3

Care and feeding of source trees

In Chapter 2, *Unpacking the goodies*, we saw how to create an initial source tree. It won't stay in this form for long. During a port, the source tree is constantly changing:

- Before you can even start, you may apply *patches* to the tree to bring it up to date.
- After unpacking and possibly patching, you may find that you have to clean out junk left behind from a previous port.
- In order to get it to compile in your environment, you perform some form of *configuration*, which modifies the tree to some extent. We'll look at package configuration in Chapter 4, *Package configuration*.
- During compilation, you add many new files to the tree. You may also create new subdirectories.
- After installation, you remove the unneeded files, for example object files and possibly the final installed files.
- After cleaning up, you may decide to archive the tree again to save space on disk.

Modifying the source tree brings uncertainty with it: what is original, what have I modified, how do I remove the changes I have made and get back to a clean, well-defined starting point? In this chapter we'll look at how to get to a clean starting point. Usually this will be the case after you have extracted the source archive, but frequently you need to add patches or remove junk. We'll also look at how to build a tree with sources on CD-ROM, how to recognize the changes you have made and how to maintain multiple versions of your software.

Updating old archives

You don't always need to get a complete package: another possibility is that you might already have an older version of the package. If it is large—again, for example, the GNU C compiler—you might find it better to get *patches* and update the source tree. Strictly speaking, a patch is any kind of modification to a source or object file. In UNIX parlance, it's almost always a *diff*, a file that describes how to modify a source file to produce a newer version. Diffs are almost always produced by the *diff* program, which we describe in Chapter 10, *Where to go from here*, page 144. In our case study, we have *gcc* version 2.5.6 and want to update to 2.5.8. We discover the following files on the file server:

```
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
-rw-rw-r-- 1 117 1001   10753 Dec 12 19:15 gcc-2.5.6-2.5.7.diff.gz
-rw-rw-r-- 1 117 1001   14726 Jan 24 09:02 gcc-2.5.7-2.5.8.diff.gz
-rw-rw-r-- 1 117 1001 5955006 Dec 22 14:16 gcc-2.5.7.tar.gz
-rw-rw-r-- 1 117 1001 5997896 Jan 24 09:03 gcc-2.5.8.tar.gz
226 Transfer complete.
ftp>
```

In other words, we have the choice of copying the two *diff* files *gcc-2.5.6-2.5.7.diff.gz* and *gcc-2.5.7-2.5.8.diff.gz*, a total of 25 kB, and applying them to your source tree, or copying the complete 6 MB archive *gcc-2.5.8.tar.gz*.

Patch

diff files are reasonably understandable, and you can apply the patches by hand if you want, but it's obviously easier and safer to use a program to apply the changes. This is the purpose of *patch*. *patch* takes the output of the program *diff* and uses it to update one or more files. To apply the patch, it proceeds as follows:

1. First, it looks for a file header. If it finds any junk before the file header, it skips it and prints a message to say that it has done so. It uses the file header to recognize the kind of diff to apply.
2. It renames the old file by appending a string to its name. By default, the string is *.orig*, so *foo.c* would become *foo.c.orig*.
3. It then creates a new file with the name of the old file, and copies the old file to the new file, modifying it with the patches as it goes. Each set of changes is called a *hunk*.

The way *patch* applies the patch depends on the format. The most dangerous kind are *ed* style diffs, because there is no way to be sure that the text is being replaced correctly. With context diffs, it can check that the context is correct, and will look a couple of lines in each direction if it doesn't find the old text where it expects it. You can set the number of lines it will look (the *fuzz factor*) with the `-F` flag. It defaults to 2.

If the old version of the file does not correspond exactly to the old version used to make the diff, *patch* may not be able to find the correct place to insert the patch. Except for *ed* format diffs, it will recognize when this happens, and will print an error message and move the corresponding hunk to a file with the suffix *.rej* (for reject).

A typical example are the patches for X11R5. You might start with the sources supplied on the companion CD-ROM to *X Window System Administrator's Guide* by Linda Mui and Eric Pearce. This CD-ROM includes the complete X11R5 sources to patch level 21. At the time of writing, five further patches to X11R5 have been released. To bring the source tree up to patch level 26, you would proceed as follows:

First, *read the header of the patch file*. As we have seen, *patch* allows text before the first file header, and the headers frequently contain useful information. Looking at patch 22, we see:

```
$ gunzip < /cd0/x11r5/fix22.gz | more
                      X11 R5 Public Patch #22
                         MIT X Consortium

To apply this patch:

cd to the top of the source tree (to the directory containing the
"mit" and "contrib" subdirectories) and do:

        patch -p -s < ThisFile

Patch works silently unless an error occurs.  You are likely to get the
following warning messages, which you can ignore:
```

In this example we have used *gunzip* to look at the file directly; we could just as well have used GNU *zcat*. The patch header suggests the flags `-s` and `-p`. The `-s` flag to *patch* tells it to perform its work silently—otherwise it prints out lots of information about what it is doing and why. The `-p` flag is one of the most complicated to use: it specifies the *pathname strip count*, how to treat the directory part of the file names in the header. We'll look at it in more detail in the section *Can't find file to patch* on page 36.

