Rocky Enterprise Linux 9.2 Manual Pages on command 'fuse.4'

$ man fuse.4

FUSE(4)   Linux Programmer's Manual   FUSE(4)

NAME

    fuse - Filesystem in Userspace (FUSE) device

SYNOPSIS

    #include <linux/fuse.h>

DESCRIPTION

    This device is the primary interface between the FUSE filesystem driver

    and a user-space process wishing to provide the filesystem (referred to

    in the rest of this manual page as the filesystem daemon). This manual

    page is intended for those interested in understanding the kernel in‐

    terface itself. Those implementing a FUSE filesystem may wish to make

    use of a user-space library such as libfuse that abstracts away the

    low-level interface.

    At its core, FUSE is a simple client-server protocol, in which the

    Linux kernel is the client and the daemon is the server. After obtain‐

    ing a file descriptor for this device, the daemon may read(2) requests

    from that file descriptor and is expected to write(2) back its replies.

    It is important to note that a file descriptor is associated with a

unique FUSE filesystem. In particular, opening a second copy of this
device, will not allow access to resources created through the first
file descriptor (and vice versa).

The basic protocol

Every message that is read by the daemon begins with a header described
by the following structure:

```
struct fuse_in_header {
    uint32_t len;      /* Total length of the data,

                          including this header */
    uint32_t opcode;    /* The kind of operation (see below) */
    uint64_t unique;    /* A unique identifier for this request */
    uint64_t nodeid;    /* ID of the filesystem object

                          being operated on */
    uint32_t uid;       /* UID of the requesting process */
    uint32_t gid;       /* GID of the requesting process */
    uint32_t pid;       /* PID of the requesting process */
    uint32_t padding;
};
```

The header is followed by a variable-length data portion (which may be
empty) specific to the requested operation (the requested operation is
indicated by opcode).

The daemon should then process the request and if applicable send a re?
ply (almost all operations require a reply; if they do not, this is
documented below), by performing a write(2) to the file descriptor.

All replies must start with the following header:

```
struct fuse_out_header {
    uint32_t len;      /* Total length of data written to

                          the file descriptor */
    int32_t  error;    /* Any error that occurred (0 if none) */
    uint64_t unique;   /* The value from the

                          corresponding request */
};
```

This header is also followed by (potentially empty) variable-sized data

depending on the executed request. However, if the reply is an error reply (i.e., error is set), then no further payload data should be sent, independent of the request.

Exchanged messages

This section should contain documentation for each of the messages in the protocol. This manual page is currently incomplete, so not all messages are documented. For each message, first the struct sent by the kernel is given, followed by a description of the semantics of the message.

FUSE_INIT

```
struct fuse_init_in {
    uint32_t major;
    uint32_t minor;
    uint32_t max_readahead; /* Since protocol v7.6 */
    uint32_t flags;        /* Since protocol v7.6 */
};
```

This is the first request sent by the kernel to the daemon. It is used to negotiate the protocol version and other filesystem parameters. Note that the protocol version may affect the lay‐out of any structure in the protocol (including this structure).

The daemon must thus remember the negotiated version and flags for each session. As of the writing of this man page, the high‐est supported kernel protocol version is 7.26.

Users should be aware that the descriptions in this manual page may be incomplete or incorrect for older or more recent protocol versions.

The reply for this request has the following format:

```
struct fuse_init_out {
    uint32_t major;
    uint32_t minor;
    uint32_t max_readahead;  /* Since v7.6 */
    uint32_t flags;         /* Since v7.6; some flags bits
                            were introduced later */
```

```
        uint16_t max_background;  /* Since v7.13 */

        uint16_t congestion_threshold;  /* Since v7.13 */

        uint32_t max_write;       /* Since v7.5 */

        uint32_t time_gran;       /* Since v7.6 */

        uint32_t unused[9];

    };
```

If the major version supported by the kernel is larger than that
supported by the daemon, the reply shall consist of only
uint32_t major (following the usual header), indicating the
largest major version supported by the daemon. The kernel will
then issue a new FUSE_INIT request conforming to the older ver?
sion. In the reverse case, the daemon should quietly fall back
to the kernel's major version.

