BACKING UP LINUX AND OTHER UNIX(-LIKE) SYSTEMS

There are two kinds of people: those who do regular backups and those who never had a hard drive failure — Unknown.

1. Introduction

The topic of doing backups of a (live) Un*x (mostly Linux) system regularly comes up on Linux mailing lists and forums and invariably the advice to simply do `tar cvfz backup.tgz /bin /boot /etc ...` is given. Unfortunately, a good backup takes more effort than that. In this article I will outline a great deal (but not necessarily all) of the pitfalls and details you will have to be watchful of when making backups.

Note that this is not an application how-to, so you should not use the given examples verbatim, nor does it give an exhaustive list of backup programs and examples. It also doesn't give step-by-step instructions. It is meant to create awareness for people who already have a general understanding of Un*x systems. Reading all of the documentation of the tool itself is and remains important, for it may make you think of things you wouldn't otherwise have considered.

Also note that this article mostly describes the process of making backups to an external device or location. If data protection is important to you, I also highly recommend using RAID. While RAID offers no protection against fires, earthquakes, data corruption or humans, it does offer protection against failing disks. It has saved me more than once. Additionally, I'd advice you to consider using a UPS.

Although my personal experience is limited to Linux, the issues I'll discuss should (could) work as well on all or most Un*x systems.

---

2. A backup is more than data

A proper backup contains far more than just your data. It also contains the data about your data: the meta data. It will also contain all the specific file system attributes and special devices to make your operating system work. It is vital that the target medium/software of your backup supports all these. As an extreme example, you shouldn't backup an Ext3 file system (standard file system on Linux machines) on FAT32/FAT16 (ancient Microsoft file systems, still used on USB sticks and similair devices, even though devices like USB sticks can be formatted with any choice file system, of course). This chapter discusses these meta data and special files.

2.1. File meta data

On an Ext3 partition, the meta data of a file consists of: file modification time, inode modification time, last access time, user and group ID's and permissions. When you have extended attributes, this can be a whole lot more, most notably Access Control List information. You need to backup as much of this as you can. Obviously, when you don't store *and* restore proper permissions, you can end up with a buggered installation. This is even true for something simple as the mtime. The Gentoo Linux distribution, for example, uses mtimes to determine if files belong to the installation of a certain package, or if they have been replaced with something new. If you don't restore proper mtimes, the package management will be completely shot.

It depends very much on the software used what you should do to include all this information. You need to make sure that you're backing it up *and* restoring it. When using tar, this only goes right by default when you're root (although I haven't tested how it deals with extended attributes and ACL's).

Ownership information can be stored in two ways: numerically or textually. A lot of backup programs find it user friendly to use textual matches, but for making backups of an entire system, this is very undesirable. It's very likely that you will restore the backup using some kind of live CD, while the original backup was made on the system you're backing up itself. On restoring the backup, files belonging to user *bin* will be given the ID on the file system based on the */etc/passwd* file of the live CD. If this ID is 2 for example, but ID 2 is user *daemon* on the system you are restoring, all the files that used to belong to *bin*, now belong to *daemon*. Therefore, *always* store owner information numericly. Tar has the *--numeric-owner* option for that. Rdiff-backup has the *--preserve-numerical-ids* option, added to version 1.1.0 per my request. Dar will never support textual matches. I discussed the issue with the author, and he agreed with my reasoning.

Certain backup software have the ability to set back atimes after files are read when doing a backup (dar and tar for example). The purpose of this is to leave everything behind exactly as it was. One should be very careful with this behavior, because setting back the atime, changes the ctime. There is no way to change this, because ctimes cannot be set artificially. According to the dar man page, Leafnode NNTP caching software relies on the preserveration of atimes, but normally the necessity for setting back the atime is very rare. I would like to add that in my opinion, any program which relies on preservation of atimes, is flawed. Atimes can be changed very arbitrarily, even by users who have no write permissions on the file. Also, automatic indexing software like Beagle can cause atimes to change. Futhermore, a change in ctime can trip certain security software. As I said, ctimes cannot be set artificially, meaning that if a file has a new ctime but an identical mtime, since it was last checked, it's possibly replaced by a different file, usually one that is part of a rootkit. Therefore, don't preserve atimes unless you know what you are doing. Dar preserves the atime per default. This behavior has been changed in the CVS repository, and is likely to be released in version 2.4.0. Until the default has changed, use the --*alter=atime* option.

2.2. Special files

2.2.1. Links

Links come in two forms: symbolic links and hard links. A symbolic link is simply a reference to another path. A hard link is an additional reference to an inode.

For preserving symbolic links, all you have to do, is make sure the backup application stores the link information, instead of the file it links to. This is not always the default, so be careful.

Hard links require a bit more attention. As I said, a hard link is basically a second (or third, or fourth...) name for a file. When you have a file *A*, link it to *B*, you have what appears to be two files. If these two files are 1 GB big, it will only consume 1 GB of space, even though applications can claim they take up 2 GB. Because the file *B* is not just a link to *A*, but a second name for it, you can safely delete *A*. The file *B* will still exist after the deletion of*A*.

Most backup applications have support for hard links, *but only when they are all in the same source tree*. This means, if you copy */bin*, */etc*, */usr*... with separate *cp -*

a commands, hard link information is not detected and copied. Because hard links cannot span accross file systems, suppling one backup and restore command per partition will work fine. For example, if you have your */home* on a separate partition, you could make a separate archive for */* with */home* excluded, and another archive of */home* alone. If you choose to make one archive with all mount points included, you may have to take special actions to make sure that upon restoration, the data is restored to the proper partitions. If the program in question doesn't complain about existing directories, creating the mount points in the new file system with identical names as before should do it. Otherwise, restoring to one partition first and copying parts to another partition with *cp -a* later, will most likely work for you. Don't use *mv* to move the data. You can imagine what will happen if the command fails half-way...

Linux, and all Unix machines, use hard links extensively, so make 100% sure you maintain link integrity. Rsync for instance, needs the special flag *--hard-links*, even when you've also specified *--archive* (as the man page says, *--archive* still lacks *--hard-links*, *--acls* and *--xattrs*).

2.2.2. Sparse files

A sparse file is a file of which the zeros aren't stored on disk as zeros, but are not allocated. Therefore, it's possible that a 1 GB file with a lot of empty space takes up only 1 MB, for example. A program which uses sparse files is Azureus, a Bittorrent download client.

Support for sparse files varies widely in backup software. When you use a program which doesn't support sparse files, the file is read in the regular way. The data of the file remains the same, but it can take up a lot more space. You therefore have to be careful, because it's possible a backup won't fit on the disk anymore when you restore it, since the sparse files are created as normal ones.

For Bittorrent download files, it's not really an issue that they're restored as normal files, because they will be filled up with data as the download progresses anyway. But, if you have a lot of sparse files which should remain sparse, selecting a backup program that supports sparse files is essential. Note however, that when a file is determined as sparse, the copy is not sparse in the exact same way and places as the original, because this information can't be retrieved. Instead, it's created as a new sparse file, where non allocated space is used as the backup tools sees fit. This shouldn't be a problem however; I can't think of a situation where this matters.

2.2.3. Others

There are some other special files, like FIFO's, named pipes, block devices, etc. These are pretty unremarkable, and most applications know how to deal with them. However, you do have to supply the correct option(s). Using*cp* without *-a* on a named pipe for example, will try to copy the data of the pipe, and not recreate the pipe.

There is also a special kind of directory: *lost+found* (part of Ext2/3/4). This is actually not a directory at all, and should not be made with *mkdir*. Instead, use *mklost+found*. In case you were wondering, lost+found is used to store files "recovered" with e2fsck when the file system is damaged.

---

3. What to exclude

To save space on your backup medium, you can choose not to back up certain locations. For my Gentoo Linux system, these include */usr/portage/* and */var/tmp/portage*.

There are also special file systems mounted within the root file system, which are created dynamically upon boot, and shouldn't be backed up. For my system, these are */sys*, */proc*, */lost+found*, */media* (which contains only dynamically created directories for removable media) and */dev* (because I use udev). I also exclude */mnt*, but the need to backup */mnt* can vary from system to system.

---

4. Application data

When making a backup of a live system, you have to be mindful of programs which can change their data files during the backup. A good example is a database, such as MySQL or PostreSQL, but also the data of e-mail programs (mbox files are more sensitive than maildir). The data files (stored somewhere in */var* usually) can undergo change on a live system. This can be because of normal transactions, or automatic database clean-up. Never trust these data files of a running database, LDAP server, Subversion repository, or whatever similar software you may use.

If shutting down such software before the backup is not an option, schedule jobs which periodicly dump the data of the database (using *pg_dump* for Postgresql, *slapcat* for OpenLDAP, *svnadmin dump* or *svn-backup-dumps*for Subversion, etc) into (date stamped) files. These files are then backed up and you should be safe. Use the software's native dump utility whenever possible, as pg_dump and slapcat are for Postgresql and OpenLDAP respectively.

Doing this (scheduled dumps) is always a good idea, regardless of the situation. Should the data suddenly get corrupted, you still have dumps of past situations, so not everything is lost. And, when you dump them somewhere in the local file system, you don't have the hassle of searching through your backups when the need arrises to restore the database (or other application data).

Source : http://www.halfgaar.net/backing-up-unix