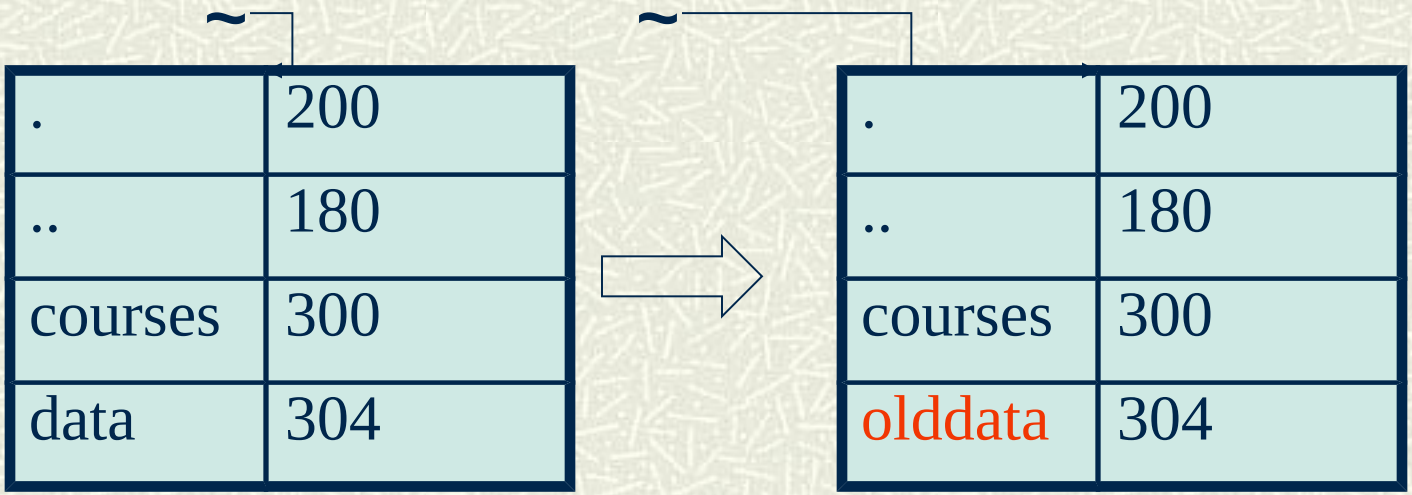
Changing the Contents of Directories

Creation, Deletion, and Renaming of Files and Directories

Changing what is in a directory

- When you ***create***, ***rename*** or ***delete*** a file, whether it is a regular file or a directory, you are really ***changing the contents of its parent directory***. Give it a moment's thought.
- For example, if I change the name of a file named **`data`** in my home directory to **`olddata`**, the only change that takes place is this:



Changes to directories

- The same is true if you create or delete a file. This either adds a new entry to a directory, or deletes an existing entry from a directory.
- This implies that ***in order to create, delete, or rename a file of any kind, you need to have permission to modify the directory in which the changes are to be made***. I.e., you need ***write permission*** on that directory.
- The rest of this lesson is about the details of deletion, copying, and renaming of files and directories. You already know how to create files using **`touch`**.

Creating directories

- UNIX provides a single command to create a new directory: **`mkdir`**. **`mkdir`** (pronounced "make dir") has a list of names of directories to create:

  `mkdir my_newdir1 newdir2 newdir3`

  will create a 3 new directories named **`my_newdir`**, **`newdir2`** and **`newdir3`**.

- Directory names can have almost any character in them, including spaces and newlines, but it is best to avoid using any characters other than letters, digits, and punctuation marks.

More about `mkdir`

- If you type a name but the directory already exists, **`mkdir`** will not replace it. It will warn you instead.
- **`mkdir`** can be given any pathname for an argument, either relative or absolute, and it will create the directory in the place you specify in the pathname. For example,

  ```
  mkdir  /tmp/mydir  ../food
  ```

  will create the directory **`mydir`** in the **`/tmp`** directory and the directory **`food`** in the parent of the current working directory.

Default permissions for directories

- The value of your **umask** determines the permissions of the directories that you create with **mkdir**. To see your **umask**, type the command

  umask -S.

```
$ umask -S
u=rwx,g=rx,o=rx
$ mkdir testdir
$ ls -ld testdir
drwxr-xr-x  2 sweiss cs132  4096 Sep  7 21:15 testdir/
```

- This confirms that **testdir** was created according to the umask.