The negotiated minor version is considered to be the minimum of
the minor versions provided by the daemon and the kernel and
both parties should use the protocol corresponding to said minor
version.

FUSE_GETATTR

```
    struct fuse_getattr_in {

      uint32_t getattr_flags;

      uint32_t dummy;

      uint64_t fh;     /* Set only if

                  (getattr_flags & FUSE_GETATTR_FH)

    };
```

The requested operation is to compute the attributes to be re?
turned by stat(2) and similar operations for the given filesys?
tem object. The object for which the attributes should be com?
puted is indicated either by header->nodeid or, if the
FUSE_GETATTR_FH flag is set, by the file handle fh. The latter
case of operation is analogous to fstat(2).

For performance reasons, these attributes may be cached in the
kernel for a specified duration of time. While the cache time?
out has not been exceeded, the attributes will be served from

the cache and will not cause additional FUSE_GETATTR requests.

The computed attributes and the requested cache timeout should
then be returned in the following structure:

```
struct fuse_attr_out {
    /* Attribute cache duration (seconds + nanoseconds) */
    uint64_t attr_valid;
    uint32_t attr_valid_nsec;
    uint32_t dummy;
    struct fuse_attr {
        uint64_t ino;
        uint64_t size;
        uint64_t blocks;
        uint64_t atime;
        uint64_t mtime;
        uint64_t ctime;
        uint32_t atimensec;
        uint32_t mtimensec;
        uint32_t ctimensec;
        uint32_t mode;
        uint32_t nlink;
        uint32_t uid;
        uint32_t gid;
        uint32_t rdev;
        uint32_t blksize;
        uint32_t padding;
    } attr;
};
```

FUSE_ACCESS

```
struct fuse_access_in {
    uint32_t mask;
    uint32_t padding;
};
```

If the default_permissions mount options is not used, this re?

quest may be used for permissions checking. No reply data is expected, but errors may be indicated as usual by setting the error field in the reply header (in particular, access denied errors may be indicated by returning -EACCES).

FUSE_OPEN and FUSE_OPENDIR

```
struct fuse_open_in {
    uint32_t flags;     /* The flags that were passed
                    to the open(2) */
    uint32_t unused;
};
```

The requested operation is to open the node indicated by header->nodeid. The exact semantics of what this means will de?pend on the filesystem being implemented. However, at the very least the filesystem should validate that the requested flags are valid for the indicated resource and then send a reply with the following format:

```
struct fuse_open_out {
    uint64_t fh;
    uint32_t open_flags;
    uint32_t padding;
};
```

The fh field is an opaque identifier that the kernel will use to refer to this resource The open_flags field is a bit mask of any number of the flags that indicate properties of this file handle to the kernel:

FOPEN_DIRECT_IO   Bypass page cache for this open file.

FOPEN_KEEP_CACHE Don't invalidate the data cache on open.

FOPEN_NONSEEKABLE The file is not seekable.

FUSE_READ and FUSE_READDIR

```
struct fuse_read_in {
    uint64_t fh;
    uint64_t offset;
    uint32_t size;
```

```
            uint32_t read_flags;

            uint64_t lock_owner;

            uint32_t flags;

            uint32_t padding;

        };
```

The requested action is to read up to size bytes of the file or

directory, starting at offset. The bytes should be returned di‐

rectly following the usual reply header.

FUSE_INTERRUPT

```
        struct fuse_interrupt_in {

            uint64_t unique;

        };
```

The requested action is to cancel the pending operation indi‐

cated by unique. This request requires no response.   However,

receipt of this message does not by itself cancel the indicated

operation. The kernel will still expect a reply to said opera‐

tion (e.g., an EINTR error or a short read). At most one

FUSE_INTERRUPT request will be issued for a given operation.

After issuing said operation, the kernel will wait uninterrupt‐

ibly for completion of the indicated request.

