SANDEEP SAHORE

reclaim disk space by shrinking files

Sandeep Sahore holds a Master's degree in computer science from the University of Toledo and has nearly 15 years of experience in the computing industry. He specializes in low-level and C programming, systems engineering, and system performance.

ssahore@yahoo.com

The source for cfsize.c may be found at http://www.usenix.org/publications/login/2008-10/cfsize.c.

SYSADMINS FROM TIME TO TIME ARE faced with the problem of reclaiming disk space, a problem that lurks in the shadows waiting to buzz the pager into life. The typical response is either to remove files or to compress them, or to invoke some combination of the two approaches. But there are many situations where the choice is not so cut-and-dried. Let's say there is a file filling up a storage partition that cannot be removed because its data should be kept for a rolling window of one year or because its contents are sensitive or because it is being held open by a process. In such cases it is better to shrink the file in size instead of removing or compressing it.

Having faced such scenarios a countless number of times I decided to write a program that would shrink files in size from the beginning or "head." This program is called cfsize, which is short for "change file size"; it is described in detail in the sections that follow. cfsize is written in the C language for the UNIX platform, but it works equally well on Linux.

The Need for Cfsize

So what is the need for designing and developing cfsize, when a standardized utility, csplit, already exists? Though cfsize and csplit look similar they are poles apart functionally.

Cfsize was ostensibly designed to change a file in size by deleting part of its contents from the "head." By shrinking the input file in place, it reclaims space from a file system that is at its threshold limit. Behind the scenes, cfsize siphons off the data that needs to be kept into a temporary file and when finished it replaces the input file with the temporary one. This retains the latest data in the file while the oldest data is thrown away.

In contrast, csplit makes a copy of the input file and splits the copy into smaller parts based on user-specified criteria. The smaller files thus created can be concatenated to reproduce the original file, since csplit follows "the whole is the sum of the parts" method. It does not alter or shrink the original file nor does it reclaim space. In fact, by creating a copy of the original file, it uses more space.

From the preceding discussion it should be clear that cfsize and csplit are two very different tools. Cfsize reclaims disk space by shrinking files in size, whereas csplit is mostly an intermediate step of a multi-step process.

Among its lesser-known rivals is the trunc.pl utility written in Perl. A keyword search for trunc.pl on the Internet pulls up its Web site. It is similar in functionality to cfsize but uses external UNIX utilities such as tail and system instead of Perl built-ins, thereby incurring overhead owing to the repeated invocation of the `fork()` and `exec()` system calls, not to mention from their small internal buffers. It also takes as its argument a discrete number of lines instead of the new file size, which makes the calculations for reclaiming disk space cumbersome. The abstraction provided by trunc.pl is offset by the performance penalty incurred for shrinking files, especially large ones.

Compilation and Execution

After obtaining the source code, assemble the cfsize executable using an ANSI C compiler (either cc or gcc) as:

```
# cc cfsize.c -o cfsize
```

Store the executable in /usr/local/bin or a directory of your choice and invoke it without any arguments to obtain usage:

```
# cfsize
usage: cfsize -s filesize[k|m|g] file ...
```

Cfsize takes the following options, along with a list of files that need to be reduced in size:

```
-s filesize[k|m|g]
```

That is, the new size of the file is in bytes, kilobytes, megabytes, or gigabytes, with bytes being the default, as specified, respectively, with no suffix or with the k, m, or g suffix as shown.

Program Flow and Design

Conceptually the whole cfsize program can be divided neatly into three distinct parts:

- Parse and process the options given on the command line.
- Open and read the file(s) supplied on the command line.
- Shrink the listed file(s) to the specified size.

Before diving into an in-depth explanation of its parts, let's go over the mode of operation supported by cfsize. As already stated, cfsize chops off the "head" of the file, implying that the latest entries in the file are kept while the oldest ones are thrown away. Another way to look at this is to think of the file being *rolled* from the top down. When the desired size is reached, the *rolled-up* portion is virtually torn off.

PARSE THE COMMAND-LINE OPTIONS

The cfsize utility takes a single mandatory command-line option `-s`, which takes the new size of the file as its argument. The input file is "chopped off" from the "head" and the program checks whether the new size of the file has been passed to `-s` followed by enabling a flag and invoking the `getfsz()` routine to calculate the desired size of the file. The flag is checked to see whether the mandatory `-s` switch has been provided on the command line. If any of these checks evaluates to false the program errors out:

```
long getfsz(int s, char *sarg)
{
  int c;
   long n = 0;

  while (c = tolower(*sarg)) {
   switch (c)
   {
     case '0': case '1': case '2': case '3': case '4':
     case '5': case '6': case '7': case '8': case '9':
       n = 10 * n + (c - '0');
       break;
     case 'k': case 'm': case 'g':
       if (*(sarg-1) >= '0' && *(sarg-1) <= '9' && !*(sarg+1)) {
        if (c == 'k')
           kb++;
        else if (c == 'm')
           mb++;
        else if (c == 'g')
           gb++;
       }
       else
        fprintf(stderr, "%s: invalid argument to option --
           %c\n", prog, s), usage(prog);
       break;
     default:
       fprintf(stderr, "%s: invalid argument to option -- %c\n",
                 prog, s), usage(prog);

   }
  ++sarg;
  }
  return n;
}
```

FIGURE 1

The `getfsz()` function listed in Figure 1 ensures that the argument to the `-s` option is a valid number. It scans the `filesize` argument string one character at a time, converting it into an integer while checking for the presence of characters that are not numerical. It also figures out whether the file size reduction is specified in kilobytes, megabytes, or gigabytes. It terminates abnormally if the argument string contains any characters outside the acceptable range.

