Filesystems

COSC301 Laboratory Manual

This lab should be reasonably relaxed, assuming you read the lab before you come to class. Hopefully you have started working on your first assignment by now when you have time. We will have a stroll around the standard Unix file hierarchy on your virtual machine Client1, and study some of the things we find there. For people familiar with the Unix file system, you may find it rather easy.

We will talk of file-system permissions and how they can be managed, and finally we shall learn about creating archives and backups.

The Design of File-Systems

As a small aside into the design of file-systems (and operating systems in general), it is useful to look at other possible solutions. History is a good teacher here. There are two books you should read when you have some time spare. The first is The Art of Unix Programming [http://www.catb.org/esr/writings/taoup/html/], and the other is Unix Haters Handbook [http://www.simson.net/ref/ugh.pdf], both of which are good reading. The latter book gives some good historical insight into other systems, but beware that much of its criticisms are at the historical Unix, some of the problems have been remedied in modern systems.

1. Tour the Virtual File System (VFS)

1.1. Items Found in the Filesystem

Unlike some other Operating Systems you may have heard of, the Unix filesystem is a tree structure. It has only one root (/). It doesn't use any concept of drive letters. Every filesystem you use is *mounted* (attached, or made available) onto a directory (*mount-point*) of the filesystem.

There are various types of items you find in the file-system.

Regular files and directories
: Nothing unusual about these. In **ls -F** output (**ls** is short for list), directories have / appended to them. In **ls -l** output, the first character on the line is a - for a regular file, or d for a directory.

Hard link
: Recall that Unix filesystems are *inode* based. The inode is a number that points to the data. Directory entries point to an inode, not to the data directly. A hard link, as created using the **ln** command (short for link), creates another directory entry that points to the same inode.

  A hard link can only point to a file on the same filesystem, and are invisible to the naked eye in standard **ls -l** output. You can tell by using **ls -li** command to display the inode column, and looking for files with the same inode..

***Symbolic links, soft link* or *symlinks*.**
: These are similar to shortcuts under Windows, or aliases under MacOS or Mac OS X, but are more transparent. When a program opens a symbolic link, it opens the file it points

to. Symbolic links can also point to directories. Actually, symbolic links can contain any text. If the target does not exist, the link is said to be a *broken link* or *dangling link*.

Symbolic links are created with the **ln -s** command. In **ls -l** output the first character on the line is a `l`, the link target is also printed after the filename.

The only way to remember in what order the arguments of **ln** go is to think of the **ln** command a bit like **cp**, where the source goes first, and the destination (or *new thing*) comes last.

Device node

One of the design philosophies of Unix is "Everything is a file". While this isn't entirely true in Linux (network interfaces, for example) all other devices are represented by a device node in `/dev`, which might or might not be a virtual filesystem.

For example, the first serial port on a machine would be presented by `/dev/ttyS0` (equivalent to COM1 in DOS). An application that interacts with the serial port, such as **minicom**, a serial terminal emulator, will open `/dev/ttyS0`.

This allows access-control to be placed on such devices, in the same way as you would any other file-system object.

Device nodes are defined with a *major number* and *minor number*, and whether or not the device is a *block special file* or a *character special file*.

ls -l output that starts with a `b` is a block special file, a `c` indicates a character special file. A device such as a hard disk is a block device, most others are character devices.

The major number indicates the type of device, such a hard disk. The minor number indicates the particular device, such as a particular partition on the a particular hard disk. It is this `major:minor`, not the name, that tells the operating system kernel which device the user is opening, and thus which driver to pass the request to.

Socket

Unix systems have a form of network connection known as a *Unix domain socket*[1], which is parallel to an *IP socket*, but is entirely local to the machine. Unix domain sockets can do some things that can't be done over other socket types, such as passing open files and credentials, and because they use the filesystem, additional access control can be applied.

A server process would create a *socket file* as its socket address, and client processes on the same machine can connect to the server by using the socket file as the destination address (i.e. the server address).

We can know which file is a socket file with the following commands. In **ls -F** output, a = is printed after a socket file. In **ls -l** output, a `s` at the first character means a socket file.

There is no command to create a socket file. It is created automatically when a program binds a socket to the file with the system call bind(2) .

Named pipe* or *FIFO

Pipes are very useful in Unix. *Anonymous pipes* in particular form of the most important constructs for the Unix command-line environment. They allow the output of one process to be fed into the input of another, such as when we use the command-line construct such as `ls -l | sort -n +4`, which sorts the output of **ls -l**. Unlike sockets, pipes are unidirectional.

