18.7 Removing a File or Directory: *remove()*

The *remove()* library function removes a file or an empty directory.

```
#include <stdio.h>

int remove(const char *pathname);
```

Returns 0 on success, or −1 on error

If *pathname* is a file, *remove()* calls *unlink()*; if *pathname* is a directory, *remove()* calls *rmdir()*.

Like *unlink()* and *rmdir()*, *remove()* doesn't dereference symbolic links. If *pathname* is a symbolic link, *remove()* removes the link itself, rather than the file to which it refers.

If we want to remove a file in preparation for creating a new file with the same name, then using *remove()* is simpler than code that checks whether a pathname refers to a file or directory and calls *unlink()* or *rmdir()*.

> The *remove()* function was invented for the standard C library, which is implemented on both UNIX and non-UNIX systems. Most non-UNIX systems don't support hard links, so removing files with a function named *unlink()* would not make sense.

18.8 Reading Directories: *opendir()* and *readdir()*

The library functions described in this section can be used to open a directory and retrieve the names of the files it contains one by one.

> The library functions for reading directories are layered on top of the *getdents()* system call (which is not part of SUSv3), but provide an interface that is easier to use. Linux also provides a *readdir(2)* system call (as opposed to the *readdir(3)* library function described here), which performs a similar task to, but is made obsolete by, *getdents()*.

The *opendir()* function opens a directory and returns a handle that can be used to refer to the directory in later calls.

```
#include <dirent.h>

DIR *opendir(const char *dirpath);
```

Returns directory stream handle, or NULL on error

The *opendir()* function opens the directory specified by *dirpath* and returns a pointer to a structure of type *DIR*. This structure is a so-called *directory stream*, which is a handle that the caller passes to the other functions described below. Upon return from *opendir()*, the directory stream is positioned at the first entry in the directory list.

The *fdopendir()* function is like *opendir()*, except that the directory for which a stream is to be created is specified via the open file descriptor *fd*.

```
#include <dirent.h>

DIR *fdopendir(int fd);
                                    Returns directory stream handle, or NULL on error
```

The *fdopendir()* function is provided so that applications can avoid the kinds of race conditions described in Section 18.11.

After a successful call to *fdopendir()*, this file descriptor is under the control of the system, and the program should not access it in any way other than by using the functions described in the remainder of this section.

The *fdopendir()* function is specified in SUSv4 (but not in SUSv3).

The *readdir()* function reads successive entries from a directory stream.

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
                        Returns pointer to a statically allocated structure describing
                                next directory entry, or NULL on end-of-directory or error
```

Each call to *readdir()* reads the next directory from the directory stream referred to by *dirp* and returns a pointer to a statically allocated structure of type *dirent*, containing the following information about the entry:

```
struct dirent {
    ino_t d_ino;          /* File i-node number */
    char  d_name[];       /* Null-terminated name of file */
};
```

This structure is overwritten on each call to *readdir()*.

> We have omitted various nonstandard fields in the Linux *dirent* structure from the above definition, since their use renders an application nonportable. The most interesting of these nonstandard fields is *d_type*, which is also present on BSD derivatives, but not on other UNIX implementations. This field holds a value indicating the type of the file named in *d_name*, such as DT_REG (regular file), DT_DIR (directory), DT_LNK (symbolic link), or DT_FIFO (FIFO). (These names are analogous to the macros in Table 15-1, on page 282.) Using the information in this field saves the cost of calling *lstat()* in order to discover the file type. Note, however, that, at the time of writing, this field is fully supported only on *Btrfs*, *ext2*, *ext3*, and *ext4*.

Further information about the file referred to by *d_name* can be obtained by calling *stat()* on the pathname constructed using the *dirpath* argument that was specified to *opendir()* concatenated with (a slash and) the value returned in the *d_name* field.

The filenames returned by *readdir()* are not in sorted order, but rather in the order in which they happen to occur in the directory (this depends on the order in which the file system adds files to the directory and how it fills gaps in the directory list after files are removed). (The command *ls −f* lists files in the same unsorted order that they would be retrieved by *readdir()*.)

> We can use the function *scandir(3)* to retrieve a sorted list of files matching programmer-defined criteria; see the manual page for details. Although not specified in SUSv3, *scandir()* is provided on most UNIX implementations.

On end-of-directory or error, *readdir()* returns `NULL`, in the latter case setting *errno* to indicate the error. To distinguish these two cases, we can write the following:

```
errno = 0;
direntp = readdir(dirp);
if (direntp == NULL) {
    if (errno != 0) {
        /* Handle error */
    } else {
        /* We reached end-of-directory */
    }
}
```

If the contents of a directory change while a program is scanning it with *readdir()*, the program might not see the changes. SUSv3 explicitly notes that it is unspecified whether *readdir()* will return a filename that has been added to or removed from the directory since the last call to *opendir()* or *rewinddir()*. All filenames that have been neither added nor removed since the last such call are guaranteed to be returned.

The *rewinddir()* function moves the directory stream back to the beginning so that the next call to *readdir()* will begin again with the first file in the directory.

```
#include <dirent.h>

void rewinddir(DIR *dirp);
```

The *closedir()* function closes the open directory stream referred to by *dirp*, freeing the resources used by the stream.

```
#include <dirent.h>

int closedir(DIR *dirp);
```
Returns 0 on success, or −1 on error

Two further functions, *telldir()* and *seekdir()*, which are also specified in SUSv3, allow random access within a directory stream. Refer to the manual pages for further information about these functions.

Directory streams and file descriptors

A directory stream has an associated file descriptor. The *dirfd()* function returns the file descriptor associated with the directory stream referred to by *dirp*.

```
#include <dirent.h>

int dirfd(DIR *dirp);
```
Returns file descriptor on success, or −1 on error

We might, for example, pass the file descriptor returned by *dirfd()* to *fchdir()* (Section 18.10) in order to change the current working directory of the process to the corresponding directory. Alternatively, we might pass the file descriptor as the *dirfd* argument of one of the functions described in Section 18.11.

The *dirfd()* function also appears on the BSDs, but is present on few other implementations. It is not specified in SUSv3, but is specified in SUSv4.

At this point, it is worth mentioning that *opendir()* automatically sets the close-on-exec flag (`FD_CLOEXEC`) for the file descriptor associated with the directory stream. This ensures that the file descriptor is automatically closed when an *exec()* is performed. (SUSv3 requires this behavior.) We describe the close-on-exec flag in Section 27.4.

Example program

Listing 18-2 uses *opendir()*, *readdir()*, and *closedir()* to list the contents of each of the directories specified in its command line (or in the current working directory if no arguments are supplied). Here is an example of the use of this program:

```
$ mkdir sub                    Create a test directory
$ touch sub/a sub/b            Make some files in the test directory
$ ./list_files sub             List contents of directory
sub/a
sub/b
```

Listing 18-2: Scanning a directory

dirs_links/list_files.c

```
#include <dirent.h>
#include "tlpi_hdr.h"

static void             /* List all files in directory 'dirPath' */
listFiles(const char *dirpath)
{
    DIR *dirp;
    struct dirent *dp;
    Boolean isCurrent;          /* True if 'dirpath' is "." */

    isCurrent = strcmp(dirpath, ".") == 0;

    dirp = opendir(dirpath);
    if (dirp  == NULL) {
        errMsg("opendir failed on '%s'", dirpath);
        return;
    }

    /* For each entry in this directory, print directory + filename */

    for (;;) {
        errno = 0;              /* To distinguish error from end-of-directory */
        dp = readdir(dirp);
        if (dp == NULL)
            break;

        if (strcmp(dp->d_name, ".") == 0 || strcmp(dp->d_name, "..") == 0)
            continue;           /* Skip . and .. */

        if (!isCurrent)
            printf("%s/", dirpath);
        printf("%s\n", dp->d_name);
    }

    if (errno != 0)
        errExit("readdir");

    if (closedir(dirp) == -1)
        errMsg("closedir");
}

int
main(int argc, char *argv[])
{
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [dir...]\n", argv[0]);

    if (argc == 1)              /* No arguments - use current directory */
        listFiles(".");
```

```
    else
        for (argv++; *argv; argv++)
            listFiles(*argv);

    exit(EXIT_SUCCESS);
}
```

dirs_links/list_files.c

The *readdir_r()* function

The *readdir_r()* function is a variation on *readdir()*. The key semantic difference between *readdir_r()* and *readdir()* is that the former is reentrant, while the latter is not. This is because *readdir_r()* returns the file entry via the caller-allocated *entry* argument, while *readdir()* returns information via a pointer to a statically allocated structure. We discuss reentrancy in Sections 21.1.2 and 31.1.

