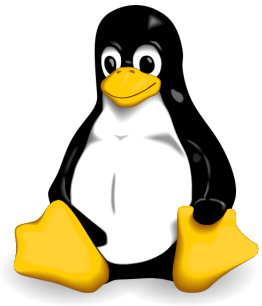
LINUX USER REFERENCE GUIDE

Professional Skills (*CS*133)



ZAK EDWARDS

Department of Computer Science
University of Warwick
December 2014

'Tux' image courtesy of L. Ewing, S. Budig and A. Gerwinski. Available at Wikimedia Commons.

CONTENTS

1 PROCESSES AND SCRIPTING IN LINUX

1.1 PROCESSES

Processes are, of course, an integral part of any Linux system; they are responsible for carrying out tasks within the operating system. Essentially, a *program* is constituted by a set of machine code instructions and data stored in an executable image on disk; A *process*, then, can be considered a computer program in action.[1]

A typical tree of processes is shown in Figure 1[2] below.

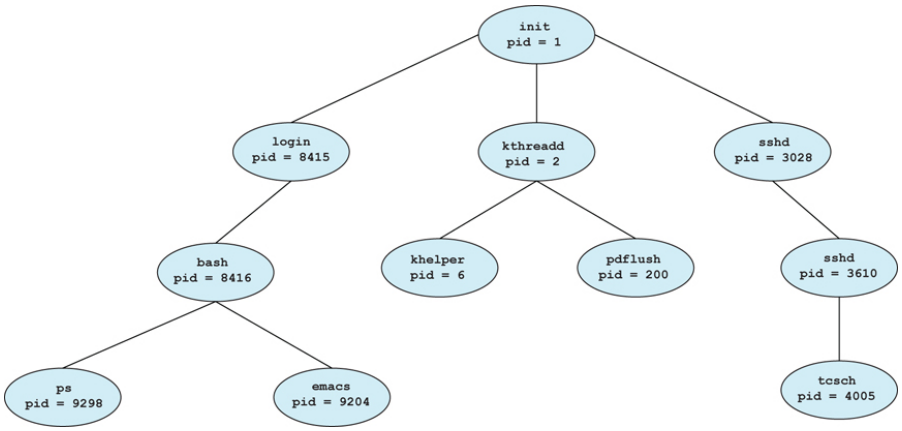


Figure 1: *A tree of processes on a typical linux system*

In this section we will be concerned with the management of these processes; that is, both the monitoring and the termination of processes on a Linux system.

1.1.1 *Monitoring Processes*

The linux user's most valuable tool for monitoring process is `ps`. In short, `ps` displays information about a user-specified selection of the system's active processes.

Concerning this specified selection, `ps` selects, by default, all processes with the same *Effective User ID* as the current user *and* associated with the same active terminal as the user responsible for invoking said terminal. Thus, simply running `ps` will print a rather small amount of information. One can expect something similar to the following:

```
$ ps
#   PID TTY          TIME CMD
# 31706 pts/5    00:00:00 zsh
# 31717 pts/5    00:00:00 ps
```

To generate a more informative output, use the following command:

```
$ ps -aux
```

Here, the `a` option lifts the *'only yourself'* restriction, and, provided the processes are associated with a terminal, list the processes associated with all users on the system. The `u` option provides a more informative output, whereas `x` lifts the restriction that listed processes must have an associated terminal.

To find the *Process ID* (PID) of a particular process, the commands `ps` and `grep` can be combined by utilising the "pipe" symbol (the pipe operator is discussed further in 2.1.3). The following command, for instance, takes the output of `ps` and uses it as input for `grep vim`:

1 This characterisation is adapted from Rusling [4]; *cf. ch.* 4.

2 Image adapted from Silberschatz, Gagne and Galvin, *'Operating System Concepts, Ninth Edition'*, (*cf.* [6], *ch.* 3); accessed at `http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html`.

```
$ ps | grep vim
```

`grep`, as is discussed further in 2.2.1, searches the specified file for the specified string. Here, the string is *'vim'*; a file, however, is not specified. Rather the output of `ps` is used as input for `grep`. Thus, the above command searches the output of `ps` for lines containing a match with the pattern *'vim'*. *(For further information concerning the redirection of standard input, see 2.1.2.)*

One must note that `ps`, used in this manner, does not necessarily list all instances of `vim`. One can execute vim as per usual and, in a seperate terminal, execute both the command vim & and the above command: it can be observed that only one such vim process is identified by grep; namely, the process running in the background of the present terminal.

