Permissions and Links

- The `root` account
- Setuid and Setgid Permissions
- Setting Setuid and Setgid with `chmod`
- Directory Access Permissions
- Links
  - Two Types of Links
  - The `ln` command
  - Removing a link

The root Account

- On every Unix or Linux system, there is a special account named **root**
- This account is sometimes referred to as the **superuser**
- *root* can:
  - access any file
  - or run any program
- *root* is an administrator account
- It is used for system configuration and maintenance

The root Account

- Even a system administrator should not log in as *root*
- Instead, he or she should use a *regular* Unix account -- and should switch to the *root* account ***only*** when they need its power
- This can be done with the ***su*** (**s**witch **u**ser) command
- To become the *root* user, you would enter the following on the command line

```
su -
```

- This will only work if you know the *root* password

The root Account

- A better way of doing this is to use the ***sudo*** command
- ***sudo*** stands for **s**uper**u**ser **do**
- You type ***sudo*** and then the command that only the *root* user can run
- You use it like this

```
sudo COMMAND_ONLY_ROOT_CAN_RUN
```

- You will then be prompted for **your** password – not the root password

The root Account

- This will only work if the system administrator has added you to the **sudo-ers** list
- ***sudo*** is safer than using ***su*** because the person using it does not have to know the *root* password
- If all administrators knew the *root* password, then you would have to change it *every time* one of them left
- If they all use ***sudo***, all you would have to do when they left would be to delete their entry in the sudo-ers list

Setuid and Setgid Permissions

- Sometimes, a program needs to *read* or *modify* a file to do the work it was designed to do
- For example, the ***passwd*** command – which is used to change the password for an account – has to make changes to the file **/etc/shadow**
- When you need to change *your* password, you run ***passwd***
- But, **/etc/shadow** is owned by the **root account**, so no other account can change it
- To deal with situations like this, two special permissions were created
  - **setuid**
  - **setgid**

Setuid and Setgid Permissions

- If a file has **setuid permission**, then anyone who runs the file has all the permissions of the owner of that file -- but ***only*** while the file is running
- This permission means it can change files – that the script or program needs to do its job – that ordinary users cannot change
- If a file has **setgid permission** set, then anyone who runs the file has the permissions of the group assigned to that file – *while* the file is running
- **`setuid`** and **`setgid`** permissions only apply to **executable** files – that is, programs and scripts

Setuid and Setgid Permissions

- A file with **`setuid`** permission will have **`s`** in place of **`x`** in the column for the *owner's* execute permission
- A file with **`setgid`** permission will have **`s`** in place of **`x`** for the *group* execute permission
- Since **`setuid`** and **`setgid`** permission apply ***only*** to executable files, there is no ambiguity in replacing **`x`** with **`s`**
- For example, consider the ***`passwd`*** command…

Setuid and Setgid Permissions

- The executable file for this utility is **/usr/bin/passwd**
- Running **ls -l** on this file we get

```
ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 42824 2011-02-20 19:18
                                        /usr/bin/passwd
```

- Notice the **s** in the place where the owner's execute permission should be

Setuid and Setgid Permissions

- The **passwd** command needs to modify **/etc/shadow** — which is a file that stores the encrypted passwords
- Only `root` can change this file

```
ls -l /etc/shadow
-rw-r--r-- 1 root shadow 926 Jul 16  2013
                                    /etc/shadow
```

- But you need to change this file for your entry only when you change your password
- The **setuid** permission allows you to run ***passwd*** and modify the file, but ***passwd*** knows who you really are and it will only let you modify your own entry not the entry for any other account

Setting Setuid and Setgid with *chmod*

- You assign **setuid** or **setgid** permissions to a file with *chmod* and an extra digit
- If I wanted to assign **755** permission to a script along with **setuid** permission, I would use **4755**
- When 4 is the first digit in a series of four digits used with *chmod*, **setuid** permission is assigned to the file

```
$ chmod 4755 work.sh

$ ls -l work.sh
-rwsr-xr-x 1 ghoffmn grad 0 Mar  3 12:30 work.sh
```

- To assign the **setgid** permission I would use 2 instead

```
$ chmod 2755 work.sh

$ ls -l work.sh
-rwxr-sr-x 1 ghoffmn grad 0 Mar  3 12:30 work.sh
```

