

## About this course

## Program, September 27<sup>th</sup>



09:30 – 10:00 Morning coffee & registration  
10:00 – 10:15 Introduction to the course (whereabouts, etc.)  
10:15 – 10:45 What is UNIX/Linux (basic concepts, multi-user, multi-tasking, multi-processor)  
10:45 – 11:30 Linux on my own computer (native installation, dual-boot, virtual appliances)  
11:30 – 12:00 1st utilization of Linux - GUI based (opening terminal from GUI, creating shortcuts)  
12:00 – 13:00 Lunch  
13:00 – 14:15 A first glimpse of the shell (simple navigation, listing, creating/removing files and directories)  
14:15 – 14:45 Coffee  
14:45 – 15:15 Text editors (vi, emacs and nano)  
15:15 – 16:00 File permissions (concepts of users and groups, changing permissions/groups)  
16:00 – 16:30 Troubleshooter: Interactive session to deal with open questions and specific problems

1

2

## Program, September 28<sup>th</sup>



09:00 – 09:30 Job management (scripts and executables, suspending/killing jobs, monitoring)  
09:30 – 10:00 Coffee  
10:00 – 11:15 Setup of your system (environment variables, aliases, rc-files)  
11:15 – 12:15 Lunch  
12:15 – 13:30 A second look at the shell (finding content, accessing and copying from/to remote hosts)  
13:30 – 14:00 Linux security (best practices, user management, firewall, ssh keys)  
14:00 – 14:30 Coffee  
14:30 – 15:30 Hands-on exercises  
15:30 – 16:15 Troubleshooter: Interactive session to deal with open questions and specific problems

## How we teach







- All topics are presented with interactive demonstrations
  - This is a course for beginners, hence we try to adopt a possibly slow pace
  - Please, indicate immediately, if pace is too fast. We want to have everyone with us all the time
- Additionally, exercises to each of the sections will be provided
- The *Troubleshooter* section is meant for personal interaction and is (with a time-limit to 16:30 or 16:15) kept in an open end style

3

4

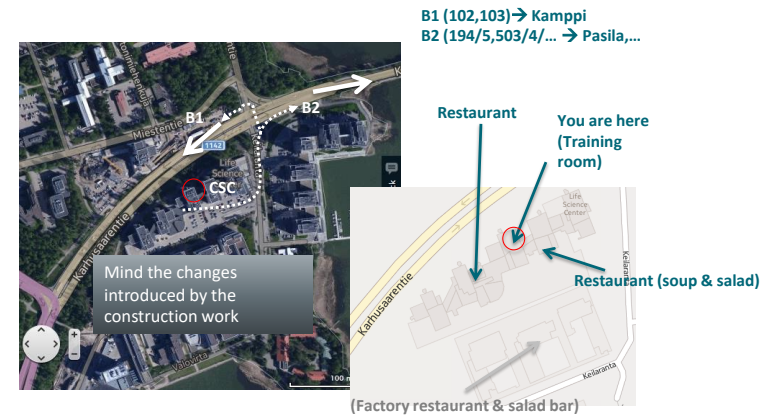
## Practicalities

- Keep the name tag visible
- Lunch is served in the same building
- Toilets are in the lobby 
- Network: 
  - WIFI: eduroam, HAKA authentication
  - Ethernet cables on the tables
  - CSC-Guest accounts upon request
- Public transport: 
  - Other side of the street (102,103) -> Kamppi/Center
  - Metro station at Keilaranta (but no metro!)
  - Same side, towards the bridge (194,195,503-6) -> Center/Pasila
  - Bus stops to arrive at CSC at the same positions, just on opposite sides

- If you came by car: parking is being monitored - ask for a temporary parking permit from the reception (tell which workshop you're participating) 
- Visiting outside: doors by the reception desks are open
- Room locked during lunch
  - You can leave stuff inside room
  - else, use lockers in lobby
  - lobby remains open
- Username and password for *workstations*: given on-site



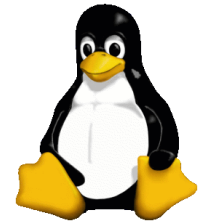
## Around CSC



## What is UNIX/Linux

### Linux ≠ UNIX®

- Linux is a **free** and **open-source software** operating system built around the *Linux kernel*.
  - The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system.
- Linux was originally developed for personal computers based on the Intel x86 architecture, but has since been ported to more platforms than any other operating system.
- Linux is a derivative of the original AT&T Unix operating system.
- The Linux kernel is an **Unix-like** operating system kernel.



By lewing@isc.tamu.edu  
Larry Ewing and The GIMP

7

8

### The “Unix philosophy”

- Unix was designed to be portable, multi-tasking and multi-user in a time-sharing configuration.
- Unix (and thus, unix-like) systems are characterized by various concepts:
  - the use of plain text for storing data;
  - a hierarchical file system;
  - treating devices as files;
  - the use of a large number of small programs that can be strung together through a command-line interpreter, as opposed to using a single monolithic program that includes all of the same functionality.

### Multitasking

- Multitasking is the concurrent execution of multiple tasks (also known as *processes*) over a certain period of time.
  - As a result, a computer executes segments of multiple tasks in an interleaved manner.
- Multitasking automatically interrupts the running program, saving its state (partial results, memory contents and computer register contents) and loading the saved state of another program and transferring control to it.
  - This is called *context switching*.



Time →

9

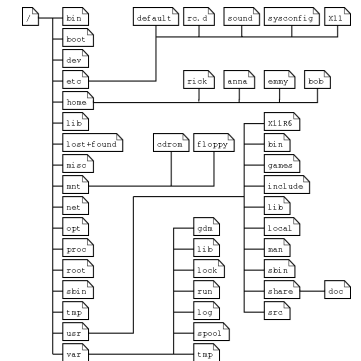
10

## Multi-user

- Multi-user system is operating system software that allows access by multiple users of a computer, typically simultaneously.
- The operating system provides isolation of each user's processes from other users, while enabling them to execute concurrently.
- The filesystem supports multiple users by providing permissions or access rights to specific users and groups for all the files stored on the system.

## Filesystem

- Unix-like operating systems create a virtual file system, which makes all the files on all the devices appear to exist in a single hierarchy. This means there is one *root directory*, and every file existing on the system is located under it somewhere.
  - To gain access to files on another device, the operating system must be informed where in the directory tree those files should appear. This process is called *mounting a file system*.
- Linux supports numerous file system formats, most common ones being ext\* family (ext2, ext3 and ext4), XFS, ReiserFS and btrfs.

