操作系统方案二设计文档

——XV6 系统 GUI 的重新实现

耿正霖	Geng Zhenglin	2012013295
王宇炜	Wang Yuwei	2012013301
袁扬	Yuan Yang	2012013335
曾华	Zeng hua	2012013324
刘桐彤	Liu Tongtong	2012013331

目录

1 实验概述

本次实验的目的是根据一字班做的 GUI、Shell、音频播放这几个优秀作业，进行内核的升级和功能的整合。在评估一字班原有的代码之后，我们认为原先 GUI 中存在以下两个问题

1. 硬编码的显存地址

在一字班的 GUI 实现中，通过某种途径查到了 Ubuntu 10.04 下显存地址为 0xE0000000,之后在 gui.c 中，他们就直接往改地址写。这种不合理的硬编码方式导致该 GUI 无法适应不同版本的操作系统，代码的移植性差。

2. 内核态的 GUI

最新版（第八版）xv6 对内存机制进行了较大的更改，不同于第六版，kernel 部分的内存大小被严格限制为 4M。一字班的 GUI 实现是通过将所有图片内容转成数组保存在.h 文件中，然后将所有代码编译进操作系统内，即保存在 kernel 当中。这种方法仅仅是实现了一层外壳，是一种不合理的 GUI 实现。

前期在解决一字班 GUI 的问题时花费了很长的时间，但是在探索的同时也为后面 GUI 的设计提供了许多参考。作为第一个探坑者，我们也为其他 GUI 组提供了一些起步的建议，提供了一定的帮助。

由于移植非常困难，所以我们决定在最新版 xv6 上，重新设计 GUI 架构，使其能适应不同平台不同系统，并加以实现。

在开发过程中，为了提高团队开发的效率，我们使用 Git 进行代码版本的管理，项目托管在 GitHub 上，项目地址为 https://github.com/xv6Group/new_xv6。

2 实验环境

在开发过程中，我们主要使用 Ubuntu 14.04 系统进行开发，在测试过程中，在 Ubuntu 10.04、Ubuntu 12.04、Ubuntu 14.04 和 Windows 的 Qemu 上进行了测试。

3 系统架构

我们实现的 GUI 系统基本可以分为三层。其中最底层的是和硬件直接相关的键盘、鼠标驱动和显存绘制。中间层的是消息管理器和窗口管理器。这两层都运行在内核态，最上层的是运行在用户态的 GUI 库函数和各种应用程序。

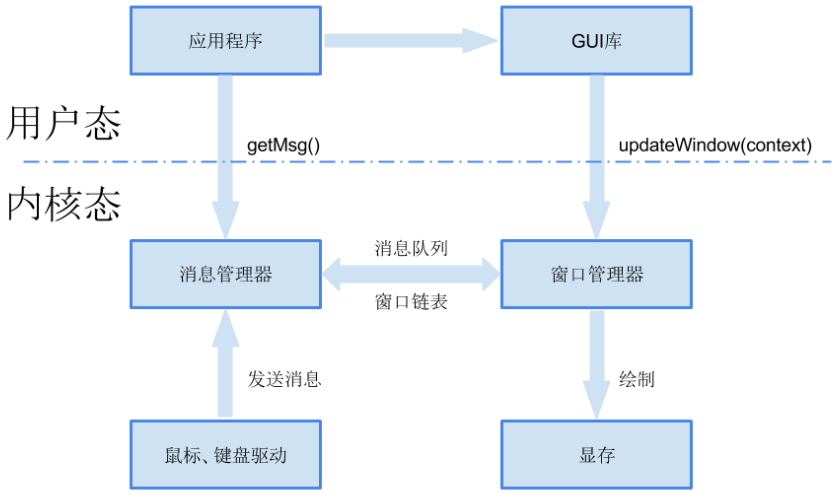

图 1 GUI 系统架构

键盘鼠标驱动程序由中断处理程序调用，它们会将中断事件封装成统一的消息数据，传递给消息管理器。消息管理器再根据窗口位置信息将消息分发给相应的进程。应用程序获取到消息之后，会调用 GUI 库函数。这些库函数都是对绘制窗口的系统调用的封装。窗口管理器响应这些调用之后，就会在显存中绘制出正确的界面。在我们实现的 GUI 中，所有带有图形界面的应用程序都需要通过在主函数中不断循环接受消息来响应键盘、鼠标和更新窗口的事件。此外，同时可能会有多个窗口打开。因此，我们在内核中编写了消息管理器和窗口管理器来管理系统中所有的消息和接口。图 1 展示了我们的系统架构。

