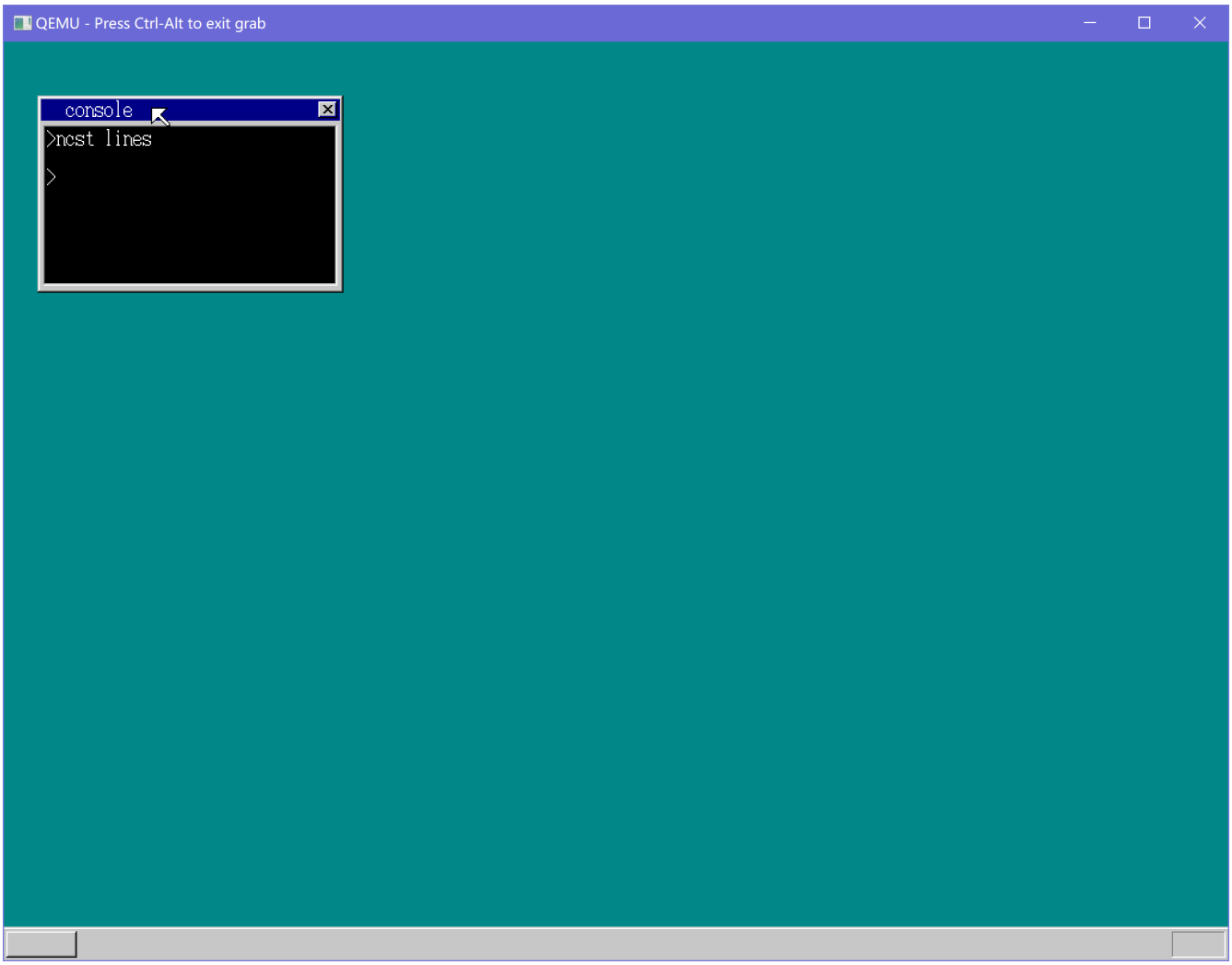
褚逸豪 学号2017211123 序号17

Day 27

之前我们完成的那一版中，使用ncst启动的程序无法关闭：无论是点击`X`还是按`Shift`+`F1`，都无法关闭程序。不过看起来我们Day26的代码当中并没有什么问题，所以我们应当把查错范围放大一些。

我们的解决方案是在`Shift`+`F1`和`X`的关闭逻辑后再加上`task_run(task, -1, 0);`也就是唤醒要关闭的应用程序。原因到现在显而易见了：如果程序一直在休眠的话，那么他就无法运行关闭程序的逻辑，从而无法退出了。这也是为什么那些具有光标闪烁的窗体能够被正常关闭的原因：闪烁的光标保证这些任务每隔一段时间就被唤醒一次。



lines被关闭了

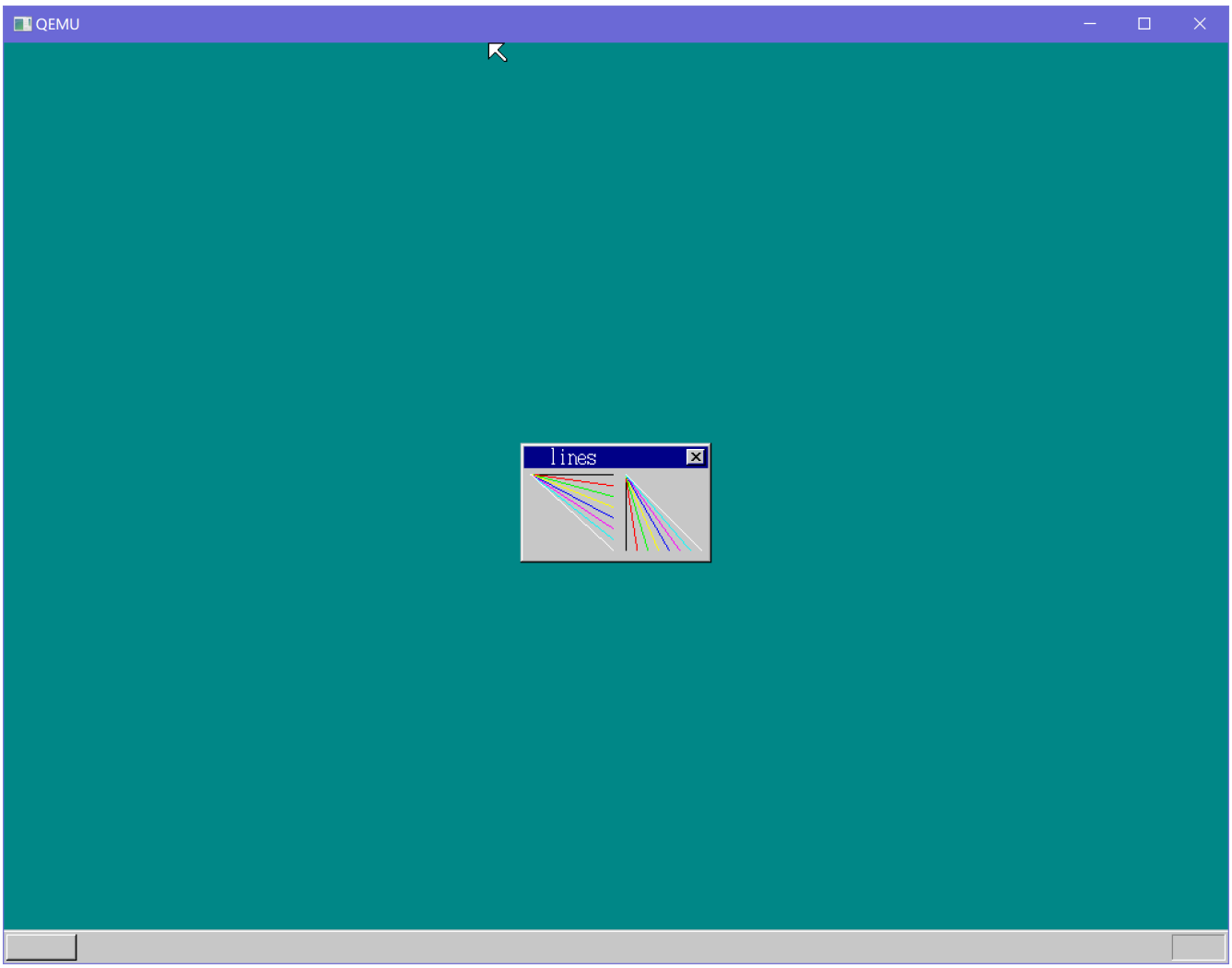
然后我们来实现运行时关闭命令行窗口。目前我们的程序在运行时，它所对应的命令行窗口是不响应的，没有什么用，我们来实现运行时关闭命令行。

