褚逸豪 2017211123 序号17

Day 21

我们之前第20天的时候已经提前解决了今天开头的这个小问题，所以我们先跳过了。

之前我们所谓的应用程序都是用汇编语言写的，不过用汇编语言写程序真是太累了，我们得想办法。

我们的思路是这样的：用汇编语言写API，c语言写程序，然后都编译成符号文件，然后进行链接。最后用bim2hrb转换一下。

我们写了一个输出一个字符的程序，装入映像中。make run。然而系统却没了反应。

咋回事呢？根据作者提供的方法，我们修改了程序的前6个字节为`E8 16 00 00 00 CB`，就能顺利运行了。

这六个字节，对应的nask命令如下

```
call 0x1b
retf
```

相当于调用0x1b位置的函数，然后进行far return

0x1b位置是什么呢？他正好是我们的写的main函数所处的位置。

这样我们就搞明白整个的工作原理了。

C语言（或者是bim2hrb，不确定）编译（链接）出来的程序的main函数并不存在于逻辑地址的0x00位置，而是再0x1b的位置上，而我们的操作系统则是从文件的开始位置执行的。于是我们在开始位置替换两条命令，使得应用程序能够调用main，并且运行完之后返回系统。

不过每次替换程序的开头的6个字节实在是太麻烦，所以我们把他交给操作系统来做吧！

在进行farcall之前，做如下操作

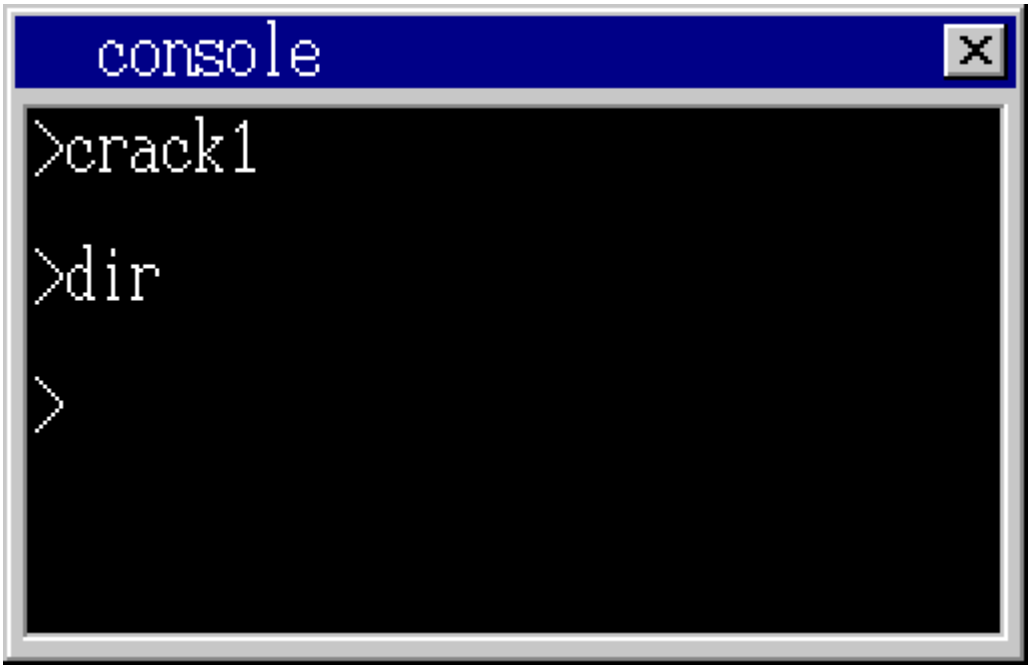
```
if (finfo->size >= 8 && strncmp(p + 4, "Hari", 4) == 0) {
    p[0] = 0xe8;
    p[1] = 0x16;
    p[2] = 0x00;
    p[3] = 0x00;
    p[4] = 0x00;
    p[5] = 0xcb;
}
```

为什么size要至少为8呢，因为我们要比较4到7byte，看这个是不是c语言的程序（bim2hrb生成的文件这4个字节一定是"Hari"）（要是汇编语言写的也做替换的话会出问题）。

接下来我们搞搞系统保护。由于应用程序的开发不受操作系统的控制，我们不能确保每个开发者不具有恶意并且水平足够高，所以我们要进行操作系统保护以防止操作系统遭到意外破坏无法正常运行。

我们进行一个小小的破坏系统的测试，讲0x00102600位置修改为0

```
void HariMain(void)
{
    *((char *) 0x00102600) = 0;
    return;
}
```



可见dir命令已经坏掉了。

如何保护系统呢？我们应当把应用程序对内存的操作和访问限制在一定的范围内。我们创建应用程序专用的数据段

操作系统用代码段	2 * 8
操作系统用数据段	1 * 8
应用程序用代码段	1003 * 8
应用程序用数据段	1004 * 8
TSS段	3 * 8 ~ 1002 * 8

```
if (finfo != 0) {
    p = (char *) memman_alloc_4k(memman, finfo->size);
    q = (char *) memman_alloc_4k(memman, 64 * 1024); // 为数据段分配空间
    *((int *) 0xfe8) = (int) p;
    file_loadfile(finfo->clustno, finfo->size, p, fat, (char *) (ADR_DISKIMG + 0x003e00));
    set_segmdesc(gdt + 1003, finfo->size - 1, (int) p, AR_CODE32_ER);
    set_segmdesc(gdt + 1004, 64 * 1024 - 1,   (int) q, AR_DATA32_RW); // 设定gdt表，把刚刚分配的空间注册了
    if (finfo->size >= 8 && strncmp(p + 4, "Hari", 4) == 0) {
        p[0] = 0xe8;
        p[1] = 0x16;
        p[2] = 0x00;
```

```
        p[3] = 0x00;
        p[4] = 0x00;
        p[5] = 0xcb;
    }
    start_app(0, 1003 * 8, 64 * 1024, 1004 * 8); // 指定cs,  ds
    memman_free_4k(memman, (int) p, finfo->size);
    memman_free_4k(memman, (int) q, 64 * 1024); // 释放空间
    cons_newline(cons);
    return 1;
}
```

