# (1.25)第五天 结构体、文字显示与GDT/IDT

## 接收启动信息与结构体

将显存地址，分辨率等相关参数写在asmhead.nas文件中，C代码从其指定的地址读取，而不是写死在C代码中。并使用结构体来读取信息。

**本将将asmhead.nas中的注释翻译成了中文**。

|  |  |
| --- | --- |
| struct BOOTINFO {  char cyls, leds, vmode, reserve;  short scrnx, scrny;  char \*vram;  }; | void HariMain(void)  {  **struct BOOTINFO \*binfo;**  init\_palette();  **binfo = (struct BOOTINFO \*) 0x0ff0;**  init\_screen(binfo->vram, binfo->scrnx, binfo->scrny);  for (;;) {  io\_hlt();  }  } |

访问结构体的2种方式：

**(\*binfo).scrnx**

**binfo->scrnx**

## 显示字符

void putfont8(char \*vram, int xsize, int x, int y, char c, char \*font)

{

int i;

char \*p, d /\* data \*/;

for (i = 0; i < 16; i++) {

p = vram + (y + i) \* xsize + x;

d = font[i];

if ((d & 0x80) != 0) { p[0] = c; } // dealed by each pixel.

if ((d & 0x40) != 0) { p[1] = c; }

if ((d & 0x20) != 0) { p[2] = c; }

if ((d & 0x10) != 0) { p[3] = c; }

if ((d & 0x08) != 0) { p[4] = c; }

if ((d & 0x04) != 0) { p[5] = c; }

if ((d & 0x02) != 0) { p[6] = c; }

if ((d & 0x01) != 0) { p[7] = c; }

}

return;

}

## 增加osask字体

程序：harib02e, harib02f。

hankaku.txt中存储了256个ASCII码的字体信息。每个字符16byte。使用其ASCII码为key。程序加载时使用：**extern char hankaku[4096]**。访问时某个字符的字体数据地址为：

**hankaku + ’char’ \* 16**。如A的数据为：**hankaku + ’A’ \* 16**；或者**hankaku + 0x41 \* 16。**



程序：

void HariMain(void)

{

struct BOOTINFO \*binfo = (struct BOOTINFO \*) 0x0ff0;

init\_palette();

init\_screen(binfo->vram, binfo->scrnx, binfo->scrny);

putfonts8\_asc(binfo->vram, binfo->scrnx, 8, 8, COL8\_FFFFFF, "ABC 123");

putfonts8\_asc(binfo->vram, binfo->scrnx, 31, 31, COL8\_000000, "Haribote OS.");

putfonts8\_asc(binfo->vram, binfo->scrnx, 30, 30, COL8\_FFFFFF, "Haribote OS.");

for (;;) {

io\_hlt();

}

}

void **putfonts8\_asc**(char \*vram, int xsize, int x, int y, char c, unsigned char \*s)

{

extern char hankaku[4096];

for (; \*s != 0x00; s++) {

putfont8(vram, xsize, x, y, c, hankaku + \*s \* 16);

x += 8;

}

return;

}

void putfont8(char \*vram, int xsize, int x, int y, char c, char \*font)

{

int i;

char \*p, d /\* data \*/;

for (i = 0; i < 16; i++) {

p = vram + (y + i) \* xsize + x;

d = font[i];

if ((d & 0x80) != 0) { p[0] = c; }

if ((d & 0x40) != 0) { p[1] = c; }

if ((d & 0x20) != 0) { p[2] = c; }

if ((d & 0x10) != 0) { p[3] = c; }

if ((d & 0x08) != 0) { p[4] = c; }

if ((d & 0x04) != 0) { p[5] = c; }

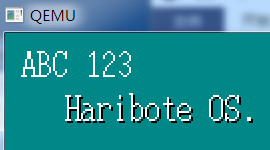
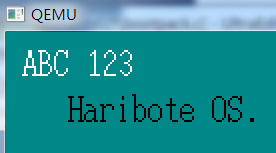
if ((d & 0x02) != 0) { p[6] = c; }

if ((d & 0x01) != 0) { p[7] = c; }

}

return;

}

左图为正常输出，右图为主程序中去掉对COL8\_FFFFFF的输出。

## 显示变量值

为了显示变量值，需要使用sprintf函数，用于**把格式化的数据写入某个**[**字符串**](http://baike.baidu.com/view/56072.htm)[**缓冲区**](http://baike.baidu.com/view/266782.htm)。因此需要在主函数开头，增加库函数：

**#include <stdio.h>**

在主函数中，打印输出末尾增加如下3行：

char s[40];

sprintf(s, "scrnx = %d", binfo->scrnx);

putfonts8\_asc(binfo->vram, binfo->scrnx, 16, 64, COL8\_FFFFFF, s);

结果：

|  |  |
| --- | --- |
|  | 整体的**工作方式**是：  1）sprintf将要打印的内容到某个字符串缓冲区  2）再将此缓冲区的内容copy到显存中。 |

### sprintf

头文件：[**stdio.h**](http://baike.baidu.com/view/538727.htm)