Alternatively, one can use `pgrep` to perform a similar task. Running

```
$ pgrep vim
```

will produce the PIDs of all vim processes currently running, irrespective of their association with the present terminal; `pgrep`, however, lists only the PID, and provides a less complete output in comparison to the relative verbosity of the combination `ps | grep`.

1.1.2 *Terminating Processes*

There are many ways to terminate an active process on a Linux system – aside from the commonly used approach of `<Ctrl-C>` – each with their own strengths and shortcomings. In this section, we consider the most widely practised such methods.

- `pkill`

`pkill` terminates all processes that contain the specified string in their name. This can prove undesirable; running `pkill ls`, for instance, can inadvertently kill the process `alsa`.

To avoid this possibility, use the following:

```
$ pkill -x ls
```

The above, by virtue of the x option, "[...] will only match processes whose name exactly matches the pattern" (*cf.* the `pkill man` page). In our example, then, `alsa` would remain functional.

- `kill`

There remains, however, an element of inaccuracy to `pkill`, in that *all instances* of the specified process are terminated. If this is undesirable, one can use `kill`. The below code shows, by example, a preferred method of terminating precisely the process – and only the process – one wishes to terminate:

```
$ ps -a | grep vim
# 2678 pts/3 00:00:00 vim
# 2814 pts/4 00:00:00 vim
$ kill 2814
```

Where '2814' is the PID of the process one wishes to terminate.

- `killall`

`killall` is closely comparable to `pkill -x` insofar as it too terminates all instances of the specified process. It is preferable to `kill` if one does not wish to firstly identify the PID, yet as a result, it is less accurate.

```
$ killall vim
```

- `xkill`

Alternatively still, `xkill` allows for a more convenient method of process termination, as the user can select a certain process manually, *via* cursor. For example:

```
$ xkill
# Select the window whose client you wish to kill with button
    1....
# xkill: killing creator of resource 0x2200009
```

Each of these processes are effective in their own right, but their value is contingent on the situation wherein they are used. Below is a summary of each discussed method, with advantages and disadvantages for each:

Method	Advantages	Disadvantages
`pkill`	Convenient; only part of the process name needs to be specified	Imprecise; kills all *processes* containing the specified string
`kill`	Precise; only the process one wishes to terminate is terminated	Inconvenient; one must first identify the PID
`killall`	Convenient; specifying the PID is not necessary	Imprecise; kills all *instances* of the specified process
`xkill`	Convenient and precise; simple 'click' method for a single instance	Can only be used on the X Windows system

1.2 BASIC SCRIPTING

The shell is most familiar as the *'insulating layer'* between the Linux kernel and the user. It has the additional capacity, however, to function as a powerful programming language. Shell programs, or *scripts*, are composed by arranging and incorporating a variety of utilities, command line tools, system calls, compiled binaries, *etc.* In this section we will, by way of example, examine some basic elements of shell scripting.

Essentials of Scripting: `sleep` and `kill`

In this brief section we will consider the invocation of `sleep` and `kill` in shell scripting. Consider the following script:

```
1 #!/bin/bash
2 javac Loop.java
3 java Loop &
4 sleep 5
5 kill $!
```

The purpose of the `sleep` command is to delay the progression of the code for a specified period of time – here, 5 seconds. The `sleep` command takes a numerical value and a suffix as an argument; for seconds, this suffix is `s`. However, if no suffix is specified, the given numerical value is assumed to be in seconds, so the `s` would here be superfluous.