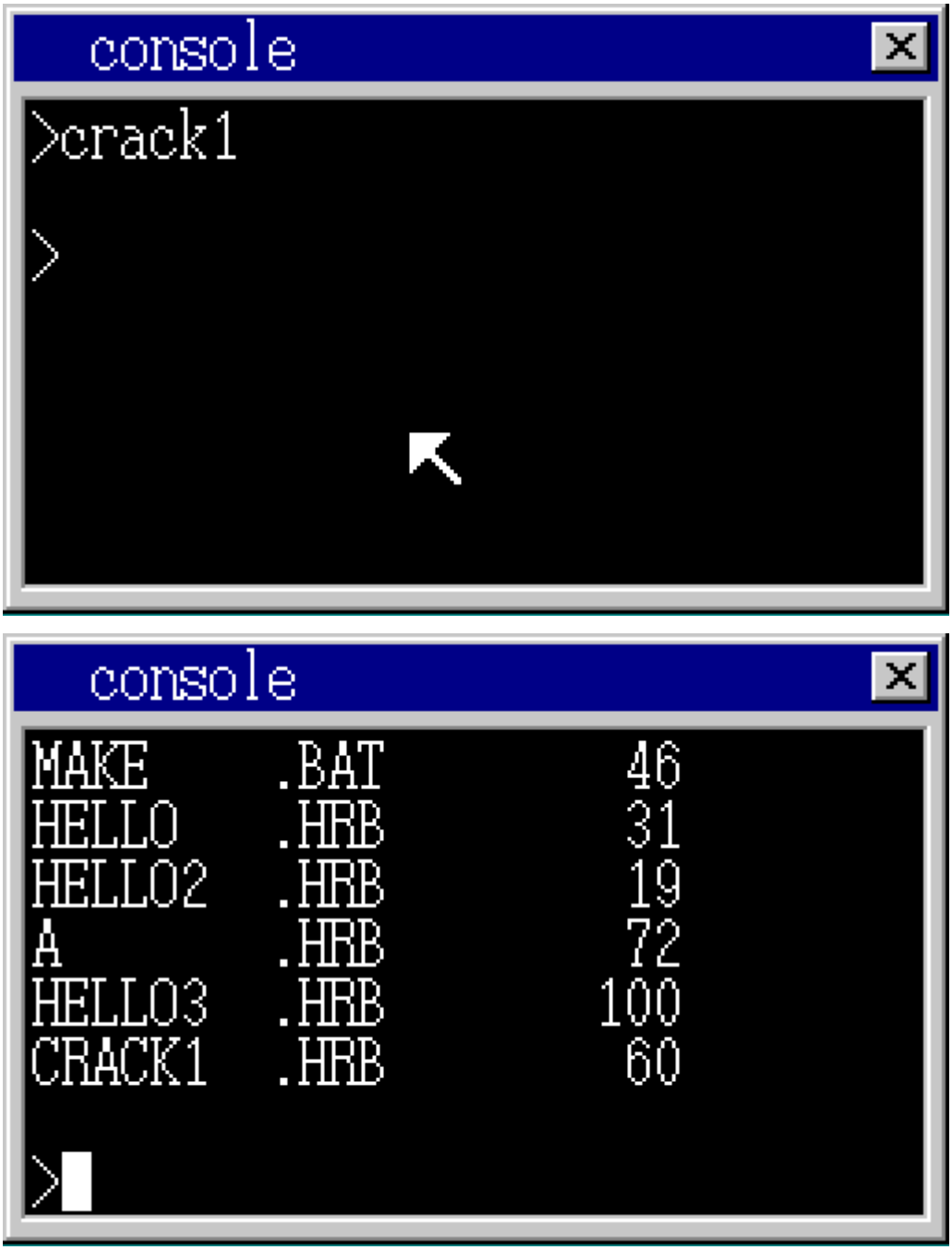
```
_start_app:     ; void start_app(int eip, int cs, int esp, int ds);
        PUSHAD      ; 保存现场
        MOV     EAX,[ESP+36]    ; 保存参数eip
        MOV     ECX,[ESP+40]    ; 保存参数cs
        MOV     EDX,[ESP+44]    ; 保存参数esp
        MOV     EBX,[ESP+48]    ; 保存参数ds
        MOV     [0xfe4],ESP     ; 保存操作系统的esp到内存中
        CLI         ; 切换时关中断
        MOV     ES,BX
        MOV     SS,BX
        MOV     DS,BX
        MOV     FS,BX
        MOV     GS,BX
        MOV     ESP,EDX ; 导入应用程序esp
        STI         ; 切换完成恢复中断
        PUSH    ECX             ; cs
        PUSH    EAX             ; eip
        CALL    FAR [ESP]       ; 调用应用程序

        MOV     EAX,1*8         ; 恢复操作系统的ds
        CLI         ; 切换，关中断
        MOV     ES,AX
        MOV     SS,AX
        MOV     DS,AX
        MOV     FS,AX
        MOV     GS,AX
        MOV     ESP,[0xfe4]
        STI         ; 恢复中断
        POPAD   ; 恢复现场
        RET
```

别忘了hrb_api，它应当工作在内核态下，我们应该将段切换回系统，否则无法正常工作

对了，还有中断处理的也需要改！因为我们的程序在运行过程中中断可不会停止产生。

总之，任何可能返回内核态的位置都需要进行修改。真是太麻烦了！

我们就不在这里写代码了。方法和start_app大同小异。



操作系统没有被破坏，好极了！

我们日常使用的操作系统在程序产生错误的时候都可以自动结束并报错，我们也引入这样的功能。

> 当应用程序试图破坏操作系统，或者试图违背操作系统的设置时，就会自动产生0x0d中断

也就是说，我们只要注册0x0d中断，就可以对应用程序的错误进行响应。

由于是中断，我们还是要进行切换的设置（微笑）

```
_asm_inthandler0d:
        STI
        PUSH ES
        PUSH DS
        PUSHAD
        MOV AX,SS
```

```
        CMP AX,1*8
        JNE .from_app ; 判断是操作系统还是应用程序产生的0x0d中断
; 当操作系统活动时产生中断的情况和之前差不多
        MOV EAX,ESP
        PUSH SS ; 保存中断时的SS
        PUSH EAX ; 保存中断时的ESP
        MOV AX,SS
        MOV DS,AX
        MOV ES,AX
        CALL _inthandler0d
        ADD ESP,8
        POPAD
        POP DS
        POP ES
        ADD ESP,4 ; 在INT 0x0d中需要这句
        IRETD
        .from_app:
; 当应用程序活动时产生中断
        CLI
        MOV EAX,1*8
        MOV DS,AX ; 先仅将DS设定为操作系统用
        MOV ECX,[0xfe4] ; 操作系统的ESP
        ADD ECX,-8
        MOV [ECX+4],SS ; 保存产生中断时的SS
        MOV [ECX ],ESP ; 保存产生中断时的ESP
        MOV SS,AX
        MOV ES,AX
        MOV ESP,ECX
        STI
        CALL _inthandler0d
        CLI
        CMP EAX,0
        JNE .kill
        POP ECX
        POP EAX
        MOV SS,AX ; 将SS恢复为应用程序用
        MOV ESP,ECX ; 将ESP恢复为应用程序用
        POPAD
        POP DS
        POP ES
        ADD ESP,4 ; INT 0x0d需要这句
        IRETD
.kill:
; 将应用程序强制结束
        MOV EAX,1*8 ; 操作系统用的DS/SS
        MOV ES,AX
        MOV SS,AX
        MOV DS,AX
        MOV FS,AX
        MOV GS,AX
        MOV ESP,[0xfe4] ; 强制返回到start_app时的ESP
        STI ; 切换完成后恢复中断请求
        POPAD ; 恢复事先保存的寄存器值
```

```
        RET
```

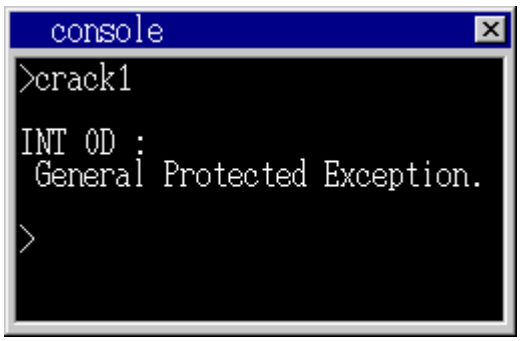
这个函数与20号中断处理函数的区别在于用STI/CLI禁止、允许终端，根据handler的结果决定是否结束程序。

> 在强制结束时，尽管中断处理完成了但却没有使用IRETD指令，而且还把栈强制恢复到start_app时的状态，使程序返回到cmd_app。可能大家会问，这种奇怪的做法真的没问题吗？是的，完全没问题。

然后我们写中断服务程序并将其注册到IDT中

```
int inthandler0d(int *esp)
{
    struct CONSOLE *cons = (struct CONSOLE *) *((int *) 0x0fec);
    cons_putstr0(cons, "\nINT 0D :\n General Protected Exception.\n");
    return 1;
}
```

make run(由于提前知悉了qemu的问题，所以放在virtual box上面跑了)



保护成功!

