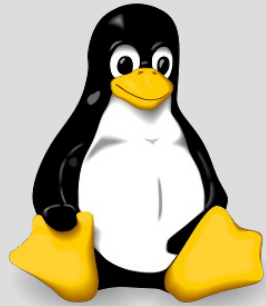
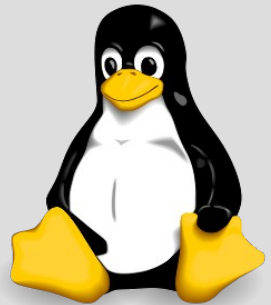


- Log into the Moodle site
- Enter the “Lecture 5” area
- At 14:00, choose “Daily Quiz 4”
- Answer the multiple choice quiz
(you have 10 min to finish)



Removing files and directories, copying and moving files, links, controlling programs



J. M. P. Alves

Laboratory of Genomics & Bioinformatics in Parasitology

Department of Parasitology, ICB, USP

Copying files and directories

- Sometimes you need a copy of **a file or directory**, say, to edit or experiment on, without modifying the original
- You can use **cp** to **copy** files and directories
- **cp** takes **at least** two arguments: what is being copied and where it is being copied to. For example:

```
cp file_1 file_2
```

- This command will get **file_1** and create an exact copy but with a different name, **file_2**. **Try (in the remote server):**

```
cp dummy_file dummy_file2
```



Copying files and directories

- If the last argument to **cp** is a **directory**, the file(s) given will be copied to the directory (**keeping the old names**):

```
cp file1 fileA file~ dir1
```

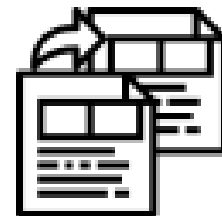
- Now, directory **dir1** will contain copies of those three files
- Very useful options for **cp** are **-u**, **-p**, and **-i**
- **-u** is for **update**: only copies files that are **missing or are newer** than their destination (saves time when there are lots of files!)
- **-i** is for **interactive**: always ask for confirmation before overwriting an existing file (by default, **cp** **quietly overwrites**!) – different from **-n** (**--no-clobber**), which simply does not overwrite, and never asks!
- **-p** **preserves** file attributes (ownership, timestamps, permissions) as they are in the original file, as much as possible

Copying files and directories

- Now, being in the remote server and inside directory **PE11**, try:

```
cp manifest_2007 2007-01
```

- Copy file **manifest_2008** to directory **2008-01**, but **this time, preserve timestamps**
- List (or **stat**) the **manifest_2007** file from **PE11** and the one from **PE11/2007-01** – how do they compare? What is the same and what is not?
- Same comparison for the **manifest_2008** files... What happened now?



Copying files and directories

- How about directories? Let's try (inside **PE11** as usual):

```
cp 2007-01 2007-02
```

- It did not work, did it? You got:

```
cp: omitting directory '2007-01'
```

- By default, **cp** does not copy directories!
- What can we do now?



Quiz time!



Go to the course page and choose **Quiz 10**

Copying files and directories

- How about directories? Let's try (inside **PE11** as usual):

```
cp 2007-01 2007-02
```

- It did not work, did it? You got:

```
cp: omitting directory '2007-01'
```

- By default, **cp** does not copy directories!
- What can we do now?
- To **also copy directories**, we need to give **cp** the **-r** (for recursive) option:

```
cp -r 2007-01 2007-02
```



Moving files and directories

- If instead of copying you actually want to **move files or directories**, then you need the **mv** command, which we have already played with before:



```
mv file1 dir1
```

- Now, directory **dir1** will contain **file1**, which will **NOT** be in the current directory anymore
- **mv** behaves very similarly to **cp**, and many of the options are **the same**
- Very useful options for **mv** are **-u** and **-i** (no need for **-p** now!)
- **-u** is for **update**: only moves files that are **missing or are newer** than their destination (saves time when there are lots of files!)
- **-i** is for **interactive**: always ask for confirmation before overwriting an existing file (by default, **mv** quietly overwrites!) – different from **-n** (**--no-clobber**), which simply does not overwrite, and never asks!

