Principles of Programming

The GNU/Linux or UNIX Command-line and Programming in Bash

Technical University of Denmark

LICENSE

Table of Contents

1 About this Document

1.1 Overview

This document is a course material for introducing people to using the UNIX and GNU/Linux (hereafter written as *NIX) command-line. The *NIX command line is supported by a "shell", of which there are many.

The document describes the Bourne Again Shell, usually noted by its executable filename `bash`, which is the most common on *NIX systems.

The document frequently mentions the Linux computational systems available at the Technical University of Denmark as one of the main purposes of this document is to assist students of the university accessing those systems.

1.2 Structure

The document moves forward with a focus on practical usage, and brings in new concepts along the way. It is conversational in style.

There is an additional technical stream, presented progressively though the document. It is given major section Titles of "Breakout: (something)". These introduce the relationship between the command-line bash (the command-line interpreter) and the kernel (the core of the *NIX operating system). This technical stream attempts to provide a fundamental framework on which to hang the understanding of the command-line and its functionality.

This document starts by introducing you to fundamental "at the command line" things that you need to know. At this stage you are thinking from your ***own perspective***. As the "Breakout" sections continue you begin to look from the ***perspective of bash*** itself. Finally, at the end of the breakout section, you should be able to think from the ***perspective of the kernel***. Attaining all 3 of these perspectives is a key objective of this document. Thus, the course objective, apart from enabling you to use and program in bash, is really that you can:

- think about what is happening during the execution of a single bash command from ***any of these perspectives*** and describe exactly what is happening.

1.3 Programming

The vast majority of this document is concerned with understanding in detail what happens when a *single command* is issued to bash.

Right at the end of the document the "programming" issue is addressed. The programming parts of bash are very similar to other imperative languages, with conditional, switching and looping statements.

2 Introduction

This document is about learning how to use a *NIX based system for computational work. It is a crash course, which means that:

- you *will* have more questions than answers during the course, which is expected
- you will often be confused *during* the course, but it is hoped that you will develop methods to resolve your confusion. The primary methods proposed are Read The Manual (RTM) and consult the community (in that order)

2.1 Community

The most effective method of self-help is to find a community with which one can interact to learn and contribute. See Appendix A.

2.2 History: UNIX, GNU/Linux and Hackers

GNU/Linux is a derivative of UNIX (of which there are many), which are derivatives of MULTICS. There are two main branches of UNIX, BSD (Berkeley Software Distribution) and System V (System five). As mentioned, all GNU/Linux and UNIX type systems are written *NIX in this document.

*NIX were written by Hackers. This term is eternally misused by the media. Hackers are those who exhibit playfully cleverness. The people who break into other people's computers and cause chaos are called Crackers. For a pleasant and eminently readable history of *NIX and the hackers, see:

http://www.catb.org/~esr/writings/homesteading/hacker-history/

I cannot recommend this historical account highly enough. Please take the 60 mins to read it in entirety.

Hereafter GNU/Linux is written merely as Linux. In truth, Linux is a kernel, with which you can do almost nothing. It is the GNU part that makes it useful. (Topic for research).

2.3 Advice / Warning

If you're wishing to use Linux then you are almost certainly in for a new experience in computing. If all you have ever used are graphical user interfaces (GUI's), then that new experience is going to leave your 'mouse hand' lost and wondering what to do. The keyboard is your assistant, your mind is the master, and the mouse is hiding (cat present).

2.4 The End Result

*NIX is a steep learning curve for a person who is only used to GUI's.

After this course, and a few months of practice and frustration, but with the help of the community, you will be able to achieve results on Linux using larger datasets than you can on Windows. You will learn that you need to tell Linux *exactly* what you want it to do, because it will do exactly what you ask it to do with all of your privileges.

You will have precise control over what you want done, and the flexibility to actually do that.

3 Putty: Accessing *NIX via ssh from Windows

To access the Linux modelling systems you need to use the ssh (secure shell) protocol. A nice tool for doing that from Windows is called Putty.

Once you have authenticated (logged in) via ssh you will then be presented with a command-line (the shell).

3.1 Download and Install

Download from:

http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Download: A Windows installer for everything except PuTTYtel

Run the installer: Next, ..., Install

(Its an open source tool. It will not install nasty toolbars on your brower or do anything else nasty; Next, Next …).

3.2 Installation and Configuration

Start it (Start menu): Putty, Putty

1. Add to Host Name '<username>@<the-server>' (e.g username@login2.gbar.dtu.dk)
2. Click on Window and change the Row to 40 (or more than 24)
3. Expand 'Window' to 'Colors' and select 'Use system colors' (this gives you a white background)
4. Back to 'Session'
5. In 'Saved Sessions' enter a name for this connection (e.g hpc)
6. Click 'Save'
7. Click 'Open'
8. Enter password
9. ... do stuff on the Linux server ...
10. Type 'exit' to close the connection

Note that Putty will probably be your access mechanism for Linux. Get to know Putty. Customise it to be as comfortable as possible.

3.3 The Bash Command-Line

Welcome to your interface to the Linux system. Why don't we use a GUI? Because it takes up lots of memory that we would rather use for the actual work! A GUI would not help you much at all, because the majority of software that you will be using on the Linux modelling systems, particularly for parallel processing, is better used on the command-line.

Your mouse is going to be next to useless here, so get used to loving your keyboard.

(If you never learned to touch type … ?)

3.4 Security and Stability

You may feel uncomfortable, but you need not feel afraid. Whatever you do, you cannot kill the system or damage anyone else's things. *NIX has always been a multi-user operating system, and was build from the beginning with a “separation of powers” between different user accounts, their files and processes.

4 Help and Basic Commands

4.1 Manual Pages

Every general command-line tool that you will use has a help page, telling you what it does and how to use its command-line switches and arguments to tell it to do exactly what you want.

The 'help' pages in *NIX speak are called Manual Pages and can be accessed with the 'man' command. Try:

```
man man
```

That is, show me the manual page for the 'man' command which shows manual pages. Look at the manual page in structure.

Each manual page comes with sections, and the sections are fairly consistent across Manual Pages (generally known just as man pages).

NAME, SYNOPSIS, DESCRIPTION, ... ,SEE ALSO, ...

Man pages are 'terse'. i.e they say what they mean in the least number of words possible. Reading man pages requires patience, but with a little experience becomes much easier, and you may end up being happy not having to read a book rather than 5 pages.

To start understanding the man pages that you will use, do:

```
man 1 intro
```

The '1' in the above command tells the man command which section of the manual you wish to look at. Section 1 is for all the command-line tools, in which you are mostly interested.

Each of the sections of the manual pages are described in the manual page for the man command. One refers to a manual page by the page name and the manual section. E.g. man(1) refers to the ‘man’ command from section 1 of the manaual pages, or fork(2) refers to the ‘fork’ man page from section 2 of the manual.

4.1.1 The Crossroads

At this point most people will have looked at `man man` and thought “how the **** is that going to help me?”.

This is the fork in the road:

1. “I cannot be bothered to 'waste' my time reading this complicated stuff and will ask other people to give me the solution to the specific problem that I to need solve. I am under pressure and do not believe that investing my time in understanding this antiquated insanity is to my, or anyone else's, benefit.”
2. “Crickey, not (Organic Chemistry 101, Calculus and Linear Algebra 101, … <insert your nightmare course here>) again!” Sadly, yes. Know that the people who wrote these pages are mostly the ***authors*** of the tool; i.e they **are** the expert. They almost always list their contact addresses, and respond to **well formed questions**.

This is a new world, and this is its initiation. Battle on! Like inorganic chemistry or calculus, the investment of time will be intensely rewarded, though it may seem light years away at the outset.

Read The Manual (**RTM**) and then ask questions.

Say, command 'foo' is interesting:

- `man foo`
- `foo -h`
- `foo –-help`

Nothing there? Go to *your community.* The web is the 'catch all' (last resort) and will invariable require much time finding the exact answer to the exact question that you posed. Pre-investing the time into reading and learning to read the man pages gets you directly to the canonical source of information for *NIX command-line utilities. It is this information upon which all the useful answers that you see on the web are based.

4.2 Essential Commands

This section introduces some 'commands' that you will certainly need. In all cases, you should **RTM** for the command to understand it better. Please read the man page about a command before asking others about how to use the command. Failure to understand a man page is part of learning – ask for help. Failure to *read* it is disrespectful.

4.2.1 Listing the contents of directories:

`ls`

`ls -l` (list in (long) detail)

`ls -a` (show everything; normally any file/dir that starts with the character '.' is not shown)

`ls -al` (combination of the two above: everything in long listing)

4.2.2 Changing directories:

```
cd (changes to your home directory)
cd .. (changes to the parent directory)
cd – (changes to the last directory that you were in)
cd /tmp  (change to the /tmp directory)
```

Relative vs. Absolute path names

'.' is where you are

'..' is the directory above

'/' is the root (start of all file paths)

'/' is the directory separator in *NIX ('\' for windows).

Examples

You are the 'abcd' user, and you are in your home directory `/home/abcd` :

Path	**Explanation**
`foo/bar.txt`	The `bar.txt` file in the `foo` directory under where you are
`./foo/bar.txt`	The explicit (relative path) version of the above
`/home/abcd/foo/bar.txt`	The absolute path to the same file
`~/foo/bar.txt`	The '~' character expands to the full path to the home directory.
`$HOME/foo/bar.txt`	$HOME also expands to the full path to the home directory. We meet environment variables, or which this is one, later.
`/tmp/baz.pdf`	Absolute path to `/tmp/baz.pdf`
`../../tmp/baz.pdf`	The relative path to `baz.pdf` from the home dir that was assumed above.

4.2.3 Using 'Tab' for path completion

While typing in a file path on the command-line you can use the Tab key to ask the shell to 'finish' the path:

- if you type 'Tab' and there is only one possible completion, the completion is done
- if you type 'Tab' and nothing happens, then there are more than one possible completions, so type 'Tab' again (quickly) and the shell will show you all of the possible options. Add more characters by typing and with another 'Tab' the shell will do its best to complete the path from the additional letters that you have typed.

