计算机系统软件(1)大作业方案二实验报告

简介

本次大组实验，我们在mit-pdos/xv6-public的基础上，进行了多项功能扩展。项目的源代码见src文件夹，使用手册见report/用户手册。本片文档主要介绍了各个小组的工作内容。

组队情况

`标准库移植`：凌精望，王兆伟，张汝婷，叶葳蕤

`内存管理`：张洋，郝天翔，张玉君，方梓唯

`文件系统`：王天旸，刘文华，汪祯寅，潘翰林

`多线程`：丁润语，李沁恬，胡天易，刘昀瑞

`Vim&命令扩展`：程嘉梁，江俊广，卢北辰，毛誉陶

`任务管理器`：顾国驰，董学昊，刘家维，曾铮

`文件检索&Basic解释器`：赵哲晖，陈倩，刘译键，徐天依

`GUI（文件浏览器&文本编辑器）`：薛广信，萧霭静，俞李金海，韦晶晶

`GUI（图片浏览器）`：蔡文涛，太志伟，陈年，秦海强

`GUI（音乐播放器）`：许逸凡，周耀华，喻琳颖，李岚芃

`Merge小组`：王泽宇，游凯超，纳鑫，丁郑

各组实际工作

标准库移植

技术实现&技术难点

1. 迁移五字班标准库

   为了充分汲取学长在迁移标准库方面的经验和成果，我们小组先将五字班的标准库移植到了方案2的xv6上。需要特别说明的是start.c中的*start是每个用户态程序的入口。所有用户态程序都会从*start开始，进行一些标准库的初始化后再进入main函数。

   我们在过程中遇到了一个难点：使用标准库后，有些文件超过了70KB，超过了xv6文件系统允许的单个文件大小。我们使用-s参数在链接过程中去掉了生成的二进制中的符号来缩小文件大小，使得生成的用户态程序不超过70KB。

   最终，我们让每个用户态程序都用上了这个版本的标准库（而newlib只被我们的lua使用）。

2. 迁移`newlib`

   我们小组在调研之后认为，为了在`xv6`上运行通用软件，迁移一个现有的标准库是比自己实现一个新的标准库更有效可靠的做法。在尝试迁移`glibc`遇到困难之后，我们小组决定迁移文档比较完善的`newlib`。

   a.添加目标系统

   在`newlib`的目标系统列表中添加目标系统`xv6`和目标机器架构`i686`。在这个过程中需要修改一系列`autotools`的配置文件并需要重新运行`autotools`。由于`newlib`的历史比较久，使用的文件树和新版本的`automake`不兼容，所以一定要编译一份`automake 1.12`。机器架构相关的代码因目标架构不同而不同。由于`i686`这种架构比较常见，我们找到了一份现成的实现。

   b.实现系统调用

   添加了`gettime`，`isatty`，`lseek`，`execve`，`stat`，`times`，`gettimeofday`系统调用。对于每个`xv6`中现成的系统调用，均在标准库中给出了对应的包装函数。在`crt0.c`中实现了用户态程序的入口，在运行`main`之前先进行signal和time的初始化。

   c.编译`newlib`

   使用了`barebones-toolchain`中的`i686-elf`交叉编译器编译`newlib`，得到`libc.a`，`libm.a`，`libg.a`，`libnosys.a`和`crt0.o`供用户态程序链接。至此这套标准库和交叉编译器形成了一个交叉编译的工具链。

3. 编译`lua`

   `lua`是一种完全基于C的脚本语言，可以通过它很方便的测试C标准库库函数的各项功能，能充分展示`xv6`运行通用软件的能力。在阅读`lua`源码过程中对`lua`做出了如下的适配，使其能使用前述的工具链编译通过：

   - 由于`xv6`没有完整的`posix`实现，去掉了`lua`的`posix`支持，使得`lua`不能使用文件读写锁（`flockfile`和`funlockfile`）和管道子进程（`popen`）。
   - 由于`xv6`没有`libdl`，去掉了动态链接支持。这导致了`lua`不能调用C，而C语言调用`lua`时会产生巨大的可执行文件。
   - 没有迁移`libreadline`，去掉了`readline`支持。这导致了`lua`在交互式界面下不能通过按上下键查看之前的输入记录。

   除了以上提到的阉割以外，`lua`其余的功能得到保留。小组成员已经对照`lua`官网上的文档对大多数`API`进行了测试。部分测试效果可以通过运行`xv6`中的测试脚本得到。

4. 解决栈过小的问题

   `lua`一启动就会占用约50KB的栈空间，而`xv6`使用的是4KB的固定大小栈。这部分感谢做内存管理的张洋小组提供的支持，他们使`xv6`栈的大小能够动态增长，解决了这个问题。

5. 解决文件系统不支持大文件的问题

   `lua`编译并静态链接后的大小约有2MB，而`xv6`单个文件的最大大小为70KB。在查阅论坛和github后我们终于找到了一种使用`ld -b`的方案：将`lua`的代码链接进内核中，绕过文件系统，直接在系统启动时随内核加载进内存并常驻在内存中。通过`nm kernel`找到标识这段代码在内存中位置的符号。需要运行`lua`时在`exec()`中进行特判，将这段代码从内核态的内存中拷贝进用户态的内存中进行执行。至此，`lua`已经能够在`xv6`中运行。

6. 合并库并编写测试脚本

   将`barebones-toolchain`、修改后的`newlib`、修改后的`lua`这三个库在`git`中合并到`xv6-public`中。对照`lua`文档测试`API`是否正常并编写用于测试和展示的脚本。

成员分工

- 凌精望：迁移五字班的标准库、构建交叉编译工具链、修改lua以支持xv6、编写测试并测试lua各项API。
- 张汝婷：阅读、修改和编译newlib、实现newlib使用的系统调用。
- 王兆伟：迁移五字班的标准库、调研解决xv6不支持大文件的问题并实现了`ld -b`和对应的`exec()`。
- 叶葳蕤：迁移五字班的标准库、阅读、修改和编译newlib。

参考文献

- https://www.embecosm.com/appnotes/ean9/ean9-howto-newlib-1.0.html迁移`newlib`的教程
- https://www.cs.ccu.edu.tw/~pahsiung/courses/esd/resources/newlib.pdf 针对嵌入式平台的`newlib`迁移教程
- https://wiki.osdev.org/OS_Specific_Toolchain 构建特定OS工具链的教程
- https://wiki.osdev.org/Porting_Newlib 一个迁移标准库的教程
- https://github.com/techno/newlib-xv6/blob/xv6/README-xv6.md 在这里面我们找到了`ld -b`的提示

使用的外部库

- ftp://sourceware.org/pub/newlib/index.html `newlib`的代码仓库
- https://github.com/lua/lua `lua`的github镜像
- https://github.com/rm-hull/barebones-toolchain 使用的交叉编译器