Removing directories

- The **`rmdir`** command removes ***empty*** directories. (**`rmdir`** is pronounced "remove dir"). ***You can only remove a directory if it is empty and if you have write permission on its parent directory.***

```
rmdir mydir
```

  removes **`mydir`** provided that it is empty, and that you have write permission on the current working directory.

- If a directory contains any entries other than **`.`** and **`..`**, **`rmdir`** will not succeed.

Removing directories (2)

- You can remove many empty directories with a single command:

  ```
  rmdir mydir class/stuff  /tmp/tempdir
  ```

  removes **`mydir`**, **`class/stuff`**, and **`/tmp/tempdir`** provided that all are empty, and you have write permission on **`.`** and **`class`** and **`/tmp`**.
- What about non-empty directories? How do we remove them?
- Soon you will see.

Removing files with `rm`

- `rm` is a very dangerous command in UNIX. Unlike Windows and Macintosh operating systems, UNIX does not use a trash bin concept (***although the GUI built on top of it does.***) So deleting a file is irreversible.
- Therefore, I always use `rm -i` to remove files, which works just like `cp -i` and `mv -i`; the command prompts me and asks if I want to remove the file:

```
$ rm -i oldfile
rm: remove regular file 'oldfile'?
```

to which I can reply '`y`' or '`n`'.

Using `rm`

- The rules for using **`rm`** are simple. You supply it one or more regular file names on the command line, and it removes all of them, if you have permission to remove them.
- You need write permission on the parent directory to remove a file.

```
rm file1 file2 file3 … fileN
```

  removes **`file1`**, **`file2`**, up to **`fileN`**.
- Even though you cannot recover the removed file, forensic techniques can be used to recover its data.

Using `rm` to remove directories

- There is a recursive option to `rm` that will allow you to use it to remove ***non-empty*** directories: the directory to be removed does not have to be empty, which is the requirement of `rmdir`.

```
rm -r mydir
```

  will remove all files recursively from `mydir`, and then remove `mydir`.

- This is, of course, dangerous. If you use `rm -ir` then it will ask you before removing each file, (which might take a long time.)

Copying files and directories: **cp**

- In UNIX, files and directories can be copied easily using the **cp** command (**cp** for copy). It has two different forms. The first is:

  ```
  cp source source_copy
  ```

  in which **source** is an existing file and **source_copy** is the name you wish to give to the new copy of it. In this form, **cp** has just two arguments.
- **source_copy** will have the same size and permissions as **source**, but the time-stamps and ownership will change.

Preserving attributes with **cp**

- If you want to preserve the ownership of the file as well as the times of last modification, last access, and so on, use the **-p** option to **cp** :

```
cp -p source source_copy
```

which will make **source_copy** and preserve ownership and timestamps and permissions.

One danger of **cp**

- Suppose you already have a file named **source_copy** when you type

  cp source source_copy

- The **cp** command will silently replace your old file with a copy of **source**, and you cannot get it back. It's gone. It is safer to type

  cp -i source source_copy

- which will ask you first if you want to replace source_copy:

  cp: overwrite 'source_copy'?

  to which you can type '**y**' or '**n**'.

Copying multiple files into a directory

- Another form of **`cp`** is

  `cp file1 file2 file3 … fileN destdir`

  which, if the first N arguments are regular files (not directories, and the last is an ***existing directory***, will make copies of **`file1`**, **`file2`**, … **`fileN`** and put those copies into **`destdir`** .

Copying all files into a directory (1)

- If you want to copy ***all of the files*** in one directory into another directory, you can do it in one of two ways.
- Suppose **`olddir`** is the directory whose files you want to copy, and that **`destdir`** is an ***existing*** directory into which you want to place the copies. This will do it:

```
cp olddir/* destdir
```

It will not copy hidden files though. If you want to copy the hidden files too, you need to type

```
cp olddir/.?* destdir
```

but you must wait for an explanation of why it works.

