System Security

The Boring Introduction

Mohamed Sabt

Univ Rennes, CNRS, IRISA

2023 / 2024

Part I

Introduction

Diversity of Operating Systems

What is an Operating System?

- Operating System
  - is a system software which acts as an intermediary between user and hardware.
  - keeps the complicated details of the hardware hidden from the user and provides user with easy and simple interface.
  - performs functions which involves allocation of resources efficiently between user program, file system, Input Output device.

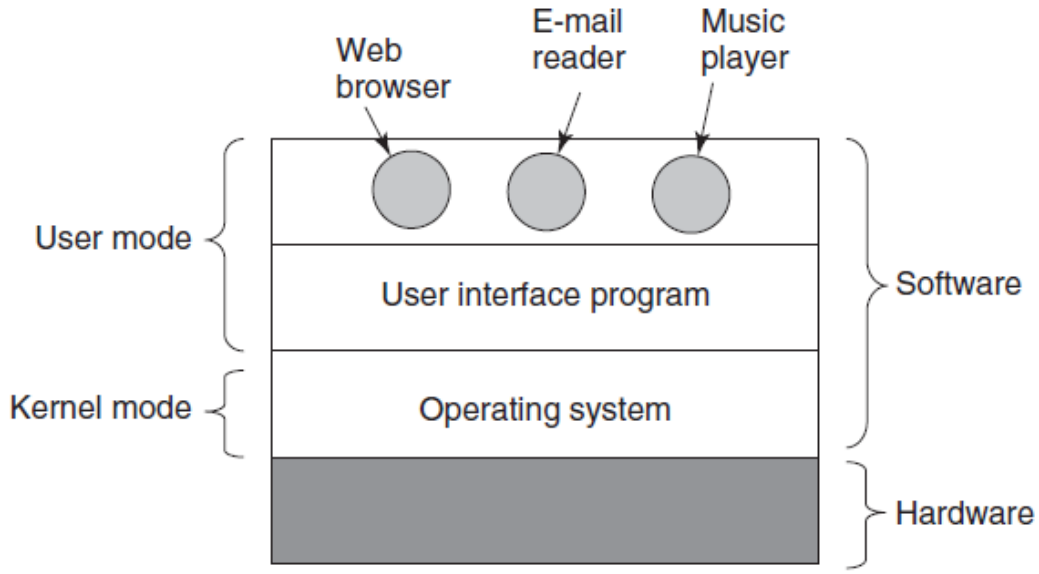
- Operating system goals:
  - Execute user programs.
  - Make the computer convenient to use.
  - Use the computer hardware in an efficient manner.

Computer System Components

- Hardware – provides basic computing resources
  - CPU, memory, I/O devices.
- Operating System
  - Controls and coordinates use of hardware among various applications and users.
- Application programs – define the ways in which the system resources are used to solve users computing problems.
  - Word processors, compilers, web browsers, video games.
- Users
  - People, machines, other computers.

Abstract View of Operating System

- The hardware components lie at the bottom of the diagram. It is considered as The most crucial part of computer system



- Operating system runs in the kernel mode of the system wherein the OS gets an access to all hardware and can execute all machine instructions. Other part of the system runs in user mode.

The OS as an Extended Machine

- The structure of computers system at the machine-language level is complicated to program, especially for input/output.
  - Programmers don't deal with hardware, so a level of abstraction is supposed to be maintained.
  - Abstraction allows a programs to create, write, and read files, without having to deal with the messy details of how the hardware actually works

- Good abstractions turn a nearly impossible task into two manageable ones.
  - The first is defining and implementing the abstractions.
  - The second is using these abstractions to solve the problem at hand.

The OS as a Resource Manager

- Modern computers consist of processors, memories, disks, network interfaces, IO, and a wide variety of other devices.
  - Operating system allows multiple programs to be in memory and run at the same time.
  - The operating system provides for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs.

- Resource management includes multiplexing (sharing) resources in two different ways: in time and in space.

- In time multiplexed, different programs takes turns using CPU. First one of them gets to use the resource, then the another, and so on.

- In space multiplexing, Instead of the customers taking turns, each one gets part of the resource.

Design Goals 1/2

- Abstraction
  - Make the system convenient to use.

- Performance
  - Minimize the overheads: both time and space.

- Protection
  - Isolating processes from one another.

Design Goals 2/2

- Reliability
  - Run non-stop.
- Security
  - An extension of protection.
  - Against malicious applications.
- Energy-efficiency.
- Mobility.

