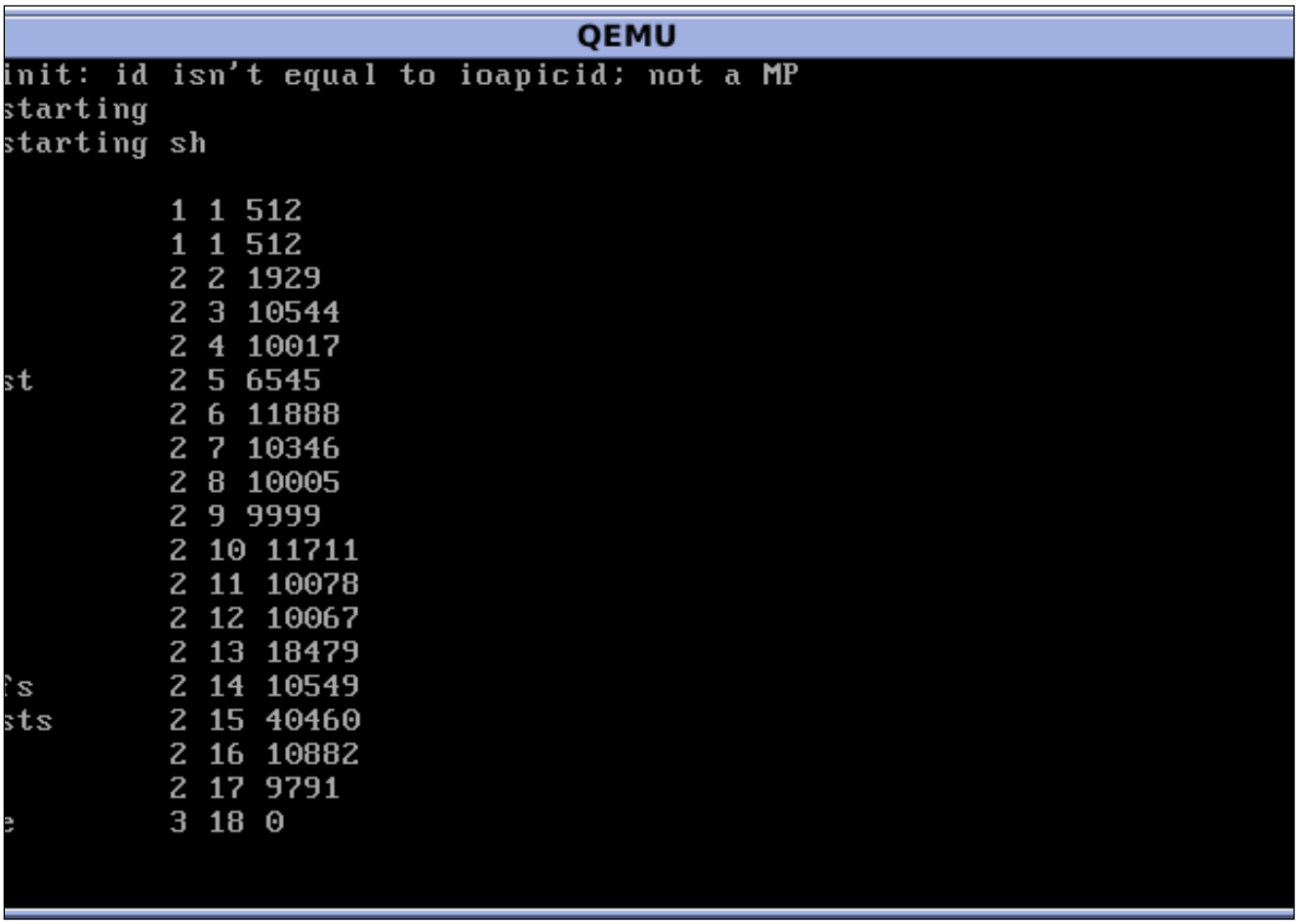
XV6 实验报告

基于xv6的mp3音频解码与播放

卢凌铜 肖剑楠 董亮 杨志强 - January 23, 2015



导语

本次实验，我们小组在0字班与1字班的基础上。进一步实现了xv6平台上，对于mp3文件的解码与播放。

关于音频的播放，主要分为以下几个模块

- 硬件初始化
- 文件系统修改
- 播放逻辑与进程通信

其中，前两个模块的工作几乎完全基于前两届学长的工作。在这里做简要陈述：

1. 硬件初始化　绝大多数笔记本的声卡硬件使用的是Intel旗下的ac97声卡。为了使xv6能够正常调用硬件声卡。首先我们需要在qemu虚拟机里面实现其声卡驱动。具体过程时找到相应设备的PCI总线地址，找到ac97相应的位置。最后在Makefile里添加相应的虚拟声卡代码段，即可成功配置。详细流程可参见上一届文档，这里不再赘述。

```
QEMUOPTS = -soundhw ac97 -hdb fs.img -hda xv6.img -smp $(CPUS)
```

