Plan 9 Mkfiles

R. Flandrena

Introduction

Every Plan 9 source directory contains a file, called `mkfile`, specifying the rules for building the executable or library that is the product of the directory. *Mk*(1) interprets the rules in the file, calculates the dependencies and executes an *rc*(1) script to construct the product. If necessary components are supplied by subtending or neighboring directories, the mkfiles in those directories are first executed to build the components before the local construction proceeds.

Most application source directories produce one of four types of product: a single executable, several executables, a local library, or a system library. Four generic mkfiles define the normal rules for building each type of product. The simplest mkfiles need only list the components and include the appropriate generic mkfile to do the work. More complex mkfiles may supply additional rules to augment, modify or override the generic rules.

Using a Mkfile

To build a product, change to the directory containing its source and invoke *mk* with the appropriate target as an argument. All mkfiles provide the following standard targets:

`all`	Build a local version of the product or products for the current architecture. If the product is a single program, the result is stored in file `$O.out`. If the directory produces multiple executables, they are stored in the files named `$O.`*progname*, where *progname* is the name of each executable. A product may be built for a different architecture by prefacing the `mk` command with `objtype=`*architecture*, where *architecture* is the name of the target architecture. Directories producing system libraries always operate directly on the installed version of the library; in this case the target `all` is equivalent to the target `install`.
`install`	Build and install the product or products for the current architecture.
`installall`	Build and install the product or products for all architectures.
`clean`	Rid the directory and its subdirectories of the by-products of the build process. Intermediate files that are easily reproduced (e.g., object files, `yacc` intermediates, target executables) are always removed. Complicated intermediates, such as local libraries, are usually preserved.
`nuke`	Remove all intermediates from the directory and any subdirectories. This target guarantees that a subsequent build is performed from scratch.

If no target is specified on the `mk` command line, the `all` target is built by default. In a directory producing multiple executables, there is no default target.

In addition to the five standard targets, additional targets may be supplied by each generic mkfile or by the directory's mkfile.

Creating a Mkfile

The easiest way to build a new mkfile is to copy and modify an existing mkfile of the same type. Failing that, it is usually possible to create a new mkfile with minimal effort, since the appropriate generic mkfile predefines the rules that do all the work. In the simplest and most common cases, the new mkfile need only set a couple of variables and include the appropriate architecture-specific and generic mkfiles.

There are four generic mkfiles containing commonly used rules for building a product: `mkone`, `mkmany`, `mklib`, and `mksyslib`. These rules perform such actions as compiling C source files, loading object files, archiving libraries, and installing executables in the `bin` directory of the appropriate architecture. The generic mkfiles are stored in directory `/sys/src/cmd`. Mkfile `mkone` builds a single executable, `mkmany` builds several executables from the source in a single directory, and `mklib` and `mksyslib`, maintain local and system libraries, respectively. The rules in the generic mkfiles are driven by the values of variables, some of which must be set by the product mkfile and some of which are supplied by the generic mkfile. Variables in the latter class include:

Variable	*Default*	*Meaning*
`CFLAGS`	`-w`	C compiler flags
`LDFLAGS`		Loader flags
`YFLAGS`	`-d`	Yacc flags
`AFLAGS`		Assembler flags

The following variables are set by the product mkfile and used by the generic mkfile. Any may be empty depending on the specific product being made.

`TARG`	Name(s) of the executable(s) to be built
`LIB`	Library name(s)
`OFILES`	Object files used in build
`HFILES`	Common header files
`YFILES`	`Yacc` input files
`BIN`	Directory where executables are installed

Mkfile Organization

All mkfiles share the following common structure:

```
</$objtype/mkfile           #architecture-dependent definitions

variable definitions        #TARG, OFILES, HFILES, etc.

</sys/src/cmd/generic       #mkone, mkmany, mklib, or mksyslib

extra rules                 #overrides, augmented rules, additional targets
```

The variables driving the expansion of the generic mkfile may be specified in any order as long as they are defined before the inclusion of the generic mkfile. The value of a variable may be changed by assigning a new value following the inclusion of the generic mkfile, but the effects are sometimes counter-intuitive. Such variable assignments do not apply to the target and prerequisite portions of any previously defined rules; the new values apply to all recipes and to all parts of any rules following the assignment in the mkfile. Thus, assigning a new value to a variable following the inclusion of the generic mkfile results in the new value applying to the recipes in the generic file, but not to the dependency specifications.