4 模块实现

4.1 显存位置判断

通过设置相关寄存器中不同的值，在触发 VESA 相应终端时可以达到不一样的效果。具体如图 2 所示（bootasm.S 文件）：

```
39  # Get VEGA mode
40  movw    $0x1000, %di
41  movw    $0x4f01, %ax
42  movw    $0x4114, %cx
43  int     $0x10
44
45  # Set VEGA mode
46  movw    $0x4f02, %ax
47  movw    $0x4114, %bx
48  int     $0x10
```

图 2 VESA 的触发设置

在 40-43 行，在%ax 中存储 0x4f01，触发 0x10 号中断，这时在%di 所表示的位置会储存%cx 中的值所指示的 VESA 模式的信息结构体。

在 46-48 行，在%ax 中存储 0x4f02，触发 0x10 号中断，这时会启动%bx 中的值所指示的 VESA 模式。

本题中 0x4114 指示的是分辨率为 800*600，颜色模式为 16 位（5:6:5）的 VESA 模式。

执行上述汇编语句后，即可获得保存 VESA 信息的结构体。在 vesamode.c 文件中实现了 vesamodeinit()函数，可以获取显示模式的显存地址、长度、宽度等信息。在 main.c 中系统启动时调用该函数即可获取显存地址后再进行界面显示。

具体实现见 bootasm.S，vesamode.h，vesamode.c，main.c 文件。

4.2 键盘、鼠标驱动

4.2.1 键盘驱动

键盘驱动的执行过程中，需要判断输入是否为一个有用按键，具体执行过程如下所示。

1. 假设当前按下的字符为 ch,对于 ch 的不同和当前 flag 的不同做出如下反应。
   a) 若 shiftcode[ch] != NO,将 flag_shift 赋值为 shiftcode[ch],返回。
   b) 若 togglecode[ch] == CAPSLOCK,将 flag_caps 赋值为 1,返回。
   c) 根据 flag_caps 是否被标记,flag_shift 是否为 SHIFT,可以判断当前按下的字符的大小写。
2. 假设当前松开的字符为 ch,对于 ch 的不同标记 flag。
   a) 若 shiftcode[ch] == flag_shift,将 flag_shift 赋值为 NO,返回。
   b) 若 togglecode[ch] == CAPSLOCK,将 flag_caps 赋值为 0,返回。

4.2.2 鼠标驱动

4.2.2.1 PS/2 协议

标准的 PS/2 协议数据格式为 3 字节，其主要内容如表格 1 所示。

表格 1 PS/2 协议数据内容

y_overflow	x_overflow	y_sign	x_sign	1	middle_press	left_press	right_press
x_displacement							
y_displacement							

其中第一个字节记录了鼠标有是否越界，鼠标位移方向，以及左键、右键、中键的情况。第二个字节记录了鼠标在 x 轴方向的位移，第三个字节记录了鼠标在 y 轴方向的位移。

