

LPI certification 102 (release 2) exam prep, Part 1

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.

1. Before you start	2
2. Shared libraries	4
3. Compiling applications from sources	7
4. Package management concepts	15
5. rpm, the (R)ed Hat (P)ackage (M)anager	16
6. Debian package management	24
7. Summary and resources	30

Section 1. Before you start

About this tutorial

Welcome to "Compiling sources and managing packages," the first of four tutorials designed to prepare you for the Linux Professional Institute's 102 exam. In this tutorial, we'll show you how to compile programs from sources, how to manage shared libraries, and how to use the Red Hat and Debian package management systems.

This tutorial on compiling sources and managing packages is ideal for those who want to learn about or improve their Linux package management skills. This tutorial is particularly appropriate for those who will be setting up applications on Linux servers or desktops. For many readers, much of this material will be new, but more experienced Linux users may find this tutorial to be a great way to "round out" their important Linux system administration skills. If you are new to Linux, we recommend that you start with [Part 1](#) and work through the series from there.

By the end of this *series* of tutorials (eight in all, covering the LPI 101 and 102 exams), you'll have the knowledge you need to become a Linux Systems Administrator and will be ready to attain an LPIC Level 1 certification from the Linux Professional Institute if you so choose.

For those who have taken the [release 1 version](#) of this tutorial for reasons other than LPI exam preparation, you probably don't need to take this one. However, if you do plan to take the exams, you should strongly consider reading this revised tutorial.

The LPI logo is a trademark of Linux Professional Institute.

About the authors

For technical questions about the content of this tutorial, contact the authors:

- Daniel Robbins, at [drobbins@gentoo.org](mailto:d Robbins@gentoo.org)
- Chris Houser, at chouser@gentoo.org
- Aron Griffis, at agriffis@gentoo.org

Daniel Robbins lives in Albuquerque, New Mexico, and is the Chief Architect 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 their daughter, Hadassah.

Chris Houser, known to his friends as "Chouser," has been a UNIX proponent since 1994 when joined the administration team for the computer science network at Taylor University in Indiana, where he earned his Bachelor's degree in Computer Science and Mathematics. Since then, he has gone on to work in Web application programming, user interface design, professional video software support, and now Tru64 UNIX device driver programming at Compaq. He has also contributed to various free software projects, most recently to Gentoo Linux. He lives with his wife and two cats in New Hampshire.

Aron Griffis graduated from Taylor University with a degree in Computer Science and an award that proclaimed him the "Future Founder of a Utopian UNIX Commune". Working towards that goal, Aron is employed by Compaq writing network drivers for Tru64 UNIX, and spending his spare time plunking out tunes on the piano or developing Gentoo Linux. He lives with his wife Amy (also a UNIX engineer) in Nashua, NH.

Section 2. Shared libraries

Introducing shared libraries

On Linux systems there are two fundamentally different types of Linux executable programs. The first are called *statically linked* executables. Static executables contain all the functions that they need to execute -- in other words, they're "complete." Because of this, static executables do not depend on any external library to run.

The second are *dynamically linked* executables. We'll get into those in the next panel.

Static vs. dynamic executables

We can use the `ldd` command to determine if a particular executable program is static:

```
# ldd /sbin/sln
not a dynamic executable
```

"not a dynamic executable" is `ldd`'s way of saying that `sln` is statically linked. Now, let's take a look at `sln`'s size in comparison to its non-static cousin, `ln`:

```
# ls -l /bin/ln /sbin/sln
-rwxr-xr-x  1 root  root      23000 Jan 14 00:36 /bin/ln
-rwxr-xr-x  1 root  root    381072 Jan 14 00:31 /sbin/sln
```

As you can see, `sln` is over *ten* times the size of `ln`. `ln` is so much smaller than `sln` because it is a *dynamic executable*. Dynamic executables are *incomplete* programs that depend on external shared libraries to provide many of the functions that they need to run.

Dynamically linked dependencies

To view a list of all the shared libraries upon which `ln` depends, use the `ldd` command:

```
# ldd /bin/ln
libc.so.6 => /lib/libc.so.6 (0x40021000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

As you can see, `ln` depends on the external shared libraries `libc.so.6` and `ld-linux.so.2`. As a rule, dynamically linked programs are much smaller than their statically-linked equivalents. However, statically-linked programs come in handy for certain low-level maintenance tasks. For example, `sln` is the perfect tool to modify various library symbolic links that exist in `/lib`. But in general, you'll find that nearly all executables on a Linux system are of the dynamically linked variety.

The dynamic loader

So, if dynamic executables don't contain everything they need to run, what part of Linux has the job of loading them along with any necessary shared libraries so that they can execute correctly? The answer is something called the *dynamic loader*, which is actually the `ld-linux.so.2` library that you see listed as a shared library dependency in `ln`'s `ldd` listing. The dynamic loader takes care of loading the shared libraries that dynamically linked executables need in order to run. Now, let's take a quick look at how the dynamic loader finds the appropriate shared libraries on your system.

ld.so.conf

The dynamic loader finds shared libraries thanks to two files -- `/etc/ld.so.conf` and `/etc/ld.so.cache`. If you `cat` your `/etc/ld.so.conf` file, you'll probably see a listing that looks something like this:

```
$ cat /etc/ld.so.conf
/usr/X11R6/lib
/usr/lib/gcc-lib/i686-pc-linux-gnu/2.95.3
/usr/lib/mozilla
/usr/lib/qt-x11-2.3.1/lib
/usr/local/lib
```

The `ld.so.conf` file contains a listing of all directories (besides `/lib` and `/usr/lib`, which are automatically included) in which the dynamic loader will look for shared libraries.

ld.so.cache

But before the dynamic loader can "see" this information, it must be converted into an `ld.so.cache` file. This is done by running the `ldconfig` command:

```
# ldconfig
```

When `ldconfig` completes, you now have an up-to-date `/etc/ld.so.cache` file that reflects any changes you've made to `/etc/ld.so.conf`. From this point forward, the dynamic loader will look in any new directories that you specified in `/etc/ld.so.conf` when looking for shared libraries.

ldconfig tips

To view all the shared libraries that `ldconfig` can "see," type:

```
# ldconfig -p | less
```

There's one other handy trick you can use to configure your shared library paths. Sometimes, you'll want to tell the dynamic loader to try to use shared libraries in a specific directory before trying any of your `/etc/ld.so.conf` paths. This can be handy in situations where you are running an older application that doesn't work with the currently-installed versions of your libraries.

LD_LIBRARY_PATH

To instruct the dynamic loader to check a certain directory first, set the `LD_LIBRARY_PATH` variable to the directories that you would like searched. Separate multiple paths using commas; for example:

```
# export LD_LIBRARY_PATH="/usr/lib/old:/opt/lib"
```

After `LD_LIBRARY_PATH` has been exported, any executables started from the current shell will use libraries in `/usr/lib/old` or `/opt/lib` if possible, falling back to the directories specified in `/etc/ld.so.conf` if some shared library dependencies are still unsatisfied.

We've completed our coverage of Linux shared libraries. To learn more about shared libraries, type `man ldconfig` and `man ld.so`.

Section 3. Compiling applications from sources

Introduction

Let's say you find a particular application that you'd like to install on your system. Maybe you need to run a very recent version of this program, but this most recent version isn't yet available in a packaging format such as rpm. Perhaps this particular application is only available in source form, or you need to enable certain features of the program that are not enabled in the rpm by default.

Whatever the reason, whether of necessity or simply just because you *want* to compile the program from its sources, this section will show you how.

Downloading

Your first step will be to locate and download the sources that you want to compile. They'll probably be in a single archive with a trailing .tar.gz, tar.Z, tar.bz2, or .tgz extension. Go ahead and download the archive with your favorite browser or ftp program. If the program happens to have a Web page, this would be a good time to visit it to familiarize yourself with any installation documentation that may be available.

The program you're installing could depend on the existence of any number of other programs that may or may not be currently installed on your system. If you know for sure that your program depends on other programs or libraries that are not currently installed, you'll need to get these packages installed first (either from a binary package like rpm or by compiling them from their sources also.) Then, you'll be in a great position to get your original source file successfully installed.

Unpacking

Unpacking the source archive is relatively easy. If the name of your archive ends with .tar.gz, .tar.Z, or .tgz, you should be able to unpack the archive by typing:

```
$ tar xzvf archivename.tar.gz
```

(*x* is for extract, *z* is for gzip decompression, *v* is for verbose (print the files that are extracted), and *f* means that the filename will appear next on the command line.)

Nearly all "source tarballs" will create one main directory that contains all the program's sources. This way, when you unpack the archive, your current working directory isn't cluttered with lots of files -- instead, all files are neatly organized in a single, new directory and don't get in the way.

Listing archives

Every now and then, you may come across an archive that, when decompressed, creates tons of files in your current working directory. While most tarballs aren't created this way, it's been known to happen. If you want to verify that your particular tarball was put together correctly and creates a main directory to hold the sources, you can view its contents by typing:

```
$ tar tzvf archivename.tar.gz | more
```

(`t` is for a *text* listing of the archive. No extraction occurs.)

If there is no common directory listed on the left-hand side of the archive listing, you'll want to create a new directory, move the tarball inside it, enter the directory, and only then extract the tarball. Otherwise, you'll be in for a mess!

Unpacking bzip2-compressed archives

It's possible that your archive may be in `.tar.bz2` format. Archives with this extension have been compressed with bzip2. Bzip2 generally compresses significantly better than gzip. Its only disadvantage is that compression and decompression are slower, and bzip2 consumes more memory than gzip while running. For modern computers, this isn't much of an issue, which is why you can expect bzip2 to become more and more popular as time goes on.

Because bzip2 has been gaining popularity, modern versions of GNU tar recognize the `j` option to mean "this tarball is compressed with bzip2." When tar encounters the `j` option, it will auto-decompress the tarball (by calling the "bzip2" program) before it tries to open the tarball. For example, here's the command to view the contents of a `.tar.bz2` file:

```
$ tar tjvf archive.tar.bz2 | less
```

And here is the command to view the contents of a `.tar.gz` file:

```
$ tar tzvf archive.tar.gz | less
```

And here is the command to view the contents of a `.tar` (uncompressed) file:

```
$ tar tvf archive.tar | less
```

bzip2 pipelines

So, your version of tar doesn't recognize those handy bzip2 shortcuts -- what can be done? Fortunately, there's an easy way to extract the contents of bzip2 tarballs that will work on nearly all UNIX systems, even if the system in question happens to have a non-GNU version of tar. To view the contents of a bzip2 file, we can create a shell pipeline:

```
$ cat archive.tar.bz2 | bzip2 -d | tar tvf - | less
```

This next pipeline will actually extract the contents of `archive.tar.bz2`:

```
$ cat archive.tar.bz2 | bzip2 -d | tar xvf -
```

bzip2 pipelines (continued)

In the previous two examples, we created a standard UNIX pipeline to view and extract files from our archive file. Since tar was called with the `-` option, it read tar data from stdin, rather than trying to read data from a file on disk.

If you used the pipeline method to try to extract the contents of your archive and your system complained that bzip2 couldn't be found, it's possible that bzip2 isn't installed on your system. You can download the sources to bzip2 from <http://sources.redhat.com/bzip2>. After installing the bzip2 sources (by following this tutorial), you'll then be able to unpack and install the application you wanted to install in the first place :)

Inspecting sources

Once you've unpacked your sources, you'll want to enter the unpacked directory and check things out. It's always a good idea to locate any installation-related documentation. Typically, this information can be found in a README or INSTALL file located in the main source directory. Additionally, look for README.platform and INSTALL.platform files, where platform is the name of your particular operating system -- in this case "Linux."

Configuration

Many modern sources contain a *configure* script in the main source directory. This script (typically generated by the developers using the GNU autoconf program) is specially designed to set up the sources so that they compile perfectly on your system. When run, the configure script probes your system, determining its capabilities, and creates *Makefiles*, which contain instructions for building and installing the sources on your system.

The configure script is almost always called "configure." If you find a configure script in the main source directory, odds are good that it was put there for your use. If you can't find a configure script, then your sources probably come with a standard Makefile that has been designed to work across a variety of systems -- this means that you can skip the following configuration steps, and resume this tutorial where we start talking about "make."

Using configure

Before running the configure script, it's a good idea to get familiar with it. By typing `./configure --help`, you can view all the various configuration options that are available for your program. Many of the options you see, especially the ones listed at the top of the `--help` printout, are standard options that will be found in nearly every configure script. The options listed near the end are often related to the particular package you're trying to compile.

Take a look at them and note any you'd like to enable or disable.

The --prefix option

Most GNU autoconf-based configure scripts have a `--prefix` option that allows you to control where your program is installed. By default, most sources install into the `/usr/local` prefix. This means that binaries end up in `/usr/local/bin`, man pages in `/usr/local/man`, etc. This is normally what you want; `/usr/local` is commonly used to store programs that you compile yourself.

Using --prefix

If you'd like the sources to install somewhere else, say in `/usr`, you'll want to pass the `--prefix=/usr` option to configure. Likewise, you could also tell configure to install to your `/opt` tree, by using the `--prefix=/opt` option.

What about FHS?

Sometimes, a particular program may default to installing some of its files to non-standard locations on disk. In particular, a source archive may have a number of installation paths that do not follow the Linux Filesystem Hierarchy Standard (FHS). Fortunately, the configure script doesn't just permit the changing of the install prefix, but also allows us to change the install location for various system components such as man pages.

This capability comes in very handy, since most source archives aren't yet FHS-compliant. Often, you'll need to add a `--mandir=/usr/share/man` and a `--infodir=/usr/share/info` to the configure command line in order to make your source package configure itself to eventually install its files in the "correct" locations.

Time to configure

Once you've taken a look at the various configure options and determined which ones you'd like to use, it's time to run configure. Please note that you may not *need* to include any command-line options when you run `configure --` in the majority of situations, the defaults will work (but may not be *exactly* what you want).

Time to configure (continued)

To run configure, type:

```
$ ./configure <options>
```

This could look like:

```
$ ./configure
```

or

```
$ ./configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info --with-threa
```

The options you need will depend on the particular package you're configuring. When you run `configure`, it will spend a minute or two detecting what features or tools are available on your system, printing out the results of its various configuration checks as it goes.

config.cache

Once the configuration process completes, the `configure` script stores all its configuration data in a file called `config.cache`. This file lives in the same directory as the `configure` script itself. If you ever need to run `./configure` again after you've updated your system configuration, make sure you `rm config.cache` first; otherwise, `configure` will simply use the old settings without rechecking your system.

configure and Makefiles

After the `configure` script completes, it's time to compile the sources into a running program. A program called *make* is used to perform this step. If your software package contained a `configure` script, then when you ran it, `configure` created Makefiles that were specially customized for your system. These files tell the `make` program how to build the sources and install the resultant binaries, man pages, and support files.

Makefile intro

Makefiles are typically named `makefile` or `Makefile`. There will normally be one `makefile` in each directory that contains source files, in addition to one that sits in the main source directory. The autoconf-generated Makefiles contain instructions (officially called *rules*) that specify how to build certain *targets*, like the program you want to install. `make` figures out the order in which all the rules should run.

Invoking make

Invoking `make` is easy; just type "make" in the current directory. The `make` program will then find and interpret a file called `makefile` or `Makefile` in the current directory. If you type "make" all by itself, it will build the *default target*. Developers normally set up their `makefiles` so that the default target compiles all the sources:

```
$ make
```

Some makefiles won't have a default target, and you'll need to specify one in order to get the compilation started:

```
$ make all
```

After typing one of these commands, your computer will spend several minutes compiling your program into object code. Presuming it completes with no errors, you'll be ready to install the compiled program onto your system.

Installation

After the program is compiled, there's one more important step: installation. Although the program is compiled, it's not yet ready for use. All its components need to be copied from the source directory to the correct "live" locations on your filesystem. For example, all binaries need to be copied to `/usr/local/bin`, and all man pages need to be installed into `/usr/local/man`, etc.

Before you can install the software, you'll need to become the root user. This is typically done by either logging in as root on a separate terminal or typing `su`, at which point you'll be prompted for root's password. After typing it in, you'll have root privileges until you exit from your current shell session by typing "exit" or hitting control-D. If you're already root, you're ready to go!

make install

Installing sources is easy. In the main source directory, simply type:

```
# make install
```

Typing "make install" will tell make to satisfy the "install" target; this target is traditionally used to copy all the freshly created source files to the correct locations on disk so that your program can be used. If you didn't specify a `--prefix` option, it's very likely that quite a few files and directories will be copied to your `/usr/local` tree. Depending on the size of the program, the install target may take anywhere from several seconds to a few minutes to complete.

In addition to simply copying files, make install will also make sure the installed files have the correct ownership and permissions. After `make install` completes successfully, the program is installed and ready (or *almost* ready) for use!

Once it's installed

Now that your program is installed, what's next? Running it, of course! If you're not familiar with

how to use the program you just installed, you'll want to read the program's man page by typing:

```
$ man programname
```

It's possible that a program may require additional configuration steps. For example, if you installed a Web server, you'll need to configure it to start automatically when your system boots. You may also need to customize a configuration file in /etc before your application will run.

Ta da!

Now that you've fully installed a software package from its sources, you can run it! To start the program, type:

```
$ programname
```

Congratulations!

Possible problems

It's very possible that configure or make, or possibly even make install, aborted with some kind of error code. The next several panels will help you correct common problems.

Missing libraries

Every now and then, you may experience a problem where configure bombs out because you don't have a certain library installed. In order for you to continue the build process, you'll need to temporarily put your current program configuration on hold and track down the sources or binary package for the library that your program needs. Once the correct library is installed, configure or make should be happy and complete successfully.

Other problems

Sometimes, you'll run into some kind of error that you simply don't know how to fix. As your experience with UNIX/Linux grows, you'll be able to diagnose more and more seemingly cryptic error conditions that you encounter during the configure and make process.

Sometimes, errors occur because an installed library is too old (or possibly even too new!). Other times, the problem you're having is actually the fault of the developers, who may not have anticipated their program running on a system such as yours -- or maybe they just made a typo :)

Other problems (continued)

For problems such as these, use your best judgment to determine where to go for help. If this is your first attempt at compiling a program from source, this may be a good time to select another, easier program to compile. Once you get the simpler program compiled, you may have the necessary experience to fix your originally encountered problem. As you continue to learn more about how UNIX works, you'll get closer to the point where you can actually "tweak" Makefiles and sources to get even seemingly flaky code to compile cleanly!

Section 4. Package management concepts

Package management advantages

Beyond building applications from sources, there's another method for installing software on your Linux system. All Linux distributions employ some form of package management for installing, upgrading, and uninstalling software packages. Package management offers clear advantages over installing directly from source:

- Ease of installation and uninstallation
- Ease of upgrading already-installed packages
- Protection of configuration files
- Simple tracking of installed files

Package management disadvantages

Before jumping into instructions for using the most popular package management tools, I'll acknowledge that there are some Linux users who dislike package management. They might propose some of the following downsides:

- Binaries built for a specific system perform better
- Resolving package dependencies is a headache
- Package database corruption can render a system unmaintainable
- Packages are hard to create

There is some truth to these statements, but the general consensus among Linux users is that the advantages outweigh the disadvantages. Additionally, each stumbling block listed above has a corresponding rebuttal: Multiple packages can be built to optimize for different systems; package managers can be augmented to resolve dependencies automatically; databases can be rebuilt based on other files; and the initial effort expended in creating a package is mitigated by the ease of upgrading or removing that package later.

Section 5. rpm, the (R)ed Hat (P)ackage (M)anager

Getting started with rpm

The introduction of Red Hat's rpm in 1995 was a huge step forward for Linux distributions. Not only did it make possible the management of packages on Red Hat Linux, but due to its GPL license, rpm has become the defacto standard for open source packaging.

The rpm program has a command-line interface by default, although there are GUIs and Web-based tools to provide a friendlier interface. In this section we'll introduce the most common command-line operations, using the Xsnow program for the examples. If you would like to follow along, you can download the Xsnow rpm below, which should work on most rpm-based distributions.

- [xsnow-1.41-1.i386.rpm](#)

Note: If you find the various uses of the term "rpm" confusing in this section, keep in mind that "rpm" usually refers to the program, whereas "an rpm" or "the rpm" usually refers to an rpm package.

Installing an rpm

To get started, let's install our Xsnow rpm using `rpm -i`:

```
# rpm -i xsnow-1.41-1.i386.rpm
```

If this command produced no output, then it worked! You should be able to run Xsnow to enjoy a blizzard on your X desktop. Personally, we prefer to have some visual feedback when we install an rpm, so we like to include the `-h` (hash marks to indicate progress) and `-v` (verbose) options:

```
# rpm -ivh xsnow-1.41-1.i386.rpm
xsnow #####
```

Re-installing an rpm

If you were following along directly, you might have seen the following message from rpm in the previous example:

```
# rpm -ivh xsnow-1.41-1.i386.rpm
package xsnow-1.41-1 is already installed
```

There may be occasions when you wish to re-install an rpm, for instance if you were to

accidentally delete the binary `/usr/X11R6/bin/xsnow`. In that case, you should first remove the rpm with `rpm -e`, then re-install it. Note that the information message from rpm in the following example does not hinder the removal of the package from the system:

```
# rpm -e xsnow
removal of /usr/X11R6/bin/xsnow failed: No such file or directory

# rpm -ivh xsnow-1.41-1.i386.rpm
xsnow #####
```

Forcefully installing an rpm

Sometimes removing an rpm isn't practical, particularly if there are other programs on the system that depend on it. For example, you might have installed an "x-amusements" rpm which lists Xsnow as a dependency, so using `rpm -e` to remove Xsnow is disallowed:

```
# rpm -e xsnow
error: removing these packages would break dependencies:
       /usr/X11R6/bin/xsnow is needed by x-amusements-1.0-1
```

In that case, you could re-install Xsnow using the `--force` option:

```
# rpm -ivh --force xsnow-1.41-1.i386.rpm
xsnow #####
```

Installing or removing with --nodeps

An alternative to using `--force` in the previous panel would be to remove the rpm using the `--nodeps` option. This disables rpm's internal dependency checking, and is *not recommended* in most circumstances. Nonetheless, it is occasionally useful:

```
# rpm -e --nodeps xsnow

# rpm -ivh xsnow-1.41-1.i386.rpm
xsnow #####
```

You can also use `--nodeps` when installing an rpm. To re-iterate what was said above, using `--nodeps` is not recommended, however it is sometimes necessary:

```
# rpm -ivh --nodeps xsnow-1.41-1.i386.rpm
xsnow #####
```

Upgrading packages

There is now an rpm of Xsnow version 1.42 on the Xsnow author's Website. You may want to upgrade your existing Xsnow installation for your particular version of Linux. If you were to use `rpm -ivh --force`, it would appear to work, but rpm's internal database would list *both* versions as being installed. Instead, you should use `rpm -U` to upgrade your installation:

```
# rpm -Uvh xsnow-1.42-1.i386.rpm
xsnow #####
```

Here's a little trick: we rarely use `rpm -i` at all, because `rpm -U` will simply install an rpm if it doesn't exist yet on the system. This is especially useful if you specify multiple packages on the command-line, where some are currently installed and some are not:

```
# rpm -Uvh xsnow-1.42-1.i386.rpm xfishtank-2.1tp-1.i386.rpm
xsnow #####
xfishtank #####
```

Querying with rpm -q

You might have noticed in the examples that installing an rpm requires the full filename, but removing an rpm requires only the name. This is because rpm maintains an internal database of the currently installed packages, and you can reference installed packages by name. For example, let's ask rpm what version of Xsnow is installed:

```
# rpm -q xsnow
xsnow-1.41-1
```

In fact, rpm knows even more about the installed package than just the name and version. We can ask for a lot more information about the Xsnow rpm using `rpm -qi`:

```
# rpm -qi xsnow
Name           : xsnow                Relocations: (not relocateable)
Version        : 1.41                Vendor: Dan E. Anderson http://www.dan.dry
Release       : 1                    Build Date: Thu 10 May 2001 01:12:26 AM EDT
Install date: Sat 02 Feb 2002 01:00:43 PM EST   Build Host: danx.drydog.com
Group         : Amusements/Graphics      Source RPM: xsnow-1.41-1.src.rpm
Size          : 91877                  License: Copyright 1984, 1988, 1990, 1993-1
Packager      : Dan E. Anderson http://dan.drydog.com/
URL           : http://www.euronet.nl/~rja/Xsnow/
Summary      : An X Window System based dose of Christmas cheer.
Description  :
The Xsnow toy provides a continual gentle snowfall, trees, and Santa Claus flying his sleigh around the screen on the root window.
Xsnow is only for the X Window System, though; consoles just get coal.
```

Listing files with rpm -ql

The database maintained by rpm contains quite a lot of information. We've already seen that it

keeps track of what versions of packages are installed, and their associated information. It can also list the files owned by a given installed package using `rpm -ql`:

```
# rpm -ql xsnow
/etc/X11/applnk/Games/xsnow.desktop
/usr/X11R6/bin/xsnow
/usr/X11R6/man/man1/xsnow.1x.gz
```

Combined with the `-c` option or the `-d` option, you can restrict the output to configuration or documentation files, respectively. This type of query is more useful for larger rpms with long file lists, but we can still demonstrate using the Xsnow rpm:

```
# rpm -qlc xsnow
/etc/X11/applnk/Games/xsnow.desktop

# rpm -qld xsnow
/usr/X11R6/man/man1/xsnow.1x.gz
```

Querying packages with rpm -qp

If you had the information available with `rpm -qi` *before* installing the package, you might have been able to better decide whether or not to install it. Actually, using `rpm -qp` allows you to query an rpm file instead of querying the database. All of the queries we've seen so far can be applied to rpm files as well as installed packages. Here are all the examples again, this time employing the `-p` option:

```
# rpm -qp xsnow-1.41-1.i386.rpm
xsnow-1.41-1

# rpm -qpi xsnow-1.41-1.i386.rpm
[same output as rpm -qi in the previous panel]

# rpm -qpl xsnow-1.41-1.i386.rpm
/etc/X11/applnk/Games/xsnow.desktop
/usr/X11R6/bin/xsnow
/usr/X11R6/man/man1/xsnow.1x.gz

# rpm -qp1c xsnow-1.41-1.i386.rpm
/etc/X11/applnk/Games/xsnow.desktop

# rpm -qp1d xsnow-1.41-1.i386.rpm
/usr/X11R6/man/man1/xsnow.1x.gz
```

Querying all installed packages

You can query all the packages installed on your system by including the `-a` option. If you pipe the output through `sort` and into a pager, then it's a nice way to get a glimpse of what's installed on your system. For example:

```
# rpm -qa | sort | less
[output omitted]
```

Here's how many rpms we have installed on one of our systems:

```
# rpm -qa | wc -l
287
```

And here's how many files are in all those rpms:

```
# rpm -qa | wc -l
45706
```

Here's a quick tip: Using `rpm -qa` can ease the administration of multiple systems. If you redirect the sorted output to a file on one machine, then do the same on the other machine, you can use the `diff` program to see the differences.

Finding the owner for a file

Sometimes it's useful to find out what rpm owns a given file. In theory, you could figure out what rpm owns `/usr/X11R6/bin/xsnow` (pretend you don't remember) using a shell construction like the following:

```
# rpm -qa | while read p; do rpm -ql $p | grep -q '^/usr/X11R6/bin/xsnow$' && echo $p; done
xsnow-1.41-1
```

Since this takes a long time to type, and even longer to run (1m50s on one of our Pentiums), the rpm developers thoughtfully included the capability in rpm. You can query for the owner of a given file using `rpm -qf`:

```
# rpm -qf /usr/X11R6/bin/xsnow
xsnow-1.41-1
```

Even on the Pentium, that only takes 0.3s to run. And even fast typists will enjoy the simplicity of `rpm -qf` compared to the complex shell construction :)

Showing dependencies

Unless you employ options such as `--nodeps`, rpm normally won't allow you to install or remove packages that break dependencies. For example, you can't install Xsnow without first having the X libraries on your system. Once you have Xsnow installed, you can't remove the X libraries without removing Xsnow first (and probably half of your installed packages).

This is a strength of rpm, even if it's frustrating sometimes. It means that when you install an

rpm, it should just *work*. You shouldn't need to do much extra work, since rpm has already verified that the dependencies exist on the system.

Sometimes when you're working on resolving dependencies, it can be useful to query a package with the `-R` option to learn about everything it expects to be on the system. For example, the Xsnow package depends on the C library, the math library, the X libraries, and specific versions of rpm:

```
# rpm -qpR xsnow-1.41-1.i386.rpm
rpmLib(PayloadFilesHavePrefix) <= 4.0-1
ld-linux.so.2
libX11.so.6
libXext.so.6
libXpm.so.4
libc.so.6
libm.so.6
libc.so.6(GLIBC_2.0)
libc.so.6(GLIBC_2.1.3)
rpmLib(CompressedFileNames) <= 3.0.4-1
```

You can also query the installed database for the same information by omitting the `-p`:

```
# rpm -qR xsnow
```

Verifying the integrity of a package

When you download an rpm from the Web or an ftp site, for the sake of security you may want to verify its integrity before installing. All rpms are "signed" with an MD5 sum. Additionally, some authors employ a PGP or GPG signature to further secure their packages. To check the signature of a package, you can use the `--checksig` option:

```
# rpm --checksig xsnow-1.41-1.i386.rpm
xsnow-1.41-1.i386.rpm: md5 GPG NOT OK
```

Wait a minute! According to that output, the GPG signature is *NOT OK*. Let's add some verbosity to see what's wrong:

```
# rpm --checksig -v xsnow-1.41-1.i386.rpm
xsnow-1.41-1.i386.rpm:
MD5 sum OK: 8ebe63b1dbe86ccd9eaf736a7aa56fd8
gpg: Signature made Thu 10 May 2001 01:16:27 AM EDT using DSA key ID B1F6E46C
gpg: Can't check signature: public key not found
```

So, the problem is that we couldn't retrieve the author's public key. After we retrieve the public key from the [package author's Website](#) (shown in the output from `rpm -qi`), the signature checks out:

```
# gpg --import dan.asc
gpg: key B1F6E46C: public key imported
gpg: /root/.gnupg/trustdb.gpg: trustdb created
gpg: Total number processed: 1
gpg:             imported: 1
```

```
# rpm --checksig xsnow-1.41-1.i386.rpm
xsnow-1.41-1.i386.rpm: md5 gpg OK
```

Verifying an installed package

Similarly to checking the integrity of an rpm, you can also check the integrity of your installed files using `rpm -V`. This step makes sure that the files haven't been modified since they were installed from the rpm:

```
# rpm -V xsnow
```

Normally this command displays no output to indicate a clean bill of health. Let's spice things up and try again:

```
# rm /usr/X11R6/man/man1/xsnow.1x.gz
# cp /bin/sh /usr/X11R6/bin/xsnow
# rpm -V xsnow
S.5...T /usr/X11R6/bin/xsnow
missing /usr/X11R6/man/man1/xsnow.1x.gz
```

This output shows us that the Xsnow binary fails MD5 sum, file size, and mtime tests. And the man page is missing altogether! Let's repair this broken installation:

```
# rpm -e xsnow
removal of /usr/X11R6/man/man1/xsnow.1x.gz failed: No such file or directory
# rpm -ivh xsnow-1.41-1.i386.rpm
xsnow #####
```

Configuring rpm

Rpm rarely needs configuring. It simply works out of the box. In older versions of rpm, you could change things in `/etc/rpmrc` to affect run-time operation. In recent versions, that file has been moved to `/usr/lib/rpm/rpmrc`, and is not meant to be edited by system administrators. Mostly it just lists flags and compatibility information for various platforms (e.g. i386 is compatible with all other x86 architectures).

If you wish to configure rpm, you can do so by editing `/etc/rpm/macros`. Since this is rarely

necessary, we'll let you read about it in the rpm bundled documentation. You can find the right documentation file with the following command:

```
# rpm -qld rpm | grep macros
```

Additional rpm resources

That's all we've got in this tutorial regarding the Red Hat Package Manager. You should have enough information to administer a system, and there are a lot more resources available. Be sure to check out some of these links:

- [Xsnow Website](#)
- [rpm Home Page](#)
- [Maximum RPM - an entire book](#)
- [The RPM HOWTO at the Linux Documentation Project](#)
- [Red Hat's chapter on package management with rpm](#)
- [developerWorks article on creating rpms](#)
- [rpmfind.net -- a huge collection of rpms](#)

Section 6. Debian package management

Introducing apt-get

The Debian package management system is made up of several different tools. The command-line tool apt-get is the easiest way to install new packages. For example, to install the program Xsnow, do this as the root user:

```
# apt-get install xsnow
Reading Package Lists... Done
Building Dependency Tree... Done
The following NEW packages will be installed:
  xsnow
0 packages upgraded, 1 newly installed, 0 to remove and 10 not upgraded.
Need to get 17.8kB of archives. After unpacking 42.0kB will be used.
Get:1 http://ftp-mirror.internap.com stable/non-free xsnow 1.40-6 [17.8kB]
Fetched 17.8kB in 0s (18.4kB/s)
Selecting previously deselected package xsnow.
(Reading database ... 5702 files and directories currently installed.)
Unpacking xsnow (from .../archives/xsnow_1.40-6_i386.deb) ...
Setting up xsnow (1.40-6) ...
```

Skimming through this output, you can see that Xsnow was to be installed, then it was fetched it from the Web, unpacked, and finally set up.

Simulated install

If apt-get notices that the package you are trying to install depends on other packages, it will automatically fetch and install those as well. In the last example, only Xsnow was installed, because all of its dependencies were already satisfied.

Sometimes, however, the list of packages apt-get needs to fetch can be quite large, and it is often useful to see what is going to be installed before you let it start. The `-s` option does exactly this. For example, on one of our systems if we try to install the graphical e-mail program balsa:

```
# apt-get -s install balsa
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  esound esound-common gdk-implib1 gnome-bin gnome-libs-data implib-base libart2
  libaudiofile0 libesd0 libglib1.2 libgnome32 libgnomesupport0 libgnomeui32
  libgnorba27 libgnorbagtk0 libgtk1.2 libjpeg62 liborbit0 libpng2 libproplist0
  libtiff3g libungif3g zlib1g
The following NEW packages will be installed:
  balsa esound esound-common gdk-implib1 gnome-bin gnome-libs-data implib-base
  libart2 libaudiofile0 libesd0 libglib1.2 libgnome32 libgnomesupport0
  libgnomeui32 libgnorba27 libgnorbagtk0 libgtk1.2 libjpeg62 liborbit0 libpng2
  libproplist0 libtiff3g libungif3g zlib1g
```

0 packages upgraded, 24 newly installed, 0 to remove and 10 not upgraded.

It then goes on to list the order in which the packages will be installed and configured (or set up).

Package resource list: apt-setup

Since apt-get is automatically fetching packages for you, it must know something about where to find packages that haven't yet been installed. This knowledge is kept in `/etc/apt/sources.list`. Although you can edit this file by hand (see the `sources.list` man page), you may find it easier to use an interactive tool:

```
# apt-setup
```

This tool walks you through the process of finding places to get Debian packages, such as CDROMs, Web sites, and ftp sites. When you're done, it writes out changes to your `/etc/apt/sources.list` file, so that apt-get can find packages when you ask for them.

From apt-get to dselect

The apt-get tool has many command-line options that you can read about in the apt-get man page. The defaults usually work fine, but if you find yourself using the same option frequently, you may want to add a setting to your `/etc/apt/apt.conf` file. This syntax for this configuration file is described in the `apt.conf` man page.

apt-get also has many other commands besides the `install` command we've used so far. One of these is `apt-get dselect-upgrade`, which obeys the Status set for each package on your Debian system.

Starting dselect

The Status for each package is stored in the file `/var/lib/dpkg/status`, but it is best updated using another interactive tool:

```
# dselect
Debian GNU/Linux `dselect' package handling frontend.

* 0. [A]ccess          Choose the access method to use.
  1. [U]pdate          Update list of available packages, if possible.
  2. [S]elect          Request which packages you want on your system.
  3. [I]ninstall       Install and upgrade wanted packages.
  4. [C]onfig          Configure any packages that are unconfigured.
  5. [R]emove          Remove unwanted software.
  6. [Q]uit            Quit dselect.
```

Move around with ^P and ^N, cursor keys, initial letters, or digits;
Press <enter> to confirm selection. ^L redraws screen.

Version 1.6.15 (i386). Copyright (C) 1994-1996 Ian Jackson. This is
free software; see the GNU General Public License version 2 or later for
copying conditions. There is NO warranty. See dselect --license for details.

Using dselect Select mode

You can view and change each package's Status by choosing the Select option. It will then display a screenful of help. When you're done reading this, press space. Now you will see a list of packages that looks something like this:

```

EIOM Pri Section Package      Inst.ver   Avail.ver  Description
  All packages
    Newly available packages
      New Important packages
        New Important packages in section admin
n* Imp admin   at          <none>     3.1.8-10   Delayed job execution and
n* Imp admin   cron        <none>     3.0p11-57.3 management of regular bac
n* Imp admin   logrotate   <none>     3.2-11     Log rotation utility
        New Important packages in section doc
n* Imp doc     info        <none>     4.0-4      Standalone GNU Info docum
n* Imp doc     manpages    <none>     1.29-2     Man pages about using a L
        New Important packages in section editors
n* Imp editors ed          <none>     0.2-18.1   The classic unix line edi
n* Imp editors nvi         <none>     1.79-16a.1 4.4BSD re-implementation
        New Important packages in section interpreters
n* Imp interpre perl-5.005 <none>     5.005.03-7. Larry Wall's Practical Ex
        New Important packages in section libs
n* Imp libs    libident    <none>     0.22-2     simple RFC1413 client lib
n* Imp libs    libopenldap- <none>     1.2.12-1   OpenLDAP runtime files fo
n* Imp libs    libopenldap1 <none>     1.2.12-1   OpenLDAP libraries.
n* Imp libs    libpcre2    <none>     2.08-1     Philip Hazel's Perl Compa

```

The package Status

The Status for each package can be seen under the somewhat cryptic heading EIOM. The column we care about is under the M character, where each package is marked with one of the following:

To change the Mark, just press the key for the code you want (equal, dash, or underline), but if you want to change the Mark to * (asterisk), you have to press + (plus).

When you are done, use an upper-case Q to save your changes and exit the Select screen. If you need help at any time in dselect, type ? (question mark). Type a space to get back out of a help screen.

Install and Configure (dpkg-reconfigure)

Debian doesn't install or remove packages based on their Status settings until you run something like `apt-get dselect-upgrade`. This command actually does several steps for you at once -- Install, Remove, and Configure. The Install and Remove steps shouldn't need to stop to ask you any questions. The Configure step, however, may ask any number of questions in order to set up the package just the way you want it.

There are other ways to run these steps. For example, you can choose each step individually from the main `dselect` menu.

Some packages use a system called `debconf` for their Configure step. Those that do can ask their setup questions in a variety of ways, such as in a text terminal, through a graphical interface, or through a Web page. To configure one of these packages, use the `dpkg-reconfigure` command. You can even use it to make sure *all* `debconf` packages have been completely configured:

```
# dpkg-reconfigure --all
debconf: package "3c5x9utils" is not installed or does not use debconf
debconf: package "3dchess" is not installed or does not use debconf
debconf: package "9menu" is not installed or does not use debconf
debconf: package "9wm" is not installed or does not use debconf
debconf: package "a2ps" is not installed or does not use debconf
debconf: package "a2ps-perl-ja" is not installed or does not use debconf
debconf: package "aalib-bin" is not installed or does not use debconf
```

This will produce a very long list of packages that do not use `debconf`, but it will also find some that do and present easy-to-use forms for you to answer the questions that each package asks.

Getting the status of an installed package

The Debian package management tools we've reviewed so far are best for handling multi-step operations with long lists of packages. But they don't cover some of the nuts-and-bolts operations of package management. For this kind of work, you want to use `dpkg`.

For example, to get the complete status and description of a package, use the `-s` option:

```
# dpkg -s xsnow
Package: xsnow
Status: install ok installed
Priority: optional
Section: non-free/x11
Installed-Size: 41
Maintainer: Martin Schulze <joe@debian.org>
Version: 1.40-6
Depends: libc6, xlib6g (>= 3.3-5)
```

Description: Brings Christmas to your desktop
Xsnow is the X-windows application that will let it snow on the root window, in between and on windows. Santa and his reindeer will complete your festive-season feeling.

The link between a file and its .deb

Since a .deb package contains files, you would think there would be a way to list the files within the package. Well, you would be right; just use the `-L` option:

```
# dpkg -L xsnow
/.
/usr
/usr/doc
/usr/doc/xsnow
/usr/doc/xsnow/copyright
/usr/doc/xsnow/readme.gz
/usr/doc/xsnow/changelog.Debian.gz
/usr/X11R6
/usr/X11R6/bin
/usr/X11R6/bin/xsnow
/usr/X11R6/man
/usr/X11R6/man/man6
/usr/X11R6/man/man6/xsnow.6.gz
```

To go the other way around, and find which package contains a specific file, use the `-S` option:

```
# dpkg -S /usr/doc/xsnow/copyright
xsnow: /usr/doc/xsnow/copyright
```

The name of the package is listed just to the left of the colon.

Finding packages to install

Usually, apt-get will already know about any Debian package you might need. If it doesn't, you may be able to find the package among these [lists of Debian packages](#), or elsewhere on the Web.

If you do find and download a .deb file, you can install it using the `-i` option:

```
# dpkg -d /tmp/dl/xsnow_1.40-6_i386.deb
```

If you can't find the package you're looking for as a .deb file, but you find a .rpm or some other type of package, you may be able to use *alien*. The alien program can convert packages from various formats into .debs.

Additional Debian package management resources

There is a lot more to the Debian package management system than we covered here. There is also a lot more to Debian than its package management system. The following sites will help round out your knowledge in these areas:

- [*Debian home page*](#)
- [*Debian installation guide*](#)
- [*Lists of Debian packages*](#)
- [*Alien home page*](#)
- [*Guide to creating your own Debian packages*](#)

Section 7. Summary and resources

Summary

Congratulations, you've reached the end of this tutorial! We hope it has helped solidify your knowledge of compiling programs from sources, managing shared libraries, and using the Red Hat and Debian package management systems. Please join us in our next tutorial, entitled "Configuring and compiling the kernel," in which we show you how to modify and compile the Linux kernel from source code. By continuing in this tutorial series, you'll soon be ready to attain your LPIC Level 1 Certification from the Linux Professional Institute.

On the next page, you'll find a number of resources that you will find helpful in learning more about the subjects presented in this tutorial.

Resources

At linuxdoc.org, you'll find linuxdoc's collection of guides, HOWTOs, FAQs and man-pages to be invaluable. Be sure to check out [Linux Gazette](#) and [LinuxFocus](#) as well.

The Linux System Administrators guide, available from [Linuxdoc.org's "Guides" section](#), is a good complement to this series of tutorials -- give it a read! You may also find Eric S. Raymond's [Unix and Internet Fundamentals HOWTO](#) to be helpful.

In the *Bash by example* article series on *developerWorks*, Daniel shows you how to use `bash` programming constructs to write your own `bash` scripts. This `bash` series (particularly Parts 1 and 2) will be excellent additional preparation for the LPIC Level 1 exam:

- [Bash by example, Part 1: Fundamental programming in the Bourne-again shell](#)
- [Bash by example, Part 2: More bash programming fundamentals](#)
- [Bash by example, Part 3: Exploring the `ebuild` system](#)

We highly recommend the [Technical FAQ for Linux users](#) by Mark Chapman, a 50-page in-depth list of frequently asked Linux questions, along with detailed answers. The FAQ itself is in PDF (Acrobat) format. If you're a beginning or intermediate Linux user, you really owe it to yourself to check this FAQ out. We also recommend [Linux glossary for Linux users](#), also from Mark.

If you're not familiar with the `vi` editor, we strongly recommend that you check out Daniel's [Vi intro -- the cheat sheet method tutorial](#). This tutorial will give you a gentle yet fast-paced introduction to this powerful text editor. Consider this must-read material if you don't know how to use `vi`.

Feedback

Please send any tutorial feedback you may have to the authors:

- Daniel Robbins, at [drobbins@gentoo.org](mailto:d Robbins@gentoo.org)
- Chris Houser, at chouser@gentoo.org
- Aron Griffis, at agriffis@gentoo.org

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial [Building tutorials with the Toot-O-Matic](#) demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.