by Jeffreys Copeland and Haemer

Hoist with Our Own Petard

Jeffrey Copeland (copeland@alumni.caltech.edu) *is a member of the technical staff at QMS' R&D group in Boulder, CO. He's been a software consultant to the Hugo award administrators for several years. He spends his spare time raising children and cats.*

Jeffrey S. Haemer (jsh@canary.com) *now works for QMS, too, and is having a great time. 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.*

Several months ago, we wrote our column twice. Not because we thought it was a cool idea, but because we typed `rm 17.mm` when we meant `rm 17.ps` and, as a result, deleted the source file for our column, rather than the formatted output. (We came back to this thought because QMS is currently moving its Boulder office and after 90 years of experience of moving houses and offices between us, we've come to expect disasters.) In any event, we had to ask ourselves what we could have done to prevent a slip of the fingers from doubling our workload.

We typically use the GNU file utilities layered on top of whatever UNIX we run on our workstation. This gives us a couple of advantages: Not only do we get the same commands with the same options on each machine, but the GNU utilities allow us to write and run POSIX.2-conforming scripts even on old systems like our still-BSD-based Suns. (This is also why Interactive Systems Corp.'s Programmer's Work Bench, which provided UNIX utilities for VMS, was a good idea, and why we install Mortice Kern Systems Inc.'s MKS toolkit on all our DOS and Windows machines.)

Haemer, being older and wiser than Copeland, has installed Linux on his desktop machine so that he gets the GNU tools as his regular commands.

A few years ago when widespread computer voice recognition was forecast, some wag suggested that a particularly effective act of corporate terrorism would be to stand in the middle of a bank of cubicles and yell: "File … Delete … Yes." Most of us are still safe from this kind of joke, yet we terrorize ourselves by deleting files we wanted to keep more frequently than we'd like.

When we started to figure out how to unshoot ourselves in the foot, we quickly discovered that GNU's fileutils have a backup capability that lets us preserve versions of files we overwrite. For example, we can say `cp -b foo bar`, and if there is a file named `bar` in our directory, it is renamed `bar~` before `foo` is copied to `bar`.

Why is the backup file named `bar~`? Because the Free Software Foundation (FSF) folks are `emacs`-centric and use the `emacs` backup file conventions. Fortunately, if the environment variable `SINGLE_BACKUP_SUFFIX` is set, its value is used for the backup name. Thus,

```
$ SINGLE_BACKUP_SUFFIX=.bak
$ ls
foo   bar
$ mv -b foo bar
$ ls
bar   bar.bak
```

The truly paranoid, or the recent immigrant from VMS-land, can even set the `VERSION_CONTROL` environment variable to add version numbers to the backup files.

Alas, of all the fileutils, `rm` is the only one that does not provide backup capabilities. So that brings us back to square one.

Defusing the Petard

Why don't fileutils perform backup for `rm`? We're not sure. Still, it seems like a small enough hole to fill, so we'll tackle it here. Rather than trying to layer the backup functionality into the FSF code, we can just write a shell script replacement for `rm` built from `mv` and `cp`. We do this because we're lazy–it saves us wasting time on two columns explaining the FSF fileutils library functions–and because doing so illustrates a useful feature of the shell: error handling with the `trap` command.

First, let's build something that just removes a single file while backing it up. We can use something like this:

```
#!/usr/local/bin/bash

TMPFILE=$$
touch $TMPFILE
mv -b $TMPFILE $1
rm $1
```

This has the advantage that our new command–for now, we'll call it `saferm`–understands the same environment variables as GNU's `mv`.

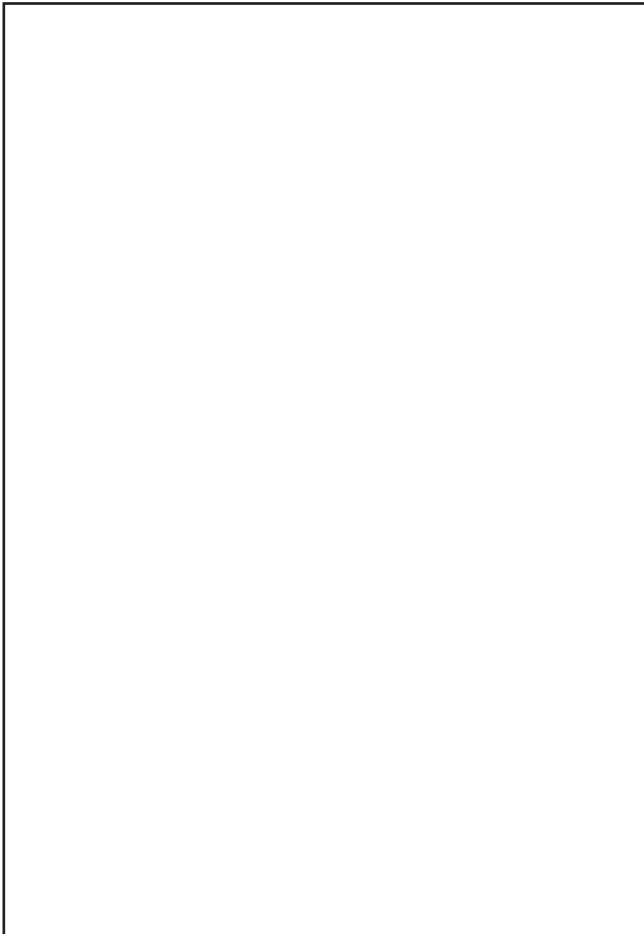
Is that enough? What happens if, by analogy with `rm`, we say `saferm foo bar mumble`? Clearly, only `foo` gets backed up. Unlike `mv` and `cp`, which require two file names as arguments, `rm` takes one or more.

