Lab0:不依赖于操作系统的游戏

戴威，111220016

南京大学，计算机科学与技术系

1 实验内容

本次实验的实验内容是在提供的框架代码的基础上，实现一个不依赖于操作系统以及库函数的游戏。

2 实验目标

- 初步掌握 Linux 的使用与开发方法
- 初步认识 IA-32 体系结构与中断机制
- 培养阅读文献、使用搜索工具、解决问题的能力
- 培养良好的编程习惯、代码风格

3 预备知识

- 框架代码 https://github.com/jiangyy/OSLab0.git
- 计算机加电启动引导过程 http://cslab.nju.edu.cn/opsystem/#OS2013_2
- 跟我一起写 Makefile http://www.chinaunix.net/old_jh/23/408225.html
- Linux Man Pages.

  我主要用到了 gcc，gdb，ld，objdump，cat 等几个 man page。

4 实验设计

4.1 游戏简介

我设计了一个飞行类游戏，游戏的参考来源是人人网上前一阵子的一个热门应用“是男人就坚持 20 秒”。游戏的内容为操控一个小飞机上下左右移动飞行，躲避四面八方飞来的小炸弹。

4.2 游戏设计

游戏大致分成主逻辑、游戏效果、屏幕刷新三部分设计。在 src/game 路径下的源文件中定义，在 include/game.h 头文件中声明。

4.2.1 游戏主逻辑设计

根据框架代码，从 src/game/game.c 中的 main_loop()函数开始，进入游戏逻辑。我的主逻辑实现同样也是在 src/game/game.c 文件中，主要包含以下三个函数：

```
void main_loop(void);
void game_loop(void);
void game_initial(void);
```

其中 game_loop()是我的游戏的帧刷新逻辑函数，和框架代码的 main_loop()函数大致相同。设计游戏时间节点变量 now，不断循环刷新游戏主逻辑（帧），每次刷新游戏帧时，now 自增，以追赶时钟中断产生的记时变量 tick。并在刷新游戏帧之中，按一定间隔插入刷新屏幕、刷新游戏内容的过程。

我做的一点改进是设立了游戏可重复开始的机制。这一点在 main_loop()里实现。main_loop()中设立变量 startflag，一旦该变量为 true，则循环执行 game_loop()。为了实现游戏重复开始，在每一次完整的游戏过程之前，必须对游戏中要用到的一些变量进行初始化，这个功能在 game_initial()里实现。

4.2.2 游戏效果设计

游戏逻辑效果的设计在 src/game/effect.c 中，主要包括以下几个全局变量与函数:

```
typedef struct plane_body          //飞机结构体定义
{   float x;          float y;
}   plane_body;
typedef struct bomb_body           //炸弹结构体定义
{   float x;   float y;
    float vx;   float vy;
}   bomb_body;
static bomb_body bomb[NUM];//NUM 为宏方式定义的炸弹个数
static plane_body plane;
void plane_initial(void);
 void bomb_initial(void);
void protect_center(void);//使炸弹不生成在屏幕中心区域
void protect_border(void);//使炸弹不飞出屏幕
void update_bomb_pos(void);
 bool check_crash(void);//检测是否发生撞机
bool keyboard_plane_control(void);
bool keyboard_game_start(void);
```

结构体 bomb_body 和 plane_body 的定义是我的游戏的核心效果逻辑。bomb_body 结构体由 x,y,vx,vy 四个 float 型变量构成，以描述一个炸弹的状态信息。plane_body 结构体由 x,y 两个 float 型变量构成，用于描述飞机的位置。炸弹位置随时钟增长而按速度发生变化，飞机位置仅在键盘操作时发生变化。

由于使用的两组变量 bomb[NUM]和 plane 都是作为 effect.c 文件的全局变量而存在的，故我设计的大部分游戏效果函数都不需要参数传递，而是对这两个量进行访问和操作。