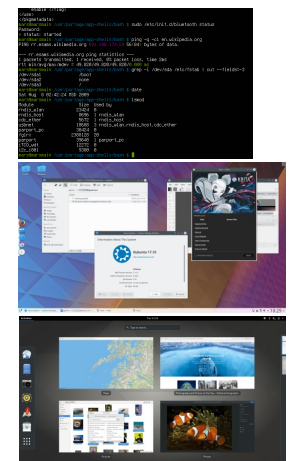


## Linux Distributions

- A Linux distribution (often abbreviated as *distro*) is an operating system made from a software collection, which is based upon the *Linux kernel* and, often, a *package management system*.
- A typically comprises of a Linux kernel, GNU tools and libraries, additional software, documentation, and a desktop environment.
- Almost six hundred Linux distributions exist, with close to five hundred out of those in active development.
  - Debian (Ubuntu, Mint, Knoppix) and Red Hat (Fedora, RHEL, CentOS) are the most common ones.
  - Whether Google's Android counts as a Linux distribution is a matter of definition.
  - <http://www.distrowatch.org/>

## User Interfaces: CLI and GUI

- *Command-line interfaces*, or CLI shells, are text-based user interfaces, which use text for both input and output.
  - The dominant shell used in Linux is the Bourne-Again Shell (bash).
  - Most low-level Linux components use the CLI exclusively.
  - The CLI is particularly suited for automation of repetitive or delayed tasks, and provides very simple inter-process communication.
- On desktop systems, *graphical user interfaces*, or GUIs, are the most common ones providing extensive desktop environments.
  - Typical ones are the K Desktop Environment (KDE), GNOME, MATE, Cinnamon, Unity, LXDE, Pantheon and Xfce, though a variety of additional user interfaces exist.



## Linux on my own computer

## Running your own Linux

- Basically, three options:

1. Run native Linux on your computer
  - Includes the option of *dual boot* (two OS's side-by-side, but optionally booting into one of them)
  - Not recommended: run as live-system (boot from USB/CD)
2. Run it inside a Virtual Machine
3. Run it remotely over the network
  - Includes remote login and remote desktops
  - Needs a network connection

15

16

## Dual boot

- Boot loader in the beginning gives choice of which OS to load
- Pros:
  - native Linux works faster and all resources of the computer are dedicated to a single OS
  - Windows file-system can be mounted in Linux
- Cons:
  - changing between OS's needs reboot of machine
  - Mounting of Linux/Unix file-systems on Windows at least problematic

17

## Dual boot

- I have a Windows machine, what do I have to do to install Linux in parallel (as dual boot) to it?:
  1. Provide a separate disk(-partition) on computer
    - It is possible (e.g., in Ubuntu) to install into existing Windows system, but you loose performance
    - Some installation medias allow for live-mode (Linux running from USB/CD) and have a repartitioning program within (always backup your data!)
  2. Download the image of your favorite Linux distribution (see later)
  3. Installation generally guides you also through boot-loader configuration

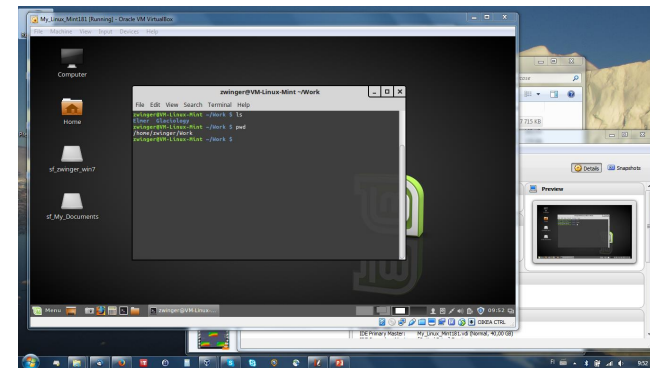
18

## Virtual machines



- Running an application inside your native OS that emulates hardware on which you can install another OS
- Pros:
  - Seamless integration of Linux (guest) in host system
  - Data exchange between guest and host
  - Suspend system (no new boot, leave applications open)
  - Backup and portability (copy to new computer)
- Cons:
  - Performance loss of guest system (SW layer between emulated and real hardware)
  - Shared resources between guest and host

## Virtual Machines



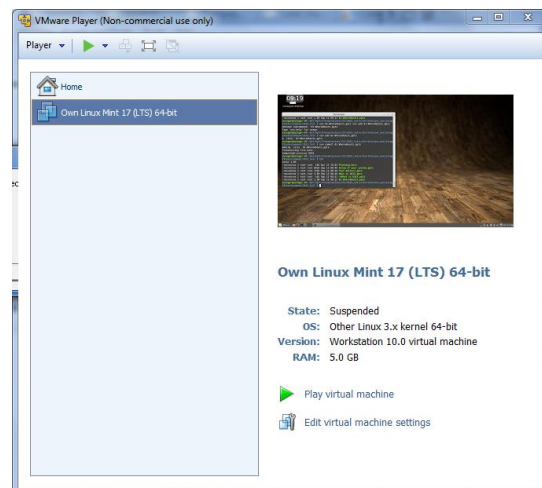
19

20

## Virtual Machines



- The machine can be suspended as is
- Upon relaunch, the user gets the system as she/he left it



21

## Virtual machines



- I have a Windows computer. How can I install Linux running in a Virtual Machine (VM)?
  1. Make sure you have the hardware to support a VM (CPU, memory > 2GB, disk-space)
    - Some older CPUs do not support virtualization
  2. Download a VM software (see next slide) and install it
  3. Download an image of your favorite Linux distribution (see later)
  4. Mount the medium in your VM and install as if it would be a normal computer
  5. Instead of 3+4: Download a ready made virtual appliance (~virtual computer system)

22

## Virtual machines

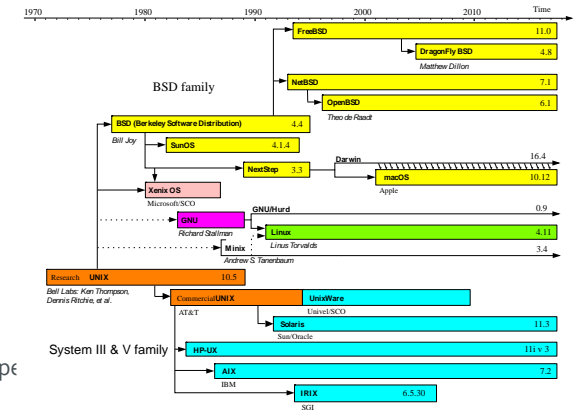
- Two main vendors for VM packages:
  - VMware™ Player** (free-of-charge)
    - Only max 4 cores supported in VM
    - Restricted to non-commercial use
  - Oracle (former Sun) **VirtualBox** (open-source)
    - Supports even VMWare virtual disks
- Usually, additional tools (e.g. VMware-tools) have to be installed
- Important to know the hardware
  - especially CPU type (32- or 64bit)
  - Might need adjustments in BIOS (virtualization)
- Virtual Appliances: Google or **FUNET**
  - Only download appliances you trust!