内存管理

功能实现说明

按照完成时间顺序排列。

内存页面写入时复制功能

使用场景

内存页面的写入时复制是一个比较重要的功能。常见的场景是，在进行了多次 fork 操作之后，多个进程将会有重复的数据（特别是代码段和数据段），但是 xv6 的设计没有共享页面的功能，所以每次 fork 的时候都会将每个进程的所有的内存全部复制一份，子进程将会具有父进程的内存的网站拷贝。如果这些内存中有一部分注定是不会被修改的话，这种复制是没有必要的。写入时复制这一想法的提出可以让进程在进行 fork 后暂时共享内存页面，只有当这个页面真的需要被修改才进行真实的复制，随后即可自由改动。这个功能对于用户来说是透明的。

实现方式

大部分涉及到共享的东西都要维护一个引用计数，在这个功能的实现中我们使用了一个全局的数组用来记录每个物理页框一共被几个进程所引用（即多少个进程共用这个物理页面）。

内存中的某一页在只有一个进程可以使用的情况下（典型情况是执行了 exec），这个进程自然有对自己所有内存的写权限。一旦发生 fork，新的进程就会将自己的页表映射到和父进程的内存相同的地址上，同时增加这些页框的引用计数，并将这些页面的权限改成只读（对两个进程都是只读的）。这样一来就避免了 fork 时发生的复制。随后，一旦某个进程想要修改某个页面上的值，就会因为权限问题触发一个缺页中断，在缺页中断处理的函数中我们如果判断出这个缺页中断是写权限造成的，就会将对应的页面真正地复制一遍，然后修改触发中断的那个进程的页表。这时会减小复制的那个页面的引用计数，如果到 1 的话就会赋予写权限。另外，一个进程退出后，清理内存本质上是将它占用的那些页面中引用计数大于 1 的页面减 1（减 1 后如果为 1 则赋予写权限），并将已经为 1 的页面删去。

测试方式

见 cowtest.c 文件。为了对这个功能进行测试，我们实现了一个系统调用用来实时获取当前系统中还剩下多少物理内存页面。这个系统调用的实现思路进行使用一个全局的计数器整数。每次分配了页面就将这个计数器减一，回收则加一，这样总是可以知道整个系统中剩下的页面数。测试过程如下：首先在用户进程的数据区放一个整数（通过全局变量实现），然后执行 fork，统计剩余页面数。在子进程中，修改这个整数，然后再统计剩余页面数。此时可以发现少了一个页面，可见确实发生了共享和写时复制。后面的测试基本上是在验证正确性，例如多个 fork 嵌套时功能的正确性以及一个进程中的修改是否会影响到其它进程。

内存页面惰性分配功能

使用场景

在实际的用户软件中用户可能会申请大量的堆内存，但是不一定完全使用它们。例如，某些质量低下的程序可能会使用二维数组来存放一个非常稀疏的矩阵，但是编译器只能按照二维数组的情况进行处理，这样会导致大量不必要的堆内存浪费。通过使用惰性的内存页面分配，可以在程序不需要特殊改动的情况下减小这种浪费现象的发生，提高内存的利用价值。这个功能对于用户来说是透明的，现有程序无须进行改动即可具有这种特性。

实现方式

原始 xv6 的实现中有一个系统调用 sbrk，这个系统调用是用来增大程序占用的内存的。一些如像 malloc 的函数本质上依赖于这个系统调用。每个进程都具有一个 sz 域，这个域记录了当前进程的内存大小。 sbrk 的实现是先提高进程的 sz 值，然后使用 kalloc 将页面分配给这个进程，最后更新该进程的页表，使得进程能够正常使用这些内存。在我们的实现中，我们只进行第一步即只提高进程的 sz 值，这样在一些临界判断中，该进程将会“误以为”自己

已经拥有了自己需要的内存页面数，实际上我们还没有分配给它。在稀疏矩阵的例子中就是它认为自己已经被分配了一个存放矩阵的数组，实际上只是有了这个记录，真实的分配没有发生。

当进程尝试访问自己的相应页面时，因为这个页面没有出现在页表中，会触发一个缺页中断。在缺页中断处理函数中一旦能够确定这个中断是由惰性分配产生的，就可以计算出产生缺页中断时进程想要访问的虚拟页面，给它分配一个实际的物理页面并且更新页表即可。在稀疏矩阵的例子中，只有那些非零元所在的页面真的被分配了页面。

测试方式

见 lalloctest.c 文件。这个功能的测试使用了前面提到的查询系统剩余页面数的系统调用。在测试程序中，我们首先开一个很大的数组，然后依次对这个数组中的元素赋值，赋值过程中每经过一定的时间就会查询当前系统中剩余的页面数。在原始 xv6 实现中，开完数组后页面数会突然减小很多，此后不管怎么操作这个数组，系统剩余的页面数应该是不会变的；但是在我们的测试程序中可以看到开完数组后页面数基本不变化，在后面的赋值过程中页面数才逐渐减小，所以这个功能可以被认为是正确实现了的。

零指针保护功能

使用场景

零指针就是指向地址为 0 的指针。在平时的编程中习惯将零地址用作一个特殊的地址，表示“无、没有”之类的意思，表示对应的指针不指向任何对象。但是 C 语言中并没有规定这一点，只是在语言层面上定义了 NULL 宏，一个地址是否有意义是由平台和操作系统来规定的。为了简单起见，xv6 系统中并没有规定零地址是不可使用的，所以对这个地址的读写都是可以进行的。

但是在工程中，具有零指针这个特殊的指针是很重要的，虽然说可以通过高质量的编程来保证不会访问零地址，但是总会有出错的时候，这不利用用户程序的开发。在用户进程看来 0 将是一个普通地址，如果某段代码写错了导致访问了零地址，在没有零指针保护的情况下程序会继续运行，而有零指针保护的情况下则可以提前将这个错误暴露出来。

实现方式

为了实现用户态进程的零指针保护，我们的解决方案是将用户进程从自己的第二个虚列内存页面中开始加载，这样第一个虚拟页面就腾了出来。然后将第一个页面对应的映射在页表中去掉。这样，如果进程尝试访问零地址（其实是第一个页面中的任何地址），就会触发一个缺页中断。在缺页中断的处理函数中，检查一下这个进程是否是因为访问第一个虚拟页面而导致的缺页中断，就可以判定出它是否是访问了零指针，如果是的话，就将这个进程杀掉即可。

实现过程分成两部分。从第二个页面开始加载是在 `exec` 函数中进行的。这个函数会从磁盘上读入一个 ELF 格式的可执行文件装入到内存中，通过修改这个装入的过程即可完成内核态的更改。第二个更改是要对用户态进程的行为进行更改，因为编译器在进行编译的时候默认认为程序的转载点是 0x0，通过修改 Makefile 让编译器在编译用户程序的时候总是将所有的地址加上一个偏移即可。这样用户在进行程序编写的时候不需要知道这个实现细节，零指针保护功能也能正常运行。整个功能对用户是完全透明的。