原型 ： **int sprintf( char \*buffer, const char \*format, [ argument] … );**

参数列表

buffer：[char](http://baike.baidu.com/view/1006519.htm)型指针，指向将要写入的字符串的缓冲区。

format：格式化字符串。

%d:123; %5d: 123; %05d:00123

%x:abc; %5x: abc; %05x:00abc

%X:ABC; %5X: ABC; %05X:00ABC

[argument]...：可选参数，可以是任何类型的数据。

返回值：字符串长度（[strlen](http://baike.baidu.com/subview/736226/5092486.htm)）

eg:

n=**sprintf**(buffer,"%d plus %d is %d",a,b,a+b);/\*赋予数值\*/

printf("[%s]is a string %d chars long\n",buffer,n);/\*“格式输出函数”\*/

## 鼠标

void init\_mouse\_cursor8(char \*mouse, char bc)

{

static char cursor[16][16] = {

"\*\*\*\*\*\*\*\*\*\*\*\*\*\*..", // 边界用\*，实体用O，空白用…

"\*OOOOOOOOOOO\*...",

} // according cursor[16][16] to gen output info at char \*mouse

// \* -- > black; O -- > white; . -- > back-color

void HariMain(void)

{

struct BOOTINFO \*binfo = (struct BOOTINFO \*) 0x0ff0;

char s[40], mcursor[256];

int mx, my;

init\_palette();

init\_screen8(binfo->vram, binfo->scrnx, binfo->scrny);

**mx = (binfo->scrnx - 16) / 2; // center for mouse, 16 is length of mouse, 152**

**my = (binfo->scrny - 28 - 16) / 2; // 28 is scroll bar, 16 is mouse, 78**

init\_mouse\_cursor8(mcursor, COL8\_008484);

putblock8\_8(binfo->vram, binfo->scrnx, 16, 16, mx, my, mcursor, 16);

sprintf(s, "(%d, %d)", mx, my);

putfonts8\_asc(binfo->vram, binfo->scrnx, 0, 0, COL8\_FFFFFF, s);

for (;;) {

io\_hlt();

}

}

void putblock8\_8(char \*vram, int vxsize, int pxsize,

int pysize, int px0, int py0, char \*buf, int bxsize)

{ // 将buf[16\*16]里的鼠标图，写到显存中(px0, py0) 像素为起点的位置。

int x, y;

for (y = 0; y < pysize; y++) {

for (x = 0; x < pxsize; x++) {

vram[(py0 + y) \* vxsize + (px0 + x)] = buf[y \* bxsize + x];

} //复制像素空间，基本上都是**(y0+y)\*xsize \*(x0+x)**

}

return;

}



## GDT and IDT

### 概念

**GDT**

将内存分成多人块（block），每一块的起始地址都看作0处理。

指令：**MOV AL, [EBX]** or **MOV AL, [DS:EBX]**

16位模式：数据段寄存器DS乘16再加BX中的偏移量，

32位模式：数据段寄存器DS存的段基址，

分段需要知道段的大小，段的起始地址，以及段的管理属性。需要**8字节**。但指定段的register只有16位。并且在指定段号与段关系时只能使用**13位**，即2^13=8192个段。

因此要描述分段信息，共需要空间：8192\*8＝2^13 \* 2^3=2^16=**64K**。此64K地址即**GDT**（global (segment) descriptor table），全局段号记录表。此**GDT的首地址及有效个数**放在**GDTR**（global (segment) descriptor table register）寄存器中。

**IDT**

IDT（interrupt descriptor table）中断记录表。

主动查询 VS 被动中断。

IDT记录了255个中断号码与调用函数的对应关系。

时钟周期，指令周期，机器周期。

1 **时钟周期**: 一个**时钟脉冲**所需要的时间，通常称为**节拍脉冲或T周期**。

2 CPU**周期** : 又称**机器周期**，指CPU完成一个基本操作所需的时间。用**从内存读取一条指令字的最短时间来定义**。

3 **指令周期** : CPU**从内存取出一条指令并执行这条指令的时间总和**。通常包括取指令，分析指令，执行指令。至少需要2个机器周期。

示例：2GHz的CPU主频，前端总线为800MHz，和DDR内存主频为400MHz。则

**时钟周期：**

1/2ghz=0.5ns

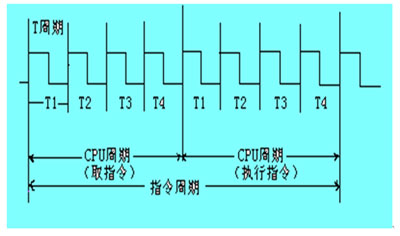
**CPU周期：**

400M 的DDR内存其等效频率为200MHz（脉冲上下沿皆可传数据），

而前端总线是800M，因此内存的速度慢，使用双通道的情况下，2条DDR 400M的内存可以刚好匹配800M的前端总线。此时CPU周期为：1/800mhz=**1.25us＝1250ns**。

**指令周期**：

最快也需要**2.5us**。即1秒可执行1/2.5us=1000/2.5K=**40W条内存中的指令**。



假设长按键盘，1秒能输入50个字符，单次按键1/50=20ms，如上分析的指令周期2.5us也已经可以执行20ms/2.5us=20/2.5K=8K条内存指令。

### 代码

struct SEGMENT\_DESCRIPTOR {

short limit\_low, base\_low;

char base\_mid, access\_right;

char limit\_high, base\_high;

};

struct GATE\_DESCRIPTOR {

short offset\_low, selector;

char dw\_count, access\_right;

short offset\_high;

};

void init\_gdtidt(void);

void set\_segmdesc(struct SEGMENT\_DESCRIPTOR \*sd, unsigned int limit, int base, int ar);

void set\_gatedesc(struct GATE\_DESCRIPTOR \*gd, int offset, int selector, int ar);

void load\_gdtr(int limit, int addr);

void load\_idtr(int limit, int addr);

