HyperionDev

Cyber Security Tools – Linux

Task

Visit our website

Introduction

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

Linux

It's time to switch gears a little bit.

Up until now, you have most likely used Windows, or perhaps a Mac (running macOS), as these are the two most common operating systems. These systems are popular due to their user-friendliness and compatibility with the large majority of commercial proprietary software. However, they can be inadequate or somewhat complicated to use for certain kinds of tasks. Sometimes you need 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 perfectly configured to meet your needs. That operating system is Linux. Linux comes in many different flavours, called **distributions or "distros"**, which are based on the same Unix foundations but have evolved to be quite different from each other.

A brief history

The foundation for Linux is an operating system that was developed in the AT&T Bell Labs in 1969, called Unix. At the time, Unix was used for time-sharing and very simple word processing. Eventually, its capabilities were expanded to enable it 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 captivate 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 and many user-level programs, and required program-level features such as input/output (I/O). Operating systems that mostly conform to POSIX are labelled as POSIX-compliant and include macOS and most Linux distributions, including **Android**. Note that, although macOS may share many foundational 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 (GUI) and enabled tasks to be run from the command line. This combination is GNU/Linux, but many people 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 more about daemons and `systemd` if your course includes the cron jobs task.

The shell

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

There are console-based shells such as Bash and Zsh, and GUI-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 input, 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.

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.

From source code to machine code

Depending on the kind of program you have written, your application may run directly or through an interpreter or virtual machine. Let's take a closer look at these two new concepts.

Interpreted vs compiled programs

An interpreted program is a program that is executed line-by-line by an **interpreter** rather than being compiled into machine code that runs directly on the hardware. The interpreter reads each line of the program, translates it into lower-level instructions, and immediately executes those instructions before moving on to interpret the next line. This allows interpreted programs to run on any platform supported by the interpreter, without having to be compiled separately for each platform.

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 (although 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 approaches have pros and cons. Interpreted languages generally do not execute programs more efficiently than compiled languages. Interpreted languages translate high-level code into machine code line-by-line at runtime, which introduces additional computational overhead. Some interpreters use just-in-time (JIT) compilation to optimise performance by compiling parts of the code during execution, but this often does not match the efficiency of pre-compiled programs. Compiled programs are directly converted into machine code before execution, allowing them to run on the system hardware with minimal overhead. This direct execution typically makes compiled programs faster and more efficient compared to interpreted ones.

So, which is better? It depends on the use case. 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 to be done.

The Python virtual machine

A Python virtual machine turns your lines of Python code 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 a fixed set of functions that can do anything you need your Python program to do.

For instance, imagine you write the following program:

```python
# Open the file in write mode

file = open("something.txt", "w")

# Write a line to the file

file.write("I am a line inside a file.")

# Close the file

file.close()
```

The interpreter will generate the following bytecode:

```
        0           0 RESUME                    0

        3           2 PUSH_NULL

                    4 LOAD_NAME                 0 (open)

                    6 LOAD_CONST                0 ('something.txt')

                    8 LOAD_CONST                1 ('w')

                   10 PRECALL                   2

                   14 CALL                      2

                   24 STORE_NAME                1 (file)
```

```
  5          26 LOAD_NAME                1 (file)
             28 LOAD_METHOD              2 (write)
             50 LOAD_CONST               2 ('I am a line inside a file.')
             52 PRECALL                  1
             56 CALL                     1
             66 POP_TOP
  7          68 LOAD_NAME                1 (file)
             70 LOAD_METHOD              3 (close)
             92 PRECALL                  0
             96 CALL                     0
            106 POP_TOP
            108 LOAD_CONST               3 (None)
            110 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 cannot do, you can always add the functionality to the language, as Python is completely open source. There's an incredibly slim chance that you'll ever need to do this, though, as most things you want to do are things that someone else has needed to do before you, and so the functionality has already been created.

Let's recap. The following happens in a Python virtual machine when you run a Python program:

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

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 that acted as a client, both would have the screen name of `/usr/bin/python3`, but they would have different PIDs.

Extra resource

Later, you will install and run Kali Linux. You can use the **HTOP tool** on your Kali Linux machine to view current processes and their IDs.

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".

By contrast, the kernel uses signals to notify processes of various events, such as I/O functionality becoming available, or an illegal memory-access attempt. If you have the appropriate permissions, you can also use syscalls to communicate with other processes. You can write code that handles signals received from the kernel however you 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 for sharing large volumes of data),
- Sockets, which you will learn about soon, and
- 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(Central Processing Unit) 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) can be in one of five states at any given time: `running and runnable`, `interruptible_sleep`, `uninterruptible_sleep`, `stopped`, and `zombie`.

Running and Runnable

- **Running**: In this state, the process is actively executing and has been assigned to a CPU. It is performing tasks and using CPU resources.
- **Runnable**: A process in this state is ready to run and is waiting for CPU time. It is not currently executing, but is in a state where it can be scheduled to run as soon as the CPU becomes available.

The key difference between "running" and "runnable" is that running processes are currently being executed, while runnable processes are queued and waiting for CPU time.

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 instance, 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 uninterruptible sleep, it does happen; for example, a process waiting for I/O operations over a network can be in this state. Processes in uninterruptible sleep 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 the stopped 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. Stopped processes are typically used in situations where the user or system needs to temporarily pause a task without terminating it entirely.

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 if your course variant includes the cron job scheduling task.

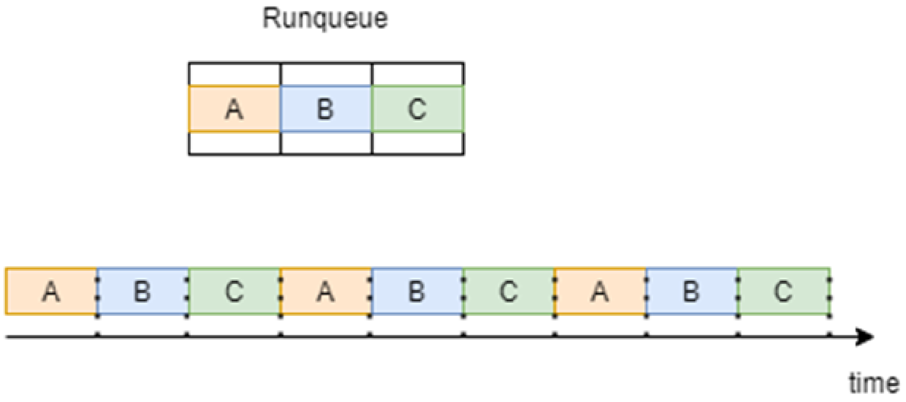
Now that we have learned about the different states a process can be in, we can finally get down to how to schedule processes! The scheduler deals with two kinds of processes: real-time processes and conventional processes.

Real-time processes

Real-time processes have work that needs to be done within 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 (CFS) to ensure that all these processes get a more or less equal amount of CPU time.

You can read more about the CFS **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 instance, 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 cannot 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
```

