**main()函数解析（一）——Linux-0.11 学习笔记（五）**

经过了前面的各种铺垫，终于来到了main函数。这篇博客的任务是把init/main.c讲清楚。由于牵扯到很多的函数调用，要想一次就说明白是很难的，所以我们把目标定得低一点，把脉络理清楚就行。

1. 宏定义\_syscall0

文件开头的头文件包含等就不多说了。对于C语言比较熟悉的朋友，我想第一个拦路虎就是“GCC内嵌汇编”。

static inline \_syscall0(int,fork)

static inline \_syscall0(int,pause)

static inline \_syscall1(int,setup,void \*,BIOS)

static inline \_syscall0(int,sync)

原理都是类似的，说清楚一个，其他的也就迎刃而解了。

static inline \_syscall0(int,fork)

\_syscall0()是在文件unistd.h中定义，它以内嵌汇编的形式调用 Linux 的系统调用中断 int 0x80。

系统调用（通常称为syscalls）是 Linux内核与上层应用程序进行交互通信的唯一接口。用户程序通过直接或间接（通过库函数）调用中断int 0x80（在eax寄存器中指定系统调用功能号），即可使用内核资源，包括系统硬件资源。

\_syscall0()其实是一个宏，这个宏定义在include/unistd.h 文件第 133 行：

#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; \

}

第5行：汇编语句，表示系统调用，0x80号中断；

第6行：输出部分，把eax的值传给变量\_\_res；

第7行：输入部分，把\_\_NR\_name的值赋给eax，即指明系统调用功能号；

第8~9行： 如果返回值>=0,则直接返回该值；

第10~11行： 否则置出错号errno(全局变量)，并返回-1。

顺便提一下，内嵌汇编语法如下。对此不熟悉的朋友可以专门找资料学习。

\_\_asm\_\_(汇编语句模板: 输出部分: 输入部分: 破坏描述部分)

1

根据\_syscall0()的宏定义，我们把static inline \_syscall0(int,fork)展开，得到：

static inline int fork(void) { long \_\_res; \_\_asm\_\_ volatile ("int $0x80" : "=a" (\_\_res) : "0" (2)); if (\_\_res >= 0) return (int) \_\_res; errno = -\_\_res; return -1; }

实际上展开结果就是上面一行。

可以手工展开，也可以用命令展开。用命令展开的方法是：

首先进入到 Linux-0.11 源码路径下，比如~/oslab/linux-0.11，然后输入命令：

gcc -E init/main.c -o main.i -I./include

如果你还没有实验环境，那赶紧弄一个吧，方法是 Linux 0.11 实验环境搭建或者Linux 0.11 实验环境搭建与调试

以上的展开结果实在是太长了，分行写如下：

static inline int fork(void)

{

long \_\_res;

\_\_asm\_\_ volatile ("int $0x80"

: "=a" (\_\_res)

: "0" (2));

if (\_\_res >= 0)

return (int) \_\_res;

errno = -\_\_res;

return -1;

}

第6行：括号里的“2”是因为在文件unistd.h中有#define \_\_NR\_fork 2

gcc会把上述“函数”体中的语句直接插入到调用fork()语句的代码处，因此执行fork()不会引起函数调用。另外，宏名称字符串syscall0中最后的0 表示无参数，1表示带1个参数。如果系统调用带有1个参数，那么就应该使用宏\_syscall1()。

2. setup.s读取的参数

/\*

\* This is set up by the setup-routine at boot-time

\*/

#define EXT\_MEM\_K (\*(unsigned short \*)0x90002)

#define DRIVE\_INFO (\*(struct drive\_info \*)0x90080)

#define ORIG\_ROOT\_DEV (\*(unsigned short \*)0x901FC)

以上三行，右侧的地址其实是setup.s运行时，读取了一些参数，并保存到了相应位置。忘了的同学可以参考我的博文 bootsect.s 分析—— Linux-0.11 学习笔记（一）



EXT\_MEM\_K (0x9002)：系统从 1MB 开始的扩展内存大小，以KB为单位；

DRIVE\_INFO (0x90080) ：硬盘参数表，包括第1个和第2个硬盘，共32字节；

ORIG\_ROOT\_DEV ：根文件系统所在的设备号3.

