Easy Automated Snapshot-Style Backups with Rsync

41-52 minutes

---

page last modified 2004.01.04

Updates: As of `rsync-2.5.6`, the `--link-dest` option is now standard! That can be used instead of the separate `cp -al` and `rsync` stages, and it eliminates the ownerships/permissions bug. I now recommend using it. Also, I'm proud to report this article is mentioned in Linux Server Hacks, a new (and very good, in my opinion) O'Reilly book by compiled by Rob Flickenger.

Contents

Abstract

This document describes a method for generating automatic rotating "snapshot"-style backups on a Unix-based system, with specific examples drawn from the author's GNU/Linux experience. Snapshot backups are a feature of some high-end industrial file servers; they create the *illusion* of multiple, full backups per day without the space or processing overhead. All of the snapshots are read-only, and are accessible directly by users as special system directories. It is often possible to store several hours, days, and even weeks' worth of snapshots with slightly more than 2x storage. This method, while not as space-efficient as some of the proprietary technologies (which, using special copy-on-write filesystems, can operate on slightly more than 1x storage), makes use of only standard file utilities and the common rsync program, which is installed by default on most Linux distributions. Properly configured, the method can also protect against hard disk failure, root compromises, or even back up a network of heterogeneous desktops automatically.

Motivation

Note: what follows is the original sgvlug DEVSIG announcement.

Ever accidentally delete or overwrite a file you were working on? Ever lose data due to hard-disk failure? Or maybe you export shares to your windows-using friends--who proceed to get outlook viruses that twiddle a digit or two in all of their .xls files. Wouldn't it be nice if there were a `/snapshot` directory that you could go back to, which had complete images of the file system at semi-hourly intervals all day, then daily snapshots back a few days, and maybe a weekly snapshot too? What if every user could just go into that magical directory and copy deleted or overwritten files back into "reality", from the snapshot of choice, without any help from you? And what if that `/snapshot` directory were read-only, like a CD-ROM, so that nothing could touch it (except maybe root, but even then not directly)?

Best of all, what if you could make all of that happen automatically, using *only one extra, slightly-larger, hard disk*? (Or one extra partition, which would protect against all of the above except disk failure).

In my lab, we have a proprietary NetApp file server which provides that sort of functionality to the end-users. It provides a lot of other things too, but it cost as much as a luxury SUV. It's quite appropriate for our heavy-use research lab, but it would be overkill for a home or small-office environment. But that doesn't mean small-time users have to do without!

I'll show you how I configured automatic, rotating snapshots on my $80 used Linux desktop machine (which is also a file, web, and mail server) using only a couple of one-page scripts and a few standard Linux utilities that you probably already have.

I'll also propose a related strategy which employs one (or two, for the wisely paranoid) extra low-end machines for a complete, responsible, automated backup strategy that eliminates tapes and manual labor and makes restoring files as easy as "cp".

Using `rsync` to make a backup

The `rsync` utility is a very well-known piece of GPL'd software, written originally by Andrew Tridgell and Paul Mackerras. If you have a common Linux or UNIX variant, then you probably already have it installed; if not, you can download the source code from rsync.samba.org. Rsync's specialty is efficiently synchronizing file trees across a network, but it works fine on a single machine too.

Basics

Suppose you have a directory called `source`, and you want to back it up into the directory `destination`. To accomplish that, you'd use:

```
rsync -a source/ destination/
```

(Note: I usually also add the `-v` (verbose) flag too so that `rsync` tells me what it's doing). This command is equivalent to:

```
cp -a source/. destination/
```

except that it's much more efficient if there are only a few differences.

Just to whet your appetite, here's a way to do the same thing as in the example above, but with `destination` on a remote machine, over a secure shell:

```
rsync -a -e ssh source/ username@remotemachine.com:/path/to/destination/
```

Trailing Slashes *Do* Matter...Sometimes

This isn't really an article about `rsync`, but I would like to take a momentary detour to clarify one potentially confusing detail about its use. You may be accustomed to commands that don't care about trailing slashes. For example, if `a` and `b` are two directories, then `cp -a a b` is equivalent to `cp -a a/ b/`. However, `rsync` *does* care about the trailing slash, but only on the source argument. For example, let `a` and `b` be two directories, with the file `foo` initially inside directory `a`. Then this command:

```
rsync -a a b
```

produces `b/a/foo`, whereas this command:

```
rsync -a a/ b
```

produces `b/foo`. The presence or absence of a trailing slash on the destination argument (`b`, in this case) has no effect.

Using the `--delete` flag

If a file was originally in both `source/` and `destination/` (from an earlier `rsync`, for example), and you delete it from `source/`, you probably want it to be deleted from `destination/` on the next `rsync`. However, the default behavior is to leave the copy at `destination/` in place. Assuming you want `rsync` to delete any file from `destination/` that is not in `source/`, you'll need to use the `--delete` flag:

```
rsync -a --delete source/ destination/
```

Be lazy: use `cron`

One of the toughest obstacles to a good backup strategy is human nature; if there's any work involved, there's a good chance backups won't happen. (Witness, for example, how rarely my roommate's home PC was backed up before I created this system). Fortunately, there's a way to harness human laziness: make `cron` do the work.

