Extension Framework for File Systems in User space（简称ExtFUSE）

1.FUSE 是什么?

FUSE是开发用户文件系统的最新框架。Fuse框架理论上包含一个内核模块（fuse.ko）和一个用户空间守护进程（下文称FUSE file-system daemon，其基于libfuse库实现）。

但FUSE 框架具体包含了 3 个组件：

- 内核模块 fuse.ko ： 用来接收 vfs 传递下来的 IO 请求，并且把这个 IO 封装之后通过管道发送到用户态；
- 用户态 lib 库 libfuse ： 解析内核态转发出来的协议包，拆解成常规的 IO 请求；
- mount 工具： fusermount

内核模块fuse.ko

内核模块fuse.ko加载时被注册成 Linux 虚拟文件系统的一个 fuse 文件系统驱动。此外，还注册了一个/dev/fuse的块设备。该块设备作为FUSE file-system daemon与内核通信的桥梁，FUSE file-system daemon通过/dev/fuse读取fuse request，处理后将reply写入/dev/fuse。

libfuse库

该库为开发者提供了两组不同的API。首先，一个fuse_lowlevel_ops接口，该接口导出所有VFS操作，例如查找路径到inode映射。它被需要访问低级抽象的文件系统使用(例如，inodes)用于自定义优化。第二，一个构建在低级API之上的高级fuse_operations接口。它隐藏了复杂的抽象，并提供了一个简单的API，便于开发。根据它们的特定用例，开发人员可以采用这两组API中的任何一种。此外，这两组API中的许多操作都是可选的。