测试方式

见 npptest.c 文件。测试程序中我们直接进行 `printf(1, "%d", *(int*)(0));` 这个语句，然后看整个程序是否会被杀死。如果被杀死的话就表示功能成功实现，否则认为实现错误。

栈自动扩展功能

使用场景

xv6 中一个进程的栈有两个，系统栈和用户栈。因为一个进程的内存空间中的动态内存有两个，栈和堆都要增长，可能是为了简单起见，xv6 并没有提供栈的自动增长功能，而是写死了是一个页面。这样一来整个内存空间就只有堆可以增长，所以内存空间中高段地址是没有用上的。系统栈是在系统调用的时候使用的，4KB 是够用的，但是用户栈也只有 4KB，这就在很大程度上限制了系统能够进行的实际操作。通过栈自动扩展就可以让栈和堆共享整个进程的虚拟内存空间，这样就突破了栈空间有限的限制。

实现方式

和很多操作系统的处理方式类似，我们将用户栈的栈底放到用户进程内存空间的最顶端，这样在这个内存空间中，栈是从高到低增长的，堆是从低到高增长的，二者共享虚拟内存空间。通过记录栈顶和堆顶的位置可以维护住它们不会相交，只要保证栈顶始终在堆顶以上并且相差至少一个页面即可。栈的增长按照页面进行分配，初始情况下只有一个页面，当栈增长到一定程度时突破这个大小，内核会收到一个缺页中断，经过检查如果发现可以给它继续分配一页当作栈时就会进行分配，以此类推。

除了这个更改外，xv6 在使用系统调用时是将参数压在栈上面的，而在获取参数的时候系统会对参数的位置进行检查。由于原来的内存布局和修改后的内存布局不一样，所以这些检查都需要重写，这个过程比较麻烦。

测试方式

见 sagtest.c 文件。栈的增长主要来自于两个方面：函数调用和栈变量。在测试程序中我们使用一个递归函数，递归大概 16000 层，在每一层中会开一个整数数组（以栈变量形式开启），这样起码会占用 12MB 的栈内存。在原始的 xv6 上这个程序是无法运行的，但是在我们修改后的系统上可以顺利执行，故认为这个功能实现得没有问题。

虚拟页式存储和页面交换

使用场景

虚拟存储管理是为了解决整个系统中需要的内存比物理内存要大的问题，通过将一部分内存转移到硬盘上来完成。xv6 中一共只有 224MB 的物理内存，但是单个进程的内存空间中留给用户的就有 2GB，所以添加了虚拟存储管理之后才能完全发挥出这 2GB 内存空间的能力。因为 xv6 使用的是页式存储管理，我们将会实现一个虚拟页式存储管理的系统。

实现方式

实现虚拟页式存储管理的方式有两种，一个是换页可以跨进程换页，另一种是每个进程使用单独的换页的空间。第一种实现方式中需要内核能够在换页时访问其它进程的页表并作改动，实现起来比较复杂，所以我们选择第二种实现方式，每个进程单独负责自己的内存的换页。为了实现这一点，每个进程需要维护两张表，一个是在内存中的页面构成的链表，这张链表决定要将哪些页面换出；另外一张是已经换出的页面在交换文件中的位置。

我们实现的是 FIFO 换页方式，在内存中的页面构成的链表是双向链表。每个进程能够在内存中留下的页面数受到限制，否则因为每个进程要维护这样的一张表本身就有额外的内存开销。每次要进行换页时，将这个链表的最后一个元素换出，同时将要换入的页面换入。选用 FIFO 是因为我们没有时间研究 qemu 的硬件系统，根据操作系统相关教材的描述，其它的一些换页策略都是需要硬件进行支持的，例如 LRU 算法需要每次访问页面的时候都进行一次更新。我们没有这种硬件资源的话只能手动进行模拟，这会导致系统的性能很低，综合考虑后，选用了在每次访问页面时不会更新，只有在需要换页的时候才会更新的 FIFO 算法。虚拟页式存储功能的代码经过仔细设计，可以很方便地切换为其它策略，在实现完 FIFO 之后我们发现还可以实现时钟算法，但是受精力所限没有实现。

在记录换出页面方面，我们没有使用常规的在硬盘和内存中都留有页面的方式，因为这样会导致交换文件存在冗余。现在一个页面要么在内存中，要么在硬盘中。交换文件不是以分区形式存在的，而直接是以文件形式存在的，地址相邻的两个页面在文件中的位置不一定相同，这个是由记录已换出页面在文件中的位置来实现的。受限于 xv6 羸弱的文件系统，单个文件的大小不能超过 70KB，所以我们使用了多个文件作为一个进程的交换文件，通过对文件这一层进行封装让交换的接口变得比较简单。然而 xv6 中每个进程能够持有的文件句柄（16 个）和整个系统中能同时存在的文件句柄数（只有几十个）也是有限制的，这些限制使我们不得不缩减每个进程的交换内存容量，最

后每个进程可用的交换空间只有 1MB 左右。但是这个问题是文件系统造成的，按照我们的设计整个虚拟页式存储管理可扩展性是较强的，只要文件系统能够支持更大的文件，这个问题立刻就会得到解决。

对于这样的两张表，他们存在于系统内存空间中。以第一张表为例，每个进程有 25 张页面用来存放这个链表，这些页面自己又构成了一张链表，并且在进程建立时才会被分配，这样可以尽量减小这些表本身的额外开销。这些表必须动态地被创建，因为如果硬编码进内核会导致内核代码超过 4MB，即一个超级页面的大小，这样会导致内核无法正常启动。

在最后我们实现了内存的虚拟页式存储管理，能够在物理内存不够时进行换页。不过因为文件系统的限制，这种换页操作发挥的功能比较有限。

测试方式

见 pgswptest.c 文件。前面已经提到了每个进程的能够驻留的页面数是有限的。测试时，我们让一个进程申请超过这么多页面数的页面（通过使用递归和在递归中开启栈数组），并且在换页相关部分加入一些日志，此时可以观察到换页现象确实发生。因为使用的是 FIFO 策略，所以被换出的页面是最低的几页，而程序要退出时需要这几页，所以程序如果要正常退出需要将这些页面换回来。我们观测到了程序正常结束，可以认为这个功能实现得没有问题。