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!

Now, let’s look at what it does. In the above 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 for that device. You will find out more about the `sudo` command towards the end of this document (or you can also type `man sudo` into your Linux terminal and read the man page that comes up).

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).

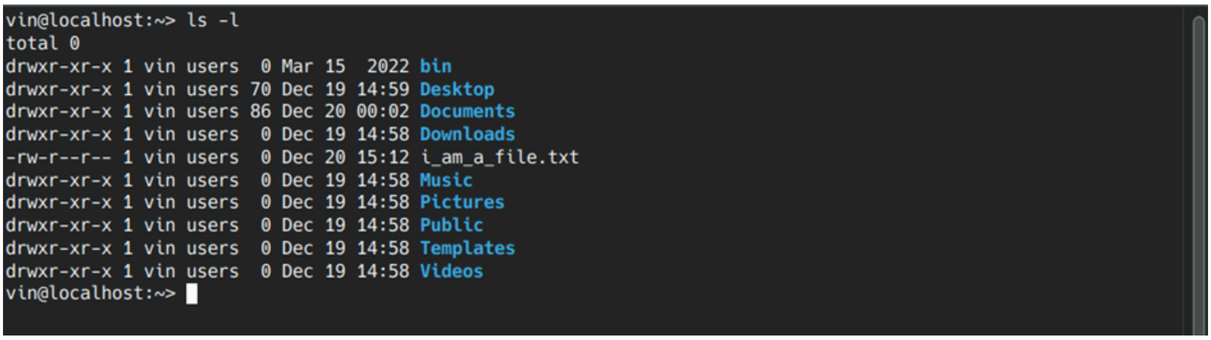
Security

The open-source nature of Linux contributes significantly to its security. With many people having access to its source code, vulnerabilities can be identified and patched by a global community of developers. This collaborative approach allows for rapid discovery and resolution of security issues, as contributors can submit fixes and improvements that are integrated into future versions of the kernel. The transparency of open source means that, while the mechanisms and source code are publicly accessible, the actual security of the system is maintained through rigorous scrutiny and constant updates.

Linux security has many, many layers, and if you are interested in reading about them, you may do so **here**. For now, we will talk briefly 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, and
- `-`: a specific permission is missing. For files, the first thing in the permissions matrix is always a dash, to indicate that it is not a directory but a file.

The permissions matrix specifies permissions for the user, group, and people who are neither the user nor a 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 `users`' group 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` (**ch**ange **mod**e) 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've 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 more reusable than they are currently.

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

```python
import sys

# Get the number of command-line arguments

number_of_args = len(sys.argv)

arg_count = 0

# Print the number of command-line arguments

print("Number of arguments: " + str(number_of_args))

# Iterate over each command-line argument and print it

for arg in sys.argv:

    print("Argument #" + str(arg_count) + ": " + arg)

    arg_count += 1
```

