不依赖于操作系统的游戏

1. 实验内容的简述。

本次实验完成的是一个飞机躲避子弹的游戏,不依赖于操作系统和运行库，直接操纵显存与帧缓冲绘制图形，使用键盘和时钟中断响应异步事件来实现一个游戏。

2.实验目标。

编写一个飞机躲避子弹的游戏，且游戏是即时的，即通过中断异步地响应键盘和时钟中断。并且给游戏增加一些额外的功能，比如 restart，计时，还有放大招是屏幕子弹消失等功能。

3.完成实验涉及的额外准备知识。

对熟悉掌握基本的终端命令，对 vim 编辑器的使用有一定的熟悉和掌握，同时对游戏部分的代码，键盘响应等要很清楚其中各函数的定义以及功能，同时尽量学会用 git 管理自己的代码，对 linux 下的 c 语言编程有一定的了解。

4.实验设计。

首先要知道，自己要写的游戏有两个模型，一个是飞机，一个是子弹，第一步先实现子弹向中间飞的部分，因为打字游戏中，已经有了很多字母下落的过程的实现，所以修改一下链表中每一个字母节点的 text，并对照 8*8 矩阵显示字母的表，这样就可以成功的使下落的字母显示成一个点 (.) 来代替子弹。这样就完成了一个方向向下的子弹的飞行。然后给节点加一个 position 的 int 型数，用来判断子弹在屏幕上是上下飞还是左右飞，然后在重新初始化一下子弹的初始位置和速度方向，这样每一个子弹就有了四种飞行的可能，在设定一个随机数，让它对四取模，这样每次子弹进入链表中时就可以确定它的速度放向和初始位置，以及其运行轨迹，所以子弹的部分得以实现。

然后就定义飞机的模型，开始定义了一个坐标点（plane_x，plane_y）每次都是对这个坐标进行操作，然后再在这个坐标点上画出完整的飞机。飞机用 8*8 的矩阵画出。建立好飞机的模型之后，就讲键盘相应加入进去，分别是键盘的上下左右来控制飞机，实际上是每一次按上下左右键，得到键盘响应后，都相应的改变 plane_x，plane_y 的数值，也就是飞机的位置，让飞机动起来。

接下来就是判断 game over。这个就需要写一个函数，判断飞机是否碰到子弹，也就是把链表中的所有子弹都检测一遍，每次画面的更新都需要检测，这样就可以判断 game over 了。

然后就是几个额外的功能，放大招就是检测键盘相应，如果按到空格键，并且还有大招的机会，就使所有的子弹都消失。接下来就是 restart 的实现，每次 game over 后，如果有‘R’的键盘响应，则游戏重新开始。

