DoggyOS 操作系统

目录

摘要

DoggyOS 是一个原型操作系统，有进程管理模块，内存管理模块，启动加载模块和 I/O 输入输出模块等组成。DoggyOS 原型系统秉着基于模仿，勇于创新的原则，在 Tinix[1]的基础之上，实现了多任务多用户的运行环境，为用户提供了类 linux 的文本交互界面，现在已经支持 ls,ll,ps,kill,lu,who,quit,clean 八个命令，同时 DoggyOS 还提供了一定数量的系统调用，支持应用程序的开发。

DoggyOS 主要由 c 语言和 nasm 汇编语言编写完成，在 bochs 和 qemu+gdb 的环境下进行调试。我们的工作主要有以下几个部分组成，独立完成了应用程序的动态加载功能、完整的内存管理模块、简单的 shell 系统、实用的内存文件系统，在 Tinix 的基础之上修改完善了启动加载过程、内核启动初始化过程、进程调度算法、系统调用过程、I/O 功能，同时添加了某些中断响应函数。

下面将分几个章节，分别介绍 DoggyOS 的启动加载，内核初始化，内存管理，进程管理和 I/O 系统。

关键词：进程管理、内存管理、内核初始化、引导启动、I/O 输入输出

第1章 引导启动及内核初始化

1.1. 概述

本章主要描述 boot/目录中的两个汇编代码文件，见表格 1.1 所示。这两个汇编文件采用的是 NASM 汇编语法，可能大部分人开始学汇编时用的是 MASM，其实 NASM 的格式跟 MASM 总体上是差不多的。

表 1-1 doggyos\boot\目录

文件名	长度（字节）	最后修改时间
boot.asm	9,429 字节 20	07 年 12 月 24 日, 9:38:19
loader.asm	30,633 字节 20	07 年 12 月 24 日, 9:38:19

这里总体说明一下操作系统启动部分的主要执行流程。当 PC 的电源打开后，80X86 结构的 CPU 将自动进入实模式，并从地址 0xFFFF0 开始自动执行程序代码，这个地址通常是 ROM-BIOS 中的地址。PC 机的 BIOS 将执行某些系统的检测，并在物理地址 0 处开始初始化中断向量。此后，它将可启动设备的第一个扇区（磁盘引导扇区，512 字节）读入内存绝对地址 0x7C00 处，并跳转到这个地方。启动设备通常是软驱或是硬盘。这里的叙述是非常简单的，但这已经足够理解内核初始化的工作过程了。

Doggyos 的最前面部分是用 nasm 编写的 boot/boot.asm，它将由 BIOS 读入到内存绝对地址 0x7C00（31KB）处，当它被执行时就会把 Loader.bin 加载到绝对地址 0x90000（576KB）处。然后把控制权交给 Loader.bin，由他来负责加载 Kernel.bin 和 Application，加载好以后并进入保护模式，然后把 Application 移到绝对地址 0x100000 处（1M），因为实模式下只能访问 1M 地址空间，所以只能到进入保护模式以后才能移动 Application。至于为什么要把 Application 也加载到内核中来的原因是在保护模式下，我们的操作系统还没有实现读取软盘的接口，所以只能把要运行的程序都统一加载到内存中，然后直接在内存中访问执行。之后内核解压到 0x30000 处，并开始执行内核。图 1 清晰地显示出 Doggyos 系统启动时这几个程序在内存中的动态位置。其中每一竖条框代表某一时刻内存中各程序的映像位置图。在系统加载时将显示信息“Loading…”。

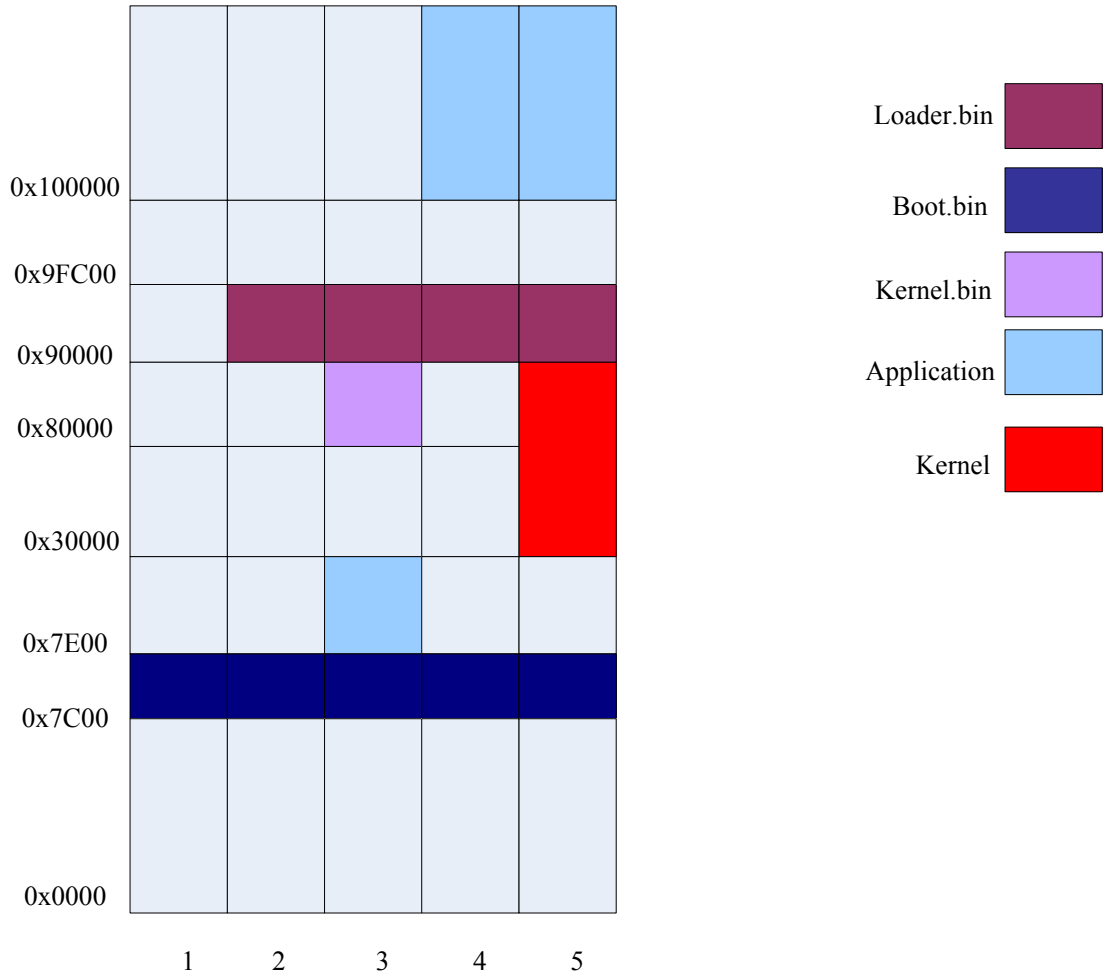


图 1-1 启动引导时内核在内存中的位置和移动后的位置情况

1.2. Boot.asm 程序

1.2.1. 功能描述

在内核开始执行之前不但要加载内核，而且还要准备保护模式等一系列工作，如果全部交给引导扇区来做，512 字节和可能是不够用的，所以，不妨把这个过程交给另外的模块来完成，我们把这个模块叫做 Loader，引导扇区负责把 Loader 加载入内存，并把控制权交给它，其他工作放心地交给 Loader 来做，因为它没有 512 字节的限制，将会灵活得多。所以 boot.asm 这个汇编程序的功能就是把 Loader 加载到内存的指定地址，它就完成了使命，然后把控制权交给 Loader，由它来负责后续的初始化和引导工作。

Boot 代码是磁盘引导块程序，驻留在磁盘的第一个扇区中（引导扇区，0 磁道（柱面），0 磁头，第一个扇区）。1.44MB 软盘上的数据分布如图 2 所示。在 PC 机加电 ROM BIOS 会把引导扇区代码 Boot.bin 加载到内存地址 0x7C00 开始处执行之。该程序的主要作用是把 Loader.bin 从软盘中加载到内存的 0x90000 处，接着在屏幕上显示“Booting…”字符串。最后跳转到 Loader，向 Loader 交出控

制权。

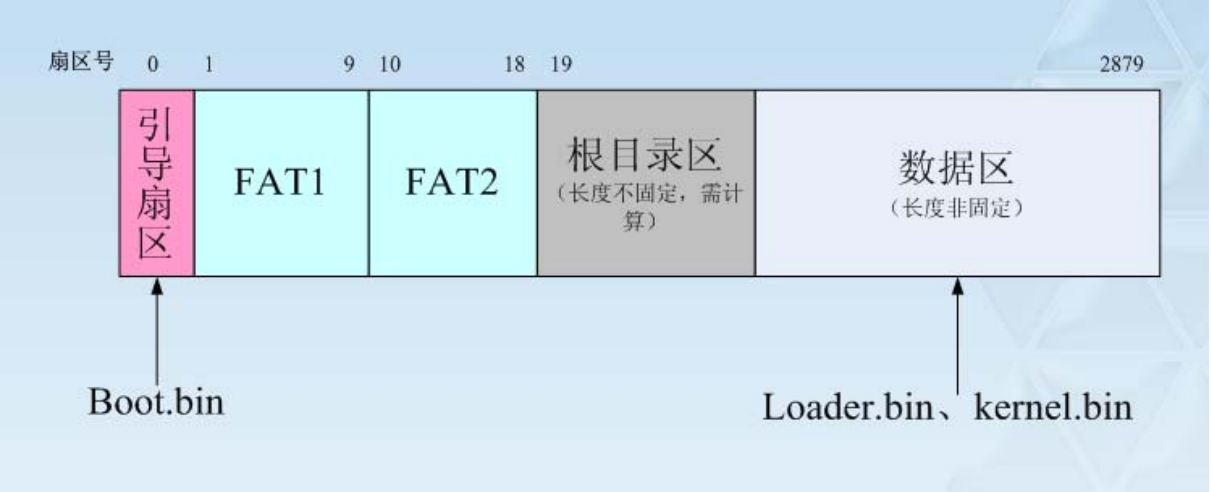


图 1-2 内核在 1.44MB 软盘上的分布

1.2.2. 加载 Loader 入内存

要加载一个文件入内存的话，免不了要读软盘，这时候就用到了 BIOS 中断 int 13h 。它的用法如表 1-2 所示。

表 1-2 BIOS 中断 int 13h 的用法

中断号	寄存器		作用
13h	ah=00h d	l=驱动器号(0 表示 A 盘)	复位软驱
	ah=02h ch=柱面(磁道)号 dh=磁头号 es:bx->数据缓冲区	al=要读扇区数 cl=起始扇区号 dl=驱动器号(0 表示 A 盘)	从磁盘读入数据 es:bx 指向的缓冲区

我们看到，中断需要的参数不是原来提到的从第 0 扇区开始的扇区号，而是柱面号、磁头号以及在当前柱面上的扇区号 3 个分量，所以需要我们自己来转换一下。对于 1.44MB 的软盘来讲，总共有两面（磁头号 0 和 1），每面 80 个磁道（磁道号 0～79），每个磁道有 18 个扇区（扇区号 1～18）。下面的共识就是软盘容量的由来：

$$2 \times 80 \times 18 \times 512 = 1.44\text{MB}$$

于是，磁头号、柱面（磁道）号和起始扇区号可以用图 1.3 所示的方法来计算。

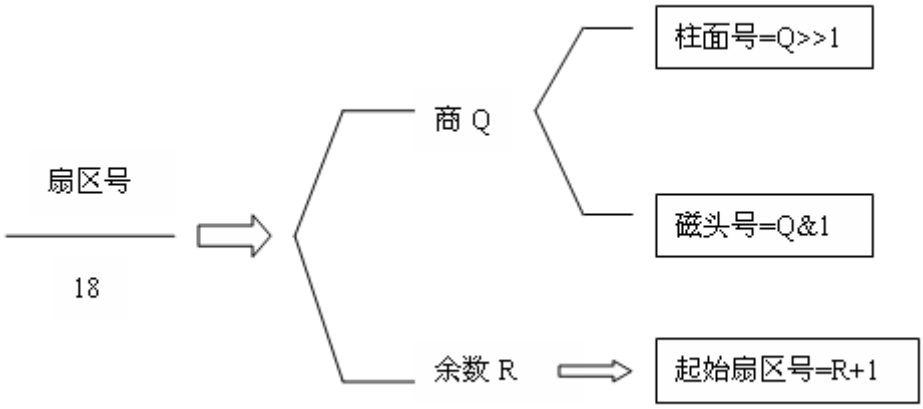


图 1-3 磁头号、柱面号、起始扇区号的计算方法图示

有了这些预备知识以后，我们可以来实现读软盘扇区的函数了：

```
//-------------------------------------------------------------------------
// 函数名: ReadSector
//-------------------------------------------------------------------------
// 作用:
// 从第 ax 个 Sector 开始, 将 cl 个 Sector 读入 es:bx 中
ReadSector:
    // ---------------------------------------------------------------------
    // 怎样由扇区号求扇区在磁盘中的位置(扇区号-> 柱面号, 起始扇区, 磁头号)
    // ---------------------------------------------------------------------
    // 设扇区号为 x
    //                          ┌柱面号= y >> 1
    //          x               ┌商 y ┤
    // ---------- => ┤          └磁头号= y & 1
    //  每磁道扇区数            |
    //                          └余 z => 起始扇区号= z + 1
    push bp
    mov bp, sp
    sub  esp, 2                 // 辟出两个字节的堆栈区域保存要读的扇区数: byte [bp-2]
    mov byte [bp-2], cl
    push bx                     // 保存 bx
    mov bl, [BPB_SecPerTrk]     // bl: 除数
    div  bl                     // y 在 al 中, z 在 ah 中
    inc  ah                     // z ++
    mov cl, ah                  // cl <- 起始扇区号
    mov dh, al                  // dh <- y
    shr  al, 1                  // y >> 1 (其实是 y/BPB_NumHeads, 这里 BPB_NumHeads=2)
    mov ch, al                  // ch <- 柱面号
```

```
    and  dh, 1                  // dh & 1 =  磁头号
    pop  bx                     //  恢复 bx
    //  至此, "柱面号, 起始扇区, 磁头号"  全部得到^^^^^^^^^^^^^^^^^^^^^^^^
    mov dl, [BS_DrvNum]     //  驱动器号(0  表示 A  盘)
.GoOnReading:
    mov ah, 2
    mov al, byte [bp-2]         //  读 al  个扇区
    int   13h
    jc    .GoOnReading          //  如果读取错误 CF  会被置为 1, 这时就不停地读, 直到正确为止
    add  esp, 2
    pop  bp
    ret
```

在实现了读扇区的函数之后，下面开始实现在软盘中寻找 Loader.bin 代码。

```
 .......
    xor   ah, ah     // ┐
    xor   dl, dl     //  ├软驱复位
    int   13h        // ┘
//  下面在 A  盘的根目录寻找 LOADER.BIN
    mov word [wSectorNo], SectorNoOfRootDirectory
LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
    cmp word [wRootDirSizeForLoop], 0    // ┐
    jz    LABEL_NO_LOADERBIN             //  ├判断根目录区是不是已经读完
    dec  word [wRootDirSizeForLoop]      // ┘ 如果读完表示没有找到 LOADER.BIN
    mov ax, BaseOfLoader
    mov es, ax                  // es <- BaseOfLoader
    mov bx, OffsetOfLoader // bx <- OffsetOfLoader  于是, es:bx = BaseOfLoader:OffsetOfLoader
    mov ax, [wSectorNo]     // ax <- Root Directory  中的某 Sector  号
    mov cl, 1
    call  ReadSector

    mov si, LoaderFileName // ds:si -> "LOADER    BIN"
    mov di, OffsetOfLoader  // es:di -> BaseOfLoader:0100 = BaseOfLoader*10h+100
    cld
    mov dx, 10h
LABEL_SEARCH_FOR_LOADERBIN:
    cmp dx, 0                                       // ┐ 循环次数控制,
    jz    LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR //  ├如果已经读完了一个 Sector,
    dec  dx                                         // ┘ 就跳到下一个 Sector
    mov cx, 11
LABEL_CMP_FILENAME:
    cmp cx, 0
    jz    LABEL_FILENAME_FOUND                  //  如果比较了 11  个字符都相等, 表示找到
```

```
dec  cx
     lodsb                              // ds:si -> al
     cmp al, byte [es:di]
     jz     LABEL_GO_ON
     jmp  LABEL_DIFFERENT               // 只要发现不一样的字符就表明本 DirectoryEntry 不是
//  我们要找的 LOADER.BIN
LABEL_GO_ON:
     inc   di
     jmp  LABEL_CMP_FILENAME            // 继续循环

LABEL_DIFFERENT:
     and  di, 0FFE0h                    // else ┐  di &= E0  为了让它指向本条目开头
     add  di, 20h                       //      │
     mov si, LoaderFileName             //      ├ di += 20h   下一个目录条目
     jmp  LABEL_SEARCH_FOR_LOADERBIN//       ┘

LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:
     add  word [wSectorNo], 1
     jmp  LABEL_SEARCH_IN_ROOT_DIR_BEGIN

LABEL_NO_LOADERBIN:
     mov dh, 2                          // "No LOADER."
     call   DispStr                     // 显示字符串
%ifdef     _BOOT_DEBUG_
     mov ax, 4c00h                      // ┐
     int    21h                         // ┘ 没有找到 LOADER.BIN, 回到 DOS
%else
     jmp  $                             // 没有找到 LOADER.BIN, 死循环在这里
%endif
LABEL_FILENAME_FOUND:                   // 找到 LOADER.BIN  后便来到这里继续
```

