Securing File Access Permissions on Linux

Ramses van Zon

SciNet HPC, University of Toronto

4 October 2024

Motivation

Sharing

I want my

- supervisor
- colleague
- friend

to be able to see or change files that I own on a shared-use cluster.

Security

I don't want others to be able to

- access
- modify, or
- delete my files,

neither on purpose nor by accident.

I.e., I don't want to be vulnerable to cyberattacks.

- The Linux operating system has built-in tools to control file access.
- These controls specify which users and groups can access which files and directories.
- Very powerful, but needs to be used properly.
- That's where today's session comes in.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Understanding Linux Permissions

User organization in Linux

- Users in Linux are organized in groups, so called posix groups.
- A user can be part of several posix groups.
- This is very useful if you want to share files with a specific set of users (e.g. your research group).

Permissions

Try this: Find out what groups you're part of!

Could be this:

```
$ whoami
rzon
```

```
$ groups
scinet ccstaff
```

```
$ id -gn
scinet
```

But on another system:

```
$ whoami
rzon
```

```
$ groups
rzon def-rzon ccstaff ...
```

```
$ id -gn
rzon
```

The latter command returns your **primary group**.

Ownership and sharing in Linux

The purpose of these groups is to control access and sharing.

- All files and directories are always 'owned' by a specific user.
- In addition, files have a 'group ownership' property, which shows to which group they belong.

Try this: Find out what group your files belong to.

```
$ ls -al .
total 24
drwxrwxr-x 2 rzon scinet  4096 Sep 11 00:17 .
drwxrwxr-x 4 rzon scinet  4096 Sep 11 00:17 ..
-rw-rw-r-- 1 rzon scinet  6251 Sep 11 10:07 slide1.pdf
-rw-rw-r-- 1 rzon ccstaff 8245 Sep 11 10:08 slide2.pdf
```

This is a long listing (`-l`) of the current folder (`.`) showing all (`-a`) files and folders (even hidden ones).

Note: if you want to only see the permissions of the directory instead of its content, add the `-d` *option to* `ls`.

Understanding the output of `ls -al`

```
$ ls -al .
total 24
drwxrwxr-x  2 rzon scinet  4096 Sep 11 00:17 .
drwxrwxr-x  4 rzon scinet  4096 Sep 11 00:17 ..
-rw-rw-r--  1 rzon scinet  6251 Sep 11 10:07 slide1.pdf
-rw-rw-r--  1 rzon ccstaff 8245 Sep 11 10:08 slide2.pdf
```

This is a table with one row per file or directory with the following fields:

1. The mode, a cryptic-looking permission strings that will be explained soon;
2. A number showing how many places in the file system link to it;
3. The owner of the file or directory;
4. The group membership of the file or directory;
5. The size of a file;
6. Its date of last modification;
7. Its name.

drwxrwxrwx (i.e. file permissions)

```
$ ls -al .
total 24
drwxrwxr-x 2 rzon scinet  4096 Sep 11 00:17 .
drwxrwxr-x 4 rzon scinet  4096 Sep 11 00:17 ..
-rw-rw-r-- 1 rzon scinet  6251 Sep 11 10:07 slide1.pdf
-rw-rw-r-- 1 rzon ccstaff 8245 Sep 11 10:08 slide2.pdf
```

a b c d b c

d

The symbolic mode consists of 10 characters that represent the permissions set for each file/directory.

a) The first character is `d` is it's a directory, `l` if it's a link, otherwise `-`.

b) The next three are the permission for the user, i.e., the owner.
They can be `rwx`, for read, write, and execute permission.
If a permission is not "set", the character at its position is `-`.

c) The following three are the permissions for members of the group to which the file belongs.

d) The final three are the permissions for others a.k.a. the world.

There are more possibilities, but more on that later, basics first.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Now look at the listing again:

```
$ ls -al .
total 24
drwxrwxr-x 2 rzon scinet  4096 Sep 11 00:17 .
drwxrwxr-x 4 rzon scinet  4096 Sep 11 00:17 ..
-rw-rw-r-- 1 rzon scinet  6251 Sep 11 10:07 slide1.pdf
-rw-rw-r-- 1 rzon ccstaff 8245 Sep 11 10:08 slide2.pdf
```

Let's understand what different users are allowed to do with these files.