4.2.4 Making directories

```
mkdir <path>
```

4.2.5 Removing directories

```
rmdir <path>
```

will complain if the directory is not empty. See 'rm' for a forceful way to remove entire directory trees (careful!).

4.2.6 Renaming files (or directories)

Renaming in *NIX is known as 'moving'.

```
mv <old-name> <new-name>
```

The names are paths, thus you can do:

```
mv foo/bar.txt /tmp
```

and it will move the file `foo/bar.txt` into the `/tmp` directory, becoming `/tmp/bar.txt`

4.2.7 Copying files

```
cp <original> <new-file>
```

Recursion

In `cp` and `rm` there is a 'recursive' flag, meaning do this for **everything** under each argument. The flag is `-r`. Read the man page!!!

```
cp -r foo /tmp
```

Copy all of the `foo` subdirectory into `/tmp.`

4.2.8 Deleting (Removing) files/directories

```
rm <file> ...
rm -r <directory> ...
```

Be very careful with your `<path>.` The only way to recover from an error with rm(1) is backup!

4.2.9 Changing the 'mode' of a file

In *NIX you can change files to protect them from being changed or deleted, or you can even make them executable.

Make read-only

```
chmod a-w <file> ...
```

Make executable

```
chmod a+x <file> ...
```

See chmod(1).

You can, of course, do this to entire directory trees with the 'recursive' switch (for `chmod` it is a capital -R, because -r already has a meaning for `chmod`):

```
chmod -R a-w <dir> ...
```

Doing these things may not see important now, but will likely be useful later.

4.3 Shell Parsing: the problem of spaces (and other special characters)

This topic is *extremely important*. Read carefully and practice at a command-line to ensure that you understand what is being said.

The shell's job is to run *processes* for you. That generally means an executable file and some arguments to be given to it. Thus, the shell needs to be able to work out what the arguments are as opposed to the command itself.

The standard shells, by default, treat any *space* or *tab* as something that separates **words**. The command-line is chopped up into words. The first word on the command-line is the command (to be executed), and the other words are its *arguments*.

This creates potential problems. Say you have a file 'my holiday.txt' and you want to edit it with an editor 'vi' (see below on Text File Editing). You might type this:

vi my holiday.txt

The shell will see that and chop it up into **three** words: vi, my, holiday.txt

Thus, it runs 'vi' and gives it two arguments, 'my' and 'holiday.txt'. But you wanted to edit the single file 'my holiday.txt'.

This problem will also raise its ugly head later when you are using bash as a programming language. In a 'normal' programming language to assign 'hello' to the variable 's' you would write:

s = “hello”

In bash, that will not do an assignment. Bash takes that as 3 words and tries to work out what to do with them (can it find a program called 's'?). To assign the value you need to ***remove the spaces***:

s=”hello”

There are three basic ways to solve the 'spaces in file names' problem: dont use them, escaping and quoting.

There is no way to solve the 'assignment' problem. Just remove the spaces.

4.3.1 Escaping

If you place a '\' character before **any** character you are telling the shell to treat the next character as an exact literal: no special meaning is to be applied to it. Thus:

```
vi my\ holiday.txt
```

will edit the file because the ' ' (space character) has had it's normal meaning (i.e it separates words) removed.

If you are using 'Tab' for auto-completion, you will see that during the auto-completion, the shell actually puts the \ in there for you !!!

4.3.2 Quoting

You can group many characters into one **word** by placing single, or double quotes around the characters. Note the quote marks.

```
vi 'my holiday.txt'
```

or

```
vi “my holiday.txt”
```

In the above two examples there is no difference because the string “my holiday.txt” contains only one type of 'special' character, the space. But there are other special characters, the most common for the shell you are using are each of:

```
! $ # & ( ) * ?
```

For the single quoted string 'my holiday.txt', the single quoting means: treat the entire string as just plain text – no special meaning *anywhere*. For the double quoted string “my holiday.txt” it tells the shell to look in there for **some other** special characters. This is very useful in advanced use of the shell which we meet at the end.

In summary, use single quotes for any string that you want to be grouped into a single **word** (argument) and all will be good.

4.3.3 Exercise for the Curious

Do the following, exactly, on a command-line and watch what is printed out for the filename completion at the end:

```
mkdir foo
cd foo
touch 'a$!#&*(){}?.rubbish'
ls a<Tab>
```

The completion should look something like: `a\$\!\#\&\(\)\{\}\?.rubbish`

The shell is telling you that these symbols have a special meaning to it.

Then, clean up:

```
cd ..
rm -rf ./foo
```

5 Breakout: Command-line to Kernel and Back

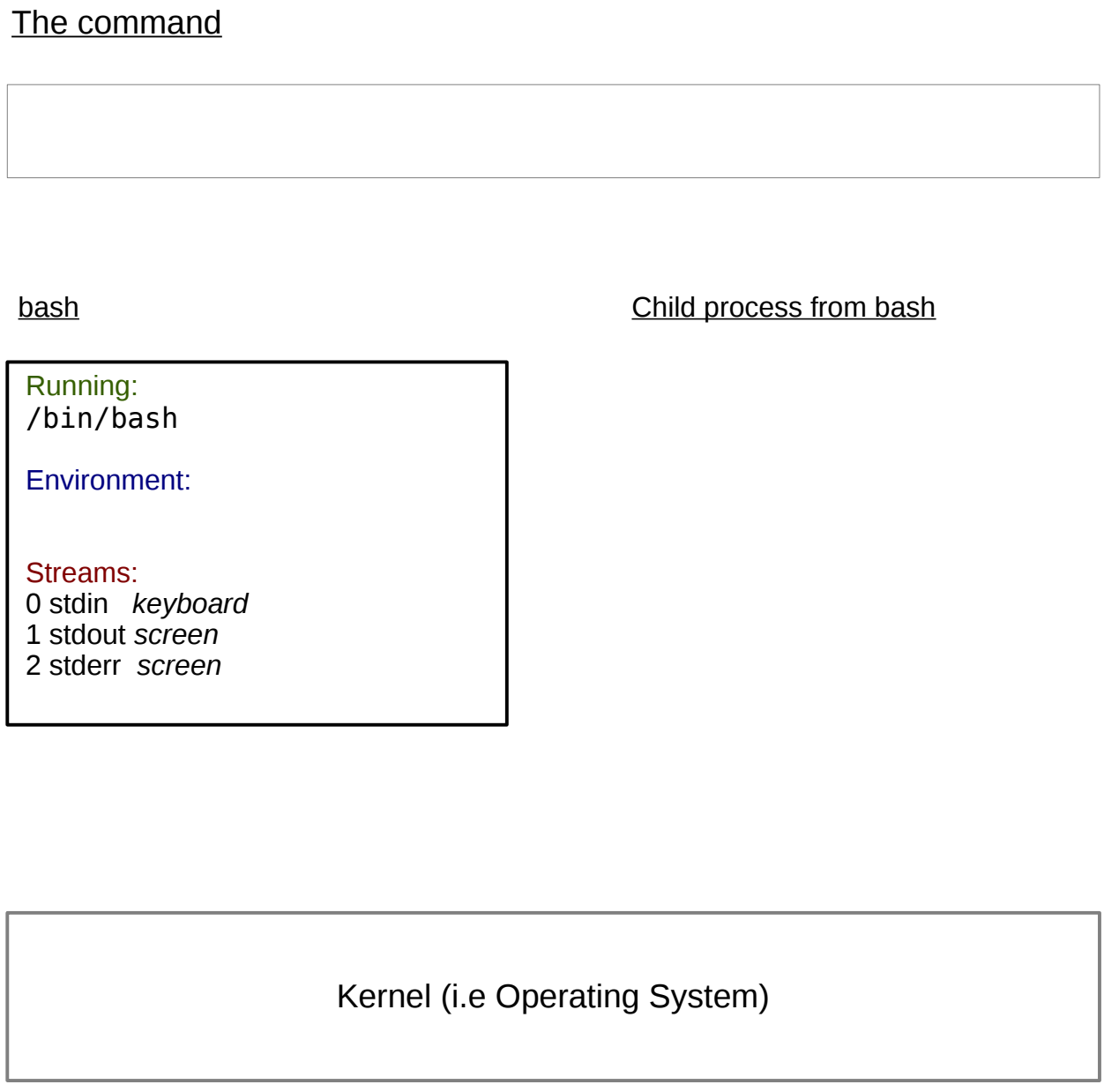
Each Breakout chapter introduces the relationship between the command-line, the bash command-line interpreter, and how it uses the kernel.

The kernel is the core of the *NIX operating system. It provides all services to processes. A process is a running program. For example, bash is a program, and many users on the same computer may be running many copies of bash. Each of those running copies is a process.

The breakout sections are a pictorial representation of what is happening, with colour coding. There are three levels, the command, processes (bash and its child) and the kernel.

We have not met Streams or Environment yet, but we will in time. For now, just ignore that.