Moving files and directories

- If the last argument to **mv** is a directory, the file(s) given will be moved to the directory (**keeping the old names**):



```
mv file1 dirA file~ dir1
```

- Now, directory **dir1** will contain those two files and one directory, which will disappear from the current working directory
- As happens with **cp**, if there are more than two arguments to **mv** and the last one is **not** a directory, you will get **an error**
- There is no need for a **-p** option like to one that **cp** takes because the file/directory being moved is still the same, just located in a different place and/or with a different name, so no change in attributes will take place

Moving/renaming files and directories

- **mv** also takes **at least** two arguments: what is being moved and where it is being moved to, just like **cp**, but...

```
mv regular_file_1 regular_file_2
```

- This command will get **regular_file_1** and change its name to **regular_file_2**; same behavior for directories if the second one does not exist
- **Try** (in the remote server):

```
mv dummy_file smart_file
```

- There is a much more **flexible and powerful** renaming program that is frequently included with Linux, but to be able to use it we need to know the basics of **regular expressions**

Safety tip

Whenever you are doing something that modifies (or deletes) files or directories, specially if it involves complicated shell expansions, TEST it first using `ls` or `echo` to make sure only the intended files will be affected!



Now you do it!

Go to the course site and enter **Practical Exercise 12**

Follow the instructions to answer the questions in the exercise

Remember: in a PE, you should do things in practice before answering the question!



The \$PATH

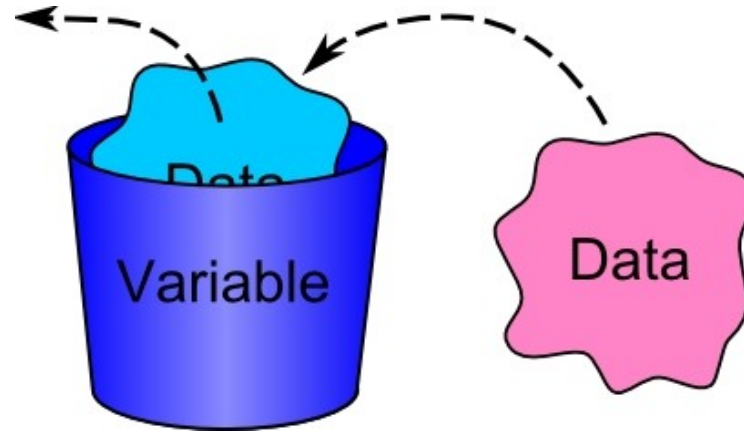
PATH (which is written with **all upper case letters**) should **not be confused** with the term **path** (lower case letters), which we saw earlier

PATH is an **environmental variable** in Linux and other Unix-like operating systems that tells the shell **which directories to search for executable files (i.e., ready-to-run programs)** when the user tries to run a command; run **echo \$PATH** to see what this variable contains in your machine

By the way:

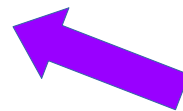
Environmental variables are a class of variables (i.e., items whose values can be changed) that tell the shell how to behave as the user works at the command line or with shell scripts (the short programs written in the shell programming language). You can use the **env** command to see all environmental variables.

First of all... what's a variable?



- Variable: A piece of memory which has **a name** and might contain some information (the variable's **value**) – think of it as a labeled bucket for your data
- You have all seen variables before, in a **mathematical** context...

$$y = 3x + 10 - z$$



x, y, and z are variables

The environment



- During the shell session, Bash maintains a **body of information** called the **environment**
- Data stored in the environment can be **used by programs** to determine things about system configuration, e.g., where to look for executables (the **\$PATH** variable we have seen earlier), language settings, locale, etc.
- The environment contains **variables**, **aliases**, and **functions**

Other important environmental variables

- Besides **\$PATH**, there are other useful and important environment variables that appear a lot and are specially useful for **automating the shell** using scripts
- A variable can be printed with **echo**; to see all, use **printenv** or **env**
 - **\$USER** stores the current user's name
 - **\$HOME** stores the current user's home directory
 - **\$HISTSIZE** stores the number of commands to keep in history
 - **\$SHELL** stores the path to the shell's executable
 - **\$BASH_VERSION** stores the version of the Bash program itself
- You can create your own environment variables too!

But... how do you run a program that is not in the \$PATH?

- Let's say you downloaded **a new program**, but you have no permission to place it in any of the directories in your **\$PATH**
- This is specially important given that **specialized analyses** will use many programs that **did not ship** with the system!
- You have different options to solve this:
 - 1) **Change** your **\$PATH** to include one of your own directories
 - 2) Always run the program from **within the directory** where it is (using **./** before the command's name)
 - 3) Always **give the path (relative or full)** to the program

Let's do it!

