刘峻琳 彭友 李映辉 刘博格

2015.1.23

xv6音乐播放器之实现与改进

目录

[1 总述 1](#_Toc410071576)

[2 环境配置 1](#_Toc410071577)

[3 文件系统的修改 2](#_Toc410071578)

[4 硬件初始化 3](#_Toc410071579)

[5 MP3格式数据解码 4](#_Toc410071580)

[6 难点与关键问题的解决 6](#_Toc410071581)

[6.1 MP3解码算法的正确性验证 6](#_Toc410071582)

[6.2 改进代码使之适应xv6的用户栈大小 6](#_Toc410071583)

[6.3 GCC4.4不支持bool类型 6](#_Toc410071584)

[6.4 变量名混用 7](#_Toc410071585)

[6.5 GCC 4.4指针判断异常 7](#_Toc410071586)

[7 数学函数的实现 7](#_Toc410071587)

[8 多级缓冲的多进程播放 8](#_Toc410071588)

[8.1 多进程实现 8](#_Toc410071589)

[8.2 一级缓冲原理：共享内存 9](#_Toc410071590)

[8.3 一级缓冲的具体实现 9](#_Toc410071591)

[8.4 二级缓冲原理：缓存播放 10](#_Toc410071592)

[8.5 二级缓冲的具体实现 10](#_Toc410071593)

[9 成果总结 10](#_Toc410071594)

[10 实验感想与经验 11](#_Toc410071595)

# 1 总述

在本次实验中，我们以上届学长代码为基础，力求在xv6中实现音乐播放器的功能。上一版的代码修改了xv6操作系统的文件系统，使之支持足够大的音乐文件，此外还实现了声卡驱动。

然而上一版的代码仍有如下不足：WAV格式音乐只能在高配电脑上流畅播放、MP3根本无法播放（存在较多BUG）以及没有真正实现多进程调度。我们的目标就是在这几个方面对播放器做出改进。

# 2 环境配置

一级虚拟机：Ubuntu 12.04或10.04（不支持14.04）

二级虚拟机：QEMU

编译器：GCC 4.4

运行步骤：

1）安装GCC 4.4，使用命令sudo apt-get install gcc-4.4

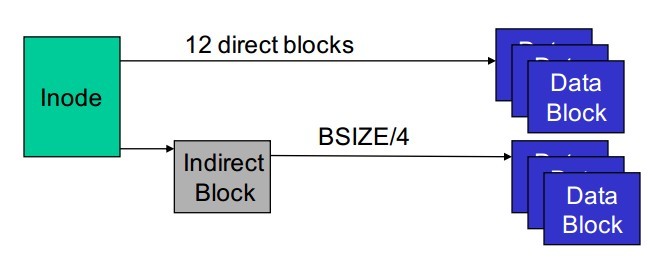
2）安装QEMU，使用命令sudo apt-get install qemu

3）进入code文件夹，使用命令make qemu启动xv6

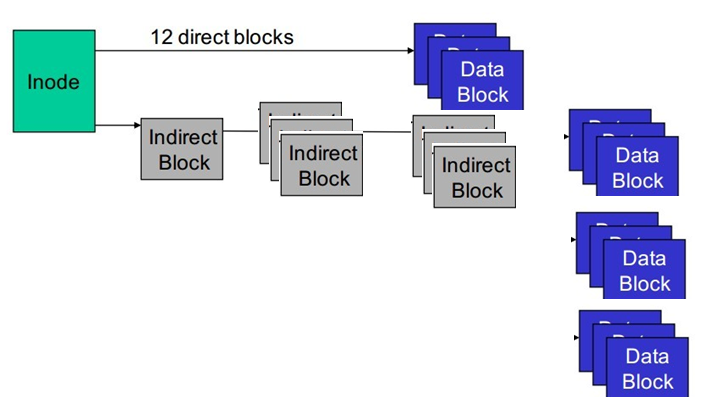
4）使用命令 play qian.wav即可收听千与千寻的WAV格式音乐；使用命令playmp3 in.mp3可收听You Raise Me Up的MP3格式音乐。如需加载其它音乐，应在Makefile文件第157行添加相应的音乐文件。另需注意，为了方便调试目前代码在playmp3.c第435行将MP3的播放文件写死为in.mp3，若要播放其它音乐，此处仍需修改，相信聪明的你一定知道怎么改成播放输入的歌曲。

# 3 文件系统的修改

基本上用的是上一届的代码，所以下面的说明也是参照他们的文档：Xv6的文件系统只能支持最大70K的文件，必须修改Xv6的文件系统。经过修改，Xv6的文件系统有512MB的磁盘空间，最大能够支持128 \* 128 \* 128 \* 0.5KB + 6KB = 1GB的文件。原生的Xv6系统结构如图：



