操作系统实验报告

基于 xv6 用户图形界面(GUI)的实现

<开发者文档>

BingoLingo (包煜 王安琪 赵雷彧 钟仰新)
2015/1/23

For source code please clone git repository:
git@github.com:publicres/xv6_gui.git

目录

For source code please clone git repository:
git@github.com:publicres/xv6_gui.git

实验背景

- 实验平台(xv6)

xv6(Unix Version 6)为操作系统实验用 Sample OS，为对 unix 设计的重写，代码量小，但具备操作系统所应该具有的几乎全部功能。

本次实验将在该系统上进行。

- 实验目的

本次试验 BingoLingo 组的目的在于给已有的 xv6 操作系统增加图形用户界面(GUI)。这种增加并不止于能在内核态根据 OS 开发者意图更新显存并获得界面，更在于给操作系统增加一整套开发框架以供用户态应用程序使用，从而在用户态开发图形化应用程序。

同时操作系统应该能在不同图形化进程间进行调度、切换。这才是一个真正图形化操作系统所应有的：良好的扩展性以及平台框架的弱耦合性。

- 现行平台下的挑战

在制作前/制作中，我们发现在现行 xv6 实验平台上有诸多局限，对于 GUI 的实现都是相当大的障碍：

1. 缓慢的运行速度。qemu 在虚拟机下运行虚拟机，并用单核模拟多核，性能必定大打折扣。
2. 受限的系统堆管理。系统堆的分配每次只能得到一页，如果需要大空间则不能保证其连续性。
3. 硬盘的严重限制。由于文件系统原因，能存在硬盘上的数据(程序、图片等)均不能超过 72kb，同时硬盘总容量不能超过 512kb，这是对 GUI 编写的最大瓶颈。
4. 内核栈/用户栈的限制。Xv6 的设计中，进程的内核栈以及用户栈均只有一页(4kb)，对于图形渲染、消息传递等显然是不够的。

上述限制的解决方式将在下文详述。

特别提醒

由于采用的是 COM 口鼠标，而鼠标和键盘都有同一个控制器 i8042 来控制，所以在实际运行的时候发现鼠标在以下两种情况下会产生乱飞现象：

1.如果你是在虚拟机环境下（如 VMware 的 Ubuntu）运行 QEMU 虚拟机，那么很可能鼠标会不断乱飞，这时不仅鼠标发送的数据包是错位的、有缺失的，而且其数据也是错误的，推测其原因可能是因为虚拟机环境下有一个虚拟鼠标，而 QEMU 下也有一个虚拟鼠标，这两个虚拟鼠标的信号干扰了鼠标向 QEMU 模拟的主机发送的数据包。

解决之道：不要虚拟机环境下运行 QEMU，可以在真机 Ubuntu 或真机 Windows 中运行 QEMU，这个问题就不会出现了。

2.**如果你在移动鼠标的同时按键盘**，那么键盘信号和鼠标信号会相互干扰：键盘信号不打紧，干扰过后还能正常使用，但是鼠标信号就极易错位，比如明明是第一位的状态字节往

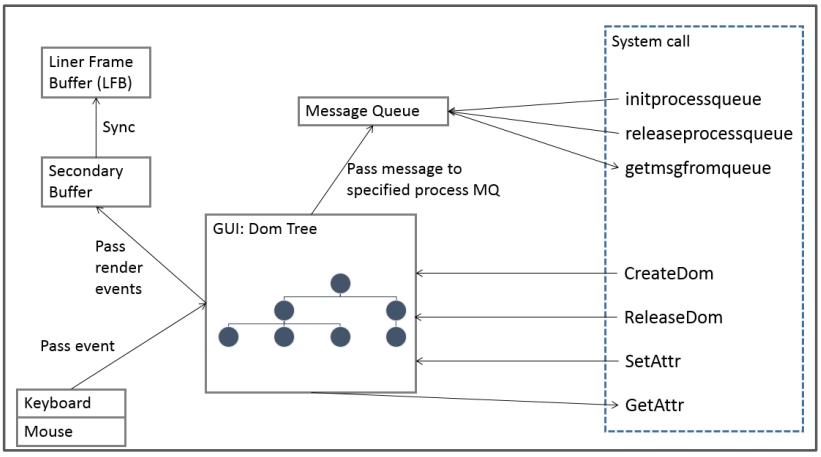
后延到了第二位的 x 位移字节，导致鼠标中断处理函数的逻辑错误，显示在 GUI 上就是鼠标乱飞的情况。

