



西安邮电大学  
XI' AN UNIVERSITY OF POSTS & TELECOMMUNICATIONS

## 项目：bt-fuse

团队	萤火虫
项目成员	尚凡、杨传江、谢佳月
年级	研一

指导导师：郑昱笙  
校内老师：陈莉君

2024. 8. 15

# 目录

一、前言 .....	2
二、fuse 性能优化 .....	2
2.1 传统 fuse 介绍 .....	2
2.2 目前已有对 fuse 的优化 .....	3
2.2.1 extfuse 介绍 .....	3
2.3 bt-fuse 介绍 .....	6
2.3.1 数据结构 .....	7
2.3.2 bt-fuse 接口实现 .....	11
2.3.3 bt-fuse 优化思路 .....	12
2.4 extfuse 与 bt-fuse 比较总结 .....	12
三、技术框架 .....	14
3.1 技术框架介绍 .....	14
3.1.1 bt-fuse 执行流程分析 .....	14
3.1.2 bt-fuse 架构 .....	14
3.2 ebpf 截断系统调用 .....	15
3.2.1 kprobe 截断系统调用 .....	15
3.2.2 bpftime 截断系统调用 .....	17
3.3 bt-fuse 文件系统的实现 .....	19
3.3.1 bt-fuse 核心逻辑实现 .....	19
3.3.2 bt-fuse 运行结果 .....	22
3.4 进程通信-共享内存 .....	23
3.4.1 进程通信 .....	24
3.4.2 共享内存 .....	24
3.4.3 bt-fuse 文件系统使用共享内存 .....	30
3.4.4 原有 fuse 文件系统和 bt-fuse 通信比较分析 .....	33
四、性能测试和监控 .....	34
4.1 Grafana 和 Prometheus 性能监控 .....	34
4.2 性能测试 .....	34
五、总结与展望 .....	36
5.1 bt-fuse 未来的发展前景 .....	36
5.2 bt-fuse 缺陷 .....	36
5.3 bt-fuse 设计遇到的问题 .....	37

## 一、前言

原有 fuse 文件系统在用户空间运行，这使得开发者可以快速原型化和修改文件系统逻辑，而不需要修改内核代码。这种灵活性降低了开发复杂性和风险。

但是原有 fuse 文件系统由于在用户空间运行，每次文件操作都需要在进行用户空间与内核空间之间的上下文切换，这会引入额外的性能损失和延迟，特别是在高负载和高并发情况下表现得更为明显。此外，用户空间的处理会导致更高的内存和 cpu 使用，从而影响系统的整体性能。

目前已有的文件系统除了 Dfuse、并行日志文件系统之外，还未有新的方案框架提出。

但是本项目提出新方案，bt-fuse 使用 bpftime 在用户态将系统调用截断，从而减少系统调用的开销，进而通过共享内存，把数据请求直接转发到 fuse 文件系统。通过这样的方式，减少系统调用、进程上下文切换的开销。

## 二、fuse 性能优化

### 2.1 传统 fuse 介绍

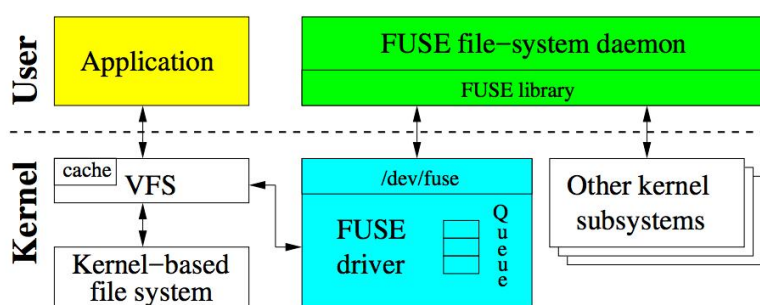


图 1 fuse 架构设计图

Fuse 包含两个部分，kernel 和用户态 daemon。内核部分是一个 Linux 内核模块，它会在 Linux 的 VFS 层上面注册一个 Fuse 的文件系统驱动。这个 Fuse 驱动可以认为是一个 proxy，它会将请求转发到后面的用户态 daemon 上面。

Fuse 内核模块也会注册一个 /dev/fuse 的块设备，这个就是 kernel 和用户态 daemon 交互的接口。通常 Daemon 会从 /dev/fuse 上面读取到 Fuse 的请求，

处理并且将数据写回到/dev/fuse。

一个简单的 Fuse 流程如下：

- 1、应用程序在挂载的 Fuse 的文件系统上面进行操作。
- 2、VFS 会将操作转发到 Fuse 的 Kernel Driver 上面。
- 3、Fuse 的 kernel driver 分配一个 request，并且将这个 request 提交到 Fuse 的 queue 上面。
- 4、Fuse 的用户态 daemon 会从 queue 里面通过/dev/fuse 将这个 request 取出来并且处理。这里，处理 request 的时候仍然可能进入这个 kernel，譬如说可能将 request 发送到 Ext4 去实际处理。
- 5、当请求处理完毕，Daemon 会将结果回写到/dev/fuse。
- 6、Fuse 的 Kernel 标记这个 request 结束，然后唤醒用户应用程序。

## 2.2 目前已有对 fuse 的优化

Extfuse	Dfuse	并行日志文件系统
提供对'ext4'文件系统的增强支持	提供了对分布式文件系统的高效支持	通过日志结构化存储来优化写入性能
优化了文件系统性能，通过减少系统调用次数和提高缓存效率来减少 I/O 开销	支持透明的分布式文件系统操作，并通过优化网络 I/O 和缓存来提升性能	在并发环境下提供高效性能的日志记录和读取性能,适用于大规模并行计算任务

表 1 fuse 优化分析表

因为 bt-fuse 文件系统的设计是利用 bpftime 工具和原有 fuse 文件系统框架设计实现。而 extfuse 也是利用 ebpf 程序以及原有 fuse 文件系统框架进行设计，我们这里的对比主要是利用 extfuse 文件系统与 bt-fuse 文件系统进行对比。

### 2.2.1 extfuse 介绍

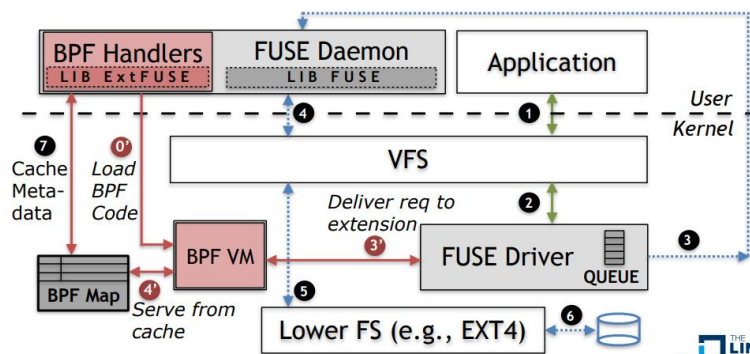


图 2 extfuse 设计框架图

ExtFUSE 的实现框架图如上所示，它由三个核心组件启用，即内核文件系统（驱动程序）、用户库（libExtFUSE）和内核内 eBPF 虚拟机运行时（VM）。

ExtFUSE 驱动程序使用插入技术在低级文件系统操作中与 FUSE 兼容。然而，与 FUSE 驱动程序只是将文件系统请求转发到用户空间不同，ExtFUSE 驱动程序能够直接将请求传递到内核处理程序（扩展）。它还可以将一些受限的请求集（例如读、写）转发到主机（下）文件系统（如果存在的话）。后者对于在主机文件系统之上添加精简功能的可堆叠用户文件系统是必需的。libExtFUSE 导出一组 api 和抽象，用于在内核中服务请求，隐藏低层实现细节。

libExtFUSE 的使用是可选的，独立于 libfuse。向 libfuse 注册的现有文件系统处理程序驻留在用户空间中。因此，我们将它们的执行称为慢路径。使用 ExtFUSE，用户空间还可以注册内核扩展，当从 VFS 接收到文件系统请求时立即调用这些扩展，以便允许在内核中提供这些扩展。我们将内核执行称为快速路径。根据快速路径的返回值，可以将请求标记为已服务，或者通过慢路径将请求发送到用户空间守护进程，以便根据需要进行复杂的处理。快速路径还可以返回一个特殊值，指示 ExtFUSE 驱动程序插入并将请求转发到下层文件系统。但是，此特性仅适用于可堆叠的用户文件系统，并且在内核中加载扩展时进行验证。

### ExtFUSE 的扩展核心机制：eBPF map 通信

eBPF map 是 eBPF（扩展伯克利数据包过滤器）框架中用于存储和检索数据的内存数据结构。eBPF 是一种技术，允许在内核中安全地执行用户定义的代码，而 map 是 eBPF 运行时环境的关键组成部分。eBPF map 可以被视为键值存储。它们用于在内核和用户空间中运行的 eBPF 程序之间传递数据，以及在不同的 eBPF 程序之间传递数据。可以使用 BPF 系统调用创建和操作映射。eBPF map 是内核应用程序开发人员的强大工具。它们允许在内核和用户空间之间以及不同的 eBPF 程序之间进行高效和安全的数据共享。

### ExtFUSE 的工作流程

在装入用户文件系统后，FUSE 驱动程序会向用户空间守护进程发送 FUSE\_INIT 请求。此时，用户守护进程内通过在请求参数中查找 FUSE\_CAP\_EXTFUSE 标志来检查操作系统内核是否支持 ExtFUSE 框架。如果受到支持，守护进程必须调用 libExtFUSE init API 来将包含专门处理程序（扩展）

的 eBPF 程序加载到内核中，并向 ExtFUSE 驱动程序注册它们。这是通过使用 `bpf_load_prog` 系统调用来实现的，它会调用 eBPF 验证器来检查扩展的完整性。如果失败，该程序将被丢弃，并通知用户空间守护进程出现错误。然后，守护进程可以退出或继续使用默认的 FUSE 功能。如果验证步骤成功，并且启用了 JIT 引擎，则 JIT 编译器将处理扩展，以生成根据需要准备执行的机器装配代码。

扩展被安装在一个 `bpf_prog_type` 映射（称为扩展映射）中，它可以有效地充当一个跳转表。为了调用扩展，FUSE 驱动程序只需使用 FUSE 操作代码（例如，`FUSE_OPEN`）作为扩展映射的索引来执行 `bpf_tail_call`（跳远）。一旦加载了 eBPF 程序，守护进程必须通过 `reply` 包含到扩展映射的标识符的 `FUSE_INIT` 来通知 ExtFUSE 驱动程序有关内核扩展。

一旦收到通知，ExtFUSE 就可以在运行时在 eBPF VM 环境下安全地加载和执行扩展。每个请求首先传递到快速路径，快速路径可以决定（1）服务它（例如，使用在快速路径和慢路径之间共享的数据），（2）将请求传递到较低的文件系统（例如，在修改参数或执行访问检查之后），或（3）根据需要采用慢路径并将请求传递到用户空间进行复杂的处理逻辑（例如，数据加密）。由于执行路径是按请求独立选择的，并且总是首先调用快速路径，因此内核扩展和用户守护进程可以协同工作，同步对请求和共享数据结构的访问。需要注意的是，ExtFUSE 驱动程序仅充当 FUSE 驱动程序和内核扩展之间的薄插入层，在某些情况下，还充当 FUSE 驱动程序和低层文件系统之间的薄插入层。因此，它不执行任何 I/O 操作，也不尝试自己为请求提供服务。