After five seconds, then, the script executes the command `kill`. One would normally, for more complex code, assign a name to each executed command after the execution of each command – for instance:

```
1 #!/bin/bash
2 javac Loop.java
3 java Loop &
4 command1 = $!
5 sleep 5
6 kill $command1
```

Where `$!` references the PID of the most recently executed asynchronous command. Thus, the above code is functionally identical to the previous excerpt of code. However, as only one significant process has been ran in the background, this would be superfluous.

Basic Mathematical Operations

The most basic mathematics operations can be performed with a shell script, by virtue of `expr`. Here, three user-input numbers are summed, and the final value is printed:

```
1 #!/bin/sh
2 sum=0
3 sum=$(expr $1 + $2 + $3)
4 echo "$sum"
5 exit 0
```

`sum=0`, of course, initialises the value of the variable `sum` to zero. The following line allows three user-input numerical values to be assigned to the variable `sum`; `expr` evaluates the values `$1`, `$2` and `$3`, respectively the first, second and third names defaultly assigned to parameters given *via* the terminal. Parentheses in this expression are crucial for assigning the output of `expr` to the variable sum, rather than printing the output; the character `$`, when prefixing a variable name, indicates the value the variable holds.

This charaterisation of "`$`", therefore, allows the following line `echo` "`$sum`" to print a *numerical value* rather than a *string*. `exit 0` sets the return value of the script to zero.

1.2.1 *`if`-statements and `for`-loops*

The downfall of the previously discussed script is its inability to account for more than three parameters. Entering greater than three parameters merely sums the first three values and ignores subsequent values. For example:

```
$ ./sum 1 2 3 4
# 6
```

Executing the script with no arguments prints a rather nondescript string:

```
$ ./sum
# +
```

This illustrates the importance of the `if`-statement in shell scripting. Here, it can be used to print an error message in the case where no arguments are input (*i.e.*, when `$# = 0`; `$#` stores the number of inputted parameters):

```
1 if [[ $# = 0 ]]
2     then
3         echo "Usage: sum <number> <number> <number>"
4     exit 1
5 fi
```

It is useful to draw a comparison with Java. To achieve a similar result in Java, one could define a counter variable of type `int` that increments for each and every user-input value, and write an `if`-statement for the condition `counter == 0`.

One should keep in mind the vital importance of whitespace in BASH. If, in the above code, one were, for instance, to type `if [[$# = 0]]`, the script would fail as "`[[`" and "`]]`" would not be recognised as *commands* surrounding an *argument*.

To allow this script to accommodate any number of variables, we construct the following `for`-loop:

```
1 for a in $*
2     do
3       sum=$(expr $sum + $a)
4     done
```

The initial line in the above code allows the code to execute for any value, without restriction; "`$*`" represents any and all positional parameters. "`sum=$(expr $sum + $a)`" straightforwardly adds the value of the user-input argument to the value of the variable `sum`, assigning this new sum to the *variable* `sum`, for each iteration.

`for`-loops in Java and in BASH are separated by the following syntactical difference:

`for(initialization; termination; increment)`	Java
`for arg in [list]`	BASH

One immediate significance of this difference concerns termination; a BASH `for` is restricted to finitude, as the `[list]` must end. For all intents and purposes, however, `for` loops in Java need not end if the termination condition (i.e., the Boolean condition) is never satisifed.

2 ADVANCED SCRIPTING AND REGULAR EXPRESSIONS

2.1 ADVANCED SCRIPTING

Here, we will discuss some further topics in shell scripting, slightly more advanced than those discussed in the previous chapter.

2.1.1 *Functions*

Similar to higher-level programming languages yet somewhat more limited in implementation, BASH has *functions*. A function is a subroutine: a block of code implementing a set of operations; a *'black box'* that performs a specified task. When one desires a repetition of code, such that a task repeats with only slight variations in procedure, functions are extremely useful.