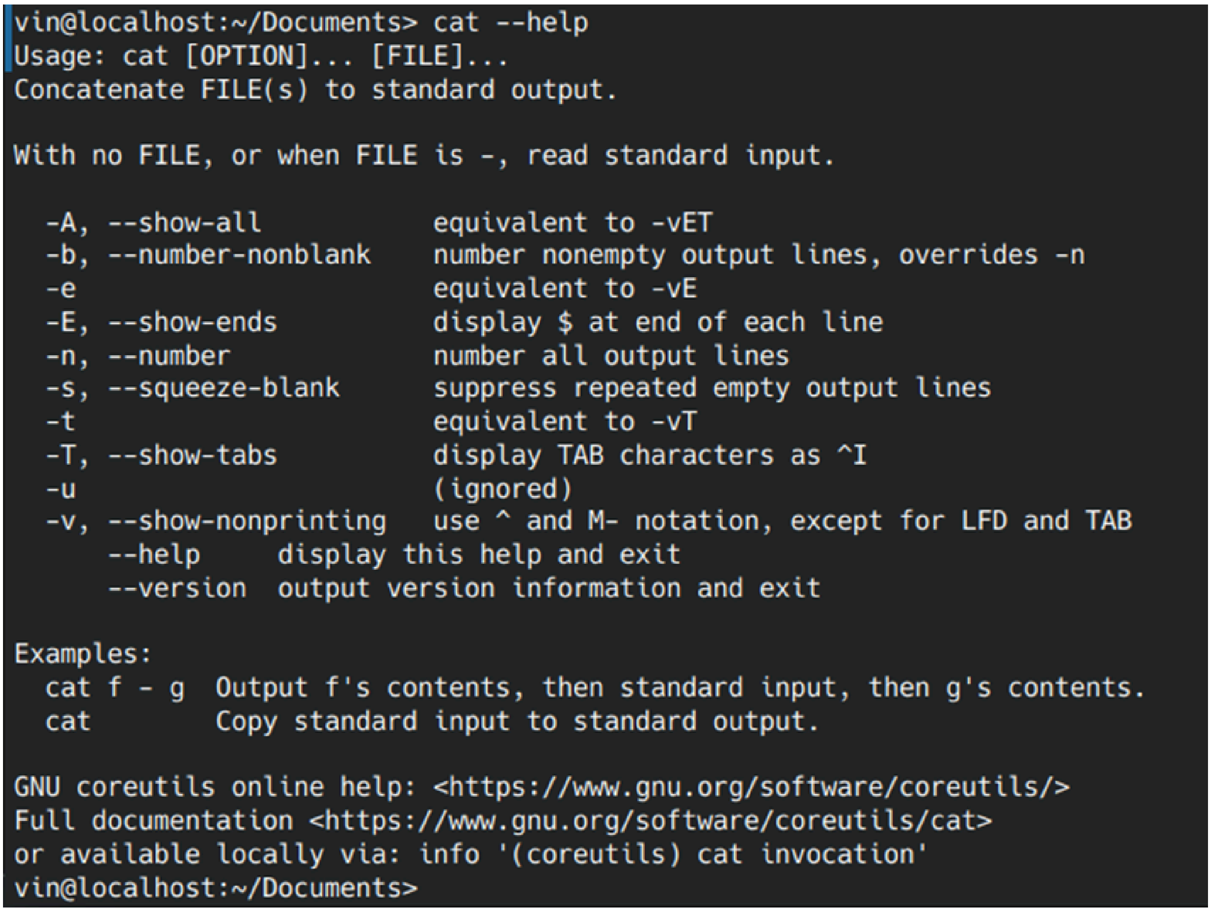
When we run it, this is the output:

```
vin@localhost:~/Documents> python 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 0th argument is the program's name. Arguments one to nine 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.

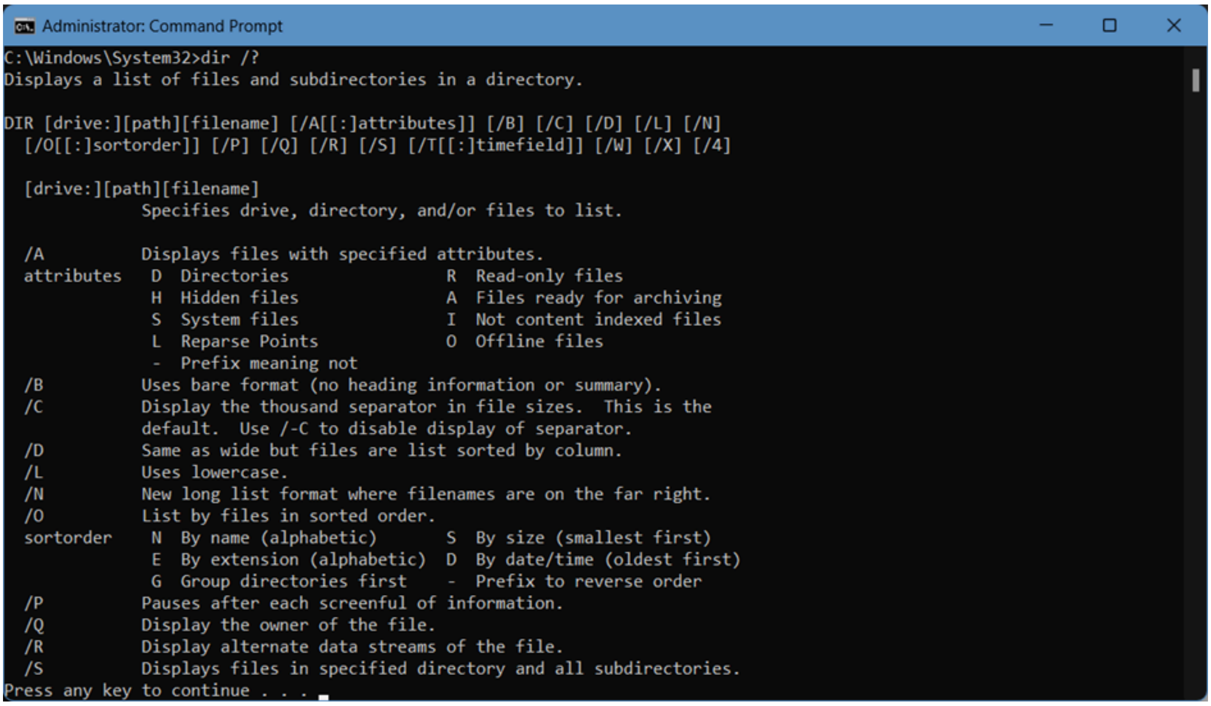
For example, let's take a look at the cat command's usage:



```
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 want to use GNU conventions, use them throughout. If you want to use POSIX conventions, use them throughout. If you want to use both, make sure that arguments have both POSIX and GNU equivalents.

