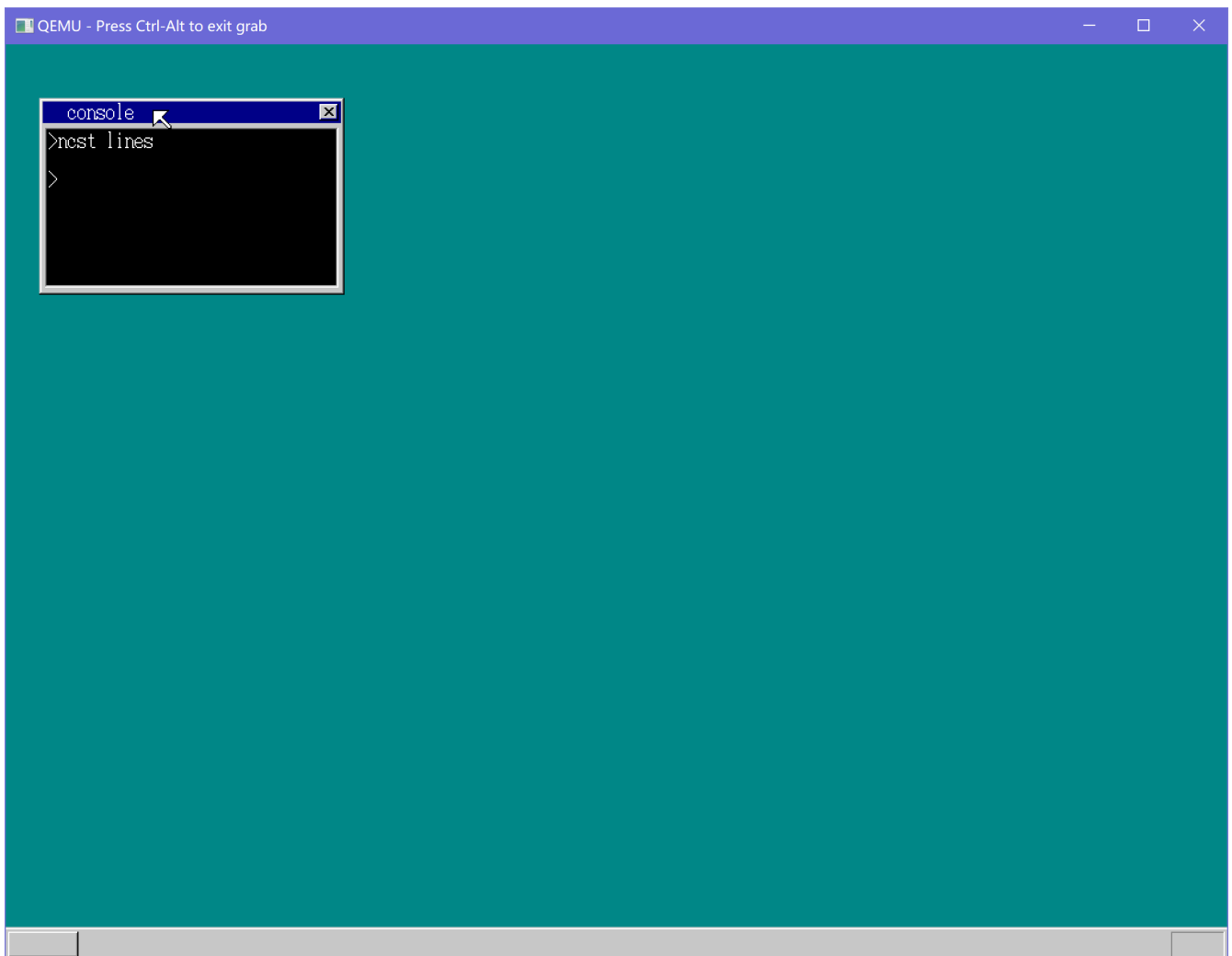


Day 27

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大问题，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

