5

Building the package

Now we have configured our package and we're ready to build. This is the Big Moment: at the end of the build process we should have a complete, functioning software product in our source tree. In this chapter, we'll look at the surprises that *make* can have in store for you. You can find the corresponding theoretical material in Chapter 19, *Make*.

Preparation

If you're unlucky, a port can go seriously wrong. The first time that error messages appear thick and fast and scroll off the screen before you can read them, you could get the impression that the packages were built this way deliberately to annoy you.

A little bit of preparation can go a long way towards keeping you in control of what's going on. Here are some suggestions:

Make sure you have enough space

One of the most frequent reasons of failure of a build is that the file system fills up. If possible, ensure that you have enough space before you start. The trouble is, how much is enough? Hardly any package will tell you how much space you need, and if it does it will probably be wrong, since the size depends greatly on the platform. If you are short on space, consider compiling without debugging symbols (which take up a lot of space). If you do run out of space in the middle of a build, you might be able to save the day by *stripping* the objects with *strip*, in other words removing the symbols from the file.

Use a windowing system

The sheer size of a complicated port can be a problem. Like program development, porting tends to be an iterative activity. You edit a file, compile, link, test, go back to edit the file, and so on. It's not uncommon to find yourself having to compare and modify up to 20 different files in 5 different directories, not to mention running *make* and the debugger. In adddition, a single line of output from *make* can easily be 5000 or 10000 characters long, many times the screen capacity of a conventional terminal.

All of these facts speak in favour of a windowing system such as X11, preferably with a high-resolution monitor. You can keep your editor (or editors, if they don't easily handle multiple files) open all the time, and run the compiler and debugger in other windows. If multiple directories are involved, it's easier to maintain multiple *xterm*s, one per directory, than to continually change directories. A correctly set up *xterm* will allow you to scroll back as far as you want—I find that 250 lines is adequate.

Keep a log file

Sooner or later, you're going to run into a bug that you can't fix immediately: you will have to experiment a bit before you can fix the problem. Like finding your way through a labyrinth, the first time through you will probably not take the most direct route, and it's nice to be able to find your way back again. In the original labyrinth, Theseus used a ball of string to find his way both in and out. The log file, a text file describing what you've done, is the computer equivalent of the ball of string, so you should remember to roll it up again. If you're running an editor like *emacs*, which can handle multiple files at a time, you can keep the log in the editor buffer and remove the notes again when you back out the changes.

In addition to helping you find your way out of the labyrinth, the log will also be of use later when you come to install an updated version of the software. To be of use like this, it helps to keep additional information. For example, here are some extracts from a log file for the *gcc*:

```
Platform:        SCO UNIX System V.3.2.2.0
Revision:        2.6.0
Date ported:     25 August 1994
Ported by:       Greg Lehey, LEMIS
Compiler used:   rcc, gcc-2.6.0
Library:         SCO


0. configure i386-unknown-sco --prefix=/opt. It sets local_prefix to
   /usr/local anyway, and won't listen to --local_prefix.  For some
   reason, config decides that it should be cross-compiling.

1. function.c fails to compile with the message function.c: 59: no
   space. Compile this function with ISC gcc-2.5.8.

2. libgcc.a was not built because config decided to cross-compile.
   Re-run config with configure i386-*-sco --prefix=/opt, and do an
   explicit make libgcc.a.

3. crtbegin.o and crtend.o were not built. Fix configure:

   --- configure~  Tue Jul 12 01:25:53 1994
   +++ configure   Sat Aug 27 13:09:27 1994
   @@ -742,6 +742,7 @@
                  else
                          tm_file=i386/sco.h
                          tmake_file=i386/t-sco
   +                      extra_parts="crtbegin.o crtend.o"
```

```
fi
truncate_target=yes
;;
```

Keeping notes about problems you have with older versions helps a lot: this example represents the results of a considerable time spent debugging the *make* procedure. If you didn't have the log, you'd risk tripping over this problem every time.

