xv6 上的音频播放器

大作业-8：杨碧茹（组长），董文冲，闵安娜，林智鑫

一、 概述

项目说明

本项目以教学用 xv6 操作系统为基础，在其上添加了模块化的声卡驱动及音频解码库支持，在 qemu 模拟器下可正确解析 wav 文件并将声音通过虚拟声卡正常输出至主机，并实现了暂停、音量调节、变速等常用功能，拥有完善的使用提示。

任务分工

前期调研 - 全组

WAV 文件解析 - 董文冲

文件系统改进 - 闵安娜

控制器与播放库 - 林智鑫

系统调用与声卡驱动 - 杨碧茹

成果代码

仓库地址：https://github.com/wav-readers/audio-player-for-xv6

1. 考虑到 wav 文件尺寸，修改了文件系统，使其支持较大的文件；

   修改了 `Makefile`，`param.h`，`fs.c`，`fs.h`

2. 添加了声卡驱动，并将相关代码封装成系统调用；

```
+ kernel/pci.h
+ kernel/pci.c // 声卡初始化与底层接口
+ kernel/sound_card.c // 与系统调用连接的声卡中层接口
```

修改了 `kernel/main.c, kernel/defs.h, kernel/trap.c, kernel/plic.c, kernel/memlayout.h, kernel/printf.c`

添加系统调用修改了 `kernal` 下的 `syscall.h, syscall.c, usyS.pl`

3. 添加了播放器与模块化的播放库，添加新的解码库即可支持新的格式，高内聚低耦合；

```
+ user/aplay.c // 控制器(音频播放器)
+ user/aplaycore.h
+ user/aplaycore.c // 播放库核心逻辑
+ user/audio.h // 音频信息通用部分
+ user/wav.h
+ user/wav.c // wav 文件解析
```

4. 添加或修改了部分函数以辅助实现功能。

   修改了 `user/user.h, user/ulib.c`

二、 代码运行

运行环境

经测试，代码在搭载了 Ubuntu 22.04 的虚拟机中运行良好。

注：WSL 无自带音频播放功能，请使用虚拟机测试。

环境配置

```
git clone https://github.com/wav-readers/audio-player-for-xv6.git
cd audio-player-for-xv6
make qemu
```

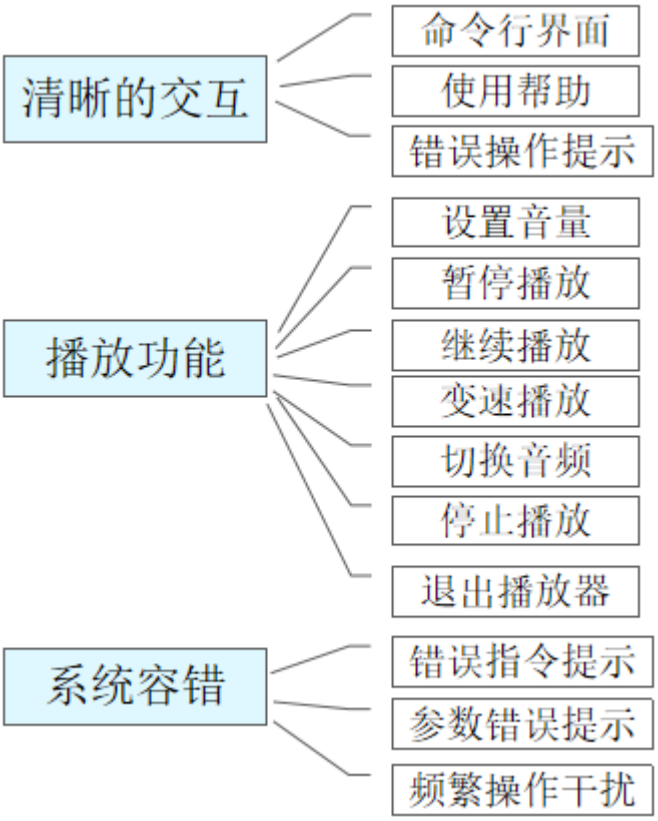
```
init: starting sh

$ aplay

>>> open test2.wav
```

