The `bak` backup utility: design and testing

Andrew Hamilton-Wright

Thu Jan 29, 2009

The script `bak` is intended to be a simple copy-based backup utility for temporarily managing duplicate copies of files. The simplest method of backing up a file is to make a copy in a "backup" directory, and then use a versioning scheme to ensure that there are a few copies of the file around for historical analysis.

The `bak` command will place all files indicated into a local directory called "backup", while ensuring a rotating order of versions of a similarly-named files.

Options to the script allow the use of several variations on the filename extension and rotation:

Option	Meaning
`-d`	use the date in the extension
`-t[24]`	use the time in the extension (12 hour format default, 24-hour format optional)
`-T<tag>`	allow an arbitrary tag to be used to identify version
`-v`	verbose mode
`-z[<filter>[,<ext>]]`	this flag, with optional filter will compress the files

Design:

The goals of the design are: portability, simplicity and understandability.

The script is written as a Bourne shell script. It should run in any version of the Bourne or bash shells, as it makes few (if any) assumptions about extended features that some older shells may not have.

Commands that the `bak` script depend on are all part of the standard POSIX.2 environment (IEEE Std 1003.2), and are as follows: `cp, date, expr, mkdir, rm, sed, test`

Additionally, the script is written to expect to use compression based on the `gzip` command, commonly available on UNIX-alike platforms for at least 10 years. Note that if `gzip` is unavailable, any `gzip`-like compression filter can be substituted from the command line using the `-z` option, or via the environment variables `BAKCOMP` and `BAKCOMPEXT` for command and preferred extension respectively.

Structure:

The overall control structure of the script is as follows: * initial setup of expected values occurs at the top of the script; * parsing of flag options is handled next; * finally, the command line is re-parsed to handle filename type arguments.

This double-pass of the command-line is performed to allow all flags to be applied to all files regardless of whether the flags appear to the right or the left of the filenames. All filenames passed to the `bak` script in a single call will therefore be treated by the same storage convention.

The overall strategy of the main loop involves the construction of a variable `dostr` which represents "the thing I am going to try to do". It is used to test for name conflicts, and ensure that new versions do not overwrite old.

Major sections of the source code will be identified in the annotation that follows.

Top of file – this is the script header and version information

```
 1  #! /bin/sh
 2
 3  #       Andrew Hamilton-Wright : original version Summer 94
 4  #
 5  #   backup tool -- puts files in the local "backup" dir, creating
 6  #   this dir if it has to
 7  #
 8  #   v1.1    revision numbers -- will add a revision number for
 9  #           multiple instances of the same file per day
10  #   v1.2    compression -- -z option will compress files
11  #   v1.3    timestamp -- use time as well as date
12  #   v1.4    tag -- use tag as well as date
13  #   v1.5    ensure that backup subdir exists if backed up file is
14  #           in a relative directory
15
16
17
```

This portion of the file sets up default values prior to parsing command line arguments so that at argument parsing time these defaults can be overwritten.

```
18  #   defines -- change these at will
19  #   directory name to use for backups
20  DIRNAME=backup
21
22  #   maximum number of file revisions/day (mostly a sanity check)
23  MAXVERSION=19
24
```

Note here the use of the "default" option on variable assignment, so that the local variable `COMPRESSUTIL` is assigned to the value of `${BACKCOMP}` if it is defined as an inherited environment variable from the parent environment; otherwise `COMPRESSUTIL` is set to be the filter `gzip -9`.

```
25  #   compression util to use:  Note that this _must_ direct output
26  #   to stdout, as a pipe is used in this code to collect output
27  COMPRESSUTIL="${BACKCOMP:-gzip -9}"
28
29  #   extension desired by compressutil, above
30  COMPRESSEXT="${BAKCOMP:-.z}"
31
```

All the possible date formats are set up here – this is relatively cheap to do, but could certainly be done below.

```
32  #   create extension to add
33  DATEEXT=`date +".%b%d.%y"`
34  TIMEEXT=`date +".%I%M%p"`
35  TIMEEXT24=`date +".%H%M"`
36
37
```

Ensure that there is a blank space after the command so that our messages below are easily visible.

```
38  echo " "
39
40
```

The values that will get changed are initialized here. The comment is intended to ward off a would-be upgrader from breaking a working variable when they think that they are setting up default values. Also in this section is the default value of Vecho, which will simply take its argument and silently do nothing with it – this is done so that in verbose mode (below) we can substitute an actual `echo` statement.

```
41  #   runtime variables : do not modify
42  compress=0
43  compressext=""
44  Vecho="test -z "
45  fileext=""
46
```

