

FUSE mostly-passthrough filesystems

Amir Goldstein · CTERA Networks

Linux Plumbers Conference 2026 · BoF · 7 Oct 2026

<https://lpc.events/event/20/contributions/2367/>

Agenda

- Intro – Problem + use cases
- libfuse_passthrough architecture
- Live demo
- Kernel direction (FUSEX)
- **Discussion**

Goal: when to use the library, how modules work, what kernel changes we need.

Intro – The pattern

Many production FUSE filesystems need to:

- Mirror an existing directory tree with **full fidelity**
- Intercept **only a small subset** of operations

They do *not* want to reimplement: lookup, rename, xattr, readdir, mmap, locking, ...

Intro – Case study: cachegwfs

cachegwfs — caching gateway on the **CTERA Edge Filer**: fast local cache in front of a slow tier **cloud filesystem**.

Built as a **monolithic** low-level FUSE server to meet two hard requirements:

- **Performance** on the fast local cache path (passthrough + kernel I/O where possible)
- **NFS export** of the caching gateway mount (persistent handles, daemon restart)

Intro – Case study: cachegwfs (in practice)

The monolith worked, but:

- **Very complex** code — long tail of subtle bugs
- **Hard to maintain**
- **Impossible to share** — highly specialized interfaces tied to CTERA Cloud filesystem

That pushed us toward **libfuse_passthrough** — a reusable passthrough library, separated from the specialized Cloud filesystem module.

Intro – What builders need

- **Small** custom FS code — implement only the ops you intercept
- **Simple** custom FS code — hooks on a module chain, not a monolithic FUSE server
- **NFS export** — persistent file handles, not an open fd per inode
- **Daemon restart** — clients keep working across passthrough daemon reload

Plus: **fast** common-case I/O via kernel passthrough

Intro – What exists today

libfuse examples: `passthrough`, `passthrough_ll`, `passthrough_hp`

→ Great references — you still own **all** FUSE plumbing

Kernel: `FUSE read/write passthrough`

→ Open files can bypass the daemon for I/O

Gap: no library for *passthrough* + *selective hooks* at production quality

→ **libfuse_passthrough**

libfuse_passthrough in one slide

C++ library on libfuse that:

1. Mirrors `source/` at `mountpoint/`
2. Dispatches ops through a **module chain**
3. Falls back to default `*at(2)` passthrough
4. Enables **kernel read/write passthrough** when supported
5. Tracks inodes via **persistent file handles** (`open_by_handle_at`)

```
App → FUSE mount → libfuse_passthrough → [your module] → default → source
                        ↓ kernel passthrough
                    backing file I/O (no daemon)
```

Module chain architecture

High-level libfuse supports a loadable module stack.

Low-level FUSE has no equivalent.

→ `libfuse_passthrough` provides a low-level module chain (not loadable).

- Register a `fuse_passthrough_module`
- Fill `fuse_passthrough_operations` — **only the hooks you need**
- Each hook receives `fuse_path_at`
- Forward with `next_op(op)`
- Per-module states: `get_module_inode_state()` / `set_module_file_state()`

Passthrough fs example

Example: `example/passthrough_fs.cc` + optional `trace` module (`--debug`)

```
fuse_passthrough_module *modules[] = { trace_module() };  
int num_modules = fuse_passthrough_get_opts().debug; // trace when --debug  
  
return fuse_passthrough_main(&args, modules, num_modules,  
                             sizeof(fuse_passthrough_operations));
```