每一个文件系统有12+128个DataBlock指针。修改后，文件系统的结构如图：

 还需要在mkfs.img中进行相应的修改。XV6编译自己的文件系统的时候，将系统的程序编译为可执行文件，然后将程序拷贝到文件系统中。我们按照我们设计的文件系统在mkfs.c中将总的block数目设置为1024 \* 1024， 即512MB。同时我们修改了iappend和balloc的逻辑，使得在创建文件系统时，系统以该种方式进行存储，这样在我们的文件系统就可以使用这个系统了。文件系统的修改逻辑见fs.c。

C:\Users\Xu\AppData\Roaming\Tencent\Users\2290826962\QQ\WinTemp\RichOle\W]S9VNCPM5Q4I8EI7{IHVC9.jpg 为了能够像xv6中添加文件，只要修改Makefile即可，如图将要添加的文件放到在Makefile中即可。

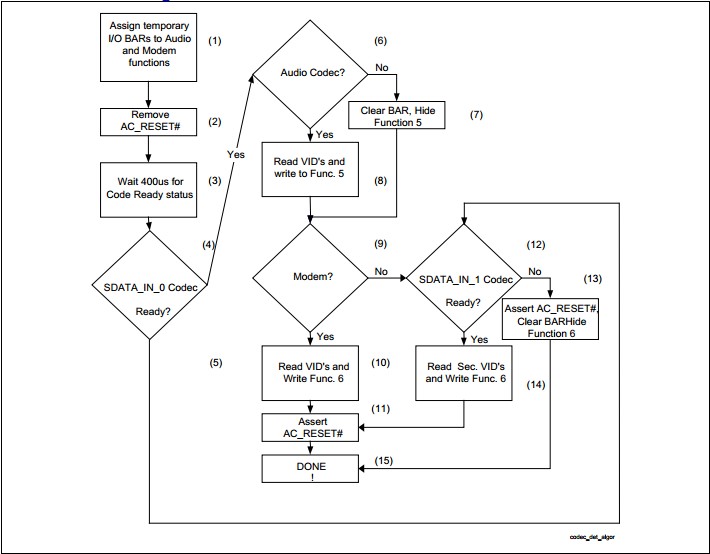
# 4 硬件初始化

这部分也基本上用的上一阶段的代码，在qemu虚拟机中实现xv6 ac97声卡驱动，分为以下几步：

1）让qemu提供虚拟的声卡。实现这个步骤需要在用qemu运行xv6时输入以下的参数：qemu –soundhw ac97 –hda xv6.img –hdb fs.img

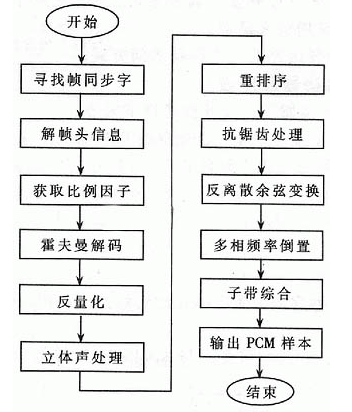
2）找到声卡设备在PCI的总线地址。PCI 配置空间为CPU控制外部设备提供了方便。在操作系统中控制声卡，首先需要找到PCI的设备。做法是询问所有PCI地址的标识，找到与AC97匹配的一项（0x24158086），该地址就是声卡在PCI配置空间的地址。具体代码实现见pci.c (soundinit函数)。

3）声卡初始化：查阅[Intel 82801AA AC'97 programmer's manual](http://wjyl.sf.net/resource/intel_82801AA_programmer's_manual.pdf" \o "http://wjyl.sf.net/resource/intel_82801AA_programmer's_manual.pdf)，可以获得ac97声卡初始化的流程。在这里，我们仅仅需要实现1-8步，就可以成功初始化声卡。每一步需要写入PCI控制段和数据段的地址可以从该文献中获得。具体代码见sound.c(soundcardinit函数)。



# 5 MP3格式数据解码

为了播放MP3，我们需要对该格式的数据文件执行一套非常繁锁的解码流程（即解压缩），将之解为原始PCM数据送入声卡。这一部分难度极大，但是好在网上有着比较成熟的开源程序，因此我们借鉴了开源程序。解码流程分为以下若干个步骤：



