



HoitFS: A Norflash based filesystem implemented on SylixOS

成员： 潘延麒、胡智胜、张楠

指导教师：夏文、江仲鸣、陈洪邦、蒋太金

摘 要

HoitFS 是一款基于 SylixOS 的 Norflash 文件系统，它以 JFFS2 作为原型，承接了 JFFS2 的各种优缺点。与类似 EXT2 的磁盘文件系统将索引结构布局在磁盘固定位置不同，JFFS2 将内存结构布局到了内存中，该设计很大一部分原因是由于 Norflash 的擦写寿命有限，不能原地更新，只能异地更新(Out-of-Place Update)。这意味着 JFFS2 的可拓展性较差，原因有二：首先，为了维护内存索引，JFFS2 需要在挂载时扫描整个介质来构建内存结构；其次，Norflash 通常被用于嵌入式设备，这意味着并没有大量内存供 JFFS2 使用。于是，当介质容量提升，JFFS2 的性能便会有明显的下降。此外，由于 JFFS2 采用异地更新模式，因此当介质即将写满后将触发垃圾回收机制 (Garbage Collection, GC)，此时整个文件系统的写入带宽将受到严重影响。

针对 JFFS2 的上述缺点，HoitFS 设计实现了三个优化方案：① 擦写总结块 (Erasable Summary Block) 实现。将提升挂载性能；② 后台 GC 实现 (Background GC)。将减少前台 GC 次数，从而降低文件系统 GC 负载；③ 可合并红黑树 (Mergeable Tree) 实现。将显著降低顺序小数据写入带来的红黑树内存负载。

此外，为了测试 HoitFS 文件系统性能，我们移植了新型 Norflash 文件系统 SpifFS，并制作了基于 Norflash 的基准测试工具 fstester。最终测试结果表明 HoitFS 在数据写入、垃圾回收等方面都有占有优势，这得益于 HoitFS 的索引分布在内存中；SpifFS 却在挂载、顺序读方面表现突出，这得益于 SpifFS 的索引分布在物理介质中。从 SpifFS 的设计中，我们获取了更多 HoitFS 优化灵感，这将在未来展望一节中做出阐述。

总结 HoitFS 项目做出的贡献如下：

- **完成赛题基本要求**。包括：① 文件系统基本接口完成 (文件读、写、目录操作等)；② 软、硬连接，读写平衡实现；③ 掉电安全实现；
- **移植 SpifFS**。除了开发 HoitFS，我们将 SpifFS 移植到 SylixOS 中，使得我们能够进行性能评测；
- **EBS + MT 机制**。我们为 HoitFS 实现了 EBS 与多线程 (Multi Thread, MT) 挂载机制，使得挂载速率相比未采用该机制有了显著的提升。同时，EBS 的存在在一定程度上保证了文件系统写入的一致性；

- **Background GC**。我们为 HoitFS 加入了后台 GC 机制，使得我们在写入相同大小文件的情况下，能够相对减少前台 GC 次数，从而提升写效率；
- **Mergeable Tree**。为了解决小数据写入带来的内存爆炸性增长，我们设计并实现了 Mergeable Tree，并能够大大缓解小数据写入带来的内存开销；

接下来的章节将这样安排：第一小节介绍**项目背景**，包括 Norflash 设备，JFFS2 以及 SpifFS 文件系统；第二小节介绍 **HoitFS 设计与实现**，包括 HoitFS 上层设计与在 SylixOS 上的实现；第三小节介绍 **HoitFS 的优化设计**，包括 EBS、Background GC 以及 Mergeable Tree 设计动机与实现方案；第四小节介绍 **HoitFS 实验测试**，测试前准备与测试结果；第五小节提出**未来展望**，给出更多的头脑风暴方案；第六小节对整个 HoitFS 项目历程做出**回顾与总结**，谈谈这一年来的收获与体会。

目 录

1. 项目介绍.....	6
1.1 Norflash 设备.....	6
1.2 JFFS2 与 SpifFS 文件系统	8
1.2.1 JFFS2	9
1.2.2 SpifFS	9
2. HoitFS 设计与实现.....	10
2.1 HoitFS 上层设计	11
2.1.1 节点设计.....	11
2.1.2 节点管理结构设计.....	11
2.1.3 文件组织.....	12
2.1.4 可靠性设计.....	14
2.2 HoitFS 实现	16
2.2.1 驱动适配.....	16
2.2.2 HoitFS 接入 SylixOS	17
3. HoitFS 优化设计	19
3.1 EBS + MT 机制	19
3.2 Background GC 机制	20
3.3 Mergeable Tree	20
4. HoitFS 实验测试	22
4.1 实验准备.....	22
4.1.1 环境准备.....	22
4.1.2 实验策略.....	22
4.2 实验结果.....	24
4.2.1 基本读写测试结果.....	24
4.2.2 挂载测试结果.....	27
4.2.3 Background GC 测试结果	28
4.2.4 Mergeable Tree 测试结果	29
5. 未来展望.....	29

5.1 冷热文件分离.....	30
5.2 虚拟文件描述符.....	30
5.3 受 SpifFS 启发的新型索引结构设计	31
6. 回顾与总结.....	31
7. 参考文献.....	33

1. 项目介绍

HoitFS 项目以企业需求为背景、以赛题为驱动，旨在实现一款基于 SylixOS 的 Norflash 文件系统。题目要求较为简单：① 文件系统基本 I/O 接口实现；② 软、硬链接实现，磨损均衡实现；③ 掉电安全实现。在开始正式介绍前，先自问自答三个问题：

- 为什么要选择该题目？

SylixOS 是国人自研大型实时嵌入式操作系统，在航空航天、工业国防等领域均有突出的贡献，我们希望能够一览现代国产操作系统的雄姿，学习其架构思想，为国产 OS 的成长献一份绵薄之力；

- 为何题目要求设备是 Norflash 而非 Nandflash？

尽管 Nandflash 无论在市场规模还是在综合性能方面都优于 Norflash，但为何题目仍然是在 Norflash 上开发文件系统呢？原因有二：首先，SylixOS 并没有类似 Linux 的 MTD（Memory Technology Device）设备，因此 SylixOS 还不支持对 Norflash 的访问；其次，SylixOS 的工业合作伙伴对 Norflash 设备有所需求，因此，需要我们完成一个支持 Norflash 的文件系统。

- 为什么叫 HoitFS？

Hoit 可以被拆为 Hot HITSZer，意为热情的哈工深入，此外，我们团队名原本就是 Hoit-23o2，这是我们团队的仓库地址 [Hoit-23o2 \(github.com\)](https://github.com/Hoit-23o2)，因此，称其为 HoitFS 也就顺理成章了。

1.1 Norflash 设备

Norflash 和 Nandflash 是现在市场上两种主要的非易失闪存技术。Intel 于 1988 年首先开发出 Norflash 技术，打破了由 EPROM 与 EEPROM 一统天下的局面。紧接着，1989 年，东芝公司发表了 Nandflash，Nandflash 的设计思想是强调降低每比特成本、更高的性能收益以及像磁盘一样可以通过接口轻松升级。

Norflash 的特点是片上执行（eXecute In Place, XIP），应用程序可以直接在 flash 闪存上运行，不必再把代码读入 RAM 中，如此 Norflash 的传输效率较传统磁盘自然有较大的提升[1]。其中 1 ~ 4MB 小容量 Norflash 具有最大的成本收

益。但 Norflash 的写入和擦除效率实在过于低下，远不如 Nandflash，这是由于 Norflash 要求在进行写入前，需要将目标块内的位擦除为 1，而 Norflash 的擦除单位是 64 ~ 128KB 的块，因此执行一个“擦除-写入”时间约为 5s；而 Nandflash 的擦除单位是 8 ~ 32 KB 的块，且写入操作较为直接简单，因此执行一个“擦除-写入”时间最多只需 4ms。

Norflash 相较于 Nandflash 的优势在于它读速率较快，因为 Norflash 具有随机访问特点，而 Nandflash 只能通过按页的方式访问。

本次大赛使用的 Norflash 芯片是 Am29LV160DB，该芯片搭载于 mini 2440 开发版上，大小仅具 2MB。据 mini 2440 官方描述，该 Norflash 芯片主要用于嵌入式学习使用。此外，该 Norflash 还常被用于存放 U-boot，利用 Norflash 的 XIP 特性能够加快 OS 启动速率。表 1.1 总结了 Am29LV160DB 芯片的读写性能带宽，可见，与传统磁盘设备相比，Am29LV160DB 芯片的带宽要低很多。

表 1.1 Am29LV160DB 芯片读写性能参数[2]

	读	按字写	按字节写
耗时（us）	Max. 0.07 ~ 0.12	7 ~ 210	5 ~ 150
带宽（kb / s）	Max. 8,138 ~ 13,950	9 ~ 279	6.5 ~ 195

注：Max.代表至多，读性能参数手册没有给出具体参数，只能通过时序图计算，这里计算时只考虑 Addresses Stable Time

Am29LV160DB 的研发至今也长达 20 年之久，因此 Am29LV160DB 并不能作为工业界衡量 Norflash 性能标准，事实上，华邦电子（Winbond Elections）生产的 Norflash 在性能上有着显著提升，以 W25Q256JW_DTR Norflash 芯片为例 [3]，其读速率高达 66MB/s，也就是 Am29LV160DB 的 6 倍，而其最快擦写吞吐量在 133MHZ 的机器上也高达 1330 次/s。此外 W25Q256JW_DTR 的存储容量也有 32MB，远大于 Am29LV160DB。

那么，在 Am29LV160DB 上进行开发是否与现代生产脱节呢？关于这一点，赛题出题方也没能给出一个令我们满意的解释。大概理由有两点：① SylixOS 目前支持的带 Norflash 的单板 mini 2440 便是其中之一，如果用其他单板做开发可能事倍功半；② 我们团队认为 SylixOS 可能更希望的是项目 Demo 而非真正地

