An Introduction to Linux

Slides available at

umuzi.gitlab.io/linux-slides

What is Linux?

“Linux is a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX and Single UNIX Specification compliance.”

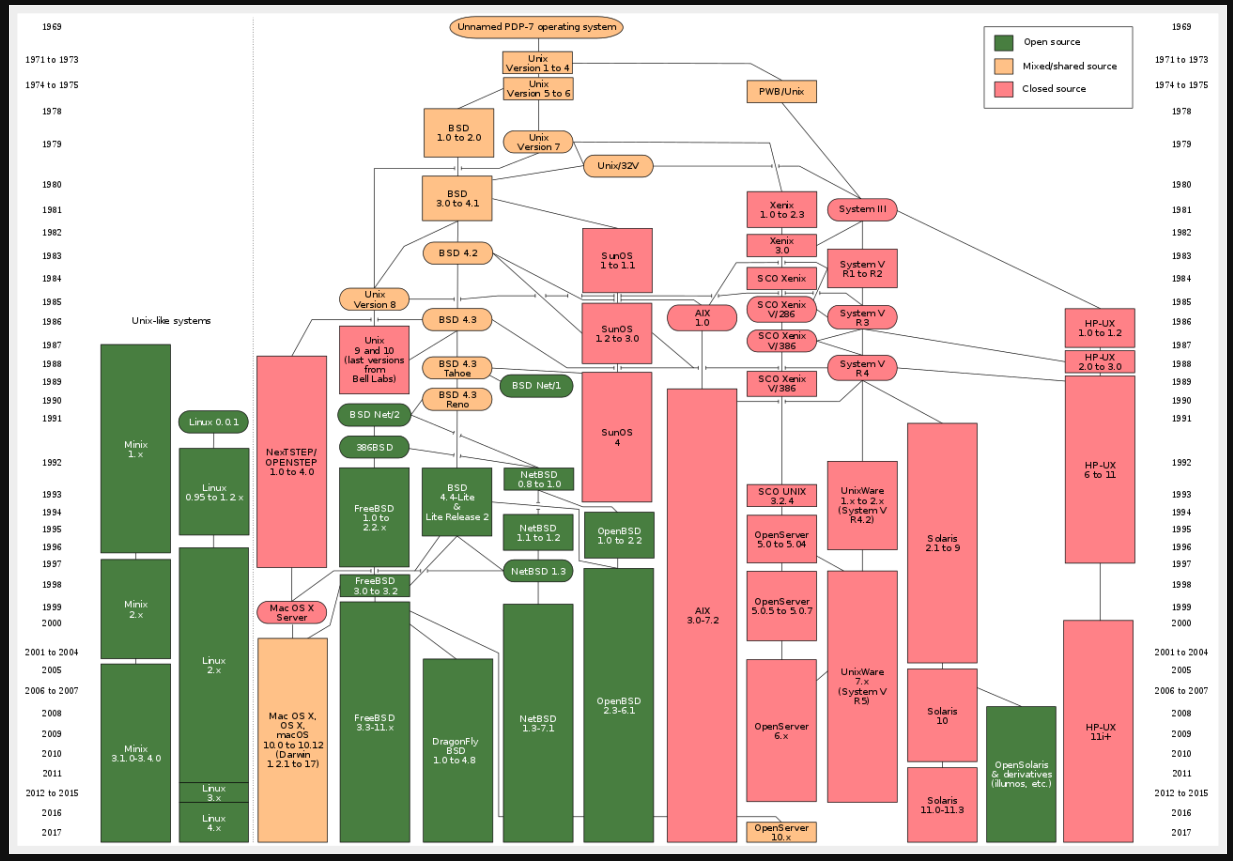
kernel.org

“Linux is a clone of the operating system Unix…”

- Family of operating systems developed in the 1970s at Bell Labs
- Established important design principles, *the Unix Philosophy*
  - Each program should do one thing, and do it well
  - Expect the output of every program to become the input of another
  - Programs should handle human readable text

