Operating systems

Filesystem

Created by
Enrico Fraccaroli
enrico.fraccaroli@gmail.com

Table of Contents

File

File

Operations

Open/Create a file (1/4)

The `open` system call either opens an existing file. Alternately, it can first create and then open a new file.

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

// Returns file descriptor on success, or -1 on error
int open(const char *pathname, int flags, .../*mode_t mode */);
```

If `open` succeeds, it returns a file descriptor that is used to refer to the file in subsequent system calls.
The file to be opened/created is identified by the pathname argument.

Open/Create a file (2/4)

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

// Returns file descriptor on success, or -1 on error
int open(const char *pathname, int flags, .../*mode_t mode */);
```

The `flags` argument is a bit mask of one or more of the following constants that specifies the access mode for the file.

Flag	Description
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_TRUNC	Truncate existing file to zero length
O_APPEND	Writes are always appended to end of file
O_CREAT	Create file if it doesn't already exist
O_EXCL	With O_CREAT, ensure that this call creates the file.

When a new file is created, then also the system call's `mode` argument is considered.

Open/Create a file (3/4)

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

// Returns file descriptor on success, or -1 on error
int open(const char *pathname, int flags, .../*mode_t mode */);
```

The `mode` argument is a bit mask of one or more of the following constants that specifies the permissions for the new file.

Flag	Description
S_IRWXU	user has read, write, and execute permission
S_IRUSR	user has read permission
S_IWUSR	user has write permission
S_IXUSR	user has execute permission
S_IRWXG	group has read, write, and execute permission
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	others has read, write, and execute permission
S_IROTH	others has read permission
S_IWOTH	others has write permission
S_IXOTH	others has execute permission

Be aware of the process's umask! (1/2)

The user file-creation mask (`umask`) is a process attribute that specifies which permission bits should always be turned off when new files (directories, FIFOs, ...) are created by the process. In most shells, the umask has the default value 022 (−−−−−w−−w−).

The permissions assigned to a new file is: (mode & ∼umask)

```
Requested file perms:    rw-rw----   (<- this is what we asked)
Process umask:           ----w--w-   (<- this is what we are denied)
Actual file perms:       rw-r-----   (<- So, this is what we get)
```

Be aware of the process's umask! (2/2)

You can manage the `umask` in a shell by using the command:

```
user@localhost[~]$ umask [-S] [expression]
```

You can view the current mask:

```
user@localhost[~]$ umask
022
```

You can set the current mask:

```
user@localhost[~]$ umask 022
```

You can also display the current mask using symbolic notation:

```
user@localhost[~]$ umask -S
u=rwx,g=rx,o=rx
```

Open/Create a file (4/4)

Examples:

```
int fd;
// Open existing file for only writing.
fd = open("myfile", O_WRONLY);

// Open new or existing file for reading/writing, truncating
//  to zero bytes; file permissions read+write only for owner.
fd = open("myfile1", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);

// Create and open a new file for reading/writing; file
//  permissions read+write only for owner.
fd = open("myfile2", O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
```

Read from a file descriptor (1/3)

The `read` system call reads data from a file descriptor.

```
#include<unistd.h>

// Returns number of bytes read, or -1 on error
ssize_t read(int fd, void *buf, size_t count);
```

The `count` argument specifies the maximum number of bytes to read from a file descriptor `fd`. The `buf` argument supplies the address of a memory buffer into which the read input data is stored.

Read from a file descriptor (2/3)

Example: reading up to MAX_READ bytes from a file.

```
// Open existing file for reading.
int fd = open("myfile", O_RDONLY);
if (fd == -1)
    errExit("open");

// A MAX_READ bytes buffer.
char buffer[MAX_READ + 1];

// Reading up to MAX_READ bytes from myfile.
ssize_t numRead = read(fd, buffer, MAX_READ);
if (numRead == -1)
    errExit("read");
```

Note: with a file, the zero value returned by `read` means end-of-file EOF

Read from a file descriptor (3/3)

Example: reading up to `MAX_READ` bytes from a terminal!