Part II

Brief History

Early Operating Systems

- In the beginning, the operating system was just a set of libraries of commonly-used functions, and thus made life easier for developers.
  - for example, instead of having each programmer of the system write low-level I/O handling code, the “OS” would provide such APIs.

- Computers, as of that point, were not used in an interactive manner.
  - **batch processing**, as a number of programs (or jobs) were set up and then run in groups, called a “batch”, by a human operator.

- Computers were known as mainframes and were kept in separate rooms.

The Era of Multiprogramming

- Multiprogramming became commonplace due to the desire to make better use of machine resources.

- Instead of just running one job at a time, the OS would load a number of jobs into memory and switch rapidly between them, thus improving CPU utilization.
  - This switching was particularly important because I/O devices were slow; having a program wait on the CPU while its I/O was being serviced was a waste of CPU time.

- **Memory protection** became important; we wouldn't want one program to be able to access the memory of another program.

The Advent of UNIX

- One of the major practical advances of the time was the introduction of the UNIX operating system.
  - UNIX took many good ideas from different operating systems (particularly from Multics from MIT), and made them simple.

- The UNIX environment was friendly for programmers and developers alike, also providing a compiler for the new C programming language.

- The spread of UNIX was slowed a bit as companies tried to assert ownership and profit from it.
  - Many companies had their own variants: **SunOS** from SunMicrosystems, **AIX** from IBM, etc.
  - The legal wrangling cast a dark cloud over UNIX, while Windows was introduced and took over much of the PC market.

The ACM Turing Award (UNIX)

- ACM is the association for Computing Machinery.
  - World's largest educational and scientific computer society.
  - You can become a student member too ***www.acm.org***.

- The ACM awards the Turing Award every year.

It is the "Nobel Prize" of computing.

- Names after British mathematician

Alain M. Turing (1912 -- 1954).

The Modern Era

- Beyond the minicomputer came a new type of machine, cheaper, faster, and for the masses:
  - the personal computer, or **PC** as we call it today.

- Led by Apple's early machines (e.g., the Apple II) and the IBM PC, this new breed of machine would soon become the dominant force in computing,
  - as their low-cost enabled one machine per desktop instead of a shared minicomputer per workgroup.

It was Better Before!

- Unfortunately, for operating systems, the PC at first represented a great leap backwards, as early systems forgot (or never knew of) the lessons learned in the era of minicomputers.

- **DOS** (the **Disk Operating System**, from **Microsoft**) didn't think memory protection was important;
  - thus, a malicious (or perhaps just a poorly-programmed) application could scribble all over memory.

- The first generations of the **Mac OS** (v9 and earlier) took a cooperative approach to job scheduling;
  - thus, a thread that accidentally got stuck in an infinite loop could take over the entire system, forcing a reboot.

And Then Came Linux

- Fortunately for UNIX, a young Finnish hacker named **Linus Torvalds** decided to write his own version of UNIX, but not from the code base, thus avoiding legal issues.
  - He took advantage of the sophisticated GNU tools that already existed

- As the internet era came into place, most companies (such as Google, Amazon, Facebook, and others) chose to run Linux.

- Steve Jobs took his UNIX-based **NeXTStep** operating environment with him to Apple, thus making UNIX popular on desktops

- As smart phones became a dominant user-facing platform, Linux found a stronghold there too (via Android).

Part III

Use-Case
Virtualizing Memory

Memory and Programs

- Memory is just an array of bytes;
  - to **read** memory, one must specify an **address** to be able to access the data stored there;
  - to **write** (or **update**) memory, one must also specify the data to be written to the given address.

- A program keeps all of its data structures in memory, and accesses them through various instructions,
  - like loads and stores or other explicit instructions that access memory in doing their work.

- Don’t forget that each instruction of the program is in memory too; thus memory is accessed on each instruction fetch.

Simple Example 1/2

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main(int argc, char *argv[]) {
    int * p = malloc(sizeof(int)); // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n", getpid(), p); // a2
    *p = 0; // a3

