# xv6 音乐播放器之实现与改进

刘峻琳 彭友 李映辉 刘博格

2015.1.23

# 目录

# 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

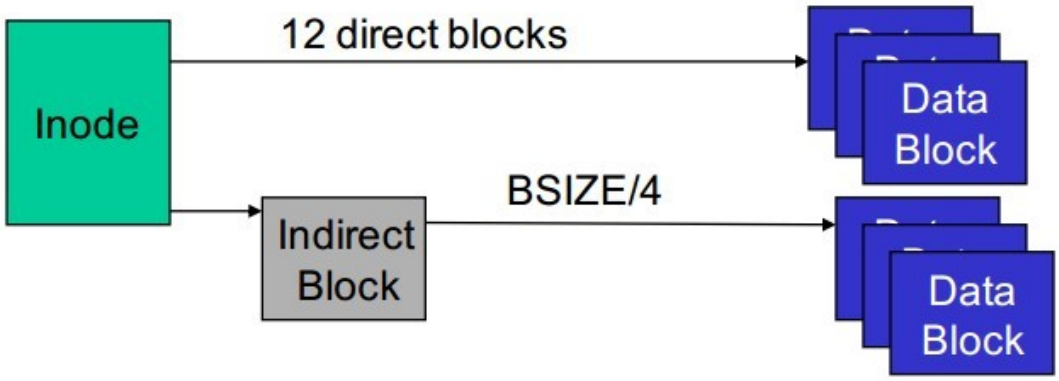
3）进入 code 文件夹，使用命令 make qemu 启动 xv6

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。

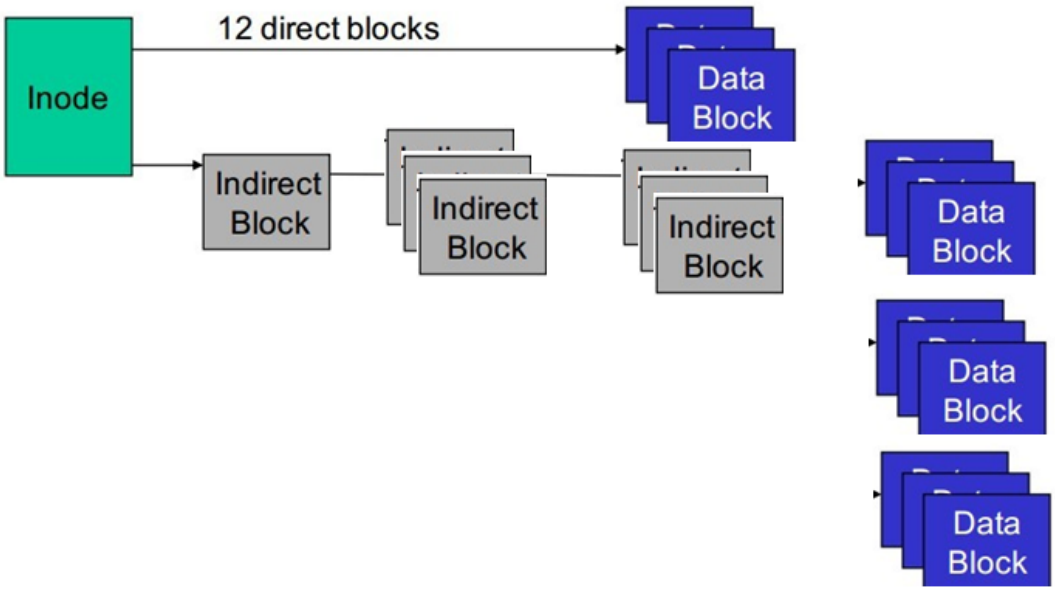
# 3 文件系统的修改

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。

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。

```
fs.img: mkfs README $(UPROGS)
        ./mkfs fs.img README $(UPROGS) wav1.wav back.wav
```