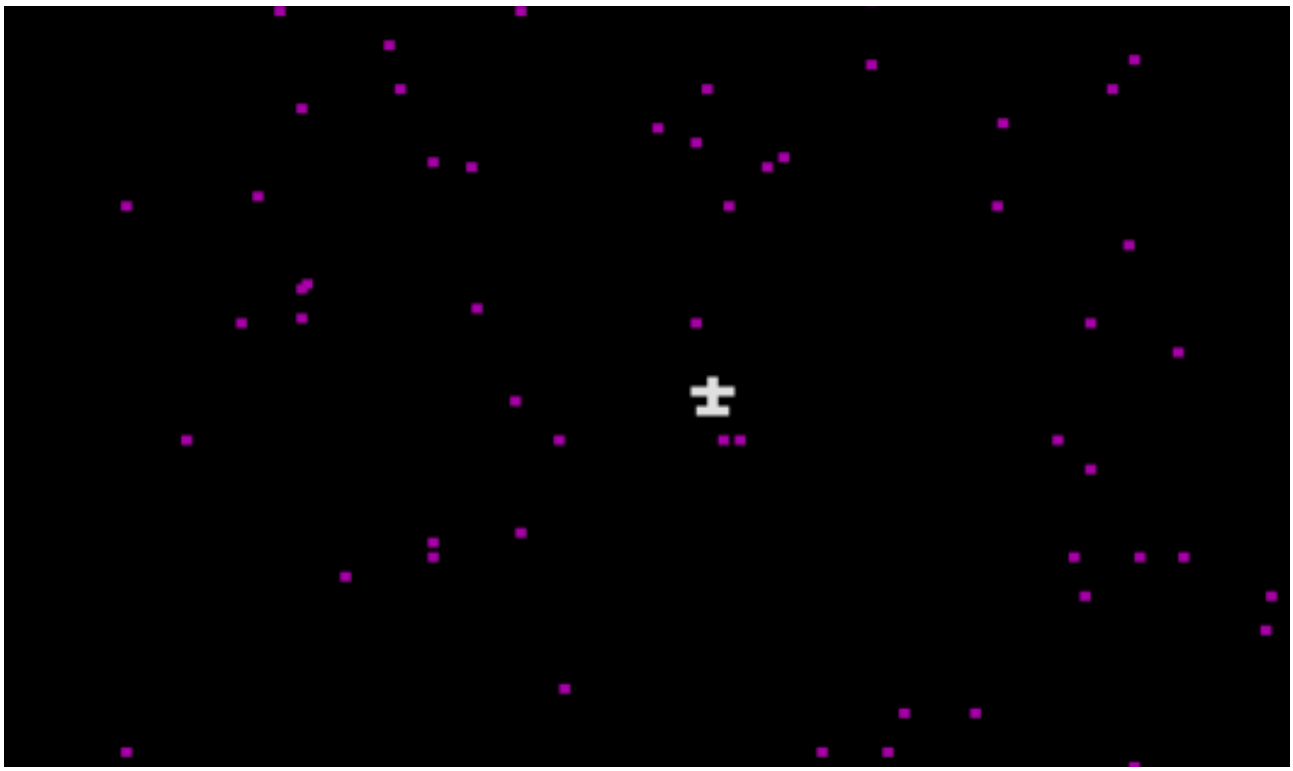
游戏运行时，左上角显示时间，左下角显示剩余大招次数。

5.具体实现。

子弹部分的实现，把所有屏幕上的子弹加入到创建的链表中去。

```
if (head == NULL) {
        head = fly_new(); /* 当前没有任何子弹，创建新链表 */
} else {
        fly_t now = fly_new();
        fly_insert(NULL, head, now); /* 插入到链表的头部 */
        head = now;
}
```

然后再通过随机数使子弹从不同方向出来向中间飞行。



然后是实现飞机，飞机初始化位置是

```
static int plane_x= SCR_HEIGHT/2 , plane_y=SCR_WIDTH/2;
```

通过键盘响应确定飞机下一步的位置。接下来就是判断 game over，只是检测所有链表中的子弹坐标是否在飞机所在的范围内。

```
                fly_t temp = head;
                for(;temp!=NULL;temp=temp->_next)
                {
                        if(((temp->x - plane_x<=7 && temp->y - plane_y >=3 && temp->x - plane_x>=0 && temp->y - plane_y <=4) || (temp->x - plane_x >=2 && temp->x - plane_x <=3 && temp->y - plane_y >= 0 && temp->y - plane_y<=7) || (temp->x - plane_x >=6 && temp->x - plane_x <= 7 && temp->y - plane_y >= 1 && temp->y - plane_y <= 6)) && temp->text==0)
                                return 0;
                }
                return 1;
```

游戏结束后，屏幕显示 game over，为了实现 restart 并且不让游戏中断，所以游戏结束后使其进入死循环，只有得到’R‘键盘响应后才可以初始化屏幕链表等并且跳出循环。

初始化

```
void restart(void){
                clear_screen(SCR_HEIGHT,SCR_WIDTH);
                clear_time();
                chance=1;
                fly_t temp1 = head;
                for(;temp1!=NULL;)
                {
                        fly_t temp2=temp1;
                        temp1=temp1->_next;
                        fly_remove(temp2);
                        fly_free(temp2);
                }
                head=NULL;
                plane_x= SCR_HEIGHT/2 , plane_y=SCR_WIDTH/2;
}               //初始化屏幕，置空链表更新飞机位置，大招次数
```

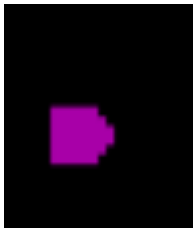
'R'控制 restart

```
}
bool
is_restart(void){

                int temp =last_key_code();
                 if(temp==19){
                        return TRUE;
                }
                        return FALSE;
}           //判断是否重新开始，按‘R’重新开始
```

```
 game_over();          //显示游戏结束
 while(1){             //这个死循环是使屏
进行游戏的目的
          if(is_restart()){
                  restart();
                  break;
                  }
          }
```