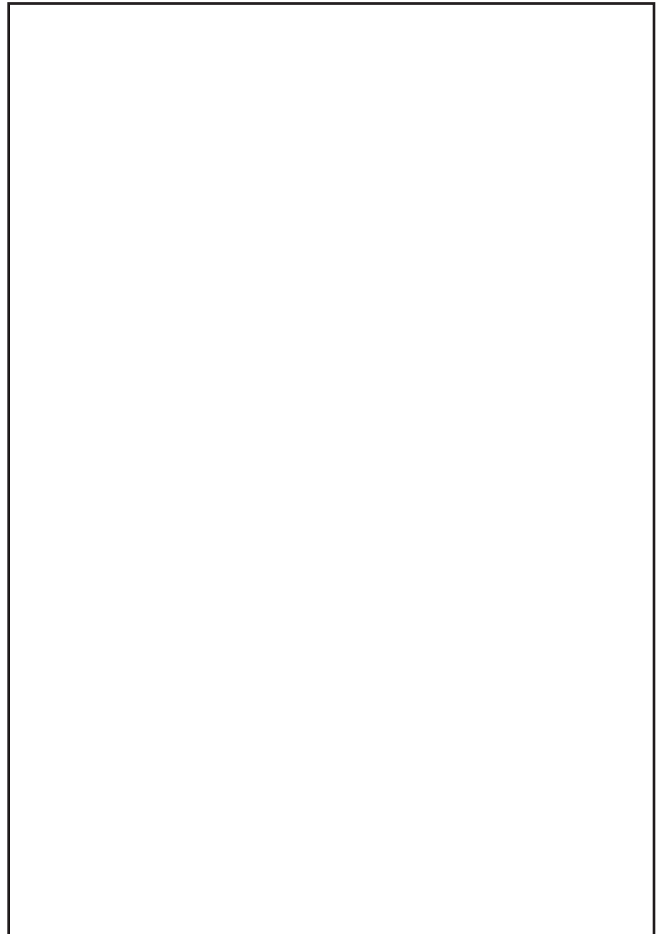
Just as annoying is a misstep such as `rm -rf foo`, which translates into `mv $TMPFILE -rf`, and then into the following complaint:

```
mv: illegal option — r
Try 'mv —help' for more information.
```

We need to both handle non-file name arguments and to provide for graceful failures. We'll show you how in the next section.

You could argue that people using our new shell script should know better than to type things like that, and should





just RTFM. That's not a very persuasive argument to those of us who have read the manual for `rm` and still mistakenly type `rm 17.mm`.

Argument Handling

First things first. We can handle multiple files by just looping through the arguments:

```
#!/usr/local/bin/bash

TMPFILE=$$
for i
do
  touch $TMPFILE
  mv -b $TMPFILE $i
  rm $i
done
```

This requires rather a lot of temporary file creation and removal, however, which we can short-circuit by changing the algorithm slightly:

```
#!/usr/local/bin/bash

TMPFILE=$$
touch $TMPFILE
for i
do
  mv -b $TMPFILE $i
  TMPFILE=$i
done
rm $TMPFILE
```

Any scripts we write always check for the correct number of arguments, and this one will be no exception. We ensure that our utility is being passed at least *one* file name with this simple test:

```
test $# -gt 0 || die $USAGE
```

We have discussed the magic of shell functions, parameter expansion and shortcut evaluation of logical expressions, all of which are used by this line, in our previous columns–most recently, last month. If you don't have a copy of the May 1997 issue handy, then you can read about all these topics in the man page, `shell(1)`.

Handling optional and incorrect arguments is a little trickier. We could use `getopts`, but for simple argument handling, we prefer the flexibility of the shell's `case` statement, whose ability to recognize shell "glob" expressions such as `*foo*` lets us whip up quick-and-dirty argument parsers.

For starters, let's reject *all* switches. The regular expression `-*` will recognize any leading switches, but to recognize a bad call such as `saferm foo -r` requires a slightly trickier expression, `*' -'*`. Combining all of these, we get this:

```
#!/usr/local/bin/bash

ARGV0=${0##*/}
USAGE="usage: ${0##*/} filename [filename]"
warn() { echo $ARGV0: $* 2>&1 }
die() { echo $* 2>&1; exit 1; }

test $# -gt 0 || die $USAGE
case $* in
  -*|*' -'*) die $USAGE ;;
esac

TMPFILE=$$
touch $TMPFILE
for i
do
  if mv -b $TMPFILE $i 2>/dev/null
  then
     TMPFILE=$i
  else
     warn "cannot back up $i"
  fi
done
rm $TMPFILE
```

Exercise for the reader: Figure out what switches `saferm` should take, and add them.