解码的主流程在mp3dec.c中实现，在该文件中调用各个模块的函数接口。而各个函数的具体实现在decodemp3.c和common.c中。各函数功能如下：

seek\_sync：寻找帧同步字

decode\_info：解帧头信息

hdr\_to\_frps：进一步解出帧头信息

III\_get\_side\_info：解出帧的边信息

III\_get\_scale\_factors：获取比例因子

III\_hufman\_decode：哈夫曼解码

III\_dequantize\_sample：反量化采样

III\_stereo：立体声处理

III\_reorder：重排序

III\_antialias：抗锯齿处理

III\_hybrid：反离散余弦变换

SubBandSynthesis：子带合成

最终解出的PCM数据保存在数组pcm\_sample中，调用out\_fifo函数将其送入声卡缓冲区从而播放出声音。

# 6 难点与关键问题的解决

这部分是我们工作的重点，第3、4、5部分是实现MP3播放器的三个非常重要的基础步骤，然而仅仅完成还不够，还必须要保证正确性。在上一版的播放器中，播放MP3的一些基本bug并未得到解决，因此我们的任务是要对出现的问题一一进行解决。

## 6.1 MP3解码算法的正确性验证

MP3解码算法非常复杂，我们沿用了上一版代码中引入的开源代码。由于上一版的MP3根本无法播放，因此我们首先需要验证这一套开源代码的正确性。我们找到了该开源代码的windows版本，在windows下对MP3文件进行了解码，将解码得到的PCM数据导入Audacity软件，声音如期播放，因此该开源代码的正确性得到验证，接下来我们要做的是就是让它在xv6上跑起来。

## 6.2 改进代码使之适应xv6的用户栈大小

在上一版的xv6中，运行MP3播放进程会遇到系统抛出的异常，如果使用printf进行单步调试，则printf的数量和位置将会对系统行为造成影响，具体表现为printf会随机输出一些乱码并且不同的pritnf，系统抛出的异常会不同。经过不断地分析思考和查阅xv6官方文档，我们发现并确定造成该问题的原因是xv6的用户栈大小只有一页大小（4KB）且不能动态增长，而整套解码流程所需要的栈开销远远大于4KB，主要体现在多处局部二维数组的定义上。因此，我们小心翼翼地修改整个算法的代码，尽最大可能减少其对栈的利用，把大的二维数组放到了全局变量中。完成该项工作后，系统异常没有了，但是我们并没有得到正确的解码结果。

## 6.3 GCC4.4不支持bool类型

在整个项目中，我们使用的编译器版本是gcc 4.4，是一个非常早的版本，不支持bool类型的变量。因此，代码采用的办法是用宏定义的TRUE来代替布尔值，将TRUE定义为1。这就带来了问题，在有布尔值的情况下，任何非0的整型数和true都是等价的，然而和宏定义的TRUE并不等价。但是在解码流程中，有大量用非0整型数和TRUE作比较的地方，因此会带来一些错误结果。尽管找到这个问题并不容易，但是解决它却非常简单，只需直接将非0数作为条件判断句就可以了，不需要将它和TRUE作比较。

## 6.4 变量名混用

上一版的代码对解码算法稍稍做了修改，在修改的过程中出现了变量名混用的情况，我们对这样的情况做了修改。例如在sysaudio.c代码的第452行，在上一版的代码中，corebuf.buf被写成了buf，前一个是读文件和解码的缓冲区，而后一个buf是解码和播放的缓冲区，两者写混必定会造成程序运行时错误，然而在编译阶段却并不会报错。

## 6.5 GCC 4.4指针判断异常

在代码的某些地方，我们发现gcc4.4对某些空指针和NULL的比较出现了问题，指针明明是空的，但是和NULL（宏定义为0）比较时却判断为不相等，因此造成了代码执行流程出现了混乱。对于这类情况，我们同样小心翼翼地修改了代码，但却保持了原有算法逻辑，从而解决了问题。

# 7 数学函数的实现

MP3的解码算法需要用到大量浮点数的数学运算，然而xv6里并没有任何现成的数学库的支持。如果引入C的标准库，则其相互依赖太多，引进难度非常大。而自己手动实现数学函数根本无法达到需要的精度和速度。

为此，我们翻遍了互联网上的资源，终于找到了一些可用的开源代码，再加上自己写的一点代码，提供了一套用内联汇编实现的数学函数。

对这套数学函数，我们必须对它进行足够的精度和速度测试。我们用千万数量级的测试数据进行了测试，该套数学函数精度可以达到10-12，速度大概为标准库的1.5倍左右，已经达到了可以用的性能。因此，我们可以放心地引入。