State machine to ensure we have several ways to make the help appear.

```
47  # print help and exit?
48  printhelp="NO"
49
50
51
52  if [ $# -lt 1 ]
53  then
54      printhelp="YES"
55  fi
56
57
58
```

Sanity checks to be sure that there is a directory, and that it will work for our purposes.

```
59  #   Check if directory for bakups exists, and create one
60  #   if it does not
61
62  if [ -d $DIRNAME ]
63  then
64      :
65  else
66      if [ -f $DIRNAME ]
67      then
68          echo "ERROR:  file '$DIRNAME' exists - cannot create backup directory" >&2
69          echo " " >&2
70          exit 1
71      fi
72
73      echo " "
74      echo "Creating backup directory $DIRNAME"
75      if
76          mkdir $DIRNAME
77      then
78          :
```

```
79      else
80          echo "WARNING:" >&2
81          echo "Could not create dir $DIRNAME" >&2
82          echo "Backup Aborted" >&2
83          echo " " >&2
84          exit 1;
85      fi
86  fi
87
88
89
90
```

First pass through the options begins here. This section overwrites the default values in UPPERCASE variable names set up above. Extentions will pile on in the order in which they are specified here.

```
91  #   strip out options
92  for arg in "$@"
93  do
94      case $arg in
95      -z*)
96          # default values already set up above, so determine
97          # if they are being overridden here
98          compress=1
99          compressext=$COMPRESSEXT
100         if [ X"${arg}" != X"-z" ]
101         then
102             compressutil=`echo ${arg} | sed -e 's/,.*//' -e 's/^-z//'`
103             if [ `expr ${arg} : ".*,.*"` -gt 0 ]
104             then
105                 compressext=`echo ${arg} | sed -e 's/.*,//'`
106             fi
107         fi
108         ;;
109     -v*)
110         Vecho="echo"
111         ;;
112     -t24*)
113         $Vecho "Timestamp mode : time will be appended to filename"
114         $Vecho " "
115         fileext="${fileext}$TIMEEXT24"
116         ;;
117     -t*)
118         $Vecho "Timestamp mode : time will be appended to filename"
```

```
119         $Vecho " "
120         fileext="${fileext}$TIMEEXT"
121         ;;
122     -d*)
123         $Vecho "Timestamp mode : time will be appended to filename"
124         $Vecho " "
125         fileext="${fileext}$DATEEXT"
126         ;;
127     -n*)
128         $Vecho "Noext mode : only version will be appended to filename"
129         $Vecho " "
130         fileext=""
131         ;;
132     -T*)
133         fileext=`echo ${arg} | sed -e 's/-T/./'`
134         $Vecho "Tag mode :  tag $fileext will be appended to filename"
135         $Vecho " "
136         ;;
137     -[hH?]*)
138         printhelp="YES"
139         ;;
140     -*)
141         echo "Unknown option $arg"
142         ;;
143     *)
144         :
145         ##  filename - ignore on this pass  ##
146         ;;
147     esac
148 done
149
150
```

Now we determine whether things are bad enough to warrant printing help – or if we have been requested to. For this program, I decided that exitting with success when help appears is a good thing, and I also decided that help should appear on standard output (not error), in case someone wanted to put it into a pipe.

```
151 if [ X"${printhelp}" = X"YES" ]
152 then
153     echo "Use:  bak <file> . . ."
154     echo " "
155     echo "Backs up files <file> to the local directory '$DIRNAME'."
156     echo " "
```

```
157     echo "bak will create the directory if needed, and if there are other copies with"
158     echo "the same name in the directory, will begin version numbering files with the"
159     echo "further extension '.v<version_no>'"
160     echo " "
161     echo "As a sanity check, the maximum version is `expr $MAXVERSION + 1`."
162     echo " "
163     echo "Options:"
164     echo " "
165     echo " -z  : For space savings, this option will compress the files using the utilit
166     echo "     '$COMPRESSUTIL', and appending the extension '$COMPRESSEXT' to all the fi
167     echo " -v  : verbose mode -- prints out all tests done"
168     echo " -d  : use date, extension will be ($DATEEXT)"
169     echo " -t24: use time stamp, extension will be ($TIMEEXT)"
170     echo " -t  : use time stamp, extension will be ($TIMEEXT)"
171     echo " -T  : use tag as well as date, extension will be specified tag"
172     echo " -n  : add no extension -- only extension will be version #"
173     echo " "
174     exit 0
175 fi
176
177
178
```

Now we rework the options, ignoring all flags and handling each file. This could be put into a function, but there are some very old versions of sh that do not support such things, so I simply put the whole works here in the case statement.