这段代码看上去稍微有一点复杂，但是逻辑是很清晰的，就是遍历根目录区所有的扇区，将每一个扇区加载入内存，然后从中寻找文件名为 Loader.bin 的条目，知道找到为止。找到的那一刻，es:di 是指向条目中字母 N 后面的那个字符。

等确信找到文件这一模块是正确的后，我们就开始下面的工作——将 Loader.bin 加载入内存。现在我们已经有了 Loader.bin 的起始扇区号，我们需要用这个扇区号来做两件事：一件是把起始扇区装入内存，另一件则是通过它找到 FAT 中的项，从而找到 Loader 占用的其余所有扇区。装入一个扇区对我们来说已经是轻松的事情了，可从 FAT 中找到一个项还多少有些麻烦。而且，如果 Loader 占用多个扇区的话，我们可能需要重复从 FAT 中找到相应的项，所以我们通过一个 GetFATEntry 函数来做这件事。函数的输入就是扇区号，输出则是其对应的 FAT 项的值。

好了现在一切具备了，让我们开始加载 Loader 吧。

```
LABEL_FILENAME_FOUND:           // 找到 LOADER.BIN 后便来到这里继续
    mov ax, RootDirSectors
    and di, 0FFE0h              // di -> 当前条目的开始
    add di, 01Ah                // di -> 首 Sector
    mov cx, word [es:di]
    pushcx                      // 保存此 Sector 在 FAT 中的序号
    add cx, ax
    add cx, DeltaSectorNo       // 这句完成时 cl 里面变成 LOADER.BIN 的起始扇区号(从 0 开始数的序号)
    mov ax, BaseOfLoader
    mov es, ax                  // es <- BaseOfLoader
    mov bx, OffsetOfLoader // bx <- OffsetOfLoader 于是，es:bx = BaseOfLoader:OffsetOfLoader = BaseOfLoader * 10h + OffsetOfLoader
    mov ax, cx                  // ax <- Sector 号

LABEL_GOON_LOADING_FILE:
    pushax              // ┐
    pushbx              // │
    mov ah, 0Eh         // │每读一个扇区就在"Booting  " 后面打一个点，形成这样的效果:
    mov al, '.'         // │
    mov bl, 0Fh         // │Booting ......
    int 10h             // │
    pop bx              // │
    pop ax              // ┘
    mov cl, 1
    call ReadSector
    pop ax              // 取出此 Sector 在 FAT 中的序号
    call GetFATEntry
    cmp ax, 0FFFh
    jz LABEL_FILE_LOADED
    pushax              // 保存 Sector 在 FAT 中的序号
    mov dx, RootDirSectors
    add ax, dx
    add ax, DeltaSectorNo
    add bx, [BPB_BytsPerSec]
    jmp LABEL_GOON_LOADING_FILE
LABEL_FILE_LOADED:
    mov dh, 1           // "Ready."
    call DispStr        // 显示字符串
    jmp BaseOfLoader:OffsetOfLoader     // 这一句正式跳转到已加载到内存中的 LOADER.BIN 的开始处
                                        // 开始执行 LOADER.BIN 的代码
                                        // Boot Sector 的使命到此结束
```

最后在成功加载好Loader.bin之后，通过一个长跳转，开始执行Loader，从而完成由boot到loader的过渡。

1.3. Loader.asm 程序

1.3.1. 功能描述

在设计 boot 的时候是否还记得 Loader 需要做的工作有哪些？Loader 要做两项工作。

- 加载内核到内存
- 跳入保护模式

我们将分步来研究这两个任务。

1.3.2. Loader 加载内核

加载内核到内存这一步和引导扇区的工作非常相似，只是处理内核时我们需要根据 Program header table 中的值把内核中的相应的段放到正确的位置。我们可以这样来做，首先像引导扇区处理 Loader 那样把内核放入内存，只要内核进入了内存，如何处理它便是一件容易的事情了，我们可以在保护模式下挪动它的位置。

依旧是寻找文件、定位文件以及读入内存，实际上，单就把内核读入内存这一部分，除了文件名和读入的内存地址变了，其余其实都是一样的。之所以没有把它写成一个函数分别字啊 boot.asm 和 loader.asm 中调用，是因为函数在调用的时候堆栈操作会占用更多的空间，在引导扇区中，每一个字节都是珍贵的。加载内核的代码和 boot.asm 其实是差不多的，其中用到的函数如 DispStr、ReadSector 以及 GetFATEntry 和 boot.asm 中是完全一样的。添加了一个新函数是 KillMotor，用来关闭软驱马达，不然软驱的灯会一直亮着的。

最后需要做的工作是加载软盘上的应用程序到内存指定的位置。

1.3.3. 跳入保护模式

现在内核已经被我们加载到内存了，该是跳入保护模式的时候了。

在 IA32 下，CPU 的有两种工作模式：实模式和保护模式。直观地看，当我们打开自己的 PC，开始时 CPU 是工作在是模式下的，经过某种机制之后，才进入保护模式。在保护模式下，CPU 有着巨大的寻址能力，并为强大的 32 位操作系统提供了更好的硬件保障。如果你还是不明白，我们不妨这样类比，实模式到保护模式的转换类似于政权更迭，开机时是在实模式下，就好像皇帝 A 在执政，他有他的一套政策，你需要遵从他订立的规则，否则就可能被杀头。后来通过一种转换，类似于革命，新皇帝 B 登基，登基的那一刻类似于程序中的那个跳入保护模式。之后，B 又有他的一套完全

不同的新政策。当然，新政策比老政策好得多，对人民来说有更广阔的自由度，虽然它要复杂得多，这套新政策就是保护模式。我们要学习的，就是新政策是什么，我们在新政策下应该怎样做事。

In tel 8086 是 16 位的 CPU，它有着 16 位的寄存器、16 为的数据总线以及 20 位的地址总线和 1M 的寻址能力。一个地址是由段和偏移两部分组成的，物理地址遵循这样的计算公式：

物理地址（Physical Address）=段值（Segment）×16+偏移（Offset）

其中，段值和偏移都是 16 位的。

从 80386 开始，Intel 家族的 CPU 进入 32 位时代。80386 有 32 位地址线，所以寻址空间可以达到 4GB。如今我们有了 32 位寄存器，一个寄存器就可以寻址 4GB 的空间，是不是从此段值就被抛弃了呢？实际上并没有，新政策下的地址仍然使用“SEG：OFFSET”这样的形式来表示，只不过保护模式下“段”的概念发生了根本的变化。保护模式下，虽然段值仍然由原来 16 位的 cs、ds 等寄存器表示，但此时它仅仅变成了一个索引，这个索引指向了一个数据结构的表项，表项中详细定义了段的起始地址、界限、属性等内容。这个数据结构，就是 GDT。GDT 中的表项也有一个专门的名字，叫做描述符（Descriptor）。GDT 的作用是用来提供段式存储机制，这种机制是通过段寄存器和 GDT 中的描述符共同提供的。图 1.4 提供了描述符的结构。

存储段描述符	m+7	m+6	m+5	m+4	m+3	m+2	m+1	m+0
	Base(31...24)	Attributes		Segment Base(23...0)			Segment Limite(15...0)	

存储段描述符	Byte m+6								Byte m+5							
	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
属 性	G	D	0	AVL	Limit(19...16)				P	DPL		DT1	TYPE			

图 1-4 代码段和数据段描述符

首先是在 Doggyos 中定义的 GDT 以及对应的选择子，我们只定义三个描述符，分别是一个 0～4GB 的可执行段、一个是 0～4GB 的可读写段和一个指向显存开始地址的段（见下面的代码）。

```
// GDT
//                                                段基址       段界限       , 属性
// 段界限粒度可为字节或 K, 段基址粒度只能为字节
LABEL_GDT:              Descriptor                    0,           0,              0     // 空描述符
LABEL_DESC_FLAT_C:   Descriptor                    0,       0fffffh,   DA_CR   | DA_32 | DA_LIMIT_4
// 0 ~ 4G, DA_LIMIT_4K 设置段界限粒度为 K
LABEL_DESC_FLAT_RW:      Descriptor                0,       0fffffh,   DA_DRW | DA_32 | DA_LIMIT_4K
      // 0 ~ 4G
LABEL_DESC_VIDEO:        Descriptor          0B8000h,     0ffffh,     DA_DRW | DA_DPL3

// 显存首地址, 段界限粒度为字节
// GDT ------------------------------------------------------------------------------------
```

这里再对选择子进行一些说明。逻辑地址的选择子部分用于指定一个描述符，它是通过指定一个描述符表并且索引其中的一个描述符项完成的。图 1.5 给出了选择子的格式。各字段的含义说明如下。

图 1-5 段选择子格式

- 索引值（Index）：用于选择指定描述符表中 8192（2^13）个描述符中的一个。处理器将该索引值乘上 8(描述符的字节长度),并加上描述如的基地址即可访问表中指定的段描述符。
- 表指示器(Table Indicator-TI)：指定选择符所引用的描述符表。值为 0 表示指定的 GDT 表，值为 1 表示指定当前的 LDT 表。
- 请求者的特权级（Request's Privalege Level-RPL）：用于保护机制。

在进入保护模式之前，我们必须首先设置好将要用到的段描述符表，例如全局描述符表 GDT。然后使用指令 lgdt 把描述符表的基地址告知 CPU（GDT 表的基地址存入 gdtr 寄存器）。再将机器状态字的保护模式标志位置位即可进入 32 位保护运行模式。

```
;*************************************************
; 下面准备跳入保护模式 -------------------------------------------

; 加载 GDTR
  lgdt  [GdtPtr]

; 关中断
  cli

; 打开地址线A20
  in  al, 92h
  or  al, 00000010b
  out 92h, al

; 准备切换到保护模式
  mov eax, cr0
  or  eax, 1
  mov cr0, eax

; 真正进入保护模式
  jmp dword SelectorFlatC:(BaseOfLoaderPhyAddr+LABEL_PM_START)
```

图 1-6 跳入保护模式代码

1.4. 内核初始化

内核刚获得系统的控制权时，需要对自身进行初始化，然后系统的控制权就完全交给调度程序和进程。内核初始化的步骤为：

1. 初始化全局描述符表（GDT）。先复制 Loader 时的 GDT，然后增加相应的其他描述符。
2. 初始化中断芯片（主从 8259A）及中断描述符表（IDT），Doggy 所用到的向量号有 20h——时钟时钟，21h——键盘，80h——系统调用。
3. 初始化 TSS，Doggy 所用到是其中的 ss0，并设定没有 I/O 许可位图。
4. 初始化进程表，主要初始化 PID 和选择子。
5. 初始化内存管理模块。
6. 创建服务进程（TTY），负责所有的输入输出。
7. 启动时钟中断，进程调度机制开始。
8. 通过 iret 跳入用户态，此时第一个进程 TTY 开始，内核初始化完成。

内核初始化过程中，值得一提的是第 4 步建立起 GDT 中 LDT 表描述符和进程表中每个进程描述符表（ldt）的对应关系。如图 1.7：

图 1-7 GDT 和进程表

GDT 中包含每个进程 LDT 表的描述符，而这些描述符将在此时指向相应的进程表项中的 LDT 表，这样，内核初始化之后，Doggy 中的 GDT 再也不会改变。而相应的进程表项中的选择子将指向相应的 GDT 中的 LDT 表描述符。

第2章 内存管理

2.1. 概要

内存管理模块主要是按照一定的算法管理内存。DoggyOS 内存管理模块只采用了保护模式下的分段机制，没有打开分页机制，所以不支持虚拟内存和页面置换功能，系统运行过程中逻辑地址和线性地址是相同的，这样做的原因是现在 DoggyOS 系统中还不具备硬盘驱动。运行时，系统的地址空间和应用程序的地址空间是通过全局描述表（gdt）和局部描述符表（ldt）区分开来的，它们之间的相互转换则由软件层来实现。

D oggyOS 的内存管理模块总共有三个文件组成，如表 2-1：

表 2-1 内存管理模块代码文件列表

文件名	说明
Memory.c	内存管理的主要函数定义
Memory.h	内存管理的主要数据结构定义和函数声明
Proto.h	内存管理的系统调用

2.2. 总体功能描述

DoggyOS 内存管理模块主要有四个部分组成。（1）系统主内存（系统堆）的管理；（2）应用程序的堆管理；（3）内存管理的初始化；（4）内存管理的系统调用。

系统主内存管理：所谓的系统主内存，就是系统可使用的堆（heap）空间，它是从地址空间 2M 开始到内存末地址的所有空间，如图 2.1 所示。内存管理模块对主内存的管理主要涉及到内存分配，内存释放，统计信息维护等操作，现在这些操作主要用在应用程序装载时的空间申请，销毁时的内存释放，内核执行时的信息保存等。

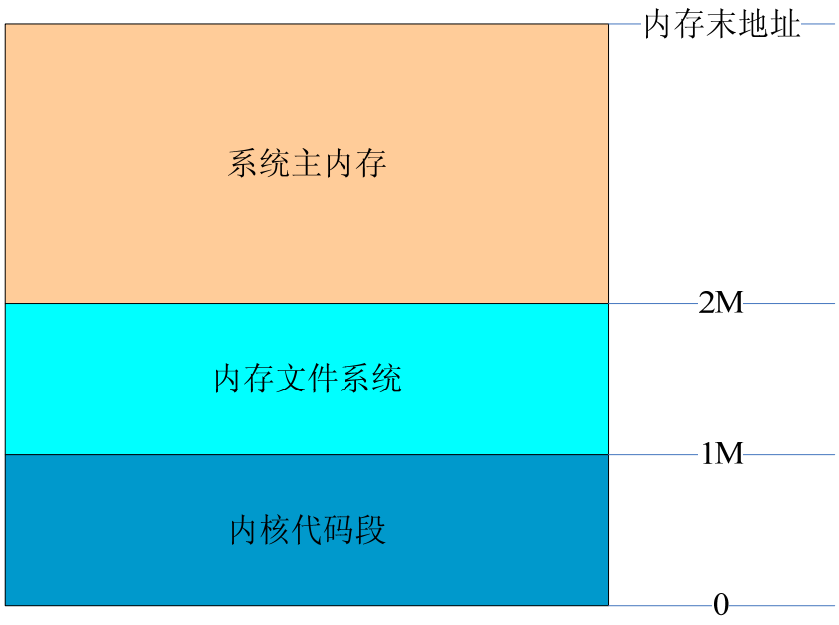


图 2-1 系统运行时内存映射图

应用程序的堆管理：每一个应用程序都有自己的堆空间，如图 2.2 所示，在装载应用程序时，内核首先从主内存分配一段空间（图中的大矩形），然后内核又将这个空间分为三个段，其中，中间的绿色段就是应用程序的堆空间。应用程序所有 malloc 和 free 操作所涉及的空间都是从这个堆空间中分配得到的，内存管理模块就负责对这个堆空间的管理，其管理算法与内核的堆管理方法算法相同，同时模块还负责两者地址空间的相互转换，整个过程对应用程序而言这是完全透明的。

