In this chapter:

- *Allowing Nondecimal Input Data*
- *Two-Way Communications with Another Process*
- *Using gawk for Network Programming*
- *Using gawk with BSD Portals*
- *Profiling Your awk Programs*

10

Advanced Features of gawk

This chapter discusses advanced features in *gawk*. It's a bit of a "grab bag" of items that are otherwise unrelated to each other. First, a command-line option allows *gawk* to recognize nondecimal numbers in input data, not just in *awk* programs. Next, two-way I/O, discussed briefly in earlier parts of this book, is described in full detail, along with the basics of TCP/IP networking and BSD portal files. Finally, *gawk* can *profile* an *awk* program, making it possible to tune it for performance.

The section "Adding New Built-in Functions to gawk" in Appendix C, *Implementation Notes*, discusses the ability to dynamically add new built-in functions to *gawk*. As this feature is still immature and likely to change, its description is relegated to an appendix.

Allowing Nondecimal Input Data

If you run *gawk* with the *––non–decimal–data* option, you can have nondecimal constants in your input data:

```
$ echo 0123 123 0x123 |
> gawk --non-decimal-data '{ printf "%d, %d, %d\n", $1, $2, $3 }'
83, 123, 291
```

For this feature to work, write your program so that *gawk* treats your data as numeric:

```
$ echo 0123 123 0x123 | gawk '{ print $1, $2, $3 }'
0123 123 0x123
```

The `print` statement treats its expressions as strings. Although the fields can act as numbers when necessary, they are still strings, so `print` does not try to treat them numerically. You may need to add zero to a field to force it to be treated as a number. For example:

```
$ echo 0123 123 0x123 | gawk --non-decimal-data '
> { print $1, $2, $3
>   print $1 + 0, $2 + 0, $3 + 0 }'
0123 123 0x123
83 123 291
```

Because it is common to have decimal data with leading zeros, and because using it could lead to surprising results, the default is to leave this facility disabled. If you want it, you must explicitly request it.

Use of this option is not recommended. It can break old programs very badly. Instead, use the `strtonum` function to convert your data (see the section "Octal and Hexadecimal Numbers" in Chapter 5, *Expressions*). This makes your programs easier to write and easier to read, and leads to less surprising results.

Two-Way Communications with Another Process

It is often useful to be able to send data to a separate program for processing and then read the result. This can always be done with temporary files:

```
# write the data for processing
tempfile = ("/tmp/mydata." PROCINFO["pid"])
while (not done with data)
    print data | ("subprogram > " tempfile)
close("subprogram > " tempfile)

# read the results, remove tempfile when done
while ((getline newdata < tempfile) > 0)
    process newdata appropriately
close(tempfile)
system("rm " tempfile)
```

This works, but not elegantly.

Starting with Version 3.1 of *gawk*, it is possible to open a *two-way* pipe to another process. The second process is termed a *coprocess*, since it runs in parallel with *gawk*. The two-way connection is created using the new `|&` operator (borrowed from the Korn shell, *ksh*):*

```
do {
    print data |& "subprogram"
    "subprogram" |& getline results
} while (data left to process)
close("subprogram")
```

The first time an I/O operation is executed using the `|&` operator, *gawk* creates a two-way pipeline to a child process that runs the other program. Output created with `print` or `printf` is written to the program's standard input, and output from the program's standard output can be read by the *gawk* program using `getline`. As is the case with processes started by `|`, the subprogram can be any program, or pipeline of programs, that can be started by the shell.

There are some cautionary items to be aware of:

- As the code inside *gawk* currently stands, the coprocess's standard error goes to the same place that the parent *gawk*'s standard error goes. It is not possible to read the child's standard error separately.
- I/O buffering may be a problem. *gawk* automatically flushes all output down the pipe to the child process. However, if the coprocess does not flush its output, *gawk* may hang when doing a `getline` in order to read the coprocess's results. This could lead to a situation known as *deadlock*, where each process is waiting for the other one to do something.

It is possible to close just one end of the two-way pipe to a coprocess, by supplying a second argument to the `close` function of either `"to"` or `"from"` (see the section "Closing Input and Output Redirections" in Chapter 4, *Printing Output*). These strings tell *gawk* to close the end of the pipe that sends data to the process or the end that reads from it, respectively.

This is particularly necessary in order to use the system *sort* utility as part of a coprocess; *sort* must read *all* of its input data before it can produce any output. The *sort* program does not receive an end-of-file indication until *gawk* closes the write end of the pipe.