For comparison, here is the output of the `dir` command in Windows:



```
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 . . .
```

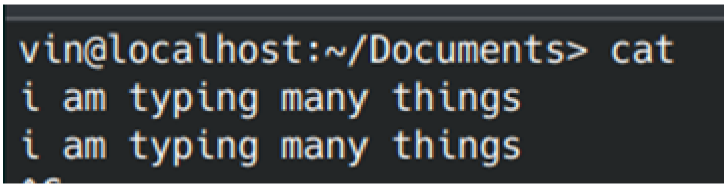
Windows uses its own slash convention. Some applications have several letters after the slash, and others allow you to precede arguments with a dash instead of a forward slash. However, these implementations are inconsistent. This is not to criticise Windows for its (many) shortcomings, but is to show 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 in most Linux programs: **stdin**, **stdout**, and **stderr**.

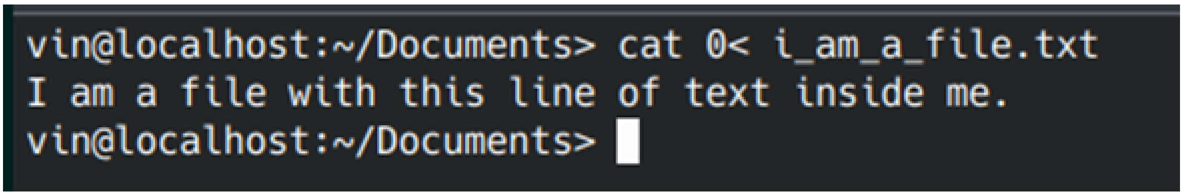
stdin

Usually, the stdin comes from the keyboard, a device file descriptor labelled `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. 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
```

As previously mentioned, we can also use an actual file to act as stdin.



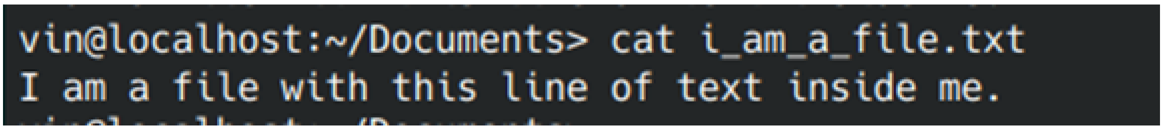
```
vin@localhost:~/Documents> cat 0< i_am_a_file.txt
I am a file with this line of text inside me.
vin@localhost:~/Documents> █
```

(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 `1`.

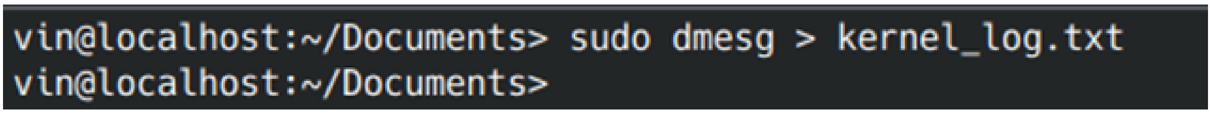
Going back to `cat`, our terminal is our stdout, so any output that is "written" to the `1` file should be displayed in our terminal window. Let's give **i_am_a_file.txt** as an argument to `cat`:



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

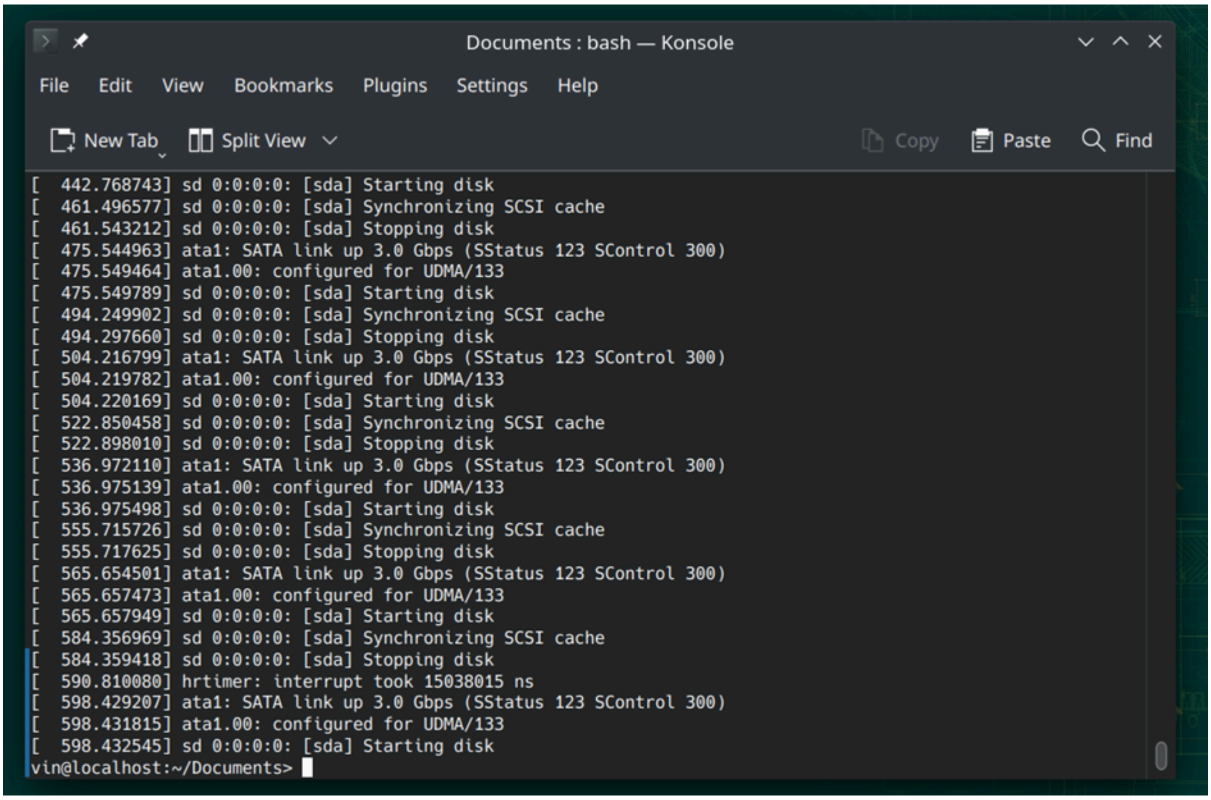
Predictably, we see the contents of this text file again.

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



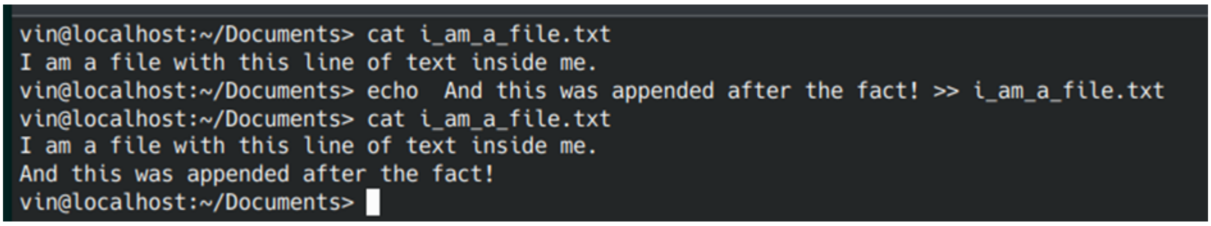
```
vin@localhost:~/Documents> sudo dmesg > kernel_log.txt
vin@localhost:~/Documents>
```

Nothing was written to the terminal window, because the file **kernel_log.txt** was our stdout, and everything that `dmesg` returned has been entered into 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> █
```

