



南开大学
Nankai University

操作系统功能挑战赛（初赛设计报告）

Proj209: 设计一个 **Linux** 文件系统并实现文件和目录读写

开发报告

成员：谢畅 张景瑞

学校：南开大学

指导老师：宫晓利 张金

2024 年 7 月 31 日

目录

1	目标描述	1
2	比赛题目分析和相关资料调研	1
2.1	赛题分析	1
2.2	背景调研	2
2.3	文件系统知识调研	4
2.3.1	Linux 内核模块编程原理概述	5
2.3.2	fat32 文件系统	8
2.3.3	simplefs 文件系统	11
2.3.4	调研结果	12
3	系统框架设计	13
3.1	核心重要组件	13
3.2	框架特点	14
3.3	总体框架布局	14
3.4	核心原理	15
3.4.1	块组原理	15
3.4.2	哈希 B+ 树原理	16
3.4.3	扩展树原理	18
3.5	核心数据结构实现	20
3.5.1	超级块	20
3.5.2	块组描述符	23
3.5.3	inode 索引节点	24
3.5.4	目录项	26
3.5.5	哈希 B+ 树相关	26
3.5.6	扩展树相关	28
4	开发计划	30
4.1	磁盘格式化工具的编写	30
4.1.1	块组的划分	31
4.1.2	超级块信息写磁盘	32
4.1.3	块组描述符写磁盘	32
4.1.4	初始化第 0 个块组的 inode 位图	32
4.1.5	初始化第 0 个块组的 block 位图	33
4.1.6	初始化第 0 个块组的 inode_store	34
4.2	填充与 VFS 层对接的函数接口	35

4.2.1	super_operations	35
4.2.2	inode_operations	35
4.2.3	file_operations	37
4.3	实现文件系统模块	39
4.3.1	文件系统卸载	39
4.3.2	文件系统挂载	39
4.3.3	模块初始化和卸载	40
4.4	编写测试脚本进行测试	40
5	比赛过程中的重要进展	40
6	系统测试情况	40
6.1	测试概况及脚本编写	40
6.2	make test 测试	43
6.3	具体测试情况	44
7	项目测试结果功能与创新性分析	48
7.1	项目结构	48
7.2	创新性分析	50
7.3	结果与性能对比	52
8	遇到的主要问题和解决办法	53
9	分工与协作	56
10	未来计划	56
11	项目目录	57
12	比赛收获	57

1 目标描述

项目的基本目标是**设计并实现一个新的 Linux 文件系统**，新设计的文件系统名字为 **XCraft**，不仅实现了传统 Linux 文件系统的读、写、增、删、改、查等全部功能，还实现了内核日志打印到用户空间和文件系统可视化等创新性功能。

在功能上，我们设计的文件系统能够支持一般 Linux 文件系统的全部功能，包括：**文件系统的挂载和卸载功能，文件和目录的增删读写等操作，文件权限属性控制等**。

在技术细节上，我们创建了一个专门的文件系统 Linux 内核模块，用于将新设计的文件系统与现有的虚拟文件系统 VFS 框架进行对接，并实现了 VFS 框架要求的超级块、索引结点、目录等数据结构以及相适应的读写接口。

此外，我们编写了测试脚本，用于将特定的存储设备格式化为新设计的文件系统，同时将其进行挂载测试了一系列的文件系统操作。

本赛题预定的目标如下：

- 设计实现一个 Linux 内核模块，此模块完成如下功能：
 - 将新创建的文件系统的操作接口和 VFS 对接。
 - 实现新的文件系统的超级块、dentry、inode 的读写操作。
 - 实现新的文件系统的权限属性，不同的用户不同的操作属性。
 - 实现和用户态程序的对接，用户程序
- 设计实现一个用户态应用程序，可以将一个块设备（可以用文件模拟）格式化成自己设计的文件系统的格式。
- 设计一个用户态的测试用例应用程序，测试验证自己的文件系统的 open/read/write/ls/cd 等通常文件系统的访问。

对于上述目标，我们全部完成。并且我们还在其基础上完成了文件目录创建与删除、复制和剪切等功能的实现。项目链接在[此处](#)。

2 比赛题目分析和相关资料调研

2.1 赛题分析

我们小组选题为 **proj209**，即”自己设计一个 Linux 文件系统并实现文件和目录读写操作”。其题目的描述链接为[此处](#)。题目要求如我们之前目标描述所示。简而言之，即为要求设计一个基于 Linux 的文件系统，能够实现文件系统的基本功能，能够对相应的磁盘块设备进行格式化，并对其挂载后验证功能。

本项目的核心目标在于设计并实现一个全新的文件系统 **XCraft**，该文件系统应该具备基础的文件操作功能，良好的性能和可用性，以及和用户态应用程序之间的交互。于是我们根据我们文件系统的工作原理和层次结构，项目需求可以分成三个主要部分：

1. **新文件系统与 Linux 虚拟文件系统 VFS 的接口对接**：这是 XCraft 实现的核心部分，我们需要在新的文件系统中实现一套完整的 VFS 接口，以此与 Linux 内核进行无缝对接。这包括文件和目录的创建、删除、读写以及目录内容的遍历等基础操作，以及文件的复制、剪切、文件权限、硬链接和软链接等高级功能；
2. **新文件系统磁盘空间的划分与管理**：文件系统是对磁盘存储空间的管理者，新文件系统需要对磁盘的存储管理实现一套高效且可靠的方案。这涉及到对我们的文件系统核心数据结构的构建和管理，文件和目录数据的组织以及良好的数据查询策略；
3. **新文件系统与用户态程序的交互接口**：虽然文件系统主要在内核空间运行，但是其提供的服务主要是为用户态的应用程序所用。因此，新设计的文件系统必须通过系统调用提供一套完整的 API，以此满足应用程序对文件操作的各种需求。

2.2 背景调研

为了弄清楚设计的文件系统应该满足什么应用场景，以及充分说明此项目的开发缘由，我们进行背景调研。

当今，随着互联网的发展、数据密集型应用（如大数据分析、虚拟化和云计算）的普及，多媒体内容（如高清视频、音频文件）的增多，虚拟化和云计算技术的飞速进步，物联网的发展，存储需求急剧增加，要求需要大量的空间来处理和保存数据。这些变化要求文件系统能够：

- 支持更大的存储容量；
- 提供更高的性能和效率；
- 提供更好的容错性和数据完整性。

这为传统的文件系统带来了前所未有的挑战，早期的文件系统如 fat、ext2 等，在面对这些不断增长的数据存储需求时表现出来许多局限性：

- **容量限制**：fat32 单个文件最大只能为 4GB，文件系统最大只能为 2TB。这在当今已经无法满足需求；
- **性能瓶颈**：传统文件系统处理文件时效率和性能过低，当处理大文件时效率更是急剧下滑，在随机读写的情况下性能很难让人满意；

- **实现数据完整性和错误检查困难**：早期文件系统缺少错误检查机制，很难处理崩溃和数据读写错误。这在面对海量数据读写时其准确性难免得到质疑，由于恢复困难使得维护成本过高。

这期间硬件技术也产生了巨大进步：

- 硬盘和各种 SSD、flash 闪存等容量增加：存储设备的容量和速度不断提升；
- 处理器和内存性能提升：支持文件系统更复杂的管理方式和优化算法的实现。

同时各种各样的新型应用场景正在出现：

- 分布式文件系统：如 Hadoop HDFS 和 Ceph，要求文件系统能够管理大量分布在不同节点上的数据；
- 高性能计算（HPC）：需要文件系统提供高并发、高吞吐量的访问能力；
- 内容分发网络（CDN）：需要快速、高效地分发大量媒体内容。

这一切都要求产生大文件系统来支持更大的存储容量，提供更高的性能和效率，维护更好的数据完整性。可是大文件系统也会面临不一样的难处：

- 复杂的管理方式，如何实现元数据的统一管理来实现高效访问；
- 庞大的目录意味着众多的目录项，如何进行高效地查找和访问目录项；
- 如何构建大文件的数据块组织形式来在数据块的随机访问下提高效率；
- 格式化磁盘的工作量加大，难度增大；
- 海量的数据存储意味着必须要有错误检查和崩溃处理机制，不然数据完整性难以保证。

因此，结合上述背景的驱动，我们最终确定了我们文件系统实现的方向：

1. 循环块组的分配方式；
2. 目录哈希树快速查询；
3. 块组延迟初始化，减轻格式化磁盘压力；
4. 文件数据块的组织方式有扩展树进行管理；
5. 日志来处理崩溃和故障恢复。

综上，便是我们想要实现的方向，即实现一个高性能的保证数据完整性的大文件系统，并且减轻初始化磁盘的工作量，从而达到高效、精简的目的。我们在调研 ext4 的过程中有了思路。

我们结合目前已有的一些大型文件系统，经过详细调研，提出了下面的设计思路：

1. **dir hash**。目录哈希指的是将每个目录项与一个 hash 值对应，一个根 inode 下面索引目录项的结构以一个哈希 B+ 树存在，可以有效避免海量目录造成的查询速度下降的问题。
2. **Extents**。Extent 指的是一连串连续实体 block，这种方式可以增加大型文件的效率并减少分裂文件。单一 inode 中可存储 4 个 Extent，超过 4 个的 Extent 会以树的方式被索引；
3. **块组初始化延迟**。ext4 的块组采取延迟初始化的方式，按需分配。这不仅良好的组织了元数据，并且还减少了管理难度和初始化磁盘的工作量；
4. **日志校验和**。日志校验和进一步提高文件系统的可靠性和数据完整性，其确保了日志记录的完整性，使文件系统能够在崩溃或其他异常情况发生时进行更可靠的恢复。其原理简单概述便是在进行故障恢复时我们会重新计算校验和与日志中每条记录的校验和进行比较，如果一致便代表我们可以重放日志此记录操作；
5. **突破子目录数量限制**。ext4 采取 Htree 的索引功能，实现了大量目录的存储，并且突破了 ext3 的一个目录下最多只有 **32000** 个子目录的限制，达到了 **64000**。

综上，我们经过反复思考讨论，提出了本段最开头我们的文件系统实现方向。目前我们已经成功实现了前三个方向（循环块组、目录哈希树、延迟初始化），其中第四个方向我们现在是以直接索引块、一级间接索引块和二级间接索引块的管理方式代替，第五个方向还并没有实现。

2.3 文件系统知识调研

针对赛题，我们进行了深入的资料调研，主要集中在如下方面：

1. 研究 Linux 内核文件系统框架，看懂 VFS 的工作原理以及如何将文件系统与 VFS 进行对接；
2. 看懂 Linux 内核模块编程原理，可以动态地加载和卸载模块以此来注册和卸载我们的文件系统；
3. 参考经典开源的 Linux 文件系统项目，例如 simplefs，fat32 等，这些源代码为我们提供了宝贵的参考资料，帮助我们理解如何设计和实现一个功能完备的文件系统。

2.3.1 Linux 内核模块编程原理概述

内核模块总述

Linux 内核模块式可以根据需要加载和卸载到内核中的代码片段。它们扩展了内核的功能，而且无需重启系统。例如，一种类型的模块是设备驱动程序，它允许内核访问连接到系统的硬件。没有模块，我们必须重构内核代码并将新功能添加到内核映像中，这样十分麻烦，并且内核笨重。有了模块，我们的内核便是微内核的一种体现：

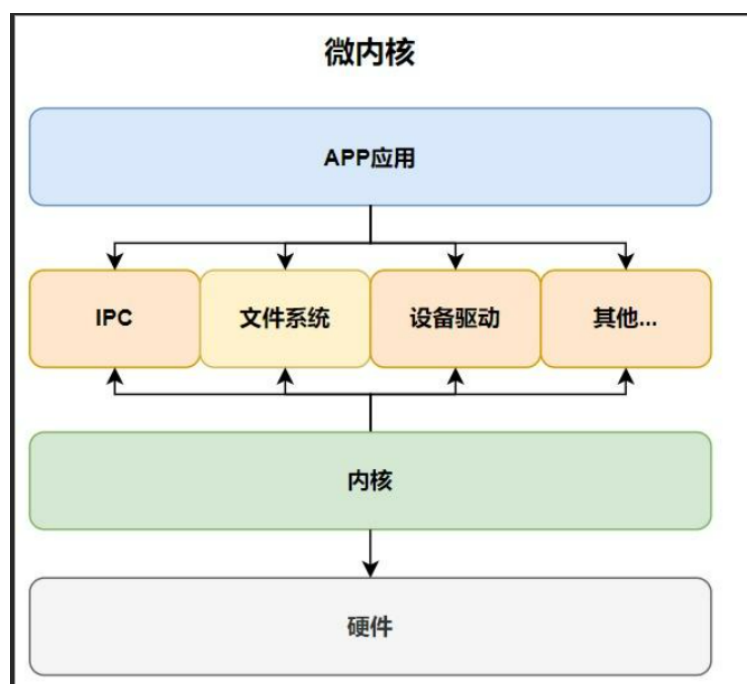


图 2.1: linux 内核模块

如上所示，我们要编写的便是一个文件系统模块，将其作为新功能加载到内核中的代码片段，在不需要时将其卸载即可。

模块进入内核

经过学习，我们得知可以运行 `lsmod` 命令来查看已经加载到内核中的模块，其是通过读取文件 `/proc/modules` 来获取信息。如下所示，成功显示了目前我们内核中的模块。


```

zjr@zjr-virtual-machine:~$ lsmod
Module                  Size  Used by
isofs                   49152  1
rfcomm                  81920  4
bnep                    28672  2
vsock_loopback         16384  0
vmw_vsock_virtio_transport_common 40960  1 vsock_loopback
vmw_vsock_vmci_transport 32768  2
vsock                  45056  7 vmw_vsock_virtio_transport_common,vsock_loopback,vmw_vsock_vmci_transport
nls_iso8859_1          16384  1
snd_ens1371             32768  2
snd_ac97_codec         155648  1 snd_ens1371
gameport               24576  1 snd_ens1371
ac97_bus               16384  1 snd_ac97_codec
intel_rapl_msr         20480  0
intel_rapl_common      40960  1 intel_rapl_msr
snd_pcm                135168  2 snd_ac97_codec,snd_ens1371

```

图 2.2

同时我们的模块文件都是 .ko 文件，当我们想将某一模块加载进入到内核中时，我们使用 **insmod** 命令即可。如下所示，我们导入模块 **xcraft**，执行命令：

```
1 sudo insmod xcraft.ko
```

显示结果如下，根据内核打印消息和 **lsmod** 命令的显示结果可以看到我们成功的加载 **xcraft** 模块到内核。

```

• zjr@zjr-virtual-machine:~/project2210132-232458/build$ sudo insmod xcraft.ko
• zjr@zjr-virtual-machine:~/project2210132-232458/build$ dmesg | tail -n 5
[ 4739.495576] usb 2-2.1: SerialNumber: 000650268328
[ 4867.637028] hrtimer: interrupt took 20117532 ns
[ 5173.582029] xcraft: module loaded
[ 5213.290563] xcraft: module unloaded
[ 5224.727973] xcraft: module loaded
• zjr@zjr-virtual-machine:~/project2210132-232458/build$ lsmod | grep "xcraft"
xcraft                  40960  0