- `.` and `..` are directories (in fact, the current directory and its parent).

  They can be written to, read from, and executed by the user, or anyone in the `scinet` group. The rest of the world can only read and execute.

- `slide1.pdf` is not a directory, and cannot be executed, but can be written to and read from by `rzon` and members of the `scinet` group, and read by others.

- Similar for `slide2.pdf`, but writing is restricted to members of the `ccstaff` group.

What do r, w, and x really mean?

File permissions

read	you (user, group member, other) can see the content of the file.
write	you can make changes to this file.
execute	you may run this file as an app.

For sharing content, **read** is often exactly what you want (but must have executable path).

For app sharing, **execute** is enough for binary applications, but for scripts you also need **read**.

Only the owner needs **write**, usually.

Directory permissions

read	you (user, group member, other) can see the names of files in this directory.
write	you can make add files **and remove** any files in this directory.
execute	you may descend into this directory.

Every directory in the path to shared content requires **execute** or the access with be blocked.

For *exploring* shared content (e.g. `ls`), **read** is often exactly what you want.

When a folder is a shared repo, **write** for a group may work (but make it sticky – later).

Let's look at another example

List the top directory

Who's this "root"?

- "root" is the administrative account.
- It owns OS files.
- Users can access these files and directories due to the permissions.
- On a shared system, you won't have access to the root account.

```
$ ls -l /
drwxr-xr-x   4 root root     0 Sep 11 00:51 cvmfs/
drwxr-xr-x  18 root root  3200 Oct  6 20:00 dev/
drwxr-xr-x  93 root root  4340 Oct  4 18:20 etc/
drwxr-xr-x   1 root root    14 Oct  4 18:11 home/
lrwxrwxrwx   1 root root     7 Oct  4 18:09 lib -> usr/lib/
drwxr-xr-x   2 root root    40 Apr 11  2018 media/
drwxr-xr-x   2 root root    40 Apr 11  2018 mnt/
drwxr-xr-x  10 root root   220 Oct  4 18:11 opt/
dr-xr-xr-x 821 root root     0 Oct  4 18:08 proc/
drwxrwxrwx   1 root root    17 Oct  4 18:11 project/
dr-xr-x---   3 root root   260 Oct  7 14:39 root/
drwxr-xr-x  23 root root  1060 Oct  4 18:20 run/
lrwxrwxrwx   1 root root     8 Oct  4 18:09 sbin -> usr/sbin/
drwxr-xr-x   1 root root    17 Oct  4 18:11 scratch/
drwxr-xr-x   2 root root    40 Apr 11  2018 srv/
dr-xr-xr-x  13 root root     0 Oct  4 18:11 sys/
drwxrwxrwt 516 root root 26100 Sep 11 00:55 tmp/
drwxr-xr-x  14 root root   300 Oct  4 18:09 usr/
drwxr-xr-x  21 root root   500 Oct  4 18:11 var/
```

Note: Symbolic links have `lrwxrwxrwx`. In reality they inherit permissions from the linked object.

More Linux commands to inspect permissions

stat [FILE|DIR]

Provides detailed information like ownership, access, modify, and change times, permissions, and permission changes.

```
$ stat $HOME
  File: /home/rzon
  Size: 33792       Blocks: 66         IO Block: 131072 directory
Device: 3510,432756 Inode: 144115205289280723  Links: 42
Access: (0700/drwx------) Uid: (3000558/rzon) Gid: (2000000/scinet)
Access: 2024-10-02 12:38:11.000000000 -0400
Modify: 2024-10-02 13:33:46.000000000 -0400
Change: 2024-10-02 13:33:46.000000000 -0400
```

tree [OPTIONS] [DIR]

Recursively shows files and directories, optionally with user and group (`-u` and `-g`), permissions (`-p`) and time (`-D`).

Limit search depth using `-L 2`.

```
$ tree -pD -L 2 $SCRATCH
[drwx------ Sep 20  2024]  /scratch/rzon
  [drwxr-s--- Sep 20  2024]  sourcecode
     [-rw-r----- Sep  2 13:39]  bestappever.cpp
  [drwxr-x--- Mar 17  2024]  clones
     [drwxr-x--- Aug 15  2024]  rarray
     [drwxr-x--- Aug 15  2024]  ish
  [drwxr-x--- Oct 1  2024]  world
      [drwxr-xr-x Oct 1  2024]  hello
```

More Linux commands to inspect permissions

find

Will recursively look for files and directories based on selection criteria.