进程间共享内存

使用场景

在现代操作系统中，需要进行非常频繁的进程间通信。进程间的通信方式包括低级通信：信号量(semaphore)和信号(signal)；高级通信：共享内存(shared memory)、消息传递(message passing)和管道(pipe)。在xv6 系统中已经实现了用管道的方式来用于通信，不过这种方法也有着相当多的缺点，比如只能在具有亲缘关系的进程之间通信、只能单向传输数据、缓冲区有限等等，并且不适合于大量数据的并发式传输。为了改善这种情况，我们实现了高速的共享内存通信机制。对于共享内存的读写可以由用户选择进行并行读写或串行读写。并行模式适用于有多个进程同时写入或读入同一块共享内存区域的不同偏移(offset)处的数据，它为高并发性的程序提供了编程接口；而串行模式则可以更普遍地适用于两个或多个进程之间的互斥通信。

实现方式

我们部分地借鉴了管道通信的思想，用签名(signature)来标识一块共享内存区域。使用一个全局数组`shmlist`来保存256个共享内存区域，每一个区域有一个自己的签名，一个区域内部最多有1024个页面，相当于每一块共享内存最大为1024*4=4MB大小。每一块共享内存区域包含一个1024项的指针数组，这个指针数组指向的是用户申请的共享页面(最多1024个)，它单独用一个页面存放，相当于一个索引页，在对共享内存进行读写时通过索引页号和页内偏移量来找到读写的位置。所有的共享页面均通过动态申请和释放，每一个进程拥有最多5块共享内存区域的使用权(使用权即有权利对这块共享区域中的数据进行读写)，当第一个进程申请这块共享内存的使用权时进行动态内存分配；进程退出时自动取消对其申请的动态内存区域的使用权，当某块共享内存区域的被使用次数为0时该块区域的内存将被回收。

在申请共享区域使用权时，通过一个标志位来标志这块区域是一块串行读写区域还是一块并行读写区域。这一标志位由第一次申请这个块的进程来指定，后续申请该块的进程需要提供同样的标志位才能成功申请使用权，否则失败。

测试方式

我们编写了一个`shmtest.c`文件，对大量数据的读写、并行读写、串行读写、不同偏移量的读写分别做了不同的测试，从显示的结果来看功能的实现没有任何问题。

`vmstat` 命令

使用场景

在Linux系统中，有很多实用的命令可以帮助用户和操作系统内核进行交互，来得到一些内核才能访问的状态信息。比如内存使用状况、进程运行状况等等。其中有一条命令叫做 `vmstat`，它可以报告进程、内存、I/O等系统整体运行状态。而在原生xv6中并没有实现类似的功能，用户只能根据自己运行的进程来猜测系统当前的状态，这在很多场景下是非常不便的，会给xv6系统上后续的用户态功能扩展带来极大的限制。因此我们基于对xv6内核的了解，部分地实现了一个类似Linux系统上的 `vmstat` 的命令。

实现方式

考虑到原生xv6中写死了物理内存为224MB的大小，它并不能自动侦测当前机器可用的内存大小，但是却强行告诉用户有2G的虚拟内存可以用，这两个值相差很大，如果还按照原生 `vmstat` 命令那样显示虚拟内存的使用情况可能会给用户带来对本机情况相当程度的误解。因此我们决定在 `vmstat` 命令中显示实际的物理内存使用情况而不是虚拟内存使用情况。通过在 `kalloc` 和 `kfree` 函数中添加一条修改当前可用物理页面数量的语句来更新可用页面数量，这样就能够实时获得系统的物理内存使用情况；通过遍历 `shmlist` 中每一个共享内存区域，读取其保存的共享页面数量可以得到总的共享内存使用情况；通过遍历 `proc` 数组可以得到每一个进程的内存使用情况。最终提供给用户一个接口来通过几个系统调用得到系统中实时的上述信息。

测试方式

通过显示调用 `vmstat` 命令，观测到显示的数据确实与当前系统状态一致。

成员分工

- 张洋：内存写入时复制、栈自动扩展、虚拟页式存储和页面交换
- 方梓唯：资料查找、内存惰性页面分配、测试
- 郝天翔：进程间共享内存、 `vmstat` 命令
- 张玉君：零指针保护、接口变更整理、测试

文档部分由各功能负责人撰写。

参考资料

- https://pdos.csail.mit.edu/6.828/2017/homework/xv6-zero-fill.html
- https://blog.csdn.net/woxiaohahaa/article/details/50540322
- https://www.cnblogs.com/ggjucheng/archive/2012/01/05/2312625.html
- https://github.com/amcolash/cs537/tree/master/xv6-3
- https://github.com/asafch/xv6-paging
- https://github.com/rschmukler/cs537-p3
- https://github.com/amcolash/cs537/tree/master/xv6-3
- https://github.com/shixuan-fan/xv6-kernel-modified
- https://github.com/hayleejane3/shared-memory-xv6
- https://github.com/theSparta/copy-on-write-xv6
- https://stackoverflow.com/questions/50343431/how-to-know-which-code-caused-a-trap-in-xv6-when-debugging

文件系统

技术实现

1、制作镜像

根据FAT32的磁盘组织形式，制作`fs.img`镜像。这一部分需要重写`mkfs.c`，实验中另写了一个文件fat_mkfs.c，同时修改了`sysfile`中的部分内容。打开镜像文件后向第0扇区写入DBR，留下保留扇区的空间，写入FAT表表头标记，紧接着FAT表是根目录区，然后向数据区写入需要预先加载的文件，包括`_init, _cat`等，最后将文件系统信息，如启动程序最后分配的簇号，剩余簇数量等信息保存至保留扇区。具体实现详见源代码文件。包含的主要函数如下：

```
void wsect(uint sec, void *buf) 向sec号扇区中写入buf
void rsect(uint sec, void *buf) 从sec号扇区中读入数据到buf
uint cnallloc()                 分配一个空闲簇号
uint appendBuf(uint clus, void *buf, int size, int curWSize) 向簇号为clus的簇中追加数据buf
struct direntry mkFCB(uchar type, char *name, int size, uint *clusNum)  生成FDT
```

2、因为在整个xv6系统的很多地方都会涉及到对文件系统内函数的调用，因此在原有文件系统的基础上修改为fat32文件系统的前提就是在尽量不改变这些函数功能的情况下实现符合于fat32文件系统的寻址等等。因此将在原有函数的基础上，实现如下函数。

