Lab0: 移植 NES 模拟器

徐栋

南京大学，计算机科学技术系

学号：**121220312**，邮箱：**dc.swind@gmail.com**

时间：**2014.3.24**

摘要：将 **NES** 模拟器实现的一个游戏移植到一个没有操作系统的计算机中。

关键字：**NES** 模拟器；移植；操作系统；中断；

1 实验介绍

在经历过操作系统与计算机体系结构的课程之后，我们对计算机（硬件层面）是如何运行的有了基本的了解。同时对 **Intel IA32** 体系结构有了一些模糊的认识，同时对中断机制、**I/O** 机制有了一定的了解。那么本次实验的目的之一就是要更加深入的了解操作系统是如何调用硬件来完成基本功能的。我们要把一个在操作系统中运行的 **NES** 模拟器，移植到一个没有操作系统、没有运行库 [1] 的计算机中，使其能够运行。也就是让这部分代码以操作系统的方式完成操作系统的一些功能。

2 实验目标

1. 熟悉编程环境及 **Linux** 下的各种开发工具。如 **git**、**vim**、**make** 等，为后续实验奠定基础。

2. 复习体系结构、**C** 语言、汇编语言的知识。将在复习过程中遇到的问题加以解决，使得对体系结构、**C** 语言、汇编语言有更深的了解。

3. 了解并掌握 **CPU** 中断、**GDT**、段等概念，清楚整个实验的执行过程、计算机的启动过程等。

4. 培养阅读文献、搜索资料、独自学习的能力。

5. 将 **NES** 模拟器移植到 **Lab0** 框架中，并进行性能及一些细节上的调优，从而掌握移植方法。

6. 掌握键盘、显示的原理。

3 涉及知识

3.1 NES 模拟器 [2] [3]

3.2 Git[4]

3.3 操作系统运行过程

以 Linux 为例，第一步、加载内核。操作系统接管硬件后，首先读入/boot 目录下的内核文件；第第二步、启动初始化进程。内核文件加载以后，就开始运行第一个程序/sbin/init，它的作用是初始化系统环境；第三步、确定运行级别。许多程序需要开机启动，这些在 Windows 下叫做服务（service）的程序在 Linux 下叫做 daemon。init 进程的任务就是运行这些开机启动程序。但是因为不同情况下启动的程序不同，所以需要分配不同的开机启动程序，这就叫运行级别（runlevel）；第四步、加载开机启动程序。……

在本实验中，其原理与以上 Linux 的例子基本相似。

3.4 Make[5]

3.5 键盘及显示原理简介

键盘：在每个按键被按下，键盘会向计算机发送一个键盘码，告诉计算机某个按键被按下。同时在释放这个按键的时候，键盘会发送一个中断码，告诉计算机某个按键被释放。所以我们可以通过捕捉这两个唯一标识某个按键的码来记录某个按键的状态并以此来作出相应的响应。

显示：显示是通过一个一个像素点的赋值完成的。计算机计算好每一个像素点在某个时刻应该显示的颜色，并将颜色代码写入到显示端口对应的内存区域。(内存映射 I/O)

4 设计思路

4.1 外部库的去除

在阅读完部分源码以及手册的基础上，我有了一个比较清晰的设计思路。首先，在 NES 模拟器（即 litenes 工程）中，尽量去掉可去除的外部调用，例如：stdio.h、stdlib.h、signal.h 等。因为涉及到 ROM 文件的读入，所以考虑将 ROM 作为常量的方式写入程序中，以达到去除输入输出的外部调用。其次是 allegro 库的去除，因为在 hal.c 中 allegro 库的几个函数在 Lab0 中有相同功能的实现。所以考虑以 Lab0 中相同功能代码替换的方式来去掉 allegro 库。第三类是一些需要用到的外部库，例如：string.h、stdbool.h 等。因为只用到了其中的少量代码，所以自行实现是可行的。这样就使得工程不再需要外部调用。

