main函数解析（二）——Linux-0.11 学习笔记（六）

4.6 blk\_dev\_init函数

void blk\_dev\_init(void)

{

int i;

for (i=0 ; i<NR\_REQUEST ; i++) {

request[i].dev = -1; //表示空闲

request[i].next = NULL;

}

}

这里的request是一个全局的数组。

/\*

\* The request-struct contains all necessary data

\* to load a nr of sectors into memory

\*/

struct request request[NR\_REQUEST]; //请求项数组

#define NR\_REQUEST 32

...

struct request {

int dev; //设备号，若为-1，则表示空闲

int cmd; // 命令 READ or WRITE

int errors; //操作时产生的错误次数

unsigned long sector; // 起始扇区

unsigned long nr\_sectors; // 读/写扇区数

char \* buffer; //数据缓冲区

struct task\_struct \* waiting; // 任务等待操作执行完成的地方

struct buffer\_head \* bh; // 缓冲区头指针

struct request \* next; //指向下一个struct request，构成单向链表

};

以上的代码要想完全理解，就牵扯到块设备驱动程序了。



每个块设备的当前请求指针与请求项数组中该设备的请求项链表共同构成了该设备的请求队列。项与项之间利用字段next指针形成链表。因此块设备项和相关的请求队列形成如图所示结构。请求项采用数组加链表结构的主要原因是为了满足两个目的：一是利用请求项的数组结构在搜索空闲请求块时可以进行循环操作，搜索访问时间复杂度为常数，因此程序可以编制得很简洁；二是为满足电梯算法 (Elevator Algorithm) 插入请求项操作，因此也需要采用链表结构。图中画出了硬盘设备当前具有4个请求项，软盘设备具有1个请求项，而虚拟盘设备目前暂时没有读写请求项。

对于一个当前空闲的块设备，当 ll\_rw\_block()函数为其建立第一个请求项时，会让该设备的当前请求项指针current\_request直接指向刚建立的请求项，并且立刻调用对应设备的请求项操作函数开始执行块设备读写操作。当一个块设备已经有几个请求项组成的链表存在，ll\_rw\_block()就会利用电梯算法，根据磁头移动距离最小原则，把新建的请求项插入到链表适当的位置处。

以上内容摘自赵炯博士的《Linux内核完全剖析》。

虽然看完不甚理解，但至少明白blk\_dev\_init函数了。其实很简单，就是把数组的每一项的设备号设为 -1，表示空闲，然后把 next 设为 NULL（因为空闲，所以没有被插入链表）。

4.7 chr\_dev\_init函数

此函数实现为空。

void chr\_dev\_init(void)

{

}

4.8 tty\_init函数

void tty\_init(void)

{

rs\_init(); // 初始化串行中断程序和串行接口1、2

con\_init(); // 初始化控制台终端

}

4.8.1 rs\_init函数

void rs\_init(void)

{

set\_intr\_gate(0x24,rs1\_interrupt); // 设置串行口1的中断门向量

set\_intr\_gate(0x23,rs2\_interrupt); // 设置串行口2的中断门向量

init(tty\_table[1].read\_q.data); // 初始化串行口1,参数是端口基地址

init(tty\_table[2].read\_q.data); // 初始化串行口2,参数是端口基地址

outb(inb\_p(0x21)&0xE7,0x21); // 允许8259A主片响应IRQ3、IRQ4中断请求, 0x21 对应 8259A 主片的中断屏蔽寄存器

}

4.8.1.1 设置中断门

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

\_set\_gate(&idt[n],14,0,addr)

set\_intr\_gate(n,addr)的宏展开是\_set\_gate(&idt[n],14,0,addr)。

 

上图是中断门描述符的格式，根据 [11:8] = 14，可以知道代码中的14表示中断门。

宏\_set\_gate(gate\_addr,type,dpl,addr)用于设置门描述符。具体的分析参见 main函数解析（一）——Linux-0.11 学习笔记（五）

这个宏根据参数中的

门描述符类型 type

描述符特权级 dpl

中断或异常处理过程的偏移地址地址 addr

设置位于地址 gate\_addr 处的门描述符。

所以， set\_intr\_gate(n,addr)表示在元素idt[n]（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。）位置安装一个中断处理过程的偏移地址地址为 addr 的、特权级为0的中断门描述符。



Linux-0.11 系统

把主片的中断号设置为 0x20~0x27；

把从片的中断号设置为 0x28~0x2f。

根据上图可以知道，串行口1的中断号是0x24，串行口2的中断号是0x23；

顺便提一下：使用中断门（interrupt gate） 来构造中断调用机制的，当 processor 进入 interrupt handler 执行前， 会将 eflags 的值压入栈中保存并且会清除 eflags.IF 标志位，这意味着进入中断后不再响应其他的可屏蔽中断。

4.8.1.2 串口的初始化

static void init(int port) // port 是端口基地址

{

...