实现函数	原有函数	修改思路
readDbr(int, struct FAT32_DBR *)	readsb(int dev, struct superblock *sb)	用以获取系统引导记录区
uint getFATStart(uint cnum, uint *offset)		获取一个簇号在FAT表中的扇区号和对应的位移
void updateFATs(struct buf* sp)		用以更新FAT表的备份
uint getFirstSector(uint cnum)		获取一个簇号对应的首个扇区号
fat32_calloc(uint dev)	uint balloc(uint dev)	分配一个空闲的簇，与原有函数在BMP中查找类似，在FAT表中进行查询。可以根据保留扇区记录的FSInfo信息更快捷得检索
fat32_iinit(int dev)	void iinit(int dev)	类似，初始化inode节点缓冲区
fat32_iupdate(struct inode *ip)	void iupdate(struct inode *ip)	用以更新磁盘上的inode节点信息，因此要通过inode的上级目录来查找inode的文件信息的存储位置，在进行更新
fat32_iget(uint dev, uint inum, uint dirCluster)	static struct inode* iget(uint dev, uint inum)	获取inode节点，除参数需要增加上级目录起始簇号外，基本类似
fat32_idup(struct inode *ip)	struct inode* idup(struct inode *ip)	增加inode节点引用次数，完全一样
fat32_ilock(struct inode *ip)	void ilock(struct inode *ip)	加锁，并获取磁盘上的文件的详细信息，除了需要考虑在磁盘中寻址外基本类似
void fat32_iunlock(struct inode *ip)	void iunlock(struct inode *ip)	去锁，完全一致
void fat32_itrunc(struct inode *ip)	void itrunc(struct inode *ip)	清空文件内容，需要在上级目录中删除该文件信息，并且的fat表中删除所占用的簇的信息
void fat32_iput(struct inode *ip)	void iput(struct inode *ip)	引用次数减一，当为0是则删除文件，基本类似
void fat32_iunlockput(struct inode *ip)	void iunlockput(struct inode *ip)	完全一致
void fat32_stati(struct inode *ip, struct stat *st)	void stati(struct inode *ip, struct stat *st)	获取信息，基本一致
int fat32_readi(struct inode *ip, char *dst, uint off, uint n)	int readi(struct inode *ip, char *dst, uint off, uint n)	读取文件内容到dst中，初始偏移off，长度为n。即寻址方式的改变，从起始簇开始扫描每个扇区，判断是否需要读取。

实现函数	原有函数	修改思路
int fat32_writei(struct inode *ip, char *src, uint off, uint n)	int writei(struct inode *ip, char *src, uint off, uint n)	向文件ip写入src的内容，起始偏移off,长度为n。同为寻址方式改变，当长度不够时调用fat32_calloc分配一个新的簇空间
int namecmp(const char *s, const char *t)	int namecmp(const char *s, const char *t)	比较两个字符串
struct inode* fat32_dirlookup(struct inode *dp, char *name, uint *poff)	struct inode* dirlookup(struct inode *dp, char *name, uint *poff)	查询dp目录下是否有文件名为name的文件。Fat32_readi直接获取，基本类似
int fat32_dirlink(struct inode *dp, char *name, struct inode*dp1)	int dirlink(struct inode *dp, char *name, uint inum)	在dp目录下创建名为name的文件。因为不能直接通过inum指向磁盘中的inode，因此需要文件内容的详细信息，即dp1
static char* skipelem(char *path, char *name)	static char* skipelem(char *path, char *name)	解析路径名，完全一致
static struct inode* fat32_namex(char *path, int nameiparent, char *name)	static struct inode* namex(char *path, int nameiparent, char *name)	获取对应路径下文件的inode，基本类似
struct inode* fat32_ialloc(struct inode *dp, short type)	struct inode* ialloc(uint dev, short type)	为创建一个inode节点，先通过fat32_calloc分配一个起始簇，再通过fat32_iget得到一个inode

遇到的困难

二、五字班部分对整体系统改动较大，且相应的改动没有详细说明，导致代码阅读和提取十分困难，五字班整体对xv6框架做了修改，导致难以找到其文件系统耦合的部分，且其修改的方式比较暴力，难以整合。

成员分工

- 王天旸：推进组内进度；对组内代码的整合；实验报告整合和撰写。
- 潘涵林：整合了三字班代码；相应部分实验报告的撰写。
- 汪祯寅：尝试整合五字班部分代码，遇到较大困难；汇报PPT大纲的制作，汇报
- 刘文华：尝试整合二字班代码，遇到较大困难

多线程

技术实现

模仿了linux中的`jobs`指令：打印出后台运行的进程信息。包括运行状态、进程名、pid。

实现系统调用`fgproc()`，并修改sh.c来跟踪前台进程。每当按下CTRL+C时，将SIGNIT发送到前台进程。

扩展了`kill`指令的功能，通过传入后台进程的序号，可以杀死后台进程。

通过添加更改父进程的系统调用实现了`bg`命令，使得&能够记录后台的进程。

技术难点

1. 环境配置与版本兼容。
2. 实现快捷键功能。

成员分工

- 李沁恬：实现`jobs`指令。
- 刘昀瑞：实现CTRL+C快捷键功能。
- 丁润语：扩展`kill`指令，增加文件读取函数`readline()`。
- 胡天易：实现`bg`、`&`指令。

参考引用

1、https://en.wikibooks.org/wiki/A_Quick_Introduction_to_Unix/Job_Control

2、https://stackoverflow.com/questions/8021774/how-do-i-add-a-system-call-utility-in-xv6

Vim&命令扩展

技术实现

`shutdown命令`：向系统指定地址写入指定内容，以触发关机命令。

`login命令`：修改系统初始化，使得系统在初始化时进入登录界面。用一个文件记录用户设置的用户名和密码，用户在成功登陆之后可以对用户名和密码进行修改。

`more命令`：通过`lseek`函数定位文件阅读位置，从而输出用户希望阅读的内容（页面）。

`命令历史机制`：历史通过两种方式储存，一是磁盘上的`.history`文件，二是内存中，在内存中通过`history`结构体存储。`H_NUMBER`和`H_LENGTH`分别指可记录在内存中的命令数量和命令最大长度。文件中的命令历史作为长期存储，shell每次启动时会从该文件加载最近的历史命令到内存中。内存中的命令历史作为临时存储，可以更快响应用户对最近历史的请求。用户每输入一个命令则将命令记录到内存和文件中。

`上下箭头快捷键`：在`sh.c`的`getcmd`函数中对输入的命令做判断，若是向上或向下的箭头则去掉上下箭头的显示，从内存中的命令历史读取要求的命令显示。

`实现!!快捷命令`：获取命令之后对命令做判断，若是“!!”则从内存中的命令历史读取上一条命令运行。

`实现history (N)命令`：从磁盘上的`.history`文件中获取文尾N行输出，若无N参数则输出10条（若总共的命令历史少于10条则全部输出）。

`实现history -c 清空命令`：清空磁盘上的.history文件。

`实现ls中.*文件不显示（如.history）`：增加判断语句，若文件名以‘.’开头则不显示。

`show系统调用`：将文件置为显示状态。

`hide系统调用`：将显示状态的文件置为隐藏状态，或将隐藏状态的文件彻底删除。

`sys_open`：以写方式打开文件时，按照C语言标准，如果原先存在该文件，在下一次保存前应该先将该文件清空。但是`sys_open`中没有将文件清空。导致了vim在保存时存在bug。最终的解决方案是，每次vim保存一个文件时，先将文件删除，然后重新创建一个文件，这种操作等效于`sys_open`先将该文件清空。