4.2.2.2 处理流程

鼠标中断程序的处理流程如图 3 所示。

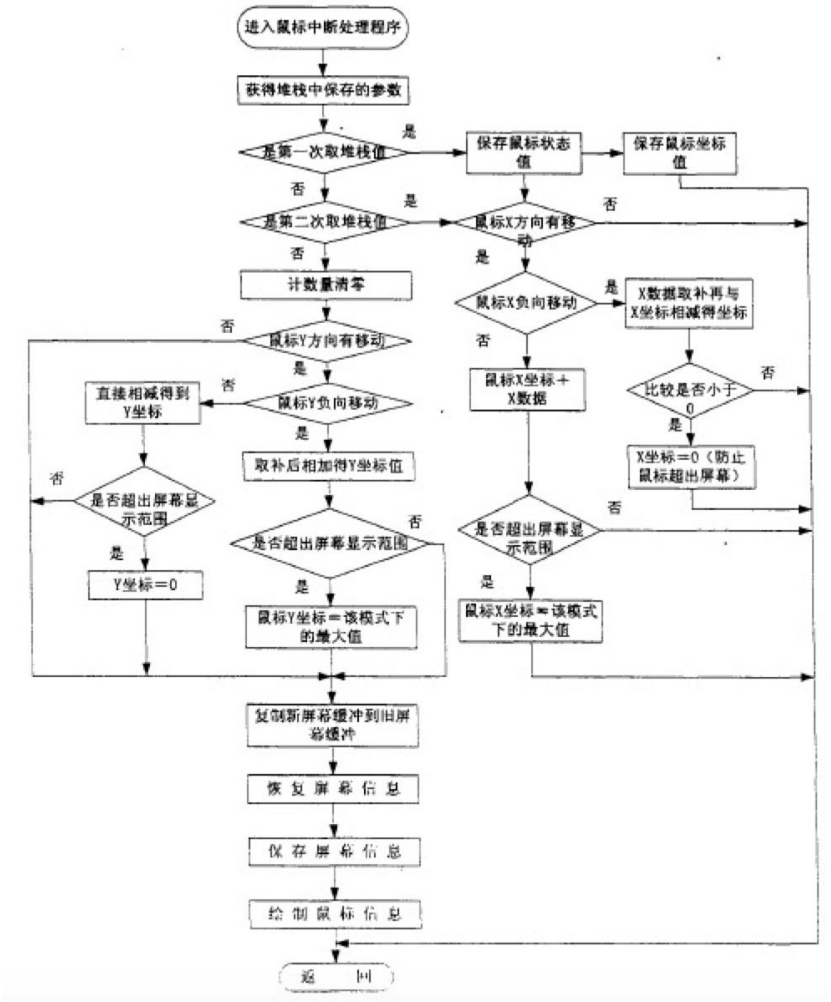

图 3 鼠标中断事件的判断流程

每次发生鼠标中断时,读取鼠标数据。按顺序依次为第一字节、第二字节和第三字节。用一个状态 state 记录读取的阶段。

数据读取完成后，进行鼠标事件的判断，然后创建鼠标消息，由系统分发给各个进程，同时完成鼠标的绘制。

4.2.2.3 异常处理

在实际操作中，有时发生鼠标中断会读取错位或者位移量溢出的异常。

读取错位即读取鼠标数据时顺序会发生错误。此时第一字节的第 4 位始终为 1 是一个重要的判断。其次由于我们只使用鼠标左键和右键，第一字节的第 3 位始终为 0 也可用于判断第一字节。

位移量溢出即实际操作中，鼠标移动速度过快，导致鼠标中断产生的位移数据出现数据溢出。这种情况下，会在短时间内产生多次中断，每次中断中位移的数据量都为 127。因此可以根据这一点，将这多次中断的位移进行累加。最后只创建经过累加的鼠标消息。

通过处理异常，在实际运行中，鼠标可以流畅地操作，不会出现类似于闪动的异常情况。

4.3 消息管理

在内核中，系统维护一个进程消息，每个进程都对应着一个消息队列。每次一个新的鼠标和键盘中断产生的时候。驱动程序都会调用消息管理器的一个函数，将中断事件类型和相应的数据封装成一个消息。然后，消息管理器会根据窗口位置等信息，判断出这个消息应该发送给哪个进程，再将这个消息添加到这个进程的消息队列中。每当进程调用 getMsg()系统调用来获取最新的一个消息。如果队列不空，那么消息管理器需要将这个消息返回给进程，并将这个消息从队列中删除。如果队列为空，那么消息管理器会返回一个代表没有新消息的消息。

进程可以根据返回的消息采取相应的操作。但是操作系统并不会主动调用进程的函数。

4.4 窗口管理

在图形界面中可能同时会出现多个窗口，因此需要一个窗口管理器来管理所有窗口的绘制。窗口管理器通过维护一个窗口列表来实现。列表的具体实现为一个双向链表。链表的顺序反映窗口的层次关系，最底层的窗口位于链表首。在本系统中，最底层的窗口始终是系统初始化的过程中创建的第一个图形界面进程——桌面进程，如图 4 所示。最顶层的窗口是当前的活跃窗口，在响应事件的过程中需要通过访问窗口链表将消息发送给适当的进程。