我们现在可以防御这种简单的bug/攻击了，如果恶意程序直接修改DS寄存器的内容的话，我们的操作系统还是会中招。具体的防御办法是为段加上访问权限

> 在段定义的地方，如果将访问权限加上0x60的话，就可以将段设置为应用程序用。当CS中 的段地址为应用程序用段地址时，CPU会认为“当前正在运行应用程序”，这时如果存入操作系 统用的段地址就会产生异常。

这样CPU就可以自动帮我们做切换了!

```
int cmd_app(struct CONSOLE *cons, int *fat, char *cmdline)
{
    （中略）
    char name[13], *p, *q;
    struct TASK *task = task_now();      /*这里！*/

    （中略）

    if (finfo != 0) {
        /*找到文件的情况*/
        （中略）
        set_segmdesc(gdt + 1003, finfo->size - 1, (int) p, AR_CODE32_ER + 0x60);     /*从此开始*/
        set_segmdesc(gdt + 1004, 64 * 1024 - 1,   (int) q, AR_DATA32_RW + 0x60);
        （中略）
        start_app(0, 1003 * 8, 64 * 1024, 1004 * 8, &(task->tss.esp0));              /*到此结束*/
        （中略）
    }
    /*没有找到文件的情况*/
    return 0;
}
```

还有一个问题，使用这种方法的话，我们就没法进行farcall和farjmp了，怎么办呢？我们使用了一个小trick。我们将跳转地址压入栈中，然后retf。这样就可以规避x86的限制了。

```
_start_app: ; void start_app(int eip, int cs, int esp, int ds, int *tss_esp0);
        PUSHAD ; 将32位寄存器的值全部保存下来
        MOV EAX,[ESP+36] ; 应用程序用EIP
        MOV ECX,[ESP+40] ; 应用程序用CS
        MOV EDX,[ESP+44] ; 应用程序用ESP
        MOV EBX,[ESP+48] ; 应用程序用DS/SS
        MOV EBP,[ESP+52] ; tss.esp0的地址
        MOV [EBP ],ESP ; 保存操作系统用ESP
        MOV [EBP+4],SS ; 保存操作系统用SS
        MOV ES,BX
        MOV DS,BX
        MOV FS,BX
        MOV GS,BX
        ; 下面调整栈，以免用RETF跳转到应用程序
        OR ECX,3 ; 将应用程序用段号和3进行OR运算
        OR EBX,3 ; 将应用程序用段号和3进行OR运算
        PUSH EBX ; 应用程序的SS
        PUSH EDX ; 应用程序的ESP
        PUSH ECX ; 应用程序的CS
        PUSH EAX ; 应用程序的EIP
        RETF
        ; 应用程序结束后不会回到这里
```

由于我们并非使用farcall跳转到应用程序，所以我们从应用程序返回系统就不能使用retf了，需要使用别的办法。

我们搞一个结束应用程序的API，给他功能号4

```
} else if (edx == 4) {          /*这里！*/
    return &(task->tss.esp0);   /*这里！*/
}
```

然后我们把中断处理程序改回去。CPU替我们做过切换了，不改回去会出问题的。

然后我们修改一下IDT，给中断加上权限，只允许应用程序调用hrb_api中断(0x40)

```
set_gatedesc(idt + 0x0d, (int) asm_inthandler0d, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x20, (int) asm_inthandler20, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x21, (int) asm_inthandler21, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x27, (int) asm_inthandler27, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x2c, (int) asm_inthandler2c, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x40, (int) asm_hrb_api, 2 * 8, AR_INTGATE32 + 0x60);
```

应用程序也得修改了，因为我们不能用retf结束程序了，要用api。

Day 22

今天我们来测试一下我们的操作系统到底能防御多少攻击

重新设定定时器

```
MOV AL,0x34
OUT 0x43,AL
MOV AL,0xff
OUT 0x40,AL
MOV AL,0xff
OUT 0x40,AL
MOV EDX,4
INT 0x40
```

防御成功，在应用程序模式下是不允许执行in、out指令的。

关中断

```
CLI
fin:
HTL
JMP fin
```

防御成功，CLI是不允许的

---

farcall

```
CALL 2*8:0xac1
MOV EDX,4
INT 0x40
```

防御成功，far call是不允许的

操作系统有后门，调用特定API

```
} else if (edx == 123456789) {
    *((char *) 0x00102600) = 0;
}
```

```
MOV EDX,123456789
INT 0x40
MOV EDX,4
INT 0x40
```

防御失败，放恶人进来，CPU也救不了你。毕竟你给了应用程序调用0x40中断的权限

bug处理

程序开发人员可能会出各种各样的疏忽，导致应用程序有bug，无意的破坏操作系统。

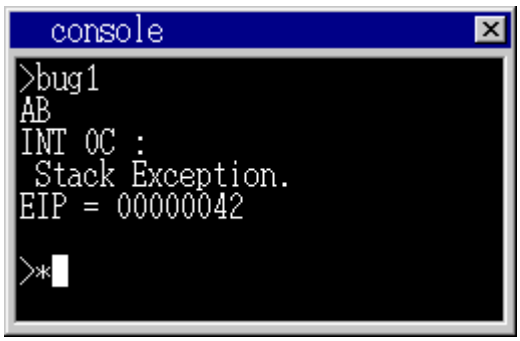
一个很常见的事情就是数组下标访问越界。

```
void api_putchar(int c);
void api_end(void);
void HariMain(void)
{
    char a[100];
    a[10] = 'A';
    api_putchar(a[10]);
    a[102] = 'B'; /*越界*/
    api_putchar(a[102]);
    a[123] = 'C'; /*越界*/
    api_putchar(a[123]);
    api_end();
}
```

virtual box运行，虚拟机重启了

这是因为这个bug触发了新的中断，0x0c，我们没有对他进行处理。

处理函数的代码和0x0d的几乎一模一样，只有输出不太一样。不要忘记注册它到IDT中



我们还可以加上相应的调试信息。

为什么显示了AB之后才产生exception呢?

> 可能有人会问，为什么“C”会被判定为异常而“B”就可以被放过去呢？下面我们就 来简单讲一讲。 a[102]虽然超出了数组的边界，但却没有超出为应用程序分配的数据段的边界，因此虽 然这是个bug，CPU也不会产生异常。另一方面，a[123]所在的地址已经超出了数据段的边界， 因此CPU马上就发现并产生了异常。 其实，CPU产生异常的目的并不是去发现bug，而是为了保护操作系统，它的思路是： “这个程序试图访问自身所在数据段以外的内存地址，一定是想擅自改写操作系统或者其他 应用程序所管理的内存空间，这种行为岂能放任不管？”因此，即便CPU不能帮我们发现所 有的bug，也不可以责怪它哦。

死循环处理

```
void HariMain(void) {
    for (;;) { }
}
```

遇到这样的空循环怎么办？操作系统和用户都无可奈何。我们可以仿照我们日常使用的操作系统中的设计，使用特定的按键来终结正在运行的程序。

命令行窗口在运行的时候并不会检查他的FIFO队列，所以我们不能在console当中进行处理。因为console并不会知道我们按下了结束程序的快捷键。我们把对强制结束的处理放在bootpack。

我们把强制结束键设置成`Shift`+`F1`吧

```
if (i == 256 + 0x3b && key_shift != 0 && task_cons->tss.ss0 != 0) { /* Shift+F1 */
    cons = (struct CONSOLE *) *((int *) 0x0fec);
    cons_putstr0(cons, "\nBreak(key):\n");
    io_cli();
    task_cons->tss.eax = (int) &(task_cons->tss.esp0);
    task_cons->tss.eip = (int) asm_end_app;
    io_sti();
}
```

