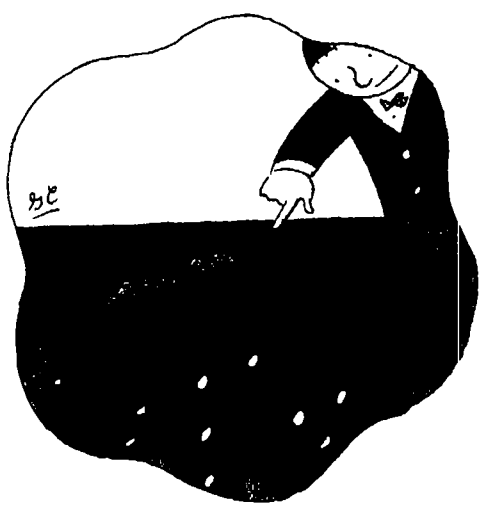
Try, Try, Try Again

by Jeffreys Copeland and Haemer

Earlier this month, Jeffrey Haemer received email from an old friend, Tom Schneider, who's a research biologist at the National Cancer Institute. In it, Tom says, "You always pointed out the importance of tool building...I have built a shell script called `waitforchange` that hangs in a loop watching a file for any change (first date change then `diff`)." Doesn't sound like much, but, Tom continues, "on top of that I can build some neat things."

Tom then describes how he uses `waitforchange` inside another script, `atchange`, that waits for a file to change and then executes a command. `atchange` has become an integral part of Tom's computing environment. He can edit programs in one window, while another window running `atchange` will recompile the file whenever he writes it out.

But Tom is having a few implementation problems. He asks, "Maybe you would see a way to



make it faster, although that isn't really an issue since at the moment it uses 100% of the CPU...However, it only detects change, not the completion of file writing, so will bomb on occasion because it will try to run a program that is incompletely written...Perhaps you have an idea about this?"

Perhaps.

OK, remember we had promised you a wrap-up of what we started

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.*

last month: How to make HTML documents look good instead of letting the browser do whatever it likes? Well, we've decided to put that back a month and attack this problem instead. After all, we told you last month where to get the code for the formatter, and besides, helping our friends always comes first.

atchange, Cut One

Tom's original scheme used a pair of C-shell scripts. (Hey, Tom still programs in Pascal.) When we attacked the problem, we started with one of our favorite hammers, `perl`, reasoning that it would be easier to pick a neutral language than to engage in shell wars or learn how to make the C shell work as a programming language.

Still, we attempted to match his variable names, logic and command-line syntax. Tom may have to enhance it and fix a few bugs.

Listing 1 shows our first rewrite.

Listing 1

```
#!/bin/perl

$0 =~ s(.*/)();              # basename
$usage = "usage: $0 command";
@ARGV > 1 || die $usage;  # check for proper invocation

$file = shift;               # peel off the filename
$cmd = join(" ", @ARGV); #   and the command

$old = (stat($file))[9]; # now get the mod time
while(1) {
    sleep 1;
    $new = (stat($file))[9];
    if ($old != $new) {          # if it's changed,
        while (1) {
            $old = $new;
            sleep 1;
            $new = (stat($file))[9];
            if ($old == $new) {  # but not still changing,
                system($cmd); #   do the command
                last;
            }
        }
    }
}
```

We have commented heavily because Tom doesn't know `perl`, but we'll also do a dramatic reading for you here.

The first paragraph constructs a usage message and checks that the command has been properly invoked.

```
$ atchange
usage: atchange command
```

We use the basename of the command because we prefer it to messages such as

```
$ atchange
usage: /usr/local/bin/atchange command
```

but we use $0 because we sometimes agonize over what to call a command. (In three days, we changed its name to watch, haunt and back to atchange again.)

Having guaranteed ourselves that there are at least two arguments, the second paragraph grabs the first argument for the name of the file to watch and concatenates the remainder of the command line to get the command to execute when that file changes.

The third paragraph does the real work. Instead of trying to diff the files, we'll just track the modification time of the file. As we discussed in detail in an earlier article (see "In Which We Write *in*," October 1993, Page 34), a file's stat structure has three times. Of these, the mod time is the last time the file was written–the time shown when we do an ls -l (except that in the stat structure the time is stored to the second). This means that if someone reads the file and writes it out unchanged, it will still trigger atchange. We can live with that, as long as we document it.

At each iteration of our infinite loop, we sleep for a second, and then compare the current modification time to the last modification time, which we've stored away. If the modification time has changed, we loop again, cat-napping until it stops changing and then execute the command.

There are a handful of problems with this design. First, it can take up to two seconds after a change before the command executes. An advantage of atchange over something like make is that actions are triggered immediately and automatically. The longer the delay, the smaller that advantage.