### ExtFUSE 框架的具体实现

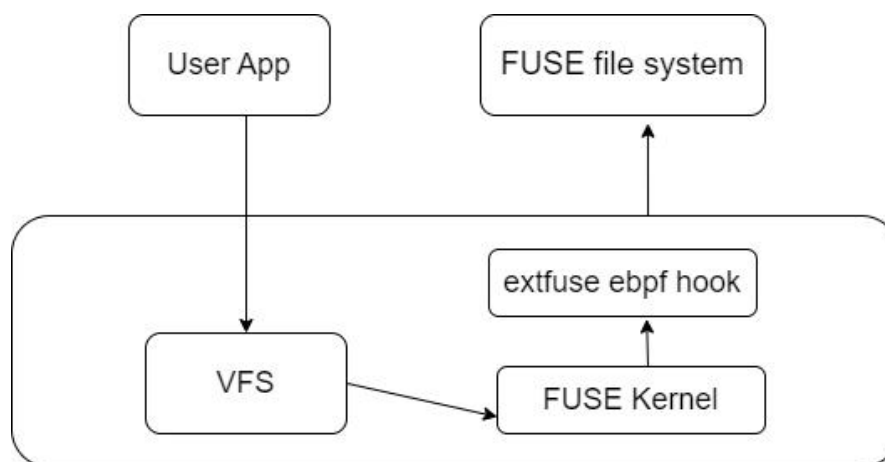


图 3 extfuse 实现原理

ExtFUSE 的 eBPF 框架分为 4 个部分：

- 1) linux 内核增加 ExtFUSE 的 eBPF 的程序类型；
- 2) 在 FUSE 的内核中增加 ebpf 的挂载点以及相应的钩子函数，并增加辅助函数；
- 3) 设计相应的 eBPF 挂载函数；
- 4) 在用户态文件系统建立与内核共同使用的 eBPF map。并在相关的元数据操作中维护 eBPF map。

### ExtFUSE 框架的优势和劣势

使用 eBPF 函数可以降低元数据操作使用用户态文件系统接口的频率，如果能根据文件系统或者应用的操作提前预加载相关的元数据确实能降低用户态文件系统的元数据开销，提升文件操作的性能。

存在的问题是需要对 FUSE 的内核做改动以及 linux 的内核 bpf 部分做相应的修改，优化场景会有比较的限制。

### ExtFUSE 总结

ExtFUSE 框架提供了优化 FUSE 文件系统开销的思路，通过使用 eBPF 提供的 hash map 建立了 inode cache，降低了元数据操作的开销，用户态文件系统可以通过统计以及预测 app 的行为来预加载 inode cache 到 VFS 中，从而提升了文件系统的元数据操作性能。用户态文件系统需要改动的地方也不多，主要是维护用户态与内核态共享的 eBPF hash map(用于存放 inode cache)。

## 2.3 bt-fuse 介绍

bt-fuse 文件系统是基于 bpftime、fuse、共享内存协同开发的一款文件系统。

bt-fuse 目录：

inode.h: 包含用于 bt-fuse 文件系统设计的 inode 管理相关的数据结构和定义，这个头文件主要定义了与文件系统中 inode（索引节点）相关的结构体和管理方式。

memory.h: 包含用于文件数据块管理的数据结构和定义。这个头文件主要定义了内存池以及管理的数据块结构体，以及数据块的分配和释放。

fuse\_example.c: 实现了 Fuse 文件系统的具体代码示例，该文件包含了 Fuse

文件系统的核心功能实现，包括文件和目录操作的处理逻辑，以及如何将这些操作与 Fuse 框架进行集成。

inode.c: 实现了 inode 管理相关的功能函数，并且提供了 fuse\_example.c 文件系统实现代码所需的 inode 操作接口。该文件包含了与 inode 相关的管理和操作逻辑，包括创建、查找、更新、删除 inode 等功能。

memory.c: 实现了内存池的管理功能，并且在 fuse\_example.c 中进程初始化，包括内存池的初始化、数据块的分配和释放。该文件提供了内存池的创建和管理逻辑，以支持文件系统的数据块操作和内存分配需求。

### 2.3.1 数据结构

inode.h:

```

1.  #define INODE_TABLE_SIZE 500
2.  #define HASH_TABLE_SIZE 715
3.  typedef struct dhmp_inode {
4.      char path[PATH_MAX]; // 完整路径
5.      ino_t ino;           // inode 编号
6.      mode_t mode;         // 文件类型和权限
7.      uid_t uid;           // 文件所有者
8.      gid_t gid;           // 文件组
9.      off_t size;          // 文件大小
10.     struct timespec atime; // 文件访问时间
11.     struct timespec mtime; // 文件修改时间
12.     struct timespec ctime; // 文件状态改变时间
13.     nlink_t nlink;         // 链接数
14.     struct dhmp_block *blocks; // 指向数据块链表
15.     pthread_mutex_t lock;
16.     struct dhmp_dir *dir_entries; // 指向目录项链表的头指针
17. } dhmp_inode_t;
18.
19. typedef struct dhmp_dir{
20.     char name[PATH_MAX];
21.     struct dhmp_inode *inode;
22.     struct dhmp_dir *next; // 下一个目录项的指针
23. }dhmp_dir_t;
24.
25. // 哈希表中的一个桶
26. typedef struct hash_entry{
27.     char name[PATH_MAX];

```



```

28.     dhmp_inode_t *inode;
29.     struct hash_entry *next;
30. }hash_entry_t;
31.
32. //哈希表的结构体
33. typedef struct hash_table{
34.     hash_entry_t **buckets;
35.     size_t bucket_count;
36.     pthread_mutex_t *bucket_mutexes;
37. }hash_table_t;
38.
39. //红黑树
40. typedef struct rb_node {
41.     char name[PATH_MAX];
42.     dhmp_inode_t *inode;
43.     struct rb_node *left;
44.     struct rb_node *right;
45.     struct rb_node *parent; // 添加 parent 指针
46.     int color; // 0 for black, 1 for red
47. } rb_node_t;
48.
49. typedef struct rb_tree {
50.     rb_node_t *root;
51.     pthread_mutex_t lock;
52. } rb_tree_t;
53.
54. typedef struct dhmp_superblock{
55.     struct dhmp_dir *root_directory;
56.     struct hash_table *hash_table; // 哈希表
57.     rb_tree_t *rb_tree; // 红黑树
58. }dhmp_superblock_t;
59.
60. typedef struct dhmp_file_system{
61.     struct dhmp_superblock *sb;
62. }dhmp_file_system_t;

```

dhmp\_file\_system\_t: 是整个文件系统的主结构体，包含一个指向 dhmp\_superblock\_t 的指针，是文件系统的入口点。

dhmp\_superblock\_t: 包含对文件系统的根目录、哈希表和红黑树的指针。用于管理文件系统的全局数据。

dhmp\_inode\_t: 表示对文件或目录的元数据，包括指向数据块和目录项链表的指针。

dhmp\_dir\_t:表示目录项, 包含名称和指向 inode 的指针。

hash\_table\_t:维护一个哈希表,用于快速查找 inode,桶包含 hash\_entry\_t。

hash\_entry\_t: 是哈希表中的条目, 包含文件名、指向 dhmp\_inode\_t 的指针和下一个条目的指针。

rb\_tree\_t:是用于存储 inode 的红黑树, 每个节点 (rb\_node\_t)存储 inode 的相关信息。

rb\_node\_t: 是红黑树的节点。

如下是 inode 管理部分的数据结构之间关系:

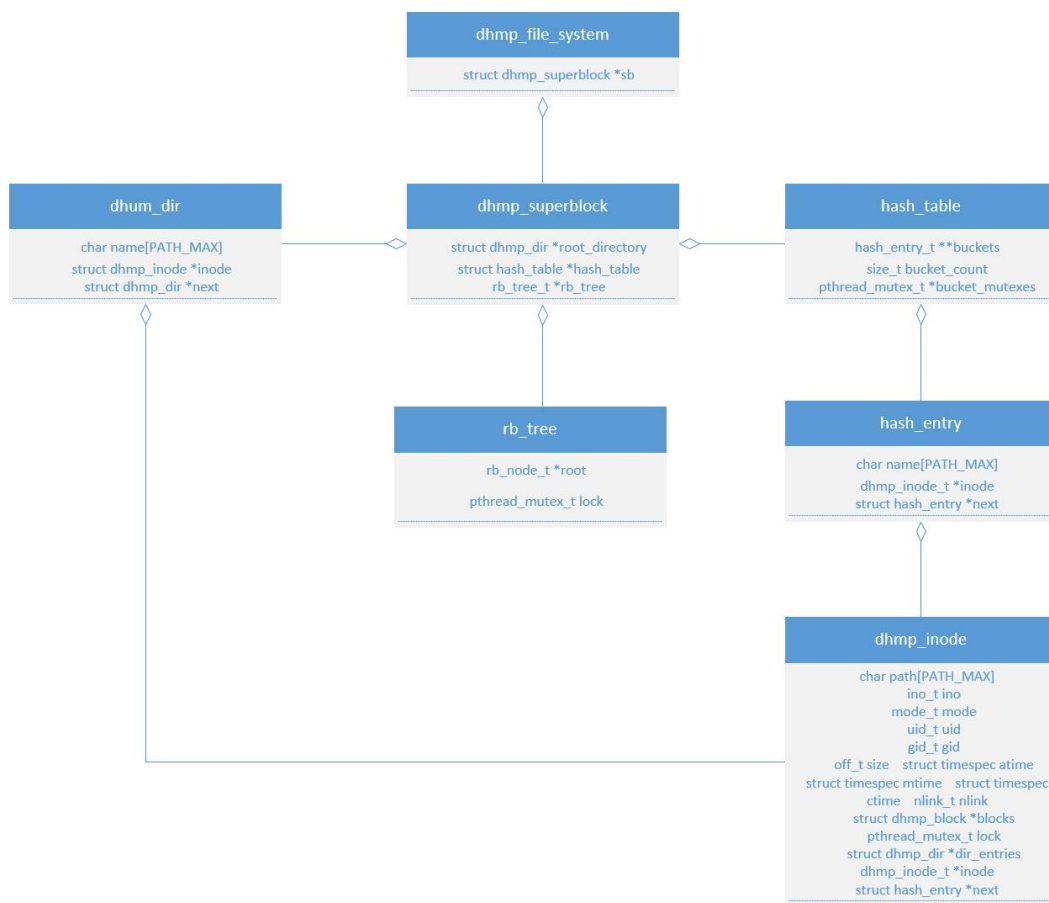


图 4 inode 数据结构逻辑关系图例

memory.h: 用于开辟内存池, 设计数据结构如下

1. `#define DATA_BLOCK_SIZE 4096`
2. `#define INITIAL_MEMORY_POOL_SIZE (100 * 1024 * 1024)` // 初始内存池大小为 100MB
3. `#define MAX_BLOCKS (INITIAL_MEMORY_POOL_SIZE / DATA_BLOCK_SIZE)` // 最大数据块数量
4. // 结构体定义



### 2.3.2 bt-fuse 接口实现

```
1. static struct fuse_operations my_ops = {  
2.     .getattr = my_getattr,  
3.     .read = my_read,  
4.     .write = my_write,  
5.     .create = my_create,  
6.     .utimens = my_utimens,  
7.     .readdir = my_readdir,  
8.     .mkdir = my_mkdir,  
9.     .init = my_init,  
10.    .destroy = my_destroy,  
11. };
```

