Backing up your Linux machines

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

Should I take this tutorial?

This tutorial gives you techniques for covering your back in the worst-case scenario. Even new, high-quality hard drives will occasionally fail. Regular system backups are essential, especially for busy developers who make continual improvements to their code. This tutorial shows you how to protect yourself from losing huge amounts of critical data.

About the author

For technical questions about the content of this tutorial, contact the author, Daniel Robbins, at *drobbins@gentoo.org* .

Residing in Albuquerque, New Mexico, Daniel Robbins is the President/CEO of *Gentoo Technologies, Inc.* , the creator of **Gentoo Linux**, an advanced Linux for the PC, and the **Portage** system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at **SONY Electronic Publishing/Psygnosis**. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah.

Section 2. Backup basics

Do I really need to back up?

I'm sure some of you are wondering whether you actually need to perform system backups. After all, Linux is a fairly reliable operating system, and maybe you haven't lost any data ever before. You may think that if you're careful, you'll never lose any data and you'll be fine. I beg to differ :)

While I was in the process of writing this tutorial, my development server's hard drive died. When I got cvs.gentoo.org back up, I sent my developers the e-mail on the next panel...

The hard drive failure

```
To: gentoo-dev
Date: Sun, 22 July 2001 11:58:34 -0700
From: drobbins
Subject: cvs.gentoo.org hard drive failure

Important information:
=======================

At about 10:00 AM this morning, I walked into my home office and noticed that
cvs.gentoo.org's hard drive light was stuck on.  Turning on the monitor, I
found a number of IDE drive error kernel messages on the screen.  I quickly
rebooted the server, and found that the hard drive (a quite new IBM 45Gb
Ultra/100 drive) was making a weird chirping sound and was unable to spin up...
the system wouldn't recognize the drive; it was dead.

So, it's 11:51 AM now, and the system has been restored onto another IBM 30Gb
Ultra/100 drive.  *Fortunately for everyone*, I performed a full system backup
to tape about a day ago.  Ironically, I'm in the process of writing a backup
tutorial for IBM developerWorks.
```

No one is immune

No one is immune to component failures, and they often occur when you least expect them. Thankfully, only a very small amount of data was lost, and the restoration went quickly. However, if I had not performed a recent backup, we would have lost a tremendous amount of critical data, and the Gentoo Linux project would have been in serious trouble. I hope this illustrates how important regular system backups can be, especially for busy developers who make continual improvements to their code.

No one is immune, part 2

No matter how careful you are, some things are out of your control. A failed hard drive or a large power surge can easily destroy data. Sure, you may have a nice UPS (uninterruptible power supply) and high-quality hard drives in your system, and that's great. These reduce the probability of data loss but do not eliminate it; even new, high-quality hard drives will

occasionally fail, which is exactly what happened to me.

No one is immune, part 3

Maybe you're using a RAID array (a redundant array of inexpensive disks) --   also wonderful! RAID is an excellent technology for increasing your system's availability. However, RAID volumes only protect you against a spontaneous complete drive failure; if your filesystem becomes corrupt or someone accidentally deletes some files, RAID won't help you get your data back. So don't turn to RAID as a "way out" of performing a system backup, because it will only protect you against a specific kind of failure, whereas a good tape backup will prepare you for anything.

Some questions

Here are some good questions to ask yourself: If your computer exploded today, how would you cope? How much time would it take to recreate the data you lost? And how great an impact would this have on you personally (if you're considering your personal Linux box) or your employer? If you had to attach a dollar amount to the data you could lose, what would it be? Having answered this question, are you willing to accept that loss? If not, then you have a backup crisis that needs to be addressed immediately.

Section 3. A backup strategy

What files should I back up?

The first obvious approach is to back up all files that are important to you. One easy way to do this is to back up the contents of your /home tree; then, you can make sure that all your important files reside there, and they'll automatically get backed up.

A full backup

Here's another approach that you could seriously consider. If unnecessary downtime is a concern for you, then consider how much time it would take to reconfigure your operating system from scratch. Remember, restoring your Linux box may not only consist of popping in your favorite distribution CD, but could also include compiling special versions of several applications, tweaking important configuration files, and more.

