NAME

inotify − monitoring filesystem events

DESCRIPTION

The *inotify* API provides a mechanism for monitoring filesystem events. Inotify can be used to monitor individual files, or to monitor directories. When a directory is monitored, inotify will return events for the directory itself, and for files inside the directory.

The following system calls are used with this API:

* **inotify_init**(2) creates an inotify instance and returns a file descriptor referring to the inotify instance. The more recent **inotify_init1**(2) is like **inotify_init**(2), but has a *flags* argument that provides access to some extra functionality.
* **inotify_add_watch**(2) manipulates the "watch list" associated with an inotify instance. Each item ("watch") in the watch list specifies the pathname of a file or directory, along with some set of events that the kernel should monitor for the file referred to by that pathname. **inotify_add_watch**(2) either creates a new watch item, or modifies an existing watch. Each watch has a unique "watch descriptor", an integer returned by **inotify_add_watch**(2) when the watch is created.
* When events occur for monitored files and directories, those events are made available to the application as structured data that can be read from the inotify file descriptor using **read**(2) (see below).
* **inotify_rm_watch**(2) removes an item from an inotify watch list.
* When all file descriptors referring to an inotify instance have been closed (using **close**(2)), the underlying object and its resources are freed for reuse by the kernel; all associated watches are automatically freed.

With careful programming, an application can use inotify to efficiently monitor and cache the state of a set of filesystem objects. However, robust applications should allow for the fact that bugs in the monitoring logic or races of the kind described below may leave the cache inconsistent with the filesystem state. It is probably wise to do some consistency checking, and rebuild the cache when inconsistencies are detected.

Reading events from an inotify file descriptor

To determine what events have occurred, an application **read**(2)s from the inotify file descriptor. If no events have so far occurred, then, assuming a blocking file descriptor, **read**(2) will block until at least one event occurs (unless interrupted by a signal, in which case the call fails with the error **EINTR**; see **signal**(7)).

Each successful **read**(2) returns a buffer containing one or more of the following structures:

```
struct inotify_event {
    int      wd;       /* Watch descriptor */
    uint32_t mask;     /* Mask describing event */
    uint32_t cookie;   /* Unique cookie associating related
                          events (for rename(2)) */
    uint32_t len;      /* Size of name field */
    char     name[];   /* Optional null-terminated name */
};
```

wd identifies the watch for which this event occurs. It is one of the watch descriptors returned by a previous call to **inotify_add_watch**(2).

mask contains bits that describe the event that occurred (see below).

cookie is a unique integer that connects related events. Currently, this is used only for rename events, and allows the resulting pair of **IN_MOVED_FROM** and **IN_MOVED_TO** events to be connected by the application. For all other event types, *cookie* is set to 0.

The *name* field is present only when an event is returned for a file inside a watched directory; it identifies the filename within to the watched directory. This filename is null-terminated, and may include further null bytes ('\0') to align subsequent reads to a suitable address boundary.

The *len* field counts all of the bytes in *name*, including the null bytes; the length of each *inotify_event* structure is thus *sizeof(struct inotify_event)+len*.

The behavior when the buffer given to **read**(2) is too small to return information about the next event depends on the kernel version: in kernels before 2.6.21, **read**(2) returns 0; since kernel 2.6.21, **read**(2) fails with the error **EINVAL**. Specifying a buffer of size

```
sizeof(struct inotify_event) + NAME_MAX + 1
```

will be sufficient to read at least one event.

inotify events

The **inotify_add_watch**(2) *mask* argument and the *mask* field of the *inotify_event* structure returned when **read**(2)ing an inotify file descriptor are both bit masks identifying inotify events. The following bits can be specified in *mask* when calling **inotify_add_watch**(2) and may be returned in the *mask* field returned by **read**(2):

IN_ACCESS (+)
File was accessed (e.g., **read**(2), **execve**(2)).

IN_ATTRIB (*)
Metadata changed—for example, permissions (e.g., **chmod**(2)), timestamps (e.g., **utimensat**(2)), extended attributes (**setxattr**(2)), link count (since Linux 2.6.25; e.g., for the target of **link**(2) and for **unlink**(2)), and user/group ID (e.g., **chown**(2)).