**void init\_gdtidt(void)**

{

struct SEGMENT\_DESCRIPTOR \*gdt = (struct SEGMENT\_DESCRIPTOR \*) 0x00270000;

struct GATE\_DESCRIPTOR \*idt = (struct GATE\_DESCRIPTOR \*) 0x0026f800;

int i;

/\* GDT初始化 \*/

for (i = 0; i < 8192; i++) { //gdt是结构体指针，地址基数是8字节，+1则是+8byte

set\_segmdesc(gdt + i, 0, 0, 0);

}

set\_segmdesc(gdt + 1, 0xffffffff, 0x00000000, 0x4092);

set\_segmdesc(gdt + 2, 0x0007ffff, 0x00280000, 0x409a);

load\_gdtr(0xffff, 0x00270000);

/\* IDT初始化 \*/

for (i = 0; i < 256; i++) {

set\_gatedesc(idt + i, 0, 0, 0);

}

load\_idtr(0x7ff, 0x0026f800);

return;

}

set\_segmdesc(gdt + 1, 0xffffffff, 0x00000000, 0x4092);

//段号为1的段，地址为0，上限为4G，表示CPU能管理的全部内存空间。

set\_segmdesc(gdt + 2, 0x0007ffff, 0x00280000, 0x409a);

//段号为2的段，地址为0x280000，大小为2^19＝512KB，为bootpack.hrb准备的

load\_gdtr(0xffff, 0x00270000);

//向270000地址写0xFFFF，最终写到GDTR寄存器。

至此，haribote.sys因为加了4096的字体，已经7632byte了。

# (1.26)第六天 分割编译与中断处理

## 分割源文件

优点：

按处理内容分类，方便查找

多个小文件优于单个大文件

通过优化Makefile，可提高编译速度



以上是整体的编译关系，对于C代码，由之前的一个分隔成三个。



## 理解dsctbl.c

这是第五天留下的代码，没有解释。

### load\_gdtr()

init\_gdtidt(void)函数的前面已经简要说明，再补充如下实现在naskfunc.nas中的函数：

|  |  |
| --- | --- |
| **\_load\_gdtr**:  ; void load\_gdtr(int limit, int addr);  MOV AX,[ESP+4] ; limit  **MOV [ESP+6],AX**  LGDT [ESP+6]  RET |  |

此函数将16位的段上限limit与32位的地址addr赋值给GDTR的48位register。只能使用LGDT将一个内存地址开始的6个字节赋值给GDTR。

### set\_segmdesc()

|  |  |
| --- | --- |
| struct SEGMENT\_DESCRIPTOR {  short limit\_low, base\_low;  char base\_mid, access\_right;  char limit\_high, base\_high;  };  void set\_segmdesc(  struct SEGMENT\_DESCRIPTOR \*sd,  unsigned int limit,  int base, int ar )  {  if (limit > 0xfffff) {  ar |= 0x8000; /\* G\_bit = 1 \*/  limit /= 0x1000;  }  sd->base\_low = base & 0xffff;  sd->base\_mid = (base >> 16) & 0xff;  sd->base\_high = (base >> 24) & 0xff;  sd->limit\_low = limit & 0xffff; sd->limit\_high = ((limit >> 16) & 0x0f)  | ((ar >> 8) & 0xf0);  sd->access\_right = ar & 0xff;  return;  } | 根据CPU的规格，需要将段的信息归结为8个字节。信息包括：  段大小limit； 段起始地址base，  访问权限属性ar。  为此定义了结构体SEGMENT\_DESCRIPTOR。  因为一共只有8字节来存储段信息，而段地址已经用掉4个。故**段长与访问权限共用4字节**。而最大空间4G需要使用32bit，**为了保证12位属性，段长只有20位**。  段长20位最大空间为1MB。为了标示更大空间，设置了Gbit (Granularity)位，表示段长的单位为4K。这样就可以表示4GB了。  左为当limit在于1MB时处理之。  段地址。  Base为32位整数；但CPU里需要将其分成三段：低2字节，次1字节，高1 字节。这是为了与80286系统兼容而增加的  低16bit段长；  低4bit为段长的高4bit。高4bit为访问权限属性。  访问权限属性。 |

**访问权限access\_right**共12位，低8位在sd->access\_right，高4位在sd->limit\_high的高4位中。函数参数ar虽在C中定义成int，实际上是当成16位整数，且8－11位是为0的。即：XXXX**0000**XXXXXXXX形式。

最高4位在80286中是没有的，386中增加的，内容为GD00，G即上述之Gbit位，1表示段长单位为4K，0表示意为为1B。D＝1表示32位模式。

低8位为0X9A时，表示系统模式，为0XFA时为应用模式。

根据如下定义：

#define ADR\_GDT 0x00270000

#define ADR\_BOTPAK 0x00280000

#define LIMIT\_BOTPAK 0x0007ffff

#define AR\_CODE32\_ER 0x409a

struct SEGMENT\_DESCRIPTOR \*gdt = (struct SEGMENT\_DESCRIPTOR \*) **ADR\_GDT**;

