Unix V6++文件系统与配套工具

技术说明

2051565 龚天遥

在邓蓉老师的指导下完成

2022 年 8 月 7 日

目录

前言

Unix V6++操作系统是由同济大学计算机系的多位学长和老师们共同完成并持续维护的一套教学操作系统。该系统借鉴经典操作系统 Unix V6，并用 C++重新实现，借助面向对象的设计，令其更适合于课堂教学。该系统是同济大学计算机系课程《操作系统》的重要组成部分。

作为一款操作系统的重要组成部分，Unix V6++的文件系统在整个项目中发挥不可替代的作用。该文件系统具有一套精良的设计思路，并在操作系统内部对该套思路完成实现。为给尚未启动过的操作系统提供一个可以正常加载的，即按照 Unix V6++的文件系统思路格式化过的磁盘映像文件，往届研发师生们配套制作了磁盘格式化工具。

从系统 makefile 的内容推知，最初的格式化工作由 partcopy 程序和 MakeImage 程序完成。前者负责将内核和启动引导程序拷贝到磁盘映像文件内，后者负责处理 SuperBlock 及后续内容。partcopy 使用 C 语言编写，MakeImage 使用 C#语言编写。

一段时间后，一款更新的工具诞生，其被命名为 build。该程序同时可以完成启动引导文件的拷贝、内核文件的拷贝以及 SuperBlock 及后续内容的构建。该程序具有更好的可读性，且很易用。该程序使用 C#语言编写。

然而，前述三个程序的注释皆较为混乱，且由于年代久远，难以继续维护。同时，C#语言的选择令其几乎完全无法跨平台运行，对未来可能发生的环境迁移带来很大的不便。

为此，在邓蓉老师的指导下，我参考 Unix V6++文件系统的设计思路，借鉴 build 程序的源码，使用 C++语言重新编写一套文件系统构建工具。该工具只使用标准库（但需要启用 C++17 语言支持），使用 makefile 完成项目管理（而不是 Visual Studio Solution），这一切令其具有更好的可移植性。同时，该工具非直接针对格式化目标制作，而是作为一个文件系统读写器，可使用其内置交互式命令行完成文件读写等操作。源码层面，该程序思路更为清晰，注释更为完备，可通过阅读该程序深入理解 Unix V6++文件系统的结构。我将其命名为 fsedit。

由于 fsedit 只是一个交互式处理工具，仅用其无法完成文件系统的自动构建，且手动构建过程较为繁琐，特额外制作一个文件扫描器，负责扫描指定位置的文件，并生成可以控制 fsedit 内置命令行的指令。我将其命名为 filescanner。

使用管道将 filescanner 和 fsedit 连接，即可优雅地完成文件系统的自动构建。

该套工具已在 2022 年 8 月 7 日完成测试，由其构建的系统盘成功驱动 Unix V6++系统。该过程十分坎坷，结果令人满意。

Unix V6++文件系统设计解析

磁盘、扇区和盘块

众所周知，机械硬盘内有磁道、扇区等繁琐的概念。通过讨论这些值来研究磁盘，实在是太刺激了。我们不妨假装磁盘是一个个扇区线性排开的，不考虑其重叠和循环等物理结构。每个扇区大小固定是 512 字节。当我们要讨论磁盘的某个扇区时，只需要说其线性扇区号即可，而不是同时描述磁头、柱面和物理扇区三个复杂的值。

事实上，我们真的可以通过这种方式直接描述磁盘。该方法被称为 LBA 方式。如果希望通过那三个值描述一个物理扇区，则被称为 CHS 模式[4]。为方便描述，后续我们只讨论 LBA 方式的磁盘读写，即只描述线性扇区的值来获取物理扇区位置。

我们可以使用磁盘映像文件，即 img 文件，来模拟一个物理磁盘。该磁盘文件内部只是简单地将一个个扇区按照线性编号依次排列。我们在该文件内构建文件系统，虚拟机将该文件当作机械硬盘，供操作系统使用。

我们为操作系统准备一块大小为 10800KB 的 img 文件。显然，该文件模拟的硬盘具有 21600 个扇区。我们可以描述的扇区线性地址范围是[0, 21599]。

扇区大小是 512 字节，这个是不能改的。同时，这也是磁盘读写的最小单位，即一次读取操作可以一次性读取 512 个字节，一次写入操作一定会写满 512 个字节。对于一些系统，512 字节无法满足它们的需求。因此，操作系统将多个扇区视作一个整体，称为盘块。如，Linux 的盘块大小是 4KB，即 8 个扇区。

