



Rocky Enterprise Linux 9.2 Manual Pages on command 'fanotify.7'

C:\>man fanotify.7

FANOTIFY(7) Linux Programmer's Manual FANOTIFY(7)

NAME

fanotify - monitoring filesystem events

DESCRIPTION

The fanotify API provides notification and interception of filesystem events. Use cases include virus scanning and hierarchical storage management. Currently, only a limited set of events is supported. In particular, there is no support for create, delete, and move events. (See inotify(7) for details of an API that does not notify those events.)

Additional capabilities compared to the inotify(7) API include the ability to monitor all of the objects in a mounted filesystem, the ability to make access permission decisions, and the possibility to read or modify files before access by other applications.

The following system calls are used with this API: fanotify_init(2), fanotify_mark(2), read(2), write(2), and close(2).

fanotify_init(), fanotify_mark(), and notification groups

The fanotify_init(2) system call creates and initializes an fanotify notification group and returns a file descriptor referring to it.

An fanotify notification group is a kernel-internal object that holds a list of files, directories, filesystems, and mount points for which events shall be created.

For each entry in an fanotify notification group, two bit masks exist: the mark

mask and the ignore mask. The mark mask defines file activities for which an event shall be created. The ignore mask defines activities for which no event shall be generated. Having these two types of masks permits a filesystem, mount point, or directory to be marked for receiving events, while at the same time ignoring events for specific objects under a mount point or directory.

The `fanotify_mark(2)` system call adds a file, directory, filesystem or mount point to a notification group and specifies which events shall be reported (or ignored), or removes or modifies such an entry.

A possible usage of the ignore mask is for a file cache. Events of interest for a file cache are modification of a file and closing of the same. Hence, the cached directory or mount point is to be marked to receive these events. After receiving the first event informing that a file has been modified, the corresponding cache entry will be invalidated. No further modification events for this file are of interest until the file is closed. Hence, the modify event can be added to the ignore mask. Upon receiving the close event, the modify event can be removed from the ignore mask and the file cache entry can be updated.

The entries in the fanotify notification groups refer to files and directories via their inode number and to mounts via their mount ID. If files or directories are renamed or moved within the same mount, the respective entries survive. If files or directories are deleted or moved to another mount or if filesystems or mounts are unmounted, the corresponding entries are deleted.

The event queue

As events occur on the filesystem objects monitored by a notification group, the fanotify system generates events that are collected in a queue. These events can then be read (using `read(2)` or similar) from the fanotify file descriptor returned by `fanotify_init(2)`.

Two types of events are generated: notification events and permission events. Notification events are merely informative and require no action to be taken by the receiving application with the exception being that the file descriptor provided within a generic event must be closed. The closing of file descriptors for each event applies only to applications that have initialized fanotify without using `FAN_REPORT_FID` (see below). Permission events are requests to the receiving application to decide whether permission for a file access shall be granted. For these

events, the recipient must write a response which decides whether access is granted or not.

An event is removed from the event queue of the fanotify group when it has been read. Permission events that have been read are kept in an internal list of the fanotify group until either a permission decision has been taken by writing to the fanotify file descriptor or the fanotify file descriptor is closed.

Reading fanotify events

Calling `read(2)` for the file descriptor returned by `fanotify_init(2)` blocks (if the flag `FAN_NONBLOCK` is not specified in the call to `fanotify_init(2)`) until either a file event occurs or the call is interrupted by a signal (see `signal(7)`).

The use of the `FAN_REPORT_FID` flag in `fanotify_init(2)` influences what data structures are returned to the event listener for each event. After a successful `read(2)`, the read buffer contains one or more of the following structures:

```
struct fanotify_event_metadata {
    __u32 event_len;
    __u8 vers;
    __u8 reserved;
    __u16 metadata_len;
    __aligned_u64 mask;
    __s32 fd;
    __s32 pid;
};
```

In the case where `FAN_REPORT_FID` is supplied as one of the flags to `fanotify_init(2)`, you should also expect to receive the structure detailed below following the generic `fanotify_event_metadata` structure within the read buffer:

```
struct fanotify_event_info_fid {
    struct fanotify_event_info_header hdr;
    __kernel_fsid_t fsid;
    unsigned char file_handle[0];
};
```

For performance reasons, it is recommended to use a large buffer size (for example, 4096 bytes), so that multiple events can be retrieved by a single `read(2)`.

The return value of `read(2)` is the number of bytes placed in the buffer, or -1 in

case of an error (but see BUGS).