}

串口初始化，说白了就是配置一些寄存器，使串口可以收发数据。

说到这个串口，内容还挺多，不了解的一定要参考我的博文 PC 机 UART（NS8250）详解

init(tty\_table[1].read\_q.data);传入的参数是tty\_table[1].read\_q.data，如果没有猜错的话，tty\_table[1].read\_q.data一定是端口的基地址。为了验证猜测，我们找找和tty\_table有关的定义。

果然，在kernel\chr\_drv\tty\_io.c中看到了以下代码：

struct tty\_struct tty\_table[] = {

{

...

},{

...

{0x3f8,0,0,0,""}, /\* rs 1 \*/

{0x3f8,0,0,0,""},

{0,0,0,0,""}

},{

...

{0x2f8,0,0,0,""}, /\* rs 2 \*/

{0x2f8,0,0,0,""},

{0,0,0,0,""}

}

};

0x3f8和0x2f8正是串口1和2的端口基地址。

static void init(int port) // port 是端口基地址

{

outb\_p(0x80,port+3); /\* set DLAB of line control reg \*/

outb\_p(0x30,port); /\* LS of divisor (48 -> 2400 bps \*/

outb\_p(0x00,port+1); /\* MS of divisor \*/

outb\_p(0x03,port+3); /\* reset DLAB \*/

outb\_p(0x0b,port+4); /\* set DTR,RTS, OUT\_2 \*/

outb\_p(0x0d,port+1); /\* enable all intrs but writes \*/

(void)inb(port); /\* read data port to reset things (?) \*/

}

目前遇到的读写端口的宏有4个：

outb\_p(value,port)—— 把value写入端口port.

inb\_p(port)——读取端口port的值。

outb(value,port)—— 把value写入端口port.

inb(port)——读取端口port的值。

前2个带延时，后2个不带延时。

这4个宏，具体的定义和分析可以参考我的博文 main函数解析（一）——Linux-0.11 学习笔记（五）

第3行：outb\_p(0x80,port+3); 把0x80写入端口（0x3f8+3=0x3fb，即 LCR 寄存器）,也就是使 DLAB=1。

第4行：outb\_p(0x30,port);写波特率因子的低字节为0x30

第5行：outb\_p(0x00,port+1);写波特率因子的高字节为0x00

所以，波特率因子为 0x0030=48。

波特率和因子之间的关系是：



公式变形一下：



所以

波特率=1.8432MHz/(48∗16)=1843200768=2400

第6行：outb\_p(0x03,port+3);无奇偶校验位、8 位数据位和 1 位停止位，同时复位 DLAB；

第7行：outb\_p(0x0b,port+4);若要让 UART 的中断请求信号能够送到 8259A 中断控制器，就需要把MODEM 控制寄存器 MCR 的位3(OUT2) 置位。因为在PC 机中，该位控制着 INTRPT 引脚到 8259A 的电路。MCR的位 1 和位 0 分别用于控制 MODEM ， 当这两位置位时，UART的数据终端就绪引脚（DTR）和请求发送引脚（RTS）输出有效。

第8行：outb\_p(0x0d,port+1);写IER（Interrupt enable register，中断允许寄存器 ）

0x0d = 1101b

[3] = 1 允许modem状态中断；

[2] = 1 允许接收器线路状态中断；

[1] = 0 禁止发送保持寄存器空中断；

[0] = 1 允许接收到数据中断;

第9行：(void)inb(port);读接收缓存寄存器 ，以进行复位？？其实这句话我也不清楚是否必要，先在这里留个疑问。

4.8.2 con\_init()函数

con\_init()函数（在文件linux/kernel/console.c中）首先根据 setup.s 程序取得的系统硬件参数初始化几个本函数专用的静态全局变量。然后根据显示卡模式（单色还是彩色）和显示卡类型（EGA/VGA、CGA、MDA 等) 分别设置显示内存起始位置以及显示索引寄存器和显示数值寄存器的端口号。最后设置键盘中断陷阱描述符并打开对键盘中断的屏蔽位。

显卡的历史

磨刀不误砍柴工，在理解这个函数之前，不妨先了解一下显卡的历史。

显卡的前身

MDA

最早的显卡称为显示适配器，在“黑底白字”的DOS年代，对显示的要求是极低的。最早的显示类型是 MDA（Monochrome Display Adapter），只能区别出黑白两色。早期的8080、8088，一直到80286都是使用这种类型的显示适配器。它的功能极为简单，一般集成 16KB 显存，是不为人关注的电脑配件。