* This is very different from the same operator in the C shell, *csh*.

When you have finished writing data to the *sort* utility, you can close the `"to"` end of the pipe, and then start reading sorted data via `getline`. For example:

```
BEGIN {
    command = "LC_ALL=C sort"
    n = split("abcdefghijklmnopqrstuvwxyz", a, "")

    for (i = n; i > 0; i--)
        print a[i] |& command
    close(command, "to")

    while ((command |& getline line) > 0)
        print "got", line
    close(command)
}
```

This program writes the letters of the alphabet in reverse order, one per line, down the two-way pipe to *sort*. It then closes the write end of the pipe, so that *sort* receives an end-of-file indication. This causes *sort* to sort the data and write the sorted data back to the *gawk* program. Once all of the data has been read, *gawk* terminates the coprocess and exits.

As a side note, the assignment `LC_ALL=C` in the *sort* command ensures traditional Unix (ASCII) sorting from *sort*.

Using gawk for Network Programming

In addition to being able to open a two-way pipeline to a coprocess on the same system (see the section "Two-Way Communications with Another Process" earlier in this chapter), it is possible to make a two-way connection to another process on another system across an IP networking connection.

You can think of this as just a *very long* two-way pipeline to a coprocess. The way *gawk* decides that you want to use TCP/IP networking is by recognizing special filenames that begin with `/inet/`.

The full syntax of the special filename is */inet/protocol/local-port/remote-host/remote-port*. The components are:

protocol
: The protocol to use over IP. This must be either `tcp`, `udp`, or `raw`, for a TCP, UDP, or raw IP connection, respectively. The use of TCP is recommended for most applications.

The use of raw sockets is not currently supported in Version 3.1 of *gawk*.

local-port
: The local TCP or UDP port number to use. Use a port number of `0` when you want the system to pick a port. This is what you should do when writing a TCP or UDP client. You may also use a well-known service name, such as `smtp` or `http`, in which case *gawk* attempts to determine the predefined port number using the C `getservbyname` function.

remote-host
: The IP address or fully-qualified domain name of the Internet host to which you want to connect.

remote-port
: The TCP or UDP port number to use on the given *remote-host*. Again, use `0` if you don't care, or else a well-known service name.

Consider the following very simple example:

```
BEGIN {
    Service = "/inet/tcp/0/localhost/daytime"
    Service |& getline
    print $0
    close(Service)
}
```

This program reads the current date and time from the local system's TCP `daytime` server. It then prints the results and closes the connection.

Because this topic is extensive, the use of *gawk* for TCP/IP programming is documented separately. See Chapter 14, *Internetworking with gawk*, for a much more complete introduction and discussion, as well as extensive examples.

Using gawk with BSD Portals

Similar to the */inet* special files, if *gawk* is configured with the *--enable-portals* option (see the section "Compiling gawk for Unix" in Appendix B, *Installing gawk*), *gawk* treats files whose pathnames begin with `/p` as 4.4 BSD-style portals.

When used with the `|&` operator, *gawk* opens the file for two-way communications. The operating system's portal mechanism then manages creating the process associated with the portal and the corresponding communications with the portal's process.

Profiling Your awk Programs

Beginning with Version 3.1 of *gawk*, you may produce execution traces of your *awk* programs. This is done with a specially compiled version of *gawk*, called *pgawk* ("profiling *gawk*").

pgawk is identical in every way to *gawk*, except that when it has finished running, it creates a profile of your program in a file named *awkprof.out*. Because it is profiling, it also executes up to 45% slower than *gawk* normally does.

As shown in the following example, the *--profile* option can be used to change the name of the file where *pgawk* will write the profile:

```
$ pgawk --profile=myprog.prof -f myprog.awk data1 data2
```

In the above example, *pgawk* places the profile in *myprog.prof* instead of in *awkprof.out*.