    while (1) {
        sleep(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p); // a4
    }
    return 0;
}
```

Simple Example 2/2

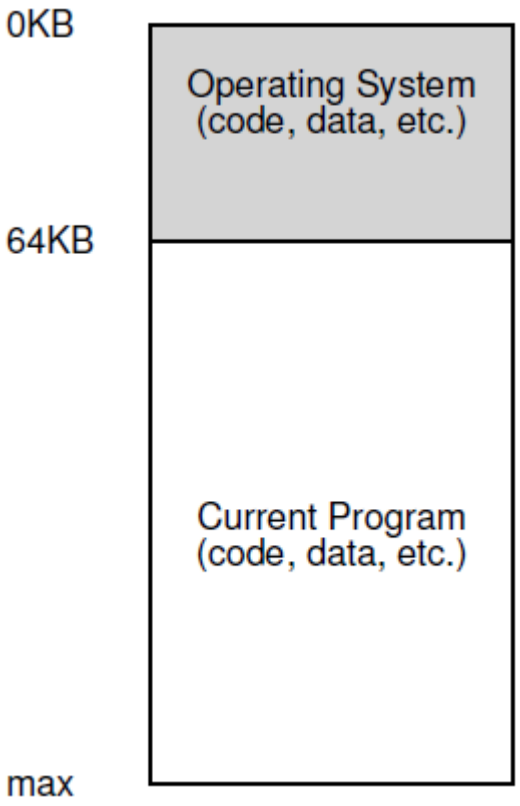
```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Example Explained

- We run multiple instances of this program to see what happens.
- We see from the example that each running program has allocated memory at the same address, and yet each seems to be updating the value independently!
  - It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs.
- Each process accesses its own private **virtual address space** (sometimes just called its **address space**), which the OS somehow maps onto the physical memory of the machine.
- A memory address within one running program does not affect the address space of other processes (or the OS itself); as far as the running program is concerned.
  - The reality, however, is that physical memory is a shared resource, managed by the operating system.

Memory in Early Systems

- From the perspective of memory, early machines didn't provide much of an abstraction to users. Basically, the physical memory of the looked something like what you see in Figure
  - The OS was a set of routines (a library, really) that sat in memory (starting at physical address 0 in this example), and there would be one running program (a process) that currently sat in physical memory (starting at physical address 64k in this example) and used the rest of memory.



Multiprogramming and Time Sharing

- After a time, because machines were expensive, the era of **multiprogramming** was born.
  - in which multiple processes were ready to run at a given time, and the OS would switch between them.

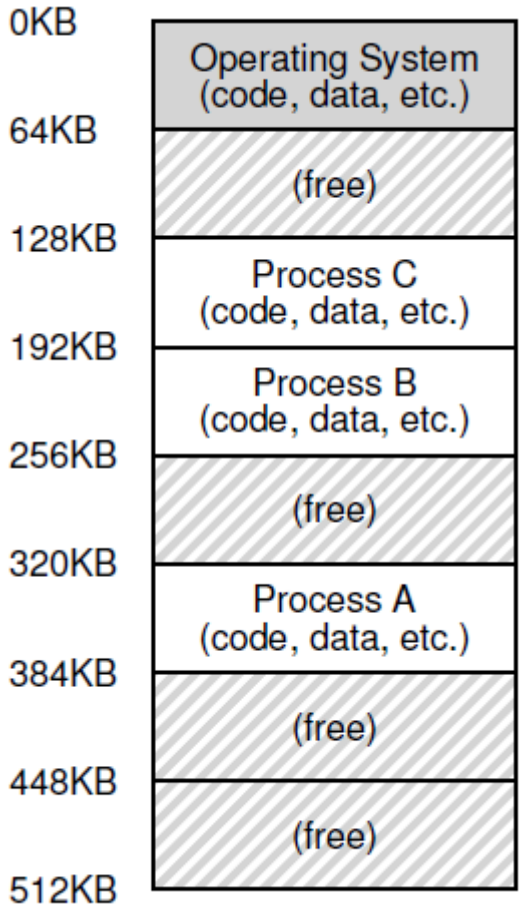
- Soon enough, however, people began demanding more of machines, and the era of **time sharing** was born.

- One way to implement time sharing would be to run one process for a short while, giving it full access to all memory, then stop it, save all of its state to some kind of disk (including all of physical memory), load some other process's state, run it for a while, etc.
  - Unfortunately, this approach has a big problem: it is way too slow, particularly as memory grows.

Sharing Memory

- In the diagram, there are three processes (A, B, and C) and each of them have a small part of the 512KB physical memory carved out for them.
  - Assuming a single CPU, the OS chooses to run one of the processes (say A), while the others (B and C) sit in the ready queue waiting to run.

- Multiple programs to reside concurrently in memory makes **protection** an important issue; you don't want a process to be able to read, or worse, write some other process's memory.

