Bash for Fun

Bash Programming: Principles and Examples

Libor Spacek

This book is for sale at http://leanpub.com/bashforfun

This version was published on 2022-01-08

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2021 - 2022 Libor Spacek

Contents

Introduction

The intended audience of this book are all those who want to gain a better understanding of their computers. Those who wish to use Bash effectively and with confidence. This book spans the full range of Bash programming skills: from beginners to advanced, with amusing anecdotal insights and examples of doing useful things.

The ignorant 'clicking of buttons' has done much disservice to laudable ambitions of programming education over the past four decades or so. It is time to rediscover the ancient lore! You, too, will become a wizard of *retro programming*. Armed with this book, you will massively increase your productivity and gain true ownership of your device.

This book is for you if in the past you got waylaid and discouraged from programming, if you never even started or if you ever thought: "surely, there must be a simpler and better way of doing this!".

Above all, what you will discover here is that one most rare and precious thing: *understanding*.

This book is definitely not another boring reference manual. Real programming is a skilful art. This applies especially to programming in Bash. Art is not easily learnt from manuals. Not even from pedagogically perfectly organised treatises full of exhortations and exercises. Manuals are for looking things up, same as are search engines on the internet. By all means, look up the details as and when needed (1)[1] but please read this first.

This book is based on the author's lifetime experience with using and teaching numerous programming languages. The experience has shown that the most effective way to learn programming is by playing with fun examples.

[1] https://www.gnu.org/software/bash/manual/bash.html

Examples from cheat sheets (2)[2], examples of 'boiler plate', examples of powerful techniques that actually accomplish useful things, examples of advanced tips and tricks, examples of amazing solutions to interesting problems. Examples, examples, examples.

It is playing with those inspiring snippets of code that brings on those 'eureka' moments, when we want to shout:

"Wow, I did not know that!"

"So, this is how to do it! Yes now I really get it!"

"What fun, let me try this myself!"

"Now, let me just change this a bit and it will solve my own fascinating problem!"

Typesetting conventions

- technical terms and software packages that are not in common English use, inline bits of code, etc, are all literalised in a fixed width 'computer' font like this: `wget`
- bigger chunks of Bash code are additionally block set and partially coloured
- special terms are single quoted, e.g.: 'guru'. Sometimes this may involve a measure of gentle irony
- author's emphasis is indicated by *italicised text*
- acronyms are capitalised and introduced after their first use in parenthesis (round brackets): (REPL)
- double quotes "..." mark, as usual, speech
- link references are placed in round brackets (1)

[2]https://devhints.io/bash

Program in Bash anywhere

One of the objectives of this book is to make programming accessible to everybody. It is actually quite easy to stop clicking on the icons for a while and start running powerful, automated, general Bash scripts instead. It is the secret of the computing professionals, known as the Command Line Interface (CLI).

There are some Unix/Linux utilities (commands) within Bash, for example `wget`, that put at your fingertips direct access to the whole of the internet and all the servers and cloud services that you may have been using. All in an organised and automated fashion.

Under `Linux` and `MacOS`, there is no need to install anything. Just open the default terminal and you are ready to go. The terminal is an interactive Bash session.

Under `Windows` (10-11), install Windows Subsystem for Linux (`WSL`). Then you get most of the benefits of a proper operating system, without having to partition your disks and dual boot your computer. Most of all, you get a little Linux (Ubuntu) Icon, which activates the real Bash terminal with full access to the whole file system.

It is even possible to program in Bash on a mobile Android phone. Install an app called `F-Droid`. That in turn will allow the installation of the latest version of `Termux`, which is a fully fledged Linux with Bash included. You can also install any other Linux compilers and applications, even though they will probably eat up a fair chunk of your phone's memory. Let us hope that the developers of `Termux` will, in time, make it possible to use an SD card instead. Anyway, even the basic install is perfectly adequate for enjoying Bash programming.

It is probably true to say that no other useful programming language is so simple and so easily and universally available.

Dialects and styles

When searching the internet for answers to questions and problems related to Bash, it quickly becomes apparent that there are many questions but even more answers. This is partly due to the long and active history of shell programming. At first glance, it is not at all obvious, which of the suggested methods or styles are 'deprecated', outdated, pertaining to some obscure dialect fallen to disuse, or just plain wrong. At the same time, there are often more than one valid solutions with different merits. All that can be somewhat confusing.

Accordingly, this book prefers a simple, unified, rational, style. Insignificant alternatives are rarely offered or discussed, in order to keep the complexity down. That is not to say that other valid styles and solution are not 'out there'.

Why Bash?

> "To Bash or not to Bash, that is the question!"

Bash shell script is one of the 'old school' languages, going back to mid 1980s and beyond. It was originally developed as part of the Unix project in conjunction with C. That is why some of its (Unix/Linux) functions/utilities are in fact the same as in C. For example, the print command `printf`. Bash is the user facing heart of any UNIX/Linux based Operating System (OS). That is why today it remains important to grey bearded systems admin gurus and skilled users alike. Its longevity speaks volumes about its proven strength, usability and versatility.

It is because of this close integration with the OS that, in fact, all Unix/Linux commands, plus anything that you might install or implement yourself, are all directly available and trivially simply invokable in Bash.

Notes for the professionals

On the other hand, if you need a lot of 'hand holding' in case you do something silly, if you work on large collaborative projects and if you do not mind a much steeper learning curve, long compilations and large executables, then probably you should go to the other extreme and use Rust.

Actually, there is a lot to be said for combining within the same overall project Rust and its prodigious speed and safety, with Bash. Bash is great for effectively automating the overall project administration: files marshalling, moving, renaming, archiving, updating, uploading, downloading, compiling, backing up, program executions, etc. Thus one can get the best of both worlds.

This book took shape when the author started using Bash within his TokenCrypt project (4)[3], originally just for the prosaic convenience of it. Warming up to it, it came as a bit of a shock to realise that two modern programming styles, namely Logic Programming and Functional Programming, since then much loved and hyped in the academic circles, are in fact already available in Bash. Admittedly in a dormant and embryonic form but they are there. In retrospect view, Bash can be said to have been seminal to their development.

Bash is said to be the language which many use (or need to use, or wish to use) but few understand. Maybe this book will help to change that.

First steps

Anyone can certainly list a few commands like `cd ..; ls -al` in a script file and then run it. In fact, doing just that, is an excellent way to get started with Bash programming.

[3] https://github.com/liborty/TokenCrypt

Create a file called `myscript`, put some commands in it and make it executable with `chmod +x myscript`. Then just typing `./myscript` will make Bash execute all the commands contained in it.

Apart from Bash, there are several other dialects of shell scripting languages. The convention dictates that the scripts should always start on the very first line with the so called 'shriek-bang' `#!`, which indicates which particular interpreter (shell) is wanted and where exactly it is to be found. Most often, it will be just this: `#!/bin/bash`

However, on Android phones, Bash will be installed in a non-standard location with a longer path. The path specification should therefore always point to exactly where on the filesystem Bash resides, otherwise it may not run.

Any normal commands will then follow, exactly as you might type them to the terminal prompt. Hash `#` turns the rest of the line into a comment. Bash is very terse. A lot can be happening in just one line of Bash code, so it is a good idea to be generous with explanatory comments.

