The Problem with Menus, Part 2

by Jeffreys Copeland and Haemer

Last month, we began our discussion of menus. Because we found ourselves needing to prototype the front-panel menus for a system with an embedded controller, we tried to build a simple analog in Perl.

Basically, we began with the insight that a menu is analogous to a file system: We have branches and nodes, with files standing in for individual menu items. We had already explored the Perl package for Inodes and had begun exploring the package for Directories. As our real-world example, we've been thinking about the menus to program our VCR–an example that works for those of us whose VCRs aren't blinking "12:00" at us.

When we left off last month, we had just glanced at the function for listing the elements in our directory class. We gave a quick sketch of the function, which we realize was not a good explanation. So let's look at that function again (see Figure 1).

We can invoke the function with an inode number and some optional flags–the available flags are identical in functionality to the analogous ones for the UNIX `ls` command.

For example, to obtain a recursive listing of the whole directory tree, we'd say

```
$dir->ls(1,'-R');
```

We begin this routine by extracting the flags. Then, if we're using the -d flag, we just list the file we're pointing to, invoking the `Inode::ls` routine for it. Without the -d flag, we list the name of every file at this level.

Jeffrey Copeland (copeland@alumni.caltech.edu) *is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting, cats and children.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

Figure 1. Listing Elements in a Directory Class

```
sub ls {   # always takes an indenting level
    my($self, $indent, @args) = @_;
    local(@dflags) = grep(/^-/, @args);
    local(@margs) = grep(!/^-/, @args);

    if (grep(/-d/, @dflags)) {
        Inode::ls($self, $indent, @margs);
    } else {
        foreach $file (@{$self->{'data'}}) {
           local($name) = $file->{'name'};
           print "\t" x $indent .
               "Name: $name\n";
           $ino = $file->{'ino'};
           if ($ino->{'ftype'} eq 'regular') {
               $ino->ls($indent+1, @margs);
           } else {# a little complicated
               if (dotname($name)) {
                   $ino->ls($indent+1, @margs,
                   @dflags, '-d');
               } elsif (grep(/-R/, @dflags)) {
                   $ino->ls($indent+1, @margs,
                   @dflags);
               } else {
                   $ino->ls($indent+1, @margs,
                   @dflags, '-d');
               }
           }
        }
    }
}
```

If we have a regular file, we can use Inode::ls to deal with it. If we do *not* have a regular file (that is, a directory), we do something a little complicated. First, we must check if the file entry is either . or .. to prevent infinite recursion.

Why? Think about what happens if we list a directory recursively, including the directory itself–we list the directory, and then list the directories in the directory, beginning with the pointer to ourselves. Then we list the directories in *that* directory, beginning with the pointer to ourselves. So, we make a special case, listing the current directory pointer and the pointer to our parent with the -d flag. If we aren't examining one of the special directories, we simply invoke the inode listing with the appropriate flags.

For our next trick, we add another familiar UNIX utility with a two-character name:

```
sub rm {
    my($self, $name) = @_;
    my($i);

    $data = $self->{'data'};
    for ($i = 0; $i < @$data; $i++) {
       $file = @{$data}[$i];
       if ($file->{'name'} eq $name) {
          splice(@$data,$i,1);
          return 1;
       }
    }
}
```

This function takes our directory inode and a file name and puts them on the stack in our local block (remember, that's what the my() construct is for!).

Next, it extracts the data from the directory into an array. It then loops through the entries in the directory, until it finds the one we named in our argument list, and removes it from the data array with the splice() routine. We return a 1 to indicate success.

We'll skip cp and move directly to ln. Actually, we don't bother with cp or mv: Longtime readers will remember from our series about POSIX.1 that ln, mv and cp are effectively the same command.

```
sub ln {    # doesn't do sanity checking
    my($self, $ino, $name) = @_;

    $self->rm($name);

    unless (dotname($name)) {
       $ino->chdotdot($self)
           if ($ino->{'ftype'} ne "regular");
    }

    $entry = {'ino'=>$ino, 'name'=>"$name"};

    $self->append($entry);
}
```

Just like in the UNIX kernel, we attach a name to an inode. We begin by removing the existing name. Note that UNIX doesn't do this: Say touch foo bar; ln foo bar and you get an error. The odd unless clause ensures that if we are linking the current directory, we link it into the parent directory, too. Of course, we could have written it using a compound if, but we're showing off. To do that link, we need a utility routine:

```
sub chdotdot {
    my($self, $new) = @_;

    ln($self, $new, "..")
}
```

And last, for our directory package, we need to emulate the namei function from the kernel. If you're not familiar with it, namei takes a path and returns an inode number.

```
sub namei {
    my($self, $name) = @_;
    my($i);

    $data = $self->{'data'};
    for ($i = 0; $i < @$data; $i++) {
         $file = @{$data}[$i];
         if ($file->{'name'} eq $name) {
                 return $file->{'ino'};
         }
    }
    return(-1);
}
```

We look for the named file in the current directory, looping through the directory entries with a for. When we find it, we return the inode number entry from the file structure. If we don't find the named file–an error–we return -1.