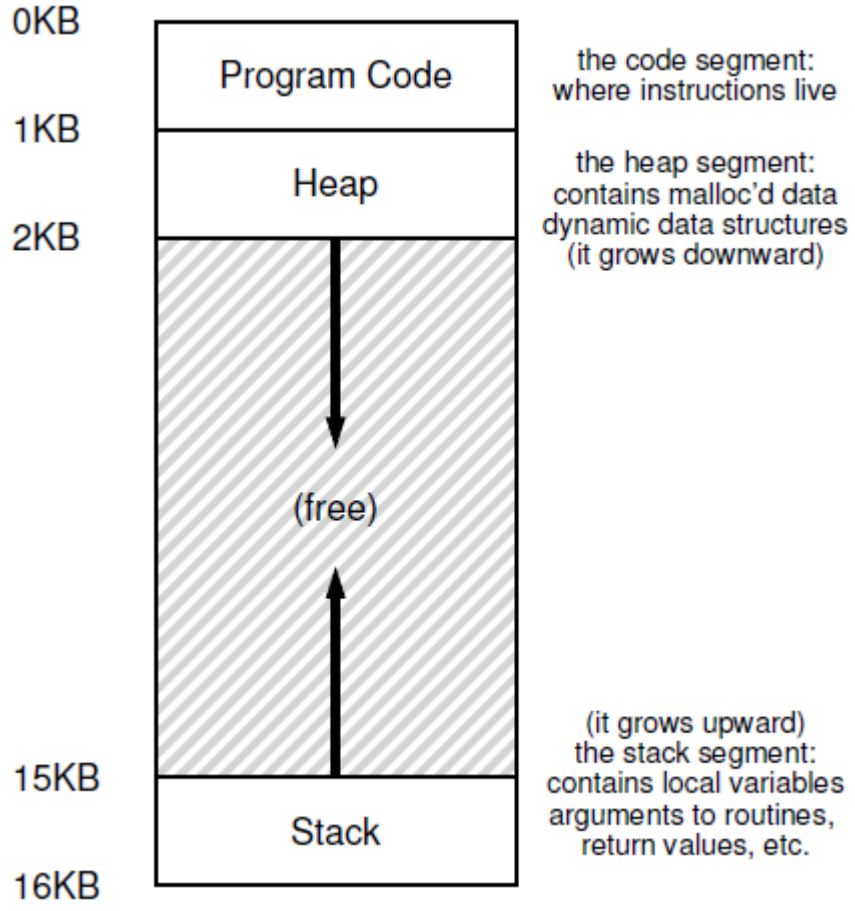


The Address Space

- The address space of a process contains all of the memory state of the running program.
  - Code.
  - Stack: for temporary execution state.
  - Heap: dynamically-allocated, user-managed memory.

- We say the OS is **virtualizing memory**, because the running program thinks it is loaded into memory at a particular address (say 0) and has a potentially very large address space.

- When, for example, process A tries to perform a load at address 0, somehow the OS, in tandem with some hardware support, will have to make sure the load doesn't actually go to physical address 0 but rather to physical address 320KB (where A is loaded into memory).



The Principle of Isolation

- Operating systems strive to isolate processes from each other and in this way prevent one from harming the other.
  - Isolation is a key principle in building reliable systems.

- By using memory isolation, the OS further ensures that running programs cannot affect the operation of the underlying OS.
  - When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect in any way the memory contents of any other process or the OS itself

- Some modern OS's take isolation even further, by walling off pieces of the OS from other pieces of the OS.
  - Such **microkernels** thus may provide greater reliability than typical monolithic kernel designs.

Part IV

The Unix Shell

Need for Shell

- The most widely used way to interact with personal computers is called a **graphical user interface** (GUI)

- The Unix shell is both a **command-line interface** (CLI) and a scripting language, allowing such repetitive tasks to be done automatically and fast.

What is a Shell?

- Shell is the human interface point for Unix.

- The shell is a program where users can type commands.

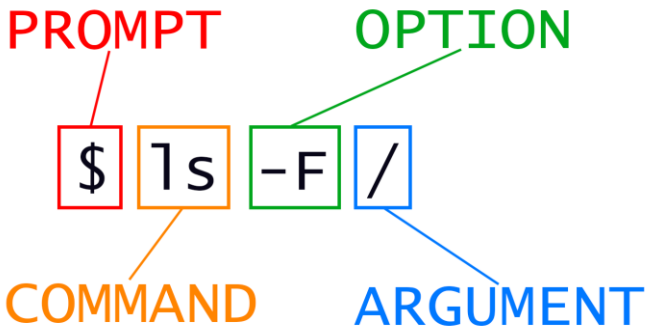
- Unix has provided different shells: they mainly differ in their scripting capabilities.
  - Bourne (sh).
  - Bourne Again (bash; the default one in most Linux distributions)
  - C shell (csh)
  - Z shell (zsh)

Shell Commands

- General form of a command:

  Example: ls -al or

  ls -a -l or ls -l -a



- Arguments can be optional or mandatory.
- Options can be short (single dash) or long (two dashes).
- All commands have a return code (0 if OK)
  - Read return code: echo $?
  - The return code can be used as part of control logic in shell scripts.