4.2 移植及细节调整

在此基础上，将 litenes 工程的代码先移植到 Lab0 中，修改一些 include 的路径问题后，需要做的就是 hal.c 中的几个 allegro 库的重新实现了。具体实现在下一部分【具体实现】中进行详细说明，在此只提供思路。

之后可以尝试进行 make，根据 make 的报错信息进行细节上的修改。编译通过后，理论上讲是移植基本成功了。接下来就要进行键盘、显示的优化。

4.3 显示及键盘处理

显示的处理。NES 中的图像是 256*240 像素，而 VGA 则是 320*200 像素，所以要进行缩放，使得 NES 返回的图像可以在 VGA 中更加友好的显示。首先想到的是直接删除无用行及保留黑边，这样看起来效果虽然是最好的，但是其有局限性。在不同的 ROM 下，删除的行并不一定都是无效的。所以考虑平滑一些的缩放，二维线性插值法是一种比较基础的效果相对可以的处理方法。

键盘的处理。在 NES 的源码中可以看到，在询问键盘某个键是否被按下的时候，是通过一个返回键盘按键状态的函数来返回的。而在 Lab0 中 lastkey 仅仅是保存了上一次按键的码，仔细观察可以发现 lastkey 中保存的是触发和中断两个码，所以可以通过设置 1 个 bool 型的变量来进行按键状态的保存，这样就可以解决多个按键的组合问题。

5 具体实现

具体的实现依照设计思路的顺序进行。

5.1 外部库的去除

首先，在 NES 模拟器（即 litenes 工程）中，去掉 stdio.h、stdlib.h、signal.h 等外部调用。自己写一个 c 程序将 ROM 文件的读入然后强制类型转换为 int 后输出到一个.h 文件中（rom.h）。稍作修改，在 rom.h 中定义一个数组并初始化为打印出的 int 数组。然后就可以将此数组强制转换回 char 型后使用了。

allegro 库的去除，则是将 hal.c 中关于 allegro 库的调用全部删除，然后进行其中几个函数的重新实现。wait_for_frame() 可以理解为等待下一帧的意思，在 Lab0 中 play() 中的 while(i) 循环则是实现相同功能的部分，将其写入 wait_for_frame() 中。需要注意的是要在 while 之前执行 prepare_buffer() 函数来准备显示的缓冲数组。nes_set_bg_color() 根据函数的参数，使用 for 循环将所有像素点赋值为参数颜色。nes_draw_pixel() 可以调用相同功能的 draw_pixel() 进行实现，但是因为 nes 中的调色板和 VGA 不同，所以要进行调色板的重新设置及 c 的重新计算。因为 c 只有 8 位有效所以 RGB 采用 332 的方式存放（即通过移位操作删除 RGB 各个值的低位，同时使用移位操作将 RGB 拼接）。调色板的初始化在 Lab0 执行前进行初始化，代码写在 palette.h 中。nes_hal_init() 可以直接不予实现，因为在 Lab0 中已经进行了必要的初始化操作。nes_flip_display() 则是由相同功能的函数 display_buffer() 实现。nes_key_state() 则是返回自定义的每个按键的 bool 型状态变量。最后对于 string.h、stdbool.h 等。在 Lab0 中给出实现的直接调用，没有实现的 bool 以及 string 中的 memcmp 函数由自己实现。

需要说明的是，上面是进行不平滑缩放的处理方式，其在 draw_pixel() 函数的实现中，利用简单的坐标变换使得图像居中，且删除部分行。在采用线性插值后，我开辟了一个新的数组 temp_draw_buffer 来暂存 NES 中的 256*240 像素的 rgb。在 nes_draw_pixel() 中，没有调用 draw_pixel() 函数，而

是先暂存于新数组中。然后在 nes_flip_display() 中将所有点经过线性插值计算后调用 draw_pixel() 函数更新点。

5.2 移植及细节调整

将 litenes 工程的代码利用 cp 命令复制到 Lab0 中，维持工程结构（即 litenes 工程中的 include 中的.h 文件 cp 到 Lab0 的 include 中）。修改 include 的路径问题。