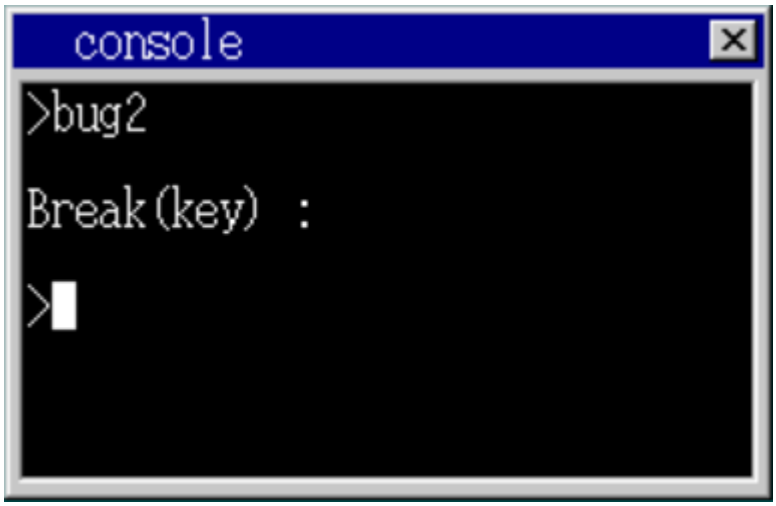
这段代码的原理是，当按下`Shift`+`F1`时，修改命令行窗口的寄存器值，然后goto到asm_end_app。光这样做还不够，我们在应用程序没有运行的时候按下`Shift`+`F1`键会出问题，也会进行跳转。

我们可以用tss中的ss0来进行标记

```
struct TASK *task_alloc(void)
{
    int i;
    struct TASK *task;
    for (i = 0; i < MAX_TASKS; i++) {
        if (taskctl->tasks0[i].flags == 0) {
        task = &taskctl->tasks0[i];
        task->flags = 1;
        task->tss.eflags = 0x00000202; /* IF = 1; */
        task->tss.eax = 0;
        // ........
        task->tss.iomap = 0x40000000;
        task->tss.ss0 = 0; /*这里! */
        return task;
```

```
        }
    }
    return 0; /*已经全部正在使用*/
}
```

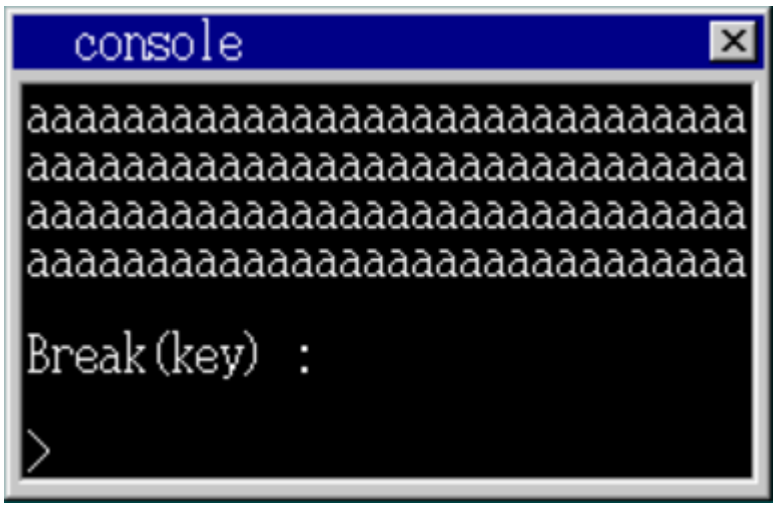
make run



按下 Shift + F1 。成功结束程序

```
void api_putchar(int c);
void api_end(void);
void HariMain(void)
{
    for (;;) {
        api_putchar('a');
    }
}
```

测试一下这个



我们来实现写入字符串的API

由于现在使用API来结束程序，所以我们修改开头6字节的操作也不必要了。我们直接在start_app时设定起始运行地址就ok了

遇到了一些麻烦.....

```
void api_putstr0(char *s);
void api_end(void);
void HariMain(void)
{
    api_putstr0("hello, world\n");
    api_end();
}
```

这段程序并没有正确的打印helloworld，我们的API看起来也似乎没什么问题。

通过阅读我们了解到，程序出现问题的主要原因是无法定位数据部分

hrb前32个字节的内容和定义

0x0000 (DWORD)	请求操作系统为应用程序准备的数据段的大小
0x0004 (DWORD)	“Hari ”（.hrb 文件的标记）
0x0008 (DWORD)	数据段内预备空间的大小
0x000c (DWORD)	ESP 初始值& 数据部分传送目的地址
0x0010 (DWORD)	hrb 文件内数据部分的大小
0x0014 (DWORD)	hrb 文件内数据部分从哪里开始
0x0018 (DWORD)	0xe9000000
0x001c (DWORD)	应用程序运行入口地址 - 0x20
0x0020 (DWORD)	malloc

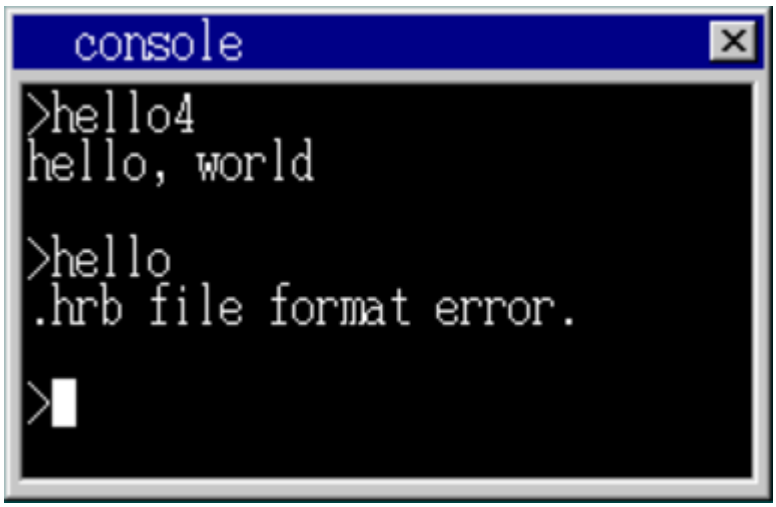
我们要完成以下几个任务

- 文件中找不到“Hari”标志则报错。
- 数据段的大小根据.hrb文件中指定的值进行分配。
- 将.hrb文件中的数据部分先复制到数据段后再启动程序。

```
if (finfo->size >= 36 && strncmp(p + 4, "Hari", 4) == 0 && *p == 0x00) {
    segsiz = *((int *) (p + 0x0000));
    esp = *((int *) (p + 0x000c));
    datsiz = *((int *) (p + 0x0010));
    dathrb = *((int *) (p + 0x0014));  //  根据上面的约定获取各种信息
    q = (char *) memman_alloc_4k(memman, segsiz);
```

```
    *((int *) 0xfe8) = (int) q;
    set_segmdesc(gdt + 1003, finfo->size - 1, (int) p, AR_CODE32_ER + 0x60);
    set_segmdesc(gdt + 1004, segsiz - 1, (int) q, AR_DATA32_RW + 0x60);
    for (i = 0; i < datsiz; i++) {  // 拷贝数据区数据过去
        q[esp + i] = p[dathrb + i];
    }
    start_app(0x1b, 1003 * 8, esp, 1004 * 8, &(task->tss.esp0));
    memman_free_4k(memman, (int) q, segsiz);
}
```

测试



再写一个汇编语言的程序

```
[FORMAT "WCOFF"]
[INSTRSET "i486p"]
[BITS 32]
[FILE "hello5.nas"]
        GLOBAL  _HariMain
[SECTION .text]
_HariMain:
        MOV     EDX,2
        MOV     EBX,msg
        INT     0x40
        MOV     EDX,4
        INT     0x40
[SECTION .data]
msg:
        DB  "hello, world", 0x0a, 0
```