This information is important: *patch* is rather like a chainsaw without a guard, and if you start it without knowing what you are doing, you can make a real mess of its environment. In this case, we should find that the root of our source tree looks like:

```
$ cd /usr/x11r5
$ ls -FC mit
Imakefile     RELNOTES.ms   extensions/   rgb/
LABEL         bug-report    fonts/        server/
Makefile      clients/      hardcopy/     util/
Makefile.ini  config/       include/
RELNOTES.PS   demos/        lib/
RELNOTES.TXT  doc/          man/
...  that looks OK, we're in the right place
$ gunzip < /cd0/x11r5/fix22.gz | patch -p -s
```

We've taken another liberty in this example: since the patch file was on CD-ROM in compressed form, we would have needed to extract it to disk in order to patch the way the file header suggests. Instead, we just *gunzip* directly into the *patch* program.

It's easy to make mistakes when patching. If you try to apply a patch twice, *patch* will notice, but you can persuade it to reapply the patch anyway. In this section, we'll look at the havoc that can occur as a result. In addition, we'll disregard some of the advice in the patch header. This is the way I prefer to do it:

```
$ gunzip < /cd0/x11r5/fix23.gz | patch -p &> patch.log
```

This invocation allows *patch* to say what it has to say (no `-s` flag), but copies both the standard output and the error output to the file *patch.log*, so nothing appears on the screen. You can, of course, pipe the output through the *tee* program, but in practice things happen so fast that any error message will usually run off the screen before you can read it. It certainly would have done so here: *patch.log* had a length of 930 lines. It starts with

```
Hmm...  Looks like a new-style context diff to me...
The text leading up to this was:
--------------------------
|                    Release 5 Public Patch #23
|                         MIT X Consortium
...  followed by the complete header
|Prereq: public-patch-22
```

This last line is one safeguard that *patch* offers to ensure that you are working with the correct source tree. If *patch* finds a *Prereq:* line in the file header, it checks that this text appears in the input file. For comparison, here's the header of *mit/bug-report*:

```
To: xbugs@expo.lcs.mit.edu
Subject: [area]: [synopsis]   [replace with actual area and short description]

VERSION:
    R5, public-patch-22
    [MIT public patches will edit this line to indicate the patch level]
```

In this case, *patch* finds the text. When it does, it prints out the corresponding message:

```
|
|*** /tmp/,RCSt1006225  Tue Mar  9 14:40:48 1993
|--- mit/bug-report     Tue Mar  9 14:37:04 1993
--------------------------
Good.  This file appears to be the public-patch-22 version.
```

This message shows that it has found the text in *mit/bug-report*. The first hunk in any X11 *diff* changes this text (in this case to *public-patch-23*), so that it will notice a repeated application of the patch. Continuing,

```
Patching file mit/bug-report using Plan A...
Hunk #1 succeeded at 2.
Hmm...  The next patch looks like a new-style context diff to me...
The text leading up to this was:
--------------------------
|*** /tmp/,RCSt1005203  Tue Mar  9 13:45:42 1993
|--- mit/lib/X/Imakefile        Tue Mar  9 13:45:45 1993
--------------------------
Patching file mit/lib/X/Imakefile using Plan A...
Hunk #1 succeeded at 1.
Hunk #2 succeeded at 856.
Hunk #3 succeeded at 883.
Hunk #4 succeeded at 891.
Hunk #5 succeeded at 929.
Hunk #6 succeeded at 943.
Hunk #7 succeeded at 968.
Hunk #8 succeeded at 976.
Hmm...  The next patch looks like a new-style context diff to me...
```

This output goes on for hundreds of lines. What happens if you make a mistake and try again?

```
$ gunzip < /cd0/x11r5/fix23.gz | patch -p &> patch.log
This file doesn't appear to be the public-patch-22 version--patch anyway? [n] y
bad choice...
Reversed (or previously applied) patch detected!  Assume -R? [y] RETURN pressed
Reversed (or previously applied) patch detected!  Assume -R? [y] RETURN pressed
Reversed (or previously applied) patch detected!  Assume -R? [y] ^C$
```

The fi rst message is printed because *patch* didn't fi nd the text `public-patch-22` in the fi le (in the previous step, *patch* changed it to read `public-patch-23`). This message also appears in *patch.log*. Of course, in any normal application you should immediately stop and check what's gone wrong. In this case, I make the incorrect choice and go ahead with the patch. Worse still, I entered `RETURN` to the next two prompts. Finally, I came to my senses and hit `CTRL-C`, the interrupt character on my machine, to stop *patch*.

The result of this is that *patch* removed the patches in the fi rst two fi les (the `-R` flag tells *patch* to behave as if the fi les were reversed, which has the same effect as removing already applied patches). I now have the fi rst two fi les patched to patch level 22, and the others patched to patch level 23. Clearly, I can't leave things like this.