这些功能函数通过 `fuse_operations` 结构体提供了文件系统的具体实现，涵盖了文件和目录的基本操作、文件系统的初始化和销毁。

1、`getattr(my_getattr)`: 获取文件或目录的属性信息。常用于返回文件或目录的元数据，如权限、大小、修改时间等。

2、`read(my_read)`: 读取文件的内容。处理读取请求，从文件中提取指定字节范围的数据并返回。

3、`write(my_write)`: 向文件写入数据。处理写入请求，将数据写入文件的指定位置。

4、`create(my_create)`: 创建新文件。处理文件创建请求，包括文件的初始设置和资源分配。

5、`utimes(my_utimes)`: 更新文件的访问时间和修改时间。处理时间戳的更新请求。

6、`readdir(my_readdir)`: 读取目录内容。处理读取目录请求，并返回目录中的文件和子目录列表。

7、`mkdir(my_mkdir)`: 创建新目录。处理目录创建请求，包括目录的初始设置和资源分配。

8、`init(my_init)`: 初始化文件系统。处理文件系统的初始化请求，用于设置和准备文件系统环境。

9、`destroy(my_destroy)`: 销毁文件系统。处理文件系统的卸载请求，清理和释放文件系统使用的资源。

### 2.3.3 bt-fuse 优化思路

**减少系统调用开销：**传统的文件系统直接处理系统调用可能会导致性能瓶颈。通过 bpftime 进行系统调用在用户态截断，可以减少系统调用的直接开销，提高整体效率。

**减少进程上下文切换：**传统的系统调用涉及从用户态到内核态的上下文切换。通过 ebpf 在用户态捕获和处理数据，然后将其放入共享内存，FUSE 文件系统可以在用户态直接读取数据，减少了用户态和内核态之间的上下文切换。

**提高数据处理速度：**将系统调用数据放入共享内存，使得 FUSE 文件系统可以快速访问和处理这些数据，避免了重复的数据复制和处理。

**增强文件操作效率：**共享内存中的数据允许 FUSE 文件系统快速获取必要的信息，从而提高文件操作的响应速度和整体效率。

## 2.4 extfuse 与 bt-fuse 比较总结

### extfuse 优化的方面

extfuse 作为一个优化 FUSE 文件系统性能的框架，主要在以下几个方面进行了优化：

#### 1、减少用户态和内核态切换：

●通过在用户态实现更多的文件系统逻辑，extfuse 减少了用户态和内核态之间的频繁切换，这种优化直接减少了上下文切换的开销。

#### 2、高效的 I/O 操作：

●优化了 I/O 请求的处理，比如通过批量处理或合并小的 I/O 请求，减少系统调用次数，从而提高 I/O 性能。

#### 3、缓存优化：

●在用户态实现更高效的缓存管理，减少对低层文件系统的频繁访问，进一步提升性能。

#### 4、更好的并发处理：

●通过改进 FUSE 的并发处理机制，extfuse 提高了并发访问效率，减少了锁竞争和线程切换的开销。

### 5、灵活的扩展性：

●通过在用户态实现文件系统功能，可以更方便地添加定制功能，增强了文件系统的灵活性。

### bt-fuse 的优化方式

#### 1、eBPF 拦截系统调用：（利用 bpftime 这个工具）

●使用 eBPF 程序在用户态拦截系统调用，然后将请求转发到用户态。这种方式减少了传统的 FUSE 文件系统中从内核态传递请求到用户态的开销。

#### 2、共享内存通信

●使用共享内存进行数据传输，避免了用户态与内核态之间的上下文切换吗，提高了通信速度。共享内存提供了低延时和高吞吐量的数据传输机制。

#### 3、用户态 inode 管理

●bt-fuse 在用户态实现了 inode 管理，这减少了内核态和用户态之间的交互，降低了系统调用的开销，并且允许更灵活地管理文件系统元数据。

### bt-fuse 和 extfuse 的优化区别

#### 1、技术手段的不同

●bt-fuse：

●依赖 eBPF 来拦截系统调用，将操作转发到用户态，并通过共享内存进行高效的通信。这种方法着重于减少系统调用的开销和提升通信速度。

●extfuse：

●通过在用户态扩展 FUSE 的能力，减少用户态和内核态之间的切换，并优化 I/O 操作和缓存管理。extfuse 更侧重于优化 FUSE 的基本框架和运行机制。

#### 2、性能提升的侧重点

●bt-fuse：

●主要针对通信效率和系统调用开销进行优化，重点在于提升数据传输速度和减少上下文切换的次数。这对高频次 I/O 操作的场景特别有效。

●extfuse：

●通过优化 FUSE 内部机制，如 I/O 请求处理、缓存管理和并发处理，提升整体性能。它更侧重于改善 FUSE 文件系统在各种应用场景下的通用性能。

### 结论

●bt-fuse 主要是通过减少系统调用和提高通信速度来提升性能，适合对通信效率要求高的场景，如频繁的小文件操作。

●extfuse 则通过优化 FUSE 内部机制和框架，适合广泛的使用场景，整体性能提升较为均衡。

## 三、技术框架

### 3.1 技术框架介绍

#### 3.1.1 bt-fuse 执行流程分析

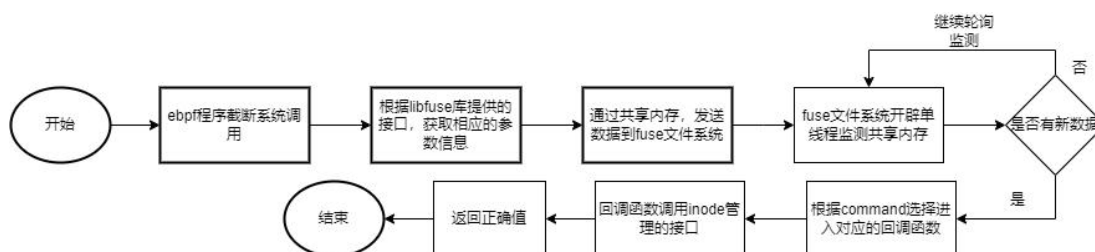


图 6 bt-fuse 执行流程图

#### 3.1.2 bt-fuse 架构

如下图，红色箭头代表的是 bt-fuse 的架构设计，黑色箭头代表的是原有 fuse 架构设计。

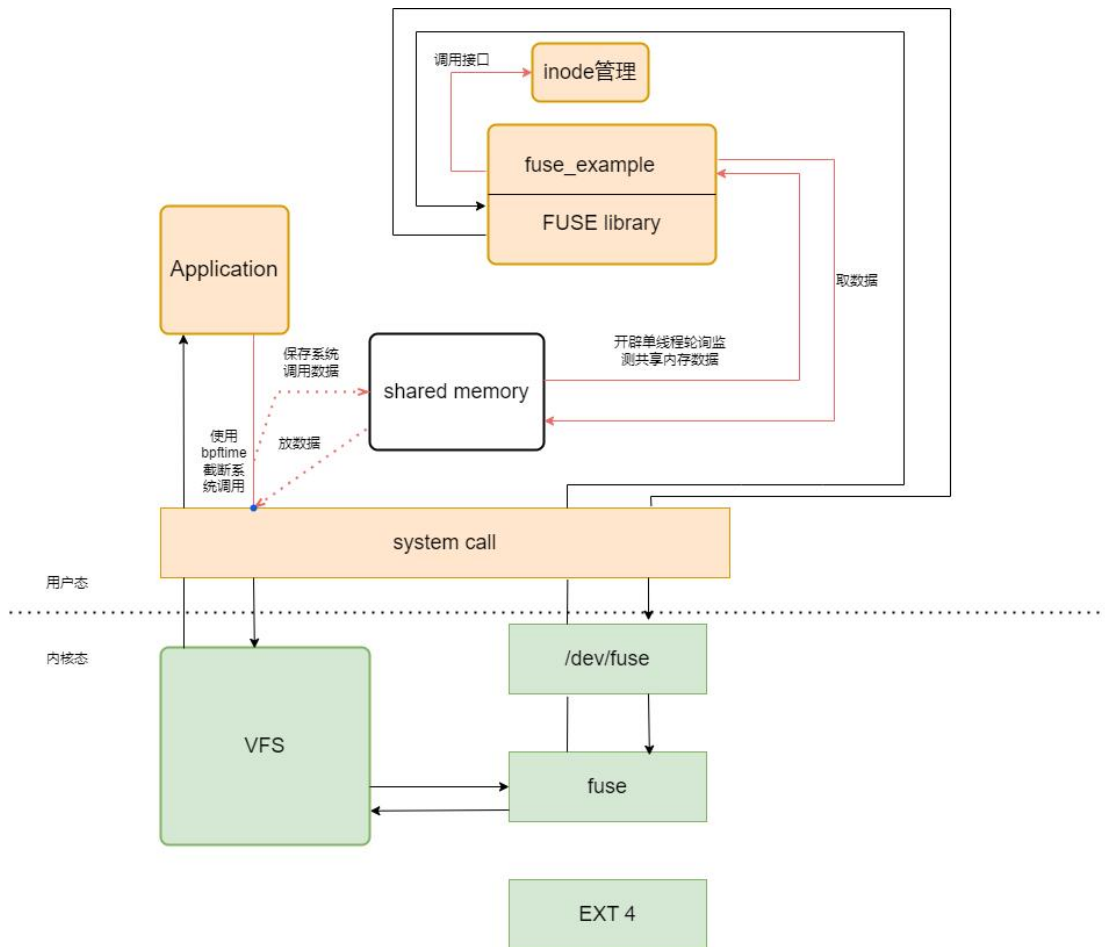


图 7 bt-fuse 架构

## 3.2 eBPF 截断系统调用

### 3.2.1 kprobe 截断系统调用

有一个 eBPF 辅助函数，`bpf_override_return` 允许程序覆盖返回值。恶意软件开发人员可以利用它来阻止它们认为不受欢迎的操作。例如，如果你想运行 `kill -9 <pid -of -ebpf - malware>`，恶意软件可以将 `kprobe` 附加到适当的函数以处理 `kill` 信号，返回错误，并有效地阻止系统调用的发生。`ebpfkit` 使用它来阻止可能导致发现控制 eBPF 程序的用户空间进程的操作。

限制:

- 有一个内核构建时选项可以启用它：`CONFIG_BPF_KPROBE_OVERRIDE`
- `ALLOW_ERROR_INJECTION` 它适用于使用宏的函数
- 目前仅支持 x86



### ●它只能与 kprobes 一起使用

最初设计的时候，使用 bpftime 去截断系统调用，但是对于 bpftime 的研究不够深入，截断功能无法实现的时候，考虑过用这个方案去代替 bpftime 截断系统调用。我们做过测试，打开内核开关，编译安装内核。如下图为对 ls 命令系统调用截断的图示：

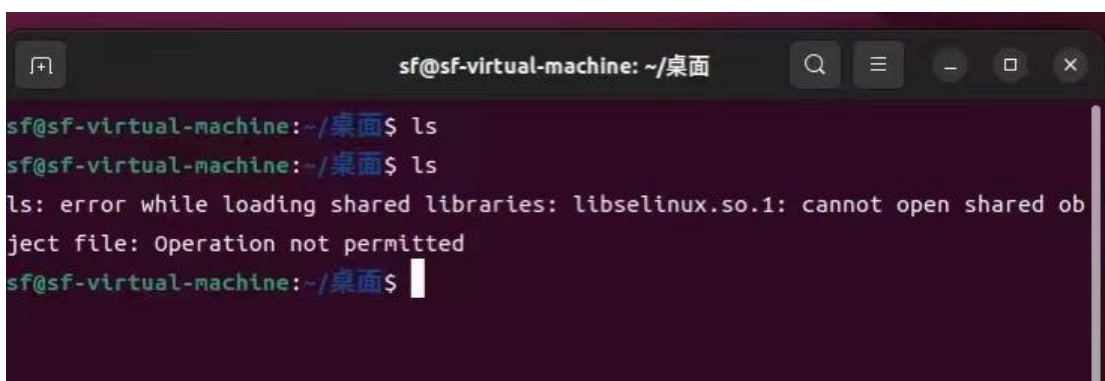


图 8 kprobe 截断 sys\_enter\_openat 系统调用示例

如图为我们使用 kprobe 拦截系统调用，截断 ls 命令系统调用的结果示例图。

但是这种实现方式，由于 kprobe 拦截系统处于内核态，会引发安全问题，例如，拦截和覆盖系统调用的返回值时没有正确处理边界情况或错误条件，可能会导致系统行为异常。