Many selection criteria, check the `man` page.

Syntax to search for specific permissions

```
find [DIR] [-maxdepth N] -perms PERMISSION [ACTION]
```

Here:

- `DIR` is the directory to scan; if omitted, starts from current directory.
- `N` is the depth, i.e, how mnay directory to go into.
- `PERMISSION` is a symbolic or numeric mode to look for. Prepend a slash to make it a mask.
- `ACTION` is what to do with each file. If omitted, `find` only lists the filename+path.
  `-ls` is a common choice for `ACTION`, it gives a long listing.

E.g. to find scratch files and folders that are world readable but at most two levels deep:

```
$ find $SCRATCH -maxdepth 2 -perm /o+r -ls
8190     25 drwxr-xr-x  22 rzon rzon  25600 Mar 17  2023 /scratch/rzon/world/hello
```

Controlling Ownership and Permissions

There is only one owner for each file

- After you are logged in as a user, when you create a file, you are its owner.
- Its group is whatever your primary group is (unless the directory has setgui - more later).
- The owner can change the group ownership, but not user ownership.
- The owner can set permissions such that other group members, or even anybody, have nearly the same permissions as the owner, except for the permission to set permissions.
- This include the possibilty to allow removal of the files and directories by anyone.

This has happened, even very recently. Users with directories that were writable by "other" had their files removed by an erroneous command by another user. You have full control to set what is allowed with your files, but with great power...

Default permissions

- The settings for default permissions of new files and the user's primary group vary from system to system.
- On Cedar, Beluga, Graham, and Narval, each user has its own private group as their primary group. By default, new files in home and scratch are not shared with others in the research group.
- On Niagara, your primary group is that of the sponsor of your primary role, and your files are readable and executable by other group members by default.
- New file permissions are restricted by the umask (about which more later).

With so many possibilies, and with defaults that may not work for everyone, we need tools to change and control the permissions.

Changing Permissions with `chmod`

Suppose we create a new bash script to print “hello world!” to the console:

```
$ echo 'echo hello world!' > newfile.sh
$ ls -l newfile.sh
-rw-r--r-- 1 rzon scinet 18 Sep 11 10:26 newfile
```

This file cannot be executed:

```
$ ./newfile.sh
bash: ./newfile.sh: Permission denied
```

With `chmod +x` we can change this:

```
$ chmod +x newfile.sh
$ ls -l newfile.sh
-rwxr-xr-x 1 rzon scinet 18 Sep 11 10:26 newfile
$ ./newfile
hello world!
```

Note that everyone got execution permissions!
(Don’t worry if you don’t know why yet - we’ll get to that.)

Controlling permissions of existing files

Add permissions:

```
$ chmod +r newfile.sh
$ chmod +w newfile.sh
$ chmod +rxw newfile.sh
```

Adding permissions just for owner

```
$ chmod u+r newfile.sh
$ chmod u+w newfile.sh
$ chmod u+rxw newfile.sh
```

Adding permissions just for group

```
$ chmod g+r newfile.sh
```

Adding permissions for all others

```
$ chmod o+r newfile.sh
```

Removing permissions:

```
$ chmod -r newfile.sh
$ chmod -w newfile.sh
$ chmod -rxw newfile.sh
```

Removing permissions just for owner

```
$ chmod u-r newfile.sh
$ chmod u-w newfile.sh
$ chmod u-rxw newfile.sh
```

Removing permissions just for group

```
$ chmod g-r newfile.sh
```

Removing permissions just for all others

```
$ chmod o-r newfile.sh
```

Numeric options for `chmod`

Each character in the permission string is on or off, so it's a bit.

