1. System Administration

1.1 Secure shell

Video

your machine - ssh --- sshd - remote machine

ssh [USERNAME@]HOSTNAME

beware of man in the middle (cryptography):

- ssh servers have a key pair for digital signatures
- asks if you're sure when connecting to a new server
- warns if key has changed

can setup own keys:

- convenient: no need for passwords
- secure: key never leaves your machine, even connecting to evil server

Exercises

ssh is a protocol that allows remote logins. Most common is the OpenSSH implementation.
Uses TCP port 22 by default.

Check your client

ssh localhost

Connect to the lab

bastion host: seis.bris.ac.uk. Reachable over ssh from the internet, lets you connect further to lab machines.
load balancer: rd-mvb-linuxlab.bristol.ac.uk. Connects you to a lab machine.
ssh USERNAME@seis.bris.ac.uk:

- uname -a prints "all" info about the system
  then ssh rd-mvb-linuxlab.bristol.ac.uk:
- no need for username again here (try whoami)
  try whoami / hostname
  exit twice to get back to own machine

ssh -J USERNAME@seis.bris.ac.uk USERNAME@rd-mvb-linuxlab.bristol.ac.uk:

- -J for jump through this host

Setting up ssh keys

when connecting, client and daemon run a cryptographic protocol to exchange keys and set up a shared secret key.
three authentication factors:

- something you know (passwords)
- something you have (key)
- something you are (biometrics)
  ssh keys implement digital signatures. Each key comes as a pair of files:
- private key in a file named id_CIPHER (CIPHER is cipher in use)
- public key in id_CIPHER.pub. Store this on every machine you want to login to.
  creating a key pair:
- on own machine: ssh-keygen -t ed25519. Normally stored in .ssh folder in home directory.
- -t parameter selects the cryptographic algorithm to use. Some older machines don't accept ed25519 - use rsa.
- Don't overwrite. Press enter straight away when asked for password.
  known_hosts stores the public keys of computers already connected to.

Set up key access on SEIS

First, upload public key to ~/.ssh on seis:

- Try ls -al ~/.ssh. If not exists, mkdir ~/.ssh
  Then secure copy from own machine:
- scp ~/.ssh/id_ed25519.pub "USERNAME@seis.bris.ac.uk:~/.ssh/"
  Now go to seis:
  ```
  cd .ssh
  cat id_ed25519.pub >> authorized_keys
  chmod 600 authorized_keys
  ```
  If anything's wrong, use -v to enable debugging info.

Setting up keys for lab machines

-A for Agent forwarding avoids uploading private key to seis.
Create ~/.ssh if not exists on lab machine.
From seis:

- scp ~/.ssh/id_ed25519.pub "rd-mvb-linuxlab.bristol.ac.uk:~/.ssh/"
  Login to lab machine:
  ```
  cd .ssh
  cat id_ed25519.pub >> authorized_keys
  chmod 600 authorized_keys
  ```

Now from own machine:
```
ssh -A -J USERNAME@seis.bris.ac.uk USERNAME@rd-mvb-linuxlab.bristol.ac.uk
```

Setting up a configuration file

ssh reads two configuration files:

- `/etc/ssh/ssh_config` for all users
- `~/.ssh/config` for one user
- checkout: `man ssh_config | less`
  Create a file called config in your .ssh directory (touch config):

```
Host seis
HostName seis.bris.ac.uk
User USERNAME

Host lab
HostName rd-mvb-linuxlab.bristol.ac.uk
ProxyJump seis
User USERNAME
```

Now:
`ssh lab`
Or ssh seis to update keys there.

Using different keys

Managing different systems and accounts - use a different key file for each one:

- -i FILENAME: select a private key file
- IdentifyFile FILENAME: select a file for a particular host
  By default, ssh searches for files in .ssh with names id_CIPHER.

