File and Directory Management

CISC 3350 — CUNY Brooklyn College
Lecture Notes

File and Directory Management: Intro

While topics 4, 5, and 6 focused on reading from files and writing to files, this topic deals with manipulating/managing files and their **metadata** (= data about the files, e.g., filenames, types, sizes, owner names, locations on disk, etc.).

Each file's metadata is stored in an **inode** data structure, which the OS can access via a unique numerical value called an **inode number** or **i-number**.

Surprisingly, the filename and the directory name of the file aren't stored in the file's inode: this chapter explains where/how these details are actually stored.

All the inodes of all the computer's files are stored on the disk, but not necessarily right next to the files' contents.

To see the inode numbers of all the files in the current folder, use the following Linux command:

```
$ ls -i
```

Getting/Reading a File's Metadata

<table>
<tr><td>Name: <code>stat()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: reads the metadata of the file with filename <code>path</code> and pastes it into <code>buf</code> (an instance of the <code>stat</code> structure.)</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <sys/stat.h> // Defines stat().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int stat (const char *path, struct stat *buf);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>path</code>: a double-quoted filename.
<code>buf</code>: a <code>stat</code> structure that will afterwards contain the file's metadata.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:

```
struct stat file_status;
if (stat ("/cisc3350/assignments/text.txt", &file_status) == -1)
    perror("stat");
```

</td></tr>
<tr><td colspan="3">Other variations: <code>fstat()</code>, <code>lstat()</code></td></tr>
</table>

Getting/Reading a File's Metadata

<table>
<tr><td colspan="2">Name: <code>fstat()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: reads the metadata of the file with file descriptor <code>fd</code> and pastes it into <code>buf</code> (an instance of the <code>stat</code> structure.)</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <sys/stat.h> // Defines fstat().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int fstat (int fd, struct stat *buf);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>fd</code>: the file descriptor (<code>fd</code>) of a file that was open with <code>open()</code>.
<code>buf</code>: a <code>stat</code> structure that will afterwards contain the file's metadata.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<pre>struct stat file_status;
int fd = open("/cisc3350/assignments/text.txt", O_RDWR);
if (fstat (fd, &file_status) == -1)
    perror("fstat");</pre></td></tr>
<tr><td colspan="4">Other variations: <code>stat()</code>, <code>lstat()</code></td></tr>
</table>

Getting/Reading a File's Metadata

<table>
<tr><td>Name: <code>lstat()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: reads the metadata of the file with filename <code>path</code> (or, if the file is a symbolic link, reads the metadata of the symbolic link, not the target file) and pastes it into <code>buf</code> (an instance of the <code>stat</code> structure.)</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <sys/stat.h> // Defines fstat().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int lstat (const char *path, struct stat *buf);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>path</code>: a double-quoted filename.
<code>buf</code>: a <code>stat</code> structure that will afterwards contain the file's metadata.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:

```
struct stat file_status;
if (stat ("/cisc3350/assignments/sym-link", &file_status) == -1)
    perror("stat");
```

</td></tr>
<tr><td colspan="3">Other variations: <code>stat()</code>, <code>fstat()</code></td></tr>
</table>

More Info on the `stat` Structure

After one of the previous 3 `stat()` syscalls runs, the `stat` structure will contain the file's metadata information:

```
struct stat {
    dev_t             st_dev;     // ID of the device where the file is stored.
    ino_t             st_ino;     // File's inode number.
    mode_t            st_mode;    // File type and permissions to use the file (read, write, and execute.)
    nlink_t           st_nlink;   // Number of hard links the file has.
    uid_t             st_uid;     // User ID of the file's owner user.
    gid_t             st_gid;     // Group ID of the file's owning group.
    dev_t             st_rdev;    // Device ID (if the file is itself a device. Otherwise, it will be zero (0).
    off_t             st_size;    // Size of the file in bytes.
    blksize_t         st_blksize; // Block size for the file system's I/O operations.
    blkcnt_t          st_blocks;  // Number of blocks the file occupies on the disk, in units of 512 bytes.
    struct timespec   st_atime;   // Last time of accessing the file.
    struct timespec   st_mtime;   // Last time of modifying (writing to) the file.
    struct timespec   st_ctime;   // Last time of modifying the file's metadata/inode.
};
```