但是我们借助工具，bpftool 排除这个风险，如下所示：

### 使用 bpftool 排除风险

bpftool 是一个工具，用于管理和查看 eBPF 程序及其附加的挂钩。作为程序开发者，您可以使用 bpftool 来帮助识别和排除潜在的安全风险。以下是一些步骤和命令，帮助您监控和管理 eBPF 程序，以减少安全风险：

#### 1、查看附加的 eBPF 程序：

查看当前挂载的 eBPF 程序和它们附加的位置，例如：bpftool prog list  
这个命令将列出所有的 eBPF 程序及其详细信息，包括程序类型和附加的钩子位置。

#### 2、检查 eBPF 程序的详细信息：

获取特定 eBPF 程序的详细信息，以确保它没有被不当修改。例如：bpftool prog show id <prog\_id>（替换 prog\_id 为你感兴趣的 eBPF 程序的 ID）。

#### 3、查看挂载的 eBPF 钩子：bpftool

查看哪些系统调用被 eBPF 程序钩住，以确保没有未授权的拦截。例如：bpftool kprobe list

这些命令将会列出所有挂载的 kprobes，可以检查是否有相关的 eBPF 程序。

#### 4、审查 eBPF 程序的代码：

如果进一步检查 eBPF 程序的内容，可以查看其字节码。这有助于确认程序是否包含 bpf\_override\_return 或其他敏感操作。

bpftool prog dump xlated id <prog\_id>

#### 5、删除不必要的 eBPF 程序：

一旦需要了不需要的或有风险的 eBPF 程序，可以使用以下命令删除它们：

bpftool prog delete id <prog\_id>

### 总结

使用 bpftool，可以列出、查看、检查和删除 eBPF 程序。确保定期检查已加载的 eBPF 程序，以防止恶意程序或不必要的程序影响系统的安全性和稳定性。如果发现任何可疑的程序或不符合预期的行为，及时采取措施进行删除或修复。

因此我们在做截断选择的时候，考虑到系统安全问题，不希望能在内核里面直接操作，所以我们最终还是舍弃了这版方案，选择了改用这个 bpftime 在用户态进行系统调用截断。

### 3.2.2 bpftime 截断系统调用

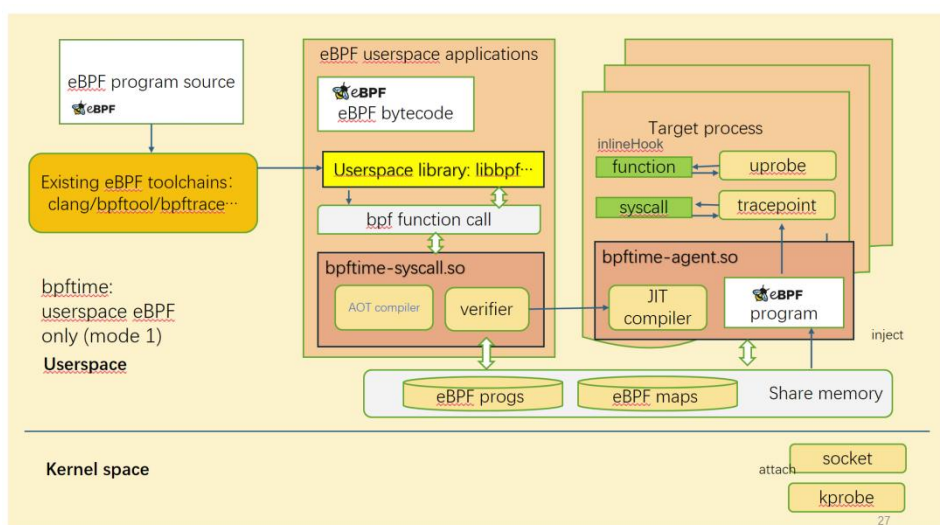


图 9 bpftime 用户态模式架构

通过 bpftime 运行在用户态的模式，在此模式下，bpftime 可以在不依赖内

核的情况下在用户空间运行 eBPF 程序，因此可以移植到低版本的 Linux 甚至其他系统，并且不需要 root 权限即可运行。它依赖于用户空间验证器来确保 eBPF 程序的安全性。所以 bt-fuse 文件系统是使用这个 bpftime 在用户态进行截断。

并且我们在用户态截断系统调用，不会影响到内核，安全性也是有保障的。

1、通过 error\_inject\_syscall.bpf.c、error\_inject\_syscall.c、error\_inject\_syscall.h 完成系统调用截断。

error\_inject\_syscall.h: 头文件定义了 eBPF 程序在内核态提取数据到用户态所需的数据结构

error\_inject\_syscall.bpf.c: 包含了 ebpf 程序代码，提取数据并传送到用户态。

error\_inject\_syscall.c: 包含了加载和管理 ebpf 程序的用户态代码，负责将 ebpf 程序附加到用户态并处理数据的传输。

3、bpftime 具体拦截系统调用方式：通过 bpf\_override\_return 进行截断。

4、bpf\_override\_return 功能：

作用：bpf\_override\_return 允许 ebpf 程序在系统调用完成后修改其返回值。

这个功能用于拦截和修改系统调用的结果，而不是直接拦截系统调用本身。

```

, flags 0, type=1
libbpf: elf: section(7) .BTF.ext, size 240, lin
k 0, flags 0, type=1
libbpf: looking for externs among 7 symbols...
libbpf: collected 0 externs total
libbpf: map '.rodata.str1.1' (global data): at
sec_idx 4, offset 0, flags 80.
[2024-07-23 11:20:58][info] [syscall_serv
r utils.cpp:24] Initialize syscall server
[2024-07-23 11:20:58][info][38227] Global shm c
onstructed. shm_open_type 0 for bpftime_maps_sh
m
[2024-07-23 11:20:58][info][38227] Global shm i
nitialized
[2024-07-23 11:20:58][info][38227] Enabling hel
per groups ufunc, kernel, shm map by default
[2024-07-23 11:20:58][info][38227] bpftime-sysc
all-server started
libbpf: map 0 is ".rodata.str1.1"
libbpf: map '.rodata.str1.1': created successfu
lly, fd=4
[]
l trace setup exiting..
[2024-07-23 11:20:40.524] [info] [agent-transfo
rmer.cpp:85] Transformer exiting, trace will be
usable now
123.txt
attach_override.h
error_inject
error_inject.bpf.c
error_inject.c
error_inject_syscall
error_inject_syscall.bpf.c
error_inject_syscall.c
Makefile
README.md
victim
victim.c
sf@LAPTOP-23HVJ9AS:~/bpftime/example/error-inje
ct$ sudo bpftime --install-location /home/sf/.b
pftime start -s echo '345' > 123.txt
echo: 写入错误: 不允许的操作
sf@LAPTOP-23HVJ9AS:~/bpftime/example/error-inje
ct$

```

图 10 截断 write 系统调用示例图

如图红框圈出来所示，这个 echo 的写入命令，也就是调用 write 调用被阻止，从而给出报错提示：写入错误，不允许的操作。可以看到系统调用截断成功。

### 3.3 bt-fuse 文件系统的实现

如 2.2.2 所示，我们目前已经实现的接口函数有这些。具体可以概括为可以实现的命令有：mkdir、touch、cat、echo、ls、fusermount。基本上都是关于目录和文件操作的执行命令。

#### 3.3.1 bt-fuse 核心逻辑实现

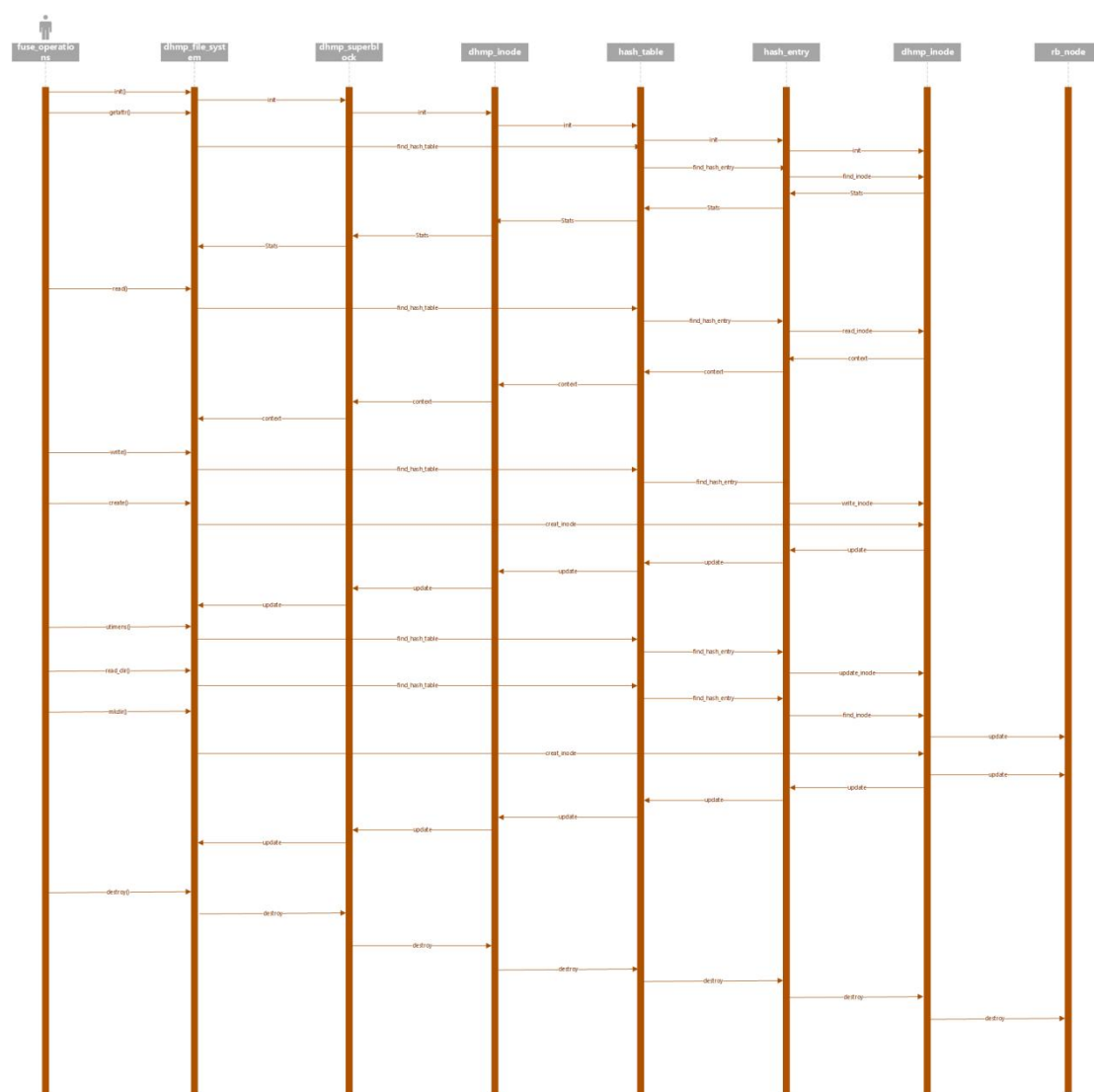


图 11 各个回调函数执行的时序图

1、对于目录，包括 mkdir、ls 命令，也就是要实现 my\_mkdir、my\_readdir、my\_getattr、my\_init 这些回调函数，可以理解为，一个命令的触发，通常要引起 fuse 文件系统好几个回调函数的响应。