```
#include <dirent.h>

int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
```

Returns 0 on success, or a positive error number on error

Given *dirp*, which is a directory stream previously opened via *opendir()*, *readdir_r()* places information about the next directory entry into the *dirent* structure referred to by *entry*. In addition, a pointer to this structure is placed in *result*. If the end of the directory stream is reached, then NULL is placed in *result* instead (and *readdir_r()* returns 0). On error, *readdir_r()* doesn't return −1, but instead returns a positive integer corresponding to one of the *errno* values.

On Linux, the *d_name* field of the *dirent* structure is sized as an array of 256 bytes, which is long enough to hold the largest possible filename. Although several other UNIX implementations define the same size for *d_name*, SUSv3 leaves this point unspecified, and some UNIX implementations instead define the field as a 1-byte array, leaving the calling program with the task of allocating a structure of the correct size. When doing this, we should size the *d_name* field as one greater (for the terminating null byte) than the value of the constant NAME_MAX. Portable applications should thus allocate the *dirent* structure as follows:

```
struct dirent *entryp;
size_t len;

len = offsetof(struct dirent, d_name) + NAME_MAX + 1;
entryp = malloc(len);
if (entryp == NULL)
    errExit("malloc");
```

Using the offsetof() macro (defined in <stddef.h>) avoids any implementation-specific dependencies on the number and size of fields in the *dirent* structure preceding the *d_name* field (which is always the last field in the structure).

The `offsetof()` macro takes two arguments—a structure type and the name of a field within that structure—and returns a value of type *size_t* that is the offset in bytes of the field from the beginning of the structure. This macro is necessary because a compiler may insert padding bytes in a structure to satisfy alignment requirements for types such as *int*, with the result that a field's offset within a structure may be greater than the sum of the sizes of the fields that precede it.

18.9 File Tree Walking: *nftw()*

The *nftw()* function allows a program to recursively walk through an entire directory subtree performing some operation (i.e., calling some programmer-defined function) for each file in the subtree.

> The *nftw()* function is an enhancement of the older *ftw()* function, which performs a similar task. New applications should use *nftw()* (*new ftw*) because it provides more functionality, and predictable handling of symbolic links (SUSv3 permits *ftw()* either to follow or not follow symbolic links). SUSv3 specifies both *nftw()* and *ftw()*, but the latter function is marked obsolete in SUSv4.
>
> The GNU C library also provides the BSD-derived *fts* API (*fts_open()*, *fts_read()*, *fts_children()*, *fts_set()*, and *fts_close()*). These functions perform a similar task to *ftw()* and *nftw()*, but offer greater flexibility to an application walking the tree. However, this API is not standardized and is provided on few UNIX implementations other than BSD descendants, so we omit discussion of it here.

The *nftw()* function walks through the directory tree specified by *dirpath* and calls the programmer-defined function *func* once for each file in the directory tree.

```
#define _XOPEN_SOURCE 500
#include <ftw.h>

int nftw(const char *dirpath,
         int (*func) (const char *pathname, const struct stat *statbuf,
                      int typeflag, struct FTW *ftwbuf),
         int nopenfd, int flags);
```

Returns 0 after successful walk of entire tree, or −1 on error, or the first nonzero value returned by a call to *func*

By default, *nftw()* performs an unsorted, preorder traversal of the given tree, processing each directory before processing the files and subdirectories within that directory.

While traversing the directory tree, *nftw()* opens at most one file descriptor for each level of the tree. The *nopenfd* argument specifies the maximum number of file descriptors that *nftw()* may use. If the depth of the directory tree exceeds this maximum, *nftw()* does some bookkeeping, and closes and reopens descriptors in order to avoid holding open more than *nopenfd* descriptors simultaneously (and consequently runs more slowly). The need for this argument was greater under older UNIX implementations, some of which had a limit of 20 open file descriptors per process. Modern UNIX implementations allow a process to open a large number of file descriptors, and thus we can specify a generous number here (say 10 or more).