Note: sometimes it is good to try things out by typing them directly to the Read Eval Print Loop (REPL), which is what an interactive Bash terminal is. It can happen that after typing out the correct answer, the REPL will add a complaint: `command not found`. For example, try this little bit of arithmetic: `$(( 1+1 ))`. There is nothing wrong, really. The message is just Bash trying to evaluate the answer `2` as well. You can stop it happening by using `echo` or `printf` commands, which drop their values (in this case do not pass them back to REPL), like this: `echo $(( 1+1 ))`

Moving beyond this, to really understand the subtle points of Bash as a 'full-on' programming language, that opens up entirely new horizons.

I hear some of you exclaim in horror: "Why not Python?". It has to be said that Bash still holds some big advantages over Python and other similar interpreted scripting languages (Ruby, etc.), making Bash easier to learn as the first programming language:

1. Bash needs no installations and voluminous libraries searching.
2. The 'Read-Eval-Print-Loop' (REPL) for trying things out is right there, in any terminal. It is recommended to open a terminal alongside this book and experiment with the code.
3. Bash is tightly integrated with the Operating System (OS), right out of the box. This has the side benefit that learning Bash, you will also get to understand much better how your OS works. This is somewhat like what learning Latin used to do for european human languages. The same can not be said for Python, etc.
4. The EIJAS principle = 'Everything is just a string' (more on this later).

Chapters outline

- **Introduction** - the why and how of Bashing.

1. **Quoting Quoting Quoting** - develops an understanding of the fundamental principles: what is most needed for confident and successful Bash programming.
2. **Bash Branching** - the practicalities of Bash: conditionals, program control, functions, loops. An example of applying them to perform the common task of processing and interpreting program arguments.
3. **Input-Output** (I/O) handling is an important part of any programming language. In Bash this applies especially. Its streams, redirections and pipes act like glue that ties everything together. That means internally as well as taking care of the traditional role of communicating with the outside world. After this chapter follow increasingly sophisticated real applications and projects. With examples of code, of course.

4. **Amazing Arithmetic** is a useful project exploiting the EIJAS property of Bash to implement fixed point arithmetic. This is achieved by inserting the decimal point in the right place in the strings of digits. All without any external utilities, just using the default Bash integer arithmetic.
5. **SubShell Scheduling** is an advanced project that shows how to automatically orchestrate background jobs in separate threads. It deploys an interesting technique of program self modification. It is also a good template code for using files and directories effectively.

- **Conclusion**
- **Link References**

Quoting, Quoting, Quoting

Bash is nowadays considered by sloppy programmers to be prone to programming errors but it is really not so bad, as long as we watch out for a few pitfalls and ‘gotchas’.

The main potential problem are *spaces in the wrong places*. This is the price we pay for programs and data all being one and the same thing to Bash. Youmaystillbeabletoreadthis. However, computers are ‘syntax nazis’. They ‘hate’ any typing errors or ambiguities. Spaces are important to the structure of any text, that means they are also important to the structure of the Bash programming language.

Everything in Bash is just a string of text (the EIJAS principle). That means, Bash only knows about and understands chunks of text. We will see later how this simplicity brings about some amazing programming opportunities and advantages.

The first and foremost advantage is simplicity: neither the user, nor Bash, have to worry about all kinds of different types and about checking them all. There is really only text and sometimes explicitly annotated text interpreted as digits.

However, with power comes responsibility. We have to be prepared to get more errors and therefore it is wise to test small units of program at a time. In case of big trouble, that means hard to find bugs, it is highly recommended to
introduce extra `printf` statements into the program code to print out values of variables that you suspect may be wrong. This method is almost always more focussed and therefore more successful than uninformed general debugging, in any language. Afterwards, it is easy to comment out or to delete those extra statements.

In order to handle text effectively in different situations, there are several different ways of annotating or *quoting* it. Each has its own specific effects and determines how is the quoted text to be interpreted and/or acted upon. It is important to understand them all and the differences between them.

Individual characters

First of all, there is 'escaping' of some special and non-printable individual characters. Funny example:

```
printf "My book sales reached 200%%\n"
```

There are specific rules which only apply within printf formatting strings.
Here the non-printable character 'newline' is encoded as `\n`. The printable character `%` is escaped by being doubled, as otherwise it has a different interpretation within the printf formatting string. In other words, it is a 'reserved character', just in this context.

The complete character 'escaping' rules are the one thing that nobody even claims to remember fully. They are what the search engines are good for. Most people manage pretty well most of the time with just these two, plus maybe the tab `\t` for tidy printouts.

The general rule for double quoted strings anywhere (not just after `printf`), states that the following Bash reserved characters must be escaped by being preceded by a backslash: dollar, backtick, double-quote, backslash, newline. The escaped 'newline' here means an actual newline, such as at the end of a line of code, which we want to continue uninterrupted onto the next line.

Let us move on to strings (sequences of characters).

Literal string

Single quotes `'...'` enclose a simple unit of literal text. Nothing inside the single quotes is evaluated. Example:

```
'the cat sat on the mat'
```

To make a single quote part of the string, without terminating it prematurely, it is necessary to use outer double quotes, like this:

```
 "the cat's mat was eaten by a rat"
```

Interpreted string

Double quotes have the additional function of evaluating (expanding) any variables or expressions found within them. The *expansion* is indicated by sticking a dollar sign `$` immediately in front of the variable/expression. In fact, the dollar sign causes the *expansion* anywhere else as well, apart from within those literal strings above. Example:

```
"The cat's name is $NAME"
```

Capitalising the variables is just a useful convention.

Note: Double quoting of just a variable alone, e.g. `"$FILENAME"`, is often advised in all evaluation contexts to prevent filenames containing spaces (a horrific Windows 'invention'), from being interpreted as several different files, instead of just one file as was intended. Similarly, in tests, double quoting a variable ensures that it always returns at least an 'empty string' value. Otherwise an unset variable just disappears after expansion and this can sometimes cause program syntax errors.

Assignment

When assigning to variables, the left hand side of an assignment is unevaluated, so it does not take the dollar sign:

```
NAME='Bob'; ANOTHERBOB=$NAME
```

Also, be very careful not to use any spaces around the assignment operator =.

Variable expansion

`${VAR}`, using braces with `$` outside, is another way of quoting variables, called the variable expansion. After the usual evaluation of the variable, it can also apply some special functions to its value. Those functions may involve, for example, searching and selecting specific parts of a path/filename, as in:

`EX="${FILESPEC##*.}"`

This assigns the last file extension to `EX`, by deleting everything in `$FILESPEC` value before and including the last dot. The wildcard character `*` is use to match everything before that last dot.

`${V#*"$C"}` deletes everything from the value of `$V`, from its beginning up to and including the first occurrence of the character specified in `$C`.

`${V##*"$C"}` deletes everything from the beginning up to and including the last occurrence of `$C`.

`${V%"$C"*}` deletes everything from the end up to and including the first occurrence (from the end) of `$C`.

`${V%%"$C"*}` deletes everything from the end up to and including the last occurrence (from the end) of `$C`.

`${V:=value}` initialises V to (default) value if it was unset.

`${V:?error message}` prints error message and exits if V is unset. The message can be a double quoted interpreted string.

`${VAR@Q}` automatically transforms the value of $VAR, by applying the individual character backslash escaping rules, where necessary. When in doubt about which characters require escaping, then ask the expert (Bash itself). This is useful whenever some text, (possibly containing special characters), is read into $VAR and it is then to be interpreted in some way by Bash.