```

图 2.3

当我们卸载内核模块时，执行 **rmmod** 命令即可。如下所示，我们执行命令：

```
1 sudo rmmod xcraft
```

卸载刚刚导入的 **xcraft** 内核模块。显示结果如下：

```

• zjr@zjr-virtual-machine:~/project2210132-232458/build$ sudo rmmod xcraft
• zjr@zjr-virtual-machine:~/project2210132-232458/build$ dmesg | tail -n 5
[ 4867.637028] hrtimer: interrupt took 20117532 ns
[ 5173.582029] xcraft: module loaded
[ 5213.290563] xcraft: module unloaded
[ 5224.727973] xcraft: module loaded
[ 5492.134387] xcraft: module unloaded
• zjr@zjr-virtual-machine:~/project2210132-232458/build$ lsmod | grep "xcraft"

```

图 2.4: 卸载模块显示图

由内核打印消息可以看到 **xcraft** 模块成功卸载，由 **lsmod** 显示内核中已经没有 **xcraft** 模块可以看到执行成功。

模块编程

如下是截取的我们编写的有关模块编程方面的代码内容，其属于 **fs.c** 中的结尾代码。注意在编写时务必包含如下头文件：

```
1 #include <linux/module.h>
```

其中 **module_init**、**module_exit** 两个宏函数分别接收编写的模块初始化函数和模块卸载函数。其中，当模块加载进入内核时会调用模块初始化函数，在模块卸载时候调用模块卸载函数。其中 **MODULE** 开头的三个宏函数是表示该模块的相关信息

```
1 static int __init XCraft_init(void){
2
3     // 初始化分配 inode 的仓库
4     int ret = XCraft_init_inode_cache();
5     if(ret){
6         pr_err("inode cache creation failed\n");
7         goto end;
8     }
9
10    ret = register_filesystem(&XCraft_fs_type);
11    if(ret){
12        pr_err("register_filesystem() failed\n");
13        goto end;
14    }
15
16    pr_info("module loaded\n");
17 end:
18    return ret;
19 }
20
21 static void __exit XCraft_exit(void){
22     int ret = unregister_filesystem(&XCraft_fs_type);
23     if(ret)
24         pr_err("unregister_filesystem() failed\n");
25
26     XCraft_destroy_inode_cache();
```

```

27     pr_info("module unloaded\n");
28 }
29
30 module_init(XCraft_init);
31 module_exit(XCraft_exit);
32
33 MODULE_LICENSE("GPL");
34 MODULE_AUTHOR("pray4u");
35 MODULE_DESCRIPTION("XCraft by pray4u");

```

1. `MODULE_LICENSE("GPL")`: 表示该模块的许可证是 `GPL`, 意味着其遵守 `GPL` 的条款;
2. `MODULE_AUTHOR("pray4u")`: 表示该模块的作者是 `pray4u`;
3. `MODULE_DESCRIPTION("XCraft by pray4u")`: 表示该模块的描述是“XCraft by pray4u”, 描述了该模块的功能和用途。

2.3.2 fat32 文件系统

总体概述

`fat` 文件系统是 `windows` 操作系统所使用的一种文件系统。FAT 文件系统用“簇”作为数据单元。一个“簇”由一组连续的扇区组成, 簇所含的扇区数必须是 2 的整数次幂。簇的最大值为 64 个扇区, 即 32KB。所有簇从 2 开始进行编号, 每个簇都有一个自己的地址编号。用户文件和目录都存储在簇中。

在 `fat` 文件系统中, 同时使用“扇区地址”和“簇地址”两种地址管理方式。这是因为只有存储用户数据的数据区使用簇进行管理, 所有簇都位于数据区。其他文件系统管理数据区域是不以簇进行管理的, 这部分区域使用扇区地址进行管理。文件系统的起始扇区为 0 号扇区。

整体布局

`fat` 文件系统的整体布局如下图2.5, 我们以其中的 `fat32` 为例:



图 2.5: fat 布局效果图

说明:

- 保留区中有一个重要的数据结构——系统引导扇区 (**DBR**)，它是 **fat32** 文件系统的第一个扇区，也称为 **DBR** 扇区。它包含了一系列重要信息，相当于超级块的作用。通过这些信息可以精确定位后面每个区域的大小和位置：
 - 每个扇区的字节数
 - 每簇扇区数
 - 保留扇区数
 - FAT 表的个数
 - 文件系统大小 (扇区数)
 - 每个 FAT 表大小 (扇区数)
 - 根目录起始簇号
 - 其他附加信息
- FAT 区由各个大小相等的 FAT 表组成，每一个相当于是对 FAT1 的备份；
- fat32 根目录通常位于 2 号簇，数据区由”簇地址”管理方式管理。

功能说明

由上面可以 fat 文件系统有两种地址管理方式，分别是”**扇区地址**”和”**簇地址**”两种。那么它们两者的转化计算公式如下：

某簇起始扇区号 = 保留扇区数 + 每个 FAT 表大小扇区数 × FAT 表个数 + (该簇簇号 - 2) × 每簇扇区数

此文件系统的功能与 fat 表紧密相关，下面介绍 FAT 表：

FAT 表由一系列大小相等的 FAT 表项组成，总的说来 FAT 表有如下特性：

1. fat32 中每个簇的簇地址，是有 **32bit (4 个字节)** 记录在 FAT 表中。FAT 表中的所有字节位置以 4 字节为单位进行划分，并对所有划分后的位置由 0 进行地址编号。0 号地址与 1 号地址被系统保留并存储特殊标志内容。从 2 号地址开始，每个地址对应于数据区的簇号，FAT 表中的地址编号与数据区中的簇号相同。我们称 FAT 表中的这些地址为 FAT 表项；
2. 如果某个簇未被分配使用，它所对应的 FAT 表项内的 FAT 表项值即用 0 进行填充，表示该 FAT 表项所对应的簇未被分配；
3. 当某个簇已被分配使用时，则它对应的 FAT 表项内的 FAT 表项值也就是该文件的下一个存储位置的簇号。如果该文件结束于该簇，则在它的 FAT 表项中记录的是一个文件结束标记，对于 fat32 而言，代表文件结束的 FAT 表项值为 **0x0FFFFFFF**；

4. 如上面所提到的，0 号表项和 1 号表项有着特殊含义，并不与簇进行对应。一般将 0 号表项值置为 **F8FFFF0F**，1 号表项置为 **FFFFFF0F**。如下是格式化后的 FAT 表表项内容：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	F8	FF	FF	0F	FF	FF	FF	0F	FF	FF	FF	0F	00	00	00	00	ayy.yyy.yyy.....
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

图 2.6: 格式化后的 FAT 表项内容

5. 新建文件时，如果新建的文件只占用一个簇，为其分配的簇对应的 FAT 表项将会写入结束标记。如果新建的文件不只占用一个簇，则在其所占用的每个簇对应的 FAT 表项中写入为其分配的下一簇的簇号，在最后一个簇对应的 FAT 表项中写入结束标记；
6. 新建目录时，只为其分配一个簇的空间，对应的 FAT 表项中写入结束标记。当目录增大超出一个簇的大小时，将会在空闲空间中继续为其分配一个簇，并在 FAT 表中为其建立 **FAT 表链**以描述它所占用的簇情况。

根据以上描述，下面我们展示如下 fat 表，并模拟读取起始于 3 号簇的文件：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	F8	FF	FF	0F	FF	FF	FF	FF	FF	FF	FF	0F	04	00	00	00	ayy.yyyyyyy.....
00000010	05	00	00	00	06	00	00	00	07	00	00	00	08	00	00	00
00000020	FF	FF	FF	0F	00	00	00	00	00	00	00	00	00	00	00	00	yy.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

图 2.7: fat 表

- 由文件的目录项中得知其第 1 簇存储在 3 号簇，查看 3 号 FAT 表项得知存储文件的下一个簇为 4 号簇，读取 4 号簇内容；
- 接下来读取 4 号表项得知存储文件的下一个簇为 5 号簇，读取 5 号簇内容，继续查看 5 号表项；
- 重复上述过程直到我们看到 9 号表项内容为 **FFFFFF0F**，说明已经是最后一簇，读取完毕。

结果评估

通过对 fat32 文件系统的调研，我们发现其适用于小文件系统。精巧、简便是其最大的特点。但是其不适用于大文件系统，因为其功能严重依赖于 **FAT** 表，这样会导致在大文件系统中 **FAT** 表所占大小过大，并且每次的 **FAT** 表项访问加跳转到磁盘扇区访问严重耗时。

但是我们从其中学习到了与扇区地址不同的地址管理方式，有时可以方便管理，提高效率。

2.3.3 simplefs 文件系统

总体概述

simplefs 的项目地址在[此处](#)。此文件系统比较精巧、简便，我们通过学习此文件系统掌握了各种 vfs 层的接口实现，并在此基础上实现了我们的方案。

整体布局

simplefs 文件系统的磁盘布局如下：



图 2.8: simplefs 文件系统

其中第 0 块是超级块，接下来是连续 inode 存储区，inode 位图，磁盘块位图和数据块。

功能说明

simplefs 的功能依赖于他的 **extent** 结构，每一个 inode 中都会包含一个 extent 索引块。如下展示目录 inode 的相关结构：

inode 中的 **ei_block** 字段存储 **extent** 索引块的物理块号，然后 **extent** 索引块中存储了一系列表示 **extent** 信息的结构。其中每一个 **extent** 占 8 个连续的磁盘块，**ee_block** 存储的是此 extent 的起始块的逻辑块号，**ee_start** 存储的是此 extent 的起始块的物理块号。每一个磁盘块中存储了一系列目录项 (inode 号以及 inode 索引的文件名)。如果 inode 索引的是文件的话，磁盘块里面就是存储的文件数据。

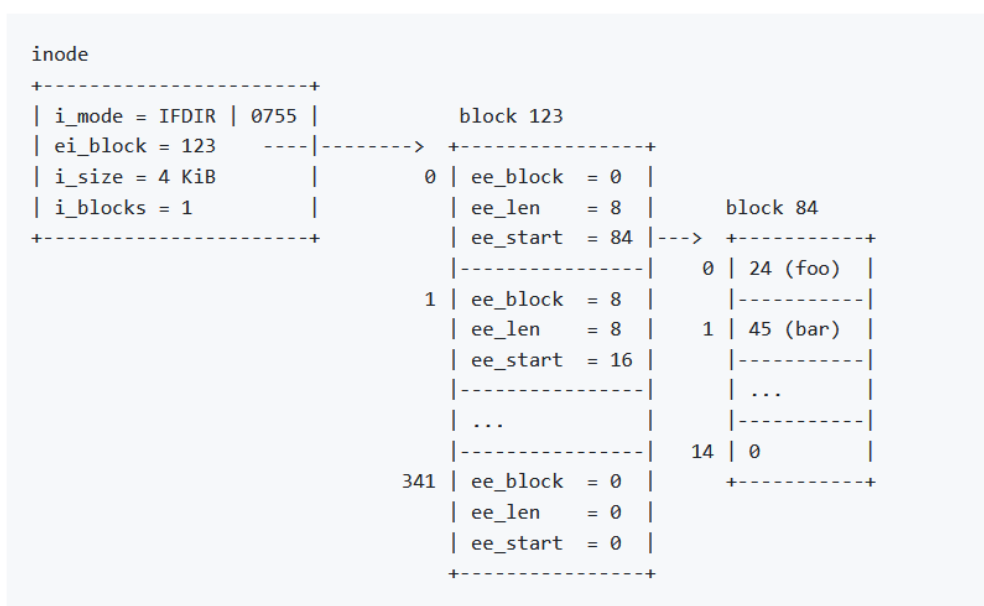


图 2.9: simplefs 的 extent 结构

结果评估

通过对 **simplefs** 文件系统的调研，我们系统掌握了各种 **vfs** 层的接口和核心数据结构实现。这对之后我们文件系统的代码编写提供了基础，并且在此基础上实现了我们的相关想法。

2.3.4 调研结果

我们经过调研之后，掌握 **Linux** 内核虚拟文件系统工作原理和与 **VFS** 层的对接，会使用 **Linux** 内核模块编程来导入和卸载我们的文件系统模块。并且通过对优秀文件系统的设计思路和源码学习，我们也确定了我们的设计思路和方案设计。具体的新文件系统的设计思路如下：

1. 实现与 **VFS** 层的一系列接口函数；
2. 对文件系统的核心数据结构进行实现；
3. 实现权限属性管理；
4. 实现与用户程序的接口；
5. 格式化工具的编写；
6. 编写测试脚本对我们的文件系统进行一系列测试工作。

3 系统框架设计

3.1 核心重要组件

我们新设计的文件系统的系统框架主要包括以下重要模块：

1. 超级块 (superblock):

超级块作为文件系统元数据的核心，携带了文件系统的全局信息，比如文件系统的大小，可用的磁盘块数量以及索引节点的数量等等。这些信息是文件系统运行的关键，为文件系统的正常运行提供了基础。因此在新的文件系统中，超级块扮演着极其重要的角色，是文件系统管理的核心；

2. 索引节点 (inode):

文件系统中，每个文件和目录都会有一个对应的索引节点 (inode)。索引节点包含了文件或者目录的元数据，例如：权限，所有者，大小，创建和修改时间等等。最重要的一点是索引节点包含了指向文件数据的指针；

3. 目录 (Directory):

目录中包含了一系列目录项，每一个目录项都是对一个文件或者子目录的索引节点的引用。每次通过目录项，便可以更快速地访问到目标文件或者目录；

4. 数据块 (Blocks):

数据块是文件系统中存储文件数据的基本单位。每一个数据块都有一个固定的大小，在我们设计的文件系统中，这个大小被设置为了 4KB。文件数据会被分成多个块，然后分别存储在这些数据块中；

5. 位图 (Bitmaps):

位图用来管理 inode 和 Blocks 的使用情况。例如，索引节点位图中的每一位都对应一个索引节点。如果这个索引节点被使用了，那么相应的位就会被置为 1，否则就是 0。数据块位图的工作方式与此类似；

6. VFS 接口 (VFS Interface):

新设计的文件系统需要在 linux 中运行，这需要我们实现一系列 VFS 接口函数。这些函数包括 read、write、mkdir 以及 rmdir 等等。通过实现这些接口，我们的文件系统便可以成功与 Linux 内核进行交互；

7. 块组 (block group):

我们设计的文件系统中引入了块组，通过块组我们可以做到按需分配。当我们的现存资源足够的情况下，我们不申请新的块组；当现存资源不够的情况下，我们才申请新的块组。块组的引入也减小了我们 mkfs 格式化磁盘的工作量，提高了效率；

8. 块组描述符 (group descriptor):

我们设计的文件系统中块组描述符用来描述块组的相关信息，每一个指定编号的块组对应一个指定编号的块组描述符。块组描述符中存储了所对应块组中的空闲 inode 以及空闲 block 的个数，inode bitmap 和 block bitmap 的位置信息等。这对我们的 inode 和 block 的分配十分有用。

