19

MONITORING FILE EVENTS

Some applications need to be able to monitor files or directories in order to deter-
mine whether events have occurred for the monitored objects. For example, a
graphical file manager needs to be able to determine when files are added or
removed from the directory that is currently being displayed, or a daemon may
want to monitor its configuration file in order to know if the file has been changed.

Starting with kernel 2.6.13, Linux provides the *inotify* mechanism, which allows
an application to monitor file events. This chapter describes the use of *inotify*.

The *inotify* mechanism replaces an older mechanism, *dnotify*, which provided a
subset of the functionality of *inotify*. We describe *dnotify* briefly at the end of this
chapter, focusing on why *inotify* is better.

The *inotify* and *dnotify* mechanisms are Linux-specific. (A few other systems
provide similar mechanisms. For example, the BSDs provide the *kqueue* API.)

> A few libraries provide an API that is more abstract and portable than *inotify*
> and *dnotify*. The use of these libraries may be preferable for some applications.
> Some of these libraries employ *inotify* or *dnotify*, on systems where they are
> available. Two such libraries are FAM (File Alteration Monitor, *http://
> oss.sgi.com/projects/fam/*) and Gamin (*http://www.gnome.org/~veillard/gamin/*).

19.1 Overview

The key steps in the use of the *inotify* API are as follows:

1. The application uses *inotify_init()* to create an *inotify instance*. This system call returns a file descriptor that is used to refer to the *inotify* instance in later operations.

2. The application informs the kernel about which files are of interest by using *inotify_add_watch()* to add items to the watch list of the *inotify* instance created in the previous step. Each watch item consists of a pathname and an associated bit mask. The bit mask specifies the set of events to be monitored for the pathname. As its function result, *inotify_add_watch()* returns a *watch descriptor*, which is used to refer to the watch in later operations. (The *inotify_rm_watch()* system call performs the converse task, removing a watch that was previously added to an *inotify* instance.)

3. In order to obtain event notifications, the application performs *read()* operations on the *inotify* file descriptor. Each successful *read()* returns one or more *inotify_event* structures, each containing information about an event that occurred on one of the pathnames being watched via this *inotify* instance.

4. When the application has finished monitoring, it closes the *inotify* file descriptor. This automatically removes all watch items associated with the *inotify* instance.

The *inotify* mechanism can be used to monitor files or directories. When monitoring a directory, the application will be informed about events for the directory itself and for files inside the directory.

The *inotify* monitoring mechanism is not recursive. If an application wants to monitor events within an entire directory subtree, it must issue *inotify_add_watch()* calls for each directory in the tree.

An *inotify* file descriptor can be monitored using *select()*, *poll()*, *epoll*, and, since Linux 2.6.25, signal-driven I/O. If events are available to be read, then these interfaces indicate the *inotify* file descriptor as being readable. See Chapter 63 for further details of these interfaces.

> The *inotify* mechanism is an optional Linux kernel component that is configured via the options CONFIG_INOTIFY and CONFIG_INOTIFY_USER.

19.2 The *inotify* API

The *inotify_init()* system call creates a new *inotify* instance.

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

int inotify_init(void);
```
                              Returns file descriptor on success, or –1 on error

As its function result, *inotify_init()* returns a file descriptor. This file descriptor is the handle that is used to refer to the *inotify* instance in subsequent operations.

Starting with kernel 2.6.27, Linux supports a new, nonstandard system call, *inotify_init1()*. This system call performs the same task as *inotify_init()*, but provides an additional argument, *flags*, that can be used to modify the behavior of the system call. Two flags are supported. The IN_CLOEXEC flag causes the kernel to enable the close-on-exec flag (FD_CLOEXEC) for the new file descriptor. This flag is useful for the same reasons as the *open()* O_CLOEXEC flag described in Section 4.3.1. The IN_NONBLOCK flag causes the kernel to enable the O_NONBLOCK flag on the underlying open file description, so that future reads will be nonblocking. This saves additional calls to *fcntl()* to achieve the same result.

The *inotify_add_watch()* system call either adds a new watch item to or modifies an existing watch item in the watch list for the *inotify* instance referred to by the file descriptor *fd*. (Refer to Figure 19-1.)

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

int inotify_add_watch(int fd, const char *pathname, uint32_t mask);
```
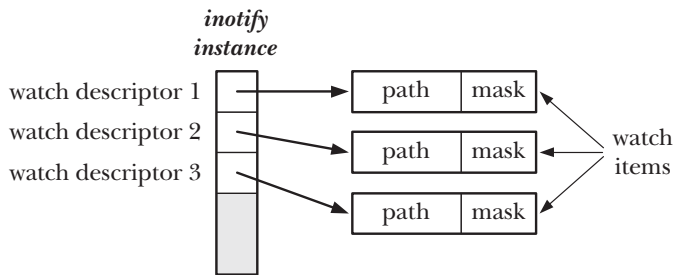   Returns watch descriptor on success, or –1 on error

