Archiving and Backup

These are the file compression programs:

- gzip – Compress or expand files
- bzip2 – A block sorting file compressor

These are the archiving programs:

- tar – Tape archiving utility
- zip – Package and compress files

This is the file synchronization program:

- rsync – Remote file and directory synchronization

Compressing Files

- Data compression is the process of removing redundancy from data. Let’s consider an imaginary example. Say we had an entirely black picture file with the dimensions of 100 pixels by 100 pixels. In terms of data storage (assuming 24 bits, or 3 bytes per pixel), the image will occupy 30,000 bytes of storage.
- 100 * 100 * 3 = 30,000
- An image that is all one color contains entirely redundant data. If we were clever, we could encode the data in such a way that we simply describe the fact that we have a block of 10,000 black pixels.
- So, instead of storing a block of data containing 30,000 zeros (black is usually represented in image files as zero), we could compress the data into the number 10,000, followed by a zero to represent our data.
- Such a data compression scheme is called run-length encoding and is one of the most rudimentary compression techniques. Today’s techniques are much more advanced and complex, but the basic goal remains the same — get rid of redundant data

Compression algorithms are the mathematical techniques used to carry out the compression and fall into two general categories.

- Lossless: Lossless compression preserves all the data contained in the original. This means that when a file is restored from a compressed version, the restored file is exactly the same as the original, uncompressed version.
- Lossy: Lossy compression, on the other hand, removes data as the compression is performed to allow more compression to be applied. When a lossy file is restored, it does not match the original version; rather, it is a close approximation. Examples of lossy compression are JPEG (for images) and MP3 (for music).

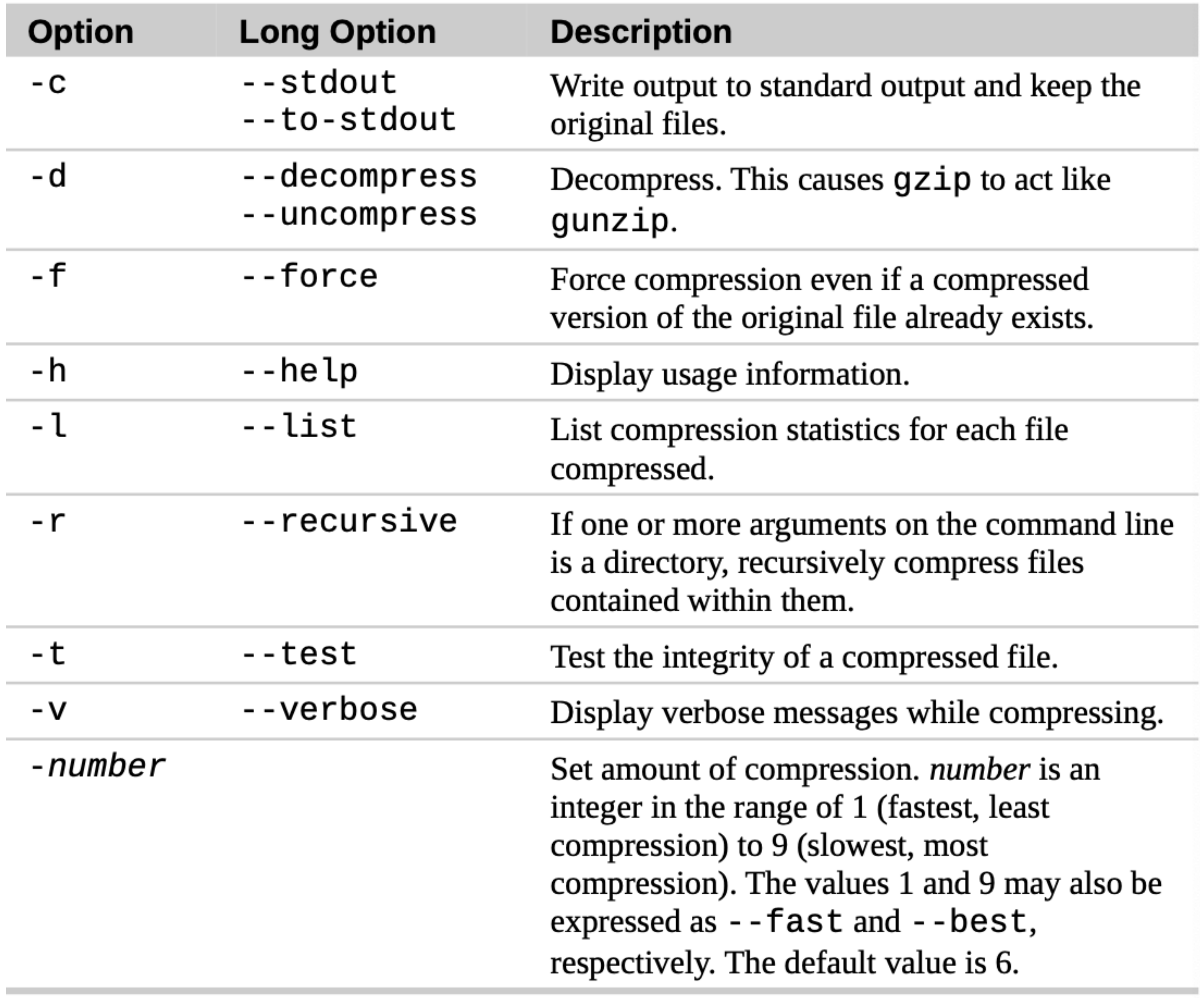
gzip

- The `gzip` program is used to compress one or more files. When executed, it replaces the original file with a compressed version of the original. The corresponding `gunzip` program is used to restore compressed files to their original, uncompressed form. Here is an example:

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    15738 2025-10-14 07:15 foo.txt
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    3230 2025-10-14 07:15 foo.txt.gz
[me@linuxbox ~]$ gunzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    15738 2025-10-14 07:15 foo.txt
```

- In this example, we create a text file named `foo.txt` from a directory listing. Next, we run `gzip`, which replaces the original file with a compressed version named `foo.txt.gz`. In the directory listing of `foo.*`, we see that the original file has been replaced with the compressed version and that the compressed version is about one-fifth the size of the original.
- The compressed file has the same permissions and time-stamp as the original.

gzip Options



Option	Long Option	Description
`-c`	`--stdout` `--to-stdout`	Write output to standard output and keep the original files.
`-d`	`--decompress` `--uncompress`	Decompress. This causes `gzip` to act like `gunzip`.
`-f`	`--force`	Force compression even if a compressed version of the original file already exists.
`-h`	`--help`	Display usage information.
`-l`	`--list`	List compression statistics for each file compressed.
`-r`	`--recursive`	If one or more arguments on the command line is a directory, recursively compress files contained within them.
`-t`	`--test`	Test the integrity of a compressed file.
`-v`	`--verbose`	Display verbose messages while compressing.
`-`*`number`*		Set amount of compression. *number* is an integer in the range of 1 (fastest, least compression) to 9 (slowest, most compression). The values 1 and 9 may also be expressed as `--fast` and `--best`, respectively. The default value is 6.

```
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ gzip -tv foo.txt.gz
foo.txt.gz:    OK
[me@linuxbox ~]$ gzip -d foo.txt.gz
```

- In the aboce example, we replaced the file foo.txt with a compressed version named foo.txt.gz.
- Next, we tested the integrity of the compressed version, using the -t and -v options. Finally, we uncompressed the file to its original form. gzip can also be used in interesting ways via standard input and output.

```
[me@linuxbox ~]$ ls -l /etc | gzip > foo.txt.gz
```

- This command creates a compressed version of a directory listing. The gunzip program, which uncompresses gzip files, assumes that filenames end in the extension .gz, so it’s not necessary to specify it, as long as the specified name is not in conflict with an existing uncompressed file.

```
[me@linuxbox ~]$ gunzip foo.txt
```

- If our goal were only to view the contents of a compressed text file, we could do this:

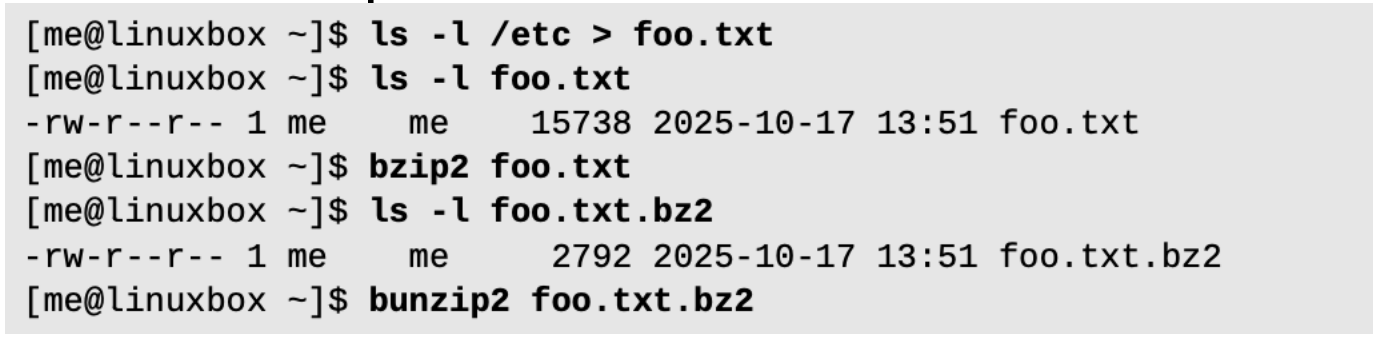
```
[me@linuxbox ~]$ gunzip -c foo.txt | less
```

- Alternately, there is a program supplied with gzip, called zcat, that is equivalent to gunzip with the -c option. It can be used like the cat command on gzip compressed files.

```
[me@linuxbox ~]$ zcat foo.txt.gz | less
```

bzip2

- The bzip2 program, by Julian Seward, is similar to gzip but uses a different compression algorithm that achieves higher levels of compression at the cost of compression speed. In most regards, it works in the same fashion as gzip.
- A file compressed with bzip2 is denoted with the extension .bz2.



```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-r--r-- 1 me    me    15738 2025-10-17 13:51 foo.txt
[me@linuxbox ~]$ bzip2 foo.txt
[me@linuxbox ~]$ ls -l foo.txt.bz2
-rw-r--r-- 1 me    me     2792 2025-10-17 13:51 foo.txt.bz2
[me@linuxbox ~]$ bunzip2 foo.txt.bz2
```

- bzip2 can be used the same way as gzip. All the options (except for - r) that we discussed for gzip are also supported in bzip2. Note, however, that the compression-level option (-number) has a somewhat different meaning to bzip2.
- bzip2 comes with bunzip2 and bzcat for uncompressing files.
- bzip2 also comes with the bzip2recover program, which will try to recover damaged .bz2 files.

Archiving Files

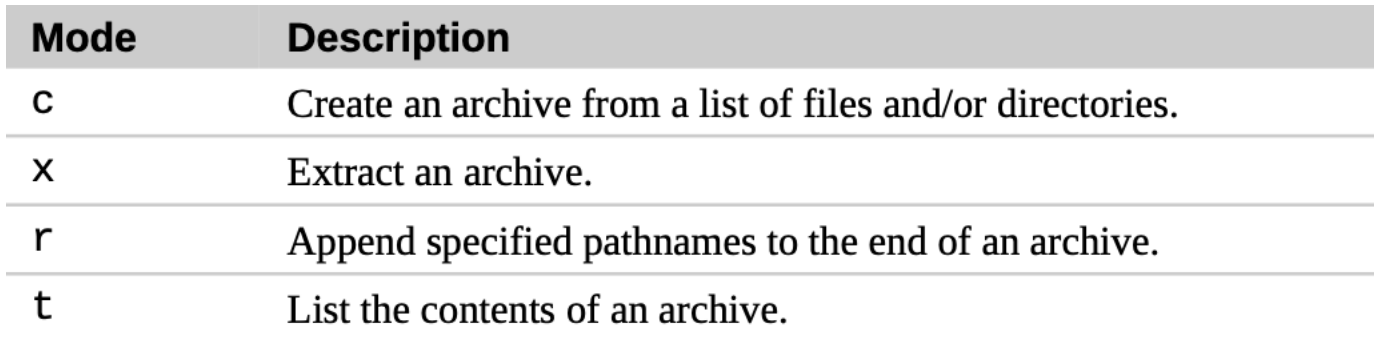
- Archiving is the process of gathering up many files and bundling them together into a single large file.
- Archiving is often done as part of system backups. It is also used when old data is moved from a system to some type of long-term storage.

tar

- In the Unix-like world of software, the tar program is the classic tool for archiving files.
- Its name, short for tape archive, reveals its roots as a tool for making backup tapes. While it is still used for that traditional task, it is equally adept on other storage devices.
- We often see filenames that end with the extension `.tar` or `.tgz`, which indicate a “plain” tar archive and a gzipped archive, respectively. A tar archive can consist of a group of sepa-
- rate files, one or more directory hierarchies, or a mixture of both. The command syntax works like this:

```
tar mode[options] pathname...
```

- Here mode is one of the following operating modes listed in the table



Mode	Description
`c`	Create an archive from a list of files and/or directories.
`x`	Extract an archive.
`r`	Append specified pathnames to the end of an archive.
`t`	List the contents of an archive.

- let’s re-create our playground from the previous chapter.

```
[me@linuxbox ~]$ mkdir -p playground/dir-{001..100}
[me@linuxbox ~]$ touch playground/dir-{001..100}/file-{A..Z}
```

- Next, create a tar archive of the entire playground.

```
[me@linuxbox ~]$ tar cf playground.tar playground
```

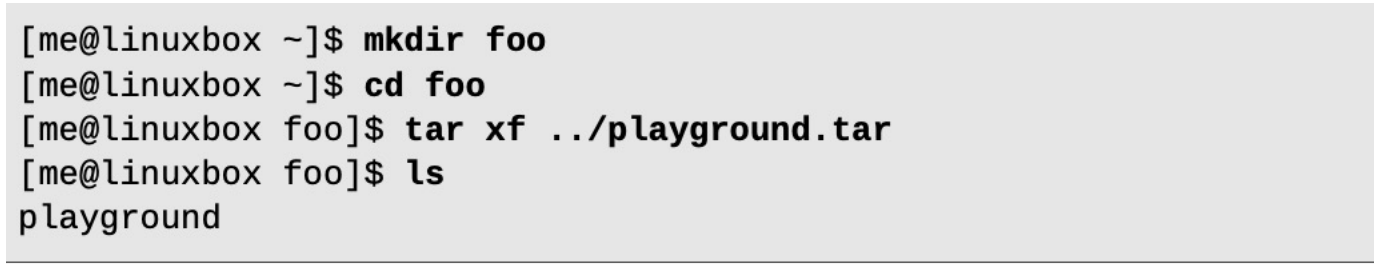
- This command creates a tar archive named `playground.tar` that contains the entire playground directory hierarchy.
- We can see that the mode and the f option, which is used to specify the name of the tar archive, may be joined together and do not require a leading dash.
- Note, however, that the mode must always be specified first, before any option
- To list the contents of the archive, we can do this:

```
[me@linuxbox ~]$ tar tf playground.tar
```

- For a more detailed listing, we can add the v (verbose) option

```
[me@linuxbox ~]$ tar tvf playground.tar
```

- let’s extract the playground in a new location. We will do this by creating a new directory named foo, changing the directory and extracting the tar archive.



```
[me@linuxbox ~]$ mkdir foo
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground.tar
[me@linuxbox foo]$ ls
playground
```

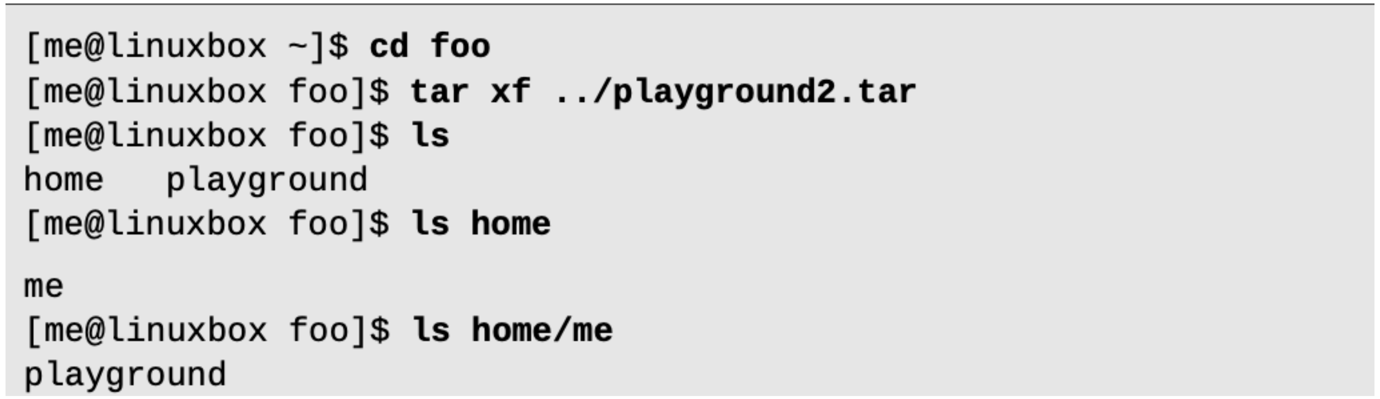
- If we examine the contents of ~/foo/playground, we see that the archive was successfully installed, creating a precise reproduction of the original files.
- There is one caveat, however. Unless we are operating as the superuser, files and directories extracted from archives take on the ownership of the user performing the restoration, rather than the original owner.

- Another interesting behavior of tar is the way it handles pathnames in archives. The default for pathnames is relative, rather than absolute. tar does this by simply removing any leading slash from the pathname when creating the archive. To demonstrate, we will re-create our archive, this time specifying an absolute pathname.

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ tar cf playground2.tar ~/playground
```

- Remember, ~/playground will expand into /home/me/playground when we press the Enter key, so we will get an absolute pathname for our demonstration.

- Next, we will extract the archive as before and watch what happens.



```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar
[me@linuxbox foo]$ ls
home   playground
[me@linuxbox foo]$ ls home

me
[me@linuxbox foo]$ ls home/me
playground
```

- Here we can see that when we extracted our second archive, it re-created the directory home/me/playground relative to our current working directory, ~/foo, not relative to the root directory, as would have been the case with an absolute pathname.
- This may seem like an odd way for it to work, but it’s actually more useful this way, because it allows us to extract archives to any location rather than being forced to extract them to their original locations.
- Repeating the exercise with the inclusion of the verbose option (v) will give a clearer picture of what’s going on.

- Let’s consider a hypothetical, yet practical, example of tar in action. Imagine we want to copy the home directory and its contents from one system to another and we have a large USB hard drive that we can use for the transfer. On our modern Linux system, the drive is “automagically” mounted in the /media directory. Let’s also imagine that the disk has a volume name of BigDisk when we attach it. To make the tar archive, we can do the following:

```
[me@linuxbox ~]$ sudo tar cf /media/BigDisk/home.tar /home
```

- After the tar file is written, we unmount the drive and attach it to the second computer. Again, it is mounted at /media/BigDisk. To extract the archive, we do this:

```
[me@linuxbox2 ~]$ cd /
[me@linuxbox2 /]$ sudo tar xf /media/BigDisk/home.tar
```

- What’s important to see here is that we must first change directory to / so that the extraction is relative to the root directory, since all pathnames within the archive are relative. When extracting an archive, it’s possible to limit what is extracted from the archive. For example, if we wanted to extract a single file from an archive, it could be done like this:

```
tar xf archive.tar pathname
```

- By adding the trailing pathname to the command, tar will restore only the specified file.
- Multiple pathnames may be specified. Note that the pathname must be the full, exact relative pathname as stored in the archive. When specifying pathnames, wildcards are not normally supported; however, the GNU version of tar (which is the version most often found in Linux distributions) supports them with the --wildcards option.

- Here is an example using our previous playground.tar file:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar --wildcards 'home/me/pla
yground/dir-*/file-A'
```

- This command will extract only files matching the specified pathname including the wildcard dir-*.
- tar is often used in conjunction with find to produce archives. In this example, we will use find to produce a set of files to include in an archive.

```
[me@linuxbox ~]$ find playground -name 'file-A' -exec tar rf
playground.tar '{}' '+'
```

- Here we use find to match all the files in playground named file-A and then, using the -exec action, we invoke tar in the append mode (r) to add the matching files to the archive playground.tar.

- Using tar with find is a good way of creating incremental backups of a directory tree or an entire system. By using find to match files newer than a timestamp file, we could create an archive that contains only those files newer than the last archive, assuming that the timestamp file is updated right after each archive is created.
- tar can also make use of both standard input and output. Here is a comprehensive example:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name 'file-A' | tar cf - --files-
from=- | gzip > playground.tgz
```

- In this example, we used the find program to produce a list of matching files and piped them into tar. If the filename - is specified, it is taken to mean standard input or output, as needed. (By the way, this convention of using - to represent standard input/output is used by a number of other programs, too). The --files-from option (which may also be specified as -T) causes tar to read its list of pathnames from a file rather than the command line. Lastly, the archive produced by tar is piped into gzip to create the compressed archive playground.tgz. The .tgz extension is the conventional extension given to gzip-compressed tar files. The extension .tar.gz is also used sometimes.

- While we used the gzip program externally to produce our compressed archive in the above example, modern versions of GNU tar support both gzip and bzip2 compression directly with the use of the z and j options, respectively. Using our previous example as a base, we can simplify it this way:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar czf
playground.tgz -T -
```

- If we had wanted to create a bzip2-compressed archive instead, we could have done this:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar cjf
playground.tbz -T -
```

- By simply changing the compression option from z to j (and changing the output file’s extension to .tbz to indicate a bzip2-compressed file) we enabled bzip2-compression.

- Another interesting use of standard input and output with the tar command involves transferring files between systems over a network. Imagine that we had two machines running a Unix-like system equipped with tar and ssh.
- In such a scenario, we could transfer a directory from a remote system (named remote-sys for this example) to our local system.

```
[me@linuxbox ~]$ mkdir remote-stuff
[me@linuxbox ~]$ cd remote-stuff
[me@linuxbox remote-stuff]$ ssh remote-sys 'tar cf - Documents' | tar
xf -
me@remote-sys’s password:
[me@linuxbox remote-stuff]$ ls
Documents
```

- Here we were able to copy a directory named Documents from the remote system remote-sys to a directory within the directory named remote-stuff on the local system. How did we do this? First, we launched the tar program on the remote system using ssh. You will recall that ssh allows us to execute a program remotely on a networked computer and “see” the results on the local system — the standard output produced on the remote system is sent to the local system for viewing.
- We can take advantage of this by having tar create an archive (the c mode) and send it to standard output, rather than a file (the f option with the dash argument), thereby transporting the archive over the encrypted tunnel provided by ssh to the local system. On the local system, we execute tar and have it expand an archive (the x mode) supplied from standard input (again, the f option with the dash argument).

zip

- The zip program is both a compression tool and an archiver. The file format used by the program is familiar to Windows users, as it reads and writes .zip files. In Linux, however, gzip is the predominant compression program, with bzip2 being a close second.
- In its most basic usage, zip is invoked like this:

```
zip options zipfile file...
```

- For example, to make a zip archive of our playground, we would do this:

```
[me@linuxbox ~]$ zip -r playground.zip playground
```

- Unless we include the -r option for recursion, only the playground directory (but none of its contents) is stored. Although the addition of the extension .zip is automatic, we will include the file extension for clarity.

- During the creation of the zip archive, zip will normally display a series of messages like this:

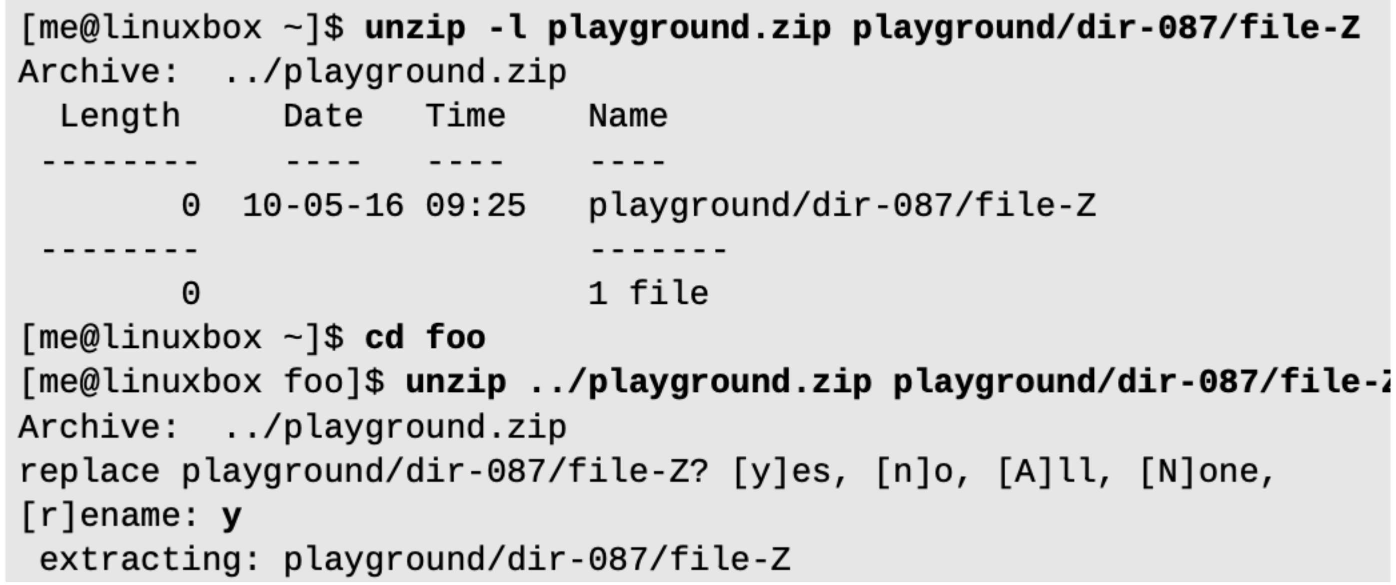
```
adding: playground/dir-020/file-Z (stored 0%)
adding: playground/dir-020/file-Y (stored 0%)
adding: playground/dir-020/file-X (stored 0%)
adding: playground/dir-087/ (stored 0%)
adding: playground/dir-087/file-S (stored 0%)
```

- These messages show the status of each file added to the archive. zip will add files to the archive using one of two storage methods: either it will “store” a file without compression, as shown here, or it will “deflate” the file that performs compression.
- The numeric value displayed after the storage method indicates the amount of compression achieved. Since our playground contains only empty files, no compression is performed on its contents.

- Extracting the contents of a zip file is straightforward when using the unzip pro

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip
```

- One thing to note about zip (as opposed to tar) is that if an existing archive is specified, it is updated rather than replaced. This means the existing archive is preserved, but new files are added and matching files are replaced.
- Files may be listed and extracted selectively from a zip archive by specifying them to unzip.



```
[me@linuxbox ~]$ unzip -l playground.zip playground/dir-087/file-Z
Archive:  ../playground.zip
  Length      Date    Time    Name
 --------    ----    ----    ----
        0  10-05-16 09:25   playground/dir-087/file-Z
 --------                   -------
        0                   1 file
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip playground/dir-087/file-
Archive:  ../playground.zip
replace playground/dir-087/file-Z? [y]es, [n]o, [A]ll, [N]one,
[r]ename: y
 extracting: playground/dir-087/file-Z
```

- Using the -l option causes unzip to merely list the contents of the archive without extracting the file. If no files are specified, unzip will list all files in the archive.
- The –v option can be added to increase the verbosity of the listing. Note that when the archive extraction conflicts with an existing file, the user is prompted before the file is replaced.
- Like tar, zip can make use of standard input and output, though its implementation is somewhat less useful. It is possible to pipe a list of filenames to zip via the -@ option.

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name "file-A" | zip -@ file-A.zip
```

- Here we use find to generate a list of files matching the test -name "file-A" and then pipe the list into zip, which creates the archive file-A.zip containing the selected files.
- zip also supports writing its output to standard output, but its use is limited because few programs can make use of the output.
- Unfortunately, the unzip program does not accept standard input. This prevents zip and unzip from being used together to perform network file copying like tar.
- zip can, however, accept standard input, so it can be used to compress the output of other programs.

```
[me@linuxbox ~]$ ls -l /etc/ | zip ls-etc.zip -
 adding: - (deflated 80%)
```

- In this example, we pipe the output of ls into zip. Like tar, zip interprets the trailing dash as “use standard input for the input file.”
- The unzip program allows its output to be sent to standard output when the -p (for pipe) option is specified.

```
[me@linuxbox ~]$ unzip -p ls-etc.zip | less
```

- Both zip/unzip have a lot of options that add to their flexibility, though some are platform specific to other systems.
- The man pages for both zip and unzip are pretty good and contain useful examples. However, the main use of these programs is for exchanging files with Windows systems, rather than performing compression and archiving on Linux, where tar and gzip are greatly preferred.

Synchronizing Files and Directories

- A common strategy for maintaining a backup copy of a system involves keeping one or more directories synchronized with another directory (or directories) located on either the local system (usually a removable storage device of some kind) or a remote system.
- We might, for example, have a local copy of a website under development and synchronize it from time to time with the “live” copy on a remote web server.
- In the Unix-like world, the preferred tool for this task is rsync. This program can synchronize both local and remote directories by using the rsync remote-update protocol, which allows rsync to quickly detect the differences between two directories and perform the minimum amount of copying required to bring them into sync. This makes rsync very fast and economical to use, compared to other kinds of copy programs.

- rsync is invoked like this:

```
rsync options source destination
```

- where source and destination are one of the following:
  - A local file or directory
  - A remote file or directory in the form of [user@]host:path
  - A remote rsync server specified with a URI of rsync://[user@]host[:port]/path
- Note that either the source or the destination must be a local file. Remote-to-remote copying is not supported.
- Let’s try rsync out on some local files. First, let’s clean out our foo directory.

```
[me@linuxbox ~]$ rm -rf foo/*
```

- Next, we’ll synchronize the playground directory with a corresponding copy in foo.

```
[me@linuxbox ~]$ rsync -av playground foo
```

- We’ve included both the -a option (for archiving — causes recursion and preservation of file attributes) and the -v option (verbose output) to make a mirror of the playground directory within foo. While the command runs, we will see a list of the files and directories being copied.

- At the end, we will see a summary message like this indicating the amount of copying performed:

```
sent 135759 bytes  received 57870 bytes  387258.00 bytes/sec
total size is 3230  speedup is 0.02
```

- If we run the command again, we will see a different result.

```
[me@linuxbox ~]$ rsync -av playground foo
building file list ... done

sent 22635 bytes  received 20 bytes  45310.00 bytes/sec
total size is 3230  speedup is 0.14
```

- Notice that there was no listing of files. This is because rsync detected that there were no differences between ~/playground and ~/foo/playground, and therefore it didn’t need to copy anything. If we modify a file in playground and run rsync again: we see that rsync detected the change and copied only the updated file.

```
[me@linuxbox ~]$ touch playground/dir-099/file-Z

[me@linuxbox ~]$ rsync -av playground foo
building file list ... done
playground/dir-099/file-Z
sent 22685 bytes  received 42 bytes  45454.00 bytes/sec
total size is 3230  speedup is 0.14
```

- There is a subtle but useful feature we can use when we specify an rsync source. Let’s consider two directories.

```
[me@linuxbox ~]$ ls
source        destination
```

- Directory source contains one file named file1 and directory destination is empty. If we perform a copy of source to destination like so:

```
[me@linuxbox ~]$ rsync source destination
```

- then rsync copies the directory source into destination.

```
[me@linuxbox ~]$ ls destination
source
```

- However, if we append a trailing / to the source directory name, rsync will copy only the contents of the source directory and not the directory itself.

```
[me@linuxbox ~]$ rsync source/ destination
[me@linuxbox ~]$ ls destination
file1
```

- let’s consider the imaginary external hard drive that we used earlier with tar. If we attach the drive to our system and it is mounted at /media/BigDisk once again, we can perform a useful system backup by first creating a directory named /backup on the external drive and then using rsync to copy the most important stuff from our system to the external drive.

```
[me@linuxbox ~]$ mkdir /media/BigDisk/backup
[me@linuxbox ~]$ sudo rsync -av --delete /etc /home /usr/local
/media/BigDisk/backup
```

- In this example, we copied the /etc, /home, and /usr/local directories from our system to our imaginary storage device. We included the --delete option to remove files that may have existed on the backup device that no longer existed on the source device (this is irrelevant the first time we make a backup but will be useful on subsequent copies).
- Repeating the procedure of attaching the external drive and running this rsync command would be a useful (though not ideal) way of keeping a small system backed up. Of course, an alias would be helpful here, too. We could create an alias and add it to our .bashrc file to provide this feature.

```
alias backup='sudo rsync -av --delete /etc /home /usr/local
/media/BigDisk/backup'
```

Using `rsync` Over a Network

- One of the real beauties of rsync is that it can be used to copy files over a network. After all, the r in rsync stands for “remote.” Remote copying can be done in one of two ways.
- The first way is with another system that has rsync installed, along with a remote shell program such as ssh.
- Let’s say we had another system on our local network with a lot of available hard drive space and we wanted to perform our backup operation using the remote system instead of an external drive.
- Assuming that it already had a directory named /backup where we could deliver our files, we could do this:

```
[me@linuxbox ~]$ sudo rsync -av --delete --rsh=ssh /etc /home
/usr/local remote-sys:/backup
```

- There were two changes to our command to facilitate the network copy. First, we added the `--rsh=ssh` option, which instructs rsync to use the ssh program as its remote shell.
- In this way, we were able to use an ssh-encrypted tunnel to securely transfer the data from the local system to the remote host. Second, we specified the remote host by prefixing its name (in this case the remote host is named remote-sys) to the destination pathname.
- The second way that rsync can be used to synchronize files over a network is by using an rsync server. rsync can be configured to run as a daemon and listen to incoming requests for synchronization. This is often done to allow mirroring of a remote system.
- For example, Red Hat Software maintains a large repository of software packages under development for its Fedora distribution. It is useful for software testers to mirror this collection during the testing phase of the distribution release cycle. Since files in the repository change frequently (often more than once a day), it is desirable to maintain a local mirror by periodic synchronization, rather than by bulk copying of the repository.

```
[me@linuxbox ~]$ mkdir fedora-devel
[me@linuxbox ~]$ rsync -av –delete rsync://archive.linux.duke.edu/
fedora/linux/development/rawhide/Everything/x86_64/os/ fedora-devel
```

- In this example, we use the URI of the remote rsync server, which consists of a protocol (rsync://), followed by the remote host-name (archive.linux.duke.edu), followed by the pathname of the repository.