The fields of the `fanotify_event_metadata` structure are as follows:

`event_len`

This is the length of the data for the current event and the offset to the next event in the buffer. Without `FAN_REPORT_FID`, the value of `event_len` is always `FAN_EVENT_METADATA_LEN`. With `FAN_REPORT_FID`, `event_len` also includes the variable length file identifier.

`vers` This field holds a version number for the structure. It must be compared to `FANOTIFY_METADATA_VERSION` to verify that the structures returned at run time match the structures defined at compile time. In case of a mismatch, the application should abandon trying to use the fanotify file descriptor.

`reserved`

This field is not used.

`metadata_len`

This is the length of the structure. The field was introduced to facilitate the implementation of optional headers per event type. No such optional headers exist in the current implementation.

`mask` This is a bit mask describing the event (see below).

`fd` This is an open file descriptor for the object being accessed, or `FAN_NOFD` if a queue overflow occurred. If the fanotify file descriptor has been initialized using `FAN_REPORT_FID`, applications should expect this value to be set to `FAN_NOFD` for each event that is received. The file descriptor can be used to access the contents of the monitored file or directory. The reading application is responsible for closing this file descriptor.

When calling `fanotify_init(2)`, the caller may specify (via the `event_f_flags` argument) various file status flags that are to be set on the open file descriptor that corresponds to this file descriptor. In addition, the (kernel-internal) `FMODE_NONOTIFY` file status flag is set on the open file descriptor. This flag suppresses fanotify event generation. Hence, when the receiver of the fanotify event accesses the notified file or directory using this file descriptor, no additional events will be created.

`pid` If flag `FAN_REPORT_TID` was set in `fanotify_init(2)`, this is the TID of the thread that caused the event. Otherwise, this the PID of the process that

caused the event.

A program listening to fanotify events can compare this PID to the PID returned by `getpid(2)`, to determine whether the event is caused by the listener itself, or is due to a file access by another process.

The bit mask in `mask` indicates which events have occurred for a single filesystem object. Multiple bits may be set in this mask, if more than one event occurred for the monitored filesystem object. In particular, consecutive events for the same filesystem object and originating from the same process may be merged into a single event, with the exception that two permission events are never merged into one queue entry.

The bits that may appear in `mask` are as follows:

FAN_ACCESS

A file or a directory (but see BUGS) was accessed (read).

FAN_OPEN

A file or a directory was opened.

FAN_OPEN_EXEC

A file was opened with the intent to be executed. See NOTES in `fanotify_mark(2)` for additional details.

FAN_ATTRIB

A file or directory metadata was changed.

FAN_CREATE

A child file or directory was created in a watched parent.

FAN_DELETE

A child file or directory was deleted in a watched parent.

FAN_DELETE_SELF

A watched file or directory was deleted.

FAN_MOVED_FROM

A file or directory has been moved from a watched parent directory.

FAN_MOVED_TO

A file or directory has been moved to a watched parent directory.

FAN_MOVE_SELF

A watched file or directory was moved.

FAN_MODIFY

A file was modified.

FAN_CLOSE_WRITE

A file that was opened for writing (O_WRONLY or O_RDWR) was closed.

FAN_CLOSE_NOWRITE

A file or directory that was opened read-only (O_RDONLY) was closed.

FAN_Q_OVERFLOW

The event queue exceeded the limit of 16384 entries. This limit can be overridden by specifying the FAN_UNLIMITED_QUEUE flag when calling fanotify_init(2).

FAN_ACCESS_PERM

An application wants to read a file or directory, for example using read(2) or readdir(2). The reader must write a response (as described below) that determines whether the permission to access the filesystem object shall be granted.

FAN_OPEN_PERM

An application wants to open a file or directory. The reader must write a response that determines whether the permission to open the filesystem object shall be granted.

FAN_OPEN_EXEC_PERM

An application wants to open a file for execution. The reader must write a response that determines whether the permission to open the filesystem object for execution shall be granted. See NOTES in fanotify_mark(2) for additional details.

To check for any close event, the following bit mask may be used:

FAN_CLOSE

A file was closed. This is a synonym for:

FAN_CLOSE_WRITE | FAN_CLOSE_NOWRITE

To check for any move event, the following bit mask may be used:

FAN_MOVE

A file or directory was moved. This is a synonym for:

FAN_MOVED_FROM | FAN_MOVED_TO

The fields of the fanotify_event_info_fid structure are as follows:

hdr This is a structure of type fanotify_event_info_header. It is a generic

header that contains information used to describe additional information attached to the event. For example, when a fanotify file descriptor is created using `FAN_REPORT_FID`, the `info_type` field of this header is set to `FAN_EVENT_INFO_TYPE_FID`. Event listeners can use this field to check that the additional information received for an event is of the correct type. Additionally, the `fanotify_event_info_header` also contains a `len` field. In the current implementation, the value of `len` is always `(event_len - FAN_EVENT_METADATA_LEN)`.