Module examples

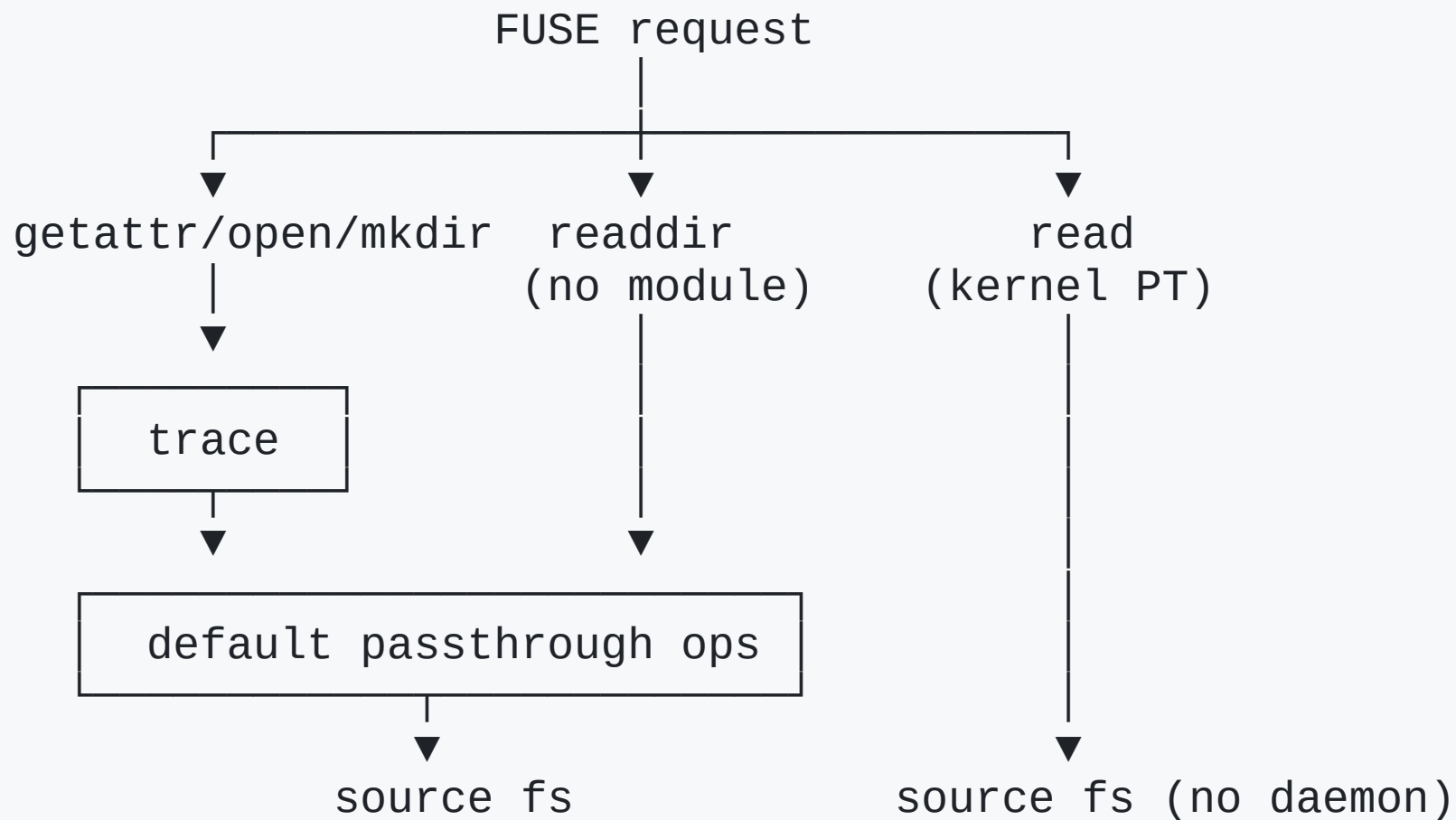
trace module:

```
static int xmp_getattr(const fuse_path_at &at, struct stat *attr,
                      fuse_file_info *fi) {
    trace_fd_path_at(at);           // your logic
    return next_op(getattr)(at, attr, fi); // chain continues
}
```

HSM sketch (whiteboard):

- `lookup()` → manifest stub before `next_op(lookup)`
- `open()` → clear `passthrough_read` if serving from slow tier
- Everything else → automatic passthrough

Execution flow: module · library · kernel



`fuse_path_at`

High-level FUSE: ops take a string path (`"/a/b/c"`).

Low-level FUSE: ops take `nodeid` + `filename` .

libfuse_passthrough: ops take a `fuse_path_at` — the arguments for an `*at(2)` `syscall( dirfd + relative path + at_flags )`.

Modules call syscalls; they do not walk mount-relative paths.

```
fstatat(at.dirfd(), at.path(), &st, at.flags());  
/* e.g. dirfd=inode_fd, path="", flags=AT_EMPTY_PATH */
```

I/O: default is the fast path

By default: read / write / readdir handled by library (or kernel) — **module chain skipped**

To intercept I/O, a module must:

1. Implement `read_buf()` / `write_buf()` (or `readdir()`)
2. At `open()` / `opendir()`, clear `fi->passthrough_read` / `fi->passthrough_write`

Caching / tiering modules **opt in** per file; everything else stays on the fast path.

Kernel read/write passthrough

Originally from Android. Upstream since **Linux 6.9**.

OPEN-time model:

1. Daemon opens backing fd → `fuse_passthrough_open()`
2. Kernel assigns `backing_id`
3. `FUSE_OPEN` reply: `FOPEN_PASSTHROUGH` + `backing_id`
4. `read()` / `write()` → kernel → backing file (**no daemon**)

Requires: `FUSE_CAP_PASSTHROUGH` , `CAP_SYS_ADMIN` for registration

Persistent file handles

Kernel export handles for `open_by_handle_at(2)` — not `fuse_file_info->fh`.

Track inodes without an open fd per inode.

Enables NFS export and daemon restart without breaking client handles.