为公司提供可合并代码，因此使用 Am29LV160DB 就够了。

1.2 JFFS2 与 SpifFS 文件系统

传统 Norflash 文件系统有很多：JFFS 系列、YAFFS 系列等，它们都是基于日志结构的文件系统，这是由于 flash 设备擦除寿命有限所致——例如 Norflash 的擦写寿命仅在 10 万次左右，文件系统不能做原地更新，所有的更新都应该是异地的、追加的写在介质上。此外，为了支持异地更新特性，flash 文件系统的一切数据都以节点形态存在，每个节点带有元数据，用于记录它的逻辑信息。这些文件系统间最大的不同在于节点索引结构的设计。例如以 JFFS2 为典型的索引结构分布在内存中，如图 1.1 所示。这样的结构能够提升文件查找、更新效率。

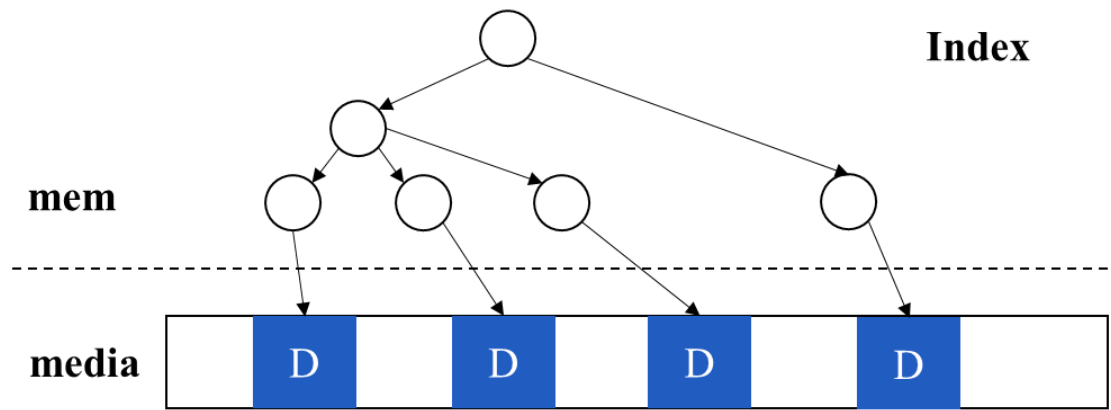


图 1.1 索引在内存中

另外便是以 YAFFS 为例的索引结构分布在介质中，如图 1.2 所示。这样的结构有利于文件系统的拓展，但在更新、查找时效率会有所下降。

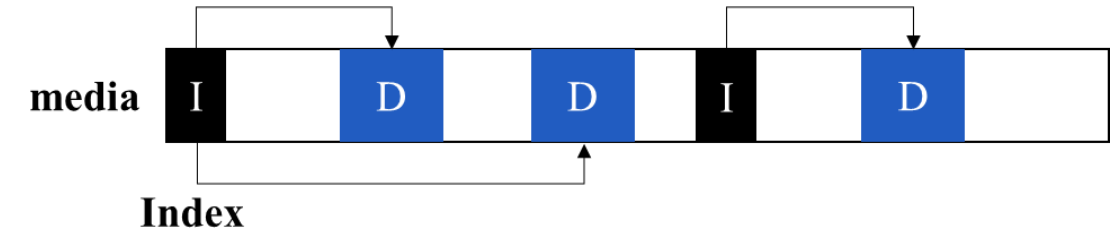


图 1.2 索引在介质上

SpifFS 是基于 YAFFS2 改良的文件系统，由 [pellepl](#) 于 2013 年开发，其因 RAM 开销小、快速读特性，目前已经被广泛应用于嵌入式领域。华为自研操作

系统 Lite OS 便搭载了 SpifFS 文件系统能力[4]。

接下来，我们着重介绍 JFFS2 以及 SpifFS 文件系统的关键设计。

1.2.1 JFFS2

JFFS2 的关键设计在于将索引结构放在内存中，从而加速文件读写、更新速率。JFFS2 的内存结构分为 3 层。

首先是**物理层**，物理层收集整个 Norflash 上的有效节点，并构建指向节点的 Ref 结构；其次是**逻辑层**，逻辑层将物理层属于同一个文件的节点收集起来，并记录它们在文件中的逻辑位置；最后是**逻辑管理层**，该层将逻辑层节点按逻辑顺序组合起来，为上层接口提供便捷的访问手段。

可见，JFFS2 的内存开销不小，随着介质增大，JFFS2 在内存中需要管理的结构将会线性增长，这是 JFFS2 的最大弊端。

1.2.2 SpifFS

有关 SpifFS 设计实现与移植，我们有更详细的文档说明，见[5]，这里仅对 SpifFS 关键思想做出阐述。

SpifFS 将介质划分为 Sector，Sector 再被划分为页面，每个 Sector 均具有 Lookup 页面，Lookup 页面类似位图，记录了该 Sector 中每个页面的元信息，大大提升查找效率，SpifFS 布局如图 1.3 所示。其中 Index 页面表示一个文件的索引结构，而 Data 页面自然表示一个文件的数据。

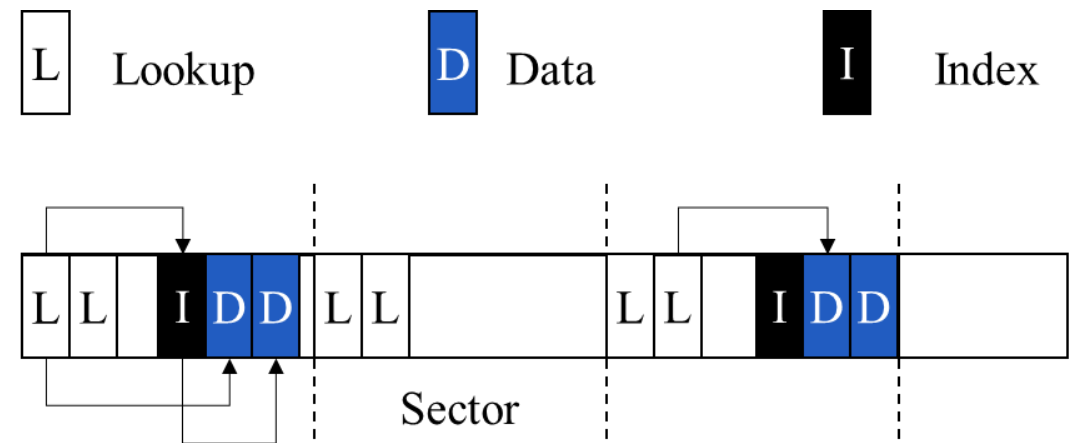


图 1.3 SpifFS 布局

SpifFS 的设计目标是尽可能节省内存开销，因此，SpifFS 的上层结构中只缓存一个 Lookup 页面与一个 Index 页面用于加速当前查找，此外 SpifFS 采用 Cache 结构来缓存 Data 页面，每个文件可将字节页面加载到 Cache 中从而加速当前写入与读出效率，如图 1.4 所示。

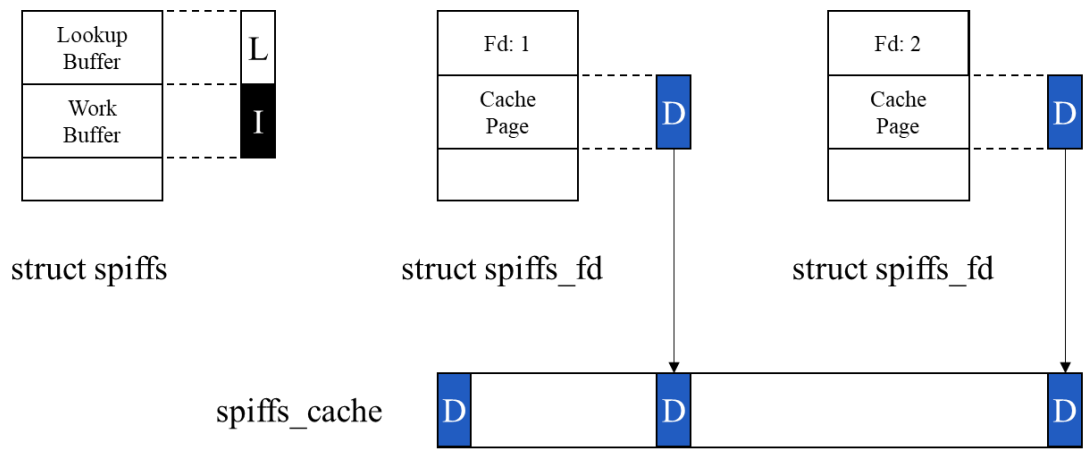


图 1.4 SpifFS 内存结构

可见 SpifFS 的内存开销远低于 JFFS2 内存开销，但有得必有失，SpifFS 设计虽然将内存开销降到了一个极低的水平，但是它的读写性能也会因此受到一定的影响，很大一个原因在于它在内存中只缓存了一个 Index 页面，当发生随机读写的时候，由于 Index 页面的不断更新，将会导致 SpifFS 需要不断扫描介质，这造成了它的瓶颈。而 JFFS2 由于整个索引结构都在内存中，因此其随机读写效率将远高于 SpifFS。

2. HoitFS 设计与实现

HoitFS 的设计承接自 JFFS2，在理解了 JFFS2 的原理后，我们从 0 到 1 在 SylixOS 上自行开发实现了 HoitFS。本节从较高的维度去介绍 HoitFS 的设计与实现，更详细的设计实现文档可以参考[6]。

该结构由两部分组成：① 描述物理节点信息的 HOIT_RAW_INFO 结构；② 快速定位 HOIT RAW INFO 的 HOIT INODE CACHE。其中，HOIT RAW INFO

