About this course

Program, May 15th

09:30 – 10:00 Morning coffee & registration
10:00 – 10:15 Introduction to the course (whereabouts, etc.)
10:15 – 11:00 What is UNIX/Linux (history and basic concepts, multi-user, multi-tasking, multi-processor)
11:00 – 11:45 Linux on my own computer (native installation, dual-boot, virtual appliances)
11:45 – 12:15 1st utilization of Linux - GUI based (opening terminal from GUI, creating shortcuts)
12:15 – 13:15 Lunch
13:15 – 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

Program, May 16th

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

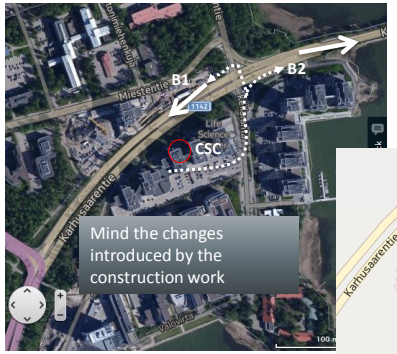
- All topics are presented with interactive demonstrations
  - This is a course for beginners, hence we try to adopt a possibly slow pace
  - Please, indicate immediately, if pace is too fast. We want to have everyone with us all the time
- Additionally, exercises to each of the sections will be provided
- 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

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
  - lobby open, use lockers
- Username and password for *workstations*: given on-site

Around CSC



B1 (102,103)→ Kamppi
B2 (194/5,503/4/... → Pasila,...

(Factory restaurant & salad bar)

What is UNIX/Linux: History and Basic Concepts

What Is UNIX®?

- UNIX is a family of computer operating systems, which are
  - … multitasking,
    - Multitasking is a concept of performing multiple tasks over a certain period of time by executing them concurrently. This allows many more tasks to be run simultaneously than there are CPUs available.
  - … multi-user,
    - Multi-user software is software that allows access by multiple users of a computer at any given time.
  - … derived from original Unix developed at AT&T Bell labs; and
  - … conforms to *Single UNIX Specification* from The Open Group.

Unix-like Operating Systems

- The Open Group owns the UNIX trademark and administers the Single UNIX Specification.
  - Currently (5/2017) there are six operating systems branded as UNIX: Oracle Solaris, Inspur K-UX, IBM AIX, HP HP/UX 11i, Huawei EulerOS, and Apple macOS.
- A Unix-like (referred to as UN*X or *nix) operating system is one that behaves in a manner similar to a Unix system, while not conforming to or being certified to Single UNIX Specification.
  - Any system that is roughly consistent with the UNIX specification: Minix, Plan 9, all BSD variants, GNU/Hurd, GNU/Linux, Cray Linux Environment, …

In the Beginning…

- Unix was born at AT&T's Bell Labs in 1969 as AT&T backed from developing the Multics operating system.
  - A team of Bell Labs researchers developed a hierarchical file system, the concepts of computer processes and device files, a command-line interpreter, and some small utility programs.
  - Started as a non-official project, it was called Unics as a pun for being an emasculated Multics.
- In 1970, text editing and word processing capabilities were added and Bell Labs started using the operating system, now named Unix, internally.

Out from the Labs

- AT&T could not commercialize Unix because an old antitrust decree denied Bell – a part of AT&T – from entering computer business.
  - AT&T began licensing Unix source code for educational institutes.
- Being distributed in source code made Unix a perfect study and research operating system for universities.
- During the turn to 1980s, the influence of Unix in academic circles led to a large-scale adoption of Unix by commercial start-ups.
  - Some notable ones being Solaris by Sun Microsystems, HP-UX by HP, AIX by IBM and Xenix by Microsoft.

