The WTCHG Research Computing Core

Talk 2: A basic introduction to Linux

Robert Esnouf (robert@strubi.ox.ac.uk; robert@well.ox.ac.uk),
Jon Diprose (jon@well.ox.ac.uk) & Colin Freeman (cfreeman@well.ox.ac.uk)

Generic emails: support rescomp@well.ox.ac.uk
users rescomp-users@well.ox.ac.uk

A series of introductory talks…

A set of six talks of about 45 minutes each (plus questions)

- Talk 1: What is the ResComp Core?
  (Mon 23/1 10:00 Room B; Robert Esnouf)
- Talk 2: A basic introduction to Linux
  (Wed 25/1 10:00 Room B; Robert Esnouf)
- Talk 3: Submitting jobs to the cluster
  (Thu 26/1 10:00 Room B; Robert Esnouf)
- Talk 4: Monitoring and troubleshooting
  (Mon 30/1 11:00 Room B; Robert Esnouf)
- Talk 5: ResComp centrally-managed applications
  (Wed 1/2 10:00 Room B; Jon Diprose)
- Talk 6: Doing your own thing (compiling and customizing)
  (Thu 2/2 10:00 Room B; Jon Diprose)

The story so far…

The ResComp Core supports genomics, statistical genetics and electron microscopy computing at scale

- High-memory, high-data volume, lots of I/O, real HPC
- Want cluster to be usable by non-HPC specialists

Cluster Computing

- A nodes: 288 cores @ 2.67GB/core (32GB/node; 70% C speed)
- B nodes: 640 cores @ 8GB/core (96GB/node; 70% C speed)
- C nodes: 1728 cores @ 16GB/core (256GB/node; 100% C speed)
- D nodes: 768 cores @ 16GB/core (384GB/node; 100% C speed)
- H nodes: 96 cores @ 42.66GB/core (2TB/node; 115% C speed)

Login nodes: rescomp1 and rescomp2

- General use: compiling programs, submitting jobs, analysing results

Project servers

- Like rescomp servers, but access restricted by group

The story so far…

High-performance storage (GPFS)

- /gpfs0: 6.7 TB @ 400MB/s do not use for data
- /users/<*group*>/<*user*>, /apps/well, /mgmt
- /gpfs1: 1291 TB @ >8GB/s paid quota
- /gpfs2: 1784 TB @ >12GB/s paid quota
- /gpfs1 and /gpfs2 presented as /well/<*group*>
- Charging rate £35 per usable TB per 6 months

"Archive" Storage (XFS)

- /arc1[a-f], /arc2[a-f], /arc3[a-g] 1417TB total paid quota
- Mounted read-only in specific places
- Not visible across the cluster
- Charging rate £14.16 per usable TB per 6 months

Overview of this talk…

- A Unix/Linux account
- Shells, profiles, environments and commands
- Filesystems, directories and pathnames
- Unix file permissions
- File streams, redirection and pipes
- Foreground and background jobs
- Getting started with shell scripts

… *i.e.* just the concepts you need to be ready to use the cluster

A Unix/Linux account

Elements of a Unix account

- a username (and an associated UID)
- a primary group (and an associated GID)
- optional membership of secondary groups (other GIDs)

You log in with your username and password, (e.g. using an ssh client)