1.2 Installing Vagrant and Debian

Video

Vitualisation:

- emulate a different stack
- reproducible build environment
- cost/scalability
  Vagrant:
  folder contains Vagrantfile and .vagrant folder:
- Host: folder with Vagrantfile (written with Ruby)
- allows access using ssh
- can share folders between host and guest
  Vagrantfile:

```
Vagrant.configure("2") do |config|
config.vm.box = "generic/alpine317"
end
```

  Commands:

- vagrant up: note that 22 (guest) => 2222 (host)
- vagrant ssh: login
- vagrant halt
- vagrant reload: stop and start machine
- vagrant destroy
  Try:
  `ssh -p 2222 vagrant@127.0.0.1`
- this won't work because we don't have the key to access this machine. Vagrant creates own key pair at vagrant up.
- You *could* find that key and specify it with ssh -i, but why bother? Just use vagrant ssh
  Storage of virtual machines:
- Linux: ~/.vagrant.d
- some configuration goes in the .vagrant folder
  Storage for lab machines:
- in /tmp : may not survive host reboots. Treat as disposable
  Alpine linux:
- Minimal installation - no gcc, git etc.
- Strong on security - good choice in production
- Small: container can fit in 8mb
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Exercises

Configuring a box

Vagrantfile:
Vagrant.configure("2") do |config|
config.vm.box = "generic/debian12" // Selects vm image (box)
config.vm.synced_folder ".", "/vagrant" // Setup shared folder

```
  config.vm.provision "shell", inline: <<-SHELL // runs a provisioning command at
first download and install. Runs as root
     echo "Post-provision installs go here"
  SHELL
```

end
Note: this is a script in Ruby. <<-SHELL treats the following as a string until SHELL.

Running vagrant

vagrant up : starts vm configured in current folder, if not downloaded and provisioned yet then does this.

vagrant ssh
Errors:

- can't find a provider: not installed virtualbox, or not rebooted since installing it.
- hypervisors: see https://www.vagrantup.com/docs/installation, section Running Multiple Hypervisors. Can't run when another program using virtualisation subsystem.

Shutting down cleanly

exit
vagrant halt
Always do this before turning off host.

Running on a lab machine

mkdir softwaretools
cd softwaretools
wget https://raw.githubusercontent.com/cs-uob/COMSM0085/master/exercises/part1/src/resources/Vagrantfile
Note:

- Private key: .vagrant/machines/default/virtualbox/private_key
- Public key: /home/vagrant/.ssh/authorized_keys
  vagrant ssh is essentially: ssh -i KEYFILE vagrant@localhost -p 2222

Warning about lab machines - read carefully!

/vagrant maps folder containing Vagrantfile on host.
Always cd /vagrant and use that instead of home folder to avoid losing files.

1.3 Debian system adminstration

Video

Package manager: contain instructions for installing from repositories. Keep track of dependencies
apt: advanced package tool
.deb files (for Debian)
Debian/Ubuntu/Mint/... : apt
Alpine: apk
Red Hat: rpm
Arch: pacman
Configuration file - what repositories a system searches:
cat /etc/apt/sources.list
Contends-<ARCH>.gz lists all packages for a system
Finding packages:
apt search [-v] [-d] STRING
apt info [-a] PACKAGE
apt list [-I] PACKAGE
apt [COMMAND] --help
Update and Upgrade:

sudo apt update // Download new list of packages, don't install yet
sudo apt upgrade // Upgrade all installed to latest version
Installing:
sudo apk add PACKAGE [PACKAGE ...]
Or in Vagrant directly:
$ cat /vagrant/Vagrantfile
...
apk add libc6-compat util-linux
...
Find from what package a software came from:
dpkg-query -S /bin/bash

Exercises

The file system

Linux: single file hierarchy with root folder /.
Documented in Filesystem Hierarchy Standard (FHS): ls /
/bin : binaries, ie programs to run.
which ls: finding where ls is.
/usr : historical. Essentially:

- /bin was only for binaries needed to start system
- /usr/bin was for other global
- /usr/local/bin for those installed by local administrator
  /usr and subfolders are for read-only, eg program and configuration, but not data or logs
  Debian solution: /bin is a link to usr/bin
  /etc : system-wide configuration files. Only root can change its contents.
  /lib : dynamic libraries. .dll for windows, .so for POSIX.
  /home : users' home directories. Exception - root gets /root.
  /sbin : system binaries. Programs only system administrators use.
  /tmp : temporary file system, may be stored in ram.
  /var : files that vary over time, eg logs or caches.
  /dev, /sys, /proc : virtual file systems, because in UNIX everything is a file.
- /dev : interface to devices like disks. /dev/sda - first SCSI disk in system, ./sda1 - first partition on that. /dev/mem - memory
- /proc : access to running processes.
- /sys : access to system functions. eg /sys/class/backlight/acpi_video0/brightness for screen brightness

Package managers

Repository: collection of software you can install, can be hosted everywhere (floppy disk, cd rom, over the internet)
Package manager: installs packages from a repository. A bit like app store, but allows installing specific

versions. Also installs dependencies automatically.
sudo apt remove : removes a package

Lab machines

In Vagrantfile, below echo: put apt install PACKAGE for vagrant up to install automatically.
No need for sudo because it's already root during vagrant up.