解决之道：一，不要再移动鼠标的同时按键盘！二，如果按了，可以选择再次再移动鼠标的同时按键盘，这样有一定的几率将错位用新的错位调整回来。三，我们为你制作了一套错误恢复机制，在这种情况下不要动鼠标，**直接按下鼠标滚轮**就能纠正错位的情况，恢复正常的鼠标移动。

3.并不是鼠标乱飞，而是如果你使用的是 Windows 版的 QEMU，那么其中的键盘特殊键位如上下左右方向键、delete 键等等，都会被自动转换成一些别的字符，比如键盘左键会直接转成'4'，键盘右键会直接转成'6'，delete 键会直接转成'.'等等。这样一来，你的这些特殊键位就不能正常工作了，比如我们的程序文本编辑器 fileEditor，它本来支持键盘左右键移动光标，但是如果在 Windows 版的 QEMU 中，按左右键就完全不能正常工作，而是会插入'4'或'6'这样的转换字符。

解决之道：linux 下 QEMU 暂时没发现过这个问题，可以在 linux 运行 QEMU。

架构及设计



- 总体结构图

总体结构图如左。其展现 GUI 实现中最大粒度下划分的模块以及模块间相互联系。关于每个模块的设计及介绍将在下文逐一解释。

- DOM 树

为方便事件的传递和画面渲染,在实现 GUI 的初期原创了树形结构来表示控件间关系。由于该设计来源于 HTML 的结构灵感，故将其定为 DOM 树。在内存中其表达形式为一棵多叉树转换而成的二叉树，即每个节点维护其第一个儿子和下一个兄弟。除了基本的树形结构域以外，每个 dom 节点还有 x,y(均为相对其父节点),w,h 等一系列与位置、事件函数指针、传递方式等相关的存储域。(详见 guilayout.c/h)

由于 c 语言没有继承的概念，故使用 struct 并将第一个域定为“父类”来伪造继承。从而由 dom 派生出 4 个控件，每一个拥有不同的附加属性、绘图行为等特点。

控件类型	控件作用	附加属性	控件绘画行为	其他信息
Div	普通矩形容器，用来构造 GUI 中最基础的框架性元	Bgcolor：单色背景，4 通道颜色(color32)	将背景色绘制	代码见guientity_div.c/h

	素。	isIntegral：是否作为一个整体接受事件	至要求的区域	
Img	图像控件，用于接受指定的图并显示。	Content：决定图片内容的数组(4 通道) isRepeat：是否平铺	将图片的一部分绘制	代码见guientity_img.c/h
Cha	字符控件，其本身不向用户开放接口，由 txt 控件统一管理	Content：显示于其中的字符	与图片相似	代码见guientity_cha.c/h
Txt	文本控件，管理文字排版、光标信息并控制文字输入删除等操作	(由于该部分逻辑相当复杂，下文将作为一个单独模块进行分析)		

在 DOM 树的结构保证了事件传递的可能性，鼠标、键盘、重绘、焦点等事件均在上面传递。按照传递方式(而非消息意义)不同，消息被分为以下三类：

消息类型	传递方式	具体例子
RenderEvent	按区域传递：即在(x,y,w,h)的矩形内，与其有交叠区域的任何 dom 均会收到该事件并指明交叠区域	绘图事件
PointEvent	按点传递：被(x,y)点中的最底层 dom 将会收到该消息(Integral 组件将会对其进行拦截)	鼠标单击、鼠标移动事件
FocusEvent	按焦点传递：当前焦点将会收到该消息(Integral 组件将会对其进行拦截)	获得焦点、失焦事件、键盘事件