Logical eval

`[...]` logical evaluation (test) of an expression. It is expected to return a boolean true/false value. It can use the comparator operators `-lt -le -eq -ne -gt -ge` when comparing numbers and `=`,`!=` when comparing strings. It can form part of a bigger expression, with several sets of square bracket tests joined by logical operators: `!, ||, &&` (meaning 'logical not', 'logical or' and 'logical and', respectively).

In fact, everything in Bash is an expression. Even an executed command returns an 'exit status' value: `0` (true) for success and 1-255 for user defined failures. Zero is interpreted as true because it indictates a successful exit, so that program calls can be included in the tests.

Regular expressions (regexp) comparison tests using `=~` operator should be enclosed in double brackets `[[...]]`.

Simple invocations of programs, functions and utilities can be tested for successful outcomes directly, without the square brackets, for example:

```
mkdir somedir || echo failed to create somedir
```

See the next chapter for these simple forms of conditional statements.

It is common to test variables for being set or unset:

- `[ -n "$VAR" ]` true if `$VAR` has been set to something of non zero length
- `[ -z "$VAR" ]` is true if it is unset
- `[ -f "$VAR" ]` is true if value of `$VAR` is an existing file
- `[ -d "$VAR" ]` is true if value of `$VAR` is an existing directory
- `[ -r "$VAR" ]` is a readable file
- `[ -h "$VAR" ]` is a symlink
- `[ -w "$VAR" ]` is a writable file
- `[ -s "$VAR" ]` is a file of nonzero size
- `[ -x "$VAR" ]` is an executable file
- `[ "$F1" -ef "$F2" ]` F1 is equal to F2 (same file content)
- `[ "$F1" -nt "$F2" ]` F1 is newer file than F2
- `[ "$F1" -ot "$F2" ]` F1 is older file than F2

Arithmetic eval

`$(( $NUM1+$NUM2 ))` or 'arithmetic evaluation', causes the evaluation of the enclosed arithmetic expression. In this case the sum of two integers. There must be spaces left between the double paranthesis and the expression inside.

Bash has no floating point types or calculations with them (but see in a later chapter our fixed point arithmetic project).

There are the usual (integer) arithmetic operators: `+,-,*,**,/,%` and their updating assignment versions, e.g. += which adds the right hand side (RHS) value to the existing value of the LHS variable, e.g.:

`let COUNTER+=1`. If `$COUNTER` was not initialised, it will be given initial value `0` (only in these updating assignments).

The `let` form triggers an arithmetic evaluation, somewhat like the double paranthesis above but then it drops the produced value. Therefore `let` is often used for arithmetic assignments.

When unsure about the operator's precedence, use extra paranthesis.

Eval

`eval command arg1 ... argn` is the normal evaluation. It executes `command arg1 ... argn` in the current shell. It can be used to *construct* and execute any function call, or a call to a linux external utility.

Comparison: `printf "$P: "` returns (formatted) value of variable `$P`, whereas `eval $P` in addition also executes its value as a command.

```
P=pwd; printf "$P: "; eval $P
```

Braces

```
{ P=pwd; printf "$P: ";  $P; }

{
    P=pwd
    printf "$P: "
    $P
}
```

Braces are used to bind together a number of commands into a 'block', executed at the current shell. Note the last semicolon before the closing brace, without which the value of $P would not be executed.

In the multi line syntax (the last version), the new lines substitute the role of the semicolons.

Control question: what will this print out? Try it out! Only a few other languages have this very powerful (some say downright dangerous) ‘program as data’ facility. Lisp is one of them.

In fact, all these three versions produce the same results, the differences are only in their syntax.

Subshell eval

`$( command1; command2; )&` has the same effect as the braces above, except that it starts a new subshell for the evaluation (command substitution). This has some small overhead costs associated with it. However, it is useful for ‘spinning off’ background processes by optionally appending `&` after the closing paranthesis. Such processes will then run in parallel, in separate threads.

Beware of their inputs and outputs getting out of sequence. Also, be very careful that you use the `wait` command before starting another process which relies on the results of a background job! To exploit parallelism requires generally a lot more careful thought than the traditional single sequential process does. As a rough rule of thumb, take it ‘from the top’, that is run in parallel the big independent jobs first. Then all the computations within them will run on separate cores, without getting tied up in knots over their fine grained details.

This `&` mechanism is very useful for better utilising modern multicore processors. The OS takes care of assigning the threads for execution to the available physical cores.

It is sometimes more convenient to use for simple commands the more concise backtick syntax: ‘command ‘ but this is not recommended for nested evaluations, as escaping multiple backticks can get messy and prone to typing and oversight errors.

Text juxtaposition

We can build up arbitrary strings by trivial juxtaposition of variables, characters and other strings. No messing around with explicit calls to `concatenate`, `append`, etc, like in most other programming languages:

```
H=Hello
W=World
# build up a sentence:
# insert an escaped space between the two words
# and append ! onto the end
echo $H\ $W!
```

Another common use of juxtaposition:
`$PATH/$FILE.$EXT`

Bash Branching

The `if` statement has the following structure. The `else` block is optional, `fi` is always mandatory.

```
if logical-expression; then
    statement1
    ...
    statementn
else
    neg-statement1
    ...
    neg-statementn
fi
```

The tabbing somewhat improves the readability of the code. The expression typically involves the logical tests in square brackets and logical operators.

Bash is easy to love but sometimes it is a love-hate relationship. Take this `if` statement, for example. Forget to put that semicolon *before* `then` and you are then in deep trouble. It would be a pity if this quirky syntax were to put people off shell programming.

Fortunately, there is a better way of branching, at least in simple situations. Complicated `if` statements are best avoided anyway, as they are the biggest source of errors affecting the fundamental program logic and can be hard to discover.

Use simple verification tests leading to an early `continue` or `break` from within loops, `return` from functions and `exit` from programs, whenever some necessary condition is not satisfied. This, in effect, turns the following code into an `else` branch, without a complex conditional structure dragging all the way along with it. Besides being a good coding practice, it tends to increase program efficiency. We will call it an *early break principle* (EBP).

Short circuit logic evaluation

When evaluating a logical (boolean) expression, in any context, the value of the result is often known well before all the individual parts of the expression have been evaluated. How is that possible?

In a conjunction (`&&`), the whole result is known to be false as soon as the first conjunct returns `false`.

Conversely, in a disjunction (`||`), the result is known to be true as soon as the first disjunct returns `true`.

These logical operators have equal precedence and are normally processed from left to right.

The *short circuit evaluation* property of logic is used in Artificial Intelligence (AI) for efficient evaluation of so called AND-OR trees in game-playing and search applications. However, we can use it too in our humble Bash programs.

In oder to utilise this property, we introduce two kinds of conditionals:

Positive conditional

```
[ test1 ] && [ test2 ] && [ testn ] && { posblock }
```

Here the posblock of statements at the end gets executed only when each of the tests returns true. As soon as the first test evaluates to false, the whole thing fails and quits there and then. Hence the expression ‘short circuit evaluation’.

Negative conditional

```
[ test1 ] || [ test2 ] || [ testn ] || { negblock }
```

Here the negblock of statements at the end gets executed only when each of the tests returns false. As soon as the first one evaluates to true, the whole thing succeeds and quits.