3. 读取CMOS实时时钟信息

#define CMOS\_READ(addr) ({ \

outb\_p(0x80|addr,0x70); \ // 把 (0x80|addr) 写入端口0x70

inb\_p(0x71); \ // 读端口0x71

})

要想搞清楚上面的代码，就先要弄清楚outb\_p和inb\_p。outb\_p和inb\_p都是宏，在文件\include\asm\io.h中定义。

3.1 outb\_p(value,port)

#define outb\_p(value,port) \

\_\_asm\_\_ ("outb %%al,%%dx\n" \

"\tjmp 1f\n" \

"1:\tjmp 1f\n" \

"1:"::"a" (value),"d" (port))

注意：第4行和第5行的“1”是标号。

第2行：把al的值写入端口dx；

第3行：跳转到1处，即下一句；这样写是为了延时；

第4行：同第3行；

第5行：port作为端口号，传给edx； 把eax的值传给value；

所以， outb\_p(value,port)表示把value写入端口port.

3.2 inb\_p(port)

#define inb\_p(port) ({ \

unsigned char \_v; \

\_\_asm\_\_ volatile ("inb %%dx,%%al\n" \

"\tjmp 1f\n" \

"1:\tjmp 1f\n" \

"1:":"=a" (\_v):"d" (port)); \

\_v; \

})

第3行：读端口dx到al；

第4~5行：跳转到1处，即下一句；为了延时；

第6行：port作为端口号，传给edx； 把eax的值传给\_v；

第7行：\_v的值作为整个表达式的返回值。

所以， inb\_p(port)表示读取端口port的值。

3.3 outb(value,port)和inb(port)

#define outb(value,port) \

\_\_asm\_\_ ("outb %%al,%%dx"::"a" (value),"d" (port))

#define inb(port) ({ \

unsigned char \_v; \

\_\_asm\_\_ volatile ("inb %%dx,%%al":"=a" (\_v):"d" (port)); \

\_v; \

})

既然都分析到这里了，那就把这两个宏也说了吧。这两个宏和上面的差不多，只不过不带延迟。

3.4 CMOS与RTC

PC 机的 CMOS 内存是由电池供电的 64 或 128 字节内存块，通常是系统实时钟芯片RTC (Real Time Chip) 的一部分。有些机器还有更大的内存容量。该 64 字节的CMOS原先在IBM PC-XT机器上用于保存时钟和日期信息，存放的格式是BCD码。由于这些信息仅用去 14 字节，因此剩余的字节就可用来存放一些系统配置数据。

CMOS的地址空间在基本地址空间之外，因此其中不包括可执行代码。要访问它需要通过端口 0x70、 0x71 进行。0x70 是地址端口，0x71 是数据端口。为了读取指定偏移位置的字节，必须首先使用out指令向地址端口 0x70 发送指定字节的偏移位置值，然后使用in指令从数据端口 0x71 读取指定的字节信息。同样，对于写操作也需要首先向地址端口 0x70 发送指定字节的偏移值，然后把数据写到数据端口 0x71 中去。

outb\_p(0x80|addr,0x70);把欲读取的字节地址(addr)与0x80进行或操作是没有必要的。因为那时的CMOS内存容量还没有超过128(=111\_1111b)字节，因此不需要把b7设为1。之所以会有这样的操作是因为当时Linus手头缺乏有关CMOS方面的资料，CMOS中时钟和日期的偏移地址都是他逐步实验出来的，也许在他的实验中将偏移地址与0x80进行或操作（并且还修改了其他地方）后正好取得了所有正确的结果，因此他的代码中也就有了这步不必要的操作。不过从1.0版本之后，该操作就被去除了。

下表是 CMOS 内存信息的一张简表。

CMOS 64 字节信息简表



3.5 time\_init函数

static void time\_init(void)

{

struct tm time;

do {

time.tm\_sec = CMOS\_READ(0); // 秒

time.tm\_min = CMOS\_READ(2); // 分

time.tm\_hour = CMOS\_READ(4); // 时

time.tm\_mday = CMOS\_READ(7); // 日

time.tm\_mon = CMOS\_READ(8); // 月

time.tm\_year = CMOS\_READ(9); // 年（since 1900）

} while (time.tm\_sec != CMOS\_READ(0));