2. The POSIX Shell

Videos

1. The shell

shell = terminal = console = command line = prompt
user -> request -> shell
<- response <-
$: in POSIX shell
: root
% : C shell
> : continuation line
tab : complete command
double tab : show possible completions
up/down : scroll through history
^R text : search history for command
which cd doesn't return anything because cd is builtin.
ls -a : show hidden
--help
man [SECTION] COMMAND :

- section 1: shell commands, eg man 1 printf
- section 2: system calls
- section 3: C library, eg man 3 printf
- man ascii

shell expansion

* : all filename in current scope, eg a* for all starting with a
? : single character, eg image???.jpg matches image001.jpg
[ab] : single character in list, eg [0-9]
$: variable name expansion

- $PWD : current working directory
- $TERM : terminal
  "" : turn off pattern matching, but keep variable interpolation and backslashes on
  '' : turn off everything
  \ : do not treat as pattern
  find DIR [EXPRESSION]

- eg find . -name "a*"

2. Pipes 1

UNIX philosophy: Easier to maintain 10 small programs than a large program

- Each program should do one thing well
- Programs should cooperate to perform larger tasks
- Universal interface between programs should be a text stream
  Pipes are part of libraries
- #include <unistd.h> : POSIX abstractions for reading and writing files
- programs read(fd, buffer, size) and write(fd, buffer, size)
- file descriptors: 0 for standard input, 1 standard output, 2 standard error
  sort -r | uniq : removes duplicates

3. Pipes 2

redirect to file:
cat infile | sort > outfile
sort < infile > outfile == sort infile > outfile == cat infile | sort > outfile
> : overwrites file
>> : appends to file

error redirect:
`COMMAND > FILE 2> ERRORFILE`
`COMMAND > FILE 2>&1` (notice the order here is strict)

ignore output:
`COMMAND > /dev/null`

Program expecting a filename can use stdio instead by:

- using the filename - , if supported
- using the filename /dev/stdin etc, if OS supports it

Filenames starting with dashes are bad, but if you have to use it:

- cat ./-

Advanced

tee

- e.g. ls | tee FILE
- writes a copy of input to FILE as well as to stdout

`less`

- pager, displays one page at a time
- up/down arrows scroll

- space\enter advances one page
- forward slash / opens a search
- q quits

`sed`

- stream editor, changes text using a regular expression
- s/ONE/TWO/[g] replaces first (or all with g) ONE with TWO
- e.g. echo "Hello World" | sed -e 's/World/Universe/'

need a file, want a pipe: use brackets

- PROGRAM <(SOMETHING)
- e.g. cat <(echo "Hi") returns hi
- note echo <(echo "Hi") returns like /dev/fd/63. This is because echo doesn't read into files

subshell to argument

- `COMMAND $(SOMETHING)`
- e.g. `echo $(echo Hi | sed -e s/Hi/Hello)` returns Hello
- old fashioned way with `: COMMAND `SOMETHING``

Exercises

2.1 Shell Expansion

Consider arguments.c. Compile with `gcc -Wall arguments.c -o arguments`

Whitespace

Notice multiple whitespaces are ignored. "one two" is one argument.

Pattern matching

Notice `./arguments *` returns all files in current directory.

Notice `../arguments *` from `empty` returns `*`. Equivalent output from `./arguments *` or `./arguments '*'` in parent folder.

Files with spaces in their names

Notice `touch "silly named file"` creates one file. Without quotes: 3 files.

- On terminal, it looks like silly\ named\ file

Shell variables

VARIABLE=VALUE sets variable; $VARIABLE retrieves value

`p=arguments`

`gcc -Wall $p.c -o $p`

${a}b means value variable a followed by letter b.

Good practice to put double quotes around shell variables to avoid space-in-name problems.