```
// A MAX_READ bytes buffer.
char buffer[MAX_READ + 1];

// Reading up to MAX_READ bytes from STDIN.
ssize_t numRead = read(STDIN_FILENO, buffer, MAX_READ);
if (numRead == -1)
    errExit("read");

buffer[numRead] = '\0';
printf("Input data: %s\n", buffer);
```

Note: with a terminal, `read` reads characters up to the next newline (\n) character.

Write to a file descriptor (1/3)

The *write* system call write data to a file descriptor.

```c
#include <unistd.h>

// Returns number of bytes written, or -1 on error.
ssize_t write(int fd, void *buf, size_t count);
```

The *count* argument specifies the number of bytes of a buffer pointed by `buf` that has to be written to a file descriptor referred by `fd`.

Write to a file descriptor (2/3)

Example: writing the string "Ciao Mondo" in a file

```
// Open existing file for writing.
int fd = open("myfile", O_WRONLY);
if (fd == -1)
    errExit("open");

// A buffer collecting the string.
char buffer[] = "Ciao Mondo";

// Writing up to sizeof(buffer) bytes into myfile.
ssize_t numWrite = write(fd, buffer, sizeof(buffer));
if (numWrite != sizeof(buffer))
    errExit("write");
```

Write to a file descriptor (3/3)

Example: writing the string "Ciao Mondo" in a terminal.

```
// A buffer collecting a string.
char buffer[] = "Ciao Mondo";

// Writing up tp sizeof(buffer) bytes on STDOUT.
ssize_t numWrite = write(STDOUT_FILENO, buffer, sizeof(buffer));
if (numWrite != sizeof(buffer))
    errExit("write");
```

Adjust the offset location of a file (1/2)

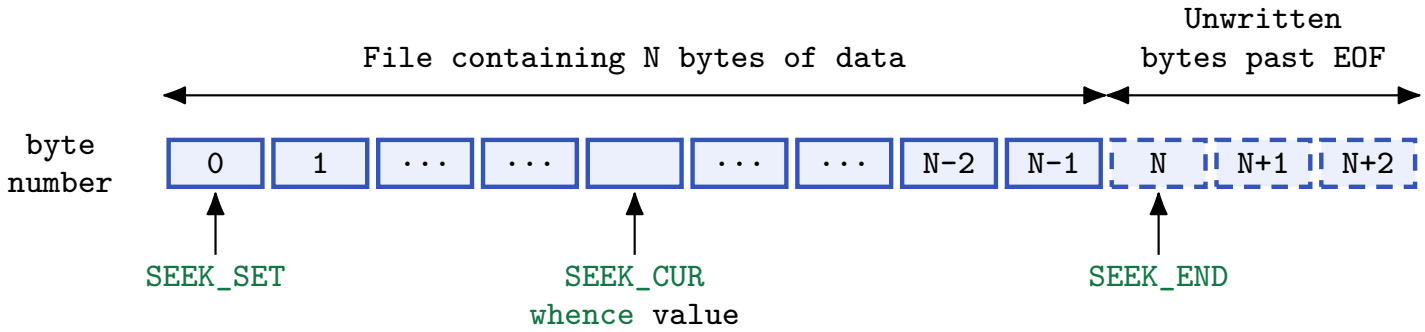
For each open file, the kernel saves a *file offset*, namely the location in the file at which the next `read`, or `write`, will start. The `lseek` system call adjusts the offset location of a open file.

```c
#include <unistd.h>

// Returns the resulting offset location, or -1 on error.
off_t lseek(int fd, off_t offset, int whence);
```

The `fd` argument speficies the file descriptor of the open file, `offset` specifies a value in byte, meanwhile `whence` indicates the base point from which `offset` is to be interpreted.

Adjust the offset location of a file (2/2)




```c
// first byte of the file.
off_t current = lseek(fd1, 0, SEEK_SET);
// last byte of the file.
off_t current = lseek(fd2, -1, SEEK_END);
// 10th byte past the current offset location of the file.
off_t current = lseek(fd3, -10, SEEK_CUR);
// 10th byte after the current offset location of the file.
off_t current = lseek(fd4, 10, SEEK_CUR);
```

Note: For `SEEK_SET`, `offset` must be a positive value.

Close a file descriptor

The `close` system call closes an open file descriptor.

