Linux0.11源码分析

Version 0.1

潘晓雷 著

Linux0.11 源码分析

Version 1.0

东北大学 信息学院 计算机

潘晓雷

pan_xiaolei@sina.com

http://pan_xiaolei.go.nease.net

二〇〇四年四月二十四日

目录

序言

写了这么久，终于可以述苦了！ 

我是从 2 月 25 日开始写的，开始仅仅是有个想法，为了逃避现实的折磨于是将它列入了日程。今天是 4 月 24 日，第一版已经写完了。我为它的完成感到快乐，我也为它的完成感到难过。因为，我是带着种种的幻想来写它的。当理想与现实差距很远的时候，自然会对现实感到一丝恐惧。但是无论如何，第一版终于在今天－－4 月 24 日完成了。这是它诞生的一天，因此我愿将它做为一份真诚的礼物送给所有在这一天过生日的人：

祝你生日快乐 

潘晓雷

2004 年 4 月 24 日 9 时

我是利用晚上时间来分析的，加上节假日，大约用了 200 个小时。在分析的过程中我学习到了非常非常多的东西。

Linux 系统的版本那么多，为什么要分析 Linux0.11 这个超级低的版本呢？答案是简单。高版本的代码量实在是让人心寒，读完有瞎子摸象的感觉。我之所以想分析操作系统是因为我不理解它，不明白那么多不可思议的功能是如何实现的。想想看，我们用了这么多年的计算机操作系统是如何让它启动起来的呢？都说现在的操作系统是分时的、并行的，它是如何实现的呢？让一个程序挂起，让另一个程序运行可能！？甚至一个函数还可能返回两个不同的值！原理课上说，操作系统真的是这么做的，我相信但是我不理解，因为听说而已，未曾亲见。当我看完了进程的管理之后，我认为我应该算是理解了。不能说理论课不重要，相反，理论课的学习是太重要了。没有原理课上讲的知识，阅读代码的时候会发现，仅仅知道代码这么写会执行什么操作，但是不知道为什么要这么执行，理解起来很是困难。

我认为，对于学计算机专业的同学来说，看一遍操作系统的源码是非常非常有用的，它能为以后的学习打下非常好的基础。以前，我用 MFC 写过一些 Windows 的小小程序，有一种在云雾中行走的感觉，总是认为自己写的程序**好像**是正确的。当时的感觉就是，编写 Windows 的应用程序与以往的程序编写方法完全不同，好像在 Windows 中自己就失去了对计算机的控制能力，完全依偎在操作系统的怀里。有一本书《win32 汇编语言程序设计》写得很好，讲解如何在 Win32 的环境下使用汇编语言。当时认为，实在是太高深没测了。还有，当时在听嵌入式系统的设计的时候真是雾里看花，我不知道老师在将什么。因为不懂的实在是太多了。

现在，我的感觉是，计算机的世界不再那么的神秘了，一切都好像变得有些透明了，就想粘了油的纸。同时，分析操作系统使我对 C 语言和汇编语言也有了一个全新的认识，感觉自己好像理解了 C 语言。这一切我实在是无法用语言来描述，这是一种感觉，没有读过源码的人是不会有的。我可以保证，只有你能够比较认真的阅读代码的话，动手能力一定会显著的提高。看完了，Linux0.11 的代码，你会说：“我终于入门了！”

如果时间允许，阅读高版本的 Linux 自然是非常好的。但是，我认为我来说这个性价比

太低了。如果我在这段时间里去分析 Linux2.*.*的代码，也许刚刚搞明白它的启动过程和系统的结构。而 Linux0.11 却让我知道了什么是操作系统。不知道实际情况如何，但我现在的感觉是，如果给我 2 个月的时间，我也可以写一个能够运行的操作系统的。

写这本书的目的绝对绝对不是让你不去读源代码，而是让你读代码的效率大大的提高。如果不去阅读代码，这本书没有任何的实际意义。由于，操作系统的实现涉及到了很多很多杂乱的知识。我当时为此而在网上东奔西跑，走了很多弯路，固然浪费了很多时间。为了节约你的时间，我将阅读的时候遇到过的困难写了下来。同时，我为这本书建立了一个 FTP 服务器，将我参考过的文献资料以及一些工具和试验的代码共享了上面。

在网上关于高版本 Linux 的分析可以说是海量的，但是关于 Linux0.11 的我就知道一本，是中文的，叫做《Linux0.11 版本完全分析》。因此我想在此感谢他为我们初学者做出的贡献。

我是在《Linux0.11 版本完全分析》这本书的启蒙下才开始阅读 Linux0.11 的代码的，这本书注释的已经很详细了。希望大家在看我的这本书的时候，也多看看他的。我在对代码注释的时候，大部分是写下了我，以一个初学者的角度，对相应代码的理解。虽然有这本书给了注释，但是实际分析的时候困难还是非常的多的。因为，我知道的太少了，需要查阅的资料太多了。这可能是每一位初学者的感觉。因此，我写下了很多我对某一个问题的理解，例如进程的创建、休眠等等。很多理解是看了这本书的注释，弄懂了代码才有的。有些注释是从这本书中粘贴过来的。如果我认为一段代码没有什么新东西可说，而不理解代码又会影响后面的阅读，为了让读者方便，我才这么做的。这本书的作者是赵炯博士，再次非常感谢他在我遇到困难的时候给我鼓励和帮助。他的网站是：http://oldlinux.org 这网站非常好，有很多资料。赵博士是一位非常非常热心的人，在百忙之中还乐于给初学者回答问题。

另外，在 Linux0.11 中的确是有一些 bug 的。对于我知道的，我写下了我的观点。

我认为 Linux0.11 可以分成两大块：进程系统和文件系统（内存管理处在进程系统中）。进程系统由我负责，文件系统由于浚泊分析。由于我比较闲，就先草草的分析完了。同时，老师留了分析源码的作业，所以我就先发表一个版本吧。

我的想法是让读这本书的人能够很好的理解它，因此我以我的一个同学为基准读者。她有着非常好的理论基础，但是动手能力相对弱了一些。因此，在有些实现的细节琐事上浪费了很多笔墨。我开始的时候先将所有我参考过的资料都附带上，但是那样这本书就会有近千页了，而且全是别人的东西，最后还要以编著结尾。国内太多的书都是编著了，反正我是尽量将自己的想法写进来，虽然也有别人的东西，但毕竟是少数，所以用了“著”这个字。也不知道是否准确。另外，我建议大家使用 Adobe Reader 6.0 来阅读这本书，因为效果比较好。

好了，还是要送大家一句话：

“RTFSC – Read The F**king Source Code ☺!”

–Linus Benedict Torvalds

预祝大家有巨大的收获！也希望能够与大家共同交流。☺

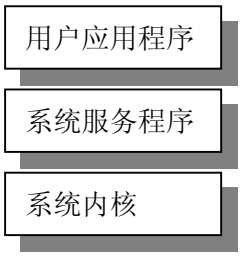
我的 ftp 地址：ftp://202.118.13.14 (主楼抢 IP 现象比较严重，因此 IP 有时会变化，请原谅)
e-mail: pan_xiaolei@sina.com
phone: 83670837

潘晓雷
二〇〇四年四月十九日

第一章概述

Linux 操作系统的版本很多，0.11 版是 Linus 第一个正式发布的版本，代码量在 1 万行左右。它包括了进程的管理、内存管理、设备管理和文件管理。在这个操作系统上可以运行.out 格式的可执行文件，甚至运行批处理文件，支持多进程。它利用了很多在原理课上提到的技术。但是，由于这个版本很低，因此也有许多技术它没有使用。例如：线程、交换内存、实现进程同步和互斥的 PV 操作等。同时，它仅仅支持 Minix1.0 的文件系统，也没有对网络和虚拟文件系统的支持。但是，这些不足不会影响你对操作系统的入门的。

操作系统是非常的复杂的，但是在 Linux0.11 中又显得那么的简单。对于用户来说，操作系统的结构是：



这里分析的仅仅是系统内核的代码，因为无论是系统服务程序还是用户应用程序，都是基于内核提供的调用来完成一些基本功能的。而这些程序的种类又千变万化，最主要的是分析它们对了解操作系统作用不是很大。需要注意的一个系统服务程序是 sh，它位于文件系统盘中。就好像 Windows 系统中的资源管理器一样，用户可以利用它来完成一些对操作系统的基本操作，例如启动一个程序等。

为了方便大家分析代码，我对 Linux0.11 中的几个文件夹做一个简要的说明：

boot:计算机的启动代码都放到了这里。Bootsec.s, setup.s, head.s

init:这里就一个文件 main.c。操作系统的初始化工作都是在这里完成的。

include:对于各种函数的声明，各类的宏定义都在这里。

kernel：与进程系统有关的各种实现几乎都在这里，另外一些底层设备的操作也在这里。

mm:对于内存的管理的文件位于这里。

Lib：链接的时候会用到的库。

Fs：文件系统的实现。

Linux0.11 中的进程模型与原理中的不太一样，不存在进程内容从内存向磁盘的转移。等待的时候也分成了两种——可中断等待和非可中断等待。同时多了两个状态 zombie 和 stop 表示即将退出运行和暂停运行的进程。

对于内核代码来说，按照功能来划分包括进程管理、内存管理、设备管理、文件管理，四个部分。模块间的交互关系如下：

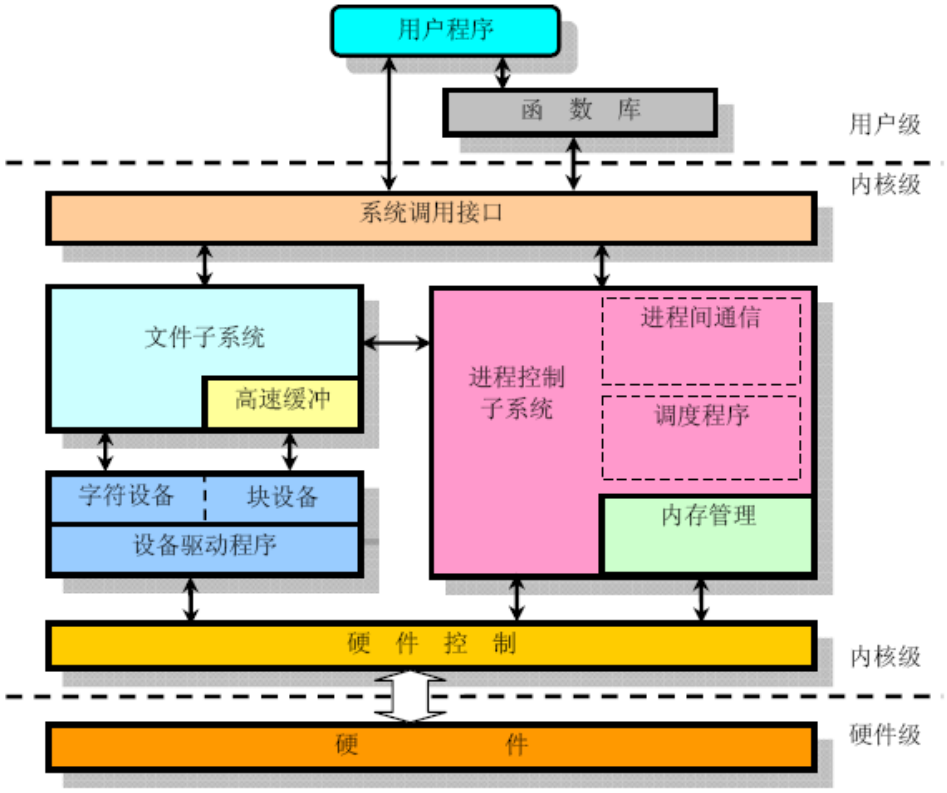


图 一.1 模块间的交互关系（出自于《Linux0.11 源码完全注释》）

可见，应用程序最终是利用系统调用来完成与操作系统的交流的。

第二章系统的引导

引导的综述

当 PC 启动时，Intel 系列的 CPU 首先进入的是实模式，CS 寄存器的值为 0xffff,IP 寄存器的值是 0x0000。因此 CPU 开始的时候是执行位于地址 0xFFFF0 处的代码，也就是 ROM-BIOS 起始位置的代码。BIOS 先进行一系列的系统自检，然后初始化位于地址 0 的中断向量表。最后 BIOS 将启动盘的第一个扇区（512 字节）装入到绝对内存地址 0x07C00，并跳转到这里开始执行此处的代码。这里说的启动盘既可以是硬盘也可以是软盘（其他的东西我不懂）。这就是对计算机启动过程的一个最简单的描述。

当 Linux 系统开始运行时，将自己装入到绝对地址 0x90000，再将其后的 2k 字节（setup 文件）装入到地址 0x90200 处，最后将内核文件装入到 0x10000。这个过程是由系统的引导程序完成的，这个程序是用汇编语言编写的，放在启动盘的第一扇区，由 BIOS 加载到内存当中。当引导程序将系统装入内存时，会显示“Loading...”这条信息。装入完成后，控制转向另一个实模式下运行的汇编语言代码/boot/Setup.S。

Setup.s 部分首先设置一些系统的硬件设备，然后将内核文件从 0x10000 处移至 0x00000 处。这时系统转入保护模式，开始执行位于 0x00000 处的代码。 内核文件的头部是用汇编语言编写的代码，对应的文件是/boot/head.s。

Head.s 这个部分会将 IDT（中断向量表）、GDT（全局段描述符表）和 LDT（局部段描述表）的首地址装入到相应的寄存器中，初始化处理器，设置好内存页面，最终调用/init/main.c 文件的 main()函数，这个函数是用 C 语言编写的。这大概是整个内核中最为复杂的部分。

看完了这一章之后，你也可以用汇编做一个简单的引导程序。具体的实现方法可以参见《操作系统引导探究》一书。

下图形象的说明了这个流程：

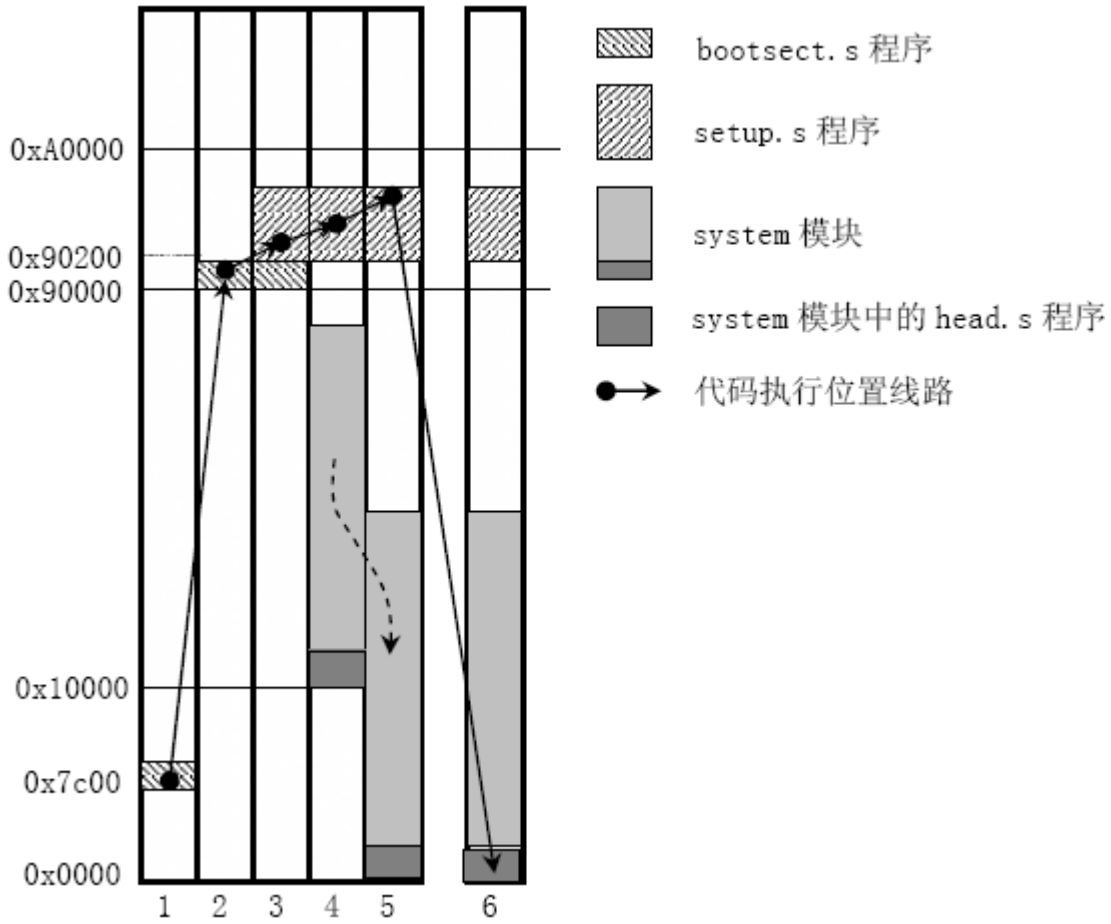


图 二.1 文件加载过程（出自于《Linux0.11 源码完全注释》）

代码综述

Bootsect.s 和setup.s 采用近似于Intel 的汇编语言语法，需要使用Intel 8086 汇编编译器as86和连接器ld86，而head.s 则使用GNU 的汇编程序格式，需要用GNU 的as 进行编译,这是一种AT&T 语法的汇编语言程序。

为了便于理解下面的代码，我在会在代码中简要的介绍一些汇编语言的语法。在看代码时建议多看看汇编的书。我参考的是东北大学高福祥老师的《汇编语言程序设计》这本书。

在看Linux汇编代码的时候可以发现，这几个汇编文件中所有的变量定义都是放在执行代码的后边，这与通常的汇编教材中给出的例子不太一样。这是因为这里的代码都是用于引导的，就是在没有任何的操作系统的情况下让计算机执行的代码。所以执行的时候就是从文件的开头顺序执行，如果将变量放到了代码的前面就会使得计算机执行发生错误。但是，如果将其变为.exe文件格式就不会发生错误了，因为它是由相应的操作系统来解释执行的。

为了便于理解为什么要将变量的定义放在程序的尾部，我们来看一个小程序。

先建立一个汇编文件test.asm，输入：

```
CSEG    SEGMENT
    ASSUME  CS:CSEG
START:
    MOV AL, DATA
DATA    DB  0FH
```

```
CSEG    ENDS
    END START
```

这个程序没有实际意义，仅仅是将DATA中的值0F放入AL寄存器中。这里是将变量的定义放到了后边。编译成TEST.COM文件后，用DEBUG反汇编得到：

```
****:0100   2E  CS:
****:0101   A00400  MOV AL, [0004]
****:0104   0F
```

TEST.COM的数据是2EA004000F，数据0F放在文件的最后边。可见与我们想要的到的程序是相同的。

我们再建立一个文件TEST2.ASM，输入：

```
CSEG    SEGMENT
    ASSUME  CS:CSEG
START:
DATA    DB  0FH
    MOV AL, DATA
CSEG    ENDS
    END START
```

同TEST.ASM文件一样，将其制作成TEST2.COM。反汇编后得到

```
****:0100   0F  DB  0F
****:0101   2E  CS:
****:0102   A00400  MOV AL, [0000]
```

如果这个程序是从****:0102开始执行的，它就是正确的。但是，它是只可能从程序头部顺序执行。0F没有相应的机器指令因此，这个程序就错误了。想让其正确执行简单，只需要加入一个跳转指令使其，跳到程序真正的起始点就可以了。

因为这个原因，Linux的引导代码中变量的定义都是放在了代码后部。

Bootsec.s 文件的说明

这个文件编译后放到引导盘的第一个物理扇区中，用于将 setup.s 和 head.c 文件加载到内存中。如果用第二张软盘做文件系统盘，当将内核文件加载完成后，就需要对软驱进行测试。这个代码就可以添加在这个文件的尾部。

代码的简要流程图

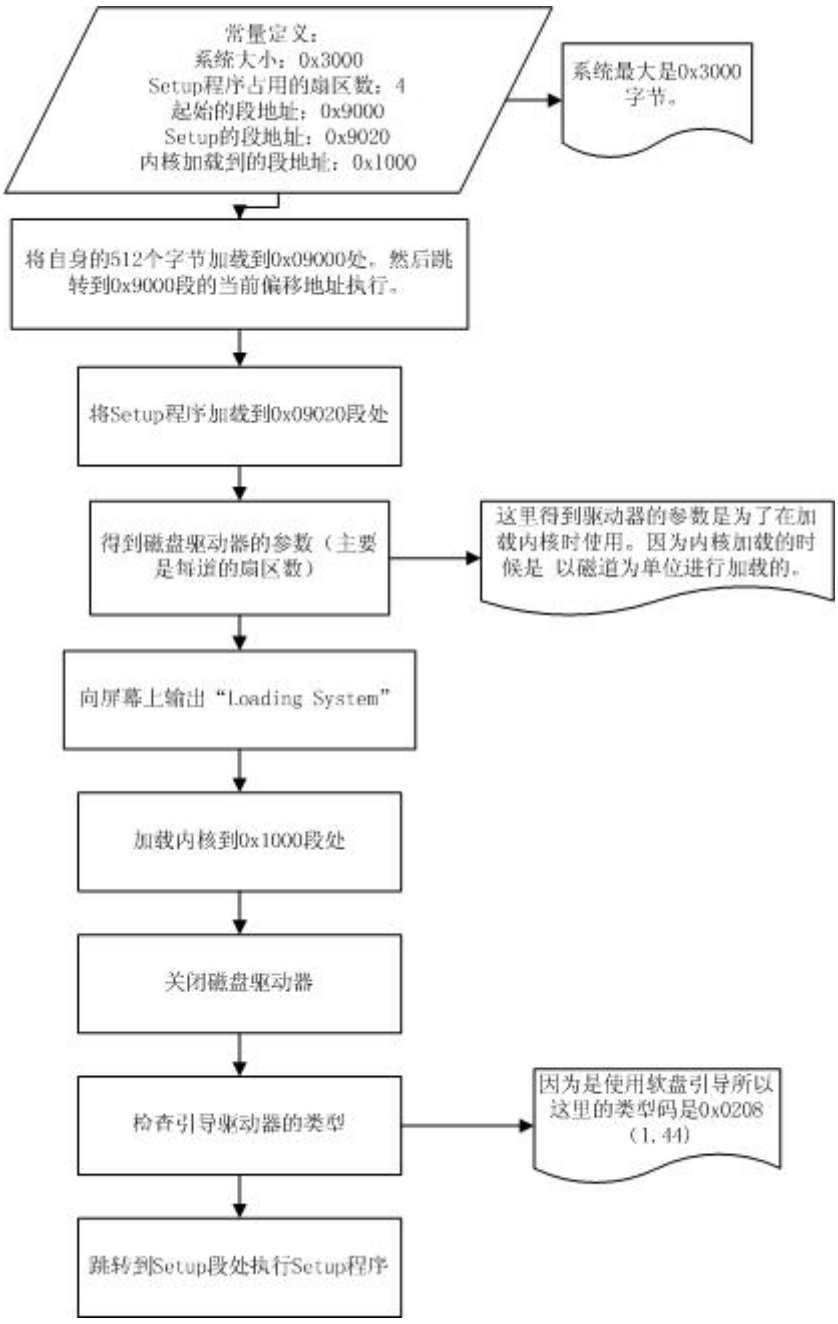


图 二.2bootsec.s 流程图

代码说明

```
!
! SYS_SIZE is the number of clicks (16 bytes) to be loaded.
! 0x3000 is 0x30000 bytes = 196kB, more than enough for current
! versions of linux
```

```
!
```
＝伪指令很像是 C 语言中的#define 语句，用于宏替换。
0x30000 对应的是 192k 字节，这个大小对于当前的内核来说已经足够了
SYSSIZE = 0x3000
```
!
!     bootsect.s       (C) 1991 Linus Torvalds
!
! bootsect.s is loaded at 0x7c00 by the bios-startup routines, and moves
! iself out of the way to address 0x90000, and jumps there.
!
! It then loads 'setup' directly after itself (0x90200), and the system
! at 0x10000, using BIOS interrupts.
!
! NOTE! currently system is at most 8*65536 bytes long. This should be no
! problem, even in the future. I want to keep it simple. This 512 kB
! kernel size should be enough, especially as this doesn't contain the
! buffer cache as in minix
!
! The loader has been made as simple as possible, and continuos
! read errors will result in a unbreakable loop. Reboot by hand. It
! loads pretty fast by getting whole sectors at a time whenever possible.

.globl begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text
```
// setup.s 编译后的文件在启动盘中占用的扇区数是 4
```
```
SETUPLEN = 4   ! nr of setup-sectors
```
```
// 启动盘的第一扇区加载到的段地址
```
```
BOOTSEG = 0x07c0   ! original address of boot-sector
```
```
// 引导程序自身就会移到 0x9000 段处
```
```
INITSEG = 0x9000   ! we move boot here - out of the way
```
```
// Setup 文件将要使用的段地址
```
```
SETUPSEG = 0x9020   ! setup starts here
```
```
// 内核文件使用的段地址
```
```
SYSSEG   = 0x1000   ! system loaded at 0x10000 (65536).
ENDSEG   = SYSSEG + SYSSIZE   ! where to stop loading

! ROOT_DEV: 0x000 - same type of floppy as boot.
!   0x301 - first partition on first drive etc
```
```

```
// 这里为根文件系统分配设备号
ROOT_DEV = 0x306

// 这里才是程序的入口，以上并不是变量的定义，编译器没有为他们分配空间，他们
// 会在由编译器自动的替换
entry start
start:
// 将引导程序自身从 0x7C00:0x0000 处移动到 0x9000:0x0000 处。这里使用了数据串操
// 作指令，可以在《汇编语言程序设计》这本书中的 7.4 节中查到。
        mov ax,#BOOTSEG
        mov ds,ax
        mov ax,#INITSEG
        mov es,ax
        mov cx,#256
        sub  si,si
        sub  di,di
        rep
        movw
// 段间跳转指令格式是:    jmpi  偏移地址，段地址。作用：段地址→CS, 偏移地址→IP。
// 这里是跳转到 0x9000：go 处，从新的内存段开始执行
        jmpi go,INITSEG
// 设置各个寄存器的值使其与代码段相同，因为这个时候所有的数据都存放在代码段
// 中
go:  mov ax,cs
        mov ds,ax
        mov es,ax
! put stack at 0x9ff00.
// 设置堆栈寄存器，这个地址比较随意，只要它的值远远大于 0x200＋0x200×4＋堆栈
// 大小就可以了
        mov ss,ax
        mov sp,#0xFF00          ! arbitrary value >>512

! load the setup-sectors directly after the bootblock.
! Note that 'es' is already set up.
// 现在是将 setup 文件加载到内存当中的时刻了，这里使用了 BIOS 的 13 号中断(int 13)
// int 13 的使用方法：
// 读扇区：
//    ah = 0x02 – ah 中放的是功能号，02 表示读磁盘扇区到内存；
//    al = 需要读出的扇区数量；
//    ch = 磁道(柱面)号的低 8 位；  cl = 开始扇区(0-5 位)，磁道号高 2 位(6-7)；
//    dh = 磁头号；  dl = 驱动器号（如果是硬盘则要置位 7）；
//     es:bx 指向数据缓冲区；  如果出错则 CF 标志为置位。
//    另外在数值前面加上#在 as86 编译器中表示这是立即数
load_setup:
```

```
        mov dx,#0x0000          ! drive 0, head 0
        mov cx,#0x0002          ! sector 2, track 0
        mov bx,#0x0200          ! address = 512, in INITSEG
        mov ax,#0x0200+SETUPLEN     ! service 2, nr of sectors
        int   0x13              ! read it
        jnc   ok_load_setup     ! ok - continue
// 磁盘的复位，使用 BIOS 的 13 号中断的第 0 号中断：
//    ah = 0     出口参数：无
        mov dx,#0x0000
        mov ax,#0x0000          ! reset the diskette
        int   0x13
        j     load_setup

ok_load_setup:

! Get disk drive parameters, specifically nr of sectors/track
// 取磁盘驱动器的参数，特别是每道的扇区数量。
//    取磁盘驱动器参数 INT 0x13  调用格式和返回信息如下：
//    ah = 0x08 dl =  驱动器号（如果是硬盘则要置位 7  为 1）。
//    返回信息：
//    如果出错则 CF  置位，并且 ah =  状态码。
//    ah = 0，   al = 0，   bl =  驱动器类型（AT/PS2）
//    ch =  最大磁道号的低 8  位，cl =  每磁道最大扇区数(位 0-5)，最大磁道号高 2  位(位
//          6-7)
//    dh =  最大磁头数，   dl =  驱动器数量，
//    es:di  指向软驱磁盘参数表。
        mov dl,#0x00
        mov ax,#0x0800          ! AH=8 is get drive parameters
        int   0x13
        mov ch,#0x00
// seg SEGMEN 指令表示下一条汇编语句使用操作数放在 SEGMEN 段中，这里是 CS
// 就是说 sectors 是放在 CS 段中的
        seg cs
        mov sectors,cx
        mov ax,#INITSEG
        mov es,ax

! Print some inane message
// 马上就要加载内核文件了，在加载这个文件的时候会在屏幕上输出“Loading
// system.…..”在屏幕上输出需要使用 BIOS 的 10 号中断。格式如下：
// 读光标位置：
//    ah  =  03    bh＝显示的页号（图形方式时是 0）
//    返回值：
//    DX,DL=行，列值        CH,CL=当前光标模式
```

```
//    显示字符串：
//        ES:BP =欲写入的字符串的首地址
//        CX＝字符串长度 DX＝光标的起始位置，（这个值刚才已经得到了）
//        BH=当前的显示页号  BL=属性
    mov ah,#0x03        ! read cursor pos
    xor  bh,bh
    int   0x10

    mov cx,#24
    mov bx,#0x0007          ! page 0, attribute 7 (normal)
    mov bp,#msg1
    mov ax,#0x1301          ! write string, move cursor
    int   0x10

! ok, we've written the message, now
! we want to load the system (at 0x10000)
// 设置内核文件将要存放到的段地址
    mov ax,#SYSSEG
    mov es,ax           ! segment of 0x010000
// 读取内核文件到内存中
    call  read_it
// 关闭驱动器马达，这样就可以知道驱动器的状态了
    call  kill_motor

! After that we check which root-device to use. If the device is
! defined (!= 0), nothing is done and the given device is used.
! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
! on the number of sectors that the BIOS reports currently.
// 此后，我们检查要使用哪个根文件系统设备（简称根设备）。如果已经指定了设备(!=0)
// 就直接使用给定的设备。否则就需要根据 BIOS 报告的每磁道扇区数来
// 确定到底使用/dev/PS0 (2,28) 还是 /dev/at0 (2,8)。
// 上面一行中两个设备文件的含义：
// 在 Linux 中软驱的主设备号是 2，次设备号 = type*4 + nr，其中
//  nr 为 0-3 分别对应软驱 A、B、C 或 D；type 是软驱的类型（2→1.2M 或 7→1.44M
// 等）。
//  因为 7*4 + 0 = 28，所以 /dev/PS0 (2,28)指的是 1.44M A 驱动器,其设备号是 0x021c
//  同理 /dev/at0 (2,8)指的是 1.2M A 驱动器，其设备号是 0x0208
    seg cs
    mov ax,root_dev
    cmp ax,#0
    jne  root_defined
    seg cs
    mov bx,sectors
    mov ax,#0x0208          ! /dev/ps0 - 1.2Mb
```

```
        cmp bx,#15
        je  root_defined
        mov ax,#0x021c          ! /dev/PS0 - 1.44Mb
        cmp bx,#18
        je  root_defined
undef_root:
        jmp undef_root
root_defined:
        seg cs
// 保存设备号
        mov root_dev,ax

! after that (everyting loaded), we jump to
! the setup-routine loaded directly after
! the bootblock:
// 现在加载结束了可以执行 setup 程序了
        jmpi 0,SETUPSEG

! This routine loads the system at address 0x10000, making sure
! no 64kB boundaries are crossed. We try to load it as fast as
! possible, loading whole tracks whenever we can.
!
! in: es - starting address segment (normally 0x1000)
!
// 这段代码是加载内核文件用的，每次都是加载一个磁道。这段代码实在是复杂我真
// 的不想些能使你理解它的注释了，而且它的实现对理解内核没有什么影响，所以就
// 允许我偷懒一次吧☺,如果你特有好奇心，又很懒，就请直接联系我吧。
sread:  .word 1+SETUPLEN    ! sectors read of current track
head:   .word 0             ! current head
track:  .word 0             ! current track

read_it:
        mov ax,es
        test ax,#0x0fff
die:  jne die               ! es must be at 64kB boundary
        xor bx,bx       ! bx is starting address within segment
rp_read:
        mov ax,es
        cmp ax,#ENDSEG      ! have we loaded all yet?
        jb ok1_read
        ret
ok1_read:
        seg cs
        mov ax,sectors
```

```
        sub ax,sread
        mov cx,ax
        shl cx,#9
        add cx,bx
        jnc ok2_read
        je ok2_read
        xor ax,ax
        sub ax,bx
        shr ax,#9
ok2_read:
        call read_track
        mov cx,ax
        add ax,sread
        seg cs
        cmp ax,sectors
        jne ok3_read
        mov ax,#1
        sub ax,head
        jne ok4_read
        inc track
ok4_read:
        mov head,ax
        xor ax,ax
ok3_read:
        mov sread,ax
        shl cx,#9
        add bx,cx
        jnc rp_read
        mov ax,es
        add ax,#0x1000
        mov es,ax
        xor bx,bx
        jmp rp_read

read_track:
        push ax
        push bx
        push cx
        push dx
        mov dx,track
        mov cx,sread
        inc cx
        mov ch,dl
        mov dx,head
```

```
        mov dh,dl
        mov dl,#0
        and dx,#0x0100
        mov ah,#2
        int 0x13
        jc bad_rt
        pop dx
        pop cx
        pop bx
        pop ax
        ret
bad_rt:    mov ax,#0
        mov dx,#0
        int 0x13
        pop dx
        pop cx
        pop bx
        pop ax
        jmp read_track

/*
  * This procedure turns off the floppy drive motor, so
  * that we enter the kernel in a known state, and
  * don't have to worry about it later.
  */
kill_motor:
        push dx
        mov dx,#0x3f2
        mov al,#0
        outb
        pop dx
        ret

sectors:
        .word 0

msg1:
        .byte 13,10
        .ascii "Loading system ..."
        .byte 13,10,13,10

.org 508
root_dev:
        .word ROOT_DEV
```

```
// 这个标志很重要，因为BIOS 只有发现了 0xAA55 这个标记之后才能认定这个引导程
// 序是正确的。这个标志要求放在 511 和 512 字节处
boot_flag:
    .word 0xAA55

.text
endtext:
.data
enddata:
.bss
endbss:
```

后记

这个引导程序非常的经典，很多引导程序同它思想很是相像。如果对引导程序的设计感兴趣的话，可以从网上找到一篇叫《操作系统引导探究》的文章，写的不错。

我也试验着写了一个引导程序，这个程序是在 MASM 6.0 下编写的。编译出来的是.exe 格式的文件，这个文件当然不能作为引导程序了。所以需要使用 exe2bin.exe 这个程序将.exe 文件转换为.com 格式的文件（在找到这个宝贝之前我一直是直接修改.exe 文件的，十分痛苦）。

引导程序必须写在第一扇区中，但是在 Windows2000 下是无法简单的完成这项工程的（至少我还不知道简单的方法）。所以，需要在 Windows98 下使用 Turbo C 来完成。调用的是直接扇区读写函数，这个函数已经不在 VC6.0 中了。

有一个很好的软件 WinHex，它可以对磁盘进行物理查看，用它来看你制作的第一扇区可以加深你对引导程序的理解。

Setup.s 的说明

setup 程序的作用主要是利用ROM BIOS 中断读取机器系统数据，并将这些数据保存到 0x90000 开始的位置（覆盖掉了bootsect 程序所在的地方），所取得的参数和保留的内存位置见下图：

内存地址	长度(字节)	名称	描述
0x90000	2	光标位置	列号（0x00-最左端），行号（0x00-最顶端）
0x90002	2	扩展内存数	系统从 1M 开始的扩展内存数值（KB）。
0x90004	2	显示页面	当前显示页面
0x90006	1	显示模式	
0x90007	1	字符列数	
0x90008	2	??	
0x9000A	1	显示内存	显示内存(0x00-64k, 0x01-128k, 0x02-192k, 0x03=256k)
0x9000B	1	显示状态	0x00-彩色, I/O=0x3dX; 0x11-单色, I/O=0x3bX
0x9000C	2	特性参数	显示卡特性参数
...			
0x90080	16	硬盘参数表	第 1 个硬盘的参数表
0x90090	16	硬盘参数表	第 2 个硬盘的参数表（如果没有，则清零）
0x901FC	2	根设备号	根文件系统所在的设备号（bootsec.s 中设置）

图 二.3 参数在内存中的存放位置（出自于《Linux0.11 源码完全注释》）

这个参数表对于以后的程序是很有用的，许多函数都是通过这个表来得到系统的信息的。

然后setup 程序将system 模块从0x10000-0x8ffff（当时认为内核系统模块system 的长度不会超过此值：512KB）整块向下移动到内存绝对地址0x00000 处。接下来加载中断描述符表寄存器(idtr)和全局描述符表寄存器(gdtr)，开启A20 地址线，重新设置两个中断控制芯片8259A，将硬件中断号重新设置为0x20 - 0x2f。最后设置CPU 的控制寄存器CR0，从而进入32 位保护模式运行，并跳转到位于内核文件最前面部分的head.s 程序继续运行。

