TASK

Introduction to Linux and DigitalOcean Droplets

Visit our website

Introduction

WELCOME TO THE INTRODUCTION TO LINUX AND USING DROPLETS TASK!

When you are finished with this task, you will understand what Linux is, understand how it works, and why we may want to use it as opposed to Windows or macOS. You will also come to understand how some very important command line tools work.

WHAT IS LINUX?

It's time to switch gears a little bit.

Up until now, you have most likely used a Windows system – possibly your entire life! While Windows is adequate for most tasks you may want to do, it can be (and in some cases, has always been) inadequate, complicated, or otherwise irritating to do certain kinds of tasks.

There is a need for a modular operating system that can do tasks by itself, allows you to plug services into it without significant configuration changes, and permits you to tweak everything until you have it dialled into your needs. That operating system is **Linux**.

A BRIEF HISTORY

The precursor to Linux was developed in the AT&T Bell Labs in 1970 and was called Unix. At the time, Unix was used for time-sharing and very simple word processing. Eventually it was expanded to do so much that it needed manuals, in the form of **man pages**. By 1977, a significant portion of the Unix codebase was written in C. During this time, Unix was fully owned by AT&T Bell.

In 1983, the GNU project was founded by Richard Stallman with the goal of creating a free (**as in libre, not gratis**) Unix-like operating system. He wrote many tools that would become the backbone of the entire operating system. However, the core of the operating system, the kernel, failed to get additional attention from the developer community and GNU was left incomplete.

In 1988, a standard called the Portable Operating System Interface (POSIX) was published by the IEEE Computer Society so that compatibility between operating systems could be maintained between both system and application development. Unix was created as the basis for the standard system interface as it was "manufacturer-neutral", and defined the functionality of command line interfaces, many user-level programs, and required program-level features such as input and output (I/O). Operating systems that mostly conform to POSIX are labelled as POSIX-compliant, and include macOS and most Linux distributions, including Android. Do note that although macOS may share many behind-the-scenes similarities with Linux, it is not a Linux distribution.

In 1991, a man named **Linus Torvalds** began a project that would later become the Linux kernel. As time went on, people would often use the software from the GNU project together with the Linux kernel to create a more complete operating system that supported a graphical user interface, and could do tasks from the command line. This combination is GNU/Linux, however many people in the field simply call it Linux.

Today, Linux is used in many applications, such as Wi-Fi routers, Android phones, and web servers powering a significant chunk of the Internet, making it the most used operating system in the world.

CLASSIC UNIX STRUCTURE

Unix and Unix-like operating systems are made up of three main parts: the kernel, shell, and userland.

The Kernel

The kernel is the brain of the operating system. It runs at the highest privilege level and is the closest you can get to the hardware itself (within the context of the operating system, of course). It handles memory allocation, communicates with devices using their drivers, handles I/O operations, schedules running processes, and a lot more. If applications want to do anything that requires interacting with hardware, they must interact with the kernel.

The Linux kernel is an example of a monolithic kernel, as it relies on an **init daemon** called systemd. Systemd is the parent process of every other process that runs in the system, and if it gets killed for whatever reason, it will take the rest of the system down with it. You will learn more about processes later in this task, and daemons and systemd in the cron jobs task.

The Shell

If the kernel is the heart, the shell is the nervous system or the messenger of the operating system. The shell is the way that users or applications can interact with the kernel. It reads input from the command line entered by you or an application. Then, it translates the input to instructions that the kernel can understand and execute.

There are console-based shells such as **bash** and **zsh**, and graphical user interface-based shells, such as the GNOME shell, which is installed by default on Ubuntu Linux.

Let's say you want to copy a file named "yes.txt" to another file called "indeed.txt". You would type into the shell "`cp yes.txt indeed.txt`", which would result in the shell telling the kernel to find and run the application called `cp`, supply `yes.txt` as an input and name the output `indeed.txt`, and then facilitate all the operations that allow `cp` to do what it needs to do (copy the file).

The userland

If the kernel and shell are the brain and nervous system of our operating system, then the userland is the skeleton and skin of our operating system.

The userland is completely separate from the kernel space. Applications that run here have their own piece of memory, and cannot access the memory of others. In the userland, you will find libraries and applications that the operating system uses to interact with the kernel.

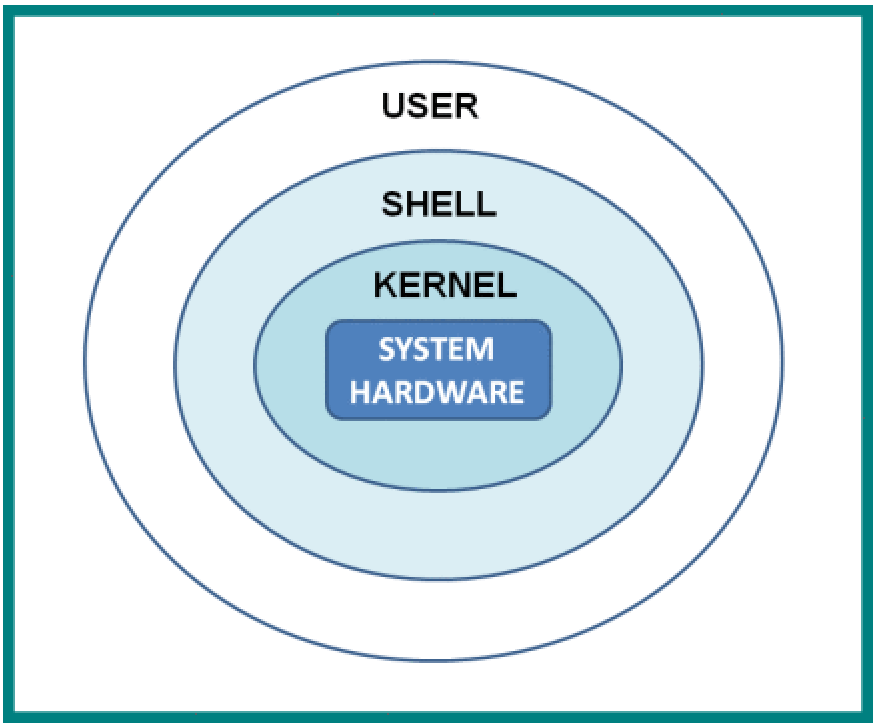


Image source:
https://medium.com/@673/what-happens-when-you-type-ls-l-in-the-shell-955104488c50

PROCESSES

When an instance of a program is running, it is referred to as a process. To help with multitasking, the kernel must manage how processes communicate with each other, which processes to run at what time, what a process is doing at any given time, and what to do with any system calls and signals that processes may produce or that the kernel may issue to any individual process.