Logical form of an if statement

Compare these two forms and understand the greater facility of logic to express diverse conditionals:

```
if [ condition ]; then posblock; else negblock; fi
[ condition ] && posblock || negblock
```

Benefits

This syntax (structure) is cleaner, more compact, more general and less prone to errors than the `if` statements provided in `bash` and most other languages.

The semantics (meaning) is more expressive, with several types of different if-statements for the price of one.

Program efficiency can be improved by ordering the individual tests to encourage an *early break*. See EPB principle introduced above. In a positive conditional, the tests most likely to fail should come first. In a negative conditional, the tests most likely to succeed should come first.

When unsure about the likelihoods of their outcomes, then ordering the tests from the fastest to the slowest is the best, in both cases.

Arbitrarily complex propositional logic expressions can be built with this framework, using `&&, ||, ! (not)`. Square brackets should be added for disambiguation, as and when needed (just as the paranthesis are used in arithmetic). So, for example, `! test1` can be used to intermix a positive `test1` into the negative conditional. Alternatively, `! test2` to add a negative `test2` into a positive conditional. The negations or even primitive negative tests can often be avoided altogether by changing the negative conditional (disjunction) to a positive one (conjunction) and vice-versa.

Chaining statements

Because in Bash there is no real difference between statements and expressions, the above logical short circuit mechanism can be just as well employed to chain together commands. Furthermore, because they return values interpreted as success (true) or failure (false), there will be similar EBP gains in efficiency.

There is a clear relationship here to the kind of programming known as 'Logic Programming', invented much later. See, for example, the programming language Prolog, which is built around the ideas of evaluating the (positive) conjuncts and backtracking on failures.

Positive (and) commands chaining

```
command1 && ... commandn
```

This is normally used in situations where it is required that all the commands should succeed.
Note that the logical 'and' operator is used here for joining up the commands, in place of the previously used semicolons. The effect of this is going to be an *early break* with false value returned, as soon as any individual command exits with failure status. Presumably, this outcome should be reported by some kind of error message, by

tacking on the end `|| printf "error message\n"`, or deploying some remedial actions.

The rest of the commands will then be ignored, whereas with the semicolon separator they would all continue being executed regardless of their return status.

Negative (or) commands chaining

```
command1 || ... commandn
```

Note that the logical 'or' operator is now used in place of the semicolons. The effect of this is going to be an *early break* from this block, as soon as any command exits with success status. The rest of the commands will then be ignored. Here the logical intent is that the success of any one of the commands is all that is required.

Alternatively, this form may be used when it is expected (hoped for?) that they should all fail, such as when checking for occurrences of security breaches. Then the whole block should return failure. Should it succeed at some point, this can be reported by tacking on the end `&& printf "Security breached!\n"` or any other remedial actions.

Functions

Bash also has `functions`, which work just like any external script, except that they are textually inserted into the script in which they are then used. The benefit of being called from the file in which they are defined is that they are not run in a subshell.

With all that convenience comes the danger of unintended global variables clashes, which generally result in horrible bugs. It is therefore strongly advised to use `local` declarations for functions' own working variables. This will insulate them, as if the function really was in its own separate script file and run in a subshell. Note that the functions also inherit any file stream identifiers declared and belonging globally to their script (shell). More on input/output later.

In a function, as in any program script, its formal arguments are identified by special variables `$1,$2...` and so on. Matching one for one, they are assigned the values of the actual arguments supplied in the function/program call.

To return values other than `0-255` from Bash functions, the global `stdout` stream `1` should normally be used. That is, in the simplest case, use `echo` or `printf` commands within the function to write its return value(s) to standard output. This will then be returned as the function call result.

Here is an example of a recursive function with one formal argument `$1`.
When writing recursive functions, the termination condition should always come first. Note the extensive but here unfortunately necessary use of the arithmetic evaluation double paranthesis: `$((...))`. To keep the number of the paranthesis down, the recursive call itself is evaluated here with the backtick subshell eval, sometimes called the command expansion, which works just like: `$(...)`.

```bash
function factorial {
  [ $1 -lt 3 ] && echo $1 || \
     echo $(( $1*`factorial $(( $1-1 ))` ))
}
time factorial 20
2432902008176640000
real    0m0.027s
user    0m0.026s
sys     0m0.005s
```

Beware that the answers will overflow without warnings for relatively small arguments, as the factorial of 21 is already bigger than the maximum 64bit integer.

An iterative implementation will be more efficient than the above stacking up of multiple subshell calls. Here we use the ‘C-like’ form of the iterative for loop, which will count down to 2, cumulatively multiplying all the (descending) integers into the local variable `F`. If the supplied argument is smaller than three, the loop will do nothing and the argument value will get immediately echoed to stdout as the answer.

```
function itfactorial {
  local F=$1
  for (( i=$1-1; i>1; i-- )); do F=$(( $F*i )); done
  echo $F
}
time itfactorial 20
2432902008176640000
real    0m0.000s
user    0m0.000s
sys     0m0.000s
```

This time it was too fast to register on the timer, thanks to being evaluated within the current shell and not using the stack.

However, it should be pointed out that Bash, being an interpreted language, is not exactly fast. Therefore, maximum speed is rarely of the main concern. Speed is relative. Nowadays, thanks to faster processors, even Bash runs faster than the fast languages, like C, did a few years ago. Another good reason for its revival? Nonetheless, the stronger points of Bash are undoubtedly convenience, versatility and compactness.

Loops and control statements

Bash is also equipped with the usual set of `while`, `until` and `for` iterative loops, plus a `case` statement. This is not a difficult or exciting topic, so we will not dwell on it much. It is far more effortlessly absorbed by simply reading some existing useful code. We have already met one form of the `for` loop. The `case` statement will be used in some of the examples in the following chapters.

An infinite loop (relying on calling `break` to exit when the data runs out or something like that) can be a useful mechanism. It is also used for human-computer interactions, a good example of which is, in fact, REPL. The user usually manually terminates such interactive sessions by closing the input with `Ctrl-D`, or closing the terminal with `exit`. Should you try the following examples in your terminal, be prepared to type `Ctrl-C` quickly to terminate them. Note that they both set up an infinite loop.

```
while true; do
  echo 'I am a lumberjack and I am OK'
done
```

```
until false; do
  echo 'I sleep all night and work all day'
done
```

Of course, the more usual form of `while` statement is to read some ordinary text file `$F`. Here the `read` statement will return `false` at the end of the file:

```
cat "$F".txt | while read LINE; do
  processline $LINE
done
```

We have already met the C-like version of the `for` loop command but it has two more useful forms. This one will apply `processfile` to all the `txt` (text) files in the current directory:

```
for F in ./*.txt; do
  processfile "$F"
done
```

This version of `for` goes through the range of values 0..100 in steps of 3 (and prints them all on the same line):

```
echo 'Multiples of three: '
for MULT3 in {0..100..3}; do
  printf "$MULT3 "
done
```

So far, we have discussed the different possibilities of evaluating and joining up commands, the conditionals, functions and loop statements.

At this point, you should be able to read real Bash code. Therefore we will now move towards some interesting examples of code and projects, with explanatory comments.

Processing arguments

Passing arguments to programs and functions is the chief way of making them more general and a lot more useful. It is an important part of the abstraction process. Therefore, most Bash scripts expect some arguments and they need to process (read) them.

