9

Installation

Finally the package has been built and tested, and it works. Up to this point, everything in the package has been in the private source tree where it has been built. Most packages are not intended to be executed from the source directory: before we can use them, we need to move the component parts to their intended directories. In particular:

- We need to put executables where a normal PATH environment variable will find them.
- We need to place on-line documentation in the correct directories in a form that the document browser understands.
- The installed software needs to be given the correct permissions to do what it has to do: all executables need to have their execute permissions set, and some programs may need *setuid* or *setgid* bits set (see Chapter 12, *Kernel dependencies*, page). In addition, software will frequently be installed in directories to which normal users have no access. In these cases, the install must be done by *root*.
- Library routines and configuration files need to be installed where the package expects them: the location could be compiled into the programs, or an environment variable could point to the location.
- If the package uses environment variables, you may also need to update *.profile* and *.cshrc* files to add or modify environment variables.
- Many packages—for example, news transfer programs—create data in specific directories. Although initially there may be no data to install, the install process may need to create the directories.
- At some future date, you may want to remove the package again, or to install an updated version. The installation routines should make some provision for removing the package when you no longer want it.

Real-life packages differ significantly in their ability to perform these tasks. Some *Makefile*s consider that their job is done when the package has been compiled, and leave it to you do install the files manually. In some cases, as when there is only a single program, this is no hardship, but it does require that you understand exactly what you need to install. On the other hand, very few packages supply an `uninstall` target.

In this chapter, we'll look at the following subjects:

- The way *Makefile*s typically install software.
- Alternatives if the *Makefile* doesn't do everything it should do.
- How to install documentation.
- How to keep track of installed software.
- How to remove installed software.

Installation is an untidy area. At the end of this chapter, you'll probably be left with a feeling of dissatisfaction—this area has been sadly neglected, and there just aren't enough good answers.

make install

The traditional way to install a pre-compiled package is with *make install*. Typically, it performs the following functions:

- It creates the necessary directories if they are not there.
- It copies all necessary files to their run-time locations.
- It sets the permissions of the files to their correct values. This frequently requires you to be *root* when you install the package. If you don't have root access, you should at least arrange for access to the directories into which you want to install.
- It may strip debug information from executables.

Some other aspects of *make install* are less unified:

- *make install* may imply a *make all*: you can't install until you have made the package, and you'll frequently see an install target that starts with

  ```
  install:  all
          installation commands
  ```

- On the other hand, *make install* may not only expect the *make all* to be completed—and fail if it is not—but remove the executables after installation. Sometimes this is due to the use of BSD *install* without the `-c` option—see the section on the *install* program below—but it means that if you want to make a change to the program after installation, you effectively have to repeat the whole build. Removing files from the tree should be left to *make clean* (see Chapter 5, *Building the package*, page 63).
- Some install targets install man pages or other on-line documentation, others leave it to a separate target with a name like `install-man`, and yet other *Makefile*s completely ignore online documentation, even if the package supplies it.

Configuring the installed package

Some packages have run-time configuration files that need to be set up before the package will run. Also, it's not always enough just to install the files in the correct place and with the correct permissions: you may need to modify the individual user's environment before they can use the package. Here are some examples:

- *sendmail*, the Internet mail transport agent, has an extremely complicated configuration file *sendmail.cf* which needs to be set up to reflect your network topology. A description of how to set up this file takes up hundreds of pages in *sendmail*, by Bryan Costales, Eric Allman and Neil Rickert.
- Many X11 clients have supplementary files that define *application defaults*, which may or may not be suitable for your environment. They are intended to be installed in a directory like */usr/X11/lib/X11/app-defaults*. Not all *Imakefile*s perform this task.
- The path where the executables are installed should be in your `PATH` environment variable.
- If you install man pages, the path should be in your `MANPATH` environment variable.
- Many packages define their own environment variables. For example, TEX defines the environment variables `TEXCONFIG`, `TEXFONTS`, `TEXFORMATS`, `TEXINPUTS`, and `TEXPOOL` to locate its data files.
- Some programs require a setup file in the home directory of each user who uses the program. Others do not require it, but will read it if it is present.
- Some programs will create links with other names. For example, if you install *pax*, the portable archive exchange program, you have the opportunity of creating links called *tar* and *cpio*. This is really a configuration option, but the *Makefile* for *pax* does not account for it.

Typical *Makefile*s are content with moving the files to where they belong, and leave such details to the user. We'll see an alternative on page 138.

Installing the correct files

At first, installation seems straightforward enough: you copy the files to where they belong, and that's that. In practice, a number of subtle problems can occur. There's no hard and fast solution to them, but if you run into trouble it helps to understand the problem.

To replace or not to replace?

Throughout the build process, we have used *make* to decide whether to rebuild a target or not: if the target exists, and is newer than any of its dependencies, it will not be rebuilt. Traditionally, installation is different: the files are installed anyway, even if newer files are already present in the destination directory.