Processes are identified using their process IDs (PIDs). PIDs are universally unique, and no process has the same PID as any other. For example, if you wrote a Python program that acted as a server, and another as a client, both would have the screen name of `/usr/bin/python3` but they would have different PIDs.

Extra resource

Use the **HTOP tool** on your Linux machine to view current processes and their IDs.

WHEN YOU RUN A PROGRAM

Depending on the kind of program you've written, your application may run directly or through an interpreter or virtual machine.

1. Interpreted vs compiled programs

Conversely, a compiled program does not need an intermediary to run. It is written in some language, and then a translator, or compiler, translates the source code into a language that some other system can understand natively. For example, C code is fed into the GCC compiler and a binary that Linux can execute directly is generated from the source code. C and Java are both examples of compiled languages (though, one can argue that Java is an interpreted language on account of it not generating machine code, but instead bytecode that is interpreted by the Java Virtual Machine. Read more about this **here**).

Both have their own pros and cons. You could argue that interpreted programs are more efficient, as you tell the language what it needs to do, and then it uses very optimised code behind the scenes to get what you need done, far more efficiently than a typical programmer could write in a compiled language. Conversely, one could argue that compiled programs are less intensive on a system, since there is no overhead introduced by an interpreter, and the code runs directly 'on the bare-metal'.

As you get better at writing code, you will realise that the best kind of language is the one that does your task the way you need it.

2. The Python virtual machine

The Python Virtual Machine is there to turn your lines of Python into machine code.

When you feed a Python source file into the interpreter, a .pyc file is generated, which is then run by the Python Virtual Machine. Inside the .pyc file, there is bytecode, which represents the fixed set of functions that can do anything you need your Python program to do.

For example, you may write the following program.

```
file = open("something.txt", "w") // open a file for writing to
file.write("I am a line inside a file.") // write this to the file
file.close() // tell the OS we're done using the file
```

The interpreter will generate the following bytecode:

```
  1          0 LOAD_GLOBAL              0 (open)
             2 LOAD_CONST               1 ('something.txt')
             4 LOAD_CONST               2 ('w')
             6 CALL_FUNCTION            2
             8 STORE_FAST               0 (file)

  2          10 LOAD_FAST               0 (file)
             12 LOAD_ATTR               1 (write)
             14 LOAD_CONST              3 ('I am a line inside a file.')
             16 CALL_FUNCTION           1
             18 POP_TOP

  3          20 LOAD_FAST               0 (file)
             22 LOAD_ATTR               2 (close)
             24 CALL_FUNCTION           0
             26 POP_TOP
             28 LOAD_CONST              0 (None)
             30 RETURN_VALUE
```

That looks almost nothing like our simple three-line program! But that's okay because this is how the Python virtual machine sees our code. Using the fixed set of instructions that the virtual machine understands, you can do almost anything you want to do in a programming language. And if there's something really specific that you need to do that the virtual machine can't do, you can always add the functionality to the language, since Python is completely open source. There's an incredibly slim chance that you'll ever need to do this, though.

So, just to recap, the following happens when you run a Python program:

- The interpreter takes the file and generates bytecode from your source code.
- The Python Virtual Machine takes the generated bytecode, and turns it into machine code that is run on your bare-metal.

SYSTEM CALLS AND SIGNALS

System calls (syscalls) and signals are mechanisms that allow the kernel to communicate with processes. Syscalls are from a process to the kernel, asking for some sort of resource. When this happens, a software interrupt is generated and the kernel is "woken up". This is called trapping into kernel space.

On the other hand, the kernel uses signals to notify processes of various events, such as I/O becoming available, or an illegal memory access attempt. If you have the appropriate permissions, you can also use them to communicate with other processes. You can write code that handles signals received from the kernel however you would want them to be handled.

The Linux kernel asks processes to close using a SIGTERM request, which allows the processes to gracefully close using the behaviour that is defined in their SIGTERM handler methods. If processes are unresponsive, the kernel can also use SIGKILL, which tells processes to cease immediately.

If you're interested, you can read more about syscalls **here** and signals **here**.

INTERPROCESS COMMUNICATION

Interprocess communication (IPC) mechanisms are provided by the kernel to allow processes to share data with each other. There are several ways to communicate with processes, including, but not limited to:

- Local files on disk
- Signals (though these are usually used for issuing commands, and not so much sharing big pieces of data)
- Sockets, which you will learn about soon
- Anonymous pipes

You can read more about IPC **here**.

PROCESS SCHEDULING

Every operating system needs a way to ensure that all running processes get sufficient CPU time. To facilitate this, the scheduler needs to know what each process is doing at any given time so that sufficient CPU time can be allocated for each process. To make this possible, Linux processes in the runqueue (the basic data structure in the scheduler) have one of 5 states at any given time: running, interuptinle_sleep, uninteruptible_sleep, stopped, and zombie.

Running

In this state, the process is actively doing something and has been allocated to a processor.

Interruptible_sleep

In this state, the process is most likely waiting for something to happen, such as a connection (in the case of a server), or a terminal or word processor waiting for the user to type something in. When a process reaches this state, the scheduler makes the process take the back seat for a bit, and other processes that need to do something are moved to the running state. When a process is in this state, it can be safely terminated.

Uninterruptible_sleep

In this state, the process is waiting for something to happen, but interrupting it would have consequences. For example, processes that are waiting for the kernel to finish an I/O operation on their behalf can be in this state. Although it's rare to find processes in this state, those that wait for I/O operations over a network can be in this state while waiting for a download to finish, or something similar.

Processes in this state cannot be killed by normal means due to inconsistencies that can happen, and Linux will refuse to let you do so if you try to.

Stopped

In this state, processes are suspended. Console applications can be put into this state by pressing Ctrl+Z, and then become unresponsive until they are brought back into the foreground using the `fg` command.

Zombie

Processes can spawn new processes, and these spawned processes are referred to as child processes. If the parent processes exit before removing their child processes from the process table, these lingering processes are called zombie processes. They have finished execution, but since they are "undead", they cannot be killed through normal means. Instead they need to be sent a SIGCHLD signal to their parent's PID. You will learn more about this in the cron job scheduling task.

