README

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

This brief document is intended to help you get started using Plan 9. It is written by a casual Plan 9 user who is not in any way part of the Plan 9 group, and is aimed at ordinary users with a Unix background.

1. Getting Started

I'm assuming that you or someone you trust has read, understood, and performed the instructions in the Plan 9 installation procedure. I'm also assuming that you have at least looked at the Plan 9 overview paper in Computing Science Technical Report 158, which explains what Plan 9 is and how it goes about it. There are also some helpful explanations and examples in the paper ''The Use of Name Spaces in Plan 9,'' which is also part of the distribution. I am further assuming that you have the Plan 9 manual near to hand, and are willing to read manual pages for commands as their names appear here.

How you get started after you turn your terminal on depends on how your Plan 9 system is set up; the details are different for a standalone system or a terminal connected to a file server and a CPU server. The latter is mostly what I'll talk about.

When Plan 9 boots, it runs a very dumb terminal program, so dumb that it doesn't even know how to scroll a screen. You won't want to use that for much of anything. In much the same way that `.profile` is executed by the shell on Unix systems, the file `lib/profile` is executed by the shell on Plan 9 when you log in. The `newuser` command normally creates a few directories (`bin`, `bin/rc`, `lib`, and `tmp`), then sets up a profile `lib/profile` that looks like this, though with some more frills:

```
bind -a $home/bin/rc /bin
font = /lib/font/bit/pelm/euro.9.font
switch($service){
case terminal
	prompt=('term% ' '	')
	exec 8½
case cpu
	bind -b /mnt/term/mnt/8½ /dev
	prompt=('cpu% ' '	')
}
```

Many of the interesting bits of Plan 9 are implicit in this file.

2. Commands

Most of the standard Unix commands exist in almost the same form on Plan 9; this includes standbys like `cat`, `ls`, `cd`, `pwd`, `cp`, `mv`, `diff`, `grep`, `awk`, and so on. You'll notice minor differences in behavior but for the most part you don't have to think about this.

3. The rc Shell

One big difference: Plan 9 uses a different shell, called `rc`. For running commands interactively, it's almost the same as the Bourne shell, so filename metacharacters like `*` and `?` behave the same, and simple redirections with > and >> are the same (but not redirection of streams by number). Quoting is simpler than in the Bourne shell: double quotes and backslash have no special meaning, and single quotes quote anything. To get a single quote into a quoted string, double the quote:

```
echo ''''
```

prints a single quote.

For programming, `rc` is almost unrelated, which is a nuisance. All shell scripts have to begin with

```
#!/bin/rc
```

Environment variables are set by

`var = '`*anything*`'`

where the quotes can be omitted if *anything* contains no spaces. Environment variables are accessed as `$var`; certain variables are initialized when your process begins, including `user` (your name), `home` (your home directory), and `service`, which is `terminal` when you are running on your terminal.

4. Directories and Search Paths

One of the unifying ideas in Plan 9 is that all resources are accessed as file systems. Central to this is management of the name space so that you can select and arrange the resources you want to use.

The `bind` and `mount` commands manipulate the name space. In particular, the command

```
bind -a $home/bin/rc /bin
```

in the profile binds the directory `$home/bin/rc` after the directory `/bin`, forming a *union directory.* (More precisely, it makes `/bin` an alias for the union). The shell searches only `/bin` for commands to run, but it searches all the directories that have been unioned together. By convention, your personal shell scripts are placed in `$home/bin/rc`.

When you first log in, several directories are bound to `/bin`, including `/rc/bin`, which contains shell scripts, and `/$cputype/bin`, which contains binaries for your current cpu type. `cputype` is the type of processor you are running on, typically one of `68020`, `sparc`, `mips`, or `386`. When you run a program like `ls`, the version for your current cpu type will be found and executed. If you subsequently execute the `cpu` command to access a cpu server, in that process and those started by it, `cputype` will be the type of the cpu server, and the `ls` command you run there will be the right binary for that cpu.