The three characters for rwx can be seen as an *octal*, 3-bit number, where

```
0 -> no permissions
1 -> r
2 -> w
3 -> rw
4 -> x
5 -> rx
6 -> wx
7 -> rwx
```

Play around and see the net effect

```
$ chmod 755 newfile.sh
$ ls -l newfile.sh
-rwxr-xr-x 1 rzon scinet 18 Sep 11  10:26  newfile.sh
```

```
$ chmod 655 newfile.sh
$ ls -l newfile.sh
-rw-r-xr-x 1 rzon scinet 18 Sep 11  10:26  newfile.sh
```

```
$ chmod 777 newfile.sh
$ ls -l newfile.sh
-rwxrwxrwx 1 rzon scinet 18 Sep 11  10:26  newfile.sh
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Changing group ownership with `chgrp`

- To change which group you could share a file/folder with, use the `chgrp` command:

```
$ chgrp GROUP NAME
```

  where `NAME` is the name of a file or directory
  and `GROUP` is the name of a group of which you are part.

  Afterwards, the group permissions of NAME are applied as pertaining to members of `GROUP`.

- Whole directory group changes can be affected with:

```
$ chgrp -R GROUP DIR
```

  This applies group membership recursively to all files in `DIR`.

  However, this does not apply to files created afterwards in that directory.

Controlling permissions of new files

The permissions of new files are subject to the user file creation mask, a.k.a. the umask.

The umask is a mode that specifies the bits that should not be set when creating new files. I.e., if the umask has a bit set, new files get that bit unset.

Inspecting the umask

Numerically: `umask`, symbolically: `umask -S`.

```
$ umask
0022
$ umask -S
u=rwx,g=rx,o=rx
```

Numerically, the disallowed bits are shown: 0 states that no user bits are masked, 22 means that the group and other write-bits are masked. But symbolically, the allowed bits are shown!

Setting the umask for new files

You can do this by specifying the mode as an argument to the umask command, e.g.

```
$ umask 0027
```

disallows write permission for group members and disallows all permissions for others.

Caveats:

- This only persist until the shell exits.
- File permissions changes using `chmod` are **not** subject to the umask.

Controlling group ownership of new files

- New files get the group ownership corresponding to the user's primary group.
- For a shell with another group as the primary group, use `newgrp`
- New files created in this shell are part of the other group.
- You can also run a command with a different primary group using `sg`.

```
$ groups
scinet ccstaff
$ touch abc
$ ls -l abc
-rw-r----- 1 rzon scinet 0  Oct  3  10:38  abc
```

```
$ newgrp ccstaff
$ groups
ccstaff scinet
```

```
$ touch def
$ ls -l abc def
-rw-r----- 1 rzon scinet  0  Oct  3  10:38  abc
-rw-r----- 1 rzon ccstaff 0  Oct  3  10:38  def
$ exit # returns to the previous shell
```

```
$ sg ccstaff 'touch ghi'
$ ls -l abc def ghi
-rw-r----- 1 rzon scinet  0  Oct  3  10:38  abc
-rw-r----- 1 rzon ccstaff 0  Oct  3  10:38  def
-rw-r----- 1 rzon ccstaff 0  Oct  3  10:38  ghi
```

Special permissions

A few common use cases missing so far

Yes: SetGID!

1. If a directory has a non-primary group membership, it's likely for sharing. So often we want all files in there to have that same group membership. Can't that be automatic?

Yes: Sticky Bit!

2. If a directory has group-write membership, it is likely a sort of repository where members of the group can deposit files. But with write permission, these same members could remove the deposits of others. Is there a way to prevent this?

Yes: SetUID!

3. If an executable has group or world executable permissions, sometimes it may need to be run as the original user. Is that possible?

These are three additional bits to the file/directory mode that can be set.

The Set Group ID Bit

If a directory has a non-primary group membership:

```
$ chgrp GROUP DIR
```

then it's likely for sharing.

So often we want all files in there to have that same group membership.

If this DIRectory's set group id a.k.a. setGID is set:

```
$ chmod g+s DIR
```

then any new file in that directory will get the group membership.

And any new directory will get group membership and be SetGID as well.

Note: files and directories created before setting SetGID will not be affected!

The Set Group ID Bit: Example

Let's create a directory for (read-only) sharing:

```
$ mkdir shrdir
$ chgrp ccstaff shrdir
$ ls -ld shrdir
drwxr-x--- 2 rzon ccstaff 4096 Oct 4 08:52 shrdir
```

Before SetGID, creating a file in this directory create a file that ccstaff cannot read:

```
$ echo "hi" > shrdir/file1
$ ls -l shrdir
-rw-r----- 1 rzon scinet  3 Oct 4 08:58 file1
```

Now SetGID, and add another file