Save make output

Typically, to build a package, after you have configured it, you simply type

```
$ make
```

Then the fireworks start. You can sit and watch, but it gets rather boring to watch a package compile for hours on end, so you usually leave it alone once you have a reasonable expectation that it will not die as soon as you turn your back. The problem is, of course, that you may come back and find a lot of gobbldegook on the screen, such as:

```
make[5]: execve: ../../config/makedepend/makedepend: No such file or directory
make[5]: *** [depend] Error 127
make[5]: Leaving directory '/cdcopy/SOURCE/X11/X11R6/xc/programs/xsetroot'
depending in programs/xstdcmap...
make[5]: Entering directory '/cdcopy/SOURCE/X11/X11R6/xc/programs/xstdcmap'
checking ../../config/makedepend/makedepend over in ../../config/makedepend first...
make[6]: Entering directory '/cdcopy/SOURCE/X11/X11R6/xc/config/makedepend'
gcc -DNO_ASM -fstrength-reduce -fpcc-struct-return -fwritable-strings -O   \
-I../../config/imake  -I../..  OSDefines -DSYSV -DSYSV386       -c include.c
gcc: OSDefines: No such file or directory
In file included from include.c:30:
def.h:133: conflicting types for 'getline'
/opt/include/stdio.h:505: previous declaration of 'getline'
Broken pipe
```

This is from a real life attempt to compile X11R6, normally a fairly docile port. The target *makedepend* failed to compile, but why? The reason has long since scrolled off the screen.* You can have your cake and eat it too if you use *tee* to save your output:

```
$ make 2>&1 | tee -a Make.log
```

This performs the following actions:

- It copies error output (file descriptor 2) to standard output (file descriptor 1) with the expression `2>&1`.
- It pipes the combined standard output to the program *tee*, which echos it to its standard output and also copies it to the file *Make.log*.

* Well, there *is* a clue, but it's very difficult to see unless you have been hacking X11 configurations longer than is good for your health. `OSDefines` is a symbol used in X11 configuration. It should have been replaced by a series of compiler flags used to define the operating system to the package. In this case, the X11 configuration was messed up, and nothing defined `OSDefines`, so it found its way to the surface.

- In this case, I specified the `-a` option, which tells *tee* to append to any existing *Make.log*. If I don't supply this flag, it will erase any previous contents. Depending on what you're doing, you may or may not want to use this flag.

If you're not sure what your *make* is going to do, and especially if the *Makefile* is complicated, consider using the `-n` option. This option tells *make* to perform a "dry run": it prints out the commands that it would execute, but doesn't actually execute them.

These comparatively simple conventions can save a lot of pain. I use a primitive script called *Make* which contains just the single line:

```
make 2>&1 $* | tee -a Make.log
```

It's a good idea to always use the same name for the log files so that you can find them easily.

Standard targets

Building packages consists of more than just compiling and linking, and by convention many *Makefile*s contain a number of targets with specific meanings. In the following sections we'll look at some of the most common ones.

make depend

make depend creates a list of dependencies for your source tree, and usually appends it to the *Makefile*. Usually it will perform this task with *makedepend*, but sometimes you will see a depend target that uses *gcc* with the *-M* flag or *cpp*. depend should be the first target to run, since it influences which other commands need to be executed. Unfortunately, most *Makefile*s don't have a depend target. It's not difficult to write one, and it pays off in the reduction of strange, unaccountable bugs after a rebuild of the package. Here's a starting point:

```
depend:
        makedepend *.[ch]
```

This will work most of the time, but to do it correctly you need to analyze the structure of the package: it might contain files from other languages, or some files might be created by shell scripts or special configuration programs. Hopefully, if the package is this complicated, it will also have a depend target.

Even if you have a depend target, it does not always work as well as you would hope. If you make some really far-reaching changes, and things don't work the way you expect, it's worth starting from scratch with a *make clean* to be sure that the make still works.

make all

make all is the normal way to perform the build. Frequently, it is the default target (the first target in the *Makefile*), and you just need to enter *make*. This target typically rebuilds the package but does not install it.

make install

make install installs the compiled package into the local system environment. The usage varies considerably; we'll look at this target in more detail in Chapter 9, *Installation*, page 126.

make clean

make clean normally removes everything that *make all* has made—the objects, executables and possibly auxiliary files. You use it after deciding to change a compiler, for example, or to save space after you have finished an installation. Be careful with *make clean*: there is no complete agreement about exactly what it removes, and frequently you will find that it doesn't remove everything it should, or it is too eager and removes lots of things it shouldn't. *make clean* should remove everything that *make all* can make again—the intermediate and installable files, but not the configuration information that you may have taken days to get right.

make stamp-halfway

Occasionally you see a target like *make stamp-halfway*. The commands perform a lot of other things, and at the end just create an empty file called *stamp-halfway*. This is a short cut to save lots of complicated dependency checking: the presence of this file is intended to indicate that the first half of the build is complete, and that a restart of *make* can proceed directly to the second half. Good examples of this technique can be found in the *Makefile* for the GNU C compiler, and in the X11 source tree, which uses the name *DONE* for the stamp file.