If you think about it, you can see that this mechanism of union directories replaces the search path of conventional Unix shells. As far as you are concerned, *all* executable programs are in `/bin`. Try

```
lc /bin
```

to see the names of all executable programs.

5. Interesting File Systems

Unix users are familiar with the idea that devices like disks and tapes are part of the file system. Plan 9 carries that idea a lot further. If you look at the directory `/dev`, you will see some familiar names. Try

```
cat /dev/time
```

a couple of times, for example. Or, after you have snarfed some text using the button 2 menu item, try

```
cat /dev/snarf
```

Note that some files like `cons` and `mouse` occur more than once. `/dev` is a union directory, and these are multiple occurrences of the same file. The first `/dev/mouse` refers to the current window, and the next one to the enclosing window (which is probably the whole screen at this point). Try

```
cat /dev/mouse
```

then move the mouse around inside and outside of the current window.

The shell environment is kept in a directory called `/env`; each shell environment variable is stored in a file. Try

```
cat /env/font
```

for example.

Running processes are found in `/proc`; each process is a directory, and each file in that directory accesses some aspect of the process. For example, the `status` file contains (all ASCII) status information about the process. Try

```
awk '{print $1}' /proc/*/status
```

to get a list of the names of the running processes.

You might also find it interesting to poke around in `/net`; all network connections are managed as file systems as well. In all of these cases, the service presents a file-system interface to its clients, although the implementation behind is not generally a traditional file system.

6. The 8½ Window System

The window system for Plan 9 is called 8½; the `terminal` case in the profile starts 8½ with the line

```
exec 8½
```

8½ provides much less of the ``flexibility'' and certainly far fewer features than `xterm` on X terminals, but I much prefer it.

The most important difference is that 8½ treats all text on the screen alike; with the mouse you can edit anything you can see, so you can fix up and resubmit commands, fiddle the output of a program or its input and resubmit it, and so on. This ability to edit the past is liberating to such a degree that once you use it, you'll never want to go back to something like `xterm`. (There is an implementation of an X server so you can run it if you want, but I never have.)

8½ does not provide any of the terminal-handling mechanisms like termcap or curses. That means that there is no support for (nor indeed implementation of) old favorites like `vi`, `emacs`, and `ksh`. In my experience, the ability to cut, paste, and edit what's on all the windows on the screen obviates the need for many of the mechanisms in these programs, so it's not as big a loss as you might think.

8½ can be called recursively: you can make a new window, run 8½ in it, and everything you do there is insulated from the surroundings. 8½ does not provide any analog of the virtual window management of, for example, VTWM, nor does it provide zillions of (or even a few) icons, but you can move a window almost off the screen, and you can hide it and then recall it from a popup menu.

You can also give 8½ a file of commands to run when it starts, which most people put into their profile:

```
exec 8½ -i lib/windows
```

Normally this is used to set up windows that you always use:

```
#!/bin/rc
window 'x0 y0 x1 y1'  command line
...
```

where *x0,y0* and *x1,y1* are the coordinates of the window in question (and *x* increases down the screen). The `window` command opens a window at the specified place, then runs the command in it. The command `wloc` will tell you the names and locations of all windows in the right format to be inserted directly in a file. Set up the windows and programs the way you want them, then run `wloc` and snarf its output.

7. Fonts

One aspect of 8½ that you can change is the font it uses for displaying text. There is a default font, but normally the variable `font` is set explicitly in the profile:

```
font = /lib/font/bit/pelm/euro.9.font
8½ -f $font
```

The font `euro.9.font` is a collection of almost any character you might find in European languages, including Cyrillic, Greek, and a bunch of special characters. There are other fonts that include oriental languages as well, and a variety of sizes.