游戏逻辑中每秒 1000 帧，其中每 10 帧调用一次 update_bomb_pos()，实现炸弹的随机飞行，并在每次更新炸弹位置时，检测是否有炸弹和飞机位置太接近，即 check_crash()函数功能。而飞机的控制靠 keyboard_game_start()函数实现，每次按下一个方向控制键，飞机向相应方向移动 4 个坐标长度。也就是说，飞机的位置仅响应键盘操作，而不响应异步的时钟中断。

4.2.3 屏幕刷新设计

游戏屏幕刷新机制在 src/game/draw.c 中定义，包含以下三个函数:

```
void draw_gamescreen(void);//绘制游戏帧
void draw_startscreen(void);
 void draw_stopscreen(void);
```

在一次完整游戏过程中，开始屏幕和结束屏幕只绘制一次，而游戏帧每秒绘制 30 次。这三个函数主要调用框架代码提供的 draw_string()函数和我自己定义的 draw_plane()，draw_bomb()函数，大同小异。

5 具体实现

5.1 游戏可重新开始的实现

我修改了 game.c 的逻辑结构，新增 startflag 全局变量，作为控制游戏开始的标志。将原本永真的死循环改为 while(startflag)，并在这层循环外面套上一层 while(true)，从而实现游戏可重新开始。注意，每一次游戏开始都要将相关的一些变量重新初始化，尤其是时钟中断维护变量 tick 和游戏时间变量 now。

为了实现上述功能，我将 now,num_draw 等变量由局部变量改为全局变量，代码的模块化性质下降了。如果我再重新做一次的话，我会考虑函数参数传递方式。

5.2 炸弹数组的初始化细节

我定义了一个包含 x,y,vx,vy 四个成员变量的结构体，名为 bomb_body，并用该结构体定义炸弹数组 bomb[50]，可以表示 50 个炸弹。对这 50 个炸弹状态的初始化，我是这样实现的:

```
bomb[i].x = rand() % ( SCR_HEIGHT / 8 - 2) * 8 + 8;
bomb[i].y = rand() % ( SCR_WIDTH  / 8 - 2) * 8 + 8;
bomb[i].vx = (rand() % 140) / 100.0 - 0.7;//generate a float num -0.7 ~ 0.7
bomb[i].vy = (rand() % 140) / 100.0 - 0.7;
```

炸弹位置随机生成在屏幕安全区域内。经反复测试，炸弹速度在-0.7 ~ 0.7 时，游戏难易度适中。

注意我在初始化时还调用了 protect_center();函数，我用这个函数巧妙地实现了将炸弹位置不初始化在屏幕中间一块的功能。代码如下：

```
if( x_diff < 30 && y_diff < 50)        //x_diff = bomb[i].x – SCR_HEIGHT/2
{   bomb[i].x = (int)(bomb[i].x * 2) % ( SCR_HEIGHT / 8 - 2) * 8 + 8;
    bomb[i].y = (int)(bomb[i].y * 2) % ( SCR_WIDTH  / 8 - 2) * 8 + 8;          }
```

检测每个炸弹是否离屏幕中心（100，160）过于接近，如果接近，则进行 if 里的操作。bomb[i].x = (int)(bomb[i].x * 2) % (SCR_HEIGHT / 8 - 2) * 8 + 8;这句代码是我认为的巧妙之处。简单的说，这句代码可以简化为以下内容：bomb.x = bomb.x * 2 % 200 ;200 是屏幕 x 轴的像素数。对 bomb.x 的值先乘以 2，再取 200 的余数。我设定当 bomb.x 的值在 70 – 130 时，对其进行上述操作。注意到屏幕 x 轴共有 200 个像素。如果 bomb.x 在 70 – 100 之间，则修改后的 bomb.x 值在 140 – 200 之间；如果 bomb.x 在 100-130 之间，则修改后的 bomb.x 值在 0-60 之间，都避开了 70-130 这个接近屏幕中心的范围。

y 方向过于靠近屏幕中心的修改方式同理。不过偏差值为 50，比 x 方向稍大一点。

5.3 炸弹与飞机的位置更新

炸弹的位置更新仅响应时钟中断，即在 game_loop()中以每秒 1000 游戏帧，每 10 个游戏帧调用一次 update_bomb_pos()函数的方式来实现。具体的代码为

```
bomb[i].x += bomb[i].vx;
bomb[i].y += bomb[i].vy;
```