可得下表set\_segmdesc的实际参数与sd的值：

|  |  |
| --- | --- |
| （gdt + 2 = 0x00270010 // sd  LIMIT\_BOTPAK = 0x0007ffff // limit  ADR\_BOTPAK = 0x00280000 // base  AR\_CODE32\_ER = 0x409a ） // ar  sd->base\_low = base & 0xffff;  sd->base\_mid = (base >> 16) & 0xff;  sd->base\_high = (base >> 24) & 0xff;  sd->limit\_low = limit & 0xffff;  sd->limit\_high = ((limit >> 16) & 0x0f)  | ((ar >> 8) & 0xf0);  sd->access\_right = ar & 0xff; | Limit=0x7ffff 小于1MB，不修改。  0x0000  0x28  0x00  0xffff  (0x0**7** & 0x0f) | (0x**4**0 & 0xf0) = **0x47**  0x409a & 0xff = **0x9a** |

## PIC

programmable interrupt controller，使用8259A。从的出口连主的IRQ2号脚。

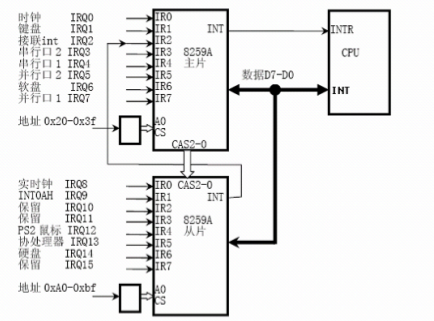
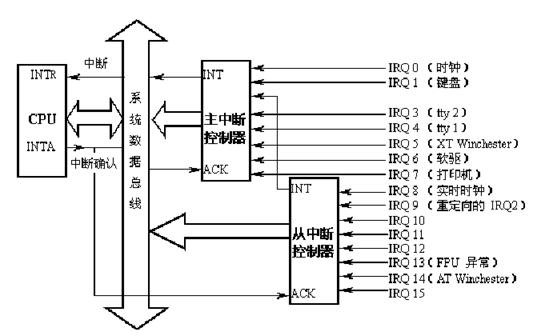
IMR： interrupt mask register. 8bit for 8 port. 1 for mask.

ICW：initial control word.

ICW2是可以设定的，用来表示15个中断对应的中断号。

ICW3是主从连接。对主而言，某位为1则表示该port对应了一个从PIC。或全为1，则表示8个从，共有64个port。此处设ICW3为0X04.

对从PIC而言，使用3位，表示从与主的几号port相连。

上为示意图，实际INT与设备关系未必如图。

当有中断时，CPU响应中断时会命令PIC发送2字节数据，此时PIC送出“0xCD 0xXX”。这两个字节的内容，与CPU从内存读的程序一样，CPU把他当成是可执行语句，并执行他，而0XCD对应的是调用BIOS的INT指令，如程序中写”INT 0x10”，会被编译成0XCD 0X10，故CPU上了PIC的当。

INT 0x00-0x1f这32个用于CPU的系统保护通知。

INT 0x20-0xXX供外部中断使用。

|  |
| --- |
| void init\_pic(void)  { // PIC0\_\* for master, PIC1\_\* for slave.  io\_out8(PIC0\_IMR, 0xff ); /\* 禁止所有中断 \*/  io\_out8(PIC1\_IMR, 0xff ); /\* 禁止所有中断 \*/  io\_out8(PIC0\_ICW1, 0x11 ); /\* edge trigger mode \*/  io\_out8(PIC0\_ICW2, 0x20 ); /\* IRQ0-7 use INT20-27 \*/  io\_out8(PIC0\_ICW3, 1 << 2); /\* PIC1 collect to IRQ2 of master \*/  io\_out8(PIC0\_ICW4, 0x01 ); /\* no buffer mode \*/  io\_out8(PIC1\_ICW1, 0x11 ); /\* edge trigger mode \*/  io\_out8(PIC1\_ICW2, 0x28 ); /\* IRQ8-15 use INT28-2f \*/  io\_out8(PIC1\_ICW3, 2 ); /\* PIC1 collect to IRQ2 of master \*/  io\_out8(PIC1\_ICW4, 0x01 ); /\* no buffer mode \*/  io\_out8(PIC0\_IMR, 0xfb ); /\* 11111011 PIC1以外全部禁止 \*/  io\_out8(PIC1\_IMR, 0xff ); /\* 11111111 禁止所有中断\*/  return;  } |

## 处理中断

harib03e目录。

IRQ12是鼠标，IRQ1是键盘。

|  |
| --- |
| void inthandler2c(int \*esp)  /\* PS/2键盘中断 \*/  {  struct BOOTINFO \*binfo = (struct BOOTINFO \*) ADR\_BOOTINFO;  boxfill8(binfo->vram, binfo->scrnx, COL8\_000000, 0, 0, 32 \* 8 - 1, 15);  putfonts8\_asc(binfo->vram, binfo->scrnx, 0, 0,  COL8\_FFFFFF, "INT 2C (IRQ-12) : PS/2 mouse");  for (;;) {  io\_hlt();  }  } |

上面这段程序不能返回中断，一直在HLT。中断需要IRETD指令返回，因此需要下面的汇编函数。

**中断返回**

|  |  |
| --- | --- |
| \_asm\_inthandler21:  PUSH ES  PUSH DS  PUSHAD  MOV EAX,ESP  PUSH EAX  MOV AX,SS  MOV DS,AX  MOV ES,AX  CALL \_inthandler21  POP EAX  POPAD  POP DS  POP ES  IRETD | 保存ES  保存DS  保存EAX,ECX,EDX,EBX, ESP, EBP,ESI,EDI  再保存当前ESP  赋值DS为SS  赋值ES为SS  丢弃暂时有ESP，  恢复EAX,ECX,EDX,EBX, ESP, EBP,ESI,EDI  恢复DS  恢复ES |