DOM 树的设置给图像提供了层次组织模型并给事件以高效的传递形式。其实现给其他各模块的沟通提供了纽带作用。

值得注意的是，为了最大程度减少画面渲染量，每个渲染域在向兄弟节点传递时会“打碎”成若干块，从而最大程度避免重渲染。为了继续渲染具有透明度的图片、非规则形状等并不能遮挡住后面的 dom 之后的控件，dom 中设置了 transparent 属性以控制渲染行为。

- 消息&消息队列

消息即被外部事物触发、在 dom 树上传递并最终通知指定进程的一个结构。是利用 dom 树完成的从中断到用户态程序获知的传输媒介。

鼠标消息(按点传递)

鼠标消息是传递鼠标操作的媒介，下面就简要分析一下这个消息类型的成员变量：msg_type 是消息类型，由于各种不同类型的消息，如鼠标消息、键盘消息、焦点消息等会在同一个队列当中,所以各种不同类型的结构体的第一个变量统一是一个 int 类型的变量，用来告诉进程它拿到的消息是什么消息。dom_id 是接收鼠标事件的 dom 的 id；x 和 y 分别是鼠标事件发生的坐标值；mouse_event_type 表示发生鼠标事件的时候鼠标的各种按键的状态；enter_or_leave 表示当前是否有鼠标进入或离开了我们上述的 dom。

键盘消息(按焦点传递)

键盘消息中包含的内容是：包含消息类型的标志位，接收键盘信息的 dom 的 id，还有刚才按下的键盘的按键的键值

焦点消息

焦点消息的传递是当有 dom 获得或是失去焦点的标志。具体的成员变量有：表示消息类型的标志位，接收消息的 dom id，当前 dom 是获得了焦点还是失去了焦点。

启动程序消息

这类消息一般都是直接发给 indexpage 进程的，它的作用是调用系统根目录下，一个名字叫做 call_process_name 的程序去打开一个路径为 file_path 的文件。这类消息一般是在其他进程需要用一个程序打开一个文件的时候用于通知 home 进程来让它自己启动我们想要启动的进程。这么做的原因是：在 home 进程中维护着一个用于进程切换的东西，但是它的维护仅限于那些由它自己打开的进程，所以，其他进程打开的其他进程是不会被维护的，所以当我们需要打开文件的时候，就需要通过向任务栏进程传递消息来通知。

另外，本次我们针对每一个进程维护一个消息队列。当然，在创建用户态程序时，如果不需要接受事件作出响应，那么我们也可以选择不初始化队列，这样就可以允许一个进程没有消息队列。下面就简单说一下消息队列的实现：

消息队列的结构

首先，当操作系统启动的时候，会有一个总的 process map 初始化，它其实是一个连续的数组空间，数组最大长度是 1024，也就是说，不支持 1024 个以上的进程同时运行，这对于我们这个操作系统来说，已经是非常够用了。这个数组的每一个元素都是一个指向消息队列结构体的指针；消息队列的结构体如下：

```
typedef struct message_queue
{
    int head;
    int tail;
    void** queue[QUEUE_SIZE];
} MsgQueue;
```

这采用的是一种循环队列的模式。这每一个进程对应一个上述消息队列结构体，由于 xv6 在启动程序的过程当中 pid 是从 1 开始依次向上每次加一，所以，进程和消息队列的一一映射是通过进程的 pid 做一个 hash，直接就映射到了相应的队列。

消息队列的入队

其实不同事件的入队机制有些不同，不过这些入队的机制都是可以通过重写一些函数，比如 onPoint， onFocus 等。下面先简单挑几种消息，看一下他们的入队方式，最后再看一下入队这个操作都做了哪些事情。