BCD\_TO\_BIN(time.tm\_sec);

BCD\_TO\_BIN(time.tm\_min);

BCD\_TO\_BIN(time.tm\_hour);

BCD\_TO\_BIN(time.tm\_mday);

BCD\_TO\_BIN(time.tm\_mon);

BCD\_TO\_BIN(time.tm\_year);

time.tm\_mon--;

startup\_time = kernel\_mktime(&time);

}

结合上面的表格，6~11行非常好懂。

第12行：while (time.tm\_sec != CMOS\_READ(0));为什么有这个do-while循环呢？

CMOS的访问速度很慢。为了减小时间误差，在读取了所有数值后，若此时CMOS中秒值发生了变化，那么就重新读取所有值。这样内核就能把与CMOS时间误差控制在1秒之内。

注意，读取的值是BCD（Binary Coded Decimal）码格式。

BCD码：是一种十进制数字编码的形式。在这种编码下，每个十进制数字用一串单独的二进制比特来存储与表示。常见的有以4位表示1个十进制数字，称为压缩的BCD码（compressed or packed）；或者以8位表示1个十进制数字，称为未压缩的BCD码（uncompressed or zoned）。

比如当前时间是10:35:20，那么读出的二进制数是：

0001\_0000b：0011\_0101b：0010\_0000b

#define BCD\_TO\_BIN(val) ((val)=((val)&15) + ((val)>>4)\*10)

// (val)&15 即 (val)&0xF, 得到个位数;

// (val)>>4)\*10 把十位上的数字乘以10;

这个宏的作用是把BCD格式的值转换成二进制（或者说十进制，总之存到PC里都是二进制）

time.tm\_mon--;

startup\_time = kernel\_mktime(&time);

第2行：调用函数kernel\_mktime()，计算从 1970 年 1 月 1 日 0 时起到现在经过的秒数，作为开机时间，保存到全局变量startup\_time 中。更具体的分析可以参考我的博文 kernel\_mktime() 详解

4. main函数

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; //0x21C

drive\_info = DRIVE\_INFO;

memory\_end = (1<<20) + (EXT\_MEM\_K<<10); //EXT\_MEM\_K = 0x3c00, memory\_end = 0x100\_0000

memory\_end &= 0xfffff000; //0x100\_0000 = 16M

if (memory\_end > 16\*1024\*1024)

memory\_end = 16\*1024\*1024;

if (memory\_end > 12\*1024\*1024)

buffer\_memory\_end = 4\*1024\*1024; //buffer\_memory\_end = 4M

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; //4M

#ifdef RAMDISK\_SIZE //=1025

main\_memory\_start += rd\_init(main\_memory\_start, RAMDISK\_SIZE\*1024);

#endif

mem\_init(main\_memory\_start,memory\_end);

trap\_init();

blk\_dev\_init();

chr\_dev\_init();

tty\_init();

time\_init();

sched\_init();

buffer\_init(buffer\_memory\_end);

hd\_init();

floppy\_init();

sti();

move\_to\_user\_mode();

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.

\*/

for(;;) pause();

}

4.1 根设备号

ROOT\_DEV = ORIG\_ROOT\_DEV;

1

在 fs/super.c 中，定义了 int ROOT\_DEV = 0;

本文件内有宏定义

#define ORIG\_ROOT\_DEV (\*(unsigned short \*)0x901FC)

ROOT\_DEV = ORIG\_ROOT\_DEV;这条语句执行后（依据我的实验环境），ROOT\_DEV = 0x21C

在bootsect.s中，有

mov %cs:root\_dev+0, %ax

cmp $0, %ax

jne root\_defined

mov %cs:sectors+0, %bx

mov $0x0208, %ax # /dev/ps0 - 1.2Mb

cmp $15, %bx

je root\_defined

mov $0x021c, %ax # /dev/PS0 - 1.44Mb, excute here when debug

cmp $18, %bx

je root\_defined

undef\_root:

jmp undef\_root

root\_defined:

mov %ax, %cs:root\_dev+0

...

