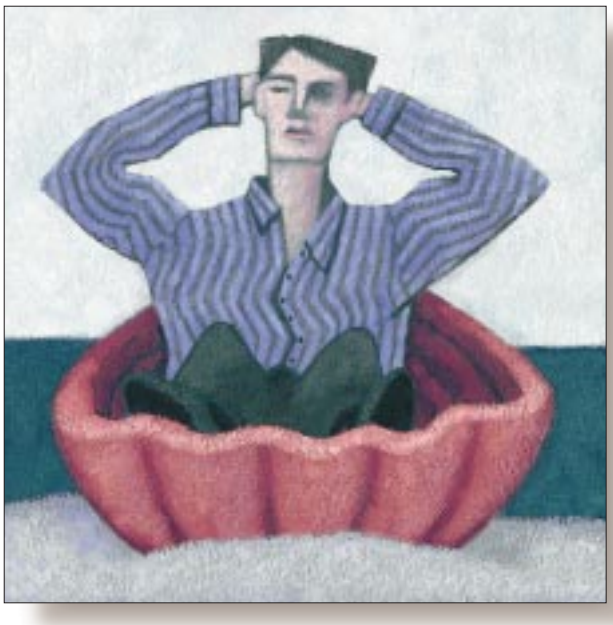
by Jeffreys Copeland and Haemer



"We will encourage you to develop the three great virtues of a programmer: laziness, impatience and hubris."
– Larry Wall

"Work is the curse of the drinking classes."
– Oscar Wilde

Work and How to Avoid It

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.html`.

If you hate doing unnecessary work, then this month's column is right for you–we'll give you some tips on sloth. On the other hand, if you like doing unnecessary work, then you should read this month's column because you'll ignore our advice and, therefore, reading it would be unnecessary work. Bertrand Russell would be proud of us.

The quotation from Larry Wall that introduces this column is taken from *Programming Perl*, his book about a language designed to save us all a lot of work. Some programmers miss this point. Given an assignment by his boss to write a Perl script to format and print a bunch of files, one of our coworkers produced the following:

```
#!/usr/bin/perl

# Allow files to be written over
system("unset noclobber");

# Set up some environment variables
$ENV{"PRINTER"} = "Viper";

# Create the files
system("groff file1 > /tmp/file1.ps");
system("groff file2 > /tmp/file2.ps");
system("groff file3 > /tmp/file3.ps");
system("groff file4 > /tmp/file4.ps");
system("groff file5 > /tmp/file5.ps");
system("groff file6 > /tmp/file6.ps");
system("groff file7 > /tmp/file7.ps");

# Print the files
system("lpr /tmp/file1.ps");
system("lpr /tmp/file2.ps");
system("lpr /tmp/file3.ps");
system("lpr /tmp/file4.ps");
system("lpr /tmp/file5.ps");
system("lpr /tmp/file6.ps");
system("lpr /tmp/file7.ps");

# Cleanup
system("/bin/rm /tmp/file1.ps");
system("/bin/rm /tmp/file2.ps");
system("/bin/rm /tmp/file3.ps");
system("/bin/rm /tmp/file4.ps");
system("/bin/rm /tmp/file5.ps");
system("/bin/rm /tmp/file6.ps");
system("/bin/rm /tmp/file7.ps");
```

What's wrong with this picture? First, while Perl eschews many sacred computer science cattle–nowhere, for example, can you find a Bachus-Naur Form grammar for Perl–loops and subroutines are both venerable control

structures that even Perl provides. All those nearly identical system calls could have been replaced with the following:

```
sub one_file {
    $file = shift;
    system("groff $file > /tmp/$file.ps");
    system("lpr /tmp/$file.ps");
    system("/bin/rm /tmp/$file.ps"); }

foreach $file (file1 file2 file3 file4
        file5 file6 file7) {
    one_file($file);
}
```

Indeed, though Perl lacks grammatical simplicity, many of us use it because it has power and elegance of expression. For example, many Perl programmers might at least pare the code down to this:

```
sub one_file {
    $file = shift;
    system "groff $file > /tmp/$file.ps";
    system "lpr /tmp/$file.ps";
    system "/bin/rm /tmp/$file.ps"; }

foreach (1..7) {
    one_file "file$_";
}
```