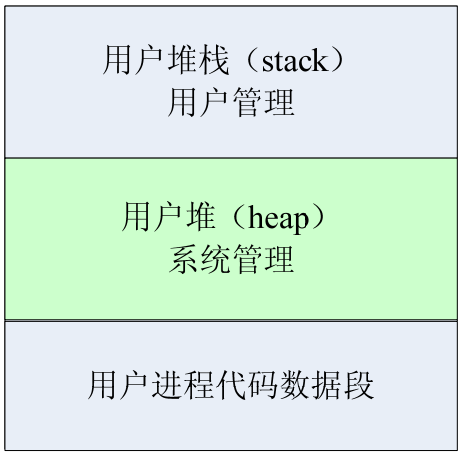


图 2-2 应用程序运行时内存映射图

内存管理的初始化：在内核初始化阶段需要对主内存进行一定的初始化工作，如放置哨兵数据结构，初始化空闲块列表等，为以后的分配和释放操作做好准备。对应用程序的堆空间也是如此，在内核为装载应用程序而分配空间时也要附带对应用程序的堆空间进行初始化。

内存管理的系统调用：现在对应用程序而言，主要存在两个系统调用来帮助程序申请和释放空间，它们分别为 malloc 和 free。

内存管理模块主要函数和功能由下表列出

表 2-2 内存管理模块主要函数

函数名	功能描述	备注
memSysLibInit	初始化内存管理模块	只在内核初始化时调用
memInitReg	初始化内存区域	
memRegAttach	初始化 region 数据结构中的各个域	
memRegAlloc	分配 region 数据结构的空间	
memRegAlignedAlloc	分配一定的内存空间，其实地址以 aligened 对齐	Aligened 的值要是 2 的幂
memRegBlockSplit	从一个空闲块分割一定大小块，要考虑对齐	
memRegFree	释放一段内存空间	
memCreateRegion	从主内存分配一段空间，用来动态加载程序	
memDestroyRegion	当程序销毁时，销毁与它相应的在主内存中的空间	
regAlloc	将初始化后内存与 region 数据结构进行绑定	
showMemRegStatus	打印内存信息	调试程序
sys_malloc Mall	oc 内核态程序	
sys_free Free	内核态程序	

2.3. 主要数据结构

2.3.1. Region 数据结构：

```
typedef struct Memory_Region
{
    LinkList freeList;              /* 空闲块列表的头结构，指向头节点和尾节点*/
    unsigned relativeAddress;     /*返回的地址要做相对处理，用于用户空间和内核空间的相互转换，
如果是系统主内存则为 0*/
    unsigned startAddress;   /* 这个 region 结构管理的内存开始地址(包括 HDR)*/
```

```
    unsigned totalBytes;          /*  这个 region 结构管理的内存的总大小*/
    unsigned minBlockBytes;       /*  在分配时，最小应该分配的 byte 数*/

    /*region 块的统计信息*/
    unsigned curBlocksAllocated;      /*  现在已经分配的块数*/
    unsigned curBytesAllocated;       /*  现在已经分配的 byte 数  */
    unsigned cumBlocksAllocated;      /*  积累分配的块数*/
    unsigned cumBytesAllocated;       /*  积累分配的 byte 数  */

}REGION;
```

每一个 region 数据结构都对应一段需要管理的内存区域，内存管理算法也是在相应的 region 结构进行的。系统主内存和每个应用程序的堆都有一个 region 数据结构与之相对应，内存管理模块通过识别 region 数据结构来确定对哪一个区域进行内存分配与释放。其中的 freeList 的域比较重要，因为每一个内存区域中的所有空闲块都是通过一个双向列表来管理，而 freeList 域就指向了这个双向列表的头节点和尾节点。relativeAddress 域是为应用程序的堆而设立的，用它可以进行用户空间和系统空间的地址转化。

2.3.2. BlockHdr 数据结构:

```
typedef struct Block_Hdr
{
    struct Block_Hdr* pPrevHdr;  /*指向相邻前一个块指针*/
    unsigned nBytes : 31;        /*这个块的大小*/
    unsigned isFree : 1;         /*表示这个内存块是否是空闲块,true 表示空闲块*/
}BlockHdr;
```

在 region 管理的一个内存区域中每一个已经分配出去和未分配出去的连续空间都是用一个 block 表示，它们在内存区域中是顺序排列的，在每一个块的最前面都有一个 BlockHdr 的数据结构来管理这个块。其中 pPrevHdr 是指向相邻前一个 block 的指针，nBytes 表示这个块的大小，通过 BlockHdr 的起始地址和 nBytes 可以计算得到相邻后一个 block 的地址，isFree 表示这个 block 是否为空闲块,true 表示空闲块。

2.3.3. FreeBlock 数据结构：

```
typedef struct Free_Block
```

```
{
    BlockHdr hdr;              /*空闲块的 BlockHdr */
    LinkNode freeList;         /*空闲列表的双向指针*/
}FreeBlock;
```

region 管理的一个内存区域中每一个空闲 block 的前面都有一个 FreeBlock 数据结构来管理它，其中 hdr 代表了这个 block 的 BlockHdr，其中的 isFree 域为 false，freeList 这个域将这个空闲块连入了 region 的空闲块链表。

2.3.4. 内存管理总体镜像图

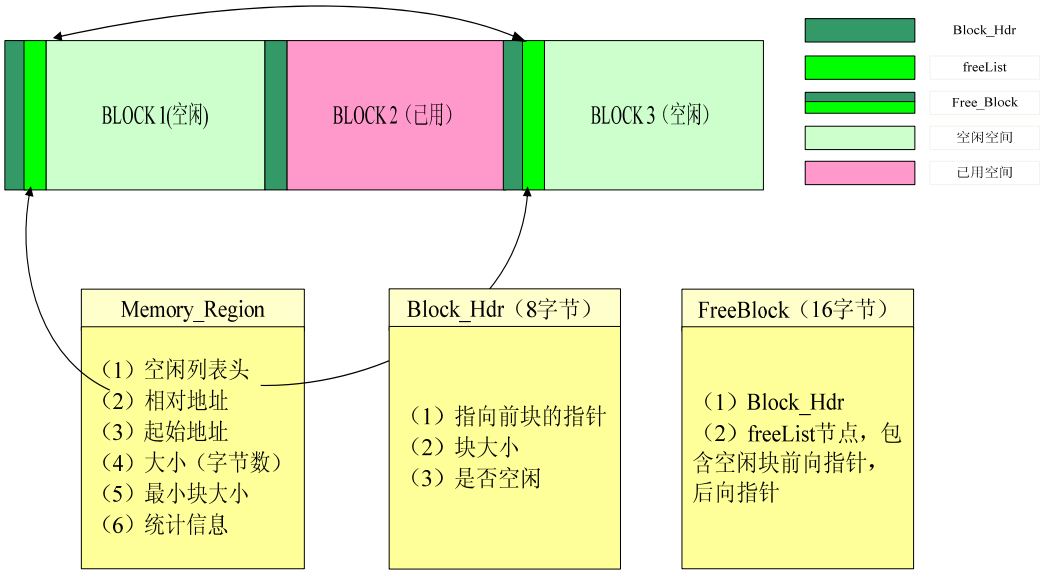


图 2-3 由 region 管理的内存镜像图（不包含头尾哨兵），其中上面的矩形表示内存的布局，下面黄色的矩形块表示数据结构

2.4. 主要算法描述

2.4.1. 内存初始化

系统主内存和每个应用程序的堆只能初始化一次。初始化过程主要完成三件事（1）得到需要管理的内存区域的起始地址，相对地址，内存大小（2）初始化相应内存区域，主要是创建两个头尾哨兵 BlockHdr，这两个 block 不带任何多于的 byte，不参与正常内存分配，它们的存在主要为了防止 region 分配的空间超出 region 所负责的区域。除去两个头尾节点，剩余的空间就作为一个大的空闲块加入到空闲列表中。(3)初始化 region 数据结构，将它和刚刚初始化的内存区域对应起来。初始化以后的内存镜像如图 2.4 所示

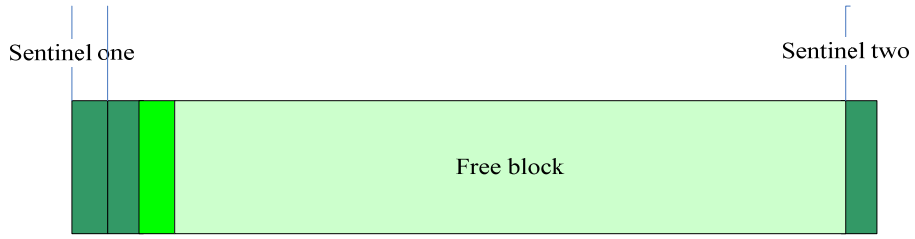
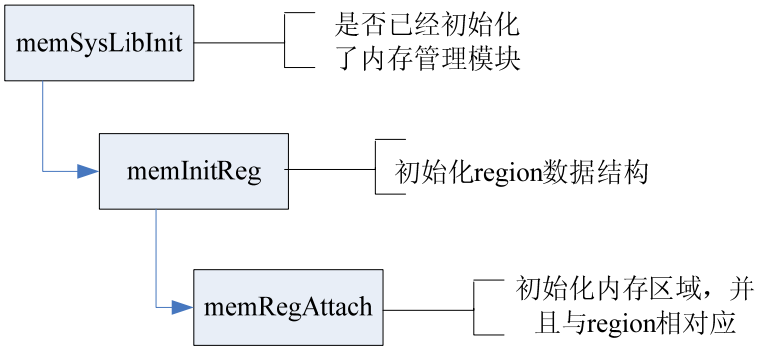


图 2-4 刚初始化完的内存镜像图

函数调用关系



初始化算法描述

```
1.  IF 还没有初始化内存管理模块
2.      得到内存区域的起始地址，大小，创建空闲列表
3.      将可管理的内存区域清零
4.      创建头哨兵节点 BlockHdr
5.      创建尾哨兵节点 BlockHdr
6.      创建中间最大空闲块，并将它加入到空闲列表中
7.      根据初始化的情况，赋值 memKernelReg（这是主内存的 region）
8.  END IF
```

2.4.2. 内存分配

内存分配主要采用 first-fit 策略，从空闲列表中找到满足分配要求（大小，对齐）的空闲块进行分裂，返回分配后的 block。一般一个空闲块经过分裂后会变成三个块，前后两个变成空闲块，中间 block 成为分配的块，管理算法之所以这样处理的原因在于一般内存分配都要求分配返回的地址与某个值相对齐，所以不能直接从空闲块的末尾分配，而要有一个地址向下对齐的过程。如果分裂后的前后两块的大小小于 region-> minBlockBytes 的值就要试图和中间块进行合并，这样做的目的是为了防止存在大量内存碎片。

整个分配过程可以用下图来表示，图 2.5 是分配之前的内存镜像图，可以看到此刻整个内存区

域存在三个 block（前后两个哨兵 block 未显示）。其中，中间粉色 block 是已经分配的块，前后绿色 block 为空闲块，用链表连接了起来。

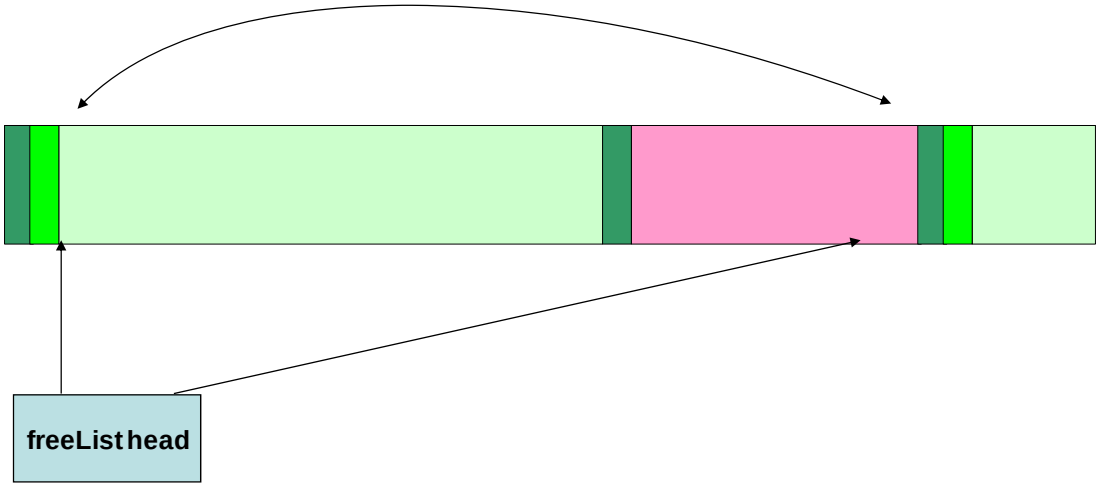


图 2-5 分配之前的内存镜像图

图 2.6 是分配以后的内存镜像图，根据 first-fit 策略从空闲列表中找到第一个符合要求的 block，图中显示为第一个空闲块，接着将第一个空闲块分裂成三个块，并且将前后两个空闲块重新放入空闲列表中。

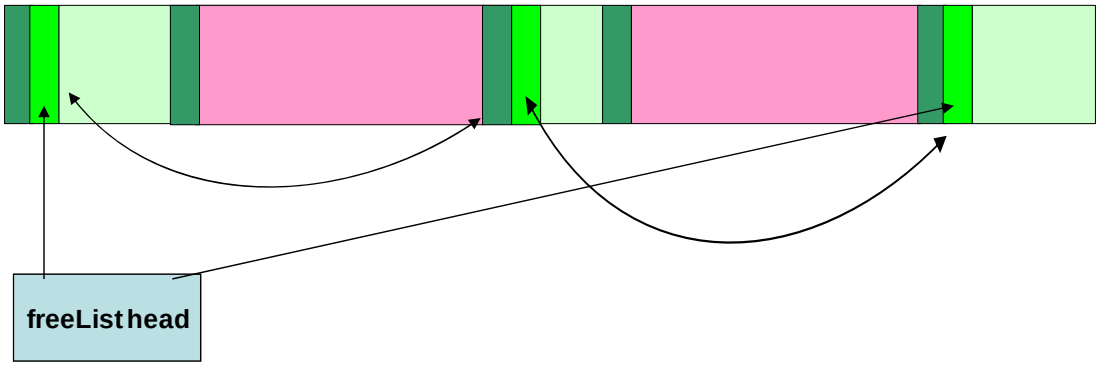
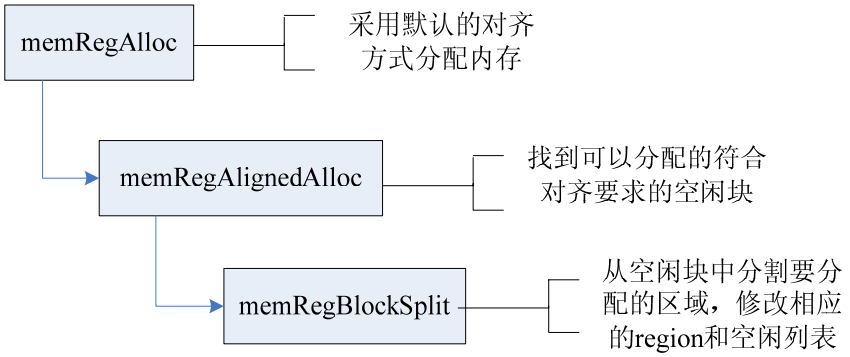