.org 508

root\_dev:

.word ROOT\_DEV ！这里存放根文件系统所在设备号（init/main.c中会用）

设备号 = 主设备号\*256 + 次设备号（也即 dev\_no = (major << 8) + minor )

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 表示1.2 MB 或 7 表示 1.44 MB 等）。

0x21C = 2<<8 + (7\*4+0)，所以根设备是 1.44M 的 A 驱动器。

4.2 计算主内存起始位置

memory\_end = (1<<20) + (EXT\_MEM\_K<<10); //EXT\_MEM\_K = 0x3c00, memory\_end = 0x100\_0000

memory\_end &= 0xfffff000; //0x100\_0000 = 16M

if (memory\_end > 16\*1024\*1024) //如果内存超过16M，则按16M计

memory\_end = 16\*1024\*1024;

if (memory\_end > 12\*1024\*1024) //如果内存超过12M，则设置缓冲区末端=4M

buffer\_memory\_end = 4\*1024\*1024; //buffer\_memory\_end = 4M

else if (memory\_end > 6\*1024\*1024)//如果内存超过6M，则设置缓冲区末端=2M

buffer\_memory\_end = 2\*1024\*1024;

else

buffer\_memory\_end = 1\*1024\*1024;//否则设置缓冲区末端=1M

main\_memory\_start = buffer\_memory\_end; //主内存起始位置=缓冲区末端

注意，代码注释部分的值是我通过实验测试出来的，你的实验环境不一定是这个值。

第1行：计算出内存大小

第2行：忽略不到4KB的内存数

在我的环境中，通过单步调试，代码执行第6行，也就是说缓冲区末端（buffer\_memory\_end）在4M处，也就是主内存的起始位置（main\_memory\_start）。

4.3 虚拟盘

#ifdef RAMDISK\_SIZE // 如果定义了虚拟盘

main\_memory\_start += rd\_init(main\_memory\_start, RAMDISK\_SIZE\*1024);

#endif

当 linux/Makefile文件中设置的RAMDISK值不为零时，表示系统会创建 RAM 虚拟盘设备。 在这种情况下，就会执行第2行，即主内存区的起始地址后移，也就是说主内存区头部还要划去一部分，供虚拟盘存放数据。

根据单步调试的结果，main\_memory\_start = 4194304（4M），RAMDISK\_SIZE = 1025



如图所示，内核程序占据在物理内存的开始部分，接下来是供硬盘或软盘等块设备使用的高速缓冲区部分（其中要扣除显卡内存和 ROM BIOS 所占用的内存，它们的地址范围是640KB~1MB)。

关于高速缓冲区：当一个进程需要读取块设备中的数据时，系统会首先把数据读到高速缓冲区中；当有数据需要写到块设备上时，系统也是先将数据放到高速缓冲区中，然后由块设备驱动程序写到相应的设备上。

内存的最后部分是供所有程序可以随时申请和使用的主内存区。内核程序在使用主内存区时，也同样先要向内核内存管理模块提出申请，在申请成功后方能使用。

对于含有 RAM 虚拟盘的系统，主内存区头部还要划去一部分，供虚拟盘存放数据。

long rd\_init(long mem\_start, int length)

{

int i;

char \*cp;

blk\_dev[MAJOR\_NR/\*=1\*/].request\_fn = DEVICE\_REQUEST;

rd\_start = (char \*) mem\_start;

rd\_length = length;

cp = rd\_start;

for (i=0; i < length; i++)

\*cp++ = '\0';

return(length);

}

第6行：MAJOR\_NR的值是1。

blk\_dev是一个数组，其成员类型是struct blk\_dev\_struct

struct blk\_dev\_struct blk\_dev[NR\_BLK\_DEV] = {

{ NULL, NULL }, /\* no\_dev \*/

{ NULL, NULL }, /\* dev mem \*/

{ NULL, NULL }, /\* dev fd \*/

{ NULL, NULL }, /\* dev hd \*/

{ NULL, NULL }, /\* dev ttyx \*/

{ NULL, NULL }, /\* dev tty \*/

{ NULL, NULL } /\* dev lp \*/

};

struct blk\_dev\_struct的定义是

struct blk\_dev\_struct {

void (\*request\_fn)(void);

struct request \* current\_request;

};

可以看出，2个成员都是指针，request\_fn指向函数，current\_request指向struct request.