具体实现思路呢，是暂时隐藏图层，然后再进行进一步的清理

```
if (sht->bxsize - 21 <= x && x < sht->bxsize - 5 && 5 <=
```

```
    y && y < 19) {/* 点击X按钮*/
    if ((sht->flags & 0x10) != 0) { /*应用程序窗口*/
        //.....................
    } else { /*命令行窗口*/
        task = sht->task;
        sheet_updown(sht, -1); /*隐藏该图层*/
        keywin_off(key_win);
        key_win = shtctl->sheets[shtctl->top - 1];
        keywin_on(key_win);
        io_cli();
        fifo32_put(&task->fifo, 4);
        io_sti();
    }
}
```

然后我们要把console_task中的sheet变量全部修改为cons.sht变量。这样可以就可以在窗口关闭后得知窗体已经关闭，从而不再在窗体上进行输出了。



接着进行今天的正式内容，LDT的设置

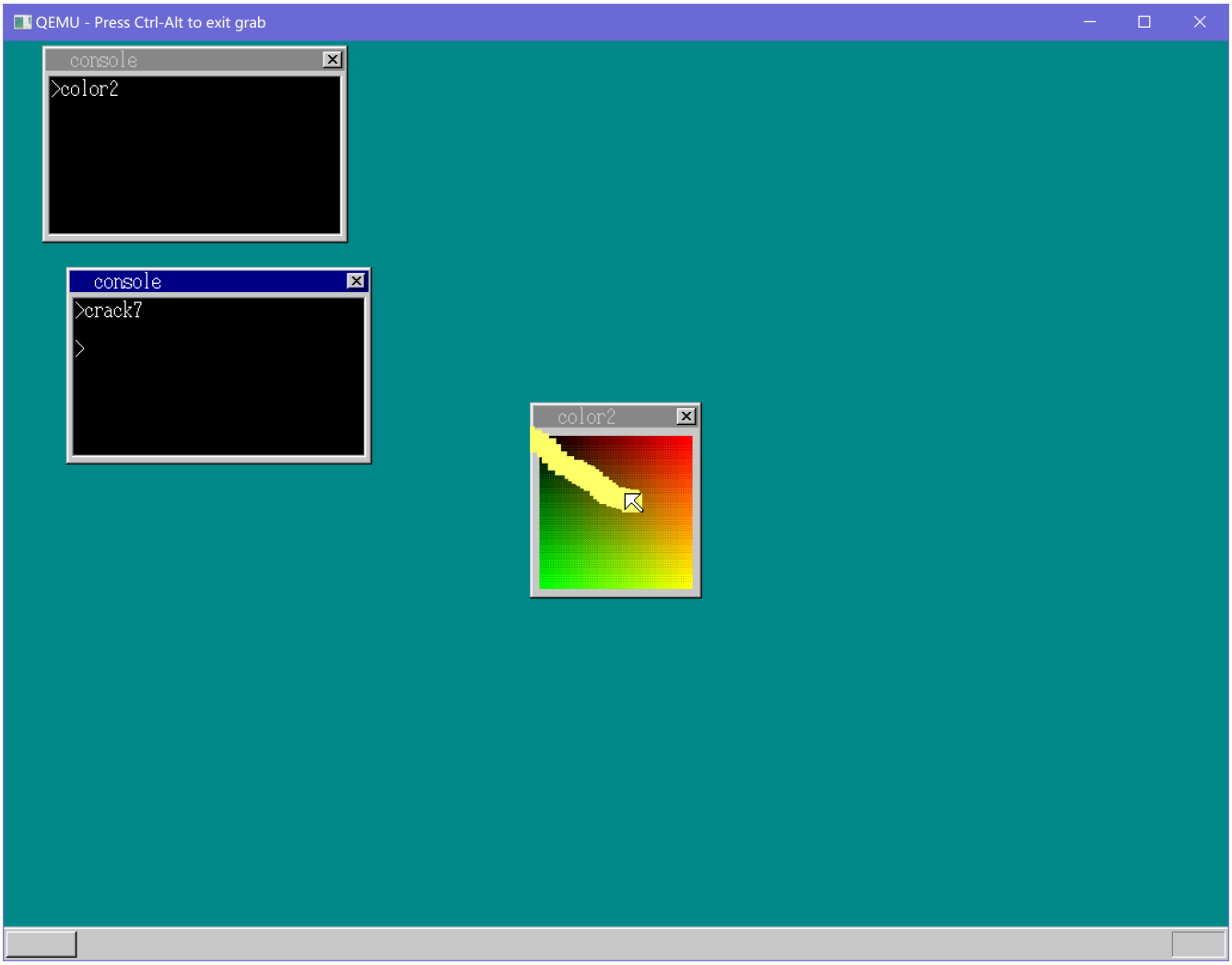
先来对系统进行破坏。

```
[FORMAT "WCOFF"]
[INSTRSET "i486p"]
[BITS 32]
[FILE "crack7.nas"]

        GLOBAL _HariMain

[SECTION .text]

_HariMain:
        MOV AX,1005*8
        MOV DS,AX
        CMP DWORD [DS:0x0004],'Hari'
        JNE fin ; 不是应用程序，因此不执行任何操作
        MOV ECX,[DS:0x0000] ; 读取该应用程序数据段的大小
        MOV AX,2005*8
        MOV DS,AX
crackloop: ; 整个用123填充
        ADD ECX,-1
        MOV BYTE [DS:ECX],123
        CMP ECX,0
        JNE crackloop
fin: ; 结束
        MOV EDX,4
        INT 0x40
```



可以看到正在运行的应用程序被破坏了。

原因是crack7修改了正在运行的color2程序的数据段。系统没法进行防护的主要原因是crack7并没有访问操作系统的段（摊手

为了解决这个问题，我们要利用CPU提供的LDT功能。只要为每个程序设置对应的LDT，CPU就可以把程序对内存的访问限制在它所对应的段中。

```
struct TASK *task_init(struct MEMMAN *memman)
{
    //.....................
    for (i = 0; i < MAX_TASKS; i++) {
        taskctl->tasks0[i].flags = 0;
        taskctl->tasks0[i].sel = (TASK_GDT0 + i) * 8;
        taskctl->tasks0[i].tss.ldtr = (TASK_GDT0 + MAX_TASKS + i) * 8;
        set_segmdesc(gdt + TASK_GDT0 + i, 103, (int) &taskctl->tasks0[i].tss, AR_TSS32);
        set_segmdesc(gdt + TASK_GDT0 + MAX_TASKS + i, 15, (int) taskctl->tasks0[i].ldt,
AR_LDT);
    }
    //.....................
}
```