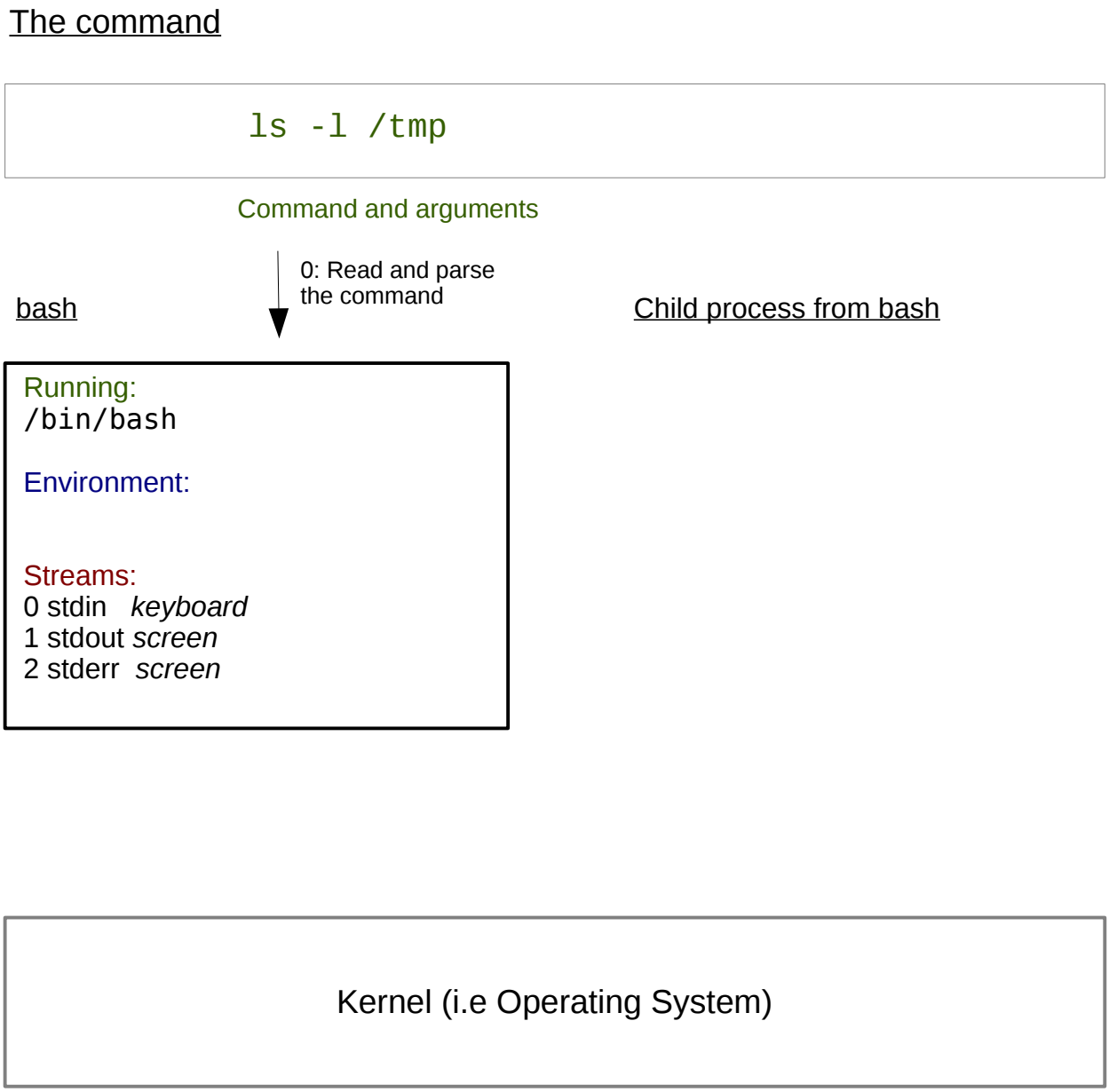
Here is the blank structure which we will be filling in as we progress.



5.1 Parsing

The user has entered a command and pressed the Return key. Bash parses the command. If it does not parse (i.e it is rubbish), bash reports an error.

Nothing has happened yet, apart from bash agreeing that it can understand (parse) the command.

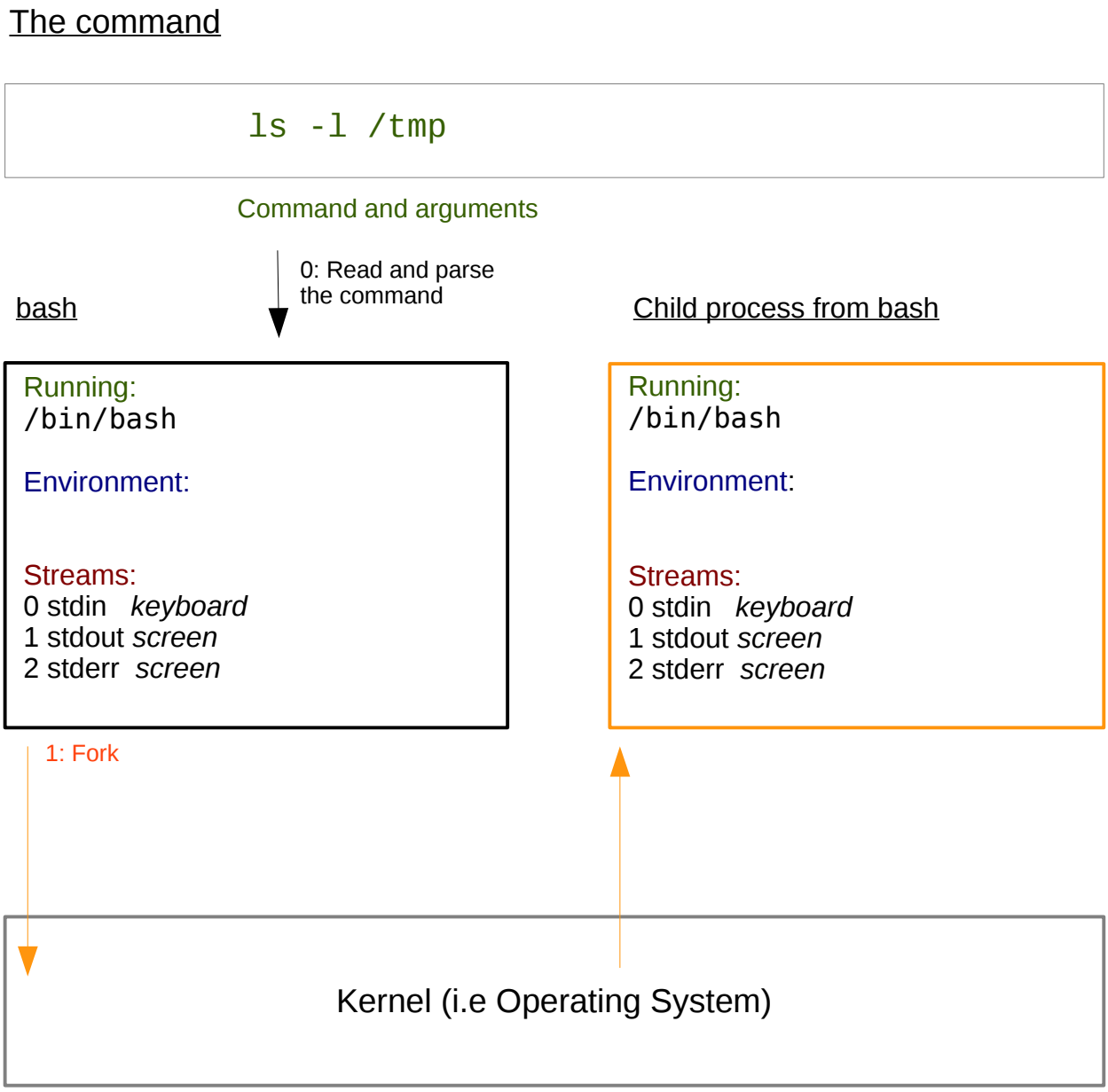


5.2 Fork

Parsing has succeeded. We have three words ('ls', '-l', and '/tmp') and by searching the directories listed in the PATH variable (a part of the environment which we meet soon) bash has found an executable 'ls' at '/bin/ls'. Thus, the command can be executed.

First, bash calls the kernel and asks for a fork. This creates an *exact copy* of the bash process that asked for the fork, as it was at the time of asking, as a *new process*. This is the **only** way to create a new process in *NIX. There are other variants of fork(2), e.g clone(2), but they essentially do the same thing of creating a new process which is a copy of the process asking for a new process.

The “original” bash is the parent process and the “new” bash is the child process, over which the parent has control until the exec call (which we meet next). This “control” is explained at the end of all Breakout chapters. It is subtle and beautiful, but for now just assume that the parent process is controlling the child process.



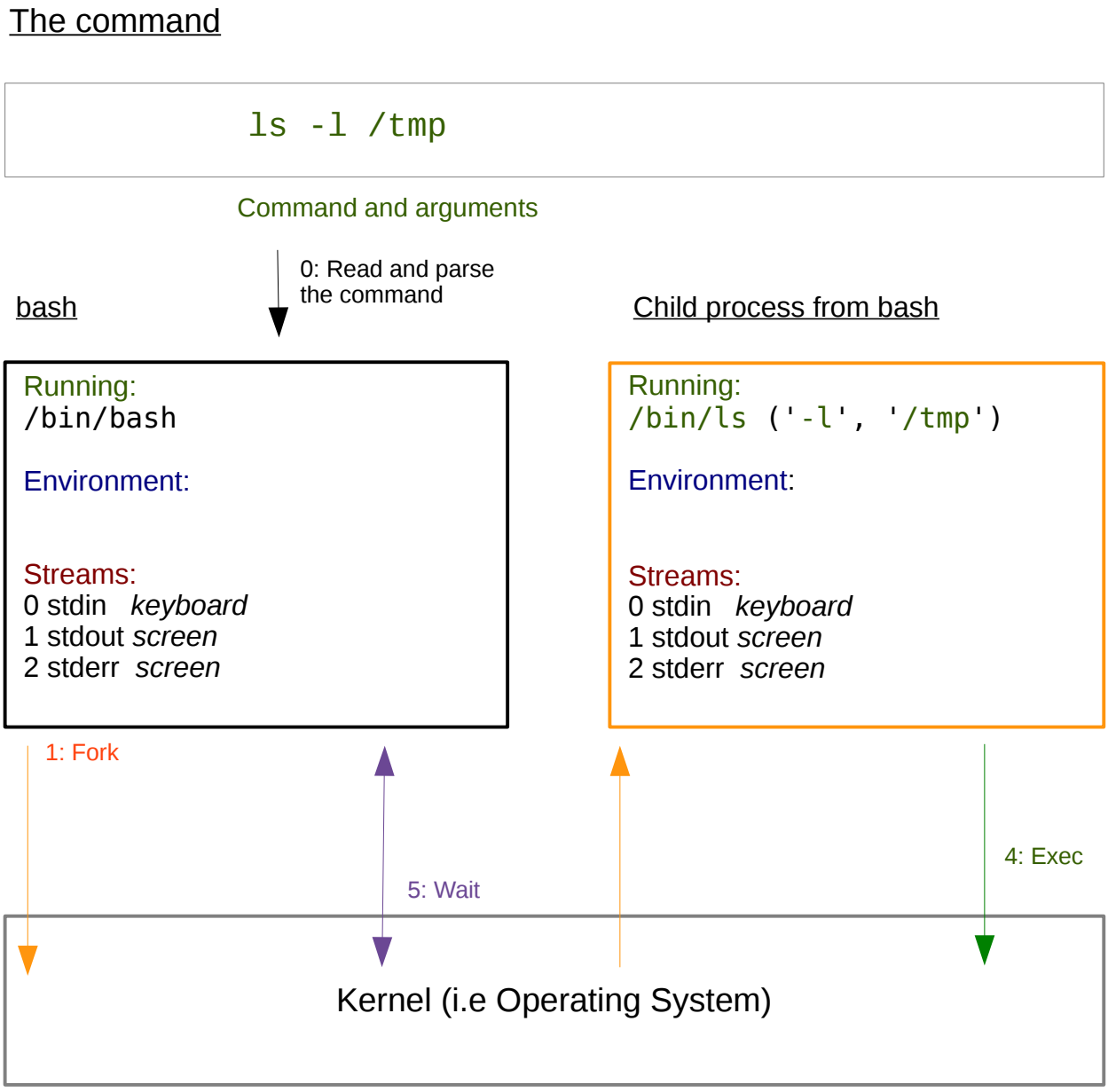
5.3 Exec and Wait

The child process (being controlled by the parent) then calls the kernel to exec with '/bin/ls' as the new program to write over the top of itself (!!!) with the arguments that were read from the command-line ('-l', and '/tmp').