Declaring a function in BASH can be done in the following two ways:

```
function function_name {
echo "Hello"
}
```

```
function_name () {
echo "Hello"
}
```

The following are two fairly elementary BASH scripts to perform mathematical calculations. (*Additional, more complex examples can be found in the Appendix (3.1).*)

1. ***Fibonacci Sequence.*** Observing mathematical formulae and comparing them to their implementations in BASH is an invaluable practice for furthering one's knowledge of scripting. Take, for instance, the formula for the nth Fibonacci number:

$$F_n = F_{n-1} + F_{n-2}$$

The Fibonacci formula has a fairly straightforward BASH analogue. Consider the following script:

```
function fib {
n=$1
if [ $n -le 1 ]
   then
       echo $n
   else
       l=‘fib $((n-1))‘
       r=‘fib $((n-2))‘
       echo $((l + r))
fi
}
```

The first conditional – where the user-input integer `n`, represented by the positional parameter `$1`, is less than or equal to `1` – covers the cases where `0` or `1` are input; the zeroth and first numbers of the Fibonacci sequence are `0` and `1` respectively, and thus the script only needs to echo back the input argument. The latter conditional (`else`) is a programmatic recursive implementation of the formula for the nth number in the Fibonacci sequence. This is immediately comparable to mathematical formula. The function `fib` must of course be implemented in a script, which, when ran, utilises `fib` in accordance with the user's command. We append the following `main` function:

```
function main {
echo "What fib should I calculate?"
read n
fib $n
exit 0
}
```

It is good practice for the *declaration* of a function to precede the first *call* to it. One must ensure that the `main` function is called by simply appending "`main`" to the end of the script.

2. ***Factorial.*** (Code adapted from Cooper [1], *p.* 369.) Consider also the following formula:

$$n! = \prod_{k=1}^{n} k$$

```
max_arg=5

if [ -z "$1" ]
   then
       echo "Usage: ‘basename $0‘ number"
   exit 0
fi

if [ "$1" -gt $max_arg ]
   then
       echo "Out of range (5 is maximum)."
   exit 0
fi

fact () {
local number=$1                          # Variable "number" must be
     declared as local
if [ "$number" -eq 0 ]
   then
       factorial=1                       # Factorial of 0 = 1
   else
       let "decrnum = number - 1"
       fact $decrnum                     # Recursive function call
       let "factorial = $number * $?"
fi
return $factorial
}

fact $1
echo "Factorial of $1 is $?."
exit 0
```

In a manner similar to the Fibonacci script, here we are using recursion to calculate the factorial of an inputted number.

2.1.2 *Input/Output Redirection*

Input/Output (I/O) Redirection is an important issue in scripting. There are three 'default' system files always open – `stdin`, `stdout` and `stderr` (represented in Figure 2[1]), with file descriptors `0`, `1` and `2`, respectively – and these files can be *redirected*; that is, their output can be captured and sent as input to other files, commands, applications, *etc.* This basic redirection can be performed with use of the `>` operator; for instance, the command `echo "Hello" > output.txt` will execute the `echo` command and write the results to a file called *output.txt*.

Successive redirections using "`>`" will overwrite (in this instance) `output.txt` solely with new output, erasing previously written content. However, using "`>>`" will append the new content to the file instead; for instance, executing `echo "World" >> output.txt` will allow for the following:

```
$ cat output.txt
# Hello
```

1 Image courtesy of 'ScotXW'. Available at Wikimedia Commons: https://upload.wikimedia.org/wikipedia/commons/thumb/7/70/Stdstreams-notitle.svg/300px-Stdstreams-notitle.svg.png.

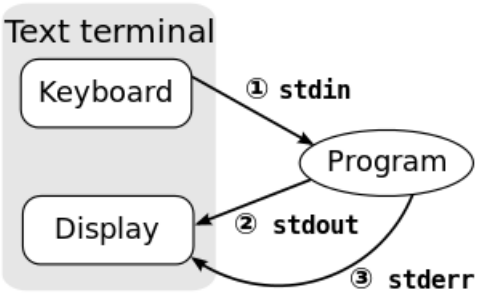


Figure 2: *Diagrammatic representation of standard input/output*

```
# World
```