Two wrongs don't normally make a right, but in this case they do. We do it again, and what we get this time looks pretty much the same as the time before:

```
$ gunzip < /cd0/x11r5/fix23.gz | patch -p &> mit/patch.log
Reversed (or previously applied) patch detected!  Assume -R? [y] ^C$
```

In fact, this time things went right, as we can see by looking at *patch.log*:

```
|*** /tmp/,RCSt1006225  Tue Mar  9 14:40:48 1993
|--- mit/bug-report     Tue Mar  9 14:37:04 1993
--------------------------
Good.  This file appears to be the public-patch-22 version.
Patching file mit/bug-report using Plan A...
Hunk #1 succeeded at 2.
Hmm...  The next patch looks like a new-style context diff to me...
The text leading up to this was:
--------------------------
|*** /tmp/,RCSt1005203  Tue Mar  9 13:45:42 1993
|--- mit/lib/X/Imakefile        Tue Mar  9 13:45:45 1993
--------------------------
Patching file mit/lib/X/Imakefile using Plan A...
Hunk #1 succeeded at 1.
(lots of hunks succeed)
Hmm...  The next patch looks like a new-style context diff to me...
The text leading up to this was:
--------------------------
|*** /tmp/d03300        Tue Mar  9 09:16:46 1993
|--- mit/lib/X/Ximp/XimpLCUtil.c        Tue Mar  9 09:16:41 1993
--------------------------
Patching file mit/lib/X/Ximp/XimpLCUtil.c using Plan A...
Reversed (or previously applied) patch detected!  Assume -R? [y]
```

This time the fi rst two fi les have been patched back to patch level 23, and we stop before doing any further damage.

Hunk #3 failed

Patch makes an implicit assumption that the patch was created from an identical source tree. This is not always the case—you may have changed something in the course of the port. The differences frequently don't cause problems if they are an area unrelated to the patch. In this example, we'll look at how things can go wrong. Let's consider the following situation: during a previous port of X11R5 pl 22,* you ran into some problems in *mit/lib/Xt/Selection.c* and fi xed them. The original text read:

```
if (XtWindow(widget) == window)
  XtAddEventHandler(widget, mask, TRUE, proc, closure);
else {
  Widget w = XtWindowToWidget(dpy, window);
  RequestWindowRec *requestWindowRec;
  if (w != NULL && w != widget) widget = w;
  if (selectWindowContext == 0)
      selectWindowContext = XUniqueContext();
```

You had problems with this section, so you commented out a couple of lines:

```
if (XtWindow(widget) == window)
    XtAddEventHandler(widget, mask, TRUE, proc, closure);
else {
/* This doesn't make any sense at all - ignore
 * Widget w = XtWindowToWidget(dpy, window); */
    RequestWindowRec *requestWindowRec;
    /* if (w != NULL && w != widget) widget = w; */
    if (selectWindowContext == 0)
        selectWindowContext = XUniqueContext();
```

Back in the present, you try to apply patch 24 to this fi le:

```
$ gunzip < /cd0/x11r5/fix24.gz | patch -p &> mit/patch.log
$
```

So far so good. But in *patch.log* we fi nd

```
|*** /tmp/da4854       Mon May 17 18:19:57 1993
|--- mit/lib/Xt/Selection.c     Mon May 17 18:19:56 1993
--------------------------
Patching file mit/lib/Xt/Selection.c using Plan A...
Hunk #1 succeeded at 1.
Hunk #2 succeeded at 70.
Hunk #3 failed at 361.
Hunk #4 succeeded at 1084.
Hunk #5 succeeded at 1199.
1 out of 5 hunks failed--saving rejects to mit/lib/Xt/Selection.c.rej
```

What does this mean? There's nothing for it but to look at the fi les concerned. In *fi x24* we fi nd

* The abbreviation *pl* is frequently used to mean *patch level*.

```
*** /tmp/da4854 Mon May 17 18:19:57 1993
--- mit/lib/Xt/Selection.c      Mon May 17 18:19:56 1993
***************
*** 1,4 ****
this must be hunk 1
! /* $XConsortium: Selection.c,v 1.74 92/11/13 17:40:46 converse Exp $ */

  /***********************************************************
  Copyright 1987, 1988 by Digital Equipment Corporation, Maynard, Massachusetts,
--- 1,4 ----
! /* $XConsortium: Selection.c,v 1.78 93/05/13 11:09:15 converse Exp $ */

  /***********************************************************
  Copyright 1987, 1988 by Digital Equipment Corporation, Maynard, Massachusetts,
***************
*** 70,75 ****
--- 70,90 ----
this must be hunk 2
   Widget w;                  /* unused */
***************
*** 346,359 ****
and this must be hunk 3, the one that failed
  {
      Display *dpy = req->ctx->dpy;
      Window window = req->requestor;
!     Widget widget = req->widget;
...  etc
***************
*** 1068,1073 ****
--- 1084,1096 ----
hunk 4
***************
*** 1176,1181 ****
--- 1199,1213 ----
and hunk 5--at least the count is correct
```

patch put the rejects in *Selection.c.rej*. Let's look at it:

```
***************
*** 346,359 ****
  {
      Display *dpy = req->ctx->dpy;
      Window window = req->requestor;
!     Widget widget = req->widget;

      if (XtWindow(widget) == window)
!       XtAddEventHandler(widget, mask, TRUE, proc, closure);
      else {
-       Widget w = XtWindowToWidget(dpy, window);
        RequestWindowRec *requestWindowRec;
-       if (w != NULL && w != widget) widget = w;
        if (selectWindowContext == 0)
            selectWindowContext = XUniqueContext();
        if (XFindContext(dpy, window, selectWindowContext,
--- 361,375 ----
  {
      Display *dpy = req->ctx->dpy;
      Window window = req->requestor;
!     Widget widget = XtWindowToWidget(dpy, window);

+     if (widget != NULL) req->widget = widget;
+     else widget = req->widget;
+
      if (XtWindow(widget) == window)
!       XtAddEventHandler(widget, mask, False, proc, closure);
      else {
        RequestWindowRec *requestWindowRec;
        if (selectWindowContext == 0)
            selectWindowContext = XUniqueContext();
        if (XFindContext(dpy, window, selectWindowContext,
```