It is a good idea to put a human readable comment just after the first line, explaining what arguments the script expects. At the same time, the conventional options `-h` and `--help` should cause the script to print out similar helpful information.

```bash
#!/bin/bash
# This program expects some options,
# followed by three mandatory directory paths.

# The program path/name is always in $0.
# Here we chop off the path
# before the program's name, with `##*/`
PROGN=${0##*/}

# Frequently used messages subsumed into functions;
# their output is redirected to stderr with >&2
function usage {
   printf "Usage: $PROGN \
   [-h][-b][-c][-x][-q][-r][-u][-v][-z] \
   indir keydir outdir\n" >&2
}

function helpmsg {
   usage
   printf "\t%s\n" \
     '-b | --b64 : test for base64 encoded files' \
     '-c | --clean : the archive' \
     '-h | --help : print this text' \
     '-q | --quiet : turn off the final summary' \
     '-r | --recurse : process also subdirectories' \
     '-u | --update : the archive' \
     '-v | --verbose : detailed reporting' \
     '-x | --hex : test for hexadecimal files' \
     '-z | --zstd : use zstd compression' >&2
   exit 1
}
```

```

# Use the utility `getops` for options parsing:
# puts listed options, that is words that can  follow -,
# into $OPT
while getopts hbcxqruvz-: OPT; do
 # Long options start with --,
 # this gets detected by the last option in the list: `-`
 # then assign the long option name to $OPT manually:
 if [ "$OPT" = "-" ]; then
 # found --: reset OPT and OPTARG
   OPT="${OPTARG%%=*}"  # extract long option name
   # extract long option argument (here empty)
   OPTARG="${OPTARG#$OPT}"
   # remove any assignments `=` from long options
   OPTARG="${OPTARG#=}"
 fi
 # now process and record all options, in any order
 # here we set their variables,
 # later detect them being set, e.g. with: [ -n "$VAR" ]
 case "$OPT" in
   h | help ) helpmsg;;
   b | b64 ) B64TEST=0;;
   c | clean ) CLEAN=0;;
   x | hex ) HEXTEST=0;;
   q | quiet ) QUIET=0;;
   v | verbose ) VERBOSE=0;;
   r | recurse ) RECURSE=0;;
   u | update ) UPDATE=0;;
   z | zstd ) COMPRESSOR=zstd;;
   ??* ) printf \
   "$PROGN quitting, bad long option: $OPT\n" >&2
      usage; exit 2;;
   # bad short option (will be reported via getopts)
   *) exit 2;;
 esac
```

```
done
# Remove all parsed options and args from their
# list $@
# so that we can parse the remaining arguments.
# The next one will be in $1,
# as if no options were given:
shift $((OPTIND-1))

# Validate the input directory
[ -d "$1" ] || {
   printf "$PROGN: input directory $1 not found\n" >&2
   exit 2
}
...
# optional final report
[ $QUIET ] || {  printf ... }
```

Now, the user can invoke this program with any combinations of the listed options, preceded by `-` and followed by three directory paths. The options, as their name suggests, give the program optional capabilities and modes of operation. For example:

```
ncrpt -rbxuc dir1 dir2 dir3
```

Input-Output

Of all the programming languages, Bash must have just about the simplest and most convenient input/output (I/O) handling.

Basic terminology

- files
  - are usually stored on a filesystem, where they are identified by their path, filename and filename extension(s)
  - when stored on the internet, then their universal resource locator (URL) fulfils the role of the path. These files need to be copied (downloaded) over the network by some utility program, such as `scp` (ssh secure copy) or `wget`. Once securely on the local filesystem, they can be read and manipulated much faster and more securely.
- streams are sequences of bytes, i.e. the actual data, with an associated internally memorised current position. Reads and writes automatically advance that position.
  - input streams are for reading
  - output streams are for writing
  - in-out streams are for both reading and writing
- file descriptor (perhaps more accurately a stream descriptor) is the means of referring internally, within a program, to a particular stream of data. A program will use these to communicate with the ‘outside world’, i.e. to read and write some data. The streams need to be named (in Bash actually by single digit characters) because several different streams can be used at the same time for different kinds of data and different purposes. For example, one stream for writing

clean results to a file and a separate one for writing out warnings (and errors) which we do not want intermixed with the results.

Opening a file associates a nominated stream descriptor with the named file and specifies if that file is to be used for input, output, or both.

Stream descriptors in Bash are just digit characters `0-9`. The first three, `0-2`, have special meanings. They are always opened automatically in every shell:

- `0` standard input (known as stdin), is the default input to any Bash command.
- `1` standard output (known as stdout), is the default output from a command.
- `2` standard error (known as stderr), is the 'error' output.
- `3` to `9` can be explicitly opened and assigned to files or other streams by the user.

`0-2` should not be opened for a real named file. In fact, they have predefined system files associated with them, see later. Their streams of data are either directed to processes and files with `<`, `>`, `>>` or sent/received to/from a pipe `|`, which is a hidden buffer inserted by Bash between two 'piped' commands, whereby the first one writes to it and the second one (asynchronously) reads from it.

The meaning of zero

This is a word of warning. Zero is for some reason very important to Bash. Its designers must have been very fond of it. Oh, seeker, realise the meaning of zero and become a Bash Master!

Variable `$0` always holds as its value the path and the name of the current program or function. It is often used by the program to identify itself as the source of an error message. `PROGN=${0##*/}` holds its root name (with the path up to the last / stripped off) in our examples.

Not to be confused with `0` written to stdout to indicate a successful outcome.

Which happens to be the same as the value of 'true'.

Which also happens to be the same as the character `0` (binary ascii value 48) in strings etc.

Not to be confused with `0` within an arithmetic evaluation context `$(( 0 ))`, where it means the integer zero (binary value 0).

Not to be confused with the character `0` where stream identifiers might be expected, as there it becomes the standard input (stdin) stream identifier!

All clear now?

Using I/O

The streams (1-2) are often simply directed (written) to real files with `> $FILENAME`, without having to 'open' those files.

`>> $FILENAME` will append to the end of an existing file instead of creating a new one (or overwriting and existing one).

To append several files at once into stream `0` (stdin), we can invoke program `cat $F1 $F2` or `tac $F2 $F1`, where `tac` presents the file's lines backwards.

Sometimes we may wish to read chunks of an input file and to remember the current reading position in-between, down to the last byte. It is better to define our own dedicated stream to do this. Let us open file `$F1` for reading on stream `3`. We use special command `exec` which replaces the default shell stream(s) with the new one(s). In this case, it adds the newly created stream `3`:

```
exec 3<$F1 || {
  printf "$PROGN: '$F1' failed to open for reading\n" >&2
  exit 2
}
```

To open an output file, use `>` and for both input and output, use `<>`.

Example: invoking utility program `head` to read just the next `$N` bytes from stream `3` and writing them to `$FILE2`:

```
head -c $N <&3 >$FILE2
```

The next execution of this command will read the following `$N` bytes from `$FILE1` and so on. Note that, unlike `cat` and `tac`, this disregards the 'lines structure' of text files and thus is particularly suitable for reading binary files.

When we no longer need stream 3, we close it with: `exec 3>&-`. The dash means 'close this stream'.

