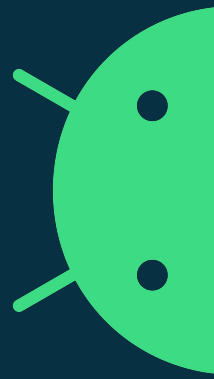
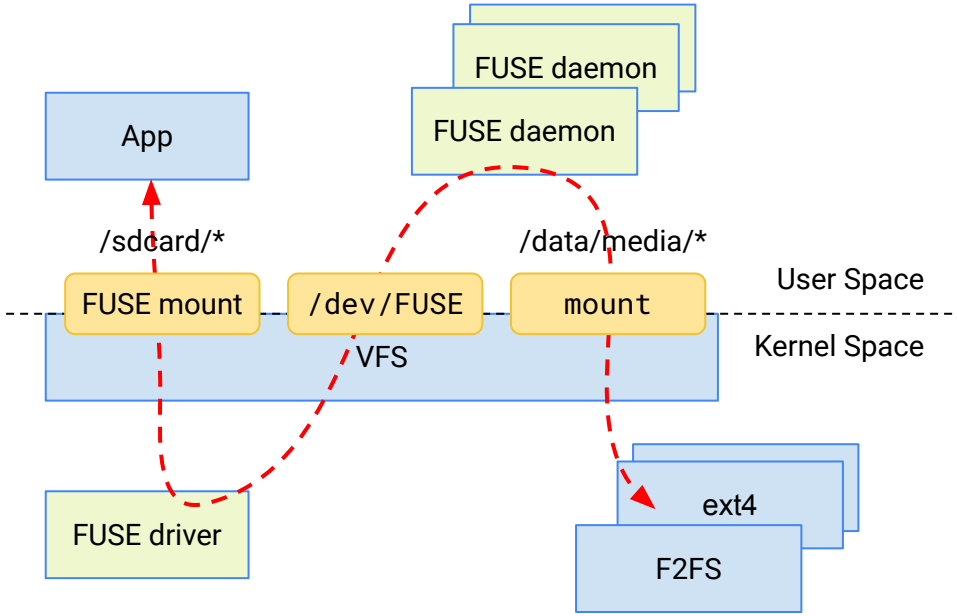
FS stacking with FUSE: performance issues and mitigations

Alessio Balsini <balsini@google.com>
Paul Lawrence <paullawrence@google.com>



Linux Plumbers Conference, 2021

FUSE in Android



Extra permission checks on shared storage access, e.g., only some apps can access some folders

Data redaction, e.g., remove metadata from pictures

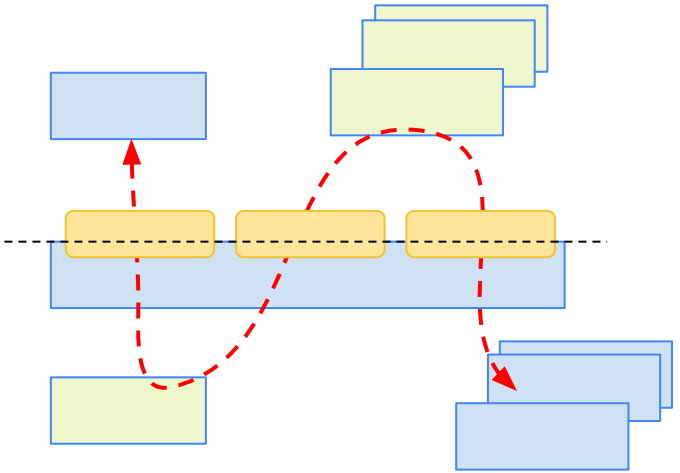
Live transcoding

Emulates the external storage regardless its location, e.g., (un)mounting external storage media