三、 功能介绍

本系统的主要功能如下图所示：



对 xv6 系统的改进

本系统扩大了镜像支持文件的最大大小，单文件最大大小与文件系统总大小均约为 56MB。

音频播放

实现了对 wav 音频文件的正常播放。进入 `aplay` 后，命令是 open + target file path。

在该系统中加入额外的文件的步骤是：

1. 将合法 wav 文件加入到 user 目录下
2. 在 Makefile 中加入对应的镜像
3. make clean
4. make qemu 重新编译

更多播放功能

本系统可以对播放的音频进行设置音量，变速播放（设置范围内），切换音频，停止/继续播放，停止播放，退出音频播放器等操作。

1. 设置音量:volume + target volume
2. 变速播放:speed + target speed(float)
3. 暂停播放：pause
4. 恢复播放：resume
5. 停止播放：stop
6. 退出播放器：quit
7. 切换音频：open + target file path

完善的提示信息

进入 aplay，输入 help 后系统有完整的功能提示信息。

```
Vendor id: 0x11001af4
init: starting sh
$ aplay
Audio Player 1.0
Type "help" for more information.
>>> help
open:
    open <file path>       open the file and play.
                           File already open will be closed first.
volume: default max volume is 100.
    volume                 print current volume
    volume <target volume> set volume
speed: default speed is 1.0
    speed <target speed>   set play speed
pause:
    pause                  pause playing
resume:
    resume                 resume playing
stop:
    stop                   stop playing
quit:
    quit                   quit the Audio Player
>>>
```

系统有一定容错能力，对于不合法输入（错误的执行命令，错误的参数或不在参数范围内，没有参数等情况）有相应的提示信息。

```
>>> op
invalid command
>>> open
open <file path>
>>> open test1.wa
invalid file extension
```

错误命令，缺参数，错误参数的对应提示

```
>>> open test1.wav
file name: test1.wav
sample rate: 22050
volume: 74
```

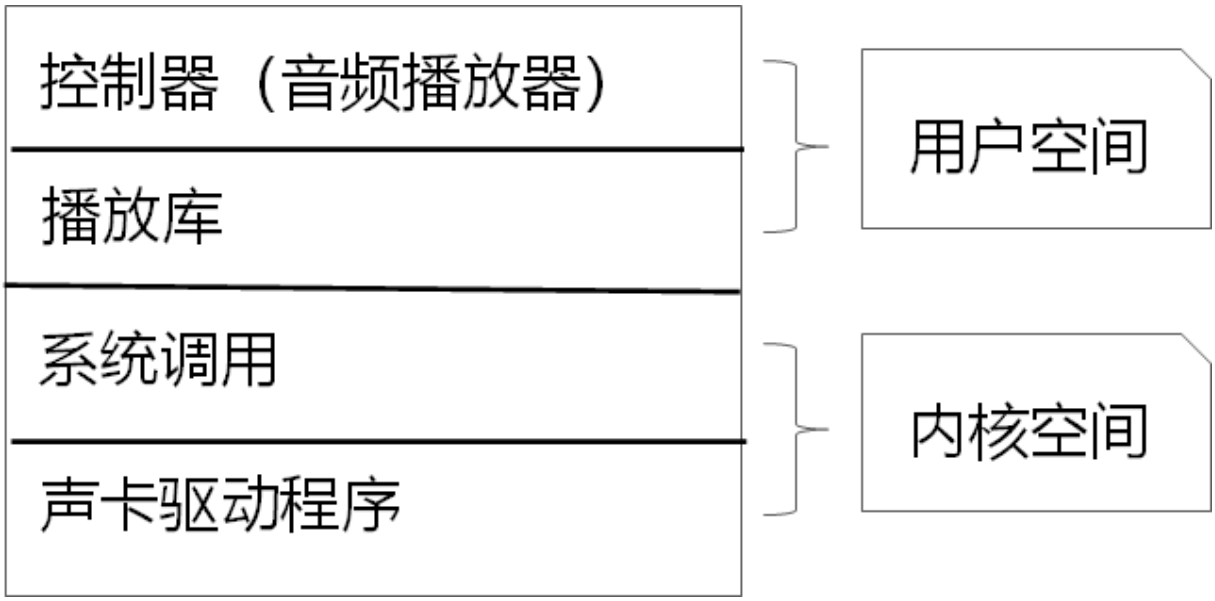
正确打开文件的提示

```
>>> speed 3
no file open
>>> open test2.wav
file name: test2.wav
sample rate: 44100
volume: 74
>>> speed 3
too high speed value for this audio
>>> speed 1.1
current speed: 1.1
```