以 x 方向和 y 方向各有一个随机的速度分量来描述炸弹的飞行轨迹，使得每个炸弹的速度、飞行方向各不相同，且飞行轨迹符合直观的简单物理规律，同时避免了数学函数的使用。

飞机的位置更新仅响应键盘中断，即在 game_loop()中调用 keyboard_plane_control()函数。具体的代码为：

```
if(query_key(9))              //J pressed （ left ）
{   plane.y -= 4;
    release_key(9);
    return TRUE;    }
```

以上是响应向左飞行按键操作的部分，其他三个方向大致相同。我并没有设计成按下一个方向键飞机一直向那个方向飞行的设计，那样的设计不符合玩家的预期操作想法和习惯。但设计成仅响应键盘更新飞机位置也带来一些缺点:

1、飞机飞行显得不流畅，每次按键飞行 4 个像素。

2、经测验，键盘响应的频率大约是 10Hz，也就是按下一个键不放，飞机位置每秒大约更新 10 次，相较于炸弹 100Hz 的位置更新频率，我认为这个次数过少了。这也是我将飞机每次飞行设置成 4 个像素的原因。

3、大量的键盘响应会使得游戏卡顿，我会在"遇到的困难与解决方法"这一部分中仔细分析。

5.4 炸弹与飞机的绘制

仿照框架代码中的 draw_character()我在 src/device/video.c 自定义了两个函数 draw_bomb()和 draw_plane()。draw_bomb()绘制一个炸弹，由五个像素组成，十字形。draw_plane()函数类似 draw_character(),不过其中的飞机的像素图像编码是我自己定义的，即 src/device/font.c 中的 char font_plane[8]。

6 遇到的困难与解决方法

6.1 在没有数学函数的基础上建立运动的物理模型

由于本次实验是在裸机上运行的，仅依赖框架代码，游戏的设计无法运用一系列库函数。我设计的游戏需要一些基本的运动学和碰撞反弹的物理模型，原本我想通过数学函数建立飞机与炸弹飞行的角度、速度、位置以及碰撞反弹的物理模型，但苦于没有库函数，想法告罄。

最终我通过设立 x，y 分量与分速度的方式描述运动与反弹的物理模型。x,y 方向分速度各自随机（一定范围内），从而运动合速度随机（一定范围内）、运动角度 360°随机，炸弹撞击屏幕边缘时，相应的一个方向的分速度量取反，即可描述完全弹性碰撞。

这种做法避免了数学函数的使用，同时符合一定程度的物理特征。局限性是模型过于简单，只能描述匀速直线运动和完全弹性碰撞。

6.2 链表与数组的使用问题

框架代码中定义了一个可使用的链表形式，我的游戏的第一个版本的炸弹和飞机都是用链表描述的。遗憾当时没使用 git 记录下来。

在写到 protect_border()这个函数时，也就是实现炸弹的边界碰撞反弹功能，我发现我的游戏逻辑中，飞机与炸弹的个数都是始终恒定不变的，因而链表的数据结构对我的游戏来说意义不大。将飞机数炸弹改为结构体一维数组后，代码量下降了不少，代码可读性也提升了不少。性能方面倒是没深究。

6.3 炸弹与飞机碰撞的准确判定

我最初的版本碰撞语判定句为 if(plane.x == bomb[i].x && plane.y == bomb[i].y)。游戏试玩后发现经常有炸弹飞过飞机机身但是没有发生碰撞响应。