23

## Mac OSX = UNIX + bling

- The underlying system to your Apple is Darwin, which is a fork from BSD and hence UNIX
  - Darwin is actually open-source
  - The rest of your Apple computer apparently not
- You can use the terminal from within your Mac just like a UNIX shell
- You can even display UNIX-type graphics (see later)



24

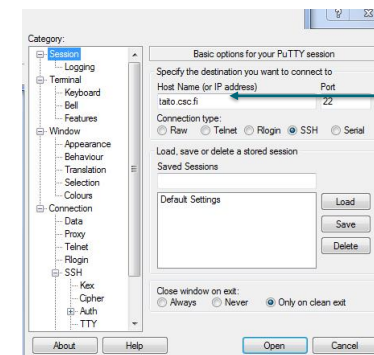
## Remote connection

- From OS X:
  - ssh and X available – like from a Linux machine
- From Windows \*:
  - Needs a ssh client: e.g. **PuTTY**
  - If graphics, needs a X11-emulator: e.g. **Xming**
- Remote desktops:
  - Needs a server running (and network connection)
  - Certain software (client + server)
  - CSC is maintaining such a service (see [CSC environment course](#)): **NoMachine**, **NX**



25

## Remote Connections from Windows

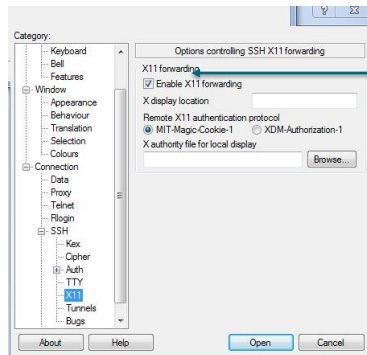


Insert host computer here



26

## Remote Connections from Windows



Tick here to forward X11 (=graphics); needs X11 emulator installed and activated

## Remote Connections from Windows



```
please use "newgrp groupname". You can find more information at:
http://tinyurl.com/kozfa6t

2014-11-27: For jobs requiring more than 16 GB memory per core, please
use the 'hugemem' queue consisting of two 1.5 TB memory nodes with
32 cores each.

2015-01-19: 407 nodes with Intel Haswell processors added to the existing
Sandy Bridge ones and available for the users. A batch job can be
constrained to either Sandy Bridge or Haswell nodes using option
--constraint=snb or =hsw, more details available at:
https://research.csc.fi/taito-constructing-a-batch-job-file#3.1.4

CRITICAL: You belong to the project hpc, which has used all its CPU time quota.
Project hpc is not allowed to submit any new jobs.
Please apply for more CPU time, the instructions are at: https://research.csc.fi/
/applying-for-computing-resources

[zwinger@taito-login3 ~]$ xterm
PuTTY X11 proxy: unable to connect to forwarded X server: Network error: Connect
ion refused
xterm Xt error: Can't open display: localhost:32.0
[zwinger@taito-login3 ~]$
```



## 1<sup>st</sup> Utilization of Linux

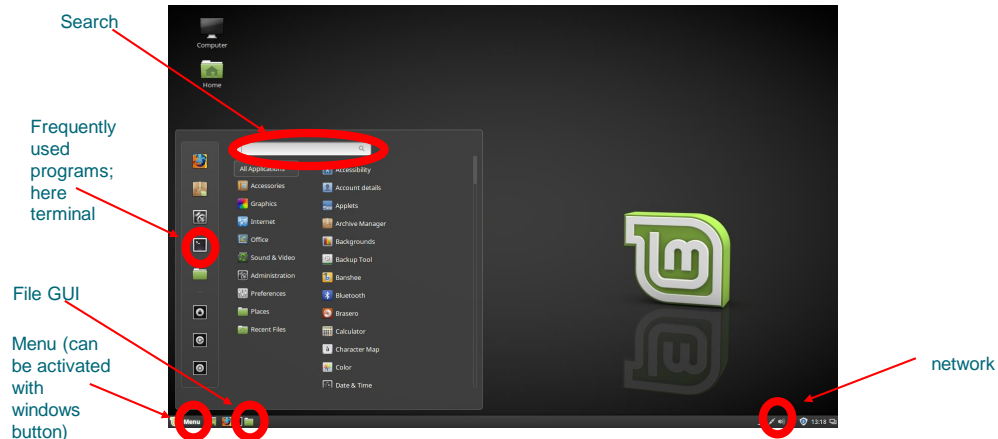
### Starting

- Terminal
- Network
- Web browser
- Install new software
- Text editor, e.g., *gedit*, *nano*
  - Take a note of what you've learnt

29

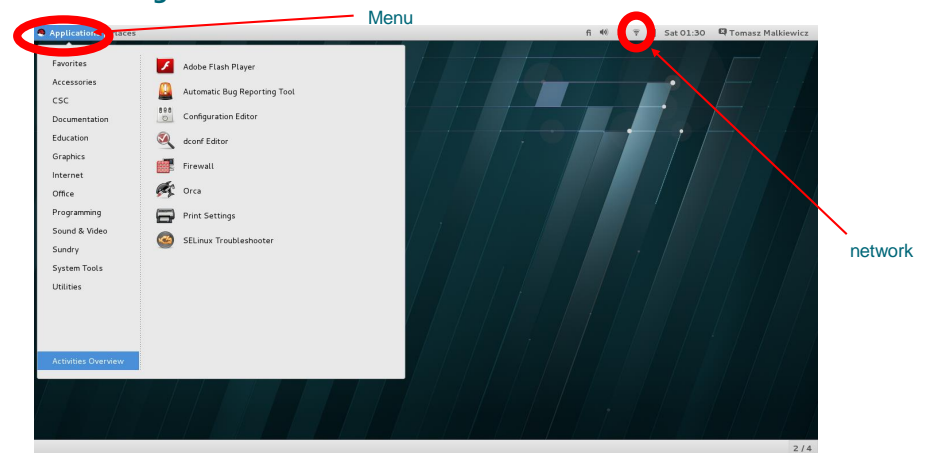
30

### Linux Mint 18



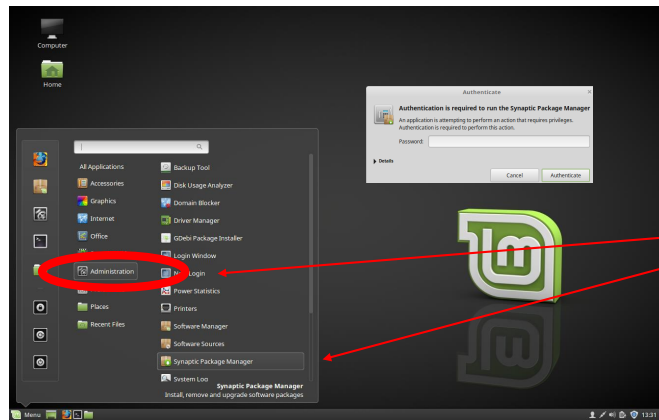
31

### Same thing on RHEL



32

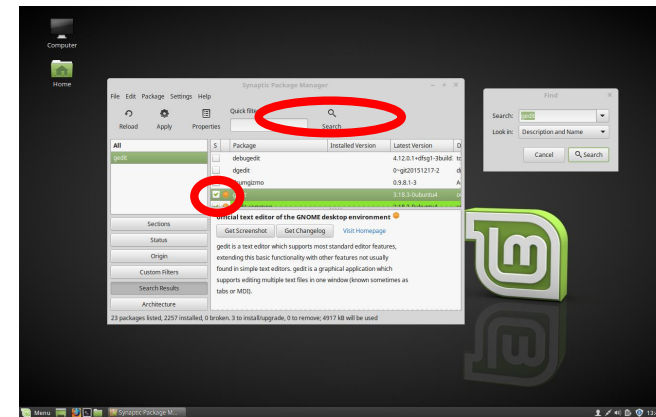
## Installation of software package



- Press  
1) Administration  
2) Synaptic ...