```c
#include <unistd.h>

// Returns 0 on success, or -1 on error.
int close(int fd);
```

Advice:
Even if the process's file descriptors are automatically close when the process terminates, it is usually good practice to close unneeded file descriptors explicitly. This makes the code more readable and reliable in the face of subsequent modifications.

Remove a file

The `unlink` system call remove a link and, if that is the last link to the file, also removes the file itself.

```c
#include <unistd.h>

// Returns 0 on success, or -1 on error
int unlink(const char *pathname);
```

Warning:
`unlink` cannot remove a directory (see `rmdir`).

Example

Example: Create, close and remove a file

```c
// Create a new file named myFile.
int fd = open("myFile", O_CREAT | O_WRONLY);
// ... only writes as myFile is open in write-only
// Close the file descriptor fd
close(fd);
// Unlink (remove) myFile
unlink("myFile");
```

File

Attributes

Retrieve the attributes of a file (1/4)

The `stat`, `lstat`, and `fstat` system calls retrieve information about a file.[1]

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

// Return 0 on success or -1 on error.
int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

These three system calls differ only in the way that the file is specified:

- `stat` returns information about a named file;
- `lstat` returns information about a symbolic link;
- `fstat` is similar to *stat*, except that a file is referred to by a file descriptor rather than its pathname.

[1] In Linux a directory is a file. The following system calls for file attributes can also be applied for directories.

Retrieve the attributes of a file (2/4)

All of these system calls return a stat structure in the buffer pointed to by `statbuf`. This structure has the following form:

```
struct stat {
    dev_t st_dev;           // IDs of device on which file resides.
    ino_t st_ino;           // I-node number of file.
    mode_t st_mode;         // File type and permissions.
    nlink_t st_nlink;       // Number of (hard) links to file.
    uid_t st_uid;           // User ID of file owner.
    gid_t st_gid;           // Group ID of file owner.
    dev_t st_rdev;          // IDs for device special files.
    off_t st_size;          // Total file size (bytes).
    blksize_t st_blksize;   // Optimal block size for I/O (bytes).
    blkcnt_t st_blocks;     // Number of (512B) blocks allocated.
    time_t st_atime;        // Time of last file access.
    time_t st_mtime;        // Time of last file modification.
    time_t st_ctime;        // Time of last status change.
};
```

Retrieve the attributes of a file (3/4)

Device IDs and i-node number: The `st_dev` field identifies the device on which the file resides. The `st_ino` field contains the i-node number of the file. The combination of `st_dev` and `st_ino` uniquely identifies a file across all file systems.

File ownership: The `st_uid` and `st_gid` fields identify, respectively, the owner (user ID) and group (group ID) to which the file belongs.

Link count: The `st_nlink` field is the number of (hard) links to the file.

File timestamps: The `st_atime`, `st_mtime`, and `st_ctime` fields contain, respectively, the times of last file access, last file modification, and last status change (i.e., last change to the file's i-node information) in time format of seconds.

Retrieve the attributes of a file (4/4)

File size, blocks allocated, and optimal I/O block size

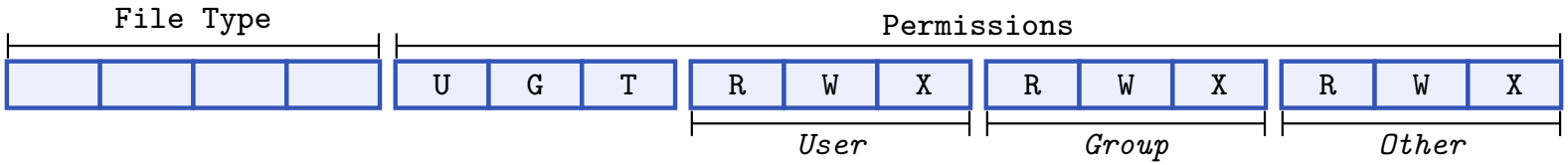
For regular files, the `st_size` field is the total size of the file in bytes. For a symbolic link, this field contains the length (in bytes) of the pathname pointed to by the link.

The `st_blocks` field indicates the number of blocks actually allocated to the file in 512-byte block units (might be smaller than expected from the corresponding `st_size` if the file contains holes).

The `st_blksize` is the optimal block size (in bytes) for I/O on files on this file system. I/O in blocks smaller than this size is less efficient. A typical value returned in `st_blksize` is 4096.

File type and permissions (1/5)

The `st_mode` field is a bit mask serving the dual purpose of identifying the file type and specifying the file permissions. The bits of this field are laid as follow:

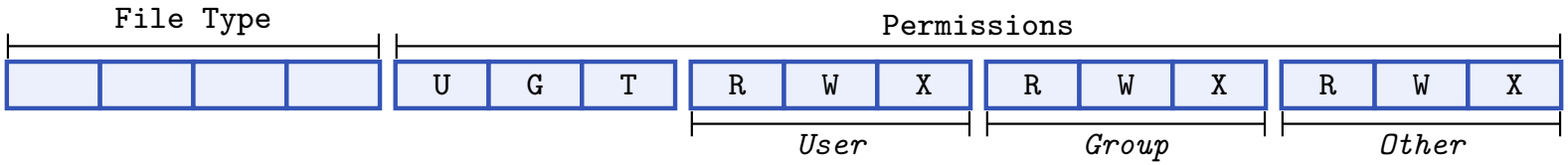


Constant	Test macro	File type
S_IFREG	S_ISREG()	Regular file
S_IFDIR	S_ISDIR()	Directory
S_IFCHR	S_ISCHR()	Character device
S_IFBLK	S_ISBLK()	Block device
S_IFIFO	S_ISFIFO()	FIFO or pipe
S_IFSOCK	S_ISSOCK()	Socket
S_IFLNK	S_ISLNK()	Symbolic link

The file type can be extracted from this field by ANDing (&) with the constant `S_IFMT`, and then comparing the result with a range of constants.

As this is a common operation, standard macros are provided.

Example



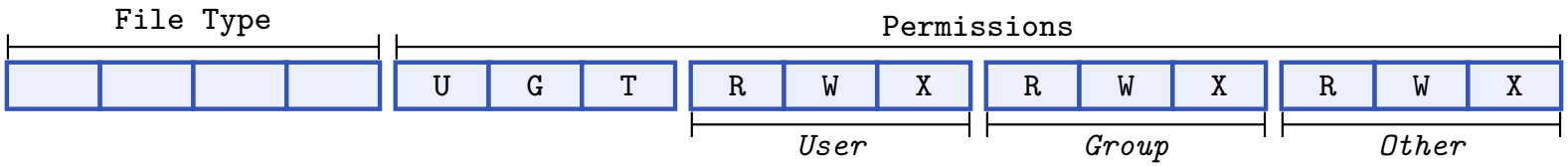
How to check if a file is a regular file

```c
char pathname[] = "/tmp/file.txt";
struct stat statbuf;
// Getting the attributes of /tmp/file.txt
if (stat(pathname, &statbuf) == -1)
    errExit("stat");

// Checking if /tmp/file.txt is a regular file
if ((statbuf.st_mode & S_IFMT) == S_IFREG)
    printf("regular file!\n");