回到函数rd\_init:

blk\_dev[MAJOR\_NR/\*=1\*/].request\_fn = DEVICE\_REQUEST;

DEVICE\_REQUEST实际上是设备请求函数do\_rd\_request

因为#define DEVICE\_REQUEST do\_rd\_request

void do\_rd\_request(void)

{

int len;

char \*addr;

INIT\_REQUEST;

addr = rd\_start + (CURRENT->sector << 9);

len = CURRENT->nr\_sectors << 9;

if ((MINOR(CURRENT->dev) != 1) || (addr+len > rd\_start+rd\_length)) {

end\_request(0);

goto repeat;

}

if (CURRENT-> cmd == WRITE) {

(void) memcpy(addr,

CURRENT->buffer,

len);

} else if (CURRENT->cmd == READ) {

(void) memcpy(CURRENT->buffer,

addr,

len);

} else

panic("unknown ramdisk-command");

end\_request(1);

goto repeat;

}

此函数的代码，我们先不深入，以后用到再说。我们关注的是rd\_init函数的以下几行：

rd\_start = (char \*) mem\_start;

rd\_length = length;

cp = rd\_start; // cp是 char \* 类型

for (i=0; i < length; i++)

\*cp++ = '\0'; //以上3行， 盘区清零

return(length);

rd\_start和rd\_length都是全局变量，定义在文件kernel\blk\_drv\ramdisk.c中：

char \*rd\_start; //虚拟盘的起始地址

int rd\_length = 0; //虚拟盘空间大小，以B为单位

4.4 mem\_init函数

该函数对1MB以上内存区域以页面为单位进行管理前的初始化设置工作。

一个页面长度为4KB字节。该函数把1MB以上所有物理内存划分成一个个页面，并使用一个页面映射字节数组mem\_map[] 来管理这些页面。对于具有 16MB 内存容量的机器，该数组共有3840( (16M-1M)/4K=3840 )项 ，即可管理3840个物理页面。

每当一个物理内存页面被占用时就把 mem\_map[]中对应的的字节值增1 ；若释放一个物理页面，就把对应字节值减 1。 若字节值为0 , 则表示对应页面空闲； 若字节值 >=1，则表示对应页面被占用或被不同程序共享占用。

在该版本内核中，最多能管理16MB的物理内存，大于16MB的内存将弃掉不用。对于具有16MB内存的PC机系统，在没有设置虚拟盘 RAMDISK 的情况下start\_mem通常是4MB，end\_mem是 16MB。因此主内存区范围是4MB~16MB，共有3072个物理页面可供分配。如果设置了 RAMDISK，那么start\_mem会大于4MB，比如我的实验环境是5243904(=5121K)即RAMDISK占用了1025K(=5121K-4096K).

void mem\_init(long start\_mem, long end\_mem)

{

int i;

HIGH\_MEMORY = end\_mem;

// 参数start\_mem是可用作页面分配的主内存区起始地址

//（已去除RAMDISK所占内存空间）。

// end\_mem是实际物理内存最大地址。

//地址范围start\_mem到end\_mem是主内存区。

for (i=0 ; i<PAGING\_PAGES ; i++) //PAGING\_PAGES = 3840

mem\_map[i] = USED;

i = MAP\_NR(start\_mem); // i=主内存区起始位置处页面号

end\_mem -= start\_mem; // 首尾相减，算出主内存区的大小

end\_mem >>= 12; // 主内存区的总页面数

while (end\_mem-->0)

mem\_map[i++]=0; // 以上2行， 主内存区页面对应字节值清零

}

第11~12行： 首先将 1MB 到 16MB 范围内所有内存页面设置为已占用状态，即各项字节值全部设置成 USED（100）

PAGING\_PAGES 被定义为(PAGING\_MEM0RY>>12)，即(15\*1024\*1024)>>12=3840

#define LOW\_MEM 0x100000

#define PAGING\_MEMORY (15\*1024\*1024)

#define PAGING\_PAGES (PAGING\_MEMORY>>12)

#define MAP\_NR(addr) (((addr)-LOW\_MEM)>>12)

#define USED 100

第13行：MAP\_NR(start\_mem) 即是(start\_mem-0x100000)>>12，计算出主内存区起始位置处页面号。

4.5 trap\_init函数

void trap\_init(void)

{

int i;

set\_trap\_gate(0,÷\_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);

...