FUSE_LOOKUP

Directly following the header is a filename to be looked up in

the directory indicated by header->nodeid. The expected reply

is of the form:

```
        struct fuse_entry_out {

            uint64_t nodeid;          /* Inode ID */

            uint64_t generation;       /* Inode generation */

            uint64_t entry_valid;

            uint64_t attr_valid;

            uint32_t entry_valid_nsec;

            uint32_t attr_valid_nsec;

            struct fuse_attr attr;

        };
```

The combination of nodeid and generation must be unique for the filesystem's lifetime.

The interpretation of timeouts and attr is as for FUSE_GETATTR.

FUSE_FLUSH

```
struct fuse_flush_in {
    uint64_t fh;
    uint32_t unused;
    uint32_t padding;
    uint64_t lock_owner;
};
```

The requested action is to flush any pending changes to the in‐dicated file handle. No reply data is expected. However, an empty reply message still needs to be issued once the flush op‐eration is complete.

FUSE_RELEASE and FUSE_RELEASEDIR

```
struct fuse_release_in {
    uint64_t fh;
    uint32_t flags;
    uint32_t release_flags;
    uint64_t lock_owner;
};
```

These are the converse of FUSE_OPEN and FUSE_OPENDIR respec‐tively. The daemon may now free any resources associated with the file handle fh as the kernel will no longer refer to it. There is no reply data associated with this request, but a reply still needs to be issued once the request has been completely processed.

FUSE_STATFS

This operation implements statfs(2) for this filesystem. There is no input data associated with this request. The expected re‐ply data has the following structure:

```
struct fuse_kstatfs {
    uint64_t blocks;
```

```
        uint64_t bfree;

        uint64_t bavail;

        uint64_t files;

        uint64_t ffree;

        uint32_t bsize;

        uint32_t namelen;

        uint32_t frsize;

        uint32_t padding;

        uint32_t spare[6];

    };

    struct fuse_statfs_out {

        struct fuse_kstatfs st;

    };
```

For the interpretation of these fields, see statfs(2).

ERRORS

E2BIG Returned from read(2) operations when the kernel's request is
       too large for the provided buffer and the request was FUSE_SETX?
       ATTR.

EINVAL Returned from write(2) if validation of the reply failed. Not
       all mistakes in replies will be caught by this validation. How?
       ever, basic mistakes, such as short replies or an incorrect
       unique value, are detected.

EIO   Returned from read(2) operations when the kernel's request is
       too large for the provided buffer.
       Note: There are various ways in which incorrect use of these in?
       terfaces can cause operations on the provided filesystem's files
       and directories to fail with EIO. Among the possible incorrect
       uses are:
       * changing mode & S_IFMT for an inode that has previously been
          reported to the kernel; or
       * giving replies to the kernel that are shorter than what the
          kernel expected.

ENODEV Returned from read(2) and write(2) if the FUSE filesystem was

unmounted.

EPERM Returned from operations on a /dev/fuse file descriptor that has

not been mounted.

CONFORMING TO

The FUSE filesystem is Linux-specific.

NOTES

The following messages are not yet documented in this manual page:

FUSE_BATCH_FORGET

FUSE_BMAP

FUSE_CREATE

FUSE_DESTROY

FUSE_FALLOCATE

FUSE_FORGET

FUSE_FSYNC

FUSE_FSYNCDIR

FUSE_GETLK

FUSE_GETXATTR

FUSE_IOCTL

FUSE_LINK

FUSE_LISTXATTR

FUSE_LSEEK

FUSE_MKDIR

FUSE_MKNOD

FUSE_NOTIFY_REPLY

FUSE_POLL

FUSE_READDIRPLUS

FUSE_READLINK

FUSE_REMOVEXATTR

FUSE_RENAME

FUSE_RENAME2

FUSE_RMDIR

FUSE_SETATTR

FUSE_SETLK

FUSE_SETLKW

FUSE_SYMLINK

FUSE_UNLINK

FUSE_WRITE

SEE ALSO

fusermount(1), mount.fuse(8)

COLOPHON

This page is part of release 5.10 of the Linux man-pages project.   A
description of the project, information about reporting bugs, and the
latest   version   of   this   page,   can   be   found   at
https://www.kernel.org/doc/man-pages/.

Linux   2018-02-02   FUSE(4)