`fsid` This is a unique identifier of the filesystem containing the object associated with the event. It is a structure of type `__kernel_fsid_t` and contains the same value as `f_fsid` when calling `statfs(2)`.

`file_handle`

This is a variable length structure of type `file_handle`. It is an opaque handle that corresponds to a specified object on a filesystem as returned by `name_to_handle_at(2)`. It can be used to uniquely identify a file on a filesystem and can be passed as an argument to `open_by_handle_at(2)`. Note that for directory entry events, such as `FAN_CREATE`, `FAN_DELETE`, and `FAN_MOVE`, the `file_handle` describes the modified directory and not the created/deleted/moved child object. The events `FAN_ATTRIB`, `FAN_DELETE_SELF`, and `FAN_MOVE_SELF` will carry the `file_handle` information for the child object if the child object is being watched.

The following macros are provided to iterate over a buffer containing fanotify event metadata returned by a `read(2)` from an fanotify file descriptor:

`FAN_EVENT_OK(meta, len)`

This macro checks the remaining length `len` of the buffer `meta` against the length of the metadata structure and the `event_len` field of the first metadata structure in the buffer.

`FAN_EVENT_NEXT(meta, len)`

This macro uses the length indicated in the `event_len` field of the metadata structure pointed to by `meta` to calculate the address of the next metadata structure that follows `meta`. `len` is the number of bytes of metadata that currently remain in the buffer. The macro returns a pointer to the next metadata structure that follows `meta`, and reduces `len` by the number of bytes

in the metadata structure that has been skipped over (i.e., it subtracts meta->event_len from len).

In addition, there is:

FAN_EVENT_METADATA_LEN

This macro returns the size (in bytes) of the structure fanotify_event_metadata. This is the minimum size (and currently the only size) of any event metadata.

Monitoring an fanotify file descriptor for events

When an fanotify event occurs, the fanotify file descriptor indicates as readable when passed to `epoll(7)`, `poll(2)`, or `select(2)`.

Dealing with permission events

For permission events, the application must write(2) a structure of the following form to the fanotify file descriptor:

```
struct fanotify_response {
    __s32 fd;
    __u32 response;
};
```

The fields of this structure are as follows:

`fd` This is the file descriptor from the structure `fanotify_event_metadata`.

`response`

This field indicates whether or not the permission is to be granted. Its value must be either `FAN_ALLOW` to allow the file operation or `FAN_DENY` to deny the file operation.

If access is denied, the requesting application call will receive an `EPERM` error.

Closing the fanotify file descriptor

When all file descriptors referring to the fanotify notification group are closed, the fanotify group is released and its resources are freed for reuse by the kernel.

Upon `close(2)`, outstanding permission events will be set to allowed.

/proc/[pid]/fdinfo

The file `/proc/[pid]/fdinfo/[fd]` contains information about fanotify marks for file descriptor `fd` of process `pid`. See `proc(5)` for details.

ERRORS

In addition to the usual errors for `read(2)`, the following errors can occur when

reading from the fanotify file descriptor:

EINVAL The buffer is too small to hold the event.

EMFILE The per-process limit on the number of open files has been reached. See the description of `RLIMIT_NOFILE` in `getrlimit(2)`.

ENFILE The system-wide limit on the total number of open files has been reached.

See `/proc/sys/fs/file-max` in `proc(5)`.

ETXTBSY

This error is returned by `read(2)` if `O_RDWR` or `O_WRONLY` was specified in the `event_f_flags` argument when calling `fanotify_init(2)` and an event occurred for a monitored file that is currently being executed.

In addition to the usual errors for `write(2)`, the following errors can occur when writing to the fanotify file descriptor:

EINVAL Fanotify access permissions are not enabled in the kernel configuration or the value of `response` in the response structure is not valid.

ENOENT The file descriptor `fd` in the response structure is not valid. This may occur when a response for the permission event has already been written.

VERSIONS

The fanotify API was introduced in version 2.6.36 of the Linux kernel and enabled in version 2.6.37. `Fdinfo` support was added in version 3.8.