Second, if anything changes the file a second time during the sleep interval, atchange will still run but the command will only run once. We're satisfied to live with these design choices. We can always go back and decrease the sleep time to, say, a quarter of a second by replacing `sleep(1);` with `select(undef, undef, undef, 0.25);`. Exercise for the reader: let the user set the sleep time with a command-line argument.

atchange, Cut Two

We sent the code to Tom, who announced that it worked better than his old code and that he'd already switched over. But he had discovered another problem. Tom often finds it necessary to run several atchange jobs at once. For example, he might have one window running atchange pc scan to recompile scan.p whenever he writes it out, and another running atchange scan scan to run scan as soon as it has been recompiled. "Can you do something about that?" Tom asks.

One approach would have been to let each file change trigger a sequence of commands That's not difficult but would still require a separate invocation of atchange for every file he wanted to watch. However, it didn't require much more code to tweak the command to permit input files such as the following:

```
#!/usr/local/bin/atchange

/tmp/foo  echo foo changed

/tmp/bar  echo bar changed
```

For backward compatibility, we allowed an argument count of more than one to trigger the original behavior. However, when our improved atchange is invoked with exactly one argument, it treats that argument as a command file.

As a bonus, this behavior makes it easy to take advantage of the !# magic cookie that we discussed in detail in an earlier column (see "Envelopes," May 1995, Page 35). Thus,

```
$ atchange /tmp/hello echo hello, world &
$ touch /tmp/hello
hello, world
```

but,

```
$ example &
$ touch /tmp/foo
foo changed
$ touch /tmp/bar
bar changed
```

The code is straightforward, but we should point out a few things. First, instead of a single file and command, we now have an array of commands, %cmd, indexed by file name. Similarly, the mod time, $old, is replaced by an array of mod times, %old. We've turned the inner loop of our earlier program into a subroutine that checks to see if the file's modification time has changed. If it has, we look up and execute the appropriate command, poking the new mod time back into the %old array for future reference.

The subroutine takes a single argument, the file name. This design means that when we catch a file changing, we still wait until it has stopped before doing anything. But if changes are rare, there's only a delay of about a second before we notice a change in any file.

Listing 2 shows the code.

Listing 2

```
#!/usr/bin/perl

$0 =~ s(.*/)();          # basename
$usage = "usage: $0 filename cmd | $0 command_file";
@ARGV || die $usage;     # check for proper invocation

if (@ARGV > 1) {         # it's a file and a command
   $file = shift;           # peel off the filename
   $cmd{$file} = join(" ", @ARGV); #    and the command
   $old{$file} = (stat($file))[9]; # mod time.
} else {            # it's a program
   open(PGM, shift) || die "Can't open $_: $!";
   while(<PGM>) {
       s/#.*//; # comments
       @F = split;
       next if (@F < 1);      # blank lines
       if (@F == 1) { warn "odd line"; next; };
       $file = shift(@F);
       $cmd{$file} = join(" ", @F);
       $old{$file} = (stat($file))[9]; # mod time.
   }
}

while(1) {
   sleep 1; # wait a second, then
   foreach (keys %cmd) {  #  rip through the whole list
       atchange($_);
   }
}

sub atchange {  # if $file has changed, do $cmd{$file}
   my($file) = @_;
   my($new);

   $new = (stat($file))[9];
   return 0 if ($old{$file} == $new);
   while (1) {              # wait until it stops changing
       $old{$file} = $new;
       sleep 1;
       $new = (stat($file))[9];
       if ($old{$file} == $new) {
           system($cmd{$file});
           return 1;
       }
   }
}
```

atchange, Cut Three

At this point, Tom is pretty happy, but we aren't satisfied. We would like to make it easier to tie a file change to an entire list of commands. We can say

```
atchange /tmp/foo 'date; echo hello, world',
```

but writing a for loop with a lot of commands, or a case statement with a lot of cases, would be inconvenient.

Also, atchange has no memory. There's no way for it to know how many times it has been called, or for what.

Then, there's the nagging issue of efficiency. We have already eliminated the need to have a separate atchange process for every file we watch, but we still fork a subshell each time a file changes.

Our latest version fixes all of these problems and more. But before we present the code, Listing 3 shows an example of an input file.

Listing 3. Example Input File

```
#!/usr/local/bin/atchange
#
# Here's a program for atchange

        HELLO="hello world"    # set a variable
        echo $PS1

/tmp/hello        echo $HELLO        # all one script

        datefn() {                 # define a function
                  echo the date: $(date)
        }

/tmp/date datefn
        echo -n "$PWD$ "

        counter=0

/tmp/counter      # commands can span multiple lines
        echo $counter
        let counter=counter+1

        CLEARSTR=$(clear)

/tmp/iterator
        echo $CLEARSTR
        let iterator=iterator+1
        echo $iterator | tee /tmp/iterator

/tmp/zero_counter
        let counter=0
        touch /tmp/counter
```

The actions for /tmp/hello and /tmp/date illustrate that our third version of atchange allows users to define variables and functions. The actions for /tmp/counter and /tmp/iterator show that this atchange has a memory.