The reasons for this behaviour are shrouded in time, but may be related to the fact that both *install* (which we will discuss below) and *cp* traditionally modify the time stamps of the files, so that the following scenario could occur:

1. Build version 1 of a package, and install it.
2. Start building version 2, but don't complete it.
3. Make a modification to version 1, and re-install it.
4. Complete version 2, and install it. Some of the file in version 2 were compiled before version 1 was re-installed, and are thus older than the installed files. As a result, they will not be installed, and the installed software will be inconsistent.

It's obviously safer to replace everything. But is that enough? We'll look at the opposite problem in the next section.

Updating

Frequently you will install several versions of software over a period of time, as the package evolves. Simply installing the new version on top of the old version will work cleanly only if you can be sure that you install a new version of every file that was present in the old version: otherwise some files from the old version will remain after installation. For example, version 1.07.6 of the GNU *libc* included a file *include/sys/bitypes.h*, which is no longer present in version 1.08.6. After installing version 1.08.6, *include/sys/bitypes.h* is still present from the earlier installation.

The correct way to handle this problem is to *uninstall* the old package before installation. For reasons we will investigate on page 133, this seldom happens.

install

install is a program that is used in installing software. It performs the tasks of creating any necessary directories, copying files, stripping executables, and setting permissions.

install originated in Berkeley, and older System V systems don't support it. It's a fairly trivial program, of course, and many packages supply a script with a name like *install.sh* which performs substantially the same functions. The source is available in the 4.4BSD Lite distribution—see Appendix E, *Where to get sources*.

Although *install* is a relatively simple program, it seems that implementors have done their best to make it differ from one system to the next. The result is a whole lot of incompatible and just downright confusing options. System V.4 even supplies two different versions with conflicting options, a BSD compatible one and a native one—the one you get depends on your other preferences, as laid down in your PATH environment variable.

System V.4 native *install* is sufficiently different from the others that we need to look at it separately—it can install only a single file. The syntax is:

```
install options file [dir dir ...]
```

If the *dir*s are specified, they are appended to the fixed list of directories */bin*, */usr/bin*, */etc*, */lib*, and */usr/lib*. *install* will search the resultant list of directories sequentially for a file with the name *file*. If it finds one, it will replace it and print a message stating in which directory it has installed the file. The `-i` option tells *install* to omit the standard directories and take only the list specified on the command line.

Other versions of *install* have a syntax similar to *mv* and *cp*, except that they take a number of supplementary options:

```
install options file1 file2
install options file1 ...  fileN dir
```

The first form installs *file1* as *file2*, the second form installs *file1* through *fileN* in the directory *dir*.

Table 9-1 contains an overview of *install* options:

Table 9−1: install options

option	Purpose
`-c`	In BSD, copy the file. If this option is not specified, the file is moved (the original file is deleted after copying). In GNU and System V.4 (BSD compatibility), this option is ignored. Files are always copied.
`-c` *dir*	System V.4 native: install the file in directory *dir* only if the file does not already exist. If the file exists already, exit with an error message.
`-d`	In GNU and SunOS, create all necessary directories if the target directory does not exist. Not available in BSD. This lets you create the directory with the command `install -d [-g group] [-m perm] [-o owner] dir`
`-f` *flags*	In 4.4BSD, specify the target's file flags. This relates to the *chflags* program introduced with 4.4BSD—see the man page *usr.bin/chflags/chflags.1* in the 4.4BSD Lite distribution.
`-f` *dir*	System V.4 native: force the file to be installed in *dir*. This is the default for other versions.
`-g` *group*	Set the group ownership to *group*.
`-i`	System V.4 native: ignore the default directory list (see below). This is not applicable with the `-c` or `-f` options.
`-m` *perm*	Set the file permissions to *perm*. *perm* may be in octal or symbolic form, as defined for *chmod(1)*. By default, *perm* is 0755 (`rwxr-xr-x`).

Table 9−1: install options (continued)

option	Purpose
`-n` *dir*	System V.4 native: if *file* is not found in any of the directories, install it in *dir*.
`-o`	System V.4 native: if *file* is already present at the destination, rename the old version by prepending the letters `OLD` to the filename. The old file remains in the same directory.
`-o` *owner*	All except System V.4 native: change the owner to *owner*.
`-s`	System V.4 native: suppress error messages.
`-s`	All except System V.4 native: strip the final binary.
`-u` *owner*	System V.4 native: change the owner to *owner*.

Other points to note are:

- *install* attempts to prevent you from moving a file onto itself.
- Installing */dev/null* creates an empty file.
- *install* exits with a return code of 0 if successful and 1 if unsuccessful.