设定范围内变速播放的提示和过大速度的错误提示

四、技术框架

项目的主要代码的结构如下：



控制器的实现在 aplay.c 文件中，提供用户界面，起到接收用户指令、控制相关操作的功能。

控制器调用播放库实现相关功能，播放库的接口独立于操作系统，故控制器的实现是独立于操作系统的。

播放库则再分为「核心逻辑」（aplaycore.h & aplaycore.c）与「解码库」(wav.h & wav.c)两部分。

核心逻辑部分，实现了音频的打开关闭、播放暂停、音量调节、变速等音频相关的常用功能。解码库则提供对特定音频格式的解析功能，核心逻辑中会检查音频格式从而从相应解码库中调用功能解析音频。

播放库的部分功能需要控制声卡，相关功能由系统调用提供。这些系统调用的接口独立于声卡类型，因此播放库的实现是独立于声卡类型的。

系统调用的实现则依赖于声卡驱动程序，声卡驱动程序是直接与声卡交互的部分。

这样，每个层次的功能都非常清晰，真正做到了高内聚低耦合，不仅易懂，而且易于添加功能。

五、实现思路

修改 xv6 最大文件大小限制

文件读写

xv6 无法直接在 ubuntu 上读文件，要把文件读入 fs.img（文件系统映像）。

修改限制的三种方式

WAV 文件大小计算公式是采样频率(kHz) x 采样位数 x 声道数 x 时间(秒) / 8 = 文件大小(kb)，如果采用如下的参数： 采样率：8kHz 采样位数：16 声道数： 2 ， 那么： 一分钟 WAV 文件的大小 = 8 x 16 x 2 x 60 / 8 = 1920KB，可近似成 2M 计算。

而目前，xv6 文件被限制在 140 个扇区，即 71,680 字节。这个限制来自于一个 xv6 inode 包含 12 个“直接”块号("direct" block number)和一个“单间接”块号，这个块号指的是一个最多可以容纳 128 个块号的块，总数为 12+128=140。这对于播放 WAV 文件是不够的。

xv6 文件系统由 inode 组成，每个 inode 是单个未命名的文件。整个磁盘读写的最小单元为 block (xv6 为 512 字节)。 xv6 文件系统采用位图块来管理磁盘中的块，每个块可以管理的大小为 BPB = BSIZE * 8，若该标志为 0，则块空闲，否则已经使用。其中整个磁盘分布如下图。

第 0 个 block 为启动区，第 1 个 block 为超级块 (也是根目录所在的块),接下来是连续分布的 dinode，最后是连续分布的 BPB(块位图)

假设块大小为 BSIZE, 则每个块包含的 dinode 结构体的数量为: IPB = BSIZE/sizeof(dinode)。等价于第 i 个 dinode 所在的 block 为 IBLOCK(i) = i/IPB+2

第 i 个 block 所在的位图块为： BBLOCK(b, ninodes) = b/BPB+ninodes/IPB+3

默认的 xv6 直接分配的时候，文件尺寸的大小为 NDIRECT·BSIZE，索引分配的时候，文件尺寸的大小为 NINDIRECT·BSIZE ,其中 NINDIRECT = BSIZE/sizeof(uint)