Prepare for a "bare metal" restore

If you really don't want to have to do these things again, I recommend that you perform a full system backup. Then, you can perform a "bare metal" restore -- in a matter of hours, everything will be *exactly* as it was before. For servers, this is generally the approach you want to use so that the servers can be restored and running as soon as possible.

What media/drive should I use?

So, what kind of media should you use for your backups?

In general, a modern tape drive will offer the best value, performance, capacity, and bang for the buck. If you need to purchase a tape drive, first figure out a ballpark maximum amount that you're willing to spend on your tape hardware, based on your finances and the value of your data. Then, start shopping around for a Linux-compatible drive. The best way to start is to head over *http://www.linuxtapecert.org* and take a look at their Linux tape drive compatibility matrix. Pick out a drive that's reliable and has enough capacity for future growth.

No tape drive?

If you can't afford to purchase a tape drive, don't despair. Tape drives are generally the most convenient-to-use   type of backup hardware, but *any type of backup* is better than no backup at all. You can backup onto CD-R, a removable hard drive, an Iomega Zip drive, or even a 3.5" disk if you're desperate. Generally, if you're using non-tape hardware, you'll want to create an on-disk archive (such as a .tar.gz file) containing your data. Then, you should copy this tarball over to your media and store it in a safe place.

With tape, things are a little more convenient since the data can be streamed to the tape

directly, so there's no need for an intermediate on-disk archive.

What backup program should I use?

There are a lot of choices when it comes to free backup software. However, two notable ones stand out: tar and dump. There are differences between these two programs, namely, tar is filesystem independent, while dump is filesystem specific. Because dump works at a lower level than tar (backing up filesystem inodes directly), you'll need to use a version of dump specifically written for your particular filesystem, such as ext2.

The advantages of...tar?

While dump generally offers some performance advantages and is often the preferred backup method on other UNIX systems, I can't recommend it for a Linux system. Why? On other UNIX systems, there's generally only one "official" hard drive filesystem format, which makes dump's filesystem dependence less of an issue. However, Linux now has a bunch of new journalling filesystems, and dump isn't supported on at least one of them, ReiserFS. Due to the wide selection of filesystems in the Linux world, it's best to ensure that your backups are filesystem independent, meaning that dump is out of the picture.

star to the rescue

OK, so we've eliminated dump. For this tutorial, I'm going to be using a program called star. star is a tar-compatible program with a reputation for being the world's fastest tar implementation. It is also very well written and has a number of enhancements to overcome limitations found in other tar programs such as GNU tar; these limitations have led many people to prefer dump. While GNU tar *can* be used to perform tape backups, star is simply more refined, faster, easier to use, and more feature rich. And it's free, which is a good thing :) While star *is* a tar implementation, it shares many of the strengths of a dump implementation: speed, ease of use, and robustness. And because it's filesystem and platform independent, you'll have an easy time restoring to other filesystems (and even other UNIX systems, such as AIX or Solaris) if necessary.

Section 4. Configuring the tools

Downloading star sources

Regardless of your backup media type (tape or otherwise), you'll need the star program. You can grab the latest star sources from *ftp://ftp.fokus.gmd.de/pub/unix/star/alpha/* (currently star-1.3a8.tar.gz is the latest). Once you have the sources, follow these steps to get star installed...

star compilation...

```
# tar xzvf star-1.3a8.tar.gz
# cd star-1.3
# make
```

Now that everything's compiled, we'll need to install star. **Don't** run "make install"; this will install a couple of additional things that we don't want installed. Instead, do the following on the next panel.

...and installation

```
# install -m0755 star/OBJ/*/star /usr/bin
# cd ../..
```

You should now be in the "star" source directory. Now, for the man page --   if your system is FHS 2.1 compliant and stores man pages in /usr/share/man, modify the last line appropriately:

```
# gzip -9 star.1
# install -m0644 star.1.gz /usr/man/man1
```

mt

Now, it's time for the mt command. mt is used to perform all sorts of tape functions, such as rewinding, ejecting, advancing to the next filemark, etc. Because it's tape specific, you'll need it only if you're backing up directly to tape. You may have a version of mt already installed; make sure that it supports the "setblk" command (type "man mt" to find out). If it doesn't, or you'd just like to make sure you're running the latest and greatest version of mt available, you can follow these instructions to install mt-st-0.6   from sources.