先比如说鼠标事件的进队，默认的进队方式是：我先看一下我当前的鼠标操作是否导致了鼠标离开或是进入了某个 dom，若是，那么我们会给被进入的那个 dom 的进程发送鼠标进入消息，并且给被离开的那个 dom 的进程发送鼠标离开的消息；若否，那么我就看一下当前的鼠标是否有键被按了，若是，那么我们会给相应的进程传相应的鼠标操作消息，若否，我

们就不会给任何进程传递消息。这样就不会把一些并没有操作而仅仅是普通鼠标移动的消息传给进程了。

键盘事件和焦点事件和鼠标事件有点类似，且它们三个的入队操作都是在内核态进行的，都比较方便。这里就不再赘述了。

而对于启动程序消息的入队，它是在用户态的，所以这里需要加一个系统调用，并在系统调用的时候把东西入队。

最后是看一下，进队操作干了啥：进队干了两件事，一个是循环队列的基本操作，另一个是看一下是否有正在等待该资源而被阻塞的进程，如果有，那么我们就唤醒它。

消息队列的出队

出队其实就是用于进程向自己所对应的消息队列要消息。它做的事情是：向消息队列要队首的消息，如果消息队列不为空，那么我就拿到队首并判断他是啥类型的消息，并根据不同的类型做一下转换赋给结果指针并返回。若消息队列为空，那么进程进入阻塞状态。

此外，在用户态进程中要获取消息队列中的东西同样需要添加系统调用。

- 多进程 GUI 下进程调度

由于任务栏进程以及其余 APP 本质上均为用户态应用程序，故在管理中采用与 sh 相同的总管理者、子进程管理模型。即 init 进程 exec 进入 indexpage(即任务栏进程)，由 indexpage 启动其他进程，监视器运行状态并在结束后释放相关资源。

当用户单击开始屏幕的某个 APP 时，将由 indexpage 着手利用 fork+exec 启动该程序，并为其建立任务栏按钮以及该进程的始祖 div。而后将该 div 的 id 作为参数传递给 exec 的进程并令其在上面绘画。当非阻塞等待系统调用通知 indexpage 有进程结束时，indexpage 将释放始祖 div、删除并释放任务栏按钮。

唯一的问题在于如果其他用户程序自身 exec 了其他进程，则 indexpage 不能获知从而不能对该进程进行跟踪。实践中我们提供了系统调用将打开文件消息传递给任务栏进程，从而解决了该问题。具体实现见下文“其他难点及解决方案”。

- 鼠标驱动

鼠标驱动采用的是 PS/2 协议，允许鼠标中断要在 xv6 初始化时调用鼠标初始化函数，此函数要先后执行以下的几个步骤：

1.向 0x64 端口发送值为 0xA8 的字节，允许鼠标操作。（鼠标是由键盘控制器(i8042)进行控制的，0x64 是其命令端口，0x60 是数据端口）

2.向 0x64 端口发送 0xD4，告诉鼠标程序准备往 0x60 写数据，要求它发给鼠标。

3.向 0x60 端口发送 0xF4，允许已经通过 BIOS 自检、先已处于 Stream 模式下的鼠标给主机主动发送数据包。然后马上从 0x60 端口读入对应的 ACK 字节。

4.向 0x64 端口发送 0x60，告诉鼠标下面向 0x60 发送的数据请写入 i8042 控制寄存器中。

5.向 0x60 端口发送 0x47，设置 i8042 的控制状态，这个数据表示允许鼠标以及键盘发送中断，并使鼠标采用 PS/2 协议，而非 AT 协议。

过后如果用户移动或点击了鼠标，鼠标就会触发相应的中断，并主动将含有鼠标点击和

移动信息的数据包一字节一字节地发送到 0x60 端口。处理这个数据包的函数是一个鼠标中断函数，那如何添加相应的鼠标中断函数呢？首先，鼠标触发的中断信号是 12，这需要在 traps.h 中定义一个 IRQ_MOUSE 的宏定义，然后在 traps.c 的 trap 函数中完成鼠标中断处理函数的注册。具体方式可以参考键盘处理函数 kbdintr 的注册方法。