我们为了实时监控这个共享内存里面的这个新数据，在 `init` 这个回调函数当中，添加了 `pthread thread_process_handle`，设置一个单线程，持续监控这个共享内存当中的新数据。

然后对于目录的创建，我们建立哈希表和红黑树来共同优化这个目录的查找和读取，实现的接口函数有：

```
1. unsigned long hash_function(const char *str);
2. dhmp_inode_t* find_inode_by_path(hash_table_t* hash_table, const char* path);
3. int insert_into_hash_table(hash_table_t * hash_table, const char * path, dhmp_inode_t *inode);
4. void rb_insert_fixup(rb_tree_t* rb_tree, rb_node_t* node);
5. void rotate_left(rb_tree_t* rb_tree, rb_node_t* node);
6. void rotate_right(rb_tree_t* rb_tree, rb_node_t* node);
7.
8. int insert_into_rb_tree(rb_tree_t* rb_tree, rb_node_t* node);
```

#### 哈希表相关函数：

①`hash_function`: 计算字符串的的哈希值。  
 ②`find_inode_by_path`: 通过 fuse 接口函数提供的参数路径，从哈希表中查找 inode。

③`insert_into_rb_tree`: 将 inode 插入到哈希表中。

#### 红黑树相关函数：

①`rb_insert_fixup`: 修正红黑树的平衡性。  
 ②`rotate_left`: 在红黑树中执行左旋操作。  
 ③`rotate_right`: 在红黑树中执行右旋操作。  
 ④`insert_into_rb_tree`: 将节点插入到红黑树中，并修正平衡性。

#### 选择哈希表的原因如下：

目的：哈希表支持高效的插入和删除操作，通常也是  $O(1)$  平均时间复杂度。

效果：能够快速添加或移除文件系统中的文件或目录，提高文件系统的动态更新性能。

#### 选择红黑树的原因如下：

目的：红黑树是一种平衡的二叉搜索树，它能够保持树的平衡，确保最坏情况下操作时间复杂度为  $O(\log n)$ 。

效果：通过平衡树的结构，保证插入、删除和查找操作的高效性，即使在大

量数据的情况下。

### 综合利用哈希表和红黑树：

哈希表用于快速查找：哈希表可以用来快速查找文件路径对应的 inode，适用于需要高效检索的场景。

红黑树用于有序管理和范围查询：红黑树可以用来管理文件系统中的节点，支持有序遍历和范围查询，同时保持操作的平衡和高效性。

2、对于文件，我们使用 touch、echo、cat 这些文件操作调用的接口函数，依然是 find\_inode\_by\_path，找到对应的 inode，然后进行操作，更新时间，给文件数据块写入数据，或者从数据块中取出数据，进行打印。

3、文件数据块具体内容，我们是开辟了内存池来管理。这里实现的接口函数有：

```
1.  dhmp_memory_pool_t* initialize_memory_pool();
2.
3.  void destroy_memory_pool(dhmp_memory_pool_t* memory_pool);
4.
5.  dhmp_block_t* allocate_block(dhmp_memory_pool_t* memory_pool);
6.
7.  void free_block(dhmp_memory_pool_t* memory_pool, dhmp_block_t* block);
```

initialize\_memory\_pool：初始化并返回一个新的内存池实例。

destroy\_memory\_pool：销毁内存池并释放其占用的资源。

allocate\_block：从内存池中分配一个新的数据块，供应用程序使用。

free\_block：释放数据块，将其返回到内存池中，以便重新使用。

### 内存池在 bt-fuse 文件系统中的应用：

●减少系统调用的开销：在 FUSE 文件系统中，频繁的文件操作可能会导致大量的内存分配和释放操作。内存池减少了对操作系统内存分配函数的依赖，降低了这些系统调用的开销。

●提高文件操作的响应速度：通过快速的内存分配和释放，内存池可以提升文件系统操作（如读取和写入）的响应速度，尤其是在高并发场景下。

●优化文件系统的稳定性：内存池的可预测性和简化的内存管理有助于提升 FUSE 文件系统的稳定性，避免因内存管理错误而导致的崩溃或性能问题。

●减少内存碎片：文件系统中的数据块通常具有固定大小，使用内存池可以



有效减少内存碎片，提高内存使用效率。

### 3.3.2 bt-fuse 运行结果

启动顺序：

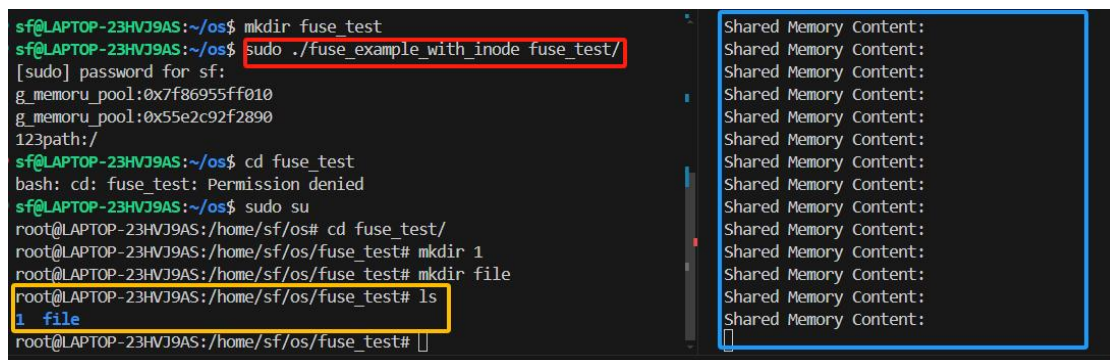
①先使用 `gcc -o shared_memory shared_memory.c` 编译共享内存代码，并且运行 `./shared_memory`。

②再用 `sudo bpftime --install-location /home/sf/.bpftime load ./error_inject_syscall` 去运行这个 `ebpf` 程序（使用的是 `bpftime` 在用户态的运行方式）。

③继续使用 `sudo bpftime --install-location /home/sf/.bpftime start -s mkdir /test` 命令创建一个目录，进行系统调用截断。

1、`mkdir` 创建一个目录：（测试 `my_mkdir` 回调函数、`my_readdir` 回调函数）

先使用这个 `bpftime` 截断系统调用，然后在 `fuse` 文件系统挂载的目录底下进行文件操作。我们将这个 `fuse` 文件系统挂载的目录设置为 `fuse_test`。



```

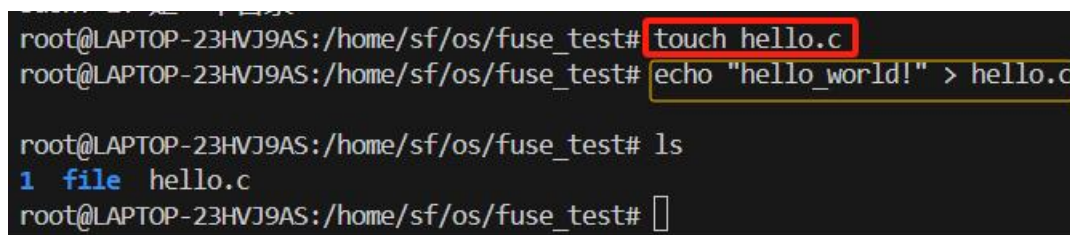
sf@LAPTOP-23HVJ9AS:~/os$ mkdir fuse_test
sf@LAPTOP-23HVJ9AS:~/os$ sudo ./fuse_example_with_inode fuse_test/
[sudo] password for sf:
g_memoru_pool:0x7f86955ff010
g_memoru_pool:0x55e2c92f2890
123path:/
sf@LAPTOP-23HVJ9AS:~/os$ cd fuse_test
bash: cd: fuse_test: Permission denied
sf@LAPTOP-23HVJ9AS:~/os$ sudo su
root@LAPTOP-23HVJ9AS:/home/sf/os# cd fuse_test/
root@LAPTOP-23HVJ9AS:/home/sf/os/fuse_test# mkdir 1
root@LAPTOP-23HVJ9AS:/home/sf/os/fuse_test# mkdir file
root@LAPTOP-23HVJ9AS:/home/sf/os/fuse_test# ls
1 file
root@LAPTOP-23HVJ9AS:/home/sf/os/fuse_test#
  
```

Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:  
 Shared Memory Content:

图 12 mkdir 命令测试

如图所示，蓝框圈出来的是我们启动共享内存程序，红框圈出来的是挂载文件系统的方式，然后黄色框圈出来的是我们创建好的两个目录文件。

2、`echo` 命令：（测试 `my_write` 回调函数、`my_create` 回调函数）



```

root@LAPTOP-23HVJ9AS:/home/sf/os/fuse_test# touch hello.c
root@LAPTOP-23HVJ9AS:/home/sf/os/fuse_test# echo "hello_world!" > hello.c

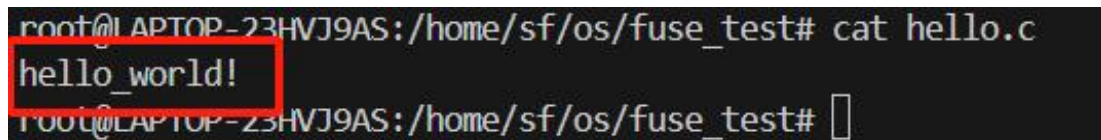
root@LAPTOP-23HVJ9AS:/home/sf/os/fuse_test# ls
1 file hello.c
root@LAPTOP-23HVJ9AS:/home/sf/os/fuse_test#
  
```

图 13 echo 命令测试

如图所示，红框圈出来的是创建一个文件，也就是 `touch` 操作，触发的是

my\_create 回调函数，黄框圈出来的是将”hello\_world!”写进去这个刚刚创建好的文件 hello.c。

3、cat 命令：（测试 my\_read 回调函数）



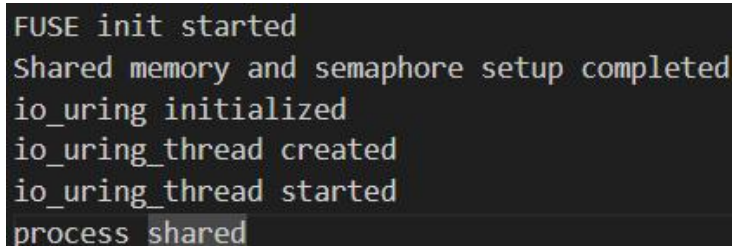
```
root@LAPTOP-23HVJ9AS:/home/sf/os/fuse_test# cat hello.c
hello_world!
root@LAPTOP-23HVJ9AS:/home/sf/os/fuse_test#
```

图 14 cat 命令测试

如图所示，红框圈出来的是我们刚刚写进文件的那个”hello\_world!”。

4、测试 my\_init 回调函数：

因为 init 回调函数，目前只设置了这个开辟一个单线程，我们测试这部分的功能，只能通过这个打印语句到日志文件，来判断这个回调函数有没有正常工作。



```
FUSE init started
Shared memory and semaphore setup completed
io_uring initialized
io_uring_thread created
io_uring_thread started
process shared
```