整套数学函数的实现在math.c，实现的数学函数有：sin、cos、tan、log2、ln、pow、sqrt、exp。

# 8 多级缓冲的多进程播放

在解决了以上诸多难题后，MP3的音乐终于可以播放出来了！然而目前的播放仍是单进程的，我们需要实现真正的多进程播放。为此，我们设计了多级缓冲的多进程播放机制。实现了多级缓冲之后，WAV格式的音乐在一台低配电脑上也能流畅播放了！

## 8.1 多进程实现

MP3的播放共涉及三个进程：读取文件进程（playmp3.c）、解码进程（mp3dec.c）、播放进程（decode.c），同时也涉及到进程间的通信（sysaudio.c）。

WAV的播放则相对简单一些，只涉及两个进程：读取文件进程（play.c）和播放进程（decode.c）。

这里，我们设计了多个系统调用来满足复杂的多进程播放程序：

setSampleRate：设置声卡采样率

kwrite：将裸的音频数据写入播放队列（二级缓冲队列）

wavdecode：从播放队列中取出数据进行播放

beginDecode：将读文件相关参数放入一级缓冲队列

waitForDecode：从一级缓冲队列取出相关参数

getCoreBuf：对一级缓冲队列的数据队列进行读写操作

* WAV播放逻辑：

play.c

循环从文件读取数据，调用kwrite将数据写入音频缓冲区队尾。

decode.c

调用wavdecode不断从音频缓冲区读出固定大小的数据块送入声卡。若暂时没有数据，则等待。

* MP3播放逻辑：

playmp3.c

循环从MP3文件中读取数据，每读完一帧数据则调用beginDecode将相关参数存入参数缓冲队列，此时桢数据已经保存在了数据缓冲队列corebuf.buf中。

mp3dec.c

不断从参数缓冲队列中取出所需要的参数，若没有，则阻塞；否则，开始当前桢的解码工作，从corebuf.buf中读取桢数据，并执行前述若干复杂的解码流程。每解完一桢数据，则调用out\_fifo将得到的PCM数据写入声音播放缓冲队列。

decode.c

播放进程，与WAV相同。

## 8.2 一级缓冲原理：共享内存

一级缓冲实际是共享内存的实现，读取文件的进程不断从数据文件一帧一帧地读取音乐数据，并将其送入共享缓冲区队尾。解码进程不断从队首读出一帧数据，对其进行解码操作。同时为了避免读文件的进程执行得过快，我们设置了一个阈值1000，即读文件的速度不能比解码的速度快过1000帧，否则将无法有足够大的缓冲区装下数据。同时，共享内存就会涉及对临界区的访问，我们也实现了互斥锁来保护临界区数据。

## 8.3 一级缓冲的具体实现

共享缓冲区实际上一个定义在sysaudio.c中的corebuf结构体。该结构体的定义在sound.h中。一个corebuf结构体中包含如下的一些域：

* + buf：缓冲区数组
  + buf\_byte\_idx：缓冲区的字节索引，表示当前读到了缓冲区的第几个字节
  + buf\_bit\_idx：缓冲区的位索引，假设当前缓冲区读到了第x个字节，那么这个域表示读到了第x个字节的第几个bit上
  + offset：缓冲区偏移量，表示当前写到了缓冲区的第几个字节
  + totbit：缓冲区读计数量，表示当前一共从缓冲区读取了多少个位
  + fr\_ps、III\_side\_info：读文件进程和解码进程通信所传递的参数

对缓冲区的操作种类有（common.c）：

* + hsstell：返回缓冲区当前的totbit值
  + hgetbits：从缓冲区的读指针处开始，读出N个位的值，并返回其对应的整型值
  + hget1bit：从缓冲区的读指针处开始，读出1个位的值，并返回其对应的整型值
  + rewindNbytes：将缓冲区的读指针回退N个字节（N\*8个位）
  + rewidnNbits：将缓冲区的读指针回退N个位

由于缓冲区实际是一段共享内存，因此以上的操作被我们封装成为一个系统调用sys\_getCoreBuf(type, para)，写在sysaudio.c中，对以上函数的调用都会转化为对getCoreBuf的系统调用。不同的操作种类以不同的type值表示，para则表示该类操作下的参数。

## 8.4 二级缓冲原理：缓存播放

二级缓冲同样也是用队列实现，解码进程不断将将码结果送入待播放队列，而播放的进程则不断从该队列读取固定大小的块数据，并将其送入声卡。有一个4096\*8字节大小的缓存区作为循环队列，头指针head（代码中用变量in表示）指向，尾指针tail（代码中用变量out表示）分别指向了当前读入读出的起始字节。读入的时候只要不超过剩余缓存大小都可以，读出是以4096字节为一个块进行操作。这样读入和读出就实现了分离，从而起到了IO繁忙和CPU繁忙的进程独立运行的局面。