利用bim2hrb我们生成上文所提到的那些信息，我们就可以提前分配好数据区了。

显示窗口。之前我们一直在跟字符界面打交道，我们现在来给我们的应用程序添加UI功能。为系统添加创建窗口的API

EDX	5
EBX	窗口缓冲区
ESI	窗口在x轴方向上的大小（即窗口宽度）
EDI	窗口在y轴方向上的大小（即窗口高度）
EAX	透明色
ECX	窗口名称

调用后的返回值：

EAX	用于操作窗口的句柄（用于刷新窗口等操作）

之前我们写的API都不具有返回值，这次我们要为API添加返回值，那么如何返回寄存器的值呢？

我们在asm_hrb_api中进行了两次push_ad。第一次是为了保存现场，第二次是为了传参。只要我们把参数修改掉，就可以通过popad获得修改后的值，从而实现返回值的功能。

```
struct SHTCTL *shtctl = (struct SHTCTL *) *((int *) 0x0fe4); /*从此开始*/
struct SHEET *sht;
int *reg = &eax + 1; // 保存的现场
/* reg[0] : EDI, reg[1] : ESI, reg[2] : EBP, reg[3] : ESP */
/* reg[4] : EBX, reg[5] : EDX, reg[6] : ECX, reg[7] : EAX */
```

```
else if (edx == 5) {
    sht = sheet_alloc(shtctl);
    sheet_setbuf(sht, (char *) ebx + ds_base, esi, edi, eax);
    make_window8((char *) ebx + ds_base, esi, edi, (char *) ecx + ds_base, 0);
    sheet_slide(sht, 100, 50);
    sheet_updown(sht, 3); /*背景层高度3位于task_a之上*/
    reg[7] = (int) sht;
}
```

```
_api_openwin: ; int api_openwin(char *buf, int xsiz, int ysiz, int col_inv, char *title);
PUSH EDI
PUSH ESI
PUSH EBX
MOV EDX,5
MOV EBX,[ESP+16] ; buf
MOV ESI,[ESP+20] ; xsiz
MOV EDI,[ESP+24] ; ysiz
MOV EAX,[ESP+28] ; col_inv
MOV ECX,[ESP+32] ; title
INT 0x40
POP EBX
POP ESI
POP EDI
RET
```

以上是我们的API设计

我们编写个测试程序：

```
int api_openwin(char *buf, int xsiz, int ysiz, int col_inv, char *title);
void api_end(void);
char buf[150 * 50];
void HariMain(void)
{
    int win;
    win = api_openwin(buf, 150, 50, -1, "hello");
    api_end();
}
```

make run

成功显示！

再接再厉，添加一些在窗口中绘制其他元素的API

显示字符API

EDX	6
EBX	窗口句柄
ESI	显示位置的x坐标
EDI	显示位置的y坐标
EAX	色号
ECX	字符串长度
EBP	字符串

绘制方块API

EDX	7
EBX	窗口句柄
EAX	x0
ECX	y0
ESI	x1
EDI	y1
EBP	色号

hrb_api部分

```
else if (edx == 6) {
    sht = (struct SHEET *) ebx;
    putfonts8_asc(sht->buf, sht->bxsize, esi, edi, eax, (char *) ebp + ds_base);
    sheet_refresh(sht, esi, edi, esi + ecx * 8, edi + 16);
} else if (edx == 7) {
    sht = (struct SHEET *) ebx;
    boxfill8(sht->buf, sht->bxsize, ebp, eax, ecx, esi, edi);
    sheet_refresh(sht, eax, ecx, esi + 1, edi + 1);
}
```

然后我们修改应用程序

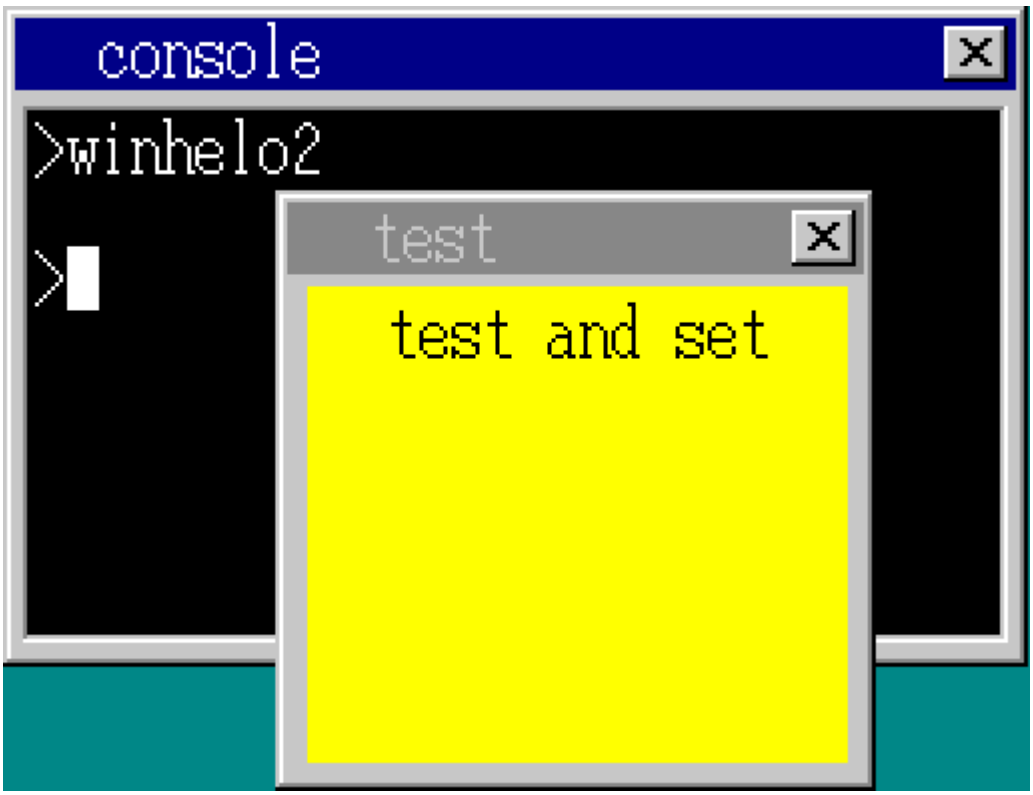
修改a_nask.nas

```
_api_putstrwin: ; void api_putstrwin(int win, int x, int y, int col, int len, char *str);
        PUSH EDI
        PUSH ESI
        PUSH EBP
        PUSH EBX
```