Directory Access Permissions

- Unix permissions work a little differently for directories
- *Read* and *write* permissions for a directory are similar to those for a file
- ***Read*** permission on a directory allows you to list the **contents** using ***ls***
- This read permission **only allows you to use *ls***
  - To read the *files* in the directory, you need read permission for each file
  - So read permission on a directory **does not** allow you to read the files in that directory
  - You need read permission on the **file** to do that

Directory Access Permissions

- ***Write*** permission on a directory allows you to create, delete, or change the *name* of any files **in that directory**
- But, you cannot change the files themselves – unless you have write permission on those *files*, too
- Note that write permission on a directory only applies to the **contents** of a directory – **not** to the directory itself
- You cannot change the name of a directory or delete it…unless you have write permission on its **parent** directory

Directory Access Permissions

- ***Execute*** permission on a directory is ***very*** different from execute permission on a file
- You can't "run" a directory from the command line, like you would a program or script file
- Execute permission on a directory allows you to do two things
  - *Enter* the directory using **`cd`**
  - Read a file in the directory for which you have read privileges or write to the file if you have write permission
- If you have read permission for the file but *not* for the directory in which it is located, then you can only read the contents if you know the name of the file
- That's because you need read access to the *directory* to run **`ls`** on it

Links

- Sometimes, it is convenient to have *more than one way* of getting to a file or directory
- Windows, for example has *shortcuts*
- You can create a shortcut anywhere, and clicking on it will take you to the *real* file somewhere else on disk
- This allows the user another way of getting to the file
- Instead of going to the directory that holds the file, you can click a shortcut in some other directory

Links

- In Unix, these pointer files are called **links**
- Links can be very useful when moving to a directory that is far away from your current directory
- If your current directory has a link to another file or directory, then you can use the link to access that file or directory
- This saves you the bother of using a long absolute or relative pathname

Links

- For example, each of you has an entry in your home directory called **it244**

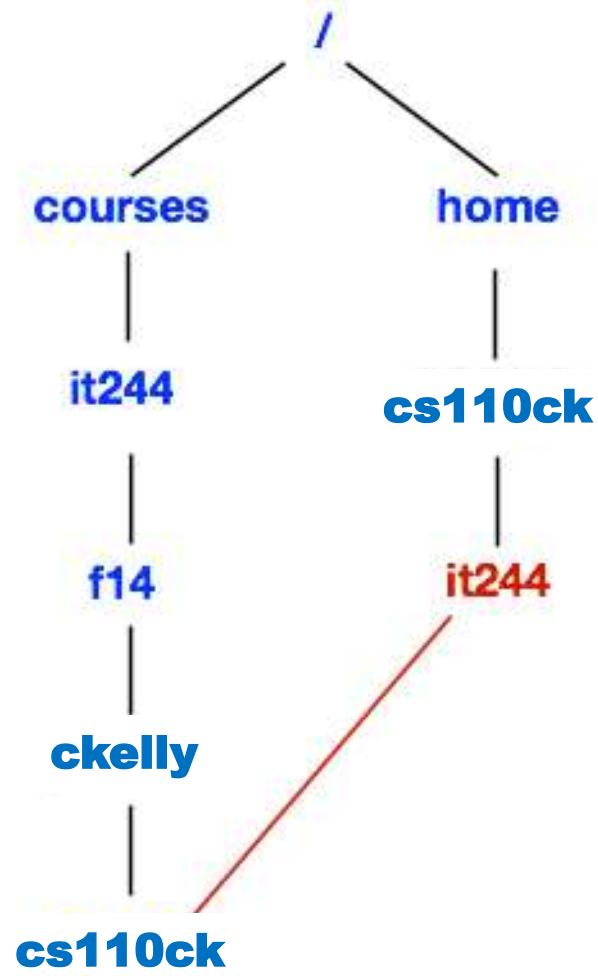
```
$ ls -l it244
lrwxrwxrwx 1 it244gh faculty  34 Jan 29 10:39 it244 ->
                                /courses/it244/sum14/it244gh
```