FUSE performance flaws analysis

- Additional FUSE daemon logic
  - That's part of the feature
- Extra VFS traversal
  - 2 file systems are accessed, extra checkings are desirable
  - Double caching for identical FUSE/lower fs files
- Data passing
  - Mostly pointers. Splicing and read-ahead help
  - Writes and rand-read/-writes should be improved
- Long pipeline
  - Communication delay, context switches, user ⟷ kernel switches
- Parallelism
  - Extra locks

...for almost every FUSE file system operation

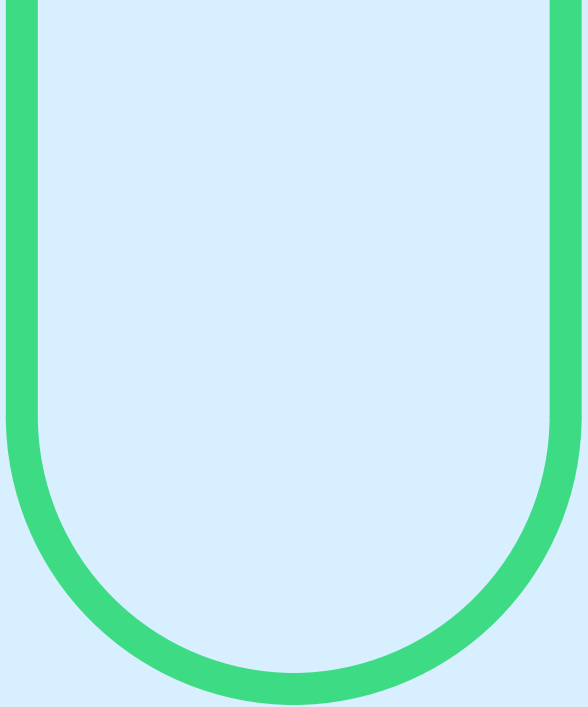


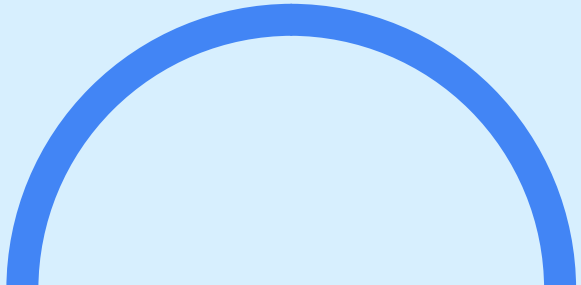
FUSE passthrough

Coming with Android 12

On LKML: V8, V9, V10, V11, V12

https://lore.kernel.org/lkml/20210125153057.3623715-1-balsini@android.com/





android

FUSE passthrough: Read, Write, MMAP

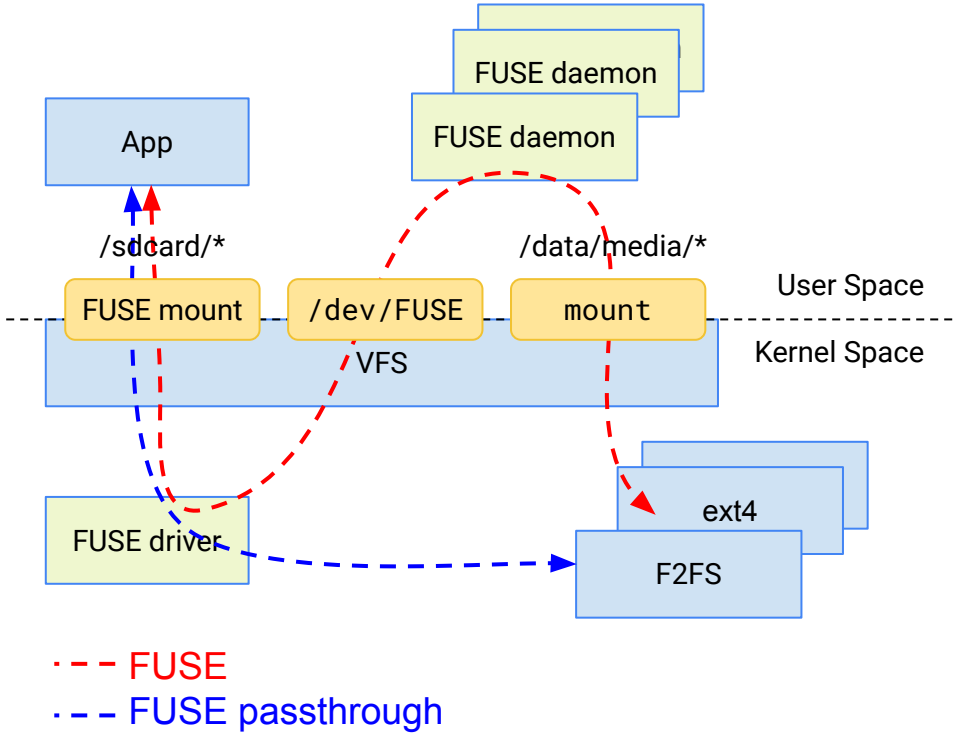
At file open, the FUSE daemon:

- `ioctl("/dev/FUSE", FUSE_PASSTHROUGH_OPEN, lower_fs_fd);`

That `fuse_file` gets a pointer to the lower file pointer

Upcoming read/write/mmap on that file:

- redirected to the lower file system
- use FUSE daemon credentials
- passthrough until close()



Performance in a nutshell: FIO on RAM block device

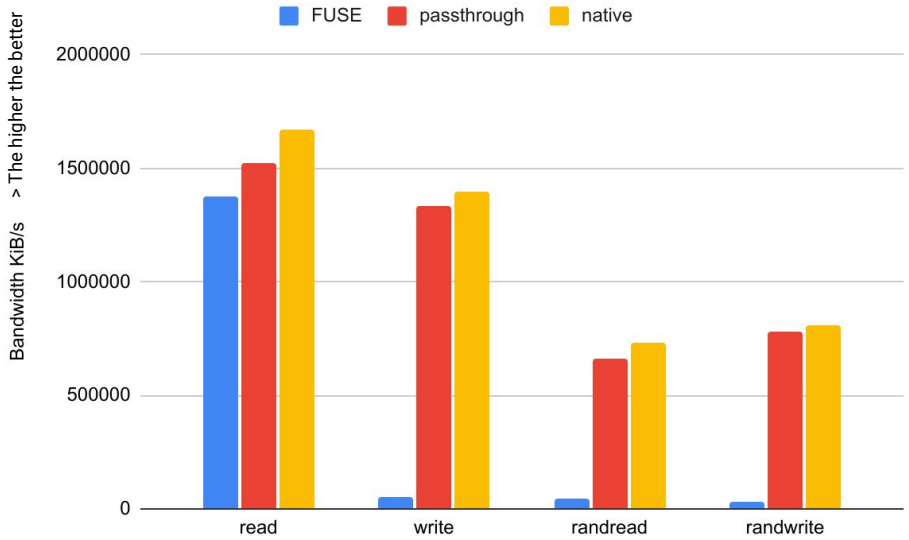
fio-3.23 on RAM device, x86_64, Linux 5.13