2. 文件系统　原生xv6文件系统只支持很小的文件。这里需要对其文件系统做比较系统的修改（具体修改见fs.c 以及 mkfs.c ）。另外，在需要添加某个文件时，只需要在Makefile里做相应的添加即可。详细的修改流程请见上一届文档。

```
fs.img: mkfs README $(UPROGS)
    ./mkfs fs.img README in.mp3 output.pcm dewindow.txt huffdec.txt $(UPROGS)
```

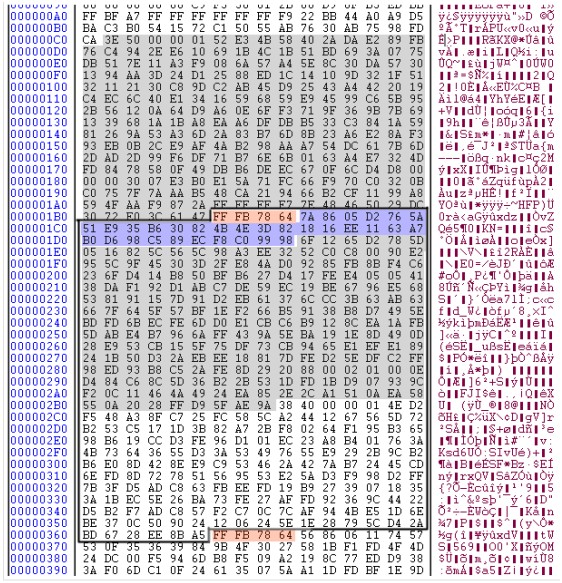
接下来的播放逻辑与进程控制将在接下来几章里做比较详细的讲解。我们将从mp3解码、播放两个模块讲解。（值得注意的一点是，由于本组作业不涉及wav文件的播放。在我们的代码中移除了wav播放的相应代码段。具体wav播放及其流程可以观看上一届学长的文档和代码）

mp3解码

在说到mp3解码时就必须要谈及mp3编码的相关知识。（相关mp3解码的知识见我的印象笔记https://app.yinxiang.com/l/ACRp1caV3s1PSJttjR14R2HvCHlQXysNN4Q）

mp3编码的中心思想是人耳其听觉范围局限在频率为20Hz到20kHz区段。而音频系统另一个重要的特点是某些音频信号会”掩盖“其相邻信号，我们就可以通过移除音频文件中一些对最终听觉效果没有影响或影响不大的音频信号，最终实现压缩文件体积的目的。

介绍完编码之后进入解码。而进入解码的第一件事就是将文件从物理帧转换为逻辑帧：



如图所示即是mp3的逻辑帧：分为header、side information、main data。header存储了采样率和声道信息，side information存储了主数据的编码信息以及方式，main data则存储了音频信息。

为了进一步压缩体积，mp3文件普遍采用了Huffman 编码。作为解码的重要一步，如何理解运用Huffman来进行相关解码操作也十分重要。

解码完成之后我们就可以获得计算机可以直接播放的源文件PCM。

本次解码的主代码来源于libmad（其是一款开源的mp3解码库，被广泛运用与各个播放器领域 http://www.underbit.com/products/mad/）。这个库拥有良好的mp3解码性质譬如：24bit PCM输出，百分百整数运算，基于ISO/IEC标准。我们此次作业基于这个库做了移植

与整合。（由于libmad本身学习成本较高，但是有着良好的学习受益。这里推荐两个中文文档方便学弟学习： http://blog.chinaunix.net/uid-29068482-id-4130006.html http://wenku.baidu.com/view/e9420e6d9b6648d7c1c7469e.html ）