`Shell脚本`：在执行前，首先将脚本拆成多行语句，然后将每一条语句输入给Shell去执行。只要在原先的Shell中可以正常执行的语句，脚本同样可以执行。后续可以考虑扩展Shell的语法（比如引入for循环、变量等），从而让Shell脚本真正发挥优势。

`Vim的颜色高亮功能`：通过提供给`setconsole`字体和背景的参数即可实现。

技术难点

1. 在添加`shutdown`命令时，发现Xv6修改了执行关机命令的地址，而有关文献都是旧版内容。最后终于在Github一个Xv6相关库中找到最新版操作地址。
2. 添加`lseek`系统调用时需要将Linux的实现代码移植到Xv6中，在实际的移植过程中遇到很多不匹配导致出错的情况。

   . vim中如何区分不同类型的单词：这个问题其实是编译原理中的词法分析问题，vim通过对每一行的语句进行一次词 法分析，得到该行不同单词的类型，然后根据该类型刷新其颜色。

   . 增加较多功能后vim的响应速度变慢：原先的vim逐个字符进行刷新，这么做的好处是刷新速度快，坏处是对于不同的操作（插入、删除等）都需要单独处理刷新策略，编写和维护代码的难度大，并且很难适 用于加入颜色高亮后的刷新。

   如果每次操作后都对vim整个窗体进行刷新，则响应速度过慢。考虑到每次操作往 往影响的是某些行的内容，因此采用了单行刷新的机制。

   同时原先文档内容修改/光标移动/视图刷新三个操作是相互耦合在一起的，代码难 以维护，而且效率较低。通过将文档内容修改/光标移动/视图刷新分成三个独立动作， 提高显示的效率（避免了重复刷新）。

成员分工

程嘉梁：整合`mv`，`touch`，`cp`，`head`，`tail`，`splice`，添加`shutdown`，`login`操作，按页阅读文件命令`more`操作，`lseek`系统调用

毛誉陶：命令历史的上下箭头快捷键的整合与修改，实现`!!`快捷命令，`history (N)`，`history -c`，ls中.*文件不显示，例如`.history`

卢北辰：文件删除确认及回退功能

江俊广：修复原先vim的bug：原先无法打开空文件、原先不支持tab键、原先在删除文件中某一行后再保存，保存不成功（由于xv6调用sysopen时没有将文件在磁盘上对应内容清空）。

实现了vim中的高亮功能、实现vim单行刷新的机制。

实现shell脚本的运行

参考引用

1、http://pdos.csail.mit.edu/6.828/2012/homework/xv6-syscall.html

2、https://blog.csdn.net/qq_25426415/article/category/6684908

3、https://www.cnblogs.com/yongheng20/p/4947702.html

4、词法分析代码参考：https://www.cnblogs.com/zyrblog/p/6885922.html

5、vim规范参考：http://www.runoob.com/linux/linux-vim.html

6、shell规范参考：http://www.runoob.com/linux/linux-shell-io-redirections.html

任务管理器

技术实现

实现了简单的命令行界面的可交互的任务管理器。

在xv6的shell内输入taskmgr指令即可进入任务管理器界面。

任务管理器会列出当前所有进程（xv6进程数目上限为64）的基本信息，包括进程的id、名称、状态和占用内存的大小。

选中的进程会被高亮显示，按上下键可移动选中的进程，按左右键可翻页，按F键可进行刷新，按K键可以删除选中的进程。

新增的系统调用函数：

`inittaskmgr`：启动任务管理器时调用，记录所有和当前控制台有关的信息，如屏幕内容、光标位置等，然后将控制台屏幕清空。

`closetaskmgr`：关闭任务管理器时调用，根据记录的所有启动任务管理器前和控制台有关的信息将控制台恢复至打开任务管理器前的状态。

`getprocinfo`：获取系统内全部（64个）进程的信息。

`updscrcont`：将计算出的应该显示在屏幕上的信息打印到屏幕上。

任务管理器主逻辑：

1.调用inittaskmgr初始化任务管理器界面。

2.调用getprocinfo获取进程信息。根据进程信息和当前页码手工计算出24*80大小的控制台上每一位应打印出的字符，再调用updscrcont将计算结果显示在屏幕上。每读入一个字符，若其为合法的非退出指令（上下左右键或k、f键）则重复第2步，直至收到退出指令（按下q键）或杀死的进程是任务管理器自己。

3.调用closetaskmgr将控制台屏幕恢复至开启任务管理器之前的状态。

具体实现时仅仅参考了往届代码的思路，并没引用任何代码原文。

技术难点

1. 往届代码可复用性极差，如何使自己编写的代码易于merge。

   往届代码为方便在其他源文件内编写系统调用函数，将console.c内的部分代码删除并移到了一个新增的头文件中，极大地影响了代码的可复用性。如果我们小组也这么做将极不利于merge。对此我们采取的措施是直接在console.c文件内编写自己的初始化任务管理器界面、获取进程信息、打印等函数，然后再将其封装成系统调用的形式。

2. 获取所有进程信息后，如何将其输出在屏幕上。

   在这个问题上我们几乎完全抛弃了往届代码的解决方案。考虑到xv6允许的最大进程数为64个，我们组计划将其分4页进行显示。而控制台屏幕大小为2*480，于是建立一个大小为96*80的缓存区，每一块大小为2*480的缓冲区内存放一整页应输出在屏幕上的内容。再根据当前页码将要打印的大小为2*480的缓冲区内的内容传入负责打印的系统调用函数updscrcont内即可。

3. 较为幸运的是，其他一些难点，例如如何使任务管理器运行时光标不显示在屏幕上、如何使选中的当前行高亮显示以及如何使用xv6的锁机制，往届代码都有了较为成熟的解决方案，因此解决起来相应的顺利些。

成员分工

- 顾国驰：阅读并编写任务管理器主逻辑；阅读并编写按键响应函数；编写新增的系统调用函数；设计控制台打印函数的工作原理，为后续的工作提供了必要的支持。
- 刘家维：函数与变量命名设计；阅读并编写控制台打印函数；git/github操作与指导。
- 曾铮&董学昊：了解xv6使用的CGA显示标准，理解新增的系统调用函数的工作原理，并对其接口给出必要的注释，为后续的工作提供了必要的支持。

参考引用

1、xv6 中文文档

2、2014级林樾小组作品

文件检索&Basic解释器

技术实现详情

`find`命令：在四字班方案一的基础上，对其进行bug fix，并对其原有功能进行拓展。初始版本 的`find`仅能实现对给定文件名从根目录开始遍历文件名与之相同的文件，且可显示的内容也 有限。完成的任务如下：