图 2-6 分配以后的内存镜像图

函数调用关系



分配算法描述

```
1.  检查申请分配的要求是否合法，如果要分配的大小太小，就直接分配 region-> minBlockBytes 大小
2.  WHILE 遍历空闲块列表
3.      IF 如果找到一个空闲块的容量大于要分配的大小
4.          试图从这个空闲块中分裂
5.              根据对齐要求将空闲块分成三块，其中中间那块是要分配的块
6.              IF 前面那块的大小小于 minBlockBytes && 这块的起始地址满足对齐要求
7.                  从这块空闲块的起始地址开始分配，这样原来的空闲块变成两块（前一块是要分配的块）
8.              ELSE
9.                  不能从这块空闲块分配，继续寻找其他空闲块
10.             END IF
11.             IF 后面那块的大小小于 minBlockBytes
12.                 将这一块与要分配的块合并，增大分配的 bytes 数
13.             END IF
14.     END IF
15.     IF 分配成功
16.         更新 region 和空闲列表
17.     END IF
18. END WHILE
```

2.4.3. 内存释放

内存释放相对比较简单，主要就是把要释放的块加入到空闲列表中。为了防止内存碎片，在这之前，要释放的块会试图跟它前面和后面的块进行合并，组成比较大的空闲块。

函数调用关系

释放算法描述

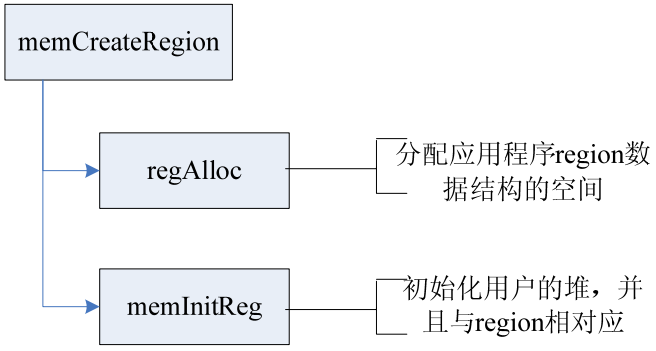
```
1.  IF 释放块的后一块是空闲块
2.      释放块与后一块合并
3.  END IF
4.  IF 释放块的前一块是空闲块
5.      释放块与前一块合并
6.  END IF
7.  将释放块联入到空闲列表中，修改 region 数据结构
```

2.4.4. 应用程序堆管理

应用程序堆初始化：

应用程序堆初始化过程与系统主内存初始化过程基本相同。唯一不同在于，主内存的 region 数据结构是一个静态变量，不需要动态分配，而应用程序的 region 数据结构是在初始化过程中从主内存中动态分配的，在应用程序销毁时需要动态释放。应用程序堆管理的算法是和主内存完全相同的，只是使用不相同的 region。

函数调用关系



应用程序堆初始化算法描述

1. 得到应用程序堆的起始地址，大小
2. 从主内存中分配 region 数据结构的空间
3. 将应用程序的堆清零
4. 在堆中创建前后哨兵节点 BlockHdr
5. 在堆中创建中间最大空闲块，并将它加入到空闲列表中
6. 初始化 region 数据结构

用户空间和系统空间的地址转换：

对应用程序堆管理是由内核来完成的，而堆分配出来的空间则是由应用程序使用的，所以要在两者之间进行地址转换。这时就要用到 region 数据结构中 relativeAddress 域，它保存了应用程序加载时在系统空间中的起始地址，任何用户空间地址都是相对于这个地址而言的。所以要进行地址相转换，只要在系统调用前后加减这个相对地址就可以了。具体的操作如图 2.7 所示：

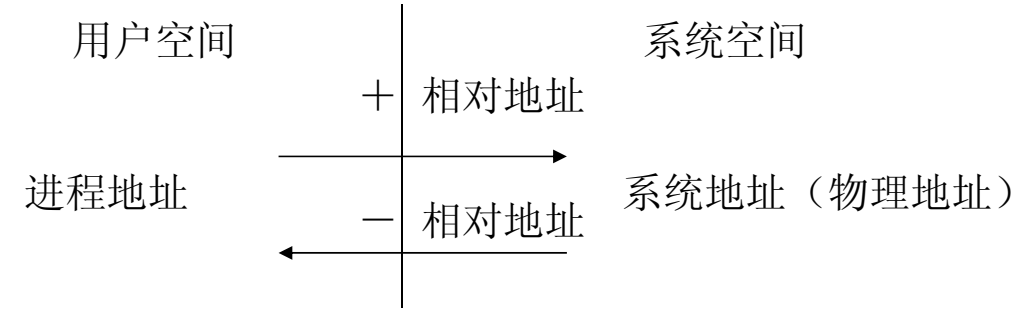


图 2-7 空间地址转换图

2.5. 相关主题

2.5.1. 内存容量的获得

DoggyOS 内存管理模块开发中遇到的第一个问题是如何得到内存大小的硬件参数。LINUX0.11 的做法是将主内存的底端地址和大小都写死在代码中，如在 memory.c 中：

```
43  #define LOW_MEM       0x100000                    // 内存底端（1MB）
44  #define PAGING_MEMORY       (15*1024*1024)    //分页内存 15MB。主内存最多 15MB
```

这样做的缺点是系统的可扩展性不是很好，不能有效的利用所有的内存，在当今内存已经以 G 来计算的年代，会造成内存的浪费。

为了合理的利用内存资源，有效的提高系统的可移植性，一个操作系统必须知道内存的容量，以便进行内存管理。DoggyOS 利用实模式下 BIOS 硬件中断 15h 来得到内存的容量（loader.asm 中实现），并将它保存到一个变量当中，在保护模式下 loader.asm 完成内核的启动初始化之后正式将控制权交给内核之前将这个参数压入堆栈。当进入 kernel 之后，进行内核堆栈切换之前，将内存容量值出栈（pop）保存，这样 DoggyOS 的内核就得到了内存的容量大小。

具体的讲，bios 中断 15h 会返回一个地址返回描述符结构（Address Range Descriptor Structure），里面有 BaseAddrLow, BaseAddrHigh, LengthLow, LengthHigh 这几个域，顾名思义，通过这几个域我们就得到内存区域的基址和长度，并且知道这个段是否可以使用的信息。将这个信息保存到变量 _szRAMSize 中，在正式将控制权交给内核之前，通过

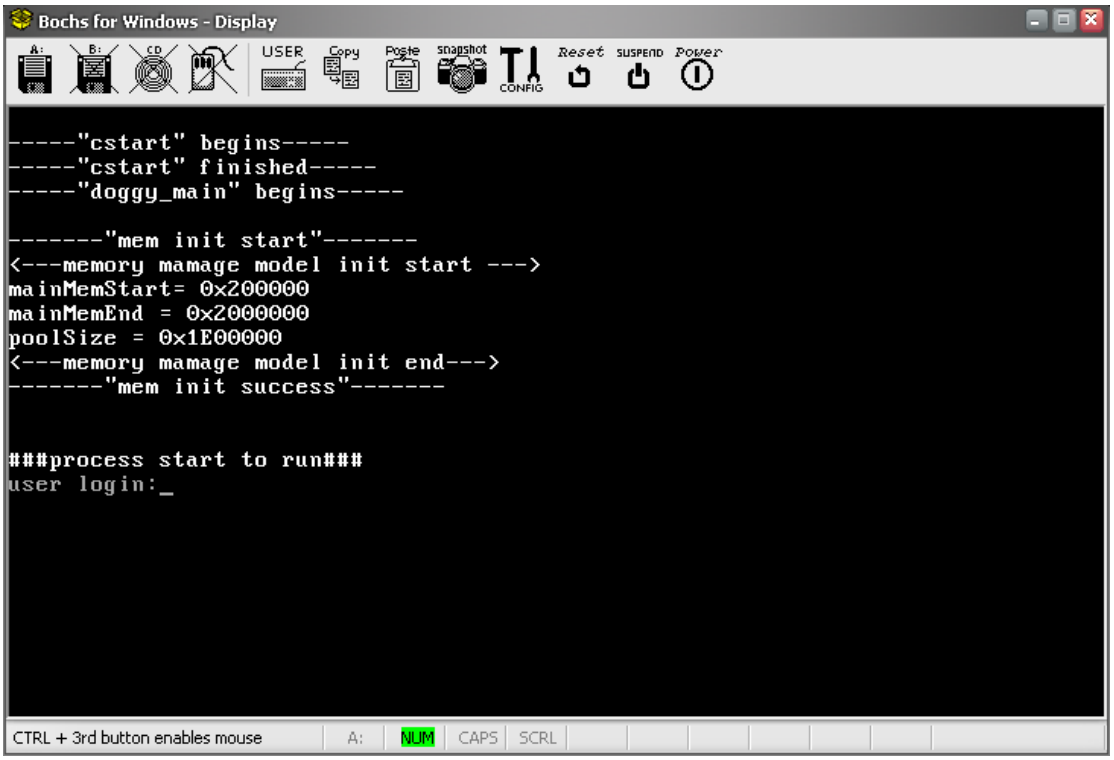
```
push dword [dwMemSize]                          ; 这里我们需要得到内存的大小
jmp  SelectorFlatC:KernelEntryPointPhyAddr      ; 正式进入内核  *
```

将大小压入堆栈，其中 dwMemSize 是 szRAMSize 在保护模式下的地址，第二条语句就正式进入内核了。

在进入内核之后，所要做的第一件事情就是

```
pop  dword [mem_size]
```

将内存大小保存到变量 mem_size 中，然后再做内核堆栈的切换。到此我们正式将内存大小参数传递给了内核，在内核的内存管理模块初始化结束之后，我们就可以从屏幕看到内存的大小了



其中 mainMemEnd 就是得到的内存的末尾地址（与我们在 bochs 的配置文件中设置的 32M 相符），mainMenStart 是系统主内存的起始地址。

2.5.2. DoggyOS 内存管理模块对 VxWorks 的创新和发展

DoggyOS的内存管理相对比较简单，想法的来源主要是实时操作系统VxWorks的内存管理模块，我们在此基础之上做相应的修改和创新。在 DoggyOS 项目的设计之初，我们选择用 Vxworks 作为 DoggyOS 内存管理的原型系统，主要原因在于 DoggyOS 第一个阶段的实现目标中将不包含硬盘驱动，所以就不能实现当今操作系统中普遍流行的虚拟内存机制（无法实现页面的 swap 功能），从而也就放弃了分页机制，我们只选择了利用保护模式中的分段机制作为我们系统的硬件支持，而 Vxworks 的内存管理也没有运用分页机制，跟我们实现目标比较一致，有利于我们学习借鉴。但是我们的内存管理模块与 VxWorks 还是存在很多不同。

首先 Vxworks 是一个实时操作系统，所以在内存管理中不能运用普通操作系统的虚拟内存机制，因为实时系统对延迟有很大的要求，而虚拟内存页面 swap 所造成的系统延迟对实时操作系统是致命的，所以在实时系统中一般都是不开分页机制的，最多只是利用了保护模式中分段机制，如 Vxworks，这样系统的逻辑地址就等于线性地址，对于一些硬实时系统来说根本上放弃了硬件 mmu 对地址转换

的支持，在系统中逻辑地址就等于物理地址，而且每一个程序的运行地址都是在编译时就已经确定的，可以说在硬实时系统中根本就不存在内存管理。而对于 DoggyOS 来说，是因为没有硬盘驱动才放弃了虚拟内存的。其次 VxWorks 是任务抢断式操作系统，所以为了保护内存管理中许多关键的数据结构，内核代码有许多申请锁，释放锁，保护关键区等操作，而且这些代码都相对比较晦涩。而 DoggyOS 系统采用的是非抢占的调度策略，所以在内核态的时候不需要繁琐的锁操作，代码相对比较清晰。第三，Vxworks 出于对性能的考虑，在系统运行时不区分内核空间和用户空间，用户空间地址和内核空间地址是相同的，虽然 VxWorks 也提供了用户程序自己管理堆空间的函数接口（在 5.4 版本中只有堆空间的申请，还没有实现释放功能），但是用户程序在运行时不需要通过系统调用就可以直接访问内核空间的数据，没有任何保护机制，系统执行时的安全性要靠编写应用程序的程序员来保证，而且所有应用程序都是随内核一起编译，不支持程序的动态加载。在 DoggyOS 系统中，我们实现了用户空间和系统空间的分离，每个应用程序都和一个本地描述符（ldt）相对应，应用程序除了系统调用，不能随便访问系统的任何地址空间，否则越界访问将抛出越界异常。在 DoggyOS 中，我们还实现利用软件技术对用户空间和系统空间的地址转换，虽然没有 mmu 直接操作的高效率，但也弥补了缺乏分页机制而导致的功能不足。DoggyOS 应用程序有完整的堆管理接口（malloc，free），每一个应用程序都有一个 region 数据结构来对应自己的堆空间。而且 DoggyOS 采用的是非抢占任务调度策略，在内存管理中就可以舍弃很多无用的同步操作。

第3章 进程管理

3.1. 概述

程序是一个可执行的文件，而进程是一个执行中的程序实例。操作系统往往利用分时技术，同时可以运行多个进程。分时技术的基本原理是把 CPU 的运行时间划分成一个个规定长度的时间片，让每个进程在一个时间片内运行。当进程的时间片用完时系统就利用调度程序切换到另一个进程去运行。

Doggy 实现了类似的进程管理机制，其内容包括 PCB，进程创建和销毁，进程调度，进程动态加载及相关的系统调用。

Doggy 的进程分为两种：用户进程和服务进程。就目前来说，仅 TTY 属于服务进程。进程运行模式包括用户态，服务态和内核态，分别对应 CPU 特权级中的 ring3，ring1 和 ring0 级，如图 3-1：

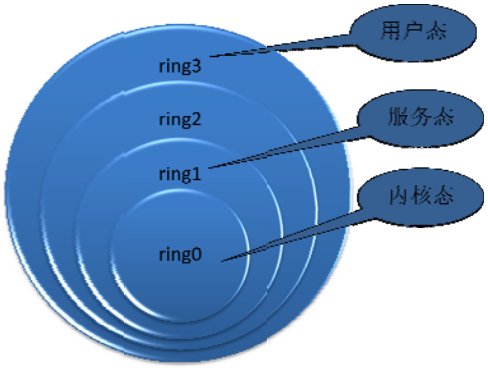


图 3-1 进程特权级和模式

进程文件主要包括（表 3-1）：

表 3-1 进程对应文件

文件名	说明
Proc.h	进程结构体、方法、常量定义
Proc.c	进程调度
Fork.h	声明进程生命周期方法的头文件
Fork.c	进程生命周期方法定义

3.2. 进程描述

3.2.1. 进程数据结构

进程相关的数据结构主要有 PCB，如下：

```
typedef struct s_proc
{
      STACK_FRAME regs;      /* 进程当前寄存器表 */

      t_16  ldt_sel;         /* 进程对应的 LDT 的选择子 */
      DESCRIPTOR ldts[LDT_SIZE];  /* LDT 描述符表 */
      int ticks;          /* 时间片 */
      int priority;       /* 优先级 */
      t_32 pid;           /* 进程 ID */
      t_32 ppid;          /* 进程父结点 ID */
      char name[16];      /* 进程名 */
      int nr_tty;         /* TTY */
      int state;  /* 进程状态 */
      t_32 cids[16];  /* 子进程 ID, 最 16 个儿子*/
      REGION* region;     /* 内存块*/
      void* base;     /* 进程块的起始绝对地址 */
}PROCESS;
```

其中进程当前寄存器表（regs）保存了进程在运行挂起时的所有寄存器状态，这保证在进程重新被调度时能正确地恢复到进程挂起时的状态，使得程序正常执行，其结构如下：

```
typedef struct s_stackframe
{     /* proc_ptr points here                                  ↑ Low        */
      t_32 gs;          /* ┐                                  │            */
      t_32 fs;          /*  │                                 │            */
      t_32 es;          /*  │                                 │            */
      t_32 ds;          /*  │                                 │            */
      t_32 edi;         /*  │                                 │            */
      t_32 esi;         /*  ├ pushed by save()                │            */
      t_32 ebp;         /*  │                                 │            */
      t_32 kernel_esp;    /* <- 'popad' will ignore it        │            */
      t_32 ebx;         /*  │栈从高地址往低地址增长            ↑            */
      t_32 edx;         /*  │                                 │            */
      t_32 ecx;         /*  │                                 │            */
      t_32 eax;         /* ┘                                  │            */
      t_32 retaddr;  /* return address for assembly code save()  │            */
      t_32 eip;         /*  ┐                                 │            */
      t_32 cs;          /*   │                                │            */
      t_32 eflags;      /*   ├  pushed by CPU during interrupt │            */
      t_32 esp;         /*   │                                │            */
      t_32 ss;          /*  ┘                                 ┴High        */
}STACK_FRAME;
```