上述组件一起构成了我们文件系统的框架。对于这些组件的设计与实现，一直贯穿于我们整个系统的框架设计之中。

3.2 框架特点

我们的项目框架具有以下特点：

1. 代码的封装性高、可读性强：项目开发过程中，我们非常注重代码的封装性和模块化。每一个功能模块我们都经过了仔细的设计来确保其逻辑清晰并能独立工作。并且我们在实现代码的基础上还添加了详细的注释，更加增加了我们代码的可读性；
2. 对第三方库依赖性弱：为了确保项目的长期稳定性，我们尽量减少对外部第三方库的依赖。通过这种方法，我们可以降低因第三方库更新或 API 变动导致的潜在风险。这同时意味着我们的文件系统在未来更有可能与各种环境和平台兼容；
3. 良好的内核兼容性：我们设计文件系统时考虑了适配多个 Linux 版本的需求。无论是主流的 Linux 发行版还是基于低版本 Linux 内核的版本，我们的文件系统都能够在其上平稳运行。这达到了较好的兼容性，使得用户可以在任意 Linux 环境中运行我们的项目；
4. 引入块组进行管理：我们的文件系统引入了块组进行管理，实现按需分配。提高了空间资源的利用率，减小了 mkfs 格式化磁盘的难度和工作量；
5. 目录文件引入哈希树来管理目录项：我们的文件系统引入了哈希树来管理目录项，提高了遍历目录项的效率，相比于之前的顺序遍历磁盘块大大降低了时间复杂度。

3.3 总体框架布局

XCRAFT 文件系统的磁盘布局如图3.10所示。

第 0 个块组包括:super 超级块、块组描述符、块组对应的 inode 位图、块组对应的 block 位图、储存 inode 的块、其他数据块。

其他块组的结构为：块组对应的 inode 位图、块组对应的 block 位图、储存 inode 的块、其他数据块。

其中一个块大小为 4096 字节，那么一个块作为 block 位图，可以跟踪 8×4096 个块，所以一个块组的块的数量是固定的，为 8×4096 个。

super	group descriptor	inode_bitmap	block_bitmap	inodes	dbs	inode_bitmap	block_bitmap	inodes	dbs	...
-------	------------------	--------------	--------------	--------	-----	--------------	--------------	--------	-----	-----

图 3.10: XCRAFT 文件系统布局

```

1  /*
2  XCraft partition layout
3  * +-----+
4  * | superbblock | 1 block
5  * +-----+
6  * | descriptor | sb->s_gdb_count blocks
7  * +-----+
8  * | ifree bitmap | bg_0(32768 blocks) 4096*8
9  * +-----+
10 * | bfree bitmap | XCRAFT_BFREE_PER_GROUP_BLO blocks
11 * +-----+
12 * | inode_str_ |
13 * |          | blocks | bg_nr_inodes/XCRAFT_INODES_PER_BLOCK blocks
14 * +-----+
15 * | data blocks | bg_nr_blocks-1-XCRAFT_BFREE_PER_GROUP_BLO-inode_str_blocks blocks
16 * +-----+
17 * | ...other bg | ...
18 */

```

3.4 核心原理

3.4.1 块组原理

我们定义的块组如上介绍的那样第 0 个块组相比于其他块组多出了两个结构，分别是超级块和块组描述符。我们的块组采用的是延迟初始化的模式，在格式化磁盘时只对第 0 个块组进行了初始化，对其他块组并没有进行初始化。

磁盘结构定义

我们对其他块组初始化的方式便是首先由其块组描述符确定磁盘块的个数和 inode 个数，由这两者便可以知道其块组中 inode 位图、磁盘块位图以及 inode 各占了多少磁盘块，然后由这还可以知道 data blocks 所占的块数，最终确定了这个块组的磁盘布局结构，并

且对 inode 位图和磁盘块位图进行写磁盘初始化，对块组描述符信息进行更新，修改其标识初始化的字段。这样，我们便完成了一次块组延迟初始化。

延迟初始化应用场景

每当我们对块组进行了一次初始化，都标志着我们目前已经初始化的块组的 inode 或者磁盘块已经不够分配。当我们希望分配磁盘块或者 inode 时，我们会访问已经初始化块组的块组描述符，如果其信息显示其已经没有空闲的 inode 和磁盘块满足我们的需求，我们便需要对新的块组进行初始化，这便是一次块组延迟初始化。通过块组这种组织结构，有着明显的优点：

- 减轻了格式化磁盘的压力，减小了工作量；
- 块组的模式使得文件元数据都尽量地集中在同一块组，在磁盘块随机访问的情形下提高了访问效率；
- 充分利用了磁盘空间，减少了磁盘碎片的产生；
- 不采用块组的情形下，位图大小过于庞大，访问需要遍历多个磁盘块，查找效率低；采用块组降低了位图大小，每一个位图管理一个块组，大大提高了查找访问效率。

3.4.2 哈希 B+ 树原理

目录条目的线性数组对性能影响较大，因此我们增加了一项新功能，以提供更快（但特殊）的平衡树，该树以目录条目名称的哈希值为关键。

通常情况下，XCraft 文件系统目录最初采用无索引方式存储目录条目。在这种无索引模式下，目录条目按顺序存储在目录的单个数据块中。当目录条目的数量包含在单个数据块中时，这种方法是有效的。

但是，一旦目录条目的数量超过了单个数据块的容量，目录就可以转换为索引树结构，即散列树。启用哈希树功能后，这种转换就会发生。

如果在 inode 中设置了哈希树标志，该目录就会使用散列 btree (htree) 来组织和查找目录条目。

哈希树结构包括根节点、中间节点和叶节点。

如果目录没有索引，其数据块将存储 dirents（目录条目）。但是，如果目录有索引（即使用哈希树），前几个数据块将存储与哈希树结构相关的信息，而其余数据块将存储 dirents。在哈希树的情况下，第 1 个数据块是哈希树根的主机，由 `struct dx_root` 表示，其后是 `struct dx_entry` 实例。这些 `dx_entry` 结构提供哈希树层次结构中下一级数据块的地址。

建树过程 随着目录中 dirents 数量的增加，用于存储这些 dirents 的树的大小和深度也会增加。

起初，目录处于无索引状态，dirents 存储在单个块中。但是，如果 dirents 的数量超过了单个块的容量，目录就会转换为索引树结构。在转换过程中，会创建两个新块。其中一个块将成为根块，并使用 dx_root 结构进行初始化。然后按升序对 dirents 进行排序，前半部分 dirents 保留在原始块中，而剩余的 dirents 则移到另一个新块中。两个 dx_entry 结构被添加到 dx_root，以保持目录的索引状态。

假设目录最初未编入索引，包含 40 个 dirents，目录的数据块已完全满载。当添加第 41 个 dirent 时，目录将转换为索引结构。

在转换过程中，会创建两个新块。其中一个块成为根块，40 个字段被排序。前 20 个 dirents 保留在原始块中，而其他 20 个 dirents 则被移到新块中。新区块的 dx_entry 的哈希值等于该区块中第一个 dirent 的哈希值。另一方面，初始数据块的 dx_entry 的哈希值为 0。

经过这样的重组，目录的索引结构中就可以容纳新增的第 41 个 dirent 了。

将目录转换为索引结构后，如果有任何叶块被 dirents 填满，就会创建一个新块来容纳新增的 dirents。已填满的叶块中的 dirents 将在已填满的块和新块之间重新平均分配。

这种重新分配可确保 dirents 在叶子区块之间均匀分布，防止任何一个区块的 dirents 超载。通过创建新块并重新分配字典，目录的索引结构可以有效地存储和管理更多字典，同时保持块的均衡使用。

举例说明下图描述了目录哈希树的初始配置。此时，散列树的深度为 0，由三个叶块 (1、2 和 3) 组成。

```
debugfs: htree dir2
Root node dump:
    Reserved zero: 0
    Hash Version: 1
    Info length: 8
    Indirect levels: 0
    Flags: 0
    Number of entries (count): 3
    Number of entries (limit): 507
    Checksum: 0x26e53965
    Entry #0: Hash 0x00000000, block 1
    Entry #1: Hash 0x7d9c8fa2, block 2
    Entry #2: Hash 0xbfe55cc0, block 3
```

图 3.11: 目录哈希树初始配置

现在，我们考虑在同一目录下创建 20 个新文件的情况。添加这些新文件后，哈希树配置更新如下：

```

debugfs: htree dir2
Root node dump:
    Reserved zero: 0
    Hash Version: 1
    Info length: 8
    Indirect levels: 0
    Flags: 0
    Number of entries (count): 4
    Number of entries (limit): 507
    Checksum: 0x1e7a9fd0
    Entry #0: Hash 0x00000000, block 1
    Entry #1: Hash 0x46aa34e4, block 4
    Entry #2: Hash 0x7d9c8fa2, block 2
    Entry #3: Hash 0xbfe55cc0, block 3

```

图 3.12: 目录下创建文件的哈希树配置

哈希树的深度保持不变，表明不需要增加层级。不过，在哈希树中的区块 1 和区块 2 之间创建并插入了一个新的叶子区块（区块 4）。这是必要的，因为一些新添加的条目完全填满了块 1。为了容纳新增的字段，又创建了一个新块（块 4），并在块 1 和块 4 之间均匀地重新分配字段，以保持哈希树结构的平衡。

碰撞 在发生哈希值碰撞时，需要注意的是，哈希值主要用于定位特定哈希值范围内的字典块。如果发生碰撞，且多个 `dirents` 共享相同的哈希值，它们将被一起存储在同一个区块中。一旦找到目标数据块，就会在该数据块内进行线性搜索，以找到特定的 `dirent`。

在添加新的 `dirent` 时，如果该 `dirent` 与完全填满的区块中的 `dirent` 发生哈希碰撞，则会创建一个新的区块。这些 `dirent` 将根据哈希值在两个区块之间重新平均分配。不过，值得注意的是，具有相同哈希值的 `dirent` 仍会保留在同一区块中。这就确保了尽管进行了重新分配，但具有碰撞哈希值的数据字段仍会保留在同一数据块中（按照 `hash` 值升序排列的），以便在线性搜索操作中进行高效检索。

总结：哈希树方案有助于快速高效地搜索、添加和删除目录条目（`dirents`）。这一功能对于容纳大量文件的目录尤为重要，因为它能最大限度地减少线性搜索的需要，确保快速访问所需的目录。

`rmdir` 只需要将 `d_entry` 删除即可，树的结构不用改变

3.4.3 扩展树原理

传统的文件 `inode` 使用的是间接索引块的方式来索引逻辑块，但是这种方式无法存储大文件，并且访问效率过低，不适合大文件系统。在此我们使用基于 B+ 树的扩展树方式来替代间接索引块的方式，扩展树不仅可以存储更大的文件，还可以减少磁盘碎片，将文件内容尽可能连续存放，当映射逻辑块时其实便是对树的搜索，查询效率大大提高。

扩展树的结构包括根节点、索引节点和叶子节点。

其中上述每个节点都会有一个头结构 `XCraft_extent_header`。然后根节点的头结构后面便是一系列索引项 `XCraft_extent_idx` 或者扩展项 `XCraft_extent`。索引节点的头结构后面便是一系列索引项 `XCraft_extent_idx`，叶子节点的头结构后面便是一系列的扩展项 `XCraft_extent`。其中根节点的内容其实存放在 `inode` 的 `i_block` 结构中，叶子节点和索引节点的内容从磁盘块起始存放，最大可以达到一个磁盘块。

建树过程 最开始我们映射逻辑块号为物理块号时，根节点此时头结构的后面是一系列的扩展项。当根节点无法容纳扩展项之后，便要开始增加深度，此时是创建一个新块，将根节点的内容复制到新块中。然后将原来的根节点的扩展项变为只有一个索引项，索引项指向新块。

之后我们将逻辑块号映射成物理块号的过程中，首先我们会从根节点开始向下对每一层进行二分查找，并将其每一层的相关信息记录在 `XCraft_ext_path` 中。然后我们对叶子节点层的 `path` 中记录的扩展项进行判断，如果映射的起始逻辑块号在扩展项的逻辑块号范围中，我们便调整映射长度，并且得到映射的起始逻辑块号。否则便是我们要插入新的 `extent`。我们先对 `map` 结构中的映射逻辑块号范围进行去重，然后我们要插入的新的 `extent` 如果起始逻辑块号比当前叶子节点中的最后一个 `extent` 的起始逻辑块号大，我们便看能不能在下一个最近的叶子节点中插入。如果下一个最近的叶子节点中有空闲的扩展项可以供我们插入，我们便直接插在最近的叶子节点中。否则，我们便要从当前的叶子节点向上顺着 `path` 回溯，回溯到有空闲索引项的层。如果一直到根节点都没有空闲索引项，我们便重新增加深度；找到有空闲索引项的层之后，我们便需要往下创建新块直到叶子节点层，形成一个子树。最后我们将这个子树接在空闲索引项的层。然后再重新遍历扩展树，插入新的 `extent` 到叶子节点即可，最后自下而上检查是否需要修改每一层对应索引项映射的起始逻辑块号。

删除逻辑块映射的物理块 我们会首先会拿起起始逻辑块号从根节点开始向下对每一层进行二分查找，将每一层的相关信息记录在 `XCraft_ext_path` 中。然后对 `path` 层记录的叶子节点层的扩展项映射的逻辑块号范围是否包含其起始逻辑块号进行判断，如果在的话，就可以释放该扩展项中此起始逻辑块号向后的所有物理块。如果起始逻辑块号恰好是此扩展项中映射的起始逻辑块号，我们便删除此扩展项，后面的扩展项前移；否则就修改此扩展项中的相关信息即可。最后检查是否需要自下而上更新 `path` 中每一层对应的索引项映射的起始逻辑块号。

删除扩展树 删除扩展树我们采用 DFS 的思想，先一直向下遍历至最后一层索引节点，删除其中每一项对应的叶子节点及其映射的扩展磁盘块。然后回溯再继续重复上述流程。最后根节点处就将其全部置 0 即可。

3.5 核心数据结构实现

我们在这里将讲述 XCraft.h 中的重要数据结构的设计构思

3.5.1 超级块

磁盘结构定义

我们超级块的磁盘结构定义如下：

```

1 struct XCraft_superblock{
2     __le32 s_inodes_count; /* number of inodes */
3     __le32 s_blocks_count; /* number of blocks */
4     __le32 s_free_blocks_count; /* number of free blocks */
5     __le32 s_free_inodes_count; /* number of free inodes */
6     __le32 s_blocks_per_group; /* number of blocks per group */
7     __le32 s_inodes_per_group; /* number of inodes per group */
8     __le32 s_groups_count; /* number of groups */
9     __le32 s_last_group_blocks; //最后一个组的块数
10    __le16 s_magic; /* magic number */
11    __le16 s_inode_size; /* inode size */
12 };

```

我们定义的超级块只占一个磁盘块，且为第 0 块，如下是所在第 0 磁盘块中的具体布局：

```

1     s_inodes_count; /* number of inodes */
2     s_blocks_count; /* number of blocks */
3     s_free_blocks_count; /* number of free blocks */
4     s_free_inodes_count; /* number of free inodes */
5     s_blocks_per_group; /* number of blocks per group */
6     s_inodes_per_group; /* number of inodes per group */
7     s_groups_count; /* number of groups */
8     s_last_group_blocks; //最后一个组的块数
9     s_magic; /* magic number */
10    s_inode_size; /* inode size */
11    padding; /* 一个磁盘块剩余的填充 */