Downloading mt

First, download the mt-st sources from *http://www.ibiblio.org/pub/linux/system/backup/* (the latest version is mt-st-0.6.tar.gz).   Then, follow the steps on the next panel to unpack,

compile, and install...

---

mt compilation and installation

```
# tar xzvf mt-st-0.6
# cd mt-st-0.6
# make
# install -m0755 mt /usr/sbin
# install -m0755 stinit /usr/sbin
# gzip -9 mt.1 stinit.8
# install -m0644 mt.1.gz /usr/man/man1
# install -m0644 stinit.8.gz /usr/man/man8
```

Now, mt (and stinit) will be installed. I recommend that you look in /usr/bin, /bin, and /sbin for any old copies of mt, and if you find any, be sure to rename them to mt.old. Now that our tools are installed, it's time to put together our backup tools.

Section 5. Tape drive technology

Block size

Before we start hacking away at our backup script, it's important to understand the format of data on tape. Linux writes data to tape in the form of equal-sized blocks (also called records). Between each record, a special marker called an *inter-record gap* is written to tape. So, when 300Mb of backup data is written to tape, this data is converted into a bunch of equal-sized records, and each pair of records is separated by an inter-record gap.

Linux block size

On Linux, the default record/block size is 1kb. Modern drives and tapes are capable of streaming several megabytes of data per second, and a 1kb block size isn't exactly optimal, for a couple of reasons. First, each inter-record gap takes up space on the tape, so the smaller the block size, the lower the "real" tape capacity. Also, most modern tape drives are optimized for larger block sizes.

Block size optimizations

For example, my *Ecrix VXA-1* tape drive is optimized for a block size of 64kb, and using this block size makes my backups about 50% faster than the default (1kb). The best way to figure out the optimal block size for your tape drive is to visit *http://www.linuxtapecert.org* . If you can't find the information there, you may want to check with your manufacturer -- or you can simply do a little experimentation with different tape block sizes. I'll show you how to set the block size in just a bit.

filemarks

So far we've taken a look at how a single glob of data is written to tape -- it's broken up into many small blocks, separated by tiny inter-record gaps. But how do you go about writing several globs of data to tape, if, for example, if you wanted to write two archives to your tape drive, one after the other? The process is actually quite easy, and relies on a special tape marker called a *filemark*.

A tale of two devices

To understand filemarks, it's important to grasp the differences between the two different types of tape devices, the rewind and no-rewind devices. In these examples, we'll be using a SCSI tape, so the rewind and no-rewind devices are /dev/st0 and /dev/nst0, respectively.

The rewind device

Here's how the rewind device (/dev/st0) works. Let's say you have a rewound tape in the

drive, and you perform a full system backup, as follows:

```
# cd /
# star -cv -f /dev/st0 .
```

A bunch of records and inter-record gaps will be written out to tape. After the backup is complete, a filemark will be written to tape, and the tape will be rewound.

The norewind device

But, if we use /dev/nst0 instead, the tape won't be rewound -- instead, the tape will remain positioned immediately after the filemark, meaning that we can put multiple archives on one tape. Assume that we have a rewound tape in the drive and execute these commands:

```
# cd /
# star -cv -V pat=home/\* -V pat=tmp/\* -V pat=proc/\* -f /dev/nst0 .
# cd /home
# star -cv -f /dev/nst0 .
```

The first star command will dump a backup to tape, consisting of all files except those found in /home, /tmp, and /proc. After this backup is complete, a filemark will be written to tape. Then, a second backup will begin that contains all the contents of the /home tree. Again, a filemark will be written after the backup is complete.