Testing Our Program

To finish our directory package, we provide a set of tests as we did for our Inode package.

In the normal case–we've included the Dir package in another program–we set $TEST = 0;, which prevents the test from being run.

```
$TEST = 0;
END {
       if ($TEST) {
```

At the END of input, if $TEST is set, we run our tests. We begin by throwing up a header and beginning a new test directory. We finish the first paragraph of our test by taking a directory.

```
    print "== Testing package Dir\n\n";
    print "Create a directory:\n";
    $dir = new Dir;
    $dir->ls(1, '-d');
```

Next, we add two files to our test directory, and again list it, to make sure the new files are actually there.

```
    print "Now add some files:\n";

    $ino = new Inode;
    $ino->chdata(["This is file 1\n"]);
    $ino->chftype("regular");
    $dir->ln($ino, "Foo");

    $ino = new Inode;
    $ino->chdata(["This is file 2\n"]);
    $ino->chftype("regular");
    $dir->ln($ino, "Bar");

    $dir->ls(1);
```

We can create files. Can we remove them?

```
    print "Delete a file:\n";
    $dir->rm("Foo");
    $dir->ls(1);
```

Next, we make sure we can add a second directory and make it a subdirectory of the first.

```
print "Create a second directory:\n";
$subdir = new Dir;
$ino = new Inode;
$ino->chdata(["File in a subdirectory\n"]);
$ino->chftype("regular");
$subdir->ln($ino, "Mumble");
$subdir->ls(1);

print "Now make it a subdirectory:\n";
$dir->ln($subdir, "Sub-directory");
$dir->ls(1); }
```

We also want to manipulate the parent directory and convince ourselves that we can do a recursive listing.

```
   print "Check that dot-dot is reset:\n";
   $subdir->ls(1);

   print "Now try a recursive list:\n";
   $dir->ls(1, '-R'); }
```

And then we close it all up.

```
     }
}

1;
__END__
```

This gives us some reassuring output that shows us our directory code is working:

```
== Testing package Dir

Create a directory:
       ftype: directory
       data: ARRAY(0xa2e54)
Now add some files:
    Name: .
         ftype: directory
         data: ARRAY(0xa2e54)
    Name: ..
         ftype: directory
         data: ARRAY(0xa2e54)
    Name: Foo
         ftype: regular
         data: ARRAY(0xc78ec)
    Name: Bar
         ftype: regular
         data: ARRAY(0xc7964)
Delete a file:
    Name: .
         ftype: directory
         data: ARRAY(0xa2e54)
    Name: ..
         ftype: directory
         data: ARRAY(0xa2e54)
    Name: Bar

         ftype: regular
         data: ARRAY(0xc7964)
Create a second directory:
    Name: .
         ftype: directory
         data: ARRAY(0xc7934)
    Name: ..
         ftype: directory
         data: ARRAY(0xc7934)
    Name: Mumble
         ftype: regular
         data: ARRAY(0xc7bbc)
Now make it a subdirectory:
    Name: .
         ftype: directory
         data: ARRAY(0xa2e54)
    Name: ..
         ftype: directory
         data: ARRAY(0xa2e54)
    Name: Bar
         ftype: regular
         data: ARRAY(0xc7964)
    Name: Sub-directory
         ftype: directory
         data: ARRAY(0xc7934)
Check that dot-dot is reset:
    Name: .
         ftype: directory
         data: ARRAY(0xc7934)
    Name: Mumble
         ftype: regular
         data: ARRAY(0xc7bbc)
    Name: ..
         ftype: directory
         data: ARRAY(0xa2e54)
Now try a recursive list:
    Name: .
         ftype: directory
         data: ARRAY(0xa2e54)
    Name: ..
         ftype: directory
         data: ARRAY(0xa2e54)
    Name: Bar
         ftype: regular
         data: ARRAY(0xc7964)
    Name: Sub-directory
         Name: .

ftype: directory

data: ARRAY(0xc7934)
         Name: Mumble

ftype: regular

data: ARRAY(0xc7bbc)
         Name: ..

ftype: directory

data: ARRAY(0xa2e54)
```

Something Completely Different

Our last step in this effort is to produce the menu subsystem itself, which will sit on top of the directory package. The code for this takes up about twice the space we have left, so we'll take a short digression to finish out this week's class.