图 15 init 回调函数测试

如图所示，init 回调函数打印的语句，都是单线程开辟的时候要打印的语句内容。

### 3.4 进程通信-共享内存

因为当系统调用在用户态被截断后，我们要将请求重新转发到这个 fuse 文件系统，所以要建立一个通信机制。综合性能优化考虑，最终选择了共享内存这种方式。

我们对于这个进程通信，刚开始设计了好多种方案验证，比如 socket、netlink 机制等，最后会选择共享内存作为通信机制的主要原因是它在性能和开销方面相较于 socket 和 netlink 机制具有显著优势。共享内存能够减少上下文切换，提升数据传输速度，降低系统调用开销，并且在高并发处理和同步方面表现出色。此外，使用共享内存还能够简化设计和实现过程，并在延迟低场景中提供更好的性能。这些因素使得共享内存成为优化 FUSE 文件系统性能的理想选择。而这个



socket、netlink 机制需要系统调用来发送和接收消息，这些系统调用你会带来额外的开销。

### 3.4.1 进程通信

进程通信（Inter-Process Communication, IPC）是指在操作系统中，多个进程之间交换数据和信息的机制。由于进程在操作系统中具有独立的内存空间和资源，它们之间的直接访问是受到限制的。因此，IPC 为进程之间的数据交换提供了必要的手段。

进程通信的作用：

- 1、数据共享：进程可能需要共享数据或信息，如数据分析程序中的多个进程需要共享数据结果。
- 2、任务协调：进程可能需要协调其执行状态，以确保任务的正确执行和资源的有效利用。
- 3、进程同步：有些进程需要协调其工作步调，以避免冲突和数据不一致。
- 4、分布式计算：在分布式系统中，IPC 用于在不同计算节点上运行的进程之间交换信息。

### 3.4.2 共享内存

#### 1、共享内存的基本原理

不同的进程想要看到同一份资源，在操作系统内部，一定是通过某种调用，在物理内存当中申请一块内存空间，然后通过某种调用，让参与通信进程“挂接”到这份新开辟的内存空间上；其本质：将这块内存空间分别与各个进程各自的页表之间建立映射，再在虚拟地址空间当中开辟空间并将虚拟地址填充到各自页表的对应位置，使得虚拟地址和物理地址之家建立起对应关系，至此这些参与通信进程便可以看到了同一份物理内存，这块物理内存就叫做共享内存。

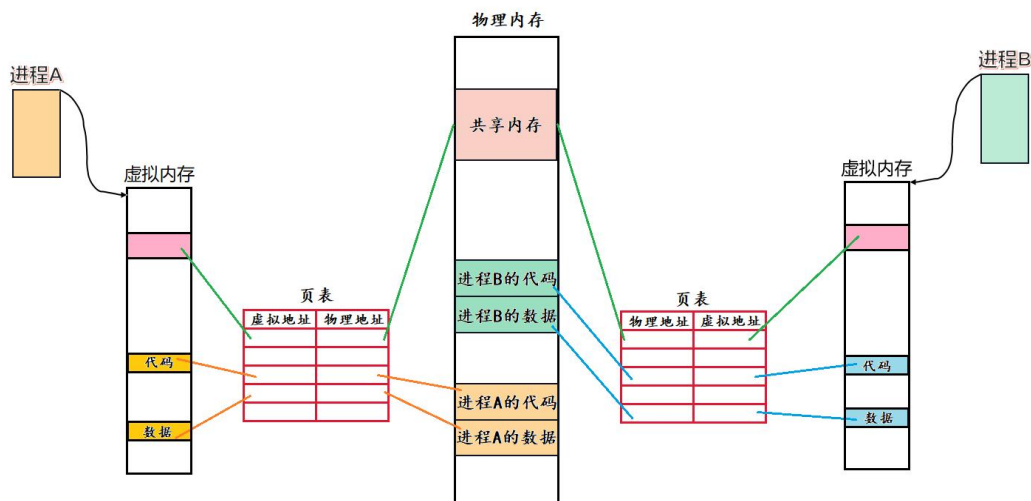


图 16 共享内存结构设计图例

shared\_memory.c 共享内存使用的函数功能如下：

```

1.  #define SHM_NAME "/my_shared_memory"
2.  #define SEM_NAME "/my_semaphore"
3.  #define SHM_SIZE 1024 // 共享内存大小
4.
5.  int shm_open(const char *name, int oflag, mode_t mode);
6.  int ftruncate(int fd, off_t length);
7.  void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
8.  sem_t *sem_open(const char *name, int oflag, ...);
9.  int sem_wait(sem_t *sem);
10. int sem_post(sem_t *sem);
11. int munmap(void *addr, size_t length);
12. int sem_close(sem_t *sem);
13. int shm_unlink(const char *name);
14. int sem_unlink(const char *name);
15.

```

**shm\_open:** 用于创建或打开共享内存对象 /my\_shared\_memory。

**ftruncate:** 用于设置共享内存的大小为 1024 字节。

**mmap:** 将共享内存映射到进程地址空间。

**sem\_open:** 用于创建或打开名为 /my\_semaphore 的信号量，并初始化为 1（表示二值信号量）。

**sem\_wait:** 用于进入临界区，即在访问共享内存之前获取信号量。

**sem\_post:** 用于离开临界区，即在访问共享内存之后释放信号量。

**munmap:** 用于解除共享内存的映射。

sem\_close:用于关闭信号量描述符。

shm\_unlink:用于删除共享内存对象 /my\_shared\_memory。

sem\_unlink:用于删除信号量 /my\_semaphore。

## 2、共享内存对优化 fuse 文件系统的优点

### ①减少进程间通信开销

共享内存允许多个进程直接访问同一块内存区域，而不需要通过内核空间的数据拷贝。这显著减少了进程间通信的开销和延迟。对于 FUSE 文件系统，这意味着用户空间的 FUSE 实现和 eBPF 程序可以高效地交换数据，减少了内核态和用户态之间的上下文切换。

### ②低延迟数据传输

共享内存提供了低延迟的数据交换机制，因为数据可以在用户空间直接读写，而不需要通过系统调用。这对于需要实时或高频次数据交互的应用（如文件系统操作）非常重要。通过共享内存，FUSE 文件系统能够更快速地处理 eBPF 程序收集的数据，从而提高整体性能。

### ③高效的资源利用

共享内存是一种高效的资源利用方式，因为它避免了数据的多次复制和转换。数据直接写入共享内存区域，减少了额外的内存使用和复制开销。这对于处理大规模数据或高并发文件操作时特别有用，可以提升系统的吞吐量和响应速度。

### ④简化的数据同步机制

利用共享内存可以简化进程间的数据同步。通过信号量、互斥锁等机制，可以有效地协调对共享内存的访问，避免数据竞争和一致性问题。这种同步机制可以在用户空间实现，减少了对内核同步机制的依赖。

### ⑤支持高并发操作

共享内存支持多个进程同时访问和修改数据，通过适当的同步机制（如信号量、互斥锁）可以有效管理并发访问。对于 FUSE 文件系统来说，这意味着能够处理更多的并发文件操作请求，提高系统的吞吐量和并发处理能力。

### ⑥降低系统调用频率

通过共享内存，用户态进程可以直接进行数据交换而无需频繁调用系统调用，这不仅减少了系统调用的开销，还降低了上下文切换的频率。这对于提高 FUSE 文件系统的性能至关重要，特别是在处理高频率的文件操作时。

总结：在 FUSE 文件系统中使用共享内存作为进程间通信机制，可以显著提升性能，减少延迟，优化资源利用，简化数据同步，并支持高并发操作。通过减少系

统调用频率和避免内核态到用户态的数据复制, 共享内存能够为 FUSE 文件系统提供一个高效的性能优化方案。

### 3、ebpf 程序当中使用共享内存

```
1.  typedef struct {
2.      char buffer[BUFFER_SIZE];
3.      atomic_size_t write_index;
4.      atomic_size_t read_index;
5.  } shared_memory_t;
```

shared\_memory\_t 结构体通过提供一个线程安全的缓冲区, 使用原子操作来管理读写索引, 使得多个进程能够高效、可靠地交换数据。它适用于生产者-消费者模式。在这里 ebpf 程序就是生产者模式。

```
1.  void write_data_to_queue(shared_memory_t *shm_ptr, const char *data, size_t size) {
2.      size_t write_index = atomic_load(&shm_ptr->write_index);
3.      size_t current_read_index = atomic_load(&shm_ptr->read_index);
4.      size_t next_write_index = (write_index + size) % BUFFER_SIZE;
5.
6.      if (next_write_index == current_read_index) {
7.          fprintf(stderr, "Error: Buffer is full, cannot write data.
8.          \n");
9.          return;
10.     }
11.     size_t first_part_size = BUFFER_SIZE - write_index;
12.
13.     if (size <= first_part_size) {
14.         memcpy(&shm_ptr->buffer[write_index], data, size);
15.     } else {
16.         memcpy(&shm_ptr->buffer[write_index], data, first_part_size);
17.         memcpy(&shm_ptr->buffer[0], data + first_part_size, size - first_part_size);
18.     }
19.     atomic_store(&shm_ptr->write_index, next_write_index);
20.
21.     // 输出写入的数据和索引位置进行调试
22.     fprintf(stderr, "Data written to shared memory at index %zu: %s\n", write_index, data);
23.
24.     // 设置 futex 变量为 1, 通知 FUSE 文件系统有新数据
```

```

25.     atomic_store(&shm_ptr->futex, 1);
26.     futex_wake(&shm_ptr->futex, 1); // 唤醒等待的 FUSE 进程
27. }

```

这段代码就是将从 eBPF 程序提取到的参数信息，放到这个缓冲区当中。  
它的执行流如下图所示：

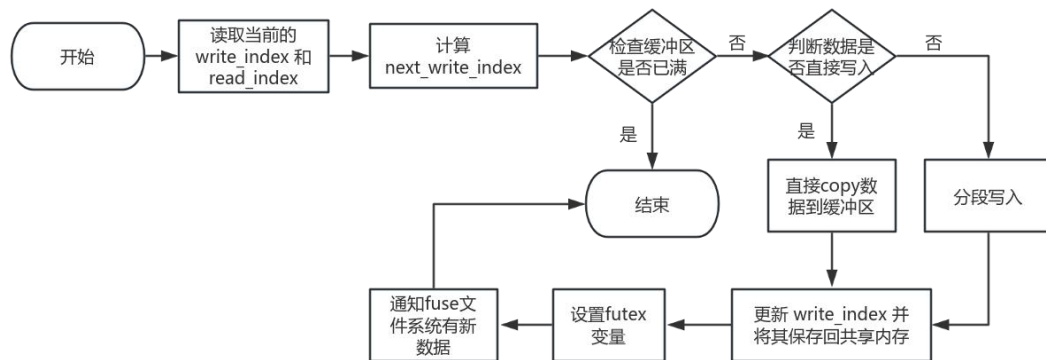


图 17 eBPF 提取数据到共享内存

#### 4、Fuse 文件系统当中使用共享内存

使用 `io_uring` 来优化 bt-fuse 文件系统从共享内存提取数据的过程。（这一方案在进一步优化中被否定，因为 `io_uring` 主要用于处理高并发、大量 I/O 操作的场景。而在你的场景中，eBPF 程序写入数据后，FUSE 文件系统立即获取并处理数据的实时性需求更适合使用一种更轻量的同步机制，而非 I/O 操作队列。）