每个进程拥有自己的 LDT，进程 PCB 中包含两个 LDT 描述符，分别描述进程的代码段和数据段。时间片和优先级的值实际是相等的，这用来实现一种简单的优先队列调度算法。PCB 中包含内存块信息，由内存模块进行管理。同时 PCB 中包含进程的状态。

3.2.2. 进程状态

进程的状态和 Linux 类似，如图 3-2：

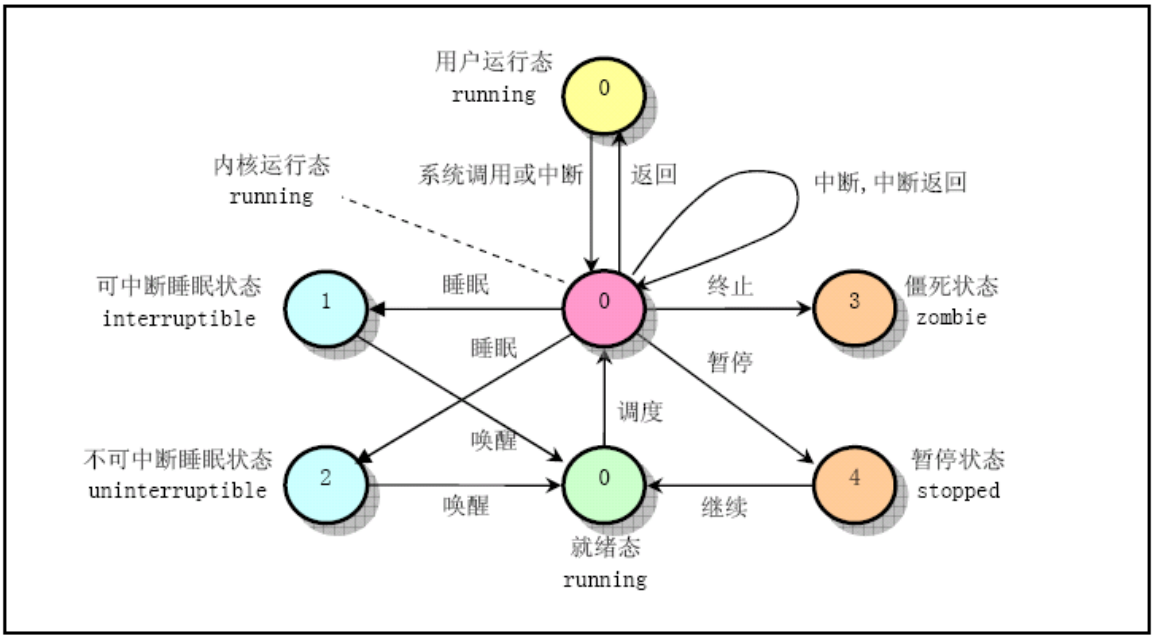


图 3-2 进程状态

最初的设计是模拟 Linux 状态转换，各状态的解释如下：

- 运行状态（PROC _RUNNING）

当进程正在被 CPU 执行，或已经准备就绪随时可由调度程序执行，则称该进程为处于运行状态（running）。进程可以在内核态运行，也可以在用户态运行。当系统资源已经可用时，进程就被唤醒而进入准备运行状态，该状态称为就绪态。这些状态（图中中间一列）在内核中表示方法相同，都被成为处于 TASK_RUNNING 状态。

- 可中断睡眠状态（PROC_INTERRUPTIBLE）

当进程处于可中断等待状态时，系统不会调度该进行执行。当系统产生一个中断或者释放了进程正在等待的资源，或者进程收到一个信号，都可以唤醒进程转换到就绪状态（运行状态）。

- 不可中断睡眠状态（PROC _UNINTERRUPTIBLE）

与可中断睡眠状态类似。但处于该状态的进程只有被使用函数明确唤醒时才能转换到可运行的就绪状态。

- 暂停状态（PROC _STOPPED）

当进程收到信号一些暂停信号时就会进入暂停状态。可向其发送唤醒信号让进程转换到可运行状态。

- 僵死状态（PROC _ZOMBIE）

当进程已停止运行，但其父进程还没有询问其状态时，则称该进程处于僵死状态。

不过 Doggy 虽然定义了这么多进程状态，但是实际的实现并没有这么复杂。就目前来说，真正使用的状态有 running，stopped，zombie。

3.2.3. 进程相关系统调用

进程相关的系统调用主要如表 3-2：

表 3-2 进程相关系统调用

系统调用	说明
exec	动态加载并创建进程。
_exit	进程退出。
kill	杀死进程。

3.3. 进程生命周期

3.3.1. 进程初始化

进程初始化的工作在刚进入 Linux 内核时进行。在第一个进程被运行之前，内核会初始化整个进程表，这也表示，在系统运行过程进程个数的最大值不会被改变。进程初始工作最主要初始化的其中的 PID 和选择子的值。所有进程表项中的 PID 都被赋值为-1，而选择子指向全局描述符表中，定义进程表项本身 LDT 的描述符。具体的初始化过程可以参考内核初始化中进程初始化的相关部分。

3.3.2. 进程创建

进程创建是一个比较简单的过程。主要分为以下几步：

1. 准备用于创建进程的结构体 PROC_DESC，该结构体包含进程创建时所需的各种信息。其中进程代码段入口 initial_eip，指示进程的第一句代码的偏移，对于那些动态加载的进程，它的值始终为 0，因为这个值是相对进程代码段的 LDT 描述符而言的，而代码的入口在 Doggy 为代码段的第一行指令。其他字段与进程 PCB 结构体中很多类似，见代码中注释，这里就不多介绍。

```
typedef struct s_proc_desc
{
    t_pf_procinitial_eip;    /* 进程代码段入口 */
    int         stacksize; /* 进程栈大小 */
    char        name[32];    /* 进程名 */
    t_8         privilege; /* 特权级 */
    t_8         rpl;             /* 特权属性 */
    int         eflags;          /* 相关属性 */
    t_16        selector_ldt;        /* LDT 选择子 */
    void*       p_proc_stack; /* 堆栈地址 */
    int         ticks;           /* 时间片 */
    int         priority;        /* 优先级 */
    int         nr_tty;          /* TTY */
    t_32        ppid;            /* 父进程 ID */
    DESCRIPTOR ldts[LDT_SIZE];  /*   描述符表*/
}PROC_DESC;
```

2. 调用 create_process（fork.c）创建进程，该方法首先调用 find_empty_process 找到空闲的进程表项和空闲的进程号。进程通过最近创建的进程号加 1 得到，该进程保存在全局变量中，供下次创建时使用。如果该进程号被使用，则继续加 1，一直找到未被使用的进程号；然后遍历整个进程表，找空闲进程表项，找不到则进程创建失败。目前 Doggy 最多允许 17 个进程运行。
3. 给找到的空闲进程表项（PCB）赋值。主要把结构体 PROC_DESC 中值赋给它。创建成功之后，设置进程的状态为 PROC_STOPPED，此时它还不能被调度。
4. 如果是动态加载的进程，还会把进程所用的堆交给内存模块管理（调用 memCreateRegion），之后把进程状态设为 PROC_RUNNING，此时进程将被正常调度，进程创建完毕。

3.3.3. 进程调度

进程调度采用非抢占优先队列调度算法，并且调度的时间片是不等长的。调度的过程如下：

优先级最高的进程先被调度，并且进程优先级越高，其时间片也越长。由于非抢占，这个进程执行不会被打断。当该进程的时间片用完之后，它的值设为 0，在其他进程全部调度一遍之前，它不会再被调度。在下次调度时，同样剩下的进程中优先级最高的进程将被调度。当一轮调度完成，即所有的进程都被调度了一遍，新的一轮调度将以同样的方式进行。

3.3.4. 进程切换

在进程调度过程中，伴随着进程切换的进行。一个进程要在被切换后，下一次重新运行仍然保持原有的状态，这就要在切换前保持上一次运行的进程的状态。同时进程的切换还伴随着用户态（服

务态）和内核态的切换，这就涉及到进程栈、进程表、内核栈三个堆栈之间的切换。如图 3-3 所示：

图 3-3 进程切换

具体的进程切换步骤为：

1. 时钟中断发生，进程 P1 时间片运行结束。系统进入内核态，堆栈切换到进程表中，这样进程 P1 被中断时的所有寄存器值保存到进程表相应表项中。
2. 堆栈被切换到内核栈，这样做可以防止内核态运行过程中进程表被错误地修改。调度算法被执行，新的进程 P2 被选中。
3. 调度算法运行完毕，系统堆栈切换到进程表中，此时堆栈指针 esp 指向进程 P1 的进程表项处。此时手动设置 esp 指向进程 P2 的进程表项，为切换到 P2 做到准备。
4. 系统通过 ired 指令切换为用户态，由于之前 esp 被指向进程 P2 的进程表项，所以返回到用户态后，进程 P2 就运行起来了。这样一个进程切换就完成了。

3.3.5. 进程动态加载

程序一开始初始化时，把所有软盘中的可执行文件加载到内存 1M~2M 处，构成一个简单的内存文件系统。进程的动态加载，就是从内存文件系统中读出某个可执行文件（ELF 格式），解析 ELF 格式，把程序加载到某个申请的内存中，然后创建进程执行它。

进程动态加载分为查找程序、解析加载程序、创建运行进程：

查找程序

进程动态加载发生成系统调用 exec 被调用之后，此时系统进入了内核态。系统必须先在内存文件系统中遍历查找相应可执行文件，找到之后才能加载。内存文件系统实际上并不能算是一个真正的文件系统，因为它只提供读操作。它是在内核被加载时，由 Loader 程序从软盘加载到内存中的。内存文件系统如图 3-4 所示：

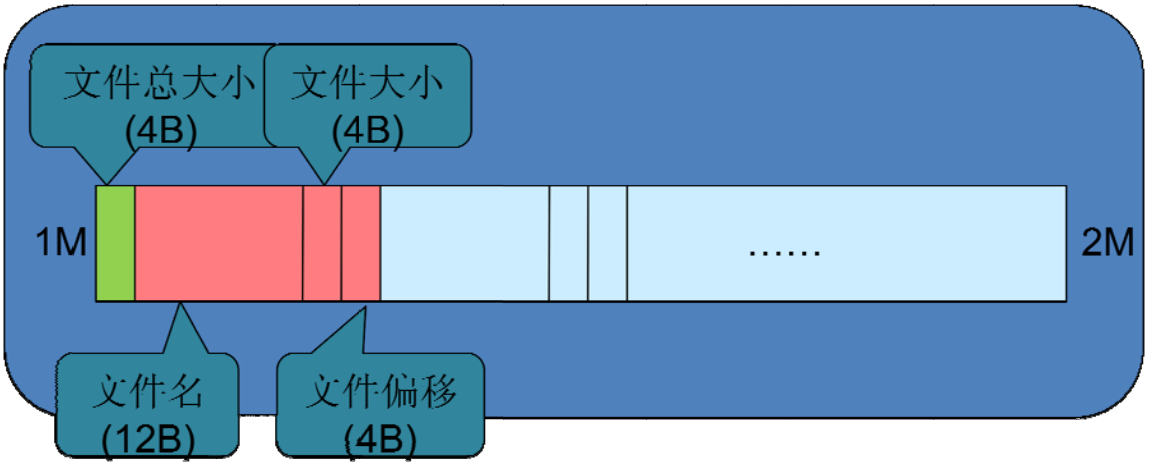


图 3-4 内存文件系统

内存文件系统包括文件目录和文件内容。在文件目录中，前四个字节为文件总大小，接下来每 20 个字节为一个文件条目。一个文件条目中，前 12 个字节为文件名，接下来 4 个字节为文件大小，再下面 4 个字节为文件偏移。通过一个文件条目就可以正确地定位和读取内存中的可执行文件。如果遍历了整个内存文件系统而未找到相应文件，则动态加载失败。

解析加载程序

找到的可执行文件是 ELF 格式，必须进行正确地解析才能正常运行。这一步所作的操作就是解析 ELF 文件，然后把它放到指定的内存中，其步骤如下：

1. 根据可执行文件的大小的向内存管理模块申请一块合适大小的内存。
2. 解析 ELF 文件，把它按 ELF 中的要求拷到申请的内存中。其中数据段和代码段从申请内存的最低地址开始存放，堆栈从最高地址开始，剩下的部分作为堆，由内存管理模块来管理。

创建运行进程

进程创建的过程和前文提到的一样，只是注意要把进程的堆交给内存管理模块来管理，并在 PCB 中保存堆的信息。

3.3.6. 进程销毁

进程的销毁主要分为下面几步：

1. 进程状态设为 PROC_ZOMBIE，但无任何价值，因为目前的实现来说，Doggy 并未处理 PROC_ZOMBIE 的状态。
2. 进程所对的进程表项的 PID 设为-1，这样它就不再被调度。
3. 如果是动态加载的内存，将通知内存管理模块释放堆，再释放整个进程所占内存区域。
4. 重新调用进程调度函数，该进程就永远消失了。

第4章 IO 模块

4.1. 总体介绍

DoggyOS 中 I/O 模块的设计目标是基于 Tinix 系统，实现基本的 IDT 描述表在内核中的实现。由于整体系统基于软盘存储，所以需要实现基本的软盘文件系统的存取封装操作。并尝试编写基本的键盘驱动和文本模式显存操作命令。

计划设计并实现基本的 shell 环境，包括以内核形态存在的基本 shell 命令。设计并实现多个终端的用户交互系统。设计并实现对现存的多用户空间操作。

为方便基于内核的进程调用，尝试设计实现部分功能 printf 系统调用，提供此 API 给用户态进程。

4.2. 软盘 I/O

4.2.1. FAT12 介绍

Fat12 是普遍使用与 DOS 时代的文件系统，现在仍然在软盘上使用。为了方便操作，将启动软盘镜像也设置成为 FAT12 格式

```
;================================================
; FAT12 文件系统头
; 这个块会让 windows 认出编译后的二进制文件为有效的引导文件
; 如果不使用这个块, 将不会将其作为引导程序处理
;================================================
bsOEM       db "ExOS0.02"          ; OEM String,任意你喜欢的 8 字节 ASCII 码
bsSectSize  dw 512                 ; Bytes per sector
bsClustSize db 1                   ; Sectors per cluster
bsRessect   dw 1                   ; # of reserved sectors
bsFatCnt    db 2                   ; # of fat copies
bsRootSize  dw 224                 ; size of root directory
bsTotalSect dw 2880                ; total # of sectors if < 32 meg
bsMedia     db 0xF0                ; Media Descriptor
bsFatSize   dw 9                   ; Size of each FAT
bsTrackSect dw 18                  ; Sectors per track
bsHeadCnt   dw 2                   ; number of read-write heads
bsHidenSect dd 0                   ; number of hidden sectors
bsHugeSect  dd 0                   ; if bsTotalSect is 0 this value is
                                   ; the number of sectors
bsBootDrv   db 0                   ; holds drive that the bs came from
bsReserv    db 0                   ; not used for anything
```

```
bsBootSign  db 29h                ; boot signature 29h
bsVolID     dd 0                  ; Disk volume ID also used for temp
                                  ; sector # / # sectors to load
bsVoLabel   db "NO NAME    "      ; Volume Label
bsFSType    db "FAT12   "         ; File System type <- FAT 12 文件系统
```

这里要对软盘的寻址方式(可以认为是读取数据的方式)是:CHS,C = Cylinder(柱面),H = Header（磁头）,S = Sector（扇区）。

0 面 0 道第 2 扇区到第 10 扇区的 9 个扇区是 FAT 表的存放位置，为了预防，0 面 0 道的第 11 扇区到 1 面 0 道第 1 扇区的 9 个扇区是第 2 个 FAT 表的存放位置，这第 2 个 FAT 是备用的，当第一个 FAT 出了问题里，可以用第 2 个 FAT。1 面 0 道的第 2 扇区起到 1 面 0 道的第 15 扇区（共 14 个扇区）用于存放 FDT 。FDT 没有备份，所以没有第二个 FDT。这里要注意的是，磁盘为了读写的速度，0 面 0 道的 18 个扇区接下来的是 1 面 0 道的扇区，而不是 0 面 1 道，因为 0 面 0 道跟 1 面 0 道同在一个柱面上（同心圆），只是用的磁头不同。