// Equivalently, checking if /tmp/file.txt is a
//  regular file by S_ISREG macro.
if (S_ISREG(statbuf.st_mode))
    printf("regular file!\n");
```

File type and permissions (2/5)

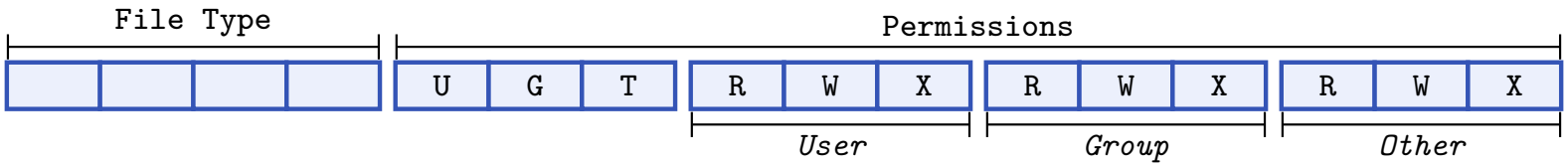


The bits labelled U, and G are applied for executables.

- *set-user-ID*: if it is set, then the effective user ID of the process is made the same as the owner of the executable;
- *set-group-ID*: if it is set, then the effective group ID of the process is made the same as the owner of the executable.

The bit labelled T, which is named Sticky-bit, acts as the *restricted deletion* flag for directory.
Setting this bit on a directory means that an unprivileged process can unlink (`unlink()`, `rmdir()`) and rename (`rename()`) files in the directory only if it has write permission on the directory and owns either the file or the directory.

File type and permissions (3/5)



The remaining 9 bits form the mask defining the permissions that are granted to various categories of users accessing the file. The file permissions mask divides the world into three categories:

- *Owner*: The permissions granted to the owner of the file.
- *Group*: The permissions granted to users who are members of the file's group.
- *Other*: The permissions granted to everyone else.

Three permissions may be granted to each user category:

- *Read*: The contents of the file may be read.
- *Write*: The contents of the file may be changed.
- *Execute*: The file may be executed.

File type and permissions (4/5)

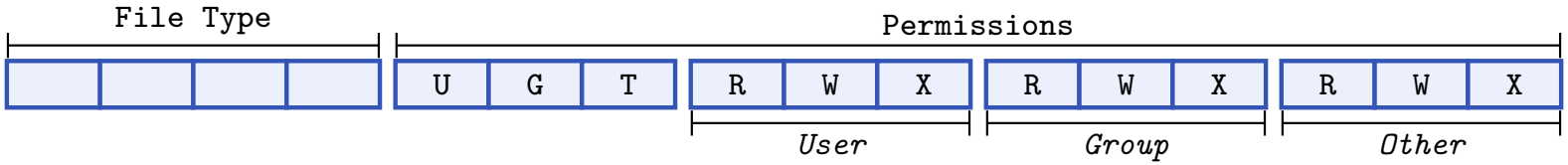
Directories have the same permission scheme as files. However, the three permissions are interpreted differently:

- *Read*: The contents (i.e., the list of filenames) of the directory may be listed
- *Write*: Files may be created in and removed from the directory
- *Execute*: Files within the directory may be accessed.

When accessing a file, execute permission is required on all of the directories listed in the pathname. Example:
Suppose we want to read the file /home/user1/secrets/passwords.txt, then we have to have the execute permission for the directories:

/ home/ user1/ secrets/

File type and permissions (5/5)



The <sys/stat.h> header file defines constants that can be ANDed (&) with st_mode of the stat structure, in order to check whether particular permission bits are set.

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

Table: Constants for file permission bits

Example

Displaying the user's permission of a file:

```
char pathname[] = "/tmp/file.txt";
struct stat statbuf;

// Getting the attributes for the executable /tmp/a.out
if (stat(pathname, &statbuf) == -1)
    errExit("stat");

// printing out the user's permissions
printf("user's permissions:  %c%c%c\n",
    (statbuf.st_mode & S_IRUSR)? 'r' : '-',
    (statbuf.st_mode & S_IWUSR)? 'w' : '-',
    (statbuf.st_mode & S_IXUSR)? 'x' : '-');
```

Check the accessibility of a file

The `access` system call checks the accessibility of the file specified in pathname based on a process's real user and group IDs.