Figure 19-1: An *inotify* instance and associated kernel data structures

The *pathname* argument identifies the file for which a watch item is to be created or modified. The caller must have read permission for this file. (The file permission check is performed once, at the time of the *inotify_add_watch()* call. As long as the watch item continues to exist, the caller will continue to receive file notifications even if the file permissions are later changed so that the caller no longer has read permission on the file.)

The *mask* argument is a bit mask that specifies the events to be monitored for *pathname*. We say more about the bit values that can be specified in *mask* shortly.

If *pathname* has not previously been added to the watch list for *fd*, then *inotify_add_watch()* creates a new watch item in the list and returns a new, nonnegative watch descriptor, which is used to refer to the watch item in later operations. This watch descriptor is unique for this *inotify* instance.

If *pathname* has previously been added to the watch list for *fd*, then *inotify_add_watch()* modifies the mask of the existing watch item for *pathname* and returns the watch descriptor for that item. (This watch descriptor will be the same as that returned by the *inotify_add_watch()* call that initially added *pathname* to this watch list.) We say more about how the mask may be modified when we describe the IN_MASK_ADD flag in the next section.

The *inotify_rm_watch()* system call removes the watch item specified by *wd* from the *inotify* instance referred to by the file descriptor *fd*.

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

int inotify_rm_watch(int fd, uint32_t wd);
```
                                          Returns 0 on success, or –1 on error

The *wd* argument is a watch descriptor returned by a previous call to *inotify_add_watch()*. (The *uint32_t* data type is an unsigned 32-bit integer.)

Removing a watch causes an IN_IGNORED event to be generated for this watch descriptor. We say more about this event shortly.

19.3 *inotify* **Events**

When we create or modify a watch using *inotify_add_watch()*, the *mask* bit-mask argument identifies the events to be monitored for the given *pathname*. The event bits that may be specified in *mask* are indicated by the *In* column of Table 19-1.

Table 19-1: *inotify* events

Bit value	In	Out	Description
IN_ACCESS	•	•	File was accessed (*read()*)
IN_ATTRIB	•	•	File metadata changed
IN_CLOSE_WRITE	•	•	File opened for writing was closed
IN_CLOSE_NOWRITE	•	•	File opened read-only was closed
IN_CREATE	•	•	File/directory created inside watched directory
IN_DELETE	•	•	File/directory deleted from within watched directory
IN_DELETE_SELF	•	•	Watched file/directory was itself deleted
IN_MODIFY	•	•	File was modified
IN_MOVE_SELF	•	•	Watched file/directory was itself moved
IN_MOVED_FROM	•	•	File moved out of watched directory
IN_MOVED_TO	•	•	File moved into watched directory
IN_OPEN	•	•	File was opened
IN_ALL_EVENTS	•		Shorthand for all of the above input events
IN_MOVE	•		Shorthand for IN_MOVED_FROM | IN_MOVED_TO
IN_CLOSE	•		Shorthand for IN_CLOSE_WRITE | IN_CLOSE_NOWRITE
IN_DONT_FOLLOW	•		Don't dereference symbolic link (since Linux 2.6.15)
IN_MASK_ADD	•		Add events to current watch mask for *pathname*
IN_ONESHOT	•		Monitor *pathname* for just one event
IN_ONLYDIR	•		Fail if *pathname* is not a directory (since Linux 2.6.15)
IN_IGNORED		•	Watch was removed by application or by kernel
IN_ISDIR		•	Filename returned in *name* is a directory
IN_Q_OVERFLOW		•	Overflow on event queue
IN_UNMOUNT		•	File system containing object was unmounted

The meanings of most of the bits in Table 19-1 are evident from their names. The following list clarifies a few details:

- The IN_ATTRIB event occurs when file metadata such as permissions, ownership, link count, extended attributes, user ID, or group ID, is changed.

- The IN_DELETE_SELF event occurs when an object (i.e., a file or a directory) that is being monitored is deleted. The IN_DELETE event occurs when the monitored object is a directory and one of the files that it contains is deleted.

- The IN_MOVE_SELF event occurs when an object that is being monitored is renamed. The IN_MOVED_FROM and IN_MOVED_TO events occur when an object is renamed within monitored directories. The former event occurs for the directory containing the old name, and the latter event occurs for the directory containing the new name.

- The IN_DONT_FOLLOW, IN_MASK_ADD, IN_ONESHOT, and IN_ONLYDIR bits don't specify events to be monitored. Instead, they control the operation of the *inotify_add_watch()* call.

- IN_DONT_FOLLOW specifies that *pathname* should not be dereferenced if it is a symbolic link. This permits an application to monitor a symbolic link, rather than the file to which it refers.

- If we perform an *inotify_add_watch()* call that specifies a pathname that is already being watched via this *inotify* file descriptor, then, by default, the given *mask* is used to replace the current mask for this watch item. If IN_MASK_ADD is specified, then the current mask is instead modified by ORing it with the value given in *mask*.

- IN_ONESHOT permits an application to monitor *pathname* for a single event. After that event, the watch item is automatically removed from the watch list.

- IN_ONLYDIR permits an application to monitor a pathname only if it is a directory. If *pathname* is not a directory, then *inotify_add_watch()* fails with the error ENOTDIR. Using this flag prevents race conditions that could otherwise occur if we wanted to ensure that we are monitoring a directory.

19.4 Reading *inotify* Events

Having registered items in the watch list, an application can determine which events have occurred by using *read()* to read events from the *inotify* file descriptor. If no events have occurred so far, then *read()* blocks until an event occurs (unless the O_NONBLOCK status flag has been set for the file descriptor, in which case the *read()* fails immediately with the error EAGAIN if no events are available).

After events have occurred, each *read()* returns a buffer (see Figure 19-2) containing one or more structures of the following type:

```
struct inotify_event {
    int      wd;        /* Watch descriptor on which event occurred */
    uint32_t mask;      /* Bits describing event that occurred */
    uint32_t cookie;    /* Cookie for related events (for rename()) */
    uint32_t len;       /* Size of 'name' field */
    char     name[];    /* Optional null-terminated filename */
};
```
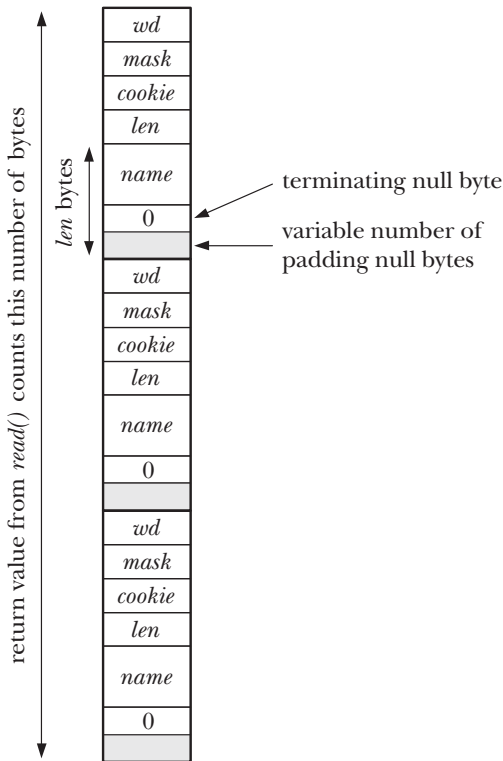
Figure 19-2: An input buffer containing three *inotify_event* structures

The *wd* field tells us the watch descriptor for which this event occurred. This field contains one of the values returned by a previous call to *inotify_add_watch()*. The *wd* field is useful when an application is monitoring multiple files or directories via the same *inotify* file descriptor. It provides the link that allows the application to determine the particular file or directory for which the event occurred. (To do this, the application must maintain a bookkeeping data structure that relates watch descriptors to pathnames.)

The *mask* field returns a bit mask that describes the event. The range of bits that can appear in *mask* is indicated via the *Out* column of Table 19-1. Note the following additional details about specific bits:

- An IN_IGNORED event is generated when a watch is removed. This can occur for two reasons: the application used an *inotify_rm_watch()* call to explicitly remove the watch, or the watch was implicitly removed by the kernel because the monitored object was deleted or the file system where it resides was unmounted. An IN_IGNORED event is not generated when a watch that was established with IN_ONESHOT is automatically removed because an event was triggered.

- If the subject of the event is a directory, then, in addition to some other bit, the IN_ISDIR bit will be set in *mask*.

- The `IN_UNMOUNT` event informs the application that the file system containing the monitored object has been unmounted. After this event, a further event containing the `IN_IGNORED` bit will be delivered.

- We describe the `IN_Q_OVERFLOW` in Section 19.5, which discusses limits on queued *inotify* events.

The *cookie* field is used to tie related events together. Currently, this field is used only when a file is renamed. When this happens, an `IN_MOVED_FROM` event is generated for the directory from which the file is renamed, and then an `IN_MOVED_TO` is generated for the directory to which the file is renamed. (If a file is given a new name within the same directory, then both events occur for the same directory.) These two events will have the same unique value in their *cookie* field, thus allowing the application to associate them.

When an event occurs for a file within a monitored directory, the *name* field is used to return a null-terminated string that identifies the file. If the event occurs for the monitored object itself, then the *name* field is unused, and the *len* field will contain 0.