FAT12 中，每个文件分配表项只占 12 位(bit)，即 1.5 字节(byte)，每个表项代表一个扇区，在这里,磁盘只有扇区的概念,磁盘里所有扇区都被类似于上一段提到的磁盘读写方式线性地编址(LBA),不再有 CHS。这里还要提一提簇的概念：DOS 会把 2 个扇区作为一簇，那么文件就要以簇为单位读写。簇的大小通常根据磁盘的大小设定，以尽可能少浪费磁盘空间为本。

FAT12 每个表项的值指出文件存放的下一个扇区号，同时也是表项入口。比如如果文件的存放的第一个扇区是 002，那系统首先找 FAT 的 002，在 002 处得到一个值 003，表示文件下一个扇区是 003 号，再接着 003 表项找，得到 006...，表项的值含义如下：000 - 此簇未用；FF8 - FFF 该簇为文件的最后一簇；FF0 - FF7，此簇为坏，不可用；其它值表示文件下一簇的簇号。

下面的图来说明 FAT 的基本原理：

```
表项编号      值(16 位)          备注.
000  |   FF0   |  <- 000 项 00 1 项为表头，1 字节 0x F0 表示存储介质
001  |   FFF   |  <- 2、3 字节为 0xFFFF ，固定值，FAT 标志
002  |   003   |  <- 文件下一簇为 003
003  |   005   |  <- 下一簇：005
004  |   FF7   |  <- 坏簇，不可用
005  |   011   |  <- 下一簇：011
.....................................
011  |   FF8   |  <- 该文件结束
012  |   000   |  <- 可用簇
.....................................
```

根据上表，我们可以知道，一个文件占用了 002 ,003,005,011 这 4 个簇。

簇号 + 31 = 逻辑扇区号 //// 31 ＝ 保留扇区数 + 隐藏扇区数 + FAT 数×每个 FAT 所占扇区数 + FDT 所占扇区数 - 2 = 1 + 0 + 2*18 + 14 -2

LBA = 逻辑扇区号 – 1

扇区 = (LBA MOD 每道扇区数) + 1

磁道 = (LBA / 每道扇区数) / 磁头数

磁头 = (LBA / 每道扇区数) MOD 磁头数

根据上面的公式，得到以下计算值：

002: S = (32 MOD 18) + 1 = 15

002: C = (32 / 18) / 2 = 0

002: H = (32 / 18) MOD 2 = 1

011: S = ((11+31-1) MOD 18) + 1 = 6

011: C = ((11+31-1) / 18) / 2 = 1

011: H = ((11+31-1) / 18) MOD 2 = 0

就此，我们已经可心根据簇号得到物理 CHS 了，那怎样才能得到一个文件的关系首簇号呢？前面我们提到了 FDT。下面说说 FDT 的结构：

每个 FDT 项占 32 字节，分配如下：

==========

0 - 7 ： 8 字节，文件名
8 - 10： 3 字节，文件扩展名
11 ： 1 字节，文件的属性
12 - 15： 4 字节，保留
16 - 21： 6 字节，保留
22 - 23： 2 字节，文件最后修改时间（时分秒，5:6:5）
24 - 25： 2 字节，文件最后修改日期（年月日，7:4:5，年取 0-119 对应 1980 - 2099）
26 - 27： 2 字节，文件首簇号，我们可以根据这个值在 FAT 中找到文件的存储位置
28 - 31： 4 字节，文件的长度，以字节为单位

==========

0 - 7 文件名含义：0 - 目录项为空，可用；E5 - 此文件已经被删除

7 - 10 ：文件名和扩展名为 8.3 格式，如果不够，必需用空格填充，即文件名如果只有 6 个字节，那剩下的 2 个字节必须以空格填充。文件名和扩展名都是大写。

11 属性字节含义：00 - 普通文件；01 - 只读；02 - 隐藏；04 - 系统文件；10(1x) - 该文件是目

录。

4.2.2. 软盘 I/O 读取

介绍完了软盘的结构，下面介绍如何读一个文件：

- 给出一个文件名，比如“KERNEL.SYS”
- 将文件名扩展为“KERNEL SYS”，即去掉点并为文件名和扩展名补充空格
- 读 FDT 到内存中（用 BIOS INT 13）
- 在 FDT 中查找到符合的文件名
- 可选，判断在 FDT 中找到的是否是目录
- 在符合的 FDT 中取出文件首簇号
- 读入 FAT，可以选择读入两个 FAT 表，以检查是否有效
- 将簇号转换为 CHS，将扇区读入内存
- 根据簇号在 FAT 中查找下一簇，并判断是否是文件最后一簇
- 如果是文件最后一簇，则文件读取完毕；如果不是，则转第 8 步

如果在引导程序中读指定的内核文件，则可以省略 1、2 步，直接给出内核文件名即可。如果给出的文件是带目录的，那这里还有必要介绍一下：实际上，目录也是一个文件，只不过这个文件是一个 FDT，FDT 指出该目录下其它文件或目录。因此，给出如下路径：/EXOS/KERNEL.BIN ，则先是在根目录中，将“EXOS ”这个“文件”读出来，然后在读出的 FDT 中找“KERNEL BIN”。为了简单的实现起见，我们直接给定内核文件名”kernel bin”,并放在软盘根目录下。

在知道磁头号，柱面号和起始扇区号后，开始采用以下的函数读取软盘扇区：

```
;-------------------------------------------------------------------------
; 函数名: ReadSector
;-------------------------------------------------------------------------
; 作用:
; 从第 ax  个 Sect or  开始, 将 cl  个 Sector  读入 es :bx  中
ReadSector:
push bp
mov bp, sp
sub esp , 2                    ; 辟出两个字节的堆栈区域保存要读的扇区数: byte [bp-2]
mov byte [bp-2], cl
push bx                        ; 保存 bx
mov bl, [BPB_SecPerTrk]        ; bl: 除数
div bl                         ; y 在 al 中, z 在 ah 中
inc ah                                 ; z ++
mov cl, ah                     ; cl <- 起始扇区号
```

```
mov dh, al                      ; dh <- y
shr al,         1               ; y >> 1 (其实是 y/BPB_Num Heads,里 BPB_NumHeads=2)
mov ch, al                      ; ch <-  柱面号
and dh,  1                      ; dh & 1 =  磁头号
pop b x                         ;  恢复 bx
mov dl, [BS_DrvNum]             ;  驱动器号 (0   表示 A   盘)
.GoOnReading:
mov ah, 2                       ;  读
mov al, byte [bp-2]             ;  读 al   个扇区
int 1   3h
jc .GoOnReading                 ;  如果读取错误 CF   会被置为 1,  这时就不停地读,  直到正确为止
```

可以读取扇区后，就可以读取根目录区的扇区来对软盘上的文件名进行查找。在实际实现过程中，通过引导扇区后查找 loader.bin,加载至 BaseOfStack 地址，在 Loader 中查找 kerneal.bin 文件加载。

查找代码如下:

```
mov sp, BaseOfStack
xor ah,  ah                     ; ┐
xor dl,  dl                     ; ├ 软驱复位
int   13h                       ;  ┘
;  下面在 A   盘的根目录寻找 LOADER.BIN
mov word [wSectorNo], SectorNoOfRootDirectory
LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
cmp word [wRootDirSizeForLoop], 0       ; ┐
jz LABEL_NO_LOADERBIN                   ; ├ 判断根目录区是不是已经读完
dec w  ord [wRootDirSizeForLoop]        ; ┘  如果读完表示没有找到 LOADER.BIN
mov ax, BaseOfLoader
mov es, ax                              ; es <- BaseOfLoader
mov bx, OffsetOfLoader                   bx <- OffsetOfLoader
mov ax, [wSectorNo]                     ; ax <- Root Directory  中的某 Sect or  号
mov cl, 1
call ReadSe ctor

mov si, LoaderFileName    ;              ds:si -> "LOADER    BIN"
mov di, OffsetOfLoader    ;             es:di -> BaseOfLoader:0100 = BaseOfLoader*10h+100
cld
mov dx, 10h
LABEL_SEARCH_FOR_LOADERBIN:
cmp dx, 0                                       ; ┐ 循环次数控制,
jz LABEL_   GOTO_NEXT_SECTOR_IN_ROOT_DIR        ; ├如果已经读完了一个 Sector ,
dec dx                                          ; ┘ 就跳到下一个 Sect or

mov cx, 11
LABEL_CMP_FILENAME:
```

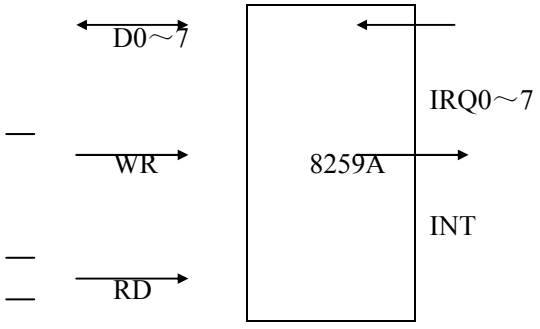
```
cmp cx, 0
jz LABEL_FILENAME_FOUND                ; 如果比较了 11  个字符都相等, 表示找到
dec cx
lodsb                                  ; ds:si -> al
cmp al, byte [es:di]
jz LABEL_GO_ON
jmp LABEL_DIFFERENT          ; 只要发现不一样的字符就表明本 D irectoryEntry  不是
                             ; 我们要找的 LOADER.BIN
LABEL_GO_ON:
inc di
jmp  LABEL_CMP_FILENAME                ;     继续循环

LABEL_DIFFERENT:
and di,  0FFE0h                  ; else ┐  di &= E0  为了让它指向本条目开头
add d i, 20h                     ;      │
mov si, LoaderFileName           ;      ├ d i += 20h    下一个目录条目
jmp LABEL_ SEARCH_FOR_LOADERBIN; ┘

LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:
add  word [wSectorNo], 1
jmp LA BEL_SEARCH_IN_ROOT_DIR_BEGIN

LABEL_NO_LOADERBIN:
mov dh, 2            ;                  "No LOADER."
```

这段代码查找 directoryEntry，查找 loader.bin 的信息，如找到则转至扇区读写函数读取相应内容至 BaseOfLoader 处。在实际应用中可以使用 esp 来传递参数，实现对任意文件的查找功能

4.3. 键盘 I/O

4.3.1. 8259 初始化

首先需要设置 8259，在微机中，利用 8259 芯片来完成中断。8259 芯片逻辑结构见图 4-1。



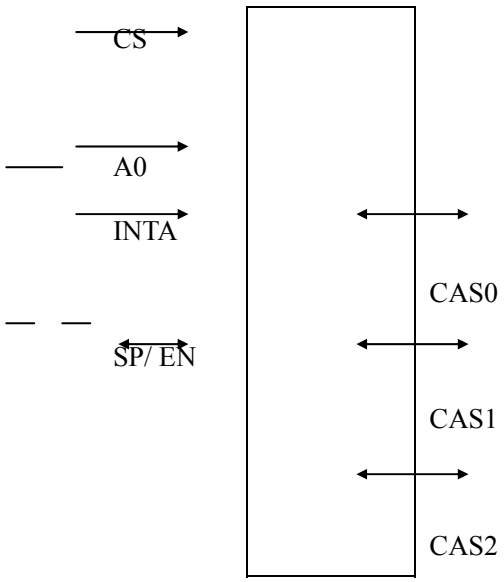


图 4-1 中断控制器 8259 逻辑结构

其中需要关心的为 IRQ0～IRQ7：8 级中断请求输入。

INT：中断请求信号（输出）。用来向 CPU 发中断请求信号。

INTA：中断应答线（输入）。当接收到 CPU 的应答信号后，8259A 就把中断向量类型号送到数据线。

1) 8259A 包括下列主要功能部件：

- 中断请求寄存器（IRR）

它接收外部中断请求。IRR 有 8 位，分别与引脚 IR0～IR7 相对应。

当某一个 IR 端呈现高电平时，则 IRR 的相应位将被置“1”。显然若最多有 8 个中断请求信号同时进入 IR0～IR7 端，则 IRR 的将被置为全“1”。至于被置“1”的请求能否进入 IRR 的下一级判优电路，还取决于控制 IRR 的中断屏蔽寄存器（IMR）中相应位是否置“0”（即不屏蔽该位请求）。

- 中断服务寄存器（ISR）

它用来存放或记录正在服务中的所有中断请求（如在多重嵌套时）。当某一级中断请求被响应，CPU 正在执行它的中断服务程序时，则 ISR 中相应位将被置“1”，并将一直保持到该级中断处理过程结束为止。在多重中断时，ISR 中可能有多位同时被置“1”。 至于 ISR 某位被置“1”的过程是这样：若有一个或多个中断同时请求，它们先由优先级判别器选出当前在 IRR 中置“1”的各种中断优先级别最高者，并用 INTA 脉冲选通送入 ISR 寄存器的对应位。显然当多重中断处于服务过程中时，ISR 中可同时记录多个中断请求。

- 中断屏蔽寄存器（IMR）

它用来屏蔽已被锁存在 IRR 中的任何一个中断请求。对所有要屏蔽的中断请求相应位将置“1”即可。

- 优先级判别器（PR）

它用来判别已进入 IRR 中的各中断请求的优先级。当有多个中断请求同时产生并经 IMR 允许进入系统后，先由 PR 判定当前那一个中断请求具有最高优先级，然后有系统首先响应这一级中断。

2) 8259A 中断过程步骤如下：

- 当一条或多条中断请求线（IRQ0～IRQ7）变成高电平，则使 IRR 相应位置“1”。
- 可用 IMR 对 IRR 进行屏蔽。通过优先级判别器（PR）把当前未屏蔽的最高优先级的中断请求从 INT 输出送到 CPU 的 INTR 端。
- 若 CPU 处于开中断状态，则在执行完当前指令后，用 $\overline{\text{INTA}}$ 作为响应信号。8259A 在收到 CPU 的第一个中断应答 INTA 信号后，将 ISR 中的中断优先级最高的那一位置“1”，而将 IR 中的相应位复位为“0”。
- 8259A 在收到第 2 个 $\overline{\text{INTA}}$ 信号后，将把对应的中断向量送到数据线，CPU 读入该中断向量即可转入执行相应的中断子程序。
- 中断响应结束后，在自动结束中断（AEOI）方式下，8259A 会将 ISR 中原来在第一个 INTA 脉冲到来时设置的“1”在第 2 个 INTA 脉冲结束时自动复位为“0”。若是非自动结束中断（EOI）方式，则该位的“1”将一直保持到中断过程结束，由 CPU 发 EOI 命令才能复位为“0”。

3) 中断类型码中断向量表(IDT)

PC 中的可屏蔽中断源及其相应的中断类型码见下表：

表 4-1 中断向量表

中断类型码	中断优先级	中断源
8 IRQ0		定时计数器
9 IRQ1		键盘
A IRQ2		保留
B IRQ3		异步通信 COM2
C IRQ4		异步通信 COM1
D IRQ5		硬盘

E IRQ6		软盘
F IRQ7		打印机

PC 在内存的前 1024 个字节（地址 00000H～003FFH）建立了一个中断向量表，可以存放 256 个中断向量，每个中断向量占 4 个字节。前两个字节为中断服务程序的入口地址的偏移量，后两个字节为中断服务程序的入口地址的段首地址。使用时这两个字节分别装入 IP 和 CS 中，以转入中断服务程序。每个中断向量用类型码加以区分，当执行中断时，CPU 将类型码乘以 4 得到中断向量地址，进而得到 IP 和 CS 的值，它就是中断服务程序的入口地址，程序由此转入中断服务程序执行。

4) 8259A 的控制字格式

8259A 中的 IRR、ISR 及 IMR 的内容可以通过写控制命令字来加以改变，有两种类型的控制命令字。

- 初始化命令字：（ICW1～ICW4），它们必须在初始化时写入，并且一旦写入，一般在系统运行过程中不再改变。
- 工作方式命令字或操作命令字：（OCW1～OCW4），它们必须在初始化命令后写入。它们用来对中断处理过程作动态控制。在一个系统运行过程中，操作命令字可以被多次设置。