Now that we have learnt about the different states a process can have, we can talk about the scheduler itself. The scheduler deals with two kinds of processes:

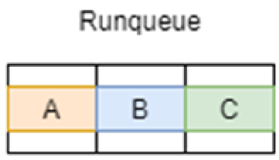
real-time processes and conventional processes.

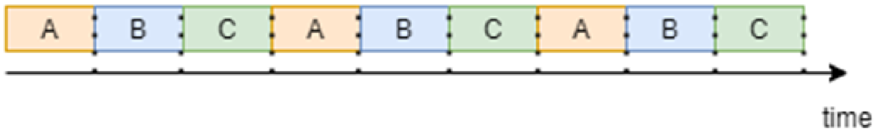
REAL-TIME PROCESSES

These kinds of processes have work that needs to be done with a strict time constraint. They can have two scheduling policies: SCHED_FIFO (First In First Out) and SCHED_RR (Round Robin).

1. **SCHED_FIFO**: In this policy, processes in the runqueue run in the order in which they arrived, and give up the CPU when they enter one of the sleep states, in which case they go to the back of the queue and the next process in the queue takes the CPU.

2. **SCHED_RR**: The Round Robin policy runs all the processes in the runqueue in a cyclic fashion, giving each process a little time to do its work before they get moved to the queue. It looks a little bit like this:





CONVENTIONAL PROCESSES

These kinds of processes aren't strict about how long they can take to do their tasks. For these processes, Linux uses the Completely Fair Scheduler to ensure that all these processes get a more or less equal amount of CPU time.

You can read more about the Completely Fair Scheduler **here**.

Files

In UNIX-like operating systems, everything is a file. This means that documents and such things are files (as you would expect), but I/O streams are also exposed through the file system using a file. For example, the hard drive in your computer (assuming it's a **SATA** drive and not an **NVMe** one) is /dev/sda. The default printer device is at /dev/lpt0. These devices can be read from and written to (depending on the type of device of course – you can't write to a keyboard!). System properties are exposed as files in the /proc folder as well, a folder that doesn't actually exist, because its contents are procedurally generated.

For example, if you were recycling your hard drive and wanted to erase it first, you would read the output from the pseudorandom number generator device and write that to your hard drive's device using this command:

```
sudo dd if=/dev/urandom of=/dev/sda
```

In this command, the **if** argument is the input file, and the **of** argument is the output file. Since these are device files, the output uses the device driver to appropriately take the data to the specified device. The advantage of this is that the programmer does not need to worry about the specific implementation of a device driver, instead they just read and write to the file to that device. Find out more about the `sudo` command towards the end of this document, or by typing `man sudo` in your Linux terminal.

It probably goes without saying, but **please do not run that command on your computer**. You will **lose all your data forever.**

Seriously. **Don't do it.** You have been warned!

LINUX DISTRIBUTION FILE SYSTEM STRUCTURE

Most distributions keep important files in a few folders:

/var

Things that are likely to change often, such as configuration files for userspace applications are kept here.

/home

All of your things are kept here. Documents, pictures, downloads, your desktop, configuration for applications installed for you only, that sort of thing.

/etc

System configuration files for systemd and such applications are kept here. The hosts file and the sudoers file are kept here as well.

/dev

Device files are kept here, including for some pseudodevices. **Pseudodevices** are devices such as /dev/null (which discards any input given to it, and returns a stream of null bytes if read from), /dev/zero (which returns a stream of zeroes if read from) and /dev/urandom, which is a cryptographically secure random number generator.

Security

The open-source nature of Linux makes it the most secure operating system kernel out right now. It has the advantage of many people having access to how its kernel works, and people who find vulnerabilities can patch them and contribute their changes to the next version of the kernel. Everybody knows exactly how the lock works, nobody has the key to it.

Linux security has many, many layers, and if you are interested in reading about them, you may do so **here**. For now, we will briefly talk about file security.

File security

Every file in Linux is owned by a user and a group user. If you open up a terminal window and type in `ls -l`, you would see something similar to this:

```
vin@localhost:~> ls -l
total 0
drwxr-xr-x 1 vin users  0 Mar 15  2022 bin
drwxr-xr-x 1 vin users 70 Dec 19 14:59 Desktop
drwxr-xr-x 1 vin users 86 Dec 20 00:02 Documents
drwxr-xr-x 1 vin users  0 Dec 19 14:58 Downloads
-rw-r--r-- 1 vin users  0 Dec 20 15:12 i_am_a_file.txt
drwxr-xr-x 1 vin users  0 Dec 19 14:58 Music
drwxr-xr-x 1 vin users  0 Dec 19 14:58 Pictures
drwxr-xr-x 1 vin users  0 Dec 19 14:58 Public
drwxr-xr-x 1 vin users  0 Dec 19 14:58 Templates
drwxr-xr-x 1 vin users  0 Dec 19 14:58 Videos
vin@localhost:~>
```

This is a directory listing of a user's folder in /home. On the left, you have the permissions matrix that every file has:

- d: the object is a directory.
- r: there are read permissions.
- w: there are write permissions.
- x: there are execute permissions
- -: a specific permission is missing. For files, the first thing in the permissions matrix is always a dash, to indicate that it is a file.

The permissions matrix specifies permissions for the user, group, and people who are neither the user nor group, in this order, from the left. Take a look at i_am_a_file.txt in the screenshot just above:

- It is a file, so the first entry is a dash.
- The user (named vin) has read and write permissions, but no execute permissions.
- The group users has read permissions, but no write or execute permissions.
- People who are neither vin nor in the users group have read permissions, but no write or execute permissions.
- To the right of the 1, you can see which user and group (respectively) owns the files and folders in the directory.

To change permissions attached to a file, the **chmod** command is used. We will talk about it in a bit more detail later.

Arguments

Console applications that are run in the terminal often take arguments and options. You've done this yourself many times when you run your Python programs, when you give the name of your program to Python as an argument. You can do this yourself in your Python programs as well, and this will make them a lot more reusable than they already are.