Rewinding

At this point, you can rewind the tape by typing:

```
# mt -f /dev/nst0 rewind
```

...or, you can rewind *and* eject it by typing:

```
# mt -f /dev/nst0 offline
```

Multi-archive advantages

So, are there any advantages to a two-archive backup approach as in the previous example? Definitely. Our first archive contained everything except the contents of /tmp, /proc, and /home, which means it contains all the files you need to get your system up and running again. This means that if your hard drive fails, you can restore the first archive to a new hard drive. Then, you can reboot under your restored OS, and restore the user data from the second archive. And, if you simply need to restore a single file that a user deleted by accident, you can extract it from the second archive, and tar won't have to wade through all the system files.

Section 6. mt in action

Accessing archives on tape

As you can see, it's easy to store two archives on a single tape. However, how do you go about accessing them? Fortunately, the "mt" command comes to the rescue. To print the contents of the second archive on tape, you can type:

```
# mt -f /dev/nst0 asf 1
# star -tv -f /dev/nst0
```

The "mt asf" command works by rewinding the tape, and then advancing the number of filemarks that you specify. As you can see, the two archives have no associated filename as they would on a disk filesystem --   instead, mt refers to them by number, starting with zero.

---

mt fsf

You could do the same thing with two mt commands, as follows:

```
# mt -f /dev/nst0 rewind
# mt -f /dev/nst0 fsf 1
# star -tv -f /dev/nst0
```

This time, we use the "fsf" command to advance just beyond the first filemark found relative to the tape's current position.

Moving around

If you wanted to list the contents of the first archive, followed by the contents of the second archive, you could type:

```
# mt -f /dev/nst0 rewind
# star -tv -f /dev/nst0
# star -tv -f /dev/nst0
```

In this example, the first star command prints the contents of archive zero, and then advances just beyond the first filemark, right at the beginning of the next archive. Then, the second star command is ready to begin reading the contents of the second archive.

Moving around, continued

Here's another, less efficient method (since it performs an additional rewind and seek) to perform the same thing:

```
# mt -f /dev/nst0 asf 0
```

```
# star -tv -f /dev/nst0
# mt -f /dev/nst0 asf 1
# star -tv -f /dev/nst0
```

You should now have a good grasp of how filemarks work. Next, I'll show you how to set the tape block size, and then we'll be ready to start working on a tape backup script.

Setting block sizes

Now, it's time to learn all about how to set and detect tape block sizes. As I mentioned earlier, Linux has a default tape block size of 1kb. And, mt has a special "setblk" command that's used to set the default block size. So, you might imagine that you use setblk to set the block size that you'd like to use, and then you're ready to go. However, in reality, things are a bit more convoluted.

Variable block size

If you want to use a different block size, the best way to go about it is to set your drive into "variable" block size mode. You can do this by using the setblk command and specifying a block size of zero:

```
# mt -f /dev/nst0 setblk 0
```

Specifying block size with star

With the block size in variable mode, you can now control the blocksize using the star program itself. So, before you'd type:

```
# star -cv -f /dev/nst0 .
```

...but now, you specify the block size as follows:

```
# star -cv bs=64k -f /dev/nst0 .
```

...and your records will now be 64k in length, which is the optimal size for my particular tape drive.

Block size quirks

Here are some interesting quirks that you need to know about tape block size. First, when reading from a tape, you must specify the proper block size. The backup program can't determine the block size automatically. So, if you wanted to list the contents of an archive that had a blocksize of 64k, you'd need to make sure the tape drive is in variable block size

mode and then specify it on the star command line:

```
# star -tv bs=64k -f /dev/nst0
```

Another quirk

Here's another quirk. Let's say you try to read the tape, but you use the wrong block size. If the read block size is *smaller* than the write block size, you'll get an IO error. However, if the read block size is an *exact multiple* of the write block size, all will be well. But, if the read block size is larger than the write block size, but isn't a multiple, then you'll also get an IO error. This all happens because the tape driver can read one or more whole blocks at once, but gets confused when it reads a partial block. For this reason, it's a very good idea to write the block size you use on the tape label for future reference.

I forgot the block size

Just for kicks, let's say you forgot the block size you used, or you're trying to restore data from an old tape and you don't know what block size the data is formatted in. Is there any way to determine the tape block size? Fortunately, there is. First, insert the tape in the drive and type the following command:

```
# dd if=/dev/nst0 of=/tmp/testblock bs=128k count=1
```

Now, type "ls -l /tmp/testblock" and take a look at the size of the file --   it'll be equal to the tape block size.