- ssh robert@rescomp1.well.ox.ac.uk

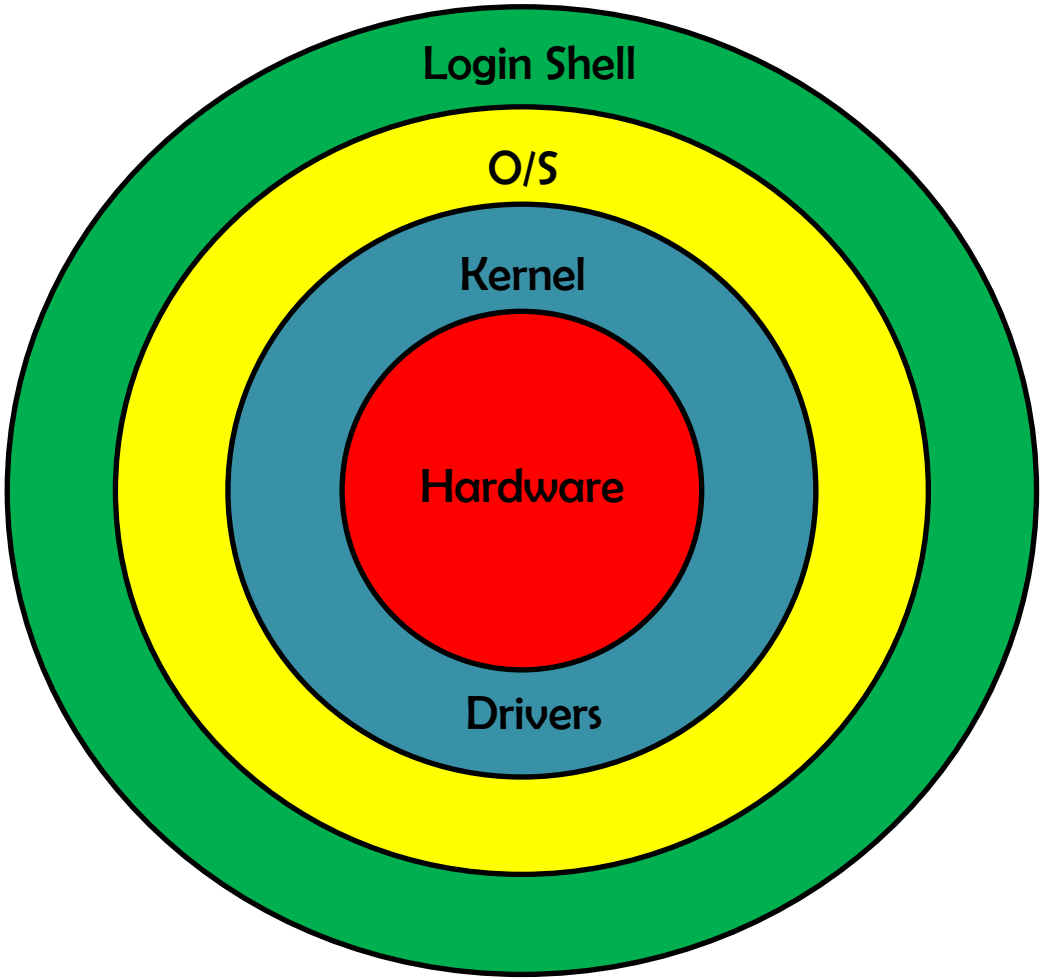
Your UID and GIDs determine what files you can see and what you can do as far as the operating system is concerned

Checking your account:

- getent passwd <*username*>
- ypcat passwd <*username*> (Centre LAN)
- id <*username*>

Welcome to your login shell…

Commands are entered at the shell prompt



Choice of shells:

bash, ksh, zsh

csh, tcsh

Profiles and environments and variables

When you log in you are placed your home directory

You are also placed in a shell (e.g. bash) with a predefined environment

- This environment can be customized by system, group or user
- e.g. editing the file .bash_profile in your home directory

Environment variables define many customizations

- $USER is username, $HOME is home directory
- $PATH defines where to look for programs
- $LD_LIBRARY_PATH defines where to look for libraries

You can see what environment variables are and change them (e.g. in bash)

- printenv $PATH
- LD_LIBRARY_PATH=/apps/well/lib64; export LD_LIBRARY_PATH

Shell prompts and executing commands

Your shell gives you a prompt where you type commands

- The prompt can be customized, but typically looks like
  robert@rescomp1> _

To execute a command you type it in at the prompt and press return

- Some commands are "intrinsic" (*i.e.* they are part of the shell)
- e.g. ls –alrt to show the contents of a directory sorted by modification time
- Most commands are the names of executable files with in directories that are in the $PATH environment variable
- Otherwise type the pathname to the file to get the command to run