Streams can also be redirected from one to another. For example, `1>&2` redirects stdout to stderr. Leaving 1 out, it will be assumed by default. Thus `>&2` has the same effect (see above).

Occasionally, we may need a stream to directly stand in for a file. Suppose we wish to pipe 100 random hexadecimal characters to an external program `prog`, which for some reason does not have the ability to read from stdin. Instead, `prog` expects an explicit input filename argument. For this situation, there are certain standard system filenames, permanently associated with the standard streams:

```
tr -dc '0-9a-f' < /dev/urandom | \
head -c 100 | prog /dev/stdin
```

Here we sent a 'lazy' stream of random valued bytes (0-255) from `/dev/urandom` to utility program `tr`, which deletes all but the specified hexadecimal characters ranges and pipes the surviving ones to `head`. Head then takes just one hundred of them and pipes them to `prog`. Prog can not read the pipe on stream `0` (stdin), as per usual but instead opens the file `/dev/stdin` and finds the bytes there.

Similarly, to throw away some unwanted messages, data, or spam, direct them to `>/dev/null`

Pipes and functional composition

Function calls, subshell evaluations, Bash commands, invocations of other programs, basically everything in Bash, are functions which write their produced values by default to stream `1`. The results are then received from stream `0` as the returned value of their call or passed on via the pipe.

This mechanism is somewhat like what happens in *functional programming*: it facilitates convenient composition. In fact, with a bit of self imposed discipline not to abuse global variables, good old Bash can be used for functional programming. Abusing global variables is a bad practice anyway, it is bound to bite you painfully when you start using `&` (background parallel execution of sub processes).

We will now demonstrate the piping to and from any Bash statements on an example of `case` command. The following text fragment is taken from TokenCrypt project (4)[4].

It shows how we can *switch pipes*. That is, factor out all common computations and put them upfront. Then pipe their results to a `case` statement, within which each branch is going to continue the pipe actions in its own specific way. Then we continue again with one common path after the `case` statement is finished. That

[4]https://github.com/liborty/TokenCrypt

is, in this example, write the decrypted decompressed result into `$OUTFILE`.

The code reads a key of known `$FSIZE` from stream `3`, decrypts $INFILE with it and then chooses the relevant method of decompression, depending on the value of `$CH`. What gets written into `$OUTFILE` in the end will depend on which branch of the case statement was selected.

```
head -c $FSIZE <&3 | symcrypt $INFILE | case "$CH" in
   h ) hexify;;
   b ) base64 -w 0;;
   i ) lzma -d;;
   j ) lzma -d | hexify;;
   k ) lzma -d | base64 -w 0;;
   l ) zstd -d -c;;
   m ) zstd -d -c | hexify;;
   n ) zstd -d -c | base64 -w 0;;
   * ) printf "$PROGN importfile: unrecognised ch\n";
       exit 2;;
esac > $OUTFILE
```

Similar pipes can be set up for any Bash statements. Think of them all as filters of some data streams.

Amazing Arithmetic

In fairness, there is one practical drawback to the 'everything is just a string' (EIJAS) philosophy of Bash. Namely, the lack of floating point arithmetic.

Suppose we want to do a simple floating point division `C=$A/$B`. That is, we are not satisfied with just the truncated answer that the Bash default integer division gives, e.g. '`5/2 = 2`' and we want greater precision (in this case 2.5).

We could reach for external Linux utilities, such as `awk`,`dc`,`bc`, etc., and most people do. However, embedding their code is somewhat messy in Bash. Most of the facilities that these external programs offer are an overkill for a simple one line calculation. Plus we have to start a sub-process for them every time, even for trivial examples, such as this one:

```
C=$( awk "BEGIN { print $A / $B }" )
```

Luckily, there is a simpler and more efficient alternative: *fixed point arithmetic*. As an exercise, we are now going to implemented it in Bash. The full source is available in (3)[5].

Fixed point arithmetic is used in situations, where the numbers being handled are all of similar magnitudes, so that they do not need the separately maintained exponents of floating point numbers.

Two good examples are calculating percentages and money. In both of these common use cases, the decimal point has a fixed meaning (and position). This is in fact preferable in many applications, as the truncation errors are more predictable and controllable.

[5]https://github.com/liborty/fdiv

Then the 'mantissa' can be, in effect, expanded to the full 64 bits word. Note that the commonly used 'Double Precision IEEE 754 Floating Point Standard' has only 52 bits allocated for the mantissa. Therefore our fixed point arithmetic implemented on top of ordinary 64 bit integer arithmetic used by Bash, will gain up to 12 bits of extra precision.

Ratio in fixed point arithmetic

Ratios are easier for internal calculations than percentages. Besides, any barely numerate person is able to mentally translate them to percentages by simply multiplying them by a hundred.

Suppose we want to calculate the ratio of the byte sizes of two directories: $INDIR and $OUTDIR. The second one, $OUTDIR, is a compressed version of the first, and we want to find its compression ratio.

By taking the ratio, where we always divide by the bigger of the two numbers, we are normalising from two hard to handle ranges of the original values `[0,infinity)`, to a much more compact and convenient range of their ratio: [0,1].

- Mathematical notation: the round closing bracket above denotes a half open interval. Computers can not handle infinity, so we excluded it from the interval. We can only ever approach it.

In general, it is easy to check that we are always dividing by the bigger of the two numbers and not by zero. That is enough to ensure that the ratio will always be well behaved and in the interval [0,1].

Back to our compression ratio application. The following line calls `du` (disk usage) to fill an array, here called `SIZES`, with the sizes and names of the two directories:

```
# create array SIZES[size1, dir1, size2, dir2]
SIZES=$( du -hbs $INDIR $OUTDIR )
```

Next we use printf formatting to turn an ordinary integer `%d` into a fixed point number `0.%05d`.

In the printf formatting string, `%s` (for strings), `%d` (for decimal numbers), etc., get filled in order by the actual values of the evaluated arguments (if any), following the formatting string. Here they are the sizes and names of the two directories and finally the calculated compression ratio (\ simply allows continuing the code onto the next line):

```
printf "Output size:\t%s (%s)\n\
   Original size:\t%s (%s)\n\
   Compressed to:\t0.%05d\n" \
   ${SIZES[0]} ${SIZES[1]} ${SIZES[2]} ${SIZES[3]} \
     $(( 100000*${SIZES[2]}/${SIZES[0]} ))
```

The last line is the actual fixed point (integer) calculation. We have chosen here to calculate the ratio to five decimal places `0.%05d`, so we premultiply by 100000. The five zeroes correspond to the five decimal places we will output via the formatting string. The numerator is the size of the smaller compressed $OUTDIR, so the true ratio will be less than one.

We get this precision by making a temporary left shift by five decimal places and then carrying out ordinary integer division. Of course, this should be followed by somehow shifting back those five decimal places to the right, to produce the final result.

Good plan but how to shift right (divide by 100000), without a numeric type capable of holding those five decimal places of the result? Are we not going around in circles with this?

Here is the trick. We can break out of the circle with simple *textual manipulation*. Forget about dividing back by 100000. We just textually construct a fixed point number, by prepending onto the integer result the literal string `0.`. We can do this whenever we know that the result is less than one. I bet you missed seeing that trick in the `printf` format string above? Note that this gives us a ‘new fixed point type’, without violating the EIJAS principle.