CGA

到了286时，PC上出现了一些和图形相关的软件，因此出现了一种四色适配器，只能识别三原色和黑白。由于这是第一种彩色的显示适配器，所以称为 CGA（Color Graphics Adaptor，彩色图形适配器）。CGA 时代对显卡的要求已经大幅度提高，但是当时的制作工艺仍然远远高于显卡芯片的需求，因此 CGA 显示适配器依旧被整合在主板上，以一块单芯片的方式来实现，所以“显卡”尚未诞生。

EGA

CGA 的分辨率太低，于是又有了 EGA（Enhanced Graphics Adapter，增强图形适配器）。在显示性能方面（颜色和分辨率），EGA 介于 CGA 和 VGA 之间，可以在高达640x350的分辩率下达到16色。

显卡的诞生与换代

以上MDA、CGA、EGA 三种标准都是以 TTL 数字信号输出，而之后的 VGA 标准采用模拟信号输出，因而其彩色显示能力大大加强，原则上可以显示无穷多的颜色。VGA 最初代表分辨率，在个人电脑的启蒙时代，能够输出 VGA（640×480）这样的分辨率并不是一件易事，VGA 标准的出现对显示输出设备首次提出了较高的要求，于是催生了 VGA Card，显卡正式诞生！

● 第一代显卡：VGA Card，支持256色显示，1988年

● 第二代显卡：Graphics Card，支持Windows图形加速，1991年

● 第三代显卡：Video Card，支持视频加速，1994年

● 第四代显卡：3D Accelerator Card，支持3D加速，1994年

● 第五代显卡：GPU图形处理器，支持硬件 T&L（Transform and Lighting，多边形转换与光源处理），1999年

● 现代和未来显卡：GPGPU（General-purpose computing on graphics processing units，简称 GPGPU 或 GP²U）通用计算图形处理器，支持几何着色、物理加速、高清解码、科学计算……

和显示有关的静态全局变量的初始化

void con\_init(void)

{

register unsigned char a;

// 定义寄存器变量 a，该变量将被保存在一个寄存器中，以便于高效访问和操作。若想指定存放的寄存器（如 eax) ，则可写成 register unsigned char a asm("ax"）；

char \*display\_desc = "????";

char \*display\_ptr;

video\_num\_columns = ORIG\_VIDEO\_COLS;

video\_size\_row = video\_num\_columns \* 2;

video\_num\_lines = ORIG\_VIDEO\_LINES; //每屏25行

video\_page = ORIG\_VIDEO\_PAGE; // 我觉得有点问题

video\_erase\_char = 0x0720; // 擦除字符（0x20是空格，0x07是属性）

...

在文件linux/kernel/console.c中有宏定义：

#define ORIG\_X (\*(unsigned char \*)0x90000)

#define ORIG\_Y (\*(unsigned char \*)0x90001)

#define ORIG\_VIDEO\_PAGE (\*(unsigned short \*)0x90004)

#define ORIG\_VIDEO\_MODE ((\*(unsigned short \*)0x90006) & 0xff)

#define ORIG\_VIDEO\_COLS (((\*(unsigned short \*)0x90006) & 0xff00) >> 8)

#define ORIG\_VIDEO\_LINES (25)

#define ORIG\_VIDEO\_EGA\_AX (\*(unsigned short \*)0x90008)

#define ORIG\_VIDEO\_EGA\_BX (\*(unsigned short \*)0x9000a)

#define ORIG\_VIDEO\_EGA\_CX (\*(unsigned short \*)0x9000c)

右边的地址，其实对应 setup.s 中取得的系统硬件参数的存放位置。如果忘了，可以参考我的博文 setup.s 分析—— Linux-0.11 学习笔记（二）

第9行：video\_num\_columns = (((\*(unsigned short \*)0x90006) & 0xff00) >> 8);

对比汇编代码，也就是把AH的值（字符列数）赋给全局变量video\_num\_columns ；

`setup.s中相关代码如下

! 获取显示卡当前的显示模式

! 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。

! ds = 0x9000

! 0x90004(l个字)存放当前页；0x90006(1字节)存放显示模式；0x90007(1字节)存放字符列数。

mov ah,#0x0f

int 0x10

mov [4],bx ! bh = 当前显示页

mov [6],ax ! al = 显示模式, ah = 字符列数（窗口宽度）

第10行：video\_size\_row = video\_num\_columns \* 2;算出每行字符需使用的字节数。

第12行：video\_page = (\*(unsigned short \*)0x90004);