然后删除task_alloc中的`task->tss.ldtr = 0;`

我们之前没设置过ldtr，我们现在进行设置。

然后我们修改启动程序的代码，在段号加上4，来表示他是LDT（也就是将Table Indicator位置成1）

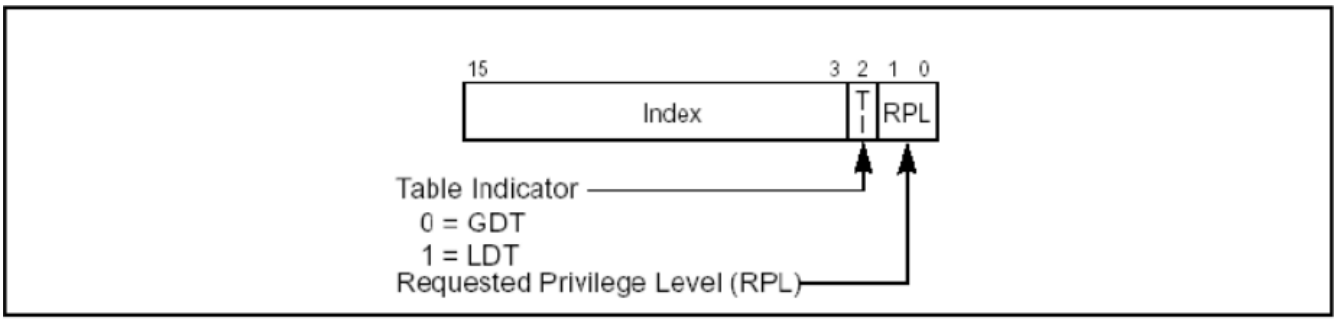
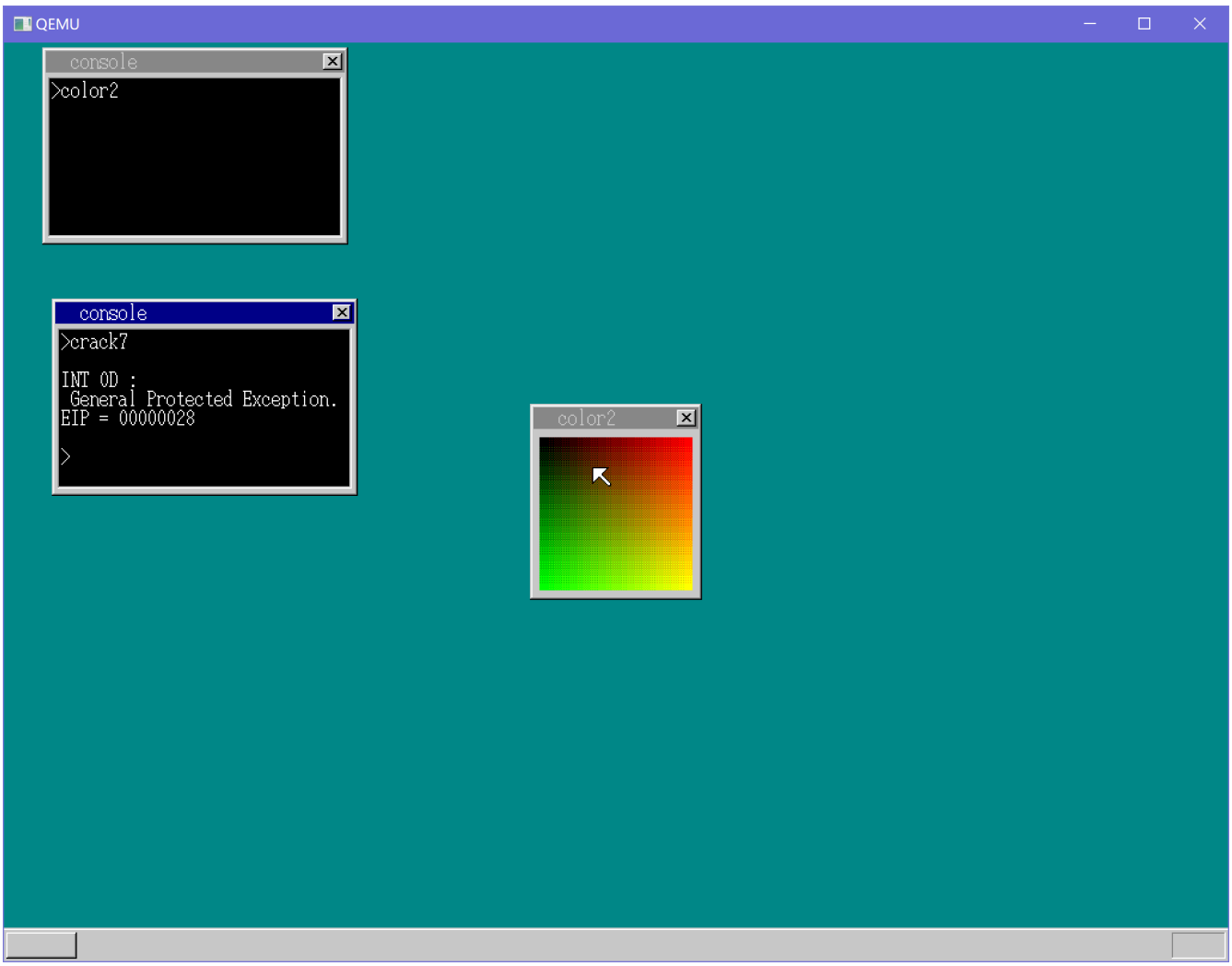


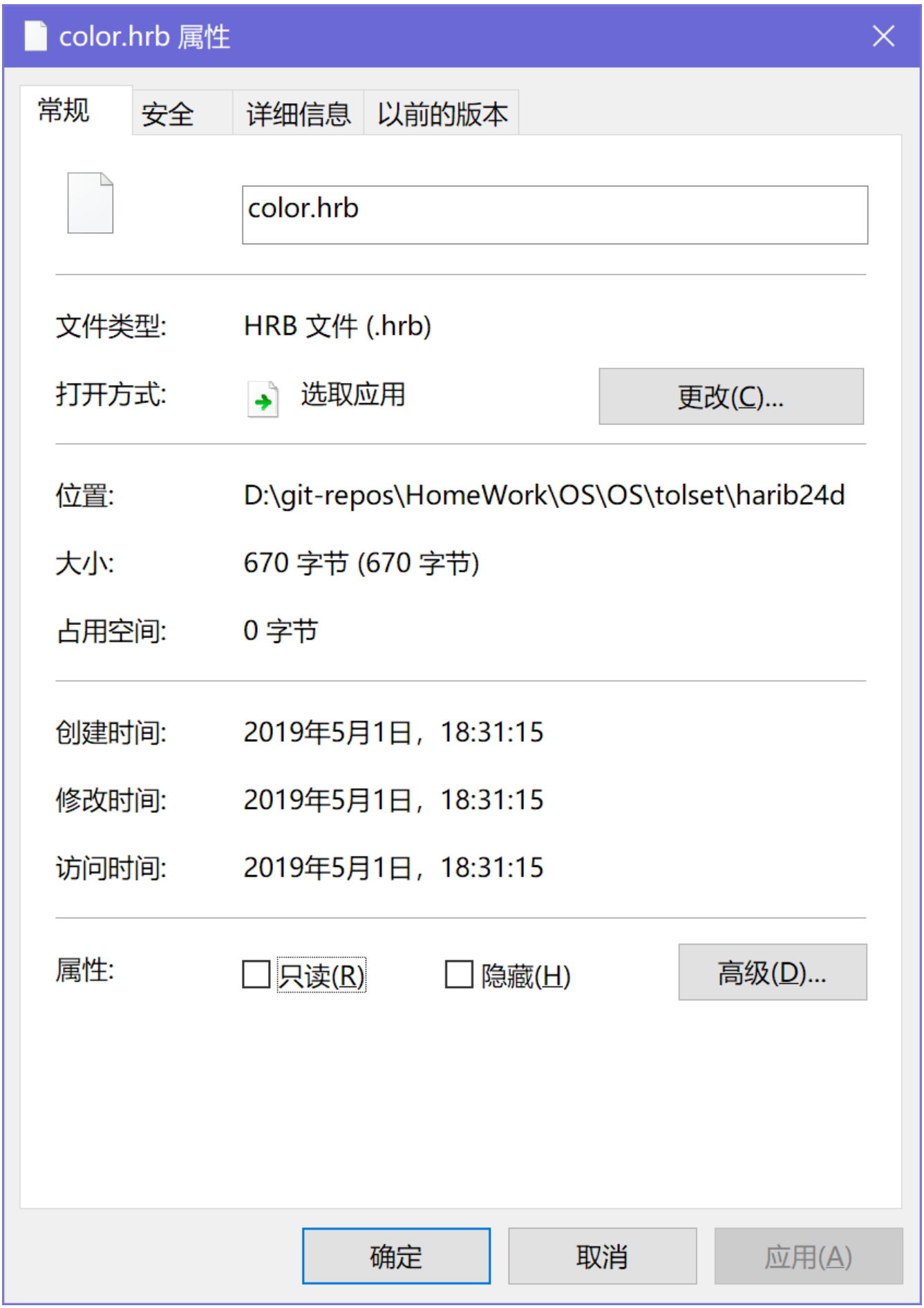
Figure 3-6. Segment Selector

```
set_segmdesc(task->ldt + 0, finfo->size - 1, (int) p, AR_CODE32_ER + 0x60);
set_segmdesc(task->ldt + 1, segsiz - 1, (int) q, AR_DATA32_RW + 0x60);
for (i = 0; i < datsiz; i++) {
    q[esp + i] = p[dathrb + i];
}
start_app(0x1b, 0 * 8 + 4, esp, 1 * 8 + 4, &(task->tss.esp0));
```