的 `phys_addr` 字段将记录节点的特定位置，`next_logic` 字段串接起所有 `ino` 相同的节点，`next_phys` 字段则是用于记录下一个在物理介质上相邻的节点位置。由 `next_logic` 字段串接起的节点构成了整个文件号为 `ino` 的文件，不过**注意**，此时只能确定该文件由这些节点组成，却并不确定这些节点如何组成该文件。

`HOIT_INODE_CACHE` 为文件访问它的所有节点提供了便利。`HoitFS` 会为每个文件都维护一个 `HOIT_INODE_CACHE` 结构，`HoitFS` 通过单链表结构管理 `HOIT_INODE_CACHE`。其中字段 `HOITC_ino` 记录了文件 `ino` 号，字段 `HOITC_nodes` 则记录了首个属于该文件的 `HOIT_RAW_INFO` 结构。通过 `HOITC_nodes` 以及 `next_logic` 字段，`HoitFS` 便能够快速获取某文件的所有物理节点。

由于 `HoitFS` 节点管理结构直接与物理节点打交道，因此，我们把它视为 `HoitFS` 的物理层结构。

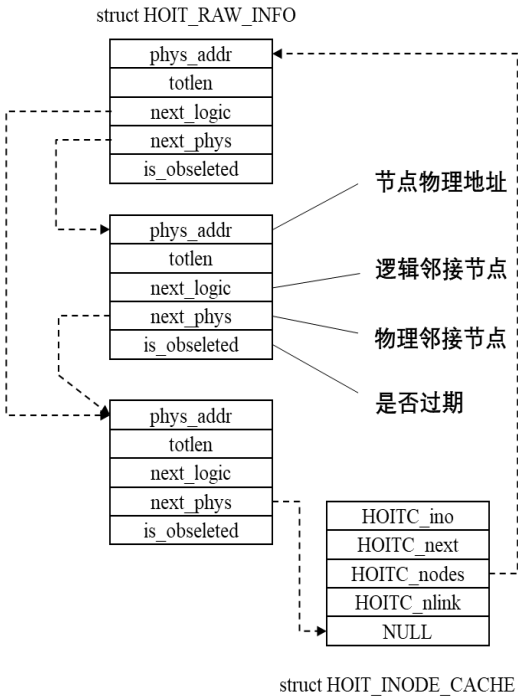


图 2.2 HoitFS 节点管理结构

2.1.3 文件组织

`HoitFS` 的文件结构在物理层的基础上搭建。首先，`HoitFS` 会通过物理节点携带的信息建立起逻辑层结构，逻辑层结构的节点如图 2.3 所示。其中，字段

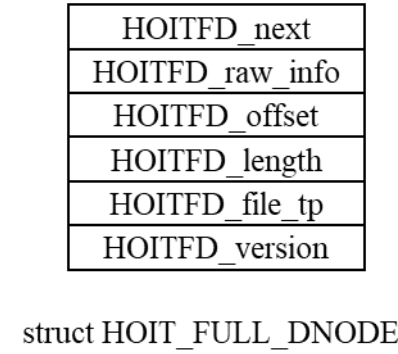


图 2.3 HoitFS 逻辑节点

`HOITFD_raw_info` 指向物理层节点，字段 `HOITFD_offset` 描述了该物理节点在文件内部的偏移，字段 `HOITFD_length` 描述了该物理节点在文件内部的大小，字段 `HOITFD_file_tp` 及字段 `HOITFD_version` 分别表示文件类型以及当前节点版本号。逻辑层结构为文件的每个物理节点找到了在文件中的归宿。但需要注意的是，此

时的逻辑节点间还没有建立起一致性联系，举个例子，假设有两个逻辑节点：第一个节点表示文件[10,100]区间，第二个节点表示文件[20,110]区间，此时两个逻辑节点发生了覆盖，再假设第二个节点 `version` 字段更高，表明第二个节点覆盖了第一个节点，因此，如果我们需要访问该文件区间[10,110]的内容，我们需要的是第一个节点[10,20]部分以及第二个节点[20,110]部分，这才保证了逻辑节点的一致性。

为了加速这种一致性维护，我们在逻辑层之上建立起了逻辑管理层。逻辑管理层的本质是一颗红黑树，红黑树节点的键值为逻辑节点的 `HOITFD_offset` 字段，在每次添加新节点时，我们便通过遍历红黑树找到产生逻辑覆盖部分，并作一致性修正，从而维护好文件的逻辑结构。算法 1 描述了红黑树插入过程中我们会做的工作。其中，`do_fix` 将会根据四种不同的覆盖方式进行一致性恢复，详情请见 [6] “3.2.3 HoitFS FragTree 管理层”，这里不再赘述。

Algorithm 1: hoitFragTreeInsert(tree, node)

```
1. min_node = get_minimum(tree)
2. traverse = min_node
3. cur_low = node.HOITFD_offset
4. cur_high = cur_low + node.HOITFD_length
5. while(traverse) {
6.     next = get_succesor(traverse)
7.     target_low = traverse.HOITFD_offset
8.     target_high = target_low + traverse.HOITFD_length
9.     if [cur_low, cur_high] overlay [target_low, target_high]
10.        do_fix()
11.    endif
12.    traverse = next
13. }
```

最终，整个 HoitFS 文件结构的全貌如图 2.4 所示。其中，`HOIT_INODE_INFO` 结构便是 HoitFS 文件的最上层描述，其中有三个字段最为重要：① `HOITN_ino`，该字段表明该文件的唯一 `ino` 号；② `HOITN_ic`，`ic` 即 `inode_cache`，该字段指向前文描述的 `HOIT_INODE_CACHE` 结构；③ `HOITN_fragtree`，该字段构建起用于逻辑层管理的红黑树，使得整个文件系统能够有条不紊的执行读写操作。

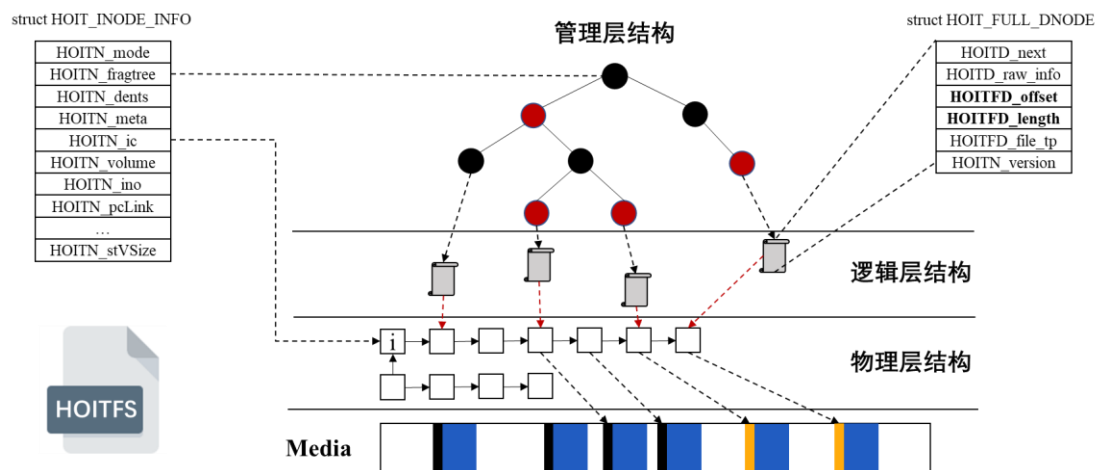


图 2.4 HoitFS 文件结构

可见，HoitFS 延续了 JFFS2 的思想，通过物理层、逻辑层以及逻辑管理层共同支撑起文件系统的核心。

2.1.4 可靠性设计

由于 Norflash 擦写寿命有限，因此我们需要小心翼翼地规划每一次落盘 IO，从而保证 HoitFS 的可靠性。为此我们设计了三种机制，它们分别是：① 多头链表选择机制；② Cache 缓存机制；③ 比特级检验算法；

2.1.4.1 多头链表选择

多头链表的设计顺延了 JFFS2 的设计思路：划分不同类别的 Sector，利用随机性保证每个 Sector 都有擦写的机会，从而保证 Norflash 的磨损均衡。为了简化设计，我们定义了三类型的链表：① Dirty List，该链表代表所有的具有过时节点（出现节点覆盖很可能产生过时节点）的 Sector；② Clean List，该链表代表所有的全为有效节点的 Sector；③ Free List，该链表代表被标记为空的 Sector。在擦除回收时，有 99% 的概率从 Dirty List 中选择 Sector，而有 1% 的概率从剩下两种链表选择 Sector。虽然暂时无法证明该方法具有多好磨损均衡性[7]，但从前人的经验来看，这样做是比较合理的。

2.1.4.2 Cache 缓存机制

Cache 机制是为了尽可能延缓落盘 IO，从而减少对 Norflash 的大量写入。我们采用双向链表结构维护 Cache Block 结构，每个 Cache Block 都具有数据缓存的能力，当所有 Cache Block 都被写满后，我们会采用 LRU 算法换入一个空的 Cache Block，并将被替换的 Cache Block 的数据回写到 Norflash 的指定位置，Cache 机制示意图如图 2.5 所示。