For example, this Python program will take your arguments and list them out by number:

```
import sys

number_of_args = len(sys.argv)
arg_count = 0
print("Number of arguments: " + str(number_of_args))

for arg in sys.argv:
    print("Argument #" + str(arg_count) + ": " + arg)
    arg_count += 1
```

When we run it, this is the output:

```
vin@localhost:~/Documents> python3 argue_with_me.py Python is one of the languages of all time
Number of arguments: 10
Argument #0: argue_with_me.py
Argument #1: Python
Argument #2: is
Argument #3: one
Argument #4: of
Argument #5: the
Argument #6: languages
Argument #7: of
Argument #8: all
Argument #9: time
vin@localhost:~/Documents>
```

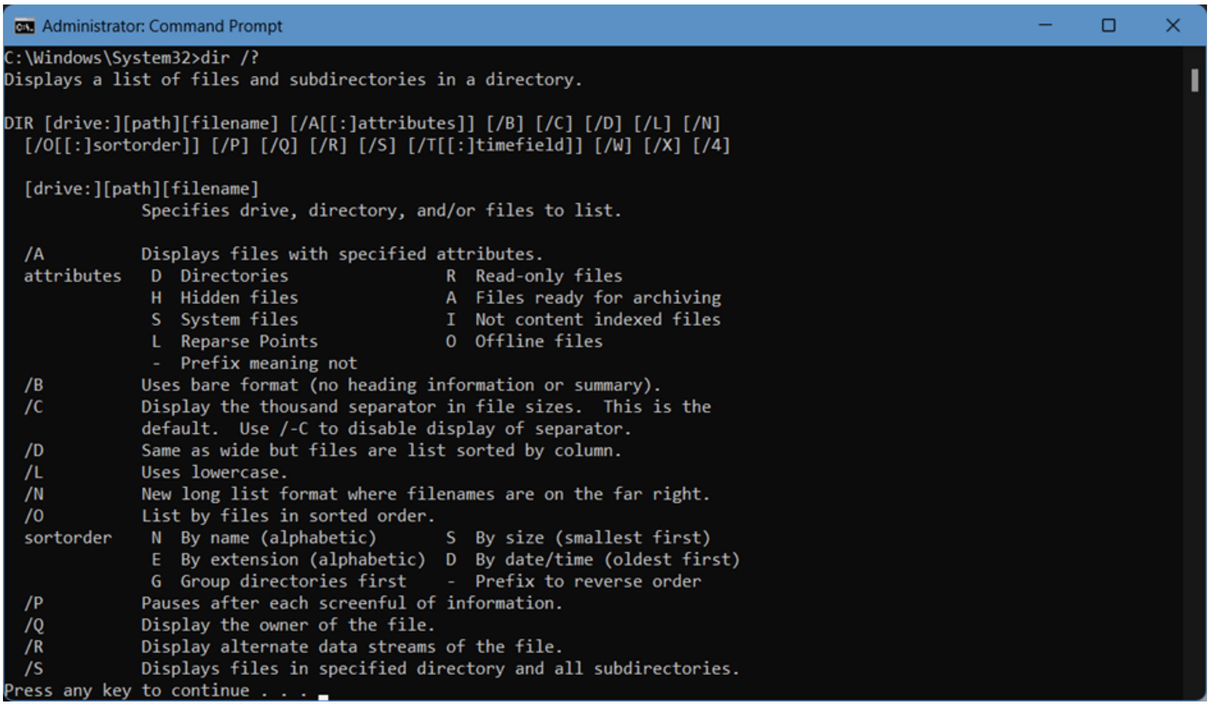
The 0^{th} argument is the program's name. Arguments 1 to 9 are everything after the program name.

While this was a fun program that shows that we can take an arbitrary number of arguments and options, actual programs take arguments and options separately. For instance, let's take a look at the **`cat`** command's usage:

```
vin@localhost:~/Documents> cat --help
Usage: cat [OPTION]... [FILE]...
Concatenate FILE(s) to standard output.

With no FILE, or when FILE is -, read standard input.

  -A, --show-all           equivalent to -vET
  -b, --number-nonblank    number nonempty output lines, overrides -n
  -e                       equivalent to -vE
  -E, --show-ends          display $ at end of each line
  -n, --number             number all output lines
  -s, --squeeze-blank      suppress repeated empty output lines
  -t                       equivalent to -vT
  -T, --show-tabs          display TAB characters as ^I
  -u                       (ignored)
  -v, --show-nonprinting   use ^ and M- notation, except for LFD and TAB
      --help     display this help and exit
      --version  output version information and exit

Examples:
  cat f - g  Output f's contents, then standard input, then g's contents.
  cat        Copy standard input to standard output.

GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
Full documentation <https://www.gnu.org/software/coreutils/cat>
or available locally via: info '(coreutils) cat invocation'
vin@localhost:~/Documents>
```

Here, we have a set of arguments in two different conventions: GNU and POSIX. GNU conventions are a single dash and a letter, which is sometimes case sensitive. POSIX conventions are two dashes and one or more words separated by dashes. For your own programs, it's important to be consistent with your notation. If you are to use GNU conventions, use them throughout. If you are to use POSIX conventions, use them throughout. If you want to use both, make sure that arguments have both POSIX and GNU equivalents.

And for comparison’s sake, here is the usage for the dir command in Windows:



```
Administrator: Command Prompt
C:\Windows\System32>dir /?
Displays a list of files and subdirectories in a directory.

DIR [drive:][path][filename] [/A[[:]attributes]] [/B] [/C] [/D] [/L] [/N]
  [/O[[:]sortorder]] [/P] [/Q] [/R] [/S] [/T[[:]timefield]] [/W] [/X] [/4]

  [drive:][path][filename]
              Specifies drive, directory, and/or files to list.

  /A          Displays files with specified attributes.
  attributes   D  Directories                R  Read-only files
               H  Hidden files               A  Files ready for archiving
               S  System files               I  Not content indexed files
               L  Reparse Points             O  Offline files
               -  Prefix meaning not
  /B          Uses bare format (no heading information or summary).
  /C          Display the thousand separator in file sizes.  This is the
              default.  Use /-C to disable display of separator.
  /D          Same as wide but files are list sorted by column.
  /L          Uses lowercase.
  /N          New long list format where filenames are on the far right.
  /O          List by files in sorted order.
  sortorder    N  By name (alphabetic)       S  By size (smallest first)
               E  By extension (alphabetic)  D  By date/time (oldest first)
               G  Group directories first    -  Prefix to reverse order
  /P          Pauses after each screenful of information.
  /Q          Display the owner of the file.
  /R          Display alternate data streams of the file.
  /S          Displays files in specified directory and all subdirectories.
Press any key to continue . . .
```

Windows uses its own slash convention. Some applications have several letters after the slash, and others allow you to precede arguments with a dash instead of a forward slash, however these implementations are inconsistent. This is not to criticise Windows for its (many) shortcomings, but is to show that there are multiple styles of formatting arguments for applications.