（1）struct fuse_lowlevel_ops的成员如下所示，其中init方法在其它所有方法之前调用，用于初始化文件系统，fuse已经实现，destroy则是在文件系统被卸载时做一些清理工作。用于大多数请求的参数都是fuse_ino_t类型的ino，而文件系统提供给用户的视图是以文件名呈现的，故lookup是实现文件系统的关键，它在parent中查找名字name对应的文件，并返回相应的信息，可使用fuse_reply_entry或fuse_reply_err作为请求的返回。

```
struct fuse_lowlevel_ops {
    void (*init) (void *userdata, struct fuse_conn_info *conn);
    void (*destroy) (void *userdata);
    void (*lookup) (fuse_req_t req, fuse_ino_t parent, const char *name);
    void (*forget) (fuse_req_t req, fuse_ino_t ino, uint64_t nlookup);
    void (*getattr) (fuse_req_t req, fuse_ino_t ino,
            struct fuse_file_info *fi);
    void (*setattr) (fuse_req_t req, fuse_ino_t ino, struct stat *attr,
            int to_set, struct fuse_file_info *fi);
    void (*readlink) (fuse_req_t req, fuse_ino_t ino);
    void (*mknod) (fuse_req_t req, fuse_ino_t parent, const char *name,
              mode_t mode, dev_t rdev);
    void (*mkdir) (fuse_req_t req, fuse_ino_t parent, const char *name,
              mode_t mode);
    void (*unlink) (fuse_req_t req, fuse_ino_t parent, const char *name);
    void (*rmdir) (fuse_req_t req, fuse_ino_t parent, const char *name);
    void (*symlink) (fuse_req_t req, const char *link, fuse_ino_t parent,
```

```
            const char *name);
    void (*rename) (fuse_req_t req, fuse_ino_t parent, const char *name,
            fuse_ino_t newparent, const char *newname,
            unsigned int flags);
    void (*link) (fuse_req_t req, fuse_ino_t ino, fuse_ino_t newparent,
            const char *newname);
    void (*open) (fuse_req_t req, fuse_ino_t ino,
            struct fuse_file_info *fi);
    void (*read) (fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
            struct fuse_file_info *fi);
    void (*write) (fuse_req_t req, fuse_ino_t ino, const char *buf,
            size_t size, off_t off, struct fuse_file_info *fi);
    void (*flush) (fuse_req_t req, fuse_ino_t ino,
            struct fuse_file_info *fi);
    void (*release) (fuse_req_t req, fuse_ino_t ino,
            struct fuse_file_info *fi);
    void (*fsync) (fuse_req_t req, fuse_ino_t ino, int datasync,
            struct fuse_file_info *fi);
    void (*opendir) (fuse_req_t req, fuse_ino_t ino,
            struct fuse_file_info *fi);
    void (*readdir) (fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
            struct fuse_file_info *fi);
    void (*releasedir) (fuse_req_t req, fuse_ino_t ino,
            struct fuse_file_info *fi);
    void (*fsyncdir) (fuse_req_t req, fuse_ino_t ino, int datasync,
            struct fuse_file_info *fi);
    void (*statfs) (fuse_req_t req, fuse_ino_t ino);
    void (*setxattr) (fuse_req_t req, fuse_ino_t ino, const char *name,
            const char *value, size_t size, int flags);
    void (*getxattr) (fuse_req_t req, fuse_ino_t ino, const char *name,
            size_t size);
    void (*listxattr) (fuse_req_t req, fuse_ino_t ino, size_t size);
    void (*removexattr) (fuse_req_t req, fuse_ino_t ino, const char *name);
    void (*access) (fuse_req_t req, fuse_ino_t ino, int mask);
    void (*create) (fuse_req_t req, fuse_ino_t parent, const char *name,
            mode_t mode, struct fuse_file_info *fi);
    void (*getlk) (fuse_req_t req, fuse_ino_t ino,
            struct fuse_file_info *fi, struct flock *lock);
    void (*setlk) (fuse_req_t req, fuse_ino_t ino,
            struct fuse_file_info *fi,
            struct flock *lock, int sleep);
    void (*bmap) (fuse_req_t req, fuse_ino_t ino, size_t blocksize,
            uint64_t idx);
    void (*ioctl) (fuse_req_t req, fuse_ino_t ino, int cmd, void *arg,
            struct fuse_file_info *fi, unsigned flags,
            const void *in_buf, size_t in_bufsz, size_t out_bufsz);
    void (*poll) (fuse_req_t req, fuse_ino_t ino, struct fuse_file_info *fi,
            struct fuse_pollhandle *ph);
    void (*write_buf) (fuse_req_t req, fuse_ino_t ino,
            struct fuse_bufvec *bufv, off_t off,
            struct fuse_file_info *fi);
    void (*retrieve_reply) (fuse_req_t req, void *cookie, fuse_ino_t ino,
            off_t offset, struct fuse_bufvec *bufv);
    void (*forget_multi) (fuse_req_t req, size_t count,
            struct fuse_forget_data *forgets);
    void (*flock) (fuse_req_t req, fuse_ino_t ino,
            struct fuse_file_info *fi, int op);
    void (*fallocate) (fuse_req_t req, fuse_ino_t ino, int mode,
```

```
                off_t offset, off_t length, struct fuse_file_info *fi);
    void (*readdirplus) (fuse_req_t req, fuse_ino_t ino, size_t size, off_t
off,
                struct fuse_file_info *fi);
    void (*copy_file_range) (fuse_req_t req, fuse_ino_t ino_in,
                off_t off_in, struct fuse_file_info *fi_in,
                fuse_ino_t ino_out, off_t off_out,
                struct fuse_file_info *fi_out, size_t len,
                int flags);
};
```

用户实现的接口是如何跟这个结构关联起来的?

FUSE中已经实现了一组接口，在fuse_lowlevel.c中，定义了一个静态的结构数组，该数组的元素为一组（函数，名字）的结构，但没做什么实际的工作，当FUSE用户空间的FUSE file-system daemon从/fuse/dev中读取到请求之后，它通过请求号来判别各个请求，并调用这里相应的处理函数，如读取到read调用时，会调用do_read进行处理。

```
static struct {
    void (*func)(fuse_req_t, fuse_ino_t, const void *);
    const char *name;
} fuse_ll_ops[] = {
    //只列举了一部分
    [FUSE_LOOKUP]      = { do_lookup,      "LOOKUP"      },
    [FUSE_OPEN]        = { do_open,        "OPEN"        },
    [FUSE_READ]        = { do_read,        "READ"        },
    [FUSE_WRITE]       = { do_write,       "WRITE"       },
    [FUSE_STATFS]      = { do_statfs,      "STATFS"      },
    [FUSE_FLUSH]       = { do_flush,       "FLUSH"       },
    [FUSE_INIT]        = { do_init,        "INIT"        },
    [FUSE_OPENDIR]     = { do_opendir,     "OPENDIR"     },
    [FUSE_READDIR]     = { do_readdir,     "READDIR"     },
    [FUSE_RELEASEDIR]  = { do_releasedir,  "RELEASEDIR"  },
    [FUSE_DESTROY]     = { do_destroy,     "DESTROY"     }
};
```