33

## Installation of software package



- 1) Press Search  
2) Give "gedit"  
3) Mark  
4) Press apply

34

## Short exercise



- Try to now find **gedit** from the menu and launch
- Start to fill in the opened blank file with a log of your actions
- First entry could be:
  - Installation of new software: synaptic

35

## A first glimpse of the shell

### Contents

- What is a shell?
- What is a command?
- Listing of directories: `ls`
- Contents of a file: `cat`, `less`
- Moving around in file tree: `cd`, `pwd`
- Directories (creating, changing into and out, removing): `mkdir`, `rmdir`
- Files (creating, redirecting, re/moving):

36

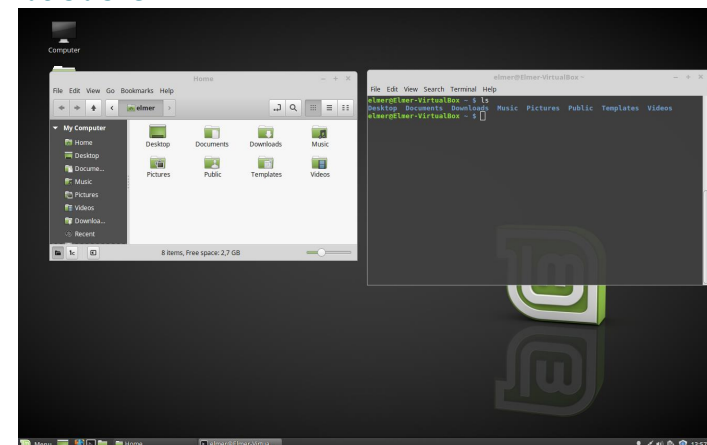
37

### What is a shell?

- “A **shell** in computing provides a **user interface** for access to an operating system’s **kernel services**.” (Wikipedia)
- Remote login:
  - Normally no GUI (Graphical User Interface)
  - Text shell: Terminal with a set of commands
- Different flavours:
  - `bash` (default), `tcsh` (old default), `zsh`, `corn-shell`, ...

programmable

### What is a shell?



38

39



## What is a command?

- A command is a small program provided by the shell
- The over-all structure of a command is:

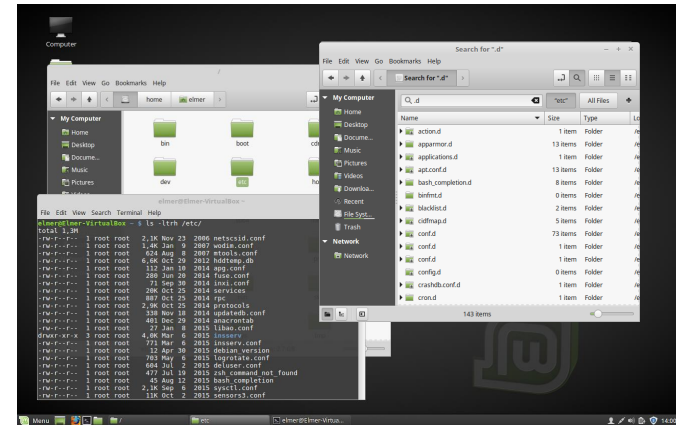
**command -option [optional input]**

- Example: " \$" is not part of the command, but depicts the command prompt

\$ ls -lsh /etc/init.d (we will see later)

- Case sensitive? Try: **Ls -lsh /etc/init.d**
- How to find a command? **\$ apropos list**
- How to find all options? **\$ man ls**

## Listing of directories



40

41



## Listing of directories

- Print contents of a directory or information on a file
- Detailed list of directory:
  - \$ **ls -lthr /etc/**
  - l displays additional information (detailed list in Windows)
  - h displays size in human readable format
  - t orders by date (use -r to reverse order, i.e., oldest first)
  - d keeps from going into sub-directories
- Only print directory/filenames matching a **wildcard** expression:
  - \$ **ls -d /etc/\*.d**
- Only print directory/filenames with a 4 char suffix: \$ **ls -l /etc/\*.????**

42



## Contents of a file

- Prints contents of file to screen:
  - \$ **cat /etc/group**
- -n to precede lines with line numbers
- What if the file doesn't fit on the screen?:
  - Open a scrollable view of a file:
    - \$ **less /etc/group**
  - Press q to quit
  - / to search forward, ? to search backwards
  - n to find the next match, N for previous

43

## Moving around in directories



- change directory: `$ cd /etc/`
- print work directory: `$ pwd → /etc`
- go to subdirectory: `$ cd ./init.d`  
`$ pwd → /etc/init.d`
- Relative path: `$ cd ../`  
`$ pwd → /etc`
- Absolute path: `$ cd /etc/init.d`
- Combination: `$ cd ../usr`  
`$ pwd → /usr`
- Where is home?: `$ cd` or `cd ~/`

44

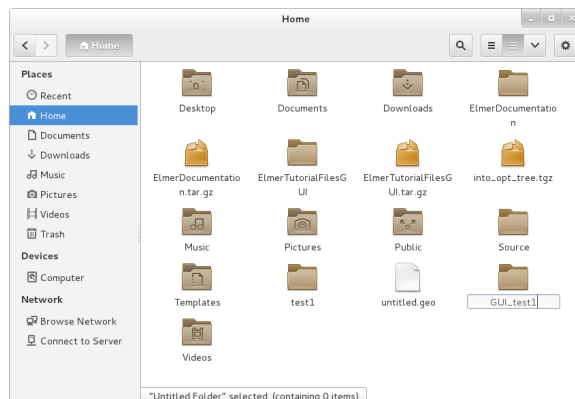
## Creating and (re-)moving directories



- Make a new directory: `$ mkdir test1`
- Relative to (existing) path: `$ mkdir test1/anotherone`
- Recursively: `$ mkdir -p test2/anotherone`
- moving a directory: `$ mv test2 test3`
- removing a directory: `$ cd test1`  
`$ rmdir anotherone`  
`$ cd ..`  
`$ rmdir test1`  
`$ rmdir test3`
- Recursively: `$ rmdir -p test3/anotherone`

45

## Creating and (re-)moving directories



46

## Creating/copying/(re-)moving files



- In UNIX: everything is text  
redirecting output of command/programs into files:  
`$ echo "hello world" > mytest.txt`
- Important: if file exists, it will be overwritten!
  - To prevent it: `$ set -o noclobber`
  - To enable it back: `$ set +o noclobber`
- Appending to existing files:  
`$ echo "hello again" >> mytest.txt`  
`$ cat mytest.txt`  
`$ cat mytest.txt > othertest.txt`