And how about making the program more flexible by replacing the hard-wired list of files with a list supplied on the command line:

```
while (@ARGV) {
    one_file shift;
}
```

We Roll up Our Sleeves to Do Even Less Work

Not enough, not enough. Each of those `system()` calls forks a subshell. We could cut it down from 21 subshells (7x3) to seven by making each subshell perform more than one action. For example,

```
sub one_file {
    $file = shift;
    system "
      groff $file > /tmp/$file.ps &&
      lpr /tmp/$file.ps
    ";
    die "one_file($file) failed" if $? != 0;
    unlink "/tmp/$file.ps" or
        die "can't unlink /tmp/$file.ps";
}

while (@ARGV) {
    one_file shift;
}
```

Here, each call to `one_file()` creates a single subshell, which performs both the `groff` and `lpr` for the source file to be printed. We've also done a couple of pieces of noteworthy sanity checking: First, the double ampersand, `&&`. The sequence

```
$ command_1 && command_2
```

translates as "do `command_1`; if that succeeds, then do `command_2`." This idiom can be a surprisingly useful safeguard. We recently helped a coworker restore his home directory from tape after he had done the following in his home directory:

```
$ cd bogus_directory_name; rm -rf *
bogus_directory_name not found
```

We recommended typing this instead:

```
$ cd bogus_directory_name && rm -rf *
```

which only does the `rm -rf` if the `cd` command succeeds. Thus in our rewritten script, we only try to print the file when our invocation of `groff` succeeds.

We've also taken the reasonable precaution of checking the exit status of the `system()` call, which is held in the predefined Perl variable, `$?`. Moreover, instead of calling the shell-level `rm` command, we use the Perl function, `unlink()`.

As long as we're trying to be careful, we ought to turn

```
#!/usr/bin/perl
```

into

```
#!/usr/bin/perl -w
```

But when we do so, we get a sharp reprimand:

```
Can't exec "unset": No such file or directory at 1 line 4
```

Here, what we're seeing is that `unset` is not an external executable command, but a shell built-in.

```
$ type unset
unset is a shell built-in
```

Just as a `cd` in one shell doesn't affect the current working directory of another unrelated shell, `set` and `unset` change the values of shell variables for a particular shell and, potentially, its children. Perl is warning you that

```
system("unset noclobber");
```

is useless, because it only affects the value of `noclobber` for the subshell invoked by that specific call to `system()`; it has no effect on the subshells invoked by the other `system()` calls.

Each `system()` call executes a miniature shell script. The original program is nothing more than a shell script broken into 23 little, one-line shell scripts, sequentially invoked by the Perl interpreter. Even experienced programmers will occasionally forget that rules in a `Makefile` have this same property.

The production

```
clean_tmp:
   cd /tmp
   rm *
```

will change directories to `/tmp` in one subshell and then, in a separate subshell, remove all the files in your *current* directory. The correct way to write this production is:

```
clean_tmp:
   cd /tmp; rm *
```

Lazier and Lazier

So this brings us to the crux of the matter: Why fork all of these subshells? Here's the more-or-less-equivalent shell script:

```
#!/bin/sh

# Allow files to be written over
unset noclobber;

# Set up some environment variables
PRINTER=Viper;

# Create the files
groff file1 > /tmp/file1.ps
groff file2 > /tmp/file2.ps
groff file3 > /tmp/file3.ps
groff file4 > /tmp/file4.ps
groff file5 > /tmp/file5.ps
groff file6 > /tmp/file6.ps
groff file7 > /tmp/file7.ps

# Print the files
lpr /tmp/file1.ps
lpr /tmp/file2.ps
lpr /tmp/file3.ps
lpr /tmp/file4.ps
lpr /tmp/file5.ps
lpr /tmp/file6.ps
lpr /tmp/file7.ps

# Cleanup
/bin/rm /tmp/file1.ps
/bin/rm /tmp/file2.ps
/bin/rm /tmp/file3.ps
/bin/rm /tmp/file4.ps
/bin/rm /tmp/file5.ps
/bin/rm /tmp/file6.ps
/bin/rm /tmp/file7.ps
```