The characters `+` and `-` at the beginning of the lines in this hunk identify it as a *unified context diff*. We'll look at them in more detail in Chapter 10, *Where to go from here*, page 147. Not surprisingly, they are the contents of hunk 3. Because of our fix, *patch* couldn't find the old text and thus couldn't process this hunk. In this case, the easiest thing to do is to perform the fix by hand. To do so, we need to look at the partially fixed file that *patch* created, *mit/lib/Xt/Selection.c*. The line numbers have changed, of course, but since hunk 3 wasn't applied, we find exactly the same text as in *mit/lib/Xt/Selection.c.orig*, only now it starts at line 366. We can effectively replace it by the "after" text in *Selection.c.rej*, remembering of course to remove the indicator characters in column 1.

Can't find file to patch

Sometimes you'll see a message like:

```
$ patch -p <hotstuff.diff &>patch.log
Enter name of file to patch:
```

One of the weaknesses of the combination of *diff* and *patch* is that it's easy to get the file names out of sync. What has probably happened here is that the file names don't agree with your source tree. There are a number of ways for this to go wrong. The way that *patch* treats the file names in diff headers depends on the `-p` flag, the so-called *pathname strip count*:

- If you omit the `-p` flag, *patch* strips all directory name information from the file names and leaves just the filename part. Consider the following diff header:

  ```
  *** config/sunos4.h~   Wed Feb 29 07:13:57 1992
  --- config/sunos4.h    Mon May 17 18:19:56 1993
  ```

  Relative to the top of the source tree, the file is in the directory *config*. If you omit the `-p` flag, *patch* will look for the file *sunos4.h*, not *config/sunos4.h*, and will not find it.

- If you specify `-p`, *patch* keeps the complete names in the headers.

- If you specify `-p`*n*, *patch* will remove the first *n* directory name components in the pathname. This is useful when the diffs contain incorrect base path names. For example, you may find a diff header which looks like:

```
*** /src/freesoft/gcc-patches/config/sunos4.h~  Wed Feb 29 07:13:57 1992
--- /src/freesoft/gcc-patches/config/sunos4.h   Mon May 17 18:19:56 1993
```

Unless your source tree also happens to be called */src/freesoft/gcc-patches*, *patch* won't be able to find the files if you use the `-p` flag with no argument. Assuming that you are in the root directory of the package (in other words, the parent directory of *config*), you really don't want to know about the */src/freesoft/gcc-patches/* component. This pathname consists of four parts: the leading / making the pathname absolute, and the three directory names *src*, *freesoft* and *gcc-patches*. In this case, you can enter

```
$ patch -p4 <hotstuff.diff &>patch.log
```

The `-p4` tells *patch* to ignore the first four pathname components, so it would read thes filenames just as *config/sunos4.h~* and *config/sunos4.h*.

In addition to the problem of synchronizing the path names, you may run into broken diffs which don't specify pathnames, even though the files belong to different directories. We'll see how easy it is to make this kind of mistake in Chapter 10, *Where to go from here*, page . For example, you may find that the diff headers look like:

```
*** sunos4.h~   Wed Feb 29 07:13:57 1992
--- sunos4.h    Mon May 17 18:19:56 1993
```

This kind of diff is a real nuisance: you at least need to search for the file *sunos4.h*, and if you're unlucky you'll find more than one and have to examine the patches to figure out which one is intended. Then you need to give this name to the prompt, and *patch* should perform the patches. Unfortunately, in a large collection of diffs, this can happen dozens of times.

I can't seem to find a patch in there

Sometimes you will get what looks like a perfectly good unified context diff, but when you run *patch* against it, you get a message:

```
$ patch <diffs
Hmm...  I can't seem to find a patch in there anywhere.
$
```

Some versions of *patch* don't understand unified diffs, and since all versions skip anything they don't understand, this could be the result. The only thing for it is to get a newer version of *patch*—see Appendix E, *Where to get sources*, for details.

Malformed patch

If *patch* finds the files and understands the headers, you could still run into problems. One of the most common is really a problem in making the diffs:

```
$ patch <diffs
Hmm...  Looks like a unified diff to me...
The text leading up to this was:
--------------------------
```

```
|--- real-programmers.ms~       Wed Dec  7 13:17:47 1994
|+++ real-programmers.ms        Wed Dec  7 14:53:19 1994
--------------------------
Patching file real-programmers.ms using Plan A...
Hunk #1 succeeded at 1.
Hunk #2 succeeded at 54.
patch: **** malformed patch at line 398:  No newline at end of file
```

Well, it tells you what happened: *diff* will print this message if the last character in a file is not `\n`. Most versions of *patch* don't like the message. You need to edit the *diff* and remove the offending line.