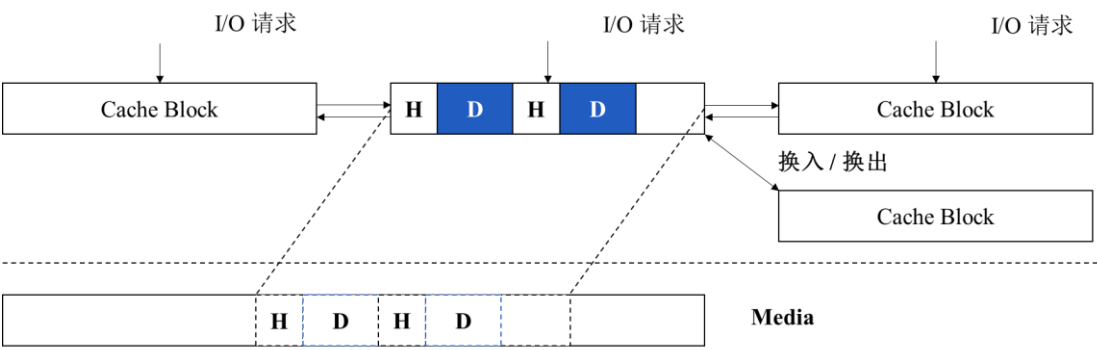


图 2.5 HoitFS Cache 缓存机制示意图

2.1.4.3 比特级检验算法

在 1.1 节中，我们提到在每次写 Norflash 前，我们会进行擦除操作，但这不完全正确。事实上，对于 Norflash 而言，其写操作是将比特 1 置为比特 0，当我们需要在原本比特 0 的位置写入比特 1 时，此时才需要擦除该比特所在的整个 Sector。因此，并非每次写入我们都需要擦除整个 Sector，基于这个思想，我们设计了比特级检验算法，其目的在于尽可能避免不必要的擦除操作，算法 2 给出了比特级检验算法的实现流程。

Algorithm 2: nor_check_should_erase (base, offset, content, size_bytes)

```
1.  for (i = 0; i < size_bytes; i++)
2.  {
3.      byte_in_flash = read_byte_from_mem(base, offset + i);
4.      byte_to_write = (UINT8)*(content + i);
5.      byte_diff = byte_in_flash ^ byte_to_write;
6.      if((byte_diff & byte_to_write) != 0){
7.          return TRUE;
8.      }
```

Algorithm 2: nor_check_should_erase(base, offset, content, size_bytes) (续)

```
9.  }  
10. return FALSE;
```

该算法的核心在于将介质上待被写入部分与即将写入部分进行字节的按位异或，并将异或后的结果与即将写入字节做与操作，如果发现结果均等于 0，说明所有的写操作都是将 0 写为 0 或是将 0 写为 1 或是将 1 写为 1，故不需要擦除，反之才需要擦除。

2.2 HoitFS 实现

2.2.1 驱动适配

Am29LV160DB 官方数据手册给出了控制 Norflash 的时序，如图 2.6 所示。根据控制时序，我们便能够轻松编写相应读写接口。

Command Sequence (Note 1)			Cycles	Bus Cycles (Notes 2–5)											
				First		Second		Third		Fourth		Fifth		Sixth	
				Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read (Note 6)			1	RA	RD										
Reset (Note 7)			1	XXX	F0										
Autoselect (Note 8)	Manufacturer ID	Word	4	555	AA	2AA	55	555	90	X00	01				
		Byte	4	AAA		555		AAA							
	Device ID, Top Boot Block	Word	4	555	AA	2AA	55	555	90	X01	22C4				
		Byte	4	AAA		555		AAA		X02	C4				
	Device ID, Bottom Boot Block	Word	4	555	AA	2AA	55	555	90	X01	2249				
		Byte	4	AAA		555		AAA		X02	49				
	Sector Protect Verify (Note 9)	Word	4	555	AA	2AA	55	555	90	(SA) X02	XX00 XX01				
		Byte	4	AAA		555		AAA		(SA) X04	00 01				
CFI Query (Note 10)		Word	1	55	98										
		Byte		AA											
Program		Word	4	555	AA	2AA	55	555	A0	PA	PD				
		Byte		AAA		555		AAA							
Unlock Bypass		Word	3	555	AA	2AA	55	555	20						
		Byte		AAA		555		AAA							
Unlock Bypass Program (Note 11)			2	XXX	A0	PA	PD								
Unlock Bypass Reset (Note 12)			2	XXX	90	XXX	00								
Chip Erase	Word	6	555	AA	2AA	55	555	80	555	AA	2AA	55	555	10	
	Byte		AAA		555		AAA		AAA		555		AAA		
Sector Erase	Word	6	555	AA	2AA	55	555	80	555	AA	2AA	55	SA	30	
	Byte		AAA		555		AAA		AAA		555				
Erase Suspend (Note 13)			1	XXX	B0										
Erase Resume (Note 14)			1	XXX	30										

图 2.6 Am29LV160DB 时序接口

由于 SylixOS 没有 MTD 设备，因此我们无法对 Norflash 进行访问，因此只


有通过 mmap 的方法将 Norflash 设备地址空间 map 到 RAM 中进行操作，从而完成 Norflash 适配。

2.2.2 HoitFS 接入 SylixOS

HoitFS 的具体开发实现可以参考[6]，在此不做赘述。我们着重介绍如何在 SylixOS 上搭载一个文件系统。与 Linux 不同，SylixOS 的搭载更加模板化，主要需要完成三个上层接口函数：

- **API_HoitFsDrvInstall**

该函数用于注册文件系统能力，直观来讲就是能够通过 shell mount 操作识别到 HoitFS 文件系统，如图 2.7 所示。其中，__hoitFSOpen 等函数就是我们需要实现的函数，它们控制文件系统的基本操作。



```
1  LW_API
2  INT  API_HoitFsDrvInstall(VOID)
3  {
4      struct file_operations    fileop;
5
6      if (_G_iHoitFsDrvNum > 0) {
7          return (ERROR_NONE);
8      }
9
10     lib_bzero(&fileop, sizeof(struct file_operations));
11
12     fileop.owner = THIS_MODULE;
13     fileop.fo_create = __hoitFsOpen;
14     fileop.fo_release = __hoitFsRemove;
15     fileop.fo_open = __hoitFsOpen;
16     fileop.fo_close = __hoitFsClose;
17     fileop.fo_read = __hoitFsRead;
18     fileop.fo_read_ex = __hoitFsPRead;
19     fileop.fo_write = __hoitFsWrite;
20     fileop.fo_write_ex = __hoitFsPWrite;
21     fileop.fo_lstat = __hoitFsLStat;
22     fileop.fo_ioctl = __hoitFsIoctl;
23     fileop.fo_symlink = __hoitFsSymlink;
24     fileop.fo_readlink = __hoitFsReadlink;
25
26     _G_iHoitFsDrvNum = iosDrvInstallEx2(&fileop, LW_DRV_TYPE_NEW_1);    /* 使用 NEW_1 型设备驱动程序 */
27
28     DRIVER_LICENSE(_G_iHoitFsDrvNum, "GPL->Ver 2.0");
29     DRIVER_AUTHOR(_G_iHoitFsDrvNum, "HITSZ.HoitGroup");
30     DRIVER_DESCRIPTION(_G_iHoitFsDrvNum, "norflash fs driver.");
31
32     _DebugHandle(__LOGMESSAGE_LEVEL, "norflash file system installed.\r\n");
33
34     __fsRegister("hoitfs", API_HoitFsDevCreate, LW_NULL, LW_NULL);    /* 注册文件系统 */
35
36     return ((_G_iHoitFsDrvNum > 0) ? (ERROR_NONE) : (PX_ERROR));
37 }
```

图 2.7 文件系统注册函数

● API_HoitFsDevCreate

该函数用于创建类似与 Linux 的 `super_block` 结构，同时也是挂载指令入口，如图 2.8 所示。其中，可以把 `PHOIT_VOLUME` 视为 `super_block` 结构。



图 2.8 文件系统挂载函数

● API_HoitFsDevDelete

该函数用于取消挂载一个文件系统，较为简单，如图 2.9 所示。

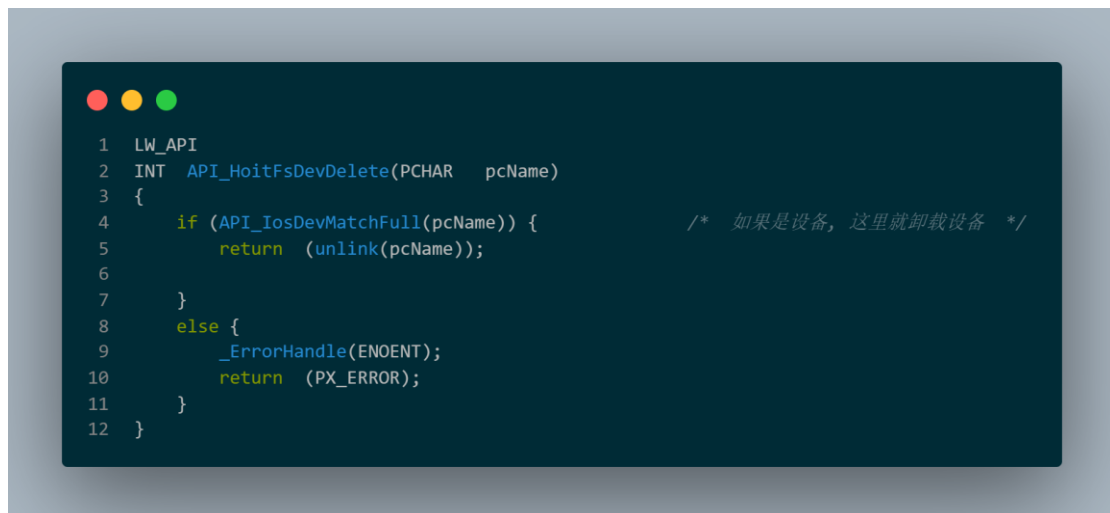


图 2.9 文件系统取消挂载函数

3. HoitFS 优化设计

3.1 EBS + MT 机制

正如 JFFS2 一样，HoitFS 也存在着拓展性差的问题。这里考虑一个情景：当 Norflash 介质不断增大，导致介质上的节点数目几何倍数上涨，我们需要扫描整个介质来建立物理层结构，这样一来时间的开销无疑是巨大的。为了改善这一点，我们不得不考虑引入固定索引机制，也就是 EBS 块。我们将 Sector 划分为数量一定的 Page，且保证每次数据写入都按 Page 对齐，如此一来，我们便能在每个 Sector 末尾放置固定大小的 EBS，来记录 Sector 上各个节点的位置情况。EBS 布局如图 3.1 所示。