```
$ chmod g+s shrdir
$ ls -ld shrdir
drwxr-s--- 2 rzon ccstaff 4096 Oct 4 08:52 shrdir
$ echo "there" > shrdir/file2
$ ls -l shrdir
-rw-r----- 1 rzon scinet  3 Oct 4 08:58 file1
-rw-r----- 1 rzon ccstaff 6 Oct 4 08:58 file2
```

Success! `file2` can be read by the ccstaff group.

The Sticky Bit

If a directory has group-write membership:

```
$ chgrp GROUP DIR
$ chmod g+rwx DIR
```

it is likely a sort of repository where members of the group can deposit files.

But with write permission, these same members could remove the deposits of others.

Prevent others deleting a file with the sticky bit

When setting the sticky bit on the containing directory:

```
$ chmod o+t DIR
```

then only the owners can remove their files from the directory.

Set User ID

If an executable has group or world executable permissions, sometimes it may need to be run as the original user.

This is possible by setting the SetUID.

An executable set a SetUID that is run by another user will run as that user.

Some system utilities need this.

But this is generally a huge security risk.

Don't do this yourself!

Numerical values of the special permissions

A fourth octal number in the mode

The special permissions form a fourth triplet in addition to the “user”, “group”, and “other” triplets.

- SetGID has value 4.
- Sticky bit has value 1.
- SetUID has value 2.

The sum of these form a fourth *octal* number.

The mode of a file consists now of 4 octal numbers.

This is why `stat` and `umask` returned 4 numbers.

Symbolic representation

This works as follows:

- If the SetGID bit of a directory is set, the “group” `x` becomes an `s`. (or `S` if `x` was not set).
- If the sticky bit is set, , the “other” `x` becomes an `t`. (or `T` if `x` was not set).
- If the SetUID is set, the “user” `x` becomes an `s`. (or `S` if `x` was not set).

Beyond POSIX: Access Control Lists

Sharing with other users

- What if you need to share with users outside your group,or with multiple groups, or even with selected users in your group?
- The standard “user, group, other” POSIX permissions do not suffice for that.
- You need to use Access Control Lists, .a.k.a. ACLs.

Access Control Lists

- ACLs are file permissions that can be overlaid on Linux files.
- They allow more granulated multiple user and multiple group access control.
- The commands differ per file system:

For Lustre and local Linux file systems (Cedar,Graham,Béluga,Narval)

- `getfacl` to see the ACL permissions of a file
- `setfacl` to set permissions on a file

For GPFS (Niagara,Mist)

- `mmgetacl` to see the ACL permissions
- `mmputacl` to alter them
- `mmdelacl` to remove any previous added ACL

ACL Examples for Lustre file systems

```
$ setfacl -m u:jdoe:rx my_script.py
```

- Gives user `jdoe` read and execute permissions to `my_script.py`.

```
$ setfacl -d -m u:jdoe:rwX /home/USER/projects/def-PI/shareddata
```

- Sets default access rules to directory `/home/USER/projects/def-PI/shareddata`, so any file or directory created within it inherits the same ACL rule. Required for new data.
- The X attribute above (compared to x) sets the **execute** permission only when the item is already executable.

```
$ setfacl -R -m u:jdoe:rwX /home/USER/projects/def-PI/shareddata
```

- Sets ACL to directory `/home/USER/projects/def-PI/shareddata` and all its current content. So it is applicable only to existing data.

```
$ setfacl -bR /home/USER/projects/def-PI/shareddata
```

- Removes the ACLs in the `shareddata` directory recursively.

ACL Examples for GPFS file systems

`mmputacl` works slightly differently:

- does not have a recursive option.
- requires use of a ***permissions file***, which could look like this:

```
user::rwxc
group::----
other::----
mask::rwxc              #must set this to the largest allowed
user:USER:rwxc
user:PI:rwxc            #read and WRITE permissions to group PI
group:OTHERGROUP:r-xc   #read-only for members of OTHERGROUP
```

A nice way to create a permission file is to use the previous ACL as a starting point:

```
$ mmgetacl /project/g/group/owner > permissions.acl
```

Then edit the file with desired permissions, and run

```
$ mmputacl -i permissions.acl /project/g/group/owner
$ mmputacl -d -i permissions.acl /project/g/group/owner
```

In the last command, the `-d` means setting the default permissions for new files and directories in that directory. New subdirectories inherit the ACL.

Recursive ACLs for GPFS

Because of the lack of a recursive option, to apply or change ACLs in an existing directory tree you need go down the tree yourself, e.g.