47



## Creating/copying/(re-)moving files

- copy a file: `$ cp mytest.txt othertest2.txt`
- Same recursively with directory:  
`$ mkdir -p test/anotherone`  
`$ cp -r test test2`
- move a file (renaming):  
`$ mv mytest.txt othertest3.txt`  
`$ mv othertest3.txt test2/anotherone`
- remove file(s): `$ rm -f mytest.txt` (-f forces the action)
- Remove recursively: `$ rm -r test2`



## Further resources

- CSC's online user guide: <http://research.csc.fi/csc-guide>
- All the man-pages of the commands mentioned in these slides
- The UNIX-wiz sitting by your side
- Else:
  - <http://www.ee.surrey.ac.uk/Teaching/Unix/index.html>
  - [http://en.wikipedia.org/wiki/List\\_of\\_Unix\\_utilities](http://en.wikipedia.org/wiki/List_of_Unix_utilities)
  - <https://v4.software-carpentry.org/shell/index.html>

## Text Editors: Nano, Emacs, vi

## Nano — the Basic (But Still a Good) Editor for Terminal



- For opening and creating files, type:  
**\$ nano filename**
  - If you are editing a configuration file use the `-w` switch to disable wrapping on long lines as it might render the configuration file unparseable by whatever tools depend on it:  
**\$ nano -w /etc/fstab**
- If you want to save the changes you've made, press **^O**.
  - The carrot symbol `^` denotes *control* key, `ctrl`. Therefore `^O` ought to be read as `ctrl+o`, meaning hold down the control key while pressing the `o` key.
- To exit nano, type **^X**.
  - If you ask nano to exit from a modified file, it will ask you if you want to save it. Just press **n** in case you don't, or **y** in case you do. It will then ask you for a filename. Just type it in and press **Enter**.

50

51

## Learning to Fly with Nano



- To cut a single line, you use **^K**. The line disappears. To paste it, you simply move the cursor to where you want to paste it and punch **^U**. The line reappears.
  - To move multiple lines, simply cut them with several **^K** in a row, then paste them with a single **^U**. The whole paragraph appears wherever you want it.
  - If you need a little more fine-grained control, then you have to mark the text. Move the cursor to the beginning of the text you want to cut. Hit **^6**. Now move your cursor to the end of the text you want to cut: the marked text gets highlighted. Press **^K** to cut the marked text. Use **^U** to paste it.
    - If you need to cancel your text marking, simply hit **^6** again.
- To search for a string, hit **^W**, type in your search string, and press **Enter**. To search for the same string again, hit **M-W**.
  - Notion 'M-' means *meta* key, which typically is the *alt* key.

52

## Editor MACroS for TECO — the GNU Emacs Editor



- For opening and creating files, type:  
**\$ emacs filename**
  - Emacs will detect whether you are working on a GUI and opens in a separate window if you are. To disregard the GUI, use the `-nw` option:  
**\$ emacs -nw filename**
- If you want to save the changes you've made, press **C-x C-s**.
  - In Emacs, `C-` denotes the *control* key. `M-` is the *meta* (usually *alt*) key.
- A file can also be opened while already in Emacs with **C-x C-f**.
  - A new file will be created if it does not exist yet.
- To exit Emacs, type **C-x C-c**.
  - If you ask Emacs to exit from a modified file, it will ask you if you want to save it. Just press **n** in case you don't (and confirm with **yes**), or **y** in case you do.

53



## Abort! And Some Other Useful Stuff in Emacs

- You can abort any command by hitting **C-g**.
- If you have multiple files open in Emacs you can switch between them with **C-x ←** and **C-x →**, or list all the buffers with **C-x C-b**.
  - To actually use the buffer list you need to switch to that window with **C-x o**.
  - Type **C-x 1** to close other windows, and **C-x 0** to close the current window.
- Emacs' cursor movement commands are useful also in bash!
  - Use **C-a** and **C-e** to jump to the beginning/end of current line.
  - **M-b** takes you backwards a word at a time, and **M-f** moves forward.
  - Hit **M-backspace** or **M-d** to delete the word left/right to the cursor.
  - Or, to kill the rest of the line from the cursor, hit **C-k**.
- Typos and editing errors are inevitable, hence **C-/** aka Undo helps.
  - Also **C-\_** and **C-x u**.



## Not for the Faint-Hearted — the vi Editor

- For opening and creating files, type:  
**\$ vi filename**
- vi is a *modal editor* and it always opens in *command mode*, so it only understands commands.
  - In this mode, you can do everything else but insert or edit the text.
  - You switch to *insert mode* with commands like **insert**, **open**, or **append**, and back to command mode with **esc**.
- To save the changes you've made, type **esc : w enter**.
- To exit vi, type **esc : q**
  - vi will not allow you to exit from a modified file. If you really want to discard your changes, you need to type **esc : q !**
  - To save and exit on one go type **esc : w q**



## File Permissions

### File permissions

- UNIX distinguishes between users, groups and others
  - Check your groups: `$ groups`
- Each user belongs to at least one group
- `ls -l` displays the attributes of a file or directory

`-rw-r--r-- 1 userid groupid 0 Jan 29 11:04 name`

type  
user  
group  
others

`r` = read, `w`=write, `x`=execute

The above configuration means: user can read + write, group and all others only read

### File permissions

- Changing permissions with `chmod`
  - `$ cp /etc/group lala`
  - `$ ls -l lala`
    - `-rw-r--r-- 1 userid groupid 0 Jan 29 11:04 lala`
  - `$ chmod o-r,g+w,u+x lala`
  - `$ ls -l lala`
    - `-rwxrw---- 1 userid groupid 0 Jan 29 11:04 lala`
  - `$ chmod u-xrw lala`
  - `$ less lala`

"\$" is not part of the command, but depicts the command prompt

### File permissions

- Changing group `chgrp` and user `chown`
  - `$ chgrp othergrp lala`
  - `$ chown otherusr lala`
  - `$ ls -l name`
    - `$ rwxrw---- 1 otherusr othergrp 0 Jan 29 11:04 lala`



## File permissions

- You can make a simple text file to be executed – **your first script**
  - Scripts are useful for workflows, when you repeatedly have to do the same sequence of commands
- Open file **befriendly.sh** and insert following lines:

```
#!/bin/bash
echo "Hello and welcome"
echo "today is:"
date
echo "have a nice day"
```

- Change to executable:

```
$ chmod u+x befriendly.sh
$ ./befriendly.sh
```

## Job management (in shell)

## Managing jobs

- By default commands (jobs) are run in foreground, e.g.,  
**\$ emacs newfile**
- Now, try to enter something in your shell
  - It does not respond. Why?
  - emacs (the currently running program) blocks the shell as long as you do not quit it
- Killing a job: in shell press **Ctrl + C**
  - That is not usually recommended, as you might lose data
  - Do that only when program gets unresponsive