这样就大功告成了

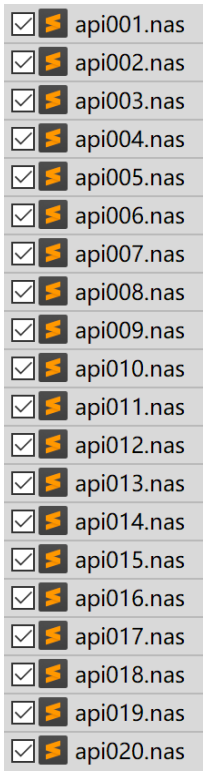


如果还想搞破坏的话，那么恶意程序只能使用lldt来重设ldtr到别的程序，但lldt是系统专用指令，所以这个操作是会被CPU拦截下来的。


color.hrb 属性
常规
安全
详细信息
以前的版本
color.hrb
文件类型:
HRB 文件 (.hrb)
打开方式:
选取应用
更改(C)...
位置:
D:\git-repos\HomeWork\OS\OS\tolset\harib24d
大小:
670 字节 (670 字节)
占用空间:
0 字节
创建时间:
2019年5月1日，18:31:15
修改时间:
2019年5月1日，18:31:15
访问时间:
2019年5月1日，18:31:15
属性:
只读(R)
隐藏(H)
高级(D)...
确定
取消
应用(A)

应用程序的大小急剧的增长，究其原因，是因为不管他用没用到某些API，所有API的代码都被添加到了应用程序当中，究其原因，是因为尽管链接器可以按需链接，但链接一次链接会一整个obj文件，并不会单独的链接其中的某个函数。

接下来我们将API拆分成单独的文件，这样就可以按需链接了。



然后我们修改makefile，把之前的a_nask.nas改成api001~020.nas

不过这样还是有些麻烦，我们使用库管理器来对这些obj文件进行管理。作者提供了一个库管理器。修改makefile，添加

```
GOLIB = $(TOOLPATH)golib00.exe

apilib.lib : Makefile $(OBJS_API)
    $(GOLIB) $(OBJS_API) out:apilib.lib
```

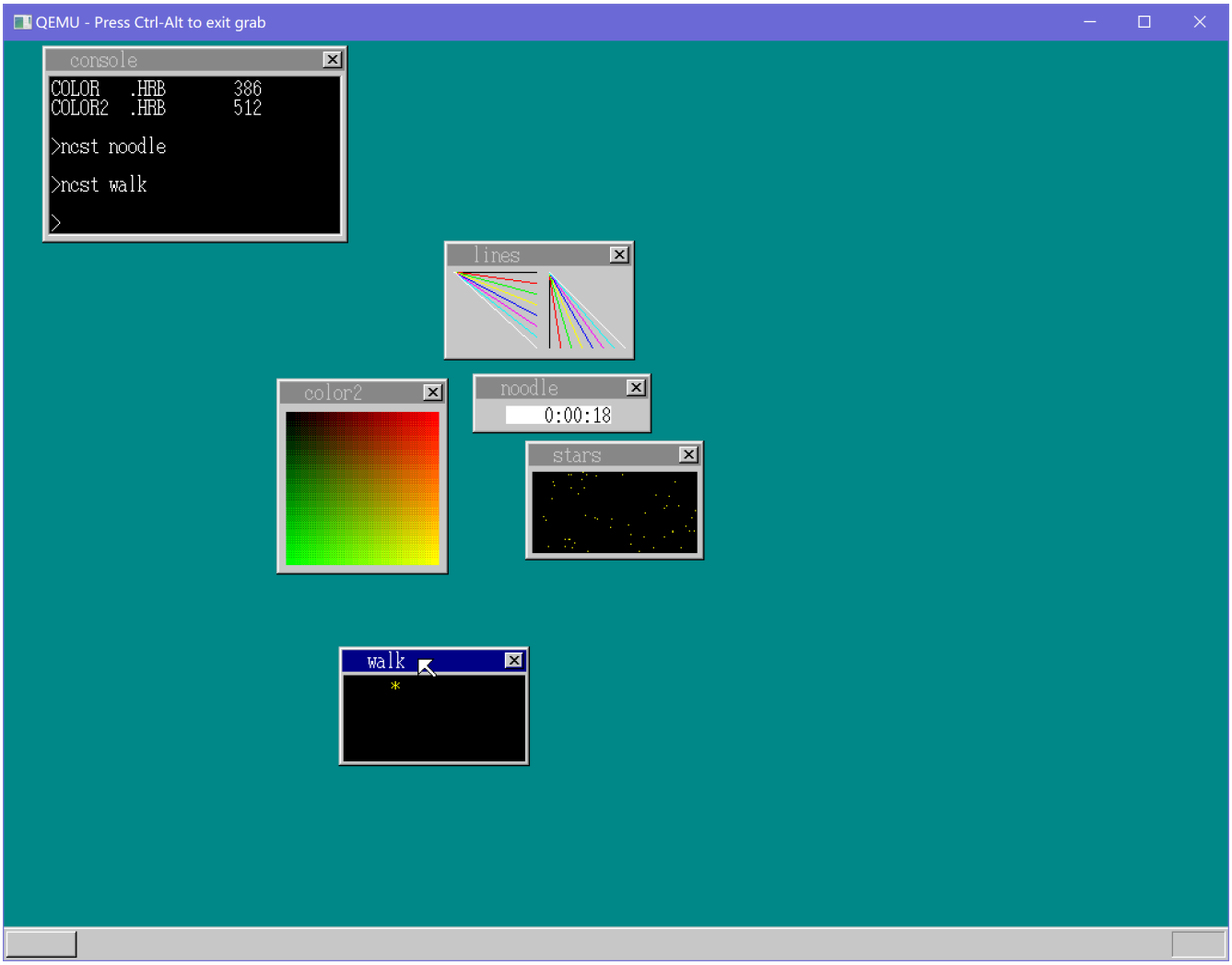
这样就可以利用作者提供的库管理器生成一个交`apilib.lib`的库

然后我们写一个头文件，囊括所有的API，然后写程序的时候只要include这个头文件就ok了，不用再进行单独的函数声明了。

然后整理makefile和我们的源码，拆分到不同的目录

make	像之前一样，生成一个包含操作系统内核及全部应用程序的磁盘映像
make run	“make”后启动QEMU
make install	“make”后将磁盘映像安装到软盘中
make full	将操作系统核心、apilib和应用程序全部make后生成磁盘映像
make run_full	“make full”后“make run”
make install_full	“make full”后“make install”
make run_os	将操作系统核心make后执行“make run”，当只对操作系统核心进行修改时可使用这个命令
make clean	本来clean命令是用于清除临时文件的，但由于在这个Makefile中并不生成临时文件，因此这个命令不执行任何操作
make src_only	将生成的磁盘映像删除以释放磁盘空间
make clean_full	对操作系统核心、apilib和应用程序全部执行“make clean”，这样将清除所有的临时文件
make src_only_full	对操作系统核心、apilib和应用程序全部执行“make src_only”， 这样将清除所有的临时文件和最终生成物。不过执行这个命令后，“make”和“make run”就无法使用了（用带full版本的命令代替即可），make时会消耗更多的时间
make refresh	“make full”后“make clean_full”。从执行过“make src_only_full”的状态执行这个命令的话，就会恢复到可以直接“make”和“make run”的状态

