

Day 18

我们今天来丰富一下我们的命令行工具，让他成为一个真正可用的命令行工具。

首先我们发现我们的多个窗口的光标是同时闪烁的，这样是不符合我们平常使用的系统的特性的，也不美观，所以我们只允许获得焦点的窗体上的光标闪烁。

我们先从harimain开始修改。

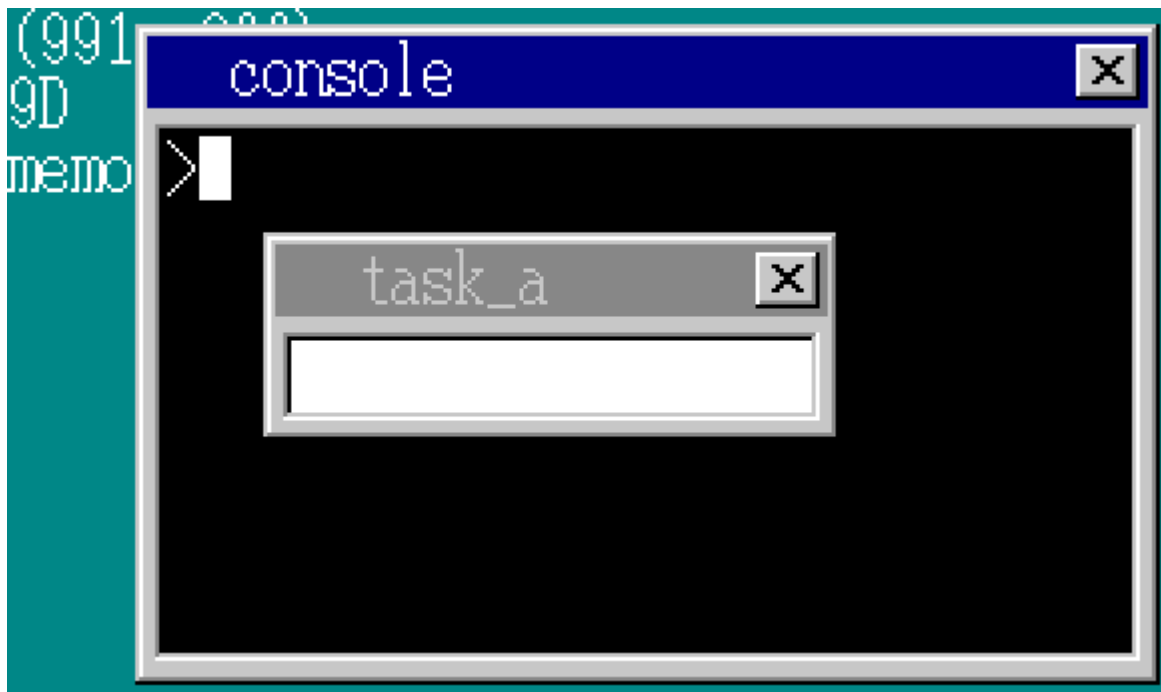
我们的主要思路是利用光标颜色的负数范围，来表明我们不想打印光标。

换句话说，如果我们的cursor_c是-1，那么我们就打印光标。

除此之外，我们还要注意，在我们切换打印光标的颜色的时候，一定要先判断一下是否大于等于0，否则我们的-1就会被改掉。

另一种比较直观的思路是另设一个变量，来标记我们是否需要打印光标。作者的做法好处在于我们少开了一个变量，充分利用现有变量的值域空间，节省内存。

```
if (i == 256 + 0x0f) { /* Tab键*/
    if (key_to == 0) {
        key_to = 1;
        make_wtitle8(buf_win, sht_win->bysize, "task_a", 0);
        make_wtitle8(buf_cons, sht_cons->bysize, "console", 1);
        cursor_c = -1; /* 不显示光标 */
        boxfill8(sht_win->buf, sht_win->bysize, COL8_FFFFFFFF, cursor_x, 28, cursor_x + 7,
43);
    } else {
        key_to = 0;
        make_wtitle8(buf_win, sht_win->bysize, "task_a", 1);
        make_wtitle8(buf_cons, sht_cons->bysize, "console", 0);
        cursor_c = COL8_000000; /*显示光标*/
    }
    sheet_refresh(sht_win, 0, 0, sht_win->bysize, 21);
    sheet_refresh(sht_cons, 0, 0, sht_cons->bysize, 21);
}
```



可以看到，当焦点在console上时只有一个光标在闪烁，两个光标不会同时闪烁了。

而当焦点不再console上的时候，还是由两个光标在闪烁，这是因为我们目前只修改了harimain，也就是task_a。

我们接着来修改task_cons也就是console。

由于两个task是相对独立的，要修改task_b的cursor_c，就需要想别的办法。我们可以通过fifo将我们想要灭掉光标的信息发送过去我们先将光标开始闪烁定义为2，停止闪烁定义为3。

我们首先需要在harimain对 `Tab` 的处理稍稍修改一下，在修改完自己的cursor_c后，向task_cons.fifo中put一个2或者3。

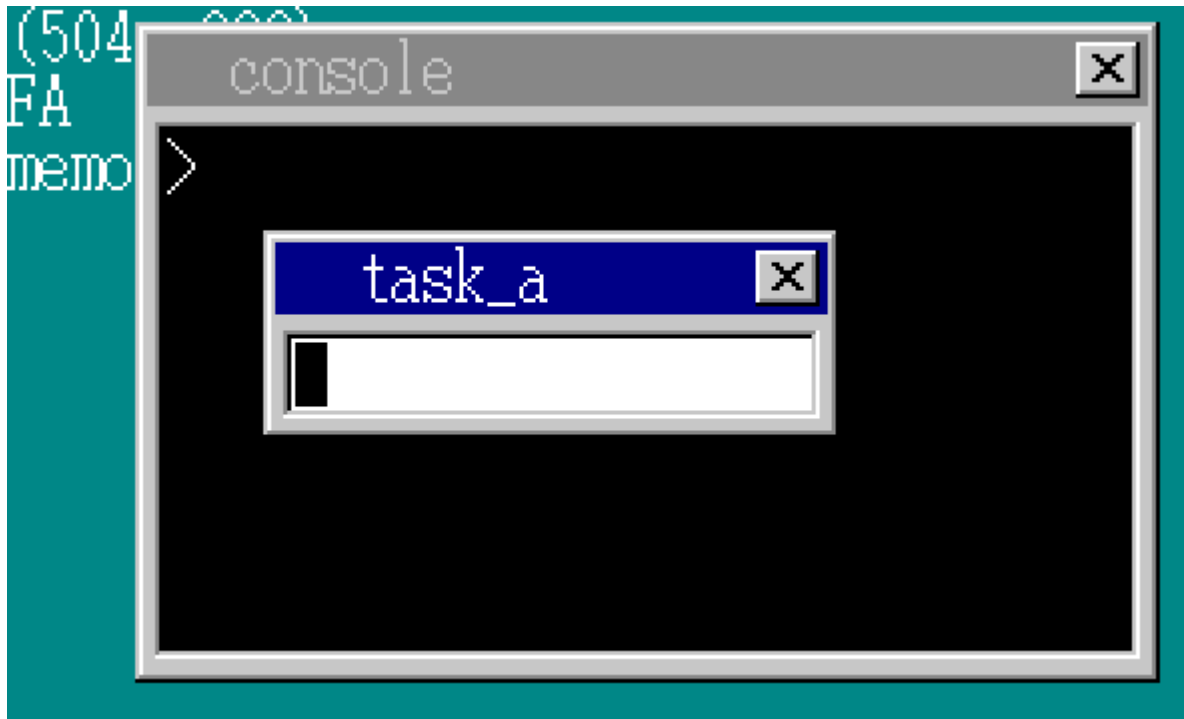
```
cursor_c = -1;
fifo32_put(&task_cons->fifo, 2);
// -----
cursor_c = COL8_000000;
fifo32_put(&task_cons->fifo, 3);
```

对了！还有初始状态！由于任意时刻只能由一个窗体有焦点，所以启动状态也是这样。我们要把task_cons的cursor_c的初始值设置成-1

```
int i, fifobuf[128], cursor_x = 16, cursor_c = -1;
```