```

下面解释我们定义的各个字段的含义：

1. **s_inodes_count**: XCraft 的 inode 总个数；

2. `s_blocks_count`: XCraft 的磁盘块总个数;
3. `s_free_blocks_count`: XCraft 现在有多少空闲磁盘块;
4. `s_free_inodes_count`: XCraft 现在有多少空闲 inode;
5. `s_blocks_per_group`: **普通块组**一个块组中有多少磁盘块;
6. `s_inodes_per_group`: **普通块组**一个块组中多少 inode;
7. `s_groups_count`: XCraft 有多少个块组;
8. `s_last_group_blocks`: XCraft 最后一个块组有多少磁盘块, 在我们后续的格式化工具编写中我们可能会把**最后一个块组和前一个块组进行合并**, 所以最后一个块组的情况和前面的普通块组的情况可能会有所不同;
9. `s_magic`: 文件系统幻数, 用于区分不同的文件系统;
10. `s_inode_size`: XCraft 一个 inode 的大小, 以字节为单位。

内存结构定义

当我们将磁盘中的超级块内容加载到内存中, 我们定义的结构如下:

```

1  typedef unsigned int xcraft_group_t;
2
3  struct XCraft_superblock_info{
4      unsigned long s_blocks_per_group; /* number of blocks per group */
5      unsigned long s_inodes_per_group; /* number of inodes per group */
6      unsigned long s_last_group_inodes; //最后一个组的 inode 数
7      unsigned long s_last_group_blocks; //最后一个组的块数
8      unsigned long s_gdb_count; /* number of group descriptor blocks */
9      unsigned long s_desc_per_block; /* number of group descriptors per block */
10     __u32 s_la_init_group; //最后一个初始化的块组
11     xcraft_group_t s_groups_count; /* number of groups */
12     __u32 s_hash_seed[4]; /* hash seed */
13     struct buffer_head *s_sbh; /* superblock buffer_head */
14     struct XCraft_superblock* s_super; /* superblock */
15     struct buffer_head **s_group_desc; //指向组描述符数组的块的指针
16     struct XCraft_ibmap_info **s_ibmap_info; //指向 bitmap 的指针
17
18 };

```

其中 `XCraft_ibmp_info` 的定义如下：

```

1 struct XCraft_ibmp_info{
2     unsigned long*ifree_bitmap;//inode bitmap
3     unsigned long*bfree_bitmap;//block bitmap
4 };

```

我们在内存中定义的超级块的结构为上述的 `XCraft_superblock_info`，下面具体介绍其中某些字段的含义：

1. `s_last_group_inodes`：最后一个块组的 inode 个数。这个与我们之前提到的 mkfs 格式化工具编写中的最后一个块组合并有关；
2. `s_last_group_blocks`：最后一个块组的磁盘块数。这个与我们之前提到的 mkfs 格式化工具编写中的最后一个块组合并有关；
3. `s_gdb_count`：XCraft 中所有块组的块组描述符所占的块数；
4. `s_desc_per_block`：一个磁盘块中可以容纳多少个块组描述符；
5. `s_La_init_group`：最后一个初始化的块组。通过这个字段我们可以知道当前已经初始化的块组有多少，并且每次当资源不够时我们也可以快速得知接下来要初始化的块组是什么；
6. `s_hash_seed`：哈希种子，后续进行文件名哈希值计算时可能会用到；
7. `s_group_desc`：指向块组描述符数组的块的指针。因为块组描述符可能占了多个块，我们在这里使用指针数组便可以读取每个块的块组描述符的内容；
8. `s_ibmap_info`：每一个块组都会对应一个 `XCraft_ibmap_info` 结构，其里面存储了每个块组加载到内存中的 inode 位图和 block 位图；
9. `s_super`：对应着超级块磁盘结构的内容。当我们后续进行操作时如果需要修改相关字段直接对 `s_super` 进行修改，写回时将 `s_super` 的内容复制到 `s_sbh` 中然后写回即可。

与 VFS 层的对接

当我们将磁盘中超级块的内容读入到内存中，我们会利用其内容填充 `XCraft_superblock_info` 结构体和 VFS 层的 `super_block` 中的字段。然后将 VFS 层的 `super_block` 与 `XCraft_superblock_info` 结构体以如下方式进行关联：

```

1 #define XCRAFT_SB(sb) ((struct XCraft_superblock_info *)((sb)->s_fs_info))

```

即最开始会把 VFS 层的 `super_block` 的 `s_fs_info` 字段指向相应的 `XCraft_superblock_info` 结构体。如上便完成了结构上与 VFS 层的对接工作。

3.5.2 块组描述符

磁盘结构定义

我们的块组描述符定义如下：

```

1  struct XCraft_group_desc{
2      __le32 bg_block_bitmap; /* block bitmap block 物理块号 */
3      __le32 bg_inode_bitmap; /* inode bitmap block 物理块号 */
4      __le32 bg_inode_table; /* 物理块号 */
5      __le16 bg_nr_inodes; /* number of inodes in this group */
6      __le16 bg_nr_blocks; /* number of blocks in this group */
7      __le16 bg_free_blocks_count; /* number of free blocks */
8      __le16 bg_free_inodes_count; /* number of free inodes */
9      __le16 bg_used_dirs_count; /* number of directories */
10     __le16 bg_flags; /*EXT4_BG_flags(INODE_UNINIT,etc) 表示该块组的信息 初始化 or 未初始化 */
11
12 };

```

如上我们解释各个字段的含义：

1. `bg_block_bitmap`。其对应块组的 block 位图所在的起始物理块号；
2. `bg_inode_bitmap`。其对应块组的 inode 位图所在的起始物理块号；
3. `bg_inode_table`。其对应块组 `inode_store` 的起始物理块号；
4. `bg_nr_inodes`。其对应的块组中的 inode 的个数；
5. `bg_nr_blocks`。其对应的块组中的 block 的个数；
6. `bg_free_blocks_count`。其对应块组中的空闲块个数；
7. `bg_free_inodes_count`。其对应块组中的空闲 inode 个数；
8. `bg_used_dirs_count`。本块组中目录的个数；
9. `bg_flags`。标志位，主要是用于判断此块组是否初始化。

由上面我们在超级块中提到的那样，我们将块组描述符读入内存是存储在 `XCraft_superblock_info` 结构体当中，即我们加载超级块到内存时便会首次读取块组描述符的内容并将其存储在 `XCraft_superblock_info` 结构体当中。

3.5.3 inode 索引节点

磁盘结构定义

我们 inode 索引节点的磁盘结构定义如下：

```

1  #define XCRAFT_N_BLOCK 15    //12 个直接索引块 2 个 1 级间接索引块 1 个 2 级间接索引块
2  #define XCRAFT_N_DIRECT 12 //直接索引块的个数
3  #define XCRAFT_N_INDIRECT 2 //1 级间接索引块个数
4  #define XCRAFT_N_DOUBLE_INDIRECT 1 //2 级间接索引块个数
5  struct XCraft_inode{
6      __le16 i_mode; /* file mode */
7      __le16 i_uid; /* owner UID */
8      __le32 i_size_lo; /* file size in bytes */
9      __le32 i_atime; /* access time */
10     __le32 i_ctime; /* creation time */
11     __le32 i_mtime; /* modification time */
12     __le32 i_nr_files; /* number of files */ // 目录下才有用
13     __le16 i_gid; /* owner GID */
14     __le16 i_links_count; /* hard links count */
15     __le32 i_blocks_lo; /* number of blocks 文件或者目录所使用的块的个数 */
16     __le32 i_flags; /* file flags B+ 树等 */
17     __le32 i_block[XCRAFT_N_BLOCK]; /* pointers to blocks */
18     char i_data[32]; /* store symlink content */
19 };

```

下面我们解释某些字段的含义：

1. **i_nr_files**：这个字段仅仅当 inode 索引的文件类型是目录时才有用，表示目录下有多少文件和子目录。如果 inode 索引的文件类型不是目录，那么将其置 0；
2. **i_block**：如果 inode 索引的文件类型是目录时，我们只会使用 **i_block[0]** 所对应的磁盘块，其他的都会置 0，这是因为我们使用哈希树；如果 inode 索引的文件类型是文件时，如上所示，我们会将前 12 个作为直接索引块，第 13、14 个作为一级间接索引块，第 15 个作为 2 级间接索引块；
3. **i_flags**：此字段仅仅当 inode 索引的文件类型是目录时有效。通过此字段可以判断此时是否生成了哈希树。是否生成了哈希树对于我们来说对应着不同的目录下文件的搜索方式。

内存结构定义

当我们将磁盘中的 inode 读入内存时，我们会定义如下结构：

```

1 struct XCraft_inode_info{
2     //ino 逻辑 Inode 号接在 inode->i_no 中
3     char i_data[32]; /* store symlink content */
4     __u32 i_nr_files; //只有其是目录才有用，用于确定其下有多少文件和目录
5     xcraft_group_t i_block_group; //所在组号
6     unsigned long i_flags; //标志位 区别哈希树和普通文件等
7     unsigned int i_block[XCRAFT_N_BLOCK]; //指向数据块的指针
8     struct inode vfs_inode;
9 };

```

接下来我们解释上述结构体中的某些字段：

1. **i_nr_files**。对应着其磁盘结构中的 **i_nr_files** 字段，只有其是目录此字段才有用，用于确定其下有多少文件和目录；
2. **i_block_group**。用于确定此 inode 所在的块组号；
3. **i_flags**。对应着其磁盘结构中的 **i_flags** 字段，也是只有其是目录才会有效，用于判断此时是否生成了哈希树；
4. **i_block**。对应着磁盘结构中的 **i_block** 字段；
5. **vfs_inode**。将 VFS 层的 inode 作为自己的成员变量，这便是 inode 与 VFS 对接的方式。

与 VFS 层的对接

当我们从磁盘获取 inode 时，会由磁盘获取的 inode 的内容填充 **XCraft_inode_info** 中的字段内容与 VFS 层 **inode** 中的内容。然后我们将 **XCraft_inode_info** 和 VFS 层的 **inode** 通过如上提到的方式进行联系。

我们由 VFS 层的 **inode** 获取 **XCrfat_inode_info** 便可以通过以下方式获取：

```

1 #define XCRAFT_I(inode) (container_of(inode, struct XCraft_inode_info, vfs_inode))

```

综上，我们的 inode 索引结构便实现了与 VFS 层的对接。

3.5.4 目录项

我们定义的目录项磁盘结构如下：

```

1 struct XCraft_dir_entry{
2     //对于 u8 类型的数据，字节序转换是不必要的。
3     //其他数据要转换 le16_to_cpu
4     //le32_to_cpu
5     __le32 inode; /* inode number */
6     __le16 rec_len; /* record length */
7     __u8 name_len; /* name length */
8     __u8 file_type; /* file type */
9     char name[XCRAFT_NAME_LEN]; /* file name */
10 };

```

如上，我们接下来解释磁盘目录项结构其中字段的含义：

1. **inode**：标识其对应的文件的索引节点的 inode 号，依据 inode 号我们很容易获取到磁盘中的 inode 结构；
2. **rec_len**：标识此目录项的大小；
3. **name_len**：标识文件名的长度；
4. **file_type**：标识其对应文件的类型是什么；
5. **name**：存储文件名。

综上这便是我们定义的在磁盘中的目录项数据组织形式。

3.5.5 哈希 B+ 树相关

dx_entry

```

1 struct dx_entry{
2     __le32 hash; /* hash value */
3     __le32 block; /* block + ? 逻辑块号 存疑 */
4 };

```

dx_root

```

1  #define XCRAFT_HTREE_VERSION 6
2  struct dx_root{
3      //对于 u8 类型的数据，字节序转换是不必要的。
4      //其他数据要转换 le16_to_cpu
5      struct dx_root_info{
6          __u8 hash_version; /* hash version */
7          __u8 indirect_levels; /* 0 if no dx_node else 1 */
8      //    __le16 limit; //最大目录项数 header + entries
9      //    __le16 count; //目录项数 header + entries
10     }info;
11     struct dx_entry entries[]; /* entries 一个块后面全部是 dx_entry */
12 };

```

dx_node

```

1  struct dx_node
2  {
3      //对于 u8 类型的数据，字节序转换是不必要的。
4      //其他数据要转换 le16_to_cpu
5      // __le16 limit; /* limit */
6      // __le16 count; /* count */
7      __le16 fake; /* fake */ // 赋值为 0
8      struct dx_entry entries[]; /* entries */
9  };

```

如上，dx_root 为我们的哈希树的根节点，其中 dx_root_info 中存储的便是哈希树的总体信息，其中 hash_version 字段为计算文件名哈希值使用的算法标识，indirect_levels 标识了哈希树的层数。当启用哈希树之后，indirect_levels 为 0 表示当前只有 dx_root，没有 dx_node。一般地说，如果 indirect_levels 为 n，则哈希树共有一级 dx_root，n 级 dx_node。

dx_node 为哈希树的中间索引节点，其中 fake 无意义，在这里赋值为 0，只起到占位的作用。

dx_root 和 dx_node 都有 dx_entry 数组结构，其中 dx_entry 的结构如上面代码所示，它的第一个字段 hash 存储的便是 hash 值，第二个 block 存储的便是其索引的下一级块的块号。

由于我们查找时不需要知道第 0 个 `dx_entry` 的 hash 值 (小于第一个 `dx_entry` 的 hash, 即在第 0 个 `dx_entry` 索引的块中), 所以我们将第 0 个 `dx_entry` 的 hash(32 位), 储存 2 个 16 位的 `limit`、`count`。`limit`(代表此 `dx_node` 或者 `dx_root`) 最多储存 `dx_entry` 的数量, 而 `count` 代表目前存在的 `dx_entry` 数量。

`dx_frame`

```

1 struct dx_frame
2 {
3     struct buffer_head *bh; //dx_entry 所在磁盘块
4     struct dx_entry *entries; //同一级的 dx_entry
5     struct dx_entry *at; //最终的 dx_entry
6 };

```

`dx_frame` 是我们在搜索目录项时使用的, 在遍历中它用于存储我们遍历到的每一级的相关内容。它有三个字段, 分别是 `bh`、`entries` 和 `at`。其中 `bh` 用于存储遍历到的那一级的磁盘块的内容的缓冲区, `entries` 用于存储每一级 `dx_entry` 数组的起始位置, `at` 存储我们通过哈希值二分查找到的对应的 `dx_entry`。于是, 有了这一级结构我们便可以通过先声明一个 `dx_frame` 数组, 在搜索过程中由这个数组存储每一级信息, 最终得知完整的遍历位置。

3.5.6 扩展树相关

`XCraft_map_blocks`

```

1 struct XCraft_map_blocks {
2     unsigned int m_pblk;
3     unsigned int m_lblk;
4     unsigned int m_len;
5 };