STANDARD INPUT, OUTPUT, ERROR, REDIRECTION, AND PIPING

There are three file descriptors that handle I/O (Input/Output) in most Linux programs: stdin, stdout, and stderr.

stdin

Usually, stdin comes from the keyboard, a device file descriptor labelled as 0. Stdin can also come from an actual file. This makes sense when you recall that everything in Unix-like operating systems is a file. If we run `cat` without any arguments, it will read from the 0 file descriptor, which is our keyboard in this case. Anything we type into `cat` will be returned to us until we kill it.

```
vin@localhost:~/Documents> cat
i am typing many things
i am typing many things
```

We can also use an actual file to act as stdin, too.

```
vin@localhost:~/Documents> cat 0< i_am_a_file.txt
I am a file with this line of text inside me.
vin@localhost:~/Documents>
```

(The **0<** symbol indicates that we are referring to stdin).

stdout

Stdout usually goes to the console, your terminal, or an X terminal. It can be any one of these depending on what started the process. The file descriptor is labelled as 1.

Going back to `cat` then, our terminal is our stdout, so any output that is "written" to the 1 file should be displayed on our terminal window. Let's give i_am_a_file.txt as an argument to cat:

```
vin@localhost:~/Documents> cat i_am_a_file.txt
I am a file with this line of text inside me.
```

Predictably, we see the contents of this text file again.

We can redirect the output of any command that writes to stdout to another file or file descriptor. For example, if we wanted to write everything that dmesg outputs to stdout to a file called kernel_log.txt, we would do it as follows:

```
vin@localhost:~/Documents> sudo dmesg > kernel_log.txt
vin@localhost:~/Documents>
```

Nothing was written to the terminal window, because the file kernel_log.txt was our stdout, and everything that dmesg returned is inside that file:

```
[  442.768743] sd 0:0:0:0: [sda] Starting disk
[  461.496577] sd 0:0:0:0: [sda] Synchronizing SCSI cache
[  461.543212] sd 0:0:0:0: [sda] Stopping disk
[  475.544963] ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
[  475.549464] ata1.00: configured for UDMA/133
[  475.549789] sd 0:0:0:0: [sda] Starting disk
[  494.249902] sd 0:0:0:0: [sda] Synchronizing SCSI cache
[  494.297660] sd 0:0:0:0: [sda] Stopping disk
[  504.216799] ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
[  504.219782] ata1.00: configured for UDMA/133
[  504.220169] sd 0:0:0:0: [sda] Starting disk
[  522.850458] sd 0:0:0:0: [sda] Synchronizing SCSI cache
[  522.898010] sd 0:0:0:0: [sda] Stopping disk
[  536.972110] ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
[  536.975139] ata1.00: configured for UDMA/133
[  536.975498] sd 0:0:0:0: [sda] Starting disk
[  555.715726] sd 0:0:0:0: [sda] Synchronizing SCSI cache
[  555.717625] sd 0:0:0:0: [sda] Stopping disk
[  565.654501] ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
[  565.657473] ata1.00: configured for UDMA/133
[  565.657949] sd 0:0:0:0: [sda] Starting disk
[  584.356969] sd 0:0:0:0: [sda] Synchronizing SCSI cache
[  584.359418] sd 0:0:0:0: [sda] Stopping disk
[  590.810080] hrtimer: interrupt took 15038015 ns
[  598.429207] ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
[  598.431815] ata1.00: configured for UDMA/133
[  598.432545] sd 0:0:0:0: [sda] Starting disk
vin@localhost:~/Documents>
```

Using the single **>** symbol to change stdout overwrites any file you may specify. If you want to append the output to the file, use **>>**

```
vin@localhost:~/Documents> cat i_am_a_file.txt
I am a file with this line of text inside me.
vin@localhost:~/Documents> echo  And this was appended after the fact! >> i_am_a_file.txt
vin@localhost:~/Documents> cat i_am_a_file.txt
I am a file with this line of text inside me.
And this was appended after the fact!
vin@localhost:~/Documents>
```

You can find out more about the **`echo`** command towards the end of this document, or by typing **`man echo`** in your Linux terminal.

stderr

Any output that any command gives can be one of two things:

1) Valid output

2) An error

Valid output goes to stdout. Errors go to stderr instead, and its file descriptor is 2.

If we run this command: **`find / -name “*” -print`**, we will get a lot of output, and many of the lines of output would say “Permission denied,” as we may not have the appropriate permissions to be viewing some of those files (which means that the permission system we mentioned earlier is working as intended!) It's great that Linux is telling us that we can't do things we're not supposed to do, but what if we don't care about that? What if we just want to see things that we do have permission to see?

In that case, we would tweak the command slightly to say the following:

```
find / -name “*” -print 2> /dev/null
```

Here, we told the shell to take everything that is written in stderr and write it to the /dev/null device. Writing to the /dev/null device is about the same as throwing something into a bottomless pit – it will be gone forever, and if you try to read from the device in hopes of getting something back, you will receive an endless stream of null bytes.

In this case, we will only see only stdout, and stderr will be thrown away.

If this is a bit unclear for you, or you like seeing why things go wrong in their own separate file, then, of course, you can still change stderr to output to an actual file, say, one called: errors.txt. That command would look as follows:

```
find / -name “*” -print 2> errors.txt
```

But what if we wanted to put stdout in one file, and stderr in another file? That's also possible, we just need to redirect them accordingly:

```
find / -name "*" -print 1> output.txt 2> errors.txt
```

Stdout will go to output.txt, and stderr will go to errors.txt.