Thus, the child is now running '/bin/ls' and the parent loses control. The parent calls the kernel to wait for the child and is stuck (blocked) until the kernel responds to the wait call.

When the child finishes (exits), the wait call returns and bash prints a prompt for the user.

This is what happens 90+% of the time that people are using bash. Note the numbering and follow the sequence: parse, fork & wait, exec. Parts 2 and 3 get filled in at later Breakouts.



6 Text File Editing

6.1 Nano

Nano is a trivial text editor. It works exactly like notepad/wordpad from Microsoft. Arrow keys to move around and backspace/delete or typing. See the bottom of the screen. You use Control characters for saving and quitting.

6.2 Vi / Vim

Vi is a stalwart of the *NIX world. It is installed by default on every *NIX, even under a minimal install.

I could wax lyrical about the wonders of vi for hours. But, the essential understanding is that it is a *modal* editor. To understand that, consider nano/notepad etc.. In these editors whenever you type a letter it appears on the screen. Vi has two modes, input mode (when letters appear on the screen) and command mode, when letters do other things, like move you around, delete things, start an input mode etc.. To move out of input mode and into command mode you press Esc.

Here is the bare minimum of the commands in command mode:

Command	**Description**
h, j, k, l	Move left, down, up, right (respectively)
W, w, B, b	Move forward: one word (ignoring funny characters), one word (honoring funny characters), and back one word (ignoring and honoring) (respectively)
(,), {, }	Move one sentence back, forward, or one paragraph, back, foreward
I, i, A, a, O, o	Insert at the start of line, or here. Append to the end of line ,or here. Start a new line above where you are, or below.
d<movement>, dd	Delete text to wherever the move takes you, delete this line.
y<movement>, yy	Copy to wherever the move takes you. Copy this line.
P,p	Paste before where you are, paste after where you are.
U,u	Undo all changes on this line, undo the last change (repeat for more undo)
X, x	Delete the character before the cursor, under the cursor
:e <path>	Edit the file <path>
:e #	Edit the file you were previously editing
:n	Edit the next file given on the command line
:w	Save this file
:q	Quit
:wq	Save then quit
:q!	Quit and don't save anything. Just quit.
:w!	Save and don't warn me, just do it.

If you are going to be using *NIX for years, then learn something better than nano. Vi is an excellent choice.

The 'Vim' from the title is 'Vi iMproved'. An excellent piece of work. Vim is now the preferred standard vi.

6.3 Emacs

Emacs is the other great *NIX editor. It is collosal. You can run a web-server with it if you are so inclined. The author has no experience with it.

Historical Note:

There are two camps in the *NIX world when it comes to text editors. The vi camp, and the Emacs camp. They love to tell each other that their editor is better. Its a bit like the Windows camp vs. the Mac camp. Religious war. :-)

7 Command-line Editing

7.1 Simple version

Use the Up and Down arrows to browse through your previous commands. Use Left and Right to move and type or Backspace/Delete.

7.2 The Vi version

```
set -o vi
```

Now your command line editing is in vi mode. Movement and editing as per vi (j, k for up, down etc.). Note you start in 'insert' mode after typing return (on the previous command). Thus, if you want to move around (k,j,w etc.) you start with 'Esc' to get into command mode.

8 Cut / Paste: The Putty mechanism

When you high-light text with the mouse it is *automatically* copied. Click the right mouse button to 'paste' it. The reason for this becomes clearer when we deal with process control and signals. The summary is that in windows 'Ctrl+c' copies the highlighted text, in *NIX it forces the currently executing process to be terminated. Thus, Putty removes the temptation to type 'Ctrl+c'.

Be careful with pasting. If you paste on the command-line, the shell will just start doing whatever the paste told it to do.

9 Streams

A stream is a sequence of bytes, followed eventually by an 'end-of-file' marker (meaning that the stream is at its end). A file is a stream. But, a stream can exist just in memory as two processes communicate with each other, and thus avoids the HEAVY overhead of using a file. A stream could be seen as an ephemeral file (of the other way around; a file is a stream stored on a disk ;-).

In *NIX each and EVERY process gets given 3 streams by default:

Stream	Usage	Number	Explanation
stdin	<	0	The standard input. That is where a program will read from, by default.
stdout	>	1	The standard output. That is where a program will write data, by default.
stderr	2>	2	The standard error. This is a where a program will write informational messages, or error notifications, by default.

9.1 Basic Stream Usage

When `cmd` is used below, it represents *any* simple command (e.g `ls -l /tmp`)

If you have a command that prints out information, and you want to save it, you just tell the shell to redirect its stdout to a file:

```
cmd > a-file.txt
```

If a command reads the information on which it will work from the stdin, then you can redirect its stdin from a file:

```
cmd < b-file.txt
```

If you just want to see what a command prints out, but want to save any *messages* it prints into a file, you can redirect its stderr:

```
cmd 2> c-file.txt
```

If you use a double greater than (>>) then the information printed will be *appended* to the file, if it already exists.

Of course, all of the above can be combined:

```
cmd < a-file.txt > b-file.txt 2>> c-file.log
```

Here we have re-directed all standard streams, but with stderr writing in append mode (>>). Note that the names of the files are purely for example. They can be anything you want (remember the 'space character' problem! You need to quote your file names if you wish to use spaces in them; Hint: don't use spaces in file names).

Finally, if you want to redirect both stderr and stdout to the same file, there is a way of doing that too. The *order* in which you write this is important. Here is how to say: redirect stdout to a-file.txt and also redirect stderr to wherever stdout is going:

```
cmd > a-file.txt 2>&1
```

The &1 means 'wherever 1 is going' (1 is stdout, remember?). Similarly, but written a different way: redirect stderr to a-file.txt and send stdout to wherever stderr is going:

```
cmd 2> a-file.txt >&2
```

If we get the order wrong, things will not be what you expect. Say, we take the first example and write it in the other order:

```
cmd 2>&1 > a-file.txt
```

Think *carefully* about this. It says, redirect stderr to whever stdout is going. At this point, stdout is going to the terminal (window), so stderr is sent there too (it was already going there, but that's what you said to do). Then stdout is redirected to a-file.txt.

This is classic *NIX. It does **exactly** what you asked it to do, even if that is not what you meant.

All of these 'tricks' are mildly useful. But where it really *rocks* is a pipeline.

9.2 Pipelines

Some command do things and print their results to stdout and other commands read the things on which they should work from stdin.

Wouldn't it be nice to be able to connect the first with the second?

One can do:

```
cmd-A > file.txt
cmd-B < file.txt
rm file.txt
```

All command-line tools use at least some of these streams, as it creates a standard manner in which you can control them, and additionally allows them to easily work together.

The above is lots of typing and we're wasting time with the file system by creating `file.txt`.

A pipeline is a connection of at least two commands together separated by the | symbol (called 'pipe' in *NIX language, for obvious reasons).

What the pipe symbol does is to tell the shell that you want the stdout of the command before the pipe symbol connected to the stdin of the one after it. I.e `cmd-B` will read what `cmd-A` printed.

To make that pipeline:

```
cmd-A | cmd-B
```

And thats it. You can make these pipelines as long as you want: `cmd-A | cmd-B | cmd-C | ...`

9.2.1 Basic Example

Say you want to know the number of files in a directory which end in .txt?

You can:

```
ls *.txt
```

and then count them all. But, there is a command that counts characters, words or lines for you called 'wc' (word count). If we use `ls -1 *.txt` then ls will print out each file name on *a separate line* (that is what the -1 switch does). Then we can use 'wc -l' (count lines). And so, to count these files we just use:

```
ls -1 *.txt | wc -l
```

9.2.2 Convert Input to Arguments: xargs

Say we want to know the number of lines in each of the text files in the directory, and we want that sorted by the number of lines? I.e we get a table of numbers in the first column and file names in the second, sorted by the number of lines.

The problem here is that we want the output of the 'ls' command to become arguments for the wc command (read its man page). There is a special tool for that called `xargs`. Here's a first part of the solution:

```
ls -1 *.txt | xargs wc -l
```

Now wc will count the lines in each *file* that ls printed out because xargs is reading what ls printed out and then putting them on the command-line as arguments for wc.

Then, to sort them, we just use the sort command (with -n for sort numerically):

```
ls -1 *.txt | xargs wc -l | sort -n
```

Now you have a list of the number of lines in each *.txt file in the directory sorted by the number of lines.

9.3 Common and Practical Examples

Say we want to count the number of files below a directory:

```
find <path to directory> -type f | wc -l
```

Say we want to find the number of filenames that match a pattern (*.txt) in their name below a directory:

```
find <path> \( -type f -a -name '*.txt' \) | wc -l
```

There are a number of subtle things happening here. The escapes (\) are there because the parenthesis ‘(‘ is a special character to the shell (as your know by now), and the ‘-a’ argument is telling find to ‘and’ these two conditions of ‘-type f’ (it is a file) and “-name ‘*.txt’” the file name matches that pattern (glob). Curious? RTM: find(1), bash(1) and glob(3). The really interesting question is “why is the file pattern ‘*.txt’ in single quotes?”.