- All Unix commands have some documentation:
  - man command or man <number> command
  - Read the ‘man’ pages before the Stack Overflow ones.

RTFM

- Man pages are the original form of documentation that exist on Unix systems;
  - they were created before the thing called the **web** existed

- Spending some time reading man pages is a key step in the growth of a systems programmer; there are tons of useful tidbits hidden in those pages.

- RTFM - Read The Man pages.
  - The F in RT**F**M just adds a little color to the phrase.

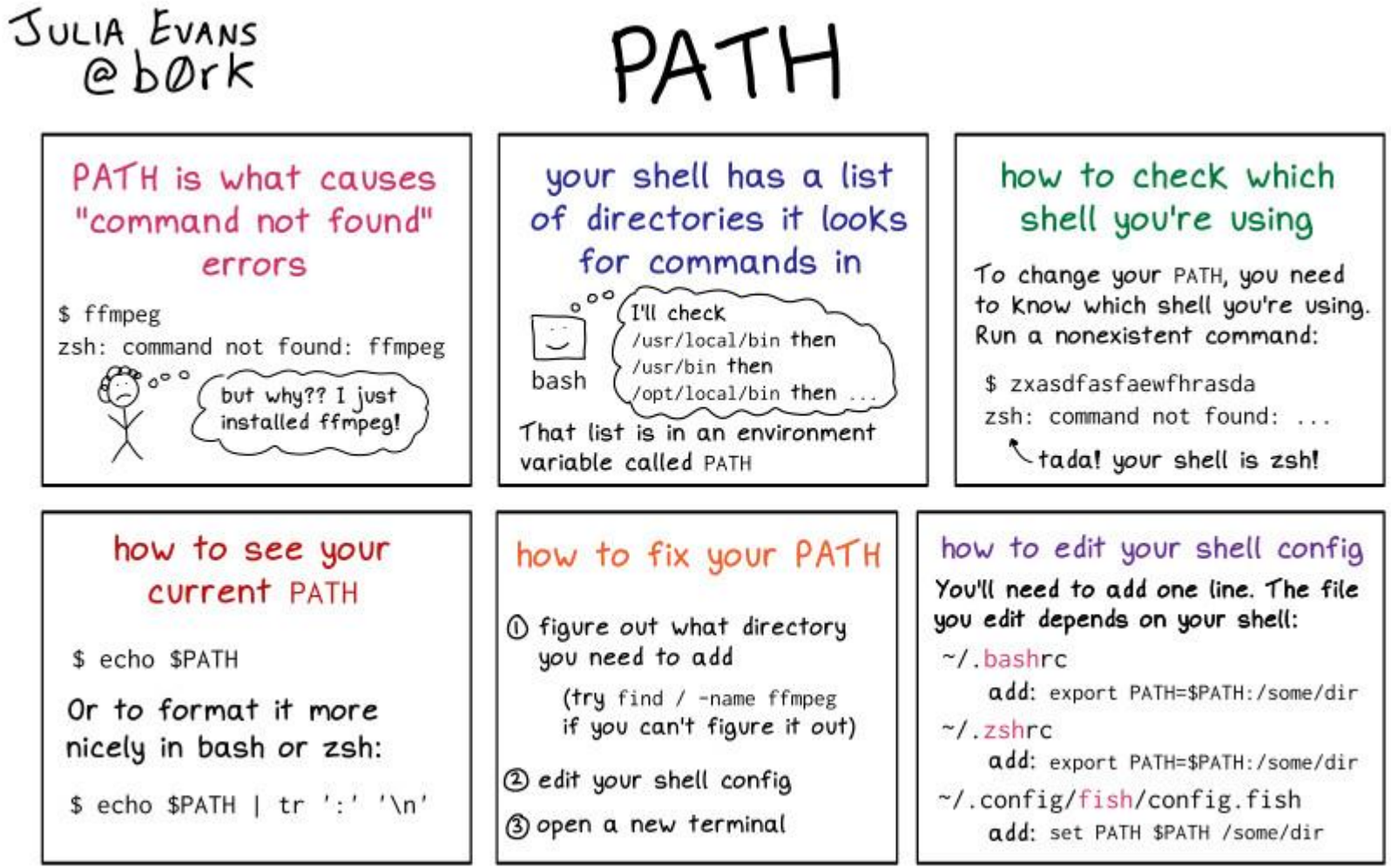
Locating Commands

- To execute a command, Unix has to locate the command before it can execute it.