Conversely, one can also redirect the *standard input* with use of the operator <. In the following example, the standard input is redirected to receive input from *file1*, rather than the keyboard device. The contents of *file1* are read into `stdin` by the redirection operation; then, `cat` reads the standard input and displays the contents of *file1*. Incidentally, this appears to be functionally identical to the command `cat file1`:

```
$ cat < file1
# Hello World
```

We can experiment with combining the redirection operations for both *standard input* and *standard output* as follows:

```
$ cat < file1 > file2
```

The `cat` command has no filename arguments; usually, `cat` receives input from the standard input and sends output to the standard output. In this instance, however, the standard input has been redirected to receive its data from *file1*, while the standard output has been redirected to write data into the empty file *file2*. Thus, no output is printed to the terminal screen, and `cat file2` yields "Hello World". A useful application of precisely this command would be to copy the contents of *file1* to *file2*, either overwriting the existing content or simply appending the data using the appropriate operator.

Operator	Action
>	Redirect standard output
>&	Redirect standard output and standard error
>>	Append standard output
>>&	Append standard output and standard error
<	Redirect standard input

2.1.3 *The Pipe Operator*

The pipe operator can be used to feed the standard output of one command to the standard input of another, as in the example:

```
$ ls -lt | head
```

The above command is particularly useful as the output of `ls -lt`, which lists the contents of a directory in descending order of modification time, is sent as input to `head` (an example of a filter), which by default lists only the first ten lines of output. Therefore, in effect, this command will list the ten most recently modified files in the current directory.

To perform precisely the opposite action that `head` performs – *i.e.,* to print only the latter ten lines of output – one can use `tail`:

```
$ ls -lt | tail
```

Thus, the ten files with the earliest modification dates in the current directory are printed.

Piping to `grep` can also prove rather useful. The following will take the output of `ls -l` and use grep to filter the data; the `-v "^d"` option will exclude those lines of data which begin with a `d`, *i.e.,* the subdirectories of the current directory:

```
$ ls -l | grep -v "^d"
```

The below command sorts the output of the file listing generated by `ls` in descending order of file size, as specified by the option `-k5` (the fifth column of `ls`'s output concerns file size). The `-n` operator specifies *numeric* sorting as opposed to the default sorting method of *alphabetic*. This is one way in which the very useful task of ordering by file size can be performed:

```
$ ls -al | sort -r -n -k5
```

Something similar can be achieved with an `awk` filter. The following command reformats the output of `ls -l` such that the file size is displayed first, perhaps preparing the data for further sorting:

```
$ ls -l | awk '{print $5, $8, $3, $6, $7}'
```

A final example uses `sed` to perform a replacement on the output of `echo`:

```
$ echo "hello" | sed 's/hello/hi/'
# hi
```

2.2 REGULAR EXPRESSIONS

A *regular expression* is a pattern composed of a sequence of characters that the 'regular expression engine' attempts to match in input text. Regular expressions prove to be particularly useful used in conjunction with `grep` or `sed`.

2.2.1 *Using `grep` and `sed`*

Perhaps the most basic example of a regular expression in practical use can be illustrated by using a `grep` filter on the output of `cat`. The following command finds all lines (in the file *./file*) that contain a number. The task is straightforwardly accomplished by specifying a range using *square brackets*:

```
$ cat ./file | grep '[0-9]'
```

`'[0-9]'`, therefore, is a basic regular expression which matches all decimal numbers. One can combine this use of square brackets with other basic regular expression operators – `^` to commence a string, `$` to end it, *etc.* – to find all words contained in */usr/share/dict/words* that end in *'ing'* and such that the first two letters are vowels:

```
$ cat /usr/share/dict/words | grep '^[aeiou]\{2\}.*ing$'
```

A `sed` filter can be used to replicate the effects of the `sum` script discussed in 1.2. The purpose of `sed` is to parse and transform text (see Figure 3[2] for a representation of this); for instance, the following `sed` filter replaces all instances (as specified by the global option, `/g`) of the string *'plus'* with the symbol '+':

```
$ echo "x is 5 plus 6" | sed 's/plus/+/g'
```