It's a Unix system

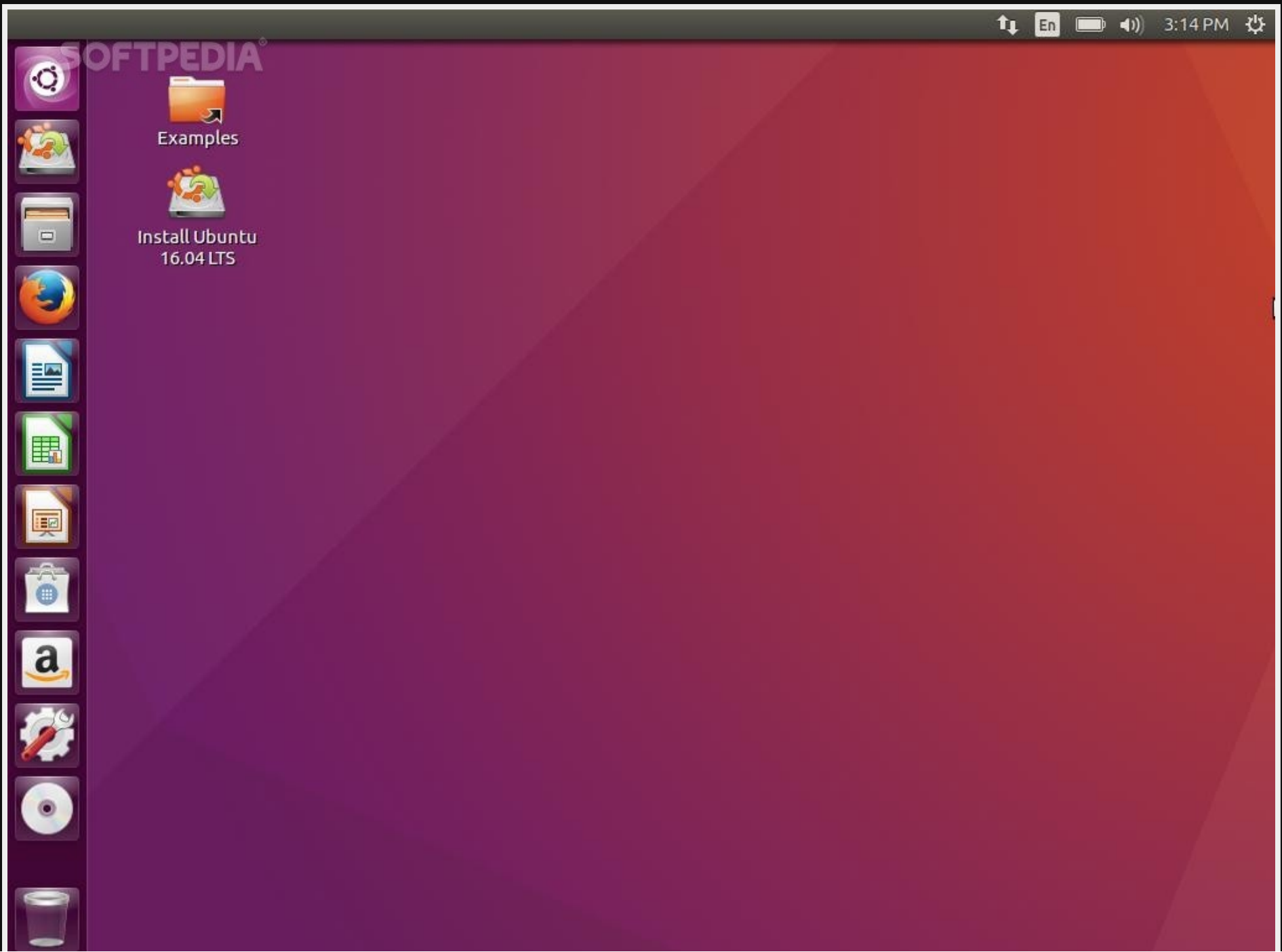
“Linux is a clone of the operating system Unix…”



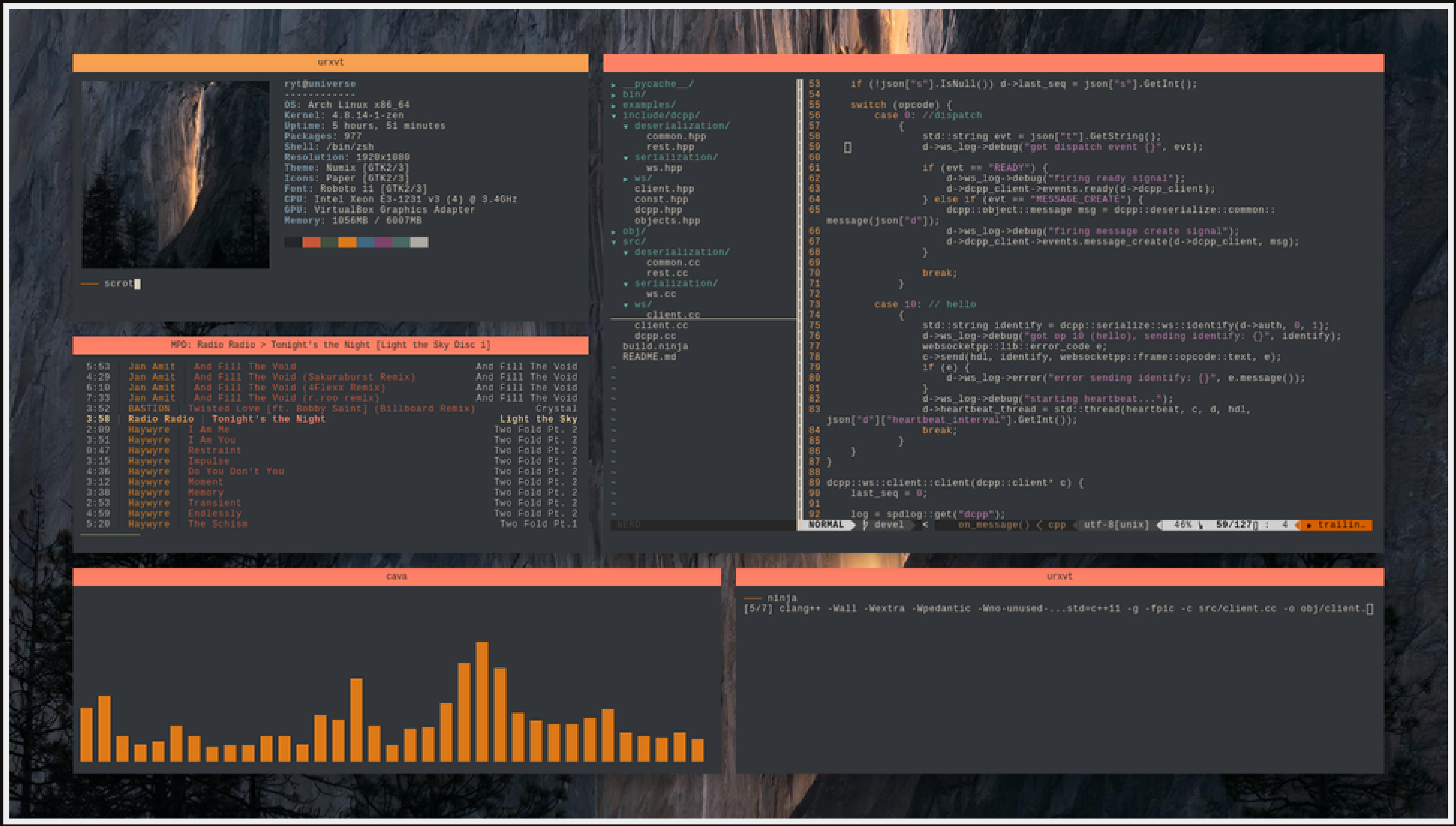
Market Share

Distributions

- Linux by itself does not give you a working system, you need userspace software
- Linux exists to the user as many different distributions: Debian, Ubuntu, CentOS, Fedora, Arch, ... (and hundreds of others)


