数的功能是利用一些较低层的指令将当前 ebp 寄存器中的值读出来，函数的返回值便是 ebp 的值，于是我们得到这个关键的 ebp 的值后便可以按照之前所讲的利用循环实现 mon_backtrace 函数的功能了。以下便是完成后的 mon_backtrace 函数的代码。

```
int mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
        uint32_t bp,ip,arg1,arg2,arg3,i;
        bp = read_ebp();      // 读取 ebp 值
        ip = *((uint32_t*)bp+1); // 从 ebp 指向的堆栈位置读取函数调用返回地址
        arg1 = *((uint32_t*)bp+2);
        arg2 = *((uint32_t*)bp+3);
        arg3 = *((uint32_t*)bp+4);// 从 ebp 指向的堆栈位置读取函数的参数
        do{
                cprintf("ebp %x eip %x args %x %x %x %x\n",bp,ip,arg1,arg2,arg3);// 按实验要求打印信息
                bp = *(uint32_t *)bp;// 读取外层函数的 ebp
                if (bp != 0){
                        ip = *((uint32_t*)bp+1);
                        arg1 = *((uint32_t*)bp+2);
                        arg2 = *((uint32_t*)bp+3);
                        arg3 = *((uint32_t*)bp+4);
                }
        }while(bp != 0);// 循环到最外层的程序位置
        return 0;
}
```

可以看到程序就是利用 ebp 的值来进行循环，最后当发现 ebp 的值为 0 时便停止循环。因为最外层的程序是 kern/entry.S 中的入口程序，记得在之前我们看到过入口程序中有一句代码是“movl $0x0,%ebp”，也就是说在入口程序调用 i386_init 函数之前便把 ebp 的值置

为 0，也就是说入口程序的 ebp 实际上为 0，ebp 代表的是指针，因为在刚进内核的时候指针的值都必须要大于 0xf0000000 才行，所以指针值为 0 是没有意义的，于是当循环到发现 ebp 值为 0 时便可以停止循环了。

图 3-11 显示了我们 mon_backtrace 函数的运行结果，在这里我们对每一个函数输出 3 个参数，而在这里我们一共输出了 8 个函数的信息，其中包括 test_backtrace 的 6 次嵌套，以及 mon_backtrace 函数与 i386_init 函数，而最外层的入口程序由于它的 ebp 值是无意义的所以我们没有打印它的信息。

当完成了 mon_backtrace 函数后，lab1 的工作实际上也就完成了，在下一章我们将要讲解 lab2 的内容，也就是有关内存管理的原理。

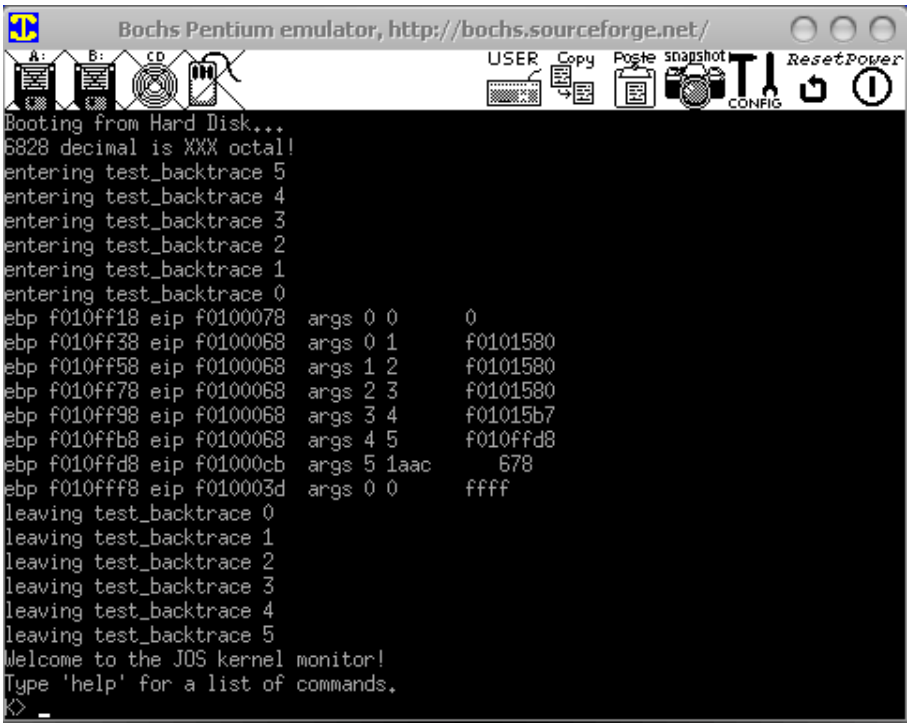


图 3-33