More on shells and environments

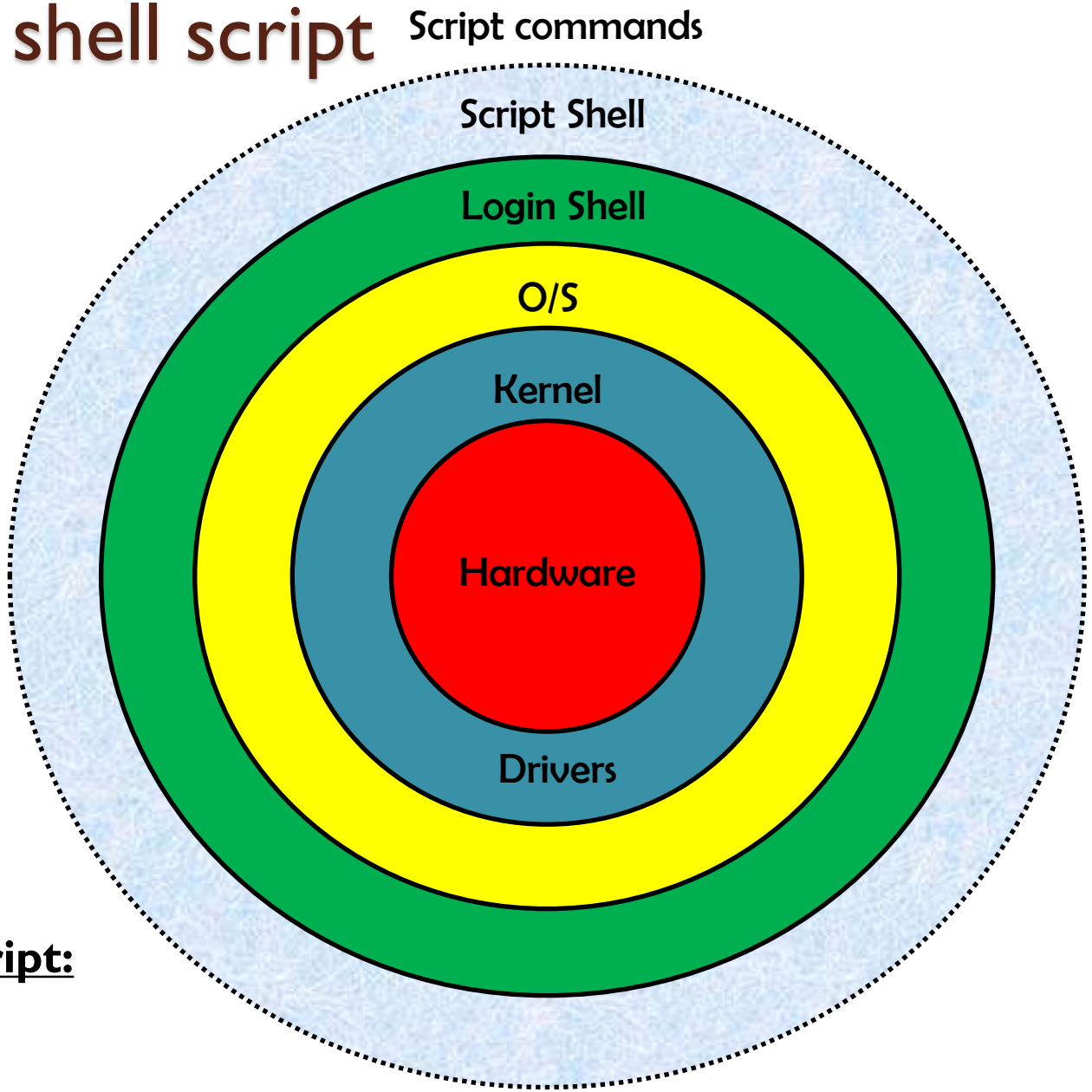
When you run a command it executes in a new child shell

Child shells inherit the environment of their parents, but can change this environment

When a child shell finishes, any environment changes are lost

- If you run a script then any changes to the environment are lost when the script finishes (since the script's shell also exits)
- If you "source" a script then changes to the environment apply to the shell from which "source" was entered
- ./setup-prog-env is probably not what you want
- source ./setup-prog-env is probably right!