有两处不明白：

1. 中断时的处理，DS,ES为什么要写SS相同？
2. set\_segmdesc()函数没有详细解释其8个字节的意义，大致认识第一个参数是中断号及对应的向量空间起址，第二个是函数地址，也是偏移量，第三个是段号，本次设置为2，最后是权限属性。

# (1.29)第七天 FIFO & MOUSE

按下和松开键盘都有中断和输入。如下是流程关系及效果图。

|  |  |
| --- | --- |
|  | 响应键盘中断：io\_out8(PIC0\_OCW2, 0X61) // 响应时输出0x60 + int\_no  获取键盘输入：io\_in8(0X0060) |

## FIFO缓冲区

|  |  |
| --- | --- |
| **fifo.c:** | void fifo8\_init(struct FIFO8 \*fifo, int size, unsigned char \*buf)  int fifo8\_put(struct FIFO8 \*fifo, unsigned char data)  int fifo8\_get(struct FIFO8 \*fifo)  int fifo8\_status(struct FIFO8 \*fifo) |
| **int.c:** | struct FIFO8 keyfifo;  void inthandler21(int \*esp)  unsigned char data;  io\_out8(PIC0\_OCW2, 0x61); /\* \*/  data = io\_in8(PORT\_KEYDAT);  fifo8\_put(&keyfifo, data);  return; |
| **bootpack.c:** | void HariMain(void)  { extern struct FIFO8 **keyfifo**;  char **keybuf**[32];  **fifo8\_init(&keyfifo, 32, keybuf);**  for (;;) {  io\_cli();  if (fifo8\_status(&keyfifo) == 0) {  io\_stihlt();  } else {  i = fifo8\_get(&keyfifo);  io\_sti();  sprintf(s, "%02X", i);  boxfill8(binfo->vram, binfo->scrnx, COL8\_008484, 0, 16, 15, 31);  putfonts8\_asc(binfo->vram, binfo->scrnx, 0, 16, COL8\_FFFFFF, s);  }  } |

### Makefile关系图



## 获取鼠标数据

### 激活鼠标

鼠标相对于键盘，是新兴一族。其中断号是IRQ12。鼠标的与键盘使用相同的控制电路，并且，必须要由CPU执行激活鼠标的指令，才允许鼠标控制器，以及鼠标产生中断。

鼠标控制电路的激活，需要先等待键盘控制电路（keyboard controller KBC）做好准备——发送控制指令。即KBC可以接受CPU指令时，会向设备号码0X0064处地址的bit1设置为0（为1则未就绪），故有如下函数：

|  |  |
| --- | --- |
| void wait\_KBC\_sendready(void)  { /\* wait KBC ready: bit1 == 0 \*/  for (;;) {  if ((io\_in8(0x64) & 0x10) == 0) {  break;  }  }  return;  } | void init\_keyboard(void)  {  /\* init KBC \*/  wait\_KBC\_sendready();  io\_out8(0x64, 0x60);  wait\_KBC\_sendready();  io\_out8(0x60, 0x47); // set mouse mode  return;  } |

经过如上设置后，KBC即已经准备就绪。可以激活mouse了。

|  |  |
| --- | --- |
| void enable\_mouse(struct MOUSE\_DEC \*mdec)  { /\* activiate mouse \*/  wait\_KBC\_sendready();  io\_out8(0x64 ,0xD4);  wait\_KBC\_sendready();  io\_out8(0x60,0xF4);  mdec->phase = 0; /\* KBC return ACK(0xfa) \*/  return;  } | 当往KBC发送指令0XD4时，下一个数据会自动发送给鼠标。  通过此方式来激活鼠标。激活后，鼠标会返回0XFA做为ACK。此ACK会引起鼠标中断。 |

### 读取鼠标数据

代码：

|  |  |
| --- | --- |
| bootpack.c: | extern struct FIFO8 keyfifo, mousefifo;  char s[40], mcursor[256], keybuf[32], mousebuf[128];  fifo8\_init(&keyfifo, 32, keybuf);  fifo8\_init(&mousefifo, 128, mousebuf);  init\_keyboard();  init\_mouse\_cursor8(mcursor, COL8\_008484);  enable\_mouse();  for (;;) {  io\_cli();  if (fifo8\_status(&keyfifo) + fifo8\_status(&mousefifo) == 0) {  io\_stihlt();  } else {  if (fifo8\_status(&keyfifo) != 0) {  i = fifo8\_get(&keyfifo);  io\_sti();  sprintf(s, "%02X", i);  boxfill8(binfo->vram, binfo->scrnx, …);  putfonts8\_asc(binfo->vram, binfo->scrnx,…);  } else if (fifo8\_status(&mousefifo) != 0) {  i = fifo8\_get(&mousefifo);  io\_sti();  sprintf(s, "%02X", i);  boxfill8(binfo->vram, binfo->scrnx, …);  putfonts8\_asc(binfo->vram, binfo->scrnx,…);  }  }  } |
| Int.c | struct FIFO8 mousefifo;  void inthandler2c(int \*esp)  { /\* PS/2鼠标中断 \*/  unsigned char data;  /\*通知PIC1 IRQ-12的中断已受理，从PIC的第4号INT \*/  io\_out8(PIC1\_OCW2, **0x64**);  /\*通知PIC0 IRQ-2 的中断已受理，主PIC的第2号INT\*/  io\_out8(PIC0\_OCW2, **0x62**);  data = io\_in8(PORT\_KEYDAT);  fifo8\_put(&mousefifo, data);  return;  } |