## 8.5 二级缓冲的具体实现

详细代码参见sysaudio.c，主要涉及的代码解释：

1）宏定义：

* + IN\_OUT 表示整个缓冲队列的块的数目
  + BLOCK\_SIZE 缓冲队列每个块的字节数

2）变量：

* + buf数组：大小为IN\_OUT\*BLOCK\_SIZE表示整个缓冲队列。
  + inNum：当前存入缓冲队列的下标。
  + in：当前存入缓冲队列的块下标。
  + out: 当前读出缓冲队列的块下标。
  + full：0表示当前缓冲队列未满，1表示已经满了。

3）函数：主要涉及了两个系统调用sys\_wavdecode\_wav和sys\_kwrite\_wav。其中sys\_wavdecode\_wav是用来读取缓冲队列中的数据，然后放到声卡缓冲区里面播放。sys\_kwrite\_wav是用来把裸的音频写入到缓冲队列中。

# 9 成果总结

综上所述，在我们的实验中，我们完成了如下成果：

* + 解决了MP3在xv6上播放遇到的诸多问题和难点，使之达到接近流畅的播放程度
  + 实现了真正的多进程协同播放
  + 实现了多级的缓冲机制，使WAV格式的音乐可以在一台低配电脑上非常流畅地播放

# 10 实验感想与经验

这次操作系统的大作业我们选择了MP3播放，众所周知MP3播放难度较大，本身是想对自我的一次挑战，我们也希望能够做出来一个比较有意义的作品。

在xv6中做测试是一件不太愉快的事情，因为基本上只能用printf语句进行测试，而且每次编译的过程都比较慢。所以，一台性能比较好的电脑是比较重要的（虚拟机的话可以试着把所有核心都分配给虚拟机，我们用了小组里性能最好的一台笔记本，处理为i7，把所有核心都分给虚拟机以后原系统其实也是可以正常运行的），这样可以大大提高编译的速度，节省很多测试的时间。

测试的时候我们采用了单元测试的方法。因为我们是在上一届的基础上修改的，而上一届的代码本身就是错误的，所以在定位错误出现位置的过程中，我们都是对代码进行一块一块的测试，这样比较费时间，但是我们暂时没有更好的办法。

同时，由于上一届代码本身是有问题的，所以我们在开发的过程中就带着很多的问题：解码流程是否正确？数学函数是否满足要求？多进程的通信写对了没有……问题之间相互牵扯，在这种怀疑一切的心态下很难进行突破。于是我们采取了这样的策略：首先，我们在windows下写出正确的解码流程，然后把这个流程迁移到xv6中进行对比，发现数学函数的精度有问题，修复这个问题以后在xv6下就可以正确地解码了；然后，我们再在xv6下实现单进程的MP3播放；最后实现的多进程的播放。经过这样的步骤，我们每一个阶段的目标都很明确，效率也非常的高。

另外，测试只能够定位到问题出现的地方，不一定能够提供足够的改进信息。有一些错误必须要对xv6内部机制有足够的了解以后才可以解决。

最后说一下优化的问题。我们优化的主要有三个地方：

第一是数学函数。我们的数学函数是利用内联汇编实现的，在精度上以及速度上都可以和库函数媲美，比上一届用C语言实现的数学函数快了很多倍，在精度上也提高了很多。如果下一届有一些库函数调用需要自己实现的话，我们推荐优先考虑汇编实现，因为汇编比C快了不知多少倍（有一些运算比如三角函数，英特尔的CPU是有对应的指令的）。

第二是IO优化。这里主要是说getbits()函数，因为read()的速度还是挺快的。我们把getbits()函数的读取速度从几百K提升到了数兆，解决的IO的瓶颈。之前读取速度慢的原因是getbits在读取磁盘数据的时候每次只读8byte的数据，大大增加了磁盘访问次数。

第三个优化也说不上是优化，只是我们提升了xv6的硬件平台。qemu是一个虚拟机，在Makefile中可以指定虚拟机分配的CPU数量以及内存大小。我们把CPU数量从双核提升到四核后，播放的速度提升了3倍，我个人猜测可能是因为播放MP3需要三个进程，只用双核CPU的话没有办法做到真正的并行。

总的来说，实现MP3播放的的确确是十分困难的。面对这样的困难，一个良好的团队不可或缺。整个过程中，我们一起拼搏，攻克了一个又一个的难关，真可谓是山重水复疑无路，柳暗花明又一村。最后有幸播放出来MP3，也算是对我们的辛苦攻关有了一个满意的交代。