System V.4 install is definitely the odd man out: if you can avoid it, do. Even Solaris 2 supplies only the BSD version of *install*. On the other hand, pure BSD *install* also has its problems, since it requires the `-c` option to avoid removing the original files.

Installing documentation

Installing man pages would seem to be a trivial exercise. In fact, a number of problems can occur. In this section, we'll look at problems you might encounter installing man pages and GNU info.

Man pages.

As we saw in Chapter 7, *Documentation*, page 99, there is not much agreement about naming, placing, or format of man pages. In order to install man pages correctly you need to know the following things:

- The name of the man directory.
- The naming convention for man files. As we saw, these are many and varied.
- Whether the man pages should be formatted or not.
- If the man pages should be formatted, which formatter should be used? Which macros should be used? This may seem like a decision to be made when building the package, but many *Makefile*s put off this operation to the install phase.
- Whether the man pages should be packed, compressed or zipped.

Typically, this information is supplied in the *Makefile* like this example from the electronic mail reader *elm*, which is one of the better ones:

```
FORMATTER       =       /usr/ucb/nroff
MAN             =       /opt/man/man1
MANEXT          =       .1
CATMAN          =       /opt/man/cat1
CATMANEXT       =       .1
TBL             =       /usr/ucb/tbl
MANROFF         =       /usr/ucb/nroff
SUFFIX          =       .Z
PACKED          =       y
PACKER          =       /bin/compress

# List of installed man pages (except for wnemail.1 - handled differently)
MAN_LIST        =       $(MAN)/answer$(MANEXT)          \
                        $(MAN)/autoreply$(MANEXT)       \
...etc
# List of installed catman pages (except for wnemail.1 - handled differently)
CATMAN_LIST     =       $(CATMAN)/answer$(CATMANEXT)$(SUFFIX)   \
                        $(CATMAN)/autoreply$(CATMANEXT)$(SUFFIX)   \
...etc

# List of formatted pages for catman
FORMATTED_PAGES_LIST =  catman/answer$(CATMANEXT)$(SUFFIX)      \
                        catman/autoreply$(CATMANEXT)$(SUFFIX)   \
...etc

# Targets
all:
        @if $(TEST) '$(CATMAN)' != none; then $(MAKE) formatted_pages ; \
          else true ; fi

formatted_pages: catman $(FORMATTED_PAGES_LIST)

catman:
        mkdir catman

install: $(LIB_LIST)
        @if $(TEST) '$(MAN)' != none; then $(MAKE) install_man ; \
           else true ;  fi
        @if $(TEST) '$(CATMAN)' != none; then $(MAKE) install_catman ; \
           else true ;  fi

install_man: $(MAN_LIST) $(MAN)/wnewmail$(MANEXT)

install_catman: $(CATMAN_LIST) $(CATMAN)/wnewmail$(CATMANEXT)$(SUFFIX)

#       Dependencies and rules for installing man pages and lib files
$(MAN)/answer$(MANEXT): answer.1
                                $(CP) $? $@
                                $(CHMOD) u=rw,go=r $@
```

```
$(MAN)/autoreply$(MANEXT):      autoreply.1
                                $(CP) $? $@
                                $(CHMOD) u=rw,go=r $@
```

This *Makefile* is in the subdirectory *doc*, which is concerned only with documentation, so all the targets relate to the man pages. The target `all` makes the decision whether to format the pages or not based on the value of the make variable `CATMAN`. If this is set to the special value `none`, the *Makefile* does not format the pages.

The target `install` uses the same technique to decide which man pages to install: if the variable `MAN` is not set to `none`, the sources of the man pages are copied there, and if `CATMAN` is not set to `none`, the formatted pages are installed there. This *Makefile* does not use *install*: it performs the operations with *cp* and *chmod* instead.

GNU info

Installing GNU *info* is somewhat more straightforward, but it is also not as clean as it could be:

- *info* is always formatted, so you need the formatter, a program called *makeinfo*, which is part of the *texinfo* package. Before you can run *makeinfo*, you need to port *texinfo*. It's not that big a job, but it needs to be done. Of course, in order to completely install *texinfo*, you need to format the documentation with *makeinfo*—a vicious circle. The solution is to port the *texinfo* executables, then port *makeinfo*, and then format the *texinfo* documentation.
- All *info* files are stored in a single directory with an index file called *dir*. This looks like:

```
-*- Text -*-
This is the file /opt/info/dir, which contains the topmost node of the
Info hierarchy.  The first time you invoke Info you start off
looking at that node, which is (dir)Top.

File: dir       Node: Top       This is the top of the INFO tree
  This (the Directory node) gives a menu of major topics.
  Typing "d" returns here, "q" exits, "?" lists all INFO commands, "h"
  gives a primer for first-timers, "mTexinfo<Return>" visits Texinfo topic,
  etc.

  Note that the presence of a name in this list does not necessarily
  mean that the documentation is available.  It is installed with the
  package in question.  If you get error messages when trying to access
  documentation, make sure that the package has been installed.
  --- PLEASE ADD DOCUMENTATION TO THIS TREE.  (See INFO topic first.) ---

* Menu: The list of major topics begins on the next line.

* Bash: (bash).            The GNU Bourne Again SHell.
* Bfd: (bfd).              The Binary File Descriptor Library.
* Bison: (bison).          The Bison parser generator.
* CL: (cl).                Partial Common Lisp support for Emacs Lisp.
```

```
...etc
```