1. 将参数输入的格式进行了改变，使之与linux 的find 命令格式相同命令格式：`find 目录 -name 文件名`
2. 在输出的文件信息中，增加了对文件创建时间的输出
3. 支持文件名模糊匹配（使用通配符）
4. 支持搜索时忽略大小写，命令格式：`find 目录 -iname 文件名`
5. 可以从指定子目录开始查找
6. 支持查找特定大小的文件（大于或小于一定值），命令格式：`find 目录 -size 文件大小范围`（+n 表示文件大小大于n，-n 表示文件大小小于n）
7. 支持查找特定创建时间的文件（n 分钟前或n 天前），命令格式：`find 目录 -cmin 时间`（n 分钟前）`find 目录 -ctime 时间`（n 天前）
8. 修改了学长查找目录深度不能超过2 的bug
9. 支持删除查找到的文件
10. 支持删除查找到的文件，`命令格式：find 目录 XX XX delete`
11. 支持查找所有空文件，`命令格式：find 目录 -empty`

bi 命令(BASIC 解释器)：对五字班方案二实现的BASIC 解释器进行了移植，并增添了一系列命令的支持。完成的任务如下：

1. 原有解释器仅支持输入文件执行，我们增添了从shell直接输入代码的功能，输入`RUN`开始运行
2. 增添了`LIST`命令，可输出之前`LIST`的所有代码
3. 增添了`GOTO`命令，可跳转至指定行号
4. 实现了`WHILE-WEND`循环语句
5. 尝试将原有的A-Z固定数量的变量改为变量表的形式，但因需要改动的地方太多未能完成。

   样例代码

```
INPUT A
WHILE A < 5
LET B = 1
WHILE B < 5
LET B = B + 1
PRINT B
WEND
WEND
```

`pwd`命令：能够实现输出系统当前工作目录的功能。

`date`命令：实现输出当前系统时间的功能

`ls`命令改进：支持输出以'.'为开头的隐藏文件，输入`ls -a`

`cat`命令改进：当cat 参数中的文件不存在时，自动创建该文件

`touch`命令bug fix：修复了程嘉梁组`touch`命令创建文件时间戳为0 的bug。

子目录下执行shell指令：由于xv6中shell指令均保存在根目录，因此通过将shell 中输入的指令名变成绝对路径，即可实现在子目录下运行shell命令的功能。

`rename`命令：实现了`rename`命令，用于文件重命名。

TAB自动补全：按下TAB 键时，shell会按照用户输入的字符串匹配前缀为该字符串的shell 命令，并输 出所有匹配到的shell命令名，如果没有找到则提示用户未找到。

技术难点

1. 如何获取系统当前时间?

   对特定文件时间的查找和`date`命令的实现都需要获取系统当前时间，其他组实现了在内 核态下获取系统时间，但用户态并不能直接获取系统时间。因此，我们添加了一个系统调用`getTimeStamp`来获取时间戳形式的系统时间，再将时间戳转换为年月日时分秒的形式，完 成了对系统当前时间的获取。

2. 如何记录当前系统的工作目录?

   xv6 系统中进程的PCB中有关于当前工作目录的记录，只不过是以`inode`来记录，并没 有当前路径名的记录，而且也无法通过`inode`来查询到当前的路径名，因此只能自己实现函 数在切换目录时对当前工作目录进行修改，并在PCB增加了一个用来记录当前路径的字符 串。之后，我们实现了一个系统调用用来获取当前工作目录，来实现`pwd`命令。

3. WHILE 循环的实现

   由于没有学习过编译原理，因此实现解释器是一个不小的挑战。在实现while 循环时， 需要一个栈结构来保存循环语句的变量、行号、判断条件等，我们在学长代码的帮助下完成 了这一任务。

成员分工

- 赵哲晖：实验分工，`find`命令移植与改进，BASIC 解释器移植与改进，`pwd`命令，`cat`命令改进，`touch`命令bug fix，报告撰写。
- 陈倩：`find`命令改进，`date`命令，`rename`命令，`ls`命令改进。
- 刘译键：BASIC 解释器改进。
- 徐天依：BASIC 解释器改进，子目录运行shell 命令，TAB 自动补全。

GUI（文件浏览器&文本编辑器）

技术实现详情

一共实现以下几点功能：

1. 现在可以从文档浏览器双击打开文件，并使用文本编辑器浏览
2. 现在文档浏览器可以返回上一级
3. 现在文本编辑器可以更好的打开文件
4. 文本编辑器增加退出前确认保存功能

技术难点

本来还想实现一个鼠标滑轮上下滑动文档浏览器窗口，但是从各个网站上找了很多关于PS/2的鼠标驱动的实现，但是怎么样都得不到正确的反馈。故弃之。

成员分工

- 薛广信：文档浏览器双击打开文件、NotePad会读取Argv[1]作为初始要打开文档的地址。
- 韦晶晶：文档浏览器扫描XV6的所有文件并且显示在文本编辑器中。
- 萧霭静：文档浏览器返回上一级。
- 俞李金海：文本编辑器退出前保存。

参考引用

1、让你自己的操作系统支持鼠标做一个支持图形界面的操作系统 https://blog.csdn.net/mirkerson/article/details/7972032

GUI（图片浏览器）

技术实现

1、JPEG解码。使用作者Martin J. Fiedler实现的C++开源简易JPEG解码器，用C语言重写后作为xv6系统中读取JPG图片的模块使用。 2、原作者用一个类来保存一次解码中的数据、执行的操作，重写主要是将其内部调用的函数参数传递的逻辑在类外正确表示，修改一些用了C++特性的代码，让memcpy、memset、malloc、free等函数与xv6中的对应，留出文件读取等接口，处理内存泄漏。 3、图片浏览器。实现了jpg、bmp格式图片的播放，实现了图片的旋转、缩放的算法。

技术难点

1、需要学习JPEG图片的解码过程，按解码步骤调试。 2、与xv6系统中的函数对应起来。 3、Gui内核实现较复杂，因此没有重新实现gui内核，而是在五字班基础上使用了XV6_GUI*吴呈*魏佳辰_刘朋组的内核。 4、不同格式的图片编码方式不同，解压算法复杂，没有实现png等格式的图片

成员分工

- 太志伟：资料搜索、重写、调试
- 秦海强：资料搜索、调试
- 蔡文涛：图片浏览器、gui内核
- 陈年：图片浏览器、文档查询

参考引用

1、Martin J. Fiedler. NanoJPEG: a compact JPEG decoder. http://keyj.emphy.de/nanojpeg/. 2009-4-29

2、shakingwaves. JPEG解码原理详解. https://blog.csdn.net/shakingwaves/article/details/77816239. 2017-9-3