...

}

以上代码主要是安装陷阱门。我们拿第5行作为例子，具体分析一下。

4.5.1 set\_trap\_gate(n,addr)

set\_trap\_gate(n,addr)其实是\_set\_gate(&idt[n],15,0,addr)，也就是下面7~15行的内嵌汇编代码。

#define set\_trap\_gate(n,addr) \

\_set\_gate(&idt[n],15,0,addr)

...

#define \_set\_gate(gate\_addr,type,dpl,addr) \

\_\_asm\_\_ ("movw %%dx,%%ax\n\t" \

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

"movl %%eax,%1\n\t" \

"movl %%edx,%2" \

: \

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

"o" (\*((char \*) (gate\_addr))), \

"o" (\*(4+(char \*) (gate\_addr))), \

"d" ((char \*) (addr)),"a" (0x00080000))

d: 表示 edx

a: 表示 eax

i: 允许一个立即整形操作数，包括其值仅在汇编时确定的符号常量。

o: 允许一个内存操作数，但只有当地址是可偏移的。即该地址加上一个小的偏移量，结果是一个有效的内存地址。

以上内嵌汇编代码没有输出部分，仅有输入部分。

上图是陷阱门的格式，上面是高4字节（代码中用 edx 表示），下面是低4字节（代码中用 eax 表示）。注意：过程入口点偏移值不是物理地址，而是线性地址。

第15行：

"d" ((char \*) (addr))表示用 addr 加载edx；此时，偏移值的[31:16]就位。

addr 是异常处理函数入口点的地址。因为内核代码段的线性基址是0，所以偏移值等于函数的线性地址，又因为内核在之前的分页中采用了恒等映射机制——线性地址等于物理地址，所以偏移值等于函数的物理地址。

"a" (0x00080000) ：表示用 0x0008\_0000 加载 eax；此时，段选择符就位。

段选择子（符）的值是0x08，为什么是这个值呢？因为在进入main函数之前，已经设置好了GDT，0x08是代码段的选择子。忘了的话可以参考我的博文head.s——第三节。

第7行的"movw %%dx,%%ax\n\t"表示用 dx 加载 ax；此时，偏移值的[15:0]就位，eax也就位。

第8行的"movw %0,%%dx\n\t"，表示用(0x8000+(dpl<<13)+(type<<8))加载 dx,

这里的 8 表示 P=1； 此时，edx 就位。

根据\_set\_gate(&idt[n],15,0,addr)的参数可知type=15（表示陷阱门）, dpl=0。(0x8000+(dpl<<13)+(type<<8))拼出了陷阱门的第4~5字节（edx的低字）。

第9行"movl %%eax,%1\n\t"表示把 eax 的值赋给\*((char \*) (gate\_addr))，就是赋给idt[n]的前4字节。

第10行"movl %%edx,%2" 表示把edx的值赋给\*(4+(char \*)(gate\_addr))，就是赋给idt[n]的后4字节。这8字节拼起来就是完整的idt[n].

4.5.2 idt数组

idt是中断描述符表（其实是数组），一共有 256 个表项，一个表项占8字节。

%1对应第13行的(\*((char \*) (gate\_addr)))

gate\_addr就是第2行的&idt[n]，那么idt是什么呢？在文件include\linux\head.h中有：

typedef struct desc\_struct {

unsigned long a,b;

} desc\_table[256];

extern desc\_table idt,gdt;

1~3行：为struct desc\_struct [256]取了一个别名——desc\_table，也就是说desc\_table的类型是“struct desc\_struct类型的数组”。

第6行，注意extern关键字，声明（而不是定义）了 idt 和 gdt，它们的类型都是desc\_table，即“struct desc\_struct类型的数组”。所以，&idt[n]是数组idt第n个元素的地址。