En
3:14 PM
SOFTPEDIA
Examples
Install Ubuntu
16.04 LTS


SOFTPEDIA
Computer
Home
Install Linux Mint
LinuXMint
17.3
Menu
19:51


urxvt
ryt@universe
OS: Arch Linux x86_64
Kernel: 4.8.14-1-zen
Uptime: 5 hours, 51 minutes
Packages: 977
Shell: /bin/zsh
Resolution: 1920x1080
Theme: Numix [GTK2/3]
Icons: Paper [GTK2/3]
Font: Roboto 11 [GTK2/3]
CPU: Intel Xeon E3-1231 v3 (4) @ 3.4GHz
GPU: VirtualBox Graphics Adapter
Memory: 1056MB / 6007MB
scrot
MPD: Radio Radio > Tonight's the Night [Light the Sky Disc 1]
5:53 Jan Amit And Fill The Void And Fill The Void
4:29 Jan Amit And Fill The Void (Sakuraburst Remix) And Fill The Void
6:10 Jan Amit And Fill The Void (4Flexx Remix) And Fill The Void
7:33 Jan Amit And Fill The Void (r.roo remix) And Fill The Void
3:52 BASTION Twisted Love [ft. Bobby Saint] (Billboard Remix) Crystal
3:58 Radio Radio Tonight's the Night Light the Sky
2:09 Haywyre I Am Me Two Fold Pt. 2
3:51 Haywyre I Am You Two Fold Pt. 2
0:47 Haywyre Restraint Two Fold Pt. 2
3:15 Haywyre Impulse Two Fold Pt. 2
4:36 Haywyre Do You Don't You Two Fold Pt. 2
3:12 Haywyre Moment Two Fold Pt. 2
3:38 Haywyre Memory Two Fold Pt. 2
2:53 Haywyre Transient Two Fold Pt. 2
4:59 Haywyre Endlessly Two Fold Pt. 2
5:20 Haywyre The Schism Two Fold Pt.1
__pycache__/
bin/
examples/
include/dcpp/
deserialization/
common.hpp
rest.hpp
serialization/
ws.hpp
ws/
client.hpp
const.hpp
dcpp.hpp
objects.hpp
obj/
src/
deserialization/
common.cc
rest.cc
serialization/
ws.cc
ws/
client.cc
client.cc
dcpp.cc
build.ninja
README.md
if (!json["s"].IsNull()) d->last_seq = json["s"].GetInt();
switch (opcode) {
case 0: //dispatch
std::string evt = json["t"].GetString();
d->ws_log->debug("got dispatch event {}", evt);
if (evt == "READY") {
d->ws_log->debug("firing ready signal");
d->dcpp_client->events.ready(d->dcpp_client);
} else if (evt == "MESSAGE_CREATE") {
dcpp::object::message msg = dcpp::deserialize::common::message(json["d"]);
d->ws_log->debug("firing message create signal");
d->dcpp_client->events.message_create(d->dcpp_client, msg);
break;
case 10: // hello
std::string identify = dcpp::serialize::ws::identify(d->auth, 0, 1);
d->ws_log->debug("got op 10 (hello), sending identify: {}", identify);
websocketpp::lib::error_code e;
c->send(hdl, identify, websocketpp::frame::opcode::text, e);
if (e) {
d->ws_log->error("error sending identify: {}", e.message());
d->ws_log->debug("starting heartbeat...");
d->heartbeat_thread = std::thread(heartbeat, c, d, hdl, json["d"]["heartbeat_interval"].GetInt());
break;
dcpp::ws::client::client(dcpp::client* c) {
last_seq = 0;
log = spdlog::get("dcpp");
NORMAL devel on_message() cpp utf-8[unix] 46% 59/127 4 trailin…
cava
urxvt
ninja
[5/7] clang++ -Wall -Wextra -Wpedantic -Wno-unused-...std=c++11 -g -fpic -c src/client.cc -o obj/client.

```
[root@SimCraftWorkShop ~]# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.2  19236  1464 ?        Ss   06:35   0:00 init
root          2  0.0  0.0      0     0 ?        S    06:35   0:00 [kthreadd/8009]
root          3  0.0  0.0      0     0 ?        S    06:35   0:00 [khelper/80096]
root        124  0.0  0.0  10664   572 ?        S<s  06:36   0:00 /sbin/udevd -d
root        460  0.0  0.5 183752  3268 ?        Sl   06:36   0:02 /sbin/rsyslogd
dbus        481  0.0  0.1  21408   916 ?        Ss   06:36   0:00 dbus-daemon --s
68          490  0.0  0.3  37224  2316 ?        Ssl  06:36   0:00 hald
root        491  0.0  0.2  20400  1300 ?        S    06:36   0:00 hald-runner
root        533  0.0  0.1  64372  1148 ?        Ss   06:36   0:02 /usr/sbin/sshd
root        540  0.0  0.1  22096   952 ?        Ss   06:36   0:00 xinetd -stayali
root        547  0.0  0.1  64568   936 ?        Ss   06:36   0:00 /usr/sbin/sasla
root        549  0.0  0.1  64568   672 ?        S    06:36   0:00 /usr/sbin/sasla
root        563  0.0  0.4  80816  2364 ?        Ss   06:36   0:00 sendmail: accep
smmsp       571  0.0  0.3  76412  2080 ?        Ss   06:36   0:00 sendmail: Queue
root        586  0.0  0.2 117216  1228 ?        Ss   06:36   0:00 crond
root        606  0.0  0.1   4068   592 tty1     Ss+  06:36   0:00 /sbin/mingetty
root        607  0.0  0.1   4068   596 tty2     Ss+  06:36   0:00 /sbin/mingetty
root      13756  0.0  0.6  94040  3868 ?        Ss   18:43   0:00 sshd: minecraft
500       13758  0.0  0.3  94040  1924 ?        S    18:43   0:01 sshd: minecraft
500       13759  0.0  0.3  55592  2264 ?        Ss   18:43   0:00 /usr/libexec/op
root      13770  0.0  0.7  94016  4432 ?        Ss   18:44   0:00 sshd: root@pts/
root      13774  0.0  0.3 108304  1900 pts/0    Ss   18:44   0:00 -bash
root      28736  0.3  0.6  92668  3888 ?        Ss   20:07   0:00 sshd: root [pri
sshd      28737  0.0  0.2  65716  1636 ?        S    20:07   0:00 sshd: root [net
root      28738  0.0  0.1 110236  1152 pts/0    R+   20:07   0:00 ps aux
[root@SimCraftWorkShop ~]# service factorio help
Usage: /etc/init.d/factorio COMMAND

Available commands:
   start                Starts the server
   stop                 Stops the server
   restart              Restarts the server
   status               Displays server status
   load-save [name]             Loads the specified save
   screen               Shows the server screen
[root@SimCraftWorkShop ~]# service factorio load-save server
DEBUG LOG: no pidfile found
DEBUG LOG: could not find a pid with invocation: "/home/minecraft/factorio/bin/x64/factorio --star
t-server server --autosave-interval 10"
```