Problems running make

Ideally, running *make* should be simple:

```
$ make all
lots of good messages from make
```

Things don't always go this smoothly. You may encounter a number of problems:

- You may not be able to find a *Makefile*, or the targets don't work the way you expect.
- *make* may not be able to make any sense of the *Makefile*.
- The *Makefile* may refer to non-existent files or directories.
- *make* seems to run, but it doesn't rebuild things it should, or it rebuilds things it shouldn't.
- You can't find anything that's wrong, but *make* still produces obscure error messages.

In the following sections we'll look at each of these problems. Here's an overview of the

types of error message we'll consider:

Table 5-1: Problems running make

Problem	page
Argument list too long	74
"$! nulled, predecessor circle"	71
"Circular dependency dropped"	71
"Commands commence before first target"	70
Comments in command lists	69
"Graph cycles through target	71
Incorrect continuation lines	73
Incorrect dependencies	68
make forgets the current directory	70
"Missing separator - stop"	70
Missing targets	66
No dependency on Makefile	68
No Makefile	64
Nonsensical targets	71
Problems with make clean	72
Problems with subordinate makes	68
Prompts in Makefiles	74
Subordinate makes	72
Syntax errors from the shell	71
Trailing blanks in variables	69
Unable to stop make	71
Wrong flavour of make	66
Wrong Makefile	66

Missing Makefile or targets

Sometimes *make* won't even let you in the door—it prints a message like:

```
$ make all
Don't know how to make all.  Stop.
```

The first thing to check here is whether there is a *Makefile*. If you don't find *Makefile* or *makefile*, check for one under a different name. If this is the case, the author should have documented where the *Makefile* comes from—check the *README* files and other documentation that came with the package. You may find that the package uses separate *Makefile*s for different architectures. For example, *Makefile* may be correct only if you are compiling in a BSD environment. If you want to compile for a System V machine, you may need to specify a different *Makefile*:

```
$ make -f Makefile.sysv
```

This is a pain because it's so easy to make a mistake. In extreme cases the compiler will successfully create objects, but they will fail to link.

Other possibilities include:

- The *Makefile* is created by the configuration process, and you haven't configured yet. This would be the case if you find an *Imakefile* (from which you create a *Makefile* with *xmkmf*—see Chapter 4, *Package configuration*, page 57), or *Makefile.in* (GNU *configure*—see page 55).
- The directory you are looking at doesn't need a *Makefile*. The *Makefile* in the parent directory, also part of the source tree, could contain rules like:

  ```
  foo/foo:        foo/*.c
          ${CC} foo/*.c -o foo/foo
  ```

  In other words, the executable is made automatically when you execute *make foo/foo* in the parent directory. As a rule, you start building in the root directory of a package, and perform explicit builds in subdirectories only if something is obviously wrong.
- The author of the package doesn't believe in *Makefile*s, and has provided a shell script instead. You often see this with programs that originated on platforms that don't have a *make* program.
- There is really nothing to build the package: the author is used to doing the compilation manually. In this case, your best bet is to write a *Makefile* from scratch. The skeleton in Example 5-1 will get you a surprisingly long way. The empty targets are to remind you what you need to fill in:

Example 5-1:

```
SRCS =                    list of C source files
OBJS = ${SRCS:.c=.o}      corresponding object files
CC=gcc                    file name of compiler
CFLAGS=-g -O3             flags for compiler
LDFLAGS=-g                flags for linker
BINDIR=/opt/bin
LIBDIR=/opt/lib
MANDIR=/opt/man
MAN1DIR=man1
INFODIR=/opt/info
PROGRAM=          name of finished program

all:    $(PROGRAM)
        ${CC} ${LDFLAGS} -o ${PROGRAM} ${OBJS}

man:

doc:

install:  all
```