Copying a directory tree to another recursively

- The second method uses the **`-r`** option of **`cp`**. The **`-r`** option is the recursive option; it descends the directory tree, copying the entire tree of files rooted in the old directory into the new one.

```
cp -r olddir newdir
```

- If **`newdir`** did not exist before then after, **`newdir`** is an exact copy of **`olddir`**, with everything in it.
- If **`newdir`** existed before, then the directory **`newdir/olddir`** will be created as a copy of **`olddir`**.

Backing up your files

- You should make backup copies of your files regularly. One easy way to do this is with the **`cp`** command. (There are better ways, but they take more time to learn.) Suppose that you have inserted some external medium into a spare USB port on your computer and it is named **`/media/backups`**. You can copy all files of a directory named **`dir`** to a directory with a suitable name on this drive, preserving all information, with the command

  `cp -av dir /media/backups/dir.2014.09.20`

- The **`-a`** option is the same as **`-rp`** in effect, and the **`-v`** means "verbose" -- you will see what **`cp`** is doing as it works.

Suggestions for using `cp`

- I usually give copies names with extensions:

```
cp file1 file1.copy
```

  or

```
cp file1 file1.bak
```

- If you read the man page for **`cp`** you will find more complicated uses of the command.

Moving files using `mv`

- The `mv` command moves files. Moving files is like renaming them; the original name is removed and the file is created with a new name. `mv` has two different forms, which parallel the way `cp` is used.
- In the first form, there are two arguments and both are regular file names:

```
mv origfile newfile
```

renames `origfile`, giving it the name `newfile` instead.

Moving multiple files: Second form of `mv`

- `mv` can be used to move multiple files, using the same syntax as `cp`:

  ```
  mv file1 file2 … fileN destdir
  ```

  moves each of files `file1`, `file2`, …, `fileN` to the directory `destdir`.
- Unlike `cp`, the arguments to `mv` can be directories, in which case they and the files within them are moved together to the new location.

Moving files to other places

- You can use **`mv`** to move a single file to a different directory. ***This is a special case of the second form, in which the last argument is a directory name***:

```
mv hwk3 ~/cs132/homework
```

will move **`hwk3`** to the directory **`~/cs132/homework`**. If that directory already has a file named **`hwk3`**, it will be replaced silently. You could use the first form

```
mv hwk3 ~/cs132/homework/hwk3_v2
```

to give it a new name in that directory instead.

Warning

- If you type

  ```
  mv olddir destdir
  ```

  where **olddir** and **destdir** are existing directories, it means, "**mv** the directory **olddir** *into* the existing directory **destdir**." When **mv** finishes, **destdir** will contain **olddir**:

  ```
  $ ls destdir
  olddir
  ```

- But if **destdir** did not exist before, it just means "rename **olddir** with the new name **destdir**."

Dangers of **mv**

- Just like **cp**, **mv** will silently overwrite destination files if they exist already.
- If **dest** is a file that already exists, and you type

```
mv oldfile dest
```

then **dest** will be completely eradicated with no hope of retrieval. It is safer to use **mv -i**, to prompt you just in case an overwrite would happen.

```
$ mv -i oldfile dest
cp: overwrite 'dest'?
```

Shell aliases

- An ***alias*** is another name for a command. All shells let you create aliases. The reason to do this is either to make shorter versions of long commands, or to prevent you from making careless mistakes.
- In **`bash`** you create an alias using the syntax

  `alias name=`*`command`*

  with NO SPACE around the '**`=`**'. For example

  `alias m='more'`

  makes **`m`** a shorthand for the **`more`** command.

More about aliases

- If the command you want to alias has white space in it, then enclose it in single quotes:

  ```
  alias remove='rm -i'
  ```

  makes **`remove`** a shorthand for **`rm -i`**. You can replace the **`rm`** command itself by making **`rm`** an alias for **`rm -i`**

  ```
  alias rm='rm -i'
  ```

- This alias is a good one to have in your **`.bashrc`** file. Take a look at the **`.bashrc`** file posted in the course home directory on **`eniac`** and see what aliases you have.

Things to try

- Try to create your own directory in the root directory.
- Look at the permissions of various directories. Go snooping.
- Create a temporary scratch directory in your home directory and populate it with a bunch of files and directories. See how **`mv`** works inside this directory by exploring its options.
- Create some useful aliases for yourself.
- Change your **`umask`** and see what happens when you create directories and files.