CONFORMING TO

The fanotify API is Linux-specific.

NOTES

The fanotify API is available only if the kernel was built with the `CONFIG_FANOTIFY` configuration option enabled. In addition, fanotify permission handling is available only if the `CONFIG_FANOTIFY_ACCESS_PERMISSIONS` configuration option is enabled.

Limitations and caveats

Fanotify reports only events that a user-space program triggers through the filesystem API. As a result, it does not catch remote events that occur on network filesystems.

The fanotify API does not report file accesses and modifications that may occur because of `mmap(2)`, `msync(2)`, and `munmap(2)`.

Events for directories are created only if the directory itself is opened, read,

and closed. Adding, removing, or changing children of a marked directory does not create events for the monitored directory itself.

Fanotify monitoring of directories is not recursive: to monitor subdirectories under a directory, additional marks must be created. (But note that the fanotify API provides no way of detecting when a subdirectory has been created under a marked directory, which makes recursive monitoring difficult.) Monitoring mounts offers the capability to monitor a whole directory tree. Monitoring filesystems offers the capability to monitor changes made from any mount of a filesystem instance. The event queue can overflow. In this case, events are lost.

BUGS

Before Linux 3.19, `fallocate(2)` did not generate fanotify events. Since Linux 3.19, calls to `fallocate(2)` generate `FAN_MODIFY` events.

As of Linux 3.17, the following bugs exist:

- * On Linux, a filesystem object may be accessible through multiple paths, for example, a part of a filesystem may be remounted using the `--bind` option of `mount(8)`. A listener that marked a mount will be notified only of events that were triggered for a filesystem object using the same mount. Any other event will pass unnoticed.
- * When an event is generated, no check is made to see whether the user ID of the receiving process has authorization to read or write the file before passing a file descriptor for that file. This poses a security risk, when the `CAP_SYS_ADMIN` capability is set for programs executed by unprivileged users.
- * If a call to `read(2)` processes multiple events from the fanotify queue and an error occurs, the return value will be the total length of the events successfully copied to the user-space buffer before the error occurred. The return value will not be `-1`, and `errno` will not be set. Thus, the reading application has no way to detect the error.

EXAMPLE

The two example programs below demonstrate the usage of the fanotify API.

Example program: `fanotify_example.c`

The first program is an example of fanotify being used with its event object information passed in the form of a file descriptor. The program marks the mount point passed as a command-line argument and waits for events of type `FAN_OPEN_PERM` and

FAN_CLOSE_WRITE. When a permission event occurs, a FAN_ALLOW response is given.

The following shell session shows an example of running this program. This session involved editing the file /home/user/temp/notes. Before the file was opened, a FAN_OPEN_PERM event occurred. After the file was closed, a FAN_CLOSE_WRITE event occurred. Execution of the program ends when the user presses the ENTER key.

```
# ./fanotify_example /home
Press enter key to terminate.
Listening for events.
FAN_OPEN_PERM: File /home/user/temp/notes
FAN_CLOSE_WRITE: File /home/user/temp/notes
Listening for events stopped.
```

Program source: fanotify_example.c

```
#define _GNU_SOURCE /* Needed to get O_LARGEFILE definition */
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <poll.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/fanotify.h>
#include <unistd.h>

/* Read all available fanotify events from the file descriptor 'fd' */
static void
handle_events(int fd)
{
    const struct fanotify_event_metadata *metadata;
    struct fanotify_event_metadata buf[200];
    ssize_t len;
    char path[PATH_MAX];
    ssize_t path_len;
    char procf_path[PATH_MAX];
    struct fanotify_response response;

    /* Loop while events can be read from fanotify file descriptor */
```

```

for (;;) {
    /* Read some events */
    len = read(fd, (void *) &buf, sizeof(buf));
    if (len == -1 && errno != EAGAIN) {
        perror("read");
        exit(EXIT_FAILURE);
    }
    /* Check if end of available data reached */
    if (len <= 0)
        break;
    /* Point to the first event in the buffer */
    metadata = buf;
    /* Loop over all events in the buffer */
    while (FAN_EVENT_OK(metadata, len)) {
        /* Check that run-time and compile-time structures match */
        if (metadata->vers != FANOTIFY_METADATA_VERSION) {
            fprintf(stderr,
                "Mismatch of fanotify metadata version.\n");
            exit(EXIT_FAILURE);
        }
        /* metadata->fd contains either FAN_NOFD, indicating a
           queue overflow, or a file descriptor (a nonnegative
           integer). Here, we simply ignore queue overflow. */
        if (metadata->fd >= 0) {
            /* Handle open permission event */
            if (metadata->mask & FAN_OPEN_PERM) {
                printf("FAN_OPEN_PERM: ");
                /* Allow file to be opened */
                response.fd = metadata->fd;
                response.response = FAN_ALLOW;
                write(fd, &response,
                    sizeof(struct fanotify_response));
            }
        }
    }
}