Debris left behind by patch

At the end of a session, *patch* leaves behind a number of files. Files of the form *filename.orig* are the original versions of patched files. The corresponding *filename*s are the patched versions. The length of the suffix may be a problem if you are using a file system with a limited filename length; you can change it (perhaps to the `emacs` standard suffix ~) with the `-b` flag. In some versions of *patch*, ~ is the default.

If any patches failed, you will also have files called *filename.rej* (for "rejected"). These contain the hunks that *patch* could not apply. Another common suffix for rejects is #. Again, you can change the suffix, this time with the `-r` flag. If you have any *.rej* files, you need to look at them and find out what went wrong. It's a good idea to keep the *.orig* files until you're sure that the patches have all worked as indicated.

Pruning the tree

Making clean distribution directories is notoriously difficult, and there is frequently irrelevant junk in the archive. For example, all *emacs* distributions for at least the last 6 years have included a file *etc/COOKIES*. As you might guess from the name, this file is a recipe for cookies, based on a story that went round Usenet years ago. This file is not just present in the source tree: since the whole subdirectory *etc* gets installed when you install *emacs*, you end up installing this recipe as well. This particular directory contains a surprising number of files, some of them quite amusing, which don't really have much to do with *emacs*.

This is a rather extreme case of a common problem: you don't need some of the files on the distribution, so you could delete them. As far as I know, *emacs* works just as well without the cookie recipe, but in many cases, you can't be as sure. In addition, you might run into other problems: the GNU General Public License requires you to be prepared to distribute the *complete* contents of the source tree if so requested. You may think that it's an accident that the cookie recipe is in the source tree, but in fact it's a political statement[*], and you are *required* by the terms of the GNU General Public License to keep the file in order to give it to anybody who might want it.

* To quote the beginning of the file: *Someone sent this in from California, and we decided to extend our campaign against information hoarding to recipes as well as software. (Recipes are the closest thing, not involving computers, to software.)*

This is a rather extreme example, but you might find any of the following in overgrown trees:

- Old objects, editor backups and core dumps from previous builds. They may or may not go away with a *make clean*.
- Test programs left behind by somebody trying to get the thing to work on his platform. These probably will not go away with a *make clean*.
- Formatted documentation. Although the *Makefile* should treat documents like objects when cleaning the tree, a surprising number of packages format and install documentation, and then forget about it when it comes to tidying it away again.
- Old mail messages, only possibly related to the package. I don't know why this is, but mail messages seem to be the last thing anybody wants to remove, and so they continue to exist for years in many trees. This problem seems to be worse in proprietary packages than in free packages.

The old objects are definitely the worst problem: *make* can't tell that they don't belong to this configuration, and so they just prevent the correct version of the object being built. Depending on how different the architectures are, you may even find that the bogus objects fool the linker, too, and you run into bizarre problems when you try to execute.

Save the cleaned archive

If you had to go to any trouble (patches or cleanup) to get to a clean starting point for the port, *save the cleaned archive*. You won't need it again, of course, but Murphy's law will ensure that if you don't save it, you *will* need it again.

Handling trees on CD-ROM

It's convenient to have your source tree on CD-ROM: you save disk space, and you can be sure that you don't accidentally change anything. Unfortunately, you also can't deliberately change anything. Normal *Makefiles* expect to put their objects in the source tree, so this complicates the build process significantly.

In the next two sections, we'll look at a couple of techniques that address this problem. Both use symbolic links.

Link trees

You can simulate a writeable tree on disk by creating symbolic links to the sources on CD-ROM. This way, the sources remain on the CD-ROM, but the objects get written to disk. From your viewpoint, it looks as if all the files are in the same directory. For example, assume you have a CD-ROM with a directory */cd0/src/find* containing the sources to *find*:

```
$ ls -FC /cd0/src/find
COPYING         Makefile        config.status*  lib/
COPYING.LIB     Makefile.in     configure*      locate/
ChangeLog       NEWS            configure.in    man/
```

```
INSTALL        README         find/          xargs/
```

The / at the end of the file names indicate that these files are directories; the * indicates that they are executables. You could create a link tree with the following commands:

```
$ cd /home/mysrc/find      put the links here
$ for i in /cd0/src/find/*; do
>  ln -s $i .
> done
$ ls -l              see what we got
total 16
lrwxrwxrwx COPYING -> /cd0/src/find/COPYING
lrwxrwxrwx COPYING.LIB -> /cd0/src/find/COPYING.LIB
lrwxrwxrwx ChangeLog -> /cd0/src/find/ChangeLog
lrwxrwxrwx INSTALL -> /cd0/src/find/INSTALL
lrwxrwxrwx Makefile -> /cd0/src/find/Makefile
lrwxrwxrwx Makefile.in -> /cd0/src/find/Makefile.in
lrwxrwxrwx NEWS -> /cd0/src/find/NEWS
lrwxrwxrwx README -> /cd0/src/find/README
lrwxrwxrwx config.status -> /cd0/src/find/config.status
lrwxrwxrwx configure -> /cd0/src/find/configure
lrwxrwxrwx configure.in -> /cd0/src/find/configure.in
lrwxrwxrwx find -> /cd0/src/find/find
lrwxrwxrwx lib -> /cd0/src/find/lib
lrwxrwxrwx locate -> /cd0/src/find/locate
lrwxrwxrwx man -> /cd0/src/find/man
lrwxrwxrwx xargs -> /cd0/src/find/xargs
```