图 4 桌面

因为内核内存限制，我们不能在内核中存储所有窗口的图像信息，所有窗口管理器需要的数据都需要进程利用 updateWindow 系统调用主动传递给内核。我们将消息与窗口结合，利用系统调用实现了复杂的绘制工作。例如，每当新建一个窗口的时候，都需要在窗口列表中创建一个新

窗口，生成一个窗口更新消息发送给进程。进程收到消息之后利用系统调用将图像信息传递给内核。最后再由窗口管理器绘制这个窗口。当发生窗口切换或者拖动的时候，我们需要重绘全屏，方法是通知底层窗口更新。更新后再逐步通知上层窗口。为了防止绘制过程出错，我们借助了一个屏幕图像缓存，先将所有图形绘制在缓存区中，绘制完毕后再复制到显存中。

4.5 GUI 库

内核提供的系统调用仅仅能够将一个矩形的区域在屏幕上绘制出来。为了能够绘制字符、线条、图片等更复杂的元素，我们在用户态编写了一个 GUI 的库，用来绘制图形。在使用绘图函数前，需要先使用相关的函数在内存中申请一个数组作为绘图上下文，然后将每一像素的颜色值保存在数组中。数组中的每一项是一个 16 位的整数，按照 5:6:5 的分配方式分别记录像素点对应的 RGB 颜色值。将数组作为参数传递给函数 updateWindow，窗口管理器就会将这片区域写入显存。

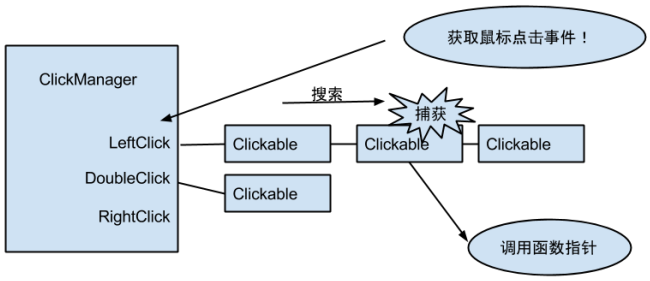

图 5 Clickable 的数据结构

在绘图部分，我们提供了绘制窗口标题栏、绘制点、绘制字符、绘制图片等功能。同时，为了处理较为复杂的鼠标点击事件，我们设计了 Clickable 结构体用于托管鼠标点击事件。具体原理如图 5 所示。Clickable 包含一个矩形区域和一个函数指针。用于处理鼠标点击事件。ClickManager 包含三个指向 Clickable 的链表，用于托管进程的鼠标点击事件。Clickable 的具体使用方式如图 6 所示，需要先绘制一个按钮图片，然后在相同的位置添加一个 Clickable 结构作为鼠标点击事件相应区域。