代码的简要流程图

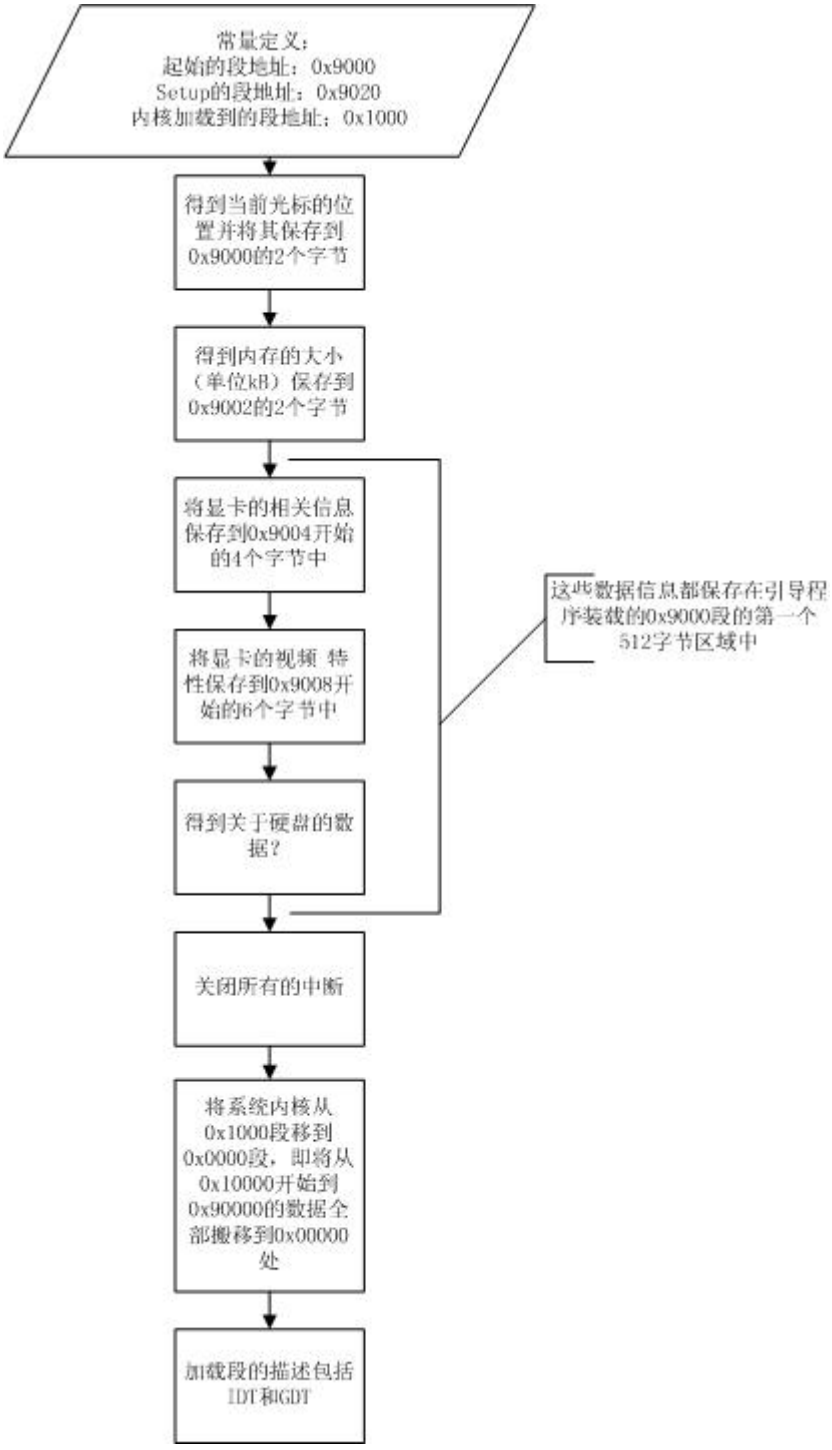


图 二.4setup.s 流程图

代码说明

!

```
!	setup.s		(C) 1991 Linus Torvalds
!
! setup.s is responsible for getting the system data from the BIOS,
! and putting them into the appropriate places in system memory.
! both setup.s and system has been loaded by the bootblock.
!
! This code asks the bios for memory/disk/other parameters, and
! puts them in a "safe" place: 0x90000-0x901FF, ie where the
! boot-block used to be. It is then up to the protected mode
! system to read them from there before the area is overwritten
! for buffer-blocks.
!

! NOTE! These had better be the same as in bootsect.s!
// 见 bootsec.s 中说明
INITSEG  = 0x9000	! we move boot here - out of the way
SYSSEG   = 0x1000	! system loaded at 0x10000 (65536).
SETUPSEG = 0x9020	! this is the current segment

.globl begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

entry start
start:

! ok, the read went well so we get current cursor position and save it for
! posterity.
// 现在将光标位置保存以备今后使用
	mov ax,#INITSEG ! this is done in bootsect already, but...
	mov ds,ax
	mov ah,#0x03 ! read cursor pos
	xor bh,bh
	int 0x10		! save it in known place, con_init fetches
	mov [0],dx		! it from 0x90000.

! Get memory size (extended mem, kB)
// 取扩展内存的大小值（KB）。
//  是调用中断 0x15，功能号 ah = 0x88
```

```
//  返回：ax = 从 0x100000（1M）处开始的扩展内存大小(KB)。
//  若出错则 CF 置位，ax = 出错码。
// 这个值将会在内存的初始化中使用
        mov ah,#0x88
        int  0x15
        mov [2],ax

! Get video-card data:
// 取显示卡当前显示模式。
//  调用 BIOS 中断 0x10，功能号 ah = 0x0f
//  返回：ah = 字符列数，al = 显示模式，bh = 当前显示页。
//  0x90004(1 字)存放当前页，0x90006 显示模式，0x90007 字符列数
        mov ah,#0x0f
        int  0x10
        mov [4],bx          ! bh = display page
        mov [6],ax          ! al = video mode, ah = window width

! check for EGA/VGA and some config parameters
// 检查显示方式（EGA/VGA）并取参数。
//  调用 BIOS 中断 0x10，
//  功能号：ah = 0x12，bl = 0x10
//  返回：bh = 显示状态
//  (0x00 - 彩色模式，I/O 端口=0x3dX)
//  (0x01 - 单色模式，I/O 端口=0x3bX)
//  bl = 安装的显示内存
// (0x00 - 64k, 0x01 - 128k, 0x02 - 192k, 0x03 = 256k)
// cx = 显示卡特性参数
// 这个值会在终端的初始化中使用
        mov ah,#0x12
        mov bl,#0x10
        int  0x10
        mov [8],ax
        mov [10],bx
        mov [12],cx

! Get hd0 data
// 取第一个硬盘的信息（复制硬盘参数表）。
//  第 1 个硬盘参数表的首地址竟然是中断向量 0x41 的向量值！而第 2 个硬盘
//  参数表紧接第 1 个表的后面，中断向量 0x46 的向量值也指向这第 2 个硬盘
//  的参数表首址。表的长度是 16 个字节(0x10)。 下面两段程序分别复制 BIOS 有关
//  两个硬盘的参数表，0x90080 处存放第 1 个 硬盘的表，0x90090 处存放第 2 个硬
//  盘的表。
        mov ax,#0x0000
        mov ds,ax
```

```
        lds   si,[4*0x41]
        mov ax,#INITSEG
        mov es,ax
        mov di,#0x0080
        mov cx,#0x10
        rep
        movsb

! Get hd1 data

        mov ax,#0x0000
        mov ds,ax
        lds   si,[4*0x46]
        mov ax,#INITSEG
        mov es,ax
        mov di,#0x0090
        mov cx,#0x10
        rep
        movsb

! Check that there IS a hd1 :-)
 // 检查系统是否存在第 2 个硬盘，如果不存在则第 2 个表清零。
 // 利用 BIOS 中断调用 0x13 的取盘类型功能。
 // 功能号 ah = 0x15；
 // 输入：dl = 驱动器号（0x8X 是硬盘：0x80 指第 1 个硬盘，0x81 第 2 个硬盘）
 // 输出：ah = 类型码;00 --没有这个盘，CF 置位；01 --是软驱，没有 change-line 支持；
 //   02 --是软驱(或其它可移动设备)，有 change-line 支持； 03 --是硬盘。
        mov ax,#0x01500
        mov dl,#0x81
        int   0x13
        jc    no_disk1
        cmp ah,#3
        je    is_disk1
no_disk1:
        mov ax,#INITSEG
        mov es,ax
        mov di,#0x0090
        mov cx,#0x10
        mov ax,#0x00
        rep
        stosb
is_disk1:

! now we want to move to protected mode ...
```

```
        cli                     ! no interrupts allowed !

    ! first we move the system to it's rightful place
// 把整个 system 模块移动到 0x00000 位置，即把从 0x10000 到 0x8ffff 的内存数据块
//  (512k)，整块地向内存低端移动了 0x10000（64k）的位置。这样就可以方便以后的内存
// 管理了，为系统内核留下一块固定的区域。剩下的就可以随便使用了。这里是一共移动了 8
// 次，一次移动 0x10000 字节。为什么不一次就把 0x80000 字节的内容移动完呢？因为
//  0x80000 占用了 20 位，而 cx 寄存器是 16 位的因此只能分 8 次移动了。
        mov ax,#0x0000
        cld                     ! 'direction'=0, movs moves forward
    do_move:
        mov es,ax               ! destination segment
        add  ax,#0x1000
        cmp ax,#0x9000
        jz    end_move
        mov ds,ax               ! source segment
        sub  di,di
        sub  si,si
        mov         cx,#0x8000
        rep
        movsw
        jmp  do_move

    ! then we load the segment descriptors

    end_move:
        mov ax,#SETUPSEG    ! right, forgot this at first. didn't work :-)
        mov ds,ax
    // 这里就是进行保护模式的设置了，后面有关于它的详细描述。
    //  lidt 指令用于加载中断描述符表(idt)寄存器，它的操作数是 6 个字节
        lidt  idt_48            ! load idt with 0,0
    //  lgdt 指令用于加载全局描述符表(gdt)寄存器，其操作数格式与 lidt 指令的相同。全
    // 局描述符 表中的每个描述符项(8 字节)描述了保护模式下数据和代码段（块）的信
    // 息。其中包括段的 最大长度限制(16 位)、段的线性基址（32 位）、段的特权级、
    // 段是否在内存、读写许可以及 其它一些保护模式运行的标志。
        lgdt  gdt_48            ! load gdt with whatever appropriate

    ! that was painless, now we enable A20
    // 开启 A20 地址线
        call  empty_8042
        mov al,#0xD1        ! command write
        out  #0x64,al
        call  empty_8042
```

```
        mov al,#0xDF        ! A20 on
        out  #0x60,al
        call  empty_8042

! well, that went ok, I hope. Now we have to reprogram the interrupts :-(
! we put them right after the intel-reserved hardware interrupts, at
! int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
! messed this up with the original PC, and they haven't been able to
! rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
! which is used for the internal hardware interrupts as well. We just
! have to reprogram the 8259's, and it isn't fun.
// 对 8259A 控制器进行初始化，参看《接口技术》的 P122
        mov al,#0x11        ! initialization sequence
// 初始化主控电路的 ICW1，设置为电平触发，级联，使用 ICW4
        out  #0x20,al       ! send it to 8259A-1
        .word       0x00eb,0x00eb           ! jmp $+2, jmp $+2
// 初始化从控电路的 ICW1，设置为电平触发，级联，使用 ICW4
        out  #0xA0,al       ! and to 8259A-2
        .word       0x00eb,0x00eb
        mov al,#0x20        ! start of hardware int's (0x20)
// 初始化主控电路的 ICW2
// 设置主控电路的中断号从 0x20 开始，这就是时钟中断对应的 int 0x20 的原因
        out  #0x21,al
        .word       0x00eb,0x00eb
        mov al,#0x28        ! start of hardware int's 2 (0x28)
// 初始化从控电路的 ICW2
// 设置从控电路的中断号从 0x28 开始
        out  #0xA1,al
        .word       0x00eb,0x00eb
// 初始化主控电路的 ICW3
// 表示主控电路的 IR2 口连接从控电路
        mov al,#0x04        ! 8259-1 is master
        out  #0x21,al
        .word       0x00eb,0x00eb
// 初始化从控电路的 ICW3
// 表示从控电路连接到主控电路的 IR2 处
        mov al,#0x02        ! 8259-2 is slave
        out  #0xA1,al
        .word       0x00eb,0x00eb
// 初始化主控电路的 ICW4
// 设置为非特殊完全嵌套，非缓冲，非自动 EOI，86/88 方式
        mov al,#0x01        ! 8086 mode for both
        out  #0x21,al
        .word       0x00eb,0x00eb
```

```
// 初始化从控电路的 ICW4
        out  #0xA1,al
        .word       0x00eb,0x00eb
        mov al,#0xFF        ! mask off all interrupts for now
// 对主从控制器发出 OCW1 命令，屏蔽一些中断
        out  #0x21,al
        .word       0x00eb,0x00eb
        out  #0xA1,al

! well, that certainly wasn't fun :-(. Hopefully it works, and we don't
! need no steenking BIOS anyway (except for the initial loading :-).
! The BIOS-routine wants lots of unnecessary data, and it's less
! "interesting" anyway. This is how REAL programmers do it.
!
! Well, now's the time to actually move into protected mode. To make
! things as simple as possible, we do no register set-up or anything,
! we let the gnu-compiled 32-bit programs do that. We just jump to
! absolute address 0x00000, in 32-bit protected mode.
// 这里设置进入 32 位保护模式运行。首先加载机器状态字(lmsw - Load Machine Status
//  Word)，到寄存器 CR0，其比特位 0 置 1 将导致 CPU 工作在保护模式。
        mov ax,#0x0001      ! protected mode (PE) bit
        lmsw        ax          ! This is it!
        jmpi 0,8            ! jmp offset 0 of segment 8 (cs)

! This routine checks that the keyboard command queue is empty
! No timeout is used - if this hangs there is something wrong with
! the machine, and we probably couldn't proceed anyway.
empty_8042:
        .word       0x00eb,0x00eb
        in      al,#0x64  ! 8042 status port
        test    al,#2             ! is input buffer full?
        jnz     empty_8042    ! yes - loop
        ret
// 全局描述符表开始处。描述符表由多个 8 字节长的描述符项组成。这里给出了 3 个
// 描述符项。第 1 项无用，但须存在。第 2 项是系统代码段描述符，第 3 项是系统数
// 据段描述符。后边有关于全局描述符表（GDT）的说明。
// .word 伪指令表示其后每个数据项占用 2 个字节。
gdt:
        .word       0,0,0,0             ! dummy

        .word       0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
        .word       0x0000          ! base address=0
        .word       0x9A00          ! code read/exec
        .word       0x00C0          ! granularity=4096, 386
```

```
        .word   0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
        .word   0x0000          ! base address=0
        .word   0x9200          ! data read/write
        .word   0x00C0          ! granularity=4096, 386

idt_48:
        .word   0               ! idt limit=0
        .word   0,0             ! idt base=0L

gdt_48:
        .word   0x800           ! gdt limit=2048, 256 GDT entries
        .word   512+gdt,0x9     ! gdt base = 0X9xxxx

.text
endtext:
.data
enddata:
.bss
endbss:
```

保护模式（段式管理部分）

Linux 是运行在保护模式下的，所以在开始执行 Linux 的内核之前需要将内存的模式由实模式改为保护模式。为了便于理解代码的执行，这里先对保护模式进行一次简单的说明。如果想了解更多的信息可以查看《intel 系列微处理器结构、编程和接口技术大全》和《保护方式下的 80386 及其编程》。

在保护模式中段寄存器的意义已经变了，它的内容是一个对基地址的一个索引并且加上了一些类型的说明和权限。因此一个指令寻找地址的过程可以概括如下：

1） **根据指令的类型来决定使用什么样的寄存器**，例如 jmp 0900h 指令就是使用 CS 段寄存器，而 mov ax，2 是使用 DS 段寄存器；

2） 根据段寄存器中的内容找到“地址段描述结构”。

Intel 是这样做到的：

在计算机中新设置两个寄存器 GDTR 和 LDTR 来存储一个结构表的基地址。GDTR 中数据的格式如下：

	BIT47—BIT16	BIT15—BIT0
全局描述符表寄存器 GDTR	基地址	界限
中断描述符表寄存器 IDTR	基地址	界限

基地址：全局（中断）描述符表的首地址，知道了这个地址就可以检索表的内容了；

界限：全局（中断）描述符表的长度限制。

Linux0.11 代码中的定义如下：

```
gdt_48:
    .word    0x800      ；gdt 界限=2048 字节，因为一个描述符需要 8 个字节
                        所以这里，最多允许 256 个 GDT 描述符。
    .word    512+gdt,0x9 ； gdt 的基地址 = 0x00090000+512+gdt
```

使用 LGDT gdt_48 更新了 GDTR。这个时候通过 GDTR 就可以找到地址段描述结构。至于使用该结构中的哪一个描述符是由段寄存器中的标号来决定的。新的段寄存器的结构如下：

选择子结构	BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
	描述符索引													TI	RPL	

由于是使用 13 个二进制位来保存描述符索引所以理论上最多允许索引 8192 个 GDT 描述符。但是在这里，从代码中可以看到最多允许 256 个。段选择子的第 2 位是引用描述符表指示位，标记为 TI，TI=0 指示从全局描述符表 GDT 中读取描述符；TI=1 指示从局部描述符表 LDT 中读取描述符。选择子的最低两位是请求特权级 RPL，用于特权检查。 RPL 字段的用法是每当程序试图访问一个段时，要把当前特权级与所访问段的特权级进行比较，以确定是否允许程序对该段的访问。

3） 从“地址段描述结构”中得到基地址。

地址段描述符占用 8 个字节其格式如下：

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

说明：

(1) P 位称为存在位。P=1 表示描述符对地址转换是有效的，或者说该描述符所描述的段存在，即在内存中；P=0 表示描述符对地址转换无效，即该段不存在。使用该描述符进行内存访问时会引起异常。

(2) DPL 表示描述符特权级共 2 位。它规定了所描述段的特权级，用于特权检查，以决定对该段能否访问。

(3) DT 位说明描述符的类型。对于存储段描述符而言，DT=1，以区别与系统段描述符和门描述符(DT=0)。

(4) TYPE 说明存储段描述符所描述的存储段的具体属性。

其中的位 0 指示描述符是否被访问过，用符号 A(Accessed)标记。A=0 表示尚未被访问，A=1 表示段已被访问。当把描述符的相应选择子装入到段寄存器时，80386 把该位置为 1，表明描述符已被访问。操作系统可测试访问位，已确定描述符是否被访问过。

其中的位 3 指示所描述的段是代码段还是数据段，用符号 E 标记。E=0 表示段为数据段，相应的描述符也就是数据段(包括堆栈段)描述符。数据段是不可执行的，但总是可读的。E=1 表示段是可执行段，即代码段，相应的描述符就是代码段描述符。代码段总是不可写的，若需要对代码段进行写入操作，则必须使用别名技术，即用一个可写的数据段描述符来描述该代码段，然后对此数据段进行

写。

在数据段描述符中(E=0 的情况),TYPE 中的位 1 指示所描述的数据段是否可写,用 W 标记。W=0 表示对应的数据段不可写。反之,W=1 表示数据段是可写的。注意,数据段总是可读的。TYPE 中的位 2 是 ED 位,指示所描述的数据段的扩展方向。ED=0 表示数据段向高端扩展,也即段内偏移必须小于等于段界限。ED=1 表示数据段向低扩展,段内偏移必须大于段界限。

在代码段描述符中(E=1 的情况),TYPE 中的位 1 指示所描述的代码段是否可读,用符号 R 标记。R=0 表示对应的代码段不可读,只能执行。R=1 表示对应的代码段可读可执行。注意代码段总是不可写的,若需要对代码段进行写入操作,则必须使用别名技术。在代码段中,TYPE 中的位 2 指示所描述的代码段是否是一致代码段,用 C 标记。C=0 表示对应的代码段不是一致代码段(普通代码段),C=1 表示对应的代码段是一致代码段。

存储段描述符中的 TYPE 字段所说明的属性可归纳为下表:

	类型值	说　　明
数据段类　型	0	只读
	1	只读、已访问
	2	读/写
	3	读/写、已访问
	4	只读、向下扩展
	5	只读、向下扩展、已访问
	6	读/写、向下扩展
	7	读/写、向下扩展、已访问

	类型值	说　　明
代码段类　型	8	只执行
	9	只执行、已访问
	A	执行/读
	B	执行/读、已访问
	C	只执行、一致码段
	D	只执行、一致码段、已访问
	E	执行/读、一致码段
	F	执行/读、一致码段、已访问

(5) G 为就是段界限粒度(Granularity)位。G=0 表示界限粒度为字节;G=1 表示界限粒度为 4K 字节。注意,界限粒度只对段界限有效,对段基地址无效,段基地址总是以字节为单位。

(6) D 位是一个很特殊的位,在描述可执行段、向下扩展数据段或由 SS 寄存器寻址的段(通常是堆栈段)的三种描述符中的意义各不相同。在描述可执行段的描述符中,D 位决定了指令使用的地址及操作数所默认的大小。D=1 表示默认情况下指令使用 32 位地址及 32 位或 8 位操作数,这样的代码段也称为 32 位代码段;D=0 表示默认情况下,使用 16 位地址及 16 位或 8 位操作数,这样的代码段也称为 16 位代码段,它与 80286 兼容。可以使用地址大小前缀和操作数大小前缀分别改变默认的地址或操作数的大小。

在向下扩展数据段的描述符中,D 位决定段的上部边界。D=1 表示段的上部界限为 4G;D=0 表示段的上部界限为 64K,这是为了与 80286 兼容。

在描述由 SS 寄存器寻址的段描述符中,D 位决定隐式的堆栈访问指令(如 PUSH 和 POP 指令)使用何种堆栈指针寄存器。D=1 表示使用 32 位堆栈指针寄存器 ESP;D=0 表示使用 16 位堆栈指针寄存器 SP,这与 80286 兼容。

(7) AVL 位是软件可利用位。80386 对该位的使用未左规定,Intel 公司也保证今后开发生产的处理器只要与 80386 兼容,就不会对该位的使用做任何定义或规定。

此外,描述符内第 6 字节中的位 5 必须置为 0,可以理解成是为以后的处理器保留的。

在 Linux0.11 代码中的 GDT 定义如下:

gdt:

```
.word    0,0,0,0        ; GDT 表中的第 0 号描述符必须是 0，并且不会被使用

.word    0x07FF         ! 8Mb - limit=2047 (2048*4096=8Mb)
.word    0x0000         ! base address=0
.word    0x9A00         ! code read/exec
.word    0x00C0         ! granularity=4096, 386
```

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)(00)			属性(C0 9A)		Base(23...0)(00 00 00)					Limite(15...0)(07 FF)			
存储段描述符属性	Byte m+6								Byte m+5					
	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
	G(1)	D(1)	0	AVL(0)	Limit(19...16)(0)				P(1)	DPL(00)		DT(1)	TYPE(A)			

```
;这个描述符说明基地址是 0，段限制的粒度为 4k 字节，段大小限制 8M（理论上最
大可达到 4G），只有级别为 00 的程序可以访问，对该段对应的内存可以执行/读。
.word    0x07FF         ! 8Mb - limit=2047 (2048*4096=8Mb)
.word    0x0000         ! base address=0
.word    0x9200         ! data read/write
.word    0x00C0         ! granularity=4096, 386
;这个描述符说明基地址是 0，段限制的粒度为 4k 字节，段大小限制 8M，只有级别
为 00 的程序可以访问，对该段对应的内存可以读/写。
```

4) **将指令发出的偏移地址同规定的段长度相比检查是否发生越界**，这样就可以避免非法访问内存。

5) **将指令发出的偏移地址同基地址相加形成物理地址。**

注意：这里介绍的仅仅是段式的存储。

A20 地址线问题[1]

1981 年8 月，IBM 公司最初推出的个人计算机IBM PC 使用的CPU 是Intel 8088。在该微机中地址线只有20 根(A0 - A19)。在当时内存RAM 只有几百KB 或不到1MB 时，20 根地址线已足够用来寻址这些内存。其所能寻址的最高地址是0xffff:0xffff，也即0x10ffef。对于超出0x100000(1MB)的寻址地址将默认地环绕到0x0ffef。当IBM 公司于1985 年引入AT 机时，使用的是Intel 80286 CPU，具有24根地址线，最高可寻址16MB，并且有一个与8088 完全兼容的实模式运行方式。然而，在寻址值超过1MB时它却不能象8088 那样实现地址寻址的环绕。但是当时已经有一些程序是利用这种地址环绕机制进行工作的。为了实现完全的兼容性，IBM 公司发明了使用一个开关来开启或禁止0x100000 地址比特位。由于在当时的8042 键盘控制器上恰好有空闲的端口引脚（输出端口P2，引脚P21），于是便使用了该引脚来作为与门控制这个地址比特位。该信号即被称为A20。如果它为零，则比特20 及以上地址都被清除。从而实现了兼容性。由于在机器启动时，默认条件下，A20 地址线是禁止的，所以操作系统必须使用适当的方法来开启它。但是由于各种兼容机所使用的芯片集不同，要做到这一点却是非常的麻烦。因此通常要在几种控制方法中选择。

对A20 信号线进行控制的常用方法是通过设置键盘控制器的端口值。这里的setup.s 程序，即使用了这种典型的控制方式。对于其它一些兼容微机还可以使用其它方式来做到对A20 线

[1] 这段说明是完全从赵炯博士的《Linux0.11版本完全分析》中摘录下来的

的控制。有些操作系统将A20 的开启和禁止作为实模式与保护运行模式之间进行转换的标准过程中的一部分。由于键盘的控制器速度很慢，因此就不能使用键盘控制器对A20 线来进行操作。为此引进了一个A20快速门选项(Fast Gate A20)，它使用I/O 端口0x92 来处理A20 信号线，避免了使用慢速的键盘控制器操作方式。对于不含键盘控制器的系统就只能使用0x92端口来控制，但是该端口也有可能被其它兼容微机上的设备（如显示芯片）所使用，从而造成系统错误的操作。还有一种方式是通过读0xee 端口来开启A20 信号线，写该端口则会禁止A20 信号线。

Head.s 的说明

head.s 程序在被编译后，会被连接成内核文件的最前面开始部分。从这里开始，内核完全都是在保护模式下运行了。heads.s 汇编程序与前面的语法格式不同，它采用的是AT&T的汇编语言格式，并且需要使用GNU 的gas 和gld进行编译连接。因此请注意代码中赋值的方向是从左到右。

这段程序已经由setup程序搬到0x0000处，所以实际上是从内存绝对地址0 处开始执行的。它的功能比较单一，首先是加载各个数据段寄存器，重新设置中断描述符表，共256 项，并使各个表项均指向一个只报错误的中断服务子程序。然后重新设置全局描述符表。接着使用物理地址0 与1M 开始处的内容相比较的方法，检测A20 地址线是否已真的开启（如果没有开启，则在访问高于1Mb 物理内存地址时CPU 实际只会访问（IP MOD 1Mb）地址处的内容），如果检测下来发现没有开启，则进入死循环。然后程序测试PC 机是否含有数学协处理器芯片（80287、80387 或其兼容芯片），并在控制寄存器CR0 中设置相应的标志位。接着设置管理内存的分页处理机制，将页目录表放在绝对物理地址0 开始处（也是本程序所处的物理内存位置，因此这段程序将被覆盖掉），紧随后面放置共可寻址16MB 内存的4 个页表，并分别设置它们的表项。最后利用返回指令将预先放置在堆栈中的/init/main.c 程序的入口地址弹出，去运行main()程序。

代码的简要流程图

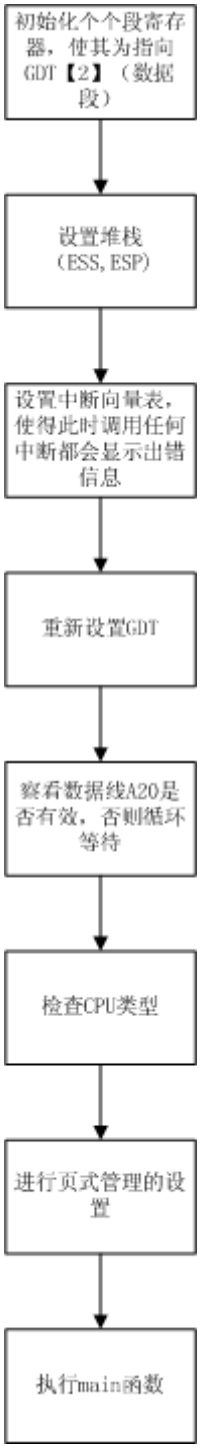


图 二.5head.s 的简要流程图

代码说明

.text

```
.globl _idt,_gdt,_pg_dir,_tmp_floppy_area
_pg_dir:
// 以下代码设置用于将所有的段寄存器(ds,es,fs,gs)的标记设置为指向 GDT 的地址
// 段，同时初始化了堆栈段。
startup_32:
    movl $0x10,%eax  // 相当于 MOV    EAX, 10H, 这里的 MOVL 表示进行 32 位移动
    mov %ax,%ds      // 相当于 MOV    DS, AX
    mov %ax,%es
    mov %ax,%fs
    mov %ax,%gs
// 表示_stack_start→ss:esp。_stack_start 是 C 编译程序为程序自动生成的存有堆栈信
// 息地方
    lss _stack_start,%esp
// 设置中断向量表，让所有的中断都指向一个子程序，这个子程序的作用是显示一
// 行“Unknown interrupt”表示中断尚未设置。关于这个函数的具体的实现方法参考
//  setup_idt 的说明。
    call setup_idt
// 重新设置 GDT，关于 GDT 的设置方法见 Setup 的说明。
    call setup_gdt
// 重新设置各个段寄存器。
    movl $0x10,%eax         # reload all the segment registers
    mov %ax,%ds       # after changing gdt. CS was already
    mov %ax,%es       # reloaded in 'setup_gdt'
    mov %ax,%fs
    mov %ax,%gs
    lss _stack_start,%esp
// 察看数据线 A20 是否有效，否则循环等待。地址线 A20 是 x86 的历史遗留问题，
// 决定是否能访问 1M 以上内存。
    xorl %eax,%eax
1:  incl %eax           # check that A20 really IS enabled
    movl %eax,0x000000    # loop forever if it isn't
    cmpl %eax,0x100000
    je 1b
/*
 * NOTE! 486 should set bit 16, to check for write-protect in supervisor
 * mode. Then it would be unnecessary with the "verify_area()"-calls.
 * 486 users probably want to set the NE (#5) bit also, so as to use
 * int 16 for math errors.
 */
// 检查 CPU 的类型，为什么，我不知道。☹
    movl %cr0,%eax          # check math chip
    andl $0x80000011,%eax       # Save PG,PE,ET
/* "orl $0x10020,%eax" here for 486 might be good */
    orl $2,%eax         # set MP
```

```
    movl %eax,%cr0
    call check_x87
    jmp after_page_tables

/*
 * We depend on ET to be correct. This checks for 287/387.
 */
check_x87:
    fninit
    fstsw %ax
    cmpb $0,%al
    je 1f                /* no coprocessor: have to set bits */
    movl %cr0,%eax
    xorl $6,%eax         /* reset MP, set EM */
    movl %eax,%cr0
    ret
.align 2
1:  .byte 0xDB,0xE4          /* fsetpm for 287, ignored by 387 */
    ret

/*
 *  setup_idt
 *
 *  sets up a idt with 256 entries pointing to
 *  ignore_int, interrupt gates. It then loads
 *  idt. Everything that wants to install itself
 *  in the idt-table may do so themselves. Interrupts
 *  are enabled elsewhere, when we can be relatively
 *  sure everything is ok. This routine will be over-
 *  written by the page tables.
 */
// 设置 IDT：首先在_idt 的位置清空了 256×8（中断向量是 8 个字节）字节的空间，
// 这个工作在编译的时候已经完成了。然后，循环 256 次将 ignore_int 的入口地址放
// 入其中。之后，设置 IDTR 寄存器，使其指向刚刚设置的 IDT 表。IDTR 寄存器的
// 格式与 GDTR 寄存器几乎一样，这里不再累述。
setup_idt:
    lea ignore_int,%edx
    movl $0x00080000,%eax
    movw %dx,%ax             /* selector = 0x0008 = cs */
    movw $0x8E00,%dx     /* interrupt gate - dpl=0, present */

    lea _idt,%edi
    mov $256,%ecx
rp_sidt:
```

```
    movl %eax,(%edi)
    movl %edx,4(%edi)
    addl $8,%edi
    dec %ecx
    jne rp_sidt
    lidt idt_descr
    ret

/*
 *  setup_gdt
 *
 *  This routines sets up a new gdt and loads it.
 *  Only two entries are currently built, the same
 *  ones that were built in init.s. The routine
 *  is VERY complicated at two whole lines, so this
 *  rather long comment is certainly needed :-).
 *  This routine will beoverwritten by the page tables.
 */
setup_gdt:
    lgdt gdt_descr
    ret

/*
 * I put the kernel page tables right after the page directory,
 * using 4 of them to span 16 Mb of physical memory. People with
 * more than 16MB will have to expand this.
 */
// 在页目录表中存放的就是这些页表项表的地址，每个表地址之间的距离是 4k。
.org 0x1000
pg0:

.org 0x2000
pg1:

.org 0x3000
pg2:

.org 0x4000
pg3:

.org 0x5000
/*
 * tmp_floppy_area is used by the floppy-driver when DMA cannot
 * reach to a buffer-block. It needs to be aligned, so that it isn't
```

```
    * on a 64kB border.
    */
// 鬼知道这是要作甚么。
_tmp_floppy_area:
    .fill 1024,1,0

after_page_tables:
// 设置堆栈段，位最后执行 main 函数作准备，main 函数使用 C 语言写的第一个函数，
// 它放在 init/main.c 当中当将 main 的入口地址放入堆栈中之后，最后执行一个 ret
// 语句就可以开始了 C 语言的生活，想到这里，幸福的感觉油然而生☺。
    pushl $0        # These are the parameters to main :-)
    pushl $0
    pushl $0
    pushl $L6           # return address for main, if it decides to.
    pushl $_main
    jmp setup_paging
L6:
    jmp L6              # main should never return here, but
                        # just in case, we know what happens.

/* This is the default interrupt "handler" :-) */
int_msg:
    .asciz "Unknown interrupt\n\r"
.align 2
ignore_int:
// 这是一个 C 语言的函数，所以使用的时候要遵守一定的规则。
    pushl %eax
    pushl %ecx
    pushl %edx
    push %ds
    push %es
    push %fs
    movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    mov %ax,%fs
    pushl $int_msg
    call _printk
    popl %eax
    pop %fs
    pop %es
    pop %ds
    popl %edx
    popl %ecx
```

```
    popl %eax
    iret


/*
 * Setup_paging
 *
 * This routine sets up paging by setting the page bit
 * in cr0. The page tables are set up, identity-mapping
 * the first 16MB. The pager assumes that no illegal
 * addresses are produced (ie >4Mb on a 4Mb machine).
 *
 * NOTE! Although all physical memory should be identity
 * mapped by this routine, only the kernel page functions
 * use the >1Mb addresses directly. All "normal" functions
 * use just the lower 1Mb, or the local data space, which
 * will be mapped to some other place - mm keeps track of
 * that.
 *
 * For those with more memory than 16 Mb - tough luck. I've
 * not got it, why should you :-) The source is here. Change
 * it. (Seriously - it shouldn't be too difficult. Mostly
 * change some constants etc. I left it at 16Mb, as my machine
 * even cannot be extended past that (ok, but it was cheap :-)
 * I've tried to show which constants to change by having
 * some kind of marker at them (search for "16Mb"), but I
 * won't guarantee that's all :-( )
 */
// 最重要的工作开始了，设置分页模式。首先从 0x0000 地址开始清空 5×4k 大小的
// 空间，包括了目录表和 4 个页面描述项表。将 4 个页面描述项表的地址放到目录
// 表中，然后回填页面描述项表。方法是循环 4k 次，每次填入 4 个字节。一共回填
// 了 4×4k 个字节，正好将 pg0—pg3 的 4 张表添完。这个时候用户能够使用的地址
// 是从 0x00000000－－0x00ffffff 共 16M。最后设置 CR0 寄存器，开始了分页模式。
.align 2
setup_paging:
    movl $1024*5,%ecx           /* 5 pages - pg_dir+4 page tables */
    xorl %eax,%eax
    xorl %edi,%edi              /* pg_dir is at 0x000 */
    cld;rep;stosl
    movl $pg0+7,_pg_dir         /* set present bit/user r/w */
    movl $pg1+7,_pg_dir+4       /*  --------- " " --------- */
    movl $pg2+7,_pg_dir+8       /*  --------- " " --------- */
    movl $pg3+7,_pg_dir+12          /*  --------- " " --------- */
    movl $pg3+4092,%edi
```

```
        movl $0xfff007,%eax         /*    16Mb - 4096 + 7 (r/w user,p) */
        std
    1: stosl            /* fill pages backwards - more efficient :-) */
        subl $0x1000,%eax
        jge 1b
        xorl %eax,%eax          /* pg_dir is at 0x0000 */
        movl %eax,%cr3          /* cr3 - page directory start */
        movl %cr0,%eax
        orl $0x80000000,%eax
        movl %eax,%cr0          /* set paging (PG) bit */
        ret             /* this also flushes prefetch-queue */

    .align 2
    .word 0
    idt_descr:
        .word 256*8-1       # idt contains 256 entries
        .long _idt
    .align 2
    .word 0
    gdt_descr:
        .word 256*8-1       # so does gdt (not that that's any
        .long _gdt          # magic number, but it works for me :^)

        .align 3
    // 这里仅仅是为中断向量表开辟了空间，并将一个报错程序放入其中。具体的填充
    // 工作是在用 C 语言实现的 trap_init()完成的，它位于/kernel/traps.c 中。
        _idt:   .fill 256,8,0           # idt is uninitialized


        _gdt:   .quad 0x0000000000000000 /* NULL descriptor */
    // 内存限制 16M，基地址 00a00000，只执行已访问。
        .quad 0x00c09a0000000fff    /* 16Mb */
    // 内存限制 16M，基地址 00200000，只执行已访问。
        .quad 0x00c0920000000fff    /* 16Mb */
        .quad 0x0000000000000000 /* TEMPORARY - don't use */
        .fill 252,8,0               /* space for LDT's and TSS's etc */
```