The rules supplied by the generic mkfile may be overridden or augmented. The new rules must be specified after the inclusion of the generic mkfile. If the target and prerequisite portion of the rule exactly match the target and prerequisite portion of a previously defined rule and the new rule contains a recipe, the new rule replaces the old one. If the target of a new rule exactly matches the target of a previous rule and one or more new prerequisites are specified and the new rule contains no recipe, the new prerequisites are added to the prerequisites of the old rule.

Following sections discuss each generic mkfile in detail.

Mkone

The `mkone` generic mkfile contains rules for building a single executable from one or more files in a directory. The variable `TARG` specifies the name of the executable and variables `OFILES` and `YFILES` specify the object files and `yacc` source files used to build it. `HFILES` contains the names of the local header files included in all source files. `BIN` is the name of the directory where the executable is installed. `LIB` contains the names of local libraries used by the linker. This variable is rarely needed as libraries referenced by a `#pragma` directive in an associated header file, including all system libraries, are automatically searched by the loader.

If no command line target is specified, the `all` target is built; it produces an executable in `$O.out.` Variable `HFILES` identifies the header files that are included in all or most or the C source files. Occasionally, a program has other header files that are only used in some source files. A header can be added to the prerequisites for those object files by adding a rule of the following form following the inclusion of generic mkfile `mkone`:

```
file.$O:        header.h
```

The mkfile for a directory producing a single executable using the normal set of rules is trivial: a list of some files followed by the inclusion of *mkone.* For example, `/sys/src/cmd/scat/mkfile` contains:

```
</$objtype/mkfile

TARG=scat
OFILES=scat.$O\
        desc.$O\
        patch.$O\
        prose.$O\
        util.$O\

HFILES=sky.h

BIN=/$objtype/bin
</sys/src/cmd/mkone
```

The more complex mkfile in `/sys/src/cmd/samterm` specifies an additional dependence between the specialized header `../sam/mesg.h` and the object file `mesg.$O` as well as providing an extra target to generate a *db*(1) script to dump a named structure:

```
</$objtype/mkfile

TARG=samterm
OFILES=main.$O icons.$O menu.$O mesg.$O\
        rasp.$O scroll.$O flayer.$O io.$O plan9.$O

HFILES=samterm.h\
        flayer.h\

BIN=/$objtype/bin/aux
</sys/src/cmd/mkone

mesg.$O:        ../sam/mesg.h

%.db:
        $CC -s$stem main.c | dbfmt > $stem.db
```

Mkmany

The `mkmany` generic mkfile builds several executables from the files in a directory. It differs from the operation of `mkone` in three respects: `TARG` specifies the names of all executables, there is no default command-line target, and additional rules allow a single executable to be built or installed.

The `TARG` variable specifies the names of all executables produced by the mkfile. The rules assume the name of each executable is also the name of the file containing its `main` function. `OFILES` specifies files containing common subroutines loaded with all executables. Consider the mkfile:

```
</$objtype/mkfile

TARG=alpha beta
OFILES=common.$O
BIN=/$objtype/bin
</sys/src/cmd/mkmany
```

It assumes the main functions for executables `alpha` and `beta` are in files `alpha.$O` and `beta.$O`, respectively and that both programs use the subroutines in file *common.*`$O`. The `all` target builds all executables, leaving each in a file with a name of the form `$O.`*progname* where *progname* is the name of the executable. In this example the `all` target produces executables `$O.alpha` and `$O.beta`.

The `mkmany` rules provide additional targets for building a single executable:

`$O.progname`	Builds executable `$O.`*progname* in the current directory. When the target architecture is not the current architecture the `mk` command must be prefixed with the customary `objtype=`*architecture* assignment to select the proper compilers and loaders.
`progname.install`	Installs executable *progname* for the target architecture.
`progname.installall`	Installs executable *progname* for all architectures.

Mklib

The `mklib` generic mkfile builds a local library. Since this form of mkfile constructs no executable, the `TARG` and `BIN` variables are not needed. Instead, the `LIB` variable specifies the library to be built or updated. Variable `OFILES` contains the names of the object files to be archived in the library. The use of variables `YFILES` and `HFILES` does not change. When possible, only the out-of-date members of the library are updated.

The variable `LIBDIR`, contains the name of the directory where the library is installed; by default it selects the current directory. It can be overridden by assigning the new directory name after the point where `mklib` is included.