```

```

/* Handle closing of writable file event */
if (metadata->mask & FAN_CLOSE_WRITE)
    printf("FAN_CLOSE_WRITE: ");
/* Retrieve and print pathname of the accessed file */
snprintf(procfd_path, sizeof(procfd_path),
         "/proc/self/fd/%d", metadata->fd);
path_len = readlink(procfd_path, path,
                   sizeof(path) - 1);
if (path_len == -1) {
    perror("readlink");
    exit(EXIT_FAILURE);
}
path[path_len] = '\0';
printf("File %s\n", path);
/* Close the file descriptor of the event */
close(metadata->fd);
}
/* Advance to next event */
metadata = FAN_EVENT_NEXT(metadata, len);
}
}
int
main(int argc, char *argv[])
{
    char buf;
    int fd, poll_num;
    nfd_t nfds;
    struct pollfd fds[2];
/* Check mount point is supplied */
if (argc != 2) {
    fprintf(stderr, "Usage: %s MOUNT\n", argv[0]);
    exit(EXIT_FAILURE);
}

```

```

}

printf("Press enter key to terminate.\n");

/* Create the file descriptor for accessing the fanotify API */
fd = fanotify_init(FAN_CLOEXEC | FAN_CLASS_CONTENT | FAN_NONBLOCK,
                  O_RDONLY | O_LARGEFILE);

if (fd == -1) {
    perror("fanotify_init");
    exit(EXIT_FAILURE);
}

/* Mark the mount for:
- permission events before opening files
- notification events after closing a write-enabled
file descriptor */
if (fanotify_mark(fd, FAN_MARK_ADD | FAN_MARK_MOUNT,
                  FAN_OPEN_PERM | FAN_CLOSE_WRITE, AT_FDCWD,
                  argv[1]) == -1) {
    perror("fanotify_mark");
    exit(EXIT_FAILURE);
}

/* Prepare for polling */
nfds = 2;

/* Console input */
fds[0].fd = STDIN_FILENO;
fds[0].events = POLLIN;

/* Fanotify input */
fds[1].fd = fd;
fds[1].events = POLLIN;

/* This is the loop to wait for incoming events */
printf("Listening for events.\n");
while (1) {
    poll_num = poll(fds, nfds, -1);

    if (poll_num == -1) {
        if (errno == EINTR) /* Interrupted by a signal */

```

```

        continue;      /* Restart poll() */
perror("poll");      /* Unexpected error */
exit(EXIT_FAILURE);
}
if (poll_num > 0) {
    if (fds[0].revents & POLLIN) {
        /* Console input is available: empty stdin and quit */
        while (read(STDIN_FILENO, &buf, 1) > 0 && buf != '\n')
            continue;
        break;
    }
    if (fds[1].revents & POLLIN) {
        /* Fanotify events are available */
        handle_events(fd);
    }
}
}
printf("Listening for events stopped.\n");
exit(EXIT_SUCCESS);
}

```

Example program: fanotify_fid.c

The second program is an example of fanotify being used with `FAN_REPORT_FID` enabled. The program marks the filesystem object that is passed as a command-line argument and waits until an event of type `FAN_CREATE` has occurred. The event mask indicates which type of filesystem object—either a file or a directory—was created.

Once all events have been read from the buffer and processed accordingly, the program simply terminates.

The following shell sessions show two different invocations of this program, with different actions performed on a watched object.