就像这样

`make run` 一下



成功了，这次任意时刻只有一个窗体的光标在闪烁了

处理回车键，我们先只让回车键完成换行吧。当harimain捕获到回车且console具有焦点，我们向命令行窗口发送10+256（换行的ASCII是10）

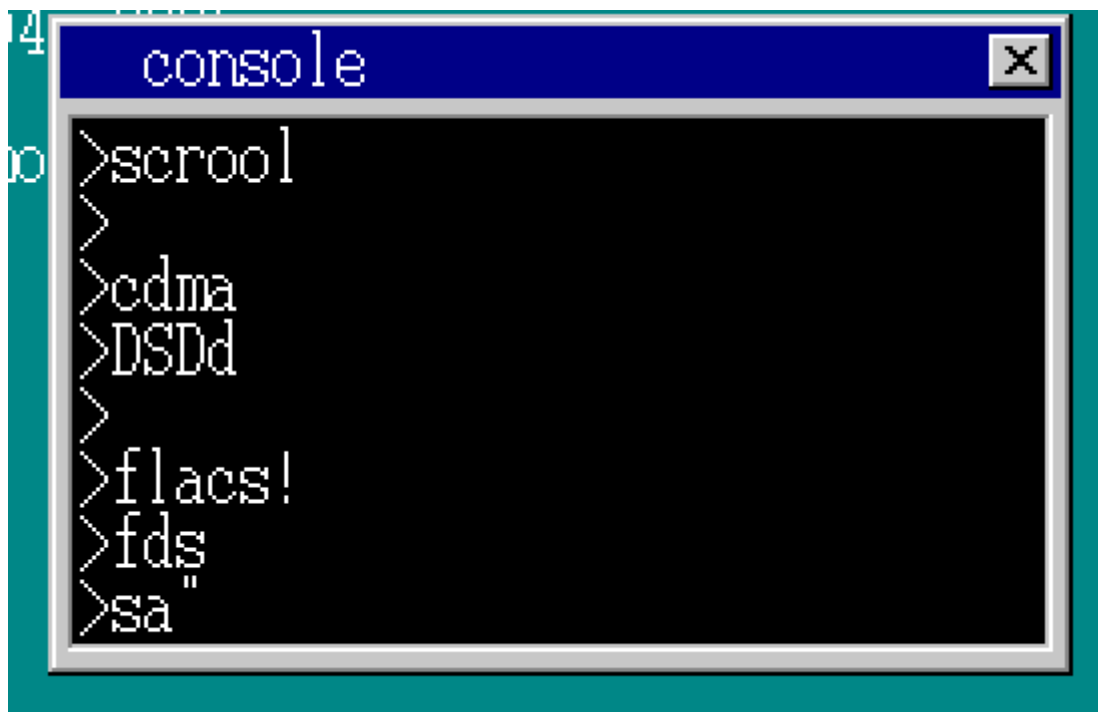
我们想下task_cons应该怎么修改，要完成换行，我们先要把旧光标删除，然后把光标（及输入位置）移到下一行的开始位置。打印一个提示符，打印光标。

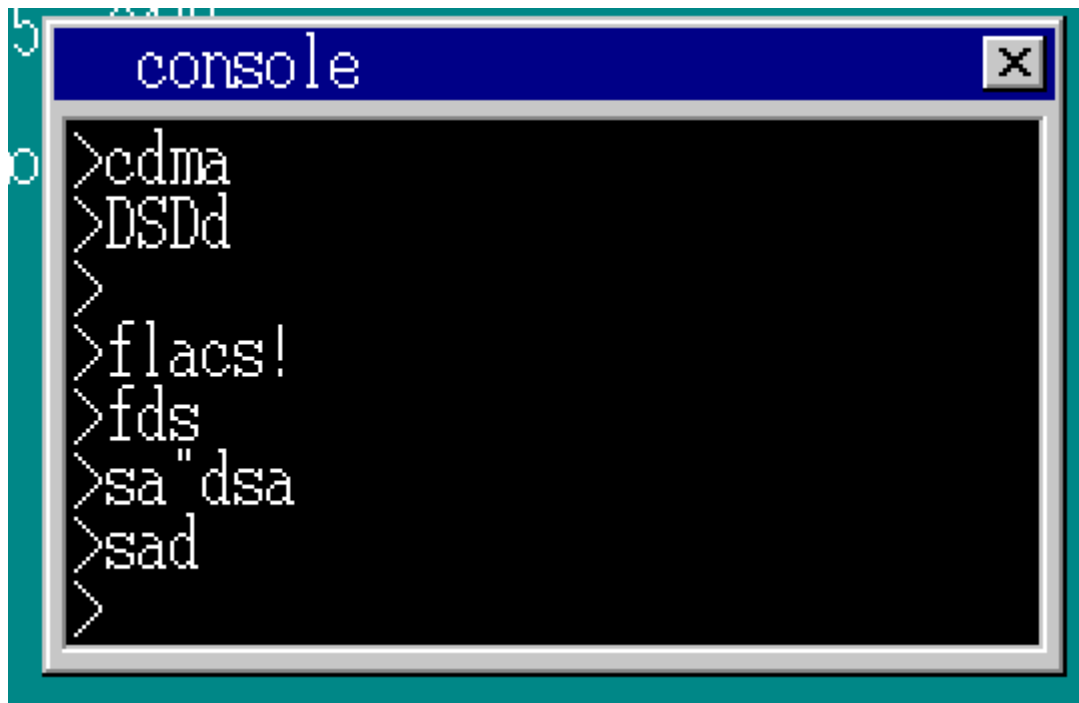
```
if (cursor_y < 28 + 112) {
    putfonts8_asc_sht(sheet, cursor_x, cursor_y, COL8_FFFFFFFF, COL8_000000,
        " ", 1);
    cursor_y += 16;
    putfonts8_asc_sht(sheet, 8, cursor_y, COL8_FFFFFFFF, COL8_000000, ">", 1);
    cursor_x = 16;
}
```



意料之内，打印到最后一行之后不再下滚了。

我们接下来要写向下滚动的逻辑。具体做法是将除第一行之外的每一行的东西都复制到上一行，然后我们用空行覆盖最后一行，就ok了。





工作正常

接下来我们要实现mem命令

mem: 在控制台中打印当前的内存使用情况

我们首先先去掉harimain当中在桌面的内存占用显示，把相关的代码注释掉就好了。

若要获取我们输入的内容，我们必须新开一个数组来记录我们都输入了什么东西。

```
if (cursor_x < 240) {
    s[0] = i - 256;
    s[1] = 0;
    cmdline[cursor_x / 8 - 2] = i - 256; // -2是因为有边框和提示符
    putfonts8_asc_sht(sheet, cursor_x, cursor_y, COL8_FFFFFFFF, COL8_000000, s,
        1);
    cursor_x += 8;
}
```

然后我们在处理回车的时候对cmdline进行一下判断。

在写这个之前，我们先把控制台换行单独拎出来写成一个函数。

```
int cons_newline(int cursor_y, struct SHEET *sheet)
{
    int x, y;
    if (cursor_y < 28 + 112) {
        cursor_y += 16;
    } else {
        for (y = 28; y < 28 + 112; y++) {
```

```

        for (x = 8; x < 8 + 240; x++) {
            sheet->buf[x + y * sheet->bysize] = sheet->buf[x + (y + 16) * sheet->bysize];
        }
    }
    for (y = 28 + 112; y < 28 + 128; y++) {
        for (x = 8; x < 8 + 240; x++) {
            sheet->buf[x + y * sheet->bysize] = COL8_000000;
        }
    }
    sheet_refresh(sheet, 8, 28, 8 + 240, 28 + 128);
}
return cursor_y;
}

```

然后修改task_cons中处理回车键的部分

```

cursor_y = console_newline(cursor_y, sheet); // 换行
if (strcmp(cmdline, "mem") == 0) { //使用strcmp来进行简化
    sprintf(s, "total %dMB", memtotal / (1024 * 1024));
    putfonts8_asc_sht(sheet, 8, cursor_y, COL8_FFFFFFFF, COL8_000000, s, 30);
    cursor_y = cons_newline(cursor_y, sheet);
    sprintf(s, "free %dKB", memman_total(memman) / 1024);
    putfonts8_asc_sht(sheet, 8, cursor_y, COL8_FFFFFFFF, COL8_000000, s, 30);
    cursor_y = cons_newline(cursor_y, sheet);
    cursor_y = cons_newline(cursor_y, sheet);
} else if (cmdline[0] != 0) {
    putfonts8_asc_sht(sheet, 8, cursor_y, COL8_FFFFFFFF, COL8_000000, "Command Illegal.",
sizeof("Command Illegal.));
    cursor_y = cons_newline(cursor_y, sheet);
    cursor_y = cons_newline(cursor_y, sheet);
}
putfonts8_asc_sht(sheet, 8, cursor_y, COL8_FFFFFFFF, COL8_000000, ">", 1); /// 打印提示符
cursor_x = 16;

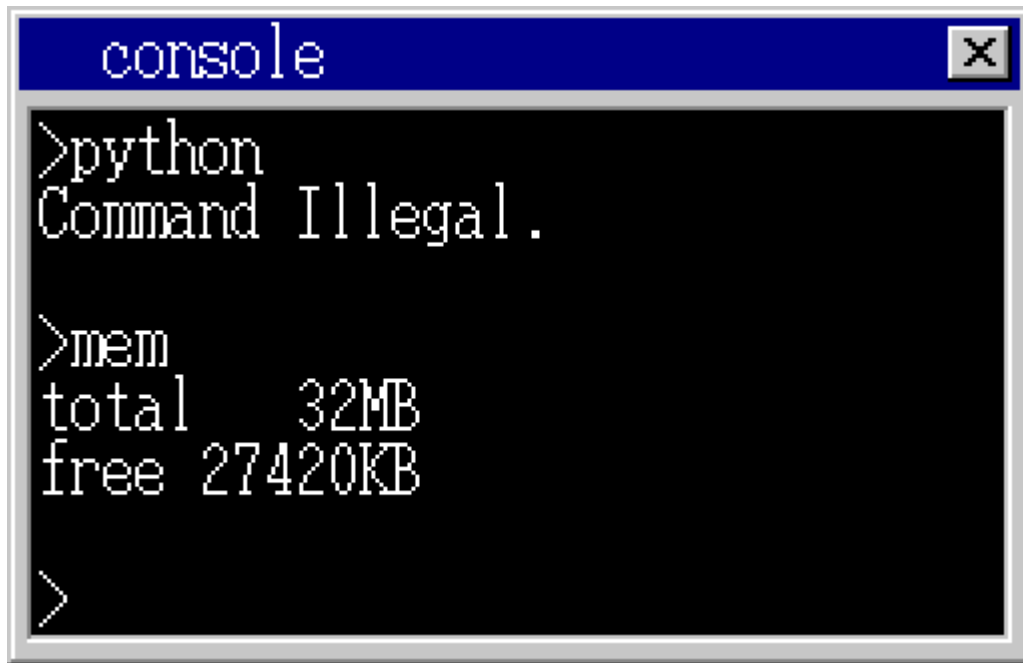
```