To run the rsync-with-backup command from the previous section every morning at 4:20 AM, for example, edit the root `cron` table: (as root)

```
crontab -e
```

Then add the following line:

```
20 4 * * * rsync -a --delete source/ destination/
```

Finally, save the file and exit. The backup will happen every morning at precisely 4:20 AM, and root will receive the output by email. Don't copy that example verbatim, though; you should use full path names (such as `/usr/bin/rsync` and `/home/source/`) to remove any ambiguity.

Incremental backups with `rsync`

Since making a full copy of a large filesystem can be a time-consuming and expensive process, it is common to make full backups only once a week or once a month, and store only changes on the other days. These are called "incremental" backups, and are supported by the venerable old `dump` and `tar` utilities, along with many others.

However, you don't have to use tape as your backup medium; it is both possible and vastly more efficient to perform incremental backups with `rsync`.

The most common way to do this is by using the `rsync -b --backup-dir=` combination. I have seen examples of that usage here, but I won't discuss it further, because there is a better way. If you're not familiar with hard links, though, you should first start with the following review.

Review of hard links

We usually think of a file's name as being the file itself, but really the name is a *hard link*. A given file can have more than one hard link to itself--for example, a directory has at least two hard links: the directory name and `.` (for when you're inside it). It also has one hard link from each of its sub-directories (the `..` file inside each one). If you have the `stat` utility installed on your machine, you can find out how many hard links a file has (along with a bunch of other information) with the command:

```
stat filename
```

Hard links aren't just for directories--you can create more than one link to a regular file too. For example, if you have the file `a`, you can make a link called `b`:

```
ln a b
```

Now, a and b are two names for the same file, as you can verify by seeing that they reside at the same inode (the inode number will be different on your machine):

```
ls -i a
  232177 a
ls -i b
  232177 b
```

So `ln a b` is roughly equivalent to `cp a b`, but there are several important differences:

1. The contents of the file are only stored once, so you don't use twice the space.
2. If you change a, you're changing b, and vice-versa.
3. If you change the permissions or ownership of a, you're changing those of b as well, and vice-versa.
4. If you overwrite a by copying a third file on top of it, you will also overwrite b, unless you tell `cp` to unlink before overwriting. You do this by running `cp` with the `--remove-destination` flag. **Notice that `rsync` always unlinks before overwriting!!**. Note, added 2002.Apr.10: the previous statement applies to changes in the file contents only, not permissions or ownership.

But this raises an interesting question. What happens if you `rm` one of the links? The answer is that `rm` is a bit of a misnomer; it doesn't really remove a file, it just removes that one link to it. A file's contents aren't truly removed until the number of links to it reaches zero. In a moment, we're going to make use of that fact, but first, here's a word about `cp`.

Using `cp -al`

In the previous section, it was mentioned that hard-linking a file is similar to copying it. It should come as no surprise, then, that the standard GNU coreutils `cp` command comes with a `-l` flag that causes it to create (hard) links instead of copies (it doesn't hard-link directories, though, which is good; you might want to think about why that is). Another handy switch for the `cp` command is `-a` (archive), which causes it to recurse through directories and preserve file owners, timestamps, and access permissions.

Together, the combination `cp -al` makes *what appears to be* a full copy of a directory tree, but is really just an illusion that takes almost no space. If we restrict operations on the copy to adding or removing (unlinking) files--i.e., never changing one in place--then the illusion of a full copy is complete. To the end-user, the only differences are that the illusion-copy takes almost no disk space and almost no time to generate.

2002.05.15: Portability tip: If you don't have GNU `cp` installed (if you're using a different flavor of *nix, for example), you can use `find` and `cpio` instead. Simply replace `cp -al a b` with `cd a && find . -print | cpio -dpl ../b`. Thanks to Brage Førland for that tip.

Putting it all together

We can combine `rsync` and `cp -al` to create what appear to be multiple full backups of a filesystem without taking multiple disks' worth of space. Here's how, in a nutshell:

```
rm -rf backup.3
mv backup.2 backup.3
mv backup.1 backup.2
```

```
cp -al backup.0 backup.1
rsync -a --delete source_directory/  backup.0/
```

If the above commands are run once every day, then `backup.0`, `backup.1`, `backup.2`, and `backup.3` will appear to each be a full backup of `source_directory/` as it appeared today, yesterday, two days ago, and three days ago, respectively--complete, except that permissions and ownerships in old snapshots will get their most recent values (thanks to J.W. Schultz for pointing this out). In reality, the extra storage will be equal to the current size of `source_directory/` plus the total size of the changes over the last three days--exactly the same space that a full plus daily incremental backup with `dump` or `tar` would have taken.