How it works

Here's how the trick works. Since dd knows all about block sizes and inter-record gaps, when we specify "bs=128k count=1", it knows that we want to read a *single* block of data. dd will attempt to read up to 128k, but if it encounters an inter-record gap before the 128k mark, it knows that the block has terminated and it stops reading. Nifty, eh?

"Ripping" data from tapes

While we're talking about dd, I want to quickly show you how to "rip" a file from the tape. Let's say you have a tape that holds two star archives (each terminated with a filemark), and that you used a block size of 64k. Here's how to copy the data from the first file on tape and dump it into a file called filezero.tar:

```
# mt -f /dev/nst0 rewind
# dd if=/dev/nst0 of=/tmp/filezero.tar bs=64k
```

In the absence of a "count=" option, dd will read until the end of a file (or in the case of a

tape, until it encounters a filemark).

The other many uses of dd

dd is a great tape tool, and can also be used to write archives to tape, as well as copy an archive directly from one tape to another. One caveat -- make sure that any archives you copy to tape are not compressed using gzip or bzip2; they should be simple ".tar" files. Otherwise, a single tape read error could prevent you from recovering *any* data from tape. Rely on your tape drive's hardware compression (normally enabled by default) instead.

Section 7. The backup script

The backup philosophy

Since your data is important, it's critical that your backup procedure is simple, consistent, and easy to execute. The best way to ensure a simple, reliable backup procedure is to create a special backup script. Instead of typing in a series of complicated commands every time you need to back up, you simply run the script. Because everything is automated and consistent, using a backup script will help to ensure a perfect backup.

---

backup-tools.sh

To make things even more flexible, I've broken this example script into two separate files. The first file is called *backup-tools.sh* (click the link to view it) and contains various generic backup utility functions to make your life easier. To use it, we'll create a bash script called "backup" that sources backup-tools.sh and performs the backup.

The backup script

Here's a basic "backup" script that creates two archives on tape. The first contains the contents of everything but /home (and /tmp and /proc, which you don't need to back up), and the second contains everything in /home. We'll step through this script line by line:

The backup script: listing

```
#!/bin/bash
TAPEDEV=/dev/nst0
BLOCKSIZE=64k

source backup-tools.sh

bt_backup() {
        #we need a "try" specially for star since it returns non-zero sometimes
        echo ">>> full system backup of" `hostname`"..."
gt;&2
        try mt -f ${TAPEDEV} rewind
        echo "    >>> rewinding tape" >&2
        #root filesystem first, to ease restore
        try cd /
        echo "    >>> backing up root filesystem" >&2
        #can't use absolute pathnames with the cd; . approach
        star -cv bs=${BLOCKSIZE} fs=16m -V pat=home/\* -V pat=tmp/\* -V pat=pro
        echo "    >>> backing up /home tree" >&2
        try cd /home
        star -cv bs=${BLOCKSIZE} fs=16m -f ${TAPEDEV} .
        echo ">>> done." >&2
        echo >&2
}

bt_backup > backup.log
```

```
bt_eject
```

The explanation

In our backup script, we first define two self-documenting environment variables, TAPEDEV and BLOCKSIZE. Then, we source backup-tools.sh, which defines "try" and "bt_eject" functions, sets the tape drive into variable block size mode, and uses the special bash "trap" command to allow you to immediately abort the entire script by pressing ^C (without it, only the currently-executing command would abort). backup-tools.sh requires that the TAPEDEV and BLOCKSIZE variables are already defined.

The explanation, part 2

After backup-tools.sh is sourced, we define a bash function, bt_backup(). This function isn't immediately executed (we call it later), but it contains all the commands required to back up our files, detect possible errors, and keep informed of what's going on.