9.4 Regular Expressions: grep and sed

Every command-line environment from any useful operating system comes with a way to match file names. You may know of *.* from DOS, meaning match anything of any length followed by a dot followed by anything of any length. The file name pattern matching rules are generally known as Globs. We have those for Bash too (RTM). Actually, DOS took this from *NIX (as is often the case).

*NIX introduced a far more powerful text pattern matching mechanism called Regular Expressions. Regular Expressions are commonly shortened to 'regex' or 'regexp'. With the additional power comes a little complexity. However, when you learn how to use just 20% of that you immediately get two of the most powerful tools: grep and sed.

9.4.1 grep

The best complete introduction to grep (and regular expressions) is maintained by GNU:

http://www.gnu.org/software/grep/manual/

The best way to learn this is to play with it on the command-line. You come up with an expression and test it using echo.

```
echo some-text | grep test-expression
```

Read the guide above. Here are some examples:

Match just simple characters:

```
prompt$ echo abcde | grep bc
abcde
prompt$ echo abcde | grep cb
prompt$
```

Note that cb does not occur in abcde, thus nothing is printed. Order matters.

```
prompt$ echo abcde | grep B
prompt$
```

So does case!

Using character classes:

```
prompt$ echo abcde 12345 | grep '[:alpha:]'
abcde 12345
prompt$ echo abcde 12345 | grep '[:digit:]'
abcde 12345
```

And now with anchors:

```
prompt$ echo abcde 12345 | grep '[:alpha:]$'
prompt$ echo abcde 12345 | grep '^[:digit:]'
```

In these cases we asked for letters (alphabetics) and the end of line, and then numbers (digits) at the start of the line, which we do not have. Thus there are no matches and nothing is printed by grep.

9.4.2 Sed

Sed (Stream Editor) allows one to use regular expressions to match text and then do stuff with it. Most commonly one will replace. So, sed is the the great 'search and replace' mechanism.

Say, you just want to insert a + symbol at the beginning of each line in a file.

```
sed 's/^/+/g' < file > tmp ; mv tmp file
```

Sed is natively built in to vi too. So to do the above in vi (whilst in command mode):

```
:%s/^/+/g
```

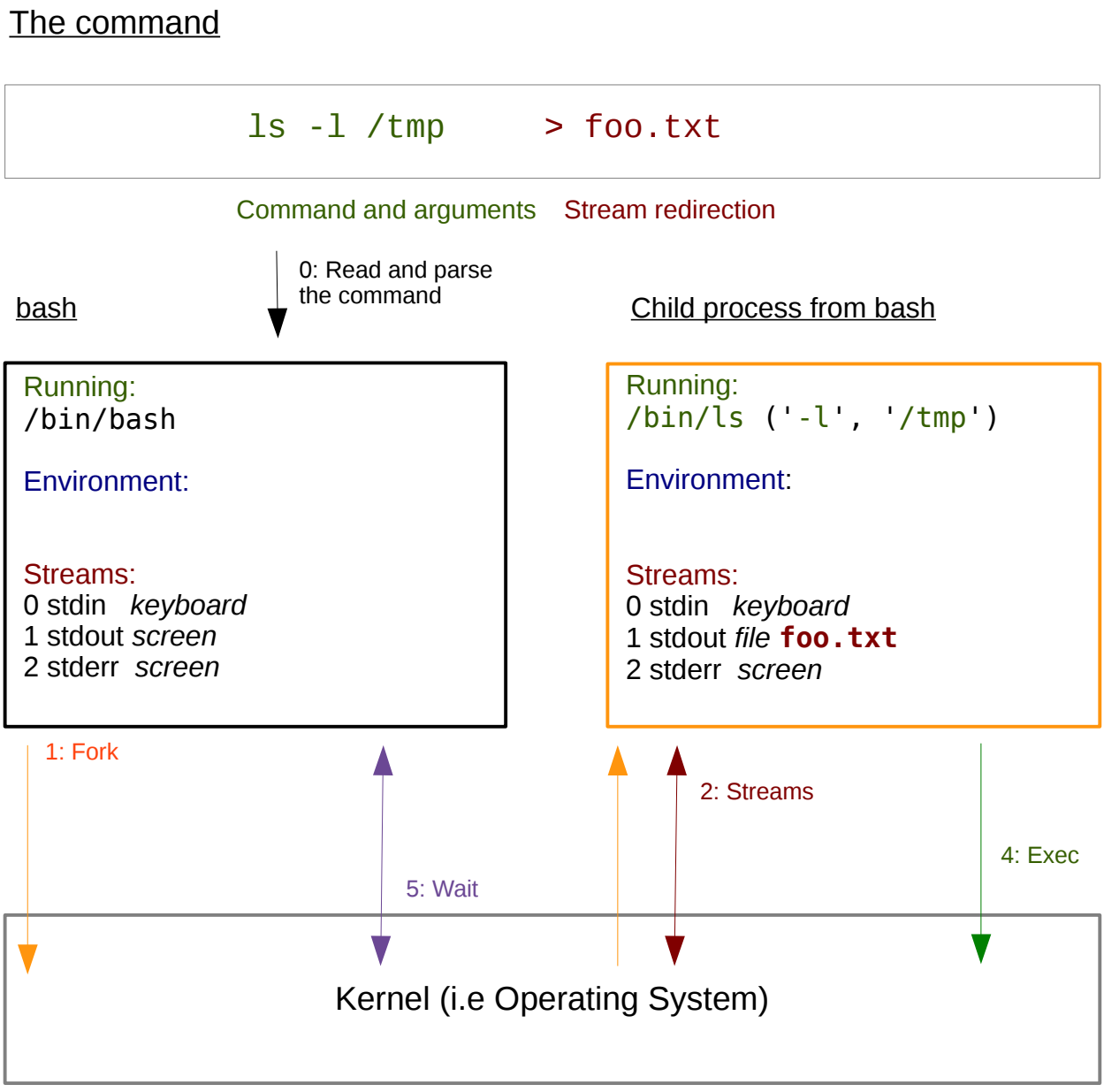
The % above means 'whole of file'. The rest is exactly the same sed argument from above.

10 Breakout: Stream Redirection

This is exactly the same as the last breakout diagram, except that we are now redirecting a stream (stdout).

After the fork the child's stdout is changed ***before*** the exec. The old stdout is closed, and then a new stdout is opened as the `foo.txt` file for writing.

Thus, when the exec is done, and 'ls' starts writing to stdout (as it always does) then that goes to the file instead of the screen.



11 The Environment

In a *NIX system every process has access to a dictionary of name / value pairs. A process can read what is already there and use that information, or change what is there, delete what is there or create new name / value pairs.

Many 'programs' use the environment as an alternative, or in addition to the command-line switches that we have seen.

Over time, as some information is equally useful to many programs, these values have been preferentially read from the environment. This is because the environment is trivial for a bunch of programs to share. More on this in Process handling.

Some examples of environment name/value pairs:

`USER=abcd` (your user name)

`HOME=/home/abcd` (where your home directory is)

One can see how these values can be useful for many programs.

By convention, all environment variables which are shared by programs are written in UPPER CASE.

To see all of the environment variables that are set in your shell use the following command:

```
env
```

An example output might be:

```
TERM=xterm
SHELL=/bin/bash
USER=hmco
USERNAME=hmco
PATH=/home/hmco/bin:/usr/local/bin:/usr/sbin:/usr/bin:/bin:
PWD=/home/hmco
LANG=en_US.UTF-8
HOME=/home/hmco
OLDPWD=/tmp
```

11.1 “Special” variables

There is a small collection of very useful environment variables that the shell itself maintains for you. See 'Special Parameters' in the bash man page. Some are listed below. Note that the man page is **the** reference.

Variable	**Meaning / Comment**
$?	The exit status of the last command executed. One uses this all the time.
$@	The arguments to a script, all together, as words.
$0, $1, …	The individual parameters from the command line. 0 is the script itself, 1 is the first argument etc.
$#	The number of arguments, not including the script itself. E.g if $# is 3 then $3 will have a value but $4 will not.
$$	The process id of the current process.

12 Breakout: Environment

This is exactly the same as the previous breakout case, with redirection, but now the command is preceded by a change to one environment variable. (There may be many!).