Unix V6++的盘块大小是 512 字节，与扇区大小一致。

Unix V6++文件系统的扇区排布

Unix V6++将磁盘分为六大部分。图中依次标出每部分的名称及所在盘块区间。盘块从 0 开始编号。考虑到 Unix V6++文件系统的盘块大小和扇区大小没有区别，后续可能会混合使用两个名词。当我们混用这两个名词时，认为它们的含义是一样的。

bootloader

该部分程序存储在硬盘的第一个扇区（即 0 号盘块）。这个位置和大小是 BIOS 规定的。

该部分为操作系统的启动引导程序。CPU 加电后，首先进入 BIOS 程序。BIOS 程序完成准备工作后，将磁盘中的第一个扇区加载到内存 0x7c00 处，之后将 eip 指向该地址，执行其中代码。

第一个盘块内存储的程序被称为 boot，其大小不能超过 512 字节。同时，这 512 字节的第 511 个字节和第 512 个字节依次必须是 0x55 和 0xaa，否则会被 BIOS 认为该盘块有问题。boot 的任务是加载 loader 程序，后者完成更多准备工作，加载内核，然后跳转进入内核代码。两个程序合在一起，被称为 bootloader。

Unix V6++内没有设计 loader 程序，仅通过一个 boot 完成基本设置，并迅速跳转到由 C++编写的程序内继续执行。

kernel

该部分存储在硬盘文件的第 1 号盘块到第 199 号盘块。

内核二进制文件。该文件需要用特殊方式编译，并通过对链接命令或链接脚本的设置，将进入点放置在指定偏移位置。该位置由操作系统开发者自行规定，以便从 bootloader 程序跳转进入。

superblock

该部分从第 200 号盘块开始，连续占用 2 个盘块的空间，即 1024 字节。

它是描述文件系统结构的最重要结构之一，记录有硬盘大小、交换区位置等信息，完成空盘块管理、空 inode 管理等任务。

inode

inode 区域从 202 号盘块开始，直到第 1023 号盘块（含）。每个 inode 占用 64 字节，因此每个盘块可以存储 8 个 inode 结构。

inode 是用来表示一个文件的重要结构。所谓文件，可以指普通文件，也可以是文件夹、块设备和字符设备。inode 内记录文件大小、文件类型等基本信息。更重要的，其内部的二级索引机制可以链接高达 6+2×128+2×128×128=16518 个盘块，令操作系统可以存储最大约 16MB 的文件。

data

紧跟在 inode 区域后，截止到盘交换区之前。

存放文件数据的盘块。也可能存储的是二级索引结构中的索引表。

swap

从第 18000 号盘块开始，用尽盘内剩余的所有空间。

该部分为 Unix V6++操作系统的盘交换区。

SuperBlock 结构

硬盘上，前 200 个盘块和最后交换区的盘块都不由文件系统直接管理。文件系统直接管理的是 superblock、inode 区和 data 区。

SuperBlock 结构占用 2 个盘块，记录文件系统的一些基本信息，并完成空闲盘块和空闲 inode 的管理。其结构如左图所示。

s_isize 登记 inode 区占用的空间，单位是盘块。该值应该是 822。

s_fsize 记录整块硬盘的盘块总数。该值应该是 21600。

s_nfree 和 s_free 数组完成对空闲数据盘块的索引管理。管理模式在后文中继续探讨。

s_ninode 和 s_inode 完成对空闲 inode 的管理。其中，s_inode 数组直接登记空闲 inode 的编号，s_ninode 表示 s_inode 数组内剩余多少个空闲 inode 可用。它们形成的栈结构总共可以直接管理 100 个空闲 inode。当我们申请一个新的 inode 时，若 s_ninode 大于 0，则直接从 s_inode 数组内选取第 s_ninode-1 号元素，并令 s_ninode 自减小。当 superblock 直接管理的空闲 inode 全部分配完毕，将前往 inode 区域重新寻找空闲 inode 以填充这个栈，直到对 inode 区搜索完毕，或直接管理的空闲 inode 重新达到 100 个。

s_flock 和 s_ilock 是两个锁，分别针对盘块操作和 inode 操作，防止在并发情况下出现问题。

由于 inode 结构同时存在于硬盘内和操作系统的内存里，操作系统需要知道是否应该将内存里的 superblock 填写回硬盘。s_fmod 标记做的就是这件事。

s_ronly 用来标记此文件系统是否是只读的。

s_time 登记最后一次更新时间。记录 unix 时间戳，以秒为单位。

在 s_time 之后还有一片区域，被称作填充区。

填充区的第一个元素是 padding_changed，用来标记填充区数据是否被更改过。

之后，填充区有 7 个 4 字节元素，依次登记磁盘扇区总数，inode 区起始位置（单位是盘块），inode 区域大小（单位是盘块），data 区起始位置（单位是盘块），data 区大小（单位是盘块），交换区起始位置（单位是盘块），交换区大小（单位是盘块）。