Example 5-1: (continued)

```
depend:
        makedepend ${SRCS}

clean:
        rm -f \#* *~ core $(PROGRAM) *.o
```

Missing targets

Another obvious reason for the error message might be that the target `all` doesn't exist: some *Makefiles* have a different target name for each kind of system to which the *Makefile* has been adapted. The *README* file should tell you if this is the case. One of the more unusual examples is *gnuplot*. You need to enter

```
$ make All
$ make x11 TARGET=Install
```

The better ones at least warn you—see Chapter 4, *Package configuration*, page 53, for an example. I personally don't like these solutions: it's so much easier to add the following line at the top of the *Makefile*:

```
BUILD-TARGET = build-bsd
```

The first target would then be:

```
all:    ${BUILD-TARGET}
```

If you then want to build the package for another architecture, you need only change the single line defining `BUILD-TARGET`.

make doesn't understand the Makefile

Sometimes *make* produces messages that make no sense at all: the compiler tries to compile the same file multiple times, each time giving it a different object name, or it claims not to be able to find files that exist. One possible explanation is that various flavours of *make* have somewhat different understandings of default rules. In particular, as we will see in Chapter 19, *Make*, there are a number of incompatibilities between BSD *make* and GNU *make*.

Alternatively, *make* may not even be trying to interpret the *Makefile*. Somebody could have hidden a file called *makefile* in the source tree. Most people today use the name *Makefile* for *make*'s description file, probably because it's easier to see in an *ls* listing, but *make* always looks for a file called *makefile* (with lower case *m*) first. If you are using GNU *make*, it first looks for a file called *GNUmakefile* before checking for *makefile* and *Makefile*.

make refers to non-existent files

Building a package refers to a large number of files, and one of the most frequent sources of confusion is a file that can't be found. There are various flavours of this, and occasionally the opposite happens, and you have trouble with a file that *make* finds, but *you* can't find.

To analyse this kind of problem, it's helpful to know how *make* is referring to a file. Here are some possibilities:

- *make* may be looking for a dependent file, but it can't find it, and it can't find a rule to build it. In this case you get a message like:

```
$ make
make: *** No rule to make target 'config.h'.  Stop.
```

- *make* may not be able to locate a program specified in a command. You get a message like:

```
$ make foo.o
/bin/cc -c foo.o -o foo.c
make: execve: /bin/cc: No such file or directory
make: *** [foo.o] Error 127
```

- The compilers and other programs started by *make* also access files specified in the source. If they don't find them, you'll see a message like

```
$ make foo.o
gcc    -c foo.c -o foo.o
foo.c:1: bar.h: No such file or directory
make: *** [foo.o] Error 1
```

No matter where the file is missing, the most frequent reasons why it is not found are:

- The package has been configured incorrectly. This is particularly likely if you find that the package is missing a file like *config.h*.
- The search paths are incorrect. This could be because you configured incorrectly, but it also could be that the configuration programs don't understand your environment. For example, it's quite common to find *Makefiles* with contents like:

```
AR = /bin/ar
AS = /bin/as
CC = /bin/cc
LD = /bin/cc
```

  Some older versions of *make* need this, since they don't look at the PATH environment variable. Most modern versions of *make* do look at PATH, so the easiest way to fix such a *Makefile* is to remove the directory component of the definitions.

Problems with subordinate makes

Occasionally while building, the compiler complains about a file that doesn't seem to be there. This can be because the make is running in a subdirectory: large projects are frequently split up into multiple subdirectories, and all the top level *Makefile* does is to run a number of subordinate *make*s. If it is friendly, it also echos some indication of where it is at the moment, and if it dies you can find the file. Newer versions of GNU *make* print messages on entering and leaving a directory, for example:

```
make[1]: Entering directory '/cdcopy/SOURCE/Core/glibc-1.08.8/assert'
make[1]: Nothing to be done for 'subdir_lib'.
make[1]: Leaving directory '/cdcopy/SOURCE/Core/glibc-1.08.8/assert'
```

If neither of these methods work, you have the option of searching for the file:

```
$ find .  -name foo.c -print
```

or modifying the *Makefile* to tell you what's going on.

make doesn't rebuild correctly

One of the most insidious problems rebuilding programs occurs when *make* doesn't rebuild programs correctly: there's no easy way to know that a module has been omitted, and the results can be far-reaching and time-consuming. Let's look at some possible causes of this kind of problem.