get_file_size.c: Getting a File's Size with `stat()`


```c
// A program outputing the size of a file whose name is
//     passed as the 1st argument (argv[1]) to the
//     program.

// This program is taken from Linux System Programming:
//     Talking Directly to the Kernel and C Library,
//     2nd Edition, by Love. ISNB: 978-1-44933953-1,
//     page 244.

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
```

find_file_type.c: Getting a File's Type with `stat()`


```c
// A program outputing the type of a file whose name is
//     passed as the 1st argument (argv[1]) to the
//     program.

// This program is taken from Linux System Programming:
//     Talking Directly to the Kernel and C Library,
//     2nd Edition, by Love. ISNB: 978-1-44933953-1,
//     pages 244-245.

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
```

Setting a File's Permissions

<table>
<tr><td>Name: <code>chmod()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: changes the permissions of the file <code>path</code> to <code>mode</code>. Works only if your effective UID matches the file's owner ID.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <sys/stat.h> // Defines chmod().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int chmod (const char *path, mode_t mode);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>path</code>: a double-quoted filename.
<code>mode</code>: the new permissions you want to give to the file.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:
<pre>// Question: what kinds of permissions are we setting below?
if (chmod ("text.txt", S_IRUSR | S_IWUSR) == -1) // Same as $ chmod u+rw "text.txt"
    perror("chmod");</pre></td></tr>
<tr><td colspan="3">Other variations: <code>fchmod()</code></td></tr>
</table>

Setting a File's Permissions

<table>
<tr><td>Name: <code>fchmod()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: changes the permissions of the file with descriptor <code>fd</code> to <code>mode</code> only if your effective UID matches the owner's ID.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <sys/stat.h> // Defines fchmod().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int fchmod (int fd, mode_t mode);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>fd</code>: the file descriptor (<code>fd</code>) of a file that was open with <code>open()</code>.
<code>mode</code>: the new permissions you want to give to the file.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:
<pre>int fd = open("text.txt", O_RDWR);
if (fchmod (fd, S_IRUSR | S_IWUSR) == -1) // Same as $ chmod 600 "text.txt", too.
    perror("fchmod");</pre></td></tr>
<tr><td colspan="3">Other variations: <code>chmod()</code></td></tr>
</table>

permissions_changing.c: Setting a File's Permissions


```c
/* A program changing the permissions of a file
 *    and printing the update to the terminal.
 *
 *    Miriam Briskman, 5/8/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
```

Setting a File's Permissions

Any questions so far?

Setting a File's Owning User and Group

<table>
<tr><td colspan="2">Name: <code>chown()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: sets the owning user and the owning group of the file with filename <code>path</code> to <code>owner</code> and <code>group</code>, respectively.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <sys/stat.h> // Defines chown().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int chown (const char *path, uid_t owner, gid_t group);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>path</code>: a double-quoted filename.
<code>owner</code>: the user ID of the new owner of the file. If you pass <code>-1</code>, owner won't change.
<code>group</code>: the group ID of the new owning group of the file. If you pass <code>-1</code>, group won't change.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>if (chown ("text.txt", 4343, -1) == -1) // owner = 4343, but don't change group.
perror("chown");</code></td></tr>
<tr><td colspan="4">Other variations: <code>fchown()</code>, <code>lchown()</code></td></tr>
</table>

Setting a File's Owning User and Group

Name: `fchown()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet
What it does: sets the owning user and the owning group of the file with descriptor `fd` to `owner` and `group`, respectively.		
What libraries you must include: `#include <sys/stat.h> // Defines fchown().`		
Syntax: `int fchown (int fd, uid_t owner, gid_t group);`		
Description of arguments: `fd`: the file descriptor (`fd`) of a file that was open with `open()`. `owner`: the user ID of the new owner of the file. If you pass `-1`, owner won't change. `group`: the group ID of the new owning group of the file. If you pass `-1`, group won't change.		
What type it returns: `int`	**On success:** `0`	
	On failure: `-1`	
	If failure, does it set `errno`? Yes	
Quick example: `if (fchown (fd, -1, 259) == -1) // group = 259, but don't change owner.` `  perror("fchown");`		
Other variations: `chown()`, `lchown()`		