然而，上述所有内容加起来并未达到 1024 字节。因此，superblock 结尾还有 156 个纯 0 字节，作为空白填充。

Inode 结构

inode 是表征文件的重要结构。结构如左图所示。

其中，permission 表示对该文件的操作权限。三个 permission 依次针对系统内所有用户、文件主同组用户及文件主自己。权限的三个字节依次是读权限、写权限、执行权限。

isvtx 登记该文件是否是可执行程序。

当 isgid 为 1 时，d_gid 记录该文件所属的用户组号。当 is_uid 为 1 时，d_uid 记录该文件所属的用户 id。

ilarg 用于识别该文件是否为巨型文件。即，大于 3KB 的文件。

file_type 用于标记文件类型。从 0 到 3 依次为普通文件、字符设备（如文本模式的 CRT 显示器）、文件夹、块设备（如硬盘）。

ialloc 表示此 inode 是否已被使用。为 0 时，此 inode 可被申请使用。

d_nlink 登记该 inode 的链接数。每当用户或程序创建指向此 inode 的软连接，该值自增。其默认值为 1。当我们删除一个软连接时，该值自减。只有当该值减为 0 时，一个文件才彻底被删除。

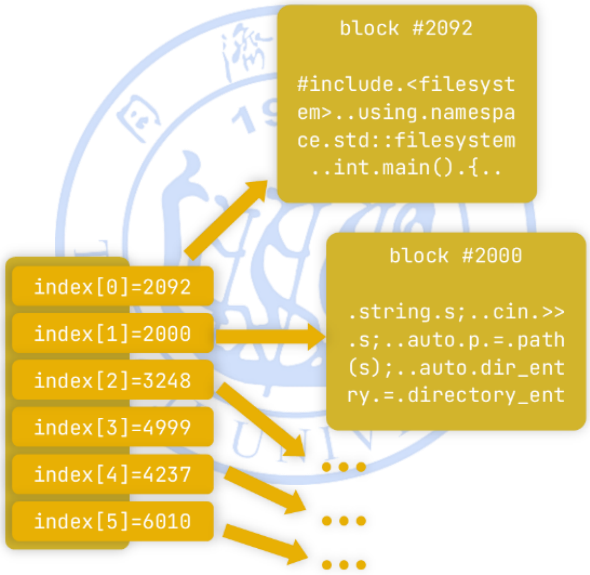
d_size 登记该文件的大小。单位是字节。

d_atime 和 d_mtime 都是以秒为单位的 unix 时间戳，依次登记文件最近访问时间和文件最近修改时间。

index、fir index 和 sec index 共同构造成二级索引结构，实现存储最大 8MB 的文件。

Inode 内的二级索引结构

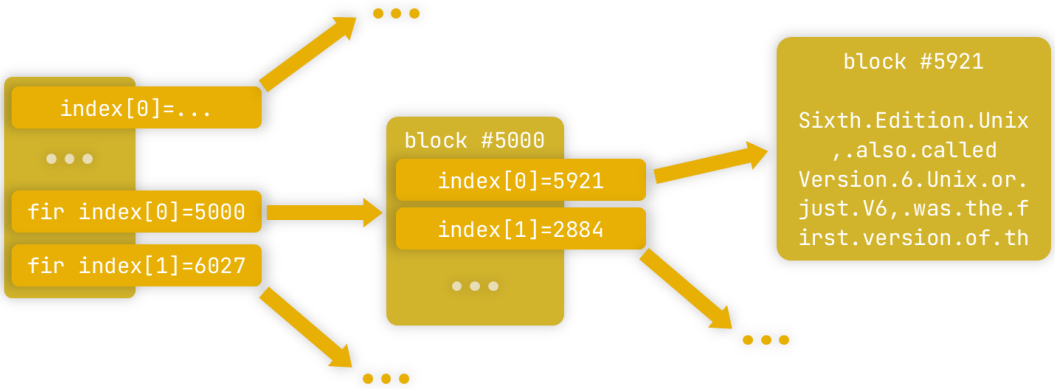
3KB 及更小的文件



当文件大小不超过 3KB，即 6 个盘块时，inode 只需要使用 index 数组即可实现对数据盘块的管理。该数组内每个元素的值都是一个盘块号。例如，当 index[1]=2000 时，表示该文件的第 513 个字节至第 1024 个字节存储在第 2000 号盘块内。显然，index 数组元素的值不能小于 1024，不能大于 17999。