- bs=4Ki
- size=20Gi
- ioengine=sync
- fsync_on_close=1
- randseed=0

This highlights FUSE bottlenecks

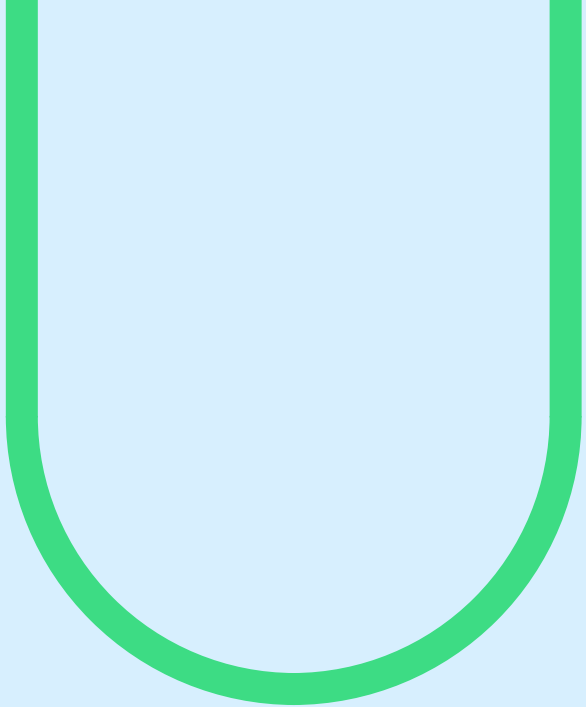
- If we increase the storage device speed, FUSE performance doesn't change!

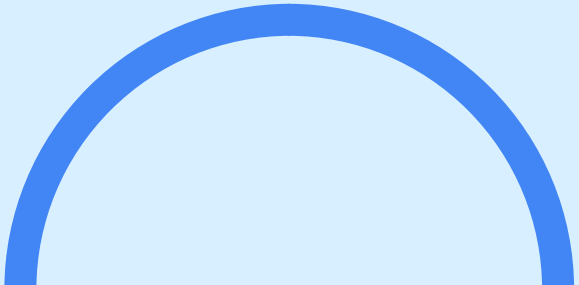
FUSE read performance is the result of good read-ahead



FUSE BPF

Experimental, soon on LKML





android

FUSE BPF: stacking fs + passthrough + extFUSE ?

Implement a **generic stacking file system**

Allow **requests** to either be **handled by FUSE or** the **backing file system**

Allow **pre** and **post filtering** of backing file system request

Filtering can be either **by the kernel**, **or** through FUSE-style requests to **userspace**

Overcome **FUSE passthrough limitations** (per-file, read/write/mmap only)

Inspiration from

- **extFUSE**, presented by Ashish Bijlani at Plumbers in 2019 (https://linuxplumbersconf.org/event/4/contributions/415/)
- **FUSE passthrough**
- Stacking file systems, e.g., **incremental fs**

FUSE BPF at a glance

Add to `fuse_inode`:

- `struct inode *backing_inode;`
- `struct bpf_prog *bpf;`

These may be set at mount time for root, at lookup time for all other inodes

If backing_inode exists, **all** requests will be conditionally sent **to the backing inode**, **else** we are in **classic FUSE** mode

If **no bpf**: simply forward as is (pure **passthrough** mode)

If **bpf**: format `fuse_args` with `in_args` and send to BPF, which may redirect request to **classic FUSE or**

1. Optionally request user-mode pre-filter with same modifiable in_args
2. (Potentially modified) request is sent to backing file system
3. Optionally pass `in_args` & `out_args` to BPF post-filter
4. Optionally pass `in_args` & `out_args` to user-mode post-filter

Early prototypes being tested within Android team

FUSE passthrough

How can we do better for Linux?

Do we really want *FUSE_PASSTHROUGH_CLOSE*?

Can be done with a mapping container (e.g., *IDR*), but is not as simple as *fuse2* (extra spinlocks)

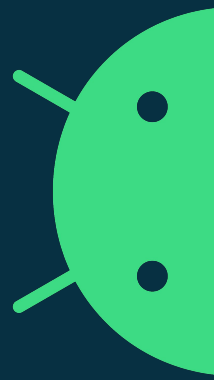
FUSE BPF

BPF is a good compromise between user space and kernel space (good fit for FUSE)

Would the Linux community benefit from this?

Is such architecture upstreamable?

Thanks!
Questions?



FS stacking with FUSE: performance issues and mitigations

Alessio Balsini <balsini@google.com>
Paul Lawrence <paullawrence@google.com>

Linux Plumbers Conference, 2021