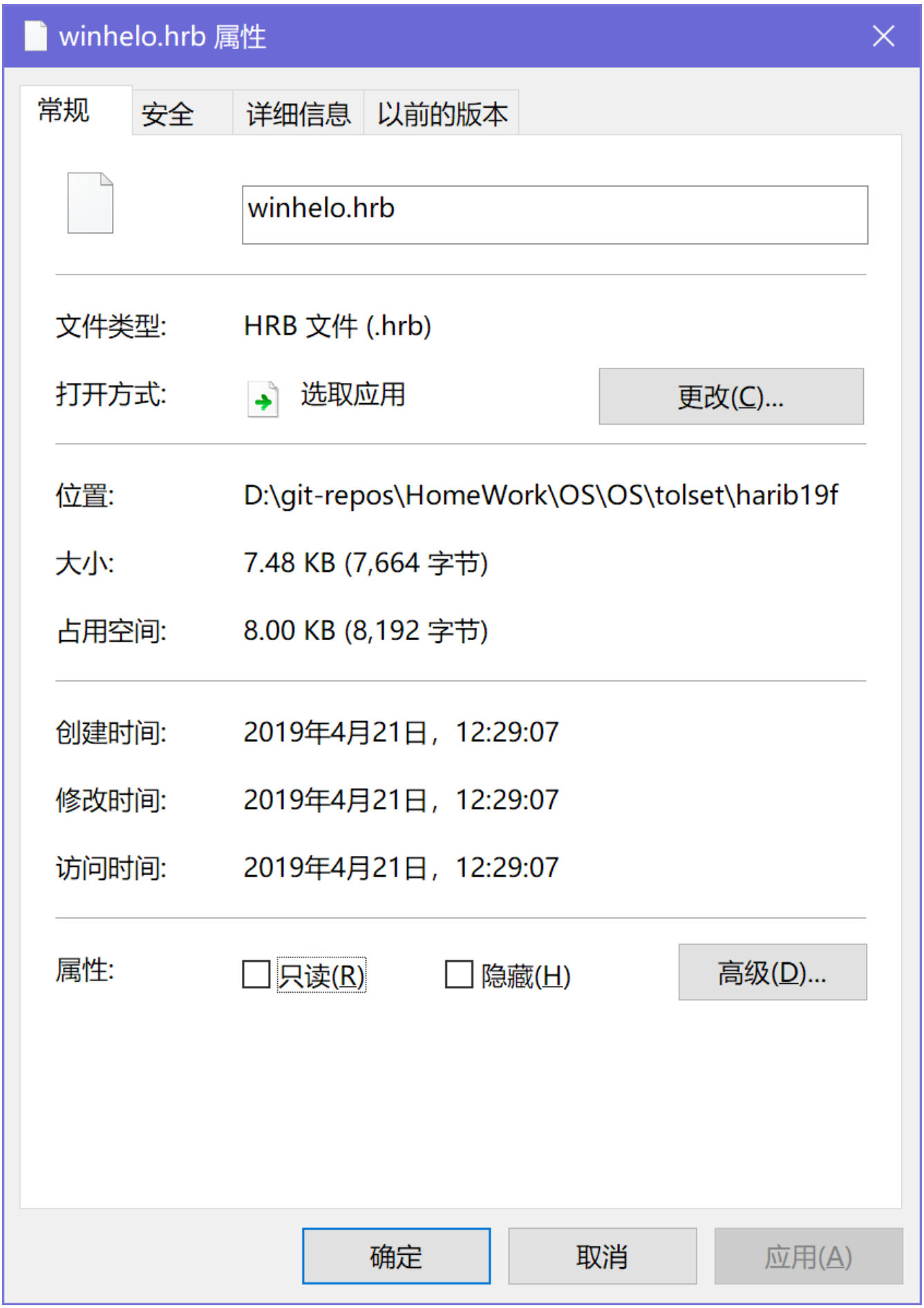
```
        MOV EDX,6
        MOV EBX,[ESP+20] ; win
        MOV ESI,[ESP+24] ; x
        MOV EDI,[ESP+28] ; y
        MOV EAX,[ESP+32] ; col
        MOV ECX,[ESP+36] ; len
        MOV EBP,[ESP+40] ; str
        INT 0x40
        POP EBX
        POP EBP
        POP ESI
        POP EDI
        RET
_api_boxfilwin: ; void api_boxfilwin(int win, int x0, int y0, int x1, int y1, int col);
        PUSH EDI
        PUSH ESI
        PUSH EBP
        PUSH EBX
        MOV EDX,7
        MOV EBX,[ESP+20] ; win
        MOV EAX,[ESP+24] ; x0
        MOV ECX,[ESP+28] ; y0
        MOV ESI,[ESP+32] ; x1
        MOV EDI,[ESP+36] ; y1
        MOV EBP,[ESP+40] ; col
        INT 0x40
        POP EBX
        POP EBP
        POP ESI
        POP EDI
        RET
```

我们测试如下程序

```
void HariMain(void)
{
    int win;
    win = api_openwin(buf, 150, 150, -1, "test");
    api_boxfilwin(win,  8, 24, 142, 142, 3);
    api_putstrwin(win, 28, 28, 0, 12, "test and set");
    api_end();
}
```


console
>winhelo2
>
test
test and set

Day 23



这个是我们之前实验当中winhelo程序的详细信息，可以看到这个文件达到了惊人的7.48KB大小。使用二进制查看器

```
00000000  00 01 02 03  04 05 06 07  08 09 0a 0b  0c 0d 0e 0f
00000000  00 30 00 00  48 61 72 69  00 00 00 00  00 04 00 00   .0..Hari........
00000010  5c 1d 00 00  94 00 00 00  00 00 00 e9  6b 00 00 00   \...?.......閏...
00000020  60 21 00 00  55 89 e5 68  00 04 00 00  6a ff 6a 32   `!..U夂h....j j2
00000030  68 96 00 00  00 68 10 04  00 00 e8 2a  00 00 00 83   h?...h....?  ...葸
00000040  c4 14 c9 e9  1a 00 00 00  ba 01 00 00  00 8a 44 24     .砷 ....?  ...奄$
00000050  04 cd 40 c3  53 ba 02 00  00 00 8b 5c  24 08 cd 40   .虮 胚 ?  ...嫕$.虮
00000060  5b c3 ba 04  00 00 00 cd  40 57 56 53  ba 05 00 00   [煤 .... 虮 WVS?  ..
00000070  00 8b 5c 24  10 8b 74 24  14 8b 7c 24  18 8b 44 24   .嫕$.娕$.媚$.婦$
00000080  1c 8b 4c 24  20 cd 40 5b  5e 5f c3 55  89 e5 5d e9   .娩$ 虮 [^_胱夂]閻
00000090  90 ff ff ff  68 65 6c 6c  6f 00 00 00  00 00 00 00        hello.......
000000a0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000000b0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000000c0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000000d0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000000e0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000000f0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000100  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000110  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000120  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000130  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000140  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000150  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000160  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000170  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000180  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000190  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000001a0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000001b0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000001c0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000001d0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000001e0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
```

可以看到这个文件当中有大量的00，这显然有很多是没有用的。这些0是哪里来的呢?

原因在于winhelo2.c的 `char buf[150 * 50];` 这相当于在可执行文件中插入了150×50 = 7500个字节的00。

如果我们的操作系统可以在运行的时候动态给程序分配内存的话就好了。我们仿照c语言的命名，把这个api叫做malloc。

api_malloc

malloc不可以直接调用操作系统的memman_malloc，这是因为mamman_malloc分配的空间是在系统段当中，应用程序是无法访问这个区域的。

作者给出了一个什么样的解决方案呢？应用程序提前指定好自己会使用的最大内存量，提前给他准备好，当应用程序请求分配内存的时候就从这段内存当中拿出来一部分给他就好了。

在哪里给定这样的值呢？作者写的bim2hrb工具接受的第三个参数可以进行指定

```
$(BIM2HRB) <filename>.bim <filename>.hrb <预存空间大小>
```

而这个指定的参数，则保存在hrb文件的0x0020处

mamman设计

初始化api

EDX	8
EBX	memman的地址
EAX	memman所管理的内存空间的起始地址
ECX	memman所管理的内存空间的字节数

分配api

EDX	9
EBX	memman的地址
ECX	需要请求的字节数
EAX	分配到的内存空间地址

释放api

EDX	10
EBX	memman的地址
EAX	需要释放的内存空间地址
ECX	需要释放的字节数

hrb_api修改的部分

```
else if (edx == 8) {
    memman_init((struct MEMMAN *) (ebx + ds_base));
    ecx &= 0xfffffff0; /*以16字节为单位*/
    memman_free((struct MEMMAN *) (ebx + ds_base), eax, ecx);
} else if (edx == 9) {
    ecx = (ecx + 0x0f) & 0xfffffff0; /*以16字节为单位进位取整*/
    reg[7] = memman_alloc((struct MEMMAN *) (ebx + ds_base), ecx);
} else if (edx == 10) {
    ecx = (ecx + 0x0f) & 0xfffffff0; /*以16字节为单位进位取整*/
    memman_free((struct MEMMAN *) (ebx + ds_base), eax, ecx);
}
```