图 6 Clickable 的具体使用方法

由于 GUI 库函数较多，在正文中不再赘述，在附录中有一份函数的列表，以供参考。

4.6 应用程序

我们在用户态实现了两个较为复杂的应用程序文件查看器和 Shell 作为示例。将来还可以利用我们的 GUI 库实现更多的功能。它们的具体功能和结构如下所示。

4.6.1 Finder

4.6.1.1 基本介绍

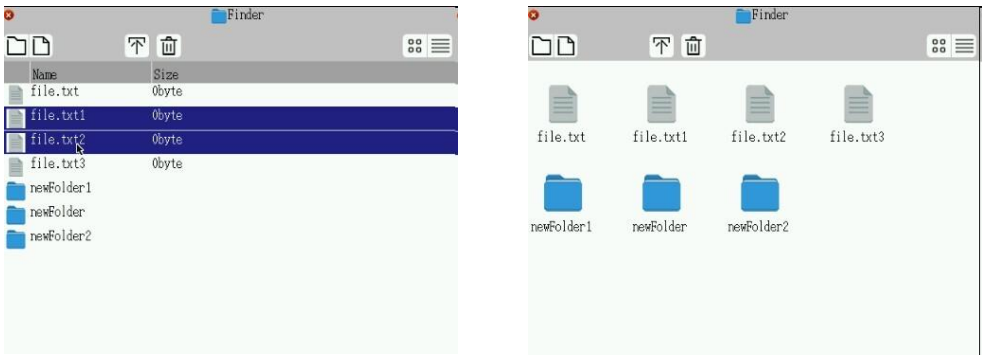

图 7 文件查看器

对于一个操作系统，基于文件的操作自然是必不可少的。因此我们在 xv6 的 GUI 中实现了类似于 Mac OS X 系统中的 Finder 的文件资源管理器。

我们的 Finder 实现了文件系统的各种基本操作，如新建文件夹，新建文件，进入目录，回到上级目录，删除文件或文件夹。同时支持两种布局模式，图标布局模式和列表布局模式。

4.6.1.2 实现细节

- 窗口绘制

  Finder 窗口用我们实现的 GUI 库来绘制。

  Finder 窗口被分成两部分，窗体和内容。窗体包含顶栏，工具栏，用于提供各种操作的按钮。内容包含目录中的所有文件，文件根据选定的布局模式进行布局。

- 点击事件

  实现了 clickable 的类，用于实现 Finder 中点击事件。每打开 Finder 进程，就创建一个 clickableManager，用于管理点击事件，其中包含三个事件队列，分别对应单击左键，单击右键，双击右键三类事件，队列存储事件的响应函数，每个响应函数都对应一个矩形区域，表示响应这个区域的事件。

  每当收到系统分发的鼠标消息，就在对应的事件队列中查找对应坐标所对应的响应函数，处理该函数，然后重绘

- 文件链表

每进入一个目录，就使用 ls 指令获取当前目录的所有文件。用链表管理这些文件，同时记录一些附加信息，如所对应的 clickable 响应事件区域，绘制区域，是否被选中等。每次重绘 Finder 都根据当前文件链表完成重绘。

4.6.2 Shell GUI

4.6.2.1 基本介绍

在操作系统中，很多应用程序并没有图形界面，我们需要在图形界面中提供一个 Shell 来执行这些程序。同时，一字班的 Shell 中提供了很多新的命令，我们无法为这些命令都写一个 GUI 的版本，因此，我们在 GUI 中实现了一个 Shell，用于执行原本的 Shell 命令

4.6.2.2 实现细节

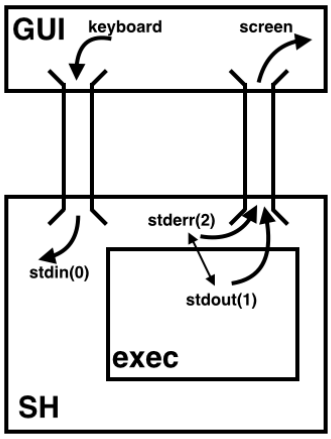

图 8 Shell 架构

如图 8 所示，Shell 架构由两部分构成，前端 GUI 部分（shell_gui.c 文件）和后台（shell_sh.c 文件）。前端和后台的交互通过管道实现。具体来说，前端通过管道连接后端的标准输入流(0)，标准输出流(1)和标准错误流(2)。前端接收键盘的输入，在屏幕显示并写入管道，后端从管道中读取键盘输入，执行相应的命令，将执行结果和提示内容写入另一管道，前端再从管道中读取并在前端显示出来。

下面通过一条命令的执行流程来解释。

1. 正确命令——执行 ls：
2. GUI 从 keyboard 读取 ls 并在屏幕显示
3. 用户按下回车
4. GUI 将命令写入 PIPE
5. SH 从 stdin(0)读入 ls
6. SH fork 出一个子进程，将子进程的 stdout(1)与另一 PIPE 的一端关联
7. 子进程执行 ls，将输出结构写入 stdout(1)