- Notice the **l** (i.e., the lowercase of "L"), the first character in the permissions string
- This tells you that you have a ***link*** named **it244**, not a directory
- The real directory is **cs110ck** in **/courses/it244/f16/ckelly**
- But, you can use the link *just as if* it were a directory

Links

- In the home directory of my test account `cs110ck` is a **link** to a directory for this account in the course directory

`/courses/it244/f16/cs110ck`



Links

- If you ***cd*** to the link, you will go to the real directory
- If you ***cd*** into this location and use ***pwd*** …

```
$ pwd
/home/cs110ck

$ cd it244

$ pwd
/home/cs110ck/it244
```

- …the path that ***pwd*** prints is *the route you took* to get to the current directory
- But… it is **not** the *real* path to the directory

Links

- You can only get the true location if you use ***pwd*** with the **-P** option

```
$ pwd
/home/cs110ck/it244

$ pwd -P
/courses/it244/f16/ckelly/cs110ck
```

- You must use a capital P, **not** a lowercase p
- Unix tries to hide your real location when you use a link so as not to confuse you

Links

- In this case, we used a link named **`it244`** inside the home directory of my it244gh account to get to **`/courses/it244/f16/ckelly/cs110ck`**
- I can get back to where I came from using **`..`**

```
$ pwd
/home/cs110ck/it244

$ cd ..

$ pwd
/home/cs110ck
```

- Why does your home directory have a link to your **`it244`** class directory?
- To make things *easier* for instructors like me

Links

- If the directory in which you do your course work were in your home directory I would have to go to many places to collect your assignments
- Instead, I only have to go one place, and if I run ***ls***, I see each of your course directories

```
$ ls
alexgri    fatalaty   hw         meteos     nle        sanf5456
skhalifa   ychen123   cdelaney   GROUP      kiwan      neko92
rangeley   sfarah     sukhi515   cs110ck    hsingh     MAIL
neoalx     rolon      sindel     wenwu10
```

- The **it244** link in your home directory points to your course directory in **/courses/it244/f16/ckelly**
- The name of your course directory is your Unix username

The Two Types of Links

- There are two types of links
  - Hard links
  - Symbolic, or soft, links
- **Hard links** are older
- But ,they are seldom used these days
- A hard link is like a *duplicate* file name
  - If you have a hard link to a file, and the original filename is deleted, then the file will *still* be there
  - The file will remain until the ***last*** hard link is removed

The Two Types of Links

- Hard links have some disadvantages
- Hard links can only point to files, **not directories**
- Our Unix filesystem *appears* to be a single hierarchy
  - In reality, it is a collection of
    - different file systems...
    - ...on different hard disc volumes
  - The different file systems are stitched together so that they ***look*** like a single system
  - Unix hides this fact from users

The Two Types of Links

- But, this causes problems for hard links
- You can only have a hard link to a file in the same volume as the link you are creating
- That means you can't link to a file on a *different* disk or partition
- **Symbolic links** are much more flexible
  - Symbolic links are sometimes called “soft” links
  - You can use either an absolute or relative pathname when creating a symbolic link

The Two Types of Links

- A symbolic link can point to a file or directory on ***any*** disk or partition
- Deleting a soft link *does not* delete the file or directory it points to
- You can delete a file or directory that has a soft link pointing to it *without* deleting the link
- The symbolic link remains, but it points to *nothing*
- Use ***ln*** to create a link

ln

- To create a symbolic (or soft) link, use ***ln*** with the **-s** option
  - Otherwise, you will create a hard link
  - That is *not* what you want
- ***ln*** takes ***two*** arguments

```
ln  -s  TARGET_PATHNAME  LINK_NAME
```

- For example…

```
$ pwd
/home/cs110ck

$ ln  -s  ~ckelly/course_files/it244_files  examples

$ ls -l examples
lrwxrwxrwx 1 it244gh libuuid 28 2012-09-17 17:53 examples ->
                                    /home/ghoffmn/examples_it244
```

Removing a Link

- To delete a link, use ***rm***
- This will work whether the link points to a file ***or*** a directory
- If you delete a symbolic link, it will not affect the file or directory it points to
- If you delete a hard link, you will not delete the file
  - Unless, of course, the link is the ***last*** connection to the file – so be careful!