The lines at the bottom of the example are menu entries for each package. They have a syntax which isn't immediately apparent—in particular, the sequence `* item:` has a special significance in emacs *info* mode. Programs that supply *info* documentation *should* supply such an entry, but many of them do not, and none of them install the line in *dir*—you need to do this by hand.

Removing installed software

For a number of reasons, you may want to remove software that you have already installed:

- You may decide you don't need the software.
- You may want to replace it with a newer version, and you want to be sure that the old version is gone.
- You may want to install it in a different tree.

If you look for a *remove* or *uninstall* target in the *Makefile*, chances are that you won't find one. Packages that supply a remove target are very rare. If you want to remove software, and you didn't take any precautions when you installed it, you have to do it manually with the computer equivalent of an axe and a spear: *ls* and *rm*.

Removing software manually

In fact, it's frequently not *that* difficult to remove software manually. The modification time-stamps of all components are usually within a minute or two of each other, so *ls* with the `-lt` options will list them all together. For example, let's consider the removal of *ghostscript*.

The first step is to go back to the *Makefile* and see what it installed:

```
prefix = /opt
exec_prefix = $(prefix)
bindir = $(exec_prefix)/bin
datadir = $(prefix)/lib
gsdatadir = $(datadir)/ghostscript
mandir = $(prefix)/man/man1
...skipping
install: $(GS)
        -mkdir $(bindir)
        for f in $(GS) gsbj gsdj gslj gslp gsnd bdftops font2c \
           ps2ascii ps2epsi; \
        do $(INSTALL_PROGRAM) $$f $(bindir)/$$f ; done
        -mkdir $(datadir)
        -mkdir $(gsdatadir)

        for f in README gslp.ps gs_init.ps gs_dps1.ps gs_fonts.ps gs_lev2.ps \
        gs_statd.ps gs_type0.ps gs_dbt_e.ps gs_sym_e.ps quit.ps Fontmap \
        uglyr.gsf bdftops.ps decrypt.ps font2c.ps impath.ps landscap.ps \
        level1.ps prfont.ps ps2ascii.ps ps2epsi.ps ps2image.ps pstoppm.ps\
```

```
showpage.ps type1ops.ps wrfont.ps ; \
  do $(INSTALL_DATA) $$f $(gsdatadir)/$$f ; done

-mkdir $(docdir)
for f in NEWS devices.doc drivers.doc fonts.doc hershey.doc \
history.doc humor.doc language.doc lib.doc make.doc ps2epsi.doc \
psfiles.doc readme.doc use.doc xfonts.doc ; \
  do $(INSTALL_DATA) $$f $(docdir)/$$f ; done
-mkdir $(mandir)
for f in ansi2knr.1 gs.1 ; do $(INSTALL_DATA) $$f $(mandir)/$$f ; done
-mkdir $(exdir)
for f in chess.ps cheq.ps colorcir.ps golfer.ps escher.ps \
snowflak.ps tiger.ps ; \
  do $(INSTALL_DATA) $$f $(exdir)/$$f ; done
```

One alternative is to make a remove target for this *Makefile*, which isn't too difficult in this case:

- First, copy the *install* target and call it *remove*.
- Move the *mkdir* lines to the bottom and change them to *rmdir*. You'll notice that this *Makefile* accepts the fact that *mkdir* can fail because the directory already exists (the `-` in front of *mkdir*). We'll do the same with *rmdir*: if the directory isn't empty, *rmdir* fails, but that's OK.
- We replace `$(INSTALL_PROGRAM) $$f` and `$(INSTALL_DATA) $$f` with `rm -f`.