经过页的初始化之后内存空间分布图：

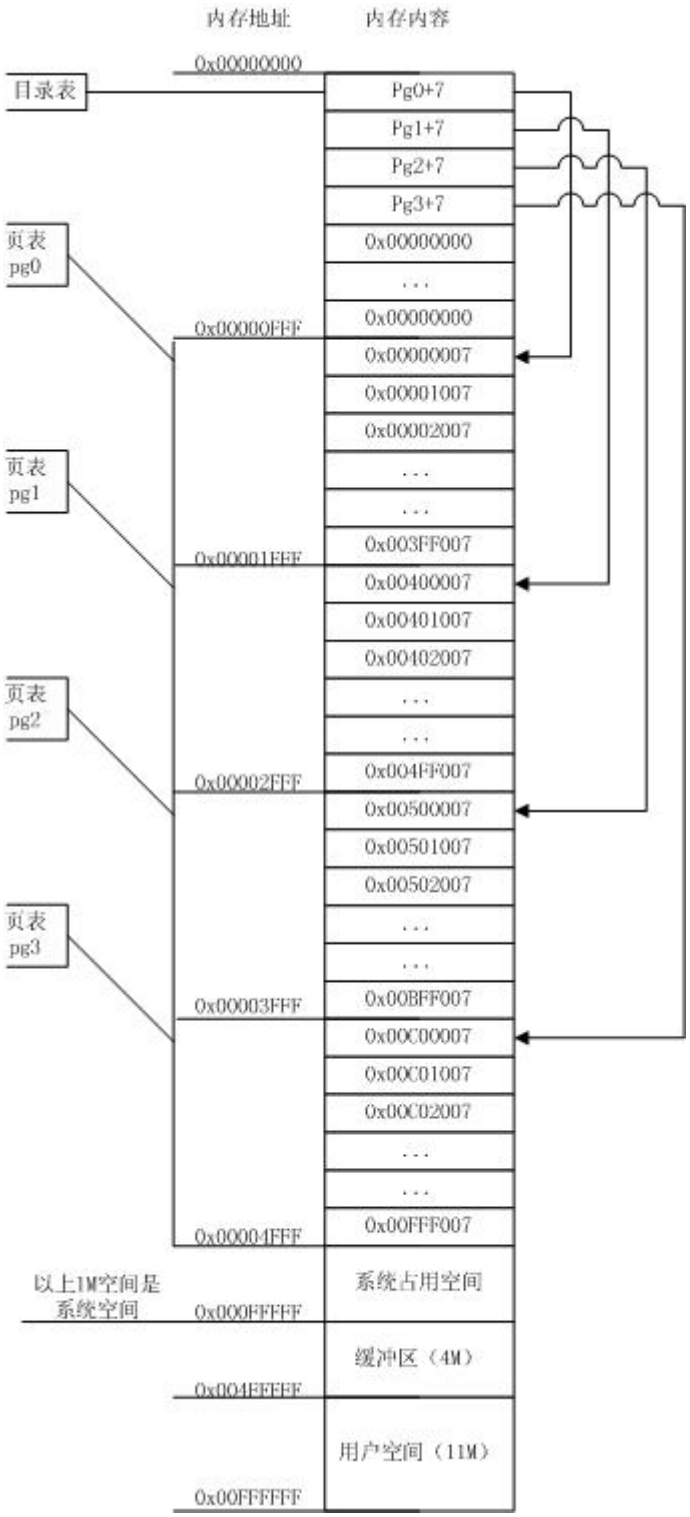


图 二.6 内存空间分布图

保护模式（页式管理）

前面已经提到了，在 setup.s 中已经将内存的模式转变为段管理，这里的页管理就是在段管理的基础上进行的。当分页模式设定了之后，物理地址的寻找就需要先经过段的变换，

然后再通过页来找到实际地址。如何实现分页管理呢？在分页管理中，对给出的 32 位的内存地址有了一个新的格式解释：

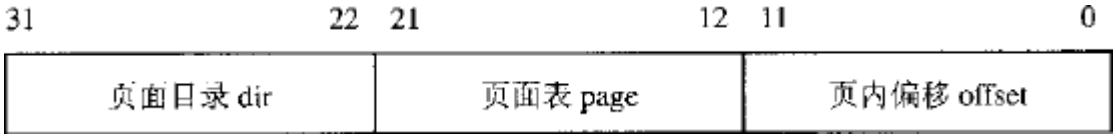


图 二.7 内存线性地址的格式（出自于《Linux 内核源代码情景分析》）

首先 Intel 设置了一个段寄存器 CR3，这个寄存器存储了页面目录的首地址（就同 GDTR 存储了全局描述符表的首地址一样）。然后以给出的内存线性地址中的 dir（页面目录）位段为下标，从目录表中找到页面表的基地址。再以 page（页面表）位段为下标取得页面描述项，最后将页面描述项中的基地址与 offset（页内偏移）位段相加得到物理地址。映射过程可以通过一个图来表示出来：

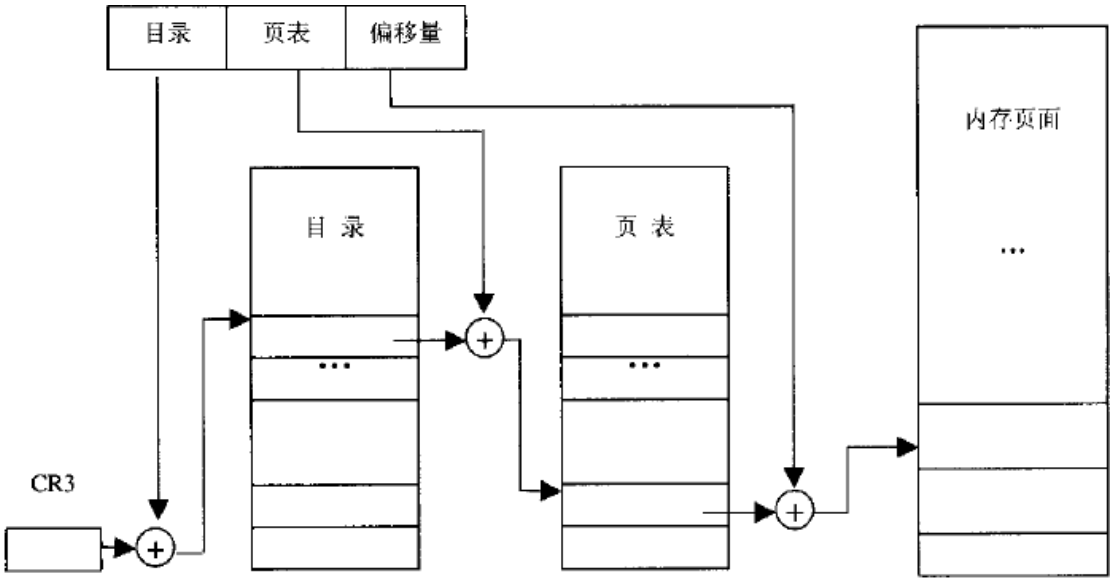


图 二.8 线性地址到物理地址的映射过程（出自于《Linux 内核源代码情景分析》）

我们再来看看目录表项和页面表项的格式，这两个格式几乎完全一样：

页目录表或页表的表项格式	BIT31—BIT12	BIT11—BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
	物理页码	AVL	0	0	D	A	0	0	U/S	R/W	P

AVL(11—9)： 字段供软件使用。

P(0)： 是存在属性位，表示该表项是否有效。P=1 表项有效；P=0 表项无效，此时表项中的其余各位均可供软件使用，80386 不解释 P=0 的表项中的任何其它的位。

R/W(1)是读写属性位，指示该表项所指定的页是否可读、写或执行。若 R/W=1，对表项所指定的页可进行读、写或执行；若 R/W=0，对表项所指定的页可读或执行，但不能对该指定的页写入。但是，R/W 位对页的写保护只在处理器处于用户特权级时发挥作用；当处理器处于系统特权级时，R/W 位被忽略，即总可以读、写或执行.

U/S(2)： 是用户/系统属性位，指示该表项所指定的页是否是用户级页。若 U/S=1，表项所指定的页是用户级页，可由任何特权级下执行的程序访问；如果 U/S=0，表项所指定的页是系统级页，只能由系统特权级下执行的程序访问。下表列出了上述属性位 R/W 和 U/S 所确定的页级保护下，用户级程序

和系统级程序分别具有的对用户级页和系统级页进行操作的权限。

	U/S	R/W	用户级访问权限	系统级访问权限
页级保护属性	0	0	无	读/写/执行
	0	1	无	读/写/执行
	1	0	读/执行	读/写/执行
	1	1	读/写/执行	读/写/执行

由上表可见，用户级页可以规定为只允许读/执行或规定为读/写/执行。系统级页对于系统级程序总是可读/写/执行，而对用户级程序总是不可访问的。与分段机制一样，外层用户级执行的程序只能访问用户级的页，而内层系统级执行的程序，既可访问系统级页，也可访问用户级页。与分段机制不同的是，在内层系统级执行的程序，对任何页都有读/写/执行访问权，即使规定为只允许读/执行的用户页，内层系统级程序也对该页有写访问权。

R/W 和 U/S： 页目录表项中的保护属性位对由该表项指定页表所指定的全部 1K 各 页起到保护作用。所以，对页访问时引用的保护属性位 R/W 和 U/S的值是组合计算页目录表项和页表项中的保护属性位的值所得。下表列出了组合计算前后的保护属性位的值，组合计算是“与”操作。

	目录表项 U/S	页表项 U/S	组合 U/S	目录表项 R/W	页表项 R/W	组合 R/W
组合页的保护属性	0	0	0	0	0	0
	0	1	0	0	1	0
	1	0	0	1	0	0
	1	1	1	1	1	1

最后，当将 CR0 寄存器的 PG 位设置位 1，就开始了分页管理。

新引入的控制寄存器格式说明：

	CRX	BIT31	BIT30—BIT12	BIT11—BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
控制寄存器	CR0	PG	0000000000000000		ET	TS	EM	MP	PE
	CR1	保留							
	CR2	页故障线性地址							
	CR3	页目录表物理页码		000000000000					

汇编语言与 C 语言的混合编程

在 head.s 文件中有两处调用了使用 C 语言书写的函数——_printk()和_main()，同时使用汇编语言分配的空间可以被 C 语言写的函数使用。另外这个汇编程序最后使需要同 C 语言写的程序一起编译到一个文件中去的。有没有想过这是如何实现的呢？

在讲解之前我们做一个试验，试验的工具是 masm 的连接器 link.exe，nasm 编译器 nasmw.exe，turbo c 2.0 的编译器 tcc.exe，.exe 文件到.com 文件的转换工具 exe2com.exe，以及 debug.exe。

这个试验是先用 C 语言写一个在通常的意义下是很奇怪的程序（没有 main 函数），然后用汇编语言写一个调用的程序。将这两个文件联合编译成一个.com 文件，使用 debug.exe

对这个.com 文件进行反汇编，来查看这个由“四不象”的编译器到底参生了一个什么怪胎。

先建立一个 C 语言文件 e.c。内容是：

```
extern void assigne(int);
void e_main()
{
    assigne(6);
}
void assigne(int c)
{
     c = 5;
}
```

这个 C 语言的程序没有 main 函数，这在通常的意义下是极其不正常的，可以说这个程序是不可能执行的。没错，使用任何一个集成环境来编译这段 C 程序都是不可能成功的（你自己写的不算）。因为这个程序没有入口点。（这个程序是我随意敲的一个，我也不知道是什么功能，只知道很简单）换一个角度来看这段代码，把它仅仅看作是用 C 写的两个函数。（很像是.dll，其实相差很多的）

现在，我想在汇编语言中调用这两个函数。建立一个汇编文件 start.asm。

```
[BITS 16]
[global start]
[extern _e_main]
start:
     call _e_main
```

这个程序就一条指令，就是调用 C 语言中的 e_main()函数。这里写成_e_main 是因为 C 语言编译的使用会在函数前自动加上‘_’。

现在进行编译了，千万不要在集成环境中编译，无法通过的。编译的使用将这两个文件仅仅编译成.obj 文件。命令格式为：

tcc –mf –oe.obj –c –e.c

表示以微型模式只进行编译，生成 e.obj 文件。

Nasmw –f obj –o start.obj –start.asm

表示以目标文件的格式编译成 start.obj 文件。

编译成了两个目标文件（e.obj 和 start.obj）后，需要将他们连接成可执行程序 e.exe。

link start.obj e.obj, e.exe,,,

输入的时候要注意这两个文件的顺序，一会儿我会讲为什么。

使用 exe2com 将这个 e.exe 转换成 e.com。因为.exe 文件由自己的一套格式，不容易分析，变成.com 文件就很好了，它就是机器码。

好了，现在让我们来看看这个怪胎是什么吧。用 debug –u 来看看反汇编这个 e.exe 文件。结果如下：

```
xxxx:0100     E80000   CALL 0003                    ;call _e_main

xxxx:0103     B80600   MOV  AX,0006                ;这是 e_main()
xxxx:0106     50       PUSH AX                     ; assigne(6)
xxxx:0107     E80200   CALL 010C
xxxx:010A     59       POP  CX
xxxx:010B     C3       RET
```

```
xxxx:010C    55          PUSH BP                          ;这是 assigne(int c)
xxxx:010D    8BEC        MOV BP, SP                       ;c = 5
xxxx:010F    C746040500  MOV WORD PTR [BP+04], 0005
xxxx:0114    5D          POP BP
xxxx:0115    C3          RET
```

产生的这个可执行文件还是挺正确的吧。前面说道在连接的时候要注意两个.obj 文件的输入顺序。现在我们看看，如果这样连接会怎么样：

link e.obj start.obj, e.exe,,,

还是将其转变成 e.com 之后，用 debug 反汇编。

```
xxxx:0100    B80600   MOV   AX,0006                      ;这是 e_main()
xxxx:0103    50       PUSH AX                           ; assigne(6)
xxxx:0104    E80200   CALL 010C
xxxx:0107    59       POP   CX
xxxx:0108    C3       RET

xxxx:0109    55       PUSH BP                           ;这是 assigne(int c)
xxxx:010A    8BEC     MOV BP, SP                        ;c = 5
xxxx:010C    C746040500  MOV WORD PTR [BP+04], 0005
xxxx:0111    5D       POP BP
xxxx:0112    C3       RET

xxxx:0113    00E8     ADD AL, CH                        ;我不知道这是什
                                                        ;么，别问我。
xxxx:0115    E9FF                                       ;没有相应的机器码
```

很明显这次编译错了。不是连接器不智能，是你表达的不正确。连接的时候需要按照程序流程进行安排。否则，必然是要出错的。

通过这个例子可以看到，汇编与 C 语言混合编程（C 语言中嵌入汇编指令不在这里讨论）的时候需要。先将汇编程序和 C 程序编译成.obj 格式的文件，然后使用连接器将目标文件连接到一起。这样，C 语言和汇编语言就可以混合使用了。（别忘了，C 的编译器会在函数名前加上‘_’）

在 Linux0.11 的代码群中有一个 makefile 文件，这个文件使用于知道编译过程的。其中有一句话：

```
$(LD) $(LDFLAGS) boot/head.o init/main.o \
$(ARCHIVES) \
$(DRIVERS) \
$(MATH) \
$(LIBS) \
```

LD 代表 gld 连接器，LDFLAGS 是连接参数。Boot/head.o 就是这个汇编写的 head.s 的目标文件，其后跟的都是相应的内核文件，于是 head.s 的代码就连接到了内核的最前面。这样在 head.s 中使用内核文件中定义的 C 语言函数就是一件很正常的事情了。同时 C 语言写的内核文件也可以使用在汇编语言中分配的变量了。如_gdt, _idt。

保护模式下内存的寻址总结

前面已经在 setup.s 和 head.s 中分别介绍了 Linux0.11 如何在保护模式下进行内存的寻址，由于比较分散，所以我在这里进行一次总结。

在保护模式下，Linux0.11 先进行段变换，然后是页变换，将绝对地址找到。可以表示成：逻辑地址→线性地址（经过段变换）→绝对地址（经过页变换）。顺便说一下，这些复杂的变换和各种检验都是由硬件来实现的，于操作系统无关。系统仅仅是进行相应的配置，为这些工作提供条件。

示意图如下：

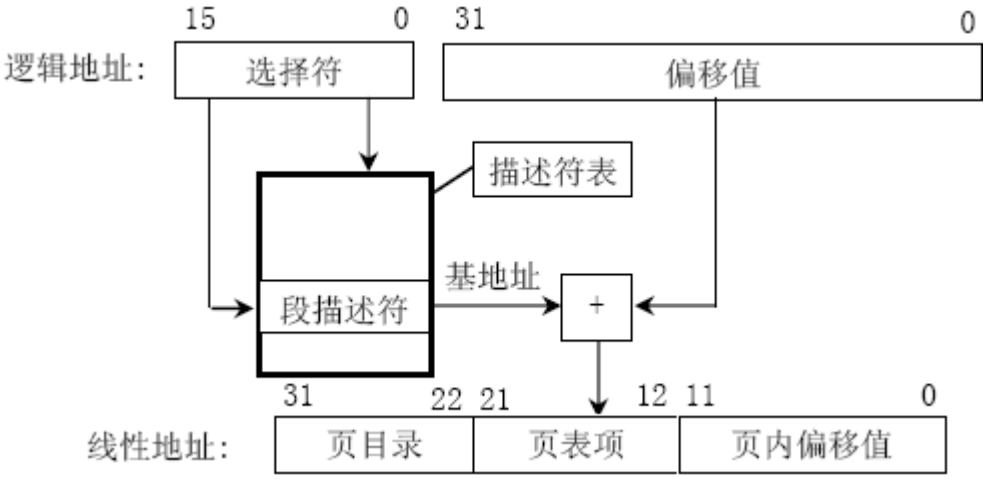


图 二.9 段变换示意图（出自于《Linux0.11 源码完全注释》）

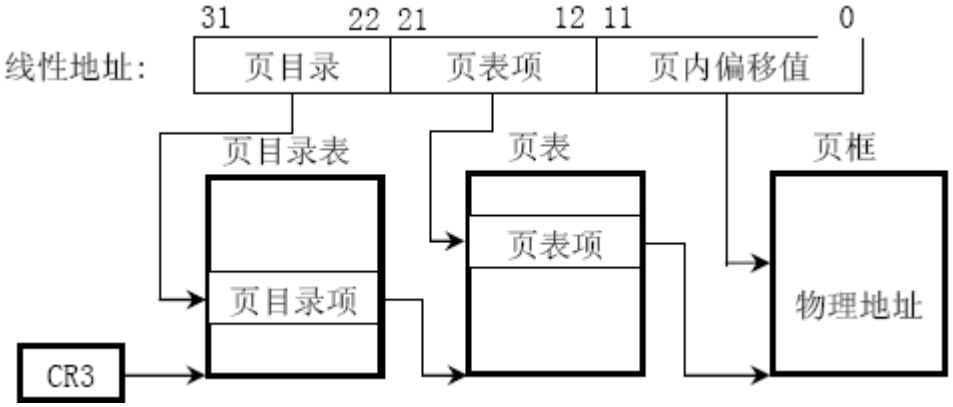


图 二.10 页变换示意图（出自于《Linux0.11 源码完全注释》）

其中段描述符表的首地址由 GDTR 给出，段描述符的选择在段寄存器中得到，页目录表的首地址由 CR3 给出，页表项的地址由页目录项得到。

在转换的过程中会经过一系列的特权级检验。

保护模式启动是通过设置 CR0 寄存器的 PE（0）位为 1 实现，分页的启动是设置 CR0 的 PG（31）位为 1 实现的。

在 Linux0.11 中是使用分页式来管理内存的，但是由于 Intel 的结构的设计使得分页式的管理一定要建立在段式管理的基础之上的。因此，Linux0.11 不得已还是要按照 Intel 的要求

去做段的管理，但是实际上这个段式管理是没有实际意义的。对于安全的考虑实际上还是由页式管理来完成的。为什么这么说呢？我们看看系统的是如何设定 GDT 的（在 head.s 中）：

```
_gdt:    .quad 0x0000000000000000 /* NULL descriptor */
         .quad 0x00c09a0000000fff  /* 16Mb */
         .quad 0x00c0920000000fff  /* 16Mb */
         .quad 0x0000000000000000 /* TEMPORARY - don't use */
         .fill 252,8,0             /* space for LDT's and TSS's etc */
```

为了清楚看到它的格式我将它转变成二进制：

代码段：

<0000 0000><1100 0000><1001 1010><0000 0000><0000 0000><0000 0000><0000 1111><1111 1111>

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)(00)			属性(C0 9A)		Base(23...0)(00 00 00)					Limite(15...0)(0F FF)			
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(1)	D(1)	0	AVL(0)	Limit(19...16)(0)				P(1)	DPL(00)		DT(1)	TYPE(A)			

基地址＝0x00000000

段限制＝0x00FFF　　4k×4k ＝ 16M

G=1　　颗粒密度 4k

DT＝1　　段描述符

Type＝A　　可读可运行

DPL＝0　　系统级的段描述符

D＝1　　32 位的偏移地址

数据段：

<0000 0000><1100 0000><1001 0010><0000 0000><0000 0000><0000 0000><0000 1111><1111 1111>

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)(00)			属性(C0 92)		Base(23...0)(00 00 00)					Limite(15...0)(0F FF)			
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(1)	D(1)	0	AVL(0)	Limit(19...16)(0)				P(1)	DPL(00)		DT(1)	TYPE(2)			

基地址＝0x00000000

段限制＝0x00FFF　　4k×4k ＝ 16M

G=1　　颗粒密度 4k

DT＝1　　段描述符

Type＝2　　可读可写

DPL＝0　　系统级的段描述符

D＝1　　32 位的偏移地址

在从逻辑地址到线性地址的转换的时候，对于使用这样的段描述符来说，由于基地址是 0x0000000，所以转换的线性地址与逻辑地址的值是相同的。这样就使得整个逻辑地址到物理地址的变换的任务交给了页式管理来完成。

第三章系统的初始化

系统初始化程序存放在/init/main.c 文件中，系统的启动过程全部集中到这一个文件中了。这个文件可以不用细看，等到进程系统看完了回头再看。这个文件主要是做系统的各个功能模块的初始化工作，然后调用 sh 完成启动。head.s 中的 iret 指令使得 CPU 的执行转到了 main()函数处开始执行。

0 号进程

0 号进程就是初始进程，用于执行 main 函数的。他会产生 1 号进程运行 init，以后的工作就由 1 号进程来完成了。

```
void main(void)          /* This really IS void, no error here. */
{                /* The startup routine assumes (well, ...) this */
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
        ROOT_DEV = ORIG_ROOT_DEV;
        drive_info = DRIVE_INFO;
//      开始内存的初始化
        memory_end = (1<<20) + (EXT_MEM_K<<10);
        memory_end &= 0xfffff000;
        if (memory_end > 16*1024*1024)
                memory_end = 16*1024*1024;
        if (memory_end > 12*1024*1024)
                buffer_memory_end = 4*1024*1024;
        else if (memory_end > 6*1024*1024)
                buffer_memory_end = 2*1024*1024;
        else
                buffer_memory_end = 1*1024*1024;
        main_memory_start = buffer_memory_end;
#ifdef RAMDISK
        main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
#endif
        mem_init(main_memory_start,memory_end);
// 中断向量初始化
        trap_init();
// 块设备初始化
        blk_dev_init();
// 字符设备初始化（这里仅仅是留了个接口，没有实现）
        chr_dev_init();
```

```
// 终端初始化
        tty_init();
// 启动时间初始化
        time_init();
// 调度程序初始化
        sched_init();
// 缓冲区初始化
        buffer_init(buffer_memory_end);
// 硬盘初始化
        hd_init();
// 软盘初始化
        floppy_init();
// 开启中断。因为在此之前中断一直是关闭的，从此开始就可以中断了。Sti()是宏命令，
// 它在/include/asm/system.h 中定义为：#define sti() __asm__ ("sti"::)
        sti();
// 将系统转入用户模式，以后内核要想工作的话也要用系统调用来完成了。
        move_to_user_mode();
// 产生 1 号进程，用来执行函数 init
        if (!fork()) {          /* we count on this going ok */
                init();
        }
/*
 *    NOTE!!    For any other task 'pause()' would mean we have to get a
 * signal to awaken, but task0 is the sole exception (see 'schedule()')
 * as task 0 gets activated at every idle moment (when no other tasks
 * can run). For task0 'pause()' just means we go check if some other
 * task can run, and if not we return here.
 */
// 0 号进程的工作到此结束，但是程序可不能退出，所以就循环吧。
        for(;;) pause();
}
```

有系统模式转向用户模式

在调用 move_to_user_mode 之前，main 函数一直在系统状态下运行，但是以后的进程是应该在用户模式下运行的，因此要在调用 fork 之前将程序的运行模式转变为用户模式。方法是模拟中断返回，我们知道在用户模式的时候，如果发生中断会调用系统的中断服务程序。中断服务程序是运行在系统模式下的，因此会发生栈转换。当中断返回的时候，会恢复到中断之前的状态，这就包括了将运行模式转换为用户模式。中断的处理不过是压栈而已，因此我们可以利用压栈来模拟中断。然后，用一个返回指令来完成现场恢复的工作。如果这个过程用图表示时这样的：

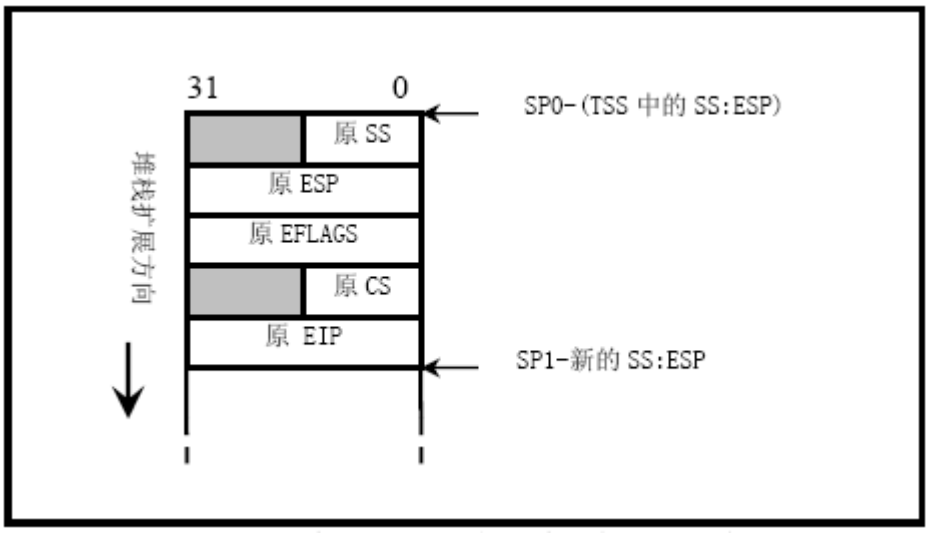


图 三.1 中断发生时的栈中情况

<include/asm/system.h>

```
#define move_to_user_mode() \
// 将原来的 esp 保存到 eax 中
__asm__ ("movl %%esp,%%eax\n\t" \
// 模拟压入用户态的 ss 寄存器的值，0x17 对应的是 LDT 的数据段
    "pushl $0x17\n\t" \
// 模拟压入用户态的 esp 寄存器的值
    "pushl %%eax\n\t" \
// 模拟压入标志寄存器的值
    "pushfl\n\t" \
// 模拟压入了用户执行的 cs，0x0f 对应的是 LDT 的代码段
    "pushl $0x0f\n\t" \
// 模拟压入中断发生时的用户 eip，标号 1 是中断返回时要执行的地址，在下面
    "pushl $1f\n\t" \
// 中断返回，这个时候会恢复现场，导致到用户态的转变
    "iret\n" \
// 中断返回时会从这里继续执行
    "1:\tmovl $0x17,%%eax\n\t" \
    "movw %%ax,%%ds\n\t" \
    "movw %%ax,%%es\n\t" \
    "movw %%ax,%%fs\n\t" \
    "movw %%ax,%%gs" \
    :::"ax")
```

1 号进程

1 号进程用于执行函数 init，这个函数会再次产生一个进程来执行 sh。1 号进程是非常重要的，因为大部分的进程退出的时候，剩余的工作都是由 1 号进程来处理的，包括获取返回值，释放 PCB 占有的 1 页内存等。

<init/main.c>

```
    // sh 的参数
    static char * argv_rc[] = { "/bin/sh", NULL };
    static char * envp_rc[] = { "HOME=/", NULL };

    static char * argv[] = { "-/bin/sh",NULL };
    static char * envp[] = { "HOME=/usr/root", NULL };

    void init(void)
    {
        int pid,i;

        setup((void *) &drive_info);
        (void) open("/dev/tty0",O_RDWR,0);
        (void) dup(0);
        (void) dup(0);
        printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
            NR_BUFFERS*BLOCK_SIZE);
        printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);
    // 产生一个用于运行 sh 的进程
        if (!(pid=fork())) {
            close(0);
            if (open("/etc/rc",O_RDONLY,0))
                _exit(1);
    // 运行可执行文件 sh，它是系统服务程序，位于文件系统盘中
            execve("/bin/sh",argv_rc,envp_rc);
            _exit(2);
        }
        if (pid>0)
    // 如果发现是 sh 执行结束，就向下执行。如果，是其它进程退出的话，由于进程号不
    // 同因此会继续循环等待。
            while (pid != wait(&i))
                /* nothing */;
        while (1) {
            if ((pid=fork())<0) {
                printf("Fork failed in init\r\n");
                continue;
            }
            if (!pid) {
                close(0);close(1);close(2);
                setsid();
                (void) open("/dev/tty0",O_RDWR,0);
                (void) dup(0);
                (void) dup(0);
```

```
                _exit(execve("/bin/sh",argv,envp));
            }
            while (1)
                if (pid == wait(&i))
                    break;
            printf("\n\rchild %d died with code %04x\n\r",pid,i);
            sync();
        }
        _exit(0);  /* NOTE! _exit, not exit() */
}
```

第四章进程的描述

从这一章开始就是对整个进程系统的分析了，这是一个大块，如果进程系统了解了，就可以说对 Linux0.11 的工作机制了解了。进程系统就好像是操作系统的大脑，一切工作的指令都要由它来发出。

从操作系统原理课上我们可以知道关于进程的理论上的定义，以及进程的组成：进程＝程序＋数据＋PCB。操作系统就是通过对 PCB 的操作来实现对进程的管理。那么在 Linux0.11 中是如何表示 PCB 这个结构的呢？Linux 将任务和进程当作一回事，它使用一个叫做 task_struct 的结构来描述进程。为了便于理解进程数据结构中各个域的作用，我讲这些域按照功能划分为几个部分，分别讲解。

<include/linux/sched.h>

与进程调度相关：

```
struct task_struct {
/* these are hardcoded - don't touch */
    long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
    long counter;
    long priority;
```

这个部分主要是在进程的调度的时候使用。在汇编写的代码中使用过这几个域的值，由于汇编使用这几个域的方法就是利用这几个域的偏移量，因此这几个值的顺序是很重要的，不能发生改变。这几个域的作用如下：

state : 用于标识进程的运行状态。它可能是一下几个值：

```
#define TASK_RUNNING        0
#define TASK_INTERRUPTIBLE  1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE         3
#define TASK_STOPPED        4
```

TASK_RUNNING 表示进程已经准备好占用 CPU。相当于“就绪态”；

TASK_INTERRUPTIBLE 表示进程正处于休眠状态，它可能在等待一个信号。这个等待通常比较长；

TASK_UNINTERRUPTIBLE 也表示进程正处于休眠状态。它可能是在等待某一个事件发生，因此所有的信号都会被挂起直到这个事件发生。通常它的等待时间比较短；

TASK_ZOMBIE 表示这个进程已经中止了，但是还没有从系统中删除掉。这是很必要的，如果一个进程结束的时候需要向父进程返回一些信息。如果这个时候就从系统中将进程删除，父进程就无法得到这个返回值了。因此是否应该删除应该由父进程来维护；

TASK_STOPPED 表示进程暂停。有的时候进程既没有等待信号，也不是中止，例如调试程序的时候。为了标识这种情况，设置了这样的状态；

counter：进程剩余的时间片。为了允许各个进程相对公平的使用 CPU 引入了分时这个概念。就是为每个进程设置一个最长运行时间，当这个时间到了就重新调度进程以允许其他的进程占有 CPU。时间片是进程运行的最小时间单位。在 Linux 中每个时间片。是 10ms。它的初始值与该进程的优先级的值相等。

priority:进程的优先级。在 Linux0.11 中优先级越高占用 CPU 的时间越长。

与信号处理有关：

long signal;
struct sigaction sigaction[32];
long blocked; /* bitmap of masked signals */

在信号处理一章我会详细的对这几个域进行说明。这里先对它们有个印象就可以了。也有汇编语言使用了这几个域，因此它们的顺序也不能改变。以后说明的域就与顺序无关了，可以根据个人爱好随意改变顺序。介绍一个概念－－位图，位图是一个数值，通常这个数值本身没有实际意义，将数值以二进制的形式表示后，每一位标识一个状态。（实际上没有明确的概念，是根据我的理解编的☺）

signal 和 **blocked** 都是 32 位的位图，分别叫做“信号请求寄存器”和“信号屏蔽寄存器”。

signaction[32]是存储与信号的处理相关信息的数组，叫做“信号向量表”。

与进程空间的安排有关：

unsigned long start_code,end_code,end_data,brk,start_stack;

进程是程序的载体，程序需要加载到内存中才能够运行。进程在内存中有着它独有的虚拟空间，Linux0.11 中这个虚拟空间是 64M。为了组织好这 64M 的内存空间，设置了代码段和数据段。Linux 中代码段与数据段的起始地址是一样的，并且它不支持代码段与数据段分开这种情况。

start_code 是代码段（数据段）的起始地址。
start_stack 是栈的起始地址。
end_code 和 **end_data** 分别是代码段的长度和数据段的长度。
brk 是总的数据长度。

说明：

1. 这里的起始地址都是指在这 64M 空间内的偏移地址，由于虚拟地址都是 32 位，因此这些起始地址使用了长整形。
2. 段的长度都是以字节为单位的。
3. 通常 start_code 是段的最低端，start_stack 是段的最高段。因为栈增长方向是向下的。

进程的一般描述信息：

/* various fields */

int exit_code;

long pid,father,pgrp,session,leader;

为了能够管理进程就需要能够唯一的标识一个进程，同时进程与进程之间可能是有连带关系的。这个关系可能是父子关系，也可能是隶属于同一个组的关系。组是很有用的，由于组的概念一个进程可以向多个进程发送同一个消息。在多用户的情况下需要为每一个访问的用户提供一个进程，同时这个进程可能会产生子进程（这可以说是一定的）。为了区分不同的用户也需要要一个标识。

exit_code 是程序返回值；

pid 是对进程的唯一标识，通过这个值可以唯一的找到这个进程；

father 是父进程的 PID 号，这样就形成了“家谱树”；

pgrp 组的标识，这个值相同的进程位于同一个组内；

session：用于标识不同的来访的用户进程，有这些进程产生的子进程与相应的父进程的 session 值是相等的；

leader:来访的进程中应该有一个“头儿”，leader 就是标识这个的。

与身份验证有关：

unsigned short uid,euid,suid;

unsigned short gid,egid,sgid;

与时间有关：

long alarm;

long utime,stime,cutime,cstime,start_time;

一个进程可能需要暂停一段时间然后再苏醒过来，因此需要有一个变量来记录这个时间，同样这个时间的单位是时间片。同时一个进程还记录了一些其他的时间信息用于查询，包括了进程的起始时间，进程处于用户态的时间，进程处于系统态的时间，子进程处于用户态的时间，子进程处于系统态的时间。

alarm 进程睡眠的时间（定时器），如果为 0 表示没睡眠；

utime,stime,cutime,cstime,start_time 分别表示进程处于用户态的时间，进程处于系统态的时间，子进程处于用户态的时间，子进程处于系统态的时间，进程的起始时间。

与程序文件有关：

/* file system info */

int tty; /* -1 if no tty, so it must be signed */

unsigned short umask;

struct m_inode * pwd;

struct m_inode * root;

struct m_inode * executable;

unsigned long close_on_exec;

struct file * filp[NR_OPEN];

局部描述符表：

/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */

struct desc_struct ldt[3];

对于系统内核来说使用的是全局描述符表，而对于每一个进程来说就要是用局部描述符表了。已经说过了 Linux 其实没有使用段式管理，仅仅使用了页式管理。但是为了响应 Intel 的号召还是要用到局部描述符表。既然有了局部描述符表就顺便用用吧，在刚开始分配进程空间的时候就利用到了。

任务状态段：

/* tss for this task */

struct tss_struct tss;

};

进程在切换的时候是需要保存进程切换前各个寄存器状态的，这些状态使用了一个叫做任务状态段的数据结构来保存。这个结构还是由 Intel 提出来的，为了保存状态信息 Intel 从硬件上提供了支持，就是说只要你将 tss（任务状态段）这个结构设置好了，当你进行某种破坏了程序执行顺序的操作的时候，Intel 会自动的将当前的寄存器状态记录到 tss 中。（详细的讲解将在后面叙述）Linux0.11 中在进行进程切换的时候就是利用了这个方便减少了很多的代码。但是在以后的版本中，Linux0.11 就不再是利用这种硬件机制了，改用了软件的方法。软件方法可扩展性比硬件方法好。不要以为有硬件方法一定就快，其实硬件方法与软件方法的执行速度差不多。

关于 tss 结构的详细介绍将在下一章讲解。

计算机对进程的感知

计算机对进程的感知是依靠 PCB 结构的（就是这个 task_struct），同时一个系统中会存在多个进程。为了组织好这些进程，Linux0.11 中使用了一个线性表来存放这些进程。这个线性表就是 task。

<kernel/sched.c>

struct task_struct * task[NR_TASKS] = {&(init_task.task), };

这样操作系统就可以通过 task 来找到任何一个进程了。可以看到在表中初始的时候就设置了一个值&(init_task.task)，这个就是指向初始进程的指针，表示初始进程存放在 task[0] 中。关于初始进程 init_task.task 将在下一章介绍。

在操作系统原理课上经常提到就绪队列，对于 Linux0.11 来说它是一个逻辑上的结构，实际上不同状态的进程是很随机的放到 task 中的。

为了清晰的描述这个线性表的头和尾，Linux 又定义了两个宏。<include/kernel/sched.h>

#define FIRST_TASK task[0]

#define LAST_TASK task[NR_TASKS-1]

第五章进程系统的初始化

在程序设计中，可以说做任何一个部分的前提就是初始化。对于代码的分析，我一向比较认可从初始化开始入手。尤其对于一些起始的数据，如果没有理解好会对以后代码的理解留下祸根。

在此我想讲一件分析代码时发生的事：当我分析 fork 函数的时候，发现它在调用一个页拷贝的函数的时候会改变映射数组 mem_map[]中所有的值，这样导致的结果就是再次要求分配内存页的时候无可用内存，这不就是 bug 了吗？这件事困扰了我很长时间，我看了很多遍代码也没有解决。多亏了赵炯博士的提示，使我感觉到问题的根源可能来至于初始化。于是我对起始进程（INIT_TASK）重新分析，发现了一处理解的错误。就是因为这个错误使我对后面的理解产生了偏差。这件事之后，我对初始化更加认真，尤其是在初始化的时候对各个变量设置的值（这是重中之重）。

因此，在分析各个功能函数之前，我需要先讲一下进程的初始化。

程序的初始化可以分为两种，一种是在分配存储空间的时候已经安排好的，例如 int mem[3]={1, 2, 3};另一种是在程序运行中设置的，例如 mem[0]=mem[1]=mem[2]=0。进程的初始化就包括了这两种。由于 Linux 中都是通过复制父进程来产生新进程的，因此必须有一个进程是一切一切进程的祖先进程，这就是初始进程。它的 task_struct 结构是直接用代码写成的。在为其分配空间的时候，值就已经设定好了。另外，进程在运行的时候会使用一些固定的寄存器，初始化的时候也要讲这些寄存器的值设定好。同时要为一些表结构分配内存空间。

初始进程

刚才已经提到了初始进程是一切进程的祖先进程，它是通过“硬编码”写到程序中的。初始进程代表的是内核的进程，但是它不会被调度程序调度的。它的目的倒不是为了让内核运行，而是为其它的进程提供一个复制的基点。代码如下：

```
<include/kernel/sched.h>
    /*
     *  INIT_TASK is used to set up the first task table, touch at
     * your own risk!. Base=0, limit=0x9ffff (=640kB)
     */
    #define INIT_TASK \
    /* state etc */ {
                    0,\         //state=0(TASK_RUNNING) 表示可以运行（就绪）
                    15,\        //counter=15  任务运行的时间片(150ms)，开始的时候与
                                //priority 的值相等
                    15, \       //priority=15  运行的优先权
    /* signals */   0,\         //signal=0     表示没有任何信号
                    {{},},\     //sigaction[32]={{},}  信号向量表为空
                    0, \        //blocked=0    表示不阻塞任何信号
    /* ec,brk... */ 0,\         //exit_code=0
```

```
        0,\         //start_code=0      代码段起始地址
        0,\         //end_code=0        代码段的长度
        0,\         //end_data=0        数据段的长度
        0,\         //brk=0
        0,\         //start_stack=0     表示还没有分配栈
/* pid etc.. */ 0,\         //pid=0             0 号进程
        -1,\        //father=-1         父进程号（表示没有父进程）
        0,\         //pgrp=0        父进程组号
        0,\         //session=0
        0,\         //leader=0
/* uid etc */   0,\         //uid=0         用户标识号（用户 id）
        0,\         //euid=0        有效用户 id
        0,\         //suid=0        保存的用户 id
        0,\         //gid=0         组标识号（组 id）
        0,\         //egid=0        有效组 id
        0,\         //sgid=0        保存的组 id
/* alarm */     0,\         //alarm=0       报警定时值（没有设定）
        0,\         //utime=0       用户态时间
        0,\         //stime=0       系统态时间
        0,\         //cutime=0      子进程用户态时间
        0,\         //cstime=0      子进程系统态运行时间
        0,\         //start_time=0  进程开始运行时刻
/* math */      0,\         //used_math=0是否使用了协处理器
/* fs info */   -1,\        //tty=-1            进程没有使用 tty
        0022,\      //umask=0022 文件创建属性屏蔽位
        NULL,\      //pwd=NULL  当前工作目录 I 节点
        NULL,\      //root=NULL  根目录 I 节点
        NULL,\      //executable=NULL       执行文件 I 节点结构
        0,\         //close_on_exec=0  执行时关闭文件句柄位图标志
/* filp */  {NULL,},\       //filp[]={NULL}     进程使用的文件表结构
/* ldt */   {\                  //ldt[3]
            {0,0},\
        {0x9f,0xc0fa00},\
            {0x9f,0xc0f200},\
        },\
/*tss*/     {0,PAGE_SIZE+(long)&init_task,0x10,0,0,0,0,(long)&pg_dir,\
         0,0,0,0,0,0,0,0,\
         0,0,0x17,0x17,0x17,0x17,0x17,0x17,\
         _LDT(0),0x80000000,\
            {}\
        },\
}
```

局部描述符表的设定

描述符表的结构已经在前面讲过了，这里又分析一遍是为了引起注意。因为它对后面的理解影响很大。

起始进程的这 3 个局部描述符展开后是：

0x00 00 00 00 00 00 00 00

0x00 c0 fa 00 00 00 00 9f

0x00 c0 f2 00 00 00 00 9f

按照描述符的格式将代码段的数据填入描述符

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)(00)			属性(C0 FA)		Base(23...0)(00 00 00)			Limite(15...0)(00 9F)	
存储段描述符属 性	Byte m+6								Byte m+5	
	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
	G(1)	D(1)	0	AVL(0)	Limit(19...16)(0)				P(1)	DPL(11)		DT(1)	TYPE(A)			

这说明，初始进程还是以 0x00000000 为基地址，但是段界限却是 0x09F。对应的是 160×4k 的访问空间。同时 DPL 也变成了 3。(我开始的时候将段界限当成是 0x07FF，对应 16M)

数据段于代码段很像，就不分析了。

任务状态段（TSS）

实模式的时候执行转移指令，CALL 指令分为段间转移和段内转移。保护模式下也一样，只是在进行段间转移的时候发生了一些变化，其中一项就是保存环境到内存中。因此引出了一个数据结构 TSS。它的格式如下：

任务状态段基本部分的格式	BIT31—BIT16	BIT15—BIT1	BIT0	OFFSET
	0000000000000000	链接字段		0
	ESP0			4
	0000000000000000	SS0		8
	ESP1			0CH
	0000000000000000	SS1		10H
	ESP2			14H
	0000000000000000	SS2		18H
	CR3			1CH
	EIP			20H
	EFLAGS			24H
	EAX			28H
	ECX			2CH
	EDX			30H
	EBX			34H

	ESP		38H
	EBP		3CH
	ESI		40H
	EDI		44H
	0000000000000000	ES	48H
	0000000000000000	CS	4CH
	0000000000000000	SS	50H
	0000000000000000	DS	54H
	0000000000000000	FS	58H
	0000000000000000	GS	5CH
	0000000000000000	LDTR	60H
	I/O 许可位图偏移	000000000000000 T	64H

TSS 的基本格式由 104 字节组成。这 104 字节的基本格式是不可改变的，但在此之外系统软件还可定义若干附加信息。基本的 104 字节可分为链接字段区域、内层堆栈指针区域、地址映射寄存器区域、寄存器保存区域和其它字段等五个区域。我分别讲解这五个区域：

1. 寄存器保存区域

寄存器保存区域位于 TSS 内偏移 20H 至 5FH 处，用于保存通用寄存器、段寄存器、指令指针和标志寄存器。当 TSS 对应的任务正在执行时，保存区域是未定义的；在当前任务被切换出时，这些寄存器的当前值就保存在该区域。当下次切换回原任务时，再从保存区域恢复出这些寄存器的值，从而，使处理器恢复成该任务换出前的状态，最终使任务能够恢复执行。

从上表可见，各通用寄存器对应一个 32 位的双字，指令指针和标志寄存器各对应一个 32 位的双字；各段寄存器也对应一个 32 位的双字，段寄存器中的选择子只有 16 位，安排再双字的低 16 位，高 16 位未用，一般应填为 0。

2. 内层堆栈指针区域

为了有效地实现保护，同一个任务在不同的特权级下使用不同的堆栈。例如，当从外层特权级 3 变换到内层特权级 0 时，任务使用的堆栈也同时从 3 级变换到 0 级堆栈；当从内层特权级 0 变换到外层特权级 3 时，任务使用的堆栈也同时从 0 级堆栈变换到 3 级堆栈。所以，一个任务可能具有四个堆栈，对应四个特权级。四个堆栈需要四个堆栈指针。

TSS 的内层堆栈指针区域中有三个堆栈指针，它们都是 48 位的全指针(16 位的选择子和 32 位的偏移)，分别指向 0 级、1 级和 2 级堆栈的栈顶，依次存放在 TSS 中偏移为 4、12 及 20 开始的位置。当发生向内层转移时，把适当的堆栈指针装入 SS 及 ESP 寄存器以变换到内层堆栈，外层堆栈的指针保存在内层堆栈中。没有指向 3 级堆栈的指针，因为 3 级是最外层，所以任何一个向内层的转移都不可能转移到 3 级。

但是，当特权级由内层向外层变换时，并不把内层堆栈的指针保存到 TSS 的内层堆栈指针区域。实际上，处理器从不向该区域进行写入，除非程序设计者认为改变该区域的值。这表明向内层转移时，总是把内层堆栈认为是一个空栈。因此，不允许发生同级内层转移的递归，一旦发生向某级内层的转移，那么返回到外层的正常途径是相匹配的向外层返回。

3. 地址映射寄存器区域

从虚拟地址空间到线性地址空间的映射由 GDT 和 LDT 确定，与特定任务相关的部分

由 LDT 确定，而 LDT 又由 LDTR 确定。如果采用分页机制，那么由线性地址空间到物理地址空间的映射由包含页目录表起始物理地址的控制寄存器 CR3 确定。所以，与特定任务相关的虚拟地址空间到物理地址空间的映射由 LDTR 和 CR3 确定。显然，随着任务的切换，地址映射关系也要切换。

TSS 的地址映射寄存器区域由位于偏移 1CH 处的双字字段(CR3)和位于偏移 60H 处的字字段(LDTR)组成。在任务切换时，处理器自动从要执行任务的 TSS 中取出这两个字段，分别装入到寄存器 CR3 和 LDTR。这样就改变了虚拟地址空间到物理地址空间的映射。

但是，在任务切换时，处理器并不把换出任务但是的寄存器 CR3 和 LDTR 的内容保存到 TSS 中的地址映射寄存器区域。事实上，处理器也从来不向该区域自动写入。因此，如果程序改变了 LDTR 或 CR3，那么必须把新值人为地保存到 TSS 中的地址映射寄存器区域相应字段中。

4. 链接字段

链接字段安排在 TSS 内偏移 0 开始的双字中，其高 16 位未用。在起链接作用时，地 16 位保存前一任务的 TSS 描述符的选择子。

如果当前的任务由段间调用指令 CALL 或中断/异常而激活，那么链接字段保存被挂起任务的 TSS 的选择子，并且标志寄存器 EFLAGS 中的 NT 位被置 1，使链接字段有效。在返回时，由于 NT 标志位为 1，返回指令 RET 或中断返回指令 IRET 将使得控制沿链接字段所指恢复到链上的前一个任务。

5. 其它字段

为了实现输入/输出保护，要使用 I/O 许可位图。任务使用的 I/O 许可位图也存放在 TSS 中，作为 TSS 的扩展部分。在 TSS 内偏移 66H 处的字用于存放 I/O 许可位图在 TSS 内的偏移(从 TSS 开头开始计算)。关于 I/O 许可位图的作用，以后的文章中将会详细介绍。

初始进程的 TSS

任务状态段基本部分的格式	BIT31—BIT16	BIT15—BIT1	BIT0	OFFSET
	0000000000000000	链接字段（0）		0
	ESP0（PAGE_SIZE+&init_task）			4
	0000000000000000	SS0(0x10)		8
	ESP1(0)			0CH
	0000000000000000	SS1(0)		10H
	ESP2(0)			14H
	0000000000000000	SS2(0)		18H
	CR3(&pg_dir)			1CH
	EIP(0)			20H
	EFLAGS(0)			24H
	EAX(0)			28H
	ECX(0)			2CH
	EDX(0)			30H
	EBX(0)			34H
	ESP(0)			38H

	EBP(0)			3CH
	ESI(0)			40H
	EDI(0)			44H
	0000000000000000	ES(0x17)		48H
	0000000000000000	CS(0x17)		4CH
	0000000000000000	SS(0x17)		50H
	0000000000000000	DS(0x17)		54H
	0000000000000000	FS(0x17)		58H
	0000000000000000	GS(0x17)		5CH
	0000000000000000	LDTR(_LDT(0))		60H
	I/O 许可位图偏移	000000000000000	T	64H

几个常用的类型和全局变量

描述符类型

由于描述符通常都是 8 个字节，因此为它定义了 8 个字节的结构体，用这个结构体类型来表示描述符类型。

<include/linux/head.h>

typedef struct desc_struct {

unsigned long a,b;

} desc_table[256];

desc_table 也是一个类型，对应的是描述符表类型。因为 IDT 和 GDT 都有 256 项。

extern desc_table idt,gdt;

idt 和 gdt 对应着 setup.s 和 head.s 中的_idt 和_gdt。因此，在类型前加上了 extern 表示声明不是定义。

init_task

它是一个在 sched.c 中定义的全局变量，代表初始进程。

union task_union {

struct task_struct task;

char stack[PAGE_SIZE];

};

static union task_union init_task = {INIT_TASK,};

current

是一个指向当前工作进程的指针，在初始化的时候设置为指向初始进程。

<kernel/sched.h>

```
struct task_struct *current = &(init_task.task);
```

进程的初始化

在初始进程中设置了 TSS 和 LDT，但是计算机对这些结构的使用是通过相应的门描述符找到的，因此需要为初始进程设置有关的 TSS 和 LDT 的门描述符。同时，因为调用这个函数的时候计算机中只有一个初始进程，因此要清空任务表以及相应的门描述符。另外，这个函数还包括了对定时器的初始化。

<kernel/sched.c>

```
void sched_init(void)
{
    int i;
    struct desc_struct * p;

    // 判断信号的执行结构是否为 16 个字节，否则说明结构不正确
    if (sizeof(struct sigaction) != 16)
        panic("Struct sigaction MUST be 16 bytes");
    //为初始进程分配 TSS
    set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
    //为初始进程分配 LDT
    set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
    //清空进程表及相应的 TSS 和 LDT
    p = gdt+2+FIRST_TSS_ENTRY;
    for(i=1;i<NR_TASKS;i++) {
        task[i] = NULL;
        p->a=p->b=0;
        p++;
        p->a=p->b=0;
        p++;
    }
/* Clear NT, so that we won't have troubles with that later on */
    __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");
    //设置 TR 寄存器，使其指向的当前进程的 TSS 为初始进程的 TSS
    ltr(0);
    //设置 LDTR 寄存器，使其指向的当前进程的 LDT 为初始进程的 LDT
    lldt(0);
    // 一下部分是初始化定时器，将在定时器一章介绍
    outb_p(0x36,0x43);          /* binary, mode 3, LSB/MSB, ch 0 */
    outb_p(LATCH & 0xff , 0x40);    /* LSB */
    outb(LATCH >> 8 , 0x40);    /* MSB */
    set_intr_gate(0x20,&timer_interrupt);
    outb(inb_p(0x21)&~0x01,0x21);
```

```
    // 设置 0x80 为系统调用的入口
    set_system_gate(0x80,&system_call);
}
```

说明：

1. **struct desc_struct：**

这是一个占 8 个字节（64 位）的类型，用来表示一个描述符类型（64 位）。它是定义在/include/linux/head.h 当中。

```
typedef struct desc_struct {
    unsigned long a,b;
}
```

这里用这个类型定义了一个变量 p 用来指向 TSS 表和 LDT 表首，然后在对进程表进行初始化的时候，将相应的 TSS 和 LDT 所对应的内存单元清空。

2. **gdt:**

gdt 是定义在/include/linux/head.h 中的 desc_struct 类型（描述符类型），占用 8 个字节。因此对于类似 gdt+n 的操作实际上是对应的地址是 gdt+n×8 字节。

这里使用了 gdt+ FIRST_TSS_ENTRY 和 gdt+ FIRST_LDT_ENTRY 对应内存的地址实际是 gdt+4×8 字节和 gdt+5×8 字节。

3. **set_tss_desc(n,addr)和 set_ldt_desc(n,addr)：**

这是两个宏定义用于设置 TSS 和 LDT 的，定义在/include/asm/system.h 中：

```
// n - 是该描述符的指针；addr - 是描述符中的基地址值。任务状态段描述符的类型
// 是 0x89。
#define set_tss_desc(n,addr) _set_tssldt_desc(((char *) (n)),addr,"0x89")
// n - 是该描述符的指针；addr - 是描述符中的基地址值。局部表描述符的类型是 0x82。
#define set_ldt_desc(n,addr) _set_tssldt_desc(((char *) (n)),addr,"0x82")
```

这里使用的_set_tssldt_desc()也是宏定义：

```
// 在全局表中设置任务状态段/局部表描述符。
// 参数：n - 在全局表中描述符项 n 所对应的地址；addr - 状态段/局部表所在内存的基
// 地址。
// type - 描述符中的标志类型字节。
// %0 - eax(地址 addr)；%1 - (描述符项 n 的地址)；%2 - (描述符项 n 的地址偏移 2 处)；
// %3 - (描述符项 n 的地址偏移 4 处)；%4 - (描述符项 n 的地址偏移 5 处)；
// %5 - (描述符项 n 的地址偏移 6 处)；%6 - (描述符项 n 的地址偏移 7 处)；
#define _set_tssldt_desc(n,addr,type) \
__asm__ (
// 将 TSS 长度 104 字节放入描述符长度域(第 0-1 字节)
    "movw $104,%1\n\t" \
// 将基地址的低字放入描述符第 2-3 字节
"movw %%ax,%2\n\t" \
// 将基地址高字移入 ax 中
"rorl $16,%%eax\n\t" \
// 将基地址高字中低字节移入描述符第 4 字节
"movb %%al,%3\n\t" \
```

```
// 将标志类型字节移入描述符的第 5 字节
"movb $" type ",%4\n\t" \
// 描述符的第 6 字节置 0
"movb $0x00,%5\n\t" \
// 将基地址高字中高字节移入描述符第 7 字节。
"movb %%ah,%6\n\t" \
// eax 清零
"rorl $16,%%eax" \
::"a" (addr), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)), \
 "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7)) \
)
```

4. FIRST_TSS_ENTRY 和 FIRST_LDT_ENTRY：

定义在/include/linux/sched.h 中：

```
#define FIRST_TSS_ENTRY 4
#define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY+1)
```

这里涉及到了 TSS 和 LDT 描述符在 GDT 中的存储关系。GDT 的结构如图：

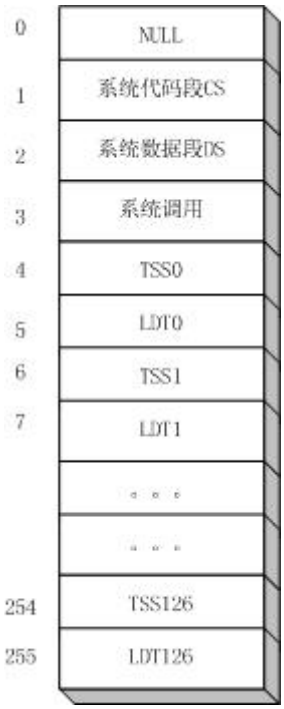


图 五.1GDT 表的结构图

5. ltr(n)和 lldt(n)：

定义在/include/linux/sched.h 中，用于加载第 n 号进程的 TSS 描述符和 LDT 描述符到 TR 和 LDR 寄存器中。

```
#define ltr(n) __asm__("ltr %%ax"::"a" (_TSS(n)))
#define lldt(n) __asm__("lldt %%ax"::"a" (_LDT(n)))
```

这里_TSS(n)和_LDT(n)表示进程 n 的 TSS 描述符和 LDT 描述符 GDT 中的偏移地址。定义如下：

```
#define _TSS(n) ((((unsigned long) n)<<4)+(FIRST_TSS_ENTRY<<3))
```

#define _LDT(n) ((((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3))

进程号同相应的 TSS 转换公式：FIRST_TSS_ENTRY×8 字节＋n×16 字节；

进程号同相应的 LDT 转换公式：FIRST_LDT_ENTRY×8 字节＋n×16 字节；

第六章进程的调度

从前几章知道了计算机是如何感知到一个进程的存在，通过这一章就可以知道操作系统是如何控制进程的工作的了。

进程的调度可以说是进程系统中最为核心的部分了，几乎进程中所有的操作都是以它为基础才得以实现的。使用了进程调度的函数包括了使进程休眠的函数 sleep_on、interruptible_sleep_on，时钟中断服务程序 do_timer，以及一些系统调用 sys_pause，sys_exit，sys_waitpid 等。

进程的调度方案有很多，Linux0.11 中仅仅使用一种方案－－最长剩余时间优先。就是说从就绪队列中选择一个剩余的时间片最多的进程优先被调度。

进程的调度可以划分为三个阶段：信号的处理、进程的选择、进程的切换。

信号的处理

在这一阶段通过判断那些休眠的进程是否接收到了没有被阻塞的信号，如果受到了就将该进程设置为就绪状态。在描述进程的时候提到了进程有个定时器，当时间到了就需要提醒进程“到点儿了！”。这个工作也放在了这个阶段来做。<kernel/sched.h>

```
/*
 *  'schedule()' is the scheduler function. This is GOOD CODE! There
 * probably won't be any reason to change this, as it should work well
 * in all circumstances (ie gives IO-bound processes good response etc).
 * The one thing you might take a look at is the signal-handler code here.
 *
 *    NOTE!!  Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
void schedule(void)
{
        int i,next,c;
        struct task_struct ** p;

/* check alarm, wake up any interruptible tasks that have got a signal */
// 判断每一个进程的定时器时间是否到了，到了就让这个进程准备工作
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                if (*p) {
// jiffies 是系统已经工作了的时间，时间到了的标准就是当前的系统的时间已经超过了
// 定时的时间。
                        if ((*p)->alarm && (*p)->alarm < jiffies) {
// 向信号请求寄存器发送请求信号，这方面的信息可以参照信号的处理一章
                                (*p)->signal |= (1<<(SIGALRM-1));
```

```
// 将 alarm 设置为 0 表示定时已经处理结束了。
                (*p)->alarm = 0;
            }
// _BLOCKABLE 定义在<include/kernel/sched.h>中，
// #define _BLOCKABLE (~(_S(SIGKILL) | _S(SIGSTOP)))
// 这个宏定义是用于从阻塞信号中清除掉对 SIGKILL 和 SIGSTOP 的阻塞，这两个信号
// 是不允许被屏蔽的。这一步判断一个处于可中断休眠状态的进程是否来了被屏蔽信
// 号意外的信号。
            if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
            (*p)->state==TASK_INTERRUPTIBLE)
                (*p)->state=TASK_RUNNING;
        }
```

进程的选择

各种调度算法讲的就是如何进行进程的选择，Linux0.11 使用的算法很容易理解，就通过检测各个处于就绪状态的进程的剩余时间以找到一个最大的，然后就将这个进程调出来准备进行进程的切换。如果发现所有的进程的时间片都已经使用完了，就利用进程的优先级重新设置时间片。然后，重新挑选。

```
/* this is the scheduler proper: */

    while (1) {
// 用于存储剩余时间中的最大值
        c = -1;
// 用于存放即将占用 CPU 的进程在进程表中的位置
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
// 忽略进程表中的空位
            if (!*--p)
                continue;
// 从进程的就绪队列中选择剩余的时间片最多的进程
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
// 如果每个进程的时间片都没有使用光，则最大的剩余时间片的值一定不为 0
// 如果值为 0 就说明所有进程的时间片都已经用光了。
        if (c) break;
// 利用进程的优先级重新为进程分配时间片，并不是只用就绪的进程的时间片会发生
// 改变，进程队列中所有的进程的时间片都会发生改变（包括了休眠的进程）。公式
// 为 counter=counter/2+prioriy。按照这中方法休眠进程的剩余时间片增多了，那么等到
// 它被唤醒的时候优先使用 CPU 的概率就增大了。
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
```

```
        if (*p)
            (*p)->counter = ((*p)->counter >> 1) +
                    (*p)->priority;
    }
```

进程的切换

进程切换是进程调度得以实现的关键技术，Linux0.11 是利用 Intel 提供的硬件方法实现这一功能的，但是高版本的 Linux 是使用软件的方法，软件方法使得程序的可扩展性增强。既然用到了硬件的知识，就先将一下预备知识吧。

保护模式下的段间转移

在保护模式下，段间转移的目标位置由选择子和偏移构成的地址表示，常把它称为目标地址指针。在 32 位代码段中，上述指针内的偏移使用 32 位表示，这样的指针也称为 48 位全指针。与实模式下相似，段间转移指令 JMP 和段间调用指令 CALL 还可分为段间直接转移和段间间接转移两类。如果指令 JMP 和 CALL 在指令中直接含有目标地址指针，那么就是段间直接转移；

在实模式下间接转移是将子程序的入口地址放到某一内存标号中，然后调用该内存标号实现段间转移。在保护模式下可以使用一个指向叫做任务门描述符的指针来实现跳转。通过这个任务门描述符可以找到一个 TSS 的地址，之后利用这个地址从 TSS 中得到子程序的入口地址。这就好像段描述符是用来得到段地址。正是因为它的功能同段描述符一样，因此任务门描述符与段描述符的格式是很像。任务门描述符属于系统段描述符，这里再介绍一下系统段描述符的格式。

系统段描述符

其格式如下：

<table>
<tr><td rowspan="2">系统段描述符</td><td>M+7</td><td>M+6</td><td>M+5</td><td>M+4</td><td>M+3</td><td>M+2</td><td>M+1</td><td>M+0</td></tr>
<tr><td>Base(31...24)</td><td colspan="2">Attributes</td><td colspan="3">Segment Base(23...0)</td><td colspan="2">Segment Limite(15...0)</td></tr>
<tr><td rowspan="3">系统段描述符的属性</td><td colspan="8">Byte m+6</td><td colspan="8">Byte m+5</td></tr>
<tr><td>BIT7</td><td>BIT6</td><td>BIT5</td><td>BIT4</td><td>BIT3</td><td>BIT2</td><td>BIT1</td><td>BIT0</td><td>BIT7</td><td>BIT6</td><td>BIT5</td><td>BIT4</td><td>BIT3</td><td>BIT2</td><td>BIT1</td><td>BIT0</td></tr>
<tr><td>G</td><td>X</td><td>0</td><td>AVL</td><td colspan="4">Limit(19...16)</td><td>P</td><td>DPL</td><td>DT0</td><td colspan="5">TYPE</td></tr>
</table>

与存储段描述符相比，它们很相似，区分的标志是属性字节中的描述符类型位 DT 的值。DT=1 表示存储段，DT=0 表示系统段。系统段描述符中的段基地址和段界限字段与存储段描述符中的意义完全相同；属性中的 G 位、AVL 位、P 位和 DPL 字段的作用也完全相同。存储段描述符属性中的 D 位在系统段描述符中不使用，现用符号 X 表示。系统段描述符的类型字段 TYPE 仍是 4 位，其编码及表示的类型列于下表，其含义与存储段描述符的类型却完全不同。

	类型编码	说　　明
系统段类型	0	未定义
	1	可用 286TSS
	2	LDT
	3	忙的 286TSS
	4	286 调用门
	5	任务门
	6	286 中断门
	7	286 陷阱门

	类型编码	说　　明
系统段类型	8	未定义
	9	可用 386TSS
	A	未定义
	B	忙的 386TSS
	C	386 调用门
	D	未定义
	E	386 中断门
	F	386 陷阱门

从上表可见，当类型域的值为 9 和 B 的时候这个描述符就表示任务门描述符。中断门描述符将再中断系统中介绍。

利用任务门实现段间跳转

段间转移指令 JMP 或段间调用指令 CALL 所含的指针是指向一个可用的任务门描述符时，就发生从当前任务到由该可用 TSS 对应任务(目标任务)的切换。目标任务的入口点由目标任务 TSS 内的 CS 和 EIP 字段所规定的指针确定。这样的 JMP 或 CALL 指令内的偏移被丢弃。

处理器采用与访问数据段相同的特权级规则控制对 TSS 段描述符的访问。TSS 段描述符的 DPL 规定了访问该描述符的最外层特权级，只有是相同级别或更高级别的程序才可以访问它。在进行任务切换的时候使用了一个叫 TR 的寄存器，它是用于存储当前正在运行的进程的 TSS 描述符。

根据指示目标任务 TSS 描述符的选择子进行任务切换的一般过程如下：

第一，测试目标任务状态段的界限。TSS 用于保存任务的各种状态信息，不同的任务，TSS 中可以有数量不等的其他信息，根据任务状态段的基本格式，TSS 的界限应大于或等于 103(104-1)。

第二，把寄存器现场保存到当前任务的 TSS。把通用寄存器、段寄存器、EIP 及 EFLAGS 的当前值保存到当前 TSS 中。保存的 EIP 值是返回地址，指向引起任务切换指令的下一条指令。但不把 LDTR 和 CR3 的内容保存到 TSS 中。

第三，把指示目标任务 TSS 的选择子装入 TR 寄存器中。同时把对应 TSS 的描述符装入 TR 的高速缓冲寄存器中。此后，当前任务改称为原任务，目标任务改称为当前任务。

第四，基本恢复当前任务(目标任务)的寄存器现场。根据保存在 TSS 中的内容，恢复各通用寄存器、段寄存器、EFLAGS 及 EIP。在装入寄存器的过程中，为了能正确地处理可能发生的异常，只把对应选择子装入各段寄存器。此时选择子的 P 位为 0。还装载 CR3 寄存器。

第五，进行链接处理。如果需要链接，那么将指向原任务 TSS 的选择子写入当前任务 TSS 的链接字段，把当前任务 TSS 描述符类型改为“忙”(并不修改原任务状态段描述符的“忙”位)，并将标志寄存器 EFLAGS 中的 NT 位置 1，表示是嵌套任务。如果需要解链，那么把原任务 TSS 描述符类型改为“可用”。如果无解链处理，那么将原任务 TSS 描述符类型置为“可用”，当前任务 TSS 描述符类型置为“忙”。由于 JMP 指令引起的任务切换不实施链接/解链处理；由 CALL 指令、中断、IRET 指令引起的任务切换要实施链接/解链处理。

第六，把 CR0 中的 TS 标志置为 1，这表示已发生过任务切换，在当前任务使用协处理

器指令时，产生自陷。由自陷处理程序完成有关协处理器现场的保存和恢复。这有利于快速地进行任务切换。

第七，把 TSS 中的 CS 选择子的 RPL 作为当前任务特权级设置为 CPL。又因为装入 CS 高速缓冲寄存器时要检测 CPL=代码段描述符的 DPL，所以 TSS 中的选择子所指示的代码段描述符的 DPL 必须等于该选择子的 RPL。任务切换可以在一个任务的任何特权级发生，并且可以切换到另一任务的任何特权级。

第八，装载 LDTR 寄存器。一个任务可以有自己的 LDT，也可以没有。当任务没有 LDT 时，TSS 中 LDT 选择子为 0。如果 TSS 中 LDT 选择子非空，则从 GDT 中读出对应的 LDT 描述符，在经过测试后，把所读的 LDT 描述符装入 LDTR 高速缓冲寄存器。如果 LDT 选择子为空，则将 LDT 的存在位置为 0，表明任务不使用 LDT。

第九，装载代码段寄存器 CS、堆栈段寄存器 SS 和各数据段寄存器及其高速缓冲寄存器。在装入代码段高速缓存之前，也要进行特权检查，处理器调整 TSS 中的 CS 选择子的 RPL=0，装入之后，调整 CS 的 RPL 等于目标代码段的 DPL。堆栈段使用的是 TSS 中的 SS 和 SP 字段的值，而不是使用内层栈保存区中的指针，即使发生了向内层特权级的变换。这与任务内的通过调用门的转移不同。

第十，把调试寄存器 DR7 中的局部启用位置为 0，以清除局部于原任务的各个断点和方式。

看到这是不是感到有些乱呢，其实利用 TSS 来实现的段间跳转的思想是很简单的。就是利用目标 TSS 找到目标地址，在转移的同时将自身的环境信息放到自身的 TSS 中。

如果还不理解，就先看看代码吧。这时请允许我引用一次一位伟大的算法大师的一句名言“假设大家把以上的内容都看了，并且假设大家把内容都看明白了。好了，让我们进行下面的任务吧。”☺

进程切换的代码

进程切换的思想就是设置 current 指向目标进程，保存当前进程的环境信息到当前进程对应的 TSS 中，同时设置 TR 寄存器指向目标进程对应的 TSS 结构，恢复目标进程的环境信息。

```
        switch_to(next);
}
```

这是一个宏定义，正是由于 Intel 的硬件支持，使得任务切换的代码量特别的少。这个宏定义位于 include/linux/sched.h。

```
/* switch_to(n)将切换当前任务到任务 n。首先检测任务 n 是不是当前任务，
* 如果是则什么也不做退出。如果我们切换到的任务最近（上次运行）使用过数学
* 协处理器的话，则还需复位控制寄存器 cr0 中的 TS 标志。
*/
// 输入：%0 - 新 TSS 的偏移地址(*&__tmp.a)； %1 - 存放新 TSS 的选择符值
// (*&__tmp.b)；
// dx - 新任务 n 的选择符；ecx - 新任务指针 task[n]。task[]是一个用于存放进程的数组。
// 其中临时数据结构__tmp 中，a 的值是 32 位偏移值，b 为新 TSS 的选择符。在任
// 务切换时，a 值没有用（忽略）。在判断新任务上次执行是否使用过协处理器时，是
// 通过将新任务状态段的地址与保存在 last_task_used_math 变量中的使用过协处理器
// 的任务状态段的地址进行比较而作出的。
```

```
#define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__(
//判断准备切换的进程是否就是当前的进程如果是就退出。_current 是一个全局变量存
//放的是当前的进程。
        "cmpl %%ecx,_current\n\t" \
        "je 1f\n\t" \
//将新任务的 TSS 描述符放入__tmp.b 中，在段间跳转的时候会用到
        "movw %%dx,%1\n\t" \
//将当前的进程与新进程交换，这步实现了 PCB 结构的交换
        "xchgl %%ecx,_current\n\t" \
//利用段间跳转实现任务的切换，因为是使用 TSS 描述符所以会忽略偏移地址
//所以__tmp.a 这四个字节是被忽略的，使用的是__tmp.b。因为这两个变量是连续存放
//的。我认为如果编译器不这么分配的话就会出问题，所以我对 Linux0.11 的可移植性
//有所怀疑
        "ljmp %0\n\t" \
        "cmpl %%ecx,_last_task_used_math\n\t" \
        "jne 1f\n\t" \
        "clts\n" \
        "1:"\
        ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
        "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

第七章进程的休眠和唤醒

进程的休眠指的就是操作系统中讲到的进程的阻塞，之所以用休眠这个词是因为与这个操作有关的函数 sleep_on 和 interruptible_sleep 都有 sleep 这个词。为什么要休眠在原理课上已经讲了很多了，我就不班门弄斧了。说一下这两个函数的区别：

有的时候一个进程需要等待的时间可能很长，例如等待键盘的输入。在这么长的等待时间内可能会有其它的事件发生来迫使这个进程工作，例如按了中止键。因此对于这类的休眠叫做可中断休眠，对应的状态是 TASK_INTERRUPTIBLE，表示在休眠的时候可以被中断。调用的函数是 intrruptible_sleep_on。

但是有的时候这个等待的时间很短，例如读取文件的时候。在这种情况的休眠是不允许被中断的，因此叫不可中断休眠，对应的状态是 TASK_UNINTERRUPTIBLE。调用的函数是 sleep_on。

另外一个进程总不至于永远的睡下去吧，因此还需要有一个唤醒的操作。对应的函数是 wake_up。

对于进程的休眠和唤醒操作，Linux0.11 做的不是很好准确的说里面有 Bug。从这一方面可以说明 Linux0.11 仅仅处于原理说明这一层次。

函数 sleep_on

如果当一个进程 1 在申请某一种资源的时候发现这个资源被锁定了，这个时候就需要等待一段时间，于是进程 1 休眠了。在进程 1 休眠的时候，如果又有一个进程 2 要求申请这个资源的时候，进程 2 也是要休眠的。于是进程 1 和进程 2 形成了一个链表。再来了进程 3、4 等的话也会连到链表的头部。从 Linux0.11 的代码中没有看到数据结构中的链表结构，它的实现是非常非常的巧妙的。第一次理解了它的实现的时候，我对程序有了新的认识。

为了便于理解，我会用一个例子按照这个算法走一遍。

函数中共牵涉到了三个任务指针：*p、tmp 和current，其中*p 是等待队列头指针，tmp 是临时指针；current 是当前任务指针。指针变化的示意图如下：

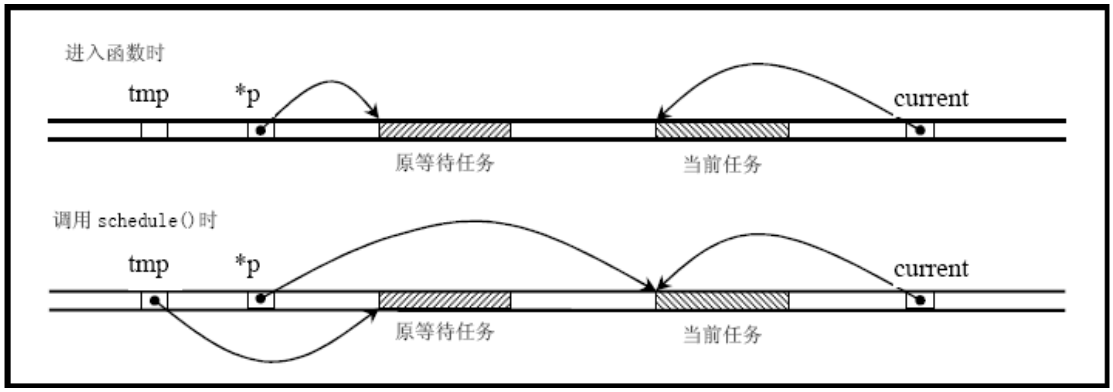
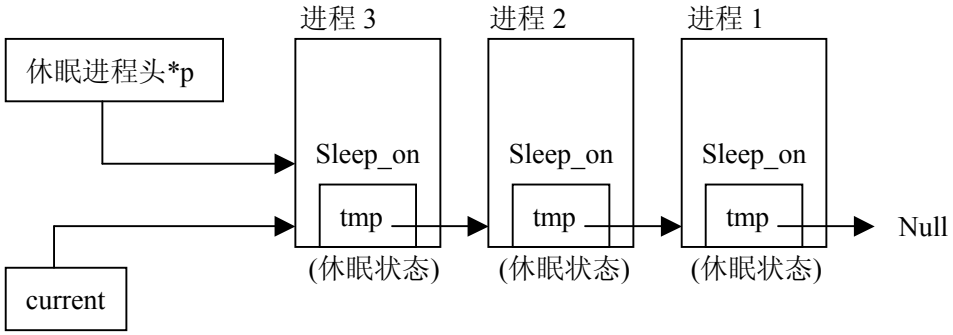


图 七.1sleep_on 中指针变化图（此图出自于《Linux0.11 完全注释》）

这里需要注意，*p和current是全局变量有所以进程共享，但是局部变量tmp却是由不同的进程所独有的，因为局部变量是利用进程的系统栈来存储的。我们把整个进程当作是一个庞大的

节点，把tmp当作是这个节点中的指向下一个进程指针(叫做next比较恰当)。按照这样的理解，假设有3个进程1，2，3依次进入休眠状态，可以形成这样的一个链表：



因为当进程 1 休眠的时候队列中还没有进程，因此进程 1 对应的局部变量 tmp 值为 NULL。

```
void sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;
// 确保进程头指针是有效的
    if (!p)
        return;
// 因为初始进程是不会去执行这个函数的因此如果发现是初始进程就会死机
    if (current == &(init_task.task))
        panic("task[0] trying to sleep");
// 让变量 tmp 指向前一个队列中的头进程，准备讲当前进程插在等待队列的头部
    tmp = *p;
// 让等待队列头指向当前进程
    *p = current;
// 让当前进程进入休眠状态，注意这里使用的是不可中断休眠状态
    current->state = TASK_UNINTERRUPTIBLE;
// 执行进程调度使得其他可用的进程占有 CPU
    schedule();
// 当进程从休眠状态中恢复的时候，就会立刻从这里开始执行了。首先判断等待队列
// 中是否还有其它的进程，如果还有的话就让下一个进程的状态是就绪状态。这样，
// 当再次执行调度程序的时候，那个进程就可能开始执行了。
    if (tmp)
        tmp->state=0;
}
```

为了能够完整的看一遍进程如何一各个的从等待队列中恢复，我需要再讲一下执行唤醒操作的函数 wake_up。

函数 wake_up

这个函数仅仅是让等待队列的头进程的状态变成就绪状态，这样在下一次进程调度的时候，这个进程就可能占有 CPU 了。

```
// *p 指向头进程
```

```
void wake_up(struct task_struct **p)
{
    if (p && *p) {
        (**p).state=0;
// 这里将*p 设为 NULL 后，就失去了对等待队列中的进程的跟踪。但是这些进程在通
// 常的情况都会执行完的。
        *p=NULL;
    }
}
```

进程从休眠到唤醒

这里就以图中的 3 个进程为例来说明这三个进程是如何进入休眠，然后又恢复的。

假设进程 1 执行的时候，等待队列是空的（1）。在申请某种资源的时候由于某种原因进程 1 需要休眠，于是调用了函数 sleep_on（2）。由于进程 2 也要求使用资源，导致进程 2 也需要休眠（3）。最后进程 3 也由于同样的原因进入了休眠（4）。

当资源可以使用的时候，调用了 wake_up 函数来恢复等待队列的头进程（进程 3），使得进程 3 为就绪态同时导致队列头部丢失（5）。

当进程调度开始的时候进程 3 开始执行，进程 3 发现 tmp 的值不空，并且指向了进程 2，于是将进程 2 的状态设置为 TASK_RUNNING。（6）当再次发生进程调度的时候，进程 2 开始执行，于是设置了进程 1 为就绪态。（7）

又一次进程调度之后，进程 1 运行。由于在进程 1 中对应的 tmp 值为 NULL，表明等待队列中的进程都已经运行了。（8）

Is 表示 interruptible_sleep_on，0 代表 TASK_RUNNING，1 代表 TASK_INTERRUPTIBLE

序号		等待队列头指向的进程 *p	进程	进程状态	这个进程对应的 tmp
1	开始的时候等待队列头为空	NULL			
2	当有一个进程 1 调用了 is 后	进程 1	1	1	NULL
3	当这个时候又有一个进程 2 调用了 is 后	进程 2	2	1	进程 1
4	最后进程 3 调用了 is	进程 3	3	1	进程 2
5	调用了了 wake_up 之后，进程 3 进入就绪态	NULL	3	0	进程 2
6	进程 3 开始执行，发现 tmp 不空。于是使得进程 2 进入就绪态	NULL	2	0	进程 1
	在一次进程调度之后				
7	进程 2 开始执行，设置进程 1 为就绪态	NULL	1	0	NULL
8	进程 1 执行时发现等待队列中已经没有进程了	NULL			

从这个流程中可以看出来，进程得唤醒原则时后进先服务（FILO）原则。

函数 interruptible_sleep_on

如果理解了使用 sleep_on 和 wake_up 让进程休眠和唤醒的话，理解 interruptible_sleep_on 就相对容易得多了。处理上的差别在于，在进程唤醒之后，进程调度之前如果又有进程进入休眠的话，interruptible_sleep_on 函数能够将这些进程也加到链表的头部。并且在进程调度的时候会先让这些进程首先执行。但是这个函数有 bug，我会提出我的改进方案。

```
void interruptible_sleep_on(struct task_struct **p)
{
        struct task_struct *tmp;

        if (!p)
                return;
        if (current == &(init_task.task))
                panic("task[0] trying to sleep");
        tmp=*p;
        *p=current;
// 上面的代码于 sleep_on 中的用法相同。
// 以下部分的设计就是为了处理后到的进程，并且让后来的进程先工作
repeat:    current->state = TASK_INTERRUPTIBLE;
        schedule();
// 当进程被唤醒的时候，先检查一下是否为空，以及是否为当前进程。如果不是就说
// 在被唤醒之后又有新的进程休眠了。这就需要让那些信到的进程工作，同时使自己
// 再次休眠
        if (*p && *p != current) {
                (**p).state=0;
                goto repeat;
        }
// 问题就出在这里，如果设置队列头为 NULL 的话，在处理完新到的进程之后就会丢
// 失原先的进程队列，因此需要将其改为*p＝tmp。同时需要设定一个新的唤醒函数
// interruptible_wake_up，于它相对应。
        *p=NULL;
        if (tmp)
                tmp->state=0;
}
```

刚才提到了要设置一个与它相对应的唤醒函数 interruptible_wake_up，它的实现如下：

```
void wake_up(struct task_struct **p)
{
        if (p && *p) {
                (**p).state=0;
        }
}
```

仅仅是将*p=NULL 一句去掉了。现在让我们看看新的搭配机制是如何工作的。

还是假设进程 1 执行的时候，等待队列是空的（1）。当进程 1 睡眠的时候（2），进程 2 也要求使用资源，导致进程 2 也需要休眠（3）。当资源可以使用调用了 interruptible_wake_up 函数之后,唤醒了进程 2（4）。在执行进程调度之前，又有了进程 3 休眠了（5）。接着又有进程 4 休眠（6）。当进程调度开始的时候进程 2 开始执行（7）。进程 2 发现头指针不是 NULL 而是指向进程 4，于是将进程 4 的状态设置为 TASK_RUNNING，然后循环并将自身设置为 TASK_UNINTERRUPTIBLE（8，9）。当在此发生进程调度的时候，进程 4 开始执行（10）。进程 4 执行的时候发现这个时候头指针指向了自身，于是就没有循环。按照修改后的代码 *p=tmp 于是头指针指向了进程 3（11）。进程 4 所对应的变量 tmp 指向了进程 3，因此进程 4 将进程 3 的状态设置为 0（12）。当又一次发生了进程调度之后，进程 3 开始运行（13）。与进程 4 一样，进程 3 发现了头指针指向自身。于是设置头指针为 tmp，由于进程 3 中的 tmp 指向了进程 2，因此头指针又指向了进程 2（14）。之后的执行顺序与 sleep_on 很相似了。

序号		等待队列头指向的进程*p	进程	进程状态	这个进程对应的 tmp
1	开始的时候等待队列头为空	NULL			
2	当有一个进程 1 调用了 is 后	进程 1	1	1	NULL
3	当这个时候又有一个进程 2 调用了 is 后	进程 2	2	1	进程 1
4	调用了了 wake_up_interruptible 之后	进程 2	2	0	进程 1
5	在进程 2 执行之前，进程 3 调用了 is	进程 3	3	1	进程 2
6	紧接着进程 4 又调用了 is	进程 4	4	1	进程 3
7	进程调度开始了，进程 2 被允许执行				
8	进程 2 发现等待队列头指针是 4	进程 4	4	0	进程 3
9		进程 4	2	1	进程 1
10	经过调度，进程 4 开始执行。				
11	由于队列头也是 4（指向了自身），就没有循环	进程 3	4	0	进程 3
12	由于进程 4 中的 tmp 是 3，执行 tmp->state=0	进程 3	3	0	进程 2
13	当再次发生调度的时候，进程 3 执行				
14	由于队列头也是 3（指向了自身），就没有循环	进程 2	3	0	进程 2
15	当再次发生调度的时候，进程 2 执行				
16	由于队列头也是 2（指向了自身），就没有循环	进程 2	2	0	进程 1
17	由于进程 2 中的 tmp 是 1，执行 tmp->state=0	进程 1	1	0	NULL
18	当再次发生调度的时候，进程 1 执行				
	进程 1 执行结束后，所有休眠的进程都执行了☺				

第八章中断系统

对于中断的处理是操作系统中一个很重要的环节，正是由于中断才使得操作系统能够与外部进行交流。可以说中断是操作系统的大门。

中断分为内部中断和外部中断两大类。外部中断称为“中断”，内部中断称为“异常”。通常在两条指令之间响应中断或异常。80386 最多处理 256 种中断或异常。

对 80386 而言，中断是由异步的外部事件引起的。外部事件及中断响应与正执行的指令没有关系。通常，中断用于指示 I/O 设备的一次操作已完成。

异常是 80386 在执行指令期间检测到不正常的或非法的条件所引起的。异常与正执行的指令有直接的联系。例如，执行除法指令时，除数等于 0。再如，执行指令时发现特权级不正确。当发生这些情况时，指令就不能成功完成。软中断指令“INT n”和“INTO”也归类于异常而不称为中断，这是因为执行这些指令产生异常事件。

80386 识别多种不同类别的异常，并赋予每一种类别以不同的中断向量号。异常发生后，处理器就象响应中断那样处理异常，即根据中断向量号，转相应的中断处理程序。（把这种中断处理程序称为异常处理程序可能更合适）

根据引起异常的程序是否可被恢复和恢复点不同，把异常进一步分类为故障(Fault)、陷阱(Trap)和中止(Abort)。我们把对应的异常处理程序分别称为故障处理程序、陷阱处理程序和中止处理程序。

故障是在引起异常的指令**之前**，把异常情况通知给系统的一种异常。80386 认为故障是可排除的。当控制转移到故障处理程序时，所保存的断点 CS 及 EIP 的值指向引起故障的指令。这样，在故障处理程序把故障排除后，执行 IRET 返回到引起故障的程序继续执行时，刚才引起故障的指令可重新得到执行。这种重新执行，不需要操作系统软件的额外参与。故障的发现可能在指令开始执行之前，也可能在指令执行期间。如果在指令执行期间检测到故障，那么中止故障指令，并把指令的操作数恢复为指令开始执行之前的值。这可保证故障指令的重新执行得到正确的结果。例如，在一条指令的执行期间，如果发现段不存在，那么停止该指令的执行，并通知系统产生段故障，对应的段故障处理程序可通过加载该段的方法来排除故障，之后，原指令就可成功执行，至少不再发生段不存在的故障。

陷阱是在引起异常的指令**之后**，把异常情况通知给系统的一种异常。当控制转移到异常处理程序时，所保存的断点 CS 及 EIP 的值指向引起陷阱的指令的下一条要执行的指令。下一条要执行的指令，不一定就是下一条指令。因此，陷阱处理程序并不是总能根据保存的断点，反推确定出产生异常的指令。在转入陷阱处理程序时，引起陷阱的指令应正常完成，它有可能改变了寄存器或存储单元。软中断指令、单步异常是陷阱的例子。

中止是在系统出现严重情况时，通知系统的一种异常。引起中止的指令是无法确定的。产生中止时，正执行的程序不能被恢复执行。系统接收中止后，处理程序要重新建立各种系统表格，并可能重新启动操作系统。硬件故障和系统表中出现非法值或不一致的值是中止的例子。

Linux 系统对异常的操作是不会去区分它是哪一种类型的，它们被一视同仁，仅仅是对应的操作不同。

中断向量表

在响应中断或者处理异常时，80386 根据中断向量号转到相应的处理程序。但是，在保护模式下，80386 不使用实模式下的中断向量表，而是使用中断描述符表 IDT。在保护模式下，80386 把中断向量号作为中断描述符表 IDT 中描述符的索引，而不再是中断向量表中的中断向量的索引。 象全局描述符表 GDT 一样，在整个系统中，中断描述符表 IDT 只有一个。中断描述符表寄存器 IDTR 指示 IDT 在内存中的位置。由于 80386 只识别 256 个中断向量号，所以 IDT 最大长度是 2K。 中断描述符表 IDT 所含的描述符只能是中断门、陷阱门和任务门。也就是说，在保护模式下，80386 只有通过中断门、陷阱门或任务门才能转移到对应的中断或异常处理程序。 将这些描述符用“门”来称呼是很形象的，因为它们已经不仅仅是中断的入口了，而是更复杂的结构，包括了权限的检测。就像一个有了锁头大门一样，是面向特定的使用者服务的。

中断向量表的空间是在程序中分配的,每个描述符占 8 个字节，为 256 个描述符预留了空间。<boot/head.s>

```
_idt:   .fill 256,8,0        # idt is uninitialized
```

数据结构

中断向量表的入口地址是依靠 IDTR 寄存器来找到的，它的格式与 GDTR 寄存器的格式相同：

	BIT47—BIT16	BIT15—BIT0
中断描述符表寄存器 IDTR	基地址	界限

在介绍 TSS 的时候已经讲解了任务门描述符，中断门与陷阱门描述符与任务门的格式是一样的，差别就在于类型域。为了便于理解，我再将其概括的描述一遍。

中断描述符占用 4 个字节，格式如下：

门描述符	M+7	M+6	M+5	M+4	M+3	M+2	M+1	M+0
	Offset(31...16)		Attributes		Selector		Offset(15...0)	

门描述符属性	Byte m+5								Byte m+4							
	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
	P	DPL		DT0	TYPE				000			Dword Count				

TYPE 是 4 位，其编码及表示的类型列于下表，其含义与存储段描述符的类型完全不同。

	类型编码	说　　明
系统段类型	0	未定义
	1	可用 286TSS
	2	LDT
	3	忙的 286TSS
	4	286 调用门
	5	任务门
	6	286 中断门
	7	286 陷阱门

	类型编码	说　　明
系统段类型	8	未定义
	9	可用 386TSS
	A	未定义
	B	忙的 386TSS
	C	386 调用门
	D	未定义
	E	386 中断门
	F	386 陷阱门

DPL 表示描述符特权级共 2 位，用于特权检查，以决定能否访问。

P 位称为存在位。P=1 表示描述符对地址转换是有效的，或者说该描述符所描述的段存在，即在内存中；P=0 表示描述符对地址转换无效，即该段不存在。使用该描述符进行内存访问时会引起异常。

Selector 用于从 GDT 中选择相应的段描述符，因此对于这里 selector 必须是代码段的选择子。

中断门与陷阱门的差别在于，当使用中断门的时候会将标志位 IF 设置为 0 来关闭中断。因此中断门用于处理 INTR 的中断。

而使用陷阱门却不会这么做，因此陷阱门用于处理异常。

初始化中断向量表

对于中断向量表的初始化在系统引导的时候就已经开始了。它将中断向量表中的每一个描述符都设定为指向一个默认的服务程序 ignore_int。<boot/head.s>

```
call setup_idt
```

setup_idt 也是用汇编写的，运行于实模式下。<boot/head.s>

在 setup_idt 中对各个寄存器的设定情况如下图：

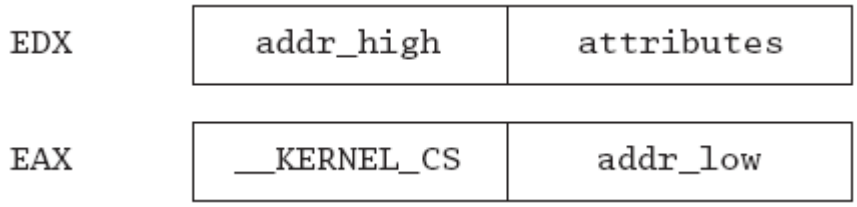


图 八.1 寄存器设定情况(此图出自于《The Linux Process Manager》)

```
/*
 *  setup_idt
 *
 *  sets up a idt with 256 entries pointing to
 *  ignore_int, interrupt gates. It then loads
 *  idt. Everything that wants to install itself
 *  in the idt-table may do so themselves. Interrupts
 *  are enabled elsewhere, when we can be relatively
 *  sure everything is ok. This routine will be over-
 *  written by the page tables.
 */
setup_idt:
// 将默认服务程序的偏移地址放入 edx 寄存器中
        lea ignore_int,%edx
// 由于代码段的选择子是 0x0008，将其保存在 eax 的高字节部分
        movl $0x00080000,%eax
// edx 中存放服务程序的 32 位偏移地址，因此 dx 中存放的是偏移地址的低 16 位
        movw %dx,%ax            /* selector = 0x0008 = cs */
// edx 的高字节部分存放的是偏移地址的高 16 位，这里是将属性信息放入 dx 中
        movw $0x8E00,%dx        /* interrupt gate - dpl=0, present */
```

```
// 将中断向量表的首地址放入 edi 中
        lea _idt,%edi
// 设定循环次数为 256，因为有 256 个描述符
        mov $256,%ecx
    rp_sidt:
// 将 eax 放入描述符的低 32 位，这时的 eax 中低字节存放的是偏移地址的低 16 位
// 高字节中存放的是段选择子 0x0008
        movl %eax,(%edi)
// 将 edx 放入描述符的高 32 位，edx 中的高字节存放偏移地址的高 16 位，低字节
// 中存放的是描述符的属性
        movl %edx,4(%edi)
// 将 edi 移到下一个要设置的描述符位处
        addl $8,%edi
        dec %ecx
        jne rp_sidt
// 加载 IDTR 寄存器使其指向中断向量表
        lidt idt_descr
        ret
```

默认服务程序 ignore_int

它也是一个汇编写的函数，但是在其实现的时候调用了 c 语言的函数 printk。用于在屏幕上输出出错信息。<boot/head.s>

```
/* This is the default interrupt "handler" :-) */
// 出错提示信息，当调用没有在系统中重新设定的中断的时候就会显示这个信息
int_msg:
    .asciz "Unknown interrupt\n\r"
.align 2
// 为了维护堆栈的格式因此在调用 printk 之前向栈中压入了也些寄存器，可以暂时不考
// 虑
ignore_int:
    pushl %eax
    pushl %ecx
    pushl %edx
    push %ds
    push %es
    push %fs
    movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    mov %ax,%fs
// 将提示信息压入栈中，做为参数传递给函数 printk
    pushl $int_msg
    call _printk
```

popl %eax
pop %fs
pop %es
pop %ds
popl %edx
popl %ecx
popl %eax
iret

中断向量表的设定

在引导阶段在中断向量表中设定了默认值使其指向默认服务程序，但是这个默认服务程序没有实际用处的。它仅仅用于提示应用程序调用了没有重新设定的中断向量。因此，在系统进行高一个层次的初始化的时候需要重新设定中断向量表。

对于前 32 个（0x00－0x19）中断向量是 Intel 保留的，专门用于处理 CPU 的异常。剩余的中断可以由操作系统自行设定，但是这种设定还是有一定的习惯的。

门描述符的设定

为了方便程序设计，Linux 提供了两层函数（宏定义）来完成门描述符的设定。底层是 _set_gate 用于完成对门描述符的设定。高层的定义是通过向_set_gate 传递不同的参数来完成的。

底层宏定义

它的设定方法与引导阶段对描述符的设定方法是一样的，只不过这里是将汇编语言嵌入到 c 中了，就需要遵守一些规则了。<include/asm/system.h>

参数：gate_addr－中断描述符的地址；type－描述符中类型域的值；dpl－描述特权值；%0－由 dpl 和 type 组成的类型标志字；%1－描述符的低 4 字节；%2－描述符的高 4 字节；%3－edx 服务程序的偏移地址；%4－eax 高字节中含有段选择符。

#define _set_gate(gate_addr,type,dpl,addr) \
__asm__ (

将服务程序偏移地址的低 4 字节放到了 eax 的低字节中，这时的 eax 的高字节中存放的是段选择子

"movw %%dx,%%ax\n\t" \

将类型标志字放到 edx 的低字节中，edx 的高字段存放的是服务程序的高 4 字节的偏移地址

"movw %0,%%dx\n\t" \

设置中断向量

"movl %%eax,%1\n\t" \
"movl %%edx,%2" \
: \

将类型标志字放到由编译器自行分配的一个寄存器中

: "i" ((short) (0x8000+(dpl<<13)+(type<<8))),

“o”表示内存地址加上偏移量

"o" (*((char *) (gate_addr))),

"o" (*(4+(char *) (gate_addr))),

将服务程序的 32 位入口地址放到 edx 中

"d" ((char *) (addr)),

将 0x00080000 放到 eax 中，高字节是 0008。这个值是段选择子，表示选择了 CS 类型的段。0008 的二进制格式是 0000 0000 0000 1000。前面已经提过了段寄存器的格式：前 13 位是用于从 GDT 中选择全局段描述符。在 GDT 中第 1 号描述符就是用于描述 CS 段的。

"a" (0x00080000))

高层宏定义

set_intr_gate()（中断门设置函数）、ste_trap_gate()（陷阱门设置函数）、set_system_gate()（系统门设置函数）用于完成对中断向量表不同类型的门描述符的设置。

set_trap_gate()设置的特权级是 0，而 set_system_gate()设置的特权级是 3。<include/asm/system.h>

```
// 设置中断门函数:
// 参数：n - 中断号；addr - 中断程序偏移地址。
// &idt[n]对应中断号在中断描述符表中的偏移值；中断描述符的类型是 14，特权级是 0。
#define set_intr_gate(n,addr) \
    _set_gate(&idt[n],14,0,addr)

// 设置陷阱门函数:
// 参数：n - 中断号；addr - 中断程序偏移地址。
// &idt[n]对应中断号在中断描述符表中的偏移值；中断描述符的类型是 15，特权级是 0。
#define set_trap_gate(n,addr) \
    _set_gate(&idt[n],15,0,addr)

// 设置系统调用门函数。
// 参数：n - 中断号；addr - 中断程序偏移地址。
// &idt[n]对应中断号在中断描述符表中的偏移值；中断描述符的类型是 15，特权级是 3。
#define set_system_gate(n,addr) \
    _set_gate(&idt[n],15,3,addr)
```

设置系统用的中断向量

系统中用到的 CPU 异常处理程序是在函数 trap_init 中设定的。<kernel/traps.c>

```
void trap_init(void)
{
    int i;
```

```
        set_trap_gate(0,&divide_error);
        set_trap_gate(1,&debug);
        set_trap_gate(2,&nmi);
        set_system_gate(3,&int3);    /* int3-5 can be called from all */
        set_system_gate(4,&overflow);
        set_system_gate(5,&bounds);
        set_trap_gate(6,&invalid_op);
        set_trap_gate(7,&device_not_available);
        set_trap_gate(8,&double_fault);
        set_trap_gate(9,&coprocessor_segment_overrun);
        set_trap_gate(10,&invalid_TSS);
        set_trap_gate(11,&segment_not_present);
        set_trap_gate(12,&stack_segment);
        set_trap_gate(13,&general_protection);
        set_trap_gate(14,&page_fault);
        set_trap_gate(15,&reserved);
        set_trap_gate(16,&coprocessor_error);
        for (i=17;i<48;i++)
            set_trap_gate(i,&reserved);
        set_trap_gate(45,&irq13);
    // 对 8259A 的处理
        outb_p(inb_p(0x21)&0xfb,0x21);
        outb(inb_p(0xA1)&0xdf,0xA1);
        set_trap_gate(39,&parallel_interrupt);
    }
```

在初始化的时候，其中前 17 个中断具体的含义见下表：

中断号	名称	类型	信号	说明
0	Devide error	故障	SIGFPE	当进行除以零的操作时产生。
1	Debug	陷阱 故障	SIGTRAP	当进行程序单步跟踪调试时，设置了标志寄存器 eflags 的 T 标志时产生这个中断。
2	nmi	硬件		由不可屏蔽中断 NMI 产生。
3	Breakpoint	陷阱	SIGTRAP	由断点指令 int3 产生，与 debug 处理相同。
4	Overflow	陷阱	SIGSEGV	eflags 的溢出标志 OF 引起。
5	Bounds check	故障	SIGSEGV	寻址到有效地址以外时引起。
6	Invalid Opcode	故障	SIGILL	CPU 执行时发现一个无效的指令操作码。
7	Device not available	故障	SIGSEGV	设备不存在，指协处理器。在两种情况下会产生该中断：(a)CPU 遇到一个转意指令并且 EM 置位时。在这种情况下处理程序应该模拟导致异常的指令。(b)MP 和 TS 都在置位状态时，CPU 遇到 WAIT 或一个转意指令。在这种情况下，处理程序在必要时应该更新协处理器的状态。
8	Double fault	故障	SIGSEGV	双故障出错。
9	Coprocessor segment overrun	故障	SIGFPE	协处理器段超出。
10	Invalid TSS	故障	SIGSEGV	CPU 切换时发觉 TSS 无效。
11	Segment not present	故障	SIGBUS	描述符所指的段不存在。
12	Stack segment	故障	SIGBUS	堆栈段不存在或寻址越出堆栈段。
13	General protection	故障	SIGSEGV	没有符合 80386 保护机制（特权级）的操作引起。
14	Page fault	故障	SIGSEGV	页不在内存。
15	Reserved			
16	Coprocessor error	故障	SIGFPE	协处理器发出的发出的出错信号引起。

图 八.2Intel 保留的中断向量（出自于《Linux0.11 源码分析》）

异常处理

现在开始中断系统中的实质的部分。异常的服务程序处理过程可以分为一下三个阶段：

1. **进入阶段：**由汇编写的主要是完成对栈的设定。
2. **处理阶段：**大部分是用 C 语言写的，完成对相应异常的处理。
3. **退出阶段：**由汇编写的，用于恢复堆栈，以及对信号的处理（在系统调用中）

对于不同异常服务程序在处理阶段差异可能是很大的，因此本章后面的部分关心的是第一和第三阶段。

在实模式中处理异常时会将 EFLAG、CS 和 EIP 分别压入栈中。但是在保护模式下可能有其它情况：如果异常带有出错码，在最后会将出错码（error_code）压栈；如果在发生栈变换最先压栈的是 SS 和 ESP。正是这个原因还可以将异常分为不带出错码异常、带出错码异常和系统调用。

进入阶段和退出阶段

在进入阶段主要是设定栈的格式，对应 3 种异常有三种函数来设定栈的格式。这三个函数是：no_error_code、error_code 和 system_call。退出阶段主要是将栈恢复到原来的状态，

对于系统调用的退出，它还有其它的操作。

不带出错码异常

不带出错码异常在进入第二阶段前栈中的情况如下：

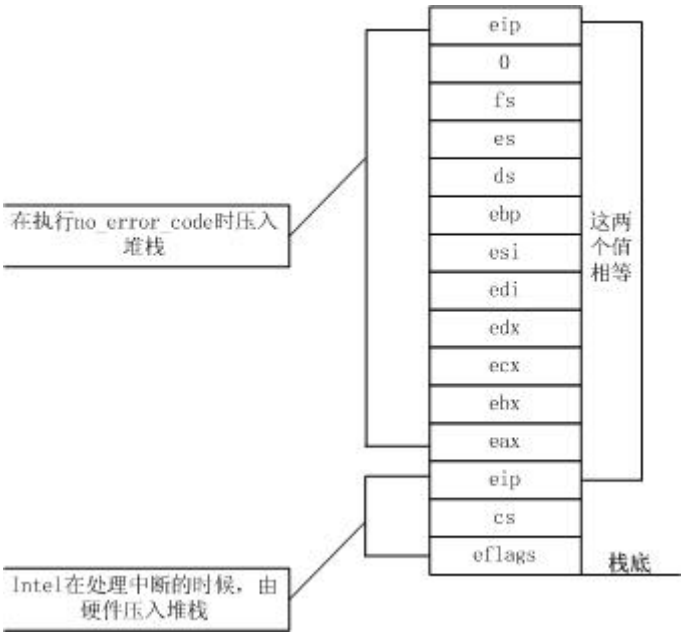


图 八.3 不带出错码异常的栈中情况

由于没有出错码，因此就在栈中压入了一个 0。对应的异常服务程序为：

<kernel/asm.s>

```
_divide_error:
        pushl $_do_divide_error
no_error_code:
// 在调用前已经将第二阶段要执行的程序的入口地址压栈了，这里是将这个入口地址
// 放入 eax 中，同时将 eax 原来的值保存到栈中
        xchgl %eax,(%esp)
        pushl %ebx
        pushl %ecx
        pushl %edx
        pushl %edi
        pushl %esi
        pushl %ebp
        push %ds
        push %es
        push %fs
        pushl $0        # "error code"
// 获取返回地址的 EIP 的值，放入 edx 中，准备作为第二阶段函数的参数
        lea 44(%esp),%edx
        pushl %edx
```

```
// 设定各个段寄存器执行系统数据段
        movl $0x10,%edx
        mov %dx,%ds
        mov %dx,%es
        mov %dx,%fs
// 调用第二阶段的处理程序
        call *%eax
// 还原各个寄存器的值
        addl $8,%esp
        pop %fs
        pop %es
        pop %ds
        popl %ebp
        popl %esi
        popl %edi
        popl %edx
        popl %ecx
        popl %ebx
        popl %eax
// 返回到栈中 EIP 指定的地址
        iret

_debug:
        pushl $_do_int3         # _do_debug
        jmp no_error_code

_nmi:
        pushl $_do_nmi
        jmp no_error_code

_int3:
        pushl $_do_int3
        jmp no_error_code

_overflow:
        pushl $_do_overflow
        jmp no_error_code

_bounds:
        pushl $_do_bounds
        jmp no_error_code

_invalid_op:
        pushl $_do_invalid_op
```

```
        jmp no_error_code

_coprocessor_segment_overrun:
        pushl $_do_coprocessor_segment_overrun
        jmp no_error_code

_reserved:
        pushl $_do_reserved
        jmp no_error_code
```

带出错码异常

带出错码异常在进入第二阶段前栈中的情况如下：

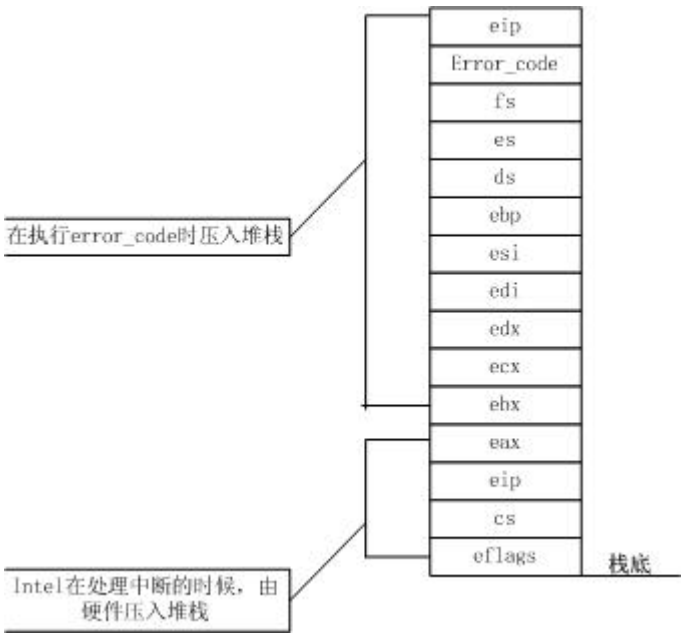


图 八.4 带出错码异常中的栈情况

将出错码放在栈的头部是用于作为参数，实现的方法与 no_error_code 的方法几乎完全一样。对应的异常服务程序为：

```
_double_fault:
        pushl $_do_double_fault
error_code:
        xchgl %eax,4(%esp)          # error code <-> %eax
        xchgl %ebx,(%esp)       # &function <-> %ebx
        pushl %ecx
        pushl %edx
        pushl %edi
        pushl %esi
        pushl %ebp
        push %ds
```

```
        push %es
        push %fs
        pushl %eax              # error code
        lea 44(%esp),%eax       # offset
        pushl %eax
        movl $0x10,%eax
        mov %ax,%ds
        mov %ax,%es
        mov %ax,%fs
        call *%ebx
        addl $8,%esp
        pop %fs
        pop %es
        pop %ds
        popl %ebp
        popl %esi
        popl %edi
        popl %edx
        popl %ecx
        popl %ebx
        popl %eax
        iret

_invalid_TSS:
        pushl $_do_invalid_TSS
        jmp error_code

_segment_not_present:
        pushl $_do_segment_not_present
        jmp error_code

_stack_segment:
        pushl $_do_stack_segment
        jmp error_code

_general_protection:
        pushl $_do_general_protection
        jmp error_code
```

系统调用

每个操作系统都提供很多种系统级的函数作为系统与外界的接口，外界对这些函数的调用就叫做系统调用。它们包括：外存文件与目录的读写，各种 I/O 设备的使用，在程序中启

动另一个程序，查询和统计系统资源使用情况等等。由于系统调用是对系统级函数的使用，为了能够保证这种功能能够正常使用，需要有一套安全机制来保证它的实现。

所有的系统调用都必须通过 int 0x80 来完成，任何其他的方法都无法使用这些系统函数。这个通道就好像是一个牢固的城堡的大门，所有的人只能通过这扇门进入城堡。这样保护城堡的工作就简单了，那就守好这扇门。

对于不同系统调用的具体实现是由相应的功能号决定的，这个功能号要在系统调用之前放入 eax 寄存器中。有的系统调用需要参数，这些参数需要使用寄存器来传递。这是因为系统调用是为了能够让用户的程序调用系统的函数，因此在产生异常的时候一定会发生权限的变化。这样就会发生栈变换。

栈变换是指将原来执行用户栈的 esp 和 ss 变换到执行系统栈，为了保存原来 esp 和 ss 中的信息，需要将原来的值保存到系统栈中。

与其它的异常一样，在系统调用的时候会保存寄存器的值。在执行系统调用的功能函数之前，向栈中压入的值如下图：

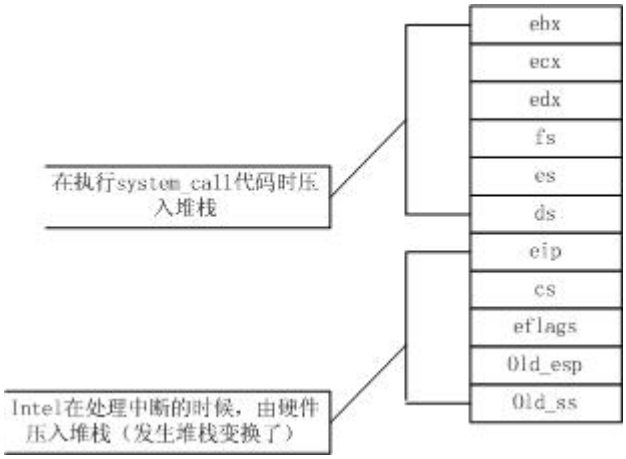


图 八.5 系统调用时栈中的情况

为了能够比较方便的使用栈中的这些值，Linux 通过宏定义了各个寄存器值在栈中的偏移地址。注意，对堆栈的使用是发生在返回阶段。而在调用功能函数之后会将 eax 中的返回值压入栈中。<kernel/system_call.s>

//定义各个寄存器在堆栈中的偏移量

EAX = 0x00
EBX = 0x04
ECX = 0x08
EDX = 0x0C
FS = 0x10
ES = 0x14
DS = 0x18
EIP = 0x1C
CS = 0x20
EFLAGS = 0x24
OLDESP = 0x28
OLDSS = 0x2C

另外，在中断返回阶段会对信号进行处理。这就需要对 PCB 进行处理，为了方便的访

问 tast_struct 结构中的一些变量，也通过宏定义了这些变量的偏移量：

```
state = 0         # these are offsets into the task-struct.
counter   = 4
priority = 8
signal     = 12
sigaction = 16       # MUST be 16 (=len of sigaction)
blocked = (33*16)

# offsets within sigaction
sa_handler = 0
sa_mask = 4
sa_flags = 8
sa_restorer = 12
```

在返回阶段，进行判断是否需要执行进程调度对信号进行处理以及恢复现场。

```
SIG_CHLD    = 17

//系统调用的总数 72 个
nr_system_calls = 72

/*
 * Ok, I get parallel printer interrupts while using the floppy for some
 * strange reason. Urgel. Now I just ignore them.
 */
.globl _system_call,_sys_fork,_timer_interrupt,_sys_execve
.globl _hd_interrupt,_floppy_interrupt,_parallel_interrupt
.globl _device_not_available, _coprocessor_error

.align 2
//当进行系统调用出错的时候就会返回－1，EAX 寄存器用于保存这个返回值
bad_sys_call:
        movl $-1,%eax
        iret
.align 2
//系统函数执行完了之后，需要执行调度程序，当调度程序执行完后，需要回到
//ret_from_sys_call 处继续执行，这里是通过对栈操作来模拟 call 操作实现这个功能的
reschedule:
        pushl $ret_from_sys_call
        jmp _schedule
.align 2
//系统调用的时候究竟是执行哪一个函数就是由这段代码实现的，它是操作系统的看门
//人。它的执行过程是先进行错误检查以保证能够正确的调用的系统函数；然后保存系
//统环境信息；通过功能号在函数指针表中将相应的系统函数找到来执行；接着就是关
//于进程调度的操作了。
```

```
_system_call:
//如果功能号超过了 71 就会出错，因为一共就 72 个系统调用函数。
        cmpl $nr_system_calls-1,%eax
        ja bad_sys_call
//开始保护现场的操作，在此之前在系统堆栈中已经保存了 EIP  CS、EFLAGS、原来
//的 ESP、原来的 SS 寄存器的值。他们是由硬件压入堆栈的。
        push %ds
        push %es
        push %fs
        pushl %edx
        pushl %ecx          # push %ebx,%ecx,%edx as parameters
        pushl %ebx          # to the system call
//让 DS,ES 指向内核数据段(全局描述符表中数据段描述符)
        movl $0x10,%edx         # set up ds,es to kernel space
        mov %dx,%ds
        mov %dx,%es
//FS 指向局部数据段(局部描述符表中数据段描述符)
        movl $0x17,%edx         # fs points to local data space
        mov %dx,%fs
//下面是通过功能号在系统函数表中找到相应的系统函数。调用地址 = _sys_call_table +
//%eax ×4。表中的每一项都占用 4 个字节因为是函数指针。这里_sys_call_table 对应
//的是 C 程序中的 sys_call_table （在 include/linux/sys.h 中），它是一个包括 72 个系
//统调用 C 处理函//数的地址数组表。
        call _sys_call_table(,%eax,4)
// 执行完第二阶段的返回值会放入 eax 寄存器中，由于后面的操作会改变 eax 寄存器
// 因此需要将其保存。
        pushl %eax
// 开始对是否执行进程调度进行判断。
// 原则是如果当前进程的状态不是就绪态，不执行进程调度；
//         如果当前进程的还剩余时间片，不执行进程调度。
        movl _current,%eax
        cmpl $0,state(%eax)             # state
        jne reschedule
        cmpl $0,counter(%eax)       # counter
        je reschedule
ret_from_sys_call:
// 无论执行了进程调度与否，都会从这里执行返回阶段
// 这里主要涉及到了对信号的处理，具体情况将在信号处理一章讲解
        movl _current,%eax              # task[0] cannot have signals
        cmpl _task,%eax
        je 3f
        cmpw $0x0f,CS(%esp)         # was old code segment supervisor ?
        jne 3f
        cmpw $0x17,OLDSS(%esp)          # was stack segment = 0x17 ?
```

```
        jne 3f
        movl signal(%eax),%ebx
        movl blocked(%eax),%ecx
        notl %ecx
        andl %ebx,%ecx
        bsfl %ecx,%ecx
        je 3f
        btrl %ecx,%ebx
        movl %ebx,signal(%eax)
        incl %ecx
        pushl %ecx
        call _do_signal
        popl %eax
    // 恢复寄存器
3:      popl %eax
        popl %ebx
        popl %ecx
        popl %edx
        pop %fs
        pop %es
        pop %ds
        iret
```

从代码中可以看到，系统调用中相应的功能是通过查找一个函数指针的数组找到的。这个表的初始化是在 include/linux/sys.h 文件中：

```
extern int sys_setup();
extern int sys_exit();
extern int sys_fork();
extern int sys_read();
extern int sys_write();
extern int sys_open();
extern int sys_close();
extern int sys_waitpid();
extern int sys_creat();
extern int sys_link();
extern int sys_unlink();
extern int sys_execve();
extern int sys_chdir();
extern int sys_time();
extern int sys_mknod();
extern int sys_chmod();
extern int sys_chown();
extern int sys_break();
extern int sys_stat();
extern int sys_lseek();
```

```
extern int sys_getpid();
extern int sys_mount();
extern int sys_umount();
extern int sys_setuid();
extern int sys_getuid();
extern int sys_stime();
extern int sys_ptrace();
extern int sys_alarm();
extern int sys_fstat();
extern int sys_pause();
extern int sys_utime();
extern int sys_stty();
extern int sys_gtty();
extern int sys_access();
extern int sys_nice();
extern int sys_ftime();
extern int sys_sync();
extern int sys_kill();
extern int sys_rename();
extern int sys_mkdir();
extern int sys_rmdir();
extern int sys_dup();
extern int sys_pipe();
extern int sys_times();
extern int sys_prof();
extern int sys_brk();
extern int sys_setgid();
extern int sys_getgid();
extern int sys_signal();
extern int sys_geteuid();
extern int sys_getegid();
extern int sys_acct();
extern int sys_phys();
extern int sys_lock();
extern int sys_ioctl();
extern int sys_fcntl();
extern int sys_mpx();
extern int sys_setpgid();
extern int sys_ulimit();
extern int sys_uname();
extern int sys_umask();
extern int sys_chroot();
extern int sys_ustat();
extern int sys_dup2();
```

```
extern int sys_getppid();
extern int sys_getpgrp();
extern int sys_setsid();
extern int sys_sigaction();
extern int sys_sgetmask();
extern int sys_ssetmask();
extern int sys_setreuid();
extern int sys_setregid();

fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
sys_setreuid,sys_setregid };
```

这些函数的实现遍布到了整个操作系统中的各个部分。

如何使用系统调用

系统调用是通过不同的功能号来指示它相应的执行函数的，因此要想使用系统调用只需设置 eax 为 2，然后执行 int 0x80 就可以了。如果有参数，只需将参数按顺序放入 ebx、ecx 和 edx 寄存器中就可以了（目前最多支持 3 个参数）。

在 Linux 下，为了使得在 C 语言中实现系统调用变得方便，设置了一个宏定义 _syscall?(int, 系统函数)。在使用的使用的时候在调用前将这个宏定义加在文件的前端就可以了。以对 fork()函数调用为例，先在使用前加上

```
static inline _syscall0(int,fork)
```

然后，在程序中就可以使用 fork()这个函数了。是不是挺神的☺。

这是如何实现的呢？

在/include/unistd.h 文件中我们可以看到这样的代码：

```
#ifdef __LIBRARY__

#define __NR_setup      0     /* used only by init, to get system going */
#define __NR_exit 1
#define __NR_fork2
#define __NR_read       3
#define __NR_write      4
#define __NR_open       5
```

```
#define __NR_close      6
#define __NR_waitpid    7
#define __NR_creat      8
#define __NR_link 9
#define __NR_unlink     10
#define __NR_execve     11
#define __NR_chdir      12
#define __NR_time13
#define __NR_mknod      14
#define __NR_chmod      15
#define __NR_chown      16
#define __NR_break      17
#define __NR_stat 18
#define __NR_lseek      19
#define __NR_getpid     20
#define __NR_mount      21
#define __NR_umount     22
#define __NR_setuid     23
#define __NR_getuid     24
#define __NR_stime      25
#define __NR_ptrace     26
#define __NR_alarm      27
#define __NR_fstat28
#define __NR_pause      29
#define __NR_utime      30
#define __NR_stty 31
#define __NR_gtty 32
#define __NR_access     33
#define __NR_nice34
#define __NR_ftime      35
#define __NR_sync       36
#define __NR_kill 37
#define __NR_rename     38
#define __NR_mkdir      39
#define __NR_rmdir      40
#define __NR_dup 41
#define __NR_pipe42
#define __NR_times      43
#define __NR_prof44
#define __NR_brk 45
#define __NR_setgid     46
#define __NR_getgid     47
#define __NR_signal     48
#define __NR_geteuid    49
```

```
#define __NR_getegid   50
#define __NR_acct51
#define __NR_phys      52
#define __NR_lock53
#define __NR_ioctl54
#define __NR_fcntl     55
#define __NR_mpx56
#define __NR_setpgid   57
#define __NR_ulimit    58
#define __NR_uname     59
#define __NR_umask     60
#define __NR_chroot    61
#define __NR_ustat     62
#define __NR_dup2      63
#define __NR_getppid   64
#define __NR_getpgrp   65
#define __NR_setsid    66
#define __NR_sigaction67
#define __NR_sgetmask68
#define __NR_ssetmask69
#define __NR_setreuid  70
#define __NR_setregid  71

#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
      : "=a" (__res) \
      : "0" (__NR_##name)); \
if (__res >= 0) \
      return (type) __res; \
errno = -__res; \
return -1; \
}

#define _syscall1(type,name,atype,a) \
type name(atype a) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
      : "=a" (__res) \
      : "0" (__NR_##name),"b" ((long)(a))); \
if (__res >= 0) \
```

```
        return (type) __res; \
    errno = -__res; \
    return -1; \
    }

    #define _syscall2(type,name,atype,a,btype,b) \
    type name(atype a,btype b) \
    { \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(a)),"c" ((long)(b))); \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
    }

    #define _syscall3(type,name,atype,a,btype,b,ctype,c) \
    type name(atype a,btype b,ctype c) \
    { \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(a)),"c" ((long)(b)),"d" ((long)(c))); \
    if (__res>=0) \
        return (type) __res; \
    errno=-__res; \
    return -1; \
    }

    #endif /* __LIBRARY__ */
```

还是以对 fork()函数的调用为例，当在调用前加上 static inline _syscall0(int,fork)之后。在编译前，编译器会对这个宏定义进行展开，得到：

```
static inline int fork(void)
{
long __res;
__asm__ volatile (“int $0x80”
                  :     “=a” (__res)
                  :     “0”  (__NR_fork));
if (__res>=0)
            return (int) __res;
errno = -__res;
return –1;
```

}

^_^原来还是使用汇编语言进行 0x80 中断，只是封装的比较好而已。在这里__NR_fork 对应的是 2，就是说 fork()对应的功能号是 2。将其值放到了 EAX 寄存器中，注意在系统调用的时候一定使用寄存器进行参数传递，因为在系统调用的时候会发生堆栈转换。

从这里可以看到，在 Linux0.11 版本中提供的函数中最多就是 3 个。相应的宏定义分别是_syscall0(type,name)、_syscall1(type,name,atype,a)、_syscall2(type,name,atype,a,btype,b)、_syscall3(type,name,atype,a,btype,b,ctype,c)。

第九章定时器

定时器的初始化

在讲定时器的初始化之前，我想再说一下在 Linux 启动的时候已经将 ICW2 设置为 0x20，因此这个时候 IR0 对应的中断向量码是 0x20，就是说定时器的中断向量码是 0x20。

定时器的初始化工作是在 sched_init()(/kernel/sched.c)中完成的。代码如下：

```
// 设置 8253 的控制字使其处于 3 号工作模式，先使用 LSB 后使用 MSB，编写通道为 0
// 8253 控制字寄存器的地址是 0x43
outb_p(0x36,0x43);              /* binary, mode 3, LSB/MSB, ch 0 */
// 设置定时器的中断间隔为 10ms（100HZ）
outb_p(LATCH & 0xff , 0x40);    /* LSB */
outb(LATCH >> 8 , 0x40);    /* MSB */
// 设置定时器的中断响应程序（中断向量）
set_intr_gate(0x20,&timer_interrupt);
// 打开对定时器的屏蔽，OCW1 的 M0 位清零（让定时器开始工作）
outb(inb_p(0x21)&~0x01,0x21);
```

说明：

1. **定时器锁存器：**

   定时器有 3 个锁存器，他们各有其则。锁存器 0 用于维护系统时钟地址为 0x40；锁存器 1 用于周期性的向 DMA 发送数据信号，供存储器刷新用，地址为 0x41；锁存器 2 用于扬声器发出声音，地址为 0x42。因此这里是向 0x40（锁存器 0）设定值。

2. **时间间隔的设定：**

   在 3 号工作模式下当锁存器的值在时钟脉冲的驱动下减至 0 的时候会发出一个信号产生中断，因此要想设定定时器的中断时间间隔，只需设定锁存器的值。定时器的时钟脉冲频率是 1193180Hz。对于 10ms 的周期对应的频率是 100Hz，因此将锁存器的值设为 1193180/100 就可以了。Linux0.11 是这么做的：

   在/include/kernel/sched.h 文件：

   #define HZ 100

   在/kernel/sched.c 文件：

   #define LATCH (1193180/HZ)

   然后将 LATCH 送入到锁存器中。

响应定时中断

定时中断的处理程序是 timer_interrupt，它位于/kernel/sytem_call.s 中：

```
.align 2
```

```
_timer_interrupt:
        push %ds        # save ds,es and put kernel data space
        push %es        # into them. %fs is used by _system_call
        push %fs
        pushl %edx          # we save %eax,%ecx,%edx as gcc doesn't
        pushl %ecx          # save those across function calls. %ebx
        pushl %ebx          # is saved as we use that in ret_sys_call
        pushl %eax
        // 让 DS，ES 指向系统的数据段
        movl $0x10,%eax
        mov %ax,%ds
        mov %ax,%es
        // 让 FS 指向用户的数据段
        movl $0x17,%eax
        mov %ax,%fs
        // 增加滴答数的值
        incl _jiffies
        // 由于 Linux 在进行系统初始化的时候将系统设置为非 EOI 方式，为了能够复位
        //  ISR 位，需要手动设置。
        movb $0x20,%al          # EOI to interrupt controller #1
        outb %al,$0x20
        // 从系统栈中得到当前程序的代码段寄存器的值，CS 寄存器的值是在中断发生的
        // 时候由硬件压入栈中。从这个值中分离出用户的权限值（CPL）作为函数
        //  do_timer()的参数
        movl CS(%esp),%eax
        andl $3,%eax        # %eax is CPL (0 or 3, 0=supervisor)
        pushl %eax
        call _do_timer      # 'do_timer(long CPL)' does everything from
        addl $4,%esp        # task switching to accounting ...
        jmp ret_from_sys_call
```

关于 EOI 的说明：

当中断发生后需要复位 ISR 位，如果是手动复位的话就必须设置命令字 OCW2（地址为 0x20）。对于时钟中断可以使用 00 1 00 000(二进制)，进行复位。

函数 do_timer:

函数 do_timer 是执行定时中断的核心部分，这个函数是实现分时功能的基础。从这个函数中可以看出来，Linux0.11 是非抢占式的，对于已经占有 CPU 的进程，除非发生中断否则在时间片用尽之前式不会让出 CPU 的。它位于/kernel/sched.h 中：

```
void do_timer(long cpl)
{
```

```
// 处理同扬声器相关的操作
    extern int beepcount;
    extern void sysbeepstop(void);

    if (beepcount)
        if (!--beepcount)
            sysbeepstop();
// 根据程序的权限值来判断目前是在用户态还是系统态
    if (cpl)
        current->utime++;
    else
        current->stime++;
// 处理用户的定时器
    if (next_timer) {
        next_timer->jiffies--;
        while (next_timer && next_timer->jiffies <= 0) {
            void (*fn)(void);

            fn = next_timer->fn;
            next_timer->fn = NULL;
            next_timer = next_timer->next;
            (fn)();
        }
    }
// 此操作同软驱有关
    if (current_DOR & 0xf0)
        do_floppy_timer();
// 如果发现用户的时间片没有用尽，就让它继续运行。否则，就要进行
// 进程调度了
    if ((--current->counter)>0) return;
    current->counter=0;
// 对于系统级的程序是不会进行系统调度的，这就是原语
    if (!cpl) return;
// 进程调度开始了
    schedule();
}
```

第十章信号处理

信号可以说是由操作系统提供的软中断，是用来在进程之间建立通讯用的。这种通讯机制是异步的，就是说一个进程在执行的时候不知道什么时候会收到信号。这就同中断非常的相像了。在中断处理中需要有中断向量表，有中断号。通过中断号可以从中断向量表中找到相应的服务程序的入口。对于信号也是一样的，不同的信号对应不同的信号类型，而且对应不同的信号有相应的处理函数。中断中有屏蔽位，信号中也有。在讲解如何使用信号之前，先看看在 Linux0.11 中信号是如何定义的。

信号的描述

信号是用于在进程间进行通讯用的，因此它应该放到进程中作为进程的一个属性，就好像是进程的耳朵。在</include/kernel/sched.h>

```
struct task_struct {
        ……
// 它是一个 32 位的位图，每一位对应着一种信号类型。就好像是中断请求寄存器一样。
// 它的第 n 位对应信号 n
        long signal;
// 信号的执行数组，好像是中断向量表，不同的信号对应这里不同的执行结构。以后
// 就称它是信号向量表吧
        struct sigaction sigaction[32];
// 信号的屏蔽位图，就像中断屏蔽寄存器
        long blocked; /* bitmap of masked signals */
        ……
};
```

在中断向量表中存放的是中断向量描述符，这里存放的是与信号处理有关的信息。它的定义放在/include/kernel/signal.h 中：

```
// 定义了一个新类型－－信号集
// 它其实是一个 32 位的位图，。在以后的新版本中，位数增多，定义的方式也变化了。
typedef unsigned int sigset_t;      /* 32 bits */
……
struct sigaction {
// 信号处理程序指针（函数指针），相当于中断服务程序
        void (*sa_handler)(int);
// 当一个信号处理工作时，可能需要屏蔽新的信号。这些需要屏蔽的信号信息存放在
// 这里。实际上它通常仅仅将自己屏蔽了，因为这个信号自身到来时，进程对这个信
// 号是开放的。但是当这个信号处理程序运行的时候，如果同一个信号再次来了怎么
// 办呢？通过设置这个值，来决定是屏蔽还是进行嵌套。
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_restorer)(void);
```

};

信号类型

在 Linux0.11 中设置了 22 种信号，其定义位于/include/kernel/signal.h 中：

```
#define SIGHUP      1
#define SIGINT      2
#define SIGQUIT     3
#define SIGILL      4
#define SIGTRAP     5
#define SIGABRT     6
#define SIGIOT      6
#define SIGUNUSED   7
#define SIGFPE      8
#define SIGKILL     9
#define SIGUSR1     10
#define SIGSEGV     11
#define SIGUSR2     12
#define SIGPIPE     13
#define SIGALRM     14
#define SIGTERM     15
#define SIGSTKFLT   16
#define SIGCHLD     17
#define SIGCONT     18
#define SIGSTOP     19
#define SIGTSTP     20
#define SIGTTIN     21
#define SIGTTOU     22
```

信号的设置

与中断一样，为了能够响应中断需要在中断向量表中设置中断服务程序的入口地址。信号在使用之前需要设置与信号对应的处理程序。为了这个操作 Linux 系统中提供了两个系统调用。将这两个系统调用之前，我还是先谈谈初始化问题。

信号的初始化

信号是由进程来维护的，所有的进程都是通过复制父进程来产生的。最原始的进程就是初始进程了 INIT_TASK，在 INIT_TASK 中有这么一段代码：</include/kernel/sched.h>

/ signals */ 0,{{},},0, *

表示 signal=0 sigaction={{},} blocked=0。说明开始的时候没有信号来，也不屏蔽任何信号。对于每一个信号处理函数的句柄都是 0。系统定义了两个句柄：<include/signal.h>

```
#define SIG_DFL     ((void (*)(int))0)    /* default signal handling */
#define SIG_IGN     ((void (*)(int))1)    /* ignore signal */
```

这就说明如果设置句柄为 0，这个信号就执行默认的操作。如果句柄为 1，就会忽略这个信号的操作。

对于以后调用 fork 的时候，会继承父进程信号向量表和屏蔽位。但是设置子进程的 signal 为 0 表示还没有接受到任何信号。这样在一个进程没有设置自己的信号服务程序的时候，如果来了信号系统会使用默认的操作。

系统调用 sys_signal

设置信号服务程序的句柄，当服务程序执行之后会将信号的句柄恢复为默认值 0。<kernel/signal.c>

```
// 参数：signum 是信号类型；handler 是函数句柄（其实就是将函数指针转换为整型）
int sys_signal(int signum, long handler, long restorer)
{
    struct sigaction tmp;
// 判断是否为有效的类型并且不能是 SIGKILL
    if (signum<1 || signum>32 || signum==SIGKILL)
        return -1;
// 设置句柄
    tmp.sa_handler = (void (*)(int)) handler;
// 在信号服务程序处理中不再次屏蔽任何信号（包括自己）
    tmp.sa_mask = 0;
// 设置信号处理方式标志为信号仅仅使用一次，不屏蔽信号本身
    tmp.sa_flags = SA_ONESHOT | SA_NOMASK;
    tmp.sa_restorer = (void (*)(void)) restorer;
// 设置返回值为原信号处理程序的句柄
    handler = (long) current->sigaction[signum-1].sa_handler;
// 将新的信号处理结构保存到当前这个进程的信号向量表中。从这可见进程这能设置
// 自己的信号服务程序。
    current->sigaction[signum-1] = tmp;
    return handler;
}
```

系统调用 sys_sigaction

sys_sigaction 与 sys_signal 很像，只是 sys_sigaction 为设置提供了更多的自由度，因为它是通过信号的执行结构来设置的。<kernel/signal.c>

```
// 参数：signum 是信号类型；action 是新的信号执行结构；oldaction：如果不空就用来
// 保存原来的信号执行结构。
int sys_sigaction(int signum, const struct sigaction * action,
    struct sigaction * oldaction)
```

```
{
    struct sigaction tmp;
// 还是先判断是否为有效的类型并且不能是 SIGKILL
    if (signum<1 || signum>32 || signum==SIGKILL)
        return -1;
// 保存原信号执行结构
    tmp = current->sigaction[signum-1];
// 将新的执行结构拷贝到进程的信号向量表中。这里没有直接复制，而是使用了一个
// 函数是因为，这个新的信号执行结构一定是位于用户区的，要使用用户态的选择子。
// 所以将相关的操作放到函数中了。
    get_new((char *) action,
        (char *) (signum-1+current->sigaction));
// 如果 oldaction 不空就将原执行结构保存进去。使用函数的原因同上
    if (oldaction)
        save_old((char *) &tmp,(char *) oldaction);
// 判断是否允许信号自身的嵌套，然后决定如何设置屏蔽位
    if (current->sigaction[signum-1].sa_flags & SA_NOMASK)
        current->sigaction[signum-1].sa_mask = 0;
    else
        current->sigaction[signum-1].sa_mask |= (1<<(signum-1));
    return 0;
}
```

辅助函数说明

1. **函数 get_new：**

   用于从用户区取得信号执行结构，用户的段选择子在系统调用的时候保存到了 fs 寄存器中。很简单不做更多的说明了。

```
    static inline void get_new(char * from,char * to)
    {
        int i;

        for (i=0 ; i< sizeof(struct sigaction) ; i++)
// 这个宏定义位于 include/asm/segment.h 中
            *(to++) = get_fs_byte(from++);
    }
```

2. **函数 save_old:**

   这个函数在执行前会检验要写入的地址空间是否足够，否则就会去申请一页空间。这些工作会有函数 verify_area, 位于<kernel/fork.c>,来做。

```
    static inline void save_old(char * from,char * to)
    {
        int i;

        verify_area(to, sizeof(struct sigaction));
```

```
        for (i=0 ; i< sizeof(struct sigaction) ; i++) {
            put_fs_byte(*from,to);
            from++;
            to++;
        }
    }
```

系统调用 sys_ssetmask

前两个系统调用是用来设置信号的执行结构的，但是信号的响应机制中还提到过两个“寄存器”：“信号请求寄存器”signal 和“信号屏蔽寄存器”blocked。对于 signal 的设置属于信号的传递后面介绍。对于 blocked 的设置就是使用这个系统调用来实现的。<kernel/signal.c>

```
int sys_ssetmask(int newmask)
{
    int old=current->blocked;
// 除了信号 SIGKILL,所有信号用户都可以屏蔽掉。因此无论如何都要将对 SIGKILL 的
// 屏蔽去掉。
    current->blocked = newmask & ~(1<<(SIGKILL-1));
// 将原来的屏蔽位图返回，同样是使用 eax 寄存器
    return old;
}
```

系统调用 sys_sgetmask

提供了对 block 设置的操作同时，还要有索取的。这个系统调用就是用于返回当前进程的“信号屏蔽寄存器”block。<kernel/signal.c>

```
int sys_sgetmask()
{
    return current->blocked;
}
```

信号的传递

与中断做类比，信号的传递应该是一件很简单的事情（在 Linux0.11 中是这样的）。只需要设置相应进程的“信号请求寄存器”signal 就可以了。为了确保安全，在传递的时候需要检查一下发送信号的进程的权限，这同是系统态还用户态不同。这里的权限指的是登陆身份。在 Linux0.11 中，可以看到执行了发送信号操作的都在文件 kernel/exit.c 中。注意：函数 send_sig 不是系统调用！

```
// 参数：sig 是信号类型；p 指向接收信号的进程；priv：权限
static inline int send_sig(long sig,struct task_struct * p,int priv)
```

```
{
// 判断信号是否有效，以及要接受信号的进程时候存在
    if (!p || sig<1 || sig>32)
        return -EINVAL;
// 如果有权发送信号或者要接收信号的进程与当前进程同级或者当前进程是超级用户
// 就设置 signal。宏定义 suer()位于 include/kernel/kernel.h 中。
// #define suser() (current->euid == 0)
//
    if (priv || (current->euid==p->euid) || suser())
        p->signal |= (1<<(sig-1));
    else
        return -EPERM;
    return 0;
}
```

信号的执行

只有在从系统调用返回的时候才会去处理信号相应程序。信号的执行可以划分为两个阶段：预处理阶段和执行阶段。

预处理阶段

信号执行的预处理阶段位于系统调用的返回阶段，当时我没有介绍是因为其中涉及到一些与信号有关的知识。在这一阶段将会根据信号屏蔽寄存器来判断是否需要相应信号请求，为执行阶段准备参数。<kernel/system_call.s>

```
ret_from_sys_call:
// 将当前进程的指针放入 eax
    movl _current,%eax          # task[0] cannot have signals
// 如果发现当前进程是初始进程就不执行信号处理，因为初始进程是不会接收信号的
    cmpl _task,%eax
    je 3f
// 由于信号服务函数需要使用原来进程的代码段和数据段，因此要判断
    cmpw $0x0f,CS(%esp)         # was old code segment supervisor ?
    jne 3f
    cmpw $0x17,OLDSS(%esp)      # was stack segment = 0x17 ?
    jne 3f
// 获取当前进程的信号请求寄存器放入 ebx
    movl signal(%eax),%ebx
// 获取当前进程的信号屏蔽寄存器放入 ecx
    movl blocked(%eax),%ecx
// 获取没有被屏蔽的信号，放入 ecx
    notl %ecx
    andl %ebx,%ecx
```

```
// 从 ecx 的第 0 位向第 31 位扫描，寻找第一个为 1 的位的偏移值（0－31），放入 ecx
    bsfl %ecx,%ecx
// 如果没有找到说明没有需要处理信号
    je 3f
// 将 ebx 中偏移量为 ecx 的位清 0，ebx 中存放的是当前进程信号请求寄存器，因此这
// 一步是复位相应的信号位
    btrl %ecx,%ebx
// 保存信号请求寄存器
    movl %ebx,signal(%eax)
// 由于信号类型是从 1 开始计数的，因此要将偏移量（在 ecx 中）加 1
    incl %ecx
// 将信号类型作为参数传递给函数 do_signal
    pushl %ecx
    call _do_signal
    popl %eax
    ……
    iret
```

信号的执行

在信号的执行阶段最主要的工作为信号处理程序设定相应的运行环境。

```
void do_signal(long signr,long eax, long ebx, long ecx, long edx,
    long fs, long es, long ds,
    long eip, long cs, long eflags,
    unsigned long * esp, long ss)
{
    unsigned long sa_handler;
    long old_eip=eip;
// 从信号向量表中获取信号执行结构
    struct sigaction * sa = current->sigaction + signr - 1;
    int longs;
    unsigned long * tmp_esp;

// 从信号执行结构中获取信号处理程序的句柄
    sa_handler = (unsigned long) sa->sa_handler;
// 如果句柄是 1，表示忽略信号的处理
    if (sa_handler==1)
        return;
// 如果句柄是 0，表示开始执行默认程序
    if (!sa_handler) {
        if (signr==SIGCHLD)
            return;
        else
            do_exit(1<<(signr-1));
```

```
        }
// 如果执行结构的标志中有 SA_ONESHOT 表示这个信号处理程序仅执行一次
        if (sa->sa_flags & SA_ONESHOT)
                sa->sa_handler = NULL;
// &eip 对应的是栈中系统调用将要返回执行的地址，将这个值设为信号处理程序的句
// 柄就能够使系统调用返回后执行信号处理程序
        *(&eip) = sa_handler;
// 设定信号处理程序将要使用的栈的长度
        longs = (sa->sa_flags & SA_NOMASK)?7:8;
        *(&esp) -= longs;
// 获取足够的空间
        verify_area(esp,longs*4);
        tmp_esp=esp;
// 为信号服务程序设定栈的内容
        put_fs_long((long) sa->sa_restorer,tmp_esp++);
        put_fs_long(signr,tmp_esp++);
        if (!(sa->sa_flags & SA_NOMASK))
                put_fs_long(current->blocked,tmp_esp++);
        put_fs_long(eax,tmp_esp++);
        put_fs_long(ecx,tmp_esp++);
        put_fs_long(edx,tmp_esp++);
        put_fs_long(eflags,tmp_esp++);
        put_fs_long(old_eip,tmp_esp++);
// 如果 sa_mask 中设定了对自身屏蔽，这一步就会将这个屏蔽设置到信号屏蔽寄存器中
        current->blocked |= sa->sa_mask;
}
```

第十一章进程的创建

前面讲了那么多与进程有关的操作，但是进程使怎么产生的呢？简单的说，新进程就是通过复制旧进程产生的。因此就将新进程叫子进程，旧进程叫父进程。这个操作是由系统调用 fork 完成的。在创建后，子进程与父进程共享同一块内存空间，就是说子进程与父进程共同运行一个程序。为了能够区分出这个程序是由父进程运行还是由子进程运行，fork 在返回的时候会为父进程和子进程设定不同的返回值。通过这个返回值就可以区分出来了。

系统调用 fork

在中断系统中讲到了一个系统调用进入阶段和退出阶段，至于系统调用执行阶段是由系统调用的功能号决定的。对于 fork 来说，它对应的功能号是 2。从系统调用表 sys_call_table[](位于 include/linux/sys.h 中)，可以看到它对应的是函数 sys_fork。它先获取一个空闲的进程 id 号，然后复制父进程。

<kernel/system_call.s>

```
// 首先调用 C 函数 find_empty_process()，取得一个进程号 pid。若返回负数则说明目前
// 任务数组已满,直接退出。否则调用 C 函数 copy_process()复制进程。这里的 C 函数都
// 是定义在/kernel/fork.c 中
.align 2
_sys_fork:
// 找到一个可用空间，将空间的索引号返回。函数是通过 eax 来传递返回值的。
    call _find_empty_process
    testl %eax,%eax
    js 1f
// 将寄存器 gs、esi、edi、ebp 和 eax 的值显示的压入栈中用作 copy_process()函数的
// 参数。然后调用 copy_process()进行进程的拷贝
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call _copy_process
// 这实际上是一个模拟弹栈的过程，因为不需要恢复这些寄存器所以仅使用了加法
    addl $20,%esp
1:  ret
```

函数 int find_empty_process(void)

```
    // last_pid 是一个全局变量用于保存最新分配的进程号
long last_pid=0;
```

```
。。。。。。

// 先找到一个目前没有使用的 PID，然后任务表中找到一个目前没有使用的空间
// 将该空间对应的索引号返回，如果没有找到空间就返回-EAGAIN(-11)
int find_empty_process(void)
{
    int i;
// last_pid 是从 0 开始计数依次递增，达到上限后重新从 1 开始计数。
// 查找没有使用的 PID 的方法是：先产生一个 last_pid，然后搜索任务表，将目前
// 存在的每一个任务的 PID 与这个 last_pid 相比较。如果发现相等，即该 PID 已经
// 分配过了，就重新产生一个 last_pid，再次搜索任务表。
    repeat:
        if ((++last_pid)<0) last_pid=1;
        for(i=0 ; i<NR_TASKS ; i++)
            if (task[i] && task[i]->pid == last_pid) goto repeat;
// 从任务表中找到可用空间，将该空间对应的索引号返回
    for(i=1 ; i<NR_TASKS ; i++)
        if (!task[i])
            return i;
    return -EAGAIN;
}
```

函数 int copy_process(... ..)

这个函数可以说是 fork 的核心，复制父进程的操作就是由它完成的。按照执行的顺序可以将其分解成 5 个部分：进程常规变量的设定、TSS 的设置、分配内存、文件设定及其它。

进程常规变量的设定

```
/*
 *  Ok, this is the main fork-routine. It copies the system process
 * information (task[nr]) and sets up the necessary registers. It
 * also copies the data segment in it's entirety.
 */
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    struct task_struct *p;
    int i;
    struct file *f;
```

```
// 为新的进程分配一页内存空间
p = (struct task_struct *) get_free_page();
// 如果分配失败就返回错误码
if (!p)
        return -EAGAIN;
// 将新进程空间的首地址赋给任务表
task[nr] = p;
// 由于当前的进程就是父进程，所以这里是对父进程的完整拷贝
*p = *current;/* NOTE! this doesn't copy the supervisor stack */
// 新进程的状态是不可中断的等待状态
p->state = TASK_UNINTERRUPTIBLE;
// 设置子进程的 ID
p->pid = last_pid;
// 设置子进程的父进程 ID
p->father = current->pid;
// 让子进程与父进程的优先级相同
p->counter = p->priority;
p->signal = 0;
p->alarm = 0;
p->leader = 0;        /* process leadership doesn't inherit */
p->utime = p->stime = 0;
p->cutime = p->cstime = 0;
// 设置进程的开始时间为当前的滴答数
p->start_time = jiffies;
```

TSS 的设置

```
// 设置新进程的 TSS 内容
p->tss.back_link = 0;
// 设置系统栈的地址，系统栈与 PCB 位于同一页中，栈底是页顶如图：
```

系统栈
未使用
PCB 结构

```
p->tss.esp0 = PAGE_SIZE + (long) p;
p->tss.ss0 = 0x10;
// 设置子进程运行时的 IP，它与父进程调用系统调用时的 IP 相同
p->tss.eip = eip;
p->tss.eflags = eflags;
// 设置对子进程执行 fork 函数后的返回值，系统调用返回值是在 eax 寄存器中
p->tss.eax = 0;
```

```
        p->tss.ecx = ecx;
        p->tss.edx = edx;
        p->tss.ebx = ebx;
        p->tss.esp = esp;
        p->tss.ebp = ebp;
        p->tss.esi = esi;
        p->tss.edi = edi;
        // 段寄存器只有低字节有效
        p->tss.es = es & 0xffff;
        p->tss.cs = cs & 0xffff;
        p->tss.ss = ss & 0xffff;
        p->tss.ds = ds & 0xffff;
        p->tss.fs = fs & 0xffff;
        p->tss.gs = gs & 0xffff;
        // 设置新进程的 LDT 地址
        p->tss.ldt = _LDT(nr);
        p->tss.trace_bitmap = 0x80000000;

        // 如果当前任务使用了协处理器，就保存其上下文
        if (last_task_used_math == current)
                __asm__("clts ; fnsave %0"::"m" (p->tss.i387));
```

分配内存

```
        // 为子进程设置代码段
        if (copy_mem(nr,p)) {
                task[nr] = NULL;
                free_page((long) p);
                return -EAGAIN;
        }
```

这里调用了一个函数 copy_mem。

内存分配函数 copy_mem

```
// 设置新任务的代码和数据段基址、限长并复制页表
// nr 为新任务号；p 是新任务数据结构的指针
int copy_mem(int nr,struct task_struct * p)
{
        unsigned long old_data_base,new_data_base,data_limit;
        unsigned long old_code_base,new_code_base,code_limit;
// 取局部描述符表中代码段描述符项中段限长
        code_limit=get_limit(0x0f);
// 取局部描述符表中数据段描述符项中段限长
```

```
        data_limit=get_limit(0x17);
        // 取原代码段的基地址
        old_code_base = get_base(current->ldt[1]);
        // 取原数据段的基地址
        old_data_base = get_base(current->ldt[2]);
        // Linux0.11 不支持数据段同代码段分开这种情况
        if (old_data_base != old_code_base)
            panic("We don't support separate I&D");
        if (data_limit < code_limit)
            panic("Bad data_limit");
        new_data_base = new_code_base = nr * 0x4000000;
        // 设置代码段的起始地址，这是一个虚拟地址，在执行了 copy_page_tables 之后
        // 它所映射到的物理空间与父进程相同
        p->start_code = new_code_base;
        set_base(p->ldt[1],new_code_base);
        set_base(p->ldt[2],new_data_base);
        if (copy_page_tables(old_data_base,new_data_base,data_limit)) {
            free_page_tables(new_data_base,data_limit);
            return -ENOMEM;
        }
        return 0;
    }
```

说明：

get_limit():

定义在/kernel/sched.h 中，用于取得

```
#define get_limit(segment) ({ \
unsigned long __limit; \
__asm__("lsll %1,%0\n\tincl %0":"=r" (__limit):"r" (segment)); \
__limit;})
```

这里用到了一个指令 LSLL，它用于从描述符中取得段限长。其格式为：

LSLL　　OPRD1,OPRD2

其中，操作数 OPRD1 可以是 16 位或 32 位通用寄存器，操作数 OPRD2 是 16 位通用寄存器或存储单元，也可以是 32 位通用寄存器或存储单元。操作数 OPRD1 和 OPRD2 的尺寸必须一致。该指令把操作数 OPRD2 视为选择子(当为 32 位时，仅使用低 16 位)，如果 OPRD2 所指示的描述符满足如下条件，那么零标志 ZF 被置 1，并把描述符内的界限字段装入 OPRD1；否则，ZF 清 0，OPRD1 保持不变。

装入到 OPRD1 的由 OPRD2 所指示的描述符中的界限字段以字节位为单位。如果描述符中的界限字段以 4K 字节为单位(G=1)，那么装入到 OPRD1 时被左移 12 位，空出的低位全部填成 1。

注意，如果指令使用 16 位操作数，那么只有段界限的低 16 位被装入到 OPRD1。该指令只影响 ZF 标志。

在这里因为是用户态选择局部描述符，所以代码段是 0x0f，数据段得选择子是 0x17。

文件设定

```
// 如果父进程中有文件是打开的，则将对应文件的打开次数增 1
    for (i=0; i<NR_OPEN;i++)
        if (f=p->filp[i])
            f->f_count++;
// 当前进程（父进程）的 pwd, root 和 executable 引用次数均增 1
    if (current->pwd)
        current->pwd->i_count++;
    if (current->root)
        current->root->i_count++;
    if (current->executable)
        current->executable->i_count++;
```

其它

```
    // 在内存的 TSS 描述符表和 LDT 描述符表中建立新进程的描述符。
    // 新进程的描述符对应的 TSS 和 LDT 是从进程中取得的。
    // 目前的 LDT 与父进程的 LDT 是相同的（复制过来的嘛）
    set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
    set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
    p->state = TASK_RUNNING;      /* do this last, just in case */
    return last_pid;
}
```

生成进程后的返回值问题

我们知道，在调用 fork()函数之后，这个函数会向父进程返回子进程的进程号 PID，而向子进程返回 0。有没有想过一个函数怎么可能有两个返回值呢？而且这两个返回值还不相同!（我第一次听到的时候，吃惊了很久）

例如这个程序应该会在屏幕上输出两行：(该程序尚未检验)

```
static inline _syscall0(int,fork)
int main ()
{
    if (fork()==0)
        printf("I'm child!\n");
    else
        printf("I'm father!\n");
    return 0;
}
```

的确，如果只用一个进程来使用这个函数的时候返回值毫无疑义就是一个。但是，fork()函数是用于产生进程的，因此实际上是有两个进程（父子进程）来共享调用 fork()的程序段。

而对于这两个进程来说就是返回两个了。为了理解这个问题，我们来重新看看这个过程在操作系统中的实现。

首先，将这里的宏定义展开得到：

```
static inline int fork(void)
{
long __res;
__asm__ volatile ("int $0x80"
                  :     "=a" (__res)
                  :     "0"  (__NR_fork));
if (__res>=0)
            return (int) __res;
errno = -__res;
return –1;
}
```

由此可见 fork()函数正确执行的返回值就是从中断返回后的 eax 寄存器的值。在执行系统调用的时候会将当前的 EIP 和 CS 压入堆栈中。在介绍 copy_process()函数的时候已经说过了，这个有硬件压入到栈中的 EIP 和 CS 会通过参数 long eip 和 long cs 传递给 copy_process()函数。然后参数 eip 和 cs 赋给了子进程。因此，新产生的子进程与父进程在内存中共享这段程序代码。换句话说，当子进程开始执行的时候，也会从调用函数 fork()之后的地方开始执行的。因此，对于同一个程序，如果相应进程对应的 eax 寄存器的值不同，就会导致函数的返回值不同。具体的情况需要到内核中才能看到。

好，我们现在开始进到系统的内核中，看看返回的时候 eax 中是什么东东。

回顾一下在/kernel/systemcall.s 中的_system_call 函数：

```
.align 2
_system_call:
        ……
        call _sys_call_table(,%eax,4)
        pushl %eax
        ……
3:      popl %eax
        ……
        iret
```

在执行 call *sys*call_table(, %eax, 4)之前，eax 中存放的是系统调用的功能号。对于 fork()函数来说，当它执行完 sys_fork()函数之后，eax 中存放的是新创建的子进程的进程号（PID）。为什么呢？别急，我们再温习一下 sys_fork()函数的实现：

```
.align 2
_sys_fork:
        ……
        push %gs
        pushl %esi
        pushl %edi
        pushl %ebp
        pushl %eax
        call _copy_process
```

addl $20,%esp

1: ret

这里使用了 copy_process()函数，而 copy_process()函数的最后一句话是 return last_pid;（返回子进程号）。这个函数的返回值就是存放在 eax 寄存器中，因此 eax 中存放的就是子进程号。有没有感觉到我在逃避责任？凭什么就说函数的返回值就是放在 eax 中，那么多寄存器呢，返回值怎么就不放在 ebx，edx，ecx 或 e 某某 x 中呢！请暂时相信它，然后带着这个疑问往下看，一会儿我会通过证明我的观点的☺。

在 sys_fork()函数的末尾之所以没有执行弹栈操作，既是因为在 system_call()函数中没有引用过 gs、esi、edi 和 ebp 寄存器（eax 在使用前已经通过压栈操作保存了），更是因为在 eax 中保存着返回值，如果弹栈了，返回值就会丢失。

当执行完 sys_fork()函数之后，会根据情况来决定是否执行调度程序。如果没有执行调度程序，那么就会直接从系统中断中返回。这个时候 eax 自然就是子程序号。如果执行了调度程序，那么父进程可能会失去占有 CPU 的权利。但是，在进程切换的时候是通过执行宏定义 switch_to(n)来实现的。对于这个宏定义我已经讲过了，它是通过段间转移来实现进程切换的。在段间转移的时候，Intel 从硬件上就实现了将各个寄存器的当前值保存到即将被切换出去的进程的 TSS 中。这里就包括了保存父进程对应的这个 eax 寄存器值到父进程 TSS 中。

从这里可以看出来要想设置子进程的返回值，只要将子进程中 TSS 结构的 eax 设为 0 就可以了。Linux0.11 的确就是这么做的。在 copy_process()中有一句：

p->tss.eax = 0;

在这里就是为子进程设置返回值。

为了理解这个过程，可以参考下图：

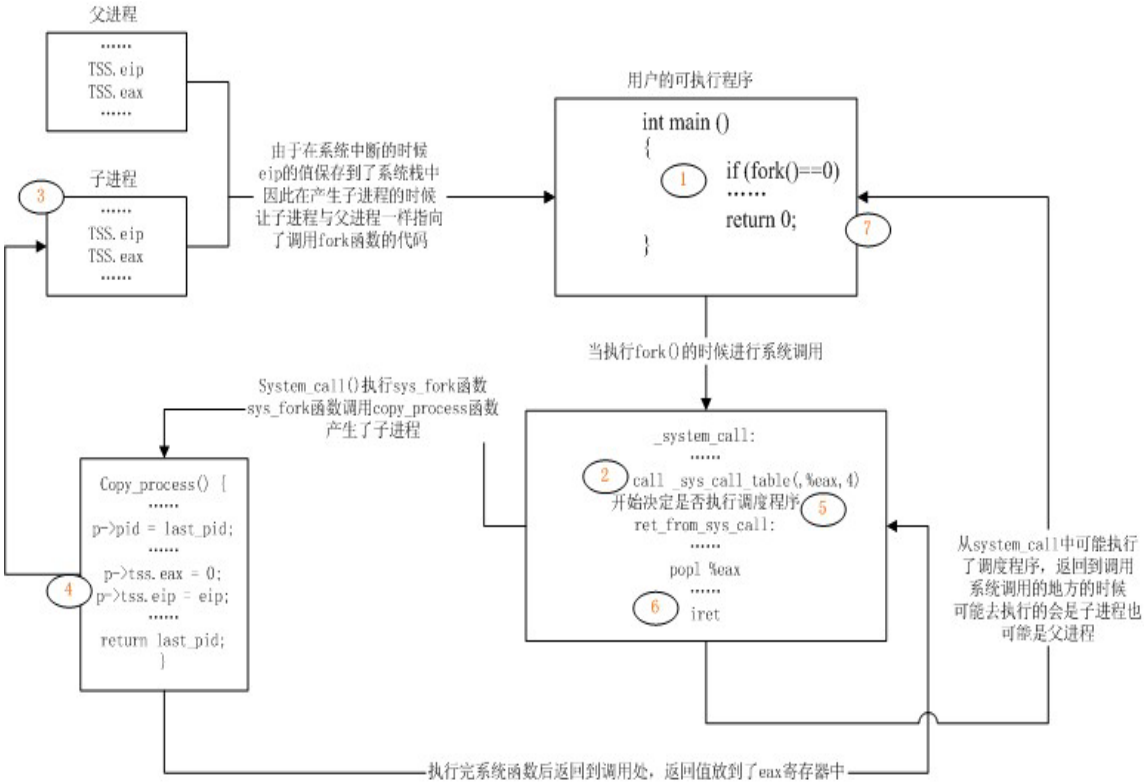


图 十一.1fork 函数生成返回值的过程

试验 关于函数的返回值问题：

试验器材：tcc.exe（V2.0）, nasmw.exe, link.exe(masmV5.0 的连接器),exe2com.exe, w32dasm(这是一个著名的 Windows 下的反汇编软件)

试验目的：证明使用 tcc 编译器编译的程序中，函数的返回值存放在 eax 寄存器中。

试验步骤：

1. 建立一个 C 语言文件 func.c，输入：

```
void copy(int    dx, int ax)
{
        int id = 1;
        return id;
}
```

2. 建立一个汇编语言文件 main.asm，输入：

```
[BITS 16]
[global start]
[extern _copy]
start:
        push dx
        push ax
        call _copy
```

3. 将两个文件编译成.obj 格式，输入：

   tcc –mf –ofunc.obj –c –func.c

   nasmw –f obj –o main.obj –main.asm

4. 将文件连接成 main.exe:

   link main.obj func.obj, main.exe,,,

5. 将 main.exe 转换为 main.com:

   exe2com main.exe

6. 使用 W32DASM 来对 main.com 反汇编，得到结果为：

```
//********************** Start of Code in Segment: 1 **************

:0001.0100 52                          push dx
:0001.0101 50                          push ax
:0001.0102 E80000                      call 0105

* Referenced by a CALL at Address:
|:0001.0102
|
:0001.0105 55                          push bp
:0001.0106 8BEC                        mov bp, sp
:0001.0108 56                          push si
:0001.0109 BE0100                      mov si, 0001
:0001.010C 8BC6                        mov ax, si
:0001.010E EB00                        jmp 0110



* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:0001.010E(U)
|
:0001.0110 5E                          pop si
:0001.0111 5D                          pop bp
:0001.0112 C3                          ret
```

图 十一.2 反汇编的运行结果

结果分析：

从 0001.0105 开始到 0001.0112 都是函数 copy()的反汇编结果，si 寄存器相当于局部变量 a，先将 1 传入 si 寄存器。然后将 si 的值赋给 ax 并弹栈返回。

由此可见，函数 copy()是通过 eax 积存器返回运行结果的。

第十二章进程的执行

进程创建成功后，会与父进程共享同一个程序。为了能够是子进程执行其它的程序就需要将要执行的程序调入内存中。这个工作是由系统调用 execve 完成的。它可以根据可执行文件中的信息为其建立相应的运行环境。同时，它会处理批处理文件。

可执行文件的格式[2]

Linux 内核0.11 版仅支持a.out(Assembley out)执行文件格式。下面全面介绍一下a.out格式。

在头文件<a.out.h>中申明了三个数据结构以及一些宏函数。这些数据结构描述了系统上可执行的机器码文件（二进制文件）。

一个执行文件共可有七个部分（七节）组成。按照顺序，这些部分是：

执行头部分(exec header)：该部分中含有一些参数，内核使用这些参数将执行文件加载到内存中并执行，而链接程序(ld)使用这些参数将一些二进制目标文件组合成一个可执行文件。这是唯一必要的组成部分。

代码段部分(text segment)：含有程序执行使被加载到内存中的指令代码和相关数据。可以以只读形式进行加载。

数据段部分(data segment)：这部分含有已经初始化过的数据，总是被加载到可读写的内存中。

代码重定位部分(text relocations)：这部分含有供链接程序使用的记录数据。在组合二进制目标文件时用于定位代码段中的指针或地址。

数据重定位部分(data relocations)：与代码重定位部分的作用类似，但是是用于数据段中指针的重定位。

符号表部分(simbol table)：这部分同样含有供链接程序使用的记录数据，用于在二进制目标文件之间对命名的变量和函数（符号）进行交叉引用。

字符串表部分(string table)：该部分含有与符号名相对应的字符串。

文件头格式

每个二进制执行文件均以一个执行数据结构（exec structure）开始。该数据结构的形式如下：<include/a.out.h>

```
struct exec {
  unsigned long a_magic;   /* Use macros N_MAGIC, etc for access */
  unsigned a_text;      /* length of text, in bytes */
  unsigned a_data;         /* length of data, in bytes */
  unsigned a_bss;       /* length of uninitialized data area for file, in bytes */
  unsigned a_syms;         /* length of symbol table data in file, in bytes */
  unsigned a_entry;        /* start address */
```

[2] 摘自《Linux0.11 版本完全分析》

```
    unsigned a_trsize;      /* length of relocation info for text, in bytes */
    unsigned a_drsize;      /* length of relocation info for data, in bytes */
};
```

各个字段的功能如下：

a_midmag: 该字段含有被N_GETFLAG()、N_GETMID 和N_GETMAGIC()访问的子部分，是由链接程序在运行时加载到进程地址空间。宏N_GETMID()用于返回机器标识符(machine-id)，指示出二进制文件将在什么机器上运行。N_GETMAGIC()宏指明魔数，它唯一地确定了二进制执行文件与其它加载的文件之间的区别。字段中必须包含以下值之一：

OMAGIC表示代码和数据段紧随在执行头后面并且是连续存放的。内核将代码和数据段都加载到可读写内存中。

NMAGIC 同OMAGIC 一样，代码和数据段紧随在执行头后面并且是连续存放的。然而内核将代码加载到了只读内存中，并把数据段加载到代码段后下一页可读写内存边界开始。

ZMAGIC 内核在必要时从二进制执行文件中加载独立的页面。执行头部、代码段和数据段都被链接程序处理成多个页面大小的块。内核加载的代码页面时只读的，而数据段的页面是可写的。

a_text 该字段含有代码段的长度值，字节数。

a_data 该字段含有数据段的长度值，字节数。

a_bss 含有‘bss 段’的长度，内核用其设置在数据段后初始的break（brk）。内核在加载程序时，这段可写内存显现出处于数据段后面，并且初始时为全零。

a_syms 含有符号表部分的字节长度值。

a_entry 含有内核将执行文件加载到内存中以后，程序执行起始点的内存地址。

a_trsize 该字段含有代码重定位表的大小，是字节数。

a_drsize 该字段含有数据重定位表的大小，是字节数。

在a.out.h 头文件中定义了几个宏，这些宏使用exec 结构来测试一致性或者定位执行文件中各个部分（节）的位置偏移值。这些宏有：

N_BADMAG(exec) 如果a_magic 字段不能被识别，则返回非零值。

N_TXTOFF(exec) 代码段的起始位置字节偏移值。

N_DATOFF(exec) 数据段的起始位置字节偏移值。

N_DRELOFF(exec) 数据重定位表的起始位置字节偏移值。

N_TRELOFF(exec) 代码重定位表的起始位置字节偏移值。

N_SYMOFF(exec) 符号表的起始位置字节偏移值。

N_STROFF(exec) 字符串表的起始位置字节偏移值。

重定位记录

重定位记录具有标准格式，它使用重定位信息(relocation_info)结构来描述：

```
/* This structure describes a single relocation to be performed.
   The text-relocation section of the file is a vector of these structures,
   all of which apply to the text section.
   Likewise, the data-relocation section applies to the data section.   */

struct relocation_info
```

```
{
  /* Address (within segment) to be relocated.  */
  int r_address;
  /* The meaning of r_symbolnum depends on r_extern.  */
  unsigned int r_symbolnum:24;
  /* Nonzero means value is a pc-relative offset
     and it should be relocated for changes in its own address
     as well as for changes in the symbol or section specified.  */
  unsigned int r_pcrel:1;
  /* Length (as exponent of 2) of the field to be relocated.
     Thus, a value of 2 indicates 1<<2 bytes.  */
  unsigned int r_length:2;
  /* 1 => relocate with value of symbol.
          r_symbolnum is the index of the symbol
       in file's the symbol table.
     0 => relocate with the address of a segment.
          r_symbolnum is N_TEXT, N_DATA, N_BSS or N_ABS
       (the N_EXT bit may be set also, but signifies nothing).  */
  unsigned int r_extern:1;
  /* Four bits that aren't used, but when writing an object file
     it is desirable to clear them.  */
  unsigned int r_pad:4;
};
```

该结构中各字段的含义如下：

r_address 该字段含有需要链接程序处理（编辑）的指针的字节偏移值。代码重定位的偏移值是从代码段开始处计数的，数据重定位的偏移值是从数据段开始处计算的。链接程序会将已经存储在该偏移处的值与使用重定位记录计算出的新值相加。

r_symbolnum 该字段含有符号表中一个符号结构的序号值（不是字节偏移值）。链接程序在算出符号的绝对地址以后，就将该地址加到正在进行重定位的指针上。（如果r_extern比特位是0，那么情况就不同，见下面。）

r_pcrel 如果设置了该位，链接程序就认为正在更新一个指针，该指针使用pc 相关寻址方式，是属于机器码指令部分。当运行程序使用这个被重定位的指针时，该指针的地址被隐式地加到该指针上。

r_length 该字段含有指针长度的2 的次方值：0 表示1 字节长，1 表示2 字节长，2 表示4 字节长。

r_extern 如果被置位，表示该重定位需要一个外部引用；此时链接程序必须使用一个符号地址来更新相应指针。当该位是0 时，则重定位是“局部”的；链接程序更新指针以反映各个段加载地址中的变化，而不是反映一个符号值的变化（除非同时设置了r_baserel，见下面）。在这种情况下，r_symbolnum字段的内容是一个n_type 值（见下面）；这类字段告诉链接程序被重定位的指针指向那个段。

r_baserel 如果设置了该位，则r_symbolnum 字段指定的符号将被重定位成全局偏移表(GlobalOffset Table)中的一个偏移值。在运行时刻，全局偏移表该偏移处被设置为符号的地址。

r_jmptable 如果被置位，则r_symbolnum 字段指定的符号将被重定位成过程链接表

(ProcedureLinkage Table)中的一个偏移值。

r_relative 如果被置位，则说明此重定位与该目标文件将成为其组成部分的映象文件在运行时被加载的地址相关。这类重定位仅在共享目标文件中出现。

r_copy 如果被置位，该重定位记录指定了一个符号，该符号的内容将被复制到r_address指定的地方。该复制操作是通过共享目标模块中一个合适的数据项中的运行时刻链接程序完成的。符号将名称映射为地址（或者更通俗地讲是字符串映射到值）。由于链接程序对地址的调整，一个符号的名称必须用来表示其地址，直到已被赋予一个绝对地址值。符号是由符号表中固定长度的记录以及字符串表中的可变长度名称组成。

符号表

符号表是nlist 结构的一个数组，如下所示：

```
#ifndef N_NLIST_DECLARED
struct nlist {
  union {
    char *n_name;
    struct nlist *n_next;
    long n_strx;
  } n_un;
  unsigned char n_type;
  char n_other;
  short n_desc;
  unsigned long n_value;
};
#endif
```

其中各字段的含义为：

n_un.n_strx 含有本符号的名称在字符串表中的字节偏移值。当程序使用nlist()函数访问一个符号表时，该字段被替换为n_un.n_name 字段，这是内存中字符串的指针。

n_type 用于链接程序确定如何更新符号的值。使用位屏蔽(bitmasks)可以将n_type 字段分割成三个子字段，对于N_EXT 类型位置位的符号，链接程序将它们看作是“外部的”符号，并且允许其它二进制目标文件对它们的引用。N_TYPE 屏蔽码用于链接程序感兴趣的比特位：

N_UNDF 一个未定义的符号。链接程序必须在其它二进制目标文件中定位一个具有相同名称的外部符号，以确定该符号的绝对数据值。特殊情况下，如果n_type 字段是非零值，并且没有二进制文件定义了这个符号，则链接程序在BSS 段中将该符号解析为一个地址，保留长度等于n_value的字节。如果符号在多于一个二进制目标文件中都没有定义并且这些二进制目标文件对其长度值不一致，则链接程序将选择所有二进制目标文件中最大的长度。

N_ABS 一个绝对符号。链接程序不会更新一个绝对符号。

N_TEXT 一个代码符号。该符号的值是代码地址，链接程序在合并二进制目标文件时会更新其值。

N_DATA 一个数据符号；与N_TEXT 类似，但是用于数据地址。对应代码和数据符号的值不是文件的偏移值而是地址；为了找出文件的偏移，就有必要确定相关部分开始加载的地址并减去它，然后加上该部分的偏移。

N_BSS 一个BSS 符号；与代码或数据符号类似，但在二进制目标文件中没有对应的偏移。

N_FN 一个文件名符号。在合并二进制目标文件时，链接程序会将该符号插入在二进制文件中的符号之前。符号的名称就是给予链接程序的文件名，而其值是二进制文件中首个代码段地址。链接和加载时不需要文件名符号，但对于调式程序非常有用。
N_STAB 屏蔽码用于选择符号调式程序(例如gdb)感兴趣的位；其值在stab()中说明。

n_other 该字段按照n_type 确定的段，提供有关符号重定位操作的符号独立性信息。目前，n_other字段的最低4 位含有两个值之一：AUX_FUNC 和AUX_OBJECT（有关定义参见<link.h>）。AUX_FUNC将符号与可调用的函数相关，AUX_OBJECT 将符号与数据相关，而不管它们是位于代码段还是数据段。该字段主要用于链接程序ld，用于动态可执行程序的创建。

n_desc 保留给调式程序使用；链接程序不对其进行处理。不同的调试程序将该字段用作不同的用途。

n_value 含有符号的值。对于代码、数据和BSS 符号，这是一个地址；对于其它符号（例如调式程序符号），值可以是任意的。字符串表是由长度为u_int32_t 后跟一null 结尾的符号字符串组成。长度代表整个表的字节大小，所以在32 位的机器上其最小值（或者是第1 个字符串的偏移）总是4。

系统调用 execve

进程的执行是依靠系统调用 execve 来实现的。第一次调用 execve 是在执行初始化函数 init()的时候调用的（位于/init/main.c）。在第一次看到调用这个函数的时候，与调用 fork 不同。因为对于 fork 调用的时候，这个函数的实现已经在文件头部使用宏实现了对其的定义：***static inline _syscall0(int,fork)***。但是，我们却只能看到对函数 execve 的声明 ***int execve(const char * filename, char ** argv, char ** envp);***（位于/include/unistd.h），没有看到他的定义。这涉及到了 C 的多文件编译，将经常使用的函数的定义和声明分别放到.c 和.h 文件中。使用的时候仅仅将包含该函数声明的头文件包括到文件中就可以使用相应的函数了。因此，对于函数 execve，可以从/lib/exec.c 中看到它的定义：***_syscall3(int,execve,const char *,file,char **,argv,char **,envp)***。

函数 execve 使用了 3 个参数，在执行系统调用的时候分别将这 3 个参数：filename 存放到 ebx 寄存器、argv 存到 ecx 寄存器、envp 存到 ebx 寄存器中。在调用相应的服务程序之前将这 3 个寄存器的值作为参数压入系统堆栈中。对应 execve 的内核函数是 sys_execve(位于 systemcall.s)：

```
.align 2
_sys_execve:
// 将执行系统调用的时候的存放 IP 值的地址作为参数传递给函数 do_execve
    lea EIP(%esp),%eax
    pushl %eax
// 调用函数 do_execve。这个时候从系统栈顶开始的 20 个字节中存放的是
// 系统调用时的 IP 的存放地址、调用_sys_execve 时的 IP 值、文件名地址、参数
// 表地址、环境表地址
    call _do_execve
    addl $4,%esp
    ret
```

函数 do_execve

进程的执行就是依靠这个函数来实现的。

```
/*
 * 'do_execve()' executes a new program.
 */
// 参数：eip 指向调用系统中断时存入系统栈的 eip 的地址；tmp：调用 do_execve 时
//          存入栈中的 eip（这个值没有用处）filename:文件名；argv:参数表指针
//          envp：环境变量表指针
int do_execve(unsigned long * eip,long tmp,char * filename,
        char ** argv, char ** envp)
{
        struct m_inode * inode;
        struct buffer_head * bh;
        struct exec ex;
// 这是一个临时的页映射表，MAX_ARG_PAGES 别定义为 32。这个表在使用的时候
// 用来存放内存页地址。参数和环境变量就是存放在这些页中。之所以说是临时的，
// 是因为，表中存放的是内存页的绝对地址程序需要的是虚拟地址。因此当完成虚拟
// 地址到绝对地址的映射之后，这个表就要被释放掉。32 页对应 32×4k＝128k 大小
// 的内存空间，可以说是足够存下任何正常的参数了。
        unsigned long page[MAX_ARG_PAGES];
        int i,argc,envc;
        int e_uid, e_gid;
        int retval;
        int sh_bang = 0;
//  p 中存放的是以页表 page 地址为基地址的偏移地址，初始的时候设置为页表最后一
// 项的相对偏移地址的下一个字节如图：
page 表：
```

0 项	……	……		MAX_ARG_PAGES-1 项

^p 指向这里

```
        unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4;

// 判断调用 execve 系统中断的是否为系统级的用户，如果是就停机。就是说这个调用
// 这个系统中断的不可以是系统态程序
        if ((0xffff & eip[1]) != 0x000f)
                panic("execve called from supervisor mode");
// 初始化临时页映射表
        for (i=0 ; i<MAX_ARG_PAGES ; i++)/* clear page-table */
                page[i]=0;
        if (!(inode=namei(filename)))           /* get executables inode */
                return -ENOENT;
// 计算出参数的个数和环境变量的个数
        argc = count(argv);
```

```
        envc = count(envp);

restart_interp:
        if (!S_ISREG(inode->i_mode)) {  /* must be regular file */
                retval = -EACCES;
                goto exec_error2;
        }
        i = inode->i_mode;
        e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
        e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;
        if (current->euid == inode->i_uid)
                i >>= 6;
        else if (current->egid == inode->i_gid)
                i >>= 3;
        if (!(i & 1) &&
                !((inode->i_mode & 0111) && suser())) {
                retval = -ENOEXEC;
                goto exec_error2;
        }
// 获得程序的第一块数据放到缓冲区中
        if (!(bh = bread(inode->i_dev,inode->i_zone[0]))) {
                retval = -EACCES;
                goto exec_error2;
        }
// 从刚才获得的数据块中取得文件头部信息
        ex = *((struct exec *) bh->b_data);     /* read exec-header */
// 对批处理文件的判断及处理
        if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
                /*
                 * This section does the #! interpretation.
                 * Sorta complicated, but hopefully it will work.  -TYT
                 */

                char buf[1023], *cp, *interp, *i_name, *i_arg;
                unsigned long old_fs;

                strncpy(buf, bh->b_data+2, 1022);
                brelse(bh);
                iput(inode);
                buf[1022] = '\0';
                if (cp = strchr(buf, '\n')) {
                        *cp = '\0';
                        for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
                }
```

```
if (!cp || *cp == '\0') {
        retval = -ENOEXEC; /* No interpreter name found */
        goto exec_error1;
}
interp = i_name = cp;
i_arg = 0;
for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++) {
        if (*cp == '/')
                i_name = cp+1;
}
if (*cp) {
        *cp++ = '\0';
        i_arg = cp;
}
/*
 * OK, we've parsed out the interpreter name and
 * (optional) argument.
 */
if (sh_bang++ == 0) {
        p = copy_strings(envc, envp, page, p, 0);
        p = copy_strings(--argc, argv+1, page, p, 0);
}
/*
 * Splice in (1) the interpreter's name for argv[0]
 *                  (2) (optional) argument to interpreter
 *                  (3) filename of shell script
 *
 * This is done in reverse order, because of how the
 * user environment and arguments are stored.
 */
p = copy_strings(1, &filename, page, p, 1);
argc++;
if (i_arg) {
        p = copy_strings(1, &i_arg, page, p, 2);
        argc++;
}
p = copy_strings(1, &i_name, page, p, 2);
argc++;
if (!p) {
        retval = -ENOMEM;
        goto exec_error1;
}
/*
 * OK, now restart the process with the interpreter's inode.
```

```
             */
            old_fs = get_fs();
            set_fs(get_ds());
            if (!(inode=namei(interp))) { /* get executables inode */
                set_fs(old_fs);
                retval = -ENOENT;
                goto exec_error1;
            }
            set_fs(old_fs);
            goto restart_interp;
        }
// 释放缓冲区
        brelse(bh);
// 下面对执行头信息进行处理。
// 对于下列情况，将不执行程序：如果执行文件不是需求页可执行文件(ZMAGIC)、或
// 者代码重定位部分长度 a_trsize 不等于 0、或者数据重定位信息长度不等于 0、或者
// 代码段+数据段+堆段长度超过 50MB、或者 i 节点表明的该执行文件长度小于代码段
//  +数据段+符号表长度+执行头部分长度的总和。
        if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
            ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
            inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
            retval = -ENOEXEC;
            goto exec_error2;
        }
        if (N_TXTOFF(ex) != BLOCK_SIZE) {
            printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
            retval = -ENOEXEC;
            goto exec_error2;
        }
// 复制参数和环境变量到临时页表中
        if (!sh_bang) {
            p = copy_strings(envc,envp,page,p,0);
            p = copy_strings(argc,argv,page,p,0);
// 如果 p 为 0，就表示参数表已经满了
            if (!p) {
                retval = -ENOMEM;
                goto exec_error2;
            }
        }
/* OK, This is the point of no return */
        if (current->executable)
            iput(current->executable);
        current->executable = inode;
        for (i=0 ; i<32 ; i++)
```

```
            current->sigaction[i].sa_handler = NULL;
        for (i=0 ; i<NR_OPEN ; i++)
            if ((current->close_on_exec>>i)&1)
                sys_close(i);
        current->close_on_exec = 0;
// 即将重新设置 LDT，需要释放原来所占有的 LDT。刚刚 fork 出来的子进程是与
// 父进程共享同一个内存空间的
        free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
        free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
        if (last_task_used_math == current)
            last_task_used_math = NULL;
        current->used_math = 0;
// 根据文件的代码段长度和参数的情况重新设置 LDT
        p += change_ldt(ex.a_text,page)-MAX_ARG_PAGES*PAGE_SIZE;
// 为参数建立参数表
        p = (unsigned long) create_tables((char *)p,argc,envc);
// 根据文件信息重新设置进程中的信息其中代码段结尾＝于文件中给出的代码段长度
// a_text，数据段结尾＝文件的代码段长度＋文件数据段长度；堆结尾＝文件的代码段
// 长度＋文件数据段长度＋文件中未初始化数据段长度
        current->brk = ex.a_bss +
            (current->end_data = ex.a_data +
            (current->end_code = ex.a_text));
// 设置进程可使用的堆栈地址
        current->start_stack = p & 0xfffff000;
        current->euid = e_uid;
        current->egid = e_gid;
// 在 bss 中清空出一页大小的空间
        i = ex.a_text+ex.a_data;
        while (i&0xfff)
            put_fs_byte(0,(char *) (i++));
// 设置该系统调用返回后的 EIP 值为程序的入口地址
        eip[0] = ex.a_entry;            /* eip, magic happens :-) */
// 设置该系统调用返回后的用户空间的栈地址
        eip[3] = p;                     /* stack pointer */
        return 0;
exec_error2:
        iput(inode);
exec_error1:
// 释放临时页映射表对应的页面
        for (i=0 ; i<MAX_ARG_PAGES ; i++)
            free_page(page[i]);
        return(retval);
}
```

函数 copy_strings

这个函数的作用是将环境变量表和参数变量表从后向前复制到临时页表中，它的实现比较琐碎。根据参数表和参数存放的空间不同可以分为 3 中情况：

0：参数 argv*在用户空间、参数表 argv**在用户空间；

1：参数 argv*在系统空间、参数表 argv**在用户空间；

2：参数 argv*在系统空间、参数表 argv**在系统空间；

对于不同的空间所需要的权限也是不同的，这里是通过使用 fs 寄存器来实现权限的变换的。对于环境参数这是一样的。

```
/*
 * 'copy_string()' copies argument/envelope strings from user
 * memory to free pages in kernel mem. These are in a format ready
 * to be put directly into the top of new user memory.
 *
 * Modified by TYT, 11/24/91 to add the from_kmem argument, which specifies
 * whether the string and the string array are from user or kernel segments:
 *
 * from_kmem     argv *          argv **
 *    0          user space      user space
 *    1          kernel space    user space
 *    2          kernel space    kernel space
 *
 * We do this by playing games with the fs segment register.  Since it
 * it is expensive to load a segment register, we try to avoid calling
 * set_fs() unless we absolutely have to.
 */
static unsigned long copy_strings(int argc,char ** argv,unsigned long *page,
                unsigned long p, int from_kmem)
{
        char *tmp, *pag;
        int len, offset = 0;
        unsigned long old_fs, new_fs;
// 判断是否有存放的空间
        if (!p)
                return 0; /* bullet-proofing */
// 由于在调用 system_call(位于/kernel/system_call.s)的时候已经在 ds 寄存器中存放了有
// 系统权限的使用全局描述符表选择子，在 fs 寄存器中存放了用户级的局部描述符表
// 选择子。在破坏之前需要保留这些值。
        new_fs = get_ds();
        old_fs = get_fs();
        if (from_kmem==2)
                set_fs(new_fs);
// 将参数从后向前将参数复制到页表的后部
```

```
        while (argc-- > 0) {
                if (from_kmem == 1)
                        set_fs(new_fs);
// 以 fs 为段地址从参数表中获得参数，由于是字符窜因此使用了字符指针
                if (!(tmp = (char *)get_fs_long(((unsigned long *)argv)+argc)))
                        panic("argc is wrong");
                if (from_kmem == 1)
                        set_fs(old_fs);
                len=0;          /* remember zero-padding */
// 统计参数的长度，这不执行完后 len 是参数的长度＋1，tmp 指向参数尾部的下一个
// 字节。
                do {
                        len++;
                } while (get_fs_byte(tmp++));
                if (p-len < 0) {/* this shouldn't happen - 128kB */
                        set_fs(old_fs);
                        return 0;
                }
// 从页尾部向前复制参数
                while (len) {
// 这一步执行之后 p 指向即将开始复制的地址，tmp 指向参数的尾部，len 为参数长度
                        --p; --tmp; --len;
// 得到此时 p 在内存某一页中的偏移地址
                        if (--offset < 0) {
                                offset = p % PAGE_SIZE;
                                if (from_kmem==2)
                                        set_fs(old_fs);
// 如果准备使用的页还没有分配呢，就分配一页内存空间
                                if (!(pag = (char *) page[p/PAGE_SIZE]) &&
                                    !(pag = (char *) page[p/PAGE_SIZE] =
                                      (unsigned long *) get_free_page()))
                                        return 0;
                                if (from_kmem==2)
                                        set_fs(new_fs);

                        }
                        *(pag + offset) = get_fs_byte(tmp);
                }
        }
        if (from_kmem==2)
                set_fs(old_fs);
// 返回复制完参数之后的临时页表的偏移值
        return p;
}
```

假设输入的参数表为 dir、pan.c；环境表为：/usr/root。那么执行完后，页表中的情况如图：

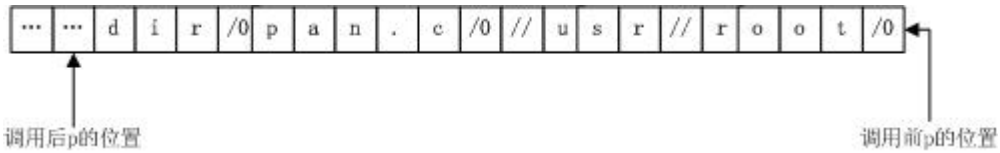


图 十二.1 参数在页表中的分布

函数 create_tables

前一个函数仅仅是将参数放入到指定的内存空间中，为了能够对这些参数进行查询就需要为他们建立一个表。通过这个表来索引这些参数，这个工作就是有函数 create_tables 来实现的。

```
/*
 * create_tables() parses the env- and arg-strings in new user
 * memory and creates the pointer tables from them, and puts their
 * addresses on the "stack", returning the new stack pointer value.
 */
static unsigned long * create_tables(char * p,int argc,int envc)
{
        unsigned long *argv,*envp;
        unsigned long * sp;
// 将当前的偏移值作为栈底，模拟压栈操作建立索引。
        sp = (unsigned long *) (0xfffffffc & (unsigned long) p);
// 为指向各个参数的指针分配空间，在分配的时候会多比时间参数多分一个用于作为
// 间隔
        sp -= envc+1;
        envp = sp;
        sp -= argc+1;
        argv = sp;
        put_fs_long((unsigned long)envp,--sp);
        put_fs_long((unsigned long)argv,--sp);
        put_fs_long((unsigned long)argc,--sp);
// 将参数的首地址存入刚刚分配的空间中
        while (argc-->0) {
                put_fs_long((unsigned long) p,argv++);
                while (get_fs_byte(p++)) /* nothing */;
        }
        put_fs_long(0,argv);
        while (envc-->0) {
                put_fs_long((unsigned long) p,envp++);
                while (get_fs_byte(p++)) /* nothing */;
        }
```

```
    put_fs_long(0,envp);
    return sp;
}
```

对于刚才的参数建立完表之后的分布如图：

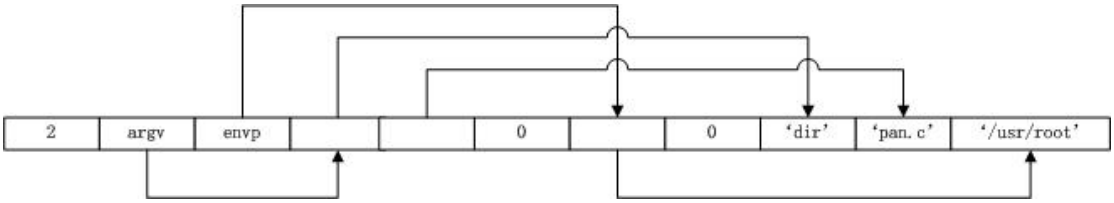


图 十二.2 参数在栈中的存储

函数 change_ldt

根据文件中给出的代码段长度 a_text 和参数表建立新的局部描述符表。

```
static unsigned long change_ldt(unsigned long text_size,unsigned long * page)
{
        unsigned long code_limit,data_limit,code_base,data_base;
        int i;
// 根据文件代码段长度求出代码段占用的长度，这是以 4k 为边界的。
        code_limit = text_size+PAGE_SIZE -1;
        code_limit &= 0xFFFFF000;
// 前面已经说过了，对于每一个进程来说它有一个虚拟的 64M 的空间，因此数据
// 段的长度是 64M
        data_limit = 0x4000000;
// 设置代码段的基地址就是当前进程（子进程）的基地址，
        code_base = get_base(current->ldt[1]);
        data_base = code_base;
// 重新设置局部描述符表
        set_base(current->ldt[1],code_base);
        set_limit(current->ldt[1],code_limit);
        set_base(current->ldt[2],data_base);
        set_limit(current->ldt[2],data_limit);
    /* make sure fs points to the NEW data segment */
        __asm__("pushl $0x17\n\tpop %%fs"::);
// 将参数放到这个虚拟空间的尾部，其实就是将虚拟空间映射到了参数表页
        data_base += data_limit;
        for (i=MAX_ARG_PAGES-1 ; i>=0 ; i--) {
            data_base -= PAGE_SIZE;
            if (page[i])
                put_page(page[i],data_base);
        }
```

```
// 将虚拟空间的尾部值作为返回值
        return data_limit;
    }
```

第十三章进程的终止

终止一个进程需要两个阶段，这是因为进程在结束的时候需要向它的父进程返回一些信息。为了能够让父进程得到这些信息，子进程在退出的时候会先释放所暂用的资源，然后通知父进程来完成剩余的工作。这些工作分别由系统调用 sys_exit 和 sys_waitpid 完成。

进程除了自己正常退出外，还可能是被其它进程强行中止。这个操作由系统调用 sys_kill 完成，但是它的具体实现还是利用了 sys_exit 的机制。

系统调用 sys_exit

它需要一个返回给父进程的值做参数，父进程仅仅接收参数最低的一个字节。具体的操作由函数 do_exit 完成。

<kernel/exit.c>

```
int sys_exit(int error_code)
{
// 获取参数最低的一个字节
    return do_exit((error_code&0xff)<<8);
}
```

do_exit

函数 do_exit 其实是一个不可能返回的函数，它会调用进程调度函数，并且在这之前会将自身的状态设置为 TASK_ZOMBIE，变成这个状态的进程就永远不会再被调用了。这个函数最主要的功能就是释放进程所占用的资源包括了内存的，文件的等等。

<kernel/exit.c>

```
int do_exit(long code)
{
    int i;
// 释放进程所占用的内存空间，这里没有包括 PCB 占用的 1 页空间，这 1 页空间由
// 父进程负责释放。
    free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
    free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
// 当一个进程消失了，它的子进程就成了孤儿，为了能够继续管理它们，就让它们的
// 老祖宗 1 号进程成为它们的父进程。另外，如果发现这些子进程中有状态是
// TASK_ZOMBIE 的，就说明这个子进程已经释放了自身占用的资源，需要由父进程来
// 完成剩余的操作。由于它的父进程已经消失了，因此就向它新的父进程－－1 号进程
// 发送信号 SIGCHILD 来激活。由 1 号进程完成剩余的工作。
    for (i=0 ; i<NR_TASKS ; i++)
        if (task[i] && task[i]->father == current->pid) {
            task[i]->father = 1;
```

```
                if (task[i]->state == TASK_ZOMBIE)
                    /* assumption task[1] is always init */
                    (void) send_sig(SIGCHLD, task[1], 1);
            }
    // 释放进程做占有的与文件有关的信息
        for (i=0 ; i<NR_OPEN ; i++)
            if (current->filp[i])
                sys_close(i);
        iput(current->pwd);
        current->pwd=NULL;
        iput(current->root);
        current->root=NULL;
        iput(current->executable);
        current->executable=NULL;
    // 如果占用了终端，这里也要释放
        if (current->leader && current->tty >= 0)
            tty_table[current->tty].pgrp = 0;
        if (last_task_used_math == current)
            last_task_used_math = NULL;
    // 如果进程是会话的头进程，这里就要释放以挂断其它以它为首的会话
        if (current->leader)
            kill_session();
    // 这只进程状态为 TASK_ZOMBIE，表示资源已经释放，等待父进程完成剩余的任务
        current->state = TASK_ZOMBIE;
    // 保存进程的给父进程的返回值
        current->exit_code = code;
    // 通知父进程去完成其它的工作
        tell_father(current->father);
    // 重新开始进程调度，以使父进程占有 CPU
        schedule();
    // 理论上是不可能执行到这里的，除非有 bug
        return (-1);    /* just to suppress warnings */
    }
```

相关函数

kill_session

用于挂断以当前进程为对话头部的进程。

<kernel/exit.c>

```
    static void kill_session(void)
    {
        struct task_struct **p = NR_TASKS + task;
```

```
    // 找到所有以当前进程为对话头的进程，向它们发送挂断信号 SIGHUP。
    // 处理这个信号的服务程序由系统服务程序提供
        while (--p > &FIRST_TASK) {
            if (*p && (*p)->session == current->session)
                (*p)->signal |= 1<<(SIGHUP-1);
        }
    }
```

tell_father

这个函数的工作仅仅是向父进程发送一个信号，好唤醒它来处理中止进程的剩余操作。

<kernel/exit.c>