##### ①异步 I/O 操作：

`io_uring` 提供了高效的异步 I/O 操作能力，可以减少等待 I/O 操作完成的时间。在处理共享内存中的数据时，`io_uring` 允许您异步提交 I/O 请求，而不需要阻塞线程，这可以提高整体系统的吞吐量和响应速度。

##### ②减少上下文切换：

传统的 I/O 操作可能涉及频繁的系统调用和上下文切换，这会增加系统的开销。`io_uring` 通过使用共享内存中的提交队列和完成队列，减少了这些开销，从而提高了 I/O 操作的效率。

##### ③高效的 I/O 完成通知：

`io_uring` 具有高效的事件通知机制，可以在 I/O 操作完成时及时通知消费者（即 FUSE 文件系统）。这避免了轮询或其他低效的等待机制，使得数据处理更加高效。

##### ④简化 I/O 操作流程

使用 `io_uring` 可以将 I/O 操作流程中的多个步骤（如提交、等待、处理）

整合到一个高效的接口中，从而简化代码和操作流程。

如下为 io\_uring 处理共享内存的代码示例：

```

1. // 处理共享内存
2. void process_shared_memory() {
3.     struct io_uring_cqe *cqe;
4.     shared_memory_t *shared_mem = (shared_memory_t *)shm_ptr;
5.     size_t write_index = atomic_load(&shared_mem->write_index);
6.     size_t read_index = atomic_load(&shared_mem->read_index);
7.
8.     while (read_index != write_index) {
9.         struct event event_in_memory;
10.        size_t first_part_size = BUFFER_SIZE - read_index;
11.
12.        if (read_index + sizeof(struct event) <= BUFFER_SIZE) {
13.            memcpy(&event_in_memory, &shared_mem->buffer[read_index], sizeof(struct event));
14.        } else {
15.            memcpy(&event_in_memory, &shared_mem->buffer[read_index], first_part_size);
16.            memcpy((char *)&event_in_memory + first_part_size, &shared_mem->buffer[0], sizeof(struct event) - first_part_size);
17.        }
18.        struct io_uring_sqe *sqe = io_uring_get_sqe(&ring);
19.        if (!sqe) {
20.            fprintf(log_file, "Failed to get SQE\n");
21.            break;
22.        }
23.
24.        io_uring_prep_nop(sqe);
25.        io_uring_submit(&ring);
26.
27.        read_index = (read_index + sizeof(struct event)) % BUFFER_SIZE;
28.        atomic_store(&shared_mem->read_index, read_index);
29.    }
30.
31.    while (io_uring_wait_cqe(&ring, &cqe) == 0) {
32.        io_uring_cqe_seen(&ring, cqe);
33.    }
34. }

```

该函数执行以下操作：

- 从共享内存读取数据（事件）。

- 使用 `io_uring` 提交 I/O 操作。
- 更新索引以保持缓冲区的一致性。
- 等待并处理 `io_uring` 完成队列中的事件，以确保所有提交的 I/O 操作都已完成。

### 3.4.3 bt-fuse 文件系统使用共享内存

利用共享内存、`futex` 和 `pthread` 线程机制，在用户态高效地接收并处理 FUSE 文件系统的请求。这种设计实现了低延迟的并发处理，对于需要快速处理大量数据的应用场景非常合适。

优化的主要方面包括：

- 利用共享内存实现高效的进程间通信；
- 利用 `futex` 实现高效的同步机制；
- 利用线程机制避免主线程阻塞，提高系统的响应性。

这种结合方案使得 FUSE 文件系统能够在高性能的场景下运行，并且保持代码的简介和可维护性。

#### 1、Futex(Fast Userspace Mutex)

`futex` 是一种轻量级的用户态锁机制，用于协调多线程访问共享内存。在这段代码中，`futex` 用于线程间的同步操作，确保当共享内存中的数据可用时，处理线程能及时被唤醒。

●`futex_wait()`：当前共享内存中的 `futex` 变量为 0 时，线程会阻塞等待，或者在超时时检查突出标志。

●`futex_wake()`：当共享内存中的 `futex` 变量被设置为 1 时，阻塞的线程会被唤醒，开始处理共享内存中的数据。

#### ●futex 的优点：

●效率高：`futex` 通过在用户态和内核态的结合操作来减少不必要的系统调用开销，只有在需要时才会触发内核态操作。

●轻量级：相比于传统的锁机制，`futex` 在无需进入内核态的情况下能提供快速的同步操作。

#### 2、POSIX 线程 (pthread)

该代码使用了 `pthread_create` 创建了一个新的线程 `data_polling_thread`, 用于异步处理共享内存中的数据。这使得文件系统的主线程不会因为等待数据处理而阻塞。

### ●线程的优点:

●**并发处理:** 通过多线程机制, 可以让文件系统并发地处理多个任务, 提高整体性能。

●**响应性:** 使用单独的线程来处理数据, 不会阻塞主线程, 保证系统的响应性。

如下为 bt-fuse 文件系统在共享内存存取数据的代码示例:

```

1.  void *data_polling_thread(void *arg) {
2.      shared_memory_t *shared_mem = (shared_memory_t *)shm_ptr;
3.
4.      struct timespec timeout;
5.      timeout.tv_sec = 0; // 超时时间 1 秒
6.      timeout.tv_nsec = 300000000;
7.
8.      while (!exiting) {
9.          // 等待 futex 变量变为 1, 设置超时以便检查退出标志
10.         if (futex_wait(&shared_mem->futex, 0, &timeout) == -ETIME
DOUT) {
11.             if (exiting) {
12.                 break; // 如果接收到退出信号, 退出循环
13.             }
14.             continue; // 超时继续等待
15.         }
16.
17.         size_t write_index = shared_mem->write_index;
18.         size_t read_index = shared_mem->read_index;
19.         size_t pos = read_index;
20.
21.         // 读取共享内存中的数据并处理
22.         while (pos != write_index) {
23.             struct event event_in_memory;
24.
25.             if (pos + sizeof(struct event) <= BUFFER_SIZE) {
26.                 memcpy(&event_in_memory, &shared_mem->buffer[pos],
sizeof(struct event));
27.             } else {
28.                 size_t first_part_size = BUFFER_SIZE - pos;

```



```

29.         memcpy(&event_in_memory, &shared_mem->buffer[pos],
first_part_size);
30.         memcpy((char *)&event_in_memory + first_part_size,
&shared_mem->buffer[0], sizeof(struct event) - first_part_size);
31.     }
32.
33.     //根据 event_in_memory.command 决定调用哪个函数
34.     if(strcmp(event_in_memory.command, "mkdir") == 0){
35.         int mkdir_result = my_mkdir(event_in_memory.pathname, event_in_memory.mode);
36.     }else if(strcmp(event_in_memory.command, "cat") == 0)
{
37.         char read_buffer[1024];
38.         int read_result = my_read(event_in_memory.pathname, read_buffer, sizeof(read_buffer), 0, NULL);
39.     }
40.     pos = (pos + sizeof(struct event)) % BUFFER_SIZE;
41. }
42.
43.     shared_mem->read_index = pos;
44.     shared_mem->futex = 0; // 重置 futex 变量
45. }
46.
47.     return NULL;
48. }

```

如下图为在共享内存取数据的流程图示例：

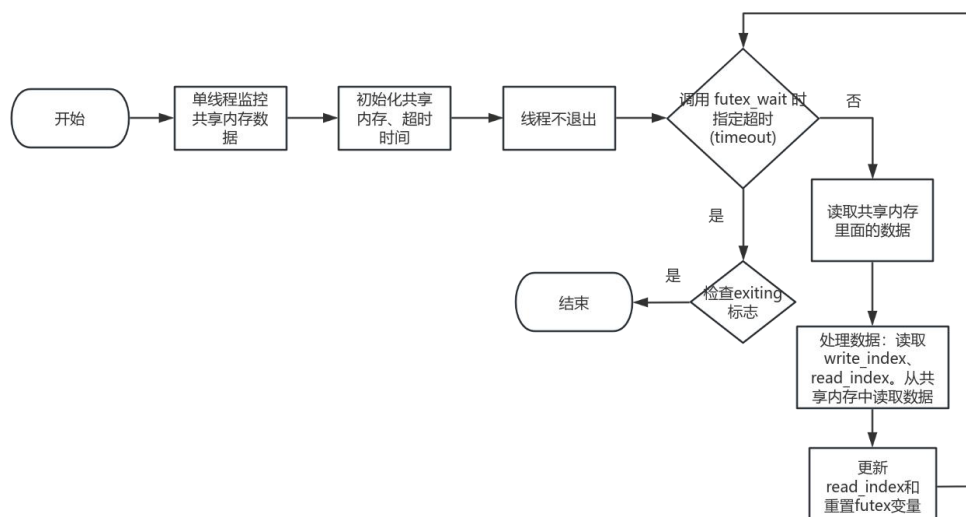


图 18 fuse 文件系统取共享内存数据

### 3.4.4 原有 fuse 文件系统和 bt-fuse 通信比较分析

#### 1、原有 FUSE 文件系统的通信方式

在原有的 FUSE 文件系统中，通信通道通常是基于内存的，具有以下特点：

- **内存通信**：FUSE 使用内存中的通信机制（例如 FIFO 或 socket）在用户态和内核态之间传输数据。这个通信机制较快，但可能会受到上下文切换和系统调用的影响。
- **系统调用开销**：每次文件系统操作都涉及到系统调用，从用户态到内核态的切换会有一定的性能开销。
- **并发处理**：原有的 FUSE 文件系统实现支持多个文件系统同时挂载，并为每个文件系统提供独立的通信通道，以支持并发访问。

#### 2、bt-fuse 的优化方案

使用 eBPF 程序在用户态拦截系统调用，并将请求通过共享内存传递给用户态的 fuse 文件系统。同时，在用户态实现 inode 管理，以减少内核态域用户态之间的交互。这种方法有以下特点：

- **用户态 inode 管理**：将 inode 管理从内核态迁移到用户态，可以减少内核态与用户态之间的交互，降低系统调用开销。用户态管理还可以更灵活地处理文件系统的元数据。
- **eBPF 程序拦截**：使用 eBPF 程序可以高效地拦截系统调用，在用户态进行操作，从而减少不必要的上下文切换。eBPF 程序提供了高效的事件捕获和处理能力。
- **共享内存通信**：使用共享内存进行进程间通信（IPC）通常比传统的系统调用通信更快，因为它避免了用户态和内核态之间的频繁切换。共享内存提供了低延迟、高带宽的数据传输能力。

比较分析：

#### 性能

- **系统调用开销**：bt-fuse 通过 eBPF 程序直接通过拦截系统调用，减少了系统调用的开销。传统的 FUSE 文件系统每次操作都涉及系统调用，可能会导致性能瓶颈。通过 eBPF 进行拦截，可以减少这种开销，从而提高性能。
- **内存通信速度**：共享内存的通信速度通常比基于文件描述符的通信（如 FIFO 或 socket）更快，因为共享内存避免了内核态的上下文切换和数据复制。

●并发处理：bt-fuse 如果能够有效地管理并发要求，并且共享内存设计得当，可以提高并发处理能力。原有 FUSE 实现支持并发，但性能可能受限于系统调用的开销和上下文切换。

### 复杂性

●实现复杂度：bt-fuse 需要在用户态实现 inode 管理，并编写 ebpf 程序，这可能增加实现和维护的复杂性。ebpf 编程和共享内存管理需要仔细设计和调试。