注册过后，每有鼠标中断，就会调用鼠标中断处理函数(mouseintr)。默认状态下鼠标使用的是二维模式（没有滚轮位移信息），每次鼠标发送到 0x60 的数据包都是 3 个。第一个是鼠标事件的状态字节，第二个是鼠标 x 位移的绝对值，第三个是鼠标 y 位移的绝对值字节。而状态字节的组成结构如下：

左键是否被按下	右键是否被按下	滚轮是否被按下	总为 1	x 位移的符号位	y 位移的符号位	x 位移是否溢出	y 位移是否溢出

低位->高位

鼠标中断处理函数的工作，就是通过位运算处理状态字节，并与后两个数据字节结合起来，获取鼠标哪个按键被按下、怎么移动。需要注意的是，鼠标 0 坐标从左下角开始，向右 x 增长，向上 y 增长。得到鼠标点击信息和位移信息后，只需将其封装成为鼠标消息，然后通过消息队列和 dom 树搜索分发给合适的进程即可。

至此，鼠标驱动简要介绍说明结束。

- TXT 文字排版引擎

在 guientity_cha 和 guientity_txt 的源文件和头文件中实现了两个 dom 的“伪”派生结构体，每个 cha 是一个字母，每个 txt 是一堆 cha 的组织，它们俩一起构成了字符潘班引擎的核心。

总体设计架构上是这样的：每一个 txt 节点是一个透明的 dom 节点，它相当于一些 cha 的容器，而从 dom 树上看，每个 cha 都是 txt 的儿子，而且字符串越靠后的字母越靠左，也就是说 txt 的第一个儿子是字符串最后一个字母的 chaDom；另外，txt 的最右边的儿子是一个 div，它表示这个 txt 的光标。而 txt 结构体内部用一个 head 指针指向它所有 cha 儿子的链表，这次，链表的顺序就是字符串的顺序；另外还有一些指针，cursorDiv 指向光标的 div，cursor 指向光标聚焦的 cha，也即 backspace 会删除的那个 cha，至于 blockHead、blockTail 以及 tail 是用于 cha 的内存的申请与释放的，由于内核态一次只能申请 4k 的堆空间，而我们一个 txt 可能会有成百上千或更多的字母，我们不能每个 cha 都占 4k，只能将 4k 合理地分配给每个 cha 以节约空间。所以说 cha 的内存申请和释放不由它来维护，而是由它父亲 txt 来维护，可以说 cha 不能脱离 txt 这个容器而单独存在。

其中内存的申请和分配策略是将 4k、4k 的堆内存的头四字节用作指针，相连起来，每次申请内存就从尾指针(tail)那直接拿，不够就再申请，然后相连，而若要从 txt 中删除一个 cha 则是将 cha 剥离 dom 树但不马上释放其内存，而是等待 txt 释放时再统一 4k、4k 地释放；而 blockHead 指向的是第一个 4k 的起始地址，用于释放内存时从头遍历；blockTail 指向的是现在 txt 已申请的最后一个 4k 的起始地址，它配合 tail 可以判断最后一个 4k 内存是否不够用，是否要继续申请 4k 内存。

由于有光标的存在，而且还需要完成移动光标的系统调用，那么我么就得涉及左移光标和右移光标，但是因为 dom 只会记录它的右兄弟，所以在 dom 树上光标左移（找一个 cha 的左兄弟）将变得很慢，于是我们又设计了一个结构体 txtContent 来封装 cha，只是多给了它一个 next 和一个 prev 指针，用于辅助维护 dom 树。而实际上存储在一个 txt 申请的堆空间中的不是 cha，而是这个封装过的结构体 txtContent。前面所说的 head、tail、blockHead、blockTail 以及 cursor 指针都是 txtContent*。说到光标，如果一个 txt 的 cursor 指向(uint)-1 的地址，代表这个 txt 目前隐藏光标；如果指向的是 0 地址，代表这个 txt 的光标目前可见是在第一个字母上。