图 3.1 EBS 布局

我们可以将 EBS 理解为一个表格，表格的每一项包含索引一个节点所需要的全部元信息，包括该节点所属的 ino 号、该节点在该 Sector 内的偏移以及该节点是否过期。有了 EBS 我们便能大大减少介质扫描的范围，保证每次介质访问到的都是有效的数据节点（如果过期，则跳过该项）。

此外，EBS 项显式地将各个 Sector 隔离开，这为 Sector 的并发访问提供了可能，如图 3.2 所示，我们可以通过多个线程共同扫描加速物理层结构的构建。

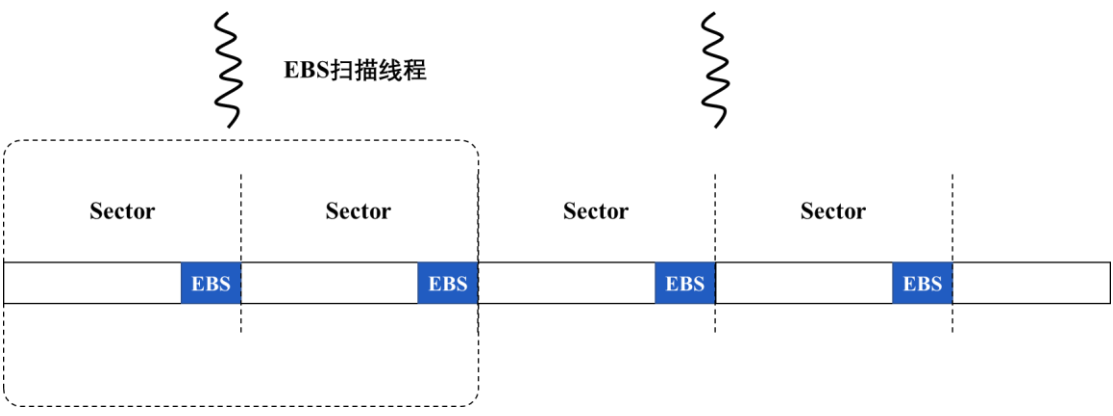


图 3.2 基于 EBS 的多线程扫描机制

3.2 Background GC 机制

由于 HoitFS 属于日志型文件系统，因此介质很容易就被写满了，当介质被写满后，GC 将被不断触发，此时文件系统的负载开销将变得格外大。如果用户对实时性要求偏高，那么这种开销是难以接受的。为此，我们引入了后台 GC 机制，其基本思想在于尽可能提前进行 GC，而不要将其推迟到最后一刻。

后台 GC 机制的实现利用了 HoitFS 的多头链表结构。当后台 GC 线程检测到整个介质占用达到某个阈值 δ 时，便会进行后台 GC 操作。由于后台 GC 对系统实时性要求不高，因此我们将从全部三条链表中去选择待 GC 的 Sector，并且每次 GC 只回收一个有效节点，从而能够充分降低后台 GC 的性能开销；当系统到达不得不调用前台 GC 地步时，我们会迅速从 Dirty List 中取出一个 Sector 用以 GC，并且一次性回收完整整个 Sector，从而充分提高系统实时性，减少用户等待时间。图 3.3 为 HoitFS GC 机制示意图。

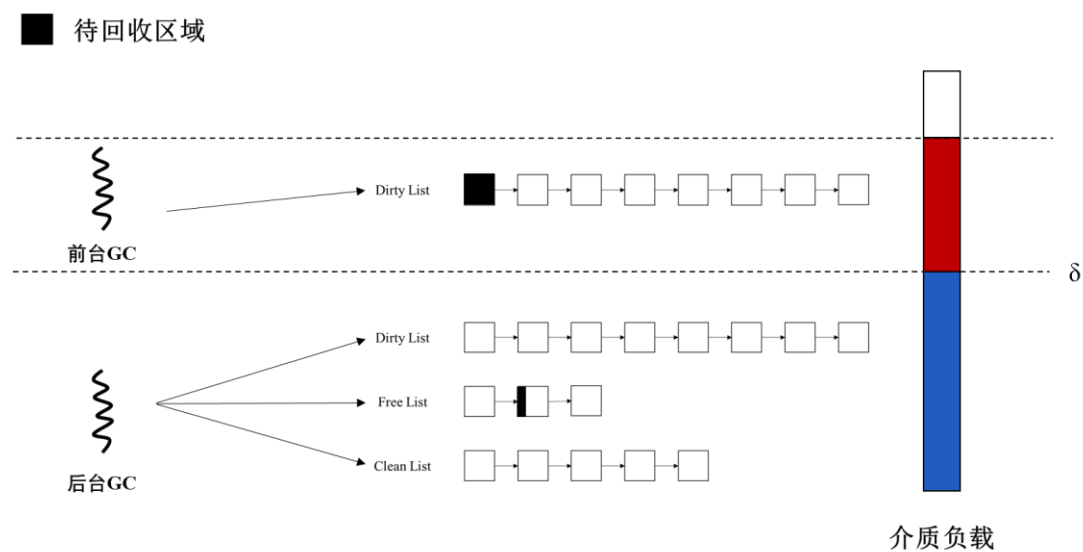


图 3.3 前后台 GC 示意图

3.3 Mergeable Tree

JFFS2 对小数据写入极不友好[7]，HoitFS 自然如此。我们来考虑如下场景：假设我们仅写入 1B 数据，我们需要为该数据创建相应的元数据头，如图 2.1 所示。可以发现，元数据头大小：数据体大小 = 32B:1B，也就是说有效数据率仅

在 3.03%，这种设计显然是不合理的。此外，由于 HoitFS 的内存索引结构很大程度上依赖于红黑树，当如是小节点增多，树的高度必然会不断增长，查找效率自然降低，但内存开销还在不断增大，由此可见优化小数据写入的必要性。Mergeable Tree 的核心思想便是利用某种机制将这些小数据节点合并，从而达到减少红黑树节点个数，减少内存开销，增大有效数据利用率的目的。

Mergeable Tree 的难点在于我们不能简单地设计一个 Write Buffer 来缓存上层向下层的小数据写入，从而达到合并节点的目的，因为正如我们在 2.1.3 节中介绍的那样，逻辑节点间会出现覆盖现象，而红黑树则需要去处理这一覆盖，从而保证逻辑节点的一致性。但引入 Write Buffer 势必会造成 Write Buffer 与红黑树的不一致性，这在无形中扩大了问题的难度。

我们提出了一种折衷办法，其核心思想在于：我们允许小数据节点的插入，但当红黑树上出现的小节点数高达某个阈值时，我们便会尝试合并这些节点，并重新写入合并后的节点，从而达到小数据节点合并的目的。

我们通过 Merge Buffer 结构来记录红黑树上的小数据节点。Merge Buffer 是一个固定长度的双向链表结构，其中，每个节点都会保存一个指向红黑树小节点的指针，当 Merge Buffer 被占满后，便会扫描其上记录的所有节点，并尝试合并那些逻辑上连续的节点。

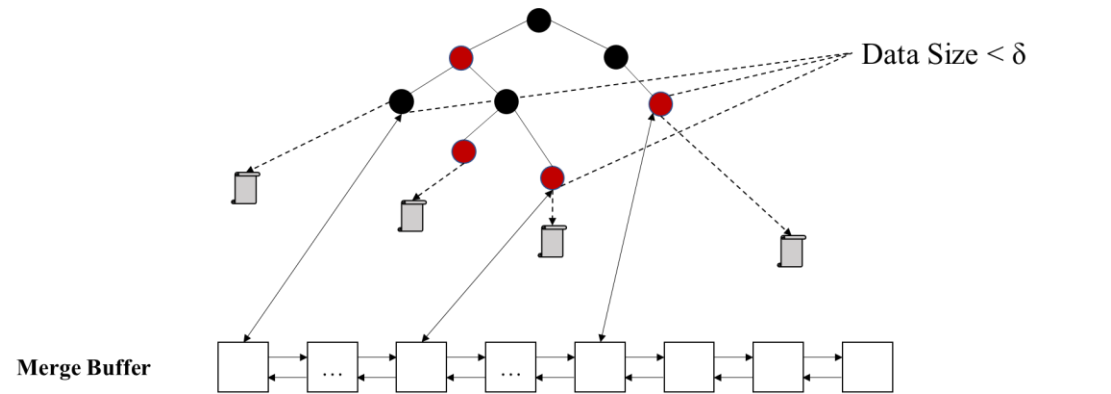


图 3.4 Mergeable Tree 结构

值得说明的是，Mergeable Tree 的存在会在小数据写入时带来较为强烈的内存抖动。假设小节点是数据大小不超过 δ 的节点，而每个红黑树节点占用 32B 内存，再假设 Merge Buffer 可容纳 1024 个小节点，那么 1 次小数据合并便能够减少 32KB 的内存开销。紧接着内存又会逐渐增大 32KB，触发 Merge 内存开销又

将减少 32KB.....如此往复，内存开销便会像锯齿一样不断抖动，这将在 4.2 节的实验现象中看到。

4. HoitFS 实验测试

由于 HoitFS 运行在 SylixOS 上，因此我们并没有像在 Linux 中有着趁手的工具来进行性能测试，我们只能自行设计开发 test bench。此外，由于 SylixOS 没有其他支持 Norflash 的文件系统，为了对比 HoitFS 的性能，我们不得不再移植一个 Norflash 文件系统，也就是 1.2 节中介绍的 SpifFS 文件系统了。