Update (2003.04.23): As of `rsync-2.5.6`, the `--link-dest` flag is now standard. Instead of the separate `cp -al` and `rsync` lines above, you may now write:

```
mv backup.0 backup.1
rsync -a --delete --link-dest=../backup.1 source_directory/  backup.0/
```

This method is preferred, since it preserves original permissions and ownerships in the backup. However, be sure to test it--as of this writing some users are still having trouble getting `--link-dest` to work properly. Make sure you use version 2.5.7 or later.

Update (2003.05.02): John Pelan writes in to suggest recycling the oldest snapshot instead of recursively removing and then re-creating it. This should make the process go faster, especially if your file tree is very large:

```
mv backup.3 backup.tmp
mv backup.2 backup.3
mv backup.1 backup.2
mv backup.0 backup.1
mv backup.tmp backup.0
cp -al backup.1/. backup.0
rsync -a --delete source_directory/ backup.0/
```

2003.06.02: OOPS! Rsync's link-dest option does not play well with J. Pelan's suggestion--the approach I previously had written above will result in unnecessarily large storage, because old files in backup.0 will get replaced and not linked. Please only use Dr. Pelan's directory recycling if you use the separate `cp -al` step; if you plan to use `--link-dest`, start with backup.0 empty and pristine. Apologies to anyone I've misled on this issue. Thanks to Kevin Everets for pointing out the discrepancy to me, and to J.W. Schultz for clarifying `--link-dest`'s behavior. Also note that I haven't fully tested the approach written above; if you have, please let me know. Until then, caveat emptor!

I'm used to `dump` or `tar`! This seems backward!

The `dump` and `tar` utilities were originally designed to write to tape media, which can only access files in a certain order. If you're used to their style of incremental backup, `rsync` might seem backward. I hope that the following example will help make the differences clearer.

Suppose that on a particular system, backups were done on Monday night, Tuesday night, and Wednesday night, and now it's Thursday.

With dump or `tar`, the Monday backup is the big ("full") one. It contains everything in the filesystem being backed up. The Tuesday and Wednesday "incremental" backups would be much smaller, since they would contain only changes since the previous day. At some point (presumably next Monday), the administrator would plan to make another full dump.

With rsync, in contrast, the Wednesday backup is the big one. Indeed, the "full" backup is always the *most recent* one. The Tuesday directory would contain data only for those files that changed between Tuesday and Wednesday; the Monday directory would contain data for only those files that changed between Monday and Tuesday.

A little reasoning should convince you that the `rsync` way is *much* better for network-based backups, since it's only necessary to do a full backup once, instead of once per week. Thereafter, only the changes need to be copied. Unfortunately, you can't rsync to a tape, and that's probably why the dump and `tar` incremental backup models are still so popular. But in your author's opinion, these should never be used for network-based backups now that `rsync` is available.

Isolating the backup from the rest of the system

If you take the simple route and keep your backups in another directory on the same filesystem, then there's a very good chance that whatever damaged your data will also damage your backups. In this section, we identify a few simple ways to decrease your risk by keeping the backup data separate.

The easy (bad) way

In the previous section, we treated `/destination/` as if it were just another directory on the same filesystem. Let's call that the easy (bad) approach. It works, but it has several serious limitations:

- If your filesystem becomes corrupted, your backups will be corrupted too.
- If you suffer a hardware failure, such as a hard disk crash, it might be very difficult to reconstruct the backups.
- Since backups preserve permissions, your users--and any programs or viruses that they run--will be able to delete files from the backup. That is bad. Backups should be read-only.
- If you run out of free space, the backup process (which runs as root) might crash the system and make it difficult to recover.
- The easy (bad) approach offers no protection if the root account is compromised.

Fortunately, there are several easy ways to make your backup more robust.

Keep it on a separate partition

If your backup directory is on a separate partition, then any corruption in the main filesystem will not normally affect the backup. If the backup process runs out of disk space, it will fail, but it won't take the rest of the system down too. More importantly, keeping your backups on a separate partition means you can keep them mounted read-only; we'll discuss that in more detail in the next chapter.

Keep that partition on a separate disk

If your backup partition is on a separate hard disk, then you're also protected from hardware failure. That's very important, since hard disks always fail eventually, and often take your data with them. An entire industry has formed to service the needs of those whose broken hard disks contained important data that was not properly backed up.

Important: Notice, however, that in the event of hardware failure you'll still lose any changes made since the last backup. For home or small office users, where backups are made daily or even hourly as described in this document, that's probably fine, but in situations where any data loss at all would

be a serious problem (such as where financial transactions are concerned), a RAID system might be more appropriate.

RAID is well-supported under Linux, and the methods described in this document can also be used to create rotating snapshots of a RAID system.

Keep that disk on a separate machine

If you have a spare machine, even a very low-end one, you can turn it into a dedicated backup server. Make it standalone, and keep it in a physically separate place--another room or even another building. Disable every single remote service on the backup server, and connect it only to a dedicated network interface on the source machine.