61

62

## Managing jobs

- Launch again into foreground  
**\$ emacs newfile**
- Type something into emacs
- Suspending a job: in shell press **Ctrl + Z**
  - Shell reports on stopped job
  - type a command into the shell: **\$ ls -ltr**
  - Try to type something into emacs. What happens?
  - The process of emacs is suspended, hence does not accept any input<sup>\*)</sup>

<sup>\*)</sup> In fact, the input buffer keeps the typed stuff and will fill it into emacs, once it is active again

63

## Managing jobs

- Sending the suspended job to background: **\$ bg**
  - type a command into the shell: **\$ ls -ltr**
  - type something into emacs
  - It works now for both!
- Fetching back to foreground: **\$ fg**
  - Shell is blocked again
  - emacs accepts input (but press exit)
- Launching directly into background:  
**\$ xterm -T "no 1" &**  
**\$ xterm -T "no 2" &**

64



## Managing jobs

- Listing jobs of shell: **\$ jobs**

```
[1] - Running      xterm -T "no 1" &  
[2]+ Running      xterm -T "no 2" &
```

- Explicitly bring to foreground: **\$ fg %2**

- Send it back again: **Ctrl + Z**    **\$ bg**

- Killing job: **\$ kill -9 %2**  
**\$ jobs**

```
[1] - Running      xterm -T "no 1" &  
[2]+ Killed        xterm -T "no 2"
```

- Playing terminator: **\$ for i in {1..5};do xterm & done**  
**\$ pkill xterm**

## Setup of your system

## The Environment

- When interacting with a host through a shell session, there are many pieces of information that your shell compiles to determine its behaviour and access to resources.
- The way that the shell keeps track of all of these settings and details is through an area it maintains, called the *environment*.
- Every time a shell session spawns, a process takes place to gather and compile information that should be available to the shell process and its child processes.

66

67

## How the Environment Works

- The environment is implemented as strings that represent key-value pairs and they generally will look something like this:  
KEY=value1:value2:...
  - By convention, these variables are usually defined using all capital letters.
- They can be one of two types, *environmental variables* or *shell variables*.
  - Environmental variables are variables that are defined for the current shell and are inherited by any child shells or processes.
  - Shell variables are contained exclusively within the shell in which they were set or defined.

68

## Printing Shell and Environmental Variables

- We can see a list of all of our environmental variables by using the **printenv** command.
- The **set** command can be used for listing the shell variables.
  - If we type set without any additional parameters, we will get a list of all shell variables, environmental variables, local variables, and shell functions.
  - The amount of information provided by set is overwhelming and there is no way limiting the output to shell variables only.
    - You can still try with  
`$ comm -23 <(set -o posix; set | sort) <(printenv | sort)`

69



## Common Environmental and Shell Variables

- Some environmental and shell variables are very useful and are referenced fairly often. Here are some common environmental variables that you will come across:
  - SHELL: This describes the shell that will be interpreting any commands you type in. In most cases, this will be bash by default.
  - USER: The current logged in user.
  - PWD: The current working directory.
  - OLDPWD: The previous working directory. This is kept by the shell in order to switch back to your previous directory by running 'cd -'.

70



## Common Environmental and Shell Variables (cont'd)

- LS\_COLORS: This defines colour codes that are used to optionally add coloured output to the ls command. This is used to distinguish different file types and provide more info to the user at a glance.
- PATH: A list of directories that the system will check when looking for commands. When a user types in a command, the system will check directories in this order for the executable.
- LANG: The current language and localization settings, including character encoding.
- HOME: The current user's home directory.
- \_: The most recent previously executed command.

71



## Common Environmental and Shell Variables (cont'd)

- COLUMNS: The number of columns that are being used to draw output on the screen.
- DIRSTACK: The stack of directories that are available with the pushd and popd commands.
- HOSTNAME: The hostname of the computer at this time.
- PS1: The primary command prompt definition. This is used to define what your prompt looks like when you start a shell session.
  - The PS2 is used to declare secondary prompts for when a command spans multiple lines.
- UID: The UID of the current user.

72



## Setting Shell and Environmental Variables

- Defining a shell variable is easy to accomplish; we only need to specify a name and a value:
 

```
$ TEST_VAR='Hello World!'
```

  - We now have a shell variable. This variable is available in our current session, but will not be passed down to child processes. We can see this by grepping for our new variable within the set output:
 

```
$ set | grep TEST_VAR
```
  - We can verify that this is not an environmental variable by trying the same thing with printenv:
 

```
$ printenv | grep TEST_VAR
```

73



## Setting Shell and Environmental Variables (cont'd)

- Let's turn our shell variable into an environmental variable. We can do this by exporting the variable:  
**\$ export TEST\_VAR**
  - We can check this by checking our environmental listing again:  
**\$ printenv | grep TEST\_VAR**
- Environmental variables can also be set in a single step like this:  
**\$ export NEW\_VAR="Testing export"**
- Accessing the value of any shell or environmental variable is done by preceding it with a \$ sign:  
**\$ echo \$TEST\_VAR**

74



## Demoting and Un-setting Variables

- Demoting an environmental variable back into a shell variable is done by typing:  
**\$ export -n TEST\_VAR**
  - It is no longer an environmental variable:  
**\$ printenv | grep TEST\_VAR**
  - However, it is still a shell variable:  
**\$ set | grep TEST\_VAR**
- To completely unset a variable, either shell or environmental, use the unset command:  
**\$ unset TEST\_VAR**

75



## Setting Variables at Login

- We do not want to have to set important variables up every time we start a new shell session, so how do we make and define variables automatically?
- Shell initialization files are the way to persist common shell configuration, such as:
  - \$PATH and other environment variables;
  - shell tab-completion;
  - aliases, functions; and
  - key bindings.

76



## Shell Modes

- The bash shell reads different configuration files depending on how the session is started. There are four modes:
  - A *login shell* is a shell session that begins by authenticating the user. If you start a new shell session from within your authenticated session, a *non-login shell* session is started.
  - An *interactive shell* session is a shell session that is attached to a terminal. A *non-interactive shell* session is one that is not attached to a terminal session.
- Each shell session is classified as either login or non-login and interactive or non-interactive.

77

## Some Common Operations and Shell Modes



Operation	Shell modes
log in to a remote system via SSH: <code>\$ ssh user@host</code>	login, interactive
execute a script remotely: <code>\$ ssh user@host 'echo \$PWD'</code>	non-login, non-interactive
execute a script remotely and request a terminal: <code>\$ ssh user@host -t 'echo \$PWD'</code>	non-login, interactive
start a new shell process: <code>\$ bash</code>	non-login, interactive
run a script: <code>\$ bash myscript.sh</code>	non-login, non-interactive
run an executable with <code>#!/usr/bin/env bash</code> shebang	non-login, non-interactive
open a new graphical terminal window/tab on Mac OS X	login, interactive
open a new graphical terminal window/tab on Unix/Linux	non-login, interactive

78

## The Aliases

- An alias is a (usually short) name that the shell translates into another (usually longer) name or command.
- Aliases allow you to define new commands by substituting a string for the first token of a simple command.
- They are typically placed in the `~/.bashrc` file so that they are available to interactive subshells.