Unix Impact

- Unix was written in a high level programming language rather than assembly language.
  - The C programming language soon spread beyond Unix.
- Unix had a drastically simplified file model.
  - Unix popularized the hierarchical file system.
- A fundamental simplifying assumption of Unix was its focus on text for nearly all file formats.
- The TCP/IP networking protocol spread quickly on versions of Unix running on relatively inexpensive computers, which contributed to the Internet explosion.

The Raise and Fall

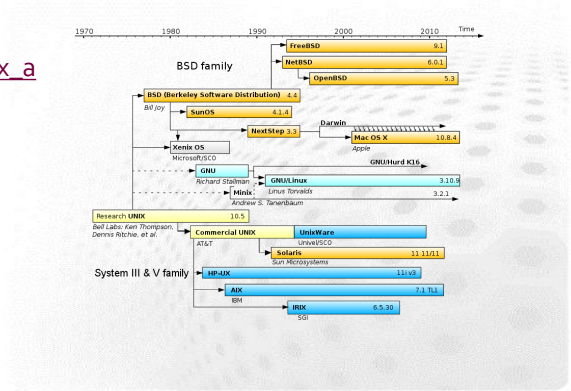
- By the early 1980s thousands of people used Unix at AT&T and elsewhere.
  - Unix was seen as a potential universal operating system, suitable for all computers.
- In 1983, the antitrust ban for AT&T was lifted and they immediately commercialised Unix, nearly killing it with poor licensing terms.
- Since the new UNIX licensing terms were not so favourable for academic use, Berkeley continued to develop BSD Unix.
  - Many contributions to Unix first appeared in BSD releases, most notably the implementation of TCP/IP network code.

The Unix Wars

- Although AT&T created Unix, by the 1980s Berkeley was the leading non-commercial Unix developer.
  - The two common versions of Unix were Berkeley's BSD and AT&T's System V and each vendor's version of Unix was different from others'.
- Since 1996, the Single UNIX Specification, the current standard for branded Unix, is now the responsibility of the Open Group.
- Meanwhile, Unix had got competition from the open source Linux operating system, a reimplementation of Unix from scratch, using parts of the GNU project that had been underway since the mid-1980s.

A Brief Unix Timeline

- The (somewhat) complete Unix history timeline can be found at http://www.levenez.com/unix/unix_a4.pdf
  - Beware, it's 32 pages
  - Updated regularly
  - Makes a nice poster ☺



Linux

- Linux is a Unix-like and mostly POSIX-compliant operating system.
  - The defining component of Linux is the Linux kernel, first released on 5 October 1991 by Linus Torvalds.
- Linux was developed as a free operating system for personal computers based on the Intel x86 architecture, but has since been ported to more computer hardware platforms than any other operating system.
  - It is the leading operating system on servers, mainframe computers and supercomputers.
  - Linux has the largest installed base of all general-purpose operating systems, due to Android being build on top of Linux kernel.

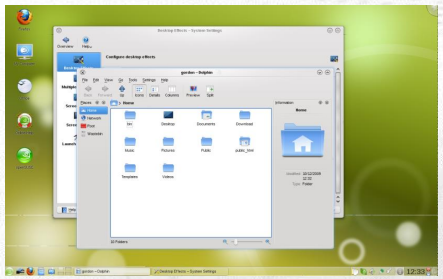
Linux Distributions

- Typically, Linux is packaged in a form known as a Linux distribution.
  - Distributions include the kernel, utilities and libraries, and usually a large amount of applications to fulfil the distribution's intended use.
- Because Linux is freely redistributable, anyone may create a distribution for any intended use – and there are numerous.
  - Some popular mainstream distributions include Debian, Ubuntu, Fedora, and the commercial Red Hat Enterprise Linux.
- Distributions for desktop use typically include X11 windowing system, and an accompanying desktop environment such as GNOME or the KDE Software Compilation.