注意为了获取memtotal，我们使用了和获取sheet相同的trick

```

// void console_task(struct SHEET *sheet, unsigned int memtotal);
// ----- HariMain(void) -----
*((int *) (task_cons->tss.esp + 8)) = memtotal;

```

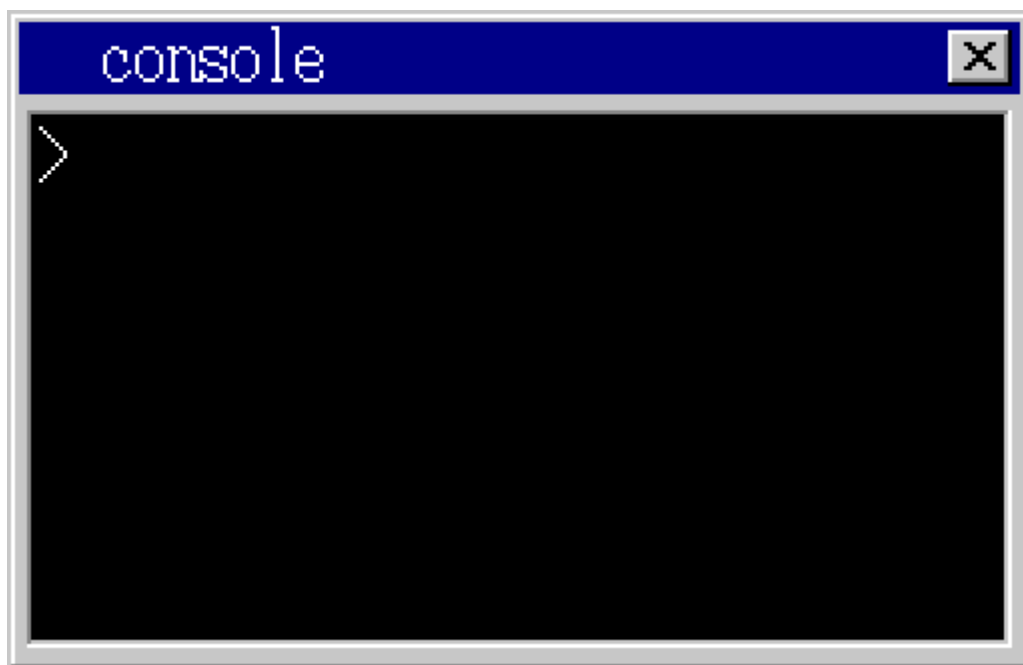


```
console
>python
Command Illegal.

>mem
total    32MB
free 27420KB

>
```

下面来实现cls命令，要点在于将cursor_x、cursor_y置为初始状态，把整个屏幕涂黑，然后打印提示符



```
console
>
```

最后来实现dir命令

dir命令是列举文件的命令

还记得文件名存储在磁盘的0x002600位置开始，也就是内存的0x00102600位置开始。

修改makefile，镜像中再添加几个文件

```
haribote.img : ip110.bin haribote.sys Makefile
$(EDIMG) imgin:../z_tools/fdimg0at.tek \
    wbinimg src:ip110.bin len:512 from:0 to:0 \
    copy from:haribote.sys to:@: \
    copy from:ip110.nas to:@: \
    copy from:make.bat to:@: \
    imgout:haribote.img
```

00002600	48 41 52 49 42 4f 54 45 53 59 53 20 00 00 00 00	H A R I B O T E S Y S
00002610	00 00 00 00 00 00 77 46 8a 4e 02 00 70 6c 00 00 w F 奇 . . p l . .
00002620	49 50 4c 31 30 20 20 20 4e 41 53 20 00 00 00 00	I P L 1 0 N A S
00002630	00 00 00 00 00 00 59 7a 42 35 39 00 95 0b 00 00 Y z B 5 9 . ? . .
00002640	4d 41 4b 45 20 20 20 20 42 41 54 20 00 00 00 00	M A K E B A T
00002650	00 00 00 00 00 00 f6 10 81 30 3f 00 2e 00 00 00 ? ? ?
00002660	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00002670	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

似乎有规律

结构是这样的

```
struct FILEINFO {
    unsigned char name[8], ext[3], type;
    char reserve[10];
    unsigned short time, date, clustno;
    unsigned int size;
};
```

开始的8个字节是文件名。文件名不足8个字节时，后面用空格补足。文件名超过8个字节的情况比较复杂，我们在这里先只考虑不超过8个字节的情况吧，一上来就挑战高难度的话，很容易产生挫败感呢。再仔细观察一下，我们发现所有的文件名都是大写的。如果文件名的第一个字节为0xe5，代表这个文件已经被删除了；文件名第一个字节为0x00，代表这一段不包含任何文件名信息。从磁盘映像的0x004200就开始存放文件haribote.sys了，因此文件信息最多可以存放224个。接下来3个字节是扩展名，和文件名一样，不足3个字节时用空格补足，如果文件没有扩展名，则这3个字节都用空格补足。扩展名和文件名一样，也全部使用了大写字母。后面1个字节存放文件的属性信息。我们这3个文件的属性都是0x20。一般的文件不是0x20就是0x00，至于其他的值，我们来看下面的说明。

0x02隐藏文件 0x04系统文件 0x08非文件信息（比如磁盘名称等） 0x10目录

接下来的10个字节为保留，也就是说，是为了将来可能会保存更多的文件信息而预留的，在我们的磁盘映像中都是0x00。话说，这个磁盘格式是由Windows的开发商微软公司定义的，因此，这段保留区域以后要如何使用，也是由微软公司来决定的。其他人要自行定义的话也可以，只不过将来可能会和Windows产生不兼容的问题。下面2个字节为WORD整数，存放文件的时间。因此即便文件的内容都一样，这里大家看到的数值也可能是因人而异的。再下面2个字节存放文件的日期。这些数值虽然怎么看都不像是时间和日期，但只要用微软公司的公式计算一下，就可以转换为时、分、秒等信息了。接下来的2个字节也是WORD整数，代表这个文件的内容从磁盘上的哪个扇区开始存放。变量名clustno本来是“簇号”（cluster number）的缩写，“簇”这个词是微软的专有名词，在这里我们先暂且理解为和“扇区”是一码事就好了。最后的4个字节为DWORD整数，存放文件的大小。

以上引用的是书上的内容。

不过我比较好奇，好多文件的扩展名不止3个字节，不知道这是怎么处理的

于是我稍微修改了下makefile


```

haribote.img : ipl10.bin haribote.sys Makefile
$(EDIMG) imgin:../z_tools/fdimg0at.tek \
    wbinimg src:ipl10.bin len:512 from:0 to:0 \
    copy from:haribote.sys to:@: \
    copy from:ipl10.nass to:@: \
    copy from:make.bat to:@: \
    imgout:haribote.img