80

## Shell Initialization Files



- A session started as a login session will read configuration details from the `/etc/profile` file first.
  - It then reads the first file that it can find out of `~/.bash_profile`, `~/.bash_login`, and `~/.profile`.
- A session defined as a non-login shell will read `/etc/bash.bashrc` and then the user-specific `~/.bashrc` file to build its environment.
- Non-interactive shells read the environmental variable called `BASH_ENV` and read the file specified to define the new environment.

79

## Listing and Creating Aliases



- The general syntax for the alias command varies somewhat according to the shell. In the case of the bash shell it is `alias [name="value"]`
  - *name* is the name of the new alias and *value* is the command(s) which the alias will initiate.
  - The alias name and the replacement text can contain any valid shell input except for the equals sign '='.
- When used with no arguments, `alias` provides a list of aliases that are in effect for the current user:  
`$ alias`

81





## Listing and Creating Aliases (cont'd)

- An example of alias creation could be the alias `p` for the commonly used `pwd` command:  
`$ alias p="pwd"`
- An alias can be created with the same name as the core name of a command; it is the alias that is called, rather than the command:  
`$ alias ls="ls --color=auto -F"`
  - Such an alias can be disabled temporarily by preceding it with a backslash:  
`$ \ls`
  - An alias does not replace itself, which avoids the possibility of infinite recursion.

82



## Listing and Creating Aliases (cont'd)

- You can nest aliases:  
`$ alias l="ls -l"`  
`$ alias lc="l | wc -l"`
  - Now you can even change the alias for `l` and have the changed behaviour in alias `lc`, too.
- Use the `unalias` built-in to remove an alias:  
`$ unalias l lc`
- Aliases are disabled for non-interactive shells (that is, shell scripts); you have to use the actual commands instead.

83



## Bringing It All Together

```
$ cat .bashrc
# .bashrc

set +o noclobber
umask 0027

PS1='\h:\W\$ '
export PS1

alias ls='ls --color=auto -F'
alias guc=globus-url-copy

unalias ll 2> /dev/null
unalias rm 2> /dev/null

function ll() { ls --color -laF "$@" | more; }
function psg() { ps -fp $(pgrep -d, -f "$@"); }
function rot13() { if [ -r $1 ]; then cat $1 | tr '[N-ZA-Mn-za-m5-90-4]' '[A-Za-z0-9]';
else echo $* | tr '[N-ZA-Mn-za-m5-90-4]' '[A-Za-z0-9]'; fi }
```

84

## A second look at the shell

### Recap: what is shell?

- text-only user interface
  - Unix: Shell, Mac: Terminal, Windows: DOS prompt
  - Optimized to work with files --- and everything is a file in Linux
- Interactive programming
  - Bash language
  - It is very easy to write new small programs/commands, scripts
- Programmable UI
  - That makes it powerful and that's why we love it!

**Be a programmer, not a computer!**

85

86

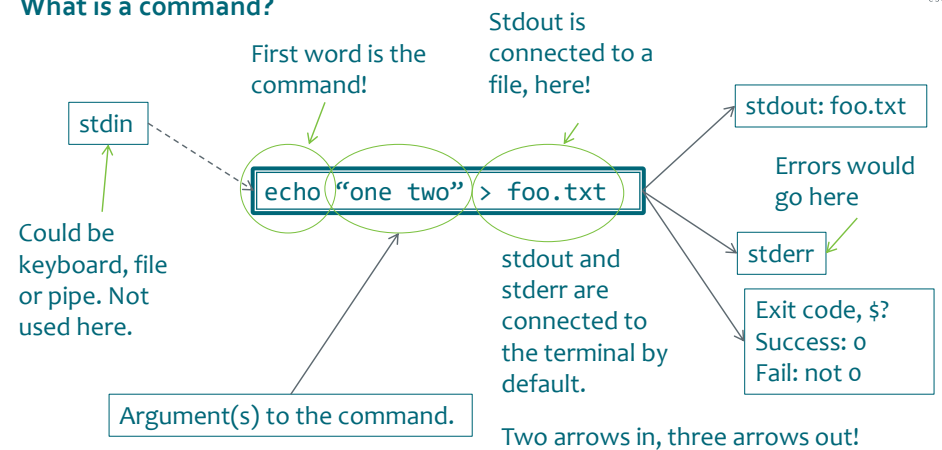
### What happens when you press ENTER?

Bash

- expands variables and file name wildcards (*globbing*)
  - splits the line into "words"
  - Connects files and pipes
  - Interprets the first word as the name of the command, and the rest of the words as the arguments to the command
- ...etc. See **man bash** for details.

*...and don't worry if this is not immediately clear...*

### What is a command?



87

88

## Quoting and escaping special characters, i.e. control expansions and word splitting!



- Try to write "Lisa" to a file named "Lunch company". How does bash interpret the command

```
$ echo Lisa > Lunch company
```

- We use single (') or double *quotes* ("), or *escape* the special meaning of space character with backslash (\), or for file names, preferably, *snake\_case* or *CamelCase*:

```
$ echo Lisa > 'Lunch company'
```

```
$ echo Lisa > Lunch\ company
```

```
$ echo Lisa > LunchCompany
```

## \$ command [arguments]



- *Arguments* are strings that the command can interpret as it pleases. There are conventions, though.
- Command *options* usually begin with one or two hyphens,  
**\$ ls -q**
- Some options are followed by an argument,  
**\$ tar -x -f foo.tar.gz**
- Anything without a hyphen is often a file name,  
**\$ cat foo**

6

89

90

## Finding files



- The hard way: **cd** yourself through the tree and **ls**
- The elegant way:

```
$ find /etc -name '*.conf'
```

- searches given directories recursively and prints matching file names
- Stresses file system. If the search takes more than a second or two, CTRL-C, and redefine the search directories

- The quicker and "safe" (cached) alternative:

```
$ locate .conf
```

## Search the contents of the files



- Search for word "network"  
**\$ grep network /etc/init.d/\***
- Recursive search:  
**\$ grep -r network /etc**
- Discard error messages:  
**\$ grep -r network /etc 2> /dev/null**
- Multiple filters using pipes  
**\$ grep -r network /etc 2> /dev/null | grep start | less**

91

92

## Managing space

- How much space is left on my filesystem?

**\$ df -h**

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/sda5	22G	20G	903M	96%	/
/dev/sda1	447M	27M	396M	7%	/boot
.host:/	12G	8.0G	4.1G	66%	/mnt/hgfs

- What are the sub-directories that consume the most disk-space?

**\$ du -sh /\***

```
1.4M bin
6.3M core
44K Desktop
696M Documents
1.2G Downloads
...
```



## Login to a remote machine from the local terminal



- Secure Shell (SSH):

**\$ ssh -X name@target.computer.fi**

o e.g.