On the source machine, export the directories that you want to back up via read-only NFS to the dedicated interface. The backup server can mount the exported network directories and run the snapshot routines discussed in this article as if they were local. If you opt for this approach, you'll only be remotely vulnerable if:

1. a remote root hole is discovered in read-only NFS, and
2. the source machine has already been compromised.

I'd consider this "pretty good" protection, but if you're (wisely) paranoid, or your job is on the line, build two backup servers. Then you can make sure that at least one of them is always offline.

If you're using a remote backup server and can't get a dedicated line to it (especially if the information has to cross somewhere insecure, like the public internet), you should probably skip the NFS approach and use `rsync -e ssh` instead.

It has been pointed out to me that `rsync` operates far more efficiently in server mode than it does over NFS, so if the connection between your source and backup server becomes a bottleneck, you should consider configuring the backup machine as an rsync server instead of using NFS. On the downside, this approach is slightly less transparent to users than NFS--snapshots would not appear to be mounted as a system directory, unless NFS is used in that direction, which is certainly another option (I haven't tried it yet though). Thanks to Martin Pool, a lead developer of `rsync`, for making me aware of this issue.

Here's another example of the utility of this approach--one that I use. If you have a bunch of windows desktops in a lab or office, an easy way to keep them all backed up is to share the relevant files, read-only, and mount them all from a dedicated backup server using SAMBA. The backup job can treat the SAMBA-mounted shares just like regular local directories.

Making the backup as read-only as possible

In the previous section, we discussed ways to keep your backup data physically separate from the data they're backing up. In this section, we discuss the other side of that coin--preventing user processes from modifying backups once they're made.

We want to avoid leaving the `snapshot` backup directory mounted read-write in a public place. Unfortunately, keeping it mounted read-only the whole time won't work either--the backup process itself needs write access. The ideal situation would be for the backups to be mounted read-only in a public place, but at the same time, read-write in a private directory accessible only by root, such as `/root/snapshot`.

There are a number of possible approaches to the challenge presented by mounting the backups read-only. After some amount of thought, I found a solution which allows root to write the backups to the directory but only gives the users read permissions. I'll first explain the other ideas I had and why they were less satisfactory.

It's tempting to keep your backup partition mounted read-only as `/snapshot` most of the time, but unmount that and remount it read-write as `/root/snapshot` during the brief periods while snapshots are being made. Don't give in to temptation!.

Bad: `mount`/`umount`

A filesystem cannot be unmounted if it's busy--that is, if some process is using it. The offending process need not be owned by root to block an unmount request. So if you plan to `umount` the read-only copy of the backup and `mount` it read-write somewhere else, don't--any user can accidentally (or deliberately) prevent the backup from happening. Besides, even if blocking unmounts were not an issue, this approach would introduce brief intervals during which the backups would seem to vanish, which could be confusing to users.

Better: `mount` read-only most of the time

A better but still-not-quite-satisfactory choice is to remount the directory read-write in place:

```
mount -o remount,rw /snapshot
[ run backup process ]
mount -o remount,ro /snapshot
```

Now any process that happens to be in `/snapshot` when the backups start will not prevent them from happening. Unfortunately, this approach introduces a new problem--there is a brief window of vulnerability, while the backups are being made, during which a user process could write to the backup directory. Moreover, if any process opens a backup file for writing during that window, it will prevent the backup from being remounted read-only, and the backups will stay vulnerable indefinitely.

Tempting but doesn't seem to work: the 2.4 kernel's `mount --bind`

Starting with the 2.4-series Linux kernels, it has been possible to mount a filesystem simultaneously in two different places. "Aha!" you might think, as I did. "Then surely we can mount the backups read-only in `/snapshot`, and read-write in `/root/snapshot` at the same time!"

Alas, no. Say your backups are on the partition `/dev/hdb1`. If you run the following commands,

```
mount /dev/hdb1 /root/snapshot
mount --bind -o ro /root/snapshot /snapshot
```

then (at least as of the 2.4.9 Linux kernel--updated, still present in the 2.4.20 kernel), `mount` will report `/dev/hdb1` as being mounted read-write in `/root/snapshot` and read-only in `/snapshot`, just as you requested. Don't let the system mislead you!

It seems that, at least on my system, read-write vs. read-only is a property of the filesystem, not the mount point. So every time you change the mount status, it will affect the status at every point the filesystem is mounted, even though neither `/etc/mtab` nor `/proc/mounts` will indicate the change.

In the example above, the second `mount` call will cause both of the mounts to become read-only, and the backup process will be unable to run. Scratch this one.

Update: I have it on fairly good authority that this behavior is considered a bug in the Linux kernel, which will be fixed as soon as someone gets around to it. If you are a kernel maintainer and know

more about this issue, or are willing to fix it, I'd love to hear from you!