```

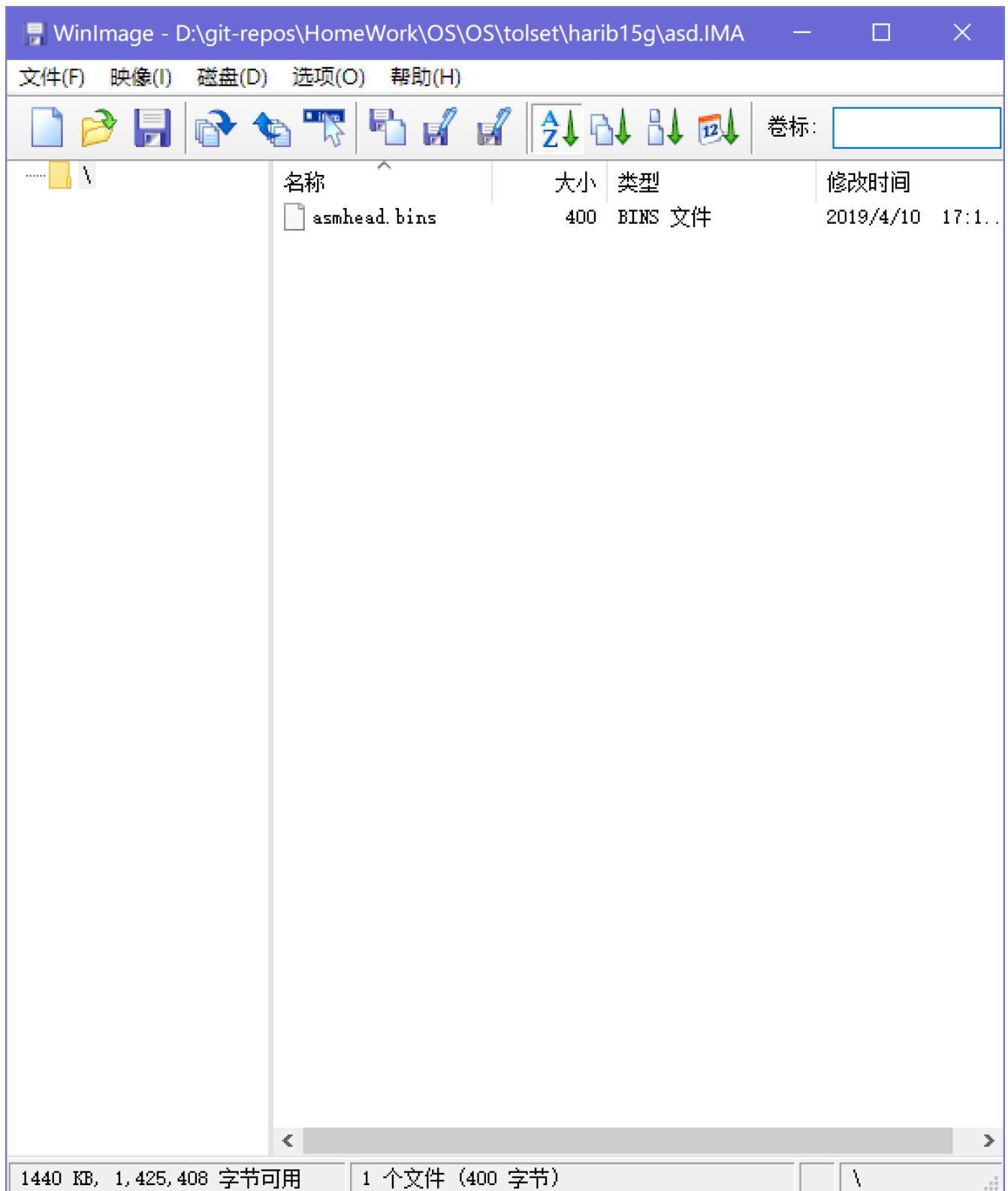
并且把ipl.nas重命名成了ipl.nass, 重新make并且用二进制编辑器查看

00002600	48 41 52 49 42 4f 54 45 53 59 53 20 00 00 00 00
00002610	00 00 00 00 00 00 aa 49 8a 4e 02 00 70 6c 00 00
0000262a	49 50 4c 31 30 20 20 20 4e 41 53 20 00 00 00 00
00002630	00 00 00 00 00 00 a2 49 8a 4e 39 00 95 0b 00 00
00002640	4d 41 4b 45 20 20 20 20 42 41 54 20 00 00 00 00
00002650	00 00 00 00 00 00 f6 10 81 30 3f 00 2e 00 00 00

H A R I B O T E S Y S
. 狗 箭	. . p l . .
I P L 1 0	N A S
. 箭	9 . ? . .
M A K E	B A T
. ?	? ? ?

emmm, 只进行了简单的截断? 我觉得是edimg的锅。

于是我下载了winimg, 并尝试进行修改



00002600	e5 53 4d 48 45 41 44 20 42 49 4e 00 18 67 6c 86	鎔 MHEAD BIN..gl 噢
00002610	8a 4e 00 00 63 03 a5 89 8a 4e 40 00 90 01 00 00	N..c. 裔 @.? ..
00002620	41 61 00 73 00 6d 00 68 00 65 00 0f 00 e6 61 00	Aa.s.m.h.e...鎔 .
00002630	64 00 2e 00 62 00 69 00 6e 00 00 00 73 00 00 00	d...b.i.n...s...
00002640	41 53 4d 48 45 41 7e 31 42 49 4e 00 18 67 6c 86	ASMHEA~1BIN..gl 噢
00002650	8a 4e 00 00 63 03 a5 89 8a 4e 40 00 90 01 00 00	N..c. 裔 @.? ..
00002660	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

似乎是ok的，关闭重新打开也可以正常读取。不过格式看不太出来，于是网搜

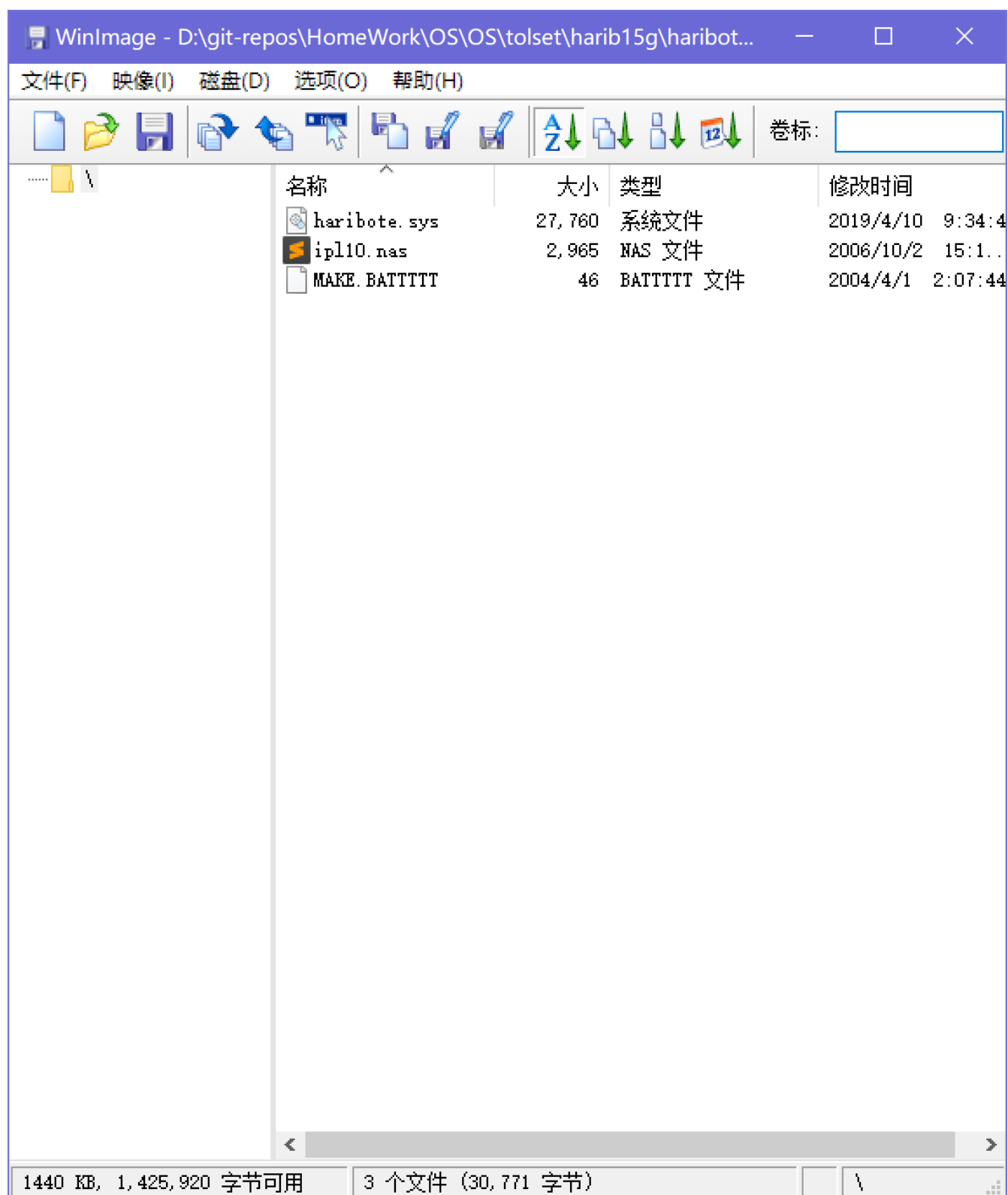
<https://blog.csdn.net/sikuon/article/details/76397831>