整个测试的思路非常简单。首先，需要测试 HoitFS 的基本读写性能，在保证功能正确性的同时验证红黑树性能的优越性。其次，需要测试 HoitFS 的三种优化效果：其一，测试 EBS 对挂载时间的优化；其二，测试 Background GC 是否真的能够提高文件系统在负载相同时写入带宽以及减少前台 GC 次数；其三，测试 Mergeable Tree 对内存开销的优化，以及测试 Mergeable Tree 的设计是否会对正常文件系统 IO 有着较大的影响。

4.1 实验准备

4.1.1 环境准备

- SylixOS 内核版本：1.12.9
- Mini2440 开发板：S3C2440A，ARM920T 405/101MHz
- Norflash 芯片：Am29LV160DB，2MB

4.1.2 实验策略

HoitFS 与 SpifFS 是两种设计目标不同的 Norflash 文件系统，前者在关注性能的同时尽可能减少内存开销，后者在尽可能降低内存开销的同时需要保证性能的均衡。

为了尽可能保证测试的公平性，我们设置测试最小 IO 按 SpifFS 页面对齐，即 256B。此外由于 SpifFS 与 HoitFS 均有 Cache 机制，为了尽可能减少 Cache 带

来的不公，我们将二者 Cache 均设置为 8 个 Sector 的大小。

根据前文所述的测试思路，我们制定如下测试策略：

● 基本读写测试

文件系统的基本读写需要测量四个标准指标：① 顺序读带宽；② 随机读带宽；③ 顺序写带宽；④ 随机写带宽。

其中**顺序读**和**随机读**测试方法相似：首先创建一个文件，并向其写入占介质容量 S%的数据，然后每次读一个最小 IO（256B）。对于顺序读，我们要求顺序读完所有 S%的文件数据；而对于随机读，我们要求随机读 $10 * S\%$ 的文件数据。这样设计的根本原因是：我们需要保证顺序读无需调整文件指针，因此读完写入数据就算结束；而随机读则需要不断调整文件指针，我们需要保证文件指针的所有位置尽可能多地覆盖整个文件，因此待读取数据量设置为 10 倍于文件数据大小。

顺序写测试方法较为简单：创建一个文件，并向其写入占介质容量 S%的数据，要注意这个 S%要设计得格外小心，我们需要避免 GC 机制触发对测试结果产生干扰。

随机写与顺序写测试方法略有不同，因为 HoitFS 目前还暂未支持文件空洞（Hole）机制，因此当一个文件内没有内容时，我们不能随机移动文件指针对其进行写入。因此，在开始测试随机写之前我们同样需要创建一个文件，并向其中写入占介质容量 S%的数据，接着我们便在这个文件范围内随机写入 N 次。这里的 S%的选择同样较为考究，由于 HoitFS 的异地更新机制，一次随机写入有可能带来好几倍的介质空间开销。考虑这种情况：文件内存在节点 A [0,4095]，此时我们向该文件中再次写入[0,255]，此时节点 A 的部分数据被新写入数据覆盖，因此节点 A 过期，我们需要再次写入一个新的节点 A' [256,4095]。这就意味着写入 256B 的数据需要额外写入 3840B 的数据，这将导致占用空间以极快的速度扩张。为了避免 GC 触发对结果造成影响，我们必须小心翼翼地估算 S%的大小。

● 挂载测试

挂载时主要测试有效节点个数对挂载时间的影响。例如，当 HoitFS 维护了占用介质 50%的文件数据时，挂载平均时间是多少，当这个值再增大又是多少。因

此，总结挂载测试的基本步骤为：① 首先创建一个文件，并向其中写入占介质容量 S% 的数据；② 关闭文件；③ 取消挂载；④ 记录开始时间；⑤ 挂载；⑥ 计算本次挂载消耗时间；⑦ 执行步骤③，直至到达测试次数要求。

● Background GC 测试

GC 性能可以通过测试在文件系统负载开销较大时写入性能来进行评估。同时，对于 HoiFS 而言，我们还可统计前台 GC 的次数，从而直观地说明后台 GC 带来的优势。

主要测试步骤为：① 首先创建一个文件，并向其中写入占介质容量 S% 的数据；② 关闭文件；③ 删除文件；④ 再次创建一个文件，并向其中写入占介质容量 M% 的数据。我们需要记录步骤④的写入带宽。

● Mergeable Tree 测试

Mergeable Tree 设计的优势在于既可以提升小数据写入的性能，又能够减少内存开销，因此针对 Mergeable Tree 测试我们需要记录两种数据：① 小数据写入带宽；② 内存开销。

Mergeable Tree 的测试思路非常简单，我们只需创建一个文件，并向其中输出 N 次大小为 S 的小数据即可。

4.2 实验结果

4.2.1 基本读写测试结果

为了方便观察，我们把对 Mergeable Tree 的性能测试合并到基本读写测试中，这样做的好处在于能够说明引入 Mergeable Tree 后对基本性能的影响。在本次测试中，我们设置小数据阈值为 16B，Merge Buffer 大小也为 16，其中各个测试点的参数如下：

● 随机读、顺序读

S% = 60%;

● 顺序写

S% = 60%;

● 随机写

$S\% = 20\%$, $N = 1000$ 。事实上,介质的 20%大概是 419,430B 左右, HoitFS 会将这 419,430B 拆分成若干个最大长度为 4096B 的节点,按最坏情况来算,每次随机写入 256B 都会额外引入 4096B 的介质开销,那么在该参数下随机写会造成空间开销估算为: $1000 * 4,096 + 419,430 = 4,515,430\text{B}$, 即 4MB 左右。实际情况会因为写入位置的随机而有所不同,会远小于 4MB,我们同时还需要考虑 GC 因素的影响,经测试,参数 $S\% = 20\%$, $N = 1000$ 是合理的。

● 小数据写入

$S = 1$, $N = 10000$;

读写测试结果如图 4.1 所示。

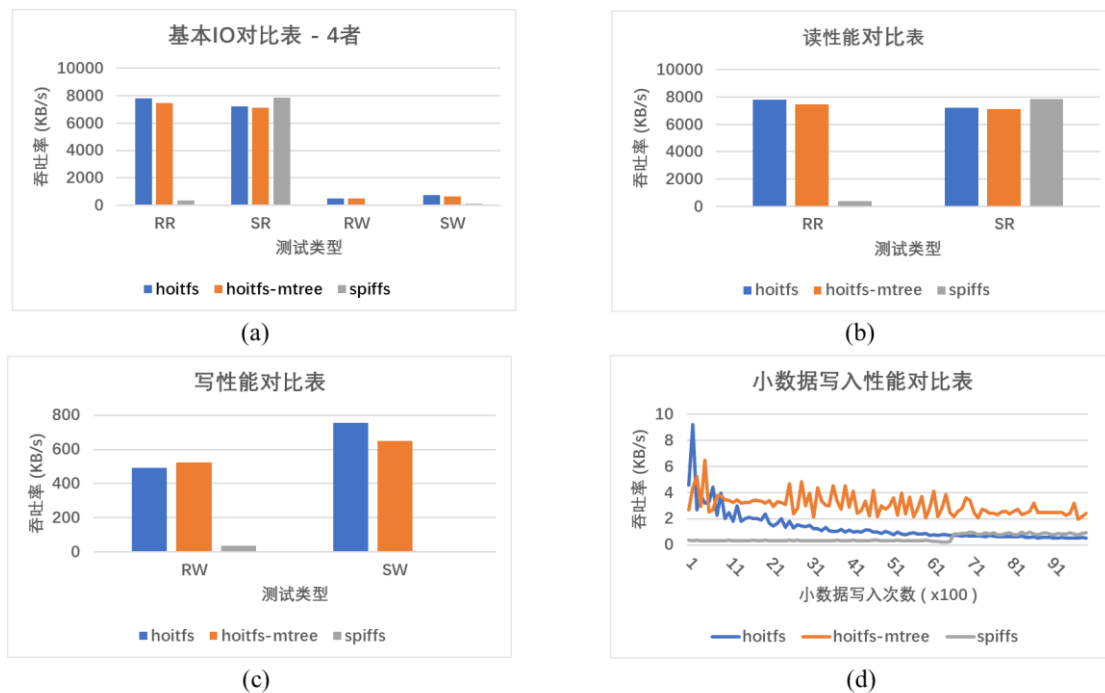


图 4.1 读写性能测试 (mtree 代表 mergeable tree)

图 4.1 (a)为随机基本 IO 测试结果表总览,可以看到,各文件系统的读带宽都远高于写带宽。

图 4.1 (b)为读性能表, HoiFS 的带宽在 7.5MB/s 到 8MB/s 左右,基本与表 1.1 的 Norflash 性能参数相符;而 SpifFS 在顺序读上能够达到 8MB/s 的带宽,甚至比具有红黑树结构的 HoiFS 性能更优,但是其随机读性能实在不敢恭维。出

现这种结果的主要原因在于：① SpifFS 顺序读快，是因为在测试顺序读前我们会先创建文件，该文件的数据块都被缓存在了 Cache 中，而顺序读不会涉及到 Index 页面反复切换，因此顺序读的开销基本就是查找 Index 位图的开销加上访问 Cache 的开销，也就是 $O(1)$ 的时间复杂度；而 HoiTFS 尽管在内存中维护了一个巨大的红黑树结构，但每次读都会从树根开始向下寻找，时间复杂度为 $O(\log n)$ ，当文件增大，红黑树上节点增多，顺序读时间势必还会降低，这是 HoiTFS 相较于 SpifFS 的弊端；② SpifFS 随机读慢，是因为上层结构只会缓存一个 Index 页面，如图 1.4 所示，但是文件指针不断变化，该 Index 页面会出现不断换入换出的情况，而每次换入时，都需要重新扫描整个介质，找到当前文件指针对应的 Index 页面，这便是 SpifFS 随机读慢的根本原因。