Running a shell script



First line of script:

#!/bin/bash

More on setting environment variables

Two environment variables that usually need setting are:

- $PATH defines where to look for programs
- $LD_LIBRARY_PATH defines where to look for libraries

Changing environment variables

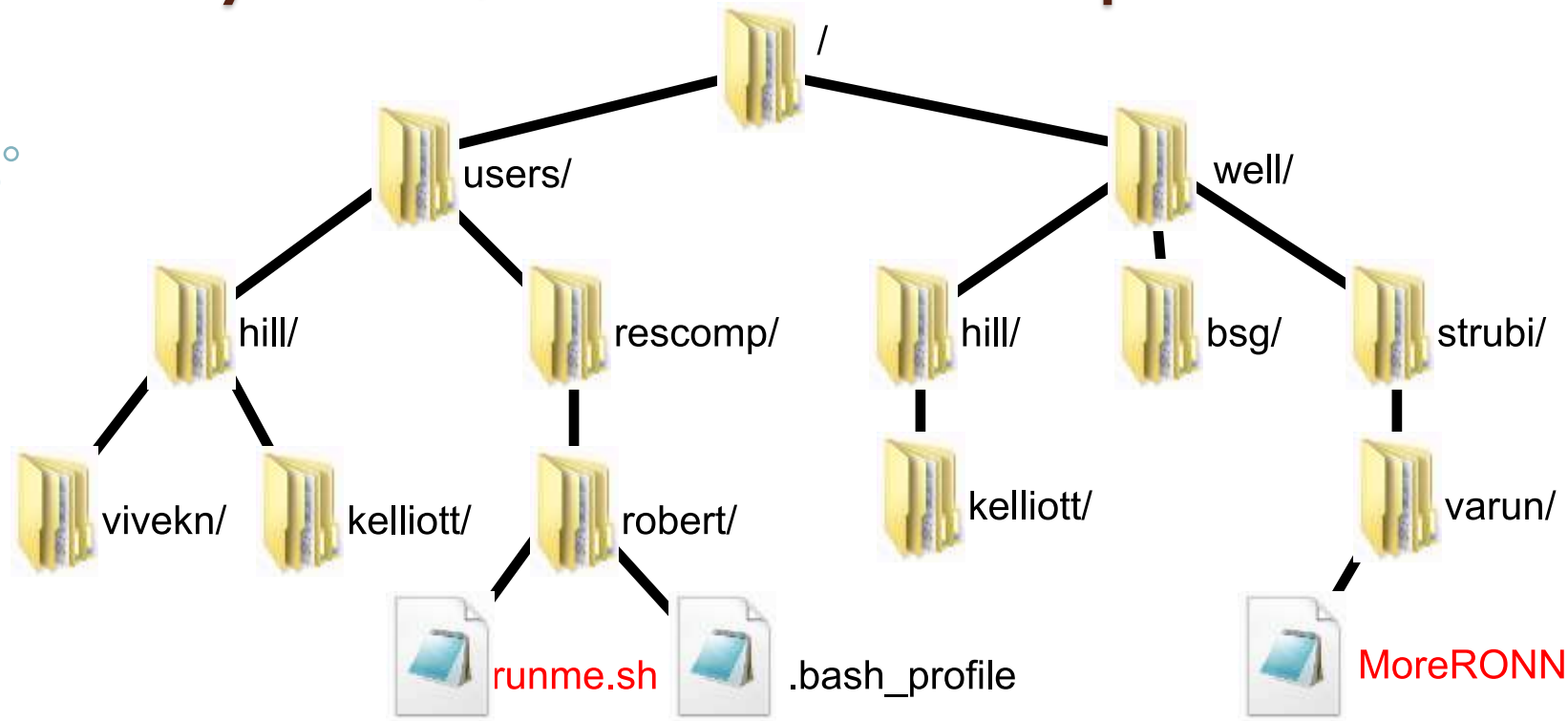
- **tcsh:** setenv PEDSYS /apps/well/pedsys/2.0

  setenv PATH .:$PATH

- **bash:** export PEDSYS=/apps/well/pedsys/2.0

  export PATH=$PATH:~/bin

Filesystems, directories and pathnames



A file is a collection of disk blocks identified by an "inode"