Setting a File's Owning User and Group

<table>
<tr><td colspan="2">Name: <code>lchown()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: sets the owning user and the owning group of the file with filename <code>path</code> to <code>owner</code> and <code>group</code>, respectively. If <code>path</code> is a symbolic link, the owner and the group of the link change, not these of the target file.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <sys/stat.h> // Defines lchown().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int lchown (const char *path, uid_t owner, gid_t group);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>path</code>: a double-quoted filename.
<code>owner</code>: the user ID of the new owner of the file. If you pass <code>-1</code>, owner won't change.
<code>group</code>: the group ID of the new owning group of the file. If you pass <code>-1</code>, group won't change.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<code>if (lchown ("symlink", 15, 7845) == -1) // owner = 15, group = 7845.
    perror("lchown");</code></td></tr>
<tr><td colspan="4">Other variations: <code>chown()</code>, <code>fchown()</code></td></tr>
</table>

Example of Changing a File's Group


```
// getgrnam() returns information on
a
// group, given the group's name:
struct group *gr = getgrnam ("cats");
if (!gr) // Probably an invalid
group.
{
    perror ("getgrnam");
    exit (EXIT_FAILURE);
}

// Set info.txt's group to 'cats':
if (chown("info.txt", -1, gr-
>gr_gid))
    perror ("chown");
```

The code on the left changes the group of the `info.txt` file to `cats`. The `group` data structure is defined as follows:

```
struct group {
    char  *gr_name;   /* group name */
    char  *gr_passwd; /* group password */
    gid_t gr_gid;     /* group ID */
    char  **gr_mem;   /* NULL-terminated pointers array
                         to names of group members */
};
```

After running this code, a call to `$ ls -l` will show that the file's group name indeed changed to `cats`.

Example of Changing a File's Group

Any questions so far?

Directories

A **directory** (folder) is a map between filenames and inode numbers. To find the inode associated with a file, the OS first reads a directory file and finds which inode is related to the given filename. This means that the directory file is the place where the names of files are stored! A directory file is stored on the disk just as other files are.

Nested directories make up a **path** (each part of a path is called a **dentry** = **d**irectory **entry**.) An example of a path: `/public_html/hello.txt`.

Directories can contain **subdirectories** (= folders inside folders.) Except for the `root` directory (that is, `/`), every directory itself is a subdirectory of another folder!

A **pathname** consists of a file's name along with all its parent directories, e.g., `/public_html/cisc3350/lecture_notes/info.txt`.

A "fully qualified" path starts at `/` (root directory) and is called an **absolute path**. A path that starts with any other directory is called a **relative path**.

Directories

File and directory names can contain any character except `/` and null characters, but it's standard practice to only use printable (keyboard) characters when naming folders.

- The reason that the slash `/` can't be a part of a folder's name is that `/` is used as a separator between folder names in a path, e.g., `/public_html/hello.txt`.

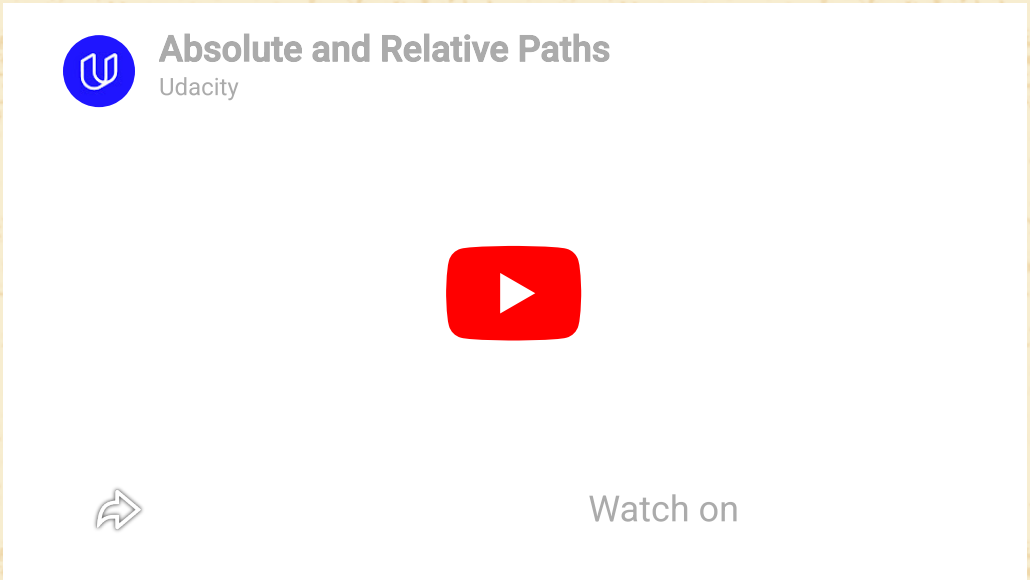
All modern Unix filesystems allow at least 255 bytes for each filename. Notice that some languages use multibyte characters, such as Java.