The action for /tmp/zero_counter shows that actions taken for one file can interact in interesting ways with actions for other files.

Because we're passing paragraphs of commands to the shell, we don't need to escape the new lines in for loops in the atchange script as we would in a Makefile. One way to provide this much functionality would have been to rewrite atchange to have a lexical analyzer and a parser, and to maintain a symbol table. We decided to let someone else do the work for us.

We began by inserting the following paragraph near the beginning:

```
$shell = $ENV{"SHELL"} ? $ENV{"SHELL"} : "/bin/sh";
open(SHELL,"|$shell")||die "Can't pipe to $shell: $!";
select(SHELL); $| = 1;
```

This spawns a single subshell and connects the default output from our program to the stdin of that shell. With this change, whenever we want to execute a command, instead of saying `system($cmd)` we can say `print $cmd`, because there's a shell waiting to execute it. Note that the statement `$| = 1` turns off buffering to make the shell get all our writes immediately.

All the triggered commands share the same shell, which runs for as long as `atchange` is running. Whenever we set an environment variable or define a function, those variables of functions are available from then on in every action triggered by a subsequent file change.

Next, we permit multiple lines per action by making `perl` read its input file in paragraph mode, as in the following example:

```
$/ = "";            # paragraph mode
while(<PGM>) {          # first read the program
    s/#.*%n/%n/g;
    ($file, $cmd) = /(%S*)%s+([^%000]+)/;
```

This reads a paragraph at a time, taking the first word to be the filename and the rest of the paragraph to be the associated command.

For convenience, we add one relatively simple rule: Any paragraph that lacks a file name (i.e., begins with white space) is executed directly.

```
unless ($file) { print $cmd; next; }
```

Looking back at our example, you'll see that this is how we define functions and set variables unconditionally. Functions, variables, control flow. We now have a little programming language. Thus, an input file for `atchange` is a single program.

We began by trying to rewrite a pair of shell scripts for a friend but, without much work, wound up with a programming language. We won't reproduce the final version of `atchange` here, but the whole thing is less than a page long. You can find it at `http://www.qms.com`.

Are we done? Maybe. We can rewrite `biff` as a trivial `atchange` script, but we can't yet write `tail -f`, which seems like a reasonable application for a program that

watches for file changes.

What ways might we want to extend what we have?

• **Other Triggers**–Changes in file modification times currently trigger atchange's actions. Perhaps we could use the access time or the inode change time instead. For example, if we used the access time, the program, `/etc/date date`, and an empty file `/etc/date` would let us do the following:

```
$cat/etc/date
Sun Jan  7 22:53:00 MST 1996

Sun Jan  7 22:53:31 MST 1996
```

Another easy extension would be to tie an action to a group of files instead of just a single file. Even more interesting might be to use a change in the file contents or even to look at things other than files. For example, changes in variable values or program output.

A good example of this is Greg Rose's `watch.curseperl`, which takes advantage of curses' ability to incrementally update screens. Invoking it as `watch.curseperl date`, it will run `date`, display the result and then update the display as the date changes, changing only the parts that have changed since its last update.

Another tool that allows this is Glenn Fowler's `nmake`, which extends `make` by allowing you to specify dependencies on things like the compilation flags.

• **Timing**–At the moment, atchange spends most of its time in a busy wait. We talked about improving performance by shortening the sleep time, but it would be nice if we made the program interrupt-driven. Doing this almost certainly requires a modification to the operating system to let user-level processes detect file changes when the filesystem sees them.

Brians Bershad and Pinkerton have done some work in this area, which they call "watchdogs." Their sample applications are mostly security-related (see "Watchdogs–Extending the UNIX Filesystem," *Computing Systems*, Vol. 1, No. 2, Spring 1988, Page 169).

Going in the other direction, we might be able to extend `atchange` to do a similar job to `make` but in reverse. Instead of specifying how to create files when they're out of date with respect to the things that go into creating them, we could specify what to do with files when they're out of date with respect to their immediate products. In addition, instead of monitoring files continuously, we could examine them at invocation of `atchange` and have `atchange` exit after a single pass.

• **Syntax**–OK, so the syntax isn't that great. Even if the shell is your favorite programming language–Haemer says that it's his–it seems a little artificial to prohibit using blank lines to help break up actions or require that you indent function definitions and variable assignments.

We're sure that Tom would appreciate a better syntax too. Of course, the best extensions are ones we haven't thought of yet. We'd love to see your ideas. Please email them to us at `jsh@canary.com` and `copeland@alumni.caltech.edu`, or to Tom Schneider at `toms@ncifcrf.gov`. While you're at it, we encourage you to visit Tom Schneider's home page at `http://www-lmmb.ncifcrf.gov/~toms/index.html`. We just write software and columns. He's curing cancer. ▲