I omitted most of the information that is printed by *ls -l* in order to get the information on the page: what interests us here is that all the files, including the directories, are symbolic links. In some cases, this is what we want: we don't need to create copies of the directories on the hard disk when a single link to a directory on the CD-ROM does it just as well. In this case, unfortunately, that's not the way it is: our sources are in the directory *find*, and that's where we will have to write our objects. We need to do the whole thing again for the subdirectory *find*:

```
$ cd ~mysource/find       change to the source directory on disk
$ rm find                  get rid of the directory symlink
$ mkdir find               and make a directory
$ cd find                  and change to it
$ for i in /cd0/src/find/find/*; do
>  ln -s $i .
> done
$ ls -l
total 18
lrwxrwxrwx Makefile -> /cd0/src/find/find/Makefile
lrwxrwxrwx Makefile.in -> /cd0/src/find/find/Makefile.in
lrwxrwxrwx defs.h -> /cd0/src/find/find/defs.h
lrwxrwxrwx find -> /cd0/src/find/find/find
lrwxrwxrwx find.c -> /cd0/src/find/find/find.c
lrwxrwxrwx fstype.c -> /cd0/src/find/find/fstype.c
lrwxrwxrwx parser.c -> /cd0/src/find/find/parser.c
lrwxrwxrwx pred.c -> /cd0/src/find/find/pred.c
lrwxrwxrwx tree.c -> /cd0/src/find/find/tree.c
lrwxrwxrwx util.c -> /cd0/src/find/find/util.c
lrwxrwxrwx version.c -> /cd0/src/find/find/version.c
```

Yes, this tree really does have a directory called *find/find/find*, but we don't need to worry about it. Our sources and our *Makefile* are here. We should now be able to move back to the top-level directory and perform the *make*:

```
$ cd ..
$ make
```

This is a relatively simple example, but it shows two important aspects of the technique:

- You don't need to create a symlink for every single file. Although symlinks are relatively small—in this case, less than 100 bytes—they occupy up to 1024 bytes of disk space per link, and you can easily find yourself taking up a megabyte of space just for the links.
- On the other hand, you *do* need to make all the directories where output from the build process is stored. You need to make symlinks to the existing files in these directories.

An additional problem with this technique is that many tools don't test whether they have succeeded in creating their output files. If they try to create files on CD-ROM and don't notice that they have failed, you may get some strange and misleading error messages later on.

Object links on CD-ROM

Some CD-ROMs, notably those derived from the Berkeley Net/2 release, have a much better idea: the CD-ROM already contains a symlink to a directory where the object files are stored. For example, the FreeBSD 1.1 CD-ROM version of *find* is stored on */cd0/filesys/usr/src/usr.bin/find* and contains:

```
total 106
drwxrwxr-x   2 bin    2048 Oct 28  1993 .
drwxrwxr-x 153 bin   18432 Nov 15 23:28 ..
-rw-rw-r--   1 bin     168 Jul 29  1993 Makefile
-rw-rw-r--   1 bin    3157 Jul 29  1993 extern.h
-rw-rw-r--   1 bin   13620 Sep  7  1993 find.1
-rw-rw-r--   1 bin    5453 Jul 29  1993 find.c
-rw-rw-r--   1 bin    4183 Jul 29  1993 find.h
-rw-rw-r--   1 bin   20736 Sep  7  1993 function.c
-rw-rw-r--   1 bin    3756 Oct 17  1993 ls.c
-rw-rw-r--   1 bin    3555 Jul 29  1993 main.c
-rw-rw-r--   1 bin    3507 Jul 29  1993 misc.c
lrwxrwxr-x   1 root     21 Oct 28  1993 obj -> /usr/obj/usr.bin/find
-rw-rw-r--   1 bin    7766 Jul 29  1993 operator.c
-rw-rw-r--   1 bin    4657 Jul 29  1993 option.c
-rw-rw-r--   1 root   2975 Oct 28  1993 tags
```

All you have to do in this case is to create a directory called */usr/obj/usr.bin/find*. The *Makefiles* are set up to compile into that directory.

Tracking changes to the tree

The most obvious modification that you make to a source tree is the process of building: the compiler creates object files* and the loader creates executables. Documentation formatters may produce formatted versions of the source documentation, and possibly other files are created as well. Whatever you do with these files, you need to recognize which ones you have created and which ones you have changed. We'll look at these aspects in the following sections.

Timestamps

It's easy enough to recognize files that have been added to the source tree since its creation: since they are all newer than any file in the original source tree, the simple command *ls -lt* (probably piped into *more* or *less*) will display them in the reverse order in which they were created (newest first) and thus separate the new from the old.