An alternative would be to build up a list of filenames, and then work through the list directly, but the only difference here is that there is the extra layer of the "case", so this seemed cleaner, in this particular instance.

```
179 #           now do work     MAIN LOOP
180
181 for filearg in $@
182 do
183     case $filearg in
184     -*)
185         :
186         ##  do nothing with options this time around    ##
187         ;;
188     *)
189
```

This checks for the easy case first, as this is probably what any programmer reading this would expect. If we cannot do it the easy way, we check for progressively harder cases.

```
190         ##  if file does not exist, then things are cool    ##
191
192         if [ ! -r $DIRNAME/$filearg$fileext -a \
193                     ! -r $DIRNAME/$filearg$fileext$COMPRESSEXT ]
194         then
195             dostr=$DIRNAME/$filearg$fileext
196         else
197
198             ## if things are not cool, try for a valid version ##
199
200             echo "bak: found $DIRNAME/$filearg$fileext$compressext" >&2
201             dostr=""
202             attempt="1"
203
```

This while loop will construct various versions of the "to do" string in `trystr`, abandoning each if they do not pan out, and placing them in `dostr` if they pass muster. This keeps the while loop exit strategy nice and simple by saying, in effect, "if dostr hasn't got something valid to do in it yet, keep trying".

```
204             while
205                 [ -z "$dostr" ]
206             do
207                 trystr="$DIRNAME/$filearg$fileext.v$attempt"
208
209                 $Vecho "##   trying $trystr"
210                 $Vecho "##      and $trystr$COMPRESSEXT"
211
212
213                 if [ ! -r $trystr -a ! -r $trystr$COMPRESSEXT ]
214                 then
215
216                     $Vecho "##   $trystr$compressext is valid"
217
218                     ##  string is ok    ##
219                     dostr=$trystr
220
221                 else
222
223                     $Vecho "##   found $trystr$compressext"
224                     if
225                     [ `expr $attempt \> $MAXVERSION ` -eq 1 ]
226                     then
227
```

Error messages (in this case that too many trials occurred) are printed on

standard output.

```
228                         ##  too many tries -- abort ##
229
230                         echo "WARNING:" >&2
231                         echo "  File $DIRNAME/$filearg$fileext already exists!" >&2
232                         echo "  Too many (`expr $MAXVERSION + 1`) versions" >&2
233                         echo "  Backup Aborted" >&2
234                         echo " " >&2
235                         exit 1
236                     fi
237                 fi
238
239                 ##  increment attempt   ##
240                 attempt=`expr $attempt + 1`
241
242             done
243
244         fi
245
246
```

If we get here, `trystr` has become `dostr`, and we are ready to go.

First task is to handle what we should do if we are backing up a file that itself has a path specification between us and the target – here I have decided to represent this path within the backup directory too, so that the difference between backing up `A/foo.txt` and `B/foo.txt` from the same directory preserves the relation, and doesn't simply call one "version 1" and the other "version 2".

```
247         ##  ok - we now have a valid name to compress to    ##
248
249         dirstr=`dirname $dostr`
250         if [ ! -d $dirstr ]
251         then
252             echo "Creating directory $dirstr" >&2
253             if
254                 mkdir -p $dirstr
255             then
256                 :
257             else
258                 echo "WARNING:" >&2
259                 echo "Could not create directory $dirstr" >&2
260                 echo "Backup Aborted" >&2
261                 echo " " >&2
262                 exit 1
```

```
263             fi
264         fi
```

Now that the directory structure is in place, we are ready to copy the file.

```
265         if
266             cp $filearg $dostr
267         then
268             echo "$filearg backed up to $dostr" >&2
269             :
270         else
271             echo "WARNING:" >&2
272             echo "Could not copy $filearg to $dostr" >&2
273             echo "Backup Aborted" >&2
274             echo " " >&2
275             exit 1
276         fi
```