Example:

```
program="silly name"
gcc -Wall $"program.c" -o "$program"
```

This expands to:

`gcc -Wall "silly name.c" -o "silly name"`

Without "" would be incorrect.

Variables must be set on a seperate line. Consider:

`file=arguments gcc -Wall "$file.c" -o "$file"`

This is wrong: shell first reads line and substitutes value of $file (unset variables expand to empty string by default) before executing the command.

2.2 Pipes

- `cat [FILENAME [FILENAME...]]` writes contents of one or more files to stdout
- `head [-n N]` reads stdin and writes first N lines. Default 10. Or skip last -N if N<0.
- `tail [-n N]` reads stdin and writes last N lines. Default 10. Or skip first N if N in the form of `+N`.
- `sort` reads stdin into memory buffer, sorts and writes.
- `uniq` removes repeated lines immediately following each other.
- `grep [-iv] EXPRESSION` reads stdin, prints only lines that matching the expression to stdout. `-i` case-insensitive, `-v` prints those not matching.
- `sed` see video Pipes 2, advanced.
- `wc [-l]` word count, `-l` counts lines.
  All these can take optional extra filename as argument, in which case they read from this file as input.

`^X` matches an X only at the start of the string.

`'j$'` matches a j only at the end of the string, single-quote to stop the shell from interpreting the dollar sign.

`.` (period) matches a single character.

`'[aeiou]'` matches any string that contains one of the bracketed characters. Quotes to stop the shell from interpreting the brackets.

`cat words | grep -i '^P.*a.*a.*a.*'` finds all starting with p/P and has at least 4 a's.

2.3 Regular Expressions

3. Git

Videos

3.1

Source Code Control System (SCCS, 1972) -> Revision Control System (1982) -> Concurrent Versions Systems (1986) -> Subversion (SVN, 2000)

- Each has an official central repostitory storing the latest versions

Around 2000, move to decentralised

- Every user has a master version of source control
- Changes are accepted from others through merges

`man 1 git`

`git init` - creates repo. Initialised empty repo in `private/tmp/tutorial.git/`

`git add` stages a file, this means

- you're saying it will be part of a new commit
- adding changes into Git's versioning
- not saving anything yet, things can still change

`git commit -m 'MESSAGE'`

- Everything staged gets written as a single change
- A note to explain change. Name associated

`git config --global user.[name, email] 'USERNAME/EMAIL'`

Commits are all identified by their hash

- `git tag`
  All commits are made to a branch which is a tag
- When commiting, branch tag is updated to point to new commit at top of branch
- default branch is main/master
  Special tag - HEAD
- Always points to current code
- Minus any unstaged work

`git checkout HEAD --hello.c` throws away changes made
`git reset --hard` removes all changes
`git clean -dfx` deletes all untracked files
`git checkout HEAD~1` code before the last commit

`git checkout main` go back to main after change
`git revert HEAD` undo all changes in a commit

3.2 Git: Working with remotes

Decentralised - every copy can act as the central repo
Patch based - Bob sends Alice his changes

- `git format-patch origin/main \ --to=alice@bristol.ac.uk` prepares the file XXX.patch. Send this via email
- Alice then `git am ../bob/XXX.patch`
  Pull based - Alice pulls Bob's changes
- Alice `git remote add bob ~bob/coursework`
- `git fetch`
- If Alice likes the changes, `git pull bob main`. Equivalent to `git fetch bob` and `git merge --ff bob/main`

Github - centralised remote (aka a forge)

- `git remote set-url ...`
- `git push`
  To use a forge, ssh-keygen is needed. Store public key in github.

When tree diverges, needs `merge`. Resolve conflict if different people changed same file.
Or alternatively, rebase - as if changes are made one after the other.

To summarize:

- `git remote`, `git clone` to work with others
- `git fetch`, `git pull` or patch files to get others' work
- `git merge`, `git rebase` to integrate changes
- `git push` to send work back to forge

3.3 Git: Branches

Main branch always works.
When doing work, take a branch off main.
Merge back to main when done.
E.g.

```
git branch new-feature main
git checkout new-feature
touch c; git add c; git commit -m 'Adds c'
git checkout main
git merge -no-ff new-feature
git branch -d new-feature
```

Useful for working on multiple features at once.
`rebase` during merging everthing together. Remember to back up. This keeps everything in main working.