代码移植与整合的一个难点在于libmad本身基于C标准库，这个在xv6下是缺失的，因此需要去掉一些代码并重新思考实现方式。

譬如Huffman树的读取，以前的实现方式是基于文件读取。而由于相关函数缺失我们采取了直接植入源代码的方式：

```
HUFFBITS dmask = (HUFFBITS)1 << (sizeof(HUFFBITS)*8-1);

unsigned char h1[7][2] = {{ 0x2, 0x1 }, { 0x0, 0x0 }, { 0x2, 0x1 }, { 0x0, 0x10 }, { 0x2, 0x1 }, { 0x0, 0x1 }, { 0x0, 0x11
};
unsigned char h2[17][2] = {{ 0x2, 0x1 }, { 0x0, 0x0 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x10 }, { 0x0, 0x1 }, { 0x2, 0x1
 { 0x0, 0x11 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x20 }, { 0x0, 0x21 }, { 0x2, 0x1 }, { 0x0, 0x12 }, { 0x2, 0x1 }, { 0x0,
x2 }, { 0x0, 0x22 }};
unsigned char h3[17][2] = {{ 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x0 }, { 0x0, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x11 }, { 0x2, 0x1
 { 0x0, 0x10 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x20 }, { 0x0, 0x21 }, { 0x2, 0x1 }, { 0x0, 0x12 }, { 0x2, 0x1 }, { 0x0,
x2 }, { 0x0, 0x22 }};
unsigned char h5[31][2] = {{ 0x2, 0x1 }, { 0x0, 0x0 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x10 }, { 0x0, 0x1 }, { 0x2, 0x1
 { 0x0, 0x11 }, { 0x8, 0x1 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x20 }, { 0x0, 0x2 }, { 0x2, 0x1 }, { 0x0, 0x21 }, { 0x0,
12 }, { 0x8, 0x1 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x22 }, { 0x0, 0x30 }, { 0x2, 0x1 }, { 0x0, 0x3 }, { 0x0, 0x13 }, {
2, 0x1 }, { 0x0, 0x31 }, { 0x2, 0x1 }, { 0x0, 0x32 }, { 0x2, 0x1 }, { 0x0, 0x23 }, { 0x0, 0x33 }};
unsigned char h6[31][2] = {{ 0x6, 0x1 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x0 }, { 0x0, 0x10 }, { 0x0, 0x11 }, { 0x6, 0x1
, { 0x2, 0x1 }, { 0x0, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x20 }, { 0x0, 0x21 }, { 0x6, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x12 }, { 0x2,
x1 }, { 0x0, 0x2 }, { 0x0, 0x22 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x31 }, { 0x0, 0x13 }, { 0x4, 0x1 }, { 0x2, 0x1 }, {
0, 0x30 }, { 0x0, 0x32 }, { 0x2, 0x1 }, { 0x0, 0x23 }, { 0x2, 0x1 }, { 0x0, 0x3 }, { 0x0, 0x33 }};
unsigned char h7[71][2] = {{ 0x2, 0x1 }, { 0x0, 0x0 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x10 }, { 0x0, 0x1 }, { 0x8, 0x1
 { 0x2, 0x1 }, { 0x0, 0x11 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x20 }, { 0x0, 0x2 }, { 0x0, 0x21 }, { 0x12, 0x1 }, { 0x6,
x1 }, { 0x2, 0x1 }, { 0x0, 0x12 }, { 0x2, 0x1 }, { 0x0, 0x22 }, { 0x0, 0x30 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x31 }, {
x0, 0x13 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x3 }, { 0x0, 0x32 }, { 0x2, 0x1 }, { 0x0, 0x23 }, { 0x0, 0x4 }, { 0xa, 0x1
 { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x40 }, { 0x0, 0x41 }, { 0x2, 0x1 }, { 0x0, 0x14 }, { 0x2, 0x1 }, { 0x0, 0x42 }, { 0x0,
x24 }, { 0xc, 0x1 }, { 0x6, 0x1 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x33 }, { 0x0, 0x43 }, { 0x0, 0x50 }, { 0x4, 0x1 }, {
x2, 0x1 }, { 0x0, 0x34 }, { 0x0, 0x5 }, { 0x0, 0x51 }, { 0x6, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x15 }, { 0x2, 0x1 }, { 0x0, 0x52
, { 0x0, 0x25 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x44 }, { 0x0, 0x35 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x53 }, { 0x0
0x54 }, { 0x2, 0x1 }, { 0x0, 0x45 }, { 0x0, 0x55 }};
unsigned char h8[71][2] = {{ 0x6, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x0 }, { 0x2, 0x1 }, { 0x0, 0x10 }, { 0x0, 0x1 }, { 0x2, 0x1
 { 0x0, 0x11 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x21 }, { 0x0, 0x12 }, { 0xe, 0x1 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0,
20 }, { 0x0, 0x2 }, { 0x2, 0x1 }, { 0x0, 0x22 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x30 }, { 0x0, 0x3 }, { 0x2, 0x1 }, { 0
, 0x31 }, { 0x0, 0x13 }, { 0xe, 0x1 }, { 0x8, 0x1 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x32 }, { 0x0, 0x23 }, { 0x2, 0x1 }
{ 0x0, 0x40 }, { 0x0, 0x4 }, { 0x2, 0x1 }, { 0x0, 0x41 }, { 0x2, 0x1 }, { 0x0, 0x14 }, { 0x0, 0x42 }, { 0xc, 0x1 }, { 0x6,
1 }, { 0x2, 0x1 }, { 0x0, 0x24 }, { 0x2, 0x1 }, { 0x0, 0x33 }, { 0x0, 0x50 }, { 0x4, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x43 }, {
0, 0x34 }, { 0x0, 0x51 }, { 0x6, 0x1 }, { 0x2, 0x1 }, { 0x0, 0x15 }, { 0x2, 0x1 }, { 0x0, 0x5 }, { 0x0, 0x52 }, { 0x6, 0x1
 { 0x2, 0x1 }, { 0x0, 0x25 }, { 0x2, 0x1 }, { 0x0, 0x44 }, { 0x0, 0x35 }, { 0x2, 0x1 }, { 0x0, 0x53 }, { 0x2, 0x1 }, { 0x0,
x45 }, { 0x2, 0x1 }, { 0x0, 0x54 }, { 0x0, 0x55 }};
```