[1] Also known as *local sockets*

However, sometimes you want to be able to create something more flexible which would be impractical to express using the shell's pipe (`|`) operator. To do this, you can create a *named pipe* somewhere in the filesystem, using **mkfifo**. In **ls -l** output, `p` at the start indicates a FIFO.

Named pipes are somewhat deprecated in favour of sockets. However, because they are more useful in shell scripts compared to sockets, they are still occassionally used, and don't appear to be in any danger of disappearing soon.

1.2. Hierarchy

The file hierarchy in Unix systems, including Linux, can be a confusing beast at times. This section will give you some familiarity with the purpose of various directories. Afterwards, you can have a brief read of hier(7), which should describe the hierarchy of the Unix filesystem on such a system.

`/`
: The root of the entire filesystem.

`/bin`
: Programs (binary files) that don't require administrative rights, and need to be available at boot time. Commands such as **ls** and **mkdir** can be found here.

`/sbin`
: Supervisor programs (binary files) that do require administrative rights to make full use of them, and need to be available at boot time. Commands such as **mount** and network configuration commands such as **ifconfig** can be found here. Note that normal users can use these commands, but for querying only, they won't be able to use the command to make changes to the system.

`/lib`
: Libraries that are needed when the system is booting (before `/usr` is mounted). It also includes kernel modules (device drivers and such).

`/usr`
: This is for static (non-changing) data. Most of the software is installed here. A system should be able to run well if this directory (which is often on its own filesystem or mounted from across the network) is mounted read-only.

  Historically, users home directories were stored in here, which may explain the peculiar name.

  This contains `bin`, `sbin` and `lib` directories, in addition to the following:

  `include`
  : Header files (`*.h`) that are included into programs used for compiling programs against libraries.

  `local`
  : Has a structure much like `/usr`, but is for software that the system administrator has compiled and installed. This directory is outside the scope of the package management system.

  `share`
  : Used for files that are shared amongst different system/processor architectures, such as documentation, pictures etc.

doc
Documentation for all the installed software. This does not include manual pages.

man
This is where the manual pages can be found, although you generally don't go there yourself, but use **man**.

src
Used to store software source code such as Linux kernel source code, though there is no requirement that source code by kept there.

/var
This is for variable (changing) data, such as databases, mail queues, logs, lock files and other things.

log
Log data produced by the system and its services.

pid
Process ID files so startup/shutdown scripts (*init scripts*) know which process to kill. PID files are generally written by *daemons* (background services) when they start.

mail
Location of email queues and mailboxes.

/etc
This is where configuration files are stored. Historically, its where anything else in the system was stored, which explains the name.

/dev
This is where device special files are stored. It may be a virtual filesystem, meaning the contents might not exist on disk, but are determined by the operating system device drivers and hardware management facilities (eg. USB insertions).

/tmp
A temporary directory that anyone can write to. The contents may be purged occasionally, or on system boot. If you want a more permanent location, use `/var/tmp`.

/proc, **/sys**
These are virtual directories that give you information about processes and the system.

/root
On Linux systems, this is the home directory of the root user.

/home
On Linux systems, this is where the home directories for the users can usually be found. It is often network mounted and possibly a symbolic link to elsewhere in the system.

1.3. Self-assessment

1. Read the ls(1) to familiarise yourself with the available options (don't try to remember them all, except `-l` (that's lowercase L, not I), `-R`, `-a` and `-h`).

Write down the **ls** command you would use to do the following; most of them will require the `-l` option and generally some others:

1. List the contents of your home directory recursively. You can use either `~` or `$HOME` to refer to your home directory from the shell.

Note

There will be a lot of output generated. You can pipe it to **less**, which will allow you to page through the output using the **Space** key and the **Page Up** and **Page Down** keys. Type **q** to exit **less**.

```
command | less
```

less is a *pager* which supercedes an older program called **more**.

2. List all files, including hidden files and directories, in your home directory, non-recursively. Hidden files or directories are those that begin with a dot, and are not usually shown in **ls** output.

3. List the contents of `/usr/bin` using **ls -l /usr/bin**. What is meant by the `foo -> bar` notation?

4. List the files in your home directory, with file sizes shown with human-friendly units (ie. bytes, kilobytes, megabytes, gigabytes).

5. List the info about a directory, *and only that directory*, without **ls** recursing into that directory. In other words, the permissions etc. of the directory itself, not the contents of the directory.