Compression here is an add-on, after the file is copied, so once the compression has happened, we delete the original. In order to make sure that the pipe for compression won't end up eating its own tail, we specifically test that the "before compression" name is not the same as the "after compression" name.

```
277         if [ $compress -eq 1 ]
278         then
279             echo "Compressing to $dostr$compressext" >&2
280             if [ $dostr = "$dostr$compressext" ]
281             then
282                 echo \
283 "NO COMPRESSION: source and target name identical $dostr$compressext" >&2
284             else
285
286                 $Vecho "## Compression being done using:"
287                 $Vecho \
288                     "## $COMPRESSUTIL < $dostr > $dostr$compressext"
289
290                 if
291                     $COMPRESSUTIL < $dostr > $dostr$compressext
292                 then
293                     $Vecho "## Removing uncompressed version $dostr"
294                     rm $dostr
295                 fi
296             fi
297         fi
298         ;;
```

```
299     esac
300 done
```

Now we are all done, so print another blank line and exit with success.

```
301
302 echo " "
303
304 exit 0
305
```

Summary:

The goals stated above are: portability, simplicity and understandability.

Portability has been achieved by using the Bourne shell environment, which is, and has been, a standard of UNIX type environments for many years, being released as part of AT&T Version 7 UNIX. External commands assumed to exist have been part of the UNIX install for many years, and have been part of the POSIX standard since version 2 (1993).

Simplicity and understandability go hand in hand. Within the constraints imposed by the use of `sh(1)`, the code has been organized into sections based on function, however no actual shell "functions" have been used, which is clearly a limitaton of this design.

Thought has gone into the error messages included and the internal structure for exceptional cases, in order that the problematic situations that the program is likely to encounter are recognized and reported, rather than introduce a fault that may be difficult to interpret.

This should then provide a rugged and useful utility for temporary file record keeping. For even more robustness and history, the reader is directed to revision management software, such as rcs(1), cvs(1), svn(1) and now git(1). The rcs(1) command is a long standby of UNIX environments, however the additional functionality provided by the newer three mean that they have subsumed the function of rcs(1) almost entirely.

Testing:

The test cases listed below have been verified for the `bak` program.

tests for basic copy behaviour

single file present, no `backup` directory present

- reasoning: standard functionality – backing up a single file
- command: `bak testfile.txt`
- expected outcome: file copied to newly created `backup` directory
- observed outcome: success

multiple files present, no `backup` directory present

- reasoning: standard functionality – backing up a set of files
- command: `bak *.txt`
- expected outcome: file copied to newly created `backup` directory
- observed outcome: success

single file not present, no `backup` directory present

- reasoning: standard functionality – incorrect filename specified
- command: `bak wrongfile.txt`
- expected outcome: error message, nothing performed
- observed outcome: failure – `backup` directory created, then error message

multiple files not present, no `backup` directory present

- reasoning: standard functionality – incorrect filename specified
- command: `bak wrongfile1.txt wrongfile2.txt wrongfile3.txt`
- expected outcome: error message, nothing performed
- observed outcome: failure – `backup` directory created, then error message

multiple files some present, some not, no `backup` directory present

- reasoning: standard functionality – some incorrect filename specified
- command: `bak testfile.txt wrongfile.txt testfile2.txt wrongfile2.txt`
- expected outcome: error message, nothing performed
- observed outcome: failure – `backup` directory created, testfile.txt backed up, then error message, second present file not backed up
- notes: this is a case that will confuse users quite significantly

all of above test cases, backup directory present, contains file with name not used in test

- reasoning: standard functionality, after at least one backup performed
- each test case above performed again
- expected outcome: **as per each expected outcome above**
- observed outcome: **as per each expected outcome above**

tests for options

tests for file versioning

single file, multiple backups

- reasoning: rotation of files is a standard function
- command set: run `bak newtestfile` seven times
- expected outcome: on first run, file is backed up simply, then a ",v#" option is appended to each subsequent name
- observed outome: success

single file, excessive multiple backups

- reasoning: rotation of files is a standard function
- command set: run `bak newtestfile` 25 times
- expected outcome: after `MAXVERSIONS` (19) backups, an error message is produced, and no backup performed
- observed outome: failure – error occurs after `MAXVERIONS+1` backups
- note: (likely fencepost error)

-z option

single file present, `-z` option used

- reasoning: standard functionality – backing up a single file
- command: `bak -z testfile.txt`
- expected outcome: compressed file copied `backup` directory
- observed outcome: success

multiple files present, `-z` option used

- reasoning: standard functionality – backing up a single file

- command: `bak -z testfile.txt`
- expected outcome: compressed file copied `backup` directory
- observed outcome: success

Notes:

- No tests were done using condition where `backup` directory needed creation, as this is assumed to be fully tested above