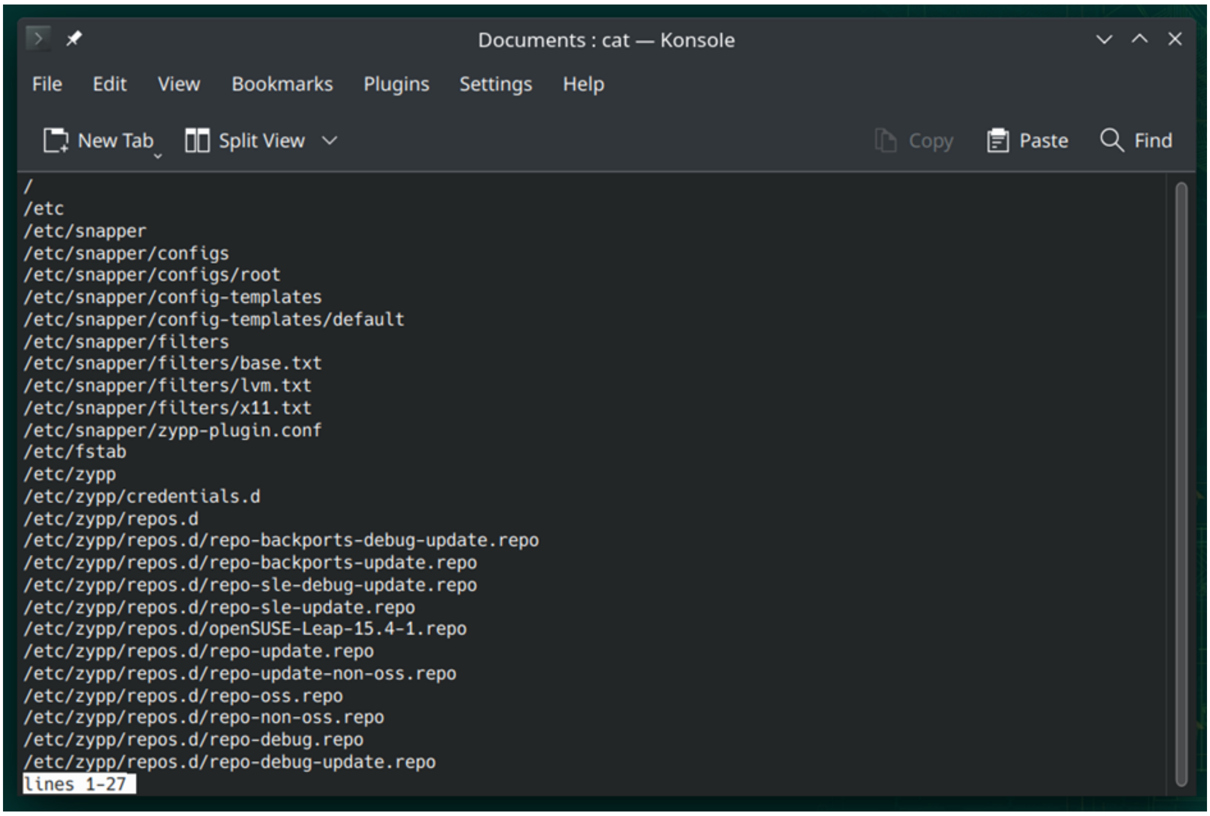
Piping

How useful would it be if we could make the output of one command be the input of another? Incredibly, and the pipe command lets us do exactly that!

Let's take a look at the output.txt file we generated just now. If we give it to `cat` as an argument, `cat` would write all of its many, many lines straight to stdout. What if we wanted to give this output to another application that allows us to scroll up and down a given input?

Less is an application that does exactly that. If given a very large input, less will let you scroll through it line by line, going both up and down. So, if we pipe cat's output to less, we can scroll through it line by line:

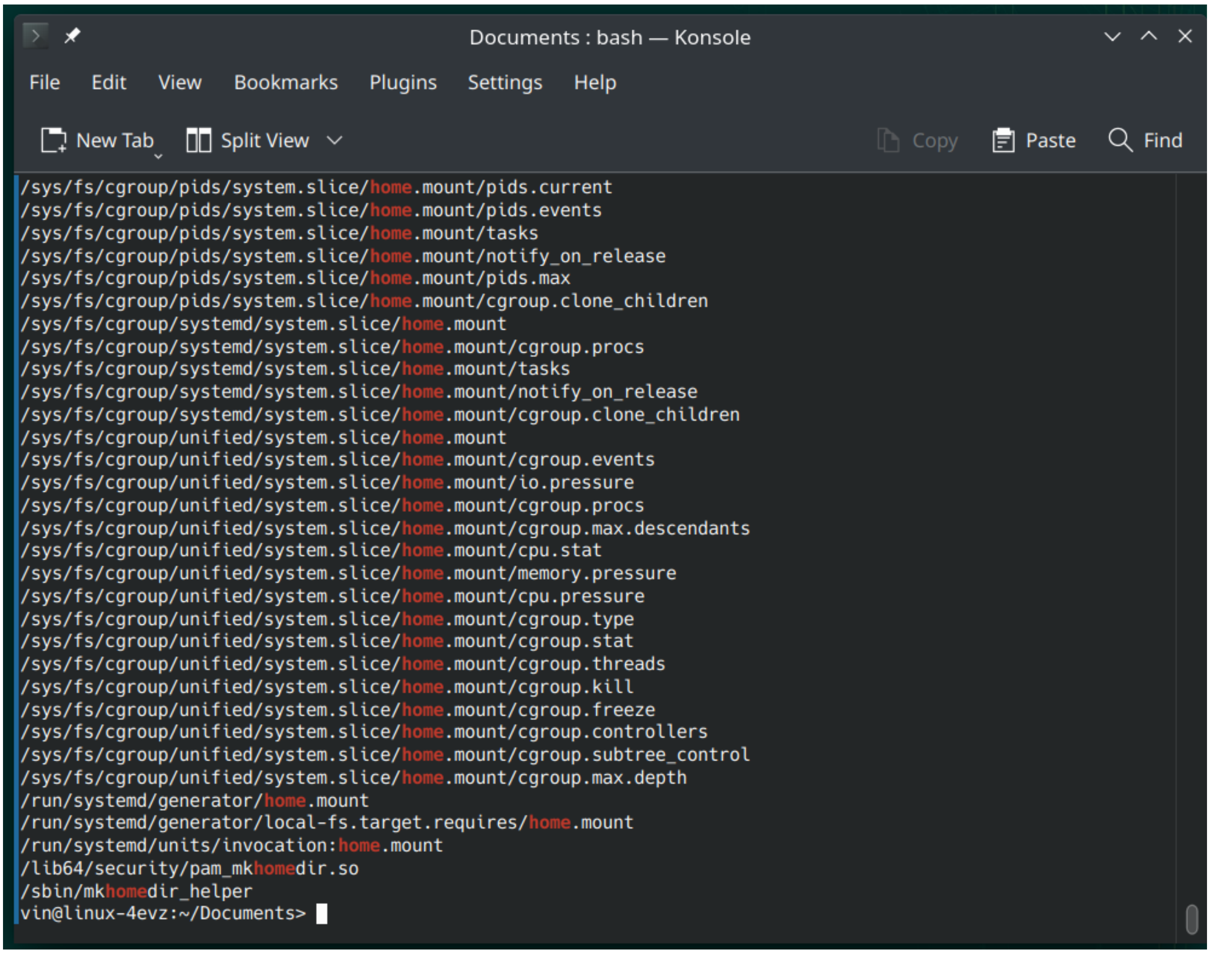
```
cat output.txt | less
```



We can now scroll through the file line by line.