IN_CLOSE_WRITE (+)
File opened for writing was closed.

IN_CLOSE_NOWRITE (*)
File or directory not opened for writing was closed.

IN_CREATE (+)
File/directory created in watched directory (e.g., **open**(2) **O_CREAT**, **mkdir**(2), **link**(2), **symlink**(2), **bind**(2) on a UNIX domain socket).

IN_DELETE (+)
File/directory deleted from watched directory.

IN_DELETE_SELF
Watched file/directory was itself deleted. (This event also occurs if an object is moved to another filesystem, since **mv**(1) in effect copies the file to the other filesystem and then deletes it from the original filesystem.) In addition, an **IN_IGNORED** event will subsequently be generated for the watch descriptor.

IN_MODIFY (+)
File was modified (e.g., **write**(2), **truncate**(2)).

IN_MOVE_SELF
Watched file/directory was itself moved.

IN_MOVED_FROM (+)
Generated for the directory containing the old filename when a file is renamed.

IN_MOVED_TO (+)
Generated for the directory containing the new filename when a file is renamed.

IN_OPEN (*)
File or directory was opened.

Inotify monitoring is inode-based: when monitoring a file (but not when monitoring the directory containing a file), an event can be generated for activity on any link to the file (in the same or a different directory).

When monitoring a directory:

* the events marked above with an asterisk (*) can occur both for the directory itself and for objects inside the directory; and

* the events marked with a plus sign (+) occur only for objects inside the directory (not for the directory itself).

Note: when monitoring a directory, events are not generated for the files inside the directory when the events are performed via a pathname (i.e., a link) that lies outside the monitored directory.

When events are generated for objects inside a watched directory, the *name* field in the returned *inotify_event* structure identifies the name of the file within the directory.

The **IN_ALL_EVENTS** macro is defined as a bit mask of all of the above events. This macro can be used as the *mask* argument when calling **inotify_add_watch**(2).

Two additional convenience macros are defined:

IN_MOVE
: Equates to **IN_MOVED_FROM | IN_MOVED_TO**.

IN_CLOSE
: Equates to **IN_CLOSE_WRITE | IN_CLOSE_NOWRITE**.

The following further bits can be specified in *mask* when calling **inotify_add_watch**(2):

IN_DONT_FOLLOW (since Linux 2.6.15)
: Don't dereference *pathname* if it is a symbolic link.

IN_EXCL_UNLINK (since Linux 2.6.36)
: By default, when watching events on the children of a directory, events are generated for children even after they have been unlinked from the directory. This can result in large numbers of uninteresting events for some applications (e.g., if watching */tmp*, in which many applications create temporary files whose names are immediately unlinked). Specifying **IN_EXCL_UNLINK** changes the default behavior, so that events are not generated for children after they have been unlinked from the watched directory.

IN_MASK_ADD
: If a watch instance already exists for the filesystem object corresponding to *pathname*, add (OR) the events in *mask* to the watch mask (instead of replacing the mask); the error **EINVAL** results if **IN_MASK_CREATE** is also specified.

IN_ONESHOT
: Monitor the filesystem object corresponding to *pathname* for one event, then remove from watch list.

IN_ONLYDIR (since Linux 2.6.15)
: Watch *pathname* only if it is a directory; the error **ENOTDIR** results if *pathname* is not a directory. Using this flag provides an application with a race-free way of ensuring that the monitored object is a directory.

IN_MASK_CREATE (since Linux 4.18)
: Watch *pathname* only if it does not already have a watch associated with it; the error **EEXIST** results if *pathname* is already being watched.

: Using this flag provides an application with a way of ensuring that new watches do not modify existing ones. This is useful because multiple paths may refer to the same inode, and multiple calls to **inotify_add_watch**(2) without this flag may clobber existing watch masks.