Every UNIX file and directory has three timestamps. The file system represents timestamps in the `time_t` format, the number of seconds elapsed since January 1, 1970 UTC. See Chapter 16, *Timekeeping*, page 270, for more details. The timestamps are:

- The *last modification* timestamp, updated every time the file or directory is modified. This is what most users think of as *the* file timestamp. You can display it with the *ls -l* command.
- The *last access* timestamp, updated every time a data transfer is made to or from the file. You can display it with the *ls -lu* command. This timestamp can be useful in a number of different places.
- The *status change* timestamp (at least, that's what my header files call it). This is a sort of kludged† last modification timestamp for the inode, that part of a file which stores information about the file. The most frequent changes which don't affect the other timestamps are change in the number of links or the permissions, which normally isn't much use to anybody. On the other hand, the inode also contains the other timestamps, so if this rule were enforced rigidly, a change to another timestamp would also change the status change timestamp. This would make it almost completely useless. As a result, most implementations suppress the change to the status change timestamp if only the other timestamps are modified. If you want, you can display the status change timestamp with the *ls -lc* command.

Whichever timestamp you choose to display with *ls -l*, you can cause *ls* to sort by it with the `-t` flag. Thus, *ls -lut* displays and sorts by the last access timestamp.

Of these three timestamps, the last modification timestamp is by far the most important. There are a number of reasons for this:

* To be pedantic, usually the assembler creates the object files, not the compiler.

† A *kludge* is a programming short cut, usually a nasty, untidy one. The *New Hacker's Dictionary* goes to a lot of detail to explain the term, including why it should be spelt *kluge* and not *kludge*.

- *make* relies on the last modification timestamp to decide what it needs to compile. If you move the contents of a directory with *cp*, it changes all the modification timestamps to the time when the copy was performed. If you then type *make*, you will perform a significant amount of needless compilation.
- It's frequently important to establish if two files are in fact the same, in other words, if they have identical content. In the next section we'll see some programmatic tools that help us with this, but as a first approximation we can assume that two files with the same name, length and modification timestamp have an identical content, too. The modification timestamp is the most important of these three: you can change the name, but if length and timestamp are the same, there's still a good chance it's the same file. If you change the timestamp, you can't rely on the two files being the same just because they have the same name and length.
- As we have seen above, the last modification timestamp is useful for sorting when you list directories. If you're looking for a file you made the week before last, it helps if it is dated accordingly.

Keeping timestamps straight

Unfortunately, it's not as easy to keep timestamps straight. Here are some of the things that can go wrong:

- If you copy the file somewhere else, traditional versions of *cp* always set the modification timestamp to the time of copying. *ln* does not, and neither does *mv* if it doesn't need to make a physical copy, so either of these are preferable. In addition, more modern versions of *cp* offer the flag `-p` (preserve), which preserves the modification timestamp and the permissions.
- When extracting an archive, *cpio*'s default behaviour is to set the modification timestamp to the time of extraction. You can avoid this with the `-m` flag to *cpio*.
- Editing the file changes the modification timestamp. This seems obvious, but you frequently find that you make a modification to a file to see if it solves a problem. If it doesn't help, you edit the modification out again, leaving the file exactly as it was, except for the modification timestamp, which points to right now. A better strategy is to save the backup file, if the editor keeps one, or otherwise to rename the original file before making the modifications, then to rename it back again if you decide not to keep the modifications.
- In a network, it's unusual for times to be exactly the same. UNIX machines are not very good at keeping the exact time, and some gain or lose as much as 5 minutes per day. This can cause problems if you are using NFS. You edit your files on one machine, where the clocks are behind, and compile on another, where the clocks are ahead. The result can be that objects created before the last edit still have a modification timestamp that is more recent, and *make* is fooled into believing that it doesn't need to recompile. Similar problems can occur when one system is running with an incorrect time zone setting.

cmp

A modification timestamp isn't infallible, of course: even if EOF, timestamp and name are identical, there still can be a lingering doubt as to whether the files really are identical. This doubt becomes more pronounced if you seee something like:

```
$ ls -l
total 503
-rw-rw-rw-   1 grog     wheel        1326 May  1 01:00 a29k-pinsn.c
-rw-rw-rw-   1 grog     wheel       28871 May  1 01:00 a29k-tdep.c
-rw-rw-rw-   1 grog     wheel        4259 May  1 01:00 a68v-nat.c
-rw-rw-rw-   1 grog     wheel        4515 May  1 01:00 alpha-nat.c
-rw-rw-rw-   1 grog     wheel       33690 May  1 01:00 alpha-tdep.c
...  etc
```

It's a fairly clear bet that somebody has done a *touch* on all the files, and their modification timestamps have all been set to midnight on May 1.* The *cmp* program can give you certainty:

```
$ cmp foo.c ../orig/foo.c          compare with the original
$ echo $?                    show exit status
0                                 0: all OK
$ cmp bar.c ../orig/bar.c
bar.c ../orig/bar.c differ: char 1293, line 39
$ echo $?                          show exit status
1                               1: they differ
```

Remember you can tell the shell to display the exit status of the previous command with the shell variable `$?`. In the C shell, the corresponding variable is called `$status`. If the contents of the files are identical, *cmp* says nothing and returns an exit status 0. If they are, it tells you where they differ and returns 1. You can use the exit status in a shell script. For example, the following Bourne shell script (it doesn't work with *csh*) compares files that are in both the current tree (which is the current working directory) and the original tree (*../orig*) and makes a copy of the ones that have changed in the directory *../changed*.

```
$ for i in *; do                          check all files in the directory
>   if [ -f ../orig/$i ]; then            it is present in the orig tree
>     cmp $i ../orig/$i 2>&1 >/dev/null   compare them
>     if [ $? -ne 0 ]; then               they're different
>       cp -p $i ../changed               make a copy
>     fi
>   fi
> done
```

There are a couple of points to note about this example:

- We're not interested in where the files differ, or in even seeing the message. We just want to copy the files. As a result, we copy both *stdout* and *stderr* of *cmp* to */dev/null*, the UNIX bit bucket.