测试make run_full，正常编译，成功运行。拆分没有出差错

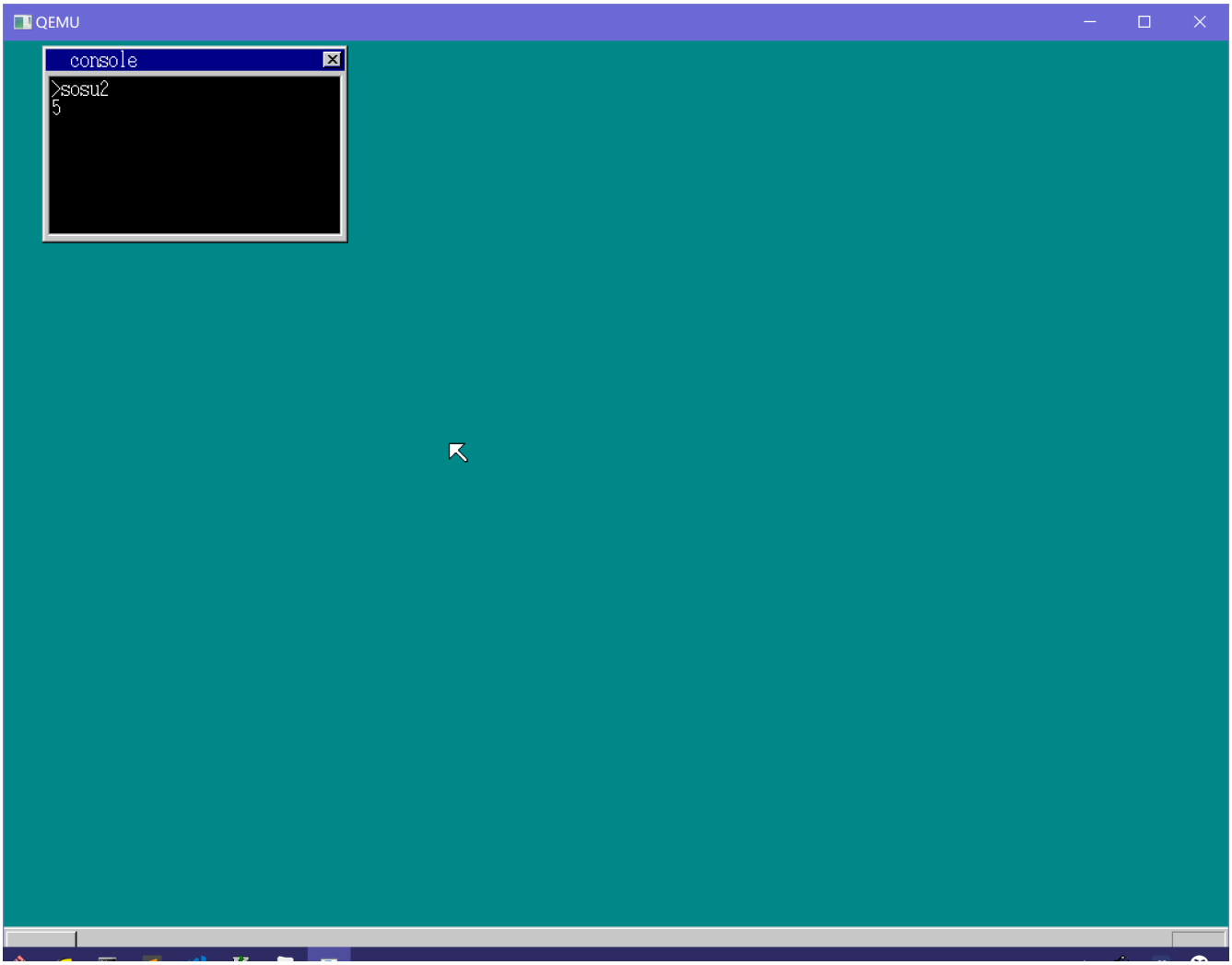

QEMU - Press Ctrl-Alt to exit grab
console
COLOR .HRB 386
COLOR2 .HRB 512
>ncst noodle
>ncst walk
>
lines
color2
noodle
0:00:18
stars
walk
*

Day 28

先来一个简单的程序

```
#include <stdio.h>
#include "apilib.h"
#define MAX 1000
void HariMain(void)
{
    char flag[MAX], s[8];
    int i, j;
    for (i = 0; i < MAX; i++) {
        flag[i] = 0;
    }
    for (i = 2; i < MAX; i++) {
        if (flag[i] == 0) {
            sprintf(s, "%d ", i);
            api_putstr0(s);
            for (j = i * 2; j < MAX; j += i) {
                flag[j] = 1;
            }
        }
    }
    api_end();
}
```

这段程序是求1000以内的素数的，当我们尝试扩大范围哦，将MAX改为10000后，程序无法正常的运行了。



输出5之后程序就不动了。

注意到编译的时候有一条警告：`Warning : can't link __alloca`

看来与他有关。为什么会这样呢？这是因为c语言编译器规定，如果栈中的变量超过4KB，就需要调用__alloca函数去获取栈中的空间

编写__alloca函数以供链接和调用

```
[FORMAT "WCOFF"]
[INSTRSET "i486p"]
[BITS 32]
[FILE "alloca.nas"]

GLOBAL __alloca

[SECTION .text]

__alloca:
        ADD EAX,-4
        SUB ESP,EAX
        JMP DWORD [ESP+EAX] ; 代替RET
```

那么什么时候__alloca会被调用呢？

- 要执行的操作从栈中分配EAX个字节的内存空间（ESP -= EAX;）
- 要遵守的规则不能改变ECX、EDX、EBX、EBP、ESI、EDI的值（可以临时改变它们的 值，但要使用PUSH/POP来复原）

以下__alloca是错误的

```
SUB ESP,EAX
RET
```

RET相当于POP EIP，ESP的值被改变了，于是EIP被装入了错误的返回地址

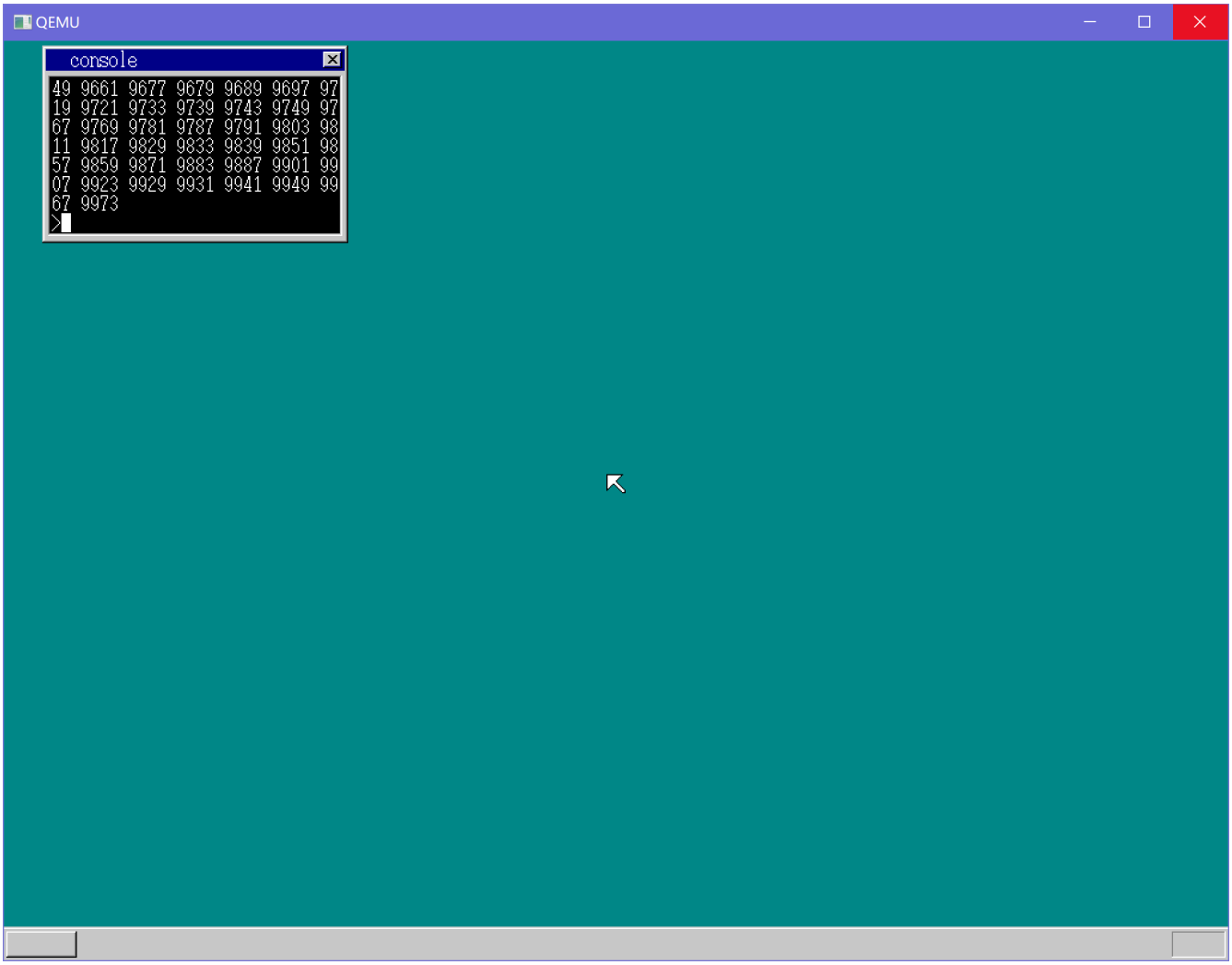
```
SUB ESP,EAX
JMP DWORD [ESP+EAX] ; 代替RET
```