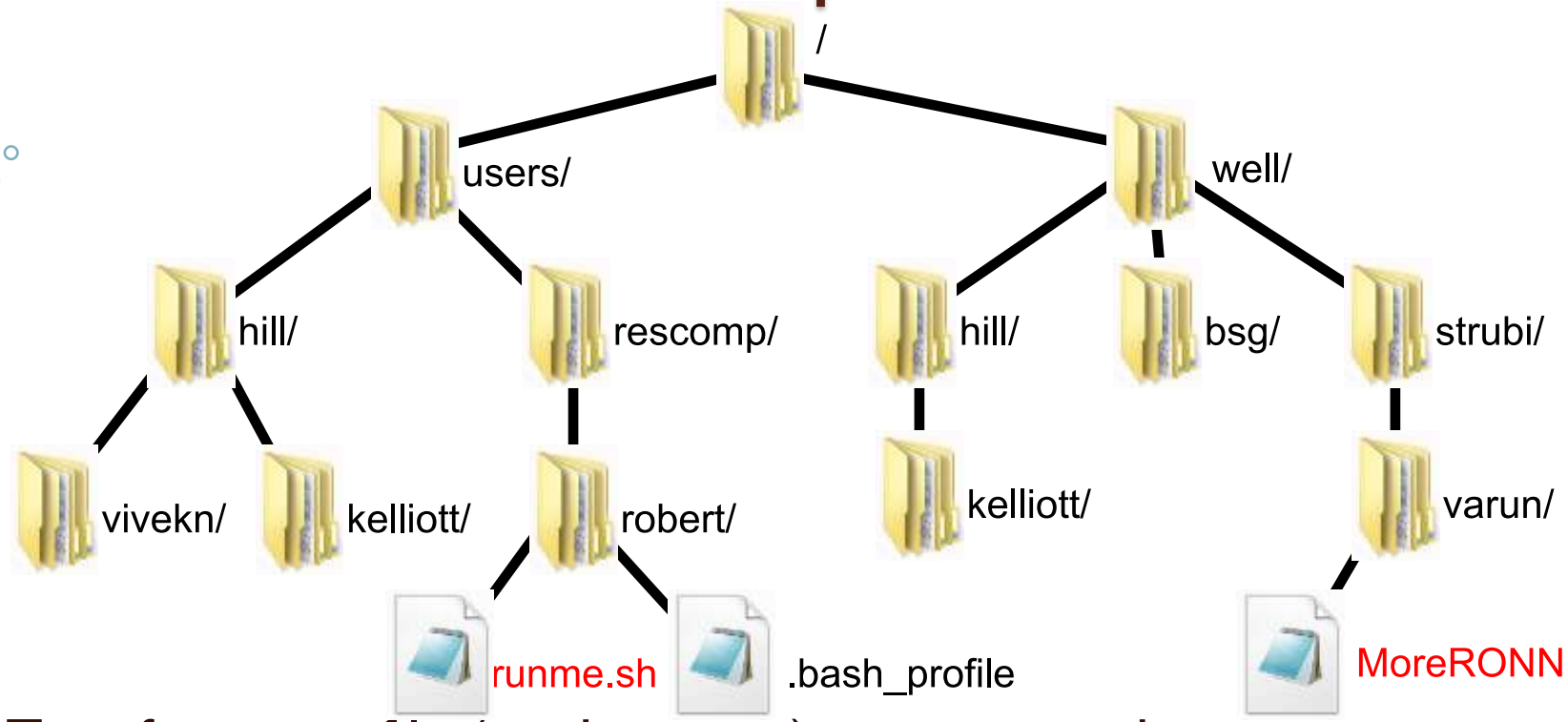
A directory is a file containing the list of filename/inode pairs

A filesystem is a hierarchy of directory and file names

A pathname is the root through the parent directories to a file

- Can be relative to current working directory or absolute from the root

Absolute and relative pathnames



To reference a file (or directory), enter its pathname

- An absolute pathname starts with a "/", the top of the tree
- A relative pathname starts with a name, "." or ".."
- "." means the current directory (output of "pwd")
- ".." means the parent of the current directory
- "~" is a shorthand for "my home directory", $HOME
- "~*<user>*" is a shorthand for *<user>*'s home directory