Every directory contains two special directories:

- A dot (`.`) represents the current directory itself.
  - That's why we run programs by typing: `./progname`, to indicate that the program we want to run is located in the current directory.
- Two dots (`..`) represent the current directory's parent directory, so `/public_html/cisc3350/..` refers to `/public_html/`.

Directories

The video at https://www.youtube.com/watch?v=ephId3mYu9o introduces the special directory symbols that you can use when referring to various files and directories on your Linux account/device:



https://www.youtube.com/watch?v=ephId3mYu9o

Directories

Any questions so far?

Finding What the Current Working Directory (cwd) Is

<table>
<tr><td colspan="2">Name: <code>getcwd()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: Pastes the absolute path to the current working directory of this running process into <code>buf</code>, which is of length <code>size</code> bytes. That is, after <code>getcwd()</code> returns, <code>buf</code> will contain the current working directory.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines getcwd().</code></td></tr>
<tr><td colspan="4">Syntax: <code>char * getcwd (char *buf, size_t size);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>buf</code>: a char array where the path to the directory will be copied to.
<code>size</code>: the size of <code>buf</code> in bytes.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>char*</code></td><td colspan="2">On success: A non-<code>NULL</code> pointer to <code>buf</code>.</td></tr>
<tr><td colspan="2">On failure: <code>NULL</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:

```
char buf[1000]; // buffer on the stack
if (!getcwd (buf, 1000))
   /* Handle the errors */
printf ("The current working directory is: %s\n", buf);
```

</td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Linux's Version of `getcwd()`

The previous slide described `getcwd()` as the POSIX standard defines it. However, Linux provides additional features and flexibilities in using `getcwd()`:

- If you pass `NULL` for `buf`, Linux will automatically allocate and return a buffer of length `size` for you!
- If you pass `0` for `size`, Linux will set the size of that newly-allocated array to exactly the current working directory's path length (plus the null character at the end.)
- After you finish using the array, you must `free()` it.

```
char *cwd = getcwd (NULL, 0);
if (!cwd) {
    perror ("getcwd");
    exit (EXIT_FAILURE);
}
printf ("cwd = %s\n", cwd);
free (cwd);
```

Changing the Current Working Directory (cwd)

<table>
<tr><td colspan="2">Name: <code>chdir()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: sets the current working directory to <code>path</code>. This is the programming way of calling the <code>$ cd</code> command on Linux. A process can change only its own current working directory but not the directories of other processes.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines chdir().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int chdir (const char *path);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>path</code>: a double-quoted path to directory you want to go to.</td></tr>
<tr><td rowspan="3" colspan="2">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<code>if (chdir ("../../") == -1)
  /* Handle the errors */</code></td></tr>
<tr><td colspan="4">Other variations: <code>fchdir()</code></td></tr>
</table>

Changing the Current Working Directory (`cwd`)

Name: `fchdir()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet
What it does: sets the current working directory to the directory file with descriptor `fd`. This is the programming way of calling the `$ cd` command on Linux. A process can change only its own current working directory but not those of other processes.		
What libraries you must include: `#include <unistd.h> // Defines fchdir().`		
Syntax: `int fchdir (int fd);`		
Description of arguments: `fd`: the file descriptor (`fd`) of a directory file that was open with `open()`.		
What type it returns: `int`	**On success:** `0`	
	On failure: `-1`	
	If failure, does it set `errno`? Yes	

Quick example:

```
int fd = open("./cisc3350-hw/", O_RDWR);
if (fchdir (fd) == -1)
   /* Handle the errors */
```