Incorrect dependencies

One weakness of *make* is that you have to tell it the interdependencies between the source files. Unfortunately, the dependency specifications are *very* frequently incorrect. Even if they were correct in the source tree as delivered, changing configuration flags frequently causes other header files to be included, and as a result the dependencies change. Make it a matter of course to run a *make depend* after reconfiguring, if this target is supplied—see page 62 for details on how to make one.

No dependency on Makefile

What happens if you change the *Makefile*? If you decide to change a rule, for example, this could require recompilation of a program. To put it in *make* terms: all generated files depend on the *Makefile*. The *Makefile* itself is not typically included in the dependency list. It really should be, but that would mean rebuilding everything every time you change the *Makefile*, and in most cases it's not needed. On the other hand, if you do change your *Makefile* in the course of a port, it's a good idea to save your files, do a *make clean* and start all over again. If everything is OK, it will build correctly without intervention.

Other errors from make

The categories we have seen above account for a large proportion of the error messages you will see from *make*, but there are many others as well. In this section, we'll look at other frequent problems.

Trailing blanks in variables

You define a *make* variable with the syntax:

```
NAME = Definition     # optional comment
```

The exact *Definition* starts at the first non-space character after the = and continues to the end of the line or the start of the comment, if there is one. You can occasionally run into problems with things like:

```
MAKE = /opt/bin/make    # in case something else is in the path
```

When starting subsidiary *make*s, *make* uses the value of the variable MAKE as the name of the program to start. In this case it is "*/opt/bin/make* "—it has trailing blanks, and the *exec* call fails. If you're lucky, you get:

```
$ make
make: don't know how to make make   .  stop.
```

This message does give you a clue: there shouldn't be any white space between the name of the target and the following period. On the other hand, GNU *make* is "friendly" and tidies up trailing blanks, so it says:

```
$ make
/opt/bin/make        subdir     note the space before the target name "subdir"
make: execve: /opt/bin/make: No such file or directory
make: *** [suball] Error 127
```

The only clue you have here is the length of the space on the first line.

It's relatively easy to avoid this sort of problem: avoid comments at the end of definition lines.

Comments in command lists

Some versions of *make*, notably XENIX, can't handle rules of the form

```
doc.dvi:        doc.tex
        tex doc.tex
# do it again to get the references right
        tex doc.tex     # same thing again
```

The first comment causes *make* to think that the rule is completed, and it stops. When you fix this problem by removing the comment, you run into a second one: it doesn't understand the second comment either. This time it produces an error message. Again, you need to remove the comment.

make forgets the current directory

Occasionally, it looks as if *make* has forgotten what you tell it. Consider the following rule:

```
docs:
        cd doc
        ${ROFF} ${RFLAGS} doc.ms > doc.ps
```

When you run it, you get:

```
$ make docs
cd doc
groff -ms doc.ms >doc.ps
gtroff: fatal error: can't open 'doc.ms': No such file or directory
make: *** [docs] Error 1
```

So you look for *doc.ms* in *doc*, and it's there. What's going on? Each command is run by a new shell. The first one executes the `cd doc` and then exits. The second one tries to execute the `groff` command. Since the `cd` command doesn't affect the parent environment, it has no further effect, and you're still in the original directory. To do this correctly, you need to write the rule as:

```
docs:
        cd doc; \
        ${ROFF} ${RFLAGS} doc.ms > doc.ps
```

This causes *make* to consider both lines as a single line, which is then passed to a single shell. The semicolon after the `cd` is necessary, since the shell sees the command as a single line.