接下来以do_read()的实现为例：

```
static void do_read(fuse_req_t req, fuse_ino_t nodeid, const void *inarg)
{
    struct fuse_read_in *arg = (struct fuse_read_in *) inarg;
    // 如果用户实现了read操作，则调用用户空间的read，否则以没有实现该调用为错误响应，这里
的op就是用户实现文件系统时实现的，并传递给fuse。
    if (req->f->op.read) {
        struct fuse_file_info fi;
        memset(&fi, 0, sizeof(fi));
        fi.fh = arg->fh;
        fi.fh_old = fi.fh;
        req->f->op.read(req, nodeid, arg->size, arg->offset, &fi);
    } else
        fuse_reply_err(req, ENOSYS);
}
```

（2）FUSE提供了一组更简单的接口fuse_operations，详细说明请参考fuse.h。这组接口的参数跟Unix提供的系统调用的参数很类似，开发者更易理解，FUSE想开发者屏蔽了底层的相关对象，直接以文件名作为参数，只有开发者按照自己的方式，把这组接口实现就可以。

```c
struct fuse_operations {
    int (*getattr) (const char *, struct stat *, struct fuse_file_info *fi);
    int (*readlink) (const char *, char *, size_t);
    int (*mknod) (const char *, mode_t, dev_t);
    int (*mkdir) (const char *, mode_t);
    int (*unlink) (const char *);
    int (*rmdir) (const char *);
    int (*symlink) (const char *, const char *);
    int (*rename) (const char *, const char *, unsigned int flags);
    int (*link) (const char *, const char *);
    int (*chmod) (const char *, mode_t, struct fuse_file_info *fi);
    int (*chown) (const char *, uid_t, gid_t, struct fuse_file_info *fi);
    int (*truncate) (const char *, off_t, struct fuse_file_info *fi);
    int (*open) (const char *, struct fuse_file_info *);
    int (*read) (const char *, char *, size_t, off_t,
             struct fuse_file_info *);
    int (*write) (const char *, const char *, size_t, off_t,
              struct fuse_file_info *);
    int (*statfs) (const char *, struct statvfs *);
    int (*flush) (const char *, struct fuse_file_info *);
    int (*release) (const char *, struct fuse_file_info *);
    int (*fsync) (const char *, int, struct fuse_file_info *);
    int (*setxattr) (const char *, const char *, const char *, size_t, int);
    int (*getxattr) (const char *, const char *, char *, size_t);
    int (*listxattr) (const char *, char *, size_t);
    int (*removexattr) (const char *, const char *);
    int (*opendir) (const char *, struct fuse_file_info *);
    int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t,
            struct fuse_file_info *, enum fuse_readdir_flags);
    int (*releasedir) (const char *, struct fuse_file_info *);
    int (*fsyncdir) (const char *, int, struct fuse_file_info *);
    void *(*init) (struct fuse_conn_info *conn,
               struct fuse_config *cfg);
    void (*destroy) (void *private_data);
    int (*access) (const char *, int);
    int (*create) (const char *, mode_t, struct fuse_file_info *);
    int (*lock) (const char *, struct fuse_file_info *, int cmd,
             struct flock *);
     int (*utimens) (const char *, const struct timespec tv[2],
             struct fuse_file_info *fi);
    int (*bmap) (const char *, size_t blocksize, uint64_t *idx);
    int (*ioctl) (const char *, int cmd, void *arg,
              struct fuse_file_info *, unsigned int flags, void *data);
    int (*poll) (const char *, struct fuse_file_info *,
             struct fuse_pollhandle *ph, unsigned *reventsp);
    int (*write_buf) (const char *, struct fuse_bufvec *buf, off_t off,
              struct fuse_file_info *);
    int (*read_buf) (const char *, struct fuse_bufvec **bufp,
             size_t size, off_t off, struct fuse_file_info *);
    int (*flock) (const char *, struct fuse_file_info *, int op);
    int (*fallocate) (const char *, int, off_t, off_t,
              struct fuse_file_info *);
    ssize_t (*copy_file_range) (const char *path_in,
                    struct fuse_file_info *fi_in,
                    off_t offset_in, const char *path_out,
                    struct fuse_file_info *fi_out,
                    off_t offset_out, size_t size, int flags);
```

```
58 };
```