emmm, 看起来不ok? 不知道这怎么实现的, 暂时留作问题

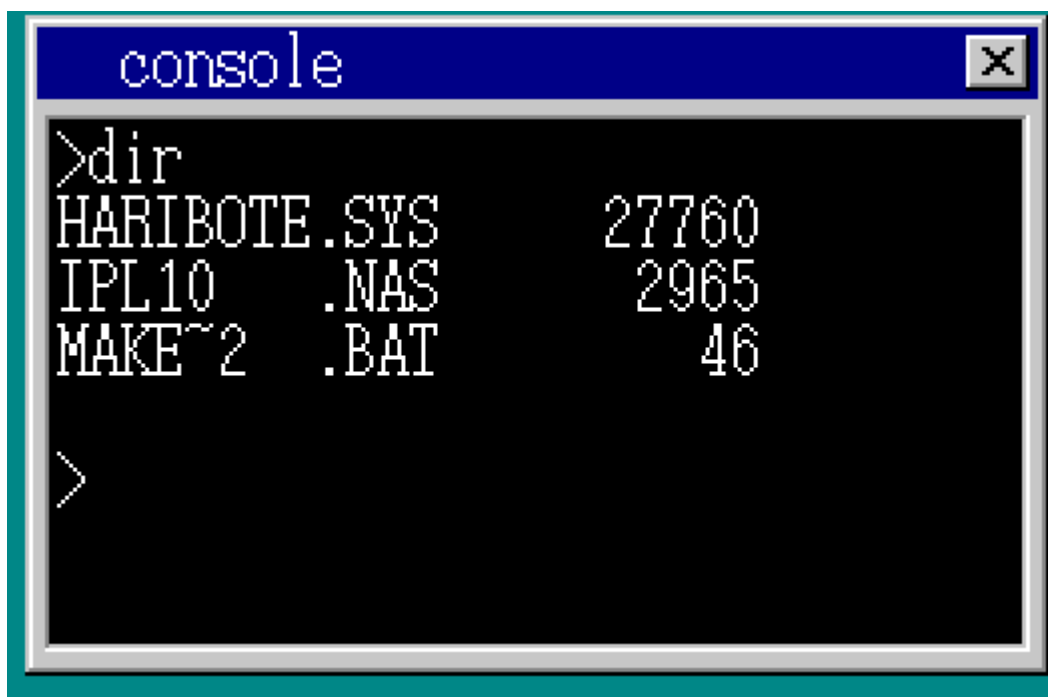
然后我们就可以来编制dir了!

```
for (x = 0; x < 224; x++) {
    if (finfo[x].name[0] == 0x00) {
        break;
    }
    if (finfo[x].name[0] != 0xe5) {
        if ((finfo[x].type & 0x18) == 0) {
            sprintf(s, "filename.ext %7d", finfo[x].size);
            for (y = 0; y < 8; y++) {
                s[y] = finfo[x].name[y];
            }
            s[ 9] = finfo[x].ext[0];
            s[10] = finfo[x].ext[1];
            s[11] = finfo[x].ext[2];
            putfonts8_asc_sht(sheet, 8, cursor_y, COL8_FFFFFFFF,
                               COL8_000000, s, 30);
            cursor_y = cons_newline(cursor_y, sheet);
        }
    }
}
```

我make之后又皮了一下, 似乎有些发现



这样改完之后



后面有个~2，又修改为make.batt，变成~1，查看二进制。

他似乎是用~作为转义符，之前的两行存储了更多的关于文件名的更多信息。

00002660	41	4d	00	41	00	4b	00	45	00	2e	00	0f	00	5f	42	00
00002670	41	00	54	00	54	00	54	00	54	00	00	00	54	00	00	00
00002680	41	41	41	45	00	00	00	40	41	54	00	00	00	00	00	00

A	M	.	A	.	K	.	E	_	B	.
A	.	T	.	T	.	T	.	T	.	T	.	T	.	T	.	T	.	.
M	A	K	E

今天就先到这吧

Day 19

首先我们要实现type命令，命令功能类似于cat，用于向控制台打印文件的内容。

先想想怎么获取文件的内容。还记得0x2600位置存放的文件信息的格式吗？有一个clustno可以告诉我们他从哪个扇区开始存放，而一个扇区是512字节，第一个文件处于2号扇区，位置是0x004200。据此，我们可以推算出如下公式

$$addr_in_image = clustno \times 512 + 0x003e00$$

再结合size字段，我们就可以把对应地址的文件给读出来了。

type指令的格式

```
type <文件名>.<扩展名>
```

我们要先在0x2600找到我们要寻找的文件，目前我们先 $O(n)$ 的找吧。

每次比较finfo[x].name与文件名的11个字节是否全部相等，相等则找到，全不相等则找不到这样的文件

```
for (y = 0; y < 11; y++) {
    s[y] = ' ';
}
y = 0;
for (x = 5; y < 11 && cmdline[x] != 0; x++) {
    if (cmdline[x] == '.' && y <= 8) {
        y = 8;
    } else {
        s[y] = cmdline[x];
        if ('a' <= s[y] && s[y] <= 'z') {
            s[y] -= 0x20;    // 转换成大写
        }
        y++;
    }
}
for (x = 0; x < 224; x++) {
    if (finfo[x].name[0] == 0x00) {
        break;
    }
    if ((finfo[x].type & 0x18) == 0) {
        for (y = 0; y < 11; y++) {
            if (finfo[x].name[y] != s[y]) {
                continue;
            }
        }
        break;
    }
}
if (x < 224 && finfo[x].name[0] != 0x00) {
```

```

// 找到了!
y = finfo[x].size;
p = (char *) (finfo[x].clustno * 512 + 0x003e00 + ADR_DISKIMG);
cursor_x = 8;
for (x = 0; x < y; x++) {
    // 打印文件内容
    s[0] = p[x];
    s[1] = 0;
    putfonts8_asc_sht(sheet, cursor_x, cursor_y, COL8_FFFFFFFF, COL8_000000, s, 1);
    cursor_x += 8;
    if (cursor_x == 8 + 240) { // 打满一行自动换行
        cursor_x = 8;
        cursor_y = cons_newline(cursor_y, sheet);
    }
}
} else {
    // 没找到
    putfonts8_asc_sht(sheet, 8, cursor_y, COL8_FFFFFFFF, COL8_000000, "File not found.", 15);
    cursor_y = cons_newline(cursor_y, sheet);
}
}

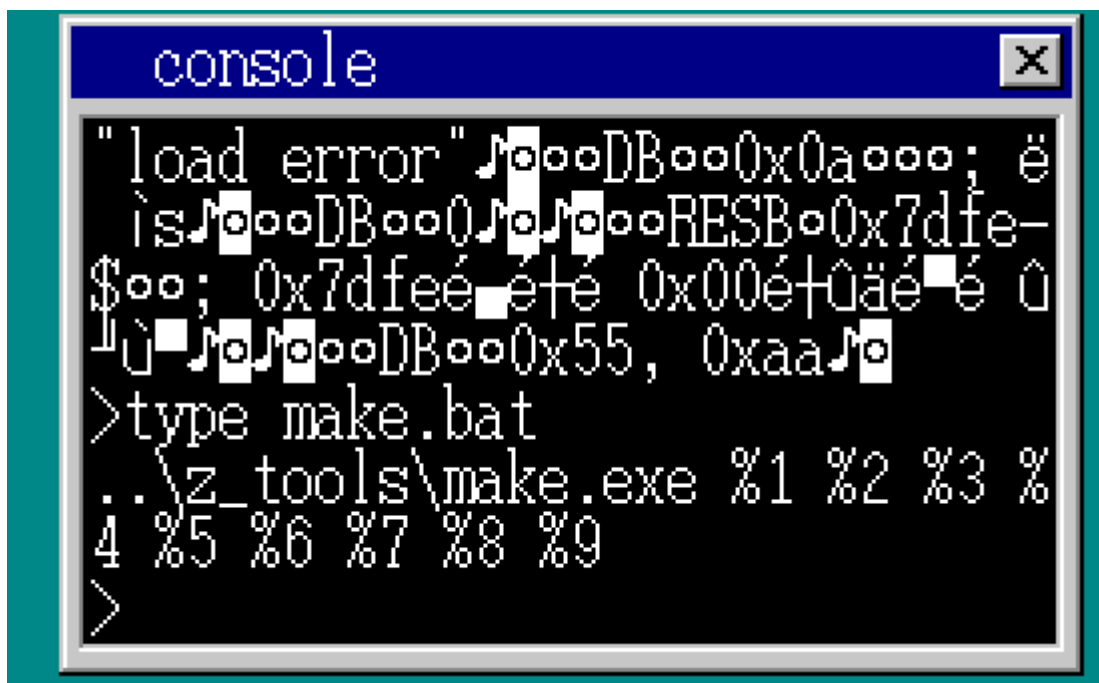
```

命令判定部分

```

else if (memcmp(cmdline, "type", 4) == 0) { // 作者这里用的是丑丑的多重条件
    // .....
}