The *len* field indicates how many bytes are actually allocated for the *name* field. This field is necessary because there may be additional padding bytes between the end of the string stored in *name* and the start of the next *inotify_event* structure contained in the buffer returned by *read()* (see Figure 19-2). The length of an individual *inotify* event is thus *sizeof(struct inotify_event) + len*.

If the buffer passed to *read()* is too small to hold the next *inotify_event* structure, then *read()* fails with the error `EINVAL` to warn the application of this fact. (In kernels before 2.6.21, *read()* returned 0 for this case. The change to the use of an `EINVAL` error provides a clearer indication that a programming error has been made.) The application could respond by performing another *read()* with a larger buffer. However, the problem can be avoided altogether by ensuring that the buffer is always large enough to hold at least one event: the buffer given to *read()* should be at least *(sizeof(struct inotify_event) + NAME_MAX + 1)* bytes, where `NAME_MAX` is the maximum length of a filename, plus one for the terminating null byte.

Using a larger buffer size than the minimum allows an application to efficiently retrieve multiple events with a single *read()*. A *read()* from an *inotify* file descriptor returns the minimum of the number of events that are available and the number of events that will fit in the supplied buffer.

> The call *ioctl(fd, FIONREAD, &numbytes)* returns the number of bytes that are currently available to read from the *inotify* instance referred to by the file descriptor *fd*.

The events read from an *inotify* file descriptor form an ordered queue. Thus, for example, it is guaranteed that when a file is renamed, the `IN_MOVED_FROM` event will be read before the `IN_MOVED_TO` event.

When appending a new event to the end of the event queue, the kernel will coalesce that event with the event at the tail of the queue (so that the new event is not in fact queued), if the two events have the same values for *wd*, *mask*, *cookie*, and *name*. This is done because many applications don't need to know about repeated instances of the same event, and dropping the excess events reduces the amount of (kernel) memory required for the event queue. However, this means we can't use *inotify* to reliably determine how many times or how often a recurrent event occurs.

Example program

Although there is a lot of detail in the preceding description, the *inotify* API is actually quite simple to use. Listing 19-1 demonstrates the use of *inotify*.

Listing 19-1: Using the *inotify* API

—————————————————————————— inotify/demo_inotify.c