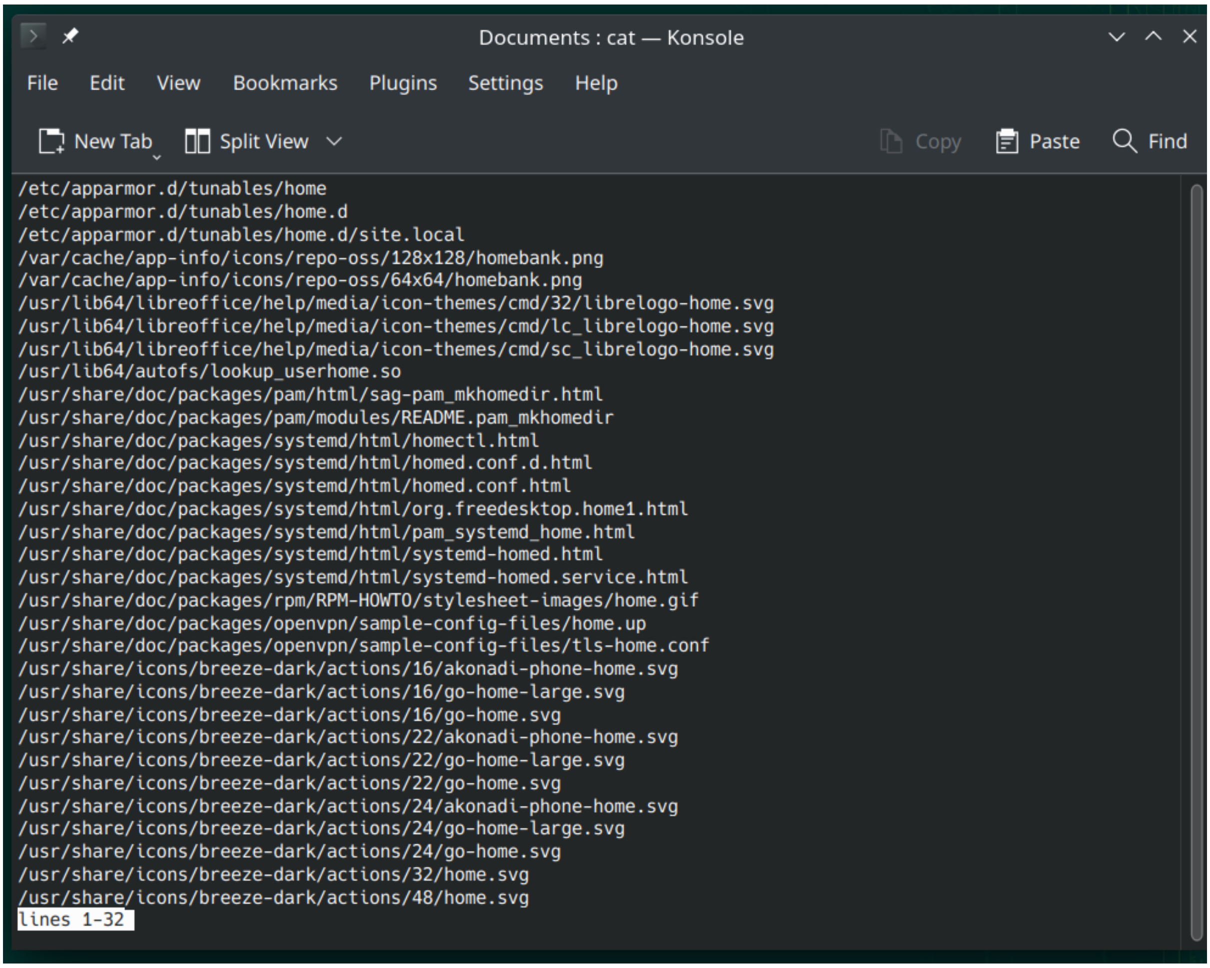
But that's not all. What if we wanted to find entries that match the "home" pattern? Grep is a tool that can do exactly that; it filters through input and writes to stdout and only provides outputs that match a given pattern. If we tweak our command slightly:

```
cat output.txt | grep home
```



Wait, but now we can't scroll through our output any more. Can we pipe the output we get from grep into yet a third command? If we tweak our command slightly.

```
cat output.txt | grep home | less
```



… we'll see that we absolutely can!

Pipes are very powerful, and with them and redirection, you can create an assembly line of commands that create very highly curated outputs from a single instruction.

The Unix philosophy: is a set of norms and cultures around minimalist software development. The philosophy originated from the Bell System Technical Journal in 1978, and was **summarised** in A Quarter-Century of Unix in 1994, as the following:

1. Write programs that do one thing and do it well.

2. Write programs to work together.

3. Write programs to handle text streams, because that is a universal interface.

Essentially, having many small programs that do one specific thing, and are interoperable with each other's outputs is preferable to writing one big program that tries to be a Swiss army knife. (Interestingly, **Busybox** is a software suite that is positioned by its creators as the Swiss Army Knife of embedded Linux. It bundles many common Linux applications into one big executable, which saves disk space and processing overhead. One could argue that Busybox violates the first tenet of the Unix philosophy, as it is literally one program that does more than 300 different things).

SPECIAL FILE VARIABLES

Some sets of symbols always refer to something specific, in the same way as a superglobal variable in other programming languages:

- `~`: This is your home folder for your account. If your account name is "vin", then your home folder will be `/home/vin`, and changing your directory to `~` (`cd ~`) will take you to `/home/vin`.
- `.`: This is the folder you are currently in. If we have an executable file called hello.sh in our folder and we would like to execute it, we would type in `./hello.sh`.
- `..`: This is the folder one level up from the folder you are in. If you are in `/home/vin/Documents`, then changing your directory with `..` (typing `cd ..`) will take you to `/home/vin`.
- `/`: This is the root of the file system. Root is the lowest level of the file system, and if you wanted to use absolute addressing, you would start from /. For example, a relative address to the output.txt file we made earlier would be `~/Documents/output.txt`, whereas an absolute address would be `/home/vin/Documents/output.txt`.