The explanation, part 3

If you take a quick look at the function, you'll notice that we're redirecting a lot of output to stderr ("&>&2"). We do this to separate these informational messages (to stderr) from the verbose backup log that the star commands spit out (to stdout). If you look at the second-to-last line in the script, you'll see that when we call bt_backup, we redirect stdout (the star output) to a backup log. But the informational messages (including any possible star error messages and summary reports) end up on the console. This way, we're able to create a backup log while at the same time getting informational output on the console.

bt_backup() internals

Now, let's look at the function internals. You'll notice that we call "mt" in an interesting way:

```
try mt -f ${TAPEDEV} rewind
```

You've seen the "mt" part before, but what's with the "try" prefix? try is actually a bash function that's defined in backup-tools.sh, and you can think of it as an automatic error-detection function. Here's how it works: try will execute its first argument ("mt") and pass its remaining arguments to mt. When "mt" returns, "try" looks at its error return code. If mt returns zero, all is well and try will return successfully. However, if mt returns a non-zero error code, then try will abort the script immediately.

bt_backup() internals, continued

We use "try" to abort execution of the script when a critical operation fails, Here's an

example. As I mentioned earlier, when we source backup-tools.sh, the tape drive gets set to variable block size automatically. However, it's only possible to set a drive into variable block mode when there's a tape in the drive; if there isn't, the attempt will be unsuccessful. Thankfully, backup-tools.sh uses "try" to gracefully exit when the "mt -f ${TAPEDEV} setblk 0" command fails:

```
# ./backup
IBM developerWorks Backup Tools 1.0
Distributed under the GPL

&gt;&gt;&gt; setting variable tape block size
/dev/nst0: Input/output error

!!! ERROR: the mt command did not complete successfully.
!!! ("mt -f /dev/nst0 setblk 0")
!!! Since this is a critical task, I'm stopping.
```

Why try?

Without "try", we'd experience a lot of messy failing star commands. Checking error codes adds an additional level of consistency to our backup, which is a good idea since a system backup is such a critical operation. You'll also notice that we *don't* use "try" with star. This is because star will exit with an error even if it was unable to back up just a single file out of thousands (possibly due to a permission problem, filesystem corruption, or a change in file size). Since this does not necessarily indicate an error (and star is able to complete the backup anyway), it doesn't make sense to automatically abort the backup script when star returns an error code.

Star control

Fortunately, in such situations, star will print an informational message to stderr. Here's an example of a situation where star can't stat a particular file: if you're running the backup script as root (as you should), then this is a sign of filesystem corruption:

```
# ./backup
IBM developerWorks Backup Tools 1.0
Distributed under the GPL

&gt;&gt;&gt; setting variable tape block size
&gt;&gt;&gt; star will use a block size of 64k
&gt;&gt;&gt; full system backup of cvs.gentoo.org...
&gt;&gt;&gt; rewinding tape
&gt;&gt;&gt; backing up root filesystem
star: No such file or directory. Cannot stat 'home/cvsroot/gentoo-x86/kde-i18n/
```

The rest of bt_backup()

You should be familiar with the rest of the bt_backup() function (we've covered it earlier). The

only thing that's probably new to you is the "fs=16m" option; this sets the fifo size used by star to 16Mb and is strictly a performance-tuning parameter. After the bt_backup() function is defined, we execute bt_backup() (redirecting stdout to our backup log) and then eject the tape by calling bt_eject().

Section 8. Backup...and restore

Running it

Once you've used my example "backup" script to create a backup script of your own, move it to a safe location (I recommend creating a /root/backup directory and storing both your "backup" script and backup-tools.sh in there.) Then, insert a tape, run the script, and verify that the backup is working. Check the backup log and make sure that it backed up everything it should have. As you're fixing your "backup" script to get it "just so", it's a good idea to verify that the data was actually stored on tape correctly. You can display the contents as shown on the next panel.

A simple verify

```
# mt -f /dev/nst0 setblk 0
# mt -f /dev/nst0 asf 0
# star -tv -f bs=[blocksize] /dev/nst0
```

This will check the first archive on tape. Substitute "asf 0" with "asf 1" to read the second archive, etc. Once your backup script is working perfectly, consider adding it to a cron job to automate your backup process. The script's output will be sent to you as an e-mail, so that you can quickly scan it for any possible problems.

Congratulations, your backup is working perfectly! Now, for something just as important, the restore!

The...restore!