GUI（音乐播放器）

技术实现详情

大文件的载入： `Fs.c` 中的 `bmap` 函数中，将 `inodes` 改成三层索引结构， `mkfs.c` 中 `balloc` 和 `iappend` 函数也做了相应的修改

音乐播放部分： `math.h, math.c, common.h, common.c` 是一些底层的函数 `sound.c, sound.h`：声卡的初始化以及将音频缓冲区转存入声卡缓冲区进行播放的一些函数

`sysaudio.c`：音乐播放相关的系统调用 主要函数 `sys_writeSoundBuf`：将wav数据写入音频缓冲区 `sys_wavSectionPlay`：等待新的音频缓冲区wav数据。等待期间阻塞，一旦有新的数据，就被唤醒，接着等待声卡空闲时转存入声卡的缓冲区

`wavBufPlay.c`：循环调用 `wavSectionPlay`

`playwav.c`：循环读取wav数据，调用 `writeSoundBuf` 存入系统音频缓冲区，并唤醒 `wavSectionPlay` 等待声卡空闲时转存入声卡的缓冲区

`huffman.h,Huffman.c,decodemp3.h,decodemp3.c`：Mp3播放的一些底层 `mp3decodePlay.c, playmp3.c`：mp3播放 `playmp3` 进程中，读入数据，同时调用进程mp3decodePlay进行解码播放 GUI部分：（若要用虚拟机运行，要将编辑->首选项->输入中鼠标的优化调为始终） API接口部分：

> `PVCConst.h`：包含各类常量的定义以及消息类型的定义等。
>
> `PVCObject.h`：包含API中各类结构体和类型的定义。
>
> `PVCPainter.h和PVCPainter.c`：各类绘制函数的实现。
>
> `PVCLib.h和PVCLib.c`：用户库。包括一些数学计算函数、字符串格式化输入函数和简单随机函数的实现。 `PVCGui.h和PVCGui.c`：操作系统内核部分GUI的实现，包括窗口链表和计时器链表的维护、各类系统调用处理程序、双缓冲和局部绘制、以及内核初始化的实现。 `PVCWindow.h和PVCWindow.c`：窗口的实现。包含窗口操作的接口、默认窗口样式、窗口消息传递链和默认窗口消息处理程序的实现。 `PVCControl.h和PVCControl.c`：窗口控件的实现。包含控件的接口、默认控件样式、默认控件消息处理和一些系统控件的实现。实现的系统控件包括：自定义控件、按钮、单行编辑框、多行编辑框、菜单、菜单栏、滑动条。 `PVCDialog.h和PVCDialog.c`：对话框的实现。包含对话框操作接口、默认对话框样式、默认对话框消息处理程序和一些系统对话框的实现。实现的系统对话框包括：自定义对话框、消息盒对话框、获取输入对话框。 `PVC.h`：包含上述各API接口模块的头文件（PVCGUI.h除外），以及xv6用户程序所需的其他头文件。编写图形界面用户程序时包含此文件即可。

用户程序部分：

> `PVCDesktop.h和PVCDesktop.c`：桌面的实现。
>
> `PVCMP3.h和PVCMP3.c`：播放器的GUI.

实现难点

1． 计划是将原来的代码整合到新的xv6版本中，有着很大的工作量，要对音频播放和GUI的代码都要仔细阅读，做一定修改才能够转移到新的xv6版本中运行。而整合的工作分配给一位同学进行，要看的代码量巨大，任务繁重。文件管理方面的代码（主要是inode的三层索引）整合过程中还遇到了许多BUG。 2． 我们不是在原有的音乐播放器GUI中来继续做。我们用的是五字班学长的一份很优秀的GUI底层，但GUI的工作也只分配到一个组员，既要看明白底层的实现和使用，又要重新写音乐播放窗口，任务也很繁重。将音乐播放的底层和GUI配合起来也要一定的调试。 3． 三字班沿用的是二字班的wav播放，而二字班文档中提到在 Ubuntu 14.04 上运行无法播放声音。而三字班的MP3播放只有在Ubuntu 12.10 或者Ubuntu 10.04 32位系统上才能正常播放，说明音乐播放和Ubuntu的版本还有着很大的关系。因此，优化的两位组员努力想要实现在Ubuntu16上能够播放。但工作过于艰巨，最后没能够完成。

成员分工

- 许逸凡：音乐播放和GUI代码整合，组内工作协调，文档撰写。
- 李岚芃：音乐播放器GUI界面的开发
- 喻琳颖、周耀华：音乐播放器的优化

Merge组

由于方案二是一个比较大的、分工合作的项目，所以为了协调各组分工，将各个组工作有机地整合在一起，我们设立了Merge组。Merge组工作主要分为两部分

协调各组工作

1. 前期协调分配各个组选题，并帮助各个组统一实现细节，例如帮助实现GUI的三个组统一GUI内核。
2. 中期保持定期开会的惯例，督促各组进度，保证大组的整体进度。

合并代码&测试

为各个组预先设立好了分支，将各个组的代码先合并到小的分支上，然后逐步合并到master分支中。例如，我们将做shell命令（程嘉梁组）、任务管理器（顾国驰组）、文件检索（赵哲晖组）的代码全都合并到shell分治中，然后再把它和其他做内存管理、标准库移植的组合并到final_shell分治中，最后加到master里。

对于合并好的代码，Merge组会检查其基本功能的实现情况，进行单元测试。例如，在加入登陆功能后，我们通过单元测试发现，某些用户名和密码不能正确的读取，经过阅读源码发现，读取时用户名和密码的指定长度使用了同一个变量，造成过长的密码被截取。

有时，多个组之间的代码也会有冲突bug。例如，xv6原本指定的文件系统块数为1000，两个组的代码分别可以正常运行，但是合并之后就会超过1000的限制，需要手动调整大小；再如，vim和任务管理器的实现都对consle.c文件进行了大量的修改，简单的合并操作会造成相互之间的功能的影响，所以需要各个部分的负责人，从逻辑层面对代码进行合并。

遇到的困难

合并两份代码很难保证不会有一些隐藏的问题，这些问题不会影响新加部分的功能，却会对之前的一些功能造成影响。由于我们很难做到一次合并检查所有的之前功能，所以这些问题往往不能被立即发现。而之后发现时，很难快速准确地找到问题的关键，并寻找到合适的负责人解决问题。

成员分工

- 王泽宇：前期的工作分配、中期督促进度，测试，简单的bug fix
- 游凯超：合并代码，测试，定位bug，协调相关组合作debug

- 纳鑫：测试，阅读其他组的代码，定位bug，bug fix
- 丁郑：合并代码，测试，阅读其他组的代码，bug fix

（由于Merge组的特殊性，组内分工并不是十分明确，几乎每一个人的工作都有重合部分）

Git链接

这里是我们的Git仓库。