Enter the virtual machine:

Address: **200.144.244.172**

- After you log in, download the following file with **wget**:

wget http://lgbp.online/average

(<https://sourceforge.net/projects/average/>)

Let's do it!

- This little program allows you to do some basic statistics directly on the command-line interface; it can give you some **help** with the **-h** option
- Let's first try the simplest option, running the program directly where it is located:

./average -h

- **What happened? Why?**
- You do not have execute permission! (remember: running a program is “executing” its instructions)
- Fix it like we learned earlier, then try again...

Let's do it!

- Now, try giving the **full path** (remember: starting from /) to the program
- Did it work this time?
- How about the **relative path**?
- Remember that the format below:

./average -h

- ...is **already a relative path!**

Changing the \$PATH

- As mentioned above, you can **add more directories** to the **\$PATH** variable
- **Traditionally**, directories containing programs have their final part as **bin** (but actually the name can be anything you want!)
- First, create a directory called simply **bin**, in your **home directory**:

```
mkdir ~/bin
```

or

```
mkdir $HOME/bin
```

- Remember that you **do not** need to put the **\$HOME/** or **~/** parts if you are already working from your home directory!

Changing the \$PATH

- Depending on the Linux distribution, the **\$PATH** variable **already** contains **\$HOME/bin** (in which case you are done, no need to change anything else!)
- For those systems where that is not the case, you have to either change the contents of the variable **or** log out and back in again
- First, use **echo** to look at the current contents:

```
echo $PATH
```

- Each directory in the list is separated from the next by a colon (:)
- The **export** command can be used to create new variables, or change existing ones (actually, recreating), and place them in the environment:

```
export PATH=$PATH:~/bin
```

Quiz time!



Go to the course page and choose **Quiz 11**

Last step!

- You now have a directory (**\$HOME/bin**) for your program files to live in
- And that directory is in the **\$PATH**, therefore it can be seen (**by you!**) automatically from anywhere in the system
- Now, use **mv** to move **average** to **\$HOME/bin** by running:

```
mv average ~/bin
```

- If you did everything correctly, pressing **TAB** should now autocomplete to **average** after you type just **av**

Setting an environmental variable

- When running **average**, we might have seen some ugly warnings... (it depends on which system you are connecting from)

```
dummy@JLabVM:~$ average -h
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
    LANGUAGE = "en_US:en",
    LC_ALL = (unset),
    LC_ADDRESS = "pt_BR.UTF-8",
    LC_NAME = "pt_BR.UTF-8",
    LC_MONETARY = "pt_BR.UTF-8",
    LC_PAPER = "pt_BR.UTF-8",
    LC_IDENTIFICATION = "pt_BR.UTF-8",
    LC_TELEPHONE = "pt_BR.UTF-8",
    LC_MEASUREMENT = "pt_BR.UTF-8",
    LC_CTYPE = "pt_BR.UTF-8",
    LC_TIME = "pt_BR.UTF-8",
    LC_NUMERIC = "pt_BR.UTF-8",
    LANG = "en_US.UTF-8"
    are supported and installed on your system.
perl: warning: Falling back to a fallback locale ("en_US.UTF-8").
Version: 0.7.2
```

average

Setting an environmental variable

- When running **average**, we might have seen some ugly warnings... (it depends on which system you are connecting from)
- If you do some research online, you will see that the (harmless, but annoying) problem is caused by your **local system** setting some language-related variables on the remote computer
- **\$LC_ALL** is the variable we want to change here
- First, to see what is in the variable right now, try:


```
echo $LC_ALL
```

Setting an environmental variable

- So, actually, you are **creating a new variable** (which happens to have a predefined meaning)
- Give **LC_ALL** the value **C** (capital c) or **en_US.UTF-8** (which are two locales the remote system does know)
- Now, try running **average -h** again and see if the warnings are gone
- They should be...