The screenshot above introduces a new command: `echo`. You can find out more about the `echo` command towards the end of this document, or by typing `man echo` into 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 similar to 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 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**.

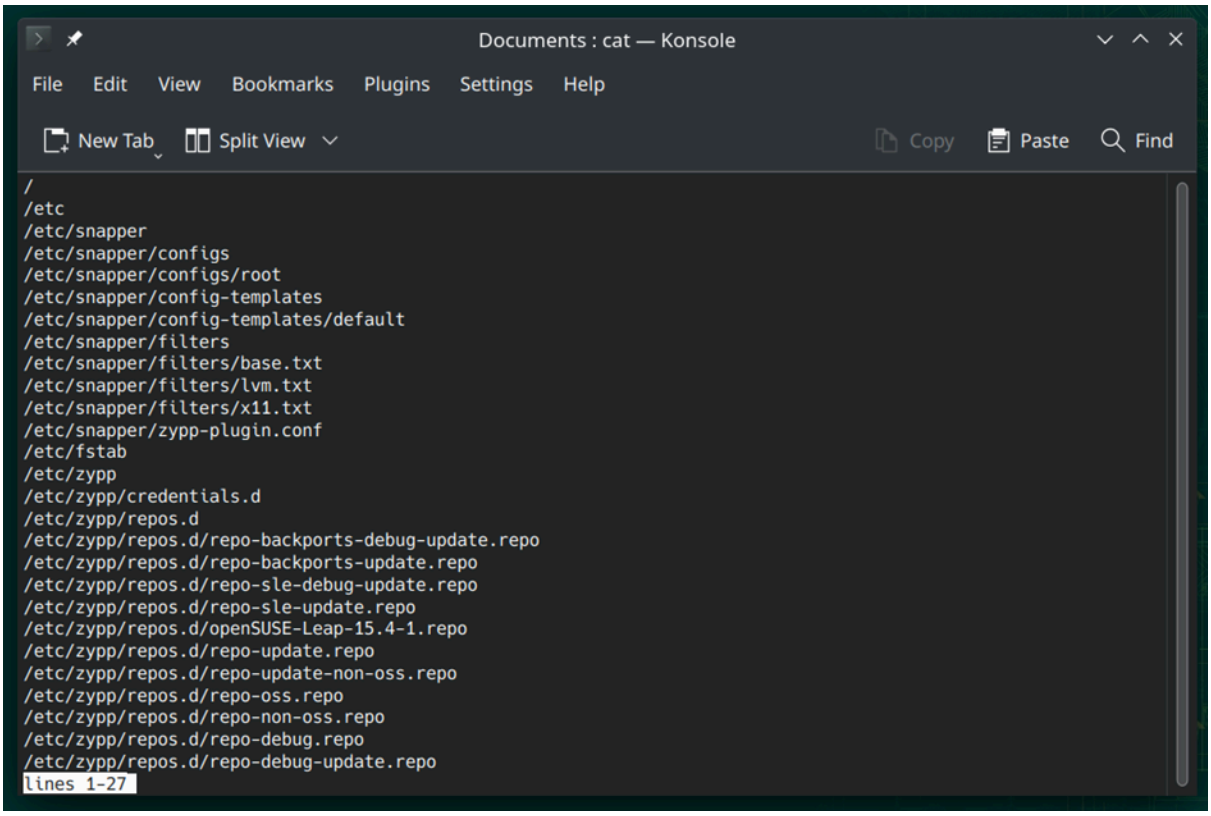
Piping

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

Let's take a look at the **output.txt** file we generated just now. If we gave 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?