放大招，初始次数设定为 chance=1，如果飞机碰到炮弹，chance++
炮弹设定其 text 为 1，然后用 8*8 矩阵画出炮弹。



6.遇到的困难与解决方法。

遇到的困难：

1）开始子弹不会从下方出来，而且子弹从四面出来后碰到左边的边框和飞机碰到四边都会蓝屏。

2）游戏结束后直接蓝屏

3）放大招后子弹全部消失后，游戏运行出错。

4）restart 不能实现，游戏结束后，没有按‘R’，游戏自动重新开始。

5）游戏结束后 restart 后。屏幕显示的 game over 并不消失。

解决的办法：

1）将从下方向上飞行的子弹的初始坐标由 SCR_HEIGHT 改为 SCR_HEIGHT－8，因为每个子弹都是由 8*8 矩阵表示的，所以要减去 8。从右边向左边飞也是一样。对于蓝屏问题，也是修改坐标，对于子弹，当它的坐标已经超出屏幕表示范围时就要及时的把该节点删掉，并释放空间（调用 remove（），free（）函数）。对于飞机超出范围时，首先要对飞机的位置进行检测，如果它也超出范围，则让飞机回到上一帧的位置。然后再画出飞机，这样我们看到的效果就是，当飞机在最右边时，再按一下右，则飞机不会动。

2）因为游戏结束后就跳出了游戏循环的 main_loop（），到 assert（0），而产生蓝屏。所以就让游戏运行在一个大的死循环中，而且每次判断游戏结束后都会进入到一个小的死循环，在小死循环中，重复判断 if 条件，只有键盘按到'R'时才会跳出循环。进入到大的死循环中，重新进行游戏，这样就不会蓝屏。

3）这是一个小错误，当删除链表后，删除所有节点，释放空间后，令 head=NULL；则问题解决。之前忘记将 head 为空了。

4）将检测‘R’的键盘相应由 if(query_key(5)){ release_key(5);

return TRUE; }　　enable_interrupt(); return FALSE;　改为 int temp = last_key_code(); if(temp==19){ return TRUE;} return FALSE;　　则问题解决。

5）由于游戏 restart 后，game over 等字样还在屏幕中，所以就在初始化子弹链表等的同时，在显示 game over 范围内，均用黑色像素覆盖掉，用双层的 for 循环即可实现。

7.总结。

开始的时候觉得本次实验无从下手，文件很多不知道该怎么改，像这种情况可以将自己能看懂的地方做细小的改动，比如开始 text 是模 26 的余数，若把它改成 0，则下来的都变成 A，改成其他的右变成别的符号，然后看到 font.c 文件知道字母都是在这里用矩阵表示的，就可以尝试着自己画个图像显示出来。再比如，字母起始位置，把 x 该一下，字母在屏幕上随机掉落，改一下速度正负号，飞行方向变化，然后再一点一点修改，就发现里预期效果越来越近。所以面对一个很大的任务时，要一点一点攻克，慢慢体会，也就大概理解其中的意思，然后一点点渗透。还有就是要选择一个好的工具，这对实验的完成是很有帮助的，比如安装 Nerdtree 插件，对于 vim 编辑器的使用就会相对方便一点。在就是细心和思维，逻辑上的错误有时是很难想出来的。由于之前写游戏时还不会使用 git，所以只在写完游戏的代码时，最后提交了一次，对于 git 只了解一下初步的知识，还要继续加强学习。

8.感谢汤顺雷对游戏逻辑的启发。

参考文献

http://coolshell.cn/articles/5426.html

http://marklodato.github.com/visual-git-guide/index-zh-cn.html

http://blog.csdn.net/gemmem/article/details/7290125

http://rogerdudler.github.com/git-guide/index.zh.html

1. Lab0 发布的代码，总共有多少个**.c** 文件，多少个**.h** 文件？你是用什么样的 **shell** 命令得到你的结果的？

有 15 个.c 文件，17 个.h 文件。在终端中键入 find . -type f -name

"*.c"命令则显示出

```
allen@allen-Aspire-4743:~/oslabwangqi$ find . -type f -name "*.c"
./OSLab0/boot/main.c
./OSLab0/src/lib/abort.c
./OSLab0/src/lib/random.c
./OSLab0/src/lib/string.c
./OSLab0/src/main.c
./OSLab0/src/device/video.c
./OSLab0/src/device/font.c
./OSLab0/src/device/timer.c
./OSLab0/src/irq/irq_handle.c
./OSLab0/src/irq/i8259.c
./OSLab0/src/irq/idt.c
./OSLab0/src/game/keyboard.c
./OSLab0/src/game/game.c
./OSLab0/src/game/effect.c
./OSLab0/src/game/draw.c
```