另外，由于字符是一个比较常用的东西，所以每个字符对应的图片（也即字体图片）应当被存在内核态的空间中；然后就像 img 控件一样，cha 也会有一个指针指向一片内存，里面存的是那个 cha 的字体图片，由于一个字母如 a 会被“实例化”为多个 cha，所以我们只需要把如 a 这样的字符的字体图片存在一个地方，让不同的实例 cha 指向这片内存就可以完成关联，进行 dom 的相关渲染。

事实上，这些字体图片都存在 guientity_txt.c 中的 fontArray 数组每个元素指向的堆空间里，实际上我们的系统是可以支持多套字体的，具体详情参考这个部分的定义，不再赘述。事实上，我们需要在 xv6 系统启动的时候，从磁盘中读取各个字符的字体图片文件到内核态的这个空间里永久存储着，具体函数请看同一源文件的 initFont 函数，它与 txt_initLock 函数在 main.c 的 main 中以及 init.c 的 initLetters 中配合使用，可以保证字符的字体图片被系统刚启动时就初始化好。每一个 txt 还有三个成员记录其字符(cha)的宽、高，以及用的是哪套字体（指针指向前文提及的字体图片空间）。

txt 这个 dom 由于可以被广泛应用到用户态的需要文字编辑类的程序中，所以需要实现比较多的系统调用，除了常规的调整坐标和宽高值，还要完成设置 txt 字体、字符串，以及关键的左移、右移、随机设置光标调用，以及在光标处插入字符、backspace 字符、delete 字符等等的调用。这些调用实现起来都相当具有难度，因为涉及了 dom 的创建、更新与删除与重绘，以及 txt 的指针、链表的各种维护（一旦涉及链表就有各种边界情况要考虑）。具体的实现细节请参考 guientity_txt.c 文件。

总体来说，因为 txt 这个 dom 的系统调用实现得比较全面，所以用户程序任何需要文字或文字编辑的地方都只要申请一个 txt，并且用好上面提及的系统调用，就能轻轻松松地完成文字显示和编辑的控制流程，例如我们实现的用户态程序：文本编辑器 fileEditor。这就是底层 API 的重要性！

- GUI 风格设计简介

鉴于本次图形界面只能通过矩形 div 块绘制，使用的图片大小仅限于 128*128，于是采用 win8 的 metro 风格进行设计。用户启动 xv6 操作系统后经过启动页面，进入主界面。在主界面即可通过选择程序从而启动进程，也可通过左侧面的进程切换窗口进行进程间切换，当鼠标移向视窗左上角时，进程切换窗口即出现。当用户正在使用某一程序时，程序界面将占据整个视窗，能通过进程切换窗口进行进程切换或者返回到主界面。当关闭该程序时视窗将回到主界面。

实现难点

绘制元素局限于矩形和小图片，通过简单的元素构建较为复杂美观的界面，这是主要的难点。

解决方案

借助 PS 和收集到的素材，发挥主观能动性，将复杂的界面分解为简单的几何图形，通过矩形实现。

其他难点及解决方案

- Vesa 编程基础以及显示模式选取

由于上一届优秀作业的 GUI 代码实在过于不可取，所以不像很多其他选题小组，我们选择了完全不参考以往代码、从头开荒。面对完全不知从何下手，我们阅读了大量资料并很大程度参考了 vesa 的 vbe 规范文档，最终确定了规范而合理的写法。

在显示模式选取上，尽管我们想尽可能追求大分辨率，但考虑到调试用机器均为 1366*768 的笔记本电脑，故最终选取了 1024*768 作为分辨率以方便开发过程中的调试、体验。在分辨率基础上我们选取了最高色数:真彩色，力求最逼近真实渲染效果。