Well, 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 either up or down. So, if we pipe (using the pipe symbol on your keyboard, |) cat's output to less, we can scroll through it line by line.

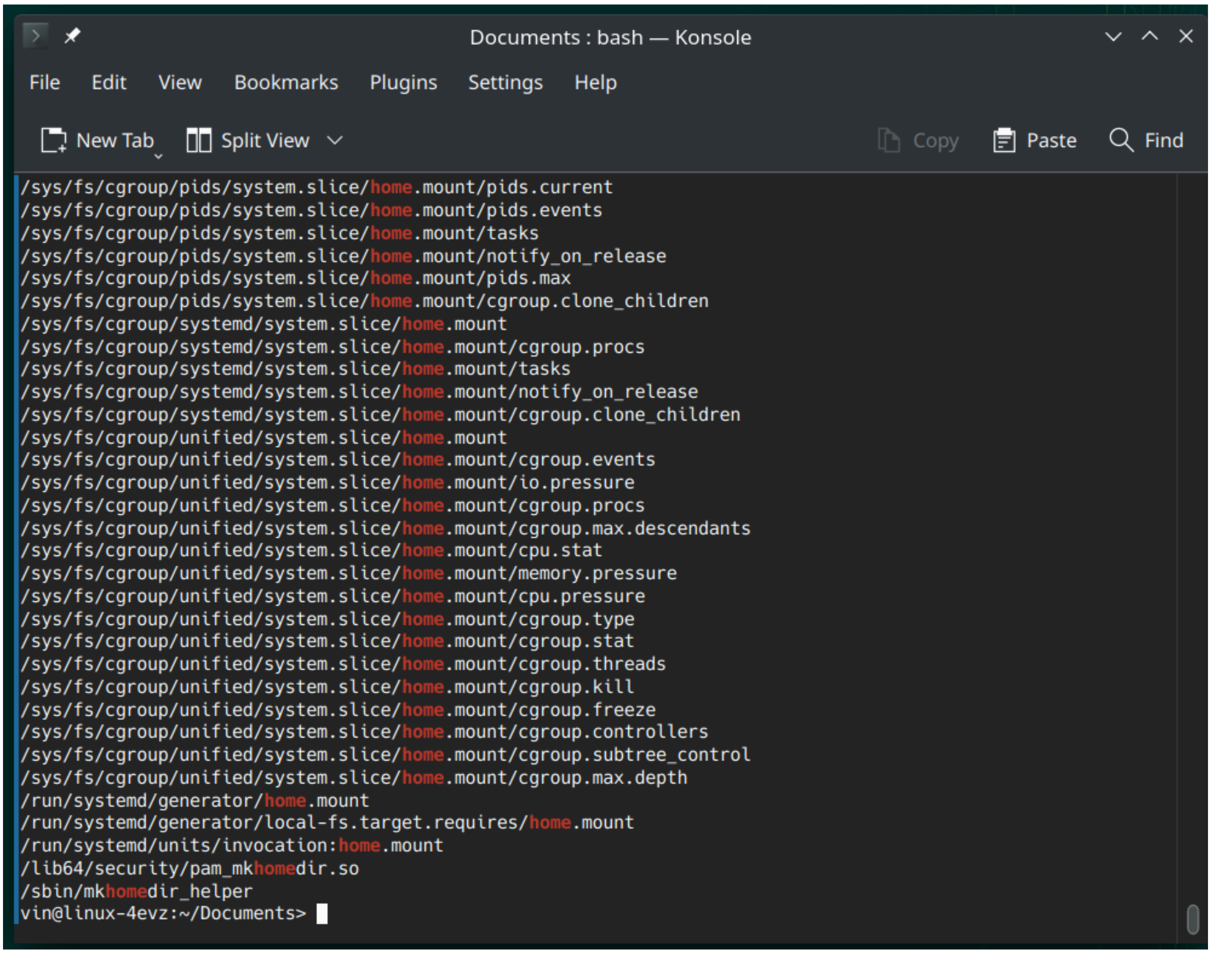
```
cat output.txt | less
```



```
/
/etc
/etc/snapper
/etc/snapper/configs
/etc/snapper/configs/root
/etc/snapper/config-templates
/etc/snapper/config-templates/default
/etc/snapper/filters
/etc/snapper/filters/base.txt
/etc/snapper/filters/lvm.txt
/etc/snapper/filters/x11.txt
/etc/snapper/zypp-plugin.conf
/etc/fstab
/etc/zypp
/etc/zypp/credentials.d
/etc/zypp/repos.d
/etc/zypp/repos.d/repo-backports-debug-update.repo
/etc/zypp/repos.d/repo-backports-update.repo
/etc/zypp/repos.d/repo-sle-debug-update.repo
/etc/zypp/repos.d/repo-sle-update.repo
/etc/zypp/repos.d/openSUSE-Leap-15.4-1.repo
/etc/zypp/repos.d/repo-update.repo
/etc/zypp/repos.d/repo-update-non-oss.repo
/etc/zypp/repos.d/repo-oss.repo
/etc/zypp/repos.d/repo-non-oss.repo
/etc/zypp/repos.d/repo-debug.repo
/etc/zypp/repos.d/repo-debug-update.repo
lines 1-27
```