诸如此类有很多小技巧，但是从整体上破坏了代码结构的美感。因此需要与其他文件小组成员进一步深化合作，完成改进。

解码部分的详尽代码请看 play.c 里面的 decode 函数以及 decodemp3.c 相关文件。

mp3播放

播放更多的涉及到的是一些进程通信和系统调用。由于我们本次并没有完美实现mp3流畅播放（我们是解码完成之后直接播放PCM文件，因为解码的过程实在是太慢），因此这里剖析一下上一届的一些代码流程。

首先我们需要一些系统调用用以实现播放：

- setSampleRate：设置声卡采样率。
- writeaudio：传递读取的文件到内核。
- setVolume：设置声音大小。
- pause：通知内核暂停播放。
- wavdecode：开始缓存用户读取的音频数据。
- beginDecode：开始进行mp3解码。
- waitForDecode：等待上一个mp3解码完成。
- endDecode：设置mp3解码状态为已完成。
- getCoreBuf：从内核态获取音频数据用于解码。

具体的播放流程就是首先 play 函数不断读取数据调用 beginDecode函数唤醒 decode 函数用以解码。根据 decode 函数状态决定阻塞与唤醒。decode 函数则从数据共享区里面拿出解压数据，予以解码并将数据放入音频缓冲区。最后 play 函数对音频缓冲区里面的文件予以播放。

我们的代码存在一些差错导致解码和播放时会出现内存越界和缺页的错误。这个错误我们会尽快予以更正。最后实现流畅的播放。

经验与教训

在这次不完美的大作业中，我们在mp3播放里遇到了很多困难，同时也收获了很多经验。这里做简要陈述：

Debug

初期，主要的调试方法为gdb。在xv6根目录下，打开两个命令行窗口，A窗口运行make qemu-gdb，B窗口运行gdb kernel，此时在B窗口中可以通过输入b[reak] func加入断点，还可以输入c[ontinue]执行下一步，p[rint] variable输出变量值。尽管gdb在linux环境下，是一个比较完善的调试工具，然而，在对xv6进行调试时，嵌套层数很多，而且问题往往在汇编层面才显现出来，因而调试难度很大。

因而后来主要采用输出特定信息的方式来确定问题所在。在需要调试的程序中加入若干printf语句输出标志量。这是一种比较原始的方式，效率很低，每次修改输出位置或者输出信息，都需要重新编译、运行。所以后来在调试解码程序的时候，便将代码移植到VS环境下调试，当调试完成，再移植到xv6中。

Memory

在把调试好的解码程序移植到xv6环境下后，在运行时出现如下问题：

```
pid 3 play: trap 14 err 4 on cpu 1
```

在traps.h中对trap 14的定义为

```
#define T_PGFLT         14      // page fault
```

所以认为问题是CPU访问内存时，发生了缺页错误，导致程序中断。

一度怀疑是xv6的Lazy Page Allocation机制的锅，因为解码程序中创建了很多大数组，可能出现了内存不够的问题。不过很快就打消了这种顾虑，在VS下运行解码程序，所占内存一直稳定在2M以下。现在一直卡在这个问题上，没法取得新的进展，还望学弟学妹们能实现吾之未竟大业！

Lock

为了进一步解决内存问题，我们当时认为是由于核心数据没有上锁导致双CPU同时修改核心数据进而引发错误。仔细研读xv6 Lock 章节的代码，运用 spinlock 在解码前后都予以上锁。问题并没有得以解决。希望学弟学妹们不要再次尝试用锁解决问题

People

本次大作业其实也涉及到很多任务分配不够清晰，日程规划不够完善的缺陷。而xv6的代码缺乏合理有效的调试手段也一直阻碍我们的进度有明显提升。

希望以后的小组能够吸收我们的经验教训。完成好xv6的整合！

人员分工

肖剑楠	声卡资料查阅，相关库函数实现，调试
董亮	播放代码整合，调试
卢凌铜	解码代码整合，调试
杨志强	调试