①初始化命令字 ICW1

ICW1 叫芯片控制初始化命令字。写 ICW1 的标记为：A0=0，D4=1。其格式如下：

表 4-2 ICW1 芯片初始化命令字格式

	D7	D6	D5	D4	D3	D2	D1	D0
0	×	×	×	1	LTIM	ADI	SNGL	IC4

ICW1 控制初始化命令字各位的具体含义如下：

D7～D5：这 3 位在 8086/8088 系统中不用，只能用于 8080/8085 系统中。

D4：此位总是设置为 1，表示现在设置的是 ICW1 的标志位。

D3：（LTIM）此位设置中断请求信号的形式。如该位为 1，则表示中断请求为电平触发；如该位为 0，则表示中断请求为边沿触发，且为上升沿触发，并保持高电平。

D2：（ADI）这 1 位在 8086/8088 系统中不用。

D1：（SNGL）这 1 位用来指定系统中用单片 8259A 方式（D1=1），还是用多片 8259A 级联方式（D1=0）

D1:（IC4）这 1 位用来指定后面是否设置 ICW4。若初始花程序中使用 ICW4，则 IC4 必须为 1，否则为 0。

②初始化命令字 ICW2

ICW2 是设置中断类型码的初始化命令字。写 ICW2 的标记为：A0=1。其格式如下：

表 4-3 ICW2 芯片初始化命令字格式

	D 7	D 6	D 5	D 4	D 3	D 2	D 1	D 0
0	A 15/ T7	A 14/ T6	A 13/ T5	A 12/ T4	A 11/ T3	A 10	A 9	A 8

ICW2 控制初始化命令字各位的具体含义如下：

A15～A8 为中断向量的高 8 位，用于 MCS80/85 系统；T7～T3 为中断向量类型码，用于 88/86 系统。中断向量类型码的低 3 位是由引入中断请求的引脚 IR0IR7 决定的。比如设 ICW2 为 40H，则 8 个中断向量类型码分别为 40H、41H、42H、43H、44H、45H、46H 和 47H。中断向量类型码的值与 ICW2 的低 3 位无关。

③ICW4

ICW4　　叫方式控制初始化命令字。写 ICW4 控制字标记为 A0=1。其格式如下：

表 4-4 ICW4 芯片初始化命令字格式

	D 7	D 6	D 5	D 4	D 3	D 2	D 1	D 0
0	0 0		0	S FNM	B UF	M /S	A EOI	μ PM

ICW4 控制初始化命令字各位的具体含义如下：

D7　～D5：这 3 位总为 0，用于表示 ICW4 的识别码。

D4　：（SFNM）如为 1 则为特殊的全嵌套工作方式，如为 0 则为非特殊的全嵌套工作方式。

D3　：（BUF）如此位置 1 则为缓冲方式。所谓缓冲方式是指在多片 8259A 级联的大系统中，8259A 通过总线驱动器和数据总线相联的一种方式。

D2　：（M/S）此位在缓冲方式下用来表示本片是主片还是从片。

D1　：（AEOI）如该位为 1，则设置中断自动结束方式，中断结束后自动复位 ISR。如该位为 0，中断结束后要求 CPU 发 EOI 命令复位 ISR。

D0 ：（μPM）如该位为 1，则表示 8259A 当前处于 8086/8088 系统中；如该位为 0，则表示 8259A 当前处于 8080/8085 系统中。

④操作命令字：（OCW1）

写的标记为 A0=1。其格式如下：

表 4-5 OCW1 芯片初始化命令字格式

	D 7	D 6	D 5	D 4	D 3	D 2	D 1	D 0
0	M 7	M 6	M 5	M 4	M 3	M 2	M 1	M 0

ICW4 控制初始化命令字各位：M7～M0 对应于 IMR 的各位，为 1 表示该位中断被屏蔽，为 0 表示该位允许中断。

一般来讲，可以为 8259 提供一个 base 的 irq 值，比如一般用 0x20，因为 CPU 的例外和陷阱是定值的，而二者都是用的 IDT 进行处理，所以，为了中断号不与 exception 重叠，一般将初始值设为 0x20(32)，这样 8259 的第零个引脚产生的中断在送入 CPU 时就是 0x20，第一个就是 0x21，类推。 初始化 8259 中断控制器代码如下

```
out_byte(INT_M_CTL,  0x11);                          // Master 8259, ICW1.
out_byte(INT_S_CTL,   0x11);                         // Slave    8259, ICW1.
out_byte(INT_M_CTLMASK, IN    T_VECTOR_IRQ0); // Master 8259, ICW2.  设置  '主 8259'  的中断入口地址为 0x 20.
out_byte(INT_S_CTLMASK, IN    T_VECTOR_IRQ8); // Sl ave  82  59, ICW2 .  设置  '从  8259'  的中断入口地址为 0x 28
out_byte(INT_M_CTLMASK, 0x    4);                    // Master 8259, ICW3. IR2  对应  '从 8259'.
out_byte(INT_S_CTLMASK, 0x    2);                    // Slave    8259, ICW3.  对应  '主 8259'  的 IR2.
out_byte(INT_M_CTLMASK,    0x1);                        //Master 82                                    59,
ICW4.out_byte(INT_S_CTLMASK, 0x    1);                // Slave    8259, ICW4.
out_byte(INT_M_CTLMASK,    0xFF);                     // Master 8259, OCW1.
out_byte(INT_S_CTLMASK,    0xFF);                     // Slave    8259, OCW1.
```

其中 out_byte 函数如下

```
out_byte:
mov edx, [esp + 4]                          ; port
mov al, [esp + 4 + 4]                       ; value
out dx,   al
nop                                         ;一点延迟
nop
ret
```

4.3.2. 键盘处理

在初始化 8259 和 IDT 后，可以为键盘中断提供相应的服务程序，代码如下

```
put_irq_handler(KEYBOARD_IRQ, keyboard_handler);    /* 设定键盘中断处理程序 */
enable_irq(KEYBOARD_IRQ);                           /* 开键盘中断 */
```

这样就可以在 keyboard_handler 中添加我们对键盘消息的处理，但事情并没有这么简单，首先介绍一下键盘的工作过程。

在键盘系统中，CPU 并不直接和 Keyboard 进行通信，而是通过一个 8042 芯片或者其它与之兼容的芯片。增加这么一个中间层，就可以屏蔽掉不同键盘之间实现的差别，并可以增加新的特性。CPU 直接和 8042 芯片进行通信，以实现对整个键盘的控制；键盘从外界输入得到的数据也可以通过 8042 芯片通知给 CPU，然后 CPU 可以通过 8042 芯片读取这些数据。另外，CPU 也直接向 8042 芯片发送命令，以使用 8042 芯片自身所提供的功能。

键盘自身也有自己的芯片（Intel8048 及其兼容芯片），此芯片的功能主要是检索来自于 Key Matrix 的外界输入（击键（Press key）或释放键（ReleaseKey））所产生的 Scan code，并将这些 Scan code 存放于键盘自身的内部缓冲；还负责和外部系统（i8042）之间的通信，以及自身的控制（SelfTest, Reset, etc）等等。

对于 PC 机的操作用户来说，与键盘的接口就是键盘上的按键。操作键盘的方式就是敲击这些按键。对于键盘系统而言，操作用户对键盘的敲击分为两种动作：Presskey 和 Release key 。这两个动作之间，还有一个时间段，被称为 Press keydelay。我们将这 2 个动作和 1 个时间段称为一个“击键过程”。

你可以设想一下这个动作——按下一个键，保持一段时间，再松开这个键——这就是你在大多数情况下快速击键动作的一个慢镜头，它明确的出现了 2 个动作和 1 个时间段。但无论你的击键动作有多快，都是上述 2 个动作和 1 个时间段的组合。

对于除了 Pause 键之外的所有键而言，键盘针对 Press Key 和 ReleaseKey 两个动作会分别产生两个 Scan code，被称作 Make Code 和 Break Code。在这两个动作之间的时间段里，会按照一定的频率产生 Repeat code。在大多数情况下，由于你的击键速度非常快，所以不会产生 Repeat code；但在任何情况下，肯定会产生 Make code 和 Break Code。

键盘都有一个 Repeat code 的“产生延迟”设置，这个“产生延迟”指的是两次产生“Repeat code”之间的时间间隔，比如,如果"产生延迟"被设置为 0.25 秒，则当一个键被保持 Press 状态时，键盘系统会每 0.25 秒产生一个针对此键的 Repeat code。

有时候，你会同时按下多个键，对于键盘系统而言，针对这些键的 Press key 动作总有现有之分。但你有可能对这些按键按下之后会保持一段时间才放开。这时候，键盘总是对你最后发生 Press key 动作的键产生 Repeat code。而对你之前按下而没有松开的键，在它的当前“击键过程”内不会再产生 Repeat code，即使在它之后被按下的键都已经完全松开。

比如，你现在按下了"A"键，产生了一个 Press key 的动作，键盘系统会为之产生一个 Make code；随后，你保持按着"A"键不放开，键盘系统将会按照一定的频率产生针对"A"键的 Repeat code。这个时候，你由按下了"B"键，键盘随即停止产生"A"键的 Repeat code，然后产生一个"B"键的 Make code。从此以后，在"A"键的当前“击键过程”中，"A"的 Repeat code 再也不会产生，即使"B"在"A"被 Release 之前松开也是这样。而之后如果“B”被保持按着的话，则键盘系统会按照一定的频率产生"B"的 Repeat code，直到"B"被松开，或又有一个键被 Press 为止。但无论"A"或"B"在任何时候被松开，都必然会产生一个 Break code。

结论是：在键盘被打开的情况下，只要一个键发生了 Press key 的动作，就一定会产生一个 Make code；只要一个键(除了"Pause/Break"键)发生了 Release key 的动作，就一定会产生一个 Break code；无论它们在什么时候发生。而对于 Repeat Code，则有两个条件，一是，到当前的时刻为止，最后被按下的键；二是，这个最后被按下的键，在被松开之前被按的时间超过键盘所设置的 Repeat code“产生延迟”。

迄今为止，IBM PC 键盘共有 3 套 Scan Code Set，最早的 IBMPC/XT 使用 Scan Code Set 1，在随后的系统中，默认的都是 Scan Code Set 2，后来出现了 Scan Code Set3，但并非所有的键盘都支持。为了保证正确性，我们应该以 Scan Code Set2 为开发对象。SCS1 扫描码如下:

表 4-6　SCS1 扫描码

KEY	通码	断码		KEY	通码	断码		KEY	通码	断码
A	1C	F0 1C		9	46	F0 46		[	54	F0 54
B	32	F0 32		`	0E	F0 0E		INSERT	E0 70	E0 F0 70
C	21	F0 21		-	4E	F0 4E		HOME	E0 6C	E0 F0 6C
D	23	F0 23		=	55	F0 55		PG UP	E0 7D	E0 F0 7D
E	24	F0 24		\	5D	F0 5D		DELETE	E0 71	E0 F0 71
F	2B	F0 2B		BKSP	66	F0 66		END	E0 69	E0 F0 69
G	34	F0 34		SPACE	29	F0 29		PG DN	E0 7A	E0 F0 7A
H	33	F0 33		TAB	0D	F0 0D		U ARROW	E0 75	E0 F0 75
I	43	F0 43		CAPS	58	F0 58		L ARROW	E0 6B	E0 F0 6B
J	3B	F0 3B		L SHFT	12	F0 12		D ARROW	E0 72	E0 F0 72

K	42	F0 42		L CTRL	14	F0 14		R ARROW	E0 74	E0 F0 74
L	4B	F0 4B		L GUI	E0 1F	E0 F0 1F		NUM	77	F0 77
M	3A	F0 3A		L ALT	11	F0 11		KP /	E0 4A	E0 F0 4A
N	31	F0 31		R SHFT	59	F0 59		KP *	7C	F0 7C
O	44	F0 44		R CTRL	E0 14	E0 F0 14		KP -	7B	F0 7B
P	4D	F0 4D		R GUI	E0 27	E0 F0 27		KP +	79	F0 79
Q	15	F0 15		R ALT	E0 11	E0 F0 11		KP EN	E0 5A	E0 F0 5A
R	2D	F0 2D		APPS	E0 2F	E0 F0 2F		KP	71	F0 71
S	1B	F0 1B		ENTER	5A	F0 5A		KP 0	70	F0 70
T	2C	F0 2C		ESC	76	F0 76		KP 1	69	F0 69
U	3C	F0 3C		F1	05	F0 05		KP 2	72	F0 72
V	2A	F0 2A		F2	06	F0 06		KP 3	7A	F0 7A
W	1D	F0 1D		F3	04	F0 04		KP 4	6B	F0 6B
X	22	F0 22		F4	0C	F0 0C		KP 5	73	F0 73
Y	35	F0 35		F5	03	F0 03		KP 6	74	F0 74
Z	1A	F0 1A		F6	0B	F0 0B		KP 7	6C	F0 6C
0	45	F0 45		F7	83	F0 83		KP 8	75	F0 75
1	16	F0 16		F8	0A	F0 0A		KP 9	7D	F0 7D
2	1E	F0 1E		F9	01	F0 01		]	58	F0 58
3	26	F0 26		F10	09	F0 09		;	4C	F0 4C
4	25	F0 25		F11	78	F0 78		'	52	F0 52
5	2E	F0 2E		F12	07	F0 07		,	41	F0 41
6	36	F0 36		PRNT SCRN	E0 12 E0 7C	E0 F0 7C E0 F0 12		.	49	F0 49
7	3D	F0 3D		SCROLL	7E	F0,7E		/	4A	F0 4A
8	3E	F0 3E		PAUSE	E1 14 77 E1 F0 14 F0 77	-NONE-				

了解键盘基本工作过程后，根据不同的扫描码，解析扫描码并进行相应处理。修改系统的键盘中断服务程序如下:

首先建立一缓冲区

```
/* Keyboard structure, 1 per console. */
    typedef struct s_kb {
        char*    p_head;            /* 指向缓冲区中下一个空闲位置 */
```

```
    char*   p_tail;         /* 指向键盘任务应处理的字节 */
    int count;              /* 缓冲区中共有多少字节 */
    char    buf[KB_IN_BYTES];   /* 缓冲区 */
}KB_INPUT;
```

然后 keyboard_handler 如下，使用 in_byte 函数读取扫描码存放在缓冲区中

```
/*======================================================================*/
                                keyboard_handler
 /*====== ======================================================================*/
PUBLIC void keyboard_handler(int irq)
{
    t_8 scan_code = in_byte(KB_DATA);

    if (kb_in.count < KB_IN_BYTES) {
        *(kb_in.p_head) = scan_code;
        kb_in.p_head++;
        if (kb_in.p_head == kb_in.buf + KB_IN_BYTES) {
            kb_in.p_head = kb_in.buf;
        }
        kb_in.count++;
    }
}
```

使用

#define MASK_RAW　0x01FF

#define FLAG_EXT　0x0100 与其它掩码后获得对应字符的 ascii 值，在输出部分显示。具体函数见 os/kernel/keyboard.c/keyborad_read()函数

4.4. 显示 I/O

4.4.1. 文本模式简介

文本模式下的显存地址是从 B800:0000 开始的

COLOR 10,3

PRINT "TEST"

这个语句就会调用一些过程比如调用 INT21 号相关的中断调用，这个相关的调用又调用了 BIOS INT10 中断.....最终的表现形式是反映到显存里，当显存里的数据发生变化的时候，显示器显示的内容也跟着发生变化，所以我们可以绕过这个语句，直接写这个显存，也可以同样达到这个效

果，但要注意显存的组织方式

显示属性定义

B800:0000——长度 2000H 字 彩色显示器文本模式的显示缓冲区。每个字中的低字节是字符的 ASCII 码，高字节是其属性（ 背景属性 前景属性 ）其组织方式如下（80×25 的 16 色文本模式）：

B800:0000 ;第一行第一列的字符的 ASCII 码

B800:0001 ;第一行第一列的字符的属性字节

B800:0002 ;第一行第二列的字符的 ASCII 码

B800:0003 ;第一行第二列的字符的属性字节

... ...