第二个版本我将判定条件放宽，语句如下：

```
if(x_diff < 3 && y_diff < 2 )        //diff 表示二者差
//crash…
```

游戏运行后发现不同角度发生的碰撞，飞机碰撞判定距离不同。原因是飞机的绘制问题。模仿draw_character()的方式绘制的飞机，是以左上角为中心画的 8*8 的图像。因而左上角的碰撞判定比右下角的判定距离要近 2-3 个像素。因而我在 draw_plane()函数中做了一个位置修正，如下：

```
x = x - 3;    y = y - 3;
```

从而实现左上角和右下角的碰撞判定距离基本相同。

6.4 游戏的卡顿问题

在整体游戏写好后，游戏存在卡顿问题。我尝试-DSLOW，情况有所好转，随之而来的问题是偶尔发生炸弹飞过飞机机身但不判定碰撞。

我总结了游戏卡顿的特征如下：

1、游戏时而卡顿时而流畅，当一个方向键被按下不放时经常卡顿。

2、游戏卡顿时，统计的 FPS 并未下降，仍保持在 30FPS

3、游戏卡顿时，我的计时器的时间也变慢了。我的计时器是通过 tick/1000 实现的。

4、游戏卡顿时，一直按下方向键，炸弹的飞行（响应时钟）变慢，但飞机的飞行（响应键盘）不变慢。

7 总结

这次试验是我真正第一次在 Linux 下编程。很多工具的使用还不会、不熟练。我第一次使用 Git 时，我的游戏已经基本完成。GDB 也没有怎么用到，程序中添加 assert 的习惯也没有培养好。现在看来，大一学习的内容还是太过基础了，Linux 博大精深，以后还有很多要学习的地方。

另外，我努力压缩篇幅，实验报告到这里还是到了第六页，深表抱歉。

最后，对应前文的四点实验目标，我认为我初步认识了 linux 下的 C 开发方法，接触了 vim，gcc，shell 等工具，对"中断"的概念有了基本的直观感受，并大量使用 man page、google 等搜索资料、文章，也在向良好的编程习惯靠拢，为以后的操作系统实验打基础。

8 实验问题

8.1 基本知识和基本概念

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？

答：

共有 15 个.c 文件，17 个.h 文件

命令：

find . | grep '\.c$' | wc –l

15

find . | grep '\.h$' | wc –l

17

命令解释：

1) find . 找到当前路径下所有文件
2) grep '\.c$' 匹配符合以.c 为结尾的输入作为输出，其中\是转义符，$表示结尾
3) wc –l 统计行数

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？

答：

"volatile"共出现 15 次

命令：

find . | grep '\.c$\|\.h$' | xargs grep 'volatile'

…

find . | grep '\.c$\|\.h$' | xargs grep 'volatile' | wc -l

15

1) volatile 的作用:

volatile 是一个类型修饰符，用在变量的定义与声明中。以 volatile 方式声明的变量，编译器不再对它进行寄存器读写优化，程序每次读/写这个变量，系统都会访问内存，而不是这个变量在寄存器中的副本，从而实现稳定的访问这个变量。

volatile 修饰符一般用于修饰可能会被意想不到的改变的变量，例如多道线程共用的变量，或者是用于中断服务的变量。

对于本题而言，用命令

find . | grep '\.c$\|\.h$' | xargs grep 'volatile'

可以发现，两个中断响应函数的变量 tick 和 key_code 都以 volatile 修饰，从而每次对这两个变量的访问，都能得到它们的准确值，而非寄存器中的副本。

2) 删除 volatile 的后果：

tick 和 key_code 这两个变量在程序运行的过程中会随着异步的中断响应而发生变化，这种变化编译器是无法预知的。如果不以 volatile 修饰，则在对这两个变量进行访问时，可能因为编译器对寄存器使用方面的优化而无法稳定访问，从而无法实现键盘响应与时钟响应的逻辑。