```

`XCraft_extent_header`

```

1 struct XCraft_extent_header {
2     __le16 eh_magic; /*magic number*/
3     __le16 eh_entries; /* number of valid entries */
4     __le16 eh_max; /* capacity of store in entries */
5     __le16 eh_depth; /* has tree real underlying blocks */

```

```

6     __le32  eh_unused;
7 };

```

XCraft_extent_idx

```

1 struct XCraft_extent_idx {
2     __le32  ei_block;    /* index covers logical blocks from 'block' */
3     __le32  ei_leaf;     /* physical block*/
4     __le32  ei_unused;
5 };

```

XCraft_extent

```

1 struct XCraft_extent {
2     __le32  ee_block;    /* first logical block extent covers */
3     __le32  ee_len;      /* number of blocks covered by extent */
4     __le32  ee_start;    /* first physical block extent covers */
5 };

```

XCraft_ext_path

```

1 struct XCraft_ext_path {
2     unsigned int p_block;
3     __u16 p_depth;
4     __u16 p_maxdepth;
5     struct XCraft_extent *p_ext;
6     struct XCraft_extent_idx *p_idx;
7     struct XCraft_extent_header *p_hdr;
8     struct buffer_head *p_bh;
9 };
10

```

如上，XCraft_map_blocks 是我们映射逻辑块号为物理块号使用的，其中 m_pblk 存放最终 m_lblk 逻辑块号映射的物理块号，m_len 为映射的连续磁盘块的数量。

XCraft_extent_header 为每个节点的头结构，其中 eh_depth 存放所处深度，定义叶子节点层深度为 0，自下而上递增。eh_entries 为索引项或者扩展项的数量，eh_magic 为扩展树幻数。

XCraft_extent_idx 为索引项结构，其中 ei_block 为逻辑块号，起中间索引的作用，ei_leaf 为指向的下一层磁盘块的物理块号。

XCraft_extent 为扩展项结构，其中 ee_block 是扩展的起始逻辑块号，ee_len 是扩展的连续物理块数量，ee_start 是起始物理块号。

XCraft_ext_path 记录的是从根节点向下遍历时每一层的相关信息，一般情况下 p_block 存放指向下一级的物理块号，p_depth 存放当前层的深度，p_maxdepth 存放最大深度，p_ext 和 p_idx 由当前节点中是索引项还是扩展项选择赋值，p_bh 存放当前节点的 buffer_head。其中如果当前层为根节点时，p_bh 为空；当前层为索引节点时，p_ext 为空；当前层为叶子节点时，p_idx 为空，如果 p_ext 不为空，则 p_block 记录的是 p_ext 中索引的起始物理块的物理块号。

4 开发计划

在我们确定了文件系统总体磁盘布局 and 核心数据结构后，我们制定了详细的设计开发计划。以下是主要的开发步骤：

1. 编写磁盘格式化工具，用于将磁盘格式化为我们的文件系统的格式；
2. 填充与 VFS 层对接的函数接口，使得我们的文件系统能够与 VFS 层交互；
3. 实现文件系统模块，用于挂载和卸载文件系统，以及初始化和退出文件系统；
4. 编写测试脚本对我们编写的文件系统的功能性进行测试。

如下，我们将详细介绍我们具体的开发计划思路 and 方案：

4.1 磁盘格式化工具的编写

我们的磁盘格式化工具在 `mkfs.c` 中实现，我们格式化磁盘的主要思路如下，顺次执行以下操作：

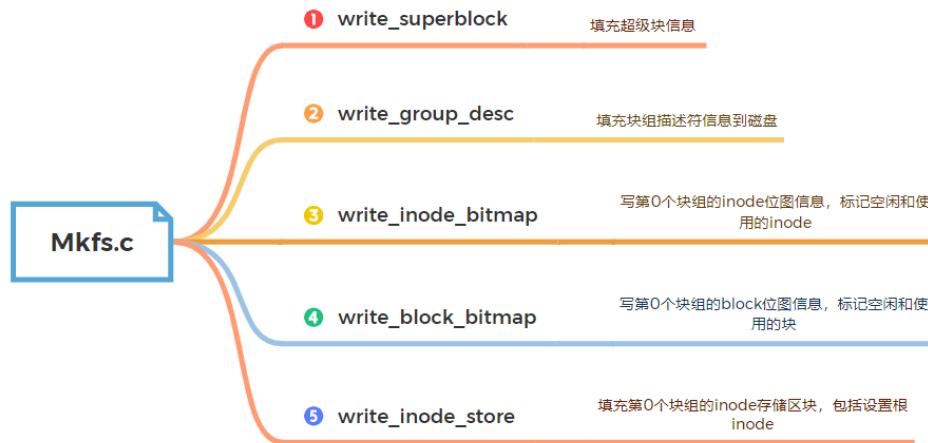


图 4.13: 格式化步骤

以上便是我们的 `mkfs.c` 的核心内容，接下来介绍我们格式化磁盘的核心思路：

我们的文件系统对于每一个普通块组包含 **8192** 个 inode，每一个 inode 索引 4 个磁盘块，于是我们认为每一个普通块组有 $8192 \times 4 = 32768$ 个磁盘块，即 **128MB**。我们在格式化磁盘时会检查磁盘文件大小，如果磁盘大小没有达到 **128MB**，我们会拒绝初始化，因为其大小不合我们的要求，我们要求至少一个普通块组大小才能初始化。

4.1.1 块组的划分

我们对于一个大小超过 128MB 的磁盘文件，首先我们会向下取整其得出其有多少个磁盘块。然后将其除以一个普通块组的磁盘块数并向上取整得出一共要分配多少个块组。注意，此时如果不是整除，那么最后一个块组的磁盘块个数会和前面所有块组的磁盘块个数不同。

此时我们会对最后一个块组的磁盘块数进行判断，如果其块数小于 1024，我们会将其并入前一个块组，此时总块组数会减 1。

具体的并入体现在我们会扩充其 inode 位图和 block 位图，使用更多的位来确定其使用情况。同时由于块组的并入，我们决定将 inode 所占的块数进行取整，不允许其有模数。具体操作如下：

```

1 // XCRAFT_INODE_RATIO 为 4，代表一个 inode 索引 4 个磁盘块
2 uint32_t last_inode_count=s_last_group_blocks/XCRAFT_INODE_RATIO;
3 s_inodes_count = (s_groups_count-1)*s_inodes_per_group + last_inode_count;
4 uint32_t mod=last_inode_count%XCRAFT_INODES_PER_BLOCK;
5 if(mod){
6     // 更新 last_inode_count
7     last_inode_count += XCRAFT_INODES_PER_BLOCK-mod;
  
```

```

8     s_inodes_count += XCRAFT_INODES_PER_BLOCK-mod;
9 }

```

如上，我们便可以使整个文件系统以及每个块组的 inode_store 所占的大小刚好是磁盘块大小的整数倍。

4.1.2 超级块信息写磁盘

结合上面块组的划分，我们便可以以此在内存中填充超级块的信息，然后通过写操作写入磁盘。

4.1.3 块组描述符写磁盘

我们上述通过块组的划分，得知了块组的数量。我们将块组描述符写磁盘时遵循的策略便是只对第 0 个块组进行信息的详细填充，其他块组的块组描述符信息先只填充其磁盘块个数和 inode 个数两个字段信息。**由于我们的块组数量可能较小，并不能撑满我们为块组描述符设置的区域。**于是一部分区域便成了保留区，此时我们便把保留区信息全部置 0。以此我们实现了按需初始化，还区分了初始化和未初始化块组以及保留区。

在我们对块组描述符信息进行填充的时候，一定要注意最后一个块组的情况。具体如下：

1. **第一个块组是最后一个块组。**这是一定要注意 block 位图大小，inode_store 大小，磁盘块数等的变化，其会影响 inode_store 的起始块号等，在对字段赋值时需要特别注意；
2. **第一个块组不是最后一个块组。**不过我们在对最后一个块组的块组描述符字段进行填充时仍然需要注意对磁盘块数和 inode 数的赋值。

4.1.4 初始化第 0 个块组的 inode 位图

经过我们上面的分析，我们的 inode 位图最多只需要一个块便可以覆盖所有 inode 的使用情况。对于第 0 个块组，我们将第 1 个 inode 作为根 inode，第 0 个 inode 我们不使用。于是我们在对位图置位时只需将其他位置 1，第 0 个 inode 和第 1 个 inode 所对应的位置 0 即可。

具体操作如下：

```

1 static int XCraft_write_inode_bitmap(int fd, struct superbblock_padding *sb){
2     char *inode_bitmap = malloc(XCRAFT_BLOCK_SIZE);
3     if(!inode_bitmap){
4         perror("malloc inode_bitmap failed");

```

```

5         return -1;
6     }
7     uint64_t *ifree=(uint64_t *)inode_bitmap;
8     memset(ifree, 0xff, XCRAFT_BLOCK_SIZE);
9     //根 inode 索引一个数据块, 同时 avoid using inode 0
10    //第一个块组的 inode_bitmap 只占据一个块
11    ifree[0] = htobe64(0xffffffffffffffffc);
12    int ret=write(fd, ifree, XCRAFT_BLOCK_SIZE);
13    if(ret!=XCRAFT_BLOCK_SIZE){
14        perror("write inode_bitmap failed");
15        free(inode_bitmap);
16        return -1;
17    }
18    free(inode_bitmap);
19    return 0;
20 }

```

我们填充 inode_bitmap 原理是通过一个 64 位数的数组来实现的, 对于每一个 8 字节, 其二进制位从低位到高位分别代表第 0 到第 63 个比特。比如我们如果要将第 0 个和第 1 个 inode 置为已使用, 那么我们便进行计算:

$$1/64 = 0 \quad 0/64 = 0$$

$$1\%64 = 1 \quad 0\%64 = 0$$

于是我们以第 1 个 inode 置为已使用为例, 需要将 64 位数的第 0 个字节中的二进制从低位到高位第 1 位置 0。

综上, 我们最后写入时 ifree[0] 置为了 htobe64(0xffffffffffffc), 标志着第 0 个 inode 和第 1 个 inode 已经被使用。

4.1.5 初始化第 0 个块组的 block 位图

经过上面块组划分处的分析, 在这里写第 0 个块组的 block 位图时同样需要注意第 0 个块组是最后一个块组的情况。

我们在这里需要将已经使用的块对应的位图 bit 置 0。即超级块, 块组描述符, inode 位图, block 位图, inode_store 和根目录 inode 初始化时为 i_block[0] 分配的磁盘块。其中为根目录 inode 的 i_block[0] 分配的磁盘块为第 0 个块组的第一个可用数据块。

我们采用了以下方式置位:

```

1 static int XCraft_write_block_bitmap(int fd, struct superbblock_padding *sb){
2     uint32_t bfree_blo;
3     if(sb->xcraft_sb.s_groups_count==1){

```

```

4         bfree_blo=XCRAFT_BFREE_PER_GROUP_BLOCK(1e32toh(sb->xcraft_sb.s_last_group_blocks));
5     }
6     else bfree_blo=1;
7     char *block_bitmap = malloc(XCRAFT_BLOCK_SIZE*bfree_blo);
8     if(!block_bitmap){
9         perror("malloc block_bitmap failed");
10        return -1;
11    }
12    ...
13    while(nr_used){
14        uint64_t line = 0xffffffffffffffff;
15        //从低位开始清 0 直至清除 nr_used 个位
16        for (uint64_t mask = 0x1; mask; mask <<= 1) {
17            line &= ~mask;
18            nr_used--;
19            if (!nr_used)
20                break;
21        }
22        bfree[i] = htobe64(line);
23        i++;
24    }
25    int ret=write(fd, bfree, XCRAFT_BLOCK_SIZE*bfree_blo);
26    if(ret!=XCRAFT_BLOCK_SIZE*bfree_blo){
27        perror("write block_bitmap failed");
28        free(block_bitmap);
29        return -1;
30    }
31    free(block_bitmap);
32 }

```

由于已经使用的块都是顺序连续的，所以我们用 `nr_used` 确定需要置位的块的个数。每次 8 字节一遍历，在遍历的同时进行置位，当置位完成时便退出循环。

4.1.6 初始化第 0 个块组的 `inode_store`

在初始化第 0 个块组的 `inode_store` 时，我们重点是初始化第 1 个 `inode`，因为我们将其作为根 `inode`。初始化的时候重点是为其分配一个磁盘块，并将磁盘块号填充在 `i_block[0]` 中，分配的这个磁盘块是第 0 个块组的首个空闲 `block`。其余的 `inode` 我们均通过置 0 处理。

4.2 填充与 VFS 层对接的函数接口

本文件系统中我们重点填充的与 VFS 层对接的函数接口是 `super_operations`, `inode_operations`, `file_operations`。

4.2.1 `super_operations`

我们填充的 `super_operations` 的内容应该如下：

```

1  struct super_operations XCraft_sops =
2  {
3      .alloc_inode = XCraft_alloc_inode,
4      .destroy_inode = XCraft_destroy_inode,
5      .write_inode = XCraft_write_inode,
6      .put_super = XCraft_put_super,
7      .sync_fs = XCraft_sync_fs,
8      .statfs = XCraft_statfs,
9  };

```

其中各个函数应该实现的功能如下：

1. `alloc_inode`: 在内存中成功分配一个 `inode`;
2. `destroy_inode`: 在内存中成功释放一个 `inode`;
3. `write_inode`: 将内存中的 `inode` 写回磁盘;
4. `put_super`: 释放超级块;
5. `sync_fc`: 同步文件系统信息到磁盘, 在这里我们主要是将超级块, 块组描述符和位图等信息写回磁盘;
6. `statfs`: 获取文件系统的信息。

4.2.2 `inode_operations`

我们的 `inode_operations` 填充的内容如下, 其中我们以 `inode` 索引文件类型是否是符号链接文件进行区分:

```

1  // 目录或者文件
2  static const struct inode_operations XCraft_inode_operations = {
3      .create = XCraft_create,

```

```

4     .lookup = XCraft_lookup,
5     .link = XCraft_link,
6     .unlink = XCraft_unlink,
7     .symlink = XCraft_symlink,
8     .mkdir = XCraft_mkdir,
9     .rmdir = XCraft_rmdir,
10    .rename = XCraft_rename,
11 };
12 // 符号链接
13 static const struct inode_operations XCraft_symlink_inode_operations = {
14     .get_link = XCraft_get_link,
15 };

```

在编写上述函数的重点其实是要实现块和 inode 的分配，哈希树的构建以及插入删除。

块和 inode 的分配

当我们要分配一个空闲块和 inode 时，我们的思路是获取目前已初始化块组的块组描述符，依据里面记录的信息查看其对应块组中是否存在空闲块和空闲 inode。如果存在就利用其位图定位我们要分配的块和 inode；不存在的话我们就需要重新初始化新块组，从新块组中获取块和 inode。

哈希树的构建以及插入删除

我们的目录最开始并没有构建哈希树，当我们在其下插入目录项时也只是在 `i_block[0]` 中索引的数据块中顺次插入目录项。当我们插入的目录项达到了其指定分裂构建哈希树的目录项数目时，此时我们便会建立哈希树。