之后可以尝试进行 make，根据 make 的报错信息进行细节上的修改。例如：include 的路径不对，变量作用域不对等问题。编译通过后，移植已经成功了一半。接下来进行键盘、显示的优化。

5.3 显示及键盘处理

显示的处理。首先我采用了直接删除部分不影响美观性的行来适应屏幕的高度，而宽度是两侧留黑边的方式，这样看起来效果较好，不会有拉伸的现象。但是正如前面提到的，这样做的一个副作用就是在更换不同 ROM 的情况下，可能当前删掉的行在另一个 ROM 中是起重要作用的行 (即便是每 6 行删除 1 行的方式，我觉得也不够好)。所以我又重新使用线性插值的方法来进行显示。其原理是对于一个目的像素，设置坐标通过反向变换得到的浮点坐标为 (i+u,j+v) (其中 i、j 均为浮点坐标的整数部分，u、v 为浮点坐标的小数部分，是取值 [0,1) 区间的浮点数)，则这个像素得值 f(i+u,j+v) 可由原图像中坐标为 (i,j)、(i+1,j)、(i,j+1)、(i+1,j+1) 所对应的周围四个像素的值决定，公式如下：

$$f(i+u,j+v) = (1-u)(1-v)f(i,j) + (1-u)vf(i,j+1) + u(1-v)f(i+1,j) + uvf(i+1,j+1)$$

其中 f(i,j) 表示源图像 (i,j) 处的的像素值。

键盘的处理。我在 handle_keyboard() 函数中，根据返回的 code 将使用的 8 个键分别设置了一个 bool 型变量记录 8 个按键的状态。则在 nes_key_state() 函数中根据 8 个 bool 变量返回按键状态。

6 问题及解决方法

Q1：allegro5 在 ubuntu12.04 中无法安装。A1：多次尝试无果后，换成 ubuntu13.10 后顺利解决。

Q2：kvm 虚拟化无法开启。A2：最后发现是虚拟机没有开 CPU 虚拟化选项。

Q3：make 的时候提示 32 位 64 位不兼容问题。A3：make clean 后，重新 make。

Q4：移植完成后运行黑屏 1-2 分钟。A4：原因是数组开太大，在缩小冗余的数组后速度大大提高属于正常范围。

Q5：颜色显示不正确。A5：因为代码中 c 的计算式 allegro 库中独自定义的计算方式，而我们没有 allegro 库的调色板，只能自己实现一个调色板，在初始化的时候将系统默认调色板修改，并自定义 c 的计算方法。

Q6：键盘无法多键组合。**A6**：代码中只是存储了 **last_key** 一个值，所以只能存储一个按键的信息，为每一个按键设置一个状态变量解决。

Q7：**ROM** 打表问题。**A7**：**ROM** 打表的过程中，试用 **char** 型打表试用了多个输入输出函数都出现错误，还企图将 **ROM** 作为一个字符串打表。最后转成 **int** 型打表后成功。

Q8：线性插值进行缩放的过程中显示始终不正确。**A8**：多次调试，发现多次头晕算错了移位位数以及弄错 **xy** 坐标。

Q9：键盘处理调试的时候，**case** 语句执行错误。**A9**：痛苦调试多个小时后发现，是 **case** 语句没加 **break**。

7 总结

很高兴的是预计要达到的实验的目标基本达成，同时更加高兴的是培养了自己查阅资料、文档的能力。同时也正如 **JYY** 学长说的那样，不知不觉中真的学到了很多东西，在阅读 **IA32** 手册的过程中，对 **CPU**、内存在硬件层面的工作有了比较深的体会。总的来说，这是一个美丽的开始。

致谢

特别感谢 **JYY** 学长在实验课中给予的正确引导！可以说没有 **JYY** 学长的指导，我不会去读那么多文献，了解这么多知识，更不可能完成这个实验。同时对在实验过程中给予我启发、与我讨论、给予我帮助的同学表示感谢！尤其感谢室友康望程、李乾科，正是多次的深夜学术交流，使得我的理解更加深刻，思路更加清晰。