`rebase -i` - interactive rebase session.
`git cherry-pick 8819c8e` - only pull this specific change. Internally this is rebase.

Exercises

3.1 Git

Git documentation

`apropos git`
`apropos git -a tutorial` for all of it

`man giteveryday`
`man gittutorial`

Configuring your identity

See above
`git-blame` to see who did the bad stuff
Drop `--global` for local config

For those of you using Vagrant

Add `git config --global user.name "YOURNAME"`
`git config --global user.email "YOUREMAIL"` under `config.vm.provision`

A sample project and repository

Do a `git status` and you see main.c in red under untracked files - this is new file, git does not know about it

Ignoring files

creating a file called .gitignore, each line says which file(s)/folders to ignore - e.g. use *.o to select all object code files

Commit and checkout

`git add .` adds all new and changed files and folders in current folder
`git checkout HASH` where HASH is the 6 or however many you need characters of the commit
`git checkout main` to return to the latest version
`git revert HASH` adds a new commit that returns the files to previous state
`git reset HASH` undoes commits by moving the HEAD pointer back to the commit with the given hash, but leaves the working copy alone

- `--hard` option to change the files as well

Part 2: Git forges

Managing SSH Keys:

- Multiple Devices: Consider using unique SSH keys for different devices and services for both security and privacy.

- SSH Config: Use `~/.ssh/config` to manage different keys for different services, specifying different IdentityFile for each host.

Creating and Cloning a Repository:

- Create Repository: Use the "New" button on GitHub to create a new repository and initialize it with a README.
- Clone Repository: Use `git clone git@github.com:USERNAME/REPONAME.git` to clone the repository locally.

Basic Git Operations:

- Remote Interaction: Use `git remote show origin` to verify remote setup.
- Branch Status: Understand different status messages (up to date, ahead, behind, diverged).

Resolve a fake conflict

`git rebase origin/main` means pretend that everything in origin/main happened before I started my local changes

Resolving a real conflict

Resolve the conflict, at which point you're "ahead by 2 commits". Merge.

Working with others

Repository Setup:

- One team member creates a new Git repository on GitHub.
- Add other team members as collaborators to the repository.
- Initial member shares the repository's clone URL with the team.
- Every team member clones the repository to their local machine using git clone <URL>.

Initial Commit:

- If repository does not contain files, create README.md or any other file, add it to the repository, commit, and push to main branch.
- Ensures that there is at least one commit in the history for the subsequent steps.

Creating the Develop Branch:

- One team member switches from the main branch to create a new `develop` branch locally using `git checkout -b develop`.
- Make a commit on this new branch to establish its history. This could be as simple as editing the README file or adding a new file.
- To push the develop branch to the remote repository and set the upstream tracking, the command `git push --set-upstream origin develop` is used.

Tracking the Develop Branch:

- After initial push, the develop branch will be available in the remote repository.
- Other team members can then fetch the latest changes from remote repository using `git pull`.
- List all branches, including remote ones, using `git branch -a`.
- To switch to the develop branch, they use `git checkout develop`, which sets up their local branch to track the remote develop branch.

Working with the Develop Branch:

- Once on the develop branch, all team members can start contributing by creating new features or fixing bugs.
- Regular commits and pushes will keep the remote develop branch updated with everyone's changes.
- Team members should frequently pull changes from the remote develop branch to ensure they are working with the most current version.

Everyone can create own branches this way to avoid conflicts.
`git branch` won't show other's branches. `git branch -a` does.

Merging:

- Commit all your changes and push.
- Fetch the latest changes from origin (a simple `git fetch` does this).
- `git checkout develop`, which switches you to the develop branch (the changes for your latest feature will disappear in the working copy, but they're still in the repository). You always merge into the currently active branch, so you need to be on `develop` to merge into it.
- `git status` to see if someone else has updated develop since you started your feature. If so, then `git pull` (you will be behind rather than diverged because you have not changed develop yourself yet).
- `git merge NAME` with the name of your feature branch.
- Resolve conflicts, if necessary.
- `git push` to share your new feature with the rest of the team.

Pull requests:

- let a team discuss and review a commit before merging it into a shared branch such as develop or main

The procedure for merging with a pull request on github:

- Commit and push your feature branch.
- On github.com in your repository, choose Pull Requests in the top bar, then New Pull Request .
- Set the base branch as the one you want to merge into, e.g. develop, and the compare branch as the one with your changes. Select Create Pull Request.