再整个画点的

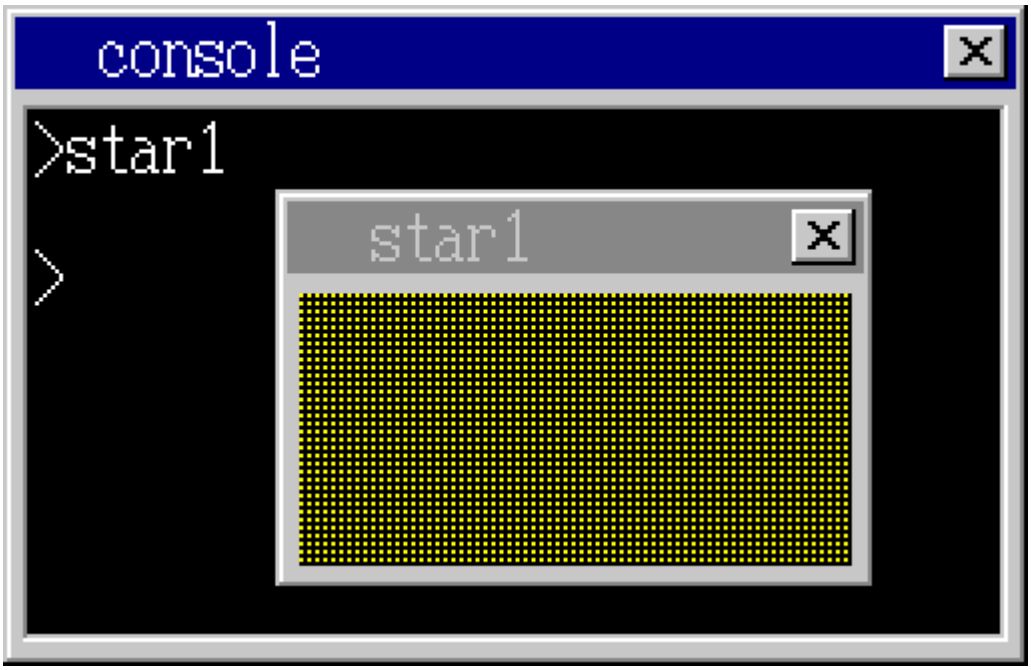
EDX	11
EBX	窗口句柄
ESI	显示位置的x坐标
EDI	显示位置的y坐标
EAX	色号

```
} else if (edx == 11) {
    sht = (struct SHEET *) ebx;
    sht->buf[sht->bxsize * edi + esi] = eax;
    sheet_refresh(sht, esi, edi, esi + 1, edi + 1);
}
```

测试程序

```
void HariMain(void)
{
    char *buf;
    int win, y, x;
    api_initmalloc();
    buf = api_malloc(150 * 100);
    win = api_openwin(buf, 150, 100, -1, "star1");
    api_boxfilwin(win,  6, 26, 143, 93, 0 /* 黑 */);
    for (y = 26; y <= 93; y += 2) {
        for (x = 6; x <= 143; x += 2) {
            api_point(win, x, y, 3);
        }
    }
    api_end();
```

```
}
```



我们发现，每次画点都会进行一次窗体刷新。这种做法效率是很低的，因为我们每次更新一个像素点，却要重新绘制整个窗口区域。可以考虑接受一个参数，从而决定是否在画点之后进行窗体刷新，并且设计一个新的刷新窗口的API

EDX	12
EBX	窗口句柄
EAX	x0
ECX	y0
ESI	x1
EDI	y1

```
else if (edx == 11) {
    if ((ebx & 1) == 0) {
        sht->buf[((struct SHEET *)(ebx))->bxsize * edi + esi] = eax;
        sheet_refresh((struct SHEET *)(ebx), esi, edi, esi + 1, edi + 1);
    } else {
        sht->buf[((struct SHEET *)(ebx ^ 1))->bxsize * edi + esi] = eax;
        sheet_refresh((struct SHEET *)(ebx ^ 1), esi, edi, esi + 1, edi + 1);
    }
} else if (edx == 12) {
    sht = (struct SHEET *) ebx;
    sheet_refresh(sht, eax, ecx, esi, edi);
}
```

为了节省参数，可以把这个信息融入到窗口句柄中。struct SHEET的地址一定是一个偶数，那 么我们可以让程序在指定一个奇数（即在原来的数值上加1）的情况下不进行自动刷新。

```
void HariMain(void)
{
    char *buf;
    int win, y, x;
    api_initmalloc();
    buf = api_malloc(150 * 100);
    win = api_openwin(buf, 150, 100, -1, "star1");
    api_boxfilwin(win,  6, 26, 143, 93, 0 /* 黑 */);
    for (y = 26; y <= 93; y += 2) {
        for (x = 6; x <= 143; x += 2) {
            api_point(win, x, y, 3);
        }
    }
    api_refreshwin(win, 6, 26, 144, 94);
    api_end();
}
```

运行正常

画直线

EDX	13
EBX	窗口句柄
EAX	x0
ECX	y0
ESI	x1
EDI	y1
EBP	色号

```
} else if (edx == 13) {
    sht = (struct SHEET *) (ebx & 0xfffffffe);
    hrb_api_linewin(sht, eax, ecx, esi, edi, ebp);
    if ((ebx & 1) == 0) {
        sheet_refresh(sht, eax, ecx, esi + 1, edi + 1);
    }
}
```

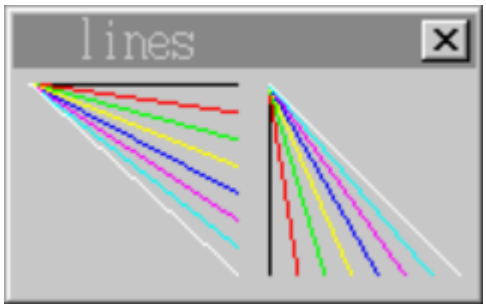
```
void hrb_api_linewin(struct SHEET *sht, int x0, int y0, int x1, int y1, int col)
{
    int i, x, y, len, dx, dy;
    dx = x1 - x0;
    dy = y1 - y0;
```

```
    x = x0 << 10;
    y = y0 << 10;
    if (dx < 0) {
        dx = - dx;
    }
    if (dy < 0) {
        dy = - dy;
    }
    if (dx >= dy) {
        len = dx + 1;
        if (x0 > x1) {
            dx = -1024;
        } else {
            dx = 1024;
        }
        if (y0 <= y1) {
            dy = ((y1 - y0 + 1) << 10) / len;
        } else {
            dy = ((y1 - y0 - 1) << 10) / len;
        }
    } else {
        len = dy + 1;
        if (y0 > y1) {
            dy = -1024;
        } else {
            dy = 1024;
        }
        if (x0 <= x1) {
            dx = ((x1 - x0 + 1) << 10) / len;
        } else {
            dx = ((x1 - x0 - 1) << 10) / len;
        }
    }
    for (i = 0; i < len; i++) {
        sht->buf[(y >> 10) * sht->bxsize + (x >> 10)] = col;
        x += dx;
        y += dy;
    }
    return;
}
```

```
int api_openwin(char *buf, int xsiz, int ysiz, int col_inv, char *title);
void api_initmalloc(void);
char *api_malloc(int size);
void api_refreshwin(int win, int x0, int y0, int x1, int y1);
void api_linewin(int win, int x0, int y0, int x1, int y1, int col);
void api_end(void);
void HariMain(void)
{
    char *buf;
    int win, i;
    api_initmalloc();
    buf = api_malloc(160 * 100);
```

```
    win = api_openwin(buf, 160, 100, -1, "lines");
    for (i = 0; i < 8; i++) {
        api_linewin(win + 1, 8, 26, 77, i * 9 + 26, i);
        api_linewin(win + 1, 88, 26, i * 9 + 88, 89, i);
    }
    api_refreshwin(win, 6, 26, 154, 90);
    api_end();
}
```