这种索引的方式很直接，每个值直接对应数据盘块。缺点是这样做难以存储大型文件。例如，对于一个 50KB 的文件，采用纯直接索引的方式，需要在 inode 内放置 100 个 4 字节的盘块号存储变量，令 inode 变得很沉重。因此，当文件大小超过 3KB 时，前 3KB 依旧采用直接索引的方式存储，后续部分采用间接索引方式存储。

超过 3KB，但不超过 131KB 的文件

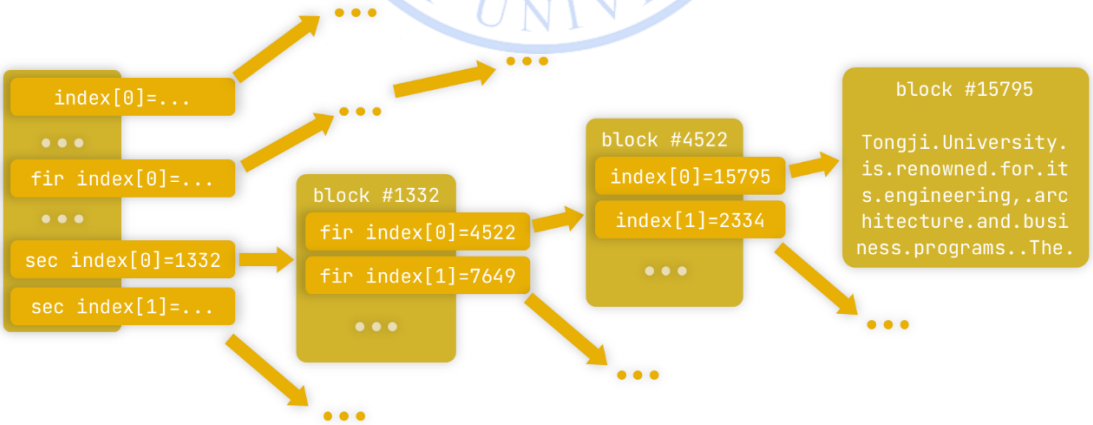


仅使用直接索引难以管理大文件。因此，inode 节点内提供 2 个一级间接索引变量。它们存储的依旧是盘块号，只不过该盘块内存储的不是文件数据，而是更多的直接索引编号。我们称之为索引块。

由于每个盘块编号只需要占用 4 字节空间，一个索引块内可以存 512÷4=128 个盘块号。它们当中的每个盘块号直接指向一个数据盘块。因此，一个索引块可以帮助管理高达 128×512 字节=64KB 的文件数据。两个一级索引块一共完成 128KB 的数据管理。配合前面的 6 个直接索引，可存储文件大小提升到 131KB。

然而，当文件更大，这种间接索引模式仍会吃不消。为此，inode 内再提供两个二级间接索引变量。

超过 131KB 的超大文件



为满足对更大文件的管理，在一级索引之后，inode 额外提供 2 个二级索引变量。它们存储的也是盘块号，对应的盘块内存储的同样是 128 条索引信息，该盘块被称为二级索引块。二级索引块内的索引信息指向的不是数据块，而是一级索引块。一级索引块内存储的盘块号指向的才是真正的数据块。

每个一级索引块可管理 64KB 空间，每个二级索引块可以管理 128 个一级索引块。因此，每个二级索引块可以管理 64KB×128=8MB 的文件数据，两个二级索引块一共可管理高达 16MB 数据。配合前面的直接索引和间接索引，一个 inode 可以存储高达 16.1MB 的文件。不过要注意的是，二级索引块也是要占用空间的。因此，一个文件在硬盘上占用的空间可能会大于其实际内容大小。

Unix V6 操作系统发布于 1975 年 5 月，发布于几年后的游戏《Super Mario Bros》大小仅为 32KB。Unix V6++文件系统提供的最大 16MB 存储能力足以满足大多数需求。