Links

- Regular files (-)
 - Directories (d)
 - Links (l)
- 
- Character device file (c)
 - Block device file (b)
 - Named pipe (p)
 - Local socket file (s)

99% of your time as a regular user, you only work with these three kinds of file

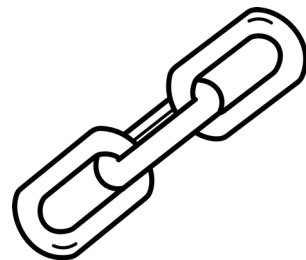
drwxr-xr-x	150	root	root
drwxr-xr-x	4	root	root
lrwxrwxrwx	1	root	root



See <https://linuxconfig.org/identifying-file-types-in-linux> for more details

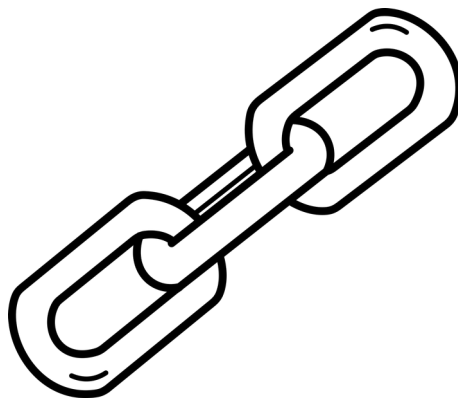
Links

- Remember **aliases**, different ways to refer to a command?
- Links are conceptually like aliases, but **for files or directories** instead of commands
- That is, links are alternative ways to point to a file or directory
- There are **two kinds** of links:
 - **symbolic** links (the most commonly used)
 - **hard** links



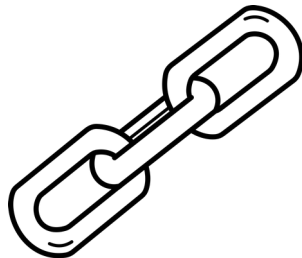
Links

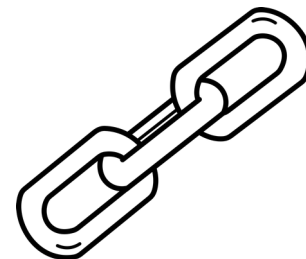
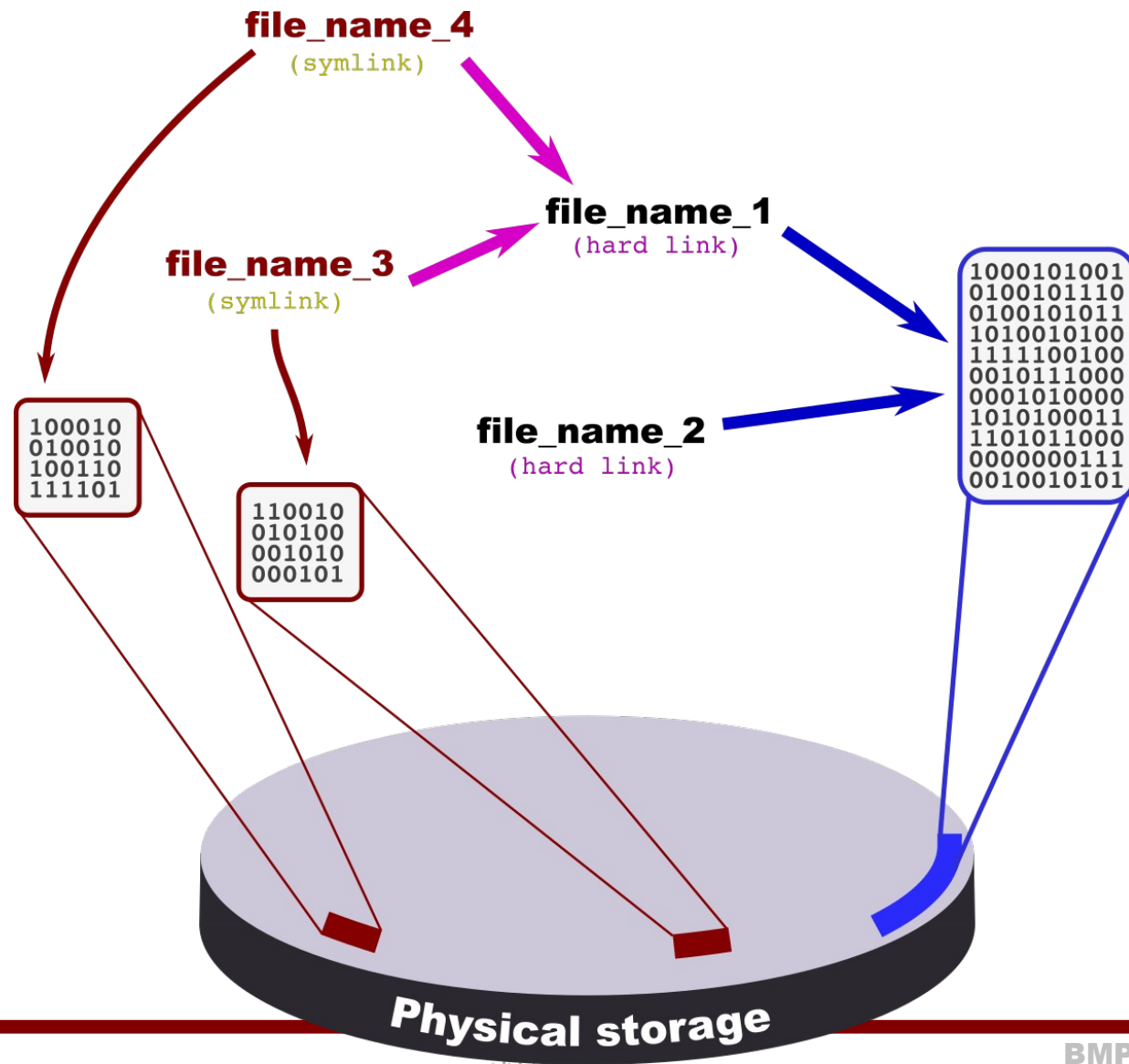
- **Hard** links are **different names for the same file**
- Hard links can **only** be made for **regular files**, **not** for **directories**!
- **Symbolic** links are **small files** that simply **point to another** file or directory:
the symlink contains a reference to another in the form of an **absolute** or **relative** path