FUSE的核心机制：内核-用户通信

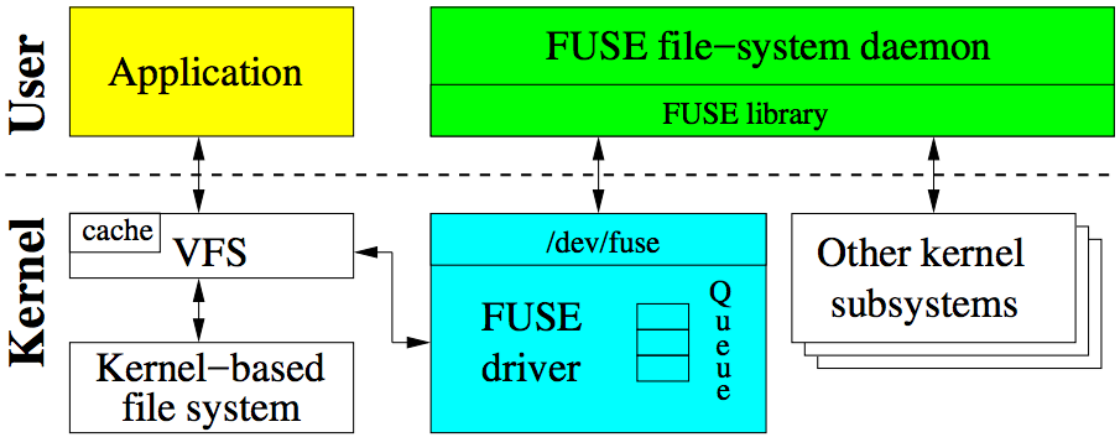


Figure 1: FUSE high-level architecture.

在FUSE中，内核和用户空间之间的通信是通过一种称为"通信通道"（Communication Channel）的机制来实现的。这个通道是在内核和用户空间之间建立的一个基于内存的、虚拟的、双向的通信管道，可以通过它进行数据的传输和命令的交互。它是FUSE实现的核心部分，负责传递文件系统操作请求和操作结果，是FUSE实现用户空间和内核之间交互的核心机制。

具体来说，当用户空间的FUSE文件系统需要向内核发送一个请求时，它会通过一个特殊的虚设备文件（/dev/fuse）向内核发起请求，并等待内核的响应。内核在接收到请求后，会通过FUSE通信通道将请求传输给用户空间的FUSE进程，FUSE进程收到请求后进行相应的处理，并将处理结果返回给内核，最终内核再将处理结果传递给FUSE文件系统。

需要注意的是，FUSE通信通道是一种基于内存的通信方式，因此数据的传输速度非常快。同时，FUSE还支持多个文件系统同时挂载的情况，每个文件系统都可以拥有自己的通信通道，以保证并发访问的效率。

用户文件系统（FUSE）的实现过程：

当application挂载（fusermount）到fuse文件系统上，并且执行一些系统调用时，VFS会将这些操作发送至fuse driver，fuse driver创建了一个fuse request结构体，并把request保存在请求队列中。此时，执行操作的进程会被阻塞，同时fuse daemon通过读取/dev/fuse将request从内核队列中取出，并且提交操作到底层文件系统中(例如 EXT4 或 F2FS)。当处理完请求后，fuse daemon会将reply写回/dev/fuse，fuse driver此时把requset标记为completed，最终唤醒用户进程。