目录文件结构

树形文件结构的实现，文件夹功不可没。事实上，文件夹自己也是一个文件。文件内每 32 个字节表示文件夹里的一个文件。其中，前 4 字节是 inode 编号，指向该文件的 inode。后 28 字节是文件名。文件名结束后，未使用的部分全部用字节 0 填充。

如此，每个文件夹可以登记高达 16MB÷32B=50 万个文件。

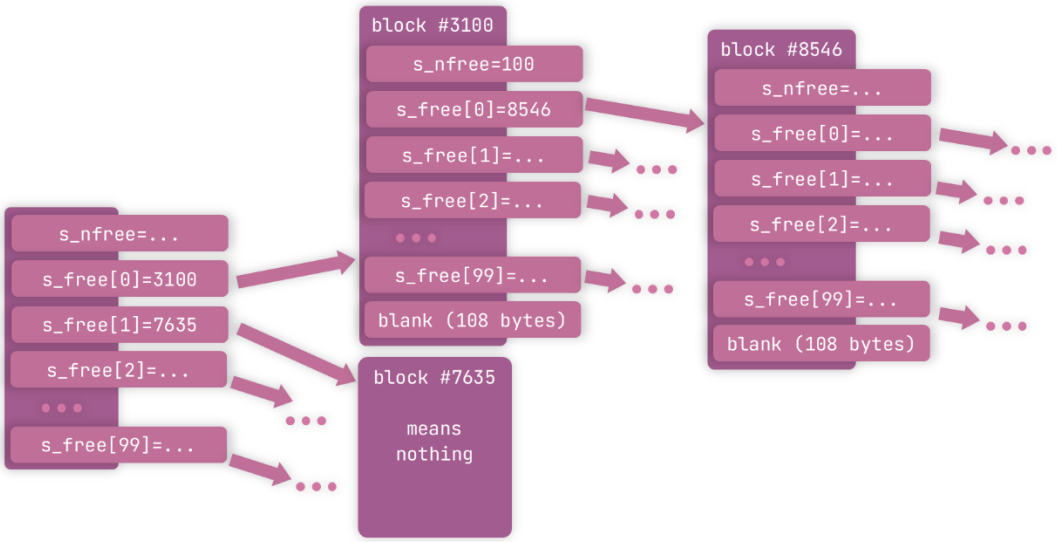
SuperBlock 内的空闲盘块管理模式

磁盘上，从第 1024 号盘块直到第 17999 号盘块都作为数据区使用。如果选择类似管理空闲 inode 的方式管理空闲盘块，搜索速度将变得很慢。更重要的，盘块上没有该盘块是否被使用的标记，使得类似管理 inode 的方式根本无法使用。为此，文件系统内采用一种“栈的栈”结构管理空闲盘块。

观察 superblock 内的 s_nfree 值和 s_free 数组。类似对 inode 的管理，s_nfree 表示 superblock 内直接管理多少个空闲盘块。s_free 数组则记录空余盘块号。

假设 s_free=99，即 superblock 直接管理 99 个空闲盘块，这些盘块的编号依次存储在 s_free[0]至 s_free[98]内。此时，如果我们释放一个盘块，不妨设释放的是第 3000 号盘块，则我们将 3000 登记到 s_free[99]，并令 s_nfree=100。

之后，如果我们继续释放盘块，如第 3100 号盘块，superblock 内已经没有登记它的地方了。于是，我们将 s_nfree 和整个 s_free 数组的值拷贝到第 3100 号盘块内，共 404 个字节。然后，我们令 superblock 内的 s_nfree=1，令 s_free[0]=3100。



当我们希望获取一个空盘块，但发现 superblock 内直接管理的盘块只剩 1 个（即 s_nfree=1）时，我们先记录将要分配的盘块号（即 s_free[0]），然后从该盘块内读取 404 字节信息，拷贝到 superblock 内，从 s_nfree 开始连续更新 404 字节的内容。

如此，构建形成“栈的栈”结构。每当即将分发手头仅剩的最后一个盘块时，从中找到下一组空闲盘块的信息，以此为后续数据继续分配盘块。每当手头空闲盘块过多，又有新盘块要释放时，就将当前的盘块信息都写入新盘块，然后重新开始登记空盘块信息。借助此结构，文件系统可以高效优雅地完成空闲盘块管理。缺陷在于没有均衡各个盘块的磨损。