再输入 find . -name "*.c" |wc -l 命令则显示出文件个数为 15 个

```
allen@allen-Aspire-4743:~/oslabwangqi$ find . -name "*.c" |wc -l
15
```

同样对于.h 文件一样输入命令 find . -type f -name "*.h"，显示出

```
allen@allen-Aspire-4743:~/oslabwangqi$ find . -name "*.h" |wc -l
17
./OSLab0/boot/boot.h
./OSLab0/include/adt/linklist.h
./OSLab0/include/string.h
./OSLab0/include/game.h
./OSLab0/include/device/timer.h
./OSLab0/include/device/font.h
./OSLab0/include/device/video.h
./OSLab0/include/irq.h
./OSLab0/include/const.h
./OSLab0/include/common.h
./OSLab0/include/types.h
./OSLab0/include/x86/x86.h
./OSLab0/include/x86/cpu.h
./OSLab0/include/x86/memory.h
./OSLab0/include/x86/io.h
./OSLab0/include/assert.h
```

再输入再输入 find . -name "*.h" |wc -l 命令则显示出文件个数为 17 个

2. 在所有的**.c** 和**.h** 文件中，单词**"volatile"**总共出现了多少次？你是用什么样的 **shell** 命令得到你的结果的？**(hint:** 尝试 **grep** 命令**)** 检查这些结果，你会发现 **volatile** 是 **C** 语言的一个关键字，这个关键字起什么样的作用？删除它有什么后果？

有 15 个。命令为 find . -name *.[ch] | xargs grep -o volatile 和 find . -name *.[ch] | xargs grep -o volatile | wc -l

```
allen@allen-Aspire-4743:~/oslabwangqi$ find . -name *.[ch] | xargs grep -o volat
ile
./OSLab0/boot/boot.h:volatile
./OSLab0/boot/boot.h:volatile
./OSLab0/boot/boot.h:volatile
./OSLab0/include/x86/cpu.h:volatile
./OSLab0/include/x86/cpu.h:volatile
./OSLab0/include/x86/cpu.h:volatile
./OSLab0/include/x86/cpu.h:volatile
./OSLab0/include/x86/cpu.h:volatile
./OSLab0/include/x86/io.h:volatile
./OSLab0/include/x86/io.h:volatile
./OSLab0/src/lib/string.c:volatile
./OSLab0/src/lib/string.c:volatile
./OSLab0/src/device/video.c:volatile
./OSLab0/src/game/keyboard.c:volatile
./OSLab0/src/game/game.c:volatile
```

```
allen@allen-Aspire-4743:~/oslabwangqi$ find . -name *.[ch] | xargs grep -o volat
ile | wc -l
15
```

volatile 变量是说这变量可能会被意想不到地改变，也就是说，优化器在用到这个变量时必须每次都从原始内存地址中读取这个变量的值，而不是使用保存在寄存器里的备份。
如果删掉 volatile 则有可能因为中断服务程序修改后其他程序再使用这个变量时与实际结果不一致，或者多线程程序中几个程序共享这一变量也可能导致这个变量与实际不符，系统也有可能报错。

3. C 语言声明和定义的区别是什么？请注意一个细节：框架代码中所有的函数的声明都在**.h** 文件中，定义都在**.c** 文件中，但有一个例外：**inline** 的函数以 **static inline** 的方式定义在**.h** 文件中。这是为什么？如果把函数或变量的定义放到头文件中，会有什么样的后果？

声明只是告诉编译器我有这个变量，我要用到这个变量，不需要分配内存，而定义则是建立在存储空间之上，要为变量分配内存。
inline 是关键字，它可使得函数定义在头文件中，而包含它的源文件调用该而不发生重定义，因为 inline 实质是在调用的地方“展开”，将函数执行的操作融合进去。而关键字 inline 必须与函数定义体放在一起才能使函数成为内联，仅将 inline 放在函数声明前面不起任

何作用。用 static 可以限制该函数只在本文件中可访问。
如果只有一个源文件包含该头文件则不会有错，但是当多个文件包含时则编译会报错，说函数或者这个变量重复定义。