内核文件系统的实现过程：

在用户态发出应用命令之后，调用read()（此处以读取文件为例）系统调用，传递给VFS层的sys_read()函数，sys_read()根据文件fd指定的索引,从当前进程描述符中取出相应的file对象,并调用vfs_read执行相应文件的file->f_op.read()读取操作，之后通过ext2_file_operations()结构（此处以EXT2文件系统为例）调用do_sync_read()，do_sync_read()函数会执行page cache操作及一系列判断，最终生成BIO。接着执行mpage_submit_bio()函数调用通用处理层的入口函数——generic_make_request()，将bio传送到IO调度层进行处理。IO调度层对bio进行合并、排序,以提高IO效率，然后,调用设备驱动层的回调函数,request_fn,转到设备驱动层处理。设备驱动层中的request函数对请求队列中每个bio进行分别处理,根据bio中的信息向磁盘控制器发送命令，处理完成后,调用完成函数end_bio以通知上层完成。

用户文件系统与内核文件系统区别：

1) FUSE除了能够使用普通用户进行挂载外，文件系统的元数据和数据操作也都是由用户空间的进程来实现，并且能够使用内核文件系统的接口进行访问；

2) FUSE的引入极大的方便了文件系统的开发和调试，相对于复杂的内核文件系统，无需编写任何内核代码，无需重新编译内核，维护上也由此变得简单，

FUSE内核队列

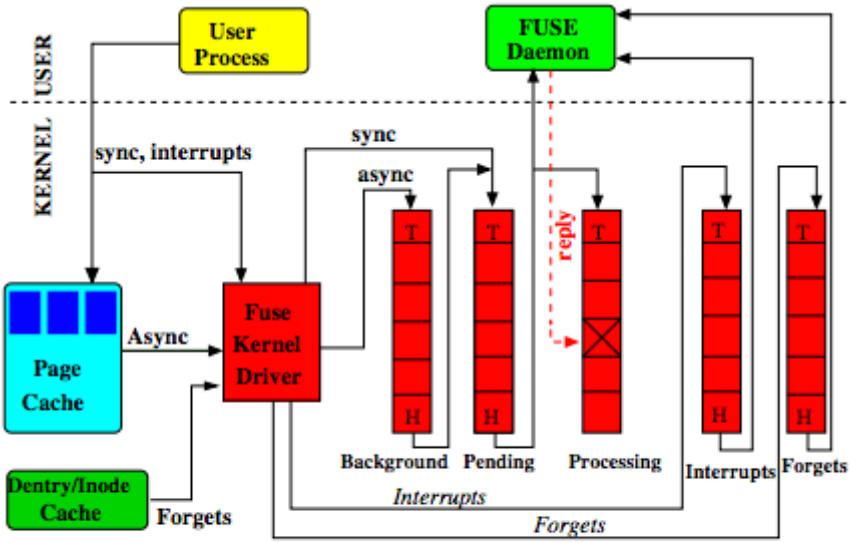


Figure 2: The organization of FUSE queues marked with their Head and Tail. The processing queue does not have a tail because the daemon replies in an arbitrary order.

FUSE在内核中维护了五个队列，分别为：Backgroud、Pending、Processing、Interrupts、Forgets。一个请求在任何时候只会存在于一个队列中。

1) Backgroud：background 队列用于暂存异步请求。在默认情况下，只有读请求进入 background 队列；当writeback cache启用时，写请求也会进入 background 队列。当开启writeback cache时，来自用户进程的写请求会先在页缓存中累积，然后当bdflush 线程被唤醒时会下刷脏页。在下刷脏页时，FUSE会构造异步请求，并将它们放入 background 队列中。