Missing separator - stop

This strange message is usually made more complicated because it refers to a line that looks perfectly normal. In all probability it is trying to tell you that you have put leading spaces instead of a tab on a command line. BSD *make* expects tabs, too, but it recovers from the problem, and the message it prints if they are missing is much more intelligible:

```
"Makefile", line 21: warning: Shell command needs a leading tab
```

Commands commence before first target

This message, from System V *make*, is trying to tell you that you have used a tab character instead of spaces at the beginning of the definition of a variable. GNU *make* does not have a problem with this—it doesn't even mention the fact—so you might see this in a *Makefile* written for GNU *make* when you try to run it with System V *make*. BSD *make* cannot handle tabs at the beginning of definitions either, and produces the message:

```
"Makefile", line 3: Unassociated shell command "CC=gcc"
Fatal errors encountered -- cannot continue
```

Syntax errors from the shell

Many *Makefile*s contain relatively complicated shell script fragments. As we have seen, these are constrained to be on one line, and most shells have rather strange relationship between new line characters and semicolons. Here's a typical example:

```
if test -d $(texpooldir); then exit 0; else mkdir -p $(texpooldir); fi
```

This example is all on one line, but you can break it *anywhere* if you end each partial line with a backslash (\). The important thing here is the placement of the semicolons: a rule of thumb is to put a semicolon where you would otherwise put a newline, but *not* after `then` or `else`. For more details, check your shell documentation.

Circular dependency dropped

This message comes from GNU *make*. In System V *make*, it is even more obscure:

```
$! nulled, predecessor circle
```

BSD *make* isn't much more help:

```
Graph cycles through docs
```

In each case, the message is trying to tell you that your dependencies are looping. This particular example was caused by the dependencies:

```
docs:   man-pages

man-pages: docs
```

In order to resolve the dependency `docs`, *make* first needs to resolve *man-pages*. But in order to resolve *man-pages*, it first needs to resolve *docs*—a real Catch 22 situation. Real-life loops are, of course, usually more complex.

Nonsensical targets

Sometimes the first target in the *Makefile* does nothing useful: you need to explicitly enter `make all` in order to make the package. There is no good reason for this, and every reason to fix it—send the mods back to the original author if possible (and be polite).

Unable to stop make

Some *Makefile*s start a number of second and third level *Makefile*s with the `-k` option, which tells *make* to continue if the subsidiary *Makefile* dies. This is quite convenient if you want to leave it running overnight and collect all the information about numerous failures the next morning. It also makes it almost impossible to stop the make if you want to: hitting the *QUIT* key (CTRL-C or DEL on most systems) kills the currently running *make*, but the top-level *make* just starts the next subsidiary *make*. The only thing to do here is to identify the top-level *make* and stop it first, not an easy thing to do if you have only a single screen.

Problems with make clean

make clean is supposed to put you back to square one with a build. It should remove all the files you created since you first typed `make`. Frequently, it doesn't achieve this result very accurately:

- It goes back further than that, and removes files that the *Makefile* doesn't know how to make.*
- Other *Makefile*s remove configuration information when you do a *make clean*. This isn't quite as catastrophic, but you still will not appreciate it if this happens to you after you have spent 20 minutes answering configuration questions and fixing incorrect assumptions on the part of the configuration script. Either way: before running a *make clean* for the first time, make sure that you have a backup.
- *make clean* can also start off by doing just the opposite: in early versions of the GNU C library, for example, it first compiled some things in order to determine what to clean up. This may work most of the time, but is still a Bad Idea: *make clean* is frequently used to clean up after some catastrophic mess, or when restarting the port on a different platform, and it should not be able to rely on being able to compile anything.
- Yet another problem with *make clean* is that some *Makefile*s have varying degrees of cleanliness, from *clean* via *realclean* all the way to *squeakyclean*. There may be a need for this, but it's confusing for casual users.

Subordinate makes

Some subordinate makes use a different target name for the subsidiary makes: you might write *make all*, but *make* might start the subsidiary *make*s with *make subdirs*. Although this cannot always be avoided, it makes it difficult to debug the *Makefile*. When modifying *Makefile*s, you may frequently come across a situation where you need to modify the behaviour of only one subsidiary *make*. For example, in many versions of System V, the *man* pages need to be formatted before installation. It's easy to tell if this applies to your system: if you install BSD-style unformatted man pages, the *man* program will just display a lot of hard-to-read nroff source. Frequently, fixing the *Makefile* is more work than you expect. A typical *Makefile* may contain a target *install* that looks like:

```
install:
        for dir in ${SUBDIRS}; do \
          echo making $@ in $$dir; \
          cd $$dir; ${MAKE} ${MDEFINES} $@; \
          cd ..; \
        done
```