**\$ ssh -X trngXX@taito.csc.fi**

o option -X (or -Y) allows "tunneling" application windows so that they open on your local machines screen. Requires also that you have X-server (Xming for Windows, Xquartz for Mac) running locally

93

94

## Copying files to or from a remote machine



- Copy a file to a remote machine:

**\$ scp lala user@taito.csc.fi:\$HOME'**

- Copy a file from remote machine:

**\$ scp user@taito.csc.fi:\$HOME/lala' .**

- If you know a source (=URL) on the internet<sup>1)</sup>:

o Usually: Open browser and download

- Elegantly from the shell:

**\$ wget http://ftp.gnu.org/gnu/hello/hello-2.7.tar.gz**

## Compressing files and entire directories with tar



- Check contents of an archive file:

**\$ tar tvf hello-2.7.tar.gz**

- Unpack:

**\$ tar xvf hello-2.7.tar.gz**

- Take the whole sub-tree and make a single compressed file:

**\$ tar cvzf myhello.tar.gz hello-2.7/**

t – text/contents, v – verbose, f – file name, x – extract, c – create

95

96



## Ideology

- Simple tool programs, commands
- Easy composability of commands (pipes)
- Complex tasks are solved by composing commands together
- And there is a command for everything: top, ps, head, tail, wc, which, time, sort, uniq, cut, paste, sed, awk, bzip2, make,...

97



## Extra tip: Executable notes

- Open one terminal window and one editor window
- Test commands on the terminal – and cut’n’past the working ones to the editor window --> executable notes!
- For example, open an editor and create file **notes.bash**:  
**#!/bin/bash**  
**echo “Counting the number of lines in files \$@”**  
**wc -l “\$@”**
- Count the lines: **\$ bash notes.bash notes.bash**  
*...and don't worry if this takes a bit of time to digest...*

98

## Introduction to Linux Security

## Information Security

- Information security is the practice of preventing unauthorized access, use, disclosure, disruption, modification, inspection, recording or destruction of information. (*Wikipedia*)
- Primary focus is the balanced protection of the *confidentiality*, *integrity* and *availability* of data while maintaining efficiency and productivity.
  - Confidentiality: don't let others access your data.
  - Integrity: don't let others modify your data.
  - Availability: make data available when it is needed.

99

100

## How You Will Be Hacked – If You Do Nothing

- Computer crime
- Vulnerability
- Eavesdropping
- Malware
- Spyware
- Ransomware
- Trojans
- ~~Viruses~~
- Worms
- Rootkits
- Bootkits
- Keyloggers
- Screen scrapers
- Exploits
- Backdoors
- Logic bombs
- Payloads
- Denial of service

## One Risk to Rule Them All

- The single most common risk for you: **loss of data**.
- It's just a matter of a small typo in a command:  
`# rm -rf / *`
- This will delete **every** file and directory on the system **without asking anything**.
- Make backups – and keep them up-to-date!

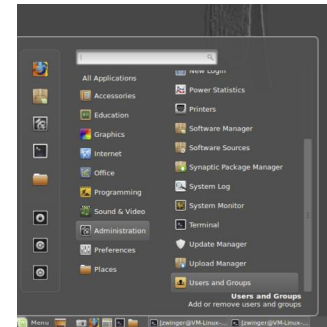
101

102

## System Administration

- Do not run any unnecessary services, like www or email servers.  
**\$ service --status-all**
- Enable firewall.
  - Firewall is a process that monitors and controls incoming and outgoing network traffic.  
**\$ sudo ufw enable && sudo ufw default deny incoming**
- Install patches regularly.  
**\$ sudo apt update && sudo apt upgrade**

## User Administration

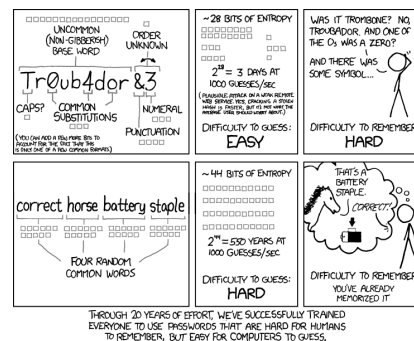


- Users can be managed from command line:  

```
$ sudo useradd -m -G cdrom \  
-c "Joe Cool" someone  
$ sudo passwd someone
```
- Most systems have an administrative graphical user interfaces for doing that job.
- Grant only permissions that are really needed, assign a password and make sure it's a good one.

## A Word About Passwords

- Do **not** use same password(s) for different services.
- Use password managers, e.g. KeePassX.
  - Pros: single master password to remember, cross-platform encrypted vault for all your passwords, great password generator, integration with browsers,...
  - Cons: you can loose all your passwords in one go.

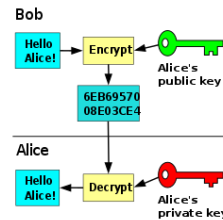


## SSH – The Secure Shell

- SSH offers a secure remote login (and more) over unsecure networks.
  - Comes (almost) always by default with Linux and macOS. On a Windows computer a 3rd party client, e.g. PuTTY, is needed.
- It ensures that all communication to and from the remote server happens in an encrypted manner, and is based on something called ssh keys.
  - Keys eliminate the need for passwords.
  - You win twice: higher convenience and increased security!

## SSH keys

- First you need to create a key pair (unless you already have one), which are used to encrypt and decrypt data:  
**\$ ssh-keygen -t rsa -b 4096**
- This creates two files: `.ssh/id_rsa` and `.ssh/id_rsa.pub` in your home directory.
  - The `id_rsa` is your private key. Keep it only to yourself. It's **private**.
  - The `id_rsa.pub` is your public key. You may think of it as a lock, which opens only with your private key. You may place it anywhere you want; it's **public**.
- You may have as many key pairs as you wish but typically only one is enough.



## Using SSH

- The public key needs to be copied over to a specific file, `~/.ssh/authorized_keys`, on the remote *host* (server) you intend to log in with ssh.  
**\$ scp ~/.ssh/id\_rsa.pub user@host:**  
**\$ ssh user@host**  
host\$ **cat id\_rsa.pub >> .ssh/authorized\_keys**
- The next time you log in to the remote host ssh will be using your keys instead of your password on that host.
  - Note: If you assigned a passphrase for your keys (highly recommended!) ssh will ask for that passphrase. In that case use ssh-agent.

107

108

## Encryption

- Encryption is the process of encoding information in such a way that only authorised parties can access it.
- There are two common methods to encrypt your data:
  - Filesystem stacked level encryption*, where files and directories are encrypted individually with tools like eCryptfs and EncFS.
  - Block device level encryption*, where the whole block device (usually a disk) is encrypted using e.g. dm-crypt and LUKS.

## Example using eCryptfs

- ```
$ mkdir Private
$ sudo mount -t ecryptfs Private Private
```
- Enter a good passphrase and **memorize it**. There is no way getting your data back should you forget your passphrase.
  - Without further ado just accept the default settings.
  - Anything you save in the directory `Private` will now be encrypted.
- ```
$ sudo umount Private
```

109

110