结果：

可以输入键盘值，

可以移动鼠标以得到输入值，但鼠标本身并未移动。

# (1.30)第八天 鼠标控制与32位模式

## 鼠标数据解读

鼠标在被激活后，首先会发送0XFA，表示已经激活。此数据需要丢弃。然后发来的数据是以3个字节为一组的。使用phase表示阶段（0表示等待0xfa，1－3分别表示3个字节），mouse\_dbuf[0-2]存储发送来的数据。

mouse\_dbuf[1]：表示鼠标X轴（左右）移动的数据

mouse\_dbuf[2]：表示鼠标Y轴（上下）移动的数据

mouse\_dbuf[0]：比较复杂。

1. 移动时：高4bit为0-3，低4bit为8且不变。
2. 点击时：高4bit未用 ，低4bit为8-F（bit3为1，**bit0-2表示左中右点击**）

bootpack.c代码：

|  |  |
| --- | --- |
| struct MOUSE\_DEC {  unsigned char buf[3], phase;  int x, y, btn;  };  int **mouse\_decode**(struct MOUSE\_DEC \*mdec, unsigned char dat)  {  if (mdec->phase == 0) {  /\* wait mouse ACK(0xfa) \*/  if (dat == 0xfa) {  mdec->phase = 1;  }  return 0;  }  if (mdec->phase == 1) {  **if ((dat & 0xc8) == 0x08)** {  mdec->buf[0] = dat;  mdec->phase = 2;  }  return 0;  }  if (mdec->phase == 2) {  mdec->buf[1] = dat;  mdec->phase = 3;  return 0;  }  if (mdec->phase == 3) {  mdec->buf[2] = dat;  mdec->phase = 1;  mdec->btn = mdec->buf[0] & 0x07;  mdec->x = mdec->buf[1];  mdec->y = mdec->buf[2];  if ((mdec->buf[0] & 0x10) != 0) {  mdec->x |= 0xffffff00;  }  if ((mdec->buf[0] & 0x20) != 0) {  mdec->y |= 0xffffff00;  }  mdec->y = - mdec->y;  return 1;  }  return -1;  }  // HariMain 节选 } else if (fifo8\_status(&mousefifo) != 0) {  i = fifo8\_get(&mousefifo);  io\_sti();  if (mouse\_decode(&mdec, i) != 0) {  sprintf(s, "[lcr %4d %4d]", mdec.x, mdec.y);  if ((mdec.btn & 0x01) != 0) { s[1] = 'L'; }  if ((mdec.btn & 0x02) != 0) { s[3] = 'R'; }  if ((mdec.btn & 0x04) != 0) { s[2] = 'C'; }  boxfill8(binfo->vram, binfo->scrnx, COL8\_008484,  32, 16, 32 + 15 \* 8 - 1, 31);  putfonts8\_asc(binfo->vram, binfo->scrnx,  32, 16, COL8\_FFFFFF, s );  }  } | /\* move: high 4bit=0-3, then & 0xc = 0  Click : bit3=1, bit0-2 some bit is 1, then & 0x8 = 8 \*/  // bit0-2  1 byte include bit0-bit7  Bit4  Only 1 byte. make bit31-8 as 0.  Bit5  鼠标的y方向是向上，显存的y方向日向下的。  Mouse有一组3字节数据需要处理了  点击了左键  点击了右键  点击了中间键 |

## 移动鼠标

代码：

|  |  |
| --- | --- |
| // HariMain 节选  } else if (fifo8\_status(&mousefifo) != 0) {  i = fifo8\_get(&mousefifo);  io\_sti();  if (mouse\_decode(&mdec, i) != 0) {  sprintf(s, "[lcr %4d %4d]", mdec.x, mdec.y);  if ((mdec.btn & 0x01) != 0) { s[1] = 'L'; }  if ((mdec.btn & 0x02) != 0) { s[3] = 'R'; }  if ((mdec.btn & 0x04) != 0) { s[2] = 'C'; }  boxfill8(binfo->vram, binfo->scrnx, COL8\_008484,  32, 16, 32 + 15 \* 8 - 1, 31);  putfonts8\_asc(binfo->vram, binfo->scrnx,  32, 16, COL8\_FFFFFF, s );  // move mouse  boxfill8(binfo->vram, binfo->scrnx, COL8\_008484,  mx, my, mx + 15, my + 15);  mx += mdec.x; my += mdec.y;  if (mx < 0) { mx = 0; }  if (my < 0) { my = 0; }  if (mx > binfo->scrnx - 16) {  mx = binfo->scrnx - 16; }  if (my > binfo->scrny - 16) {  my = binfo->scrny - 16; }  sprintf(s, "(%3d, %3d)", mx, my);  boxfill8(binfo->vram, binfo->scrnx, COL8\_008484,  0, 0, 79, 15);  putfonts8\_asc(binfo->vram, binfo->scrnx,  0, 0, COL8\_FFFFFF, s);  putblock8\_8(binfo->vram, binfo->scrnx, 16, 16,  mx, my, mcursor, 16);  }  } | /\* 隐藏鼠标 \*/  /\* 隐藏坐标 \*/  /\* 显示坐标 \*/  /\* 显示鼠标 \*/ |