The `clean` target removes object files and *yacc* intermediate files but does not touch the library. The `nuke` target, removes the library as well as the files removed by the `clean` target. The command

```
mk -s clean all
```

causes the existing library to be updated, or created if it doesn't already exist. The command

```
mk -s nuke all
```

forces the library to be rebuilt from scratch.

The mkfile from `/sys/src/cmd/upas/libString` contains the following specifications to build the local library `libString.a$O` for the object architecture referenced by `$O`:

```
</$objtype/mkfile

LIB=libString.a$O
OFILES= s_alloc.$O\
        s_append.$O\
        s_array.$O\
        s_copy.$O\
        s_getline.$O\
        s_grow.$O\
        s_nappend.$O\
        s_parse.$O\
        s_read.$O\
        s_read_line.$O\
        s_tolower.$O\

</sys/src/cmd/mklib

nuke:V:
        mk clean
        rm -f libcommon.a[$OS]
```

The override of the rule for target `nuke` removes the libraries for all architectures as opposed to the default recipe for this target which removes the library for the current architecture.

Mksyslib

The `mksyslib` generic mkfile is similar to the `mklib` mkfile but it operates directly on a system library instead of a local library. The `install` and `all` targets are the same; since there is no local copy of the library, all updates are performed on the installed library. The rule for the `nuke` target is identical to that of the `clean` target; unlike the `nuke` target for local libraries, the library is never removed.

No attempt is made to determine if individual library members are up-to-date; all members of a library are always updated. Special targets support manipulation of a single object file; the target `objfile` updates file `objfile.$O` in the library of the current architecture and the target `objfile.all` updates `objfile.$O` in the libraries of all architectures.

Overrides

The rules provided by a generic mkfile or the variables used to control the evaluation of those rules may be overridden in most circumstances. Overrides must be specified in the product mkfile after the point where the generic mkfile is included; in general, variable and rule overrides occupy the end of a product mkfile.

The value of a variable is overridden by assigning a new value to the variable. Most variable overrides modify the values of flags or the names of commands executed in recipes. For example, the default value of `CFLAGS` is often overridden or augmented and the ANSI/Posix Computing Environment is selected by setting the `CC` and `LD` variables to `pcc`.

Modifying rules is trickier than modifying variables. Additional constraints can be added to a rule simply by specifying the target and the new prerequisite. For example,

```
%.$O:   header.h
```

adds file `header.h` the the set of prerequisites for all object files. There is no mechanism for adding additional commands to an existing recipe; if a recipe is unsatisfactory, the rule and its recipe must be completely overridden. A rule is overridden only when the replacement rule matches the target and prerequisite portions of the original rule exactly. The recipe associated with the new rule then replaces the recipe of the original rule. For example, `/sys/src/cmd/movie/mkfile` overrides the `installall` target by performing the usual action before copying the `rc` script entitled `movie` to directory `/rc/bin`:

```
< /$objtype/mkfile

TARG=anim\
        fdevelop\

BIN=/sys/lib/movie/$objtype
</sys/src/cmd/mkmany
CC=pcc
LD=pcc
CFLAGS=-c

installall:V:
        for(objtype in $CPUS)
                mk $MKFLAGS install
        cp movie /rc/bin/movie
```

Special Tricks

Two special cases require an extra dollop of deviousness.

In the first, a file needed to build an executable is generated by a program that in turn, is built from a source file that is not part of the product. In this case, the executable must be built for the target architecture, but the intermediate executable must be built for the architecture `mk` is executing on. The intermediate executable is built by recursively invoking `mk` with the appropriate target and the executing architecture as the target architecture. When that `mk` completes, the intermediate is executed to generate the source file to complete the build for the target architecture. An example of this build procedure can be found in the `/sys/src/cmd/awk/mkfile`.

Another awkward situation occurs when a directory contains source to build an executable as well as source for auxiliary executables that are not to be installed. In this case the `mkmany` generic rules are inappropriate, because all executables would be built and installed. A good way to handle this situation is to used the `mkone` generic file to build the primary executable and provide extra targets to build the auxiliary executables. This approach is also useful when the auxiliary files are not executables; `/sys/src/cmd/spell/mkfile` augments to default rules that build and install the `spell` executable with elaborate rules to generate and maintain the auxiliary spelling lists.