`make $@` expands to `make install`. One of these subdirectories is the subdirectory *doc*,

* If this does happen to you, don't despair just yet. Check first whether this is just simple-mindedness on the part of the *Makefile*—maybe there is a relatively simple way to recreate the files. If not, and you forgot to make a backup of your source tree before you started, *then* you can despair.

which contains the documentation and requires special treatment for the *catman* pages: they need to be formatted before installation, whereas the *man* pages are not formatted until the first time they are referenced—see Chapter 7, *Documentation*, page 99 for further information. The simplest solution is a different target that singles out the *doc* and makes a different target, say *install-catman*. This is untidy and requires some modifications to the variable SUBDIRS to exclude *doc*. A simpler way is to create a new target, *install-catman*, and modify *all Makefiles* to recognize it:

```
install-catman install-manman:
        for dir in ${SUBDIRS}; do \
          echo making $@ in $$dir; \
          cd $$dir; ${MAKE} ${MDEFINES} $@; \
          cd ..; \
        done
```

In the *Makefiles* in the subdirectories, you might then find targets like

```
install-catman:  ${MANPAGES}
        for i in $<; do ${NROFF} -man $$i > ${CATMAN}/$i; done

install-manman:  ${MANPAGES}
        for i in $<; do cp $$i > ${MANMAN}/$i; done
```

The rule in the top-level *Makefile* is the same for both targets: you just need to know the name to invoke it with. In this example we have also renamed the original `install` target so that it doesn't get invoked accidentally. By removing the `install` target altogether, you need to make a conscious decision about what kind of man pages that your system wants.

We're not done yet: we now have exactly the situation we were complaining about on page 66: it is still a nuisance to have to remember *make install-catman* or *make install-manman*. We can get round this problem, too, with

```
INSTALL_TYPE=install-catman

install: ${INSTALL_TYPE}
```

After this, you can just enter *make install*, and the target *install* performs the type of installation specified in the variable INSTALL_TYPE. This variable needs to be modified from time to time, but it makes it easier to avoid mistakes while porting.

Incorrect continuation lines

Makefiles frequently contain numerous continuation lines ending with \. This works only if it is the very last character on the line. A blank or a tab following the backslash is invisible to you, but it really confuses *make*.

Alternatively, you might continue something you don't want to. Consider the following *Makefile* fragment, taken from an early version of the *Makefile* for this book:

```
PART1 =  part1.ms config.ms imake.ms make.ms tools.ms compiler.ms obj.ms \
         documentation.ms testing.ms install.ms epilogue.ms
```

At some point I decided to change the sequence of chapters, and removed the file *tools.ms*. I was not completely sure I wanted to do this, so rather than just changing the *Makefile*, I commented out the first line and repeated it in the new form:

```
# PART1 =        part1.ms config.ms imake.ms make.ms tools.ms compiler.ms obj.ms \
PART1 = part1.ms config.ms imake.ms make.ms compiler.ms obj.ms \
        documentation.ms testing.ms install.ms epilogue.ms
```

This works just fine—at first. In fact, it turns out that *make* treats all three lines as a comment, since the comment finished with a \ character. As a result, the variable PART1 remained undefined. If you comment out a line that ends in \, you should also remove the \.

Prompts in Makefiles

If you do the Right Thing and copy your *make* output to a log file, you may find that *make* just hangs. The following kind of *Makefile* can cause this problem:

```
all:    checkclean prog

checkclean:
        @echo -n "Make clean first? "
        @read reply; if [ "$$reply" = 'y' ]; then make clean; fi
```

If you run *make* interactively, you will see:

```
$ make
Make clean first?
```

If you copy the output to a file, of course, you don't see the prompt, and it looks as if *make* is hanging. This doesn't mean it's a bad idea to save your *make* output: it's generally a bad idea to put prompts into *Makefiles*. There are some exceptions, of course. The Linux configuration program is a *Makefile*, and to interactively configure the system you enter *make config*.