Plan 9 uses the Unicode character set throughout, which means that the system and the various programs all deal pretty comfortably with a very large character set. (Think 16 bits, or 65K characters.) So if you want to edit files in languages that use more than ASCII characters, or run `grep` or `awk` over them, it just works. (You may have trouble printing such characters on standard printers, but they will appear fine on the screen.)

8. Editing

The standard Plan 9 editor is called `sam`; it's a particularly good multi-file editor, it provides regular expression syntax the same as in the venerable `ed` (which also exists), and you can snarf text from one of its windows and paste it into other 8½ windows or vice versa. The mouse idioms for `sam` and `8½` are the same. It will also edit files on other systems if there is a network connection.

By the way, regular expressions have been cleaned up -- all programs support the same regular expressions, which are pretty close to those found in `egrep` on Unix systems.

9. The CPU Server

In the Plan 9 world view one is meant to run interactive programs like editors on the terminal and compute-intensive programs like compilers on a cpu server, which runs faster and has a higher bandwidth to the file server. The `cpu` command connects you to a cpu server so your computation runs faster (in theory), but everything else stays the same. The mechanism is quite different from either remote login (which does not preserve the name space you are currently working in) or network file system access (which does not change the processor). The line

```
bind -b /mnt/term/mnt/8½ /dev
```

in your profile arranges that all the devices (including mouse, keyboard and screen) associated with your terminal are inherited by the cpu server so they continue to work in a cpu window.

10. Connecting to Unix Systems

It is highly likely that your Plan 9 system will be connected by some network to a Unix system. The command `con` connects to another system (typically Unix); the command `rx` is rather like the `rsh` command on Unix systems, for executing a single command on another machine.

If the Unix system cooperates, it is also possible to mount a Unix file system in the Plan 9 name space so that files on the Unix side are accessible from Plan 9. The command

`9fs` *machine*

establishes the connection and mounts the files; thereafter the root of the target file system is in the Plan 9 directory at `/n/`*machine*.

There is no connection from Unix to Plan 9; security is taken seriously (no superuser, for example) and it's hard to provide a satisfactory mechanism.

11. Backup and Recovery

Normally the state of the Plan 9 file system is recorded every day or so; on our system, it's stored on an optical disk. If your Plan 9 system is suitably equipped, you should be able to run another service that makes the past state of the file system accessible (read only). The command

```
9fs dump
```

mounts this file system on `/n/dump`. At that point, I can `cd` into the past:

```
cd /n/dump/1991/0401/usr/bwk
ls -l
```

puts me in my directory as it was on April 1, 1991. This really is a file system, so all the normal commands work fine; I can `diff` a file from then with one on some other date, or copy an old version to the present.

Plan 9 has no backup or recovery programs; this mechanism subsumes them all.

12. Programming in Plan 9

Most programming in Plan 9 is done in ANSI C, with the usual supporting tools like YACC available. One difference of note: `make` has been largely supplanted by `mk`, which is cleaner but different. As with the shell, it takes time to internalize the differences.

There are separate C compilers for each supported cpu type (badly named with a single letter mnemonic), but each compiler can produce code for any object type. The `mkfile` normally encapsulates this.

Although ANSI C is supported, the Plan 9 libraries are not ANSI and the standard ANSI header files normally are not found. Compiling C programs is different enough that you should read the paper called ‘‘How to Use the Plan 9 C Compiler’’ before starting.

If you are importing or exporting a C program, you will want to use the ANSI/POSIX environment (‘‘ape’’), which really does provide for portability, including a complete set of POSIX-compatible libraries and some POSIX tools. The compiler driver is called `pcc`. The commands

```
bind −a /bin/ape /bin
bind −a /rc/bin/ape /bin
```

will bind the right files.

13. Envoi

Plan 9 is not Unix. If you think of it as Unix, you’ll often be frustrated because something doesn’t exist or works differently. If you think of it as Plan 9, however, you’ll find that most of it works very smoothly, and that there are some really neat ideas that make things much cleaner than you have seen before.