8. 子进程结束，SH 再将自己的 stderr(2)与 PIPE 一端关联，写入“$ ”
9. GUI 从 PIPE 中读取并在屏幕显示，等待下一条指令。

GUI 实现时需要对退格键进行特殊处理。当接收到退格事件时，如果已有字符正在输入，需要删除最新输入的一个字符。

具体实现见 shell_gui.c 和 shell_sh.c 文件。

5 不足之处和未来改进方向

- 在本程序中，窗口管理器被写入了内核中，但是根据现代操作系统的普遍做法（Windows、Linux 等），窗口管理器应该被实现成一个用户进程。每当一个新的消息产生的时候，应该由窗口管理器分发给相应的窗口。但是，现在 xv6 中进程间通信的方法仅有 pipe 一种，为了将窗口管理器实现在用户态中，需要实现共享内存等更高级的进程通信方法来支持进程间传递更大的数据。
- 我们的系统中，每次重绘屏幕都需要先向桌面发送一个消息，收到桌面更新数据之后再通知上层窗口更新。但是，我们无法要求每个应用程序都在响应 update 消息之后调用 updateWindow 更新。所以，更合理的做法是将所有窗口数据保存在一个窗口管理器中，重绘的操作也可以更简单一些。但是，这样需要为窗口管理器分配更多内存，或者将窗口管理器实现在用户态中。
- 目前实现的 GUI 库比较简单，依靠 Clickable 来进行鼠标事件的管理仍然是一个很不优美的实现方案。将来的扩展中，可以添加诸如 buttonView，IconView 一类的窗口控件。如果有能力的话，最好用面向对象的方法实现 GUI 库。
- 我们的系统中，使用窗口链表在管理所有的窗口，但是，在实际应用中，窗口应该被组织成一棵树而不是一个链表。因为除了窗口之外，各种控件都可以被看做是一种窗口，由统一的窗口管理器管理起来，这样可以减小应用程序的编写难度。

6 人员分工

表格 2 人员分工

王宇炜	window.h/window.c
	desktop.c
袁扬	message.h/message.c
耿正霖	drawingAPI.h/drawingAPI.c
	bitmap.h/bitmap.c
	clickable.h/clickable.c

	finder.c(与曾华合作)
曾华	mouse.h/mouse.c
	finder.c（与耿正霖合作）
刘桐彤	Shell_gui.c
	Shell_sh.c（与 Shell 组合作）
	vesamode.h vesamode.c

7 附录：GUI 库函数

- #include “drawingAPI.h”
  - void draw_point(struct Context c, unsigned int x, unsigned int y, unsigned short color);
    在指定坐标（相对于窗口左上角顶点的相对坐标）绘制点，如果坐标在窗口范围之外（窗口信息保存在 Context 中），则不会绘制。
  - void fill_rect(struct Context c, unsigned int bx, unsigned int by, unsigned int width, unsigned int height, unsigned short color)
    以 bx, by 为顶点，width 为宽度，height 为宽度绘制矩形。注意这里都是闭区间。
  - void puts_str(struct Context c, char *str, unsigned short colorNum, int x, int y)
    以 x,y 为顶点，绘制字符串。
  - void draw_picture(Context c, struct PicNode pic, int x, int y)
    绘制在 PICNODE 已加载好的图片。
  - void draw_line(Context c, int x0, int y0, int x1, int y1, unsigned short color)
    起点 x0,y0，终点 x1,y1 绘制直线。
  - void draw_window(Context c, char *title)
    绘制一个有标题有关闭按钮的窗口。
  - void initializeASCII()
    加载 ASCII 码点阵字符文件。
  - void initializeGBK()
    加载 GBK 点阵字符文件。
  - void freeASCII()
    释放 ASCII 点阵字符文件。
  - void freeGBK()
    加载 GBK 点阵字符文件。
  - void loadBitmap(PICNODE *pic, char pic_name[])
    将 bmp 格式图片从文件系统加载到内存中，在内存中保存的数据结构为 PICNODE。

- #include “clickable.h”
  - void createClickable(ClickableManager *c, Rect r, int MsgType, Handler h)
    MsgType 单击，双击还是右键。
  - void addClickable(Clickable **head, Rect r, Handler h)
    添加一个区域为 r,处理事件为 h 的 clickable,添加到链表头部。
  - void deleteClickable(Clickable **head, Rect region)
    删除起始坐标在 region 内部的所有 Clickable。
  - int executeHandler(Clickable *head, Point click)
    执行点击坐标在 area 内部的 clickable 函数。