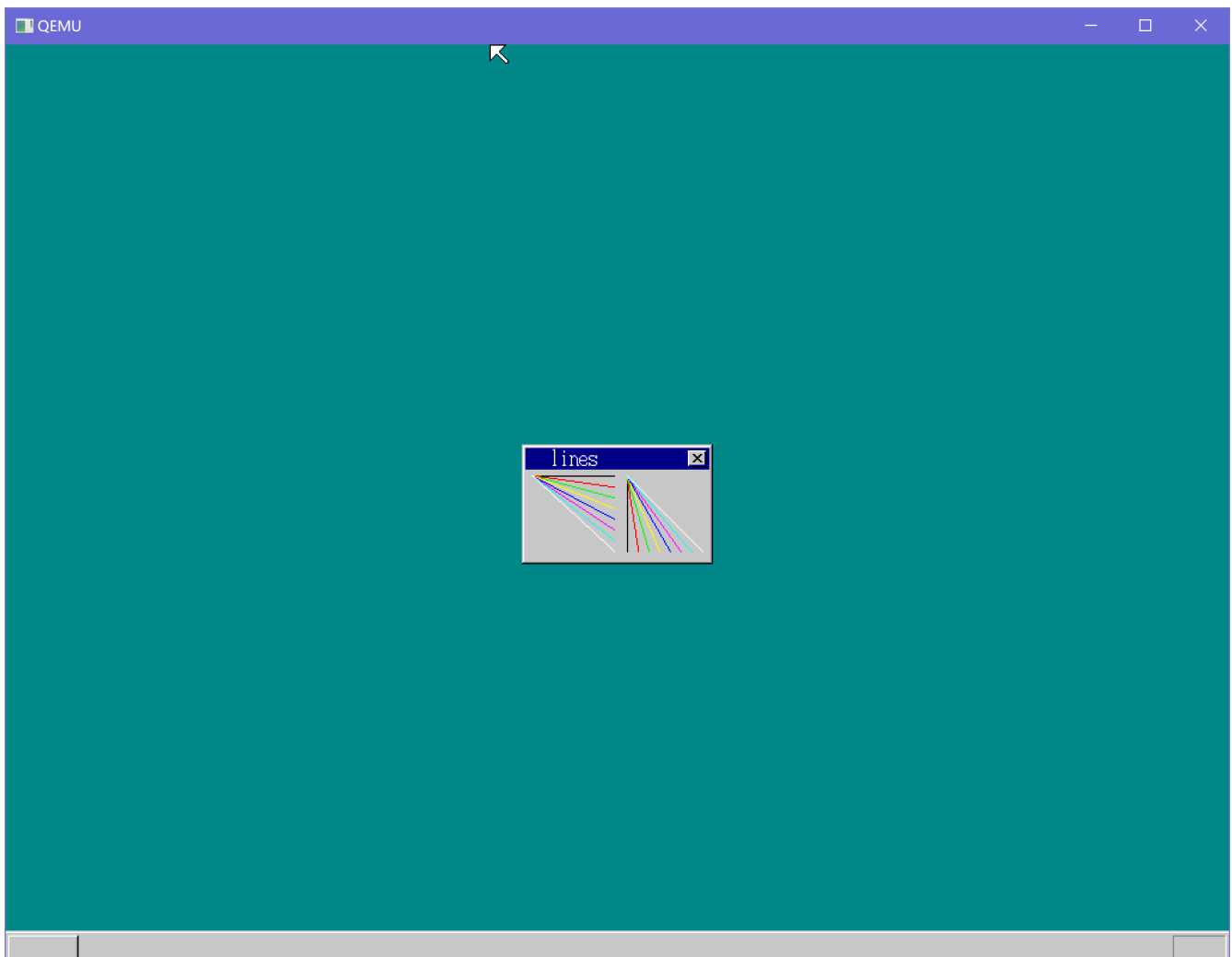
```
if (sht->bysize - 21 <= x && x < sht->bysize - 5 && 5 <=
```

```

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。