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？

答：

1) 定义与声明的区别：
这里分别讲函数、变量的定义与声明的区别。

函数的定义指的是给出函数的具体实现（代码），包括返回值类型，函数名，函数参数和函数体四个部分，从而完整的构成一个具有指定功能的函数。函数声明仅提供出函数的原型，通常包括返回值类型、函数名、函数参数三个部分，其作用有两点。一是通知编译器，在调用函数时进行检查，是否有类型、名称等错误，二是向其他源文件"介绍"这个函数，使得位于当前源文件内的函数可以作为一个公共子程序供给其他源文件使用。

变量的定义指的是为一个变量申请一块内存空间，并赋予名称和变量类型。变量可以在定义时赋值。而变量的声明仅是告诉编译器一个变量的名称和类型，并没有"建立"这样一个名称和类型的变量。

2) inline 函数例外的原因：
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 --引自《高质量 C++/C 编程指南》。对此，我的理解是，inline 函数在编译时，类似宏替换，使用函数体替换调用处的函数名，正是这一特性使得 inline 函数对编译器而言必须是可见的，所以说"关键字 inline 必须与函数定义体放在一起有效"。

对于本实验而言，将 inline 函数以 static inline 的方式定义在.h 文件中，使得这种方式定义的函数具有外部函数的效果，可以被多个源文件使用。另外注意到有一个例外，在/src/video.c 中，draw_characters 函数以 static inline 方式定义在.c 文件中，是因为这个函数仅作为内部函数供给 video.c 这一个源文件用。

3) 函数、变量的定义放到头文件中的后果
如果把函数放在头文件中定义，则这些函数会作为外部函数定义。所有 include 这个头文件的源文件都可以使用这个函数，从而失去了访问控制。

如果把变量放在头文件中定义并初始化，则会产生 multiple definition 错误。而如果不初始化，则变量在头文件中仅为定义式声明，可以被多个源文件使用，在每个源文件中具有全局作用域。可以发现，变量在头文件中定义，不仅很可能产生重复定义的错误，也会使得程序设计的模块化特性降低，所以变量不宜在头文件中定义。

8.2 主引导扇区

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)

答：

1) –e 与-Ttext 的意义

-e start 的意义是：以 start 作为程序执行的开始，start 是定义在 start.S 中的汇编代码段。

-Ttext 0x7C00 的意义是：连接时将程序的初始地址重定向至 0x7c00，对于本实验而言，结合前面的-e start 参数可知，-Ttext 0x7C00 的含义即是将 start 这个汇编函数重定向至内存 7C00H 处。通过以下 shell 命令可以证实：

```
objdump -d bootblock.o

bootblock.o:     file format elf32-i386

Disassembly of section .text:

00007c00 <start>:
    7c00:    fa                   cli
    7c01:    b8 13 00 cd 10              mov    $0x10cd0013,%eax
    7c06:    31 c0                xor    %eax,%eax
             …
```

参考 ld manual：

-e entry Use entry as the explicit symbol for beginning execution of your program, rather than the default entry point.

-Ttext org (the same as --section-start) Locate a section in the output file at the absolute address given by org.

2) objcopy 中-S, -O binary, -j .text 的意义

-S：去掉源文件的符号信息和重定位信息

-O binary：使用二进制格式来写输出文件 bootblock

-j .text：仅将.text 这个段（section）写到输出文件 bootblock 中

参考 unix man objdump 手册：

-s	Do not copy relocation and symbol information from the source file.
-O bfdname	Write the output file using the object format bfdname.
-j sectionname	Copy only the named section from the input file to the output file.