- Add a title and description to start a discussion, then press Create Pull Request again to create the request.

pull request is linked to a branch, use it for code review as follows:

- A developer creates a feature branch and submits a pull request.
- The reviewer looks at the request. If they find bugs or other problems, they add a comment to the discussion.
- The developer can address reviewer comments by making a new commit on their feature branch and pushing it, which automatically gets added to the discussion.
- When the reviewer is happy, they approve the request which merges the latest version of the feature branch into the base branch (for example `develop`).

When your changes are based on old version, but that has since been updated, you can force the push, which overwrites the old version, with `git push --force origin BRANCHNAME`. Use with caution.
For safety:

- Only rebase on private branches.
- Only force push on private branches, and only if it is absolutely necessary (for example to tidy up a rebase)

To summarise, the pull request workflow is:

- Commit and push your changes.
- If necessary, rebase your feature branch on the develop branch.
- Create a pull request.
- If necessary, participate in a discussion or review and make extra commits to address points that other developers have raised.
- Someone - usually not the developer who created the pull request - approves it, creating a merge commit in develop (or main)

4. Shell Scripting

Videos

1. Permissions

`grep -Ev '^_' /etc/passwd | column -ts` - returns table of usernames and passwords.
Try `man 5 passwd`

-rwx ... :

- first ch: b for block file, c for character block file, d for directory, l for symbolic link, p for FIFO, s for socket link.
- then user, group, other rwx

- user & group x can be s, run as UID/GID of owner. e.g updating own password
- other x can be t, sticky bits. e.g. logs

Setuid programs:

- su: switch to user (default root) with password
- sudo: switch to user if sysadmin allows you with password
- doas: modern rewrite of sudo with fewer bugs

`chown USER:GROUP FILE` - change who owns a file
`chmod go-wx FILE` - remove w, x for group, other

Linux - `capabilities` sets what things any user can do
`namespaces` allows multiple root users with different capabilities
Try `man 7 capabilities`

2. Shellscripting: basics

Start with shebang `#!` then path to interpreter of script and args. E.g.
`#! /usr/bin/env bash`
Then `chmod +x script.sh`
`./script.sh`
Run by specified interpreter. Use `sh script.sh` if don't want to / can't mark as x.
Use `env` to avoid different paths of different systems. It finds where the program is by looking through `PATH`.
`PATH` is an environment variable, tells system where all programs are. Colon seperated list of paths.

Basic syntax:

- `A; B`: run A then B
- `A | B`: run A and feed output as input to B
- `A && B`: run A and if successful run B
- `A || B`: run A and if failed run B

Programs return a one byte exit value:

- stored in variable ${?}
- 0 success; >0 failure
  Consider `test $? -eq 0 && printf "Command succeeded\n"`
  or simply `[ $? -eq 0 ] && printf "Command succeeded\n"`

3. More Shellscripting

Variables:

- Create a variable: `GREETING="Hello World!"` (no space around =)
- Use a variable: `echo "${GREETING}"`
- `export greeting`: store as environment variable

- `unset greeting`: get rid of variable

`set -o nounset`: error if unset var used.
`echo "${NAME:? variable 1 passed to program}"`: if NAME not set, display error message.

Standard variables:

- `${0}`: name of script
- `${1}`, `${2}`, `${3}` ...: arguments to script
- `${#}`: no. of arg passed
- `${@}` and `${*}`: all args

Control flow, e.g:

```
if test -x script.sh; then
   ./script
fi

for file in *.py; do
       python "${file}"
done
```

IFS (In Field Seperator) tells system how arguments in for loop are seperated, e.g. `,`.
`seq 5` `seq -s, 5` are `1 2 3 4 5` and `1, 2, 3, 4, 5`

```
case "${SHELL##*/}" in    # ${VAR##*/}" remove everything up to the last /… gives
file name
       bash) echo "bash" ;;
       zsh) echo ... ;;
       *) echo "others" ;;
esac
```

Consider replacing `"${SHELL##*/}"` with `$(basename "${shell}")`

```
echo "${SHELL}"
echo "${SHELL##*/}"
echo "$(basename "${SHELL}")"
echo "$(dirname "${SHELL}")"
```

Replacing file extensions:

```
for f in *.jpg; do
       convert "${f}" "$(basename "${f}" .jpg).png"
done
```

`ps -A | grep -i firefox | awk '{print $1, $5}' | ghead -n -1 | wc -l`
The <<< pipe takes a string and places it on standard input.