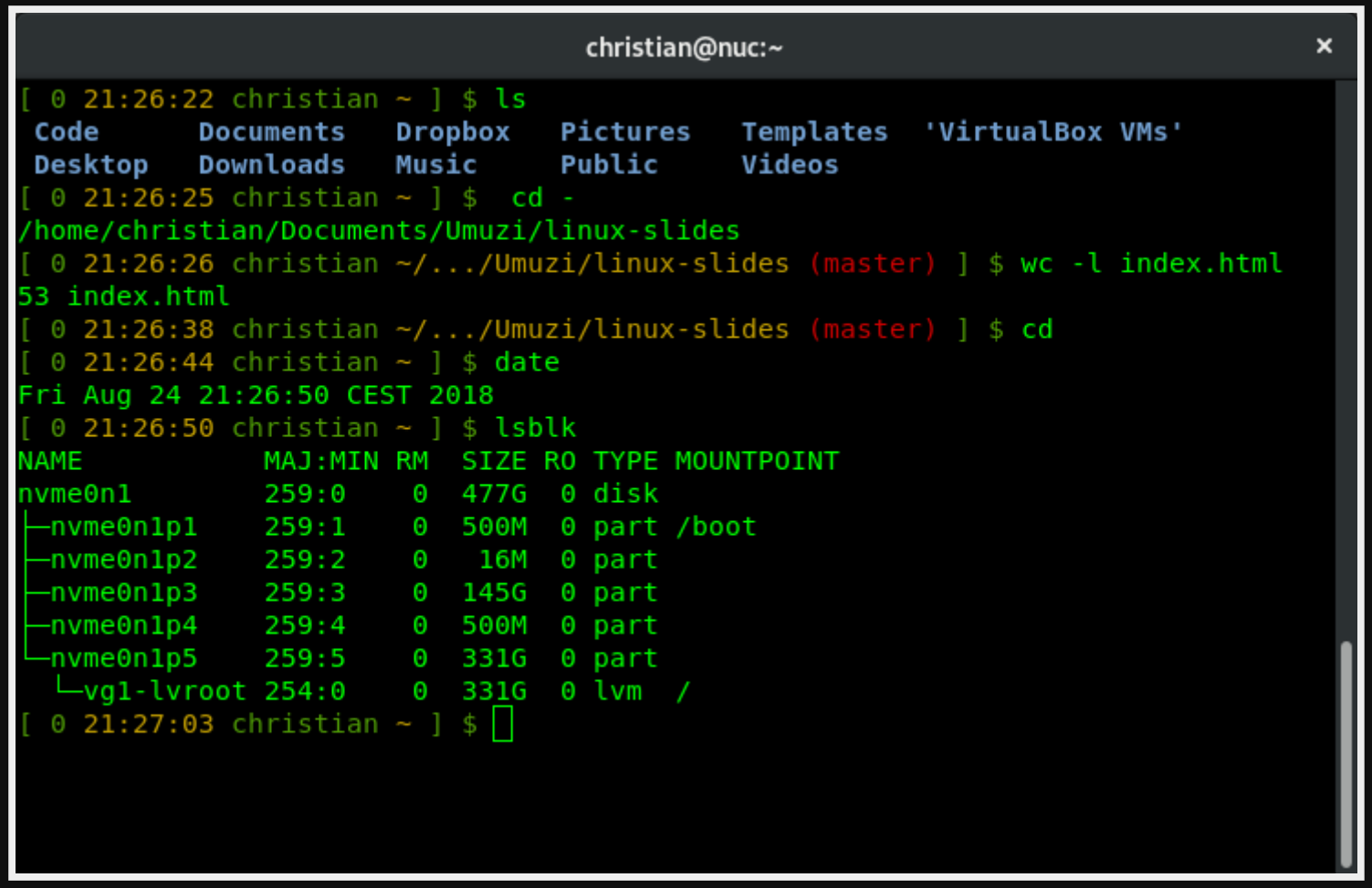
“It aims towards POSIX and Single UNIX Specification compliance.”