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？

答：

1) 代码的含义

void(*)(void)是类型描述，描述一个返回值是 void,参数是 void 的函数指针类型，这段代码即将 elf->entry 这个 unsigned int 类型的值做强制类型转换，转换为一个返回值为 void,参数为 void 的函数指针类型，并调用这个函数。

2) elf->entry 的数值

我的游戏的 elf->entry 数值为 0x100490，原本的 OSLab0 的 elf->entry 数值为 0x1001d0，我是通过在 shell 中输入 readelf –h game 这个命令得到这个数值的。

3) 为什么跳转到游戏执行

我的游戏的 elf->entry 数值为 0x100490，我尝试在 shell 中键入如下命令：

objdump -d game | grep '100490'

得到的结果为：

00100490 <game_init>:

100490: 55 push %ebp

可以发现，((void(*)(void))elf->entry)();这段代码相当于调用 game_init()这个函数，这个函数是整个游戏执行的开端，调用了这个函数，从而开始游戏。

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？

答：

1) 什么是 GDT

GDT 是全局描述符表（Global Descriptor Table）的缩写，是一种一维数据结构，用于存放保护模式下的段描述符。每个段描述符为 64bit，由 Base Address, Limit, Access 三种成分组成，一个段描述符描述保护模式下的一个段。IA-32 为了保持向下兼容，段寄存器仍为 16bit，存储不下段描述符，因而将段描述符存储在 GDT 数组中，在段寄存器中存储数组的下标以访问段描述符。

2) GDT 定义在何处？

GDT 在 boot/start.S 中 46-49 行以宏的方式定义，宏定义在 asm.h 中。可以发现，本实验的 GDT 共有三项，第一项为空，第二项为代码段描述符，第三项为数据段描述符。

3) GDT 描述符定义的定义在何处？

GDT 描述符在 boot/start.S 中 51-53 行定义，名称为 gdtdesc。

4) 游戏如何进行地址转换？

游戏运行在保护模式下，以下描述保护模式下虚拟地址 xxxx:yyyyyyyy 的地址转换过程。xxxx 为段寄存器中存储的段选择子，其中前 13 位为当前段的段描述符相对于 GDT 首地址的偏移字节数。通过 xxxx 访问段描述符，从段描述符中取出该段的 32 位 Base Address，再加上当前的 32 位偏移量 yyyyyyyy 即得到实际地址。

8.3 游戏 MAKEFILE

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？

答：

1) –Wall
   这个参数的意思是打开 gcc 的所有警告。

2) –Werror
   这个参数的意思要求 gcc 将所有的警告当成错误进行处理。

3) –MD
   这个参数的意思是要求 gcc 生成编译时每个.c 文件的依赖关系，存储在同名的.d 文件中

参考 gcc manual –wall –Werror –MD

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。

答：

首先解释 patsubst %.o, %.d, $(OBJS)，这句代码的意思是生成一系列文件名，将所有目标文件的文件名中“.o”部分替换为“.d”，我理解的这句代码的作用和 OBJS:.o=.d 相同，即-include $(OBJS:.o=.d)。由上题知.d 文件是 gcc 编译时由参数-MD 指定生成的依赖关系文件，用于 Makefile。将所有.d 文件 include 进当前 Makefile，即完善了依赖关系。

另外，我尝试用 # 将本行注释掉，本实验依然可以正常编译，即隐式规则就能反映出依赖关系。

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

答：

一次 make 执行以下两个显式的生成过程。

```
game.img: game
        @cd boot; make
        cat boot/bootblock game > game.img
game: $(OBJS)
        $(LD) $(LDFLAGS) -e game_init -Ttext 0x00100000 -o game $(OBJS)
```

以下描述 make clean 后 make 的执行过程。考察依赖关系，可以得出以下执行顺序：

1) 编译出所有目标文件(.o 文件)，编译器为 gcc，编译参数由 CFLAGS 指出。
2) 链接所有目标文件，参数为变量 LDFLAGS 和 -e game_init -Ttext 0x00100000，生成文件 game。
3) 进入目录 boot，编译并链接所有源文件(.c,.S)，生成目标文件 bootblock.o

4) 以指定格式拷贝文件 bootblock.o 中的内容，生成二进制文件 bootblock
5) 执行./genboot.pl bootblock，改造 bootblock，使之符合主引导扇区格式
6) 执行 cat 命令，将 bootblock 和 game 两个文件合并生成 game.img 文件