Notice that we've also taken care to issue our own warning message whenever one of the renames fails, instead of letting `mv` issue the message.

Oops!

So, we're done. Right? Would that we were. Early on, we remember telling a junior systems administrator that he could do what he wanted by typing `rm *.out`. After a long pause, he asked, "What does it mean when it says, '.out: no such file or directory'?" His thumb had accidentally typed in a space between `*` and `.out`. Looking on the bright side, after that, it wasn't hard to convince him of the importance of doing backups.

At this point, `saferm * .out` is safer. But what happens if we accidentally issue such a command, notice it, and quickly type `^C` to avoid renaming all our files? Consider the case where we type it right after creating the temporary file, `$TMPFILE`. All our files will be intact, but we end up with an extra file in the directory, named after our process ID. If it's run longer than that, then things are more complicated. If, for example, `xyzzy` has just been backed up to `xyzzy~` and has been replaced by the temporary file. Then, after killing `saferm`, we'll have to search through our directory listing to discover that we have both a `xyzzy` and a `xyzzy~`, in order to know that `xyzzy` is the extra, temporary file.

Of course, if this were a C program, we'd write a signal handler to delete the temporary file before exiting. But this is a shell script, so we'll write a signal handler to delete the temporary file before exiting. (Didn't know we could do that, huh?)

At your shell prompt, try typing `trap 'echo Hello, world' SIGINT`, then, type some `^C` characters to send the current shell an interrupt. Each `^C` should produce `Hello, world` as its output.

Resetting this to restore standard shell behavior is a little trickier. `trap '' SIGINT` makes the shell completely ignore `^C`,

instead of providing a new prompt. What works for us is `trap 'echo -n' SIGINT`.

Before using `trap` to put a signal handler into our program, we offer one last trick: If you list all the signals using `trap -l`, you'll see that the first one is the pseudo-signal, `EXIT`. Setting a trap for `EXIT` causes that signal handler to be executed on program completion. This means that instead of having an `rm $TMPFILE` statement at the end of our program, we can set a signal handler at the beginning to get it removed.

Our final program looks like this:

```
#!/usr/local/bin/bash

ARGV0=${0##*/}
USAGE="usage: ${0##*/} filename [filename]"
warn() { echo $ARGV0: $* 2>&1 }
die() { echo $* 2>&1; exit 1; }

test $# -gt 0 || die $USAGE
case $* in
  -*|*' -'*) die $USAGE ;;
esac

TMPFILE=$$
trap 'rm -f $TMPFILE; exit 1'  \
  SIGHUP SIGINT SIGQUIT SIGTERM
trap 'rm -f $TMPFILE' EXIT
touch $TMPFILE ||
  die "$ARGV0: cannot create $TMPFILE"

for i
do
  if mv -b $TMPFILE $i 2>/dev/null
  then
    TMPFILE=$i
  else
    warn "cannot back up $i"
  fi
done
```

We now have a full-blown `saferm` program, replete with argument parsing, usage messages and error and signal handling, in less than a page of code. Not bad.

Undeleting

OK, now we're feeling pretty confident and demo the software to our boss, who says, after we're done, "Great! You've moved all your files to some new name. Now, make an `unrm` to put them back."

How to design this isn't as obvious. For example, if we had just `saferm`'d two files, named `JJ` and `Allie`, we could imagine undeleting them by typing something like `unrm JJ Allie`. An `unrm` that worked like this might look for `JJ~` and `Allie~` and move them back to their original names. But suppose we start with hundreds of files, named `xaa` through `xzz`? If we delete them using `saferm *`, then we can't just type `unrm *` to bring them back, because this would translate into `unrm xaa~ xab~ ...`, which would, of course, look for files named `xaa~~` and so on to rename.

We could make `unrm *` work by having it look through its file name arguments for ones that end in ~, but that causes other problems. Imagine this: We've already done a few `saferm` commands in the current directory, so that there are many files with legitimate backup versions. We type `saferm '* .out'` by accident. Realizing our error, we then type `^C` to kill the command partway through. Now, we have a complete mess: some files with legitimate backup versions, some with backup versions that need to be renamed to their original names, and some with no backup versions at all.

We could expand the signal handler for `saferm` to keep track of the files that have been backed up, and either undo the damage or report which files will need `unrm`-ing, but it's not hard to invent special cases where not even that is good enough. What's worse, if we've set `SIMPLE_BACKUP_SUFFIX` or `VERSION_CONTROL`, then straightforward algorithms that look for files named `*~` aren't going to work either. What should we do? Our first instinct is to fall back on our favorite technical answer: We don't know.

Some further reflection, however, leads us to a more useful conclusion, and to another column. In concert with a safe version of `rm`, there is another strategy. If you have a lot of important files, and you want to be able to keep backup versions and restore them whenever you accidentally delete them, do what we usually do: use a source-code control system. Next month, we will look at Revision Control System, or RCS.

Until then, happy trails.