- Unix uses the concept of search path to locate the commands.
- Search path is a list of directories in the order to be searched for locating commands
  - Usually it contains standard paths (/bin, /usr/bin, …)

- Modify the search path for your environment
  - Modifying the PATH environment variable.

PATH


JULIA EVANS
@b0rk
PATH
PATH is what causes "command not found" errors
$ ffmpeg
zsh: command not found: ffmpeg
but why?? I just installed ffmpeg!
your shell has a list of directories it looks for commands in
bash
I'll check /usr/local/bin then /usr/bin then /opt/local/bin then ...
That list is in an environment variable called PATH
how to check which shell you're using
To change your PATH, you need to know which shell you're using. Run a nonexistent command:
$ zxasdfasfaewfhrasda
zsh: command not found: ...
tada! your shell is zsh!
how to see your current PATH
$ echo $PATH
Or to format it more nicely in bash or zsh:
$ echo $PATH | tr ':' '\n'
how to fix your PATH
① figure out what directory you need to add
(try find / -name ffmpeg if you can't figure it out)
② edit your shell config
③ open a new terminal
how to edit your shell config
You'll need to add one line. The file you edit depends on your shell:
~/.bashrc
add: export PATH=$PATH:/some/dir
~/.zshrc
add: export PATH=$PATH:/some/dir
~/.config/fish/config.fish
add: set PATH $PATH /some/dir

Which

- Which can be used to find whether a particular command exists in your search path.
  - It does exist, which tells you which directory contains that command.

Bash Types

- Interactive:
  - Means that the commands are run with user-interaction from keyboard.
  - E.g. The shell can prompt the user to enter input.
- Non-interactive:
  - The shell is probably run from an automated process, so it cannot assume if it can request input or that someone will see the output.
  - E.g. Maybe it is best to write output to a log-file.
- Login:
  - Shell is run as part of the login of the user to the system.
  - Typically used to do any configuration that a user wants/needs to establish their work environment.
- Non-login:
  - Any other shell run by the user after logging on.

What is a Script

- A script is a list of system commands stored in a file.

- Advantages:
  - To avoid having to retype them again and again.
  - To be able to modify and customize the script for a particular application.
  - To use the script as a program/command.

Execute the Script

- The script execution requires the script has “execute permissions”.
  - chmod +rx scriptname (gives everyone read/execute permission)
  - chmod u+rx scriptname (gives only the owner read/execute permission)

- The script can be executed issuing:
  - ./scriptname
- The script can be made available as a command:
  - Moving the script to /usr/local/bin (as root), making it available to all users as a system wide executable.
  - Including the directory containing the script in the user’s $PATH.

The Sha-bang #!

- Every script starts with the sha-bang (#!) at the head, followed by the full path name of an interpreter:
  - #!/bin/bash
  - #!/usr/bin/perl

- This tells your system that the file is a set of commands to be fed to the command interpreter indicated by the path.

- The #! Is a special marker that designates a file type, or in this case an executable shell script (type **man magic** for more details).

Special characters 1/2

- Comments [#]:
  - Lines beginning with a #.
  - Following the end of a command.

- Command separator [semicolon ;]
  - Permits putting two or more commands on the same line.