Other variations: `chdir()`

switch_directories.c: Changing Directories Twice


```
// A program switching the current directory D to
//    argv[1] and then back to the previous
//    directory D.

// This program is based on Linux System Programming:
//     Talking Directly to the Kernel and C Library,
//     2nd Edition, by Love. ISNB: 978-1-44933953-1,
//     page 265.

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
```

Creating a New Directory

<table>
<tr><td>Name: <code>mkdir()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: creates a new folder with the name (and location) <code>path</code>, which may be either an absolute or relative path, and sets the permissions at <code>mode</code> (reading, writing, and execution) for this new directory.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <sys/stat.h> // Defines mkdir().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int mkdir (const char *path, mode_t mode);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>path</code>: the name of and path to the new directory that will be created.
<code>mode</code>: the usage permissions of this directory (see more on permissions in <code>open()</code>.)</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:
<code>if (mkdir ("./my-new-dir", S_IRWXU | S_IRWXG | S_IRWXO) == -1)
  /* Handle the errors */</code></td></tr>
<tr><td colspan="3">Other variations: N/A</td></tr>
</table>

Creating a New Directory

<table>
<tr><td>Name: <code>rmdir()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: deletes the directory <code>path</code>. This directory must be empty and must not be either of the <code>.</code> and <code>..</code> directories.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <unistd.h> // Defines rmdir().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int rmdir (const char *path);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>path</code>: the name of and path to the directory that will be deleted.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:<pre>if (rmdir ("./my-new-dir") == -1)
    /* Handle the errors */</pre></td></tr>
<tr><td colspan="3">Other variations: N/A</td></tr>
</table>

Note that Linux doesn't have a function that deletes folders recursively, as the command `$ rm -r` does. You can instead implement such a function on your own using `rmdir()` and `chdir()`.

Directories

Any questions so far?

Opening a Directory File for Reading

<table>
<tr><td>Name: <code>opendir()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: opens a directory stream and returns a pointer to it.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <dirent.h> // Defines opendir().</code></td></tr>
<tr><td colspan="3">Syntax: <code>DIR * opendir (const char *path);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>path</code>: a double-quoted name of the folder or path to the folder.</td></tr>
<tr><td rowspan="3">What type it returns: <code>DIR*</code></td><td colspan="2">On success: A non-<code>NULL</code> directory pointer to the opened directory stream.</td></tr>
<tr><td colspan="2">On failure: <code>NULL</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:<pre>DIR *my_folder = opendir ("./cisc3350-hw/");
if (my_folder == NULL)
    /* Error */</pre></td></tr>
<tr><td colspan="3">Other variations: <code>dirfd()</code></td></tr>
</table>

A directory stream consists of an `fd`, some metadata, and a buffer to hold the directory's contents (= file names and their inode numbers.)

Getting a Directory File's `fd`

Name: `dirfd()`	**System call or function:** Function	**Links:** Online Manual | Course Packet
What it does: returns the `fd` of an open directory stream.		

What libraries you must include:
`#define _BSD_SOURCE // Need to use dirfd().`
`#include <dirent.h> // Defines dirfd().`		

Syntax: `int dirfd (DIR *dir);`		
Description of arguments: `dir`: a pointer to an open directory stream.		
What type it returns: `int`	**On success:** The `fd` (file descriptor) of an opened directory stream.	
	On failure: `-1`	
	If failure, does it set `errno`? Yes	

Quick example:
`int fd;`
`if ((fd = dirfd (my_folder)) == -1) // my_folder is a DIR stream.`
`  /* Error */`		

Other variations: `opendir()`		

It is advised to avoid using `dirfd()` since manipulating the directory directly via the `fd`, instead of using the directory stream, could mess up the stream.