Recently, we found a file containing a list of book titles that we wanted to alphabetize. No doubt your first reaction would be the same as ours: `sort -o books books` should do the trick. Not quite. In the normal library catalog, we want to do the sort without the articles.

Given the command above, our sample list would be sorted into:

```
A passion for excellence
In search of excellence
Penn & Teller's how to play with your food
The Klingon dictionary
The Standard C dictionary
The cartoon guide to computer science
Xenocide
```

Aside from the oddness of our library shelves, we'd also like our Klingons to appear directly after our search for excellence, for example. We'd ask "What to do?" but you already know the answer: Let's write some software.

A glance at the above list will show you that we also want to sort without regard for the nonalphabetics and uppercase and lowercase. The standard UNIX sort does that for us with the -df flags, so a command like sort -df becomes the center of our solution.

Next, we need to cause the English articles to be ignored by the sort command. The easiest way to do that is to convert them to something that the sort command will ignore–something nonalphabetic. We could try a command like this:

```
sed "s/The /\}/g" |
     sort -df |
     sed "s/\}/The /g"
```

However, this code is going to cause two types of problems. First, if we have some of those nonalphabetic characters in our list, a title with a notation like "In search of excellence {{paperback edition}}," will transform our

By the time we've included all the variations for each article, we'll need a lot of variations of nonalphabetic characters to cover them.

output lines oddly: "In search of excellence An An paperback edition The The."

Second, by the time we've included all the variations for each article–initial capital, all caps, all lowercase, at the beginning of a line, interior to a line–we'll need a lot of variations of nonalphabetic characters to cover them.

We can solve the first problem by converting the articles temporarily into control characters. For example, "the" becomes \033\001. The second problem is not completely soluble because we need to preserve some of that information–on output, we need to distinguish between "The" and "THE"–but we can solve part of the problem by relying on the more powerful regular expression matching in Perl. If we do the conversion as

```
perl -pe 's/\bThe /\033\001/g' |
     sort -fd |
     perl -pe 's/\033\001/The /g'
```

we've pretty much got things under control.

Notice that we're using the Perl regular expression \b to signify a word boundary. Aside from the English articles–notice that we're ignoring foreign ones like *La Boheme* and *Die Fledermaus*–the other variation in alphabetical order that we're used to seeing in libraries is to sort "Mc" and "Mac" as though they are identical. This means that correct alphabetical order is McDonald, MacMillan. We can also jigger our sort to take care of this variation with a conversion like s/\bMc/Ma\009c/g.

Given all that, we can put our Perl "input" and "output" scripts together, thus

```
perl -pe '
     s/\bThe /\033\001/g;
     s/\bthe /\033\002/g;
     s/\bTHE /\033\003/g;
     s/\bA /\033\004/g;
     s/\ba /\033\005/g;
     s/\bAn /\033\006/g;
     s/\ban /\033\007/g;
     s/\bAN /\033\008/g;
     s/\bMc/Ma\009c/g;
' | sort -fd | perl -pe '
     s/\033\001/The /g;
     s/\033\002/the /g;
     s/\033\003/THE /g;
     s/\033\004/A /g;
     s/\033\005/a /g;
     s/\033\006/An /g;
     s/\033\007/an /g;
     s/\033\008/AN /g;
     s/Ma\009c/Mc/g;
```

This gives us the output we want:

```
The cartoon guide to computer science
In search of excellence
The Klingon dictionary
A passion for excellence
Penn & Teller's how to play with your food
The Standard C library
Xenocide
```

Notice that we've built software that's strictly a filter. Exercise for the reader: Adjust this script so that it recognizes file names on the command line and acts correctly.

We've also written this as a shell script, where we invoke Perl twice. Exercise for the reader: Rewrite this so that it's strictly a Perl script and uses the Perl internal sort routine in its middle step. Or even better, use one of the interprocess communications tricks available in Perl 5 (see the perlipc(1) man page, for hints).

That's all for this month. Next time, we'll complete our discussion of menus. After that, we'll finish our discussion of "Work" and move onto a new series, unless any of you have a topic you feel we have missed. Meanwhile, we hope your summer is going nicely. ▲