以下有 3 种方法可以更改 xv6 文件系统代码，使每个 inode 中支持“双间接"块("doubly-indirect" block)，其中包含 128 个单间接块地址，每个单间接块最多可以包含 128 个数据块地址。其结果是，一个文件将能够由最多 16523 个扇区(或大约 8.5 mb)组成:11+128+128*128=16523。

1. 最直接的是改变 block size 大小和直接分配/索引分配时的设定大小。

2. 或者改变直接分配的模式，把 inode 结构体中 uint addrs[NDIRECT+1]中所有的索引都指向一个块（即都变成 INDIRECT 模式，这时可以最多支持(512/4) (512/4) 512 = 8M。

3. 或者在一级索引节点后增加二级索引节点，改变文件大小限制所在扇区数量。修改 bmap()，使它除了直接块和单间接块之外，还实现了双间接块，使每个 inode 中支持“双间接”块，其中包含 128 个单间接块地址，每个单间接块最多可以包含 128 个数据块地址。其结果是，一个文件将能够由最多 16523 个扇区(或大约 8.5 mb)组成:11+128+128*128=16523。

WAV 解析

wav 文件以 RIFF 为基础，该格式以 Header、FormatChunk、DataChunk 三部分组成。

Header 首先四个字符是大端序 ChunkID，它总是 RIFF，指明其格式；之后是四字节小端序 ChunkSize，表示文件的总字节数 - 8，这个 magic number 是 ChunkID 与 ChunkSize 合占八字节，即 ChunkSize 表示其之后的所有字节的大小。之后是四字节大端序 Format，对于 wav 总是 WAVE。

FormatChunk 首先是四字节大端序 Subchunk1ID，其值总是 fmt，表示 FormatChunk 从此开始。在此后是四字节小端序 Subchunk1Size，表示 FormatChunk 总字节数 - 8，这个 8 的含义与 2 中相同。之后是二字节小端序 AudioFormat，对于 wav 总是 1。NumChannels 二字节小端序，表示总声道个数。SampleRate 四字节小端序，表示每个通道上每秒包含多少帧。ByteRate 四字节小端序，大小等于 SampleRate * BlockAlign，表示每秒含多少字节。BlockAlign 二字节小端序，等于 NumChannels * BitsPerSample / 8，表示每帧的多通道总字节数。BitsPerSample 二字节小端序，表示每帧包含多少比特。

DataChunk 首先是四字节大端序 Subchunk2ID，其值总是 data，表示 DataChunk 从此开始。在此后是四字节小端序 Subchunk2Size，表示 data 段的总字节数，也就是 DataChunk 的大小 - 8。之后是小端序 data，大小为 Subchunk2Size，表示音频波形的帧数据，各声道按帧交叉排列。

声卡初始化

准备工作

1. 补写`printf`函数，使其支持对于 8、16、32、64 位整数的十六进制格式化输出。
2. 为在指定寄存器位置读写 8、16、32、64 位整数提供接口。
3. 修改`kernel/plic.c`，设置所需中断（中断号 33）的优先级，并且通过 MIT 网卡 Lab 中的魔法语句确保所需中断能发生。
4. 参考 riscv 文档，对所需读写的内核空间地址在页表进行映射。

初始化流程

1. 遍历 PCI 配置空间，通过 vendorID 和 deviceID 定位 AC97-ICH0 声卡。
2. 设置声卡对应 PCI 配置空间中的控制寄存器，写入 Native Audio Mixer 和 Native Audio Bus Mastering 对应的地址。
3. 设置 AC_RESET#，使声卡进行重置，并等待重置完成。
4. 在声卡上探测是否有对应的音频译码器。
5. 在 PCI 配置空间内填入子系统 ID。
6. 初始化缓冲描述符表的地址、采样率与音量。