为标记结尾，最后一组盘块的 s_free[0]登记为 0，该组仅提供 99 个真实可用的数据盘块。

系统盘读写工具 FsEdit 与 FileScanner 使用说明

为方便在系统外对文件系统的编辑，及取代较有年代的 build 工具，我在邓蓉老师的指导下编写一个 fsedit 工具。该工具通过友好的命令行界面，为用户提供删改及格式化 Unix V6++文件系统的能力。此外，我设计 filescanner 工具，自动扫描并生成控制 fsedit 工具工作的指令。单独使用 fsedit 可以畅游 Unix V6++的文件系统，两个工具通过管道连接可以自动完成系统盘的格式化与基本构建。

FsEdit

该工具需要在命令行环境内使用。

直接启动程序，可以看到程序使用说明。全文如下：

该提示信息可以在内置交互式界面中通过命令“h”打印。

该程序启动时，希望知道 2 个重要信息：磁盘文件是谁、要做什么准备。关于第一个信息，只需要指定文件路径即可。该路径可以是空的，这种情况下程序会新建磁盘文件。第二个参数可以是 c、m 或 e，分别表示创建文件、格式化现有文件和仅仅打开文件。注意，c 模式在创建文件后，会立即执行一次格式化操作。

例如，我们希望新建一个 Unix V6++文件系统文件，保存为 c.img，则可使用如下命令：

```
./fsedit c.img c
```

若我们希望看一下原有的文件系统内有哪些东西，则可通过以下命令打开该文件（假如文件名是 c.img）：

```
./fsedit c.img e
```

一般情况下，不使用 m 操作。

打开磁盘后，会进入一个交互式命令行界面。通过简单指令即可完成对文件系统的操作。注意，这些操作中包含写入内核和启动引导文件，它们并不属于文件系统，但本工具提供此能力。

本工具现提供这些能力：格式化、写入内核文件、写入启动引导文件、列出当前路径下的内容、切换工作目录、上传本地文件到磁盘文件、从磁盘文件下载文件到本地、新建文件夹、删除盘内内容。

特别注意，所有输入的路径需要以竖线（“|”）包裹。这是为了支持带空格的路径，同时降低程序内部路径解析的难度。

操作完毕，通过 x 命令即可存盘退出。如果对文件系统做过更改，一定要使用 x 命令退出。强制关闭可能会导致文件系统结构异常。

例，使用 e 模式打开文件系统后，通过 l 命令列出当前路径下的文件：

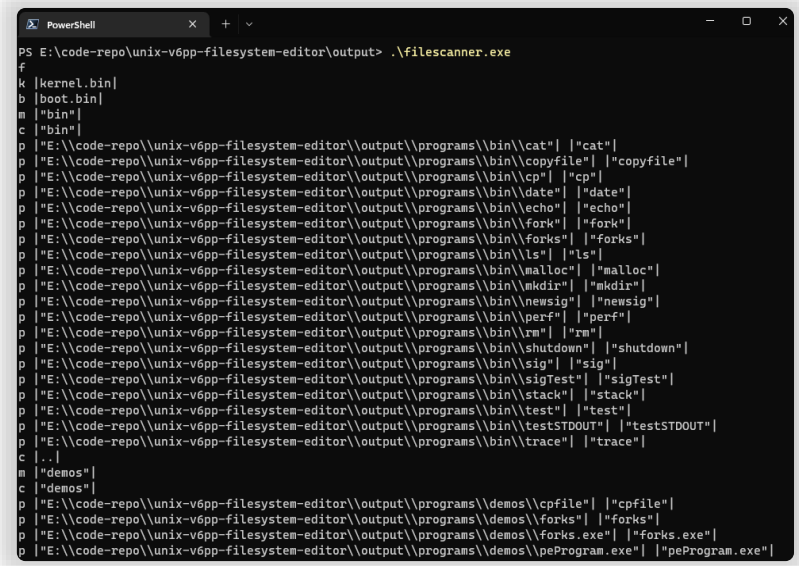
```
PS E:\code-repo\unix-v6pp-filesystem-editor\output> .\fsedit.exe .\c.img e
[] > l
drwxrwxrwx   1  0 :  0      32  1659883457.m  1659883457.a dev
-rwxrwxrwx   1  0 :  0       0  1659883457.m  1659883457.a tongji-yyds
drwxrwxrwx   1  0 :  0     640  1659883457.m  1659883457.a bin
drwxrwxrwx   1  0 :  0     256  1659883457.m  1659883457.a demos
drwxrwxrwx   1  0 :  0      32  1659883457.m  1659883457.a etc
-rwxrwxrwx   1  0 :  0   59025  1659883457.m  1659883457.a Shell.exe
drwxrwxrwx   1  0 :  0       0  1659883457.m  1659883457.a usr
drwxrwxrwx   1  0 :  0       0  1659883457.m  1659883457.a var
[] >
```