可能有人要问， idt 和 gdt的定义在哪里呢？

它们是在汇编代码boot/head.s中定义的。

在本文件末尾有：

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

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 \*/

8

另外本文件开头有

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

.globl xxx表示把符号xxx声明为全局变量/标号，以供其他源文件访问。

4.5.3 \_set\_gate(gate\_addr,type,dpl,addr)总结

#define \_set\_gate(gate\_addr,type,dpl,addr) \

\_\_asm\_\_ ("movw %%dx,%%ax\n\t" \ //将偏移地址低字与选择符组合成描述符低4字节(eax)

"movw %0,%%dx\n\t" \ //将类型标志与偏移地址高字组合成描述符高4字节(edx)

"movl %%eax,%1\n\t" \ //分别设置门描述符的低4字节和高4字节

"movl %%edx,%2" \

: \

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

"o" (\*((char \*) (gate\_addr))), \

"o" (\*(4+(char \*) (gate\_addr))), \

"d" ((char \*) (addr)),"a" (0x00080000))

\_set\_gate(gate\_addr,type,dpl,addr)此宏用于设置门描述符。

根据参数中的中断或异常处理过程地址 addr 、门描述符类型 type 和特权级信息 dpl ，设置位于地址 gate\_addr 处的门描述符。（注意：下面的“偏移”是相对于内核代码或数据段来说的。）

gate\_addr：描述符存储地址；

type：描述符类型；

dpl：描述符特权级；

addr：偏移地址。

%0：由dpl，type组合成的类型值；

%1：描述符低 4 字节的存储地址；

%2：描述符高 4 字节的存储地址；

%3：edx（程序偏移地址addr）；

%4： eax（高字中含有段选择符0x8） 。

4.5.4 set\_system\_gate(n,addr)

#define set\_system\_gate(n,addr) \

\_set\_gate(&idt[n],15,3,addr)

这个宏和set\_trap\_gate(n,addr)的区别仅有一点：前者的dpl=3，后者的dpl=0;

分析到这里， trap\_init函数的大意已经明了。

void trap\_init(void)

{

int i;

set\_trap\_gate(0,÷\_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); // 设置协处理器中断0x2d(=45)的陷阱门描述符

outb\_p(inb\_p(0x21)&0xfb,0x21); // 允许8259A主芯片的IRQ2中断请求

outb(inb\_p(0xA1)&0xdf,0xA1);

set\_trap\_gate(39,¶llel\_interrupt); //设置并行口1的中断0x27(=39)陷阱门描述符

}

5~2行：设置IDT的描述符。其中断点陷阱中断int3、溢出中断overflow、边界出错中断bounds可以由任何程序产生。

22~23行：把int 17 ~ int 48的陷阱门先设置为reserved，以后各个硬件初始化时会重新设置自己的陷阱门。

注意：set\_trap\_gate的第二个参数是中断处理函数的入口点，它们的代码在文件linux/kernel/asm.s或者linux/kernel/system\_call.s中。

第25行：outb\_p(inb\_p(0x21)&0xfb,0x21);

0x21是 8259A 主片命令字OCW1的端口地址，用于对其中断屏蔽寄存器 IMR 进行读/写操作。

inb\_p(0x21)&0xfb读出 IMR 的值，然后与0xfb(=1111\_1011b)，即清零D2位，也就是允许主片的 IRQ2 中断请求。



注意：Linux-0.11 系统把主片的 ICW2 设置为 0x20，表示主片中断请求0~7级对应的中断号是 0x20~0x27；把从片的 ICW2 设置成 0x28，表示从片中断请求8~15级对应的中断号是 0x28~0x2f。



第26行：outb(inb\_p(0xA1)&0xdf,0xA1);

0xA1是 8259A 从片命令字OCW1的端口地址。原理同上，inb\_p(0xA1)&0xdf读出从片 IMR 的值，然后与0xdf(=1101\_1111)，即清零D5位，由上图可知，允许从片 IRQ13 协处理器中断。

关于8259A的编程，可以参考我的博文： 详解8259A

囿于篇幅，对main()函数的分析先到这里，剩下的内容下次再说。谢谢您的阅读！