1. 哈希树的构建

首先我们会追加一个块，记为块 1，将 `i_block[0]` 中的目录项全部拷贝到此块 1 中。此时 `i_block[0]` 作为 `dx_root`，对 `dx_root` 中的信息进行初始化，并填充 `dx_entry[0]` 的相关信息。

然后再追加一个块，记为块 2，此时我们从块 2 底部顺次构建对应块 1 中目录项的 `dx_map_entry`，每一个 `dx_map_entry` 中存储了对应目录项中文件名的哈希值，相对于块 1 开头的偏移量以及目录项的大小。

然后我们依据哈希值对 `dx_map_entry` 进行排序并确定分裂位置，这时我们将分裂位置以下的 `dx_map_entry` 中对应的目录项从块 1 拷贝到块 2 开头，并且移除块 1 中拷贝到块 2 中的目录项。最后在 `dx_root` 添加 `dx_entry`，并将其指向的块号赋值为块 2 的物理块号。

2. 哈希树中插入目录项

首先我们会计算新插入的目录项的文件名的哈希值，然后在哈希树的每一级进行二分查找，并用 `dx_frame` 结构记录每一级的相关信息。如果此时对应的磁盘块中已经无法插入新的目录项，那么需要分裂；否则直接插入。

如果需要分裂，那么意味着需要在前一级中插入 `dx_entry`。此时需要使用我们的 `dx_frame` 数组进行判断，我们需要回溯到 `dx_entry` 的数目没有达到限制的对应级。如果没有对应级，那么我们需要添加级数。

当添加级数时，我们会追加一个新块，记为块 1，将 `i_block[0]` 中 `dx_entry` 的信息拷贝到块 1 中，并对块 1 构建 `dx_node`。然后我们对 `i_block[0]` 进行重构 `dx_root` 的工作。

如果不需要添加级数，那么我们回溯的 `dx_entry` 的数目没有达到限制的对应级的下一级需要分裂，此时我们便执行 `dx_node` 的分裂工作，并在此级插入 `dx_entry`。

我们不断重复上述过程，直到最后我们可以插入目录项为止。

3. 哈希树中删除目录项

首先我们会计算要删除目录项的文件名的哈希值，然后同样进行二分查找，并用 `dx_frame` 结构记录每一级的相关信息。我们通过 `dx_frame` 的最后一级获取删除目录项所在的磁盘块的块号。

然后我们遍历此磁盘块，找到目录项所在位置，将其之后的目录项内容全部前移一个目录项位置，以此来保证目录项连续性，并且将最开始的最后一个目录项的位置处置 0。以此我们最大程度地节省了空间，也提高了删除效率。

4.2.3 file_operations

我们的 `file_operations` 填充的内容如下，我们以文件类型是普通文件和目录进行区分：

其中普通文件的对应如下：

```

1 // address_space_operations
2 const struct address_space_operations XCraft_aops = {
3     #if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 19, 0)
4         .readahead = XCraft_readahead,
5     #else
6         .readpage = XCraft_readpage,
7     #endif
8         .writepage = XCraft_writepage,
```



```

9     .write_begin = XCraft_write_begin,
10    .write_end = XCraft_write_end,
11 };
12 const struct file_operations XCraft_file_operations = {
13     .llseek = XCraft_llseek,
14     .owner = THIS_MODULE,
15     .read_iter = XCraft_file_read_iter,
16     .write_iter = XCraft_file_write_iter,
17     .fsync = XCraft_sync_file,
18 };

```

如上，我们对普通文件的读写采用了 **page cache** 机制，所以我们填充了上述内容，并且在我们实现的函数中大多调用 linux 自带的通用 **page cache** 读写函数进行实现，简单高效。实际自己需要编写的便是 **XCraft_file_get_block** 与 **XCraft_write_end** 这两个函数：

1. **XCraft_file_get_block**: 其中 **iblock** 是某个块在 inode 数据块中的逻辑块号，这个函数需要通过 inode 和逻辑块号来建立逻辑块号和物理块号的映射，并将得到的结果读取存入到 **bh_result** 中。**create** 表示如果这个索引不存在，是否需要重新分配一个物理块；
2. **XCraft_write_end**: 这个函数主要是写操作的善后工作，因为我们每一次写操作可能会导致文件的大小变小，这时需要我们释放掉多余的块，将文件大小减少到 **inode->i_size** 表示的大小。

其中目录文件的对应如下：

```

1 const struct file_operations XCraft_dir_operations = {
2     .iterate_shared = XCraft_readdir,
3 };

```

我们针对目录文件的文件操作函数主要填充的是有关目录下遍历目录项的函数，它具体与 **ls** 命令的调用有关。

其中 **XCraft_readdir** 函数中最重要的便是遍历目录下的目录项。由于我们哈希树的设置，便需要分两种情况遍历：

- 哈希树此时没有建立，此时直接遍历 **i_block[0]** 指向的磁盘块，获取目录项；
- 哈希树此时已经建立，需要按照 **dfs** 的思想来遍历哈希树，获取目录项。

4.3 实现文件系统模块

在设计了数据结构和接口之后，我们开始实现文件系统模块。其中我们定义的文件系统如下所示：

```

1 //file_system_type define
2 static struct file_system_type XCraft_fs_type = {
3     .owner = THIS_MODULE,
4     .name = "XCraft",
5     .mount = XCraft_mount,
6     .kill_sb = XCraft_kill_sb,
7     .fs_flags = FS_REQUIRES_DEV,
8     .next = NULL,
9 };

```

其中最重要的是两个成员 `.mount` 和 `.kill_sb`，前者挂载文件系统分区，后者在卸载文件系统分区时被调用。

如下我们介绍这两个函数的实现：

4.3.1 文件系统卸载

文件系统卸载与我们编写的 `XCraft_kill_sb` 函数有关。

卸载流程我们直接调用 Linux 的 `vfs` 层中的 `kill_block_super` 函数，因为我们的文件系统是一个块设备文件系统。在此函数中我们最终会调用我们在 `super_operations` 中实现的 `put_super` 函数，来对内存中的超级块进行清除，同时还会进行对内存 `inode` 等进行释放的操作。

4.3.2 文件系统挂载

文件系统挂载与我们编写的 `XCraft_mount` 函数有关。

我们的文件系统是一个块设备文件系统，因此我们在这里选择 `mount_bdev` 函数进行挂载。此函数接受一个函数指针，需要我们自己实现，其功能为将超级块、根目录 `inode` 加载到内存。

`mount_bdev` 主要进行参数检查、获取块设备和文件系统信息、创建挂载点，并将块设备挂载到文件系统上，然后会调用我们编写的 `XCraft_fill_super` 函数来将超级块、根目录 `inode` 加载到内存，完成文件系统的初步加载工作。

上述我们编写的函数可以概括为以下几点：

- 读取磁盘超级块，加载到内存建立内存超级块结构，并与 `vfs` 层建立联系；

- 读取块组描述符和位图等信息，填充内存超级快结构；
- 获取磁盘根目录 inode, 加载到内存使用 `sb->s_root=d__make__root(root__inode)` 来建立根目录 `dentry`，同时与 `super block` 联系。

4.3.3 模块初始化和卸载

我们在 `fs.c` 实现此模块，主要的函数包括：

- `XCraft_init`：模块初始化函数，在里面我们初始化文件系统，包括创建 inode 缓存和注册文件系统

`XCraft_exit`：模块卸载函数，在里面我们退出文件系统，包括挂载和卸载文件系统，以及初始化和退出文件系统。

4.4 编写测试脚本进行测试

我们的测试脚本会对文件系统的挂载、卸载，目录及文件的创建与删除，文件读写，拷贝，剪切等功能都进行测试。其中重点会对哈希树的功能进行分析测试，观察其的建树，分裂与删除等过程。

5 比赛过程中的重要进展

1. 三月

新建学习仓库，将每次的学习成果上传至仓库；学习 Linux 模块编程；阅读《Linux 内核设计与实现》这本书的虚拟文件系统部分，弄懂文件系统 `vfs` 层结构以及如何对接；

2. 四月

学习优秀文件系统 `fat`、`simplefs`、`ext3`、`ext4` 的设计思路和源代码，讨论形成我们的设计思路；初步编写文件系统框架，写好函数声明等

3. 五月

集中编写我们的文件系统代码；一周多时间对文件系统代码以及功能实现进行 `debug`，改进并优化代码，解决 `bug`；编写开发报告，录制项目演示视频

6 系统测试情况

6.1 测试概况及脚本编写

我们对于文件系统的测试集中在如下部分：

- 文件系统的挂载和卸载
- 格式化工具
- 目录及文件的创建与删除 (哈希树要平稳运行, 并要有分裂情况)
- 文件的读写功能
- 剪切拷贝功能

我们编写了如下的测试脚本 `setup.sh` 进行测试:

```
1  #!/bin/bash
2
3  # 编译 kernel module 和 tool
4  echo " 开始编译 kernel module 和 tool..."
5  make
6  echo " 编译完成! "
7  echo -e "\n"
8  sleep 1
9
10 # 加载 kernel module
11 echo " 开始加载 kernel module..."
12 sudo insmod xcraft.ko
13 echo "kernel module 加载完成! "
14 echo -e "\n"
15 sleep 1
16
17 # 创建测试目录和测试镜像
18 echo " 开始创建测试目录和测试镜像..."
19 sudo mkdir -p /mnt/test
20 dd if=/dev/zero of=test.img bs=1M count=256
21 echo " 测试目录: /mnt/test"
22 echo " 镜像名: test.img"
23 echo " 镜像大小: 256MB"
24 echo -e "\n"
25 sleep 1
26
27 # 使用 mkfs.XCraft 工具创建文件系统
28 echo " 开始使用 mkfs.XCraft 工具创建文件系统..."
29 ./mkfs.XCraft test.img
```

```
30 echo " 文件系统 test.img 创建完成! "
31 echo -e "\n"
32 sleep 1
33
34 # 挂载测试镜像
35 echo " 开始挂载测试镜像..."
36 sudo mount -o loop -t XCraft test.img /mnt/test
37 echo " 测试镜像挂载到/mnt/test! "
38 echo -e "\n"
39 sleep 1
40
41 # 显示文件系统情况
42 echo " 显示文件系统情况..."
43 df -TH
44 echo -e "\n"
45 sleep 1
46 # 检查挂载成功的 kernel 消息
47 #dmesg | tail
48
49 # 执行一些文件系统操作
50 echo "xxxxxxxxxxxxxxxxxxxxxxxx 开始执行文件系统操作 xxxxxxxxxxxxxxxxxxxxxxxx"
51 sudo su <<EOF
52 cd /mnt/test
53 #... 相关测试代码这里省略
54
55 # 卸载测试镜像
56 # 卸载 kernel mount point 和 module
57 echo " 开始卸载 kernel mount point 和 module..."
58 sudo umount /mnt/test
59 sudo rmmod xcraft
60 echo "kernel mount point 和 module 卸载完成! "
61 echo -e "\n"
62 sleep 1
63 echo " 开始清理构建环境..."
64 make clean
65 echo " 构建环境清理完成! "
66
67
```

我们上述脚本的具体测试意义如下：

- 编译文件系统代码
- 加载注册模块，建立挂载点目录
- 生成磁盘镜像文件，用格式化工具对其进行格式化
- 挂载至挂载点目录，并进入挂载点目录下
- 进行文件系统操作
- 卸载文件系统，清理模块和环境

6.2 make test 测试

根据如下命令，我们进行测试

```
1 sudo bash 4test.sh
2 sudo make test
```

开始挂载测试镜像...
测试镜像挂载到/mnt/test!

显示文件系统情况...

文件系统	类型	大小	已用	可用	已用%	挂载点
tmpfs	tmpfs	406M	2.1M	404M	1%	/run
/dev/sda3	ext4	31G	29G	432M	99%	/
tmpfs	tmpfs	2.1G	0	2.1G	0%	/dev/shm
tmpfs	tmpfs	5.3M	4.1k	5.3M	1%	/run/lock
tmpfs	tmpfs	2.1G	0	2.1G	0%	/run/qemu
/dev/sda2	vfat	537M	6.4M	531M	2%	/boot/efi
tmpfs	tmpfs	406M	103k	406M	1%	/run/user/1000
/dev/loop21	XCraft	269M	1.1M	268M	1%	/mnt/test

xxxxxxxxxxxxxxxxxxxxxxxx开始执行文件系统操作xxxxxxxxxxxxxxxxxxxxxxxx

○ prayer@prayer-virtual-machine:~/os/project2210132-232458\$ █

图 6.14: sudo bash 4test.sh 产生 XCraft 文件系统

下面是我们 make test 的测试性能情况，目前设计了两个 test.c，运行如下，主要是对挂载目录下/mnt/test 的读写情况测试

```

prayer@prayer-virtual-machine:~/os/project2210132-232458$ sudo make test
[sudo] prayer 的密码:
#####
file content: Hello World!
file deleted
file write performance: 0.006753
#####
#####
alread write 100 files
alread delete 100 files
file write performance: 0.016024
#####
prayer@prayer-virtual-machine:~/os/project2210132-232458$

```

图 6.15: make test 测试结果

6.3 具体测试情况

可以看到，我们对我文件系统代码的编译成功：

```

LD [M] /home/prayer/os/project2210132-232458/xcraft.o
MODPOST /home/prayer/os/project2210132-232458/Module.symvers
CC [M] /home/prayer/os/project2210132-232458/xcraft.mod.o
LD [M] /home/prayer/os/project2210132-232458/xcraft.ko
BTF [M] /home/prayer/os/project2210132-232458/xcraft.ko
make[1]: 离开目录“/usr/src/linux-headers-6.2.0-32-generic”
mv *.symvers build
mv *.ko build
mv *.mod.c build
mv *.mod.o build
mv *.o build
mv *.order build
mv *.mod build
mv .Module.symvers.cmd build
mv .modules.order.cmd build
mv *.ko.cmd build
mv *.mod.cmd build
mv *.mod.o.cmd build

```

图 6.16: 编译成功显示

如图6.17，可以看到我们的模块加载成功，同时创建了磁盘镜像文件：

```

开始加载 kernel module...
kernel module 加载完成!

开始创建测试目录和测试镜像...
记录了256+0 的读入
记录了256+0 的写出
268435456字节 (268 MB, 256 MiB) 已复制, 0.239934 s, 1.1 GB/s
测试目录: /mnt/test
镜像名: test.img
镜像大小: 256MB

```

图 6.17: 模块加载成功

我们对文件系统的挂载成功，并且成功显示了挂载信息：

```

开始挂载测试镜像...
测试镜像挂载到/mnt/test!

显示文件系统情况...
文件系统      类型    大小  已用  可用  已用% 挂载点
tmpfs          tmpfs   406M  2.2M  404M    1% /run
/dev/sda3      ext4    31G   26G   4.3G   86% /
tmpfs          tmpfs   2.1G    0   2.1G    0% /dev/shm
tmpfs          tmpfs   5.3M   4.1k  5.3M    1% /run/lock
tmpfs          tmpfs   2.1G    0   2.1G    0% /run/qemu
/dev/sda2      vfat    537M   6.4M  531M    2% /boot/efi
tmpfs          tmpfs   406M  103k  406M    1% /run/user/1000
/dev/loop17    XCraft  269M   1.1M  268M    1% /mnt/test

```

图 6.18: 文件挂载信息

如下是我们对文件系统的具体操作：

```

xxxxxxxxxxxxxxxxxxxxxxxx开始执行文件系统操作xxxxxxxxxxxxxxxxxxxxxxxx
it's a test! it's a test! it's a test! it's a test
1.txt dir13 dir18 dir22 dir27 dir31 dir36 dir40 dir45 dir5 dir54 dir59 dir63 dir68 dir72 dir77 dir81 dir86 dir90
dir1 dir14 dir19 dir23 dir28 dir32 dir37 dir41 dir46 dir50 dir55 dir6 dir64 dir69 dir73 dir78 dir82 dir87 dir91
dir10 dir15 dir2 dir24 dir29 dir33 dir38 dir42 dir47 dir51 dir56 dir60 dir65 dir7 dir74 dir79 dir83 dir88 dir92
dir11 dir16 dir20 dir25 dir3 dir34 dir39 dir43 dir48 dir52 dir57 dir61 dir66 dir70 dir75 dir8 dir84 dir89
dir12 dir17 dir21 dir26 dir30 dir35 dir4 dir44 dir49 dir53 dir58 dir62 dir67 dir71 dir76 dir80 dir85 dir9
1.txt dir88 dir89 dir90 dir91 dir92
1.txt
1.txt
开始卸载 kernel mount point 和 module...
kernel mount point 和 module 卸载完成!