4. Makefile 中用 **ld** 链接 **start.o** 和 **main.o**，编译选项的**-e start** 是什么意义？**-Ttext 0x7C00** 又是什么意义？**objcopy** 中**-S, -O binary, -j .text** 又分别是什么意义？**(**请参考 **man** 手册以及我们提供的文档**)**

使用 start 标志符作为程序执行的开始端。将编译生成的二进制代码的起始段存在内存地址 0x7C00 处。-S 去掉源文件的符号信息和 relocation 信息，-O binary 输出指定文件的二进制文件，-j .text 只将由分段名称指定的 text 代码段拷贝到输出文件，可以多次指定。

5. main.c 中的一行代码实现了到游戏的跳转：

```
((void(*)(void))elf->entry)();
```

这段代码的含义是什么？在你的游戏中，**elf->entry** 数值是多少？你是如何得到这个数值的？为什么这段位于 **0x00007C00** 附近的代码能够正确跳转到游戏执行？

首先这是一个指向函数地址的函数指针，它的含义是实现启动程序向游戏应用程序的跳转，是从启动程序的地址跳转到游戏程序的入口地址。在我的游戏中它的数值是 0x100190 ，到终端游戏目录里，输入 readelf -h game 命令，即可得到有关 game 文件的信息

```
allen@allen-Aspire-4743:~/oslabwangqi/OSLab0$ readelf -h game
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x100190
  Start of program headers:          52 (bytes into file)
  Start of section headers:          35912 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         17
  Section header string table index: 14
```

其中 Entry point address 就是游戏入口地址，所以即得到 elf->entry 的数值。由于是函数指针，所以将游戏入口地址存入 elf->entry 中，最后函数返回这个地址，程序强制跳转到游戏入口，继续执行。

6. start.S 中包含了切换到保护模式的汇编代码。切换到保护模式需要设置正确的 **GDT**，请回答以下问题：什么是 **GDT**？**GDT** 的定义在何处？**GDT** 描述符的定义在何处？游戏是如何进行地址转换的？

GDT 全局描述表（ Global Descriptor Table ），可以定义在内存中的任何位置，但由于 GDT 中描述符都是 64-bit 长，也就是说都是 8 个字节，所以为了让 CPU 对 GDT 的访问速度达到最快，我们应该将 GDT 的入口地址放在以 8 个字节对齐，也就是说是 8 的倍数的地址位置，我们用寄存器 GDTR 用来存放 GDT 的入口地址。
在相应的入口地址装入段选择符子，按照这个段选择子可以到 GDT 中找到相应的段描述符，这个段描述符中记录了此段的基地址，限制和访问权限，然后加上偏移量，就得到了最后的内存地址，实现地址的跳转。

7. 为什么在编译选项中要使用**-Wall** 和**-Werror**？**-MD** 选项的作用是什么？

-Wall 打开 gcc 的所有警告。-Werror，它要求 gcc 将所有的警告当成错

误进行处理,出现任何警告即放弃编译。可以减少代码中的低级错误，减少程序中的漏洞。
-MD 和-M 选项类似首先它告诉预处理器输出一个适合 make 的规则，用于描述各目标文件的依赖关系。对于每个源文件，预处理器输出一个 make 规则，该规则的目标项(target)是源文件对应的目标文件名，依赖项是源文件中 #include 引用的所有文件。不同的地方在于它是把依赖信息输出在文件中，文件名通过把输出文件名末尾的 .o 替换为 .d 产生，同时继续指定的编译工作，而且 -MD 也不象 -M 那样阻止正常的编译任务。

8. Makefile 中包含一句：**-include $(patsubst %.o, %.d, $(OBJS))**。请解释它的功能。注意 **make** 工具在编译时使用了隐式规则以默认的方式编译**.c** 和**.S** 文件。

make 工具在编译时使用了隐式规则以默认的方式编译.c 和.S 文件，而 OBJS=$(CFILES:.c=.o)$(SFILES:.S=.o)，OBJS 就表示将所有.c 和.s 生成.o 文件，patsubst 就表示将.o 文件替换成.d 文件，最后用 include 包含进去，因为.d 文件中存储的都是.o 文件的执行所需要的头文件，也就是.c 文件和头文件的依赖关系，所以 include 后，在编译.c 文件时会检测.d 文件，看.c 文件的依赖关系是否有修改，进而判断是否需要重新编译。