- After Unix moved out of research labs, standards were made that define how a Unix operating system works (programming interface, services, file system...)
- IEEE created the Portable Operating System Interface (POSIX) standard
- The Austin Group created the Single UNIX Specification standard
- These two are now one and the same
- Several linux distributions/Linux foundation have jointly created Linux Standard Base (LSB), which largely subsumes POSIX/SuS

Further Reading:

Linux Took Over The Web. Now, It's Taking Over the World (Wired, Aug 2016)

The Shell


christian@nuc:~
[0 21:26:22 christian ~] $ ls
Code Documents Dropbox Pictures Templates 'VirtualBox VMs'
Desktop Downloads Music Public Videos
[0 21:26:25 christian ~] $ cd -
/home/christian/Documents/Umuzi/linux-slides
[0 21:26:26 christian ~/.../Umuzi/linux-slides (master)] $ wc -l index.html
53 index.html
[0 21:26:38 christian ~/.../Umuzi/linux-slides (master)] $ cd
[0 21:26:44 christian ~] $ date
Fri Aug 24 21:26:50 CEST 2018
[0 21:26:50 christian ~] $ lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
nvme0n1 259:0 0 477G 0 disk
├─nvme0n1p1 259:1 0 500M 0 part /boot
├─nvme0n1p2 259:2 0 16M 0 part
├─nvme0n1p3 259:3 0 145G 0 part
├─nvme0n1p4 259:4 0 500M 0 part
└─nvme0n1p5 259:5 0 331G 0 part
└─vg1-lvroot 254:0 0 331G 0 lvm /
[0 21:27:03 christian ~] $

- On most distributions of Linux you will have at least one application called *Terminal* (or similar)
- When you open the terminal, you get an instance of a *shell*
- A shell is what you will (usually) get if you start up a Linux machine without a desktop environment installed, or if you log into a machine remotely via SSH.

The Shell

- A shell reads commands and executes appropriate programs in response.
- Many exist, e.g.: Bourne Shell (sh), C Shell (csh), Korn Shell (ksh), Bourne Again Shell (bash), Z Shell (zsh)
- *Bash* is the most common and widely distributed shell.
- POSIX defines a standard for how a UNIX shell should behave. Bash and many others conform, but also offer extensions.

Some Preliminaries

Getting help: Man Pages

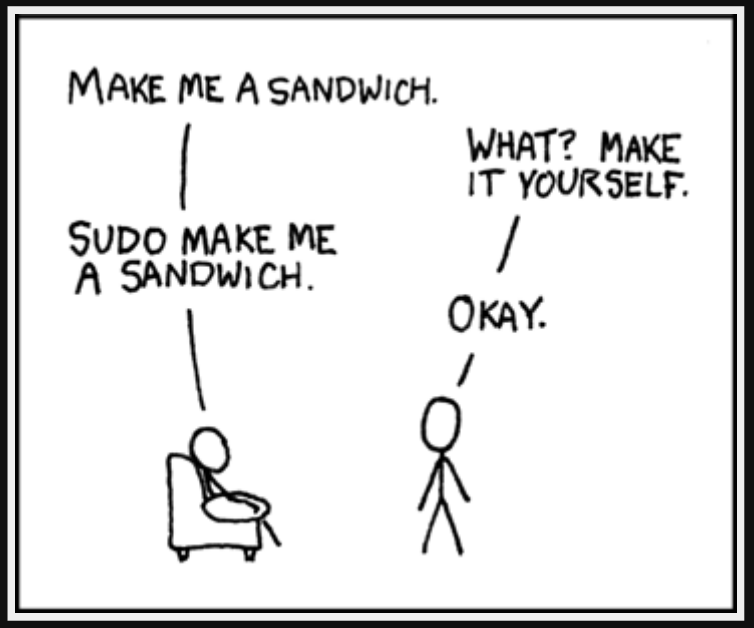
You can get information about many commands by running *man <search-term>*

The Root User

- Every Linux system has a user named *root*
- This is the superuser account, has access to all files/all commands
- Many operations are restricted to root: hardware configuration, network settings, certain services...
- Root can delegate permissions through sudo, defined in the file /etc/sudoers.
- If you have sudo access, you run a command as root by running *sudo <command>*

The Root User

The Root User



Source: XKCD

Installing Software: The Package Manager

- In most distributions, package managers are used to install software.
- Many exist: apt (Debian/Ubuntu), dnf (Fedora), yum (CentOS), ...
- Installs software from repositories considered safe
- Keeps track of dependencies

The File System

- Everything in Linux is in the same file system. All disks are too. Concept of drive letters does not exist.
- File paths look like: */dir1/.../dirN/filename*
- Top level called "root", has path */*
- A *relative path* is one without the initial /. It's interpreted relative to the directory you are in.
- Note: File names are *case sensitive*. The filenames "file.txt" and "File.txt" are distinct.