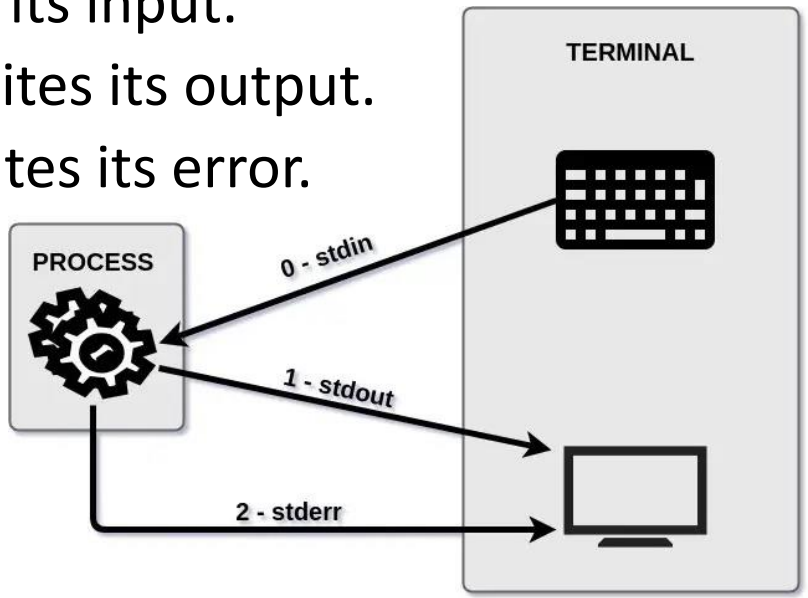
- Escape [backslash \]
  - Expresses literally a special character.
  - Example: echo This is a double quote \”

Special characters 2/2

- Command substitution [backquotes `]:
  - The `command` constructs makes available the output of command for assignment to a variable.
  - Example: a=`pwd`
- Wild card [asterisk].
  - It matches every filename in a given directory or every character in a string.
- Run job in background [and &]
  - A command followed by & will run in the background.
  - To bring the script in foreground, type fg or CTRL Z fg

Redirection 1/3

- Each Unix command is connected to three communication channels between the command and the environment:
  - Standard input (stdin) where the command reads its input.
  - Standard output (stdout) where the command writes its output.
  - Standard output (stderr) where the command writes its error.



- When a command is executed via an Interactive shell, the streams are typically connected to the text terminal on which the shell is running.

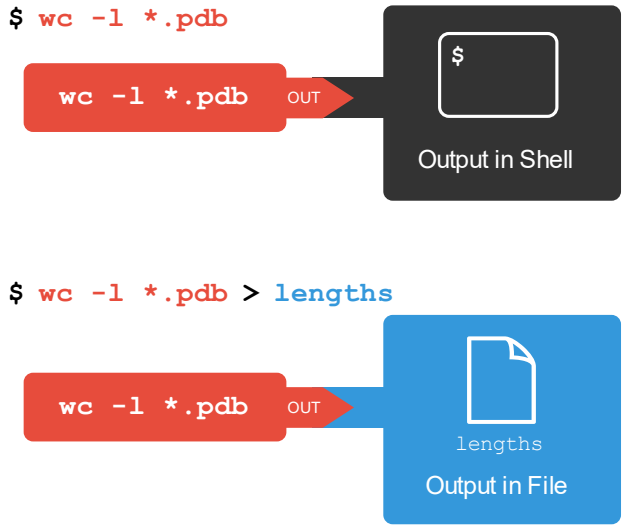
Redirection 2/3

- Redirect stdout to file (creates the file if it does not already exist):
  - command > filename (overwrite filename if it already exists)
  - command >> filename (appends the output to filename if it already exists)

- Redirect stderr to file
  - command 2> filename
- Redirect both the stdout and the stderr to file
  - command &>
- Redirect stdout to stderr
  - command >&2
- Redirect stderr to stdout
  - command 2>&1

Redirection 3/3

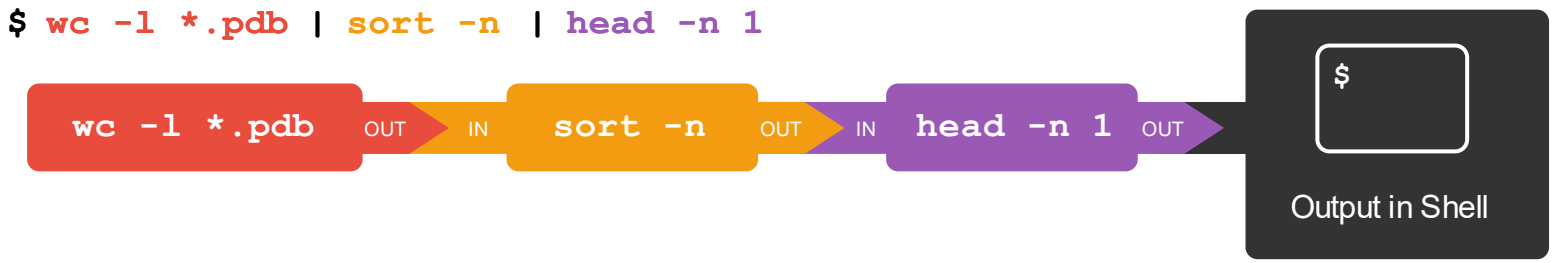
- Example: Discard any errors that are generated by command
  - command 2> /dev/null



Pipes [|]

- Passes the output (stdout) of a previous command to the input (stdin) of the next one, or to the shell.
  - Allows chaining commands together.