图 4.1 (c) 为写性能表，HoiTFS 的写入带宽在 400KB/s 到 700KB/s 之间，这与 Am29LV160DB 芯片性能参数基本一致，体现了红黑树的优越性，且加入了 Mergeable Tree 机制后对写性能的影响也处在可接受的范围内。反之，SpifFS 在写入性能方面非常差，这是因为：① 顺序写时，SpifFS Cache 全部未命中，而 SpifFS 的 Cache 策略是写不分配，这意味着必须 SpifFS 的写操作必须直接对介质进行操作，这显然导致了性能瓶颈；② 随机写时，与随机读类似，Index 页面必须反复换入换出，介质不断被扫描，这造成随机写性能的低下。

我们可以看到，SpifFS 的综合性能不如 HoiTFS，但事实上，HoiTFS 的性能收益很大程度上是因为它庞大的内存索引结构，图 4.2 显示了各文件系统在读写中索引结构产生的内存开销。

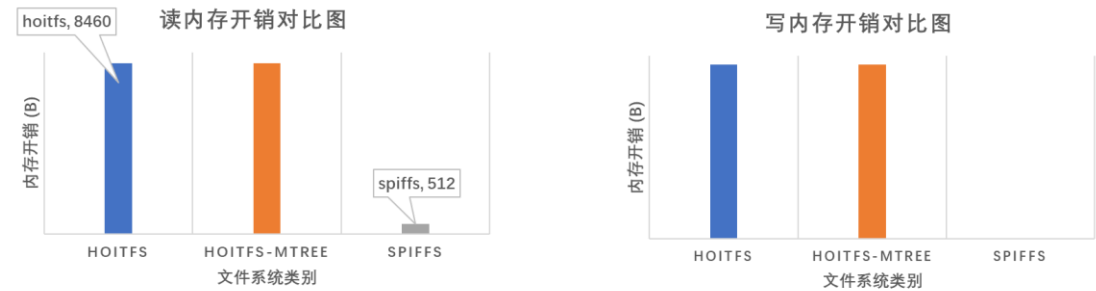


图 4.2 读写索引结构内存开销

可以看到，HoiTFS 的内存开销（8460B）高出 SpifFS（512B）几个层级，事实上，当介质增大，这种问题会愈加严重。不过现代搭载商用 Norflash 的开发板至少也

有 64MB 的内存，因此在系统正常的情况下，我们能够接受 HoitFS 的内存开销。

最后，图 4.1 (d)为小数据写入性能表，可以看到加入了 Mergeable Tree 的 HoitFS 在写入性能上明显高于普通 HoitFS 与 SpifFS，但性能曲线上有明显的尖峰抖动，这与我们的预测结果保持一致，因为 Mergeable Tree 的节点将不断经历从少到多然后突然减少的过程：当树的节点突然减少后，写入性能将迎来新一轮的提高；当树的节点不断增多的过程中，写入性能又将慢慢下降直至出现下一次尖峰。

4.2.2 挂载测试结果

我们共测试了三种情况下的挂载，它们分别是：① S% = 55%；② S% = 65%；③ S% = 85%。测试结果如图 4.3 所示。

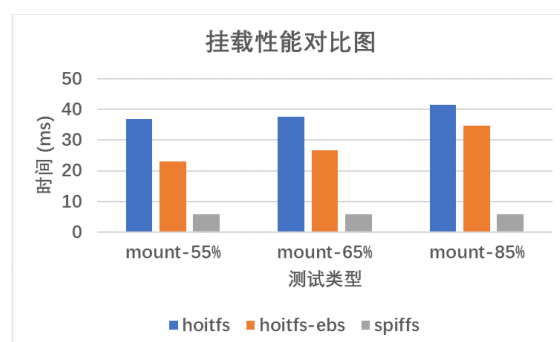


图 4.3 挂载性能对比图

从总体趋势上来看，各文件系统的挂载时间开销都随文件系统需要维护的数据量增加而增加。但 HoitFS 的挂载性能明显不如 SpifFS，这是因为 SpifFS 的索引结构保存在介质上，挂载时，SpifFS 无需知道每个页面的具体信息，只需要知道各个页面是否有效即可；但 HoitFS 的索引保存在内存中，我们必须在挂载时扫描介质上的所有节点，从而建立起相应的物理层结构。也就是说，我们不仅要知道节点存在，而且需要知道该节点是什么，上面的内容是什么，因此，这导致 HoitFS 总体上来看性能低于 SpifFS。

再来看启用了 EBS 后的 HoitFS 和未启用 EBS 的 HoitFS。可见，启用 EBS 后，HoitFS 的挂载时间有了明显的提升，提升空间在 10%~50%左右。这验证了 EBS + MT 机制的正确性，但是我们观察到一个比较奇怪的情况：当数据量不断增大，EBS + MT 的提升逐渐变得没有那么明显，这在很大程度上是因为每个

Sector 都有大量节点需要处理，EBS 扫描线程不能得到即时的释放，导致线程间严重竞争，因此才会出现上述情景。

4.2.3 Background GC 测试结果

我们同样设置了三种测试情况，它们参数分别为：① $S\% = 50\%$ ， $M\% = 30\%$ ；② $S\% = 60\%$ ， $M\% = 20\%$ ；③ $S\% = 70\%$ ， $M\% = 10\%$ ，对于这三种情况的测试结果如图 4.4 所示。

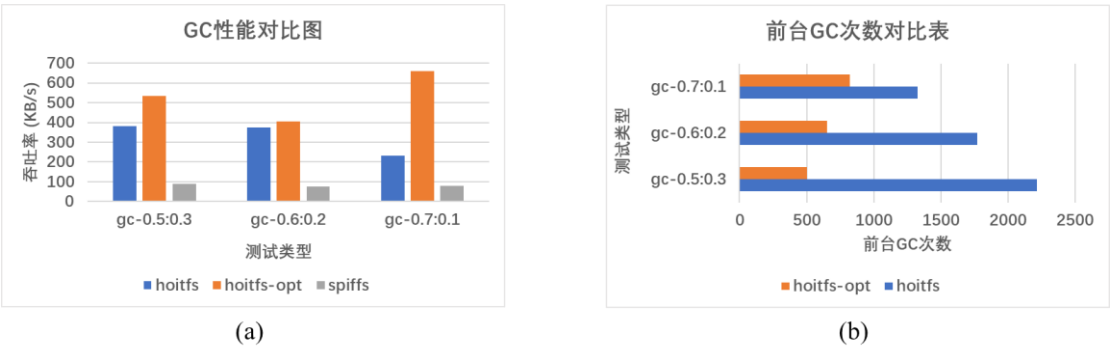


图 4.4 GC 测试结果

图 4.4 (a)为触发 GC 时写入性能对比图，可以看到 HoitFS 还是一如既往领先于 SpifFS，这是因为 SpifFS 本来写入性能就较低，此时触发 GC 后，一次写会导致额外的擦写，因此 SpifFS 在触发 GC 时性能仍然较差。而反观 HoitFS，加入了 Background GC 后，触发 GC 时的写入性能明显有了提升，这是因为 Background GC 的存在降低了前台 GC 次数，使得写入性能提升。

图 4.4 (b)可视化了在这三种情况下 HoitFS 前台 GC 的次数，可以看到，失陪了 Background GC 后，前台 GC 次数有了明显的下降。这里需要说明为什么随着 $S\%$ 的增大，Background GC 能够减少的 GC 次数也有一定的下降：这是因为一开始我们需要写入 $S\%$ 的数据，此时很可能会触发前台 GC，而此时由于写得极为频繁和迅速，后台 GC 未来得及运行，因此，这部分前台 GC 便无法被减少，这也解释了图 4.4 (b)出现这种情况的原因了。

4.2.4 Mergeable Tree 测试结果

在 4.2.1 节中，我们对 Mergeable Tree 进行了性能测试，结果表明，Mergeable Tree 在对正常读写没有太大影响的同时，能够收获到小数据写入性能的大幅提升。本节主要对 Mergeable Tree 进行内存开销评估，测试参数与前一致，测试结果如图 4.5 所示。

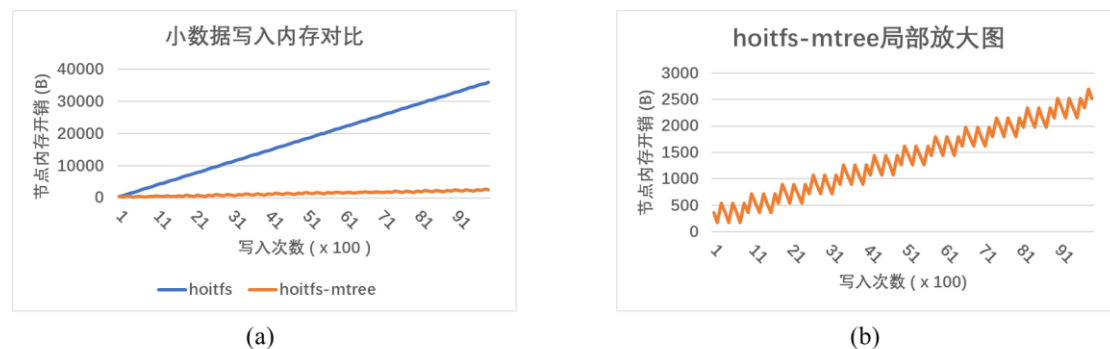


图 4.5 Mergeable Tree 内存开销示意图

图 4.5 (a)给出了 HoitFS 与加入了 Mergeable Tree 机制的 HoitFS 小数据写入内存对比，可见，未经优化的 HoitFS 内存上涨斜率远高于 Mergeable Tree HoitFS 说明 Mergeable Tree 的思路是可行的；接着，将 Mergeable Tree HoitFS 的内存开销放大来看，见图 4.5 (b)，可以看到明显的内存抖动，这也验证了我们的猜测。