The File System

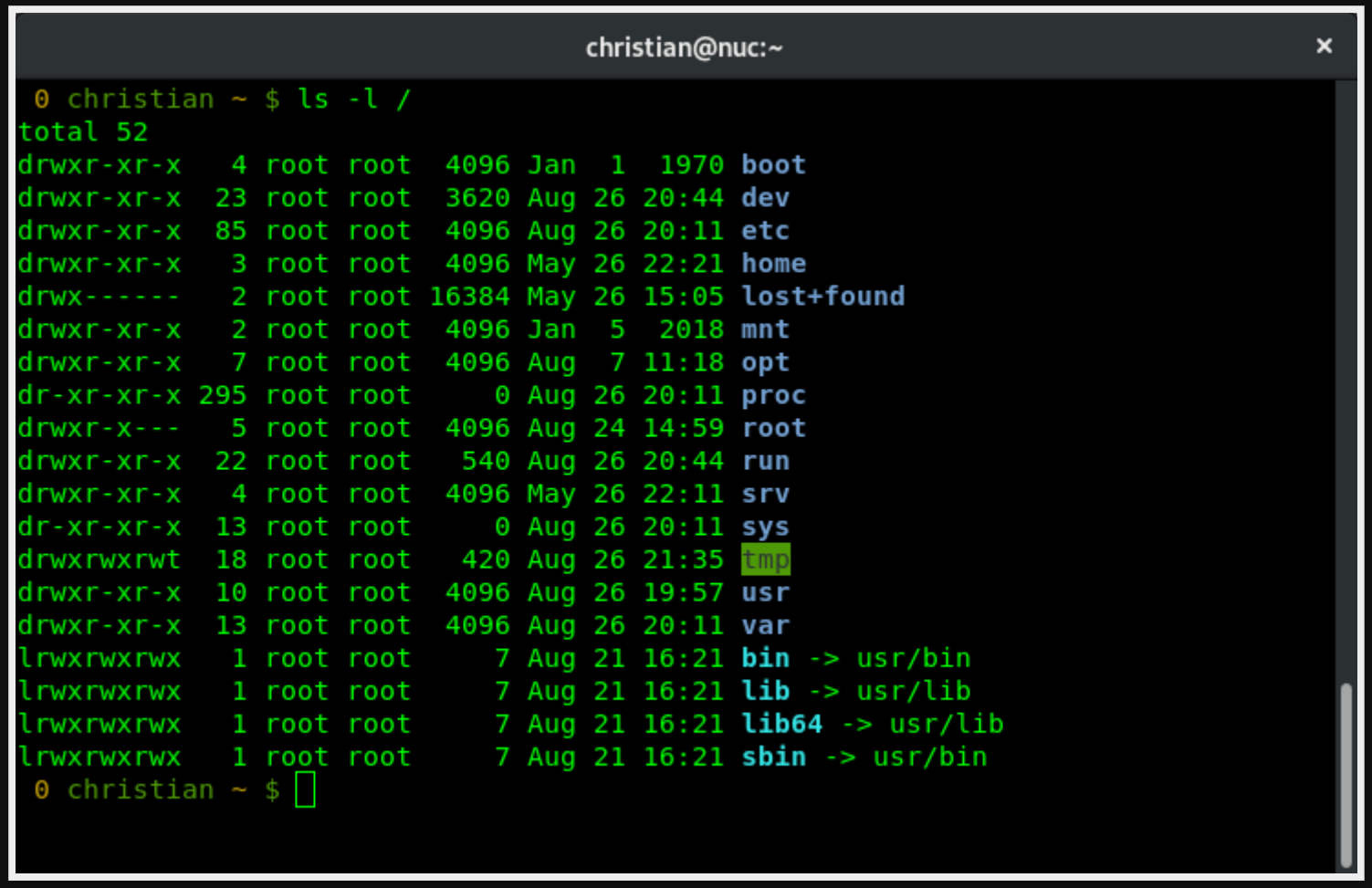
- Special symbols:
  - *~ (tilde)* home directory of current user
  - *. (dot)*: current directory
  - *.. (dot dot)*: parent directory

The File System: ls

- The command *ls* is used to list files/directories
- *ls -l* lists more detailed information
- *ls -a* also lists hidden files: Those which start with a . (period/full stop). Such files are also called *dotfiles*
- These can be combined into *ls -la* (or equivalently *ls -l -a*)

The File System: ls

The File System: Standard Directories



The File System: Standard Directories

- If you ls / on any linux system, many names will reoccur: etc, usr, home, tmp,...
- The purpose of these is defined in the Linux Filesystem Hierarchy Standard, part of the LSB.

The File System: Standard Directories

- Some directories you should know about:
  - */etc*: System-wide configuration files
  - */home*: Home dir. for everyone but root
  - */usr*: Application binaries
  - */tmp*: Temporary files
  - */var*: Files which are written to by applications, like logs and database files

The File System

- Unix philosophy: *"everything is a file"*
  - Files on the hard drive
  - Hardware: Hard-drives, keyboards, printers,...
  - Pipes for inter-process communication
  - ...
- The */dev* directory contains special device files which represent hardware. This could be hard-drives, mice, keyboards, etc.

Ownership and Permissions

- Linux has the concept of users and groups. Users can belong to many groups, groups can have many users. All users are in */etc/passwd*, all groups in */etc/group*.
- Files/directories have an associated user and group owner
- Files/directories have three permission groups: owner, group, and all users
- Each of these can be given any of three permissions: *read (r)*, *write (w)*, *execute (x)*

Changing Ownership and Permissions

- *chown* is used to change user/group owner of file: *chown <user>:<group> <filename>*. Only root can change owner.
- *chmod* is used to change permissions. Syntax:
  - Select one or more of *u*ser, *g*roup and *a*ll
  - + (add), - (remove) or =(make permissions exactly)
  - *r*ead, *w*rite, e*x*ecute.

  Examples: *chmod g+r <filename>*
  chmod ua-rw <filename>
  chmod u=rwx <filename>

Changing Ownership and Permissions

```
 0 christian /tmp/umuzi $ ls -l
total 8
-rw-rw-rw- 1 christian christian  0 Aug 27 08:58 everyone
-rwxrwxrwx 1 christian christian 54 Aug 27 09:01 executableforall
-rwxr--r-- 1 christian christian 54 Aug 27 09:00 executableforme
-rw-rw---- 1 christian christian  0 Aug 27 08:57 meandmygroup
-rw------- 1 christian christian  0 Aug 27 08:58 onlyme
-rw-r--r-- 1 root      root       0 Aug 27 09:01 onlyrootcanedit
-rw------- 1 root      root       0 Aug 27 08:57 rootssecretfile
 0 christian /tmp/umuzi $
```

- You can check permissions with ls -l
- -rwxrwxrwx: First rwx applies to owner, next to group, then all.
  E.g.: -rw-r--r-- means owner can rw, group can r, everyone else can r

Changing Ownership and Permissions

- Can also set chmod using octal permissions: Three digits, first repr. user, next group, last all. Let r = 4, w = 2, x = 1. Combine permissions by adding numbers.
- -rwxrwxrwx: chmod 777
  -rw-r--r--: chmod 644
  -rwxr-xr-x: chmod 755

Permissions on Directories

- Reading a directory means you can list files in the directory by name (with ls)
- Writing means that you are allowed to create and delete files in the directory, IF you have execute rights!
- Execute permission: This is the main permission you need to do *anything*: reading files, deleting files, enter subdirectories

Status Codes

- Whenever a program exits, it returns a numerical code to the operating system This code says whether the program shut down normally, or with an error.
- Status code 0 indicated normal shutdown (everything went as planned). Anything else signals an error.
- Status code of last command available as variable ? (accessed as $?) in the shell. E.g. *echo $?*

Output and Redirection

- Any program can access three standard file descriptors: standard in (0), out (1), error (2). A program reads from standard in, writes normal output to stdout, and error messages to stderr.
- Stdout and stderr can be redirected using >/1> and 2>, e.g. *echo Text 1>*. Both can be redirected at the same time using &>
- You can refer to descriptors as targets by using &0, &1 and &2. E.g. 2>&1 redirects stderr to stdout.