声卡交互接口

接口分为两层，底层通过读写寄存器直接与声卡交互，上层通过调用底层接口完成具体的系统调用。

设置采样率/播放/暂停：修改对应寄存器即可。

读写音量：将音量转换为 0-100 之间的整数，注意寄存器内音量按双通道表示，且数字越大音量越小。

清除缓存：触发控制寄存器的清零，并且将读缓存置空。

数据传输：

1. 在外层接口，将每次接收到的数据填入缓存页。当前缓存页满或数据写入结束时，将该页发送至底层接口。

2. 在底层接口，将每个接收到的缓存页连接至待传送数据链表的尾部。
3. 每当一个缓存页被处理完成，声卡触发中断；中断处理函数会从链表头读取新的缓存页，将其划分为 32 个数据块，并将对应信息填入 Buffer Descriptor List 中，最后设置寄存器使其继续播放。

控制器

设置 1 个缓冲区，循环接受用户输入的指令。收到一条指令后，进行解析，再按照解析结果执行相应代码。对各种错误输入提供恰当提示。

播放库核心逻辑

模块	实现思路
数据结构	使用 `enum` 记录音频文件的格式，如 `wav`。 使用 `1` 个 `struct` 存储播放信息，其中“打开状态(hasOpened)”与“音量”是始终可用的，其他 “音频文件信息”、“播放状态”、“后台信息”则只在已打开音频时可用。 音频文件信息中将“各音频格式的通用信息”与“各音频格式独有的信息”分离，后者包含在 `1` 个匿名 `union` 中。 初始化“打开状态”为 `0`，最大音量 `100`，速度倍率为 `1.0`。
打开音频	自行实现了 `strrchr` 函数，利用其提取音频文件扩展名，获取格式类型。 打开音频后，根据格式调用相应解析文件头的函数，获取音频信息。 清理声卡缓冲区，设置“打开状态”为 `1` 并确保各项信息正确，最后更新声卡的采样率。
播放音频	`fork` 出子进程，保存其 `pid`。 子进程中，设置缓冲区，循环读入音频、解码、向声卡写入解码后数据，直到无可读数据后退出。
关闭音频	根据保存的 `pid` 杀死读译进程，清空声卡缓冲区，关闭文件，设置“打开状态”为 `0`。
设置播放状态	若目前状态与当前状态相同，直接返回以减少开销。否则调用相应系统调用。
设置播放音量	检验目标音量合法性后，调用相关系统调用。 在播放信息中存有“最大音量”，默认为 `100`，播放器可根据需要将其设为其他值。本项目使用 `100`。音量调节范围为“`0`~最大

	音量”的整数。
设置播放速度	检验目标速度合法性后，通过修改声卡采样率改变播放速度。 由于修改声卡采样率要在暂停声卡后生效，故正在播放时要执行一对暂停/继续操作。

六、挑战与收获

杨碧茹

我最开始阅读声卡文档，读了二十几页后发现很难从中理解究竟声卡初始化、声卡交互的流程是什么。这一方面是因为没有相关的开发经验，对其中的许多名词并不熟悉；另一方面是因为声卡文档并非编程指南，更类似于工具书，给出的描述并不直观。但网络上与 xv6-riscv、AC97 相关的资料都不多，要获得直接的知识非常困难。我查阅了大量（不相关的）资料、问了同学、走了很多不必要的弯路，才对声卡的结构有了了解。

参考资料：

osdev 网站上对于常见声卡都有编程指南，如 https://wiki.osdev.org/AC97，值得参考。

MIT 操作系统课程网站，其上有 xv6-riscv 相关的资料，Lab 代码也很有帮助（Lab 解答可在 Github 上搜到）。

对于声卡，要结合阅读 Datasheet 和 Programmers' Reference Manual。

qemu 源代码，了解内核空间的内存结构。

在实现中途遇到了很多坑，例如：不知道究竟要完成 BIOS 初始化的哪些部分，没有映射页表，不知道应该把声卡控制器的虚拟内存映射到哪里，没有设置中断优先级，需要使用 MIT-Lab 中的魔法语句才能获得中断，因为音量设置不正确一直听不到声音（并且以为是没能正确传输数据），试图在 WSL 上配置 pulseaudio 转发声音但失败（最终换成了虚拟机）。由于 xv6-riscv 的报错非常模糊，出错时很难定位，即便定位了也很难了解错误的原因和解决方法；因此，解决这些问题的过程并不容易，在这里也要感谢别组同学提供的资料与帮助。