At a stroke of a horrible type-violating magic that would make any modern day compiler die in convulsions, we turned the upscaled integer division result, say `1234`, into `0.01234`. In Bash this is not nearly as strange as it may seem, as everything returns only strings, anyway.

Of course, any leading zeroes after the decimal point are significant, so we have to pad on the left with them. The leading zero after `%` in the format string `"%05d"` does just that.

The digit `5` (significant places) always has to agree with the number of zeroes in our multiplication left shifting factor, in this case `100000`. It should be assigned to a variable, say `SIGDIGITS=5` and used as `"%0$SIGDIGITSd"` in the formatting string and as `10**$SIGDIGITS` in the scaling factor, thus avoiding any potential oversight mismatch problems.

Percentages (in integer arithmetic)

Similarly, the following prints the compression ratio (CR) as a percentage of the original size, rounded to the nearest one hundredth of a percent. All done in integer arithmetic:

```bash
# compression ratio as a percentage: xx.xx%
# multiplying by 10**5, then taking the ratio,
# adding 5 and dividing by 10,
# rounds to one hundredth of one percent.
# SIZES is an array holding: size1,dir1,size2,dir2
SIZES=$( du -bs $INDIR $OUTDIR )
printf "$PROGN: ${SIZES[1]} ${SIZES[0]} "
printf "=> ${SIZES[3]} ${SIZES[2]} "
if [ ${SIZES[0]} = ${SIZES[2]} ]; then
   printf "(100 %%)\n"
else
   # calculate percentage rounded to two decimal places
   CR=$(( (5+100000*${SIZES[2]}/${SIZES[0]})/10 ))
   # compression ratio printed as xx.xx%
   printf "(${CR: 0:2}.${CR: 2:4} %%)\n"
fi
```

Here we are not using the printf format string substitution but directly constructing the percentage number by selecting the first two digits of `$CR`, then splicing in the decimal point character, and ending up with the next two digits: `${CR: 0:2}.${CR: 2:4}`. The selection of individual characters and their ranges is one of the functions available in variable expansion `${CR}`.

Bash is perfect for such textual hacks, precisely because ‘everything is just a string’.

General division

It could be objected that these are just specialised printing tricks and that numbers are not of much use unless we can do general calculations with them. Can we therefore write a Bash function that will do the division without any predetermined hard coded number of decimal places? Ideally, always giving the maximum precision possible?

Let us call that function `fdiv`. Here are some more detailed requirements for it:

- `fdiv` bash script takes two integer arguments: numerator and denominator. It checks the arguments and returns errors if they are missing or not integers or the divisor (denominator) is zero. It deals correctly with negative argument(s).
- it returns a fixed point number division result to the best possible precision allowed by up to two 64 bit integers, rounded to the nearest next digit, for example:

```
fdiv -2 -3
0.66666666666666667
fdiv 355 113
3.14159292035398230
```

- When the numerator is zero, 0.0 is immediately returned.
- Automatically calculates with the maximum precision. Truncation error will be less than 5E-19.
- When `numerator > denominator`, then the answer is composed from the whole number (numerator/denominator integer division), followed by the decimal point and then the upscaled remainder divided by the denominator. The last part is less than one and therefore gives the digits after the decimal point, similarly to 'Fixed Point Ratio' section above. Overall, this maintains the 5E-19 guaranteed precision even for some quite large numbers. For example:

```
fdiv $(( 2**63-1 )) 3
3074457345618258602.333333333333333333
```

- When the denominator is divisible by 10, or its multiples, then the decimal precision can be also increased, this time by the relevant number of zeroes inserted after the decimal point. In the following example, the truncation error is now less than 5E-22:

```
fdiv 5 30000
0.0001666666666666666667
```

We could do this zero shuffling just as well in binary by detecting divisibility by two but this is easier for demonstration purposes. In fact, that is just what the floating point arithmetic does, where the exponent represents 'the balance of the zeroes'. In general, the cleanest method (though not the fastest) is to carry out Euclid's greatest common divisor (GCD) algorithm and to eliminate (all) common multiples first. However, here it is not necessary, as they will be eliminated much faster by the integer divisions anyway. With our method of transferring denominator zeroes we in fact gain significant digits.

Note that `fdiv` is not like Bignum package using an unlimited number of computer words. Therefore, apart from the zeroes transfer trick, there are natural limits to the number of significant digits that can be produced, namely two 64 bit words. Still, that is quite a lot better than the 52 bits of the 'double precision' standard floating point mantissa.

Adding fixed point numbers

Without wishing to labour the point, we ought to also demonstrate that we can do arithmetic on these fixed point numbers. The next script, `fadd`, shows their addition. It is an instructive exercise in Bash textual manipulation for the unlikely purpose of arithmetic:

```
#!/bin/bash
# fadd takes two positive fixed point arguments
# each of the form whole.fract
# and returns a fixed point number addition result
# for example: fadd 9.876543 0.987 10.863543

PROGN=${0##*/}
MAXINT=9223372036854775807 # 2**63-1

# arguments verification
NUM1=${1:?$PROGN needs fnum first argument}
NUM2=${2:?$PROGN needs fnum second argument}
[[ $NUM1 =~ ^[+-]?[0-9]+[.][0-9]+$ ]] || { printf \
   "$PROGN error: first argument $NUM1 is not an fnum\n";
   exit 2
}
[[ $NUM2 =~ ^[+-]?[0-9]+[.][0-9]+$ ]] || { printf \
   "$PROGN error: second argument $NUM2 is not an fnum\n";
   exit 2
}

# parse both args into whole and fractional parts
ARR1=(${NUM1//./ })
ARR2=(${NUM2//./ })
F1=${ARR1[1]}
F2=${ARR2[1]}

# swap fractional parts so that $F1 is the longer
[[ ${#F1} -lt ${#F2} ]] && {
   FT=$F1; F1=$F2; F2=$FT
}
LF2=${#F2} #  save the shorter length

# strip trailing zeroes from the longer F1 while longer
while [[ ${#F1} -gt $LF2 ]]; do
```

```
[[ ${F1: -1:1} == 0 ]] && F1=${F1:0:-1} || break
done
LF1=${#F1} # finished chopping F1 now

# and/or pad the shorter F2 with rhs zeroes
# until of the same length
while [[ $LF1 -gt ${#F2} ]]; do F2=${F2}0; done

# do the adding
WHOLE=$(( ${ARR1[0]}+${ARR2[0]} ))
FRACT=$(( $F1+$F2 ))

# check for fractional overflow i.e. >= 1
[[ ${#FRACT} -gt $LF1 ]] && {
   let WHOLE+=${FRACT:0:1}
   FRACT=${FRACT:1}
}
expr $WHOLE.$FRACT
```

Project for the reader: implement a similar script `fsub`, that will carry out subtraction of these fixed point numbers. Hint: begin by copying this code (9)[6] and then just suitably modify it.

[6] https://github.com/liborty/fdiv/blob/master/fadd

Subshell Scheduling

As has been outlined before, running processes in the background with `&`, potentially in a different processor core, while we get on with the job on hand, is an important part of Bash. It always has been but lately, with multicore processors being the norm, it has become more topical. When you have thirty two or more cores sitting in your processor, most of them so to speak 'twiddling their thumbs', it behoves a responsible programmer to think of ways of spreading the jobs around.