Output and Redirection

- Redirecting to a file: *echo Text > filename*
- Redirecting to null device: *echo Text > /dev/null*
- To channel file as stdin, use *command < filename*
- To channel stdout from one program as stdin to another, use the pipe: |

Globbing

- A * expands to all files in the current directory
- at* expands to all filenames that start with at. *at expands to all filenames that end with at. *at* expands to all filenames that contain at.
- If you don't want the shell to expand an asterisk, enclose it in '*'

Programs And Shell-Builtins You Should Know

Files and Directories

- *pwd*: List current working directory
- *cd <directory>*: Change your shell's working directory
- *ls (<directory>)*: List content of current dir or <directory> Important flags: -l, -a
- *cp <src> <dst>*: Copy file src to dst. To copy directories use -r flag
- *mv <src> <dst>*: Move file or directory from src to dst. Note: Used to rename files.

Files and Directories

- *touch <name>*: Create empty file with given name
- *mkdir <name>*: Create directory with given name. Flags to know: -p, creates recursively to any depth.
- *rmdir <name>*: Remove an empty directory
- *rm <file>*: Remove file. Use -r to remove directory. Use -f to force, i.e. delete without prompt.
- *file <filename>*: List information about format of file

User Management and Superuser

- *sudo <command>*: Run command as superuser
- *su (<username>)*: Log in as root, or as username if supplied
- *useradd <username>*: Add user to system. Note: You have to supply some other parameters, like -m (create home directory) and -s /bin/bash (set user's shell to /bin/bash)
- *usermod <username>*: Change user settings, e.g. username, user home directory, groups. Read the man page.
- *id <username>*: Information about user and groups user belongs to

Reading, Displaying, Editing Files

- *echo <text>*: Print text to stdout (on screen)
- *vim or vi*: Text editor. Exists on almost all linux/unix systems, but must be learned
- *nano*: User friendly text editor that exists on most modern systems
- *cat <filename(s)>*: Output file content
- *tac <filename(s)>*: Like cat, but lines are printed in reverse order
- *less (<filename>)*: Pager app. Used to display long text files. Also reads from stdin.

Searching for Information

Downloading From the Internet

- *wget <url> <query>*: Downloads the file at url
- *curl <url> <query>*: Can be used to issue requests to HTTP servers

SSH

- SSH is used to connect securely to a remote host
- It's a service which usually runs on port 22
- Connect by using *ssh username@host*, where host is the IP address or domain name of the server.

Bash Programming

What is Bash Programming?

- Bash enables us to run commands. What is a program? A series of commands.
- A bash *script* is a sequence of bash commands. They can be executed manually, one by one, on the terminal, or put in a file and run from the file.
- Bash contains many features which makes it similar to other programming languages that you might know, e.g. variables and functions.

Why Learn Bash (Some Subjective Reasons)

- BASH (and similar) scripts are the glue of the Unix/Linux world, they are portable and ubiquitous
- It allows you to orchestrate programs and services that follow the unix philosophy
- Many external tools are configured with bash-like scripts (e.g. many CI/CD systems)

Scripts

- Any file which contains statements in an interpreted language is a script. E.g.: JavaScript (through Node.js), Python, Ruby, Perl, Lua...
- First line usually has a *shebang* statement. This tells Linux which interpreter to use. E.g. *#!/bin/bash* or *#!/usr/bin/python*
- Run the command *which <name>*, to find path to interpreter, e.g. which bash or which node
- To make a script executable, use chmod +x. Note: script must also be readable, otherwise the interpeter can't read it.

Syntax

```
#!/bin/bash

# This is a simple Hello World script

echo Hello Umuzi # This prints Hello Umuzi on stdout
```

Syntax

Everything after a #, before a newline, is a comment (Except for the shebang statement)

```
#!/bin/bash

# This is a simple Hello World script

echo Hello Umuzi # This prints Hello Umuzi on stdout
```

Syntax

A statement can be a program invocation of the type you know from the terminal. As you know, *echo* prints to stdout, and is a very common way of printing text. (A common, slightly more advanced alternative is the *printf* program).

```
#!/bin/bash

# This is a simple Hello World script

echo Hello Umuzi # This prints Hello Umuzi on stdout
```

Variables

Bash also has variables. Bash does however not have types: every variable is a string, but can be *interpreted* as numbers/bools/... if needed. Assignment and usage are demonstrated below:

```
#!/bin/bash

# Let's declare a variable.
# The string has a space in it, so remember ""
TEXT="Hello Umuzi"

# We can now use this variable by enclosing
# the variable name between ${ and }
echo ${TEXT}

# We can also just prepend a $,
# but the previous style has become more common
echo Again: $TEXT
```

Variables

If a variable has not been declared, it will evaluate to an empty string

```
#!/bin/bash

echo "An unset variable evaluates to: ${UNSETVAR}"

# Will print "An unser variable evaluates to:"
```

Variables

You can save user input (from stdin) as a variable using *read*

```
#!/bin/bash

echo "What is your name?"
read NAME
echo "Hello, ${NAME}, nice to meet you!"
```

Variables

You can provide default values to use when a var is not set or empty by ${VARIABLENAME:-DEFAULT}

```
#!/bin/bash

echo "What is your name?"
read NAME
echo "Hello, ${NAME:-Umuzi}, nice to meet you!"
```