另外，由于显示模式设置以及相关参数获取均为动态的，应用层和渲染层耦合度也相当低，故只需修改 bootasm.S 以及 graphbase.c/h 中的相关参数即可轻松修改显示模式。

- 内存优化-解决内核栈/堆的局限

在用户态，由于页表的存在，用户态进程可以自由申请使用连续空间。然而在 xv6 的设计中，内核态代码的页表事先已经写好，通过 kmalloc/kfree 统一管理，一次性只能无条件获取 4kb 内存空间，无法获取连续堆空间，故不能直接对其进行访问。对此，编写了大内存管理模块(详见 ex_mem.c/h)建立索引并控制访问。

另外，由于内核栈大小仅仅 4kb，对涉及较深递归的在 dom 树进行画面渲染的要求很高，单纯使用递归渲染爆栈的可能很大，故使用动态扩展内存进行动态模拟栈，即手动将每次递归的上下文压入内存，并在可用内存不足时进行动态扩展。(详见 guilayout.c 中渲染函数 passRenderEvent、stash 以及 checkout 的实现)

以上仅为需要用内存过大的情况，另一个情况为内存需求很小时，一次性申请一整页会导致内存浪费。这种情况下，可对内存进行动态管理，不足时申请，无用时释放。尽管该数据结构并未单独封装，但在字母矩阵存储、txt 控件中对文字的管理等方面均有应用。

- 磁盘大小限制以及内核访问磁盘时机

Xv6 现行的文件系统仅支持 72kb 的单个文件以及 512kb 的总大小，由于扩大单个文件尺寸涉及修改文件系统结构，较为困难，且 72kb 勉强满足 gui 的要求，故不做改进。但 512kb 的总大小无疑是不可行的，故修改 mkfs.c 以及 fs.h 相关常数，使其支持 25mb 总容量，足够应付 gui 相关图片以及程序的存储。

同时，由于打开文件需要以某个进程为基础，我们选择在 init.c(第一个用户进程)中对图像(如鼠标矩阵、字母矩阵)进行初始化，从而合理地将该流程合并在启动过程中。

- 用户态应用程序大小限制

由于硬盘单个文件大小限制，编译后的用户态应用程序如需存储于硬盘上，需要小于 72kb，然而碰到了前所未有的麻烦：

我们实现了一个较为完整的 GUI 文件系统，支持新建文件夹，新建文件，文件和文件夹的复制、剪切、删除，还有文件的重命名，基本覆盖了一个完整文件系统的应该有的所有操作，我们还可以通过 File Explorer 直接打开文本文件和图片。

这一部分的代码其实比较繁杂，xv6 本身就有很多文件操作并没有实现，所以，这里，主要是通过文件读写还有 xv6 自带的部分系统调用还有对于文件夹的递归，实现了上述的各种各样繁杂的操作。其中需要注意的就是需要时刻维护路径。还有就是在做各种文件和文件夹操作的时候还要维护好各种绝对路径。在这里面，只要路径维护的好，应该就比较容易了。

这一部分代码，量比较大，涉及到各种各样繁杂的文件操作还有 GUI 绘图，最终编译出来的程序竟然超过了 xv6 规定的单个文件 72kb 的限制。为了解决这个问题，我们采取了两种方法，同时作用：

第一个方法是把这个大的文件系统的程序拆成两部分，也就是说把它拆成了两个进程进行执行。我的拆分方法是：把新建文件、新建文件夹、重命名这三个操作放进一个新的进程当中。这个新的进程是一个全屏的透明 div 块，用来屏蔽用户对于其父进程的操作，然后透明 div 块上会有一个对话框接受用户的操作。每次用户在文件系统的主进程中进行这三个操作的时候，主进程会先 fork，然后去 exec 我们刚才说的那个新进程，新进程会根据 exec 传的参数不同采取不同的操作。父进程会通过 wait()来等待子进程直到子进程结束。当子进程结束的时候，父进程的 GUI 会刷新界面，显示用户的操作结果。