```
$ mmputacl -i permissions.acl /project/g/group/owner
$ mmputacl -i permissions.acl /project/g/group/owner/dir1
$ mmputacl -i permissions.acl /project/g/group/owner/dir1/subdir2
$ mmputacl -d -i permissions.acl /project/g/group/owner/dir1/subdir2
```

The `find` command can help you here, as it has an `-exec` option that can apply a command to each file/directory it finds (see its **man page**).

Querying and checking ACLs

Whenever you set permissions, make sure you check them afterwards!

- In `ls -l`, any files and directories with ACLs have an additional "+" appended to their mode.
- You can then check the ACLs with `getfacl` or `mmgetacl`.
- Unfortunately, `stat` and `find` do not reflect ACL permisions.

Security Pitfalls with File Permissions

Overly permissive permissions

Anytime permissions don't work, it is tempting to 'just' run `chmod 777`.

But this opens permissions completely to ***anyone***, which is undesirable, particularly on shared systems.

- Anyone can now read, change, delete or add files and directories to your account.
- You might not even be able to log in anymore.

Be careful and diligent!

Do not give away any more permissions than is necessary.

Confusing chmod with chown or chgrp

`chmod` changes permissions of a file or directory.

`chgrp` changes group membership of a file or directory.

- This changes to which group the permissions set by chmod apply.
- It also changes towards which quota the file or directory counts.

`chown` would change the ownership of a file.

- Only the `root` user can do this, or a user with `sudo` powers.
- Not possible on any on the Alliance clusters.

Trying to set permissions on other's files

- Sometimes, users try to change permissions or apply ACL to files that they do not own.
- That won't work. Only the owner of the file can do this.

So if you need to gain access to files of a collaborator, or of a student, you need to ask them to change the permissions.

Note

Files in project are, by default, in a group associated with an allocation.

On the General Purpose clusters (Cedar, Graham, Béluga, and Narval), by default, files in $HOME and $SCRATCH are in a group with the user as the only member. Thus, other research group member do not have access.

On the Large Parallel cluster Niagara, as well as on Mist, by default, files in $HOME and $SCRATCH are in the group of your sponsor, and are readable by others also in the research group of that sponsor.

Permissions do not work along directory tree

- When you open permissions at a particular sub-level (chmod or ACL), but overlook the levels above, which are oftentimes owned by the PI, the net result is that the permissions still don't work.
- Don't overreact in the worst possible way, and run `chmod -R 777`.

Set the `x` permission of all parent directories, or, if owned by someone else, ask them to set that permission.

Setting permissions on .ssh

- Giving bulk `+r` permission to `$HOME` will expose contents of `$HOME/.ssh`
- This is potentially very dangerous as it may contain private ssh keys.
- It can also prevent you from using `ssh`

Resist the urge to given bulk permissions!

Detecting and Fixing Permission Down a Tree

- Users may run the command below to find files/directories with permissions 777 (rwxrwxrwx)

```
$ find -perm -2 -not -type l
```

(search and show, world-writable files under current directory)

- Recursive ways to use chmod, e.g.

```
$ chmod -R o-rwx directory_name
```

- Recursive ways to use chgrp, e.g.

```
$ chgrp -R ccstaff directory_name
```

Final notes

Summary

- You have a lot of control over permisions on files and directories in Linux.
- The tools to control these (`chmod`, `chgrp`, `setfacl`, `mmsetacl`) are a bit intricate.
- Do not give bulk permissions!
- Keep it simple: don't start removing permissions to be safe if you're not sure what you are doing.
- Always check permissions (with `ls`, `stat`, `find`, `getfacl`, `mmgetacl`).
- Contact support if you need help: support@tech.alliancecan.ca

References

- https://docs.alliancecan.ca
- https://docs.alliancecan.ca/wiki/Sharing_data
- https://docs.alliancecan.ca/wiki/Data_management_at_Niagara (Niagara/GPFS specific)

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO