

Work

by Jeffreys Copeland and Haemer



ALEX GROSS

*“Put this in your pipe
and smoke it.”*

– Us to each other

“Say please.”

– Us to our children

A Pipe Ptool

In the beginning, operating systems provided one method of interprocess communication (IPC): the file. Process A could pass information to process B by writing to a file and letting process B read that file. Even today, when operating systems routinely provide signals, sockets, pipes, fifos, semaphores, message passing and shared memory, this method still works fine.

It is, however, a little slow. Modern operating systems are good about providing file system caches that soften some of the performance penalties, but if process B needs to look through the voluminous output of process A for example, it can be expensive to have process A write all the data to disk and then have process B read it back.

In 1972, Ken Thompson “released” UNIX v2, which provided a new IPC mechanism: pipes. Combined with `fork()` and `exec()`, `pipe()` permitted a pair of related processes to share information without requiring that the information be written to disk. Pipes

permitted and promoted the UNIX toolbox philosophy, in which small specialized tools are hooked together to perform single, collaborative jobs.

By UNIX v5, when the first important shell (the Mashey shell) was introduced, pipes were such an important part of UNIX that they merited their own command-line syntax (`'^'`).

Close reading of the paragraphs above, however, reveals a fundamental limitation of pipes: only related processes can communicate over a pipe. In theory, this is not much of a limitation: all UNIX processes on a single processor can trace back to a common ancestor. In practice, though, processes need to be relatively closely related (usually parent and child) to share a pipe. All humans are related if you go back far enough, too. For unrelated processes, the only available IPC mechanism for anything that required passing a lot of information remained the file.

In an early attempt to get around this limitation, Programmer’s Work-

bench UNIX, or PWB UNIX, introduced the “named pipe” or “fifo” (first-in, first-out). To the file system, a fifo looks as though it’s a file with a name. You can open it, close it, write to it, read from it and see it in directory listings. It does not, however, correspond to anything on the disk, and it has the semantics of a pipe. Once data are read from it, they’re gone. If a program writes into a fifo and it fills up, the writer blocks. If a program reads from it and the fifo is empty, the reader blocks. Data that go through a fifo never leave the buffer cache.

(Every once in a while, this bites us on NFS-networked systems. A fifo created on one machine will appear in the directory of other machines that have that file system mounted, but its data structures are only in the kernel of the host machine. Remembering this when we’re trying to use the fifo from another machine and trying to figure out what’s wrong always seems to take us forever.)

Listing 1. mkfifo

```
1  #!/usr/local/bin/perl -w
2  # $ID: mkfifo,v 1.1 1999/07/05 19:12:28 jsh Exp jsh $

3  use strict;
4  use POSIX "mkfifo";
5  use Getopt::Std;
6  use vars qw($opt_m);

7  $0 =~ s(.*//)(/);
8  my $usage = "usage: $0 [-m mode] filename ...\n";
9  getopts('m:') and @ARGV or die $usage;

10 my $default_mode = 0666;
11 $default_mode &= ~(umask 0);

12 sub sym_perms {
13     my $sym = shift;
14     my $mode = $default_mode;

15     my %who = (u => 0700, g => 0070, o => 0007);
16     my %what = (r => 0444, w => 0222, x => 0111);

17     my ($who, $show, $what) = split /([+]=)/, $sym;
18     $who =~ s/a/ugo/g;

19     my @who = split //, $who;
20     my $who_mask = 0;
21     foreach (@who) {
22         $who_mask |= $who{$_};
23     }

24     my @what = split //, $what;
25     my $what_mask = 0;
26     foreach (@what) {
27         $what_mask |= $what{$_};
28     }

29     #printf "%o, %o, %o\n", $who_mask, $what_mask, $change;
30     #print "$show\n";

31     if ($show eq '+') {
32         $mode |= ($who_mask & $what_mask);
33     } elsif ($show eq '-') {
34         $mode &= ~( $who_mask & $what_mask );
35     } elsif ($show eq '=') {
36         $mode = ($mode & ~$who_mask) | ($who_mask & $what_mask);
37     }
38 }

39 sub get_mode {
40     my $mode = shift;
41     my $real_mode;

42     if ($mode =~ /^0?[0-7]3$/ ) {
43         return $real_mode = oct($mode);
44     }
45     $real_mode = sym_perms $mode;
46     return $real_mode unless $real_mode < 0;
47     die "bad mode: $mode\n";
48 }
```

Continued on Page 36

Being part of the file system meant fifos incorporated the best of both worlds: unrelated processes could use them to pass large volumes of data back and forth without requiring disk activity or space. The system call required to create a fifo has varied over the years, but the POSIX version is `mkfifo(const char *path, mode_t mode)`.

mkfifo(1)

There are, however, not one but two POSIX `mkfifo` interfaces: the POSIX.1 interface and the POSIX.2 utility, which lets you create a fifo from the command line. Its syntax is trivially different:

```
mkfifo [-m mode] filename ...
```

With this command, you can create an IPC channel between unrelated processes at the shell level—well, at least on POSIX.2-conforming systems. What do you do with crippled, antiquated, non-POSIX.2-conforming UNIX systems? Or newer operating systems designed to be crippled and antiquated, like Windows NT? One possibility is to use a `mkfifo` from a tool kit like one of the ones we described last month (see “Software Ptools,” August 1999, Page 39, <http://sw.expert.com/C9/SE.C9.AUG.99.pdf>).

When we looked on the Web for a copy of `mkfifo`, we couldn't find one. This was good and bad news: bad news because we wanted it, good news because it gave us a chance to write it and then donate it to Tom Christiansen's pile of free UNIX utilities written in Perl (for the current list, see <http://language.perl.com/ppt>).

Here, then, is our implementation of `mkfifo` (see Listing 1), followed by a dramatic reading.

Scattered about is some boilerplate professionalism. We'll skim over it first for two reasons: 1) getting it out of the way is easy; and 2) it reflects how we write our code. We start by writing these parts first, stubbing out the core functionality of the program. While we're getting these parts right, we have time to think about the hard stuff.

```
49 my $mode = $opt_m ? get_mode $opt_m : $default_mode;
50 foreach my $fifo (@ARGV) {
51     mkfifo $fifo, $mode or die "can't make fifo $fifo: $!\n";
52 }
53 =head1 NAME
54 mkfifo - make named pipes
55 =head1 SYNOPSIS
56 mkfifo "-m mode" filename ...
57 =head1 DESCRIPTION
58 =over 2
59 Create one or more named pipes, in the order specified,
60 with the mode given.
61 If no mode is given, create them with mode 0666,
62 modified by the umask.
63 =back
64 =head1 OPTIONS AND ARGUMENTS
65 =over 8
66 =item I<-m>
67 The mode the fifo should be created with.
68 Numbers must be three octal digits (as for B<chmod(1)>.
69 Symbolic modes, specified the way you can for B<chmod(1)>
70 (such as C<g+w>) are also acceptable.
71 =item I<filename ...>
72 One or more fifo names to create
73 =back
74 =head1 AUTHOR
75 Jeffrey S. Haemer and Louis Krupp
76 =head1 SEE ALSO
77     chmod(1) umask(1) mkfifo(2)
78 =cut
```

In any case, the boilerplate contains the following: Line 1 is the shebang line that invokes the interpreter with the usually useful `-w` (whine copiously) option, which warns of common programming mistakes. We make a lot of mistakes, so line 3 is the `use strict` pragma, which

is even more whiny than `-w`.

Line 2 is the RCS ID. Revision Control System (RCS) is our safety net. It lets us make major modifications to working (or almost-working) versions without having to worry that we'll forget exactly what we changed if our "great new idea"

doesn't pan out the way we'd imagined.

Lines 53 through 78 are documentation. What? We actually write documentation? Well, yes. Writing man pages for everything keeps us from having to decide whether or not we need to. We're not done until it has a man page.

Lines 7 through 9 are argument processing. The call to `getopts('m:')` from the `Getopt::Std` module on line 5 says that our command will take one optional argument `-m`. The colon announces that the option demands an argument. After a successful call to `getopts()`, the value of that argument will be in the variable `$opt_m`, which we declare on line 6 to keep `use strict` from whining.

If the call to `getopts()` fails, we die with a usage message; this is better than the boring Unknown option message issued by `getopts()` itself, because it tells users what they *should* do. In fact, lines 7 and 8 arrange to issue the same usage message for all improper invocations. If you're a Perl beginner, this may seem a little complex. All the more reason to make it boilerplate and get it right at the beginning.

mkfifo(2) vs. mkfifo(1)

Now we're left with the meat (not the most fashionable way to refer to the juicy parts of our code in a vegetarian town like Boulder, Colorado, but we've given up being fashionable).

The easy part of the program is creating the fifo itself: Line 51 calls the POSIX.1 interface, `mkfifo()`, made available by line 4. The second argument to `mkfifo()` specifies the permissions, in octal, with which the fifo should be created. Line 43 translates calls like `mkfifo -m 0777 FOO` directly into the underlying `mkfifo('FOO', 0777)`.

What, though, is line 49 about? Unlike `mkfifo(2)`, which requires that permissions be integers, POSIX.2's `mkfifo` permits symbolic permissions, just like `chmod(1)`. For example, `mkfifo -m g+r FOO` translates as "create a fifo named `FOO`, with the default permissions, but add write permissions for the group." This means we have to figure out how to parse the argument to the `-m` flag.

Symbolic Permissions

One of us, Haemer, spent a little time creating ever-more-complex, incorrect parsing code that worked for most cases and failed at ever-more-obscure edges. Feeling dumber and dumber, he finally realized he was attacking the problem in the wrong way, and turned to a methodology that he has successfully used in the past to conquer difficult programming problems. First, he went to the grocery store and bought a ton of junk food. Next, he brought it back to work, started Netscape to pass the time, and waited.

It was late at night, and the first programmer to rise to the bait was Louis Krupp, who usually starts work around 4 p.m. "How's it going?" asked Louis, coming into Haemer's office and sitting down.

"Fair to partly cloudy. I'm trying to parse symbolic permissions and I can't seem to get it right for anything. I feel like a moron."

"Oh," Louis mumbled, accepting a piece of a candy bar and offering Haemer some potato chips in return. "Like, can you give me an example?"

"I just want to translate things like `g+r` to the right mods to the permissions flags, but every time I get one part right, I break something else."

"Yeah," Louis replied. Eating and thinking a little while longer, he added, "Right." After a few more minutes of silent grazing, Louis went to the whiteboard and wrote down the scheme shown in the `sym_perms()` subroutine on lines 12 through 38.

"Maybe something like that?" said Louis.

Haemer turned it into code, tested it and showed the result to Louis, who had now finished eating most of the potato chips. "Works!"

Louis nodded, smiled, then offered his opinion: "Cool."

In detail: Lines 17 through 28 begin by splitting arguments like `og=rwx` into three parts—"who" (other and group), "how" (set equal to) and "what" (read, write and execute). They then use the information about "who" and "what" to create masks that indicate exactly which parts of the permissions flags should be

changed and in what direction. Line 18 provides the special case for "all" (user, group and other).

Lines 29 and 30 are debugging code, left in but commented out, rather than removed. Even when we're completely certain there are no bugs, we remain confident there will be after enough maintenance and enhancement has taken place.

Lines 31 through 37 show how to apply the masks we've built up to give the user what he wants: setting permissions, adding permissions or taking permissions away.

With this step done, so are we.

Cheerleading

As we said at the beginning, we've donated this code to the Perl Power Tools project, although for reasons we explained last month, we think of it as "Software Ptools," the "P" being psilent.

We like the idea of a complete, freely available, UNIX tool set in Perl. So, it appears, do many other folks. Why? We're not sure, but part of it is just the

fun of doing it. Getting there really is half the fun. Take a small vacation for an evening and join us. Go to <http://language.perl.com/ppt>, take a look at what's done and what's not, and chip in. Please.

Until next time, happy trails. ✍

Jeffrey Copeland (copeland@alumni.caltech.edu) lives in Boulder, CO, and works at Softway Systems Inc. on UNIX internationalization. He spends his spare time rearing children, raising cats and being a thorn in the side of his local school board.

Jeffrey S. Haemer (jsh@usenix.org) works at QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.

Note: The software from this and past Work columns is available at <http://alumni.caltech.edu/~copeland/work> or alternately at <ftp://ftp.expert.com/pub/Work>.