Links

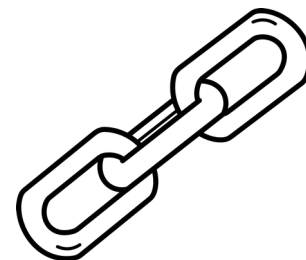
- The different **hard** links to a file have **no hierarchy** – they are all equivalent and any of them can be deleted without affecting the others
- Symbolic links are **only valid as long as the original file or directory to which they point still exists** (if the original is deleted, any remaining links are said to be **“broken”**)
- Another difference between hard and symbolic links is that **only symbolic** ones can point to a file or directory located in a **different disk or partition** from the one where the link is





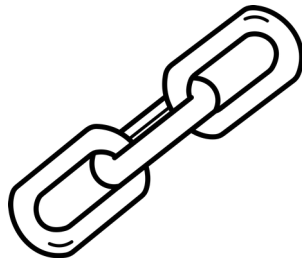
Links

- The command that makes links is **ln** (for **link**)
- The command takes **two arguments**: what you want to link, and where you want it to be linked (the link can have a different name from the original file!)
- If the second argument is missing, it is assumed to be **.** (current directory)
- The **default** for **ln** is to create **hard** links



Links

- Symbolic links are specified with the **-s** option
- If a symbolic link is created using a non-existing file as source (or if the original is deleted after link creation), the link is said to be **“broken”** or **“dangling”**
- Mostly, you can **treat the symlink as the original file** (including **permissions!**): if you do something to it, it happens to the original. The most significant exception is deleting! (only removes the link, **not** the original)



Let's make some links

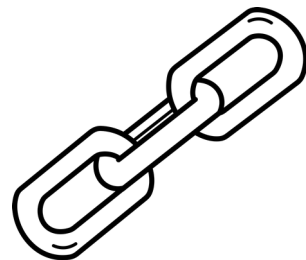
- Log into the **remote server**, in case you are not there already
- From **your home** directory, run:

```
ln -s /etc/passwd my_first_link
```

- Now list (long format) the contents of your home directory
- The system file **/etc/passwd** is now accessible using your own **my_first_link** file
- Try seeing the content of **my_first_link**:

```
more my_first_link
```

- What you see is actually the contents of **/etc/passwd**



Let's make some links

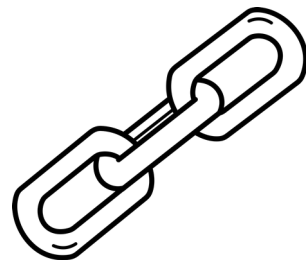
- Now, try the following:

```
ln -s /etc/blah my_second_link
```

- Now list (long format) the contents of your home directory
- Try seeing the content of `my_second_link`:

```
more my_second_link
```

- What do you see?
- The link `my_second_link` is **broken** or **dangling**



Let's make some links

- Now try a hard link:

```
ln /etc/passwd hard_link_1
```

- Usually, you can only create **hard** links to files you **own** or have **rw** permissions to
- Then, try:

```
ln /data/shaw hard_link_1
```