Today (upstream FUSE_LOOKUP by nodeid): library supports only special-case ext4/XFS handles encoded in userspace.

Falls back to open fds when persistent handles are not available.

FUSEX: LOOKUPX / REaddirPLUSX / MKOBJX identity carries a **file handle**.

Experimental – readdir passthrough

- Opt-in: `readdir_passthrough` (out-of-tree kernel patch today)
- Same `backing_id` model as read/write passthrough
- Conflicts with readdir cache / readdirplus when enabled globally

Application control (proposed): `posix_fadvise()` on the directory fd

- `POSIX_FADV_READDIR_NORMAL` — prefer kernel readdir passthrough
- `POSIX_FADV_READDIR_PLUS` — prefer readdirplus (daemon / attrs path)

Allows workloads to choose per `opendir`, not only at mount time.

Discussion: upstream this path, or fold readdir into later FUSEX work?

Demo

```
mkdir -p /tmp/src/A && echo 'hello world!' > /tmp/src/A/foo  
./build/example/passthrough_fs --debug-fuse --debug --foreground /tmp/src /tmp/mnt  
cat /tmp/mnt/A/foo
```

Compare:

FUSE daemon **without** `CAP_SYS_ADMIN` → FUSE `READ`

FUSE daemon **as root** → kernel passthrough (no `READ`)

Demo – passthrough_fs as normal user

Daemon lacks `CAP_SYS_ADMIN` → passthrough registration fails → **FUSE** `READ` serves data

```
$ cat /tmp/mnt/A/foo
...
DEBUG: xmp_open() @ ... -> /tmp/src/A/foo
fuse: passthrough_open: Operation not permitted          ◀ EPERM
DEBUG: fuse_passthrough_open failed ... disabling kernel passthrough.
    unique: 14, success, outsize: 32                      (OPEN ok, no PT)
dev unique: 16, opcode: READ (15), nodeid: 943638        ◀ daemon read path
    unique: 16, success, outsize: 29
hello world!
dev unique: 18, opcode: RELEASE (18), ...
```

Demo – passthrough_fs as root

Daemon has `CAP_SYS_ADMIN` → passthrough registered → **no FUSE READ** (kernel I/O)

```
$ cat /tmp/mnt/A/foo
...
DEBUG: xmp_open() @ ... -> /tmp/src/A/foo
DEBUG: setup shared backing file 1 for inode 943638      ◀ backing_id
DEBUG: pfs_open(): passthrough_read=1, passthrough_write=1
    unique: 14, success, outsize: 32
hello world!                                           ◀ no FUSE READ
dev unique: 16, opcode: RELEASE (18), ...
DEBUG: closed backing file 1 for inode 943638
```

Where we're headed (FUSEX)

FUSEX **identity** ops (LOOKUPX / REaddirPLUSX / MKOBJX) carry a **file handle**.

How that ties to kernel passthrough (open for discussion):

- **No-open** for regular files — passthrough has to attach at identity, not `FUSE_OPEN`
- **io_uring** can hand the kernel a backing file on that same completion — no extra `BACKING_OPEN` ioctl / `backing_id` round-trip
- Which other ops the kernel should take (passthrough mask)

Library work continues on today's OPEN + `backing_id` model.

Questions for the room

1. Which ops do you need to intercept — and which must stay on the fast path?
2. Need to change kernel passthrough ops during the lifetime of an inode?
3. Is a `posix_fadvise` hint for `readdir` passthrough worth upstreaming?
4. What else is missing for your mostly-passthrough FS?

Get involved

- **Library:** [passthrough/README.md](#)
- **Example:** `example/passthrough_fs.cc` · `modules/trace.cpp`
- **Kernel doc:** `Documentation/filesystems/fuse/fuse-passthrough.rst`

What are you building? Let's discuss.

Related – FUSE BPF

Stacked FUSE still crosses to userspace for metadata even with I/O passthrough.

Alternative: BPF `struct_ops` pre/post filters in the kernel — call VFS on the lower fs without the daemon for most ops.

- Implement only the ops you intercept (like a module chain)
- Fall back to the FUSE daemon when BPF is not enough
- Passthrough association via backing fd (or path/ `*at`)

Reference: [The FUSE BPF filesystem \(LWN\)](#)

Related – fanotify HSM alternative

HSM via FUSE overlay still pays metadata cost even with kernel I/O passthrough.

Alternative: HSM on the local filesystem via fanotify — no overlay.

- Pre-content permission events (`FAN_PRE_ACCESS` / `FAN_PRE_MODIFY`)
- Lookup permission events for directories with evicted content
- Intercept only calls on marked files and directories
- Native filesystem access after handling event

Reference: [Hierarchical Storage Management API](#)

Thank you

Questions?