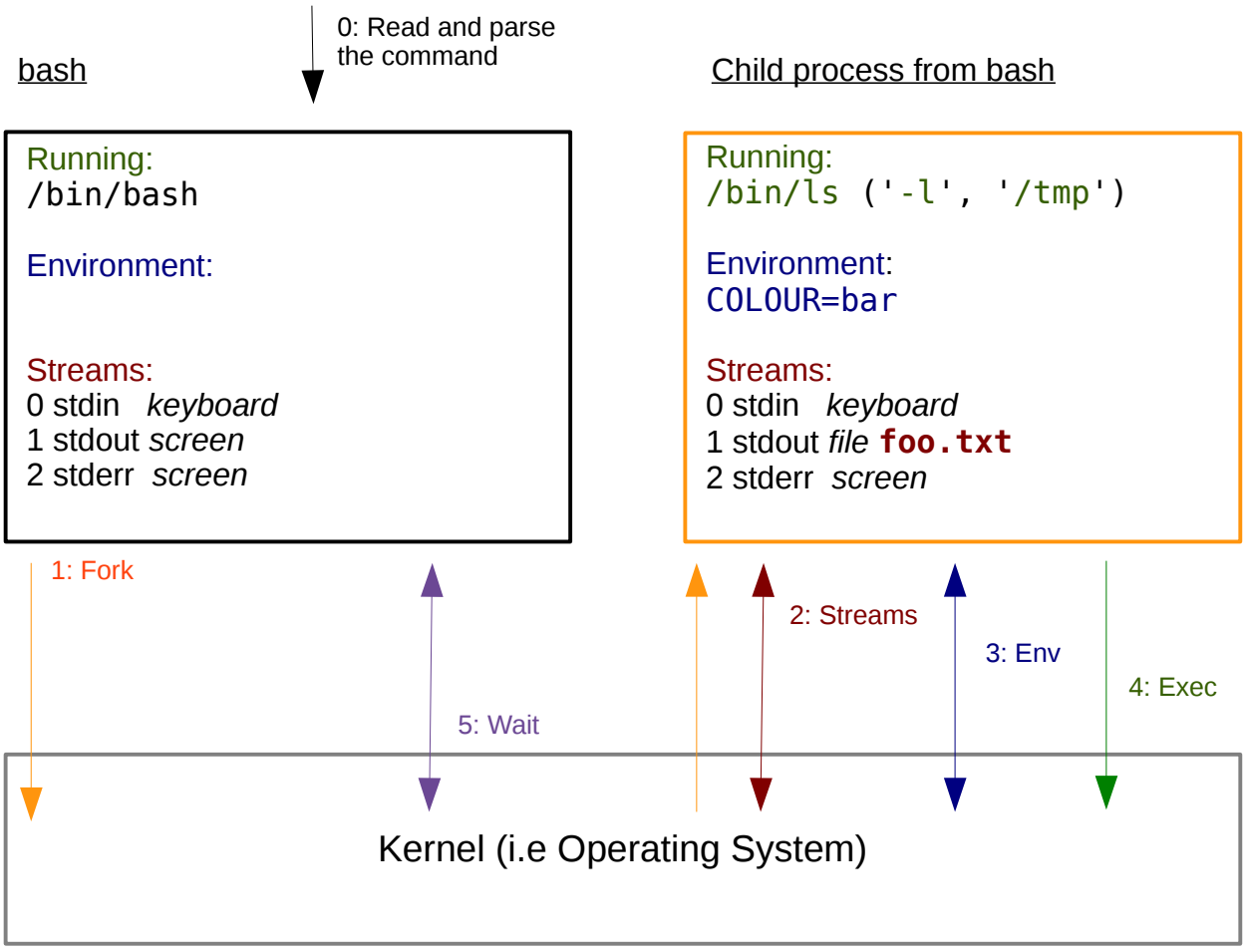
So now, after the fork and stream changes, but ***before*** the exec, the parent (which is still in control of the child) changes the environment for the child. Note that this has no effect on the parent. Only the child process' environment is modified. Then the exec occurs and the parent waits as per usual.

Note that prefixing a command with environment changes is relatively rare. It is included here to illustrate the process of what is actually happening.

The command

```
COLOUR=bar ls -l /tmp      > foo.txt
```

Environment Command and arguments Stream redirection



13 Processes and Signals

13.1 Signals

*NIX systems have a way of you sending a message to a process at any time you wish. When you do this the operating system itself acts on your behalf to stop the process doing what its doing and asking it to deal with the 'telegram' (signal).

There are many different signals, and how they are used has become standardised by convention over time. Here we talk about only the most commonly used. Why they are important will become increasingly clear in the Processes section.

The core signals are:

Signal Abbreviation	**Meaning**
INT	Interrupt: tell the process to clean up and stop as soon as it has cleaned up
KILL	Kill: terminate the process. It gets no chance to clean up and is just annihilated
USR1	User Configurable 1: Send a signal whose message is entirely dependent on the program to which you send it (read the man page).
USR2	Same as USR1, but the meaning of the signal may be different. (RTM)
HUP	Hang Up: This comes from the old modem days. It means that the user has just disconnected from the system, and thus, all of the things that they were doing should be informed. By default, each process will treat it as an INT signal (clean up and stop)

Sending a signal is easy, but you have to know which process you wish to send it to. For that, you need the 'ps' or 'jobs' commands.

The operating system will refuse to send signals to processes that you don't own (i.e you did not start them).

To send an INT signal to the currently running (foreground) process, type Ctrl-c. This means 'copy' in Windows land, but **stop** what you are doing **now** in *NIX land. (Careful)

13.2 Listing Processes

13.2.1 ps

`ps` is like 'ls' but for processes rather then files. Read the man page!

`ps` (tell me about my processes)

`ps -ef` (tell me everything about all processes)

Read the output. The number in the left most column is the process id (number). This uniquely identifies a process.

13.2.2 Jobs

Listing jobs that your shell is running in the background:

```
jobs
```

They are denoted by a percent sign (%). Thus [1] (as printed by 'jobs') is referred to as %1 when used on the command line to signal a process.

Send a Signal:

kill -(SIGNAL NAME) process_id

e.g

```
kill -INT 43231
```

or with a 'job':

`kill -INT %2` (send an INT signal to job two)

13.3 Permanently Changing your Shell itself

Of course the shell will let you completely overwrite it. This is rarely a good idea, and almost certain to annoy you unless you know exactly what you are doing. Instead of running just 'cmd' you prefix it with 'exec':

```
exec cmd
```

You will not get another prompt unless the command you chose to exec was a shell (there are many of them, of course). This is essentially how you can use another shell.

Go back to the “Breakout” sections and think about this ‘exec’. Can you understand what you are doing and why, if the ‘cmd’ that you chose was not a shell, you will not get a command prompt again?

13.4 Parallelisation and Process Control

After all that mostly techie and seemingly useless information we come to something directly useful. Running a process in the background.

Recall that when you run a command (e.g ls) the shell will fork/exec it and then WAIT for it to finish. What if the command will take lots of time and you want to do other things in the meantime?

Then you instruct the shell to run it, but do it in the *background* and to give you your prompt back immediately:

```
cmd &
```

But, there is a challenge here too. If you do not instruct the background task to do anything specific with its standard streams then they will continue to be associated with your 'window' and thus what is happening in the background gets mixed up with the other things that you are doing.

So, a smarter move is to redirect the stdout and stderr of 'cmd' to a file or two which you can watch or look at when you are ready:

```
cmd > out-file.txt 2> msg-file.txt &
```

Now you can go about your other business whilst 'cmd' goes about its.

And, you can do this again and again:

```
cmd > out-file1.txt 2> msg-file1.txt &
cmd > out-file2.txt 2> msg-file2.txt &
cmd > out-file3.txt 2> msg-file3.txt &
cmd > out-file4.txt 2> msg-file4.txt &
```

Now you have 4 processes running in the 'background'.

As mentioned in the Signals section, you can use 'jobs' to request the shell to tell you about what is happening in the background.

13.5 Hang Up: Protecting your Long Time Processes

As noted in the Signals section, when you disconnect from the computer, every process that you have created and is still running will get told about your disconnection (receive a HUP signal) and is asked to clean up and quit. This is not what you want when you're running a 3 day simulation. But, as with all other things, there is a standard solution to this well known problem.

You need to stop the HUP signal from being delivered to the process(es)! Enter 'nohup'.

All you need to do is to prefix your command with 'nohup'.

So, considering the above listed 4 background processes, we just get:

```
nohup cmd > out-file1.txt 2> msg-file1.txt &
nohup cmd > out-file2.txt 2> msg-file2.txt &
nohup cmd > out-file3.txt 2> msg-file3.txt &
nohup cmd > out-file4.txt 2> msg-file4.txt &
```

Note that nohup expects (and checks) the you have redirected stdout and stderr. That is because when you disconnect you cannot receive any information sent to the 'window' that is now gone (you are disconnecting). Thus, nohup wants you to redirect your stdout and stderr, and will do this for you automatically if you have not.

This behaviour could be restated as '*NIX will not throw away information unless you explicitly ask it to'.

13.6 Deliberately throwing away information

*NIX has a universal garbage bin called `/dev/null`. It is a file to which you can write eternally and whatever you write there will be immediately discarded. At first glance this seems silly. But, it has several standard uses.

Say, you want to run a program in the background, under nohup, and you don't care what it prints out. Maybe you know that it will create a file and write its data there. Thus you could say.

```
nohup cmd > /dev/null 2> log-file.txt &
```

What 'cmd' prints out (to stdout) is deliberately discarded (sent to /dev/null). We are possibly interested in the data from stderr and have sent that to 'log-file.txt'.

Another example may be that you have a program that prints information to stdout but you are only interested in timing the execution, but uninterested in the result. Enter the 'time' command:

```
time cmd > /dev/null
```

13.7 Being Nice: running a process deliberately slowly

Each process has a priority. Linux uses this to decide what should run next, when there are more processes than CPUs (which is almost always the case). The *lower* the priority of a process, the more likely it is to get more time actually running.

One can ask for a process to be run slower than the default scheduling. This is done using the 'nice' command (RTM).