Our `sum` script can be succinctly replicated as follows, by utilising this basic principle of replacement:

```
1 #!/bin/bash
2 expr $(echo $* | sed -e 's/[0-9]*/ + &/g' -e 's/^ +//g')
3 exit 0
```

Firstly examining the code within the parentheses, one can see that the output of `echo $*` – a positional parameter storing all the arguments entered on the command line (`$1`, `$2`, *etc.*) – is sent as input to a `sed` filter. This filter has two components, each contained within apostrophes, and the `-e` option allows these components to constitute a larger argument (from `sed`'s `man` page: *'add the script to the commands to be executed'*). The former component finds all instances of numbers within the range `[0-9]` and prepends (due to the ampersand) the character '+', ultimately allowing `expr` to interpret it; the asterisk ensures that any and all numbers with more than a single digit are interpreting, by allowing the affixation of any additional numbers within the range. The latter component merely ensures that the first '+' symbol, which by the characterisation of '&' is guaranteed to exist, is removed. Thus, the `sed` filter produces a string composed of user-input numbers, separated by '+' symbols, ready to be interpreted and summed by `expr`.

2 Image adapted from Dougherty and Robbins, *'sed & awk, Second Edition'* (*cf.* [2]); accessed at `http://docstore.mik.ua/orelly/unix/sedawk/figs/sed_0401.gif`.

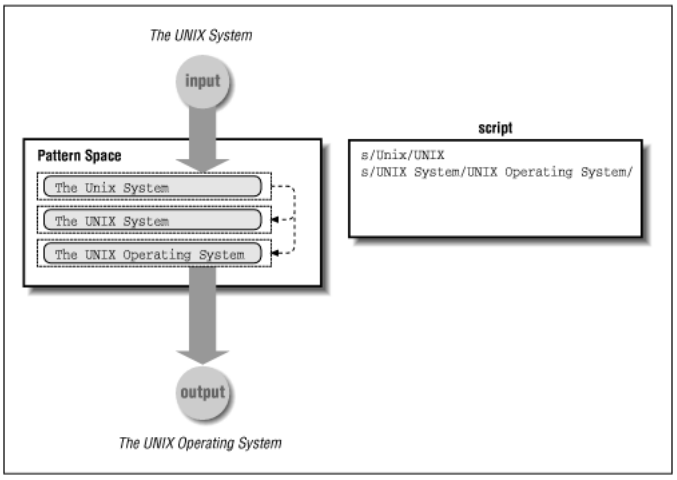


Figure 3: *A simple transformation using a* `sed` *script*

2.2.2 *Pattern matching with* `sed`

One can use a string that was matched in a pattern when making a substitution, by identifying one or more groups of characters in the match by putting escaped parentheses (`\(` and `\)`) around part of the pattern and using `\1`, `\2`, *etc.* (up to *nine* of these capture groups can be used: *cf.* `sed`'s `man` page) to designate these groups in making the substitution. The following example illustrates the most elementary usage of this kind of pattern matching:

```
$ echo 'abcabcabc' | sed 's/\(ab\)\(c\)/\1d\2/g'
# abdcabdcabdc
```

Here, the strings `ab` and `c` are seperately contained within escaped parentheses; the substitution refers to these strings respectively as `\1` and `\2`, and specifies that the character 'd' should be placed between all instances of these matches patterns.

APPENDIX 3

3.1 MORE MATHEMATICAL FORMULAE IN SHELL SCRIPTS

1. ***Quadratic Formula.*** (Courtesy of Mykhailova [3].)