SOME USEFUL COMMANDS

Note that it's okay to not know all of these by heart! Even the most experienced Linux users have to look up which command to use for what sometimes. However, the more time you spend in the terminal, the greater your intuitive sense of what tool does what will kick in, and you will find yourself looking commands up a lot less often. That said, here are some very commonly used commands:

- `man`: The Linux manual. Any installed application appends its instruction manual to the man database, and they can be accessed as follows:

  `man (utility) [section]` where the section is optional.

- `ssh`: Secure Socket Shell. This allows you to access the default shell for a specific user on a remote Linux machine that is also running an SSH server.
- `grep`: This matches patterns from data in stdin.
- `cat`: This concatenates its input to stdin.
- `echo`: This writes something to stdout.
- `read`: This reads a certain number of bytes from a file descriptor.

- `mkdir`: This creates a folder.
- `rmdir`: This deletes an empty folder.
- `touch`: This can be used to change the time attributes of a file, and to create new files.
- `rm`: Deletes a file.
- `cp`: Copies a file.
- `mv`: Moves a file.
- `ls`: Lists everything in the working directory.
- `ln`: Creates a link from one file or folder to another.
- `find`: Looks for files in a directory structure.
- `which`: Shows you the full path of any command you may run. For example, if you have Python installed in /var/opt and in /usr/bin (for some reason), then `which` would tell you exactly which Python source you're running when you type "python3".
- `chmod`: Changes permissions for a file system object.
- `chown`: Changes the owner of a file system object.
- `chgrp`: Changes the group ownership for a file system object.
- `su`: Substitute user. Briefly log on as someone else.
- `sudo`: Substitute user do. Do something as someone else. Usually, this is used for doing things that require root privileges, in which case, things are done as the superuser account.
- `who`: Shows everyone who is logged in, and on which terminals. Normally, you should see only your account here.

LINUX DISTRIBUTIONS

Linux distros (short for "distributions") are different versions of the Linux operating system. Linux is an open-source operating system, which means that anyone can download the source code and modify it to create their own version of the operating system. There are many different Linux distros available, each with its own unique set of features and characteristics.

Ubuntu is one such Linux distro that was first released in 2004. It is a popular choice for web developers because it is free, open-source, and relatively easy to use, especially for those who are new to Linux. Ubuntu is based on **Debian**, another popular Linux distro, and it has a large and active community of users and developers who contribute to its ongoing development and support.

As a web developer, you might be interested in using Ubuntu because it comes with a number of tools and features that are useful for web development, including the LAMP stack (short for Linux, Apache, MySQL, and PHP), which is a set of open-source software that is commonly used for building and running web applications. Ubuntu also has a package manager called **apt**, which makes it easy to install and manage software packages, including web development tools like Git, Node.js, and Apache.

For example, you can use apt to install Git on Unbuntu like this:

```
sudo apt install git
```

Sudo is the equivalent of running something as an admin in windows, if you're not signed in as the admin or root account.

When changing a distro of Linux it will primarily function the same. For example, commands like "**sudo**" and "**mkdir**" will function the same in all distros of Linux. However you'll need to potentially use a different package manager based on the version of Linux you're using.

For example, if we were using AWS Linux, and we want to install git, the command would change to this:

```
sudo yum install git
```

When trying out a new distro, always be sure to read the documentation related to that distro so you can familiarise yourself with any nuances that might be related to it.

Overall, Ubuntu is a great choice for web developers who are new to Linux because it is easy to use, widely supported, and comes with a lot of useful tools and features right out of the box.
You'll notice that it's supported on nearly all Virtual Machines, including AWS, MS Azure, GCP and, of course, DigitalOcean, which is why we're going to be using it as well.

WHAT ARE DROPLETS?

DigitalOcean offers a service called "Droplets," which are essentially virtual machines (VMs) that you can **rent** on a pay-per-use basis. Droplets are a popular choice for web developers who need a flexible and scalable hosting solution for their applications.

A VM is a software implementation of a computer that runs an operating system and applications. It behaves like a separate physical machine, but it is actually running on top of another physical machine.

In other words, a VM is like a computer within a computer. VMs are often used to run multiple operating systems or instances of the same operating system on a single physical machine. They are useful because they allow you to isolate applications and services from one another, and they can be easily created, moved, and replicated.

Here are some key features and benefits of DigitalOcean Droplets:

- **Easy to set up and use:** DigitalOcean provides a user-friendly web-based interface that makes it easy to create, manage, and configure Droplets. You can choose from a variety of pre-configured Droplet *images* (e.g. **Ubuntu**, CentOS, Fedora, etc.) or create a custom image from scratch.

- **Scalable and flexible:** You can easily scale your Droplet up or down depending on your needs. This makes it easy to handle traffic spikes or changes in demand for your application.

- **High performance:** DigitalOcean Droplets use SSD storage and high-speed networking to deliver fast performance for your applications.

- **Cost-effective:** DigitalOcean offers transparent pricing that allows you to pay only for what you use. You can choose from a variety of pricing plans based on the resources that you need, including CPU, memory, storage, and bandwidth.

- **Security and reliability:** DigitalOcean provides a secure and reliable hosting environment with features like automatic backups, firewalls, and monitoring tools.

Droplets provide a flexible and powerful hosting solution for web developers who need a scalable, cost-effective, and easy-to-use platform for their applications.

Compulsory Task

This compulsory task will test your understanding of command line arguments and options, and writing to stdout. Write Python programs that implement the following Linux tools (in other words, running your Python program will provide the same functionality as the tool you're emulating):

- cat
- echo
- grep

Note: that all of these tools have man pages, so if you are unsure about how any of them work, feel free to refer to the man pages, where they are documented extensively.

Each of these tools allow you to provide files as arguments, but also accept standard input as well. Your program needs to handle each of these cases individually. You have been given a code template with some light comments to help guide you.

Note: The GNU implementation of **grep** does regex pattern matches. For this task, you need to **only** do a simple exact matching system.

Completed the task(s)?

Ask an expert code reviewer to review your work!

Review work

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.