第二个方法是自己对之前的 dom 树属性设置的系统调用进行优化，采用一次性批量设置属性的方法来减少指令个数。

通过上述的这两个方法，我们最终终于解决了文件系统程序太大引起的问题！(详见 guifilesystem.c 以及 fshandlekbd.c)

- 用户态程序间通信

作为 GUI 操作系统，本不应有 IPC 需求，但由于任务栏程序(即桌面，详见 indexpage.c)也是一个独立的用户态程序，即进程，另外在文件资源管理器中有启动其他进程的需求，则文件资源管理器必须通知其父进程(即任务栏)，以便其拥有对全部进程运行状况的掌握能力。

最终，我们设计了小巧而轻便的方法进行小数据量的 IPC 方式——考虑到该消息发送方为任意进程，接收方必然为任务栏进程(即虚拟节点的第一个儿子)，则直接利用现成的消息队列将特定的字符串传递给任务栏进程，任务进程收到消息后进行窗口准备并执行。

从而，实现了在文件资源管理器里用其他应用打开文件时也拥有任务栏信息。

附录一．新增系统调用一览表

在实现 xv6 过程中，新建的系统调用如下：(详见 user.h)

	系统调用	参数类型	意义
Dom 相关	createdom	(int a,uint b,uint* c)	创建 a 类型的 dom，以 b 作为其父亲，得到的 domid 为 c
	releasedom	(int a,uint b)	释放 a 类型的 dom，其 domid 为 b，注意其下不能有 img/txt 的子节点
	setattr	(int a,uint b,int c,void* d)	设置 a 类型 dom 的属性 c，domid 为 b，设置值存在 d 指向的内存
	getattr	(int a,uint b,int c,void* d)	获取 a 类型 dom 的属性 c，domid 为 b，预先留存的 buffer 地址为 d
消息队列	initprocessqueue	(void)	进程启动时调用函数，创建一个消息队列
	releaseprocessqueue	(void)	进程终止前需调用的函数，释放该进程消息队列
	getmsgfromqueue	(void* a)	从消息队列中获得一个消息放在 a 指向的位置。如果暂无消息，则会阻塞直到有消息到达
	informhometoopenfile	(char* a, char* b)	发送一个通知任务栏进程运行指定指令的消息，程序名为 a，参数为 b
	asynwait	(void)	非阻塞 wait，如果有儿子僵死则释放并返回其 pid，如果没有马上返回-1

附录二．如何编写新的 GUI 程序？

至今为止，我们已经为在当前 xv6 下的应用程序开发准备了完整的工具。由于用户态程序均为独立于操作系统内核的，故可在不改动 OS 内核的情况下管理 GUI 应用。如需要增加一个新的用户态应用程序，只需用 c 语言编译好并将其收纳进入 xv6 的硬盘即可。编写流程如下：

1. Include 系统调用头文件”user.h”
2. Main 函数的开始使用系统调用初始化消息队列。
3. 读取 argv[0]，将该字符串解析为 uint，令其为 ancestor。
4. 以上述值为始祖，利用 4 个相关系统调用建立 GUI。
5. 进入 while 死循环，调用 getmsgfromqueue 获取消息并处理。
6. 触发某退出条件，则从下向上释放所有的 txt 及 img 控件。
7. 释放消息队列，退出程序。
8. 如需要开始屏幕到此处的链接,则需修改 indexpage.c(任务栏对应的用户态程序)中的配置信息。

附录三．DOM 树典型结构可视化

下图为正常启动后 DOM 树树形结构图，其中不同颜色代表不同进程：蓝色为内核维护、黄色为总管程序(任务栏程序)维护的 DOM、每个灰色 DOM 为一个不同的运行中进程的始祖 DOM。

其中靠左的进程在渲染时会更靠前。