FileScanner

该程序会自动扫描需要上传到文件系统内的文件，并生成可以控制 fsedit 工作的指令。

将 filescanner 程序与 boot.bin、kernel.bin 放在同一路径，并在同路径下新建 programs 文件夹，在其内部放置希望上传到文件系统内的文件（如，shell.exe）。

之后，命令行直接运行 filescanner 程序，它将自动扫描 programs 文件夹，并输出可以控制 fsedit 的一系列命令。



组合使用

filescanner 可以生成一系列用于控制 fsedit 程序工作的命令。使用管道将 filescanner 的标准输出连接到 fsedit 的标准输入，即可实现自动构建磁盘文件。

将两个程序放在同一路径，并准备好 kernel.bin、boot.bin 和 programs 文件夹，之后使用命令行运行命令：

```
./filescanner | ./fsedit c.img c
```

即可完成系统盘的构建。

系统盘读写工具技术实现细节

参考环境

编码

代码文件：UTF8

编译结果：GB18030

编译环境

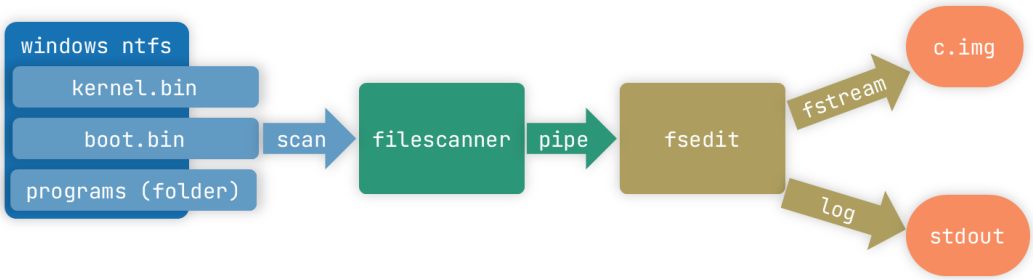
TDM-GCC 10.3 x64, C++17

测试运行

Windows 11 Pro x64, powershell 7.2.5

VM Windows 10 Enterprise LTSC 2021 x64, powershell 7.2.5

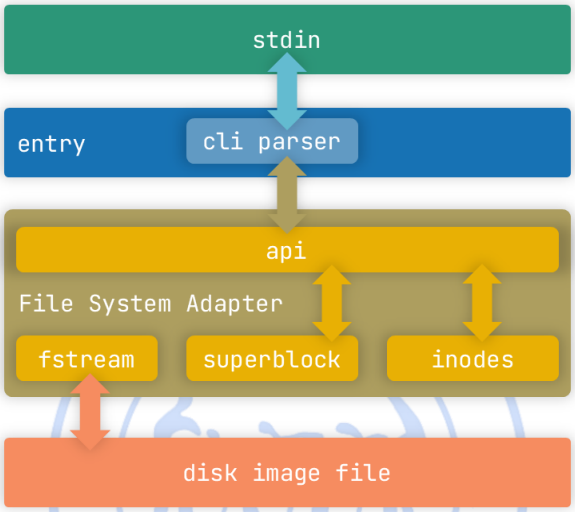
FileScanner



该程序扮演辅助角色，负责控制 fsedit 工作。

进入程序后，它会立即输出 f 命令，令 fsedit 格式化磁盘文件。之后，通过 k 命令和 b 命令，控制 fsedit 写入 boot.bin 和 kernel.bin。然后，filescanner 会在 C++ filesystem API 的帮助下，递归扫描 programs 文件夹的内容，相应生成目录构建、目录切换和文件上传指令。当一个文件夹内的文件全部上传完毕，filescanner 会发送命令 c |..|，表示切换到上级目录，继续完成该目录下其他文件的上传。直到所有文件上传完毕，filescanner 会发送 x 指令，令 fsedit 结束工作。至此，磁盘文件构建完毕。