   The "size" of a directory in the **ls** listing does not tell you how much disk space the contents of the directory consume. Use **du -s `directory`** to measure that.

2. Filesystem Permissions

Classical Unix filesystem permissions are one of the things that are often criticised as being too coarse and inflexible, and rightly so. Compared to the access-control model of other systems, such as Novell Netware™ and Microsoft Windows™, they are very coarse, and often you need something more fine-grained.

Thankfully, Linux, like any modern Unix-like operating system, generally comes with the ability to attach an *Access Control List* to files and directories. There are also other access control models available for Linux, although they are not ubiquitous, such as SELinux. We won't be covering either of those in this paper.

If, after reading this lab, you are still unsure, you might try looking at Linux File Permission Confusion [http://www.hackinglinuxexposed.com/articles/20030417.html] and Linux File Permission Confusion part 2 [http://www.hackinglinuxexposed.com/articles/20030424.html], which are both excellent articles by Brian Hatch, the author of *Hacking Linux Exposed*.

2.1. Basic Unix Permissions

In the common permissions model, there are three permissions we are concerned with; the three permissions are read, write and execute. The meaning of each differs depending on whether we are talking about a file or a directory.

If you have the read permission, you can open a file for reading, or get a directory listing. The write permission means we can open a file for writing, or change directory entries (create a new file or subdirectory; rename a file or subdirectory; or delete a file or subdirectory). The

execute permission *on a file* means you can run the program; on script files, which are text files, this causes the script to be made runnable just like a regular program. On a directory, execute means you can traverse it (such as when you **cd** into it or using **find**), but it doesn't say if you may read the directory contents; additionally, write access to a directory generally requires execute also.

Each set of permissions (read, write and execute) can be applied to three things: the user a file or directory belongs to; the Group a file or directory belongs to; and finally to everyone else (others).

Let's look at some **ls -l** output to refer to.

```
$ ls -l ~/.bashrc
-rw-r-----  1 mal mal 516 2006-03-24 13:53 /home/mal/.bashrc
```

The very first character in the permission set (the leftmost column), says what kind of filesystem object it is, which we talked about earlier in the lab.

The next three characters (`rw-` for the `~/.bashrc`) are the permissions which affect the User (the person that owns the file), which in this example is the user `mal` (the 3rd field). So the user `mal` can read and write to that file.

The three characters after that (`r--`) are for people that are in the same group as the file (the group name is displayed in the 4th field of the above **ls -l** output). These people may only read the file. It is common practice on Linux systems for users to have their own group for local accounts. For your accounts, it will likely differ if they are not local accounts. In this example, the group is `mal`

The last three affect everyone else (Other people).

The order of checking access permissions is first User, then Group, and finally Other. Only one set of permission bits is queried once matched. For example, if you are matched by the Group, but its permission bits deny the group access, the Other bits would *not* be checked, even though they may allow the access.

Look at the below **ls** output[2]. What permissions will the user `jill` be granted on this directory? Assume that `jill` is not in the group `staff`.

```
You don't need to run this command, it's just an example.
$ ls -ld /home/cosc301/teaching
drwxrwx---  3 bob staff 4096 2007-01-30 16:23 /home/cosc301/teaching
```

Because the user `jill` is not the user `bob`, who owns this directory, nor is she a member of the `staff` group, so she only gets the permissions granted to the others, which have no permissions.

Now assume that the user `alice`, who *is* a member of `staff`, tries to list the contents of the directory, which requires a read priviledge. `alice` is not the user `bob`, so is not matched against the User permissions; but `alice` is a member of `staff`, so gets the permissions `rwx`, which grants read, write and execute. This means that listing the directory will work.

Changing Ownership and Permissions

The two commands for changing the owner and group of a file or directory are **chown** and **chgrp**. **chown** can be used to change the group at the same time. Both can be used to apply the changes you specify in a recursive manner.

[2]This has been slightly modified from the true permission set on this particular directory.

chown may only be used by the root user. The user invoking **chgrp** must belong to the specified group and be the owner of the file, or be the super-user (root).

Have a quick look at the manual pages for **chown** and **chgrp**. Take notice of how you can specify both user and group with **chown** by using the `user:group` syntax.

When we come to changing the permission (or *mode*) of a file or directory, there are two syntaxes you can use, *symbolic notation* and *octal notation*.