The following bits may be set in the *mask* field returned by **read**(2):

IN_IGNORED
: Watch was removed explicitly (**inotify_rm_watch**(2)) or automatically (file was deleted, or filesystem was unmounted). See also BUGS.

IN_ISDIR
: Subject of this event is a directory.

IN_Q_OVERFLOW
Event queue overflowed (*wd* is −1 for this event).

IN_UNMOUNT
Filesystem containing watched object was unmounted. In addition, an **IN_IGNORED** event will subsequently be generated for the watch descriptor.

Examples

Suppose an application is watching the directory *dir* and the file *dir/myfile* for all events. The examples below show some events that will be generated for these two objects.

fd = open("dir/myfile", O_RDWR);
Generates **IN_OPEN** events for both *dir* and *dir/myfile*.

read(fd, buf, count);
Generates **IN_ACCESS** events for both *dir* and *dir/myfile*.

write(fd, buf, count);
Generates **IN_MODIFY** events for both *dir* and *dir/myfile*.

fchmod(fd, mode);
Generates **IN_ATTRIB** events for both *dir* and *dir/myfile*.

close(fd);
Generates **IN_CLOSE_WRITE** events for both *dir* and *dir/myfile*.

Suppose an application is watching the directories *dir1* and *dir2*, and the file *dir1/myfile*. The following examples show some events that may be generated.

link("dir1/myfile", "dir2/new");
Generates an **IN_ATTRIB** event for *myfile* and an **IN_CREATE** event for *dir2*.

rename("dir1/myfile", "dir2/myfile");
Generates an **IN_MOVED_FROM** event for *dir1*, an **IN_MOVED_TO** event for *dir2*, and an **IN_MOVE_SELF** event for *myfile*. The **IN_MOVED_FROM** and **IN_MOVED_TO** events will have the same *cookie* value.

Suppose that *dir1/xx* and *dir2/yy* are (the only) links to the same file, and an application is watching *dir1*, *dir2*, *dir1/xx*, and *dir2/yy*. Executing the following calls in the order given below will generate the following events:

unlink("dir2/yy");
Generates an **IN_ATTRIB** event for *xx* (because its link count changes) and an **IN_DELETE** event for *dir2*.

unlink("dir1/xx");
Generates **IN_ATTRIB**, **IN_DELETE_SELF**, and **IN_IGNORED** events for *xx*, and an **IN_DELETE** event for *dir1*.

Suppose an application is watching the directory *dir* and (the empty) directory *dir/subdir*. The following examples show some events that may be generated.

mkdir("dir/new", mode);
Generates an **IN_CREATE | IN_ISDIR** event for *dir*.

rmdir("dir/subdir");
Generates **IN_DELETE_SELF** and **IN_IGNORED** events for *subdir*, and an **IN_DELETE | IN_ISDIR** event for *dir*.

/proc interfaces

The following interfaces can be used to limit the amount of kernel memory consumed by inotify:

/proc/sys/fs/inotify/max_queued_events
The value in this file is used when an application calls **inotify_init**(2) to set an upper limit on the number of events that can be queued to the corresponding inotify instance. Events in excess of

this limit are dropped, but an **IN_Q_OVERFLOW** event is always generated.

/proc/sys/fs/inotify/max_user_instances
This specifies an upper limit on the number of inotify instances that can be created per real user ID.

/proc/sys/fs/inotify/max_user_watches
This specifies an upper limit on the number of watches that can be created per real user ID.

VERSIONS

Inotify was merged into the 2.6.13 Linux kernel. The required library interfaces were added to glibc in version 2.4. (**IN_DONT_FOLLOW**, **IN_MASK_ADD**, and **IN_ONLYDIR** were added in glibc version 2.5.)

CONFORMING TO

The inotify API is Linux-specific.

NOTES

Inotify file descriptors can be monitored using **select**(2), **poll**(2), and **epoll**(7). When an event is available, the file descriptor indicates as readable.

Since Linux 2.6.25, signal-driven I/O notification is available for inotify file descriptors; see the discussion of **F_SETFL** (for setting the **O_ASYNC** flag), **F_SETOWN**, and **F_SETSIG** in **fcntl**(2). The *siginfo_t* structure (described in **sigaction**(2)) that is passed to the signal handler has the following fields set: *si_fd* is set to the inotify file descriptor number; *si_signo* is set to the signal number; *si_code* is set to **POLL_IN**; and **POLLIN** is set in *si_band*.

If successive output inotify events produced on the inotify file descriptor are identical (same *wd*, *mask*, *cookie*, and *name*), then they are coalesced into a single event if the older event has not yet been read (but see BUGS). This reduces the amount of kernel memory required for the event queue, but also means that an application can't use inotify to reliably count file events.

The events returned by reading from an inotify file descriptor form an ordered queue. Thus, for example, it is guaranteed that when renaming from one directory to another, events will be produced in the correct order on the inotify file descriptor.

The set of watch descriptors that is being monitored via an inotify file descriptor can be viewed via the entry for the inotify file descriptor in the process's */proc/[pid]/fdinfo* directory. See **proc**(5) for further details. The **FIONREAD ioctl**(2) returns the number of bytes available to read from an inotify file descriptor.

Limitations and caveats

The inotify API provides no information about the user or process that triggered the inotify event. In particular, there is no easy way for a process that is monitoring events via inotify to distinguish events that it triggers itself from those that are triggered by other processes.

Inotify 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. (Applications must fall back to polling the filesystem to catch such events.) Furthermore, various pseudo-filesystems such as */proc*, */sys*, and */dev/pts* are not monitorable with inotify.

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

The inotify API identifies affected files by filename. However, by the time an application processes an inotify event, the filename may already have been deleted or renamed.

The inotify API identifies events via watch descriptors. It is the application's responsibility to cache a mapping (if one is needed) between watch descriptors and pathnames. Be aware that directory renamings may affect multiple cached pathnames.

Inotify monitoring of directories is not recursive: to monitor subdirectories under a directory, additional watches must be created. This can take a significant amount time for large directory trees.

If monitoring an entire directory subtree, and a new subdirectory is created in that tree or an existing directory is renamed into that tree, be aware that by the time you create a watch for the new subdirectory, new files (and subdirectories) may already exist inside the subdirectory. Therefore, you may want to scan the contents of the subdirectory immediately after adding the watch (and, if desired, recursively add watches for any subdirectories that it contains).

Note that the event queue can overflow. In this case, events are lost. Robust applications should handle the possibility of lost events gracefully. For example, it may be necessary to rebuild part or all of the application cache. (One simple, but possibly expensive, approach is to close the inotify file descriptor, empty the cache, create a new inotify file descriptor, and then re-create watches and cache entries for the objects to be monitored.)

If a filesystem is mounted on top of a monitored directory, no event is generated, and no events are generated for objects immediately under the new mount point. If the filesystem is subsequently unmounted, events will subsequently be generated for the directory and the objects it contains.

Dealing with rename() events

As noted above, the **IN_MOVED_FROM** and **IN_MOVED_TO** event pair that is generated by **rename**(2) can be matched up via their shared cookie value. However, the task of matching has some challenges.

These two events are usually consecutive in the event stream available when reading from the inotify file descriptor. However, this is not guaranteed. If multiple processes are triggering events for monitored objects, then (on rare occasions) an arbitrary number of other events may appear between the **IN_MOVED_FROM** and **IN_MOVED_TO** events. Furthermore, it is not guaranteed that the event pair is atomically inserted into the queue: there may be a brief interval where the **IN_MOVED_FROM** has appeared, but the **IN_MOVED_TO** has not.

Matching up the **IN_MOVED_FROM** and **IN_MOVED_TO** event pair generated by **rename**(2) is thus inherently racy. (Don't forget that if an object is renamed outside of a monitored directory, there may not even be an **IN_MOVED_TO** event.) Heuristic approaches (e.g., assume the events are always consecutive) can be used to ensure a match in most cases, but will inevitably miss some cases, causing the application to perceive the **IN_MOVED_FROM** and **IN_MOVED_TO** events as being unrelated. If watch descriptors are destroyed and re-created as a result, then those watch descriptors will be inconsistent with the watch descriptors in any pending events. (Re-creating the inotify file descriptor and rebuilding the cache may be useful to deal with this scenario.)

Applications should also allow for the possibility that the **IN_MOVED_FROM** event was the last event that could fit in the buffer returned by the current call to **read**(2), and the accompanying **IN_MOVED_TO** event might be fetched only on the next **read**(2), which should be done with a (small) timeout to allow for the fact that insertion of the **IN_MOVED_FROM-IN_MOVED_TO** event pair is not atomic, and also the possibility that there may not be any **IN_MOVED_TO** event.

BUGS

Before Linux 3.19, **fallocate**(2) did not create any inotify events. Since Linux 3.19, calls to **fallocate**(2) generate **IN_MODIFY** events.

In kernels before 2.6.16, the **IN_ONESHOT** *mask* flag does not work.

As originally designed and implemented, the **IN_ONESHOT** flag did not cause an **IN_IGNORED** event to be generated when the watch was dropped after one event. However, as an unintended effect of other changes, since Linux 2.6.36, an **IN_IGNORED** event is generated in this case.

Before kernel 2.6.25, the kernel code that was intended to coalesce successive identical events (i.e., the two most recent events could potentially be coalesced if the older had not yet been read) instead checked if the most recent event could be coalesced with the *oldest* unread event.

When a watch descriptor is removed by calling **inotify_rm_watch**(2) (or because a watch file is deleted or the filesystem that contains it is unmounted), any pending unread events for that watch descriptor remain available to read. As watch descriptors are subsequently allocated with **inotify_add_watch**(2), the kernel cycles through the range of possible watch descriptors (0 to **INT_MAX**) incrementally. When allocating a free watch descriptor, no check is made to see whether that watch descriptor number has any pending unread events in the inotify queue. Thus, it can happen that a watch descriptor is reallocated even when pending unread events exist for a previous incarnation of that watch descriptor number, with the result that the application might then read those events and interpret them as belonging to the file associated with the newly recycled watch descriptor. In practice, the likelihood of hitting this bug may be extremely low, since it requires that an application cycle through **INT_MAX** watch descriptors, release a watch descriptor while leaving unread events for that watch descriptor in the queue, and then recycle that watch descriptor. For this reason, and because there have been no reports of the bug occurring in real-world applications, as of Linux 3.15, no kernel changes have yet been made to eliminate this possible bug.

EXAMPLE

The following program demonstrates the usage of the inotify API. It marks the directories passed as a command-line arguments and waits for events of type **IN_OPEN**, **IN_CLOSE_NOWRITE** and **IN_CLOSE_WRITE**.

The following output was recorded while editing the file */home/user/temp/foo* and listing directory */tmp*. Before the file and the directory were opened, **IN_OPEN** events occurred. After the file was closed, an **IN_CLOSE_WRITE** event occurred. After the directory was closed, an **IN_CLOSE_NOWRITE** event occurred. Execution of the program ended when the user pressed the ENTER key.

Example output

```
$ ./a.out /tmp /home/user/temp
Press enter key to terminate.
Listening for events.
IN_OPEN: /home/user/temp/foo [file]
IN_CLOSE_WRITE: /home/user/temp/foo [file]
IN_OPEN: /tmp/ [directory]
IN_CLOSE_NOWRITE: /tmp/ [directory]

Listening for events stopped.
```

Program source

```
#include <errno.h>
#include <poll.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/inotify.h>
#include <unistd.h>

/* Read all available inotify events from the file descriptor 'fd'.
   wd is the table of watch descriptors for the directories in argv.
   argc is the length of wd and argv.
   argv is the list of watched directories.
   Entry 0 of wd and argv is unused. */

static void
handle_events(int fd, int *wd, int argc, char* argv[])
{
    /* Some systems cannot read integer variables if they are not
       properly aligned. On other systems, incorrect alignment may
       decrease performance. Hence, the buffer used for reading from
       the inotify file descriptor should have the same alignment as
       struct inotify_event. */

    char buf[4096]
        __attribute__ ((aligned(__alignof__(struct inotify_event))));
    const struct inotify_event *event;
```

```
int i;
ssize_t len;
char *ptr;

/* Loop while events can be read from inotify file descriptor. */

for (;;) {

    /* Read some events. */

    len = read(fd, buf, sizeof buf);
    if (len == -1 && errno != EAGAIN) {
        perror("read");
        exit(EXIT_FAILURE);
    }

    /* If the nonblocking read() found no events to read, then
       it returns -1 with errno set to EAGAIN. In that case,
       we exit the loop. */

    if (len <= 0)
        break;

    /* Loop over all events in the buffer */

    for (ptr = buf; ptr < buf + len;
            ptr += sizeof(struct inotify_event) + event->len) {

        event = (const struct inotify_event *) ptr;

        /* Print event type */

        if (event->mask & IN_OPEN)
            printf("IN_OPEN: ");
        if (event->mask & IN_CLOSE_NOWRITE)
            printf("IN_CLOSE_NOWRITE: ");
        if (event->mask & IN_CLOSE_WRITE)
            printf("IN_CLOSE_WRITE: ");

        /* Print the name of the watched directory */

        for (i = 1; i < argc; ++i) {
            if (wd[i] == event->wd) {
                printf("%s/", argv[i]);
                break;
            }
        }

        /* Print the name of the file */

        if (event->len)
            printf("%s", event->name);

        /* Print type of filesystem object */
```

```
                if (event->mask & IN_ISDIR)
                    printf(" [directory]\n");
                else
                    printf(" [file]\n");
            }
        }
    }

    int
    main(int argc, char* argv[])
    {
        char buf;
        int fd, i, poll_num;
        int *wd;
        nfds_t nfds;
        struct pollfd fds[2];

        if (argc < 2) {
            printf("Usage: %s PATH [PATH ...]\n", argv[0]);
            exit(EXIT_FAILURE);
        }

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

        /* Create the file descriptor for accessing the inotify API */

        fd = inotify_init1(IN_NONBLOCK);
        if (fd == -1) {
            perror("inotify_init1");
            exit(EXIT_FAILURE);
        }

        /* Allocate memory for watch descriptors */

        wd = calloc(argc, sizeof(int));
        if (wd == NULL) {
            perror("calloc");
            exit(EXIT_FAILURE);
        }

        /* Mark directories for events
           - file was opened
           - file was closed */

        for (i = 1; i < argc; i++) {
            wd[i] = inotify_add_watch(fd, argv[i],
                                      IN_OPEN | IN_CLOSE);
            if (wd[i] == -1) {
                fprintf(stderr, "Cannot watch '%s'\n", argv[i]);
                perror("inotify_add_watch");
                exit(EXIT_FAILURE);
            }
        }
        /* Prepare for polling */

        nfds = 2;

        /* Console input */

        fds[0].fd = STDIN_FILENO;
        fds[0].events = POLLIN;

        /* Inotify input */

        fds[1].fd = fd;
        fds[1].events = POLLIN;

        /* Wait for events and/or terminal input */

        printf("Listening for events.\n");
        while (1) {
            poll_num = poll(fds, nfds, −1);
            if (poll_num == −1) {
                if (errno == EINTR)
                    continue;
                perror("poll");
                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) {

                    /* Inotify events are available */

                    handle_events(fd, wd, argc, argv);
                }
            }
        }

        printf("Listening for events stopped.\n");

        /* Close inotify file descriptor */

        close(fd);

        free(wd);
        exit(EXIT_SUCCESS);
```

```
}
```

SEE ALSO

inotifywait(1), **inotifywatch**(1), **inotify_add_watch**(2), **inotify_init**(2), **inotify_init1**(2), **ino-tify_rm_watch**(2), **read**(2), **stat**(2), **fanotify**(7)

Documentation/filesystems/inotify.txt in the Linux kernel source tree

COLOPHON

This page is part of release 5.02 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/.