- /data/shaw and your home area are on **different disks**

```
ln: failed to create hard link 'hard_link_1' => '/data/shaw':  
Invalid cross-device link
```

Links

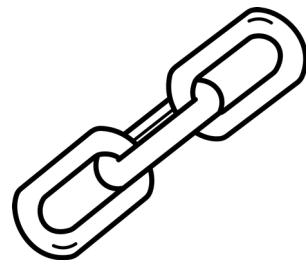
- **Why** make links?
- Links can be very useful as **shortcuts** to files that have **long names** and are **frequently needed**...
- For example:
`/usr/local/genome/store/ncbi-blast-2.4.0+/doc/TEXT`
- If you use that file all the time, typing all that will get tiring... So you can just have a short symlink in your home and access **that** instead:

```
ln -s /usr/local/genome/store/ncbi-blast-2.4.0+/doc/TEXT ~/fileX
```

Links

- Another important reason to have links: **avoid copying files**, specially **large** ones, without need
- For example, let's say you frequently read a **large, multi-GB** data file for your analyses. So, instead of wasting many GB of disk space with copies of that file, you can just create links
- Finally, links can be very useful to give **alternative names** for the same file
- This is quite common in system directories
- In the remote server, run:

```
ls -l /lib/lib*
```



Links

- You will see some library files that are regular files, and others that are simply links to other library files:



```
lrwxrwxrwx 1 root root      17 Nov  8 2014 /lib/libip4tc.so.0 -> libip4tc.so.0.1.0
-rw-r--r-- 1 root root 31416 Nov  8 2014 /lib/libip4tc.so.0.1.0
lrwxrwxrwx 1 root root      17 Nov  8 2014 /lib/libip6tc.so.0 -> libip6tc.so.0.1.0
-rw-r--r-- 1 root root 31416 Nov  8 2014 /lib/libip6tc.so.0.1.0
lrwxrwxrwx 1 root root      15 Nov  8 2014 /lib/libipq.so.0 -> libipq.so.0.0.0
-rw-r--r-- 1 root root 10544 Nov  8 2014 /lib/libipq.so.0.0.0
lrwxrwxrwx 1 root root      16 Nov  8 2014 /lib/libiptc.so.0 -> libiptc.so.0.0.0
-rw-r--r-- 1 root root  5816 Nov  8 2014 /lib/libiptc.so.0.0.0
```

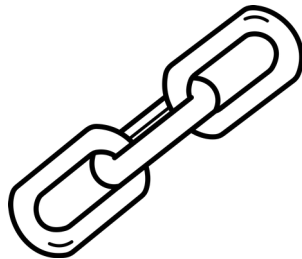
Links

- Links can be made using the **full** or the **relative** path to the source – each has its advantages, depending on the situation!
- Let's make a couple of links on the remote server
- First, create a directory called `~/link_tests` and enter it
- Then, run the following command:

```
ln -s ../../dummy/PE11/2007/manifest_2007
```

- Now run:

```
ln -s ~dummy/PE11/2008/manifest_2008
```



Quiz time!




Go to the course page and choose **Quiz 12**

Links

- Links can be investigated using **ls -l** or **stat**
- The number of hard links appears automatically for **ls -l**. For example:

```
drwxrwxr-x  2 root    root    4096 Apr  6  2016 bin
-rw-r--r--  2 dummy   dummy   4929 Mar 27 15:34 hard_link
-r--r--r--  1 jmalves jmalves    0 Mar 22 16:22 dummy_file
drwxr-xr-x 12 jmalves jmalves  4096 Mar 28 22:07 PE11
```

 number of hard
links to data

- Running **stat** on file **hard_link**, for example:

```
File: 'hard_link'
Size: 4929          Blocks: 16          IO Block: 4096   regular file
Device: ca01h/51713d Inode: 1045861     Links: 2
Access: (0644/-rw-r--r--)  Uid: ( 1015/   dummy)   Gid: ( 1017/   dummy)
Access: 2017-03-28 14:27:08.113629484 -0300
Modify: 2017-03-27 15:34:34.000000000 -0300
Change: 2017-03-29 13:04:34.011087151 -0300
Birth: -
```

Now you do it!

Go to the course site and enter **Practical Exercise 13**

Follow the instructions to answer the questions in the exercise

Remember: in a PE, you should do things in practice before answering the question!