```



上面的乱码是 type ip110.nas 的结果

我们接下来要处理特殊字符

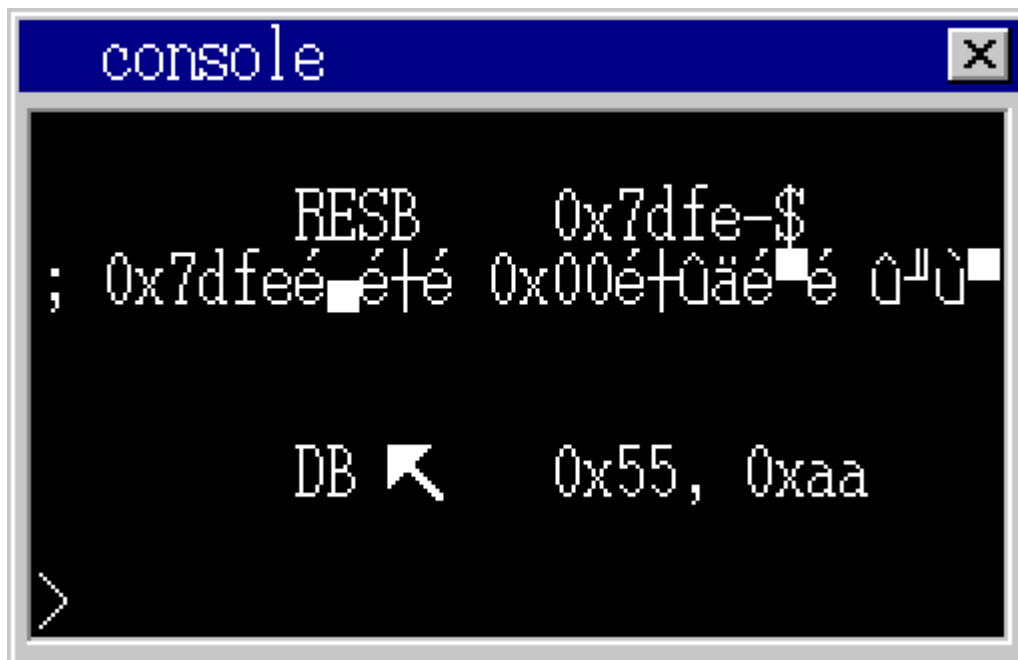
ASCII	名字	处理方法
0x09	制表符	显示空格直到x被4整除
0x0a	换行符	换行
0x0d	回车符	不管他

```
if (s[0] == 0x09) { // 制表符
    for (;;) {
        putfonts8_asc_sht(sheet, cursor_x, cursor_y, COL8_FFFFFFFF,
            COL8_000000, " ", 1);
        cursor_x += 8;
        if (cursor_x == 8 + 240) {
            cursor_x = 8;
            cursor_y = cons_newline(cursor_y, sheet);
        }
        if (((cursor_x - 8) & 0x1f) == 0) { // (cursor_x-8) % 32
            break;
        }
    }
} else if (s[0] == 0x0a) { // 换行符
    cursor_x = 8;
    cursor_y = cons_newline(cursor_y, sheet);
} else if (s[0] == 0x0d) {
    // do nothing
} else { // 普通字符
    putfonts8_asc_sht(sheet, cursor_x, cursor_y, COL8_FFFFFFFF,
        COL8_000000, s, 1);
    cursor_x += 8;
    if (cursor_x == 8 + 240) {
        cursor_x = 8;
        cursor_y = cons_newline(cursor_y, sheet);
    }
}
```

以上是特殊字符判断部分

为什么要将cursor_x减8呢？因为命令行窗口的边框有8个像素，所以要把那部分给去掉。然后，1个字符的宽度是8个像素，每个制表位相隔4个字符。4x宽度（8）=32。所以要对32取模。

再次测试



除了日文还乱码之外，其他已经正常了许多了。

接下来引入对fat的支持。

众所周知，很多文件都不止512字节，一个扇区是不足以保存他们的，那么我们该如何处理呢？我们需要读取fat(file allocation table)

fat存储在0x00200~0x013ff，我们先给他读进内存。在正式使用它之前，需要先进行解压缩

以三个字节为一组，进行4个bit为单位的换序

F0 FF FF → FF0 FFF
ab cd ef dab efc

```
void file_readfat(int *fat, unsigned char *img)
{
    int i, j = 0;
    for (i = 0; i < 2880; i += 2) {
        fat[i + 0] = (img[j + 0]      | img[j + 1] << 8) & 0xffff;
        fat[i + 1] = (img[j + 1] >> 4 | img[j + 2] << 4) & 0xffff;
        j += 3;
    }
    return;
}
```

该函数将img地址未经解压的fat解压缩并存入fat指针指向的位置

然后我们就可以查这个fat表了，他其实相当于一个链表，每当你访问一个扇区，你就访问对应的fat，从中你可以得知你应该访问的下一个扇区，以此类推，直到fat中指示的下一个扇区为0xff8~0xff的值，则说明文件结束。

接下来我们写一个函数用于读入文件到内存中，只要当读完一个扇区且文件还没读完，按照fat表进行跳转就ok了

```
void file_loadfile(int clustno, int size, char *buf, int *fat, char *img)
{
    int i;
    for (;;) {
        if (size <= 512) {
            for (i = 0; i < size; i++) {
                buf[i] = img[clustno * 512 + i];
            }
            break;
        }
        for (i = 0; i < 512; i++) {
            buf[i] = img[clustno * 512 + i];
        }
        size -= 512;
        buf += 512;
        clustno = fat[clustno];
    }
    return;
}
```

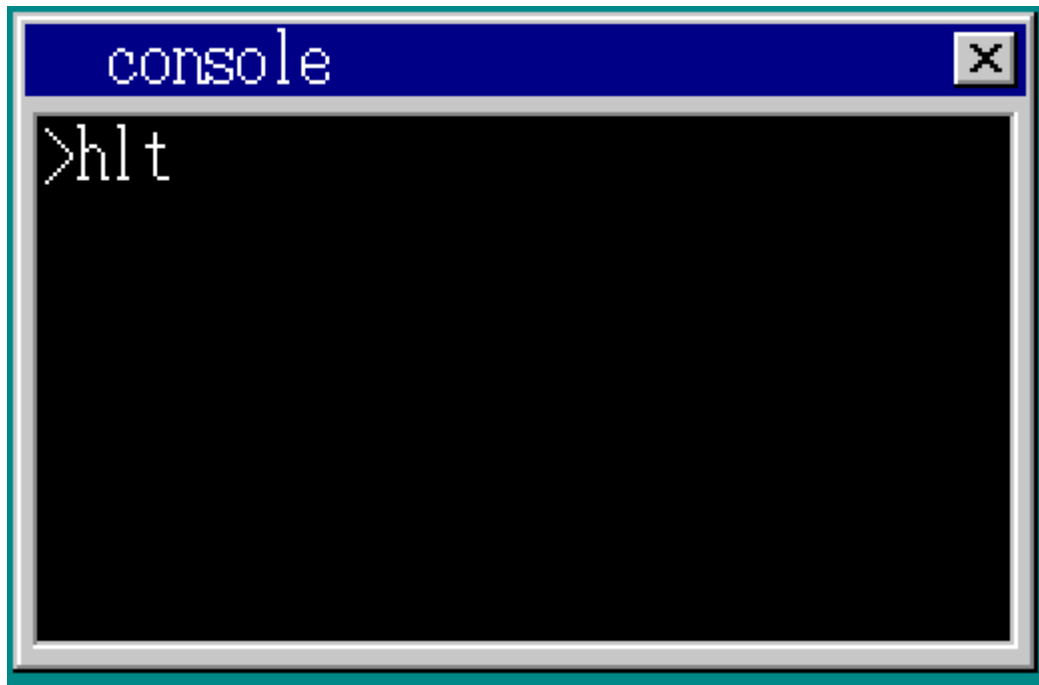
能够读取文件了，接下来我们想办法引入对应用程序的支持。

主要思路是先编写程序，然后用nask编译成文件，然后添加到镜像当中去。将文件load到另一个连续的内存区域，然后farjump到那个内存区域就ok了

下段程序运行hlt.hrb

```
strncpy(s, "HLT      HRB", 11);
for (x = 0; x < 224 && finfo[x].name[0]; x++) {
    if ((finfo[x].type & 0x18) == 0) {
        for (y = 0; y < 11; y++) {
            if (finfo[x].name[y] != s[y]) {
                continue;
            }
        }
        break;
    }
}
if (x < 224 && finfo[x].name[0]) {
    p = (char *) memman_alloc_4k(memman, finfo[x].size);
    file_loadfile(finfo[x].clustno, finfo[x].size, p, fat, (char *)
        (ADR_DISKIMG + 0x003e00));
    set_segmdesc(gdt + 1003, finfo[x].size - 1, (int) p, AR_CODE32_ER);
    farjmp(0, 1003 * 8);
    memman_free_4k(memman, (int) p, finfo[x].size);
}
```

```
} else {  
    putfonts8_asc_sht(sheet, 8, cursor_y, COL8_FFFFFFFF, COL8_000000, "File not found.", 15);  
    cursor_y = cons_newline(cursor_y, sheet);  
}  
cursor_y = cons_newline(cursor_y, sheet);
```