略去了汇编语言的部分QAQ



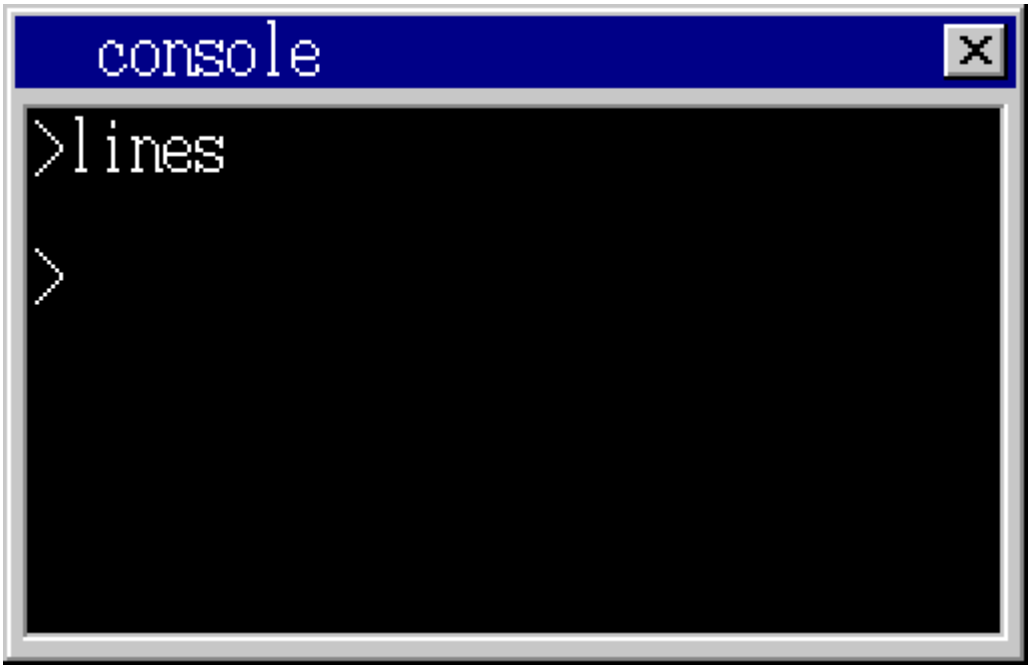
关闭窗口

应用程序结束以后，窗口也应当清除

EDX	14
EBX	窗口句柄

```
} else if (edx == 14) {
    sheet_free((struct SHEET *) ebx);
}
```

```
void HariMain(void)
{
    char *buf;
    int win, i;
    api_initmalloc();
    buf = api_malloc(160 * 100);
    win = api_openwin(buf, 160, 100, -1, "lines");
    for (i = 0; i < 8; i++) {
        api_linewin(win + 1, 8, 26, 77, i * 9 + 26, i);
        api_linewin(win + 1, 88, 26, i * 9 + 88, 89, i);
    }
    api_refreshwin(win, 6, 26, 154, 90);
    api_closewin(win); /*这里!  */
    api_end();
}
```



键盘输入

EDX	15
EAX	0……没有键盘输入时返回□1，不休眠
	1……休眠直到发生键盘输入
EAX	输入的字符编码

```
else if (edx == 15) {
    for (;;) {
        io_cli();
        if (fifo32_status(&task->fifo) == 0) {
            if (eax != 0) {
                task_sleep(task); /* FIFO为空，休眠并等待*/
            } else {
                io_sti();
                reg[7] = -1;
                return 0;
            }
        }
        i = fifo32_get(&task->fifo);
        io_sti();
        if (i <= 1) { /*光标用定时器*/
            /*应用程序运行时不需要显示光标，因此总是将下次显示用的值置为1*/
            timer_init(cons->timer, &task->fifo, 1); /*下次置为1*/
            timer_settime(cons->timer, 50);
        }
        if (i == 2) { /*光标ON */
            cons->cur_c = COL8_FFFFFF;
```

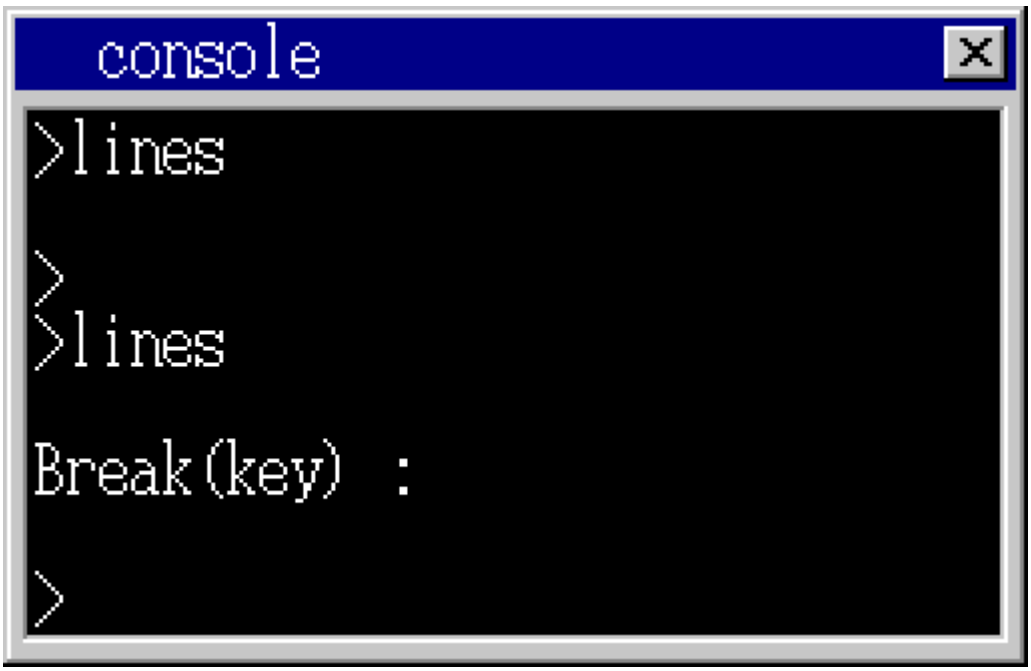
```
        }
        if (i == 3) { /*光标OFF */
            cons->cur_c = -1;
        }
        if (256 <= i && i <= 511) { /*键盘数据（通过任务A）*/
            reg[7] = i - 256;
            return 0;
        }
    }
}
```

我们将定时器放入了console当中

```
void api_closewin(int win);
int api_getkey(int mode);
void api_end(void);
void HariMain(void)
{
     (中略)
        api_refreshwin(win, 6, 26, 154, 90);
    for (;;) {
        if (api_getkey(1) == 0x0a) {
            break;
        }
    }
    api_closewin(win);
    api_end();
}
```

我们等待回车键，回车键按下后再结束程序。

对了。如果用的是`Shift`+`F1`结束程序的话，窗口还是不会自动消失，这不够好。我们可以在sheet中添加一个task字段，里面写了它对应的应用程序。我们结束程序的时候检测以下所有sheet，如果有属于这个应用程序的sheet，那么我们就把它free掉。


console
>lines
>
>lines
Break(key) :
>