Automatic multithreading

Many Bash applications just handle lots of files. As does, for example, the open-source github project TokenCrypt, (4)[7]. Many of the tasks depend on the file alone and so they can be run in parallel. Disk bottlenecks may limit the expected speedups but we will not worry about them here.

Here we shall see how to automate the spawning of background jobs within a Bash script. This technique is of general use and it can speed up many naturally parallel tasks, such as the processing of several files.

A better way of doing intricate multithreaded and asynchronous programming (and many other things besides) is Rust, of course. However, Bash remains by far the easiest for simple tasks.

[7]https://github.com/liborty/TokenCrypt

Rant about security and privacy

This project was born out of frustration with the state of encryption on the internet. Many of the offerings appear to be achieving only the objectives of obfuscation and insecurity. Amongst them even some well intentioned ones.

However, there is a very simple and effective alternative: XOR symmetric encryption. Symmetric means that the same key is used for both encryption and decryption, which much simplifies the key management. It also allows such useful operations as merging of keys for multi authorisations.

Furthermore, with the key of the same length as the message, this encryption is theoretically unbreakable, i.e. the strongest possible. It is implemented in TokenCrypt in an easily usable form.

There is one reasonable objection, that generating such long keys in effect doubles the amount of storage needed. This is mitigated by compressing the files first and some care is taken to do it effectively. So, for example, with 50% compression achieved, there is no storage overhead at all. In any case, disks are cheap and the security is worth it even with some overhead. On the plus side, the encryption and decryption are very fast, being so simple.

Encrypting many files faster

The tasks of compressing, encrypting, decompressing and decrypting all the files in a given directory are clearly I/O bound. They only depend on one file at a time, so can be performed in parallel, without communications between them.

The most suitable compression method is selected and applied for each file individually and also the key of the right length is generated on encryption or looked up on decryption. The code is general enough to take care of this. We will not dwell on the details here, they can be found in (4)[8], (5)[9], (6)[10], (7)[11].

Our aim is to fire off background jobs for longer files first and leave the shorter ones to be processed in the current shell in the end, to avoid the subshell overheads for them. This will minimise the overall script execution time. We also want simple code without too many tests and complicated 'if' statements.

First the script arguments and flags are processed and used to set up the correct input and output directories. Then the input files are sorted by their lengths:

```
# size of files to be encrypted in the subshells
BACKSZ=10000
# list sorted files in INDIR by size (in bytes)
ITEMS=$(ls -SL $INDIR)
# prepare to fire off subshells
BJ=\&
```

The last line above is a wondrous hack to set up the default textual command ending `$BJ` to `&`.
This will cause jobs over big files, followed by "$BJ", to be run in the background subshell, to start with.

What remains now is to process all the files in the order of their decreasing size and when the threshold `BACKSZ` is reached, permanently turn off the sub-shelling with `BJ=''`:

[8] https://github.com/liborty/TokenCrypt
[9] https://github.com/liborty/TokenCrypt/ncrpt
[10] https://github.com/liborty/TokenCrypt/expcrypt
[11] https://github.com/liborty/TokenCrypt/impcrypt

```
for FILEND in $ITEMS; do
   # reconstruct one full path/filename
   INFILE=$INDIR/$FILEND
   # only process a genuine file
   [ -f "$INFILE" ] || continue
   # turn off subshells when file size falls
   # below the threshold BACKSZ
   # once $BJ has been set to an empty string,
   # do not check filesizes any more
   [ -z "$BJ" ] || \
   [ $( stat -c%s $INFILE ) -ge $BACKSZ ] || BJ=''
   # the function that processes one file
   # either in the background or in this shell
   # depending on the value of $BJ
   processfile $INFILE $BJ
done
wait # for all processfiles to finish
```

Remember that Bash functions are executed in a sub shell when their invocation ends in `&`. When `$BJ` variable is programmatically unset, its value simply disappears from the program text and executions of `processfile` are thereafter done in the current shell.

Self modifying program, ha! (The beautitudes of EIJAS in Bash).

We use logical or `||` instead of `if`. It only executes the last command, unsetting `BJ=''`, once. When `$BJ` was still set to `&` but the size of the current `$INFILE` has fallen below the threshold `$BACKSZ`. The only check performed thereafter is the first part: `[ -z "$BJ" ]` is `$BJ` unset? (evaluates to true and exits `||` immediately). If there are lots of short files, it saves the relatively expensive `stat` system calls on all the rest of them. They are already sorted, so none of them will exceed the threshold any more.

Of course, in this simple example, we could have achieved the same effect by testing $BJ in an if statement and issuing two different explicit calls to `processfile`, as follows:

```
if [ -n "$BJ" ]; then
    processfile $INFILE &
else
    processfile $INFILE
fi
```

but we wanted to demonstrate the general opportunities in Bash programs for self-modification.

Hopefully, it can be seen that without too much extra difficulty, we could set up a fully fledged scheduler with a managed queue of jobs with priorities and a dispatcher for their execution. However, mostly people use a 'gunshot' technique of firing off almost anything and everything with & and letting the operating system take care of the details of their executions.

Nonetheless, as pointed out above, we need to be careful about disrupting any sequential inputs/outputs and generally anything that has to be done in specific order.

Conclusion

Possibly the most exciting thing about Bash is just how quickly we were able to move with this book ‘from zero to hero’ . That means gain confidence and to start implementing practical projects that would actually be quite challenging and long winded to do in most modern programming languages.

Some modern languages, of which Rust is a prime example, produce rather the opposite experience of going ‘from hero to zero’. That is to say, from a confident programmer who has previously mastered a number of languages, to a human wreck feeling very much at the deep end, with snail pace productivity. Don’t get me wrong, I love Rust really and will probably write my next book about it.

Another anecdote in a similar vein used to be told about Java when that one first came out: “have you heard of this wonderful new language, in which you can print ‘hello world’ in mere five pages of code”?

Behind this hilarity, there is a serious point to be made. Programming is getting more and more complex. Following current trends, we might lose ownership of it altogether. It may become a reserve for the last few dorks alive, who can still (just about) cut it and paste it, in their own narrow field, using proprietary applications libraries that someone long ago dead has bequeathed onto them. That would be a sad, dystopian world.

Is such a bleak future really inevitable? Perhaps this book has managed to inspire you towards (re)discovering the joys of ‘owning’ programming and sharing its fruits. Then its mission was a success.

Links References

(1) Gnu Foundation, Bash Reference Manual,
https://www.gnu.org/software/bash/manual/bash.html.

(2) Bash Scripting Cheatsheet,
https://devhints.io/bash.

(3) `fdiv` Fixed Point Arithmetic in Shell Scripts, github repository,
https://github.com/liborty/fdiv.

(4) TokenCrypt Encrypting Archiver, github repository,
https://github.com/liborty/TokenCrypt.

(5) `ncrpt`, bash script in (4),
https://github.com/liborty/TokenCrypt/blob/master/ncrpt.

(6) `expcrypt`, bash script in (4),
https://github.com/liborty/TokenCrypt/blob/master/expcrypt

(7) `impcrypt`, bash script in (4),
https://github.com/liborty/TokenCrypt/blob/master/impcrypt

(8) Encryption and E-Democracy, blog,
https://oldmill.cz/2020-06-10-crypt.html

(9) `fadd` bash script to add fixed point numbers, in (3),
https://github.com/liborty/fdiv/blob/master/fadd