Exercises

4.1 File permissions

Create a user and a group

Create a new user with `sudo adduser NAME`
`sudo addgroup USERNAME GROUPNAME`

Explore file permissions

`chgrp -R GROUPNAME DIRECTORY`

Setuid

Change user x to s: allow others to run as if they're the user

Sudo

By adding `%users ALL=(ALL) /sbin/reboot` to the sudoers file, members of the users group can run the reboot command.
Never edit /etc/sudoers directly without visudo to avoid syntax errors that could disable sudo and require direct system access to fix.

4.2 Shell scripting

`set -e` makes the whole script exit if any command fails. This way, if you want to run a list of commands, you can just put them in a script with set -e at the top, and as long as all the commands succeed (return 0), the shell will carry on; it will stop running any further if any command returns nonzero. It is like putting || exit $? on the end of every command.
`set -u` means referencing an undefined variable is an error. This is good practice for lots of reasons.
`set -o` pipefail changes how pipes work: normally, the return value of a pipe is that of the last command in the pipe. With the pipefail option, if any command in the pipeline fails (non-zero return) then the pipeline returns that command's exit code.

5. Build Tools

Videos

1. Make

`.PHONY: clean all` specifies these are not files

```
.PHONY: all clean

figures=$(patsubst .dot,.pdf,$(wildcard *.dot))

all: hello coursework.zip ${figures}
clean:
	git clean -dfx

hello: hello.c library.o extra-library.o
```

```
%.zip: %
	zip -r $@ $<

%.pdf: %.dot
	dot -Tpdf $< -O $@
```

2. Language specific buildtools

Make is rubbish at dealing with dependencies:

- Doesn't know how to fetch dependencies
- Doesn't track versions beyond source is newer than object

`mvn test` run the test suite
`mvn install` install the JAR into your local JAR packages
`mvn clean` delete everything

6. Debugging

Video

```
cc -Og -g journal.c -o journal
gdb ./journal
(gdb) run <<<"hello"
```

`(gdb) run <<<"hello"` or `(r)`: run as if "hello" is input by user
`(gdb) bt`: prints a list of the function calls that are currently on the stack
`break main``b file.c:42`: sets breakpoint
`delete``d 1`: delete a specific breakpoint or all if no arguments given
`continue``c`: to next breakpoint or end
`next``n`: Executes the next line of the program, stepping over function calls.
`step``s`: Executes the next line of the program, stepping into function calls.
`inspect varname`: Evaluates and prints the value of variable, function call or expression
`list``l 35`: Lists the source code. By default, shows 10 lines around the current line or the specified line.
`info``i`: Displays information about various aspects of the debugger state, such as breakpoints, threads, or variables.

- (gdb) info breakpoints
- (gdb) info locals
  `finish`: Continues running until the current function completes its execution.
  `quit``q`: Exits GDB.
  `watch`: Sets a watchpoint on a variable. Execution will stop whenever the variable changes.
  `help COMMAND`: manual

strace:
The strace tool lets you trace what systemcalls a program uses

- On OpenBSD see ktrace and kdump
- On MacOS/FreeBSD see dtruss and dtrace
  ltrace traces library calls.

8. SQL Introduction

Video

1. Intro to databases

`systemctl start mariadb`
`systemctl enable mariadb`
By default, root user with no password, but if someone is paying you money to do it:

- Set usernames and passwords
- Firewall off ports
- Add logging and intrusion detection
- Backup
- Secure backups
- Have a get out plan...

2. Relational modelling

A key for an entity is the set of attributes
needed to uniquely refer to it.

▸ A candidate key is a minimal set of
attributes needed to uniquely refer to it.

▸ The primary key for an entity is the key we
use.

If a key contains multiple attributes its called a composite key.
If a key is a meaningless ID column you added just for the sake of having a key its called a surrogate key.