8.4 游戏

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？

答：

1) linklist.h 中定义的链表

用替换命令替换掉 linklist.h 中的 NAME##，并去掉所有/号，可以得到第一种链表的定义方式。以下简述。

首先定义结构体

```
struct fly_t
{
    float x,y;
    int text;
    float v;
    struct fly_t *_prev, *_next;
};
typedef struct fly_t *fly_t;
```

在初始化时，生成一个 struct fly_t 类型的静态数组，元素个数为 SIZE+1，并将每个元素的*_next 成员赋值为下一个元素的地址（最后一个元素除外），从而形成一个初始的链表。设置一个指针 fly_free_head 指向当前数组中第一个还没被使用到的元素，以供链表的增删节点等操作。

同时定义了一系列对链表的操作的函数。

```
fly_t fly_prev(fly_t node); //获得当前节点的前一个节点
fly_t fly_next(fly_t node); //获得当前节点的后一个节点
void fly_insert(fly_t prev, fly_t next, fly_t obj); //将节点插入 prev 与 next 之间
fly_t fly_new(); //新增一个节点
void fly_remove(fly_t node); //将节点脱离链表
void fly_free(fly_t node);//将节点释放
```

其中 remove 和 free 的区别是，remove 函数将节点从链表上取下来，也就是将该节点的前一个节点和后一个节点相连接。free 函数将节点设置为未使用状态，即下次 new 操作就可能会将该节点当成未使用节点分配。

优缺点：这个方式实现了一个静态链表，具有链表的基本特征，增删元素时不需移动元素，仅修改指针即可。不过在初始化时需要分配一块连续的大内存空间，并且一个结构体内只能容下一组指针域。

2) Linux 内核的 list head 链表

查看 include/linux/list.h，可知 Linux 内核的 list head 链表定义如下：

```
struct list_head
{
struct list_head *next, *prev;
};
```

并附以相应的链表的初始化、插入节点、删除节点等函数。注意到这个结构中并没有数据域，要实际用到这个链表，应该新建一种结构体，将这个链表作为成员变量在结构体中定义。如下所示：

```
struct  test_list
{   int  *testdata;
    struct  list_head  list;
};
```

这种方式定义的链表优点有：

1、具备动态链表的基本优点

2、在一个结构体内可以定义多个链表，从而以不同方式链接各个节点

3、任何数据类型都可以使用这个链表

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。

答：

1) 中断进入前：机器状态字 MSW 入栈

2) 跳转到 irq0 时：pushl $1000　　#立即数 1000 入栈

3) call irq_handle 前：pushal #将 8 个通用寄存器（AX、CX、DX、BX、SP、BP、SI、DI）入栈，保护现场

pushl %esp #将上一个栈顶指针压入栈中，作为函数 call_irq_handle 的参数

4) irq_handle 中：通过前一步压入栈的%esp 找到前面压入栈的立即数 1000，调用时钟中断服务函数

5) iret 前：addl $4, %esp　　#esp + 4，恢复栈指针位置，指向步骤 3 之后栈的位置

popal　　#恢复 8 个通用寄存器，恢复中断现场

addl $4, %esp　　#esp + 4，恢复栈指针位置，指向中断进入之前，以供中断返回

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。

答：

redraw 控制是否重新绘制一帧，如果绘制一帧耗时过多，tick 在此间变得很大(超过 30)，那么在下一次 while (now < target)循环中，redraw 可能会被多次设置为 true，直到跳出这个循环才执行重绘屏幕。多次设置 redraw，但只执行一次重绘，这就使得过去未绘制的帧被丢弃。

9 致谢

1、 Google

2、 Wikipedia

3、 CSDN 博客

4、 ChinaUnix 论坛

5、 CoolShell 博客

6、 Linux C 编程一站式学习 http://learn.akae.cn/media/index.html

7、 课程网站 CourseWiki