Unix file permissions

```
[root@head1 robert]# ls -l
total 736
drwxr-sr-x 4 robert rescomp  16384 Jun  4  2014 SGE
drwxr-sr-x 2 robert rescomp  16384 May 15  2014 bin
-rwxr-xr-x 1 root   rescomp     85 Jun 20  2014 qalter-list.sh
-rw-r-x--- 1 root   rescomp  10688 May 28  2014 run1_ct24_f2-9ave2.log
drwxr-sr-x 2 robert rescomp 262144 Nov 25 10:51 sge-test
```

A file belongs to a user, a group, has a size & a modification date

First ten characters explain each file's permissions

- "d" means directory; "-" means normal file (there are others)
- Sets of three characters are permissions: "user", "group" & other"
- "rwx" is "read", "write" & either "execute" (file) or "traverse" (directory)
- A "-" means that permission is not granted
- The "s" (or "S") is the group "sticky bit": files created in that directory inherit the parent directory's group *NOT* the user's group.

To see a file (or directory) you must have <u>both</u>

- Read permission on the file itself
- Traverse permission on all directory parents, grandparents *etc*.

More on permissions…

```
[root@head1 ~]# ls -l /gpfs1/well
total 21248
drwxrwxr-x  11 johnb           304   32768 Oct 18 15:55 1000G
drwxrws---   7 agoel     watkins     32768 Oct 30 15:18 PROCARDIS
drwx------   3 anubha    mccarthy    32768 Nov 17 11:18 anubha
drwx------  23 ashish    got2d       32768 Feb 28  2014 ashish
drwxrwx---   4 hildr     bag         32768 Oct 15 15:52 bag
drwxrws---  23     1933 brc          32768 Dec  8 15:35 brc
```

Permissions of parent directories always add extra restrictions

Top level permissions are crucial to reliable access control

- /well/1000G is read-only to all, writeable by John Broxholme
- /well/brc can only be seen by members of the "brc" group and all files within it will be created with "brc" as the group
- /well/anubha can only be seen by Anubha and no one else

Modifying ownership and permissions – with care!

- **e.g.** chown [-R] <*user*> *file* … **or** chgrp [-R] <*group*> *file* …
- **e.g.** chmod [-R] go-w *file* … **or** chmod 755 *file* …
- find . –type d –exec chmod g+s {} \;

Hard links

Two filenames can share the same inode, this is a hard link

- ln original hardlink
- There is only one actual file!
- Changes made to the same file using either name

```
[root@head1 links]# cp .file1 original
[root@head1 links]# ln original hardlink
[root@head1 links]# cat hardlink
file1
[root@head1 links]# cp .file2 original
cp: overwrite `original'? y
[root@head1 links]# cat hardlink
file2
```

- Cannot "ln" directories because resulting permissions are ambiguous
- Cannot "ln" between filesystems (as inodes are local to a filesystem)
- Cannot "ln" between GPFS filesets (also prevents "mv" without copying)
- rsync involving hard links duplicates data by default (use "-H")

DON'T USE HARD LINKS!

Soft links

One filename can point at another filename, this is a soft link

- ln –s original softlink

```
[root@head1 links]# ln -s original softlink
[root@head1 links]# cat softlink
file2
[root@head1 links]# ls -l
total 8
-rw-r--r-- 2 root root 6 Jan 15 13:35 hardlink
-rw-r--r-- 2 root root 6 Jan 15 13:35 original
lrwxrwxrwx 1 root root 8 Jan 15 13:52 softlink -> original
```

- The file "softlink" has the "l" flag and contains the target filename
- Changes made to the original file using either name
- Can "ln" directories: soft links do not influence permissions
- Can (usefully) "ln" between filesystems (as its just a pointer)
- rsync involving soft links will not (usually) follow them
- Soft links can be "broken" *i.e.* not point to any existing file

USE SOFT LINKS WHERE THEY AID ORGANIZATION!