The first session shows a mark being placed on `/home/user`. This is followed by the creation of a regular file, `/home/user/testfile.txt`. This results in a `FAN_CREATE` event being created and reported against the file's parent watched directory object. Program execution ends once all events captured within the buffer have been

processed. Program execution ends once all events captured within the buffer are processed.

```
# ./fanotify_fid /home/user
```

```
Listening for events.
```

```
FAN_CREATE (file created): Directory /home/user has been modified.
```

```
All events processed successfully. Program exiting.
```

```
$ touch /home/user/testing      # In another terminal
```

The second session shows a mark being placed on /home/user. This is followed by the creation of a directory, /home/user/testdir. This specific action results in the program producing a FAN_CREATE and FAN_ONDIR event.

```
# ./fanotify_fid /home/user
```

```
Listening for events.
```

```
FAN_CREATE | FAN_ONDIR (subdirectory created):
```

```
    Directory /home/user has been modified.
```

```
All events processed successfully. Program exiting.
```

```
$ mkdir -p /home/user/testing    # In another terminal
```

Program source: fanotify_fid.c

```
#define _GNU_SOURCE
```

```
#include <errno.h>
```

```
#include <fcntl.h>
```

```
#include <limits.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <sys/fanotify.h>
```

```
#include <unistd.h>
```

```
#define BUF_SIZE 256
```

```
int
```

```
main(int argc, char **argv)
```

```
{
```

```
    int fd, ret, event_fd;
```

```
    ssize_t len, path_len;
```

```

char path[PATH_MAX];
char procfld_path[PATH_MAX];
char events_buf[BUF_SIZE];
struct file_handle *file_handle;
struct fanotify_event_metadata *metadata;
struct fanotify_event_info_fid *fid;
if (argc != 2) {
    fprintf(stderr, "Invalid number of command line arguments.\n");
    exit(EXIT_FAILURE);
}
/* Create an fanotify file descriptor with FAN_REPORT_FID as a flag
   so that program can receive fid events. */
fd = fanotify_init(FAN_CLASS_NOTIF | FAN_REPORT_FID, 0);
if (fd == -1) {
    perror("fanotify_init");
    exit(EXIT_FAILURE);
}
/* Place a mark on the filesystem object supplied in argv[1]. */
ret = fanotify_mark(fd, FAN_MARK_ADD | FAN_MARK_ONLYDIR,
                    FAN_CREATE | FAN_ONDIR,
                    AT_FDCWD, argv[1]);
if (ret == -1) {
    perror("fanotify_mark");
    exit(EXIT_FAILURE);
}
printf("Listening for events.\n");
/* Read events from the event queue into a buffer */
len = read(fd, (void *) &events_buf, sizeof(events_buf));
if (len == -1 && errno != EAGAIN) {
    perror("read");
    exit(EXIT_FAILURE);
}
/* Process all events within the buffer */

```

```

for (metadata = (struct fanotify_event_metadata *) events_buf;
    FAN_EVENT_OK(metadata, len);
    metadata = FAN_EVENT_NEXT(metadata, len)) {
fid = (struct fanotify_event_info_fid *) (metadata + 1);
file_handle = (struct file_handle *) fid->handle;
/* Ensure that the event info is of the correct type */
if (fid->hdr.info_type != FAN_EVENT_INFO_TYPE_FID) {
    fprintf(stderr, "Received unexpected event info type.\n");
    exit(EXIT_FAILURE);
}
if (metadata->mask == FAN_CREATE)
    printf("FAN_CREATE (file created):");
if (metadata->mask == FAN_CREATE | FAN_ONDIR)
    printf("FAN_CREATE | FAN_ONDIR (subdirectory created):");
/* metadata->fd is set to FAN_NOFD when FAN_REPORT_FID is enabled.
To obtain a file descriptor for the file object corresponding to
an event you can use the struct file_handle that's provided
within the fanotify_event_info_fid in conjunction with the
open_by_handle_at(2) system call. A check for ESTALE is done
to accommodate for the situation where the file handle for the
object was deleted prior to this system call. */
event_fd = open_by_handle_at(AT_FDCWD, file_handle, O_RDONLY);
if (ret == -1) {
    if (errno == ESTALE) {
        printf("File handle is no longer valid. "
            "File has been deleted\n");
        continue;
    } else {
        perror("open_by_handle_at");
        exit(EXIT_FAILURE);
    }
}
}
snprintf(procfd_path, sizeof(procfd_path), "/proc/self/fd/%d",

```

```

        event_fd);

/* Retrieve and print the path of the modified dentry */
path_len = readlink(procfd_path, path, sizeof(path) - 1);
if (path_len == -1) {
    perror("readlink");
    exit(EXIT_FAILURE);
}
path[path_len] = '\0';
printf("\tDirectory '%s' has been modified.\n", path);

/* Close associated file descriptor for this event */
close(event_fd);
}

printf("All events processed successfully. Program exiting.\n");
exit(EXIT_SUCCESS);
}

```

SEE ALSO

fanotify_init(2), fanotify_mark(2), inotify(7)

COLOPHON

This page is part of release 5.05 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/>.