Symbolic notation is useful when you only need to change some of the permissions. Recall that we can apply permissions to the User, Group, and Others. Recall also that we have three permissions: read (`r`), write (`w`) and execute (`x`). We can grant these permissions in an absolute or relative manner, or we can revoke these permissions. We seperate the User, Group and Other sections with commas. The syntax is most easily learned with examples as below.

Note

In the following examples, `u` represents User, `g` for Group, and `o` for Other. Don't get confused and think that `o` stands for owner!

```
This command will create an empty file if it doesn't exist.
$ touch myfile
$ ls -l myfile
-rw-r--r--  1 mal mal 0 2007-02-17 20:44 myfile

Turn on the execute bit for the owning user.
$ chmod u+x myfile
$ ls -l myfile
-rwxr--r--  1 mal mal 0 2007-02-17 20:44 myfile

Remove read bit for all.
$ chmod -r myfile
$ ls -l myfile
--wx------  1 mal mal 0 2007-02-17 20:44 myfile

Grant read to User and Group.
$ chmod ug+r myfile
$ ls -l myfile
-rwxr-----  1 mal mal 0 2007-02-17 20:44 myfile

Set multiple parts.
$ chmod u=rw,g=rw,o=r myfile
$ ls -l myfile
-rw-rw-r--  1 mal mal 0 2007-02-17 20:44 myfile
```

On the other hand, if the octal notation is used, you cannot grant or revoke individual permissions. The notation is octal because each permission set for a User, Group or Other takes three bits, and so has a digit between 0 and 7 inclusive. Therefore, we need three octal digits to specify User, Group and Other.

For each digit, read is worth 4, write is worth 2 and execute is worth 1. For example, if we want to specify read and write, but not execute, that is 4+2=6.

```
$ ls -l myfile
-rw-rw-r--  1 mal mal 0 2007-02-17 20:44 myfile
The above file has octal permissions 664.

$ chmod 755 myfile
$ ls -l myfile
-rwxr-xr-x  1 mal mal 0 2007-02-17 20:44 myfile
```

The install command

If you want a command that combines the features of **cp**, **chown**, **chgrp**, **chmod** and **mkdir**, then you might like to look at the **install** command. You may find more details about **install** using **man 1 install**.

Initial Permissions (umask)

When a file is created, the permissions that it gets are 666 (for files), or 777 (for directories), minus[3] the **umask** value. This **umask** value turns off various bits in the permission set in order to make the default permissions for new files safe. **umask** settings are inherited in the process hierarchy. The value is generally octal 022, but can be changed in the shell using the **umask** command. With the **umask** value 022, files created would have permissions `rw-r--r--` (octal 644) by default, and directories would be `rwxr-xr-x` (octal 755).

2.2. Special Permissions

In addition to the standard permissions, there are the extended permissions. These are special bits that you as an administrator need to be aware of.

The special bits take up a forth octal digit (the most significant digit), so when you see an octal permission such as 2775, the special bits are contained in the digit 2.

Set User ID bit (SetUID)

On an executable *binary file*, this lets the program run as the person who *owns* the file, often root. In contrast, in a normal situation where the SetUID bit is not set, if *you* run **ls** that is owned by root, it would run with *your* permissions.

Note that using the SetUID bit can be dangerous, and must be carefully managed, since it is often used on root-owned system programs. Also the SetUID bit won't work for scripts on many Unix-like systems for security reasons.

If the SetUID bit is used on a root-owned system program, the program has the root privilege even if it is run by a normal user. Therefore, the program should drop its root privilege as early as it can, as long as the access that requires root privilege has been done. One such program is **ping**. It uses ICMP which requires root access. Another example is **passwd**, which needs to be able to modify files normal users cannot write to. Once those files are opened, the root privilege should be dropped in the program. This is an example where the *Least Privilege Principle* is applicable.

On a directory, SetUID has no effect. The SetUID bit has a value of 4. The bit can be turned on using **chmod u+s**. In **ls -l** output, a file with the SetUID bit will appear as follows:

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x  1 root root 37140 2010-01-27 06:09 /usr/bin/passwd
```

Here, we can tell that SetUID is on because of the `s` in the User execute position. Because the `s` is lower-case, we know that there is a `x` underneath it also, so the User execute bit is also set. However, if the upper-case `S` is shown instead, it means the User execute bit is not set.

[3] Actually, the Boolean bitwise operation AND and NOT are used, as in the formula mode = 666 AND NOT(umask) for files.

Set Group ID bit (SetGID)

Much like SetUID on files, SetGID files run with the priviledge of the *group* that owns the file.

This is often used for games, that need protected access to a shared scoreboard, without giving the user write access to the scoreboard.

On the other hand, if an entry is added to a directory with the SetGID bit set, a file becomes owned by the group that owns the directory when the file is created. In contrast, in a normal situation where the SetGID bit is not set on the directory, when a new file is created, the group of the file will be the *primary group* of the user who has put it there.

SetGID, along with **umask**, is useful for groupwork, or when you need to share responsibility between multiple people (such as a staff group). You can use **umask** to allow group read/write of files, while making all files created under the shared directory automatically belong to the group of the directory by setting the SetGID bit. In this way, all files created under the shared directory are readable and writable by anyone belonging to the group of the directory.

The SetGID bit has a value of 2. It can be set using **chmod g+s**. In **ls -l** output, it appears as follows:

```
$ ls -ld /var/mail
drwxrwsr-x 2 root mail 4096 2009-10-29 09:55 /var/mail
```

This time the `s` is in the Group execute position; so SetGID is set. The `s` is lower-case, so the Group execute bit is also set. Likewise, the upper-case `S` means the Group execute bit is not set.

Sticky bit

On files this has no effect under Linux[4]. But when set on a directory, it is very special. It means that only the user who created an item inside a sticky directory may delete it. This is the behaviour of the `/tmp` directory.

The `/tmp` directory has an octal mode of 1777. This means that anyone can add or delete entries into the `/tmp` directory. If the sticky bit was not set (ie, the overall mode would be 0777), then the user `bob` could delete the user `alice`'s files in `/tmp`. But with the sticky bit, this could not happen.

The sticky bit has a value of 1. It can be set using **chmod +t**. In **ls -l** output, it appears as follows:

```
$ ls -ld /tmp
drwxrwxrwt  5 root root 1160 2007-02-18 00:36 /tmp
```

The `t` is shown in the Other execute position. The `t` is lower-case, so the Other execute bit is also set. Likewise, the upper-case `T` means the Other execute bit is not set.

2.3. Self-assessment

1. Fill in the blanks to convert between octal and symbolic notation; all but the last two you would be likely to find on a real system.

[4]Originally on UNIX systems, it told the operating system to keep a file loaded in memory. Today it is not needed due to improvements in operating system design.

Octal	Symbolic
644	
	rwxr-xr-x
1777	
	rwsr-xr-x
3070	
	---rwS---

2. If you look around the typical Unix filesystem long enough, you find some unusual permissions. Take for example the permissions that are classically set on the *files* in `/usr/games/`, and the permissions on some of the *files* in `/var/games/`.

```
You don't need to run this as it doesn't apply to Client-1
$ ls -l /usr/games/
total 2772
…
-rwxr-xr-x 1 root root    2461 2009-09-22 21:23 glchess   SetGID is not set
-r-xr-sr-x 1 root games 142716 2009-09-22 21:25 glines    SetGID is set
…
$ ls -l /var/games/
total 0
              glchess has no shared scoreboard files
-rw-rw-r-- 1 root games 0 2009-10-29 10:02 glines.Large.scores
-rw-rw-r-- 1 root games 0 2009-10-29 10:02 glines.Medium.scores
-rw-rw-r-- 1 root games 0 2009-10-29 10:02 glines.Small.scores
…
```

The files in `/var/games/` are scoreboard files, which need to be updateable by the games when run as different users. However, we don't want users to be able to edit them by themselves, only by the games that the user is running. Looking at these permissions, describe how this works. Hint: users are not meant to be in the 'games' group in order to play games; indeed that would be rather bad.

3. Assume you see a directory which **ls -l** describes as `drwxrwsr-x`. Write a paragraph to completely describe the effect of this permission set.

You should consult ls(1) for some of the subtleties of the printed permissions with regard to special bits and execute bits, if you haven't already.

4. Give a typical default umask value and describe its effect.

3. Archival and Backup

Archival and Backup are two distinct activities, though the terms are used somewhat interchangably. Archival is long-term, often indefinite, while backups are often rotated, only keeping material for a certain window of time, while archiving information is not intended to ever be deleted. Backups are generally done according to a schedule, such as a daily, while archival is often done on an as-needed basis, such as when creating a software release (eg. `foo-1.2.3.tar.gz` to upload to a website) or when you want to move your precious digital photo collection from your hard-disk onto DVD.

The media and method you use will depend on whether you are archiving or backing up. Backups should ideally be able to be done unattended to a remote location, so backing up over the network to a cheap but reliable hard disk is a useful way of handling this.

When archiving, removable media is generally the best way to go, as it scales better. However, any media has aging issues, and this can be prevalent for a number of optical media, such as recordable CDs and DVDs, so you need to occasionally check the integrity of your archives to ensure the data is still intact, and pay attention to things like the brand and type of disc, and how they are stored.

Another option is to rely on a cloud provider. There are various options depending on the scale of the backup/archival needs. Services such as Google Drive, Apple iCloud, Microsoft OneDrive are useful for an individual's documents. Some offerings, Google Suite (their Docs, Sheets, Drive, Gmail etc. for business) allow you to manage a small business and provide backup/archival solutions. Others (Google Cloud Storage, Amazon Glacier) allow you to store vast amounts of data very cheaply but cost more to retrieve[5].

USB Flash drives, DVD-ROM, USB HDDs or the cloud?

It's hard to have long-term confidence in DVDs, particularly those bought in big spindles, given the move towards removing of optical media drives from computers. USB flash drives are interesting devices; they do have a significant lifetime issue though, but only in terms of how often you write to them.

USB flash drives (unlike SD cards you find in most digital cameras) don't have a physical write-protection switch, which creates an issue regarding accidental deletion and viruses.

USB hard drives are an option, and behave like a flash drive but have the benefits and drawbacks of traditional spinning hard drives.

Recently developments in online infrastructure allow you to put a (relatively) large amount of data online for free. Services such as Google Drive, Dropbox and iCloud all offer storage of personal files, Amazon Glacier is more catered towards storage of large files (and a slow retrieval time). However, there are jurisdiction, security, and privacy issues with this approach, especially when medical records are concerned.

It may be better to "hedge your bets" and put your precious things in as many places as possible; then you can store at least one copy off-site (perhaps mailing a flash drive to a family member for safe keeping--and is often the only practical way to deal with large data sets).

Archiving and backup are otherwise technically very similar, and many tools can be used for both activities. To illustrate that point, any single archive should be usable by itself, whereas a backup *might* record changes only from a previous backup on which it becomes dependent.

Note

For the rest of the section, the terms "archive" and "backup" can be treated as synonymous. Just beware that there is a distinction, but that it only really matters when considering issues such as media, storage, etc.

In the Unix world, backup is often done using the **tar** command to create the archive/backup file, which is then burned to disc or stored elsewhere. **tar** was originally meant for use with

[5]Under an old pricing scheme, an Amazon Glacier user was surprised by the bill [https://medium.com/@karppinen/how-i-ended-up-paying-150-for-a-single-60gb-download-from-amazon-glacier-6cb77b288c3e]! Do note the pricing scheme has been changed since. Moral of the story is to plan ahead!

magnetic tapes as a destination medium. Tapes are still used today, but are quite expensive compared to hard disks.

3.1. Using tar

tar is one of those really old Unix programs, and is still quite useful for its task. The purpose of **tar** is to take a directory or a list of files, and package it as a single file. It is then usually passed to a compressor, commonly **gzip** or **bzip2**. The process is reversed to get the files out. If you have used other compression programs, such as **zip** or **jar**, please realise that those tools perform *both* the archiving and compression.

tar has three basic modes: create, list, and extract. **tar** archives, sometimes called *tarballs*[6], often have a `.tar.gz` or `.tgz` extension, which means they've also been compressed with **gzip**. There are also **bzip2** compressed files, which are slightly smaller, but take longer to process. **bzip2** compressed files have a **.tar.bz2**, or **.tbz2** extension.

To create an archive, we use the `c` option. You can specify an output file using the `f` option. `v` is for verbose output[7]. `z` passes the output (or input) through **gzip** (`j` for **bzip2** instead) to (de)compress the data.

```
This is generally a bad way of doing it, we'll see why shortly
# tar -zcvf /tmp/logs-bad.tgz /var/log
tar: Removing leading `/' from member names
/var/log/
/var/log/syslog
/var/log/apt/
/var/log/apt/history.log
/var/log/apt/term.log
/var/log/Xorg.0.log
…

Instead, cd to where you want to go first
then operate on the current directory (.)
$ cd /var/log
# tar -zcvf /tmp/logs-good.tgz .
./
./syslog
./apt/
./apt/history.log
./apt/term.log
./Xorg.0.log
…
```

We can list the contents of an tar archive using the `t` option. Note that the `v` option when listing will give output similar to **ls -l**, whereas without `v`, the output will be like plain **ls**. Additionally the command **head -5** only shows the first five lines of the output.

```
This one is listing the archive we made using the "bad" method.
Notice the path var/log contained inside the archive.
$ tar -ztf /tmp/logs-bad.tgz | head -5
var/log/
var/log/syslog
var/log/apt/
var/log/apt/history.log
var/log/apt/term.log
```

[6] To the disgust of many people.
[7] Not verify, as some mistakenly believe.

```
In the “good” method, we don’t have that problem.
We could unpack it anywhere and not get var/log
$ tar -ztf /tmp/logs-good.tgz | head -5
./
./syslog
./apt/
./apt/history.log
./apt/term.log

This shows the verbose output.
$ tar -ztvf /tmp/logs-good.tgz | head -5
drwxr-xr-x root/root         0 2010-11-19 09:31 ./
-rw-r----- syslog/adm     4139 2010-11-23 13:30 ./syslog
drwxr-xr-x root/root         0 2010-11-11 13:16 ./apt/
-rw-r--r-- root/root     55950 2010-11-11 16:21 ./apt/history.log
-rw------- root/root     94518 2010-11-11 16:21 ./apt/term.log
```

Finally, we can extract the contents using the x option. Note that if you unpack the archive as root, the Owner and Group will be set to those in the archive; however, if you unpack the archive as a normal user, they would end up with all files and directories belonging to you and your group. If you have to unpack source as root, you might like to use **chown -R root:root** to reset the User and Group on all files to the root user; otherwise, you might end up giving write permissions of some important files to some other user unintentionally, which could be disastrous. Another thing to be wary of is the permissions you give to the archive. You don’t want to allow normal users to read a backup of everyone’s home directories.

```
Make a place to extract to…
$ mkdir /tmp/restore
$ cd /tmp/restore

Time to illustrate why the “bad” method is annoying…
$ tar -zxvf /tmp/logs-bad.tgz
var/log/
var/log/syslog
var/log/apt/
var/log/apt/history.log
var/log/apt/term.log
var/log/Xorg.0.log
…

…what did we get?
$ ls -R .
.:
var

./var:
log

./var/log:
apparmor      dmesg           kern.log.1         syslog
apt           dmesg.0         lastlog            syslog.1
aptitude      dmesg.1.gz      lpr.log            udev

…bother! Everything is in var/log/FOO
and I wanted just ./FOO. Let’s start again…

$ cd ..
$ rm -rf restore
$ mkdir restore
$ cd restore

…this time using using the “good” archive.
$ tar -zxvf /tmp/logs-good.tgz
./
./syslog
```

```
./apt/
./apt/history.log
./apt/term.log
./Xorg.0.log
…
$ ls -R .
.:
apparmor      dmesg           kern.log.1         syslog
apt           dmesg.0         lastlog            syslog.1
aptitude      dmesg.1.gz      lpr.log            udev
auth.log      dmesg.2.gz      lpr.log.1          ufw.log

Great, that's more like it!
```

If we only want to extract certain members of an archive, such as a partial restore, you can list those members (*precisely* as they appear in the listing output), at the end of the **tar** command. Here's an example extracting just the `./apt/` directory:

```
$ cd /tmp
$ rm -rf restore
$ mkdir restore
$ cd restore
$ tar -zxvf /tmp/logs-good.tgz ./apt/
./apt/
./apt/history.log
./apt/term.log
```

3.2. Investigating Backups

Let us first consider some of the things we would like to have in a modern backup system.

- Remote, to protect against localised disaster, such as flooding. It should at least be in a different building. The amount of data would suggest somewhere else on the LAN. Off-site is better.

- Cheap to implement. Largest consideration here is media, drives, software, and network performance and possibly charges. Hard disks are cheap, large, and fairly reliable. Good for backups, and any host with sufficient space and able to sustain a high amount of traffic at off-peak times should suffice.

- Plentiful snapshots. You want to be able to restore at any point in time, and be able to do so with minimal loss and fuss. Storing snapshots can be done in a variety of ways, some of which are quite cheap. It can even be cheap enough to provide a snapshot every 30 minutes.

- Easy to restore. For easy, you can also read *fast*. This can also mean that it is easy to do a partial restore, often just a single file. Ideally, the user might be able to perform this operation themselves.

- Secure. The transmission must be secure, as well as how it is stored.

- Maintain file meta-data, such as access control lists. On Mac OS X for example, you can set a particular colour for a file or directory, and enter comments. It would be good if we don't have to backup to the same type of operating system or file system as the one we are backing from.

- Selection of items to include/exclude. There are a *lot* of things that don't really need backed up. But on the other hand, it can be feasible to include everything up in case you forget something important.

- Suitable for very large files, such as virtual machine disk images: you wouldn't want a small change in one area of the file to cause the entire very large file to be backed up again.

A system administrator, perhaps as part of a wider site-policy, might often use a commercial piece of software for peace of mind.

They might instead write a fairly complex shell-script which determines a list of files that need backed up, which is then fed into **tar**, as **tar** is more appropriate as a backup engine, not a "differencing" engine.

rsync, a file synchronisation tool, has some very useful functionality for doing backups, but it also has some issues that make it incomplete as a backup tool. It tries to transmit only what has changed, but it isn't really designed for backups. Other tools can use **rsync**, or the underlying mechanisms, as part of a backup solution. One such product is called **rdiff-backup**.

We won't be using **rdiff-backup** today, but you will be finding out about its feature set.

3.3. Self-assessment

1. Create a backup of Mal's home directory (`/home/mal`), storing it in `/tmp/mal-backup.tar.gz`. Set the permissions such that only Mal can read it. Ensure that it does not have `home/mal/…` at the start of each entry in the archive, as done in the previous "bad" method.

2. The above command likely has a security problem, in that the archive does not have a suitable mode when it is being created. This creates a window of opportunity for someone to open the file and read the contents before **tar** has finished and you have fixed the permissions.

How can you fix this problem? (Hint: use **umask**).

3. Part of a diet of the system administrator is to look at various products and evaluate their usefulness to solve particular situations. Have a look at the homepage of rdiff-backup [http://www.nongnu.org/rdiff-backup/]. Compare the features of rdiff-backup to the features we desire, listed above, marking each with a tick, cross or question mark, for "meets requirement", "does not meet requirement", or "unsure" respectively. Do you think rdiff-backup meets our stated needs?

3.4. Rdiff-backup Example

rdiff-backup is quite useful for most purposes (not for all though, for example it doesn't handle sparse files well), which generally includes sending them to a remote system over SSH every night. So here is one example that shows a number of the features of **rdiff-backup**. You WILL want to edit this to suit other systems.

Do not run this script. It is just an example.

```
#!/bin/bash
#
# Backup using rdiff-backup.
#

rdiff-backup \
```

```
  --exclude-if-present NOT_BACKED_UP \
  --include /home \
  --include /etc \
  --include /var/mail \
  --include /var/www \
  --include /var/log \
  --include /usr/local \
  --include /usr/lib/cgi-bin \
  --include /var/lib/dpkg \
  --exclude '**' \
  --remote-schema \
     '/usr/bin/ssh -o BatchMode=yes -C -i/path/to/.ssh/backup %s' \
  / \
  remote_user@remote_host::remote_path
```

Let us walk through this script briefly. First, to help make it clear to users, and to enable users to say what *doesn't* get backed up, we ignore any directory that has a file called `NOT_BACKED_UP` inside it. This also enables greater maintainability of this script (We could, for example, use it a template for other systems).

In this particular example, because all the software is stock Debian packages, we only want to include those parts that can't easily be reinstalled, plus any configuration files, plus any other data that might need to be backed up (We sure hope we haven't missed anything; it's generally much easier just to backup everything). Because this is a Debian system, we have backed up the directory that contains information about all the installed packages (`/var/lib/dpkg/`) which will enable us to reinstall without too much fuss.

By default, we exclude anything else that doesn't match our previous include or exclude directives; see the manual for what ** means.

Because we are backing up over SSH (which we shall learn about much later in this course), we have changed the template (schema) that **rdiff-backup** uses to run the **ssh** command. We have enabled batch operation (don't bother every trying to ask for a password), enabled compression, and pointed to a particular private key we want to use just for backing up.

We then say where we want to start our backup from (in this case, the root directory `/`, and where we want the backup to be stored.

There is a little more to this, to do with SSH public-key authentication and saying which command gets run on the server. This is configured in the remote user's `~/.ssh/authorized_keys` and looks something like the following. Don't worry if you don't understand it, as you'll learn about SSH in a later lab, and we won't be doing any assessment in this section.

```
command="/path/to/rdiff-backup-1.1.14 --server",from="client-host.domain" public key
```