Backing up is only half of the equation, the second half being the restore, which is just as (maybe more) important. Please do yourself a favor and read this entire section carefully, even if you consider yourself a tar "pro". Refamiliarize yourself with the often-forgotten "-p" option. Without it, your full filesystem restore will have the wrong directory permissions. But first, let's cover the easier stuff.

Single file restores

Restoring single files is easy. First, set variable block mode and advance to the file you want to restore from:

```
# mt -f /dev/nst0 setblk 0
# mt -f /dev/nst0 asf 0
```

We're now ready to restore a file from the first archive. To restore the file /bin/bash, type:

```
# star -xv bs=[blocksize] -f /dev/nst0 bin/bash
```

Single notes

A couple of things to note: first, notice that we used a *relative* path (no leading slash) when we specified the path to /bin/bash -- this is because we used a relative path (".") to tell star what files should be included in the archive. I recommend that you use relative paths when you create all your archives (consistency is a good thing). Just remember to always do a "cd /foo; star -cf ." rather than a "star -cf /foo", and you'll be OK.

Single notes, continued

Here's another thing: when bash is extracted, it will be placed in bin/bash (relative to the current directory). So, if you ran star in /tmp, your extracted file will be found at /tmp/bin/bash.

Another important point: due to the design of the tar archive format, when you restore a single file from tape, star will scan the entire archive, *even if your specified file is found right at the beginning* of the tarball. To speed things up, it's safe to abort star with a ^C after it lets you know that your file has been extracted successfully.

Perfect full restores

When you need to restore the contents of an entire archive to a filesystem, there's one extremely important thing you need to remember: the "-p" option. If you're like me, you extract the contents of source tarballs using tar on a daily basis, and to do this it's perfectly fine to type "tar xzvf foo.tar.gz". However, while "tar xvf" and "star -xv -f" are fine for extracting a bunch of files into your home directory, if you extract the contents of a filesystem using only these options, then your directory permissions will be messed up. Here's how to do a perfect filesystem restore. First, the basics on the next panel.

Perfect full restores

```
# mt -f /dev/nst0 setblk 0
# mt -f /dev/nst0 asf 0
# cd /mnt/newroot
```

In this example, we want to extract the contents of the first tape archive to /mnt/newroot (presumably a new mounted filesystem). Then:

```
# star -xvp bs=[blocksize] -f /dev/nst0
```

The "-p" option instructs star (and tar) to ignore the umask for setting directory permissions and simply set the perms as recorded in the archive. Don't forget "-p" or you'll be sorry!

The bare metal recovery

A "bare metal recovery" is what you need to do when you lose everything, due to a hard drive crash or massive system failure. If you're ready for a massive system failure, then you're ready for almost anything. The key to the bare metal recovery is having a good, tested Linux rescue disk or CD handy. What you'll need on it is shown on the following panel.

The rescue disk/CD checklist

These things should be on your rescue disk/CD:

*   An up-to-date   kernel
*   Kernel support for your tape device and controller
*   Kernel support for your filesystems and disk controllers
*   Up-to-date   versions of the star and mt commands **(important!)**
*   Tools to create filesystems --   mkfs.ext2, mkreiserfs, etc.
*   If you use LVM or software RAID --   kernel support and userspace tools
*   fdisk, cfdisk, gnuparted, etc.
*   lilo/GRUB --   to reinstall your boot loader

Putting together the perfect rescue disk can be a time-consuming process, but the results are well worth it. A good rescue disk will allow you to begin the restore within minutes of replacing a dead hard drive. Without a good rescue disk, you could spend hours trying to get your system to recognize your tape drive...not a good thing. With one, you can calmly get the job done, and you'll be ready for (almost) anything!

Section 9. Resources and feedback

Resources

*   Check out the very good "Unix Backup and Recovery", published by O'Reilly. There's lots of good info in the book describing how to perform *database* backups.
*   *http://www.linuxtapecert.org* is an excellent resource for Linux tape drive-related information.
*   Here's *a tarball of the backup scripts* used in this tutorial. Enjoy!

Your feedback

We look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact the author, Daniel Robbins, at *drobbins@gentoo.org* .

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.