OPEN LISTED FILES AND SHRINK TO SPECIFIED SIZE

After processing the options, cfsize moves on to reading the files listed on the command line. A `while` loop is used to open, read, and "chop off" the files from the "head." It has a built-in safety net to terminate execution of cfsize if a user mistakenly enters a file size that is greater than the current size. The size of the file being processed currently is obtained by calling the `stat()` library function with the filename as its argument. If the new file size is more than the current file size, cfsize terminates abnormally. This safety net prevents the file from being "inflated" instead of being "shrunk." An error is raised if a file cannot be opened, and the program moves on to the next file in the list until the list of files is exhausted:

```
void fsplit(long fsz, char *fnam, FILE *fin)
{
   int c;
   FILE *fout;
   long neg = -fsz;

   /* end abnormally if the temp file cannot be created */
   if (!(fout = tmpfile()))
      catcherr("tmpfile()");

   /* set file pointer to "neg" bytes from end of current file */
   if (fseek(fin, neg, SEEK_END))
      catcherr("fseek()");

   /* go to end of the line to avoid inline file breakage */
   while ((c = getc(fin)) != '\n')
      ;

   /* move the data that needs to be retained to the temp file */
   while ((c = getc(fin)) != EOF)
      putc(c, fout);

   /* truncate and prepare the current file for writing */
   if (!(fin = fopen(fnam, "w")))
      catcherr("Cannot open %s", fnam);

   /* set file pointer to the beginning of the temp file */
   if (fseek(fout, 0L, SEEK_SET))
      catcherr("fseek()");

   /* move the contents of the temp file to the current file */
   while ((c = getc(fout)) != EOF)
      putc(c, fin);

   /* flush buffered writes to the current file by closing it */
   if (fclose(fin))
      catcherr("Cannot close %s", fnam);
}
```

FIGURE 2

Figure 2 shows `fsplit()`, the function that is at the heart of cfsize and which is responsible for shrinking files. It starts by creating a temporary file, using the `tmpfile()` function, for storing the data that needs to be retained. Next it moves the file offset backward from the end of the file, stopping after exactly `filesize` bytes. The file offset is then advanced to the end of the line it currently rests in order to prevent in-line file breakage. This implies that the new size may be the same or less than the file size specified on the command line. With the file offset poised at the beginning of the line, the data to be retained is moved to a temporary file. After the data migration is complete, the temporary file replaces the input file. If any one of the system calls or library functions invoked by `fsplit()` fails, the program ends abnormally.

Examples of Usage

A good way to get familiar with any tool quickly is to understand how it is used in common scenarios, and that's the focus of this section. For example, the command to "chop off" a file ~25 MB in size from the "head" down to 10 kB would be:

```
# cfsize -s 10240 file.txt
```

Alternatively, one can use the `k` (kilobytes) suffix for the file size instead of bytes:

```
# cfsize -s 10k file.txt
```

Not just one but many files can be specified on the command line as long as you do not exceed the maximum allowable number of command-line arguments for the shell. The following command reduces all logfiles in the current directory to 2 MB:

```
# cfsize -s 2m *.log
```

The cfsize utility works on files only. Standard input (`STDIN`) has no meaning to cfsize and commands like the following should not be used because the program abends:

```
# cat file.txt | cfsize -s 10k
# cfsize -s 10k < file.txt
```

Standard output (`STDOUT`) also has no meaning to cfsize, since the input files provided on the command line are modified *in situ*. Here's an example of what not to do:

```
# cfsize -s 50k input.txt > output.txt
```

This command would reduce input.txt from 250 MB to 50 kB and create a zero-length output.txt file, which is not what is intended.

Let's wrap up this section by going over the application of the "end-of-options" switch. The command to reduce `-file` from 2 GB to 1 MB would be:

```
# cfsize -s 1m -file
cfsize: illegal option -- f
usage: cfsize -s filesize[k|m|g] file ...
```

However, cfsize thinks that it is being passed option `-f` and it terminates abnormally, as it recognizes `-s` as the only valid option. This ambiguity is the reason why the end-of-options switch `--` has been built into cfsize. To correct this pernicious situation, insert the end-of-options switch into the command line right before the processing of any files:

```
# cfsize -s 1m -- -file
```

Bugs and Shortcomings

Chopping a file from the "head" needs one important caveat, because of the way file truncation works. The data that needs to be kept is moved to a temporary file and when the desired file size is reached the original file is replaced with the temporary one. This means that the target directory, that is, the one containing the temporary file, should have enough space to hold the intermediate data; otherwise the whole operation will fail. Note that permissions on the target directory come into play when cfsize is executed without superuser privileges. Another point to keep in mind about cfsize is the fact that it does not support large files, that is, files that are greater than 2 GB. If it were used on a file bigger than 2 GB, cfsize would terminate.

Cfsize has been tried and tested on many UNIX and Linux platforms. It is designed almost exclusively with sysadmins in mind, so while using cfsize, if anyone comes across a bug or feels that redesigning the algorithm, implementing coding shortcuts, or efficiently using system resources can improve the program, please contact me via email. Please do the same if any one of you comes across a tool, besides those mentioned here, that can rival the claims of cfsize.

Conclusions

Cfsize was designed for doing a simple task, that is, reducing the sizes of the files given on the command line. It provides a much-needed respite from storage space woes to sysadmins who up to now had to either compress files or remove them. A third option in the form of a C program named cfsize is now readily available to all system admins. Future plans for cfsize may include revising it to support large files.