Environment

Every Linux process has a collection of variables called the *environment*. In particular, the shell has them, and they can be printed using *env*. A child process inherits its parents environment, so when you run a script, it inherits your terminal's enviroment. Environment variables are available just like other bash variables:

```
#!/bin/bash

echo "These are variables from the environment:"
echo "I was run by ${USER}"
echo "He/she has home dir ${HOME}"
echo "I was run from ${PWD}"
```

Environment

However, a Bash variable does not become part of the environment unless you mark it with the keyword *export*

```
#!/bin/bash
# This is file printname

echo "Hello, ${EXPORTEDNAME}"
echo "This name is blank: ${UNEXPORTEDNAME}"
```

```
#!/bin/bash

export EXPORTEDNAME="Uri"
UNEXPORTEDNAME="Christian"
```

```
./printname  # This is run as a child process.
             # It inherits our environment.
```

Environment

However, you can also run a program and *provide* new environment variables by using the syntax *VAR="value" program*

```
#!/bin/bash

EXPORTEDNAME="Peter" ./printname
```

Arguments

- When starting program from terminal, everything you write after the program name will be provided to the program as discrete arguments.
- Unless you tell bash otherwise, whitespace separates arguments
- If an argument contains whitespace, you must enclose it in double or single quotes (" vs '). " Will do substitution on inside parameters, ' will not.

Arguments

Arguments to your bash script are available as the variables 1, 2, etc. The variable 0 contains the script name. You can access all of them through @.

```
#!/bin/bash

echo "My name is: ${0}"
echo "The first argument is: ${1}"
echo "The second argument is: ${2}"
echo "All arguments: ${@}"
```

Command Substitution

You can use the stdout output of a command using the command substitution pattern: $(command). Another equivalent syntax you might see is the backtick syntax `command`.

```
#!/bin/bash

NAME="$(whoami)"
CURRENT_TIME="$(date)"
CURRENT_DIR="$(pwd)"
UMUZI_FILES="$(ls | grep umuzi)"

echo "This script was invoked by ${NAME} at ${CURRENT_TIME} inside ${CURRENT_D
echo "All files which contain the word umuzi:"
echo "${UMUZI_FILES}"
```

Command Substitution

Whenever you do command substitution, the ? variable will contain the exit status of the command that was run right after assignment.

```
#!/bin/bash

ROOTS_FILES="$(ls /root) 2>/dev/null"
echo "Exit status: ${?}"
echo "Roots files: ${ROOTS_FILES:-NO ACCESS}"
```

Numbers and Arithmetic

In order to interpret a bash string or variable as a number, enclose it between $((and))

```
#!/bin/bash

TWO="2"
THREE="3"

echo "${TWO} plus ${THREE} is $((TWO + THREE))"
echo "${TWO} minus ${THREE} is $((TWO - THREE))"
echo "${TWO} times ${THREE} is $((TWO * THREE))"
echo "${TWO} plus 7 is $((TWO + 7))"
echo "${TWO} divided by ${THREE} is $((TWO / THREE))"

# If integer arith. is not enough, use bc
echo "Floating point arithmetic this time:"
RESULT="$(echo "${TWO} / ${THREE}" | bc -l)"
echo "${TWO} divided by ${THREE} is ${RESULT}"
```

Conditional Statements

Bash lets you do conditional testing using *if*. The condition is followed by *then*, optionally *else*, and closed by *fi*. You can use if to check exit status: Exit status 0 evaluates to true, anything else to false.

```
#!/bin/bash

if ROOT_FILES="$(ls /root 2>/dev/null)"
then
    echo "Successfuly read /root. Files: ${ROOT_FILES}"
    exit 0
else
    echo "Unable to read /root. Exit status: ${?}" >&2
    exit 1
fi
```

Conditional Statements

You use if to run tests by enclosing a test between [[and]]. Many different tests exist.

- *[[-e FILE]]*: File exists
- *[[-d DIR]]*: Directory exists
- *[[-r FILE]]*: File exists and you can read it
- *[[-x FILE]]*: File exists and you can execute it
- *[[A = B]]*: Strings A and B are equal
- *[[A < B]]*: String A sorts lexicographically before B
- *[[-z A]]*: String A is empty

Conditional Statements

- *[[A -eq B]]*: A = B as integers
- *[[A -lt B]]*: A < B as integers
- *[[A -le B]]*: A ≤ B as integers

Conditional Statements

- *[[! EXPR]]*: The logical negation of EXPR
- *[[EXPR1 && EXPR2]]*: Logical and
- *[[EXPR1 || EXPR2]]*: Logical or

Conditional Statements

```
#!/bin/bash

if [[ -d "/root" ]]; then
    echo "Directory /root exists"
fi

if [[ -x "/root" ]]; then
    echo "I have execute rights on /root"
fi

if [[ -x "/root" && -e "/root/umuzi" ]]; then
    echo "I have +x on /root, and /root contains the file umuzi"
fi
```

Loops

Bash has *for* loops. For iterates through a list of values until it is exhausted

```
#!/bin/bash

NUMBERS="1 2 3 4 5"

for num in ${NUMBERS}
do
    echo "Saw number: ${num}"
done

for arg in "${@}"
do
    echo "Found argument: ${arg}"
done
```

Loops

In Bash 4 and later, you can generate a range of numbers as follows

```
#!/bin/bash

"Counting from 0 to 10:"
for i in {0..10}; do
   echo ${i}
done

echo "Counting every other number:"
for i in {0..10..2}; do
   echo ${i}
done

echo "Every third"
for i in {0..10..3}; do
   echo ${i}
done
```

Loops

Bash also has *while* loops. They take conditionals like if clauses. If you provide a program name, while runs as long as the exit code is 0. If you run a test, while continues until the test is true.

```
#!/bin/bash

ANS=""
while [[ "${ANS}" != "Please" ]]
do
    echo "Say please!"
    read ANS
done

echo "Thank you!"
```

Further Reading:

Steve Parker's Shell Scripting Guide
Bash Guide on Greg's Wiki
Many common command-line invocations explained