What if we wanted to find entries that match the "home" pattern? For this purpose, we use grep, which filters through input and writes to stdout-only outputs that match a given pattern. If we tweak our command slightly:

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

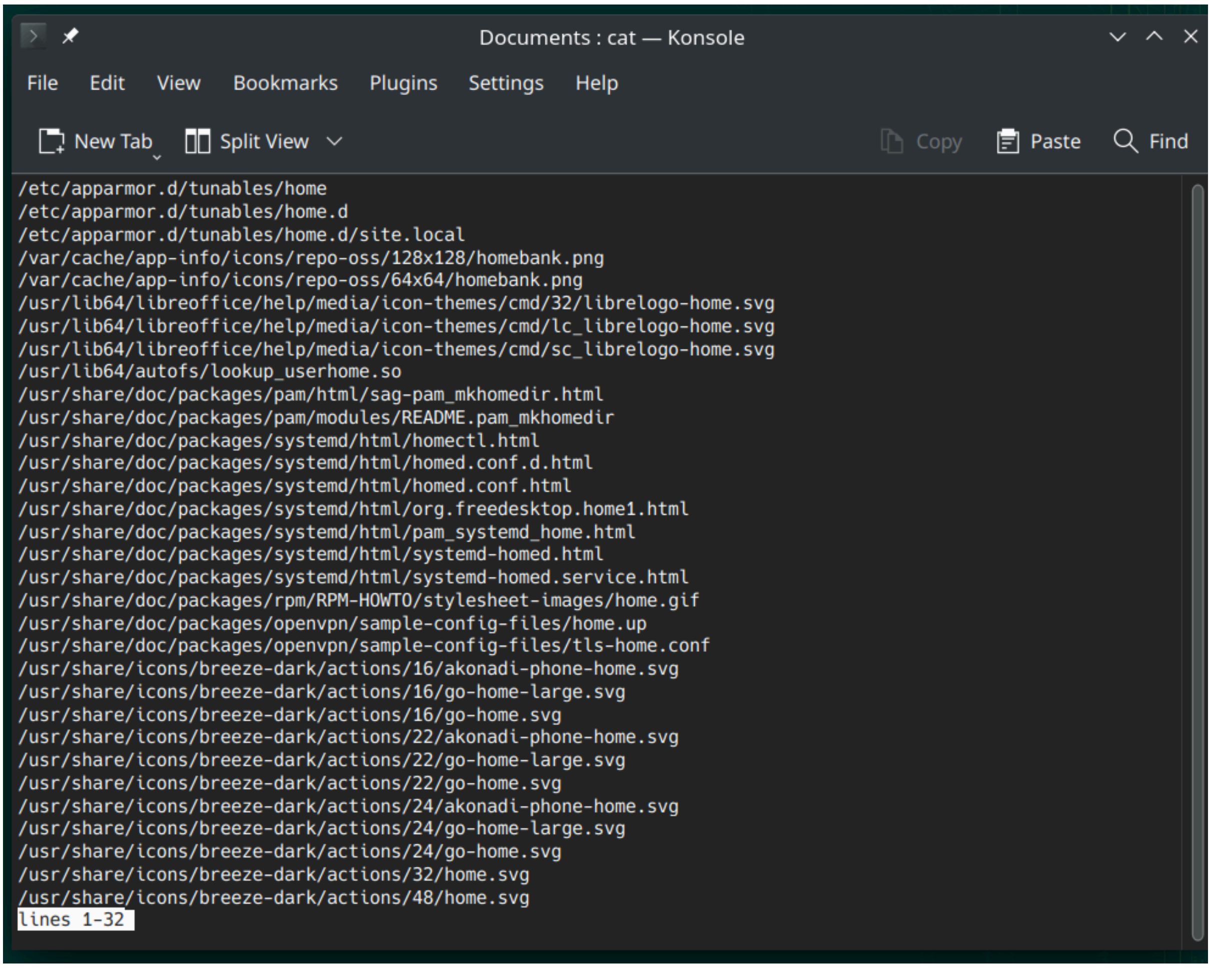


```
/sys/fs/cgroup/pids/system.slice/home.mount/pids.current
/sys/fs/cgroup/pids/system.slice/home.mount/pids.events
/sys/fs/cgroup/pids/system.slice/home.mount/tasks
/sys/fs/cgroup/pids/system.slice/home.mount/notify_on_release
/sys/fs/cgroup/pids/system.slice/home.mount/pids.max
/sys/fs/cgroup/pids/system.slice/home.mount/cgroup.clone_children
/sys/fs/cgroup/systemd/system.slice/home.mount
/sys/fs/cgroup/systemd/system.slice/home.mount/cgroup.procs
/sys/fs/cgroup/systemd/system.slice/home.mount/tasks
/sys/fs/cgroup/systemd/system.slice/home.mount/notify_on_release
/sys/fs/cgroup/systemd/system.slice/home.mount/cgroup.clone_children
/sys/fs/cgroup/unified/system.slice/home.mount
/sys/fs/cgroup/unified/system.slice/home.mount/cgroup.events
/sys/fs/cgroup/unified/system.slice/home.mount/io.pressure
/sys/fs/cgroup/unified/system.slice/home.mount/cgroup.procs
/sys/fs/cgroup/unified/system.slice/home.mount/cgroup.max.descendants
/sys/fs/cgroup/unified/system.slice/home.mount/cpu.stat
/sys/fs/cgroup/unified/system.slice/home.mount/memory.pressure
/sys/fs/cgroup/unified/system.slice/home.mount/cpu.pressure
/sys/fs/cgroup/unified/system.slice/home.mount/cgroup.type
/sys/fs/cgroup/unified/system.slice/home.mount/cgroup.stat
/sys/fs/cgroup/unified/system.slice/home.mount/cgroup.threads
/sys/fs/cgroup/unified/system.slice/home.mount/cgroup.kill
/sys/fs/cgroup/unified/system.slice/home.mount/cgroup.freeze
/sys/fs/cgroup/unified/system.slice/home.mount/cgroup.controllers
/sys/fs/cgroup/unified/system.slice/home.mount/cgroup.subtree_control
/sys/fs/cgroup/unified/system.slice/home.mount/cgroup.max.depth
/run/systemd/generator/home.mount
/run/systemd/generator/local-fs.target.requires/home.mount
/run/systemd/units/invocation:home.mount
/lib64/security/pam_mkhomedir.so
/sbin/mkhomedir_helper
vin@linux-4evz:~/Documents> 
```