FsEdit



该程序大体结构如上。核心部分为 File System Adapter (FDA)，其用文件流连接到磁盘映像文件，直接或延迟对后者进行读写操作。

为实现对文件系统内部结构的访问，该程序内仿照 build 程序和《操作系统原理》教材，实现 Inode 结构和 SuperBlock 结构。

同时，该程序额外实现一些辅助性结构。Block 结构是一个大小为 512 字节的数据块，没有任何特点。InodeDirectory 结构简化访问文件夹的难度。其内部包含 entries 数组和 length 记录，后者登记 entries 的长度。entries 采用动态内存管理，使用堆空间的内存，其每个元素都为 32 字节，含一个 4 字节 inode 号和 28 字节长度的路径名称。

FDA 内直接包含一个 superblock 结构和六千多个 inode 结构。磁盘内的这两部分内容直接投射到 FDA 中，所有修改皆在内存副本内完成。由于盘块过多，这部分采用按需取用的方式，直接从盘内取副本，写入时直接写入到磁盘文件内。因此，在异常退出后，会出现磁盘文件内 superblock 和 inode 区未更新，但 data 区被修改，造成一定的麻烦。

FDA 内部实现读写指定盘块，获取与释放 inode 或盘块，遍历 inode 的二级索引结构等内部能力，并提供 mkdir、ls 等文件系统操作，供外部调用。

其中，二级索引遍历是一大重点。删除、上传文件、下载文件、读写等操作皆要进行此操作。显然，若在每个地方都实现一遍，会令人很疲惫。Unix V6++系统内引入逻辑块号概念，并通过 Bmap 函数将文件数据逻辑块号转换为磁盘上的物理块号。本工具采用另一种方式，通过对多种情形下的遍历方式做总结，提炼大体过程，通过钩子函数改变细节处理逻辑。

提炼后的遍历过程如下：

读取文件大小
遍历直接索引
--> 已经读完？继续/结束
--> 申请盘块
--> 盘块处理
--> 盘块后处理
--> 剩余待处理内容减去 512 字节
遍历一级索引
--> 已经读完？继续/结束
--> 申请索引块
--> 读取一级索引块
--> 遍历一级索引块
-->--> （同遍历直接索引）
--> 索引块后处理
遍历二级索引
--> 已经读完？继续/结束
--> 申请索引块
--> 读取二级索引块
--> 遍历二级索引块
-->--> （同遍历一级索引）
--> 索引块后处理

其中，蓝色部分在内部实现，不可控。紫色部分留为钩子，处理逻辑从外部传入。

例如，希望释放一个文件的所有数据块，则在索引块后处理和盘块后处理中，释放传来的盘块。对于申请盘块的操作，令其保持使用原盘块即可。

当我们希望读取一个文件时，在申请盘块部分直接返回原始盘块号，在盘块处理处拷贝盘块内容。其他处理留空即可。

如此，对二级索引的遍历仅实现一次，便可支撑多个更高级的操作进行。详见源代码 FileSystemAdapter.cpp 和 FileSystemAdapter.h。

InodeDirectory 通过传入 inode 直接构造。出现问题时，通过抛出异常告诉外部信息。FDA 内处理抛出的异常，并通过返回值和标准输出告诉外部程序和用户错误信息。除此之外，整个程序内不再使用异常机制，仅通过方法返回值判断操作执行情况。

当检测到可能破坏用户物理设备的致命错误时，程序会立即输出利于寻找问题所在的排错信息，并直接结束整个程序。

为便于用户使用，在 FDA 外层包装一个简单的命令行界面。该界面接收用户命令，将用户命令转为对 FDA API 的调用，实现对文件系统的各种操作。

参考文献

[1] 同济大学. 操作系统原理. 2019

[2] 邓蓉. 格式化一个 Unix V6 文件卷. 2021

[3] 方钰. s19：文件管理（unix 文件系统的静态结构与打开结构）. 2022

[4] OSDev. ATA read/write sectors. https://wiki.osdev.org/ATA_read/write_sectors

[5] 中国色. http://zhongguose.com

致谢

本项目的完成离不开对同济大学计算机系优秀课程《操作系统》的学习。通过对 Unix V6++和 build 工具的源代码深入阅读，理解其中各部分实现，完成对本套工具的开发与调试。

在此，感谢邓蓉老师的指导、方钰老师的课程和沈坚老师的启发，更要感谢起草 Unix V6++系统的前辈们和持续维护这套系统的老师同学们。没有你们的努力，不会有这么精彩的课程。