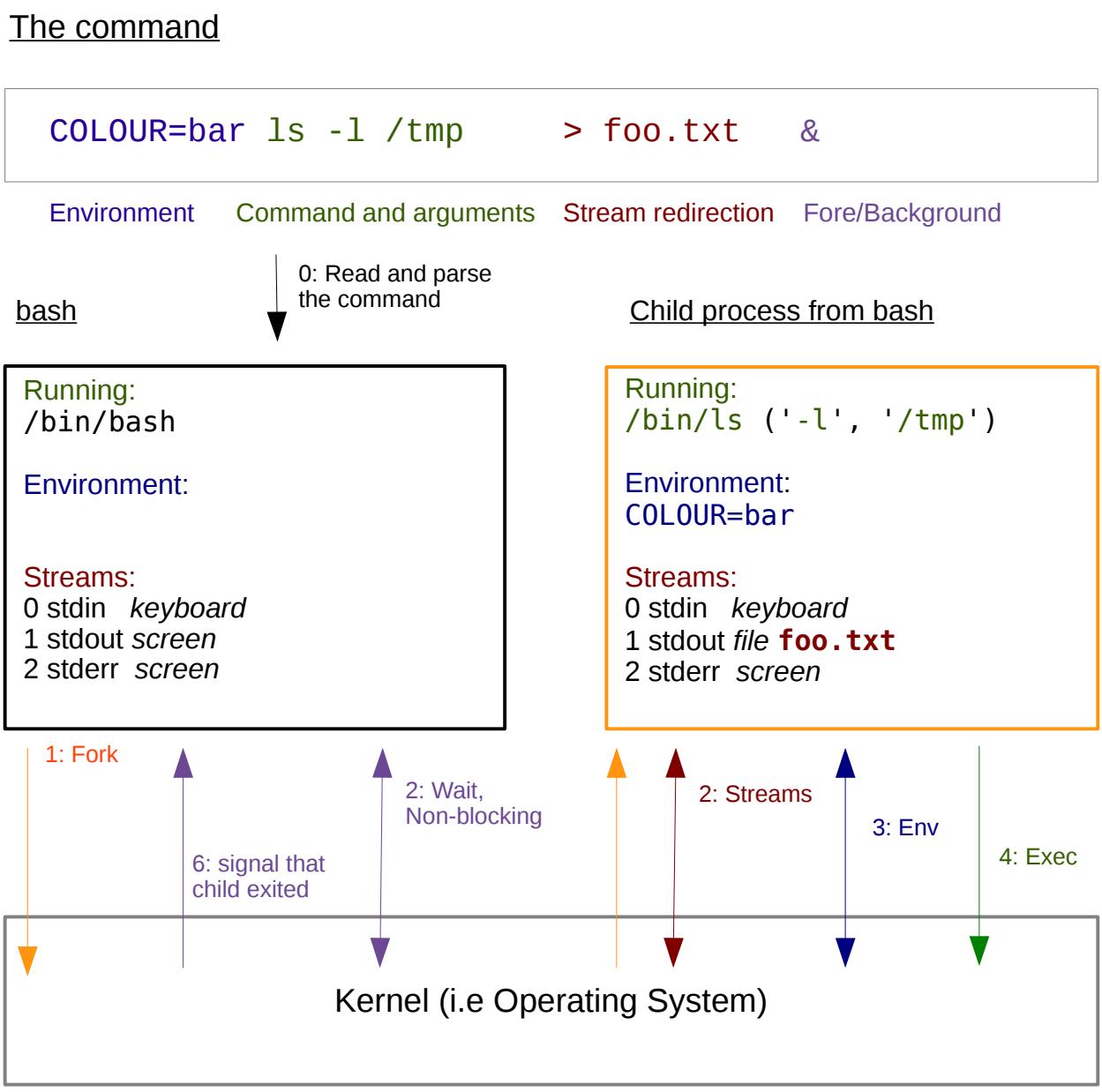
This is a nice thing to do, when you have some task that you want done that may take say an hour, but you dont really need it for at least 3 hours. So, just up the nice value and it will have less impact on other people's use of the system. For example,

```
nice -n 10 run_my_calculation
```

14 Breakout: Foreground / Background

Again, this is the same as the previous breakout case, but now we request the command to be executed in the background.

All things are exactly the same, but when the parent (bash) calls wait it uses a non-blocking variant. Thus, the kernel immediately returns the wait call rather than waiting for the child process to complete. This allows the parent to do other things. It asks the kernel to signal it when the child exits so that it can report this to the user.



14.1 “control” by the parent; or the Wonder of *NIX

There is no “influence over the child” by the parent. However, that is a good way to begin to learn about the topic and has been used for that purpose.

The truth is that when the parent calls fork it is done on a conditional branch in the code:

```
if fork
then
        ( child code )
else
        ( parent code )
```

This is a *very clever* trick and takes some thinking to understand. When bash calls the fork operation the kernel takes over. It creates an almost exact copy of the parent process as a new process called the child. If this succeeds, the kernel then starts the parent in (parent code) and the child in (child code). That is, the fork function returns in two *different processes*. Only the kernel can do this sort of magic trickery as it controls all of the computer hardware (CPU, memory etc.).

The child knows everything that the parent did (i.e what environment variables to change, which streams to modify etc.) so the child can *and does* make those changes *to itself*. Again, the kernel is doing most of this (i.e please kernel close my stdout, and please open this file for writing so it becomes the new stdout).

The child also knows which command is meant to be executed (and the arguments) so it asks the kernel to perform the exec. When the exec happens the old child code and knowledge is completely lost, but the environment and the streams are preserved. The child *becomes* the new executable. i.e it was a copy of bash executing down the child path, but at the exec call it is no longer bash, but the new executable (/bin/ls).

In the parent, which is executing down the other part of the conditional branch, it does the wait call, either blocking (normal operation, the child takes the 'foreground') or non-blocking (the child runs in the 'background' allowing the parent to do other things).

So, now you have the truth and wonder of *NIX. A fork copies environment and streams and an exec leaves them unmodified but starts a new compiled executable.

For 'full disclosure', the trick in piping (stdout of one command become stdin of the next) is also a kernel service. This itself shows the brilliance of the idea of streams. Because we abstract all flows of data as “just a sequence of bytes” (a stream) we can connect everything together with this mechanism: a keyboard, a text terminal, a file, a network socket.

Its all just a stream from the point of view of a process (program). The kernel (actually, often the device drivers which are part of the kernel) looks after all the dirty details and lets the process just get on with what it wants to do.

15 Programs, Scripts and File Types

15.1 Determining the “Type” of a File

In Windows, the extension of a file name indicates the way that that file should be handled. *NIX has a completely different mechanism that places no restrictions on how you name your files. There is no need for a file extension at all.

As there are no naming restrictions, the name, or extension, of a file cannot tell you anything certain. To determine what a file is we need to *look at its contents*. There is, of course, a tool that will do this for you. It is the 'file' command. (As an aside, the ‘file’ command is based on the shared library ‘libmagic’ – and that demonstrates again the playfullness of the ‘hackers’ who created *NIX. The shared library which can determine file types based on their contents is ‘magic’).

For example, we can ask what type of file the 'ls' program is:

```
file /bin/ls
```

```
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.32,
stripped
```

The above example tells us that 'ls' is a compiled program (and lots more technical information).

Another example:

```
file /usr/bin/ipython
```

```
/usr/bin/ipython: a /usr/bin/python script text executable
```

Here we see that ipython is a 'script text executable' meaning that it is not compiled and can be read by mortal eyes. It also tells us which language interpreter will be used to 'run' it (/usr/bin/python). i.e this is a script written in the Python language.

15.2 Compiled Programs vs. Scripts

15.2.1 A little background and history

The power at the heart of every computer is a Central Processing Unit, or CPU (possibly many of them in modern computers). Each type of CPU knows how to do a fixed number of things. These things are generally extremely low level operations like 'copy the data from this place in memory to that place over there', or 'add these two numbers'. Each type of operation is identified by some number and potentially takes some arguments (like the locations of the memory from which to copy and then write). In essence, these instructions are a series of binary strings. Not letters, just a big string of Ones and Zeros.

Early on in the use of computers the programmers had to actually write the programs in all of these Ones and Zeros. This is an arcane art.

Following that, computer scientists invented languages which were human readable and were designed to express what you wished a computer to do, but in a readable human form. But, they needed a tool to convert the human readable version into the computer readable version. This type of tool is called a compiler. Its job is to perform this conversion from human readable into computer readable. The collection of files which the compiler reads to produce the final computer

readable binary file are called source files, for they are the source (or beginning) of the final 'program'.

Once this 'compilation' is done, one can then run the program as many times as one wants, and when you run it the computer can instantly read and understand what it is being asked to do.

However, if you want to change the program you must change the source and then do the compilation all over again before you can run the changed version to 'see' the change.

15.2.2 Interpreters

The compiler is a translator between some programming language and computer (CPU) speak. Its end result is not instructing the computer, but building a file that contains these instructions.

There is another strategy and that is to translate the source code into instructions for the computer and give those instructions to the computer immediately. This is called an interpreter.

This has a good side and a bad side. The good side is that you don't have to compile the program, you can just 'run' it directly (i.e the interpreter reads it and directly instructs the CPU). The bad side is that the instructions that the CPU is given are not necessarily as optimised (fast) as they could be, and you have to run the interpreter every time you want to run the program. Interpreting is a rather heavy task, computationally speaking. So, we get convenience but you suffer in speed.

In recent years, computers have achieved two key advantages that now make the use of interpreters more attractive: computers are faster, and they have more memory.

Interpreters have been around since the beginning of human readable computer languages, but they have become more practical in recent decades due to the aforementioned increases in computing hardware (thanks, chip/chipset designers!).

15.3 Making Scripts

If you want to make a script (i.e a program that runs directly from the human readable programming language) you, of course, need to understand the language and write the 'code', but you also need to tell *NIX which interpreter it should use to run it. In Windows this is done with the filename extension. In *NIX you use the **first line** of the file to tell the shell exactly which interpreter to use.

To do that you use the following format on the first line of the file:

```
#!<path to the interpreter>
```

Examples:

```
#!/bin/bash
#!/usr/bin/python
```

To run the file, however, you need to tell the system that it is something that we wish to be *able to be run*. To do this you set the executable flag for the file (see chmod above). For a script called 'my-script' you would do:

```
chmod a+x my-script
```

15.3.1 Shell Scripts: BASH

BASH (often just written bash) is the command-line tool that you are using all the time. It is also a programming language! Any command that you type and run on the command-line can also be put in a file (script) so that you can do it again and again. Indeed, *NIX systems are full of bash type scripts as it is such a useful language for amalgamating/coordinating various related

processes/commands.

Read the manual page for bash, and consult the community.

15.3.2 Python scripts

Just like above, and you need to make sure that you set the first line to tell the system which interpreter to use:

```
#!/usr/bin/python
```

15.3.3 Perl, R, Ruby, …

There are many interpreted languages. Same deal. Set the first line to refer to the correct interpreter.

16 Useful Commands Index

Here is a list of generally useful commands in *NIX (the exact name may vary a tiny bit across all the *NIX's, but are correct for the RedHat/CentOS/Fedora-Core GNU/Linux distributions).

In all cases you should *read the manual page* if the command looks useful.