Regular *gawk* also accepts this option. When called with just *--profile*, *gawk* "pretty prints" the program into *awkprof.out*, without any execution counts. You may supply an option to *--profile* to change the filename. Here is a sample session showing a simple *awk* program, its input data, and the results from running *pgawk*. First, the *awk* program:

```
BEGIN { print "First BEGIN rule" }

END { print "First END rule" }

/foo/ {
    print "matched /foo/, gosh"
    for (i = 1; i <= 3; i++)
        sing()
}

{
    if (/foo/)
        print "if is true"
    else
        print "else is true"
}

BEGIN { print "Second BEGIN rule" }

END { print "Second END rule" }

function sing(    dummy)
{
    print "I gotta be me!"
}
```

Following is the input data:

```
foo
bar
baz
foo
junk
```

Here is the *awkprof.out* that results from running *pgawk* on this program and data (this example also illustrates that *awk* programmers sometimes have to work late):

```
        # gawk profile, created Sun Aug 13 00:00:15 2000

        # BEGIN block(s)

        BEGIN {
1               print "First BEGIN rule"
1               print "Second BEGIN rule"
        }

        # Rule(s)

5       /foo/   { # 2
2               print "matched /foo/, gosh"
6               for (i = 1; i <= 3; i++) {
6                       sing()
                }
        }

5       {
5               if (/foo/) { # 2
2                       print "if is true"
3               } else {
3                       print "else is true"
                }
        }

        # END block(s)

        END {
1               print "First END rule"
1               print "Second END rule"
        }

        # Functions, listed alphabetically

6       function sing(dummy)
        {
6               print "I gotta be me!"
        }
```

This example illustrates many of the basic rules for profiling output. The rules are as follows:

- The program is printed in the order `BEGIN` rule, pattern/action rules, `END` rule and functions, listed alphabetically. Multiple `BEGIN` and `END` rules are merged together.
- Pattern-action rules have two counts. The first count, to the left of the rule, shows how many times the rule's pattern was *tested*. The second count, to the right of the rule's opening left brace in a comment, shows how many times the rule's action was *executed*. The difference between the two indicates how many times the rule's pattern evaluated to false.
- Similarly, the count for an `if-else` statement shows how many times the condition was tested. To the right of the opening left brace for the `if`'s body is a count showing how many times the condition was true. The count for the `else` indicates how many times the test failed.
- The count for a loop header (such as `for` or `while`) shows how many times the loop test was executed. (Because of this, you can't just look at the count on the first statement in a rule to determine how many times the rule was executed. If the first statement is a loop, the count is misleading.)
- For user-defined functions, the count next to the `function` keyword indicates how many times the function was called. The counts next to the statements in the body show how many times those statements were executed.
- The layout uses "K&R" style with tabs. Braces are used everywhere, even when the body of an `if`, `else`, or loop is only a single statement.
- Parentheses are used only where needed, as indicated by the structure of the program and the precedence rules. For example, `(3 + 5) * 4` means add three plus five, then multiply the total by four. However, `3 + 5 * 4` has no parentheses, and means `3 + (5 * 4)`.
- All string concatenations are parenthesized too. (This could be made a bit smarter.)
- Parentheses are used around the arguments to `print` and `printf` only when the `print` or `printf` statement is followed by a redirection. Similarly, if the target of a redirection isn't a scalar, it gets parenthesized.
- *pgawk* supplies leading comments in front of the `BEGIN` and `END` rules, the pattern/action rules, and the functions.

The profiled version of your program may not look exactly like what you typed when you wrote it. This is because *pgawk* creates the profiled version by "pretty printing" its internal representation of the program. The advantage to this is that

pgawk can produce a standard representation. The disadvantage is that all source-code comments are lost, as are the distinctions among multiple `BEGIN` and `END` rules. Also, things such as:

```
/foo/
```

come out as:

```
/foo/   {
    print $0
}
```

which is correct, but possibly surprising.

Besides creating profiles when a program has completed, *pgawk* can produce a profile while it is running. This is useful if your *awk* program goes into an infinite loop and you want to see what has been executed. To use this feature, run *pgawk* in the background:

```
$ pgawk -f myprog &
[1] 13992
```

The shell prints a job number and process ID number; in this case, 13992. Use the *kill* command to send the `USR1` signal to *pgawk*:

```
$ kill -USR1 13992
```

As usual, the profiled version of the program is written to *awkprof.out*, or to a different file if you use the *––profile* option.

Along with the regular profile, as shown earlier, the profile includes a trace of any active functions:

```
# Function Call Stack:

#   3. baz
#   2. bar
#   1. foo
# -- main --
```

You may send *pgawk* the `USR1` signal as many times as you like. Each time, the profile and function call trace are appended to the output profile file.

If you use the `HUP` signal instead of the `USR1` signal, *pgawk* produces the profile and the function call trace and then exits.