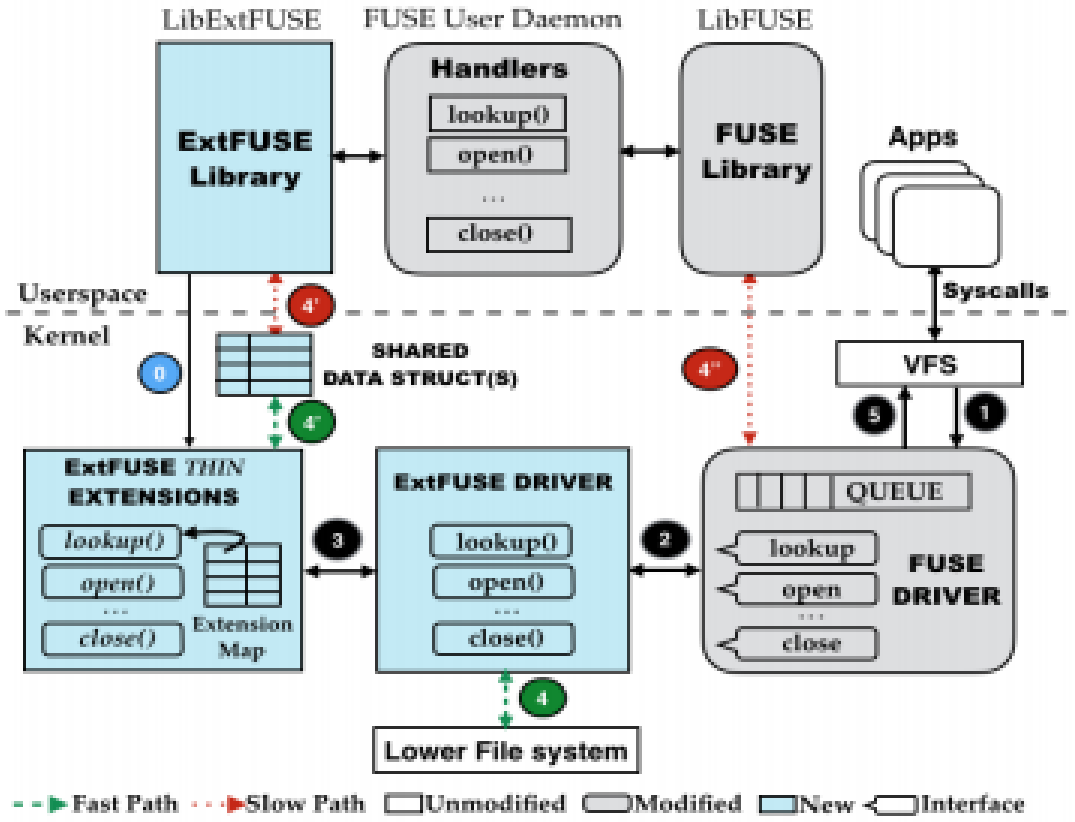
2) Pending：同步请求（例如，元数据）放在 pending 队列中，并且pending队列会周期性接收来自background 的请求。但是pending队列中异步请求的个数最大为max_background（最大为12），当pending队列的异步请求未达到12时，background队列的请求将被移动到pending队列中。这样做的目的是为了控制pending队列中异步请求的个数，防止在突发大量异步请求的情况下，阻塞了同步请求。

3) Processing：当pending队列中的请求被转发到fuse daemon的同时，也被移动到processing队列。所以processing队列中的请求，表示正在被处理fuse daemon处理的请求。当fuse daemon真正处理完请求，通过/dev/fuse下发reply时，该请求将从processing队列中删除。

4) Interrupts：用于存放中断请求，比如当发送的请求被用户取消时，内核会发送一个Interrupts请求，来取消已被发送的请求。中断请求的优先级最高，Interrupts中的请求会最先得到处理。

5) Forgets：forget请求用于删除dcache中缓存的inode。

2.ExtFUSE：使用eBPF优化FUSE的性能



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打包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，隐藏底层实现细节。

libExtFUSE的使用是可选的，独立于lifuse。向libfuse注册的现有文件系统处理程序继续驻留在用户空间中。因此，它们的调用会引起上下文切换，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。我们将内核内执行称为快速路径。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数据结构定义时一般包括以下四个要素：

- map_type: map类型
- key_size: map键大小
- value_size: map值的大小
- max_entries: map的元素最大容量

例如：

```
//下面是通过bpf系统调用函数创建BPF map的方式，传入的第一个参数是BPF_map_CREATE，第二参数是指定将要创建map的属性，第三个参数是这个map配置的大小。因此创建map之前首先要声明一个BPF map，其中有下面要素：
union bpf_attr my_map_attr {
    .map_type = BPF_map_TYPE_ARRAY,
    .key_size = sizeof(int),
    .value_size = sizeof(int),
    .max_entries = 1024,
    .map_flags = BPF_F_NO_PREALLOC,
};

int fd = bpf(BPF_map_CREATE, &my_map_attr, sizeof(my_map_attr));
```