My solution: using NFS on localhost

This is a bit more complicated, but until Linux supports `mount --bind` with different access permissions in different places, it seems like the best choice. Mount the partition where backups are stored somewhere accessible only by root, such as `/root/snapshot`. Then export it, read-only, via NFS, but only to the same machine. That's as simple as adding the following line to `/etc/exports`:

```
/root/snapshot 127.0.0.1(secure,ro,no_root_squash)
```

then start `nfs` and `portmap` from `/etc/rc.d/init.d/`. Finally mount the exported directory, read-only, as `/snapshot`:

```
mount -o ro 127.0.0.1:/root/snapshot /snapshot
```

And verify that it all worked:

```
mount
...
/dev/hdb1 on /root/snapshot type ext3 (rw)
127.0.0.1:/root/snapshot on /snapshot type nfs (ro,addr=127.0.0.1)
```

At this point, we'll have the desired effect: only root will be able to write to the backup (by accessing it through `/root/snapshot`). Other users will see only the read-only `/snapshot` directory. For a little extra protection, you could keep mounted read-only in `/root/snapshot` most of the time, and only remount it read-write while backups are happening.

Damian Menscher pointed out this CERT advisory which specifically recommends *against* NFS exporting to localhost, though since I'm not clear on why it's a problem, I'm not sure whether exporting the backups read-only as we do here is also a problem. If you understand the rationale behind this advisory and can shed light on it, would you please contact me? Thanks!

Extensions: hourly, daily, and weekly snapshots

With a little bit of tweaking, we make multiple-level rotating snapshots. On my system, for example, I keep the last four "hourly" snapshots (which are taken every four hours) as well as the last three "daily" snapshots (which are taken at midnight every day). You might also want to keep weekly or even monthly snapshots too, depending upon your needs and your available space.

Keep an extra script for each level

This is probably the easiest way to do it. I keep one script that runs every four hours to make and rotate hourly snapshots, and another script that runs once a day rotate the daily snapshots. There is no need to use rsync for the higher-level snapshots; just cp -al from the appropriate hourly one.

Run it all with `cron`

To make the automatic snapshots happen, I have added the following lines to root's `crontab` file:

```
0 */4 * * * /usr/local/bin/make_snapshot.sh
0 13 * * *  /usr/local/bin/daily_snapshot_rotate.sh
```

They cause `make_snapshot.sh` to be run every four hours on the hour and `daily_snapshot_rotate.sh` to be run every day at 13:00 (that is, 1:00 PM). I have included those scripts in the appendix.

If you tire of receiving an email from the `cron` process every four hours with the details of what was backed up, you can tell it to send the output of `make_snapshot.sh` to `/dev/null`, like so:

```
0 */4 * * * /usr/local/bin/make_snapshot.sh >/dev/null 2>&1
```

Understand, though, that this will prevent you from seeing errors if `make_snapshot.sh` cannot run for some reason, so be careful with it. Creating a third script to check for any unusual behavior in the snapshot periodically seems like a good idea, but I haven't implemented it yet. Alternatively, it might make sense to log the output of each run, by piping it through `tee`, for example. mRgOBLIN wrote in to suggest a better (and obvious, in retrospect!) approach, which is to send stdout to /dev/null but keep stderr, like so:

```
0 */4 * * * /usr/local/bin/make_snapshot.sh >/dev/null
```

Presto! Now you only get mail when there's an error. :)

Appendix: my actual configuration

I know that listing my actual backup configuration here is a security risk; please be kind and don't use this information to crack my site. However, I'm not a security expert, so if you see any vulnerabilities in my setup, I'd greatly appreciate your help in fixing them. Thanks!

I actually use two scripts, one for every-four-hours (hourly) snapshots, and one for every-day (daily) snapshots. I am only including the parts of the scripts that relate to backing up `/home`, since those are relevant ones here.

I use the NFS-to-localhost trick of exporting `/root/snapshot` read-only as `/snapshot`, as discussed above.

The system has been running without a hitch for months.

Listing one: `make_snapshot.sh`

```
#!/bin/bash
# ----------------------------------------------------------------------
# mikes handy rotating-filesystem-snapshot utility
# ----------------------------------------------------------------------
# this needs to be a lot more general, but the basic idea is it makes
# rotating backup-snapshots of /home whenever called
# ----------------------------------------------------------------------

unset PATH      # suggestion from H. Milz: avoid accidental use of $PATH
```

```
# ------------- system commands used by this script --------------------
ID=/usr/bin/id;
ECHO=/bin/echo;

MOUNT=/bin/mount;
RM=/bin/rm;
MV=/bin/mv;
CP=/bin/cp;
TOUCH=/bin/touch;

RSYNC=/usr/bin/rsync;


# ------------- file locations -----------------------------------------

MOUNT_DEVICE=/dev/hdb1;
SNAPSHOT_RW=/root/snapshot;
EXCLUDES=/usr/local/etc/backup_exclude;


# ------------- the script itself --------------------------------------

# make sure we're running as root
if (( `$ID -u` != 0 )); then { $ECHO "Sorry, must be root.  Exiting...";
exit; } fi

# attempt to remount the RW mount point as RW; else abort
$MOUNT -o remount,rw $MOUNT_DEVICE $SNAPSHOT_RW ;
if (( $? )); then
{
        $ECHO "snapshot: could not remount $SNAPSHOT_RW readwrite";
        exit;
}
fi;


# rotating snapshots of /home (fixme: this should be more general)

# step 1: delete the oldest snapshot, if it exists:
if [ -d $SNAPSHOT_RW/home/hourly.3 ] ; then                     \
$RM -rf $SNAPSHOT_RW/home/hourly.3 ;                            \
fi ;

# step 2: shift the middle snapshots(s) back by one, if they exist
if [ -d $SNAPSHOT_RW/home/hourly.2 ] ; then                     \
$MV $SNAPSHOT_RW/home/hourly.2 $SNAPSHOT_RW/home/hourly.3 ;     \
```

```
fi;
if [ -d $SNAPSHOT_RW/home/hourly.1 ] ; then                    \
$MV $SNAPSHOT_RW/home/hourly.1 $SNAPSHOT_RW/home/hourly.2 ;    \
fi;

# step 3: make a hard-link-only (except for dirs) copy of the latest
snapshot,
# if that exists
if [ -d $SNAPSHOT_RW/home/hourly.0 ] ; then                    \
$CP -al $SNAPSHOT_RW/home/hourly.0 $SNAPSHOT_RW/home/hourly.1 ; \
fi;

# step 4: rsync from the system into the latest snapshot (notice that
# rsync behaves like cp --remove-destination by default, so the
destination
# is unlinked first.  If it were not so, this would copy over the other
# snapshot(s) too!
$RSYNC                                                         \
        -va --delete --delete-excluded                         \
        --exclude-from="$EXCLUDES"                             \
        /home/ $SNAPSHOT_RW/home/hourly.0 ;

# step 5: update the mtime of hourly.0 to reflect the snapshot time
$TOUCH $SNAPSHOT_RW/home/hourly.0 ;

# and thats it for home.

# now remount the RW snapshot mountpoint as readonly

$MOUNT -o remount,ro $MOUNT_DEVICE $SNAPSHOT_RW ;
if (( $? )); then
{
        $ECHO "snapshot: could not remount $SNAPSHOT_RW readonly";
        exit;
} fi;
```

As you might have noticed above, I have added an excludes list to the `rsync` call. This is just to prevent the system from backing up garbage like web browser caches, which change frequently (so they'd take up space in every snapshot) but would be no loss if they were accidentally destroyed.

Listing two: `daily_snapshot_rotate.sh`

```
#!/bin/bash
# ----------------------------------------------------------------------
# mikes handy rotating-filesystem-snapshot utility: daily snapshots
# ----------------------------------------------------------------------
# intended to be run daily as a cron job when hourly.3 contains the
```

```
# midnight (or whenever you want) snapshot; say, 13:00 for 4-hour
snapshots.
# -------------------------------------------------------------------

unset PATH

# ------------- system commands used by this script --------------------
ID=/usr/bin/id;
ECHO=/bin/echo;

MOUNT=/bin/mount;
RM=/bin/rm;
MV=/bin/mv;
CP=/bin/cp;

# ------------- file locations -----------------------------------------

MOUNT_DEVICE=/dev/hdb1;
SNAPSHOT_RW=/root/snapshot;

# ------------- the script itself --------------------------------------

# make sure we're running as root
if (( `$ID -u` != 0 )); then { $ECHO "Sorry, must be root.  Exiting...";
exit; } fi

# attempt to remount the RW mount point as RW; else abort
$MOUNT -o remount,rw $MOUNT_DEVICE $SNAPSHOT_RW ;
if (( $? )); then
{
        $ECHO "snapshot: could not remount $SNAPSHOT_RW readwrite";
        exit;
}
fi;


# step 1: delete the oldest snapshot, if it exists:
if [ -d $SNAPSHOT_RW/home/daily.2 ] ; then                      \
$RM -rf $SNAPSHOT_RW/home/daily.2 ;                             \
fi ;

# step 2: shift the middle snapshots(s) back by one, if they exist
if [ -d $SNAPSHOT_RW/home/daily.1 ] ; then                      \
$MV $SNAPSHOT_RW/home/daily.1 $SNAPSHOT_RW/home/daily.2 ;       \
fi;
if [ -d $SNAPSHOT_RW/home/daily.0 ] ; then                      \
$MV $SNAPSHOT_RW/home/daily.0 $SNAPSHOT_RW/home/daily.1;        \
```

```
fi;

# step 3: make a hard-link-only (except for dirs) copy of
# hourly.3, assuming that exists, into daily.0
if [ -d $SNAPSHOT_RW/home/hourly.3 ] ; then                    \
$CP -al $SNAPSHOT_RW/home/hourly.3 $SNAPSHOT_RW/home/daily.0 ;  \
fi;

# note: do *not* update the mtime of daily.0; it will reflect
# when hourly.3 was made, which should be correct.

# now remount the RW snapshot mountpoint as readonly

$MOUNT -o remount,ro $MOUNT_DEVICE $SNAPSHOT_RW ;
if (( $? )); then
{
        $ECHO "snapshot: could not remount $SNAPSHOT_RW readonly";
        exit;
} fi;
```

Sample output of `ls -l /snapshot/home`

```
total 28
drwxr-xr-x   12 root     root         4096 Mar 28 00:00 daily.0
drwxr-xr-x   12 root     root         4096 Mar 27 00:00 daily.1
drwxr-xr-x   12 root     root         4096 Mar 26 00:00 daily.2
drwxr-xr-x   12 root     root         4096 Mar 28 16:00 hourly.0
drwxr-xr-x   12 root     root         4096 Mar 28 12:00 hourly.1
drwxr-xr-x   12 root     root         4096 Mar 28 08:00 hourly.2
drwxr-xr-x   12 root     root         4096 Mar 28 04:00 hourly.3
```

Notice that the contents of each of the subdirectories of `/snapshot/home/` is a complete image of `/home` at the time the snapshot was made. Despite the `w` in the directory access permissions, no one--not even root--can write to this directory; it's mounted read-only.

Bugs

Maintaining Permissions and Owners in the snapshots

The snapshot system above does not properly maintain old ownerships/permissions; if a file's ownership or permissions are changed in place, then the new ownership/permissions will apply to older snapshots as well. This is because `rsync` does not unlink files prior to changing them if the only changes are ownership/permission. Thanks to J.W. Schultz for pointing this out. Using his new `--link-dest` option, it is now trivial to work around this problem. See the discussion in the *Putting it all together* section of Incremental backups with `rsync`, above.

`mv` updates timestamp bug

Apparently, a bug in some Linux kernels between 2.4.4 and 2.4.9 causes `mv` to update timestamps; this may result in inaccurate timestamps on the snapshot directories. Thanks to Claude Felizardo for pointing this problem out. He was able to work around the problem my replacing `mv` with the following script:

```
MV=my_mv;
...
function my_mv() {
   REF=/tmp/makesnapshot-mymv-$;
   touch -r $1 $REF;
   /bin/mv $1 $2;
   touch -r $REF $2;
   /bin/rm $REF;
}
```

Windows-related problems

I have recently received a few reports of what appear to be interaction issues between Windows and rsync.

One report came from a user who mounts a windows share via Samba, much as I do, and had files mysteriously being deleted from the backup even when they weren't deleted from the source. Tim Burt also used this technique, and was seeing files copied even when they hadn't changed. He determined that the problem was modification time precision; adding --modify-window=10 caused rsync to behave correctly in both cases. *If you are rsync'ing from a SAMBA share, you must add --modify-window=10* or you may get inconsistent results. Update: --modify-window=1 should be sufficient. Yet another update: the problem appears to still be there. Please let me know if you use this method and files which should not be deleted are deleted.

Also, for those who use rsync directly on cygwin, there are some known problems, apparently related to cygwin signal handling. Scott Evans reports that rsync sometimes hangs on large directories. Jim Kleckner informed me of an rsync patch, discussed here and here, which seems to work around this problem. I have several reports of this working, and two reports of it not working (the hangs continue). However, one of the users who reported a negative outcome, Greg Boyington, was able to get it working using Craig Barrett's suggested sleep() approach, which is documented here.

Memory use in rsync scales linearly with the number of files being sync'd. This is a problem when syncing large file trees, especially when the server involved does not have a lot of RAM. If this limitation is more of an issue to you than network speed (for example, if you copy over a LAN), you may wish to use mirrordir instead. I haven't tried it personally, but it looks promising. Thanks to Vladimir Vuksan for this tip!

Contributed codes

Several people have been kind enough to send improved backup scripts. There are a number of good ideas here, and I hope they'll save you time when you're ready to design your own backup plan. Disclaimer: I have not necessarily tested these; make sure you check the source code and test them thoroughly before use!

- Art Mulder's original shell script
- Art Mulder's improved snapback Perl script, and a sample snapback.conf configuration file
- Henry Laxen's perl script
- J. P. Stewart's shell script
- Sean Herdejurgen's shell_script

- Peter Schneider-Kamp's shell script
- Rob Bos' versatile, GPL'd shell script. *Update!* 2002.12.13: check out his new package that makes for easier configuration and fixes a couple of bugs.
- Leland Elie's very nice GPL'd Python script, roller.py (2004.04.13: note link seems to be down). Does locking for safety, has a `/etc/roller.conf` control script which can pull from multiple machines automatically and independently.
- William Stearns' rsync backup for the extremely security-conscious. I haven't played with this yet, but it looks promising!
- Geordy Kitchen's shell script, adapted from Rob Bos' above
- Tim Evans' python rbackup functions and the calling script. You'll have to rename them before using.
- Elio Pizzottelli's improved version of make_snapshot.sh, released under the GPL
- John Bowman's rlbackup utility, which (in his words) provides a simple secure mechanism for generating and recovering linked backups over the network, with historical pruning. This one makes use of the --link-dest patch, and keeps a geometric progression of snapshots instead of doing hourly/daily/weekly.
- Ben Gardiner's much-improved boglin script
- Darrel O'Pry contributes a script modified to handle mysql databases. Thanks, Darrel! He also contributes a restore script which works with Geordy Kitchen's backup script.
- Craig Jones contributes a modified and enhanced version of make_snapshot.sh.
- Here is a very schnazzy perl script from Bart Vetters with built-in POD documentation
- Stuart Sheldon has contributed mirror.dist, a substantial improvement to the original shell script.
- Aaron Freed contributed two scripts from his KludgeKollection page, snapback and snaptrol.

References

- Rsync main site
- rdiff-backup, Ben Escoto's remote incremental backup utility
- The GNU coreutils package (which includes the part formerly known as fileutils, thanks to Nathan Rosenquist for pointing that out to me).
- dirvish, a similar but slightly more sophisticated tool from J.W. Schultz.
- rsback, a backup front-end for rsync, by Hans-Juergen Beie.
- ssync, a simple sync utility which can be used instead of rsync in certain cases. Thanks to Patrick Finerty Jr. for the link.
- bobs, the Browseable Online Backup System, with a snazzy web interface; I look forward to trying it! Thanks to Rene Rask.
- LVM, the Logical Volume Manager for Linux. In the context of LVM, *snapshot* means one image of the filesystem, frozen in time. Might be used in conjunction with some of the methods described on this page.
- glastree, a very nice snapshot-style backup utility from Jeremy Wohl
- mirrordir, a less memory-intensive (but more network-intensive) way to do the copying.
- A filesystem-level backup utility, rumored to be similar to Glastree and very complete and usable: storebackup. Thanks to Arthur Korn for the link!
- Gary Burd has posted a page which discusses how to use this sort of technique to back up laptops. He includes a very nice python script with it.
- Jason Rust implemented something like this in a php script called RIBS. You can find it here. Thanks Jason!
- Robie Basak pointed out to me that debian's fakeroot utility can help protect a backup server even if one of the machines it's backing up is compromised and an exploitable hole is discovered in rsync (this is a bit of a long shot, but in the backup business you really do have to be paranoid). He sent me this script along with this note explaining it.
- Michael Mayer wrote a handy and similar tutorial which is rather nicer than this one--has screenshots and everything! You can find it here.
- The rsnapshot project by Nathan Rosenquist which provides several extensions and features beyond the basic script here, and is really organized--it seems to be at a level which makes it more of a real package than a do-it-yourself hack like this page is. Check it out!

- Abe Loveless has written a howto for applying the rsync/hardlink backup strategy on the e-smith distribution.
- Mike Heins wrote Snapback2, a highly improved adapation of Art Mulder's original script, which includes (among other features) an apache-style configuration file, multiple redundant backup destinations, and safety features.
- Poul Petersen's Wombat backup system, written in Perl, supports threading for multiple simultaneous backups.

Frequently Asked Questions

- Q: What happens if a file is modified while the backup is taking place?
  - A: In rsync, transfers are done to a temporary file, which is cut over atomically, so the transfer either happens in its entirety or not at all. Basically, rsync does "the right thing," so you won't end up with partially-backed-up files. Thanks to Filippo Carletti for pointing this out. If you absolutely need a snapshot from a single instant in time, consider using Sistina's LVM (see reference above).
- Q: I really need the original permissions and ownerships in the snapshots, and not the latest ones. How can I accomplish that?
  - A: J.W. Schultz has created a --link-dest patch for rsync which takes care of the hard-linking part of this trick (instead of cp -al). It can preserve permissions and ownerships. As of `rsync-2.5.6`, it is now standard. See the discussion above.
- Q: I am backing up a cluster of machines (clients) to a backup server (server). What's the best way to pull data from each machine in the cluster?
  - A: Run sshd on each machine in the cluster. Create a passwordless key pair on the server, and give the public key to each of the client machines, restricted to the rsync command only (with PermitRootLogin set to forced-commands-only in the sshd_config file).
- Q: I am backing up many different machines with user accounts not necessarily shared by the backup server. How should I handle this?
  - A: Be sure to use the `--numeric-ids` option to rsync so that ownership is not confused on the restore. Thanks to Jon Jensen for this tip!
- Q: Can I see a nontrivial example involving rsync include an exclude rules?
  - A: Martijn Kruissen sent in an email which includes a nice example; I've posted part of it here.