**head.s 分析——Linux-0.11 学习笔记（三）**

题目：head.s 分析

head.s 程序在被编译生成目标文件后会与内核其他程序一起被链接成 system 模块，它位于 system 模块的最开始部分，这也就是为什么称其为“头部(head)”程序的原因。

从这里开始，内核完全是在保护模式下运行了。head.s 汇编程序与前面的语法格式不同，它采用的是AT&T汇编语言格式，并且需要使用 GNU 的 as 和 ld 进行编译和连接。因此请注意代码中赋值的方向是从左到右。

这段程序实际上处于内存地址0处，在理解代码的时候，请务必记住。

一、加载段寄存器

.text

.globl idt,gdt,pg\_dir,tmp\_floppy\_area

pg\_dir: # 页目录将会存放在这里，把这里的代码覆盖掉

.globl startup\_32

startup\_32:

movl $0x10,%eax

mov %ax,%ds

mov %ax,%es

mov %ax,%fs

mov %ax,%gs

lss stack\_start,%esp

第6行：0x10 是数据段的选择子，在 setup.s 文件的末尾处定义，基地址是 0，段界限是 0x7FF，粒度 4KB，可读可写，向上扩展。如果读者忘了，可以参考我的博文：setup.s 分析

7~10行：令ds,es,fs,gs指向数据段。

第11行：stack\_start 的定义在 kernel/sched.c（以后会分析）中。

为了阅读方便，截取部分代码在这里。

// kernel/sched.c

long user\_stack [ PAGE\_SIZE>>2 ] ;

struct {

long \* a;

short b;

} stack\_start = { & user\_stack [PAGE\_SIZE>>2] , 0x10 };

LSS指令

lss指令的格式是

LSS r32,m16:32

含义是用内存中的长指针加载 SS:r32

Load SS:r32 with far pointer from memory

m16:32表示一个内存操作数，这个操作数是一个长指针，由2部分组成：16位的段选择子和32位的偏移。

A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer’s segment selector. The number to the right corresponds to its offset.

注意，长指针在内存中的布局如下：低4字节是偏移，高2字节是段选择子。



stack\_start 处的6字节是long \* a和short b.

a被赋值为user\_stack[]数组最末端的地址，b被赋值为0x10.

所以，第11行代码表示用a的值加载ESP，用b的值加载SS，即栈的初始化。

二、设置中断描述符表（IDT）

call setup\_idt

1

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 # 取idt的偏移给edi

mov $256,%ecx # 循环256次

rp\_sidt:

movl %eax,(%edi) # eax -> [edi]

movl %edx,4(%edi) # edx -> [edi+4]

addl $8,%edi # edi + 8 -> edi

dec %ecx

jne rp\_sidt

lidt idt\_descr # 加载IDTR

ret

...

idt\_descr:

.word 256\*8-1 # idt contains 256 entries

.long idt # IDT 的线性基地址

...

idt:

.fill 256,8,0 # idt is uninitialized

IDT 共 256 项，作者使各个表项均指向一个只报错误的哑中断子程序ignore\_int。

2~5行：组装中断门，示意图如下，蓝色圆圈是行号。





2~5行：在edx、eax中组合设置出8字节默认的中断描述符值。eax 含有描述符低4字节，edx 含有高4字节。

9~14行：在idt表每一项中都放置该描述符，共 256 项。内核在随后的初始化过程中会替换那些真正使用的中断描述符项。

中断处理过程 ignore\_int

/\* This is the default interrupt "handler" :-) \*/

int\_msg:

.asciz "Unknown interrupt\n\r"

.align 2

ignore\_int:

pushl %eax

pushl %ecx

pushl %edx

push %ds # 这里请注意ds，es，fs，gs等虽然是16位的寄存器，

# 但仍然会以32位的形式入栈，即需要占用4个字节的栈空间。

push %es

push %fs # 以上用于保存寄存器

movl $0x10,%eax # 0x10是数据段选择子

mov %ax,%ds

mov %ax,%es

mov %ax,%fs # ds,es,fs均指向数据段

pushl $int\_msg

call printk # 该函数在 kernel/printk.c 中

popl %eax # 清理参数

pop %fs

pop %es

pop %ds

popl %edx

popl %ecx

popl %eax

iret

第17行：把 printk 函数的参数入栈。注意：若符号 int\_msg 前不加 $，则表示把 int\_msg 符号处的双字Unkn入栈。

第18行：调用 printk 函数，该函数在 kernel/printk.c 中，以后再具体分析。

第19行：清理参数 $int\_msg.

说明：汇编程序调用C函数时，函数的入口参数使用栈来传送,参数的传递顺序是从右到左。调用者负责清除参数占用的栈空间。C函数的返回值如果是32位整数，则保存在eax寄存器；如果是64位整数，则保存在edx:eax寄存器。

具体可以参考我的博文： 在汇编程序中调用C函数

三、设置全局描述符表（GDT），加载 GDTR

call setup\_gdt

1

setup\_gdt:

lgdt gdt\_descr # 加载GDTR

ret

LGDT指令的格式是：

LGDT m16&32

该指令的操作数是一个 48 位（6字节）的内存区域。在这6字节的内存区域中，要求前（低）16位是 GDT 的界限值，后（高）32 位是 GDT 的基地址 。该指令在实模式和保护模式下都可以执行。

gdt\_descr:

.word 256\*8-1

.long gdt

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 \*/

GDT 定义的描述符如下：

索引 选择子 段类型 Limit G DPL 其他

0 - - - - - -

1 0x08 代码段 0xFFF 1（表示4KB） 0 非一致，可读

2 0x10 数据段 0xFFF 1（表示4KB） 0 向上扩展，可写

段长度可以这样算：（Limit + 1）\* 4KB = (0xFFF + 1) \* 4KB = 0x1000 \* 4KB = 16MB

四、重新加载段寄存器

call setup\_idt

call setup\_gdt

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

由于段描述符中的段限长（Limit）从setup.s中的 8MB 改成了本程序设置的 16MB，因此这里再次对所有段寄存器执行加载操作是必须的。另外，如果不对 CS 再次加载，那么在执行到第1行时，CS段寄存器的“描述符高速缓存器”中的段限长还是 8MB。这样看来应该重新加载CS。

但是由于 setup.s 中的代码段描述符与本程序中重新设置的代码段描述符除了段限长以外其余部分完全一样，8MB 的限长在内核初始化阶段不会有问题，而且在以后内核执行段间跳转时会重新加载 CS，因此这里没有加载它并没有让程序出错。

针对该问题，目前内核中就在movl $0x10,%eax之前添加了一条长跳转指令ljmp $(\_KERNEL\_CS), $1f，大概的代码如下：

call setup\_idt

call setup\_gdt

# reload all the segment registers

ljmp $(\_KERNEL\_CS), $1f

1:

movl $0x10,%eax

mov %ax,%ds

mov %ax,%es

mov %ax,%fs

mov %ax,%gs

lss stack\_start,%esp

注意：以上的代码只是为了说明问题，并非源码。

ljmp $(\_KERNEL\_CS), $1f

$1f中的1是标号，紧跟在其后的f表示向前（forwards）。这条指令会跳转到第6行来确保 CS 被重新加载。

五、检测A20是否开启

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

用于测试 A20 地址线是否已开启。采用的方法是向内存地址 0x0\_0000 处写入任意一个数值，然后看内存地址 Ox10\_0000(1M)处是否也是这个数值。如果一直相同的话，就一直比较下去。死机表示 A20 线没有选通。

六、检测x87协处理器

为了弥补x86系列在进行浮点运算时的不足，Intel于1980年推出了x87系列数学协处理器，那时x87是一个外置的、可选的芯片。1989年，Intel发布了486处理器。从486开始，以后的CPU一般都内置了协处理器。这样，对于486以前的计算机而言，操作系统检测x87协处理器是否存在就非常必要了。

注：1991年，一名21岁的就读于芬兰赫尔辛基大学的计算机科学专业学生—— Linus Torvalds 基于 gcc、bash 开发了针对 386 机器的 Liniux内核。

下面这段程序用于检查数学协处理器芯片是否存在。方法是修改控制寄存器CR0,在假设协处理器存在的情况下执行一个协处理器指令，如果出错的话则说明协处理器芯片不存在。

movl %cr0,%eax # check math chip

andl $0x80000011,%eax # Save PE ET PG

orl $2,%eax # set MP=1

movl %eax,%cr0

call check\_x87

jmp after\_page\_tables

第2行：保留PE、ET、PG位，其他位都清零。

PE 指示是否开启保护模式。PG 指示是否开启分页。

关于ET，Intel 手册如是说：

Extension Type (bit 4 of CR0). Reserved in the P6 family and Pentium ® processors. (In the P6 family processors, this flag is hardcoded to 1.) In the Intel 386™ and Intel 486™ processors, this flag indicates support of Intel 387 DX math coprocessor instructions when set.

第3行，设置MP=1.

这块我不是很明白，根据下面的表格，在数学协处理器存在的时候，推荐设置EM=0，MP=1.

既然作者的意图是假设数学协处理器存在，那么就设置 EM=0，MP=1 吧。



check\_x87:

fninit # 向协处理器发出初始化命令

fstsw %ax # 把FPU的状态字保存到AX中

# 初始化后状态字应该为0,否则说明协处理器不存在

cmpb $0,%al

je 1f # 存在则跳转到标号1处

movl %cr0,%eax

xorl $6,%eax # 把 eax 的值和 0110b 异或

movl %eax,%cr0

ret

.align 2

1: .byte 0xDB,0xE4 /\* fsetpm for 287, ignored by 387 \*/

ret

第12行：0xDB,0xE4这两个字节是 80287 协处理器指令 fsetpm 的机器码。其作用是把 80287 设置为保护模式。80387 无需该指令，它会把该指令看作是空操作。

关于异或

按位异或的3个特点

0异或任何数 ＝ 任何数

1异或任何数 = 任何数取反

任何数异或自己 = 把自己置0

按位异或的几个常见用途

1. 使某些特定的位翻转

​ 例如要使 EAX 的 b1 位和 b2 位翻转：

　　　　　 EAX = EAX ^ 00000110

​ 代码第8行就是这种用法，把 EM 和 MP 翻转。

2. 不使用临时变量就可以实现两个值的交换

​ 例如 a=11110000，b=00001111，要交换a、b的值，可通过下列语句实现：

a = a^b； 　　//a=11111111

b = b^a； 　　//b=11110000

a = a^b； 　　//a=00001111

3. 在汇编语言中经常用于将变量置零

​ xor eax，eax

4. 快速判断两个值是否相等

​ 例如判断两个整数a、b是否相等，可通过下列语句实现：

​ return ((a ^ b) == 0)；

关于.align

.align是汇编语言指示符。其含义是边界对齐调整。”2”表示把随后的代码或数据的偏移位置调整到地址值最后 2 比特位为零的位置，即按 4（=2^2)字节方式对齐内存地址。不过现在 GNU as 直接写出对齐的值而非 2 的幂次。使用该指示符的目的是为了提高 32 位 CPU 访问内存中代码或数据的效率。

七、开启分页，跳转到 main()

Linus 将内核的页表直接放在页目录之后，使用了4个页表来寻址16MB的物理内存。如果你有多于16MB的内存，就需要在这里进行扩充修改。关于分页机制，说来话长，不了解的朋友可以参考我的博文：

x86分页机制

Linus 在物理地址0x0处开始存放1页页目录和4页页表。页目录是系统所有进程公用的，而其后的4页页表则属于内核专用，它们把线性地址 0x000000~0xFFFFFF 一一映射到物理地址 0x000000~0xFFFFFF。

.org 0x1000 #从偏移 0x1000 处开始放第1个页表（偏移0开始处将存放页目录）

pg0:

.org 0x2000 #从偏移 0x2000 处开始放第2个页表

pg1:

.org 0x3000 #从偏移 0x3000 处开始放第3个页表

pg2:

.org 0x4000 #从偏移 0x4000 处开始放第4个页表

pg3:

.org 0x5000 #定义下面的内存数据块从偏移 0x5000 处开始

.ORG伪指令用来表示起始的偏移地址，紧接着ORG的数值就是偏移地址的起始值。ORG伪操作常用来指定数据的存储地址，有时也用来指定代码段的起始地址。更详细的解释可以参考我的博文：

ORG 伪指令

tmp\_floppy\_area:

.fill 1024,1,0 #共保留1024项，每项1字节，填充数值0

fill伪指令的格式是 .fill repeat,size,value

表示产生 repeat 个大小为 size 字节的重复拷贝。size 最大是 8，size 字节的值是 value.