```

图 6.19: 文件系统具体测试操作

结合上述我们的脚本命令，可以看到我们的操作得到的结果完全正确。然后我们结合内核调试信息，可以观察到哈希树的建立以及级数的添加。

如下图6.20、图6.21可以看到：当我们建立 **dir16** 时此时开始生成哈希树，并且生成成功：

```
[47906.264415] filename: dir16
[47906.264418] begin mkdir
[47906.264586] XCraft_add_entry i_block: 262
[47906.264617] count:15
[47906.264618] add_dirent_to_buf retval: -28
[47906.264618] Creating index: inode 2
[47906.264714] 分配的块号279
[47906.264715] hello
[47906.264716] set dx_root success
[47906.264716] hinfo赋值成功
[47906.264716] XCraft_make_hash_tree 计算hash值成功1
```

图 6.20: dir16 建立目录项

```
[ 575.393510] dx_sort_map process correctly
[ 575.393510] dx_sort_map success!
[ 575.393510] map[0].hash=c7246bde
[ 575.393511] map[1].hash=c76d745e
[ 575.393512] map[2].hash=c796be3c
[ 575.393512] map[3].hash=c7da31b8
[ 575.393513] map[4].hash=c8431a38
[ 575.393513] map[5].hash=c878dab8
[ 575.393514] map[6].hash=c95274ce
[ 575.393515] map[7].hash=c96aff72
[ 575.393515] map[8].hash=c9bea922
[ 575.393516] map[9].hash=ca08c5ec
[ 575.393516] map[10].hash=cadfc34
[ 575.393517] map[11].hash=cb25f07a
[ 575.393517] map[12].hash=cb9593a6
[ 575.393530] map[13].hash=cc3539a0
[ 575.393531] map[14].hash=cc4bb6e4
[ 575.393531] Split block288 at c96aff72, 7/8
...
```

图 6.21: 建立哈希树的 hash 值排序过程

如图6.22可以看到：当我们建立 **dir34** 时此时哈希树添加了级数，即现在有一级 **dx_node**：

```

[66004.989540] filename: dir34
[66004.989550] begin dx_Find_entry!
[66004.989551] in dx_find_entry ,dx_probe success!
[66004.989559] begin mkdir
[66004.992621] count:0
[66004.992652] 需要添加级数
[66004.992653] 现在的级数为: 1
[66004.992654] count:0
[66004.992662] 不需要添加级数
[66004.992663] count:0
[66004.992670] dx_make_map process correctly
[66004.992671] in do_split , count :15
[66004.992672] dx_sort_map process correctly
[66004.992672] dx_sort_map success!
[66004.992672] map[0].hash=c7246bde
[66004.992673] map[1].hash=c76d745e
[66004.992674] map[2].hash=c796be3c
[66004.992674] map[3].hash=c7da31b8
[66004.992675] map[4].hash=c8431a38
[66004.992675] map[5].hash=c878dab8

```

图 6.22: 内核输出添加级数信息

如图6.23可以看到:当我们建立 dir69 时,哈希树添加了级数,即现在有两级 `dx_node`:

```

[66005.093375] filename: dir69
[66005.093382] begin dx_Find_entry!
[66005.093384] in dx_find_entry ,dx_probe success!
[66005.093389] begin mkdir
[66005.093518] count:0
[66005.093520] in add_dirent_to_buf, success
[66005.093521] dir_inode->i_nlink: 71
[66005.093522] dir_inode_info->i_nr_files: 69
[66005.093523] 添加的目录项的i_nlink: 2
[66005.093523] 添加的目录项的i_nr_files: 0
[66005.095648] filename: dir70
[66005.095660] begin dx_Find_entry!
[66005.095662] in dx_find_entry ,dx_probe success!
[66005.095669] begin mkdir
[66005.095934] count:0
[66005.096013] 需要添加级数
[66005.096015] 现在的级数为: 2

```

图 6.23: 添加级数内核信息

综上,我们看到了哈希树的建立过程以及级数的添加,验证了我们的哈希树构建成功。最后,在图6.24中,可以看到:我们卸载文件系统和注销模块成功,并且成功清理了构建环境:

```

开始清理构建环境...
make -C /lib/modules/6.2.0-32-generic/build M=/home/prayer/os/project2210132-232458 clean
make[1]: 进入目录"/usr/src/linux-headers-6.2.0-32-generic"
make[1]: 离开目录"/usr/src/linux-headers-6.2.0-32-generic"
rm -f *~ /home/prayer/os/project2210132-232458/*.ur-safe
rm -f build/mkfs.XCraft test.img
rm -rf build
构建环境清理完成!
> prayer@prayer-virtual-machine:~/os/project2210132-232458$

```

图 6.24: 卸载信息

综上，我们利用编写的脚本对文件系统实现了测试，并且测试成功。

7 项目测试结果功能与创新性分析

7.1 项目结构

- 采用 include,src,test 的结构，搭配 makefile,4test.sh，可以创建文件系统后，直接 make test 执行 .c 文件，测试文件系统。易用性与易测试性是我们项目的一大优点。

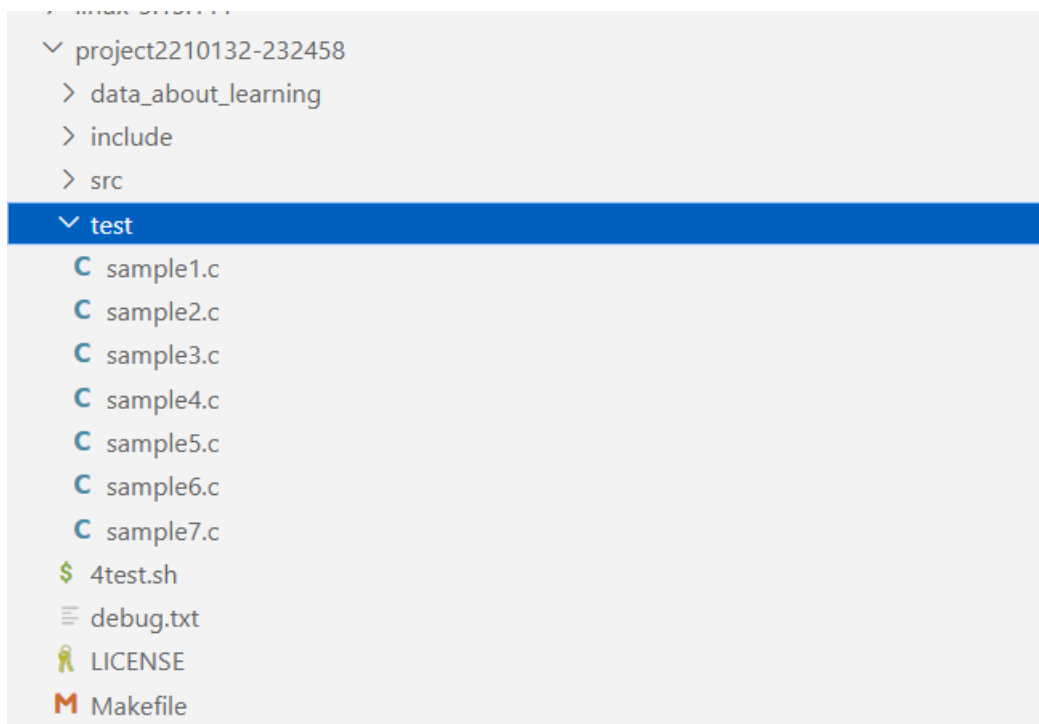


图 7.25: 项目结构图

- 简单来说，我们可以经过如下步骤测试文件系统
 - 首先执行 `bash 4test.sh` 搭建文件系统，我们文件系统默认挂载于 `/mnt/test/` 下，`df-h,df-i`，可以查看文件系统情况。

```

##### 7.26 #####
prayer@prayer-virtual-machine:~/os/project2210132-232458$ df -h
文件系统      大小  已用  可用  已用% 挂载点
tmpfs          388M  2.0M  386M   1% /run
/dev/sda3      29G   25G  2.7G  91% /
tmpfs          1.9G   0  1.9G   0% /dev/shm
tmpfs          5.0M  4.0K  5.0M   1% /run/lock
tmpfs          1.9G   0  1.9G   0% /run/qemu
/dev/sda2      512M  6.1M  506M   2% /boot/efi
tmpfs          388M  96K  388M   1% /run/user/1000
/dev/loop20    1.0G  9.9M 1015M   1% /mnt/test
prayer@prayer-virtual-machine:~/os/project2210132-232458$

```

图 7.26: df -h 显示 XCraft 文件系统使用情况

```

prayer@prayer-virtual-machine:~/os/project2210132-232458$ df -i
文件系统      Inodes  已用 I  可用 I  已用 I% 挂载点
tmpfs          495563  1111  494452   1% /run
/dev/sda3      1933312 382303 1551009  20% /
tmpfs          495563   1  495562   1% /dev/shm
tmpfs          495563   4  495559   1% /run/lock
tmpfs          495563   1  495562   1% /run/qemu
/dev/sda2       0   0   0  - /boot/efi
tmpfs          99112  145  98967   1% /run/user/1000
/dev/loop20    65536   2  65534   1% /mnt/test
prayer@prayer-virtual-machine:~/os/project2210132-232458$

```

图 7.27: df -i 显示 XCraft inodes 使用情况

```

prayer@prayer-virtual-machine:~/os/project2210132-232458$ df -T
文件系统      类型      1k的块  已用  可用  已用% 挂载点
tmpfs          tmpfs      396452  2004  394448   1% /run
/dev/sda3      ext4      30267332 27255824 1448680  95% /
tmpfs          tmpfs      1982252   0  1982252   0% /dev/shm
tmpfs          tmpfs       5120   4   5116   1% /run/lock
tmpfs          tmpfs      1982252   0  1982252   0% /run/qemu
/dev/sda2      vfat       524252  6216  518036   2% /boot/efi
tmpfs          tmpfs      396448   96  396352   1% /run/user/1000
/dev/loop20    XCraft    1048576  6912 1041664   1% /mnt/test
prayer@prayer-virtual-machine:~/os/project2210132-232458$

```

图 7.28: df -T 显示 XCraft 文件系统的块

— 然后执行 make test, 或者 make test k, 选中 test 文件夹下的某一个文件测试 (均在挂载文件系统的目录下, 执行读写), 得出文件系统的性能

```

r10: we are praye4u.Hello World! we are praye4u.Hello World! we are praye4u.Hello World! we are praye4u
.Hello World! we are praye4u.Hello World! we are praye4u.Hello World! we are praye4u.Hello World! we ar
e praye4u.Hello World! we are praye4u.Hello World! we are praye4u.Hello World! we are praye4u.Hello Wor
ld! we are praye4u.Hello World! we are praye4u.Hello World! we are praye4u.Hello World! we are praye4u.
Hello World! we are praye4u.Hello World! we are praye4u.Hello World! we are praye4u.Hello World! we are
praye4u.Hello World! we are praye4u.Hello World! we are praye4u.Hello World! we are praye4u.Hello Worl
d! we are praye4u.Hello World! we are praye4u.
已读取所有文件
已删除所有文件和目录
目录哈希测试耗时: 1.131228 秒
prayer@prayer-virtual-machine:~/os/project2210132-232458$

```

图 7.29: sample5.c 测试 1 万个目录及文件创建删除

- 采取 `pr_debug` 进行调试，为了性能可以关闭所有调试语句。

```
1 启用动态调试，显示打印调试语句：
2 echo 'module xcrafft +p' | sudo tee
   ↪ /sys/kernel/debug/dynamic_debug/control
3
4 不启用动态调试，不显示打印调试语句：
5 echo 'module xcrafft -p' | sudo tee
   ↪ /sys/kernel/debug/dynamic_debug/control
```

- `test` 文件夹下设计 7 个文件，分别针对我们需要测试的文件系统特性进行设计样例。
 - `sample2.c` 创建 10000 个子目录，同时读写文件，删除目录，测试块组以及目录哈希的功能。
 - `sample5.c` 创建 10000 个子目录，重点测试读取功能，即数据密集型的读写。
 - `sample6.c` 测试不同用户的权限管理，即禁止一个用户删除其他用户文件/目录；进制一个用户重命名其他用户的文件/目录；允许所有用户在挂载的 '/' 下创建文件/目录；每个用户只能在自己的文件夹下创建子文件夹/文件。内核会显示输出权限管理的相关提示。。
 - 其他 `sample` 全面测试文件系统功能。
 - 文件系统可能还有一点 `bug`，之后会尽力完善。

7.2 创新性分析

为了应对数据密集型场景，以及保证不同用户的权限管理，我们设计了如下创新性能

- **延迟初始化，使用块组：**

我们的文件系统采用了延迟初始化策略，只在 `mkfs` 阶段初始化第一个块组。只有当第一个块组使用完毕后才初始化其他块组，这种方法在管理大容量存储时表现出色，显著提升了文件系统的性能。

- **循环块组的分配方式：**

每个块组内部独立拥有 `inode_bitmap` 和 `block_bitmap`，相较于传统文件系统，这种分配方式使管理效率更高。在循环块组的分配模式下，文件系统能够更高效地进行资源分配和管理。

- **性能接近 Linux 自带文件系统：**

针对数据密集型应用场景，我们引入了目录哈希方法。在 `Inode` 下，当目录项数量超过一定限制时，立即构建目录的哈希 B+ 树，避免传统文件系统串行遍历查找目录项的低效问题，从而大幅提升查找目录的性能。