```
#include <unistd.h>

// Returns 0 id all permissions are granted, otherwise -1
int access(const char *pathname, int mode):
```

If `pathname` is a symbolic link, `access` dereferences it. The mode argument is a bit mask consisting of one or more of the following constants:

Constant	Description
F_OK	Does the file exist?
R_OK	Can the file be read?
W_OK	Can the file be written?
X_OK	Can the file be executed?

Example

Checking the access to a file

```
char pathname[] = "/tmp/file.txt";

// Checking if /tmp/file.txt exists, can be read and
//  written by the current process.
if (access(pathname, F_OK | R_OK | W_OK) == -1)
    printf(" It looks like that I cannot read/write file.txt :(\n)")
```

Change the permissions of a file

The `chmod` and `fchmod` system calls change the permissions of a file.

```c
// All return 0 on success, or -1 on error
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);

#define _BSD_SOURCE
#include <sys/stat.h>

int fchmod(int fd, mode_t mode);
```

The `chmod` system call changes the permissions of the file named in `pathname`. The `fchmod` system call changes the permissions on the file referred to by the open file descriptor *fd*.
The mode argument specifies the new permissions of the file by ORing (|) the permission bits listed in Table 1

Example

Changing the permission of a file

```
char pathname[] = "/tmp/file.txt";

struct stat sb;
if (stat(pathname, &sb) == -1)
    errExit("stat");

// Owner-write on, other-read off, remaining bits unchanged.
mode_t mode = (sb.st_mode | S_IWUSR) & ~S_IROTH;

if (chmod(pathname, mode) == -1)
    errExit("chmod");
```

Directory

Directory

Operations

Create a new directory

The `mkdir` system call creates a new directory.

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

// Returns 0 on success, or -1 on error.
int mkdir(const char *pathname, mode_t mode);
```

The pathname argument specifies the pathname of the new directory. This pathname may be relative or absolute. If a file with this pathname already exists, then the call fails with the error EEXIST.

The `mode` argument specifies the permissions for the new directory (see chapter file system, system call `open`)

Remove a new directory

The `rmdir` system call removes a directory.

```c
#include <unistd.h>

// Returns 0 on success, or -1 on error.
int rmdir(const char *pathname);
```

In order for `rmdir` to succeed, the directory must be empty. If the final component of pathname is a symbolic link, it is not dereferenced; instead, the error `ENOTDIR` results.

Example

Create and delete a new directory

```
// Create a new directory with name myDir.
int res = mkdir("myDir", S_IRUSR | S_IXUSR);
if (res == 0) {
    printf("The directory myDir was created!\n");

    // Remove the directory with name myDir.
    res = rmdir("myDir");
    if (res == 0)
        printf("The directory myDir was removed!\n");
}
```

Open and close a directory

The `opendir` and `closedir` system calls respectively open and close a directory.

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

// Returns directory stream handle, or NULL on error
DIR *opendir(const char *dirpath);

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

Upon return from `opendir`, the so-called *directory stream* (namely, DIR *) is positioned at the first entry in the directory list.

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

Read a directory (1/2)

The `readdir` system call reads the content of a directory.

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

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

Each call to `readdir` reads the next file/directory entry from the directory stream referred to by `dirp`. Each entry is a struct defined as follow:

```c
struct dirent {
    ino_t d_ino;            // File i-node number.
    unsigned char d_type;   // Type of file.
    char d_name[256];       // Null-terminated name of file.
    //...
}
```

Read a directory (2/2)

The C library defines the following macro constants for the value returned in d_type:

Constant	**File type**
DT_BLK	block device
DT_CHR	character device
DT_DIR	directory
DT_FIFO	named pipe (FIFO)
DT_LNK	symbolic link
DT_REG	regular file
DT_SOCK	UNIX socket

Example

Displaying only the regular files in a directory

```
DIR *dp = opendir("myDir");
if (dp == NULL) return -1;

errno = 0;
struct dirent *dentry;
// Iterate until NULL is returned as a result.
while ( (dentry = readdir(dp)) != NULL ) {
    if (dentry->d_type == DT_REG)
        printf("Regular file: %s\n", dentry->d_name);
    errno = 0;
}
// NULL is returned on error, and when the end-of-directory is reached!
if (errno != 0)
    printf("Error while reading dir.\n");
closedir(dp);
```