```
#include <sys/inotify.h>
#include <limits.h>
#include "tlpi_hdr.h"

static void                 /* Display information from inotify_event structure */
displayInotifyEvent(struct inotify_event *i)
{
    printf("    wd =%2d; ", i->wd);
    if (i->cookie > 0)
        printf("cookie =%4d; ", i->cookie);

    printf("mask = ");
    if (i->mask & IN_ACCESS)        printf("IN_ACCESS ");
    if (i->mask & IN_ATTRIB)        printf("IN_ATTRIB ");
    if (i->mask & IN_CLOSE_NOWRITE) printf("IN_CLOSE_NOWRITE ");
    if (i->mask & IN_CLOSE_WRITE)   printf("IN_CLOSE_WRITE ");
    if (i->mask & IN_CREATE)        printf("IN_CREATE ");
    if (i->mask & IN_DELETE)        printf("IN_DELETE ");
    if (i->mask & IN_DELETE_SELF)   printf("IN_DELETE_SELF ");
    if (i->mask & IN_IGNORED)       printf("IN_IGNORED ");
    if (i->mask & IN_ISDIR)         printf("IN_ISDIR ");
    if (i->mask & IN_MODIFY)        printf("IN_MODIFY ");
    if (i->mask & IN_MOVE_SELF)     printf("IN_MOVE_SELF ");
    if (i->mask & IN_MOVED_FROM)    printf("IN_MOVED_FROM ");
    if (i->mask & IN_MOVED_TO)      printf("IN_MOVED_TO ");
    if (i->mask & IN_OPEN)          printf("IN_OPEN ");
    if (i->mask & IN_Q_OVERFLOW)    printf("IN_Q_OVERFLOW ");
    if (i->mask & IN_UNMOUNT)       printf("IN_UNMOUNT ");
    printf("\n");

    if (i->len > 0)
        printf("        name = %s\n", i->name);
}
```

```c
#define BUF_LEN (10 * (sizeof(struct inotify_event) + NAME_MAX + 1))

int
main(int argc, char *argv[])
{
    int inotifyFd, wd, j;
    char buf[BUF_LEN];
    ssize_t numRead;
    char *p;
    struct inotify_event *event;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname... \n", argv[0]);

①  inotifyFd = inotify_init();                  /* Create inotify instance */
    if (inotifyFd == -1)
        errExit("inotify_init");

    for (j = 1; j < argc; j++) {
②      wd = inotify_add_watch(inotifyFd, argv[j], IN_ALL_EVENTS);
        if (wd == -1)
            errExit("inotify_add_watch");

        printf("Watching %s using wd %d\n", argv[j], wd);
    }

    for (;;) {                                   /* Read events forever */
③      numRead = read(inotifyFd, buf, BUF_LEN);
        if (numRead == 0)
            fatal("read() from inotify fd returned 0!");

        if (numRead == -1)
            errExit("read");

        printf("Read %ld bytes from inotify fd\n", (long) numRead);

        /* Process all of the events in buffer returned by read() */

        for (p = buf; p < buf + numRead; ) {
            event = (struct inotify_event *) p;
④          displayInotifyEvent(event);

            p += sizeof(struct inotify_event) + event->len;
        }
    }

    exit(EXIT_SUCCESS);
}
```
———————————————————————————————— **inotify/demo_inotify.c**

The program in Listing 19-1 performs the following steps:

- Use *inotify_init()* to create an *inotify* file descriptor ①.
- Use *inotify_add_watch()* to add a watch item for each of the files named in the command-line argument of the program ②. Each watch item watches for all possible events.
- Execute an infinite loop that:
  - Reads a buffer of events from the *inotify* file descriptor ③.
  - Calls the *displayInotifyEvent()* function to display the contents of each of the *inotify_event* structures within that buffer ④.

The following shell session demonstrates the use of the program in Listing 19-1. We start an instance of the program that runs in the background monitoring two directories:

```
$ ./demo_inotify dir1 dir2 &
[1] 5386
Watching dir1 using wd 1
Watching dir2 using wd 2
```

Then we execute commands that generate events in the two directories. We begin by creating a file using *cat(1)*:

```
$ cat > dir1/aaa
Read 64 bytes from inotify fd
    wd = 1; mask = IN_CREATE
        name = aaa
    wd = 1; mask = IN_OPEN
        name = aaa
```

The above output produced by the background program shows that *read()* fetched a buffer containing two events. We continue by typing some input for the file and then the terminal *end-of-file* character:

```
Hello world
Read 32 bytes from inotify fd
    wd = 1; mask = IN_MODIFY
        name = aaa
Type Control-D
Read 32 bytes from inotify fd
    wd = 1; mask = IN_CLOSE_WRITE
        name = aaa
```

We then rename the file into the other monitored directory. This results in two events, one for the directory from which the file moves (watch descriptor 1), and the other for the destination directory (watch descriptor 2):