If nothing else, there's a third less typing to do–639 characters instead of 900. But it isn't just "nothing else." If we do timing tests, this version runs about 12% faster. Of course, just as we did with the first script, we can make the program more general and shorter by adding loops and subroutines:

```
#!/bin/sh
unset noclobber
PRINTER=Viper
one_file() {
    groff $1 > /tmp/$1.ps
    /usr/bin/lpr /tmp/$1.ps
    /bin/rm /tmp/$1.ps
}

for i in file*
do
    one_file $i;
done
```

Oh, but that's still far too much work. Because `lpr` and `rm` will both take more than one file name, we can drop their invocations to a single call apiece, like this:

```
#!/bin/sh
set noclobber
for i in file*
do
    groff $i > $i.ps
done
/usr/bin/lpr -PViper /tmp/file*.ps
/bin/rm /tmp/file*.ps
```

We eliminate the extra variable assignment by using the `-P` argument to `lpr`. We're down to one invocation of the shell, one of `lpr`, one of `rm` and a handful of `groff`s.

Still, the restriction on file names means we might have to rename our input files to suit the script before invoking it. Moreover, in all these versions we've been assuming that the files are in the current directory, and that the only files named `file*.ps` in `/tmp` are the ones we have put there. We fix these problems by again pulling the file names from the command line, and by constructing our own temporary directory for output, removing it when we're done. In the end, we arrive at this 12-line script:

```
1)  #!/bin/sh
2)  PRINTDIR=/tmp/$$.dir
3)  mkdir $PRINTDIR || exit 1
4)
5)  for i in $*
6)  do
7)    OUTFILE=$PRINTDIR/${i##*/}.ps
8)    groff $i > $OUTFILE || rm -f $OUTFILE
9)  done
10) cd $PRINTDIR
11) lpr -P${PRINTER:-Viper} *
12) rm -r $PRINTDIR
```

Here's what it all means:

Line 1 makes UNIX pass the script to the correct shell, even if it isn't the user's login shell.

Lines 2 and 3 create a directory for our temporary files, using the process ID $$ to generate a unique name.

Lines 5 through 9 are a loop that formats all the input files named on the command line. Line 7 strips any directory information from the input file name, in case it was specified with an absolute path, so that if the file name is `$HOME/project/printfiles/foo`, `$OUTFILE` becomes `$PRINTDIR/foo.ps`.

In older shells, this sort of manipulation was done with the stand-alone executable `/usr/bin/basename`, but POSIX-conforming shells allow users to extract substrings of shell variables with a built-in facility called *parameter expansion*. In this case, the expression `${i##*/}` means "Give me the value of `$i` stripped of the longest possible prefix that matches the glob (shell) expression `*/`" – in other words, the file name, but with any directory name stripped off.

Line 8 formats an input file, taking care to remove any unsuccessful attempts.

Lines 10 and 11 print all successful formatting efforts. In line 11, we see another example of parameter expansion. The expression `${PRINTER:-Viper}` means "use the value of `$PRINTER`, if it's set in the environment (after all, you might, someday, want to print to another printer). If it isn't set, use the value `Viper` as the default." You can find more information about other kinds of parameter expansion in the shell manual page.

Line 12 removes all temporary files by removing the temporary directory. You may wonder whether you can remove a directory while you're still in it. You can, and because we have finished, we do. In a more robust version, we might use a signal handler to remove the temporary directory if the script terminates prematurely, but that seems like overkill here. Finally, because the call to `set noclobber` seemed questionable to us in the first place, we've eliminated the call altogether.

OK, where are we? We've gone from a 34-line Perl script to a 12-line shell script that's both more robust and more general. At the same time, we've made it run faster by reducing an invocation of the Perl compiler/interpreter plus 22 separate invocations of the shell, to a single invocation of the shell.

Unfortunately, all of this presupposes that we can go to our boss and say, "Please don't micromanage me by forcing me to use Perl for a job better done with a simple shell script." If you can't do that, we suggest an easier, alternative approach. First, write a simple shell script, as above, and place it in the directory `/usr/local/real_code`. Second, for your boss, write the following, easy-to-maintain Perl script:

```
#!/usr/bin/perl -w
$real = "/usr/local/real_code";
$0 =~ s(.*/)();             # basename
system "$real/$0 @ARGV";    # call real version
```

And Now We Rest

That's all for this month. Next time, we'll eschew sloth and take as the text for our sermon a sentence from the introduction to *The Art of Computer Programming* by Donald Knuth: "The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music."

Until then, happy trails.