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
read A;
if [ $A = 0 ]
   then
       echo "Not␣a␣quadratic␣equation.";
   exit 0;
fi
read B;
read C;
D=$(( ($B)*($B)-4*($A)*($C) ));
if [ $D = 0 ]
   then
       echo -n "x␣=␣"
       echo -e "scale=3\n-0.5*($B)/($A)" | bc
   exit 0;
fi
echo $D
if [ $D -gt 0 ]
   then
       echo -n "x1␣=␣"
       echo -e "scale=3\n0.5*(-($B)+sqrt($D))/($A)" | bc
       echo -n "x2␣=␣"
       echo -e "scale=3\n0.5*(-($B)-sqrt($D))/($A)" | bc
   else
       echo -n "x1␣=␣("
       echo -e "scale=3\n-0.5*($B)/($A)" | bc
       echo -n ",␣"
       echo -e "scale=3\n0.5*sqrt(-($D))/($A)" | bc
       echo ")"
       echo -n "x2␣=␣("
       echo -e "scale=3\n-0.5*($B)/($A)" | bc
       echo -n ",␣"
       echo -e "scale=3\n-0.5*sqrt(-($D))/($A)" | bc
       echo ")"
fi
```

2. ***Approximation of Pi.*** (Code adapted from Cooper [1], *pp.* 264–266. The *Monte Carlo* approximation of pi is discussed further in Schillaci [5].)

$$\pi \approx \frac{4N_c}{N}$$

```
dimension=10000        # Length of each side of the plot.
maxshots=1000          # Fire this many shots.
pmultiplier=4.0        # Scaling factor to approximate PI.

get_random () {
seed=$(head -n 1 /dev/urandom | od -N 1 | awk '{ print $2
    }')
rand=$seed
```

```
let "rnum␣=␣$rand␣%␣$dimension" # Range less than 10000.
echo $rnum
}

hypotenuse () {             # Calculate hypotenuse.
distance=$(bc -l << EOF
            scale = 0       # rounds down result to integer
                value
            sqrt ( $1 * $1 + $2 * $2 )
            EOF
          )
}

# Initialize variables.
shots=0
splashes=0
thuds=0
Pi=0

while [ "$shots" -lt "$maxshots" ]   # Main loop.
      do
        xCoord=$(get_random)         # Get random X and Y co
            -ords.
        yCoord=$(get_random)
        hypotenuse $xCoord $yCoord   # Hypotenuse = distance
            .
        ((shots++))
        printf "#%4d␣" $shots
        printf "Xc␣=␣%4d␣" $xCoord
        printf "Yc␣=␣%4d␣" $yCoord
        printf "Distance␣=␣%5d␣" $distance   # Distance from
             center of lake -- the "origin" -- coordinate
            (0,0).
        if [ "$distance" -le "$dimension" ]
           then
               echo -n "SPLASH!␣"
               ((splashes++))
           else
               echo -n "THUD!␣"
               ((thuds++))
        fi
        Pi=$(echo "scale=9;␣$pmultiplier*$splashes/$shots" |
             bc)   # Multiply ratio by 4.0.
        echo -n "PI␣~␣$Pi"
        echo
      done
echo
echo "After␣$shots␣shots,␣PI␣looks␣like␣approximately␣$Pi."
echo
exit 0
```

BIBLIOGRAPHY

[1] Mendel Cooper. *Advanced Bash-Scripting Guide*. Apr. 2007. Revision 4.3. Currently unpublished; accessed via `http://jamsb.austms.org.au/courses/CSC2408/semester3/resources/ldp/abs-guide.pdf` on 2015-1-06. (Cited on pages 6 and 11.)

[2] Dale Dougherty and Arnold Robbins. *sed & awk*. O'Reilly Media, second edition, 1997. (Cited on page 8.)

[3] Mariya Mykhailova. Quadratic equation in Unix shell, Jan. 2011. Accessed via `http://progopedia.com/example/quadratic-equation/275` on 2015-1-06. (Cited on page 11.)

[4] David A. Rusling. *The Linux Kernel*. New Riders Pub, 2000. (Cited on page 1.)

[5] Michael Jay Schillaci. *The Art of Scientific Computing: Problem Solving, Programming, and Presentation*, volume 1. 2007. Currently unpublished; accessed via `http://www.evsis.org/docs/tdpp2_chap1.pdf` on 2015-1-06. (Cited on page 11.)

[6] Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin. *Operating System Concepts*. Wiley, ninth edition, 2012. (Cited on page 1.)