* Midnight? That looks like 1 a.m. But remember that UNIX timestamps are all in UTC, and that's 1 a.m. in my time zone. This example really *was* done with *touch*.

• When copying, we use `-p` to ensure that the timestamps don't get changed again.

An example—updating an existing tree

Chances are that before long you will have an old version of *gcc* on your system, but that you will want to install a newer version. As we saw on page 29, the gzipped archive for *gcc* is around 6 MB in size, whereas the patches run to 10 KB or 15 KB, so we opt to get *diffs* from `prep.ai.mit.edu` to update version 2.6.1 to 2.6.3. That's pretty straightforward if you have enough disk space: we can duplicate the complete source tree and patch it. Before doing so, we should check the disk space: the *gcc* source tree with all objects takes up 110 MB of disk space.

```
$ cd /porting/srcmove to the parent directory
$ mkdir gcc-2.6.3        make a directory for the new tree
$ cd gcc-2.6.1           move to the old directory
$ tar cf - . | (cd ../gcc-2.6.3;tar xf -) and copy all files*
$ cd ../gcc-2.6.3        move to new directory
$ make clean                     and start off with a clean slate
$ gunzip < /C/incoming/gcc-2.6.1-2.6.2.tar.gz | patch -p | tee patch.log
Hmm...  Looks like a new-style context diff to me...
The text leading up to this was:
--------------------------
|Changes for GCC version 2.6.2 from version 2.6.1:
|
|Before applying these diffs, go to the directory gcc-2.6.1.  Remove all
|files that are not part of the distribution with the command
|
|    make distclean
|
|Then use the command
|
|    patch -p1
|
|feeding it the following diffs as input.  Then rename the directory to
|gcc-2.6.2, re-run the configure script, and rebuild the compiler.
|
|diff -rc2P -x c-parse.y -x c-parse.c -x c-parse.h -x c-gperf.h -x cexp.c -x
bi-parser.c -x objc-parse.y -x objc-parse.c
|-x TAGS -x gcc.?? -x gcc.??s -x gcc.aux -x gcc.info* -x cpp.?? -x cpp.??s -x
cpp.aux -x cpp.info* -x cp/parse.c -x cp/pa
|rse.h gcc-2.6.1/ChangeLog gcc-2.6.2/ChangeLog
|*** gcc-2.6.1/ChangeLog Tue Nov  1 21:32:40 1994
|--- gcc-2.6.2/ChangeLog Sat Nov 12 06:36:04 1994
--------------------------
File to patch:
```

Oops, these patches contain the directory name as well. As the diff header indicates, we can solve this problem by supplying the `-p1` flag to *patch*. We can also solve the problem by

* When moving directories with *tar*, it may not seem to be important whether you say `tar c .` or `tar c *`--but it is. If you say `*`, you will miss out any file names starting with `.` (period).

moving up one level in the directory hierarchy, since we have stuck to the same directory names. This message also reminds us that *patch* is very verbose, so this time we enter:

```
$ gunzip < /C/incoming/gcc-2.6.1-2.6.2.tar.gz | patch -p1 -s | tee patch.log
1 out of 6 hunks failed--saving rejects to cccp.c.rej
$
```

What went wrong here? Let's take a look at *cccp.c.rej* and *cccp.c.orig*. According to the hunk, line 3281 should be

```
                if (ip->macro != 0)
```

The hunk wants to change it to

```
                if (output_marks)
```

However, line 3281 of *cccp.orig* is

```
                if (output_marks)
```

In other words, we had already applied this change, probably from a message posted in `gnu.gcc.bugs`. Although the patch failed, we don't need to worry: all the patches had been applied.

Now we have a *gcc-2.6.2* source tree in our directory. To upgrade to 2.6.3, we need to apply the next patch:

```
$ gunzip < /C/incoming/gcc-2.6.2-2.6.3.diff.gz | patch -p1 -s | tee -a patch.log
```

We use the `-a` option to patch here to keep both logs—possibly overkill in this case. This time there are no errors.

After patching, there will be a lot of original files in the directory, along with the one *.rej* file. We need to decide when to delete the *.orig* files: if something goes wrong compiling one of the patched files, it's nice to have the original around to compare. In our case, though, we have a complete source tree of version 2.6.2 on the same disk, and it contains *all* the original files, so we can remove the *.orig* files:

```
$ find . -name "*.orig" -print | xargs rm
```

We use *xargs* instead of `-exec rm {} \;` because it's faster: `-exec rm` starts a *rm* process for every file, whereas *xargs* will put as many file names onto the *rm* command line as possible. After cleaning up the tree, we *back it up*. It's taken us a while to create the tree, and if anything goes wrong, we'd like to be able to restore it to its initial condition as soon as possible.