注意，video\_page是static unsigned char类型。

根据汇编代码第8行，应该是0x90005处的一个字节存放当前活动页码，所以我认为第12行应该改为：

video\_page = (\*(unsigned char \*)0x90005);

在所有源码文件中搜了一波发现video\_page这个变量没有被用到，好吧，暂且不管，接着往下看。

显示模式

#define ORIG\_VIDEO\_MODE ((\*(unsigned short \*)0x90006) & 0xff)

根据上面汇编代码的第9行，ORIG\_VIDEO\_MODE中的值是显示模式。

其取值对应的含义如下表。

AL Type Format Cell Colors Adapter Addr Monitor

0 text 40x25 8x8\* 16/8 (shades) CGA,EGA b800 Composite

1 text 40x25 8x8\* 16/8 CGA,EGA b800 Comp,RGB,Enh

2 text 80x25 8x8\* 16/8 (shades) CGA,EGA b800 Composite

3 text 80x25 8x8\* 16/8 CGA,EGA b800 Comp,RGB,Enh

4 graphic 320x200 8x8 4 CGA,EGA b800 Comp,RGB,Enh

5 graphic 320x200 8x8 4 (shades) CGA,EGA b800 Composite

6 graphic 640x200 8x8 2 CGA,EGA b800 Comp,RGB,Enh

7 text 80x25 9x14\* 3 (b/w/bold) MDA,EGA b000 TTL Mono

8,9,0aH PCjr modes

0bH,0cH (reserved; internal to EGA BIOS)

0dH graphic 320x200 8x8 16 EGA,VGA a000 Enh,Anlg

0eH graphic 640x200 8x8 16 EGA,VGA a000 Enh,Anlg

0fH graphic 640x350 8x14 3 (b/w/bold) EGA,VGA a000 Enh,Anlg,Mono

10H graphic 640x350 8x14 4 or 16 EGA,VGA a000 Enh,Anlg

11H graphic 640x480 8x16 2 VGA a000 Anlg

12H graphic 640x480 8x16 16 VGA a000 Anlg

13H graphic 640x480 8x16 256 VGA a000 Anlg

Notes: With EGA, VGA, and PCjr you can add 80H to AL to initialize a video mode without clearing the screen.

\*The character cell size for modes 0-3 and 7 varies, depending on the hardware. On modes 0-3: CGA=8x8, EGA=8x14, and VGA=9x16. For mode 7, MDPA and EGA=9x14, VGA=9x16, LCD=8x8.

if (ORIG\_VIDEO\_MODE == 7) // 等于7说明是单色

{

video\_mem\_start = 0xb0000; // 设置内存起始地址

video\_port\_reg = 0x3b4; // 设置索引寄存器端口

video\_port\_val = 0x3b5; // 设置数据寄存器端口

// 注意，这里使用了 BL 在调用中断 int 0x10 前后是否被改变的方法来判断卡的类型。

if ((ORIG\_VIDEO\_EGA\_BX & 0xff) != 0x10)

{ // BL之前设置的值是0x10,调用中断后发生改变，则说明是EGA

//#define VIDEO\_TYPE\_EGAM 0x20 /\* EGA/VGA in Monochrome Mode\*/

video\_type = VIDEO\_TYPE\_EGAM; // 设置显示类型

video\_mem\_end = 0xb8000; // 设置显示内存末端地址

display\_desc = "EGAm"; // 设置显示描述字符串

}

else // 没有改变说明是MDA

{

video\_type = VIDEO\_TYPE\_MDA;

video\_mem\_end = 0xb2000;

display\_desc = "\*MDA";

}

}

else // 说明是彩色

{

video\_mem\_start = 0xb8000; // 设置内存起始地址

video\_port\_reg = 0x3d4; // 设置索引寄存器端口

video\_port\_val = 0x3d5; // 设置数据寄存器端口

if ((ORIG\_VIDEO\_EGA\_BX & 0xff) != 0x10)

{

video\_type = VIDEO\_TYPE\_EGAC; // 说明是EGA或者VGA显卡

video\_mem\_end = 0xbc000; // 设置显示内存末端地址

display\_desc = "EGAc"; // EGA彩色

}

else

{

video\_type = VIDEO\_TYPE\_CGA; // 设置显示类型为CGA

video\_mem\_end = 0xba000;

display\_desc = "\*CGA";

}

}

setup.s中相关代码是：

! 检查显示方式(EGA/VGA)并获取参数。

! 调用 BIOS 中断 0x10，功能号： ah = 0xl2，子功能号： bl = 0xl0

! 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。

!