B800:07CE ;第 25 行第 80 列的字符的 ASCII 码

B800:07CF ;第 25 行第 80 列的字符的属性字节

属性字节的定义如下：

低四位是表示前景色，也就是字符颜色　　高四位中位为 1 表示闪烁，其它表示背景色

位 3 位 2 位 1 位 0	位 6 位 5 位 4
0 0 0 0 黑色	0 0 0 黑色
0 0 0 1 蓝色	0 0 1 蓝色
0 0 1 0 绿色	0 1 0 绿色
0 0 1 1 青色	0 1 1 青色
0 1 0 0 红色	1 0 0 红色
0 1 0 1 洋红	1 0 1 洋红
0 1 1 0 棕色	1 1 0 棕色
0 1 1 1 白色	1 1 1 白色
1 0 0 0 灰色	
1 0 0 1 亮蓝色	
1 0 1 0 亮绿色	
1 0 1 1 亮青色	
1 1 0 0 亮红色	
1 1 0 1 亮洋红	
1 1 1 0 黄色	
1 1 1 1 亮白色	

文本输出设置代码部分如下

```
Mov ah, 07h        ;                    0000: 黑底        0111 白字
Mov al, `p`                             ; 输出 p
Mov [gs:edi], ax                        ; 向对应显存输出
```

4.4.2. 寄存器操作

VGA 系统有很多寄存器，包括外部寄存器，CRT 控制寄存器，图形控制寄存器等，部分寄存器地址如下：

表 4-7 VGA 系统寄存器列表

VGA 寄存器地址表		
寄存器名称	单色显示模式	彩色显示模式
混合输出寄存器	3C2H	3C2H (VGA 读端口 3CCH)
待征控制寄存器 3	BAH	3DAH (VGA 读端口 3CAH)
输入状态寄存器 3C2H		3C2H
输入状态寄存器 1 3	BAH	3DAH
VGA 允许寄存器 3C3H		3C3H
定序器索引寄存器 3C4H		3C4H
定序器数据寄存器 3C5H		3C5H
CRT 控制索引寄存器	3B4H 3D	4H
CRT 控制数据寄存器	3B5H 3D	5H
图形控制索引寄存器	3CEH 3CEH	
图形控制数据寄存器	3CFH 3CFH	
属性控制器 3C0H		3C0H
DAC 接口寄存器 3C7H -3C9H		3C6H-3C9H
CRT 控制器寄存器列表		

寄存器名称	索引号(Hex)	写端口(EGA／VGA／TVGA)	读端口(EGA)	读端口(VGA／TVGA)
地址寄存器 --		3D5H／3B5H		3D4H／3B5H
水平扫描总时间	00H 3D	5H／3B5H		3D5H／3B5H
水平显示结束 01	H	3D5H／3B5H	3D	5H／3B5H
水平消隐开始 02	H	3D5H／3B5H	3D	5H／3B5H
水平消隐结束 03	H	3D5H／3B5H	3D	5H／3B5H
水平回扫开始 04	H	3D5H／3B5H	3D	5H／3B5H
水平回扫结束 05	H	3D5H／3B5H	3D	5H／3B5H
垂直扫描总时间	06H 3D	5H／3B5H	3D	5H／3B5H
溢出 07H				3D5H／3B5H
行扫描预置 08H		3D5H／3B5H	3D	5H／3B5H
最大扫描行 09H		3D5H／3B5H	3D	5H／3B5H
光标起始 0A	H	3D5H／3B5H	3D	5H／3B5H
光标结束 0BH		3D5H／3B5H	3D	5H／3B5H
显存起始地址(高)	0CH 3D	5H／3B5H 3D	5H／3B5H	3D5H／3B5H
显存起始地址(低)	0DH 3D	5H／3B5H 3D	5H／3B5H	3D5H／3B5H
光标位置(高位)	0EH	3D5H／3B5H 3D	5H／3B5H	3D5H／3B5H
光标位置(低位)	0FH	3D5H／3B5H 3D	5H／3B5H	3D5H／3B5H
垂直回扫开始 10	H	3D5H／3B5H		3D5H／3B5H

垂直回扫结束 1	1H	3D5H / 3B5H		3D5H / 3B5H
光笔地址〔高位)	10H	仅 EGA 有效只读	3D5H / 3B5H	3D5H / 3B5H
光笔地址(低位)	1 1H	仅 EGA 有效只读	3D5H / 3B5H	3D5H / 3B5H
垂直显示结束 12	H	3D5H / 3B5H		3D5H / 3B5H
偏移/逻辑屏宽度	13H 3D	5H / 3B5H	3D	5H / 3B5H
下划线位置 14H		3D5H / 3B5H		3D5H / 3B5H
垂直消隐开始 15	H	3D5H / 3B5H	3D	5H / 3B5H
垂直消隐结束 16	H	3D5H / 3B5H	3D	5H / 3B5H
模式控制 17H		3D5H / 3B5H	3D	5H / 3B5H
行比较 18H		3D5H / 3B5H		3D5H / 3B5H

我们暂时需要关心的是 0x3d5 端口，但是其中对应的 CRT 控制寄存器很多，如何通过一个端口获得对不同寄存器的访问？可以用到 address re gister,首先向 AR0x3D4 写入一个对应的索引值,然后通过 0x3D5 进行的操作就是针对向对应索引值寄存器的操作了代码如下：

```
Out_byte(0x3D4, idx);
Out_byte(0x3D5, new_value);
```

对照寄存器描述，可以看出其中比较有用的为: 显存起始地址(高)，显存起始地址(低)，光标位置(高位)，光标位置(低位)。通过这 4 个寄存器可以实现光标位置的设置，对应终端滚屏等功能。在 os/kernel/console.c 中实现了设置光标位置的函数如下：

```
/*======================================================================*
                              set_cursor
 *======================================================================*/
PRIVATE void set_cursor(unsigned int position)
{
```

```
disa      ble_int();
o      ut_byte(CRTC_ADDR_REG, CRTC_DATA_IDX_CURSOR_H);
      out_byte(CRTC_DATA_REG, (position >> 8) & 0xFF);
o      ut_byte(CRTC_ADDR_REG, CRTC_DATA_IDX_CURSOR_L);
      out_byte(CRTC_DATA_REG, position & 0xFF);
ena      ble_int();
}
```

其中宏定义如下

```
/* VGA */
#define CRTC_ADDR_REG        0x3D4    /* CRT Controller Registers - Address Register */
#define CRTC_DATA_REG        0x3D5    /* CRT Controller Registers - Data Registers */
#define CRTC_DATA_IDX_START_ADDR_H  0xC /* r egister  index  of  video m  em start a  ddress (MSB)*/
#define CRTC_DATA_IDX_START_ADDR_L  0xD /* register index of video mem start address (LSB)*/
#define CRTC_DATA_IDX_CURSOR_H    0xE  /* register index of cursor position (MSB) */
#define CRTC_DATA_IDX_CURSOR_L    0xF  /* register index of cursor position (LSB) */
```

所谓的显存开始地址，可以这么理解，如果整体现存空间从 0xB8000 开始共有 32KB，在 80*25 文本模式下一个屏幕的大小约为 80*25*2=4000 字节约 4kb.可以明显看出显存可以容纳多个屏幕，即可以通过设置现存开始地址寄存器来控制当前屏幕显示的内容范围。最常见的用途即实现滚屏功能。

doggyOS 实现了对显存的循环利用，基本的滚屏原理如下，如果当前光标地址超过了当前屏幕的显示范围，则将现存开始地址加一行。如果显存开始地址加屏幕大小超过了定义的现存限制（由多个终端引起的空间分配），则将所有当前内容 copy 至最初的显存开始地址(不同的终端所以开始地址不一定都为 0xb8000)，重新设置光标位置和当前现存开始地址。基本滚屏的函数如下：

```
/*======================================================*
                                scroll_screen
 *---------------- ----------------------------------------------------*
  滚屏.
 *---------------- ----------------------------------------------------*
direction:
S      CROLL_SCREEN_UP     : 向上滚屏
S      CROLL_SCREEN_DOWN       : 向下滚屏
      其它                : 不做处理
*=== ================================================================*/
PUBLIC void scroll_screen(CONSOLE* p_con, int direction)
```

```
{
      if (direction == SCROLL_SCREEN_UP) {
            if (p_con->current_start_addr > p_con->original_addr) {
  p_con->curr                 ent_start_addr -= SCREEN_WIDTH;
            }
}
      else if (direction == SCROLL_SCREEN_DOWN) {
 if            (p_con->current_start_addr + SCREEN_SIZE < p_con->original_addr + p_con->v_mem_limit) {
  p_con->curr                 ent_start_addr += SCREEN_WIDTH;
            }
            else{
  disp_mem_copy                   (p_con);
  p_con->curr                 ent_start_addr += SCREEN_WIDTH;
            }
}
 flush(p_con);
}
```

其中第 2 个判断条件实现了对超出现存限制的判断，disp_mem_copy 函数实现对当前内容的移动 copy.

4.5. TTY&SHELL

4.5.1. TTY 任务

doggyOS 中实现了多个 TTY 的系统，以及 TTY 与单一 console 的对应。在 TTY 任务中轮询每一个 TTY，查询对应该 TTY 的 console 是否为当前显示的 console. 如果是，则监听键盘中断和读取缓冲区，并显示在屏幕上。在整个 TTY 任务中包括 keyboard_read(), i n_process(),keyboard_write()等函数。TTY 任务示意图如下：

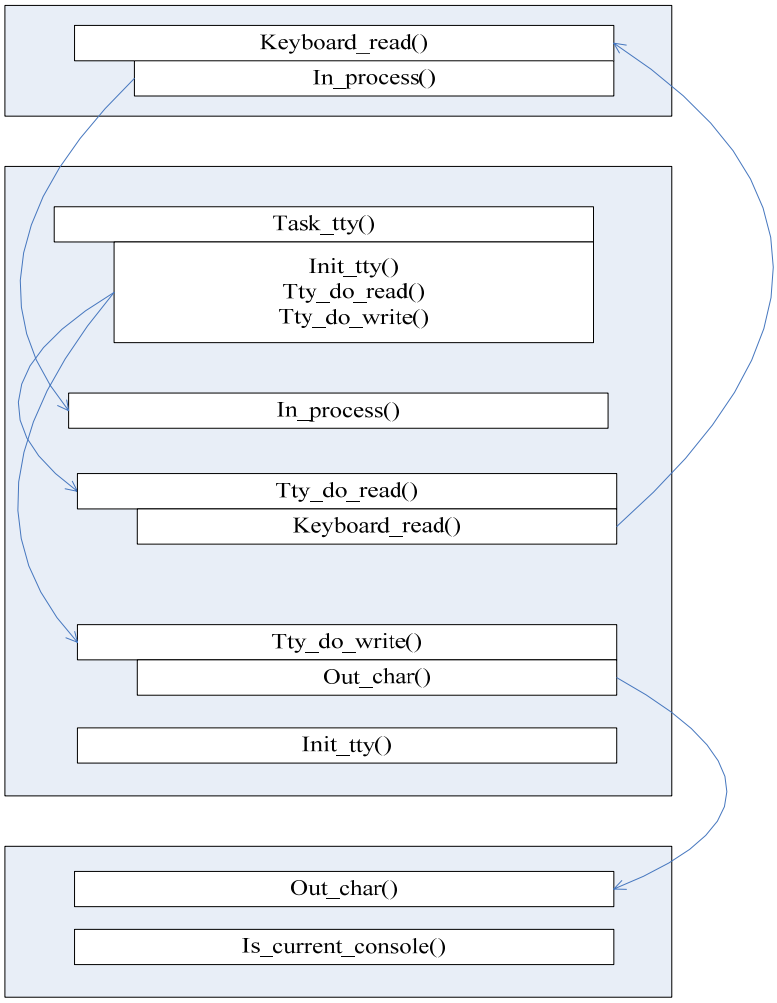


图 4-2　Tty 任务流程图

下面为 TTY 和 CONSOLE 的数据结构，其基本意义如注释所写，比较明确：

```
/* TTY */
typedef struct s_tty
{
 t_32       in_buf[TTY_IN_BYTES];          /* TTY  输入缓冲区 */
 t_3    2*      p_inbuf_head;              /*  指向缓冲区中下一个空闲位置 */
 t_3    2*      p_inbuf_tail;              /*  指向键盘任务应处理的键值 */
     int    inbuf_count;                   /*  缓冲区中已经填充了多少 */
 t_3    2 cmd_buf[TTY_IN_BYTES];           /* tty command buff space */
 t_3    2*      p_cmdbuf_head;             /* point to the next node */
 int        cmdbuf_count;                  /* num of the character */

     struct s_console *    p_console;
```

```
}TTY;

/* CONSOLE */
typedef struct s_console
{
 //stru      ct s_tty*  p_tty;
 unsi      gned int    current_start_addr;              /*  当前显示到了什么位置     */
 unsi      gned int    original_addr;              /*      当前控制台对应显存位置 */
 unsi      gned int    v_mem_limit;                /*  当前控制台占的显存大小 */
 unsi      gned int    cursor;                     /*  当前光标位置 */
 unsi      gned int    line_start;                 /*  当前行开始位置 */
 unsi      gned int    state;
 unsi      gned char    name[6];                   /*  当前登录名 */
}CONSOLE;
```

在对应键盘的处理函数中加入了对不同 TTY-console 之间的切换。暂定 alt+f1，f2, f3 为切换按钮。在输入时使用 tty 结构中的缓冲区存储，在 write 过程中清空，由于不考虑多用户同时登录的情况，因此没有对该缓冲区实现进程同步之间的保护，只是简单的存储功能，一片可以判断出用户的组合键操作。

Console 数据结构是实现终端输出的基本结构，其内部定义了有关显存的和光标位置的存储结构，其文件内部封装了对显存的操作函数，可以通过改变 console->current_start_addr 来改变终端屏幕的显示范围。终端的滚屏操作也据此而成。

在 console 中还开辟了保留当前终端状态的 state 和登录用户名的 name 结构，主要用来实现 shell 的登录功能和状态转化。基本思想是在 in_process 中除了处理输出的任务外，在 tty 的 cmd 缓存中存储相应的命令字符，在 shell 的词法分析过程中对其分析，如果是系统命令则执行，如果是可执行文件则交由相应模块 fork 新进程来执行，如果是其他信息则提示相应的错误。

4.5.2. 系统调用及命令

为了实现不同运行级任务对 I/O 的基本调用，I/O 模块实现了 printf()的系统调用，其实际的调用过程如下图:

图 3　图 4-3 printf 调用过程

在 printf()中调用了 tty_write()，将需要输出的字符放入了 tty 任务对应 console 的输出缓存，在 tty 执行输出任务时会显示相应信息。这个输出缓存与 shell 的命令缓存不同，之所以要分割为两个缓存，主要考虑到现有的 shell 环境并没有实现脚本分析功能，如果采用一个缓存，那么用户的输入信息和程序的输出显示信息会干扰 shell 对命令的词法分析。在此基础上，shell 实现了一些基本的系统命令，列表如下：

表 4-8 shell 实现命令列表

ls	打印所有已有文件
ll	打印文件详细信息
ps	打印所有执行的进程
kill	杀死一个进程
lu	打印所有的用户
who	打印所有登入用户信息
quit	注销用户
clean	清屏

doggyOS 的 shell 环境不够完善，但实现了基本的思路，对用户输入作了词法分析，以区分用户信息和系统命令，并可以对可执行文件进行执行调度。

参考文献

[1] 于渊. 自己动手写操作系统. 电子工业出版社. 2005.8

[2] 赵炯. Linux 内核完全注释. 机械工业出版社. 2004

[3] Barry B. Brey. Intel 微处理器结构编程与接口 (第六版). 电子工业出版社. 2004.1

[4] IA-32 Intel Architecture Software Developer's Manual – Volume 1: Baisic Architecture. 2004

[5] IA-32 Intel Architecture Software Developer's Manual – Volume 2: Instruction Set Reference. 2004

[6] IA-32 Intel Architecture Software Developer's Manual – Volume 3: System Programming Guide. 2004

[7] 李善平. 边干边学: Linux 内核指导. 浙江大学出版社. 2002-8

[8] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. Operating System Concepts (Siz Edition). 高等教育出版社. 2002-5