●兼容性：使用 ebpf 程序和共享内存可能会引入一些兼容性问题，特别是在不同的内核版本和操作系统上。原有 fuse 实现已经经过广泛测试，具有较好的兼容性和稳定性。

## 四、性能测试和监控

### 4.1 Grafana 和 Prometheus 性能监控

Grafana 和 Prometheus 是两个常用于监控和可视化系统性能的开源工具。

要实时监控 fuse 文件系统性能，可以通过安装一个 Prometheus Exporter 来收集相关指标，并向 Prometheus 提供数据。选择并配置适当的 Exporter，设置监听端口和挂载点，然后启动它以开始收集并暴露性能指标。

### 4.2 性能测试

使用 python 写测试代码：使用 matplotlib 绘制图表。

1、find 指令测试结果：

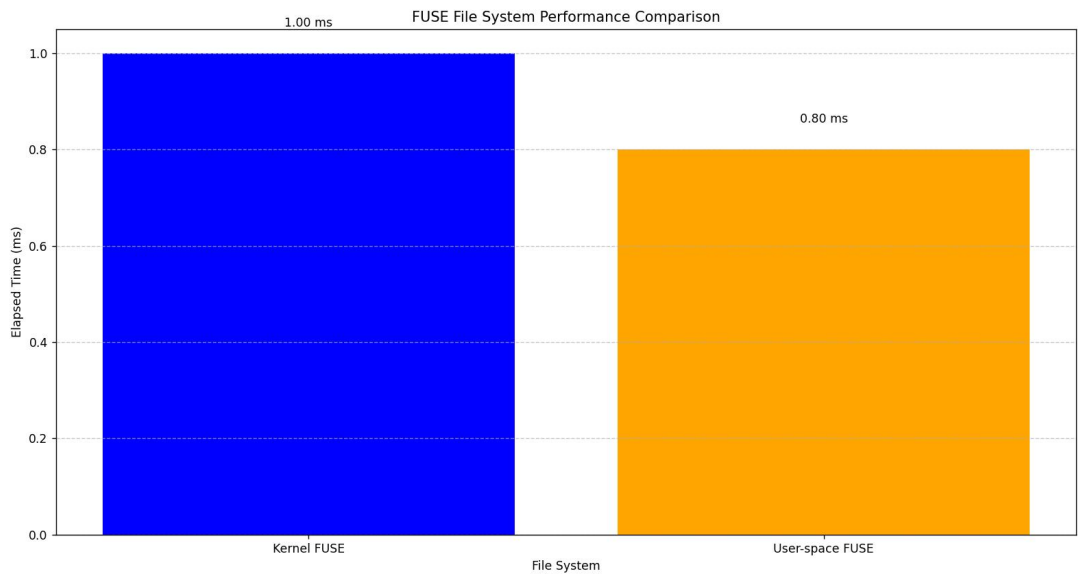


图 19 find 指令测试结果分析

test\_find\_fuse.sh 这段脚本的核心目的是通过在指定目录创建一组测试文件，反复执行 find 命令多次，并记录其平均执行时间，以评估 find 命令在该目录下搜索文件的性能。

2、传统 fuse 和 bt-fuse 对比，cat 多少万次一个文件，统计运行时间

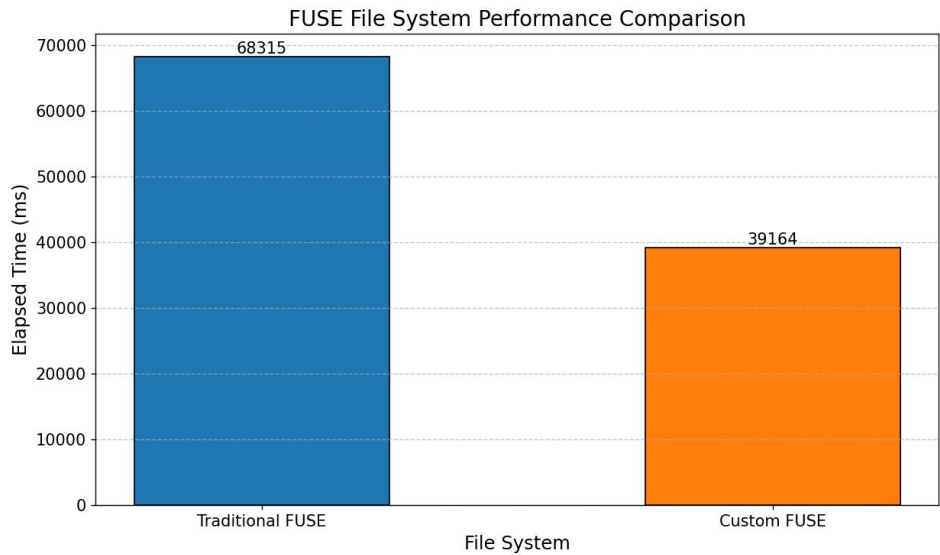


图 20 cat 文件性能分析

测试传统 FUSE 和 bt-fuse 在 cat 命令下对同一个文件进行数万次读取，并统计运行时间的目的，是评估和比较两种文件系统在高频次文件读取操作中的性能表现。通过这种性能对比，可以了解 bt-fuse 在优化文件读取效率和减少延迟方面的改进效果，以及其在高负载情况下的性能优势。

## 五、总结与展望

### 5.1 bt-fuse 未来的发展前景

#### 增强灵活性和定制化：

1、自定义功能：用户态的自定义 inode 管理可以使文件系统更加灵活，适应特定的应用需求或优化。

2、eBPF 的灵活性：eBPF 提供了强大的钩子机制，可以在系统调用层面做精细化的控制和拦截，这对于特殊的业务逻辑或性能优化有很大帮助。

#### 性能优化：

1、潜在优化：通过进一步优化 eBPF 程序和用户态 FUSE 文件系统的交互，可以减少开销。eBPF 可以用于实时监控和调整系统调用的行为，从而提高性能。

2、集成新技术：可以将这套系统与新的性能优化技术（如高效的内存管理、异步 I/O 等）集成，进一步提升性能。

#### 应用场景扩展：

1、特定场景：这种架构适合需要高性能和特殊处理逻辑的场景，如高性能计算、网络文件系统、实时数据处理等。

2、安全和审计：eBPF 还能用于安全审计和监控，结合自定义的 FUSE 文件系统可以实现强大的安全防护和监控机制。

### 5.2 bt-fuse 缺陷

#### 性能开销：

1、上下文切换：用户态和内核态之间的频繁切换可能会带来性能开销。即使自定义 inode 管理在用户态可以实现更高的灵活性，但可能也会导致额外的性能开销。

2、eBPF 的限制：eBPF 程序虽然强大，但也有一定的复杂性和限制。编写和调试 eBPF 程序可能会遇到挑战。

#### 复杂性和维护：

1、**系统复杂性**：用户态的 FUSE 文件系统和 eBPF 程序的结合增加了系统的复杂性。维护和调试这样一个系统可能会比较困难。

2、**兼容性**：需要确保 eBPF 程序和 FUSE 文件系统的兼容性，以及操作系统和内核的支持。

**安全性**：

1、**安全风险**：在用户态实现 inode 管理可能会引入额外的安全风险。例如，用户态的代码可能更容易受到攻击或出现安全漏洞。

2、**eBPF 安全性**：eBPF 程序可以访问系统调用，但也需要仔细处理权限和安全问题，以防止潜在的安全漏洞。

**测试和验证**：

**测试困难**：用户态的文件系统和 eBPF 程序的交互需要经过充分的测试，以确保系统稳定和性能符合预期。

**性能测试**：需要进行详细的性能测试，以确保系统在高负载下表现良好，并发现潜在的瓶颈。

## 5.3 bt-fuse 设计遇到的问题

**问题**：

1、bpftime 的截断存在不可避免的性能开销问题，因为是利用 bpftime 在用户态截断，发现每次在截断系统调用的时候，都要加载客户端的动态链接库，这是 bpftime 在截断系统调用的时候，不可避免的开销。

```
root@LAPTOP-23HVJ9AS:/home/sf/.bpftime# ls
bpftime          bpftime_daemon  libbpftime-agent-transformer.so  libbpftime-syscall-server.so
bpftime_agent.c  bpftime_tool    libbpftime-agent.so
```

图 21 bpftime 客户端动态链接库

如图所示，红框圈出来的就是这个客户端加载的动态链接库。

每次执行一次系统调用的截断，都需要加载一次动态链接库，这个开销非常大。我们用 time 命令监测粗略一次执行，例如：`time sudo bpftime --install-location /home/sf/.bpftime start -s mkdir /test`。

我们粗略发现运行结果如下：

```

real    0m0.717s
user    0m0.008s
sys     0m0.002s
root@LAPTOP-23HV19AS: /home/sf/bpftime/example/error

```

图 22 bpftime 加载 mkdir 命令测试

- ①real 时间为 0.717 秒，表示整个操作从开始到结束的总耗时。
- ②user 时间为 0.008 秒，表示程序在用户模式下执行的时间，这通常包括处理数据的时间。
- ③sys 时间为 0.002 秒，表示程序在内核模式下执行的时间，这通常包括执行系统调用的时间，比如文件操作、网络操作等
- 然后我们推测就是因为在执行命令时，操作系统需要加载命令所依赖的动态链接库，这通常会增加 real 时间，特别是当第一次执行时，系统可能需要从磁盘读取这些库并将它们加载到内存中。

**如何验证：（重复执行命令）**

```

real    0m0.565s
user    0m0.004s
sys     0m0.009s
root@LAPTOP-23HV19AS: /home/sf/bpftime/example/error

```

图 23 第二次测试时间

如图所示，这个执行时间明显加快了。因此可以看到一定是因为要加载这个客户端的链接库。后续是操作系统把这个动态链接库加载到了缓存里面，因此时间缩短了。

### 综合分析：

- 1、因为 bpftime 这个工具，目前对于程序开发者来说就是黑盒，因此对于为什么拦截系统调用会耗时这么长时间，其中可能涉及到 bpftime 这样的工具在执行时进行特定的设置或分析，例如启动跟踪、数据收集等操作，这些都会占用额外的时间。
- 2、启动一个新的进程本身就需要时间，特别是如果涉及复杂的命令，这会导致 real 时间增大。进程的创建、初始化、上下文切换以及最后的终止都会消耗时间。
- 3、如果命令频繁调用系统调用（syscalls）或进行用户态与内核态的切换，每次切换都会引入一定的开销，这在一个复杂工具执行时间可能会变得明显。

### 未来解决方案：

可以使用 LD\_PRELOAD 来预加载自定义的动态链接库，并通过它来拦截和重定向系统调用。这样，可以在用户空间拦截系统调用，将请求转发到共享内存，从而提高性能。

大概实现的步骤有：

- 1、创建自定义动态链接库，编译动态链接库。
- 2、使用 LD\_PRELOAD 预加载库。  
（使用 LD\_PRELOAD 环境变量来预加载动态链接库，这样会在其他库之前加载并拦截系统调用）
- 3、借助我们原有已经实现的共享内存。
- 4、在 fuse 文件系统里面监控共享内存里面的数据（这里原有 fuse 文件系统的

设计不变)。

总结:

相当于我们如果真的想要实现一个高性能的 fuse 文件系统，因为 bpftime 拦截系统调用的开销真的太大，如果想要真的提升性能，就要抛弃掉 bpftime 通过加载自身的客户端动态链接库的方式，把截断系统调用和这个请求转发，换成这里的自己实现一个动态链接库，并且使用 LD\_PRELOAD 来预加载自定义的动态链接库。