```
$ mv dir1/aaa dir2/bbb
Read 64 bytes from inotify fd
    wd = 1; cookie = 548; mask = IN_MOVED_FROM
        name = aaa
    wd = 2; cookie = 548; mask = IN_MOVED_TO
        name = bbb
```

These two events share the same *cookie* value, allowing the application to link them.

When we create a subdirectory under one of the monitored directories, the mask in the resulting event includes the IN_ISDIR bit, indicating that the subject of the event is a directory:

```
$ mkdir dir2/ddd
Read 32 bytes from inotify fd
    wd = 1; mask = IN_CREATE IN_ISDIR
        name = ddd
```

At this point, it is worth repeating that *inotify* monitoring is not recursive. If the application wanted to monitor events in the newly created subdirectory, then it would need to issue a further *inotify_add_watch()* call specifying the pathname of the subdirectory.

Finally, we remove one of the monitored directories:

```
$ rmdir dir1
Read 32 bytes from inotify fd
    wd = 1; mask = IN_DELETE_SELF
    wd = 1; mask = IN_IGNORED
```

The last event, IN_IGNORED, was generated to inform the application that the kernel has removed this watch item from the watch list.

19.5 Queue Limits and /proc Files

Queuing *inotify* events requires kernel memory. For this reason, the kernel places various limits on the operation of the *inotify* mechanism. The superuser can configure these limits via three files in the directory /proc/sys/fs/inotify:

max_queued_events
: When *inotify_init()* is called, this value is used to set an upper limit on the number of events that can be queued on the new *inotify* instance. If this limit is reached, then an IN_Q_OVERFLOW event is generated and excess events are discarded. The *wd* field for the overflow event will have the value −1.

max_user_instances
: This is a limit on the number of *inotify* instances that can be created per real user ID.

max_user_watches
: This is a limit on the number of watch items that can be created per real user ID.

Typical default values for these three files are 16,384, 128, and 8192, respectively.

19.6 An Older System for Monitoring File Events: *dnotify*

Linux provides another mechanism for monitoring file events. This mechanism, known as *dnotify*, has been available since kernel 2.4, but has been made obsolete by *inotify*. The *dnotify* mechanism suffers a number of limitations compared with *inotify*:

- The *dnotify* mechanism provides notification of events by sending signals to the application. Using signals as a notification mechanism complicates application design (Section 22.12). It also makes the use of *dnotify* within a library difficult, since the calling program might change the disposition of the notification signal(s). The *inotify* mechanism doesn't use signals.

- The monitoring unit of *dnotify* is a directory. The application is informed when an operation is performed on any file in that directory. By contrast, *inotify* can be used to monitor directories or individual files.

- In order to monitor a directory, *dnotify* requires the application to open a file descriptor for that directory. The use of file descriptors causes two problems. First, because it is busy, the file system containing the directory can't be unmounted. Second, because one file descriptor is required for each directory, an application can end up consuming a large number of file descriptors. Because *inotify* doesn't use file descriptors, it avoids these problems.

- The information provided by *dnotify* about file events is less precise than that provided by *inotify*. When a file is changed inside a monitored directory, *dnotify* tells us that an event has occurred, but doesn't tell us which file was involved in the event. The application must determine this by caching information about the directory contents. Furthermore, *inotify* provides more detailed information than *dnotify* about the type of event that has occurred.

- In some circumstances, *dnotify* doesn't provide reliable notification of file events.

Further information about *dnotify* can be found under the description of the F_NOTIFY operation in the *fcntl(2)* manual page, and in the kernel source file Documentation/dnotify.txt.

19.7 Summary

The Linux-specific *inotify* mechanism allows an application to obtain notifications when events (files are opened, closed, created, deleted, modified, renamed, and so on) occur for a set of monitored files and directories. The *inotify* mechanism supersedes the older *dnotify* mechanism.

19.8 Exercise

19-1. Write a program that logs all file creations, deletions, and renames under the directory named in its command-line argument. The program should monitor events in all of the subdirectories under the specified directory. To obtain a list of all of these subdirectories, you will need to make use of *nftw()* (Section 18.9). When a new subdirectory is added under the tree or a directory is deleted, the set of monitored subdirectories should be updated accordingly.