mov ah,#0x12 ! 功能号

mov bl,#0x10 ! 子功能号

int 0x10

mov [8],ax ! 我也不知道这个是什么(╯︵╰)

mov [10],bx ! bh=显示状态(单色模式/彩色模式),bl=已安装的显存大小

mov [12],cx ! ch=特性连接器比特位信息,cl=视频开关设置信息

注意，C代码使用了 BL 在调用中断 int 0x10 前后是否被改变的方法来判断卡的类型。BL在调用中断之前被赋值为 0x10 ，在调用中断后，其值可能不变（CGA 或 MDA），也可能变为0~3中的一个（EGA 或 VGA）。

在屏幕右上角显示显卡类型

下面的代码作用是在屏幕右上角显示描述字符串。采用的方法是直接将字符串写到显示内存的相应位置处。首先将显示指针display\_ptr指到屏幕第1行最右端起第4个字符处（每个字符需 2 个字节，因此减 8 ) ，然后循环复制字符串的字符。

display\_ptr = ((char \*)video\_mem\_start) + video\_size\_row - 8;

while (\*display\_desc) // 循环能停止是因为字符串末尾的'\0'

{

\*display\_ptr++ = \*display\_desc++; // 复制字符

display\_ptr++; // 跳过属性字节

}

在我的实验环境调试，截图如下：



和滚屏有关的变量

初始化用于滚屏的变量（主要用于EGA/VGA）：

/\* Initialize the variables used for scrolling (mostly EGA/VGA) \*/

origin = video\_mem\_start; // 滚屏起始显存地址

scr\_end = video\_mem\_start + video\_num\_lines \* video\_size\_row;// 结束地址

top = 0; // 最顶端行号

bottom = video\_num\_lines; // 最底端行号

#define ORIG\_X (\*(unsigned char \*)0x90000) // 列号

#define ORIG\_Y (\*(unsigned char \*)0x90001) // 行号

...

// 初始化当前光标所在位置（x,y）和光标对应的显存位置 pos

gotoxy(ORIG\_X,ORIG\_Y);

第1~2行对应的汇编代码（setup.s）是

mov ax,#INITSEG !INITSEG = 0x9000

mov ds,ax ! ds = 0x9000

mov ah,#0x03 ! 功能号=3，获取光标的位置

xor bh,bh ! bh = 页号 = 0（输入）

int 0x10 ! 输出： DH=行号，DL=列号

mov [0],dx ! 保存光标的行号和列号到 0x90000，共占2字节.

gotoxy函数（kernel\chr\_drv\console.c）的实现如下：

/\* NOTE! gotoxy thinks x==video\_num\_columns is ok \*/

static inline void gotoxy(unsigned int new\_x,unsigned int new\_y)

{

if (new\_x > video\_num\_columns || new\_y >= video\_num\_lines)

return;

x=new\_x; // 记录下当前光标所在的列

y=new\_y; // 记录下当前光标所在的行

pos=origin + y\*video\_size\_row + (x<<1);// 记录当前光标所在位置对应的显存地址。我调试时，video\_size\_row = 160

}

第4行：判断参数是否合法。在我的实验环境中，video\_num\_lines = 25，即new\_y的取值是[0,24]；video\_num\_columns = 80，即new\_x的取值是[0,80]，为什么可以等于80呢？目前还不知道，我想作者这样写肯定有他的道理，后面多留个心。

和显示有关的代码就暂时结束了。后面是关于键盘的。

允许键盘工作

set\_trap\_gate(0x21,&keyboard\_interrupt); //安装陷阱门，键盘的中断号是0x21, 对应8259A主片的 IRQ1

// 已经反复强调， Linux-0.11 系统把主片的中断号设置为 `0x20~0x27`；

// 把从片的中断号设置为 `0x28~0x2f`。

outb\_p(inb\_p(0x21)&0xfd,0x21); // 允许键盘中断

a=inb\_p(0x61); // 读取键盘端口 0x61(8255A端口PB)到 a(之前定义的寄存器变量)

outb\_p(a|0x80,0x61); // b7置位，禁止键盘工作

outb(a,0x61); // 再允许键盘工作，用以复位键盘

第4行：0x21是 8259A 主片命令字OCW1的端口地址，（注意：不是第1行的那个0x21）用于对其中断屏蔽寄存器 IMR 进行读/写操作。

写到这里，尽管篇幅较长，可是才分析了 2 个函数。

blk\_dev\_init();

chr\_dev\_init(); //实现为空

tty\_init();

数了数，距main函数结束，还有10多个函数呢……好了，今天就到这里，明日继续精进。

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

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/79945853

版权声明：本文为博主原创文章，转载请附上博文链接！