```

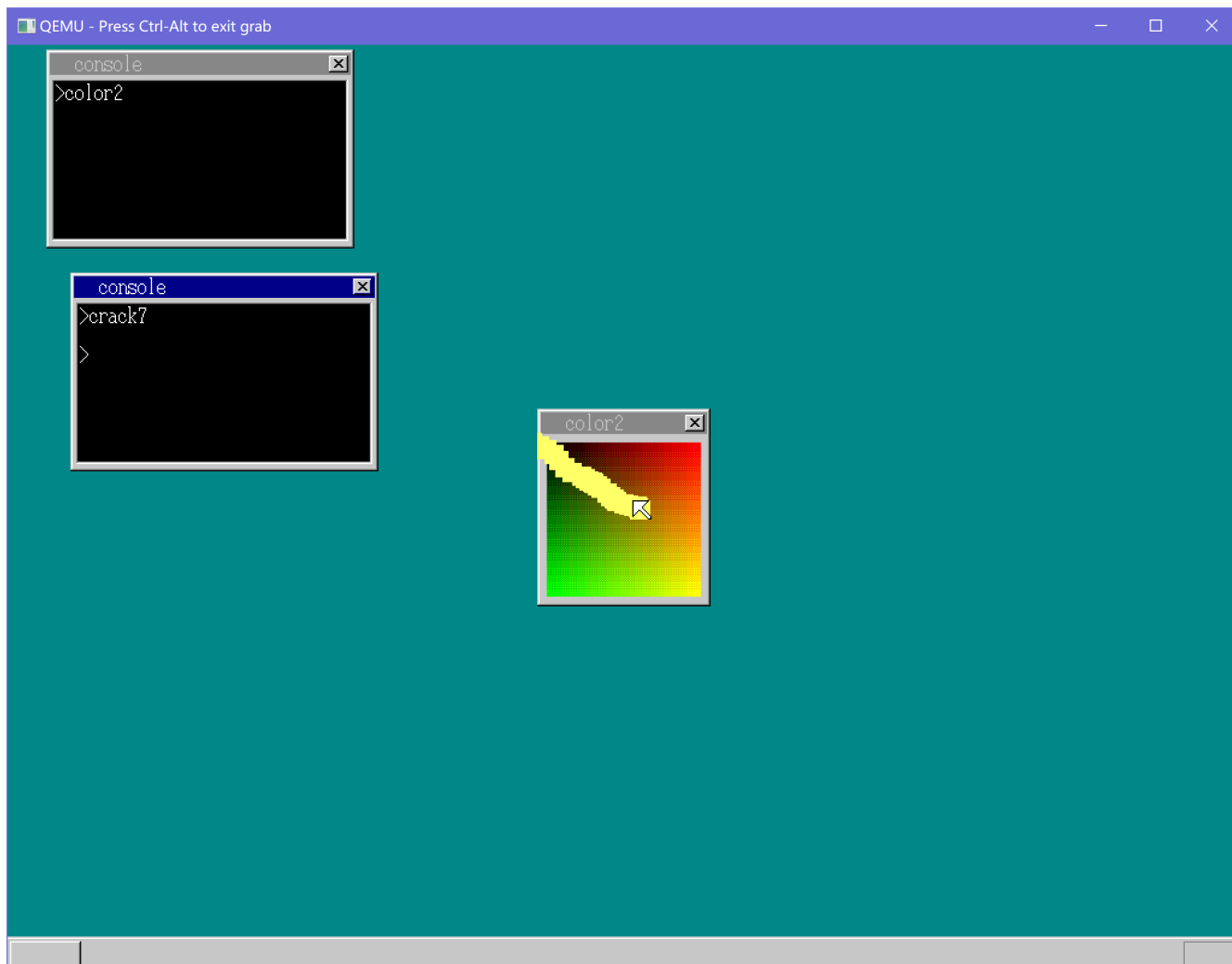
[FORMAT "WCOFF"]
[INSTRSET "i486p"]
[BITS 32]
[FILE "crack7.nas"]