POP EIP相当于`mov eip,[esp]`和`add esp,4` esp的值会出现误差

```
SUB ESP,EAX
JMP DWORD [ESP+EAX] ; 代替RET
ADD ESP,4
```

add又执行不到

```
SUB ESP,EAX
ADD ESP,4
JMP DWORD [ESP+EAX-4]
```



完成了__alloca之后，我们可以修改之前的winhelo和winhelo2了，将数组声明放到HariMain当中（之前这样做是会报错的）

接下来实现文件操作功能

打开	open
定位	seek
读取	read
写入	write
关闭	close

打开文件

EDX	21
EBX	文件名
EAX	文件句柄（为0时表示打开失败）（由操作系统返回）

关闭文件

EDX	22
EAX	文件句柄

文件定位

EDX	23
EAX	文件句柄
ECX	定位模式
	0：定位的起点为文件开头
	1：定位的起点为当前的访问位置
	2：定位的起点为文件末尾
EBX	定位偏移量

获取文件大小

EDX	24
EAX	文件句柄
ECX	文件大小获取模式
	0：普通文件大小
	1：当前读取位置从文件开头起算的偏移量
	2：当前读取位置从文件末尾起算的偏移量
EAX	文件大小（由操作系统返回）

文件读取

EDX	25
EAX	文件句柄
EBX	缓冲区地址
ECX	最大读取字节数
EAX	本次读取到的字节数（由操作系统返回）

定义数据类型

```
struct TASK {
    int sel, flags; /* sel为GDT编号*/
    int level, priority;
    struct FIFO32 fifo;
    struct TSS32 tss;
    struct SEGMENT_DESCRIPTOR ldt[2];
    struct CONSOLE *cons;
    int ds_base, cons_stack;
    struct FILEHANDLE *fhandle;
    int *fat;
};
struct FILEHANDLE {
    char *buf;
    int size;
    int pos;
};
```

在程序退出部分加入文件句柄释放部分

```
for (i = 0; i < 8; i++) {
    if (task->fhandle[i].buf != 0) {
        memman_free_4k(memman, (int) task->fhandle[i].buf, task->fhandle[i].size);
        task->fhandle[i].buf = 0;
    }
}
```

api

```
struct FILEHANDLE fhandle[8];
//.................
} else if (edx == 21) {
    for (i = 0; i < 8; i++) {
        if (task->fhandle[i].buf == 0) {
            break;
        }
```

```
    }
    fh = &task->fhandle[i];
    reg[7] = 0;
    if (i < 8) {
        finfo = file_search((char *) ebx + ds_base, (struct FILEINFO *) (ADR_DISKIMG +
0x002600), 224);
        if (finfo != 0) {
            reg[7] = (int) fh;
            fh->buf = (char *) memman_alloc_4k(memman, finfo->size);
            fh->size = finfo->size;
            fh->pos = 0;
            file_loadfile(finfo->clustno, finfo->size, fh->buf, task->fat, (char *)
(ADR_DISKIMG + 0x003e00));
        }
    }
} else if (edx == 22) {
    fh = (struct FILEHANDLE *) eax;
    memman_free_4k(memman, (int) fh->buf, fh->size);
    fh->buf = 0;
} else if (edx == 23) {
    fh = (struct FILEHANDLE *) eax;
    if (ecx == 0) {
        fh->pos = ebx;
    } else if (ecx == 1) {
        fh->pos += ebx;
    } else if (ecx == 2) {
        fh->pos = fh->size + ebx;
    }
    if (fh->pos < 0) {
        fh->pos = 0;
    }
    if (fh->pos > fh->size) {
        fh->pos = fh->size;
    }
} else if (edx == 24) {
    fh = (struct FILEHANDLE *) eax;
    if (ecx == 0) {
        reg[7] = fh->size;
    } else if (ecx == 1) {
        reg[7] = fh->pos;
    } else if (ecx == 2) {
        reg[7] = fh->pos - fh->size;
    }
} else if (edx == 25) {
    fh = (struct FILEHANDLE *) eax;
    for (i = 0; i < ecx; i++) {
        if (fh->pos == fh->size) {
            break;
        }
        *((char *) ebx + ds_base + i) = fh->buf[fh->pos];
        fh->pos++;
    }
    reg[7] = i;
```

```
}
```

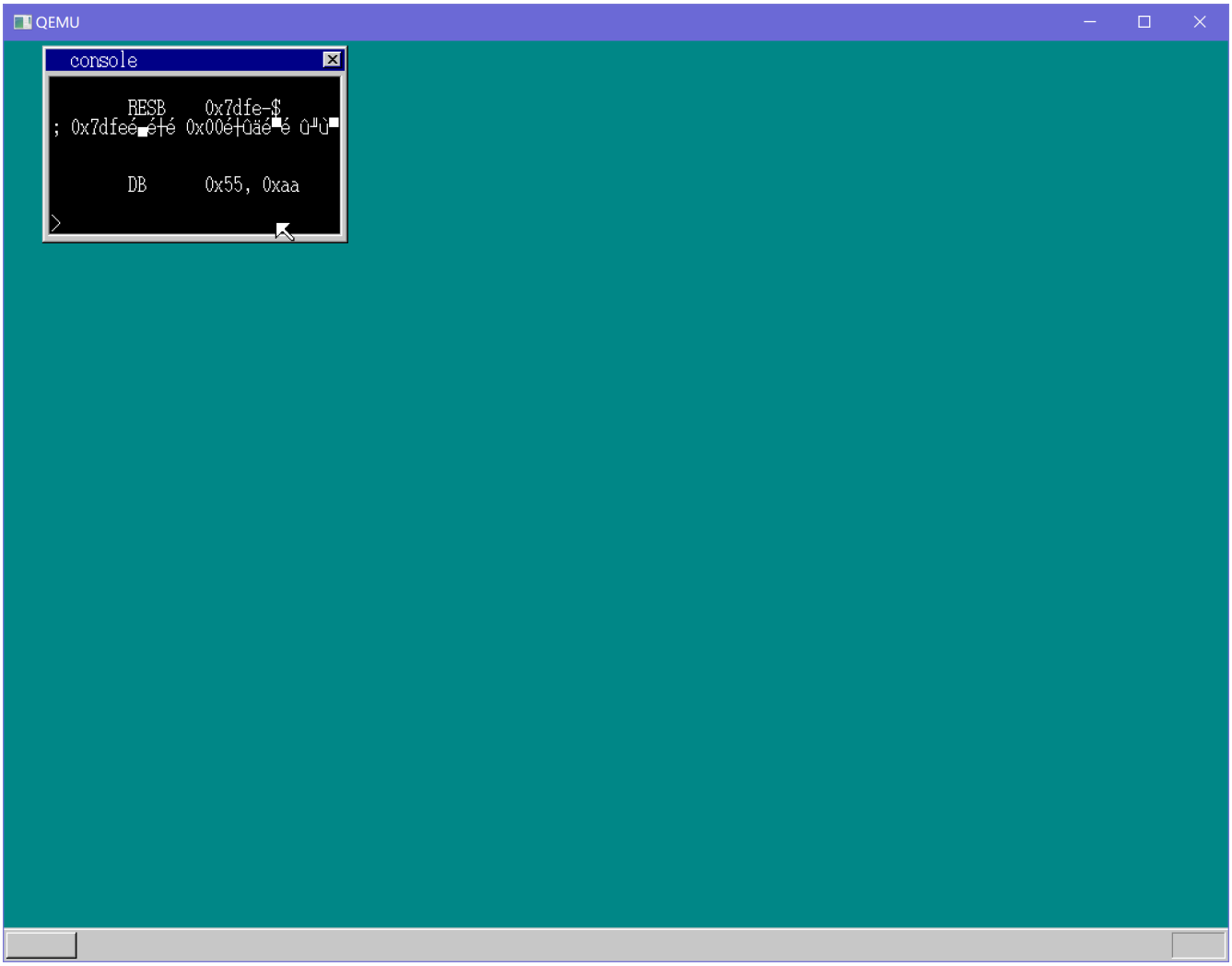
然后是apilib（就合一起写了）

```
_api_fopen: ; int api_fopen(char *fname);
        PUSH EBX
        MOV EDX,21
        MOV EBX,[ESP+8] ; fname
        INT 0x40
        POP EBX
        RET
_api_fclose: ; void api_fclose(int fhandle);
        MOV EDX,22
        MOV EAX,[ESP+4] ; fhandle
        INT 0x40
        RET
_api_fseek: ; void api_fseek(int fhandle, int offset, int mode);
        PUSH EBX
        MOV EDX,23
        MOV EAX,[ESP+8] ; fhandle
        MOV ECX,[ESP+16] ; mode
        MOV EBX,[ESP+12] ; offset
        INT 0x40
        POP EBX
        RET
_api_fsize: ; int api_fsize(int fhandle, int mode);
        MOV EDX,24
        MOV EAX,[ESP+4] ; fhandle
        MOV ECX,[ESP+8] ; mode
        INT 0x40
        RET
_api_fread: ; int api_fread(char *buf, int maxsize, int fhandle);
        PUSH EBX
        MOV EDX,25
        MOV EAX,[ESP+16] ; fhandle
        MOV ECX,[ESP+12] ; maxsize
        MOV EBX,[ESP+8] ; buf
        INT 0x40
        POP EBX
        RET
```