Arg list too long

Sometimes *make* fails with this message, especially if you are running a System V system. Many versions of System V limit the argument list to 5120 bytes—we'll look at this in more detail in Chapter 12, *Kernel dependencies*, page 169. Modern versions of System V allow you to rebuild the kernel with a larger parameter list: modify the tuneable parameter ARG_MAX to a value in the order of 20000. If you can't do this, there are a couple of workarounds:

- The total storage requirement is the sum of the length of the argument strings and the environment strings. It's very possible that you have environment variables that aren't needed in this particular situation (in fact, if you're like me, you probably have environment variables that you will never need again). If you remove some of these from your shell startup file, you may get down below the limit.
- You might be able to simplify expressions. For example, if your *Makefile* contains a line like

```
clean:
        rm -rf *.o *.a *.depend *~ core ${INTERMEDIATES}
```

you can split it into

```
clean:
        rm -rf *.o
        rm -rf *.a *.depend *~ core ${INTERMEDIATES}
```

In most large trees, the **.o* filenames constitute the majority of the arguments, so you don't need more than two lines.

- Even after the previous example, you might find that the length of the **.o* parameters is too long. In this case, you could try naming the objects explicitly:

```
clean:
        rm -rf [a-f]*.o
        rm -rf [g-p]*.o
        rm -rf [r-z]*.o
        rm -rf *.a *.depend *~ core ${INTERMEDIATES}
```

- Alternatively, you could specify the names explicitly in the *Makefile*:

```
OBJ1S = absalom.o arthur.o ...  fernand.o
OBJ2S = gerard.o guillaume.o ...  pierre.o
OBJ3S = rene.o roland.o ...  zygyszmund.o
OBJS = ${OBJ1S} ${OBJ2S} ${OBJ3S}

clean:
        rm -rf ${OBJ1S}
        rm -rf ${OBJ2S}
        rm -rf ${OBJ3S}
```

- Yet another method involves the use of the *xargs* program. This has the advantage of not breaking after new files have been added to the lists:

```
clean:
        find .  -name "*.o" -print | xargs rm -f
```

This chops up the parameter list into chunks that won't overflow the system limits.

Creating executable files

The *xargs* method is not much help if you want to build an executable file. If the command that fails looks like

```
${PROG}:
        ${CC} ${ALLOBJS} -o ${PROG}
```

there are some other possibilities. You might be able to shorten the pathnames. If you are building in a directory */next-release/SOURCE/sysv/SCO/gcc-2.6.0*, and every file name in ALLOBJS is absolute, it's much easier to exceed the limit than if the directory name was, say, */S*. You *could* use a symbolic link to solve this problem, but most systems that don't support ARG_MAX also don't have symbolic links.*

If this doesn't work, you could place the files in a library, possibly using *xargs*:

```
${PROG}:
        rm libkludge.a
        echo ${ALLOBJS} | xargs ar cruv libkludge.a
        ${CC} libkludge.a -o ${PROG}
```

This looks strange, since there's no object file, but it works: by the time it finds the name *libkludge.a*, the linker has already loaded the object file *crt0.o* (see Chapter 21, *Object files and friends*, page 368), and is looking for a symbol `main`. It doesn't care whether it finds it in an object file or a library file.

Modifying Makefiles

Frequently enough, you find that the *Makefile* is inadequate. Targets are missing, or some error occurs that is almost untraceable: you need to fix the *Makefile*. Before you do this, you should check whether you are changing the correct *Makefile*. Some packages build a new *Makefile* every time you run *make*. In particular, you frequently see *Makefile*s that start with text like

```
# Makefile generated by imake - do not edit!
```

You can follow this advice or not: it depends on you and what you are doing: If you are just trying to figure out what the *Makefile* is trying (and presumably failing) to do, it's nice to know that you can subsequently delete your modified *Makefile* and have it automatically remade.

Once you have found out why the *Makefile* is doing what it is, you need to fix the source of the *Makefile*. This is not usually too difficult: the input files to the *Makefile* generation phase typically don't look too different from the finished *Makefile*. For example, *Makefile.in* in the GNU packages is a skeleton that is processed by *m4*, and except for the *m4* parameters *Makefile.in* looks very similar to the finished *Makefile*. Finding the way back to the *Imakefile* from the *Makefile* requires a little more understanding of the *imake* process, but with a little practice it's not that difficult.

* If you are on a network with other machines with more modern file systems, you could work around this problem by placing the files on the other system and accessing them via NFS.