The result looks like:

```
remove: $(GS)
        for f in $(GS) gsbj gsdj gslj gslp gsnd bdftops font2c \
           ps2ascii ps2epsi; \
        do rm -f $(bindir)/$$f ; done

        for f in README gslp.ps gs_init.ps gs_dps1.ps gs_fonts.ps gs_lev2.ps \
        gs_statd.ps gs_type0.ps gs_dbt_e.ps gs_sym_e.ps quit.ps Fontmap \
        uglyr.gsf bdftops.ps decrypt.ps font2c.ps impath.ps landscap.ps \
        level1.ps prfont.ps ps2ascii.ps ps2epsi.ps ps2image.ps pstoppm.ps\
        showpage.ps type1ops.ps wrfont.ps ; \
          do rm -f $(gsdatadir)/$$f ; done

        for f in NEWS devices.doc drivers.doc fonts.doc hershey.doc \
        history.doc humor.doc language.doc lib.doc make.doc ps2epsi.doc \
        psfiles.doc readme.doc use.doc xfonts.doc ; \
          do rm -f $(docdir)/$$f ; done
        for f in ansi2knr.1 gs.1 ; do $(INSTALL_DATA) $$f $(mandir)/$$f ; done
        for f in chess.ps cheq.ps colorcir.ps golfer.ps escher.ps \
        snowflak.ps tiger.ps ;
          do rm -f $(exdir)/$$f ; done
        -rmdir $(bindir)
        -rmdir $(datadir)
        -rmdir $(gsdatadir)
        -rmdir $(docdir)
        -rmdir $(mandir)
```

```
        -rmdir $(exdir)
```

More frequently, however, you can't use this approach: the *Makefile* isn't as easy to find, or you have long since discarded the source tree. In this case, we'll have to do it differently. First, we find the directory where the executable *gs*, the main *ghostscript* program, is stored:

```
$ which gs
/opt/bin/gs
```

Then we look at the last modification timestamp of */opt/bin/gs*:

```
$ ls -l /opt/bin/gs
-rwxrwxr-x   1 root     wheel     3168884 Jun 18 14:29 /opt/bin/gs
```

This is to help us to know where to look in the next step: we list the directory */opt/bin* sorted by modification timestamp. It's a lot easier to find what we're looking for if we know the date. If you don't have *which*, or possibly even if you do, you can use the following script, called *wh*:

```
for j in $*; do
  for i in `echo $PATH|sed 's/:/ /g'`; do
    if [ -f $i/$j ]; then
      ls -l $i/$j
    fi
  done
done
```

wh searches the directories in the current environment variable PATH for a specific file and lists all occurrences in the order in which they appear in PATH in *ls -l* format, so you could also have entered:

```
$ wh gs
-rwxrwxr-x   1 root     wheel     3168884 Jun 18 14:29 /opt/bin/gs
```

Once we know the date we are looking for, it's easy to list the directory, page it through *more* and find the time frame we are looking for.

```
$ ls -lt /opt/bin|more
total 51068
-rw-------   1 root     bin        294912 Sep  6 15:08 trn.old
-rwxr-xr-x   1 grog     lemis      106496 Sep  6 15:08 man
...skipping lots of stuff
-rw-rw-rw-   1 grog     bin           370 Jun 21 17:24 prab~
-rw-rw-rw-   1 grog     bin           370 Jun 21 17:22 parb
-rw-rw-rw-   1 grog     bin           196 Jun 21 17:22 parb~
-rwxrwxrwx   1 grog     wheel         469 Jun 18 15:19 tep
-rwxrwxr-x   1 root     wheel          52 Jun 18 14:29 font2c
-rwxrwxr-x   1 root     wheel         807 Jun 18 14:29 ps2epsi
-rwxrwxr-x   1 root     wheel          35 Jun 18 14:29 bdftops
-rwxrwxr-x   1 root     wheel         563 Jun 18 14:29 ps2ascii
-rwxrwxr-x   1 root     wheel          50 Jun 18 14:29 gslp
-rwxrwxr-x   1 root     wheel     3168884 Jun 18 14:29 gs
-rwxrwxr-x   1 root     wheel          53 Jun 18 14:29 gsdj
-rwxrwxr-x   1 root     wheel          51 Jun 18 14:29 gsbj
```

```
-rwxrwxr-x   1 root     wheel          18 Jun 18 14:29 gsnd
-rwxrwxr-x   1 root     wheel          54 Jun 18 14:29 gslj
-rwxr-xr-x   1 root     bin         81165 Jun 18 12:41 faxaddmodem
-r-xr-xr-x   1 bin      bin        249856 Jun 17 17:18 faxinfo
-r-xr-xr-x   1 bin      bin        106496 Jun 17 15:50 dialtest
...more stuff follows
```

It's easy to recognize the programs in this format: they were all installed in the same minute, and the next older file (*faxaddmodem*) is more than 90 minutes older, the next newer file (*tep*) is 50 minutes newer. The files we want to remove are, in sequence, *font2c*, *ps2epsi*, *bdftops*, *ps2ascii*, *gslp*, *gs*, *gsdj*, *gsbj*, *gsnd* and *gslj*.