运行效果。

没有进行显示的叠加处理。包括下部的任务栏，以及键盘输入值。

|  |  |
| --- | --- |
|  |  |

## 通往32位模式

本处的代码是汇编，没全部看懂。

**内存空间分布图**

0X00000000 – 0X000FFFFF # 1MB

0X00100000 – 0X00267FFF # 用于保存软盘的内容，1440KB

0X00268000 – 0X0026F7FF # 空，30KB

0X0026F800 – 0X0026FFFF # IDT, 2KB

0X00270000 – 0X0027FFFF # GDT, 64B

0X00280000 – 0X002FFFFF # bootpack.hrb，512KB

0X00300000 – 0X003FFFFF # 栈及其他，1MB

0X00400000 – # black

汇编代码，加了注释。

|  |  |
| --- | --- |
| ; 设置显存控制信息  MOV AL,0x13 ; VGA显存320x200x8bit彩色  MOV AH,0x00  INT 0x10  MOV BYTE [VMODE],8 ; 记录画面模式  MOV WORD [SCRNX],320  MOV WORD [SCRNY],200  MOV DWORD [VRAM],0x000a0000  ; 用BIOS取得键盘上各种LED指示断的状态  MOV AH,0x02  INT 0x16 ; keyboard BIOS  MOV [LEDS],AL  ; PIC关闭一切中断  ; 根据AT兼容机的规格，如果要初始化PIC,  ; 必须在CLI之前进行，否则有时会挂机  ; 随后进行PIC的初始化  MOV AL,0xff  OUT 0x21,AL  NOP ; 如果连续执行OUT指令，有时机种无法正常运行  OUT 0xa1,AL  CLI ; 禁止CPU级别的中断  ; above equal to BELOW 3 LINE  ; io\_out8(PIC0\_IMR, 0XFF);  ; io\_out8(PIC1\_IMR, 0XFF);  ; io\_CLI()  ; 为了CPU访问1MB以上的空间，设定A20GATE  CALL waitkbdout ; equal wait\_KBC\_sendready()  MOV AL,0xd1  OUT 0x64,AL ; 先写控制命令  CALL waitkbdout  MOV AL,0xdf ; enable A20  OUT 0x60,AL ; 再写控制数据  CALL waitkbdout ; waiting for OUT finished  ; 电脑早期16位时内存只有1MB，为了兼容，在执行激活指令前，  ; 电路被限制为1MB内存，A20GATE信号线为0时即限制内存，  ; 设置为1时，则放开限制。A20GATE在键盘控制器的附属端口(0xD1),  ; 需要向其写0xDF.  ; 切换到保护模式  [INSTRSET "i486p"] ; 想要使用486指令的声明  LGDT [GDTR0] ; 设定临时GDT  MOV EAX,CR0  AND EAX,0x7fffffff ; 设bit31为0（为了禁止分页）  OR EAX,0x00000001 ;设bit0为1（为了切换到保护模式）  MOV CR0,EAX  JMP pipelineflush  pipelineflush:  MOV AX,1\*8 ; 可读写的段32bit  MOV DS,AX  MOV ES,AX  MOV FS,AX  MOV GS,AX  MOV SS,AX  ; bootpack的转送  MOV ESI,bootpack ; 转送源  MOV EDI,BOTPAK ; 转送目的地  MOV ECX,512\*1024/4  CALL memcpy  ; 磁盘数据最终转送到它本来的位置去  ; 首先从启动扇区开始  MOV ESI,0x7c00 ; 转送源  MOV EDI,DSKCAC ; 转送目的地  MOV ECX,512/4  CALL memcpy  ; 所有剩下的扇区  MOV ESI,DSKCAC0+512 ; 转送源  MOV EDI,DSKCAC+512 ; 转送目的地  MOV ECX,0  MOV CL,BYTE [CYLS]  IMUL ECX,512\*18\*2/4 ; 从柱面数变换为字节数/4  SUB ECX,512/4 ; 减去IPL  CALL memcpy  ; 转送数据是为双字（4字节）为单元的。上面三段相当于如下三个memcpy  ; memcpy(bootpack, BOTPAK, 512\*1024/4);  ; memcpy(0x7c00, DSKCAC, 512/4);  ; memcpy(DSKCAC0+512, DSKCAC+512, cyls\*512\*18\*2/4 - 512/4);  ; 必须由asmhead来完成的工作，至此全部完毕  ; 以后交由bootpack来完成  ; bootpack的启动  MOV EBX,BOTPAK  MOV ECX,[EBX+16]  ADD ECX,3 ; ECX += 3;  SHR ECX,2 ; ECX /= 4;  JZ skip ; 没有要转送的数据时  MOV ESI,[EBX+20] ; 转送源  ADD ESI,EBX  MOV EDI,[EBX+12] ; 转送目的地  CALL memcpy  skip:  MOV ESP,[EBX+12] ; 栈初始值  JMP DWORD 2\*8:0x0000001b  waitkbdout:  IN AL,0x64  AND AL,0x02  IN AL,0x60 ; 空读，以清空数据授受缓冲区的垃圾数据  JNZ waitkbdout ; AND结果不为0，就跳到waitkbdout  RET  memcpy:  MOV EAX,[ESI]  ADD ESI,4  MOV [EDI],EAX  ADD EDI,4  SUB ECX,1  JNZ memcpy ; 减法结果不为0，则继续memcpy  RET  ALIGNB 16  GDT0:  RESB 8 ; 空值选择器  DW 0xffff,0x0000,0x9200,0x00cf ; 可读写的段32位  DW 0xffff,0x0000,0x9a28,0x0047 ; 可执行的段32位（bootpack用）  DW 0  GDTR0:  DW 8\*3-1  DD GDT0  ALIGNB 16  bootpack: |  |