Reading a Directory's Contents

<table>
<tr><td>Name: <code>readdir()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: returns information about the next file in the directory with stream <code>dir</code>. You can call <code>readdir()</code> repeatedly to read in all the files in the directory, one at a time.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <dirent.h> // Defines readdir().</code></td></tr>
<tr><td colspan="3">Syntax: <code>struct dirent * readdir (DIR *dir);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>dir</code>: a pointer to an open directory stream.</td></tr>
<tr><td rowspan="3">What type it returns:
<code>struct dirent*</code></td><td colspan="2">On success: A pointer to a <code>dirent</code> structure containing info about a file in the directory (see more on this on the next slide.) <code>NULL</code> is returned if there are no more files in the directory (we read them all.)</td></tr>
<tr><td colspan="2">On failure: <code>NULL</code> and <code>errno</code> is set to a value other than <code>0</code>.</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:

```
struct *dirent = readdir (my_folder);
errno = 0;
if (dirent == NULL && errno != 0)
    /* Error */
```

</td></tr>
<tr><td colspan="3">Other variations: N/A</td></tr>
</table>

Reading a Directory's Contents

The `dirent` structure represents a directory entry and is defined in `<dirent.h>`. Here are its data fields:

```
struct dirent
{
    ino_t          d_ino;       // 1. file's inode number
    off_t          d_off;       // 2. offset to the next dirent
    unsigned short d_reclen;    // 3. length of this record
    unsigned char  d_type;      // 4. file's type
    char           d_name[256]; // 5. file's name
};
```

An application usually calls readdir() repetitively, obtaining each file in the directory, until:

- It finds the file that it was searching for, or
- The entire directory was read, but the file isn't present in this directory.

Closing the Directory Stream

Name: `closedir()`	**System call or function:** Function	**Links:** Online Manual | Course Packet
What it does: closes an open directory stream.		
What libraries you must include: `#include <dirent.h> // Defines closedir().`		
Syntax: `int closedir (DIR *dir);`		
Description of arguments: `dir`: a pointer to an open directory stream.		
What type it returns: `int`	**On success:** `0`	
	On failure: `-1`	
	If failure, does it set `errno`? Yes	
Quick example: `if (closedir (my_folder) == -1) // my_folder is a DIR stream.` `  /* Error */`		
Other variations: N/A		

my_ls.c: Implementation of the $ ls Linux Command


```c
// A program listing the contents of a directory.

// This program is based on Linux System Programming:
//     Talking Directly to the Kernel and C Library,
//     2nd Edition, by Love. ISNB: 978-1-44933953-1,
//     page 270.

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
```

my_ls.c: Implementation of the `$ ls` Linux Command

Any questions so far?

Device Nodes

Device nodes are special files that allow applications to interface with device drivers.

When an application performs I/O on a device node, the kernel passes all such requests to a device driver, which handles the I/O data transfer and returns the results to the user.

This provides the standard mechanism for accessing hardware on UNIX/Linux systems.

To identify the corresponding device driver of a device, each device node is assigned two numerical values: a **major number** and a **minor number**. This pair maps to a specific device driver software that was installed by the OS.

Examples of Special Device Nodes

Here are 3 examples of Linux special devices, which are also a part of Linux's ABI (Application Binary Interface):

- The **null device** (major #1, minor #3): `/dev/null`. The kernel silently discards any writes to the device, and all the reads return the `EOF` character (= `-1`).
- The **zero device** (major #1, minor #5): `/dev/zero`. The kernel drops all writes, but reading from the device always returns null characters (= `'\0'`).
- The **full device** (major #1, minor #7): `/dev/full`. Reads always return null characters (= `'\0'`,) but writes always trigger the `ENOSPC` error (= no space on the device), signifying that the underlying device is full.
- See more Linux devices at https://www.kernel.org/doc/Documentation/admin-guide/devices.txt

These devices have various purposes. For instance, they are useful for testing how an application handles corner and problem cases, e.g., a full filesystem situation. The null and zero devices provide an easy and fast way to discard unwanted I/O.

eject_cd_rom.c: Unmounting a CD-ROM


```
// The following program uses the CDROMEJECT request
//     to eject the media tray from a CDROM device,
//     which the user provides as the first argument
//     on the programs command line.

// This program is taken from Linux System Programming:
//     Talking Directly to the Kernel and C Library,
//     2nd Edition, by Love. ISNB: 978-1-44933953-1,
//     pages 282-283.

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

eject_cd_rom.c: Unmounting a CD-ROM

Any questions so far?

File and Directory Management: Sources

Topic 12: File and Directory Management. Briskman, Miriam. *CISC 3350 OER Course Packet.* 2024, CC BY-NC 4.0.
URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/CISC_3350_OER_Course_Packet.pdf#topic+12:+file+and+directory+management.

Partly based on the teaching of Professor **Yongqing Xiang** of Brooklyn College.