References

[1] 百度百科 -运行库 维基百科 -库

[2] 维基百科 -FC 游戏机

[3] 维基百科 -游戏模拟器

[4] wikipedia-Git 维基百科 -Git

[5] 维基百科 -Make A Simple Makefile Tutorial

附：问题回答

1. 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？(回答这个问题你需要修改程序，使用 objdump 工具观察 gcc 生成的汇编代码，并且需要很多思考。这是一个很困难的问题)

grep -r volatile | wc -l 其中前面是打印所有出现 volatile 的文件，出现多个会显示多次，而 wc -l 会统计以上的行数。

一个定义为 volatile 的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。删除后使用变量时会得到错误的值, 图像会花、出现死循环。

2.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？

声明指的是告诉编译器在程序中定义过这个标识符，而定义则是申请一块内存。inline 内联函数在编译的时候会将指定的函数体插入到调用处，从而节省调用函数带来的额外时间开销。因为涉及到拷贝，所以需要定义在头文件中使得编译器可以找到，放在其他.c 文件里可能会在编译的时候找不到从而起不到 inline 的作用。同时 inline 内联函数不会引起重复定义的错误，这是 C 的一个特例。函数、变量定义在头文件中会在多次 include 的情况下出现重定义错误。

3.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)

-e start：程序会以 start（默认的）做为入口点开始执行程序。-Ttext：用来指定目标文件中指定段 Ttext（代码段）的起始地址为 0x7C00（该地址为绝对地址，不可被重定向）。objcopy 把一种目标文件中的内容复制到另一种类型的目标文件中。-S 表示移出所有的标志及重定位信息。-O binary bootblock.o bootblock 表示由 bootblock.o 生成文件 bootblock。-j .text：只将由.text（sectionname）指定的 section 拷贝到输出文件，可以多次指定，并且注意如果使用不当会导致输出文件不可用。

4.main.c 中的一行代码实现了到模拟器代码的跳转：((void(*)(void))elf->entry)(); 这段代码的含义是什么？在你实现的模拟器中，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模拟器执行？

以函数指针的形式，跳转到模拟器程序的入口执行; 数值是 0x00106828 ；使用 make debug 输出 game_init 函数地址获得的，具体使用了以下 shell 命令：sudo make debug。sudo gdb game。target remote：1234 。b game_init 。(p game_init) ；这句代码是通过将 elf->entry 转化为一个函数指针，指向模拟器的入口函数，从而正确跳转到模拟器执行。

5.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 是全局描述符表，GDT 的定义在 start.S 中，lgdt gdtdesc；GDT 描述符的定义在 start.S 末尾位置，用 SEG_ASM() 定义。地址转换时，通过段寄存器选择 GDT 描述符，根据 GDT 描述符的基地址 base 加偏移量 (逻辑地址) 转化为物理地址。

6. 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 的作用是开启 gcc 的所有警告，更容易发现错误并规范代码。-Werror 则是将所有警告当做错误报告，这样强制我们修改不规范的地方，避免出现虽然编译通过但是结果不对的错误。-MD 的作用是生成给 makefile 用的 *.d 文件。

7.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。

首先 patsubst 将所有的 OBJS 中的.o 替换为.d，即使用的是和 OBJS 中.o 文件同名的.d 文件。而 -include 的意思是包含某个代码, 意味着编译某个文档, 需要另一个文档的时候, 能够用另一个文档中的设定。相当于 C 语言中的 #include。即在编译.c 和.S 文件的时候，可以使用 OBJS 中所有同名的.d 文件的设定。

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

首先用 find 命令找到所有 src 下的.c 和.S 文件。然后使用 gcc 进行编译，ld 进行链接。其中的参数指明要将函数 game_init 作为入口点开始执行程序，代码段开始地址为 0x00100000，并设置输出文件名为 game。然后进入 boot 目录下进行 make。将 start.S 和 main.c 进行 gcc 编译。然后用 ld 将二进制文件 link。输出文件为 bootblock，并且调用 perl 脚本，检查大小 boot 的大小是否超过 510 字节，同时在最后两个字节分别加上 55、AA。最后将 bootblock 和 game 一起拼成 game.img。