Wait, now we can't scroll through our output anymore! Can we pipe the output we get from **grep** into yet a third command? Yes, if we tweak our command as follows:

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

HyperionDev



```
/etc/apparmor.d/tunables/home
/etc/apparmor.d/tunables/home.d
/etc/apparmor.d/tunables/home.d/site.local
/var/cache/app-info/icons/repo-oss/128x128/homebank.png
/var/cache/app-info/icons/repo-oss/64x64/homebank.png
/usr/lib64/libreoffice/help/media/icon-themes/cmd/32/librelogo-home.svg
/usr/lib64/libreoffice/help/media/icon-themes/cmd/lc_librelogo-home.svg
/usr/lib64/libreoffice/help/media/icon-themes/cmd/sc_librelogo-home.svg
/usr/lib64/autofs/lookup_userhome.so
/usr/share/doc/packages/pam/html/sag-pam_mkhomedir.html
/usr/share/doc/packages/pam/modules/README.pam_mkhomedir
/usr/share/doc/packages/systemd/html/homectl.html
/usr/share/doc/packages/systemd/html/homed.conf.d.html
/usr/share/doc/packages/systemd/html/homed.conf.html
/usr/share/doc/packages/systemd/html/org.freedesktop.home1.html
/usr/share/doc/packages/systemd/html/pam_systemd_home.html
/usr/share/doc/packages/systemd/html/systemd-homed.html
/usr/share/doc/packages/systemd/html/systemd-homed.service.html
/usr/share/doc/packages/rpm/RPM-HOWTO/stylesheet-images/home.gif
/usr/share/doc/packages/openvpn/sample-config-files/home.up
/usr/share/doc/packages/openvpn/sample-config-files/tls-home.conf
/usr/share/icons/breeze-dark/actions/16/akonadi-phone-home.svg
/usr/share/icons/breeze-dark/actions/16/go-home-large.svg
/usr/share/icons/breeze-dark/actions/16/go-home.svg
/usr/share/icons/breeze-dark/actions/22/akonadi-phone-home.svg
/usr/share/icons/breeze-dark/actions/22/go-home-large.svg
/usr/share/icons/breeze-dark/actions/22/go-home.svg
/usr/share/icons/breeze-dark/actions/24/akonadi-phone-home.svg
/usr/share/icons/breeze-dark/actions/24/go-home-large.svg
/usr/share/icons/breeze-dark/actions/24/go-home.svg
/usr/share/icons/breeze-dark/actions/32/home.svg
/usr/share/icons/breeze-dark/actions/48/home.svg
lines 1-32
```

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 book entitled *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 awareness of which tool does what will become, and you will find yourself looking commands up a lot less often. That said, here are some very commonly used commands as a quick reference guide.

- `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`: This deletes a file.
- `cp`: This copies a file.
- `mv`: This moves a file.
- `ls`: This lists everything in the working directory.
- `ln`: This creates a link from one file or folder to another.
- `find`: This looks for files in a directory structure.
- `which`: This shows you the full path of any command you may run. For instance, 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`: This changes permissions for a file system object.
- `chown`: This changes the owner of a file system object.
- `chgrp`: This changes the group ownership for a file system object.
- `su`: Substitute user. This lets you briefly log on as someone else.
- `sudo`: Substitute user do. This lets you 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`: This lists everyone who is logged in, and on which terminals. Normally, you should see only your account here.

Take note

Review the code template files that accompany this task before attempting the practical task. When you complete the task, use the templates.

Practical task

Note: Use the template files that accompany this task when you attempt this practical task.

This practical 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: 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 allows you to provide files as arguments, but also accept standard input. 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: A **regular expression** (regex) is a sequence of characters that specifies a pattern for matching text. The GNU implementation of `grep` does regex pattern matches. For this task, you need only do a simple exact matching system.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Share your thoughts

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

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

Share your thoughts anonymously using this **form**.