3. Basic SQL

```
CREATE TABLE IF NOT EXISTS student (
       name TEXT NOT NULL,
       number TEXT NOT NULL,
       PRIMARY KEY (number));
CREATE TABLE IF NOT EXISTS unit (
       name TEXT NOT NULL,
       number TEXT NOT NULL,
       PRIMARY KEY (number));
CREATE TABLE IF NOT EXISTS school (
       name TEXT NOT NULL,
       PRIMARY KEY (name));
CREATE TABLE IF NOT EXISTS class_register (
       student TEXT NOT NULL,
       unit TEXT NOT NULL,
       FOREIGN KEY (student) REFERENCES student(number),
       FOREIGN KEY (unit) REFERENCES unit(name),
       PRIMARY KEY (student, unit));

DROP TABLE IF EXISTS class_register;
```

```
DROP TABLE IF EXISTS student;
DROP TABLE IF EXISTS unit;
DROP TABLE IF EXISTS school;
```

`INTEGER` whole numbers
`REAL` lossy decimals
`BLOB` binary data (images/audio/files...)
`VARCHAR(10)` a string of 10 characters
`TEXT` any old text
`BOOLEAN` True or false
`DATE` Today
`DATETIME` Today at 2pm

`NOT NULL` can't be NULL
`UNIQUE` can't be the same as another row
`CHECK` arbitrary checking (including it conforms to a regular expression)
`PRIMARY KEY` unique, not NULL and (potentially) autogenerated
`FOREIGN KEY` (IGNORED BY MARIADB) other key must exist

Can I add constraints later?
Yes with the ALTER TABLE statement

- But often easiest just to save the table somewhere else
- Drop the table
- Reimport it

```
INSERT INTO unit(name, number) VALUES ("Software Tools", "COMS100012");
```

```
SELECT artist.name AS artist,
COUNT(album.title) as albums
FROM album
JOIN artist
ON album.artistid = artist.artistid
WHERE album.title LIKE '%Rock%'
GROUP BY artist
ORDER BY albums DESC
LIMIT 5;
```

4. Normal Form

First Normal Form: Each column shall contain one (and only one) value
Second Normal Form: Every non-key attributue is fully dependent on the key
Third Normal Form: Every non-key attribute must provide a fact about the key, the whole key and nothing but the key
Boyce-Codd Normal Form: A slightly stronger form of 3NF...

- Sometimes called 3.5th Normal Form

- Every possible candidate key for a table is also in 3NF.
  - Split a 3NF table into tables with single candidate keys to get 3.5NF.

  4th Normal Form :If multiple attributes in a table depend on the same key,

▸ Then those attributes should be dependant too

▸ Otherwise split them into separate tables...

5th Normal Form

- It's in 4th normal form and you can't split it into more separate tables

Exercises

9. Intermediate SQL

Video

Cannot compare values with NULL. Don't use NULL
NATURAL JOIN is like a regular JOIN but assumes same named columns ought to be equal
FULL OUTER NATURAL JOIN

You can nest queries inside one another (subqueries!)

▸ This is a recipe for making your SQL slow

▸ Maybe just use SQL for data retrieval and leave complex stats to statistical programming languges?

```
SELECT SQRT(AVG(Deviation)) AS STDDEV
FROM (
      SELECT Fruit, Stars, Mean,
            (Stars-Mean)*(Stars-Mean) AS Deviation
      FROM ranking JOIN (
      SELECT AVG(stars) AS Mean
            FROM ranking
      )
      WHERE stars IS NOT NULL
);
```

Exercises

10. SQL and Java

Video

```
import java.sql.*;
try (final Connection conn = DriverManager.getConnection("jdbc:sqlite:database.db"))
{
conn.createStatement()
.executeQuery("CREATE TABLE users(username TEXT PRIMARY KEY, password TEXT)");
} catch (final SQLException err) {
System.out.println(err);
}
java.sql.SQLException: No suitable driver found for jdbc:sqlite:database.db

import java.sql.*;
import java.util.*;
final var users = new HashMap<String, String>();
```

```
users.put("Joseph", "password");
users.put("Matt", "password1");
users.put("Partha", "12345");
try (final Connection conn = DriverManager.getConnection("jdbc:sqlite:database.db"))
{
conn.createStatement().executeUpdate("DELETE FROM users");
final var statement = conn.prepareStatement("INSERT INTO users VALUES(?, ?)");
for (final var user : users.keySet()) {
statement.setString(1, user);
statement.setString(2, users.get(user));
statement.executeUpdate();
}
} catch (final SQLException err) {
System.out.println(err);
}
```