We're not done yet, of course: *ghostscript* also installs a lot of fonts and PostScript files, as we saw in the *Makefile*. How do we find and remove them? It helps, of course, to have the *Makefile*, from which we can see that the files are installed in the directories */opt/bin*, */opt/lib/ghostscript* and */opt/man/man1* (see the *Makefile* excerpt on page 133). If you don't have the *Makefile*, all is not lost, but things get a little more complicated. You can search the complete directory tree for files modified between *Jun 18 14:00* and *Jun 18 14:59* with:

```
$ find  /opt -follow -type f -print|xargs ls -l|grep "Jun 18 14:"
-rwxrwxr-x   1 root     wheel          35 Jun 18 14:29 /opt/bin/bdftops
...etc
-rw-rw-r--   1 root     wheel         910 Jun 18 14:29 /opt/man/man1/ansi2knr.1
-rw-rw-r--   1 root     wheel       10005 Jun 18 14:29 /opt/man/man1/gs.1
-rw-rw-r--   1 root     wheel       11272 Jun 18 14:29 /opt/lib/ghostscript/Fontmap
-rw-rw-r--   1 root     wheel       22789 Jun 18 14:29 /opt/lib/ghostscript/bdftops.ps
-rw-rw-r--   1 root     wheel         295 Jun 18 14:29 /opt/lib/ghostscript/decrypt.ps
-rw-rw-r--   1 root     wheel       74791 Jun 18 14:29 /opt/lib/ghostscript/doc/NEWS
-rw-rw-r--   1 root     wheel       13974 Jun 18 14:29 /opt/lib/ghostscript/doc/devices.doc
...many more files
```

There are a couple of points to note here:

- We used GNU *find*, which uses the `-follow` option to follow symbolic links. If your */opt* hierarchy contains symbolic links, *find* would otherwise not search the subdirectories. Other versions of *find* may require different options.
- You can't use *ls -lR* here because *ls -lR* does not show the full pathnames: you would find the files, but the name at the end of the line would just be the name of the file, and you wouldn't know the name of the directory.
- If the file is more than six months old, *ls -l* will list it in the form

  ```
  -rwxrwxrwx   1 grog     wheel          22 Feb 10  1994 xyzzy
  ```

  This may be enough to differentiate between the files, but it's less certain. GNU *ls* (in the fileutils package) includes a option `--full-time` (note the two leading hyphens). This will always print the full time, regardless of the age of the file. With this option, the file above will list as:

  ```
  $ ls --full-time -l xyzzy
  -rwxrwxrwx   1 grog     wheel          22 Thu Feb 10 16:00:24 1994 xyzzy
  ```

Removing too much

None of these methods for removing installed software can handle one remaining serious problem: some programs install a modified version of a standard program, and if you remove the package, you remove all trace of this standard program. For example, GNU *tar* and GNU *cpio* both include the remote tape protocol program *rmt*. If you install both of these packages, and then decide to remove *cpio*, *tar* will not work properly either. It's not always enough to keep track of which packages depend on which programs: in some cases, a modified version of a program is installed by a package, and if you remove the package, you need to re-install the old version of the program.

Keeping track of installed software

All the methods we've seen so far smell strongly of kludge:

- They involve significant manual intervention, which is prone to error.
- The *remove* or *uninstall* targets of a *Makefile* are based on names not contents. If you stop using a package, and install a new one with some names that overlap the names of the old package, and then remove the old package, the files from your new package will go too.
- The manual method based on the dates does not discover configuration or data files—if you remove net news from a system, you will have to remember to remove the news spool area as well, because that certainly won't have the same modification timestamp as the installed software.
- It's almost impossible to safely and automatically remove modifications to environment variables in *.cshrc* and *.profile* files.

We can come closer to our goal if we have a method to keep track of the files that were actually installed. This requires the maintenance of some kind of database with information about the relationship between packages and files. Ideally,

- It would contain a list of the files installed, including their sizes and modification timestamps.
- It would prevent modification to the package except by well-defined procedures.
- It would contain a list of the files that were modified, including *diff*s to be able to reverse them.
- It would keep track of the modifications to the package as time went by: which files were created by the package, which files were modified.

This is an ideal, but the System V.4 *pkgadd* system comes reasonably close, and the concept is simple enough that we can represent the most important features as shell scripts. We'll look at it in the next section.

System V pkgadd

UNIX System V.4 is supplied as a number of binary *packages*[*]—you can choose which to install and which not to install. You can even choose whether or not to install such seemingly essential components as networking support and man pages.

Packages can be created in two formats: *stream format* for installation from serial data media like tapes, and *file system* format for installation from file systems. In many cases, such as diskettes, either form may be used. The program *pkgtrans* transforms one format into the other. In the following discussion, we'll assume file system format.