- Examples:
  - cat *.java | sort | uniq
  - ls –l | wc -l
  - cat filename | grep search_word (or just grep search_word filename)



Bash Variables 1/2

- Variables are how languages represent data.
  - A variable is a label, a name assigned to a location holding data.
  - Variables can be set: by the system, by you, by the shell, by any program that loads another program.

- Standard Unix variables are split into two categories:
  - Environment variables: if set at login, are valid for the duration of the session.
  - Shell variables: apply only to the current instance of the shell.

- By convention, environment variables have UPPER CASE and shell variables have lower case names.
- Environment variables are a way of passing information from the shell to programs when you run them.
  - Programs look in the environment for variables, and if found, will use the variables stored.

Bash Variables 2/2

- Bash variables are untyped.
  - They can contain a number, a character, a string of characters, arrays, etc.

- There is no need to declare a variable, just assigning a value to its reference will create it.

Bash Variables: Assignment

- If variable1 is the name of a variable, then $variable1 is a reference to its value (i.e. the data item it contains).

- $variable1 is actually a simplified form of ${variable1}.
  - In contexts where the $variable syntax causes an error, the longer form ${variable} should work (this is called variable disambiguation).
- Reference (retrieving) the variable value is called variable substitution.
- No space permitted on either side of '=' sign when initializing variables.
  - What happens if there is a space? Bash will treat the variable name as a program to execute, and the '=' as its first parameter.

Bash Variables: Quoting

- Quoting means just bracketing a string in quotes.
- This has the effect of protecting special characters in the string from reinterpretation or expansion by the shell.

- Partial quoting (weak quoting) consists in enclosing a referenced value in double quotes (" … ").
  - This does not interfere with variable substitution.

- Full quoting consists in using single quotes (' … ').
  - It causes the variable name to be used literally, and no substitution will take place.

Array in Bash

- Initialize an array: arrays in Bash can contain both numbers and strings.

- Initialization with all elements of the same type (numbers).
  - myArray=(1 2 3 4 5 6)

- Initialization with mixed types elements
  - myArray=(1 2 “three” 4 “five” 6)

Retrieve Array Elements in Bash

- Although Bash variables do not generally require curly brackets, they are required for arrays.
- In turn, this allows Bash to specify the index to access.
  - echo ${myArray[1]} returns the second element of the array.

- Not including brackets
  - echo $myArray[1] leads Bash to treat [1] as a string and output it as such.

Useful Array Operations

Syntax	Result
arr=()	Create an empty array
arr=(1 2 3)	Initialize array
${arr[1]}	Retrieve second element
${arr[@]}	Retrieve all elements
${#arr[@]}	Retrieve array size
arr+=(4)	Append values(s)
str=$(ls)	Save ls output as a string
arr=($(ls))	Save ls output as an array of files
${arr[@]:s:n}	Retrieve n elements starting at index s

Bash Arithmetic Expansion 1/3

- Arithmetic expansion provides a powerful tool for performing (integer) arithmetic operations.
  - Translating a string into a numerical expression is relatively straightforward using `expr`.

- Example
  - z=1
  - z=`expr $z + 3`
  - echo $z

Bash Arithmetic Expansion 2/3

- Parentheses
  - ((EXPRESSION)) is arithmetic expansion.

- Example
  - z=0
  - ((z+=1)) # (($z+=1)) is wrong
  - echo $z

Bash Arithmetic Expansion 3/3

- `let` does exactly what (()) do.

- Example
  - z=0
  - let z=z+3
  - let “z += 3”
  - echo $z

- Quotes permit the use of spaces in variable assignment.

Debugging

- The shell gives us tools for debugging scripts. If we want to run a script in debug mode, we use a special option in our script shebang

```
1  #!/bin/bash -x
2
3  for (( i = 0; i < 3; i++ )); do
4    echo $i
5  done
6
```

```
$ ./script.sh
+ (( i = 0 ))
+ (( i < 3 ))
+ echo 0
0
+ (( i++  ))
+ (( i < 3 ))
+ echo 1
1
+ (( i++  ))
+ (( i < 3 ))
+ echo 2
2
+ (( i++  ))
+ (( i < 3 ))
```