“当 DMA (直接存储器访问）不能访问缓冲块时，tmp\_floppy\_area 内存块就可供软盘驱动程序使用。其地址需要对齐，这样就不会跨越 64KB 边界。”

这是书上的话，我不甚理解。暂不深究，以后再说。

为调用main()函数做准备

after\_page\_tables:

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 # 设置页目录和页表，并开启分页

L6:

jmp L6 # main should never return here, but

# just in case, we know what happens.

2~6行：为跳转到 init/main.c 中的 main() 函数作准备工作。

2~4行：前3个入栈 0 值应该分别表示 envp、argv 指针和 argc 的值，但 main() 没有用到。

第5行：压入返回地址。模拟调用（其实是使用JMP指令） main.c 程序时首先将返回地址入栈的操作，如果 main.c 程序真的退出，就会返回到标号 L6 处继续执行下去，即死循环。

第6行：压入 main() 函数代码的地址。当后面执行 ret 指令时，就会弹出 main() 的地址，并把控制权转移到 init/main.c 程序中。

依然可以参考我的那篇博文： 在汇编程序中调用C函数

设置页目录和页表

setup\_paging:

movl $1024\*5,%ecx # 每个页表占用1024个双字（4B），共5个页表

xorl %eax,%eax # eax = 0

xorl %edi,%edi # edi = 0

cld

rep;stosl # eax -> es:[edi],edi每次增加4，重复ecx次

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 \*/

2~5行，把页目录和页表清零。

stosl：Store EAX at address ES:EDI

7~10行，填写页目录表的前4项。关于表项的格式，可以参考我的博文 页目录项和页表项



位 名称 含义

0 P 存在位。为1表示页表或者页位于内存中，为0表示不在内存中，必须先予以创建或者从磁盘调入内存后方可使用。

1 R/W 读写标志。为1表示页面可以被读写，为0表示只读。当处理器运行在0、1、2特权级时，此位不起作用。页目录中的这个位对其所映射的所有页面起作用。

2 U/S 用户/超级用户标志。为1时，允许所有特权级别的程序访问；为0时，仅允许特权级为0、1、2的程序访问。页目录中的这个位对其所映射的所有页面起作用。

所以，根据上表，可以知道7~10行中的“+7”表示：页表存在，可读可写，允许所有特权级别的程序访问。

填写后示意图如下：



movl $pg3+4092,%edi

movl $0xfff007,%eax

std # 设置方向位DF=1

1: stosl # Store EAX at address ES:EDI

subl $0x1000,%eax # 更新下一个表项的值，因为一个表项对应 0x1000B 的内存，所以要把页基址减去0x1000

jge 1b # 用于有符号数大小的比较，eax 大于等于 0x1000 则跳转到1处

以上代码的目的是填写4个页表。

页表项的格式如下图，0、1、2比特位的含义见前文的表格。



movl $pg3+4092,%edi

一张页表最多可以容纳1024个表项，每项占4个字节。下图左边是表项的序号，从0到1023，右边是偏移地址（= 序号\*4）,4092是最后一个表项的偏移地址。

上面的代码表示把页表3（最后一个页表）的最后一项的地址传入edi. 作者的意图是从最后一个表项开始，倒着填写，直到填完页表0的第0个表项。



movl $0xfff007,%eax中的0xfff007是页表3的最后一项的值，“7”就不用再解释了，解释一下为什么是0xfff000：

页表的每一项对应4KB（2^12=4K）的内存，一个页表有1024(=1K)项，共对应4KB\*1K=4MB的内存。代码中安排了4个页表，即共可以映射4\*4MB=16M内存。

16M - 4K = 0xFFF000

或者：

16M - 1 = 0x1000000-1 = 0xFFFFFF

0xFFFFFF & 0xFFFFF000 = 0xFFF000

jge 用于有符号数大小的比较，当 DEST(这里是eax) 大于等于 SRC(这里是0x1000) 则跳转。当 eax=0x1007时，eax>=0x1000，跳转之后 eax=0x0007，这时候条件不再成立，则结束跳转。所以，最后填写的表项值是 0x0007。

xorl %eax,%eax /\* pg\_dir is at 0x0000 \*/

movl %eax,%cr3 # 把页目录的物理地址写入CR3

movl %cr0,%eax

orl $0x80000000,%eax

movl %eax,%cr0 #以上三行使 CR0 的 PG=1, 开启分页机制

ret

CR3寄存器的格式如下：



从movl %eax,%cr0执行后，段部件产生的地址就不再被看成物理地址，而是要送往页部件进行变换，以得到真正的物理地址。

注意，现在内核工作在分页机制的一个特殊情况下，线性地址和经过页部件转换后的物 理地址相同，这是作者精心安排的。

最后的ret指令有2个作用。

在改变分页处理标志后要求使用转移指令刷新预取指令队列，这里用的是返回指令ret。

将之前压入栈中的 main() 程序入口地址弹出，并跳转到 init/main.c 程序去运行。

---------------------

作者：ARM的程序员敲着诗歌的梦

来源：CSDN

原文：https://blog.csdn.net/longintchar/article/details/79513086

版权声明：本文为博主原创文章，转载请附上博文链接！