The package tools offer a bewildering number of options, most of which are not very useful. We'll limit our discussion to standard cases: in particular, we won't discuss *classes* and multi-part packages. If you are using System V.4 and want to use other features, you should read the documentation supplied with the system. In the following sections we'll look at the individual components of the packages.

pkginfo

The file *pkginfo*, in the root directory of the package, contains general information about the package, some of which may be used to decide whether or not to install the package. For example, the *pkginfo* file for an installable *emacs* package might look like:

```
ARCH=i386                       the architecture for which the package is intended
PKG=emacs                       the name of the package
VERSION=19.22                   the version number
NAME=Emacs text editor          a brief description
CATEGORY=utilities              the kind of package
CLASSES=none                    class information
VENDOR=Free Software Foundation the name of the owner
HOTLINE=LEMIS, +49-6637-919123, Fax +49-6637-919122  who to call if you have trouble
EMAIL=lemis@lemis.de            mail for HOTLINE
```

This information is displayed by *pkgadd* as information to the user before installation.

pkgmap

The file *pkgmap* is also in the root directory of the package. It contains information about the destination of the individual files. For example, from the same *emacs* package,

```
: 1 37986
1 d none /opt 0755 bin bin
1 d none /opt/README 0755 bin bin
1 f none /opt/README/emacs-19.22 0644 root sys 1518 59165 760094611
1 d none /opt/bin 0755 bin bin
1 f none /opt/bin/emacs 0755 root sys 1452488 11331 760577316
1 f none /opt/bin/etags 0755 root sys 37200 20417 760577318
```

* As used here, the term *package* is a collection of precompiled programs and data and information necessary to install them—this isn't the same thing as the kind of package we have been talking about in the rest of this book.

```
1 d none /opt/info 0755 bin bin
1 f none /opt/info/cl.info 0644 root sys 3019 62141 760094526
1 f none /opt/info/dir 0644 root sys 2847 23009 760559075
1 f none /opt/info/emacs 0644 root sys 10616 65512 760094528
1 d none /opt/lib 0755 bin bin
1 d none /opt/lib/emacs 0755 bin bin
1 d none /opt/lib/emacs/19.22 0755 bin bin
1 d none /opt/lib/emacs/19.22/etc 0755 bin bin
1 f none /opt/lib/emacs/19.22/etc/3B-MAXMEM 0644 root sys 1913 18744 574746032
```

The first line specifies that the package consists of a single part, and that it consists of 37986 512 byte blocks. The other lines describe files or directories:

- The first parameter is the part to which the file belongs.
- The next parameter specifies whether the file is a plain file (`f`), a directory (`d`), a link (`l`) or a symbolic link (`s`). A number of other abbreviations are also used.
- The next parameter is the *class* of the file. Like most packages, this package does not use classes, so the class is always set to `none`.
- The following four parameters specify the name of the installed object, its permissions, the owner and the group.
- After this come the size of the file, a checksum and the modification time in naked `time_t` format. The checksum ensures that the package is relatively protected against data corruption or deliberate modification.

Package subdirectories

In addition to the files in the main directory, packages contain two subdirectories *root* and *install*:

- *root* contains the files that are to be installed. All the files described in *pkgmap* are present under the same names in *root* (for example, */opt/bin/emacs* is called *root/opt/bin/emacs* in the package).
- The file *install/copyright* contains a brief copyright notice that is displayed on installation. *pkgadd* does not wait for you to read this, so it should really be brief.
- Optionally, there may be scripts with names like *install/preinstall* and *install/postinstall* which are executed before and after copying the files, respectively. *preinstall* might, for example, set up the directory structure */opt* if it does not already exist. *postinstall* might update *.cshrc* and *.profile* files. In some cases, it may need to do more. For example, the ISO 9660 directory standard for CD-ROMs limits allows only eight nested directories (in other words, the directory */a/b/c/d/e/f/g/h/i* is nested too deeply). *gcc* on a CD-ROM would violate this limitation, so some of the package has to be stored as a tar file, and the postinstall script extracts it to the correct position.

pkgadd

With this structure, adding a package is almost child's play: you just have to enter

```
$ pkgadd emacs
```

Well, almost. The name *emacs* is the name of the package and not a file name. By default, *pkgadd* expects to find it in */var/spool/pkg*. If your package is elsewhere, you can't tell *pkgadd* simply by prepending the name—instead, you need to specify it with the `-d` option:

```
$ pkgadd -d /cdrom emacs
```

This will install *emacs* from the directory *cdrom*.

Removing packages

One really nice thing about the System V.4 package system is the ease with which you can remove a package. Assuming that you have decided that *vi* is a better choice than *emacs*, or you just don't have the 19 MB that the emacs package takes up, you just have to type:

```
$ pkgrm emacs
```

and all the files will be removed.