成功，今天到此为止

Day 20

我们想在我们应用程序中实现一个类似于printf之类的函数，可以输出东西到控制台。但是目前为止我们做不到，因为控制台的输出是系统控制的，应用程序无法直接在控制台上输出。所以我们要设计API，应用程序通过调用API，来把他想打印的东西告诉系统，然后系统在控制台上为他进行输出。

先来实现单字符输出。

主要思路是在系统的代码中有一个cons_putchar()函数，然后我们可以使用nasm的call关键字进行跳转。我们打算把参数存在寄存器当中，但是这样又面临着另一个问题。我们操作系统中的cons_putchar()函数是用C语言来写的，C语言是无法直接获得寄存器的值的。所以我们还要给这个cons_putchar()包个壳，定义个新的nasm函数asm_cons_putchar()。该函数从EAX寄存器中获取需要打印的字符，然后将他们压入栈中（作为参数），再进行call，然后将之前压入栈中的内容弹出。这样就完成了整个API调用流程。

```
_asm_cons_putchar:
    PUSH 1                ; 参数是从EPS+4开始的，所以往里面push一个int占位
    AND EAX, 0xff         ; 把参数mask以下，避免应用程序错误的调用API造成错误
    PUSH EAX              ; 将EAX压入栈中作为C语言函数参数
    PUSH DWORD [0x0fec]   ; 要打印的控制台地址
    CALL _cons_putchar    ; c语言函数地址
    ADD ESP, 12           ; 将栈中的数据丢弃
    RET
```

控制台地址的0x0fec是怎么回事呢？应用程序没法直接获取控制台的地址，所以我们约定把这个地址存储在0x0fec位置上，到时候只要去找就ok了。

```
*((int*) 0x0fec) = (int) &cons;
```

然后我们如何调用这段程序呢？注意操作系统与应用程序的编译并不相关，在编译应用程序的时候并不知道asm_cons_putchar()的存在。

这里还需要我们处理一下。我们可以先编译操作系统，然后从中得知asm_cons_putchar()函数的地址，然后把他硬写在应用程序的代码当中。

修改makefile，使他能够输出每个函数的地址。

```
bootpack.bim : $(OBJ2_BOOTPACK) Makefile
$(OBJ2_BOOTPACK) @$(RULEFILE) out:bootpack.bim stack:3136k map:bootpack.map \
$(OBJ2_BOOTPACK)
```

查看bootpack.map

```

31 0x000000BDE : _farjmp
32 0x000000BE3 : _asm_cons_putchar
33 0x000000BFA : _init_palette
34 0x000000BFA : (.text)

```

从中我们知道了asm_cons_putchar()的地址是0x0be3，所以我们可以按与如下方式类似的方法调用我们的API

```

MOV AL, 'A'
CALL 0xbe3

```

对了！还有一个问题，仅仅这样是不行的，仅仅这样call，是call不到我们的api的。操作系统所在的段时2，所以我们还需要补充一个段号。改成

```
CALL 2*8: 0xbe3
```

使用了farcall，对应的也应该使用far ret。所以我们还要修改asm_cons_putchar()

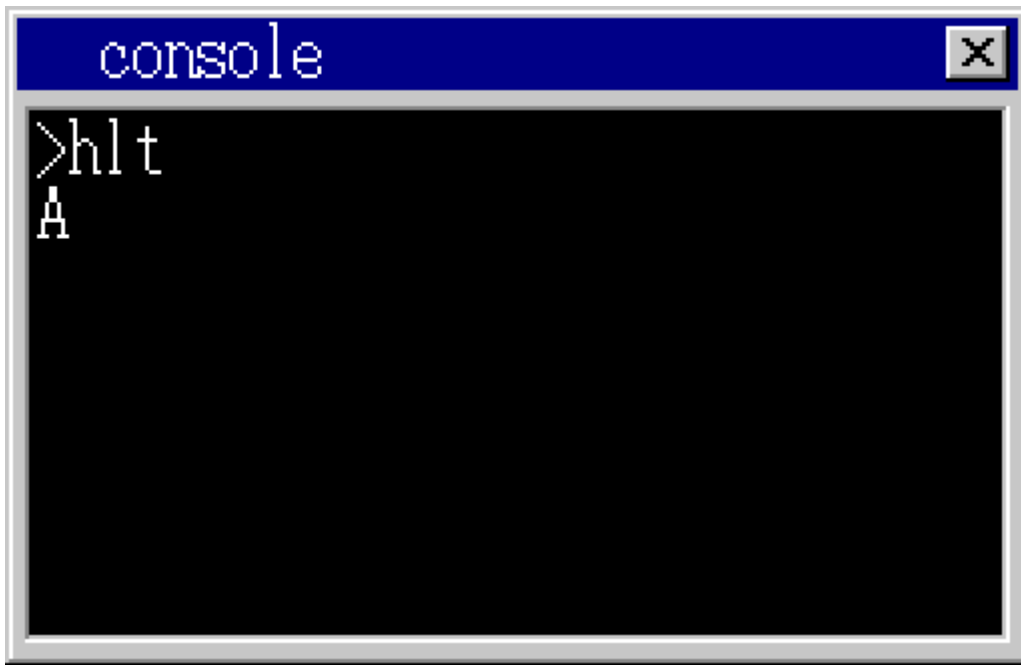
将最后的RET修改成RETF就ok了。

```

_asm_cons_putchar:
    PUSH 1                ; 参数是从EPS+4开始的，所以往里面push一个int占位
    AND EAX, 0xff          ; 把参数mask以下，避免应用程序错误的调用API造成错误
    PUSH EAX              ; 将EAX压入栈中作为C语言函数参数
    PUSH DWORD [0x0fec]    ; 要打印的控制台地址
    CALL _cons_putchar     ; c语言函数地址
    ADD ESP, 12            ; 将栈中的数据丢弃
    RETF

```

跑一下吧



成功了!

不过现在应用程序运行起来仍然有问题, 我们无法从应用程序返回操作系统了。这可不妙。为了系统能够正常的返回, 我们应当使用call和ret。由于应用程序和操作系统不在相同的段中, 所以我们要用far call和far ret。

为了在c语言中实现far call, 需要编制naskfunc。

```
_farcall: ; void farcall(int eip, int cs);  
    CALL FAR [ESP+4] ; eip, cs  
    RET
```

然后调用程序的代码修改成为

```
farcall(0, 1003 * 8);
```

应用程序代码也需要修改, 把hlt换成retf就ok了。同时需要注意, 由于我们修改了操作系统的代码, API的地址会发生变化, 所以我们需要重新查找asm_cons_putchar()的地址并写入

```
/* --- hlt.nas --- */  
    MOV AL, 'A'  
    CALL 2*8:0xbe8  
    RETF
```

不过这样好烦啊, 每次迭代源码都需要重新查表修改应用程序源码, 当API变多、API调用变多的时候, 这将成为一场灾难。

记得我们之前通过设置IDT来用C语言来处理中断吗? IDT上还有好多空闲的中断号没有使用, 我们可以用其中的一个中断号来代替我们的API地址。这样, 调用API只需要触发随营的软中断就好了。

要做的工作有:

- 在IDT中注册asm_cons_putchar()

- 修改asm_cons_putchar()推出语句为IRETD
- 在asm_cons_putchar()开始处进行sti (这是因为CPU会自动关闭中断, 导致无法响应键盘、鼠标等操作)

make run下, 运行顺利

接下来我们要让系统支持输入文件名, 运行对应的程序。

具体思路如下 如果我们输入的命令是

- mem
- cls
- dir
- type

其中之一的话, 我们执行对应的控制台功能, 否则我们查找是否存在<命令>或者<命令>.hrb文件, 如果存在的话, 我们就执行该文件。

由此, 需要编制一个cmd_app函数, 他找到并执行一个程序

```
int cmd_app(struct CONSOLE *cons, int *fat, char *cmdline)
{
    struct MEMMAN *memman = (struct MEMMAN *) MEMMAN_ADDR;
    struct FILEINFO *finfo;
    struct SEGMENT_DESCRIPTOR *gdt = (struct SEGMENT_DESCRIPTOR *) ADR_GDT;
    char name[18], *p;
    int i;
    for (i = 0; i < 13; i++) {
        if (cmdline[i] <= ' ') {
            break;
        }
        name[i] = cmdline[i];
    }
    name[i] = 0;
    finfo = file_search(name, (struct FILEINFO *) (ADR_DISKIMG + 0x002600), 224);    // 尝试
    查找文件
    if (finfo == 0 && name[i - 1] != '.') { // 找不到, 加后缀名再试一遍
        name[i] = '.';
        name[i + 1] = 'H';
        name[i + 2] = 'R';
        name[i + 3] = 'B';
        name[i + 4] = 0;
        finfo = file_search(name, (struct FILEINFO *) (ADR_DISKIMG + 0x002600), 224);

        if (finfo != 0) { // 找到则读取文件到内存并执行
            p = (char *) memman_alloc_4k(memman, finfo->size);
            file_loadfile(finfo->clustno, finfo->size, p, fat, (char *) (ADR_DISKIMG +
0x003e00));
            set_segmdesc(gdt + 1003, finfo->size - 1, (int) p, AR_CODE32_ER);
            farcall(0, 1003 * 8);
            memman_free_4k(memman, (int) p, finfo->size);
            cons_newline(cons);
            return 1;    // 找到返回1
        }
    }
}
```

```
    }  
    return 0;        // 找不到返回0  
}
```