```
    static void tell_father(int pid)
    {
        int i;

        if (pid)
    // 从 task 中找到父进程，然后将请求信号放入它的信号请求寄存器中。
            for (i=0;i<NR_TASKS;i++) {
                if (!task[i])
                    continue;
                if (task[i]->pid != pid)
                    continue;
                task[i]->signal |= (1<<(SIGCHLD-1));
                return;
            }
    /* if we don't find any fathers, we just release ourselves */
    /* This is not really OK. Must change it to make father 1 */
        printk("BAD BAD - no father found\n\r");
        release(current);
    }
```

系统调用 sys_waitpid

<kernel/exit.c>

```
    int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
    {
        int flag, code;
        struct task_struct ** p;

        verify_area(stat_addr,4);
    repeat:
    // 如果 flag 为 1 就表示接收到了信号，需要处理
```

```
        flag=0;
// 从 task 中寻找需要处理进程
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
// 忽略对自身的处理
                if (!*p || *p == current)
                        continue;
// 如果不是自己的子进程就忽略，注意这个函数仅仅会处理自身的子进程
                if ((*p)->father != current->pid)
                        continue;
// 以下是根据 pid 的情况来决定处理什么样的子进程。它没有显示的说明如果 pid=-1
// 会如何处理。实际上容易分析出来，如果 pid=-1。就会从进程表中选择出父进程对应
// 的所有子进程。

// 如果 pid>0 表示正在等待处理进程号为 pid 的子进程，不是这个号的子进程被忽略
                if (pid>0) {
                        if ((*p)->pid != pid)
                                continue;
// 如果 pid=0 表示要处理与父进程处于同一个进程组的子进程
                } else if (!pid) {
                        if ((*p)->pgrp != current->pgrp)
                                continue;
// 如果 pid<-1 表示要处理属于某一特定进程组的子进程，这个子进程的组号是 pid 的绝
// 对值。
                } else if (pid != -1) {
                        if ((*p)->pgrp != -pid)
                                continue;
                }
// 判断选择出来的这个子进程的状态，只可能是 TASK_STOPPED 或 TASK_ZOMBIE
// 这两个状态。
                switch ((*p)->state) {
                        case TASK_STOPPED:
// 对于 TASK_STOPPED 状态的子进程，如果 options 是 WUNTRACED 就忽略掉
                                if (!(options & WUNTRACED))
                                        continue;
// 如果没有忽略 TASK_STOPPED 状态的子进程，就将 0x7f 放入即将返回给调用程序
// 的变量 stat_addr。
                                put_fs_long(0x7f,stat_addr);
// 返回按照要求处理过的子进程，父进程会利用的这个值的。
                                return (*p)->pid;
                        case TASK_ZOMBIE:
// 计算子进程所占用的各类时间值，用户程序可能需要它
                                current->cutime += (*p)->utime;
                                current->cstime += (*p)->stime;
// 保存按照要求处理过的子进程号
```

```
                flag = (*p)->pid;
// 保存子进程的返回值
                code = (*p)->exit_code;
// 释放子进程的 PCB 表占用的这 1 页空间，这个函数中使用了进程调度。也许是为了
// 给其它的进程点儿工作的机会，万一其它的进程接收到信号了呢,Sys_waitpid 又使用
// 了这个久 CPU 了。由于没有改变当前进程的状态，因此不就它还会回来的。
                release(*p);
// 保存退出进程的返回值，给调用的进程
                put_fs_long(code,stat_addr);
                return flag;
            default:
// 设置 flag 为 1，表示找到过要求的进程，但是没有处理。
                flag=1;
                continue;
            }
        }
// 如果 flag 为 0 表示当前进程还没有任何子进程，这是不正常的。
        if (flag) {
// 如果 option 为 WNOHONG 表示没有发现子进程退出就返回，通常这种返回可能造成
// 死循环。以致程序挂起。
            if (options & WNOHANG)
                return 0;
// 设置当前进程为可中断休眠状态，这样其它的信号也可能将其唤醒
            current->state=TASK_INTERRUPTIBLE;
            schedule();
// 当苏醒之后，发现唤醒它的信号仅仅是 SIGCHLD 就会从头开始执行
            if (!(current->signal &= ~(1<<(SIGCHLD-1))))
                goto repeat;
// 如果发现还有其它的信号，就返回出错码
            else
                return -EINTR;
        }
// 没有找到子进程返回出错码
        return -ECHILD;
}
```

系统调用 sys_kill

系统调用 sys_kill 本质上将算作是信号的传递。它可以根据参数的情况向各类进程发送任何信号。如果对于除 SIGCHLD 以外的任何信号，默认的操作都是退出，因此可以起到消灭进程的作用。

<kernel/exit.c>

```
/*
 * XXX need to check permissions needed to send signals to process
```

```
 * groups, etc. etc.   kill() permissions semantics are tricky!
 */
// 如果 pid 值>0，则信号被发送给 pid。
// 如果 pid=0，那么信号就会被发送给当前进程的进程组中的所有进程。
// 如果 pid=-1，则信号 sig 就会发送给除第一个进程外的所有进程。
// 如果 pid < -1，则信号 sig 将发送给进程组-pid 的所有进程。
// 如果信号 sig 为 0，则不发送信号，但仍会进行错误检查。如果成功则返回 0。

int sys_kill(int pid,int sig)
{
	struct task_struct **p = NR_TASKS + task;
	int err, retval = 0;

	if (!pid) while (--p > &FIRST_TASK) {
		if (*p && (*p)->pgrp == current->pid)
			if (err=send_sig(sig,*p,1))
				retval = err;
	} else if (pid>0) while (--p > &FIRST_TASK) {
		if (*p && (*p)->pid == pid)
			if (err=send_sig(sig,*p,0))
				retval = err;
	} else if (pid == -1) while (--p > &FIRST_TASK)
		if (err = send_sig(sig,*p,0))
			retval = err;
	else while (--p > &FIRST_TASK)
		if (*p && (*p)->pgrp == -pid)
			if (err = send_sig(sig,*p,0))
				retval = err;
	return retval;
}
```

第十四章内存管理

Linux 对内存的管理是基于保护模式中的页式管理的。关于页式管理已经在引导部分讲解了，可以说与内存最主要的初始化是在引导的时候完成的。在引导的时候完成了从实模式到保护模式的转换。

为了能够有效的管理内存中的各个页，Linux 设置了一个数组叫做内存页映射表。表中的每一项代表了，内存中除系统空间之外的每页内存。数组中每一项的值表示相应的页的引用次数。当为 0 的时候，表示相应的页是空闲的。

由于使用了虚拟地址，因此 Linux 提供了从虚拟地址到实际地址的变化的操作。在创建进程的时候，子进程需要与父进程共享同一个内存空间。这个操作就是利用变换机制，将不同的虚拟地址映射到同一块内存区域中。

Linux 使用了写复制的方法来使用内存，提高了内存的利用率。因此，内存的异常处理在内存管理中也占有很重要的地位。进程的执行就是依靠它来实现的。

初始化：

在 Linux0.11 中仅仅使用 16M 内存，所以如果内存超过 16M 的话就按照 16M 计算。由于目前找到一个内存小于 16M 的计算机真困难，所以以后的分析中就当作内存都会超过 16M 来分析了。

Linux0.11 对这 16M 的内存划分如下表：

内核程序	高速缓存	虚拟盘	主内存

同样为了节省脑细胞，我就不考虑虚拟盘了，反正也没有什么用☺。

内核程序占用 1M 的地址空间，高速缓存占用随后的 3M 的地址空间，余下的 12M 地址空间就给主内存了。因此空间分配表如下：

起始地址	0X000000	0X100000	0X400000
分配情况	内核程序	高速缓存	主内存

对于内存的初始化从 main()函数(/init/main.c)中就开始了，开始时设置内存的尾地址。

memory_end = (1<<20) + (EXT_MEM_K<<10);

EXT_MEM_K 被定义为(*(unsigned short *)0x90002)。即取 0x90002 处的两个字节的数据(注意：在 VC 中的 short 类型是 4 个字节)。在 setup.s 中已经将从 BIOS 中的得到信息存放到从 0x90000 开始得内存中了，其中 0x90002 中存放得是系统从 1M 开始得扩展内存得数值，以 K 为单位。因此这个时候得到得内存大小是实际得大小，一会儿就会把它变小了。

```
memory_end &= 0xfffff000;
```

忽略其中不足 4k 字节得内存数这样便于计算页数

```
if (memory_end > 16*1024*1024)
    memory_end = 16*1024*1024;
if (memory_end > 12*1024*1024)
    buffer_memory_end = 4*1024*1024;
else if (memory_end > 6*1024*1024)
    buffer_memory_end = 2*1024*1024;
else
```

```
            buffer_memory_end = 1*1024*1024;
        main_memory_start = buffer_memory_end;
    #ifdef RAMDISK
        main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
    #endif
```

这个时候就得到了高速缓存和主内存得起始地址，同时将内存限制在 16M 以内。

```
        mem_init(main_memory_start,memory_end);
```

这个函数的实现放在了/mm/memory.c 中，这是内存初始化的主体。它的作用是将 main_memory_start 和 meory_end 之间的内存分页，放到内存映射表 mem_map[]中。代码如下：

```
    /*
     *  linux/mm/memory.c
     *
     *  (C) 1991  Linus Torvalds
     */
    void mem_init(long start_mem, long end_mem)
    {
        int i;

        HIGH_MEMORY = end_mem;
    // 根据宏定义#define PAGING_MEMORY (15*1024*1024)和
    // #define PAGING_PAGES (PAGING_MEMORY>>12)可以看出来 PAGING_PAGES
    // 是 15M 内存的页数。USED 是 100，用来标识该页已经被占用。这段代码的作
    // 用是将先将所有页标识为占用。
        for (i=0 ; i<PAGING_PAGES ; i++)
            mem_map[i] = USED;
    // MAP_NR(addr)定义为 (((addr)-LOW_MEM)>>12)用来计算 addr 的起始页号，用
    // 在这里是为了得到主存的起始页号。
        i = MAP_NR(start_mem);
        end_mem -= start_mem;
        end_mem >>= 12; // 计算主存的页数
        while (end_mem-->0)
            mem_map[i++]=0;       // 将主存这段的页面映射表清零
}
```

内存页映射表

由于系统空间（1M）是不允许使用的，因此内存页映射表仅仅对应着除系统空间以外的内存页。内存页映射表在程序中就分配了与内存页的数量相应的空间，因此表与内存页之间是一个线性的变换。公式是：页物理地址＝系统空间大小＋I×页大小。其中，I 是表中相应的索引号。

页的获取与释放

由于一个内存页是否可以使用完全由内存页映射表 mem_map[]来决定，因此对页的操作其实就是对内存页映射表的操作。

函数 get_free_page

```
/*
 * Get physical address of first (actually last :-) free page, and mark it
 * used. If no free pages left, return 0.
 */
// 这个函数从最后一页内存开始向前搜索，直到找到一页目前没有使用过的内存
// 将该内存页的首地址返回
// 参数：%1(ax=0); %2(LOW_MEM 非系统空间的首地址; %3(cx=内存页数);
//          %4(di=&mem_map[PAGING_PAGES-1])
// 返回值: 如果没有空闲的内存页就返回 0,否则返回相应页的首地址
unsigned long get_free_page(void)
{
register unsigned long __res asm("ax");
// 从最后一项开始向前搜索数组 mem_map,查找目前没有使用的项(值为 0)
// 由于这个函数的得到空闲页的地址的方法是从 mem_map 数组中找到未标识的项
// 标号，然后计算出来虚拟地址。这个虚拟地址一定是使用对应的是目录表中的前
// 4 个目录项的，又由于这 4 个目录项及其对应的页表是永远不会被程序改变的（因为
// 位于内核空间），因此这个虚拟地址是永远与内存物理地址一一对应的。就是说，
// 可以将由 get_free_page 函数返回的地址当做内存的唯一物理地址来使用。
__asm__("std ; repne ; scasb\n\t"
// 如果没有找到的话就返回,这个时候 eax 为 0
        "jne 1f\n\t"
        // 找到的话就设为 1 表示占用了,由于之前 edi 的值自动减 1 了,因此这里要加回去
        "movb $1,1(%%edi)\n\t"
        // 由于每页占用 4×1024 字节的内存因此对应该页的内存为该项号左移 12 位的值
        // 再加上用户空间的首地址
        "sall $12,%%ecx\n\t"
        "addl %2,%%ecx\n\t"
        // 清空该内存页
        "movl %%ecx,%%edx\n\t"
        "movl $1024,%%ecx\n\t"
        "leal 4092(%%edx),%%edi\n\t"
        "rep ; stosl\n\t"
        "movl %%edx,%%eax\n"
        "1:"
        :"=a" (__res)
        :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
```

```
        "D" (mem_map+PAGING_PAGES-1)
        :"di","cx","dx");
    return __res;
    }
```

函数 free_page

```
    /*
     * Free a page of memory at physical address 'addr'. Used by
     * 'free_page_tables()'
     */
    // 用于释放已经非配过的内存页，如果试图释放目前尚未分配的内存页会导致
    // 停机。它的实现很简单，就是在内存页映射表中将相应页的引用值减 1
    void free_page(unsigned long addr)
    {
        // 1M 一下使系统空间不能操作
        if (addr < LOW_MEM) return;
        // 超过了内存的范围，引起出错
        if (addr >= HIGH_MEMORY)
            panic("trying to free nonexistent page");
        // 得到该内存地址对应的内存页
        addr -= LOW_MEM;
        addr >>= 12;
        // 减少一次该页的引用数
        if (mem_map[addr]--) return;
        mem_map[addr]=0;
        panic("trying to free free page");
    }
```

对目录表的操作

对目录表的操作与对进程的操作很像，包括了生成新的目录项（拷贝已有的目录项），释放目录项。对目录表的这些操作仅仅在进程的创建和中止中使用了。它们的实现比较繁琐，需要慢慢分析，同时这里面涉及到了很多与保护模式下页式管理底层实现的细节，如果对此不熟请复习之后再看。

函数 copy_page_tables

正如 Linus 在这个函数前的注释所说的，它的确是我目前所见到的最为头痛的一个函数。Linus 在注释中写道：“but the memory management can be a bitch. See 'mm/mm.c': 'copy_page_tables()'”由于我的愤怒请允许我将 bitch 翻译成“婊子”。目前对于它的一些做法我还是想不通。幸好，只有 fork 这个函数调用了它。我先将我所理解的部分写出来。

Linux0.11 中最多允许 64 个进程，给每个进程的虚拟空间是 64M。因此，各个进程的空间间隔是 0x4000000 字节（64M）。这样求得进程空间的首地址就可以使用 nr×0x4000000 来得到。（nr 是该进程在进程表中对应的位置号）我认为这么做是很合理的，你可以计算一下 64×64M＝4G。Intel 允许使用的虚拟空间大小。

这个函数的实现思想其实很简单的，那就是先将起始地址对应的目录项和目标地址的目录项找到，然后求出欲移动的目录项数量。最后顺序的将起始目录项对应的页复制到目标目录项对应的页处。

```
int copy_page_tables(unsigned long from,unsigned long to,long size)
{
        unsigned long * from_page_table;        // 指向起始页表项的指针
        unsigned long * to_page_table;          // 指向目标页表项的指针
        unsigned long this_page;
        unsigned long * from_dir, * to_dir;     // 指向起始目录表项和目标目录表项的指针
        unsigned long nr;

        // 进程空间的地址必须是以 4M 为边界
        if ((from&0x3fffff) || (to&0x3fffff))
                panic("copy_page_tables called with wrong alignment");
        // 从线性地址中得到相应的目录表项的地址。
        // 由于目录表项的索引号位于线性地址的前 10 位，所以索引号是 addr>>22。
        // 又由于一项是 4 个字节所以对应的物理地址应该是(addr>>22)×4＝(addr>>20)
        // 与上 0xffc 是为了保证这个地址是 4 的倍数
        from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
        to_dir = (unsigned long *) ((to>>20) & 0xffc);
        // 计算要移动的内存数对应占用的目录项数量，加上 0x3fffff 是为了进位
        size = ((unsigned) (size+0x3fffff)) >> 22;
        // 开始循环的复制，由于在引导的时候已经将段界限设置为 16M，因此这里
        // size＝16。在初始化的时候，系统仅仅定义了 4 个目录项，如果复制了 16 项不
        // 就出错了吗？其实对于没有初始化过的目录项是不会复制的。
        for( ; size-->0 ; from_dir++,to_dir++) {
        // 正常情况下各个进程有其独有的目录项，因此如果发生向一个已经存在的
        // 目录项复制的时候就会出错
                if (1 & *to_dir)
                        panic("copy_page_tables: already exist");

        // 对于没有初始化过的目录项不执行复制操作
                if (!(1 & *from_dir))
                        continue;
        // 在目录项的高 20 位中存放的是目录项对应的页表的首地址
                from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
// 获得一个目前没有使用过的页的首地址，这个页地址就是目标目录项所指向的
// 页表的首地址。复制的操作就是从它开始的。页表的大小是 4k 正好一页大小
                if (!(to_page_table = (unsigned long *) get_free_page()))
                        return -1;      /* Out of memory, see freeing */
```

```
// 设置目标目录项使其指向新分配的页表，同时设置它的属性为用户态，可读、
// 写、执行，标识为已经占用。
            *to_dir = ((unsigned long) to_page_table) | 7;
        // 判断是否要复制内核的空间，对于内核空间仅仅复制 160 页（640k）
            nr = (from==0)?0xA0:1024;
        // 从起始页开始向目标页进行复制，通常一共会复制 1024 个页到目标区
        // 如果起始页没有使用就不复制，如果起始页处在用户空间中，还要到相应的
        // 内存页映射表中将相应页的引用次数加 1。由于发生了页的共享，因此这个时候
        // 不允许执行写操作。
            for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
        // 取得源页项的值
                this_page = *from_page_table;
        // 设置这个页写保护，如果向该页的空间执行写操作就会引发异常
                if (!(1 & this_page))
                    continue;
                this_page &= ~2;
                *to_page_table = this_page;
        // 如果发现被复制的页不在系统空间(1M)中，还要增加该页的引用次数。
        // 如果不释放的话，这就导致了再使用 fork 的时候无可用空间的假象
                if (this_page > LOW_MEM) {
                    *from_page_table = this_page;
                    this_page -= LOW_MEM;
                    this_page >>= 12;
                    mem_map[this_page]++;
                }
            }
        }
        invalidate();
        return 0;
}
```

函数 free_page_tables

有复制自然就有释放了，释放的实现相对于复制来说简单多了。

```
/*
 * This function frees a continuos block of page tables, as needed
 * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.
 */
int free_page_tables(unsigned long from,unsigned long size)
{
        unsigned long *pg_table;
        unsigned long * dir, nr;
// 还是判断是否是以 4M 边沿
        if (from & 0x3fffff)
```

```
            panic("free_page_tables called with wrong alignment");
// 由于内核空间对应的地址是 0x0000，因此如果发现 from 为 0 就是要释放内核空间
// 显然这是不可以的。
        if (!from)
            panic("Trying to free up swapper memory space");
// 得到段界限对应的目录项数，应该是 4 个。和目录项的首地址
        size = (size + 0x3fffff) >> 22;
        dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
        for ( ; size-->0 ; dir++) {
// 如果发现该目录项没有使用，就错过去
            if (!(1 & *dir))
                continue;
// 从目录项中的到相应的页表地址
            pg_table = (unsigned long *) (0xfffff000 & *dir);
// 将该目录项对应的 1024 个页释放
            for (nr=0 ; nr<1024 ; nr++) {
// 判断该页是否被使用过，如果使用过就去释放该页（其实就是减少该页对应的引用
// 次数）
                if (1 & *pg_table)
                    free_page(0xfffff000 & *pg_table);
// 将该页清空表明已经释放
                *pg_table = 0;
                pg_table++;
            }
// 释放该目录项从主存中取得的那一页。
            free_page(0xfffff000 & *dir);
// 将该目录项清空表明该目录项没有使用
            *dir = 0;
        }
        invalidate();
        return 0;
}
```

虚拟地址到物理地址的映射操作

由于在 Linux 中使用的地址都是虚拟地址，为了能够使得一个虚拟地址能够与物理地址形成相应的映射，Linux 使用了两个函数。Put_page 和 get_empty_page

函数 put_page

这个函数是用于将一个物理的内存页放到指定的虚拟地址中，其实准确的说应该是将一个物理的内存页与一个虚拟地址建立映射关系。这个映射的建立时很简单的，因为由虚拟地址到实际地址的变换是通过目录表和页表实现的，因此仅仅通过设置目录表和目录项就可以

实现了，无需进行整页的复制。

```
/*
 * This function puts a page in memory at the wanted address.
 * It returns the physical address of the page gotten, 0 if
 * out of memory (either when trying to access page-table or
 * page.)
 */
// page 是原页的物理地址，address 是欲存放该页的虚拟地址
// 函数将 page 值返回
unsigned long put_page(unsigned long page,unsigned long address)
{
        unsigned long tmp, *page_table;

/* NOTE !!! This uses the fact that _pg_dir=0 */
// 如果发现这个内存页地址位于内核空间或者已经超过了最大允许内存空间就会给
// 出提示
        if (page < LOW_MEM || page >= HIGH_MEMORY)
                printk("Trying to put page %p at %p\n",page,address);
// 如果发现这个物理页在内存页映射表中没有标记，既这个物理页还没有被分配或着
// 已经被释放掉了，就要给出提示信息。因为没有分配的内存页是会被覆盖掉的。
        if (mem_map[(page-LOW_MEM)>>12] != 1)
                printk("mem_map disagrees with %p at %p\n",page,address);
// 取得查找虚拟内存时所要使用的目录项地址
        page_table = (unsigned long *) ((address>>20) & 0xffc);
// 判断一下这个目录项是否已经使用过了，即是否已经指向了一个有效的页表首地址
        if ((*page_table)&1)
// 如果发现这个目录项已经有了自己的页表，让 page_table 指向目录项对应的页表的
// 首地址
                page_table = (unsigned long *) (0xfffff000 & *page_table);
        else {
// 如果发现这个目录项还没有使用，就未这个目录项分配页内存空间，使用这个 4k
// 的内存空间作为这个目录项对应的页表。然后让 page_table 指向这个页表的首地址
                if (!(tmp=get_free_page()))
                        return 0;
                *page_table = tmp|7;
                page_table = (unsigned long *) tmp;
        }
// 这个时候 page_table 已经指向了虚拟地址对应的页表首地址，然后让虚拟地址对应的
// 页项指向相应页的物理地址
        page_table[(address>>12) & 0x3ff] = page | 7;
/* no need for invalidate */
// 返回页的物理地址
        return page;
}
```

函数 get_empty_page

用于获取以虚拟地址 address 为首的一页内存。

```
void get_empty_page(unsigned long address)
{
    unsigned long tmp;

    if (!(tmp=get_free_page()) || !put_page(tmp,address)) {
        free_page(tmp);         /* 0 is ok - ignored */
        oom();
    }
}
```

异常处理

在内存管理中异常处理是很重要的部分，它是实现写时复制实现的基础。在进程创建的时候会将共享的空间设置为写保护，这样在出现写操作的时候就会引发异常。这个时候操作系统才会为其分配空间。另外，在程序执行的时候，由于要使用的虚拟内存还没有设定好，因此也会引发异常。这个时候，操作系统需要进行程序的加载。因此，在异常处理的时候需要分两种情况。

异常的进入与退出阶段

在进入阶段需要判断引发异常的起因，进而决定处理阶段由哪一个函数来完成。这部分的内容与中断系统中介绍的一样，可以参考那一章。

<mm/page.s>

```
_page_fault:
// 将出错码放入 eax 中
    xchgl %eax,(%esp)
    pushl %ecx
    pushl %edx
    push %ds
    push %es
    push %fs
    movl $0x10,%edx
    mov %dx,%ds
    mov %dx,%es
    mov %dx,%fs
// 将引起出错的虚拟地址放入 edx 中
    movl %cr2,%edx
// 作为参数压栈
    pushl %edx
```

```
        pushl %eax
// 如果出错码是 1，表示是由于缺页引发的异常
// 如果不是 1，表示是由于写保护引发的异常
        testl $1,%eax
        jne 1f
        call _do_no_page
        jmp 2f
1:      call _do_wp_page
// 还原环境变量
2:      addl $8,%esp
        pop %fs
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret
```

处理阶段

函数 do_no_page

通常是在需要加载程序到内存的时候才引发这个操作的。

```
void do_no_page(unsigned long error_code,unsigned long address)
{
        int nr[4];
        unsigned long tmp;
        unsigned long page;
        int block,i;
// 获取虚拟地址所在页面的首地址
        address &= 0xfffff000;
// 计算虚拟地址在进程空间中的偏移地址
        tmp = address - current->start_code;
// 如果偏移地址超过了程序的数据段，表示程序已经全部加载了但是需要额外的空间
// 如果 executable 为 Null 表示刚刚调用了 fork，还没有调用 execve。
// 对于以上 2 种情况不需要加载文件，只需要在内存中获取一页空间
        if (!current->executable || tmp >= current->end_data) {
                get_empty_page(address);
                return;
        }
// 为了节省空间，进程会去寻找一个使用了同一个可执行文件的进程。如果找到了当
// 前进程与找到的进程共享同一块内存空间
        if (share_page(tmp))
```

```
            return;
    // 如果没有找到可以共享的，就为这个进程分配一页物理内存
        if (!(page = get_free_page()))
            oom();
    // 计算需要加载的大小
    /* remember that 1 block is used for header */
        block = 1 + tmp/BLOCK_SIZE;
        for (i=0 ; i<4 ; block++,i++)
            nr[i] = bmap(current->executable,block);
        bread_page(page,current->executable->i_dev,nr);
    // 如果新分配一页空间之后，可用的内存超过了文件大小。需要将多余的内存清零
        i = tmp + 4096 - current->end_data;
        tmp = page + 4096;
        while (i-- > 0) {
            tmp--;
            *(char *)tmp = 0;
        }
    // 建立虚拟地址到物理地址的映射
        if (put_page(page,address))
            return;
        free_page(page);
        oom();
    }
```

共享内存

只有一个进程已经加载了可执行文件到内存中，其它的进程才可以共享。这种方法是很有用的，如果一个可执行文件由两个进程来执行，因为执行的时候仅仅是读操作，没有必要再为这个文件重新分配内存。

函数 share_page

<mm/memory.c>

```
    /*
     * share_page() tries to find a process that could share a page with
     * the current one. Address is the address of the wanted page relative
     * to the current data space.
     *
     * We first check if it is at all feasible by checking executable->i_count.
     * It should be >1 if there are other tasks sharing this inode.
     */
    static int share_page(unsigned long address)
    {
        struct task_struct ** p;
    // 如果 executable 为空，表明没有进程使用这个文件，因此无法共享
        if (!current->executable)
```

```
            return 0;
    // 如果这个文件的引用次数小于 2 表示只有一个进程使用这个文件，无法共享
        if (current->executable->i_count < 2)
            return 0;
    // 从进程组中将与当前进程使用同一个文件的进程找到
        for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
            if (!*p)
                continue;
            if (current == *p)
                continue;
            if ((*p)->executable != current->executable)
                continue;
    // 与这个进程共享一页内存
            if (try_to_share(address,*p))
                return 1;
        }
        return 0;
    }
```

函数 try_to_share

<mm/memory.c>

```
    /*
     * try_to_share() checks the page at address "address" in the task "p",
     * to see if it exists, and if it is clean. If so, share it with the current
     * task.
     *
     * NOTE! This assumes we have checked that p != current, and that they
     * share the same executable.
     */
    static int try_to_share(unsigned long address, struct task_struct * p)
    {
        unsigned long from;
        unsigned long to;
        unsigned long from_page;
        unsigned long to_page;
        unsigned long phys_addr;
    // 获取内存相对偏移地址的目录项地址
        from_page = to_page = ((address>>20) & 0xffc);

    // 虚拟地址对应的目录项＝偏移地址的目录项地址＋起始地址的目录项地址
        from_page += ((p->start_code>>20) & 0xffc);
        to_page += ((current->start_code>>20) & 0xffc);
    /* is there a page-directory at from? */
    // 取页目录项内容。如果该目录项无效(P=0)，则返回。否则取该目录项对应页表地址
        from = *(unsigned long *) from_page;
```

```
        if (!(from & 1))
                return 0;
        from &= 0xfffff000;
// 计算地址对应的页表项指针值，并取出该页表项内容
        from_page = from + ((address>>10) & 0xffc);
        phys_addr = *(unsigned long *) from_page;
/* is the page clean and present? */
// 0x41 对应页表项中的 Dirty 和 Present 标志。如果页面不干净或无效则返回
        if ((phys_addr & 0x41) != 0x01)
                return 0;
// 取页面的地址。如果该页面地址不存在或小于内存低端(1M)也返回退出
        phys_addr &= 0xfffff000;
        if (phys_addr >= HIGH_MEMORY || phys_addr < LOW_MEM)
                return 0;
// 取页目录项内容。如果该目录项无效(P=0)，则取空闲页面，并更新 to_page 所指
// 的目录项
        to = *(unsigned long *) to_page;
        if (!(to & 1))
                if (to = get_free_page())
                        *(unsigned long *) to_page = to | 7;
                else
                        oom();
// 取页表地址。如果对应的页面已经存在，则出错，死机
        to &= 0xfffff000;
        to_page = to + ((address>>10) & 0xffc);
        if (1 & *(unsigned long *) to_page)
                panic("try_to_share: to_page already exists");
/* share them: write-protect */
// 对 p 进程中页面置写保护标志(置 R/W=0 只读)。并且当前进程中的对应页表项指向
// 它
        *(unsigned long *) from_page &= ~2;
        *(unsigned long *) to_page = *(unsigned long *) from_page;
        invalidate();
// 计算所操作页面的页面号，并将对应页面映射数组项中的引用递增 1
        phys_addr -= LOW_MEM;
        phys_addr >>= 12;
        mem_map[phys_addr]++;
        return 1;
}
```

函数 do_wp_page

这个函数用于执行写时复制这个操作，当有多个进程共享一段内存空间的时候，会将这

个空间设置为写保护。如果任何一个进程试图向这个空间执行写操作的时候都会引发异常。由于这个原因导致的异常会调用这个函数。这个函数会为共享的内存页重新分配一页空间，将共享页的数据复制到新页中，同时设置页表项指向新页地址。

```
void un_wp_page(unsigned long * table_entry)
{
	unsigned long old_page,new_page;

	old_page = 0xfffff000 & *table_entry;
	if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
		*table_entry |= 2;
		invalidate();
		return;
	}
	if (!(new_page=get_free_page()))
		oom();
	if (old_page >= LOW_MEM)
		mem_map[MAP_NR(old_page)]--;
	*table_entry = new_page | 7;
	invalidate();
	copy_page(old_page,new_page);
}

/*
 * This routine handles present pages, when users try to write
 * to a shared page. It is done by copying the page to a new address
 * and decrementing the shared-page counter for the old page.
 *
 * If it's in code space we exit with a segment error.
 */
void do_wp_page(unsigned long error_code,unsigned long address)
{
#if 0
/* we cannot do this yet: the estdio library writes to code space */
/* stupid, stupid. I really want the libc.a from GNU */
	if (CODE_SPACE(address))
		do_exit(SIGSEGV);
#endif
	un_wp_page((unsigned long *)
		(((address>>10) & 0xffc) + (0xfffff000 &
		*((unsigned long *) ((address>>20) &0xffc)))));

}
```

第十五章操作系统的运行

如果看完了进程系统可以说对操作系统就会有一个相对完整的理解了。但是由于内容比较烦杂，所以我准备在这一章以尽量简短篇幅来描述一下 Linux0.11 操作系统是如何指导计算机工作的。

准备工作

Linux0.11 系统很小，用一张软盘就可以装下。将 Linux0.11 的源代码编译成计算机可以识别的二进制文件，将这个文件直接写入到软盘的第 0 面第 0 道第 1 物理扇区的 0 号偏移地址。注意：这个操作可不是一般的复制粘贴就可以完成的，需要由相应的软件完成，例如 rawrite.exer（也可以自己一个利用 c 的库函数 abswrite 在 Windows98 下就可以完成，但是这个函数在 2000 下不能用）。

好了，现在这张软盘就是系统盘了。用这张软盘就可以引导计算机工作了。但是这张软盘只有操作系统，而操作系统还需要读取一些系统服务程序，这就要用到文件系统了。因此需要再准备一张软盘，仿照系统盘的处理方法将文件系统的映射文件复制到这张软盘上。文件系统的映射文件可以从网上找到 oldlinux.org。以后就叫这张盘为文件系统盘了。

为了能够用这两张盘协调工作需要修改一下 Linux0.11 的引导代码，在文件 bootsec.s 中添加一个用于对文件系统盘测试的代码段，当将系统文件全部加载到内存之后给出提示，要求插入文件系统盘，然后再继续工作。

现在，有了这两张软盘，就可以带走 Linux0.11 操作系统了。

内核的工作

将系统盘插入软驱，在 BIOS 中将引导盘改为软盘，重新启动计算机。

计算机启动后，读取软驱的第一扇区到内存中（bootsec.s），然后执行这 512 字节的代码。Bootsec 执行的时候从系统盘中读取 setup.s 和系统内核对应的代码到内存中，然后给出提示，要求插入文件系统盘。插入了文件系统盘后，转入 setup.s 处执行。Setup.s 利用 BIOS 中断获取一些计算机的参数放到指定的内存中，然后初始化 8259A 芯片，将内存设置为保护模式，开启段式管理。最后转入系统内核的代码执行。

系统内核代码的头部是 head.s 对应的程序，于是 head.s 开始工作。Head.s 重新设置全局描述符表（GDT）和中断向量表（IDT），做一些检测工作。然后设置页目录表，页表，开启页式管理。以后，操作系统就开始在保护模式页式管理下工作了。最后，转入 init.c 中的 main 函数处执行。

Main 函数首先初始化内存页映射表，设定中断向量，初始化底层的块设备和终端，获取计算机与时间有关的信息，初始化进程表，开启时钟中断，初始化缓冲区，初始化硬盘和软驱。当这些工作完成后，开启中断，设定系统处于用户模式。生成 1 号进程运行 init 函数，0 号进程开始无限的循环等待。后面的工作就由 init 函数完成了。

函数 init 首先为标准输入输出做一些准备工作，然后产生一个子进程 2，这个子进程会从文件系统盘中找到文件 rc，并以只读的形式打开它。接着它运行程序 sh，然后 1 号进程

处于等待直到进程 2 退出。这个 sh 运行后，就是我们比较熟悉的用户界面了。它提供了一些应用程序，用户利用这些应用程序完成对操作系统的操作。而这些应用程序就是通过系统中断才与操作系统交流。因此，系统中断是程序的接口不是用户的操作接口。

在 Linux0.11 中，系统的启动是不是很容易。复杂的是内部资源的管理。为了便于理解一个用户程序是如何开始的，我再讲一讲程序的三步曲：创建、执行、死亡。

程序的一生

Linux 中有一个初始进程，0 号进程，是写到程序中的，main 函数就相当于运行在 0 号进程中。0 号进程中维护 4 个目录项对应于 Linux0.11 所能够支持的 16M 内存。

当在 main 函数中调用函数 fork 为 init 函数创建一个进程的时候，这是第一次调用 fork 函数。它创建了 1 号进程。由于，0 号进程所对应的段界限是 640k，因此 1 号进程只复制了 0 号进程的一个目录项，同时与 0 号进程共同执行 main 函数。由于对于 1 号进程返回值是 0，对于 0 号进程返回值是 1。于是，1 号进程开始执行 init 函数，0 号进程会永远的循环等待下去。1 号进程拥有的虚拟空间是 64M，段界限是 16M（与 0 号进程不同了）。

1 号进程调用 fork 创建进程 2，进程 2 复制了 1 号进程占有的 1 个目录项（最多可以拷贝 4 个，但是有 3 个目录项是空的，所以没有拷贝）。进程 2 利用 execve 执行文件 sh，execve 首先释放了刚刚设置的目录项，设定程序运行环境，设置 EIP。这样，当 execve 返回的时候会到 eip 指定的位置。由于没有为这个虚拟地址分配目录项及页表，因此这个虚拟地址会引发异常，内存异常处理程序会根据情况来加载可执行文件到内存中，于是 sh 开始运行。当 sh 退出的时候，调用 exit。Exit 会释放进程 2 占有的各种资源资源，同时会将自身的状态设置为僵死，并向 1 号进程发送信号。进程 1 将这个僵死的 2 号进程找到，获取返回值并清除进程 2 占有的 1 页内存。

参考文献

这本书对 Linux0.11 进行了非常详细的注释，没有它我是无法写这本书的

【1】 赵炯，Linux0.11 版本完全分析，2003

对 386 下的保护模式进行了详细的讲解

【2】 周明德，保护方式下的 80386 及其编程，清华大学出版社，1993

这是对 Linux2.*进行的分析，在 Linux 书籍中，可以说是最经典的之一

【3】 Daniel P. Bovet, Marco Cesati，Understanding the Linux Kernel, 2nd Edition，O'Reilly，2002

两位老师的文笔很好，写得生动形象。按照功能对 Linux2.*进行分析

【4】 毛德操、胡希明，Linux 内核源代码情景分析，浙江大学出版社，2001

这是一个仅仅对 Linux2.*得进程管理进行分析得书，有关进程方面得介绍很详细

【5】 John O'Gorman，The Linux Process Manager The internals of scheduling, interrupts and signals，2003

勿庸置疑这是个名著，Minix 是为了配合它而写的

【6】 Andrew S.Tanenbaum、Albert S.Woodhull 著， 王鹏、尤晋元、朱鹏、敖青云译，操作系统：设计与实现，电子工业出版社，1998

有关接口和中断方面的知识都可以从这里找到

【7】 Barry B.Brey，intel 系列微处理器结构、编程和接口技术大全—80x86、pentium 和 pentium pro，机械工业出版社，1998

学习汇编语言优秀书籍，经典

【8】 Randall Hyde，The Art of Assembly Language Programming，No Starch Press

看了这篇文章之后，你会在最短的时间内对操作系统有个了解

【9】 Krishnakumar R.，Writing Your Own Toy OS，2002

这是一个好人写的文章，对 GCC 与汇编的嵌套写的很详细，建议现看看

【10】 chforest_chang，AT&T 汇编语言与 GCC 内嵌汇编简介，2004

在知道《Linux0.11 版本完全分析》之前，我就靠它了。也是对 Linux2.*进行分析的。

【11】 joyfire linux 笔记，http://www.joyfire.net/

我们的教材，也是很不错的

【12】 齐志儒、高福祥，汇编语言程序设计，东北大学出版社，2001

高二时买的，以为没有用呢。没想到，在分析的时候还挺有帮助的。

【13】 徐金梧、徐科，TurboC 实用大全，机械出版社，1996

如果相对系统的引导有了解，就看它吧

【14】 谢煜波，操作系统引导探究，2004

学习接口时用的教材

【15】 高福祥、张君，接口技术，东北大学出版社，1999