5. 未来展望

从实验数据来看，HoitFS 在除了在顺序读以及挂载上不能击败 SpifFS，其他均能够完胜 SpifFS，这也与文件系统架构设计有关。我们在 HoitFS 的基础上进行了 EBS、Background GC 以及 Mergeable Tree 三种方案的优化，使得优化后的 HoitFS 相比优化前在内存开销、性能表现方面都有了一定的提升。本节主要提出一些我们在项目中头脑风暴过，但未来来得及予以实现的想法，它们或是针对 HoitFS 的内存开销进行优化，或是受到 SpifFS 的启发，就将这些想法记录下来，作为我们的未来展望。

5.1 冷热文件分离

还是回到内存开销大的问题上。Mergeable Tree 解决的主要是红黑树内存开销问题，但是我们知道 HoitFS 内存中由三层结构组成，红黑树只是逻辑管理层。事实上，物理层结构的内存开销也不可小觑。当物理介质上的节点继续增多后，物理层就需要更多的节点来管理这些物理节点，内存同样是线性增长。

冷热文件分离的提出主要是希望能够改善这一问题。该方法的提出是基于我们平日的观测，很多文件创建后我们便不再使用，例如一些 txt 文件、代码源文件等，既然这些文件都不怎么被使用，那么为何我们还要为它们建立起内存结构呢？基于这个观测，我们可以在文件系统运行过程中自行将文件分为冷、热两类，下次挂载时，我们只需要构建那些热文件的物理结构即可。如果实在要访问冷文件，那么我们便利用 EBS 去找到介质上的相应物理节点，重新为其构建物理层结构即可。冷热文件分离示意图如图 5.1 所示。

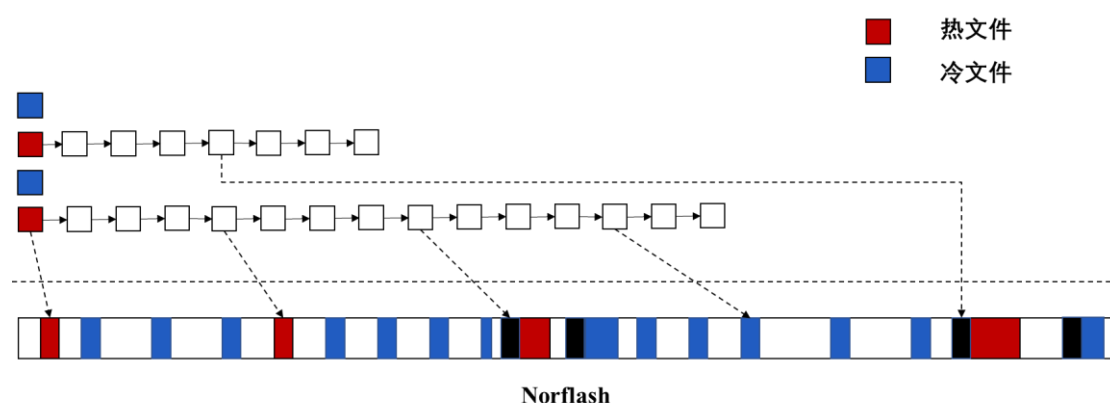


图 5.1 冷热文件分离构想图

5.2 虚拟文件描述符

该想法的出发点是因为当不同进程打开的文件愈多，内存中建立的红黑树就愈多，由此导致的内存开销就会急剧上升。基于这个问题，最简单的解决思路就是将可打开的文件限制在一定数量内，但是这种方式对上层应用并不友好，于是我们提出了一种虚拟文件描述符的方法：我们固定真正需要维护内存结构的文件数，其余打开的文件向上层提供虚拟文件描述符，只有在需要访问这些描述符时，我们才去构建该文件的红黑树，并利用 LRU 将一个维护了红黑树的文件换出。

这种想法与 Cache 机制比较相似。虚拟文件描述符示意图如图 5.2 所示。

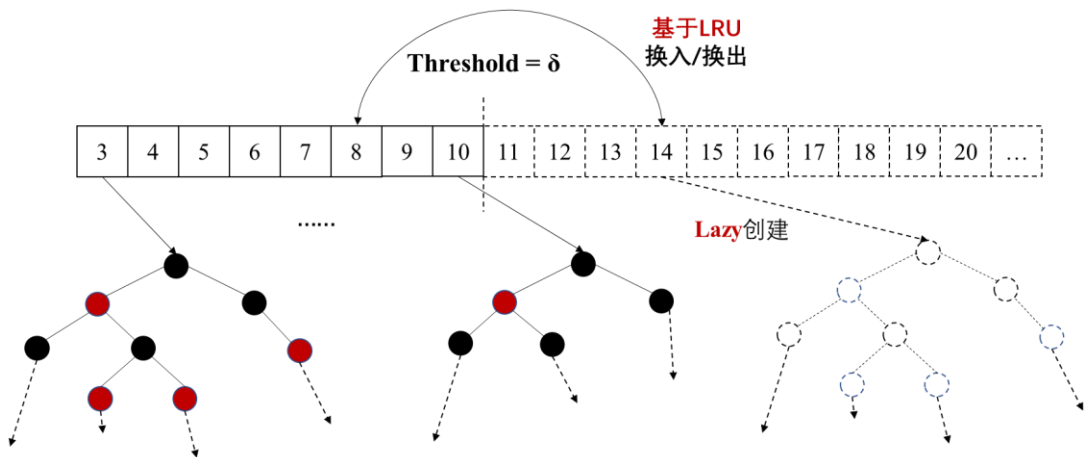


图 5.2 虚拟文件描述符构想图

5.3 受 SpifFS 启发的新型索引结构设计

从 4.2.1 节的基本 IO 测试结果来看，SpifFS 拥有性能极高的顺序读特性，这得益于 SpifFS 的位图索引。事实上，我们认为，如果 SpifFS 如果能够在内存中维护文件的所有 Index 页，那么其读写性能将会得到质的飞跃，关于这一点，由于时间原因我们并没有通过实验进行验证。既然位图如此优秀，那么我们可以考虑一种将红黑树索引更换为位图索引的方法，但碍于时间原因，我们未来得及做更加深入的思考。

6. 回顾与总结

HoitFS 项目自 12 月 31 日启程，到 8 月 18 日结束，历时近 1 年。这个过程给我们带来的不仅是能力的飞速增长，更是意志与毅力的锤炼。

在 HoitFS 开发过程中，我们遇到了很多难题：首先是对硬件的不熟悉，在捣鼓 mini2440 这样远古的开发板过程中，资料稀缺、环境古老是致命伤，期间还遇到过硬件损坏等种种问题，最终在不断询问、查阅中，捣鼓明白了 mini2440，并使我们具有了在 SylixOS 上访问 Norflash 的能力；再来就是对 SylixOS 不熟悉，我们花了近三周的时间去理解 SylixOS 的文件系统接入，以及相关函数流程，最后我们通过 SylixOS 上完成简单的 LFS 实验打通了整套 SylixOS 文件系统

自顶向下的适配流程；其三就是对 Norflash 文件系统不熟悉，在研究 SylixOS 文件系统接入的同时，我们也花费了大量时间探索 JFFS、JFFS2 文件系统的结构及其设计精髓，最终精炼出物理层、逻辑层、逻辑管理层，并据此自行实现完成了 HoitFS；最后，开发中的代码调试、团队间的沟通配合、团队的士气、队员的时间规划（6、7 月份时间非常不充分：考研的准备考研、实习的要实习、保研的准备保研）都出现过大大小小的问题，不过好在最后大家还是咬牙坚持下来了，这里，要感谢 HoitFS 团队的每个成员。

回顾这近一年的成果，虽然遇到过许多困难，我们也尽了自己的最大努力，但是我们仍然认为在比赛中还有很多地方有值得改进：首先便是性能量化意识远不够。在定题的那一天，我们认为最终的交付就是一个工程项目、一个能用的文件系统，但事实上，这远不够。正如导师说的那样，虽然我们做了很多优秀的工作，但是没有去量化工作的优越性，这等于空谈。我们从 6 月中旬开始才意识到给出一个量化测试结果的重要性，也正是从那个时候开始才想到去再移植 SpiffFS，进而去做性能对比测试。事实上，这个工作起步就晚了，导致最后时间略显仓促；再来是对代码风格的约束没有做到严格把控，导致有些代码异常丑陋，如果还有下次机会，一定要实现代码门禁工具，不予不合格代码入库的机会；最后便是对任务完成度没有严格的要求，有些被安排下去的任务并没能如期完成，这种情况需要队长好好考虑，到底是管理不力还是执行者能力不足，下次该如何改进，如果重头再来，我们相信我们能够做得更好。

最后，感谢 HoitFS 全体成员，感谢夏文老师、江仲鸣老师悉心指点，感谢陈洪邦、蒋太金老师给予我们项目上的技术支持。是你们的鼓励与支持，让 HoitFS 走到最后，为这场竞赛画上了圆满的句号。

7. 参考文献

- [1]. Nandflash 和 Norflash 在 ARM9 中的地位与连线方案. (n.d.). 豆丁网. Retrieved August 18, 2021, from <https://www.docin.com/p-533845007.html>
- [2]. Am29LV160D Datasheet [M] AMD, 2000
- [3]. W25Q256JV Datasheet [M] Winbond, 2021
- [4]. Huawei LiteOS – 轻量级物联网操作系统. (n.d.). LiteOS. Retrieved August 18, 2021, from <https://www.huawei.com/minisite/liteos/en/about.html>
- [5]. P., & P. (n.d.). spiffs/TECH_SPEC at master · pellepl/spiffs. GitHub. Retrieved August 18, 2021, from https://github.com/pellepl/spiffs/blob/master/docs/TECH_SPEC
- [6]. Yanqi, P. (n.d.). HoitFS 设计开发文档. GitLab. Retrieved August 18, 2021, from <https://gitlab.eduxiji.net/Deadpool/project325618-47064/>
- [7]. Woodhouse D. JFFS: The jouralling flash file system[C]. Ottawa linux symposium. 2001, 2001.