编写测试程序

```
/*typeipl.c*/
#include "apilib.h"
void HariMain(void)
{
    int fh;
    char c;
    fh = api_fopen("ipl10.nas");
    if (fh != 0) {
        for (;;) {
```

```
            if (api_fread(&c, 1, fh) == 0) {
                break;
            }
            api_putchar(c);
        }
    }
    api_end();
}
```



为了实现能够用应用程序打印任意文件的内容，我们还要实现读取命令行参数的功能

api设计

EDX	26
EBX	存放命令行内容的地址
ECX	最多可存放多少字节
EAX	实际存放了多少字节（由操作系统返回）

在TASK结构体中添加一个指针，cmdline，保存命令行内容

```
void console_task(struct SHEET *sheet, int memtotal)
{
    //..............
    task->cons = &cons;
    task->cmdline = cmdline;
    //..............
}
```

```
int *hrb_api(int edi, int esi, int ebp, int esp, int ebx, int edx, int ecx, int eax)
{
    //..............
    } else if (edx == 26) {
        i = 0;
        for (;;) {
            *((char *) ebx + ds_base + i) = task->cmdline[i];
            if (task->cmdline[i] == 0) {
                break;
            }
            if (i >= ecx) {
                break;
            }
            i++;
        }
        reg[7] = i;
    }
    return 0;
}
```

```
_api_cmdline: ; int api_cmdline(char *buf, int maxsize);
        PUSH EBX
        MOV EDX,26
        MOV ECX,[ESP+12] ; maxsize
        MOV EBX,[ESP+8] ; buf
        INT 0x40
        POP EBX
        RET
```

然后我们只要对cmdline进行处理，就可以获取参数了

```
#include "apilib.h"
void HariMain(void)
{
    int fh;
    char c, cmdline[30], *p;
    api_cmdline(cmdline, 30);
    for (p = cmdline; *p > ' '; p++) { }
    for (; *p == ' '; p++) { } /*跳过文件名和空格*/
    fh = api_fopen(p);
    if (fh != 0) {
```

```
        for (;;) {
            if (api_fread(&c, 1, fh) == 0) {
                break;
            }
            api_putchar(c);
        }
    } else {
        api_putstr0("File not found.\n");
    }
    api_end();
}
```

接下来引入全角字符支持。gb2312编码按区位的方式对字符进行索引。位是最小单位，然后是区

在GB2312中，字符编码的分类如下：

- 01区～09区：非汉字
- 10区～15区：空白
- 16区～55区：一级汉字
- 56区～87区：二级汉字
- 88区～94区：空白

接下来我们了解一下字库：

GB2312字库文件是HZK16，网搜下载一个。

HZK16字库是符合GB2312标准的16×16点阵字库,HZK16的GB2312-80支持的汉字有6763个，符号682个。其中一级汉字有3755个，按声序排列，二级汉字有3008个，按偏旁部首排列。

我们将压缩后的字库存入img中，然后再在系统启动后读取解压字库

```
nihongo = (unsigned char *) memman_alloc_4k(memman, 0x5d5d*32);
```

```
finfo = file_search("HZK16.fnt", (struct FILEINFO *) (ADR_DISKIMG + 0x002600), 224);
```

把汉字的langmode定为3.

注意HZK编码不同于sjis的地方在于，HZK是分上下而非左右。

修改graphic.c中的putfonts8_asc

```
if (task->langmode == 3) {
    for (; *s != 0x00; s++) {
        if (task->langbyte1 == 0) {
            if (0xa1 <= *s && *s <= 0xfe) {
                task->langbyte1 = *s;
            } else {
```

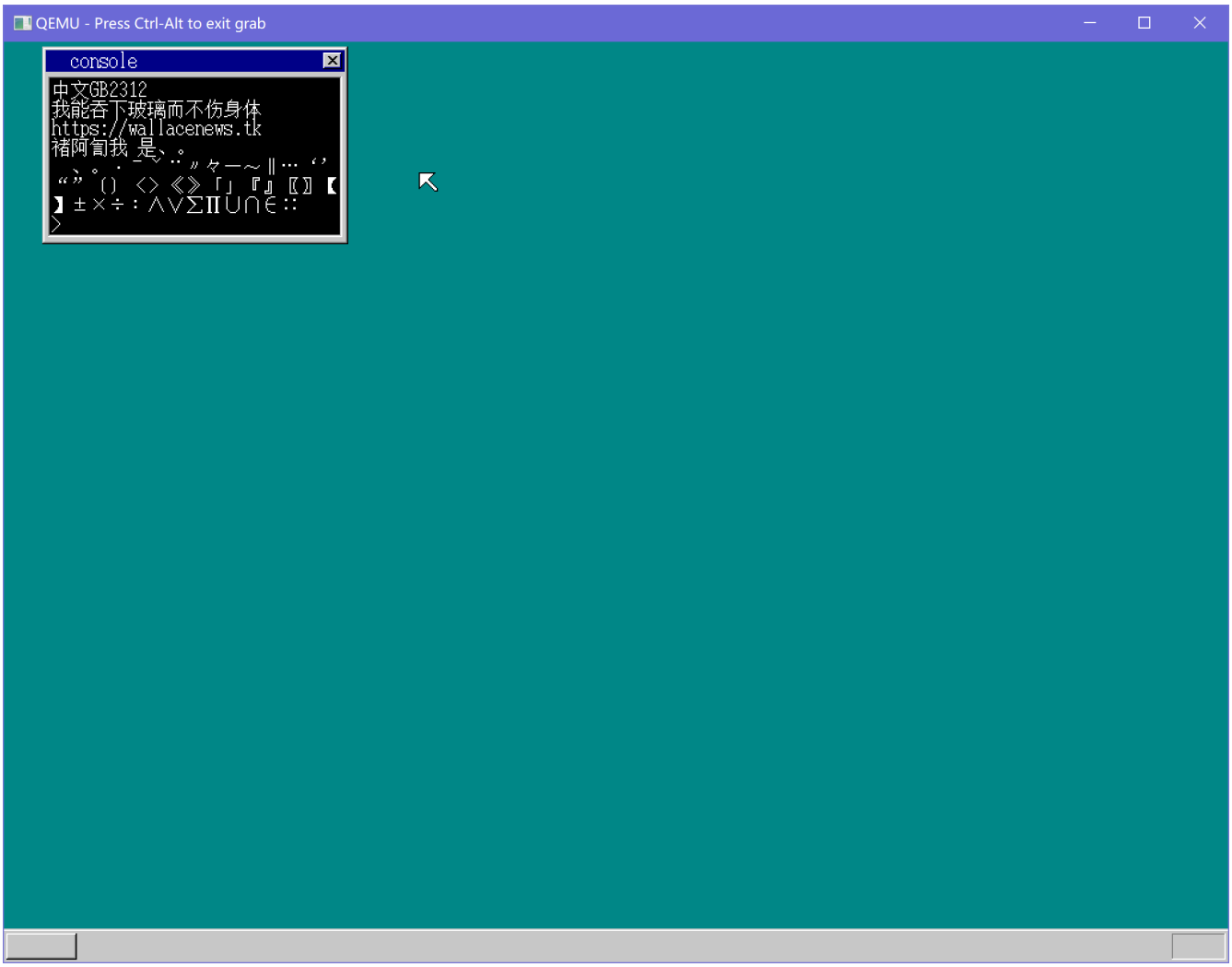
```
                putfont8(vram, xsize, x, y, c, hankaku + *s * 16);//只要是半角就使用hankaku里面的字符
            }
        } else {
            k = task->langbyte1 - 0xa1;
            t = *s - 0xa1;
            task->langbyte1 = 0;
            font = nihongo + (k * 94 + t) * 32;
            putfont32(vram,xsize,x-8,y,c,font,font+16);
        }
        x += 8;
    }
}
```

```
/*chklang.c*/
#include "apilib.h"

void HariMain(void)
{
    int langmode = api_getlang();
    static char s1[23] = {
        0x93, 0xfa, 0x96, 0x7b, 0x8c, 0xea, 0x83, 0x56, 0x83, 0x74, 0x83, 0x67,
        0x4a, 0x49, 0x53, 0x83, 0x82, 0x81, 0x5b, 0x83, 0x68, 0x0a, 0x00
    };
    static char s2[17] = {
        0xc6, 0xfc, 0xcb, 0xdc, 0xb8, 0xec, 0x45, 0x55, 0x43, 0xa5, 0xe2, 0xa1,
        0xbc, 0xa5, 0xc9, 0x0a, 0x00
    };
    static char s3[20] = {
        0xce, 0xd2, 0x20, 0xca, 0xc7, 0xa1, 0xa2, 0xa1, 0xa3, 0x0a, 0x00
    };
    int i;char j;
    if (langmode == 0) {
        api_putstr0("English ASCII mode\n");
    }
    if (langmode == 1) {
        api_putstr0(s1);
    }
    if (langmode == 2) {
        api_putstr0(s2);
    }
    if (langmode == 3) {
        api_putstr0("中文GB2312\n我能吞下玻璃而不伤身体\n");
        api_putstr0("https://wallacenews.tk\n");
        api_putstr0("褚阿訇");
        api_putstr0(s3);
        for(i=0xa1;i<0xcc;i++)
        {
            s3[0]=0xa1;
            j=i;
            s3[1]=j;
```

```
            s3[2]=0x00;
            api_putstr0(s3);
        }
    }
    api_end();
}
```