9. 请描述 **make** 工具从**.c, .h** 和**.S** 文件中生成 **game.img** 的过程。

首先 make 工具会读取.c 文件，并存到 CFILES 变量中，这一过程也会在.c 文件中找到对应的头文件，再读取.S 文件，存到 SFILES 变量中，然后再对两个变量中的文件进行编译预处理，使其替换成.o 文件，然后再用 make 工具，将.o 文件替换成.d 文件，这样就将.c 文件和.h 文件的依赖关系存到.d 文件中，预处理编译之后，在将其编译成汇编语言，再将汇编代码翻译成指令，生成.s 文件，然后再调用链接程序，将生成的目标代码链接成一个可执行程序 game，然后再在 boot 目录中执行 make 操作，也是同样的步骤预处理编译汇编链接，生成主引导程序 bootblock，最后 cat boot/bootblock game > game.img 将 bootblock 启动程序和 game 游戏程序合并成 game.img 文件，最后游戏得以运行。

10. include/adt/linklist.h 定义了一种通用链表，在游戏中我们使用到它**(**链表的结构体定义在 **include/game.h)**。另一种通 用链表的定义可以参考 **Linux** 内核中的 **list head**。这两种链表定义方式有什么不同？各自的优点和缺点是什么？

首先一个很重要的差别就是这里的 list head 没有数据域，在 Linux 内核链表中，不是在链表结构中包含数据，而是在数据结构中包含链表节点。而我们自己定义的链表则有自己的数据域。
对于定义的通用链表，其优点是节省内存空间，而且便于访问其下一个结点的数据，缺点就是，由于有时数据域不同，虽然有时可以用模板来解决，但是其他的时候则不得不为每个数据项来定义自己的链表，这样就相对麻烦。
对于 linux 内核中的 list head，优点是通用的链表结构避免了为每个数据项类型定义自己的链表的麻烦。这样就可以用相同的数据处理方式来描述所有双向链表，不用再单独为各个链表编写各种编辑函数。缺点是显然在访问其下一个节点的数据时没有自己在 include/adt/linklist.h 定义的通用链表那么方便，而且链表头中元素置为 NULL 不是初始化，与普通习惯不同，仍然需要单独编写各自的删除整个链表的函数，不能统一处理，因为不能保证所有链表元素结构中链表头结构 list_head 的偏移地址都是相同的。

11. 详细说明一次时钟中断从进入系统到处理完毕后返回的过程中，堆栈变化的情况，注意以下几个关键点：中断进入前、硬件跳转到 **irq0** 时、**call irq_handle** 前、**irq_handle** 中对堆栈内数据的使用、**iret** 前。

在中断进入前，为了保留现场，CPU 硬件会将 EFLAGS 状态寄存器，CS 段寄存器和返回指令的地址以及 IP 程序计数器保存在堆栈上，同时还会将 IF 标志位清零以暂时阻止外部中断的发生，然后处理器跳转到中断处理程序比如 irq0，在中断处理之前，保存通用寄存器，将所有的通用寄存器保存到栈中，这样中断处理程序就可以随意使用或修改寄存器了，然后用 call 指令跳转到相应的 C 语言中断处理程序，通过 TrapFrame *tf 中的 tf->irq 来判断做哪种中断处理，处理之后，再恢复现场，恢复原始寄存器现场，再将 IP，指令地址，CS 和 EFLAGS 依次出栈，恢复中断处理前的现场，程序继续执行。

12. draw.c 实现的是绘图功能。如果需要绘制的内容过多，可能会无法维持适当的 **FPS(**例如 **30FPS** 要求一帧在大约 **33ms** 内绘制完

毕)。我们的游戏设计成游戏逻辑优先更新，并丢弃过去未绘制的帧(大部分同学玩大型游戏时都有过类似的体验)。结合代码简述这个机制是如何实现的。

程序中有两个变量来控制 now 和 target，比如当一次 redraw_screen ( ) 执行后，表示进行时间的 tick 就会增加的很多，那么再循环回来时，target 就会比 now 大很多，而且也有可能接收到很多个键盘响应，这样 while (now<target) 的循环执行的次数就会变得很多，而当 if (now % (HZ / FPS) == 0) 时 redraw 就为 true，所以在这么多次的循环中，可能会有很多次的 if 条件的成立，这样就省掉了一些帧的绘制，但是此时逻辑还是继续的，所以我们看到游戏感觉是十分卡钝的。