# (2.1)第九天 内存管理

整理源文件，增加keyboard.c, mouse.c文件：



## 内存容量检查

方法：关掉CPU的高速cache，让CPU与MEM直接交互。因此需要区分是不是486及以上机型。

在检查内存时：

先保存内容的值，再写一个值，

将内存值取反，再读出来判断是否取反成功；

将内存值取反，再读出来判断是否还原了。

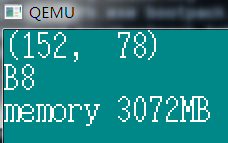
重复每4KB的最后4个字节。

i = memtest(0x00400000, 0xbfffffff) / (1024 \* 1024);

sprintf(s, "memory %dMB", i);

putfonts8\_asc(binfo->vram, binfo->scrnx, 0, 32, COL8\_FFFFFF, s);

不过，这样写的原始代码编译后，对qemu设定的32MB内存，显示却是：



因为上述的2次取反及赋值，被编译器认为没有实际修改。因此优化掉了。因此此段代码改用汇编完成。

重点是下面的代码不明白。MOV EAX,**[ESP+12+4]** ; i = start;

|  |  |
| --- | --- |
| \_memtest\_sub: ; unsigned int memtest\_sub(unsigned int start, unsigned int end)  PUSH EDI ;  PUSH ESI  PUSH EBX  MOV ESI,0xaa55aa55 ; pat0 = 0xaa55aa55;  MOV EDI,0x55aa55aa ; pat1 = 0x55aa55aa;  MOV EAX,[ESP+12+4] ; i = start;  mts\_loop:  MOV EBX,EAX  ADD EBX,0xffc ; p = i + 0xffc;  MOV EDX,[EBX] ; old = \*p;  MOV [EBX],ESI ; \*p = pat0;  XOR DWORD [EBX],0xffffffff ; \*p ^= 0xffffffff;  CMP EDI,[EBX] ; if (\*p != pat1) goto fin;  JNE mts\_fin  XOR DWORD [EBX],0xffffffff ; \*p ^= 0xffffffff;  CMP ESI,[EBX] ; if (\*p != pat0) goto fin;  JNE mts\_fin  MOV [EBX],EDX ; \*p = old;  ADD EAX,0x1000 ; i += 0x1000;  CMP EAX,[ESP+12+8] ; if (i <= end) goto mts\_loop;  JBE mts\_loop  POP EBX  POP ESI  POP EDI  RET  mts\_fin:  MOV [EBX],EDX ; \*p = old;  POP EBX  POP ESI  POP EDI  RET |  |

## 内存管理

常用的有位图管理方式，程序比较复杂些；还有种为列表管理方式，记录起始地址及长度。后者分配和释放都很快，只需要一次加法一次减法即可。但其管理即更复杂。特别是空间零散时。

需要注意：1，分配空间以4KB为基数，非不是字节，2）结构体变量可以直接赋值。

# (2.2)第十天 内存管理

# 附录

白的Windows 图片和传真查看器

# 他人笔记

## 高有用

### 1 scusjs

**### 川大计算机，创新班，杨辉 学生，**

**### 支持windows及linux，有git账号。 博客从第3天开始，到19天。**

<http://blog.csdn.net/scusjs?viewmode=contents>

<http://blog.csdn.net/scusjs/article/details/8947918>

# code

<https://github.com/scusjs/MyOS>

https://github.com/scusjs/MyOS.git

### 2 cherishsir

**### ubuntu平台**

**### cherishsir 16天， QQ 交流Group:159336168**

http://segmentfault.com/blog/cherishsir/1190000000609485?page=1

http://segmentfault.com/blog/cherishsir/1190000000609473

<https://github.com/cherishsir/ubuntu230os.git>

**creat OS**

## 参考资料

http://verynix.com/os-dev-book.html

http://wiki.osdev.org/Main\_Page

## 中可用

###　**祝威** 到**15**天。用**vmware.** bitzhuwei@qq.com

http://www.cnblogs.com/bitzhuwei/p/OS-in-30-days-01-hello-bitzhuwei-OS.html

http://www.cnblogs.com/bitzhuwei/default.html?OnlyTitle=1

### 到6天，用virtualBox

http://www.cnblogs.com/lastavengers/

### Ubuntu 环境

<http://www.cnblogs.com/dyllove98/p/3235363.html>

qq交流群：1**22358078**，此群已经进入，主要是嵌入式开发的。

##第九天链接

http://www.haodaima.net/art/2251717

http://www.tuicool.com/articles/yMVZvy

# 知乎精选

http://www.zhihujingxuan.com/19197.html

一个操作系统的实现

### JamesM's kernel development tutorials

http://www.jamesmolloy.co.uk/tutorial\_html/