有以下几种类型的eBPF map，包括：

1) Hash map：这是最常用的eBPF映射类型。它们使用哈希表数据结构实现数据的高效存储和检索。

2) Array map：这些映射将数据存储在数组中，并可以使用索引进行访问。

3) Per-CPU map：这些映射允许对每个CPU数据进行高效的存储和检索。它们对于在多个CPU之间并行化工作非常有用。

4) LRU map：这些映射实现了最近最少使用的逐出策略，允许高效地存储和检索频繁访问的数据。

5) Stack map：这些映射允许从栈中推入和弹出数据。

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。

eBPF map的更多信息详见于：(https://blog.csdn.net/qq_18643341/article/details/125233822)

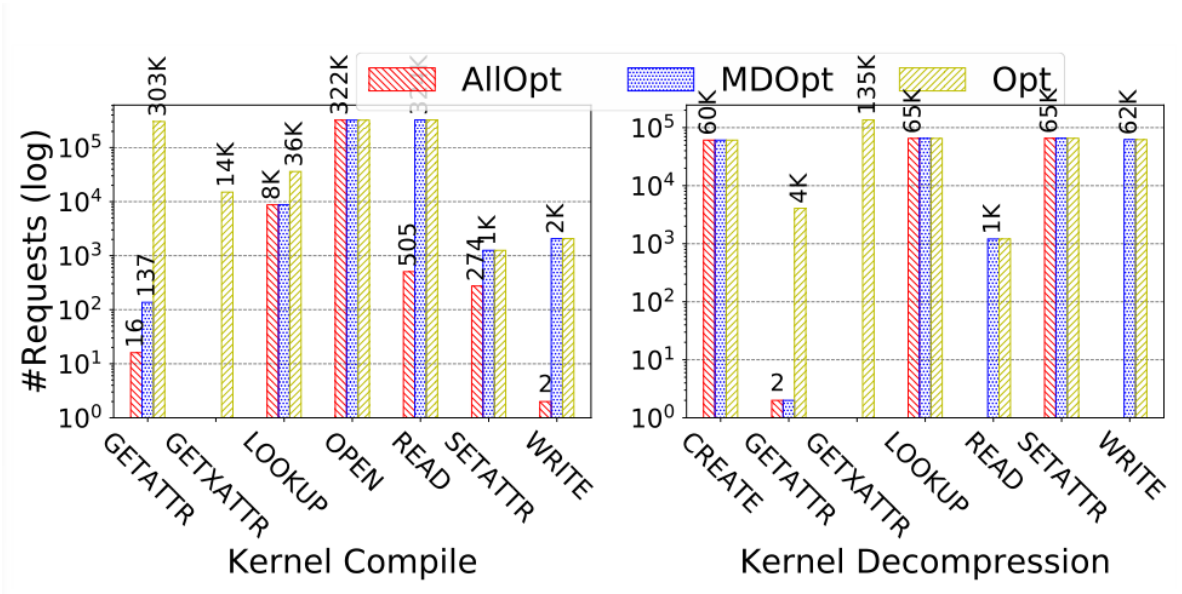
ExtFUSE的工作流程

为了了解ExtFUSE如何促进可扩展用户文件系统的实现，我们详细描述了其工作流。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驱动程序和底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。

性能优化



优化的结果如上图所示。图中显示了对内核做编译操作和解压缩操作的测试情况下监测应用层操作的对比。文件系统使用的是作者开发的用户态文件系统StackFS，AllOpt表示对READ和WRITE操作也在内核中进行了直接返回，而不发到应用层。MDOpt操作表示只是对元数据在内核中直接返回，Opt是原始的FUSE，不做eBPF的优化。

优化结果：

1) 在内核的编译测试中，可以看到getattr，getxattr，lookup的用户态操作都降低了很多。

2) 在内核的解压缩测试中，可以看到需要在用户态处理的getattr，getxattr元数据操作明显减少。

ExtFUSE框架的具体实现

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ebfp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)。不过由于它目前对linux内核和FUSE模块还是有一些改动，增加了使用的难度，降低了使用场景。