之前我们改用中断来做API其实漏了一个东西，我们应该保存CPU现场的，否则会出问题

再中断服务程序开始的地方加PUSHAD，结束的地方加POPAD。这样就OK了

我们来实现更多的API吧。由于API可能会有很多，一个API对应一个中断号不太现实。

根据课堂上学到的知识我们知道，现代操作系统的API基本上都是通过中断号加功能号来实现的，我们也来使用这样的设计。

我们编写两个新的API，一种是显示一串字符，遇到字符编码0则结束；另一种是先指定好要显示的字符串的长度再显示。

```
void cons_putstr0(struct CONSOLE *cons, char *s)  
{  
    for (; *s != 0; s++) {  
        cons_putchar(cons, *s, 1);  
    }  
    return;  
}  
void cons_putstr1(struct CONSOLE *cons, char *s, int l)  
{  
    int i;  
    for (i = 0; i < l; i++) {  
        cons_putchar(cons, s[i], 1);  
    }  
    return;  
}
```

然后我们分配功能号

功能号	功能
1	显示单个字符 (AL = 字符编码)
2	显示字符串 0 (EBX = 字符串地址)
3	显示字符串 1 (EBX = 字符串地址, ECX = 字符串长度)

为API引入统一的入口。


```

_asm_hrb_api:
    STI
    PUSHAD ; 用于保存寄存器值的PUSH
    PUSHAD ; 用于向hrb_api传值的PUSH
    CALL _hrb_api
    ADD ESP,32
    POPAD
    IRETD

```

```

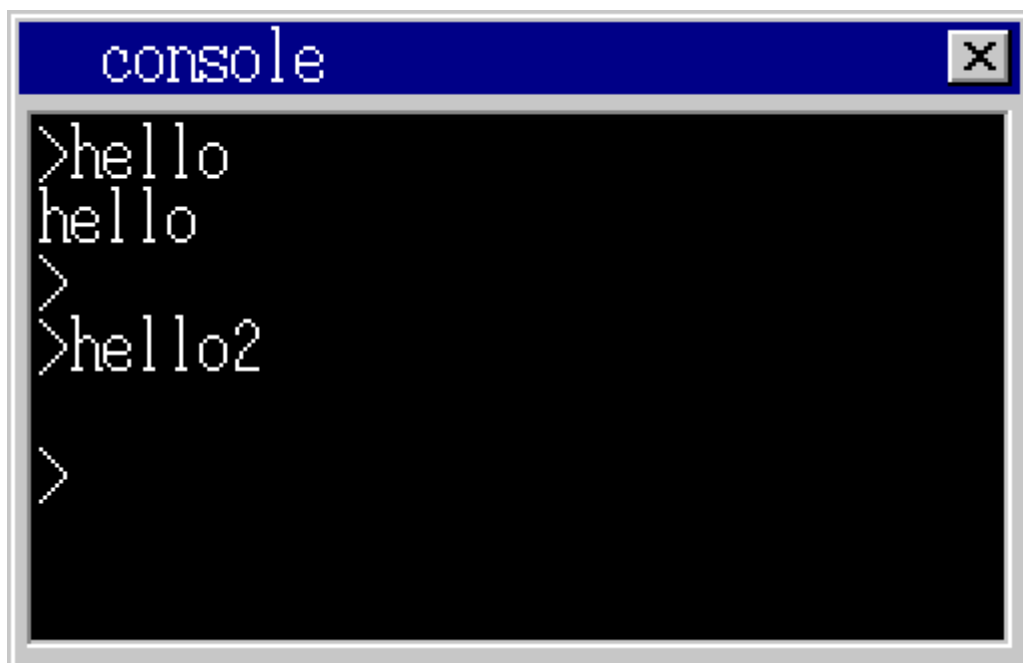
void hrb_api(int edi, int esi, int ebp, int esp, int ebx, int edx, int ecx, int eax)
{
    struct CONSOLE *cons = (struct CONSOLE *) *((int *) 0x0fec);
    if (edx == 1) {
        cons_putchar(cons, eax & 0xff, 1);
    } else if (edx == 2) {
        cons_putstr0(cons, (char *) ebx);
    } else if (edx == 3) {
        cons_putstr1(cons, (char *) ebx, ecx);
    }
    return;
}

```

然后修改IDT注册部分。

make run一下

啥也没有?????



作者要把这个放到day 21?

不成，今天必须给弄了。

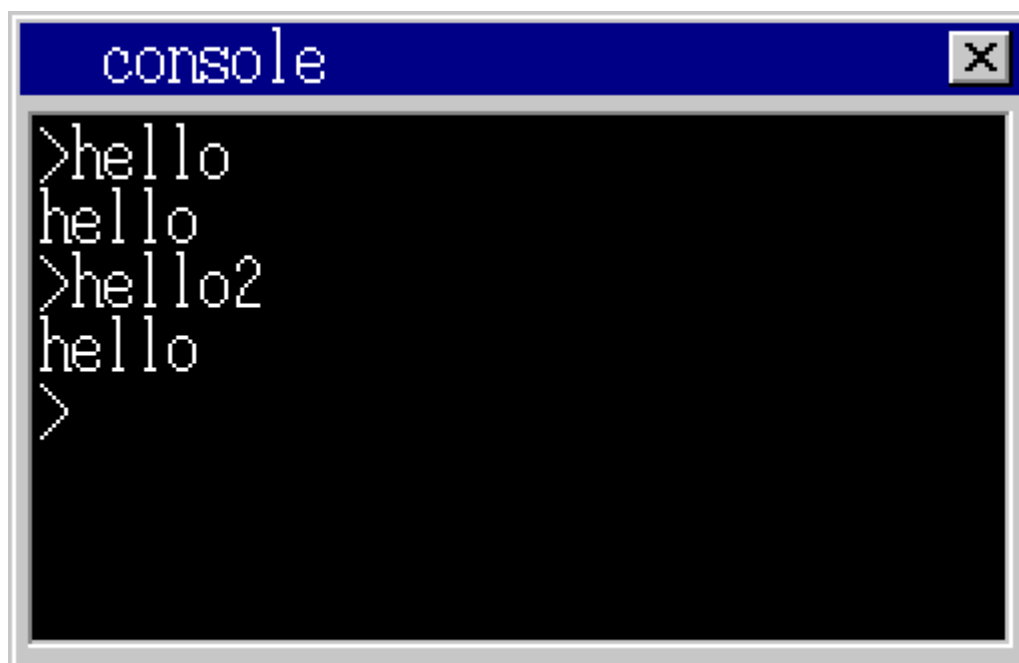
通过阅读Day21的内容，了解到原来是代码段没有正确的设置而导致了这个错误，我们只要正确的设置代码段就可以了。

我们想办法把我们之前为应用程序准备的内存地址给传过去

```
/* --- partial content of cmd_app --- */
if (finfo != 0) {
    p = (char *) memman_alloc_4k(memman, finfo->size);
    *((int *) 0xfe8) = (int) p; // 把分配的内存地址存到0xfe8位置
    file_loadfile(finfo->clustno, finfo->size, p, fat, (char *) (ADR_DISKIMG +
0x003e00));
    set_segmdesc(gdt + 1003, finfo->size - 1, (int) p, AR_CODE32_ER);
    farcall(0, 1003 * 8);
    memman_free_4k(memman, (int) p, finfo->size);
    cons_newline(cons);
    return 1;
}
```

```
void hrb_api(int edi, int esi, int ebp, int esp, int ebx, int edx, int ecx, int eax)
{
    struct CONSOLE *cons = (struct CONSOLE *) *((int *) 0xfec);
    int cs_base = *((int *) 0xfe8); // 取出段地址
    if (edx == 1) {
        cons_putchar(cons, eax & 0xff, 1);
    } else if (edx == 2) {
        cons_putstr0(cons, (char *) ebx + cs_base); // 加上段地址
    } else if (edx == 3) {
        cons_putstr1(cons, (char *) ebx + cs_base, ecx); // 加上段地址
    }
    return;
}
```

make run



OK, 成功