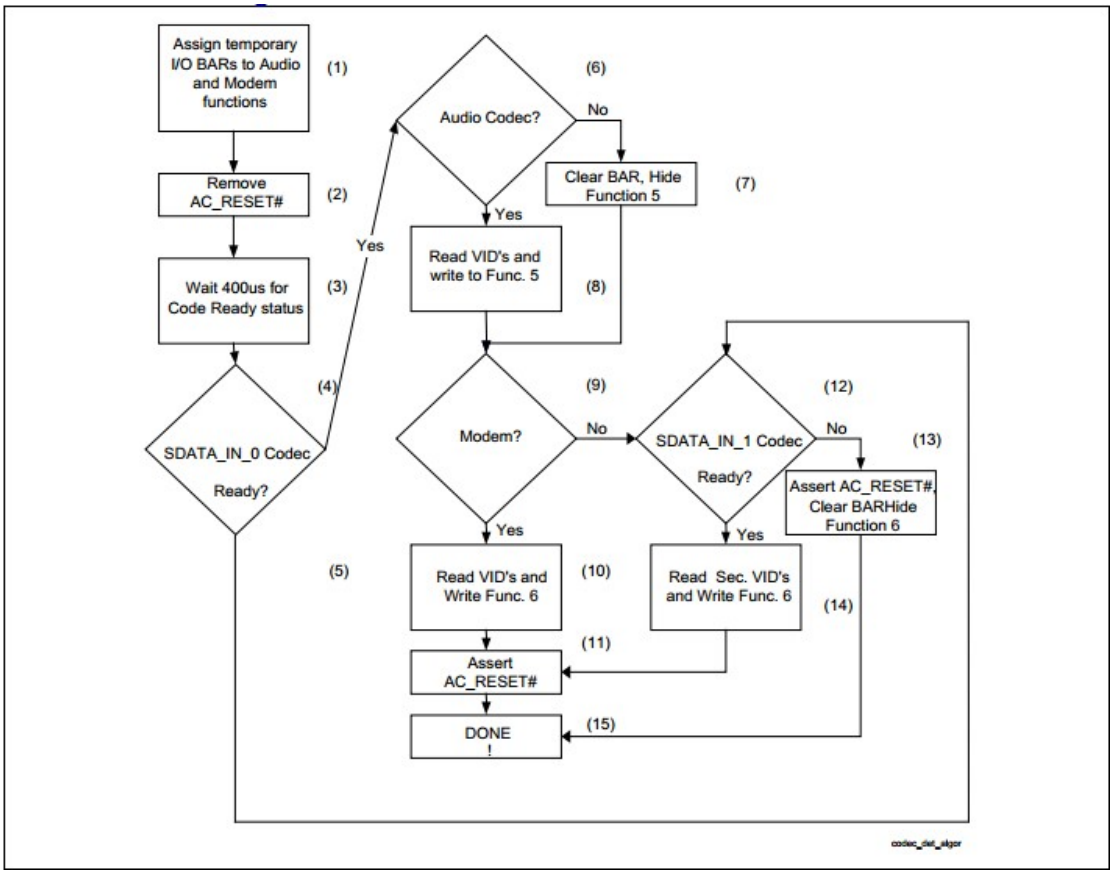
# 4 硬件初始化

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

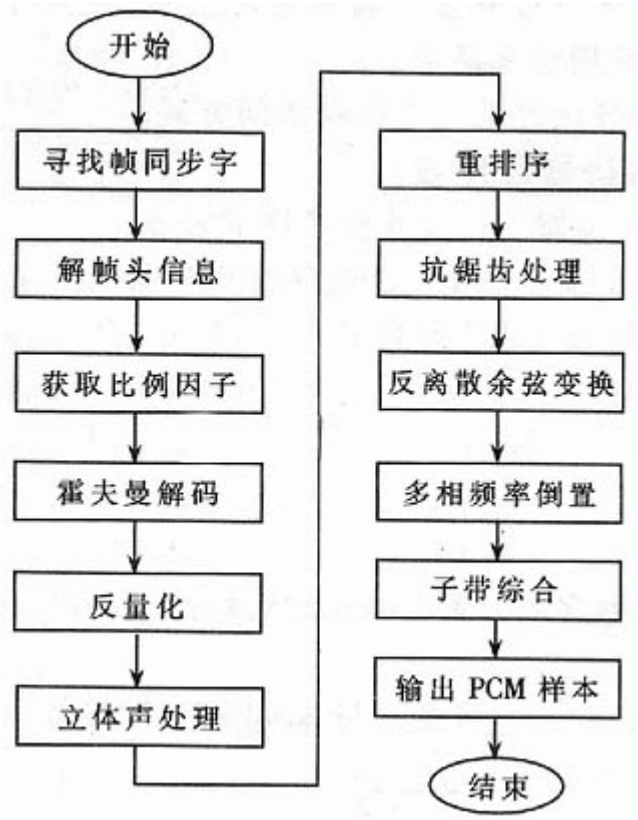
2）找到声卡设备在 PCI 的总线地址。PCI 配置空间为 CPU 控制外部设备提供了方便。在操作系统中控制声卡，首先需要找到 PCI 的设备。做法是询问所有 PCI 地址的标识，找到与 AC97 匹配的一项（0x24158086），该地址就是声卡在 PCI 配置空间的地址。具体代码实现见 pci.c (soundinit 函数)。

3）声卡初始化：查阅 Intel 82801AA AC'97 programmer's manual，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。