- **Inode 索引大文件：**

我们借鉴了 ext4 的扩展树方式，使用 B+ 树来提升大文件的索引效率。这一创新避免了大文件带来的索引效率下降问题，提高了大文件查找等操作的性能。

这里主要涉及 entent.h 以及 file.c 两个文件中。

- **自带权限管理：**

在 Linux 基础权限管理的基础上，我们设计了更严格的权限管理策略。具体包括：禁止一个用户删除或重命名其他用户的文件/目录；允许所有用户在挂载的 '/' 下创建文件/目录；每个用户只能在自己的文件夹下创建子文件夹/文件。内核会显示输出权限管理的相关提示信息，从而提升了文件系统的安全性和管理效率。

这主要是依据 inode 里面的 i_uid 与 i_gid 进行编写代码。

```

1      int XCraft_permission(struct inode *inode, int mask)
2  {
3      kuid_t fsuid = current_fsuid();
4      kgid_t fsgid = current_fsgid();
5      umode_t mode = inode->i_mode;
6      pr_debug("uid: %d %d\n", fsuid.val, inode->i_uid.val);
7
8      if (uid_eq(fsuid, GLOBAL_ROOT_UID)) {
9          return 0;
10     }
11     // 检查写权限
12     if (mask & MAY_WRITE) {
13         // 只有文件所有者/root 才可以写
14         if (uid_eq(fsuid, inode->i_uid)) {
15             return 0;
16         }
17         else{
18             return -EACCES; // 其他情况都不允许写
19         }
20     }
21
22     // 检查读权限和执行权限
23     if (mask & (MAY_READ | MAY_EXEC)) {
24         if (uid_eq(fsuid, inode->i_uid)) {
25             // 检查所有者权限
26             return (mode & mask) == mask ? 0 : -EACCES;

```

```

27     }
28     if (gid_eq(fsgid, inode->i_gid)) {
29         // 检查组权限 >>3 是因为组权限在 mode 中是从第 4 位开始的
30         return (mode & (mask >> 3)) == (mask >> 3) ? 0 : -EACCES;
31     }
32     // 检查其他用户权限 >>6 是因为其他用户权限在 mode 中是从第 7 位开始的
33     return (mode & (mask >> 6)) == (mask >> 6) ? 0 : -EACCES;
34 }
35
36 return 0; // 其他情况都允许
37 }

```

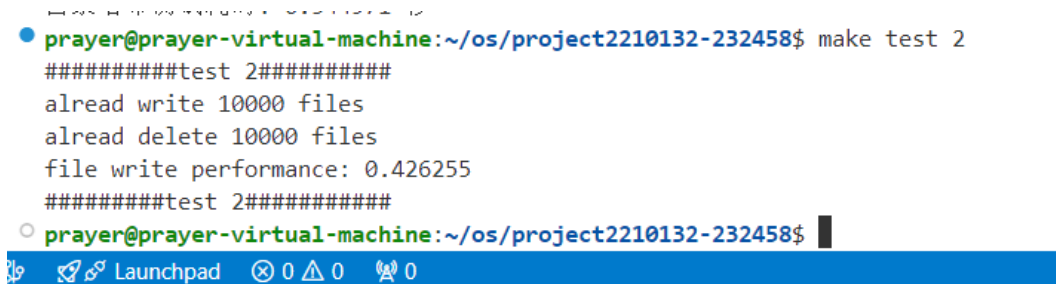
7.3 结果与性能对比

我们针对 5 个 sample.c 进行结果的对比，对比 ext4 与我们的文件系统在数据密集型场景的性能。

测试样例	ext4 性能 (s)	XCraft 性能 (s)
sample1.c	0.001823	0.002233
sample2.c	0.517606	0.725713
sample3.c	0.000823	0.003592
sample4.c	0.017394	0.062897
sample5.c	0.697304	0.806694

表 1: 性能测试结果 (单位:s)

可以看到我们的文件系统在数据密集型，尤其是 sample2.c, sample5.c, 创建 1 万个子目录，在子目录下创建文件，最后删除，性能接近 ext4，可以发现我们文件系统大致完成了最初设计的目标。



```

prayer@prayer-virtual-machine:~/os/project2210132-232458$ make test 2
#####test 2#####
alread write 10000 files
alread delete 10000 files
file write performance: 0.426255
#####test 2#####
prayer@prayer-virtual-machine:~/os/project2210132-232458$

```

图 7.30: 运行示意图

下面是不同用户权限管理的运行结果，仅测试权限管理，无性能对比：


```

prayer@prayer-virtual-machine:~/os/project2210132-232458$ make test6
User1: Successfully created file in /mnt/test/
User2: Successfully created file in /mnt/test/
mv: 无法将 '/mnt/test/user2_testfile.txt' 移动至 '/mnt/test/user2_testfile.txt': 权限不够
User1: Failed to rename user2's file (expected)
rm: 是否删除有写保护的普通空文件 '/mnt/test/user2_testfile.txt'?
rm: 无法删除 '/mnt/test/user2_testfile.txt': 权限不够
User1: Failed to delete user2's file (expected)
User2: Successfully read user1's file (expected)
-sh: 1: cannot create /mnt/test/user1_testfile.txt: Permission denied
User2: Failed to write to user1's file (expected)
prayer@prayer-virtual-machine:~/os/project2210132-232458$

```

表 2: make test 6 的运行结果

表 3: 内核中 XCraft 的权限提示

8 遇到的主要问题和解决办法

我们本次在开发我们的文件系统的过程中，遇到了许多问题，但是我们最后都对此进行了解决，主要依靠 Printk，以及 `sudo dmesg -w` 等命令，通过在内核输出打印消息，我们最终都大致消除了 BUG：

1. **mount** 时如果我们的 **root inode** 设置为第 0 个 inode 会产生无法挂载的问题，这是由于 **vfs** 层的某些特性导致的。此时我们将其设置为第 1 个 inode 即可解决；
2. 所使用的编译器要求我们在编写函数的时候要将函数中的变量声明写在函数最开头，执行代码块写在后面，不然就会出现如下警告：

```

/home/prayer/os/project2210132-232458/include/bitmap.h:255:5: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
255 |     uint32_t blocks_begin=0;
    |     ~~~~~
/home/prayer/os/project2210132-232458/include/bitmap.h:260:5: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
260 |     uint32_t ret=get_first_free_bits(sbi->s_ibmap_info[group]->bfree_bitmap,desc->bg_nr_blocks,len,blocks_begin);
    |     ~~~~~
/home/prayer/os/project2210132-232458/include/bitmap.h:264:9: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
264 |     uint32_t free_blocks_count=le32_to_cpu(sbi->s_super->s_free_blocks_count);
    |     ~~~~~

```

图 8.31: ISO C90 forbids

3. 编写 **mkfs** 格式化工具时没有注意到第 0 个块组可能是最后一个块组的情况，于是导致我们的第 0 个块组的块组描述符的字段赋值出错；
4. **mkfs** 格式化工具编写时最开始没有注意到块组描述符保留区的块的内容需要写入，只写了需要的块组的描述符所占的块。这也导致了我们的位图、inode 内容全部前推，使得获取内容时全部出错；
5. 编写 **bitmap.h** 时我们用到了 **linux** 自带的 `<linux/bitmap.h>` 头文件但是却没有包含，使得我们执行文件系统操作时被杀死，从而虚拟机崩溃；
6. 将磁盘内容导入内存定义的某些主机字节序的结构体中时，没有注意到要进行字节序的转换；

7. 内核态我们进行字节序转换时最开始使用了 `htole32` 以及 `le32toh` 等函数报错, 这是因为这些函数只能在用户态编写的程序下使用, 我们在内核态中只能使用 `cpu_to_le32` 以及 `le32_to_cpu` 等函数来进行字节序转换;

用户态时我们转字节序如下:

```
sb->xcraft_sb=(struct XCraft_superblock){
    .s_inodes_count = htole32(s_inodes_count),
    .s_blocks_count = htole32(s_blocks_count),
    .s_free_blocks_count = htole32(s_free_blocks_count),
    .s_free_inodes_count = htole32(s_free_inodes_count),
    .s_blocks_per_group = htole32(s_blocks_per_group),
    .s_inodes_per_group = htole32(s_inodes_per_group),
    .s_groups_count = htole32(s_groups_count),
    .s_last_group_blocks = htole32(s_last_group_blocks),
    .s_magic=htole16(XCRAFT_MAGIC),
    .s_inode_size=htole16(XCRAFT_INODE_SIZE),
};
```

图 8.32: 转换字节序 1

内核态时我们转字节序如下:

```
disk_inode->i_atime = cpu_to_le32(inode->i_atime.tv_sec);
disk_inode->i_mtime = cpu_to_le32(inode->i_mtime.tv_sec);
disk_inode->i_nr_files = cpu_to_le32(xi->i_nr_files);
disk_inode->i_blocks_lo = cpu_to_le32(inode->i_blocks);
disk_inode->i_links_count = cpu_to_le16(inode->i_nlink);
disk_inode->i_flags = cpu_to_le32(xi->i_flags);
```

图 8.33: 转换字节序 2

8. 编写文件系统头文件时最开始将一些函数定义写在了里面导致了报错, 后面发现在此头文件中只能编写函数声明, 其函数定义需要在其他文件中实现;
9. 我们最开始编写的代码出现了头文件包含问题, 头文件重包含会导致编译时出现函数重定义以及没有定义的情况。我们后面重新组织了头文件, 不再出现这些问题;
10. 编写代码过程中经过内核信息打印发现存在 `buffer_head` 被重复释放的问题, 后面我们通过内核打印等调试信息成功定位并解决了 `buffer_head` 重复释放的问题;
11. 我们在编写挂载过程中会用到的 `XCraft_fill_super` 函数时, 其中会读取第 0 个块组的位图, 对所有其他块组的位图在内存分配空间并进行初始化。在这过程中我们最开始仍然没有注意第 0 个块组可能是最后一个块组的情况, 其他块组的位图在分配空间时也没有注意到最后一个块组的情况, 导致位图大小分配有误。不过后续我们对此进行了解决;

12. 在内存中我们判断 inode 是否已生成哈希树的字段是 `XCraft_inode_info` 中的 `i_flags` 字段。当我们生成哈希树后，我们需要对 `i_flags` 字段进行修改，但是修改时我们 VFS 层 inode 中的 `i_flags` 字段进行了混淆。并且我们的 VFS 层的 inode 我们从来没有对其 `i_flags` 字段赋值，导致了报错；

```
static int XCraft_set_inode_flag(struct inode*dir,uint32_t flag){  
    // 获取inode_info  
    struct XCraft_inode_info *dir_info = XCRAFT_I(dir);  
    dir_info->i_flags|=flag;  
    return 0;  
}
```

图 8.34: `i_flags` 字段混淆导致 bug

13. 我们在遍历磁盘块中的目录项的过程，是依据目录项中的 `rec_len` 字段定位到下一个目录项的。我们最开始遍历整块内容时没有注意到目录项可能不存在的情况，当遍历到某一个空目录项时起 `rec_len` 字段为 0，导致其一直停留在此位置进行死循环，最终引发虚拟机崩溃。后面我们对此进行了解决，即每当发现空目录项，便跳出循环；
14. 在编写与 `ls` 命令有关的 `XCraft_readdir` 函数时，最开始没有注意到提交 `.` 和 `..` 目录项到遍历显示的上下文中，使得 `ls` 命令出错无法显示信息。我们后面再遍历目录项之前提交 `.` 和 `..` 目录项到遍历显示的上下文中如下：

```
/* Commit . and .. to ctx */  
if(!dir_emit_dots(dir, ctx))  
    return 0;
```

图 8.35: `ls` 命令相关 bug

15. 在编写哈希树中遍历目录项的函数中，借用了 `dfs` 的思想，但是在其中出现了回溯出错，没有区分 `dx_root` 和 `dx_node` 等问题，加上代码量大，产生了很大问题。不过最后通过重新分析，成功梳理了逻辑，完成了哈希树中遍历目录项的函数的编写；
16. 我们在目录下每次创建和删除文件之后忘记了更新其记录目录下有多少文件的字段信息，导致最后我们删除目录时总是显示目录不为空，无法删除目录。最后，我们修改了代码，及时更新了统计目录下有多少文件的字段信息，成功解决了此问题。比如在删除文件后，需要对相应目录的文件数减 1：

```
// 如果inode是目录，需要将dir的硬连接数减1
if (S_ISDIR(inode->i_mode))
    drop_nlink(dir);

dir_info->i_nr_files-=1;
```

图 8.36: 删除文件时 i_nr_files 处理

比如在创建文件后，需要对相应目录的文件数加 1：

```
if (S_ISDIR(mode))
    inc_nlink(dir);
dir_info->i_nr_files += 1;
```

图 8.37: 创建文件时 i_nr_files 处理

综上，我们在代码编写中确实产生了较多 bug，但是我们最后都进行了解决。其中最重要的方法是我们增加了许多内核打印信息在编写的重要函数中，通过内核信息的打印信息我们可以快速定位出错代码的位置，从而进行解决。

9 分工与协作

任务	负责人员
前期调研 Linux 内核模块化编程	谢畅
学习 fat、SimpleFS 的编写框架	张景瑞
调研总结 ext4 文件系统的特性	谢畅
学习调研块组的实现原理与方式	张景瑞
学习调研目录哈希的建树、删除操作	谢畅
完成代码编写	谢畅、张景瑞

表 4: 分工与写作过程

10 未来计划

未来我们的计划如下：

1. 增设**校验和**机制，做到及时发现磁盘的读写错误；
2. 实现**快速启动**，优化初始化机制，提高文件系统的运行效率。
3. **优化哈希树与扩展树**：尽力消除 BUG。

11 项目目录

```

├─ data_about_learning # 准备阶段学习记录
├─ include
│   ├── bitmap.h      # 位图操作
│   ├── hash.h        # 目录哈希操作
│   └─ XCRAFT.h       # 文件系统头文件
├─ src
│   ├── dir.c         # 目录操作
│   ├── file.c        # 文件操作
│   ├── fs.c          # 文件系统注册挂载与卸载
│   ├── inode.c       # inode操作
│   ├── mkfs.c        # 文件系统格式化
│   └─ super.c        # 超级块操作
├─ test
│   ├── sample1.c     # 测试文件 读写文件
│   ├── sample2.c     # 测试文件删除创建100个目录及目录下文件
│   └─ ...
├─ Makefile           # 编译文件
├─ README.md
├─ setup.sh           # 运行脚本
└─ 4test.sh           # 产生文件系统，随后利用`make test`测试文件系统

```

图 11.38: 项目目录说明

12 比赛收获

在这次比赛中我们进行了从 0 到有的突破，学习了如何在一个全新问题上进行开发项目。一开始我们并不清楚怎么进行 linux 的模块化编程，所以前期主要在学习如何构建框架，然后才是逐渐去写文件系统，基于我们的开发思路集中式的写，最后成功完成了这个项目，跨越了 5 个月。

同时我们不仅实现了一个文件系统，同时结合了具体数据密集型的使用场景，实现了延迟初始化、目录哈希、大文件扩展树、不同用户权限管理的特性，同时比对了 ext4 与我们设计的 XCraft 文件系统的性能，发现我们文件系统性能良好。

而且，我们还学习了 makefile 的一些知识，设计了 include、src、test 三个文件夹的结构，使得项目结构更加清晰。同时 make test 也能让我们对文件系统的性能测试更加顺利！具体演示见 Readme 的视频链接。