Command	Example	Very Brief Description
cat	cat a b > c	Concatenate. Print out the contents of a file, or many files and possibly save them in a single merged file.
cd	cd /tmp	Change directory.
chmod	chmod a+x foo.sh	Change permissions on files. Example adds execution.
cp	cp -a model backup	Copy a file or directory tree
cut	cut -d ';' -f 1 foo	Extract certain columnar data from a file/stream
dd	dd if=/dev/zero of=foo.dat bs=1024 count=10	Direct disk copy. Copy blocks to/from devices/files. The example creates a file of exactly 10 KB of binary zeros.
df	df –B G	File system usage. How much space is used/available on a/all file systems.
du	du -sh .	Tell me how much space a file or complete directory hierarchy is using.
echo	echo “hello, world.”	Print out the arguments given.
file	file foo	Tell me what sort of data is in a file
find	find . -type f	Search a directory tree for matching files/directories.
grep	grep foo file.txt	Search for a pattern and print out matches(*)
head	head -n 15 foo	Print out the first part of a file
less	ls -R | less	A pager. Hold the data so I can see it all, rather than scrolling off screen. Less uses vi style controls.
ls	ls -a	List information about files and directories
mkdir	mkdir bar	Create a directory
mv	mv foo bar	Rename (move) a file or directory
nice	nice -n 10 my_prog	Run a program with less 'importance'
rm	rm foo	Delete (remove) a file.
rmdir	rmdir bar	Remove (delete) and empty directory
sed	sed 's/ /\t/g'	Stream editor: perform automated editing of text (*)
sort	sort -rn foo	Sort data. Dictionary order, numeric, in reverse etc.
tail	tail foo.log	Print out the last part of a file
tar	tar xzvf foo.tgz	Create / Extract / List (from) tar archives. Warning: NEVER create a tar archive with an absolute path name. E.g tar cf /tmp/foo.tar /usr/local is WRONG. Instead, go there and use the relative path: cd /usr ; tar cf foo.tar ./local
tee	tail -f foo | tee bar	Create a T fitting in a pipe. i.e both print out what is

		coming in *and* save it to a file (bar).
which	which ls	Tell me where the program is that you would run.
who	who	Tell me who else is logged in.
xargs	find . | xargs ls -ld	Convert what is read, per line, and make it an argument to a command.
zip / unzip	unzip foo.zip	Create / Extract / List ZIP archives

(*) use a common language for the generic matching of text strings, called a Regular Experssion. This is commonly abbreviated as 'regex'. Regex's are another extremely powerful archane art.

17 Programming in Bash

17.1 About

This section provides a quick introduction to the basics of bash shell programming. Herein 'the shell' means the Bourne Again SHell (bash).

A reader should already be confident using bash as an interactive shell and be comfortable with stream redirection and piping, process control, quoting, exit status, etc..

Note that bash has many features and its manual page is ***the authoritative reference***. This document only describes the more commonly used elements of its capabilities as used in simple shell programming.

17.2 Environment and Variables

Bash provides a dictionary of key / value pairs known as environment variables. Variables can be created, modified or removed at any time. Please note, as declared way back in the 'space problem' section, there are **no spaces** between the *variable name* and the *equal to sign* (=), or the equal to sign and the *value to be assigned* (i=0 not i = 0):

```
# Create and/or modify with =
cost_AUD='$1.09'
i=0
verb=push
# Remove with unset (note that 'verb' is still set)
unset cost i
```

The value of a variable is accessed by prefixing the key (name) with '$'

```
echo $verb
echo ${verb}ing
echo $verbing
```

The curly braces are used to delimit the key from following text. The above will print:

```
push
pushing
```

The last line is the empty string (and a new-line added by echo) because there is no variable 'verbing' set. Note: you can ask echo to not add a new-line. See the man page, and beware to use `/bin/echo` rather than just `echo`. One is an independent program, the other is provided by bash (also known as a ‘builtin’).

17.2.1 Export and sub-shells

Recall that *NIX uses a process tree. Every running process has a parent, and processes can create child processes. The shell is no different and allows you to create sub-shells.

The shell limits what its sub-shells receive from its environment. Only variables that have been marked for 'export' have their values copied to the sub-shell, and these are copies so nothing that the sub-shell does can affect the variables in the parent.

Sub-shells are created by surrounding the command sequence by parentheses:

```
( cd /tmp ; ls -1F )
dir=/var
export dir
( cd $dir ; ls -1F )
```

17.2.2 Command-line Arguments

When a shell script is invoked (begins running) it receives the arguments from the command-line which invoked it. $0 has the value of the path to the script itself. $1, $2 … contain the first, second, … command-line arguments. The collection of arguments after the path to the script ($1, $2, …) can be collectively accessed as a white-space separated string via $*.

17.3 Command-line substitution

We have seen how bash can substitute the value of environment variables into a command line before it is executed.

```
echo $HOME
```

becomes

```
echo /home/abcd
```

and then echo is executed and it just prints its arguments out.

Bash can also substitute the output of a *command* into another command line! The command to substitute is contained within `$(` and `)`. For example:

```
wc -l $(find . -name \*.txt)
```

will execute the find command (search for all files that match *.txt under the current directory) and then those file names are put onto the `wc -l` command line. The end result is a line count for each file. (Note the problem of spaces again – if a file is called 'my holiday.txt' then wc will try to open 'my' and 'holiday.txt')

This is a very powerful feature, though it is not commonly used.

17.4 Conditional Branching

The shell has two basic mechanisms for branching:

- if/then/else
- case

17.4.1 if then else

The if/then/else structure chooses a branch of execution based on executing a command and assessing its ***exit status***. The `test` command is most commonly used to assess conditions. See the man page for test's capabilities. The shell provides 'syntactic sugar' for the test command so that it looks nice. Both of the following do exactly the same thing (*are the same thing*), that is check that the contents of the dir variable refers to a directory:

```
if test -d $dir
then
      # do something
fi
if [ -d $dir ]
then
      # do something
fi
```

The branching structure must start with an 'if' and can be extended as needed with repeated 'else if' (elif) conditions, and a optional final catch all case 'else':

```
if [ 0 -eq $num ]
then
      #do something
elif [ 1 -eq $num ]
```

```
then
      # do something
elif [ 2 -eq $num ]
then
      # do something
else
      # do something
fi
```

17.4.2 case

The case branching mechanism matches strings with wildcard (glob style) support, the same that is used for matching file paths. In its simplest use it is a one way branching mechanism, like if/then/else. However, it also supports conditional and non-conditional cascading. These more advanced uses are not covered here (`man bash`).

case can be used to perform command-line switch matching:

```
case $arg in
'-h'|'-?'|'–help') # give help message
      ;;
*) # glob type wildcard catch all (like else)
      ;;
esac
```

17.5 Looping

Looping is the repeated execution of a sequence of instructions (commands) – the loop body. Note that a loop body may contain any valid sequence of commands, including other loops (thus called sub-loops).

17.5.1 for

The for loop runs the loop body for each word in a list of words, in the order that they are given in the list, with a place holder variable being given the value of the current word.

```
for foo in bar baz buz
do
      echo $foo
done
```

This would print out each of the three words bar, baz and buz on separate lines as the foo variable is given each new word value.

There is a second form of 'for' which has a completely different focus and is analagous to the for loop in the group of C-like programming languages. See the bash manual page.

17.5.2 while

The while loop has a conditional expression and a loop body. The conditional expression is evaluated, and if its exit status is zero the body is executed, if not the while loop is terminated. When the loop body completes, the conditional is re-evalutated and the process continues (re-execute the loop or end the loop).

Thus:

```
while [ 1 ]
do
     # something
done
```

will continue to execute forever (as test '1' is true – exits with status 0).

To execute a loop a fixed number of times:

```
i=0
max=512
while [ $i -lt $max ]  # -lt tests 'less than' (see test(1))
do
      # something
      # increment loop counter
      i=$((i+1))
done
```

The i=$((…)) line shows the mechanism for numerical operations in bash. RTM.

A common idiom is reading lines from a file:

```
cat a-file | while read line
do
      # something with $line
done
```

Rather than piping `a-file` you can redirect stdin. Thus, the following is equivalent but more efficient as the 'cat' program is not run:

```
while read line
do
      # something with $line
done < a-file
```

17.6 Functions

The shell supports functions which may be supplied arguments and **return** a numeric value (just like an exit status). They have access to the environment (global namespace) and may also declare local variables.

For example, if one wishes to deliver informational messages to the software user one should send those to stderr. Rather than performing this redirection of echo each time, one can use a simple function

```
function msg()
{
      echo “$*” >&2
}
```

Be default the return value of a function is the return value of the last command that it executed. So, the above would return the exit status of the `echo` command. One may explicitly choose a value using the 'return' statement:

```
return 1 # error
```

17.6.1 Libraries

The source command can be used to request that a script runs another script within itself (thus allowing the new script to modify the calling script's environment and function definitions).

This enables scripts to load libraries of useful software. This can be useful if a collection of scripts have some common functions, for example error messaging, which can then be shared across the suite.

The '.' is an alias for 'source'. Thus, the following are identical

```
source common-functions.sh
. common-functions.sh
```

17.7 Template

The following is a useful template for small scripts:

```
#!/bin/bash
# Purpose:
# Written by:
# Last edited on:

# The name of the script itself
ME=$(basename $0)
# The path to the script (from wherever the user is)
DIR=$(dirname $0)

# Always provide a usage/help
function usage()
{
      echo “Usage: $ME <-a blah> …
$ME digs holes and fills them in again”
}

# simple messaging (to stderr)
function msg()
{
      echo “$ME: $*” >&2
}
# simple 'error' type message
function err()
{
      msg “Error: $*”
}

# parse args
case $1 in
'-h'|'-?'|'–help')
      usage ; exit 0
      ;;
esac

status=0
# do stuff here
#

exit $status
```

18 Appendix A: Communities

18.1 DTU Environment

DTU Environment has a community that supports its research which relies on computation. The email list is:

community-modelling@env.dtu.dk

The administrators of the list are reachable via the admin mailing list:

community-modelling-admins@env.dtu.dk

Please, just ask the admins to add you to the community, and there you are.

We have few experts and many learners, thus it is essential that the learners help each other as the experts do not have time to help all. The community will hold regular meetings (lunch-time) to discuss common problems, so that they can better help each other, and so that the experts can help the most learners as possible with the best solutions that they can develop.

Your most important resource is:

https://care.env.dtu.dk

This is available from anywhere inside the network at DTU Environment, or via the terminal servers (dont know what that means? Talk to IT or a colleague.)