这次大作业后，最大的感触就是读文档一定要注重顺序，首先读概述，其次读与实现紧密相关的部分，再递归地了解其相关概念。

董文冲

课程文档没有给出足够的指引，无论在 macOS 还是在 Linux 下给出的说明都不足以正确、合理地运行 qemu。特别是在 macOS 上，需要手动编译整条 risc-v 工具链。希望将来可以把编译结果发布给同学们，跳过这个过程。

课程没有对声卡设备相关做出指引，在声卡文档方面也必须依靠自己寻找。

在 macOS 上无法正常地初始化 qemu 中 AC97 声卡设备，疑似为系统原生不支持，没有找到合适的解决方法。

wav 头可能含拓展块，在最开始查阅的资料中没有涉及，一直出现读不出 data 块尺寸的情况。后来发现后手动剔除非标准块才解决。

闵安娜

一开始没有发现 xv6 对静态数组大小的限制，后来排错，了解了 usertrap 的机制和典型情况，发现该限制，由此对 xv6 系统了解更清晰。

关于课程中学到的文件系统我有了更深的理解，例如页面设置等对后续应用程序读取处理文件速度的影响。

一开始补充读 wav 的函数，没有考虑到后来单声道多声道进声卡默认方式播放不同的情况，导致代码没有复用性。

林智鑫

我在 WSL 上编译运行项目，但后来声卡功能实现后就跑不通了，到处查资料也未能解决，最终配置了虚拟机，通过 Ubuntu 系统与 Windows 系统共享文件夹，才顺利解决了代码运行问题。

代码框架的初步设想在第一次会议上就已经成型，但具体落地又遇到了不少困难。如何设计代码框架，使其符合逻辑，又易于添加新功能；如何将接口设计得在功能足够的前提下，尽量少暴露内部实现。最终的设计是经过几次调整得到的。

细节逻辑。如切换音频时应清理声卡缓冲区，防止声卡继续播放上一音频的残余部分，一开始我未能注意到这一点。

通过本次大作业，我学习并充分练习了 Git 与 GitHub 的使用，学习了使用虚拟机，在设计技术框架与实现相关代码时实践了操作系统书中所说的“I/O 软件的层次结构”，并且此项目代码量较大、业务逻辑较复杂，对整体编程是不小的锻炼。

七、可能的拓展

格式

支持更多音频格式。

实现思路：

1. 在 enum ApFileType 与 struct ApAudioPlayInfo 的 union 中添加该格式

2. 在 apOpenAudio 函数中支持该格式。即实现读取文件头的函数。

3. 实现解码函数，形如

```
// 从 fileData 中读取数据，解码后写入 decodedData

void decodeMp3(const char *fileData, char *decodedData);
```

4. 在 apReadDecode 函数中应用对该格式音频的解码。即在 switch 中添加 1 个选项，以将解码函数赋给函数指针 decode。

播放进度控制

实现播放进度调节、单曲循环、a-b 循环。

实现思路：

播放进度调节：

1. 实现 feek 函数。
2. 在 struct AudioInfo(音频文件通用信息)中添加 1 项保存进度。
3. 在播放库中添加 1 个功能，跳转到指定进度。

单曲循环：在播放自然结束后，跳转进度到文件开头。

a-b 循环：保存下输入的 a、b 位置，在进度越过 b 位置后，跳转进度到 a 位置。

批量播放

实现播放列表、循环播放、随机播放。

实现思路：

1. 在 struct ApAudioPlayInfo(播放信息)中添加 1 项播放列表，播放列表可用链表实现。播完 1 个音频后，链表头指针指向下一个音频。
2. 使用循环链表，播放完末尾文件继续播放开头文件，则实现循环播放。
3. 实现 1 个随机函数，从而从链表中随机选取 1 个音频。