Making installable packages

The discussion of *pkgadd* assumes that you already have an installable package. This is appropriate for System V.4, but if you have just ported a software package, you first need to create an installable binary package from it. This is the purpose of *pkgmk*. It takes a number of input files, the most important of which is *prototype*: it describes which files should be installed. It is almost identical in format to the *pkgmap* file we discussed above. For example, the *prototype* file for the *emacs* example above looks like:

```
# Prototype file created by /cdcopy/ETC/tools/mkmkpk on Wed Jan 19 18:24:41 WET 1994
i pkginfo
i preinstall
i postinstall
i copyright
# Required directories
d none /opt 755 bin bin
d none /opt/bin 755 bin bin
d none /opt/README 755 bin bin
d none /opt/man 755 bin bin
d none /opt/lib 755 bin bin
d none /opt/lib/emacs 755 bin bin
d none /opt/lib/emacs/19.22 755 bin bin
d none /opt/lib/emacs/19.22/etc 755 bin bin
d none /opt/info 755 bin bin
# Required files
f none /opt/lib/emacs/19.22/etc/3B-MAXMEM 644 root sys
f none /opt/bin/emacs 755 root sys
```

```
f none /opt/info/emacs 644 root sys
f none /opt/info/dir 644 root sys
```

This looks rather different from *pkgmap*:

- There are comment lines starting with #. The first line indicates that this file was created by a script. Later on we'll see the kind of function *mkmkpk* might perform.
- The first column (part number) and the last three columns (size, checksum and modification timestamp) are missing.
- Some lines start with the keyletter `i`. These describe installation files: we recognize the names from the discussion above. *pkgmk* copies these files into the directory tree as discussed above. What is not so immediately obvious is that *pkginfo* is placed in the main directory of the package, and the others are placed in the subdirectory *install*. It is also not obvious that some of these files are required: if they are not specified, *pkgmk* dies.

Making a prototype file

There's still a gap between the original *make install* and building an installable package. We need a *prototype* file, but *make install* just installs software. The packaging tools include a program called *pkgproto* that purports to build *prototype* files. It searches a directory recursively and creates *prototype* entries for every file it finds. If you have just installed *emacs*, say, in your */opt* directory, *pkgproto* will give you a prototype including every file in */opt*, including all the packages which are already installed there—not what you want. There are a number of alternatives to solve this problem:

- You can install into a different directory. *pkgproto* supports this idea: you can invoke it with

  ```
  $ pkgproto /tmp-opt=/opt
  ```

  which will tell it to search the directory */tmp-opt* and generate entries for */opt*. The disadvantage of this approach is that you may end up building programs with the path */tmp-opt* hard coded into the executables, and though it may test just fine on your system, the executable files will not work on the target system—definitely a situation to avoid.
- You rename */opt* temporarily and install *emacs* in a new directory, which you can then rename. This virtually requires you to be the only user on the system.
- Before installing *emacs*, you create a dummy file *stamp-emacs* just about anywhere on the system. Then you install *emacs*, and make a list of the files you have just installed:

  ```
  $ find /opt -follow  -cnewer stamp-emacs -type f -print | xargs ls -l >info
  ```

  This requires you to be the only person on the system who can write to the directory at the time. This is more not as simple as you might think. Mail and news can come in even if nobody else is using the system. Of course, they won't usually write in the same directories that you're looking in. Nevertheless, you should be prepared for a few surprises. For example, you might find a file like this in your list:

```
/opt/lib/emacs/lock/!cdcopy!SOURCE!Core!glibc-1.07!version.c
```

This is an *emacs* lock file: it is created by *emacs* when somebody modifies a buffer (in this case, a file called */cdcopy/SOURCE/Core/glibc-1.07/version.c*: *emacs* replaces the slashes in the file name by exclamation marks), and causes another *emacs* to warn the user before it, too, tries to modify the same file. It contains the pid of the *emacs* process that has the modified buffer. Obviously you don't want to include this file in your installable package.

Once you have tidied up your list of files, you can generate a *prototype* file with the aid of a shell script or an editor.

Running pkgmk

Once you have a *prototype* file, you're nearly home. All you have to do is run *pkgmk*. We run into terminology problems here: throughout this book, we have been using the term *package* to refer to the software we are building. More properly, this is the *software package*. *pkgmk* refers to its output as a package too—here, we'll refer to it as the *installable package*.

Unfortunately, *pkgmk* handles some pathnames strangely. You can read the man page (preferably several times), or use this method, which works:

- Before building the installable package, change to the root directory of the software package.
- Ignore path specifications in the *prototype* file and specify the *root path* as the root file system: `-r /`.
- Specify the *base directory* as the root directory of the package: since that's the directory we're in, just add `-b 'pwd'`.
- Choose to overwrite any existing package: `-o`.
- Specify the destination path explicitly: `-d /usr/pkg`. *pkgmk* creates packages will as subdirectories in this directory: the package *gcc* would create a directory hierarchy */usr/pkg/gcc*.

The resultant call doesn't change from one package to the next: it is

```
pkgmk -r / -b 'pwd' -o -d /usr/pkg
```

There is a whole lot more to using *pkgmk*, of course, but if you have *pkgmk*, you will also have the man pages, and that's the best source of further information.