A practical use of soft links

We have three GPFS filesystems

- /gpfs0: /users/<*group*>/<*user*>, /apps/well, /mgmt
- /gpfs1 & /gpfs2: presented as /well/<*group*>

To allow us to move things around without breaking everyone's work, we use soft links to present stable pathnames:

```
[root@rescomp1 /]# cd /
[root@rescomp1 /]# ls -l apps users mgmt well
lrwxrwxrwx. 1 root root 11 Apr  9  2014 apps -> /gpfs0/apps
lrwxrwxrwx. 1 root root 11 Apr  9  2014 mgmt -> /gpfs0/mgmt
lrwxrwxrwx. 1 root root 12 Apr  9  2014 users -> /gpfs0/users
lrwxrwxrwx  1 root root 11 Aug 28 20:18 well -> /gpfs0/well
[root@rescomp1 /]#
[root@rescomp1 /]# cd /well
[root@rescomp1 well]# ls -al
total 960
drwxr-xr-x 2 root root 16384 Jan  7 17:40 .
drwxr-xr-x 8 root root 32768 Aug 24 10:24 ..
lrwxrwxrwx 1 root root    17 Oct 30 15:33 1000G -> /gpfs2/well/1000G
lrwxrwxrwx 1 root root    21 Aug 25 17:46 PROCARDIS -> /gpfs1/well/PROCARDIS
lrwxrwxrwx 1 root root    18 Aug 25 17:43 ashish -> /gpfs1/well/ashish
lrwxrwxrwx 1 root root    15 Aug 25 17:43 bag -> /gpfs1/well/bag
lrwxrwxrwx 1 root root    15 Aug 25 17:43 brc -> /gpfs1/well/brc
lrwxrwxrwx 1 root root    19 Dec 11 15:30 brcvenn -> /gpfs2/well/brcvenn
lrwxrwxrwx 1 root root    15 Aug 25 17:43 bsg -> /gpfs1/well/bsg
```

File streams and redirection: stdin

Execution of any command (executable file; *i.e.* a program or script) is associated with three default file streams:

- 0 – stdin; 1 – stdout; 2 – stderr

Input to a command comes from stdin

- Usually the keyboard in interactive use
- Can be redirected to source from a file, *e.g.* crack < /etc/passwd
- If already in a script, "<<" indicates the end of included redirection text

```
echo "$RAWK:t SCRIPTNAME < FILE.IN > FILE.OUT"
cat << EOD

A little shell script to ease the pain of running commonly used 'awk' scripts.
The script expects one argument (SCRIPTNAME) which is the name of the script to
execute. It operates the script on the standard input and directs the output to
the standard output. The script name is the name of a file in the directory
'~robert/bin/awk' (or in the directory pointed to by the environment variable
SCRIPT_DIR, if that is set). Currently available scripts are:

EOD
if (-e $SCRIPT_DIR) then
  if (`ls $SCRIPT_DIR | wc -w`) then
    ls $SCRIPT_DIR
  else
    echo "NO SCRIPTS AVAILABLE"
  endif
else
```

File streams and redirection: stdout

Execution of any command (executable file; *i.e.* a program or script) is associated with three default file streams:

- 0 – stdin; 1 – stdout; 2 – stderr

Output from a command goes to stdout

- Usually the terminal in interactive use
- If you want a record of what you did, redirect stdout to a new file, e.g. crack < /etc/passwd > crack.log
- You can also append output to the end of an existing file, e.g. crack < /etc/passwd2 >> crack.log

Note: you can use "<" and ">" in the same command

File streams and redirection: stderr

Execution of any command (executable file; *i.e.* a program or script) is associated with three default file streams:

- 0 – stdin; 1 – stdout; 2 – stderr

Any error messages from a command go to stderr

- Usually the terminal in interactive use, even if stdout is redirected
- Both stdout and stderr can be redirected independently

Here are some perverse bash examples:

- command > output.log
- command 2> output.err
- command 1>&2 (merge stdout into stderr stream)
- command 2>&1 (merge stderr into stdout stream)
- command >& alloutput.log
- dull-command >& /dev/null

Multiple commands and using pipes

You can have multiple commands on a single line

- echo -n “Job run on ”; hostname; echo –n “Start time ”; date
- echo “Job run on `hostname`, start time `date`”

Often you want the output of one command to be the input of another command. This can be achieved using pipes, *e.g.*

- ./do-analysis < datafile > raw.out **followed by** ./prettify < raw.out > pretty.log
- ./do-analysis < datafile > raw.out; ./prettify < raw.out > pretty.log
- ./do-analysis < datafile | ./prettify > pretty.log

Pipes only send stdout, if you want to send stderr as well:

- ./do-analysis < datafile 2>&1 | ./prettify > pretty.log

Sometimes you want more than one copy of output:

- ./do-analysis < datafile | tee raw.out | ./prettify > pretty.log

Foreground and background jobs

When you run a job online, it ties up the terminal until the job finishes, even if its output is redirected

- This is called running in the foreground

Jobs can be killed by typing ctrl-C

Jobs can be stopped – not killed – by typing ctrl-Z

- These jobs sit idle unless made to run in the background, e.g. bg %1
- Background jobs can be brought back to the foreground, e.g. fg %1
- Stopped and backgrounded jobs can be killed, e.g. kill %1

```
[robert@login1 ~]$ sleep 100
<ctrl-Z>
[1]+  Stopped                 sleep 100
[robert@login1 ~]$ jobs
[1]+  Stopped                 sleep 100
[robert@login1 ~]$ fg %1
sleep 100
<ctrl-C>
[robert@login1 ~]$ jobs
[robert@login1 ~]$
```

Running jobs in the background

(Multiple) jobs can be run immediately in the background:

- ./slow-analysis < datafile > output.log &

You can then carry on with other work

- Process is still attached to the terminal
- Log out and the process dies!

You can use "nohup" to detach process from the terminal, e.g.

- nohup ./slow-analysis < datafile >& output.log &
- If stderr is not redirected, then any error messages will be lost

You can use "screen" to give a virtual terminal session

- Can manage multiple sessions from a single terminal window
- Detached "screens" do not die when you log off
- Can re-attach to these "screens" later and from other terminals
- Can keep giving input to "screen" – not possible with "nohup"

A simple shell (command) script

We created the following file using a text editor

- vi script.sh for hard-core computer users!
- "gedit" and "nano" are alternative graphical editors
- avoid using Windows-based editors!

```
[nilufer@login1 ~]$ ls -al script.sh
-rwxr--r-- 1 nilufer zondervan 474 Jun 20 09:17 script.sh
[nilufer@login1 ~]$ cat script.sh
#!/bin/bash

echo "************************************************************"
echo "Run on host: "`hostname`
echo "Operating system: "`uname -s`
echo "Username: "`whoami`
echo "Started at: "`date`
echo "************************************************************"

R --vanilla << EOD
help()
q()
EOD

echo "************************************************************"
echo "Finished at: "`date`
echo "************************************************************"
exit 0
[nilufer@login1 ~]$
```

Running the simple shell script

- ./script.sh > script.log

```
[nilufer@login1 ~]$ cat script.log
************************************************************
Run on host: login1.cluster3
Operating system: Linux
Username: nilufer
Started at: Thu Jun 20 09:23:05 BST 2013
************************************************************

R version 2.14.0 (2011-10-31)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-unknown-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
     .
     .
     .
     .
     .
     ## For programmatic use:
     topic <- "family"; pkg_ref <- "stats"
     help((topic), (pkg_ref))


> q()
************************************************************
Finished at: Thu Jun 20 09:23:05 BST 2013
************************************************************
[nilufer@login1 ~]$
```

An aside: sourcing the simple shell script

Running the script creates a child (or sub) shell

- Script must be set to be executable, e.g. chmod u+x script.sh
- Shell exits and the environment is not preserved on completion

The alternative is to "source" the script

- Executes commands in the current shell
- Does not require script to be "executable"

```
[nilufer@login1 ~]$ chmod 600 script.sh
[nilufer@login1 ~]$ source script.sh > script-source.log
[root@login1 ~]#
```

Output is the same as before

- Note that "exit" statement closed my "nilufer" login shell!

This script is already "cluster-ready"!

On a cluster submission node we could simply type

- qsub script.sh

"qsub" and its options are the subject of the next talk

```
[nilufer@login1 ~]$ cat script.sh
#!/bin/bash

echo "*************************************************************"
echo "Run on host: "`hostname`
echo "Operating system: "`uname -s`
echo "Username: "`whoami`
echo "Started at: "`date`
echo "*************************************************************"

R --vanilla << EOD
help()
q()
EOD

echo "*************************************************************"
echo "Finished at: "`date`
echo "*************************************************************"
exit 0
```

Thank you for your attention… any questions?

Next talk 10am tomorrow… "Submitting jobs to the cluster"