试一下吧



大功告成！我们支持中文了！

Day 29

今天我们为操作系统引入原生的压缩文件支持，使用作者自己鼓捣出来的tek格式。

edimg中使用的压缩格式就是tek。edimg的源码在autodec_.c中，我们直接C-cv相应的代码

我们整理一下解压缩函数

```
int tek_getsize(unsigned char *p)
{
    static char header[15] = {
        0xff, 0xff, 0xff, 0x01, 0x00, 0x00, 0x00, 0x4f, 0x53, 0x41, 0x53, 0x4b, 0x43, 0x4d,
0x50
    };
    int size = -1;
    if (memcmp(p + 1, header, 15) == 0 && (*p == 0x83 || *p == 0x85 || *p == 0x89)) {
        p += 16;
        size = tek_getnum_s7s(&p);
    }
    return size;
}
int tek_decomp(unsigned char *p, char *q, int size)
{
    int err = -1;
    if (*p == 0x83) {
        err = tek_decode1(size, p, q);
    } else if (*p == 0x85) {
        err = tek_decode2(size, p, q);
    } else if (*p == 0x89) {
        err = tek_decode5(size, p, q);
    }
    if (err != 0) {
        return -1;
    }
    return 0;
}
```

我们把file_loadfile包装一下，以实现自动解压。

```
char *file_loadfile2(int clustno, int *psize, int *fat)
{
    int size = *psize, size2;
    struct MEMMAN *memman = (struct MEMMAN *) MEMMAN_ADDR;
    char *buf, *buf2;
    buf = (char *) memman_alloc_4k(memman, size);
    file_loadfile(clustno, size, buf, fat, (char *) (ADR_DISKIMG + 0x003e00));
    if (size >= 17) {
        size2 = tek_getsize(buf);
        if (size2 > 0) {
```

```
            buf2 = (char *) memman_alloc_4k(memman, size2);
            tek_decomp(buf, buf2, size2);
            memman_free_4k(memman, (int) buf, size);
            buf = buf2;
            *psize = size2;
        }
    }
    return buf;
}
```

psize用来返回解压缩后的文件大小。

实现了自动压缩功能，我们就可以把我们的文件压缩之后再放入img中了

```
bim2bin -osacmp in:nihongo.org out:nihongo.fnt -tek2
```

其他文件同理。

为了实现原生支持，我们要把系统中所有file_loadfile的调用改成file_loadfile2

- cmd_app
- hrb_api

实现标准函数。c语言中提供了许多标准函数，例如

- printf
- putchar
- strcmp
- malloc

等

```
#include "apilib.h"
int putchar(int c)
{
    api_putchar(c);
    return c;
}
```

```
#include "apilib.h"
void exit(int status)
{
    api_end();
}
```

```

```

```c
#include <stdio.h>
#include <stdarg.h>
#include "apilib.h"
int printf(char *format, ...)
{
    va_list ap;
    char s[1000];
    int i;
    va_start(ap, format);
    i = vsprintf(s, format, ap);
    api_putstr0(s);
    va_end(ap);
    return i;
}
```

```c
void *malloc(int size)
{
    char *p = api_malloc(size + 16);
    if (p != 0) {
        *((int *) p) = size;
        p += 16;
    }
    return p;
}
```

```c
void free(void *p)
{
    char *q = p;
    int size;
    if (q != 0) {
        q -= 16;
        size = *((int *) q);
        api_free(q, size + 16);
    }
    return;
}
```

这个16是干什么的呢？为了实现只提供一个指针就可以free，我们把大小存在分配的开头的4个字节中，返回的指针是从第17个字节开始的。

于是释放的时候只要取p之前的16个字节的前4个字节，就可以得知要释放的内存到底有多大，然后就可以释放了。

使用16而不是4的原因是内存地址为16字节的倍数时，CPU的处理速度有时候可以更快

实现非矩形窗口，由于我们之前就设计了透明色机制，我们直接使用透明色就可以实现非矩形窗口了。

测试更多应用程序

```
/*bball.c*/
#include "apilib.h"
void HariMain(void)
{
    int win, i, j, dis;
    char buf[216 * 237];
    struct POINT {
        int x, y;
    };
    static struct POINT table[16] = {
        { 204, 129 }, { 195, 90 }, { 172, 58 }, { 137, 38 }, { 98, 34 },
        { 61, 46 }, { 31, 73 }, { 15, 110 }, { 15, 148 }, { 31, 185 },
        { 61, 212 }, { 98, 224 }, { 137, 220 }, { 172, 200 }, { 195, 168 },
        { 204, 129 }
    };
    win = api_openwin(buf, 216, 237, -1, "bball");
    api_boxfilwin(win, 8, 29, 207, 228, 0);
    for (i = 0; i <= 14; i++) {
        for (j = i + 1; j <= 15; j++) {
            dis = j - i; /*两点间的距离*/
            if (dis >= 8) {
                dis = 15 - dis; /*逆向计数*/
            }
            if (dis != 0) {
                api_linewin(win, table[i].x, table[i].y, table[j].x, table[j].y, 8 - dis);
            }
        }
    }
    for (;;) {
        if (api_getkey(1) == 0x0a) {
            break; /*按下回车键则break; */
        }
    }
    api_end();
}
```

运行发现一些线条显示不正常，是refresh的bug导致的，我们修改一下划线的api，swap以下大小参数

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

```
                edi = i;
            }
            sheet_refresh(sht, eax, ecx, esi + 1, edi + 1);
        }
    }
```

还有外星人游戏，这一段与操作系统相关性不大，就跳过去了