    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）

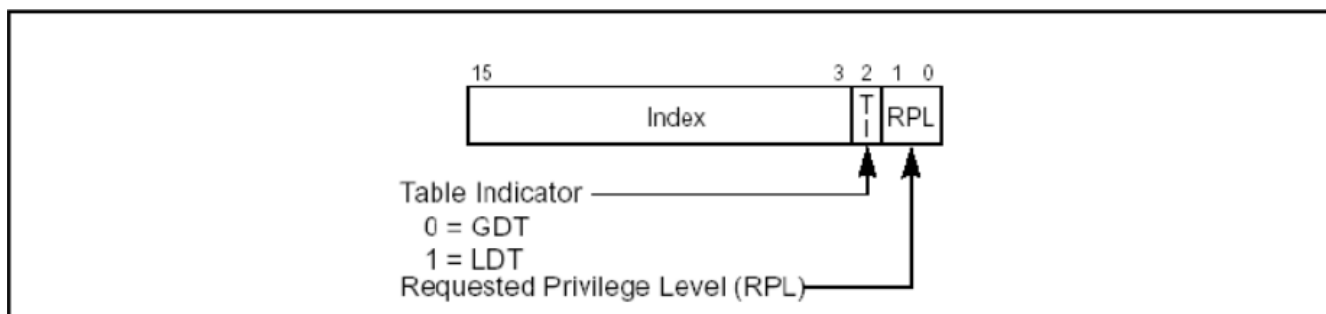
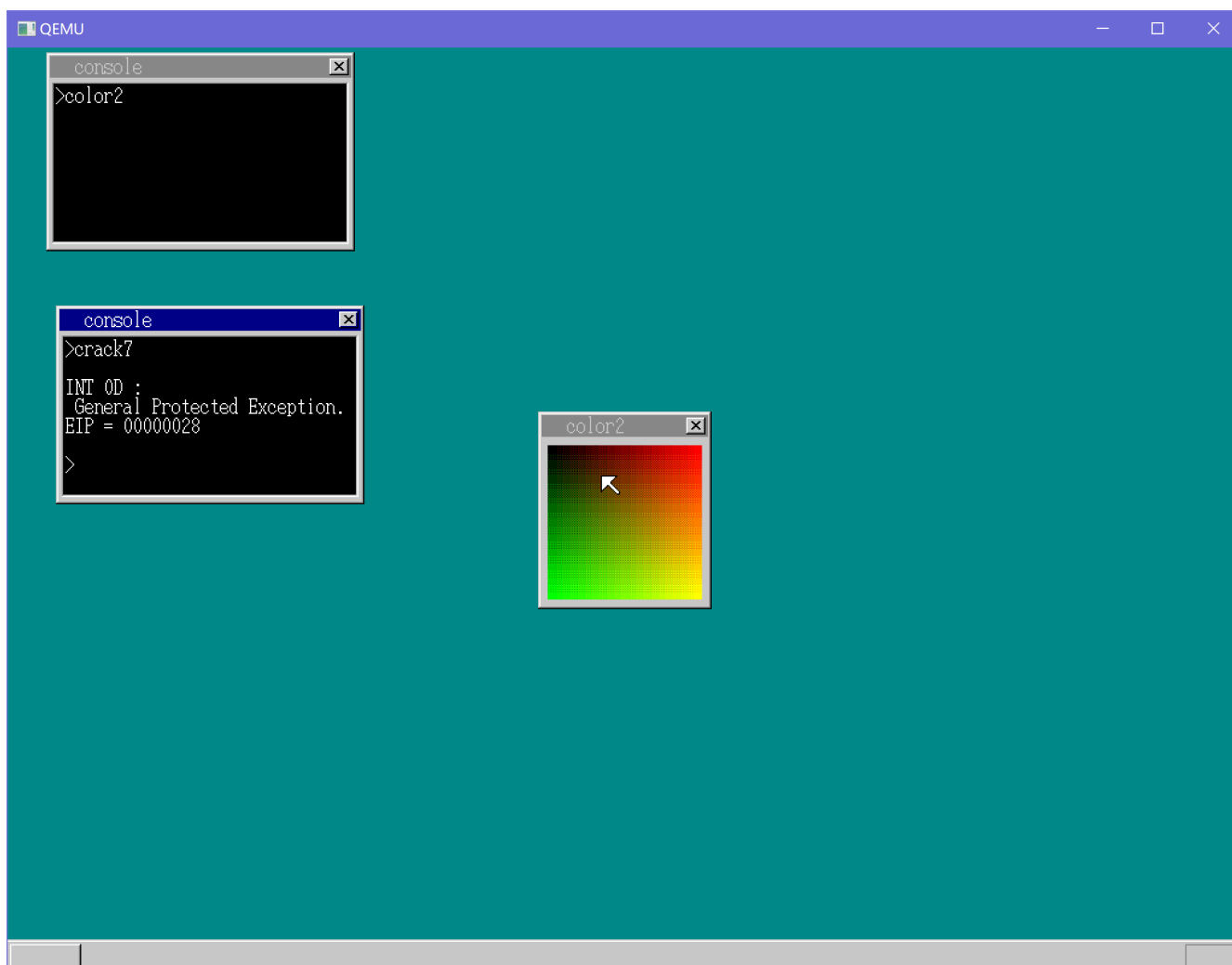


Figure 3-6. Segment Selector

```
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lidt来重设ldtr到别的程序，但lidt是系统专用指令，所以这个操作是会被CPU拦截下来的。

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)...

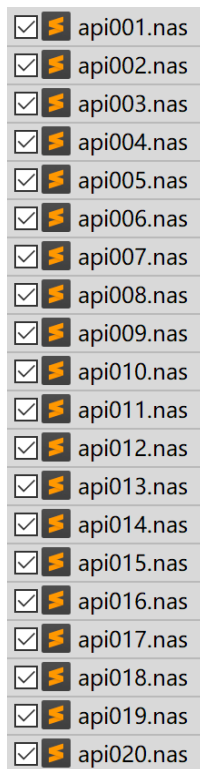
确定

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

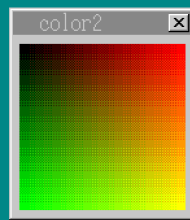
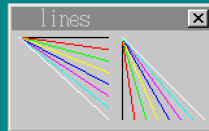
然后我们写一个头文件，囊括所有的API，然后写程序的时候只要include这个头文件就ok了，不用再进行单独的函数声明了。

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

```
console
COLOR .HEB 386
COLOR2 .HEB 512
>nest noodle
>nest walk
>
```



```
noodle
0:00:18
```