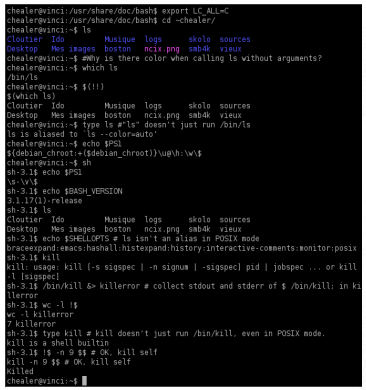
The Desktop

GNOME 3 Desktop

KDE Desktop

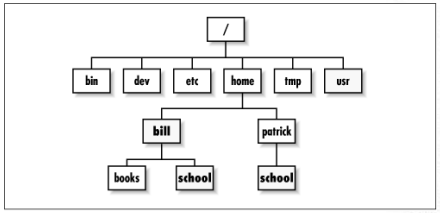


The Shell



- A Unix shell is a command-line interpreter that provides a traditional user interface for the Unix and Unix-like operating systems.
- The shell is an ordinary user-level program, with additional commands provided as separate programs.

The File System



- The hierarchical file system can have arbitrarily nested subdirectories.
- The file system hierarchy also contains machine services and devices, such as printers, terminals, or disk drives, under /dev directory.
  External disk usually *mount* in /mnt or /media.

Read The Friendly Manual

- A man page (short for manual page) is a form of online software documentation usually found on a Unix or Unix-like operating system.
- Topics covered include computer programs (including library and system calls), formal standards and conventions, and even abstract concepts.
- A user may invoke a man page by issuing the **man** command:
  $ **man bash**
  - By default, man typically uses a terminal pager program such as more or less to display its output.

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
   - Depends on network connection

Dual boot

- Boot loader in the beginning gives choice of which OS to load
- Pros:
  - native Linux works faster and all resources of 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

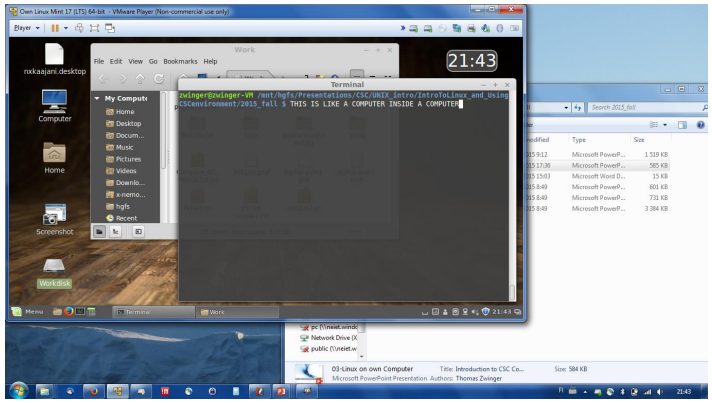
Dual boot

- I have a Windows machine, what do I have to do to install Linux 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

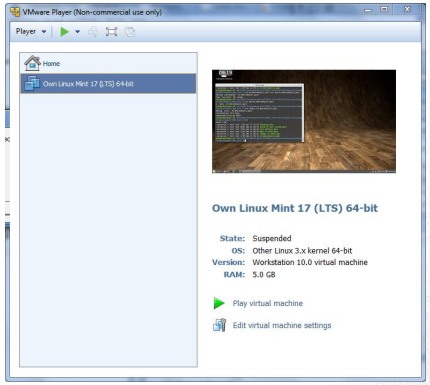
Virtual machines

- 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)
  - Shared resources between guest and host

Virtual Machines



Virtual Machines



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

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)
  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)

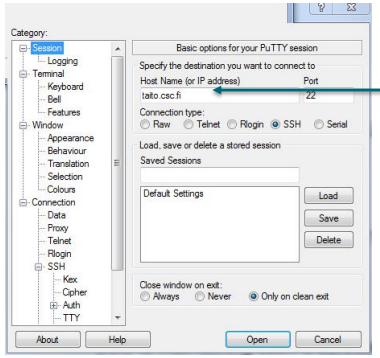
Virtual machines

- 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 (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

Remote connection

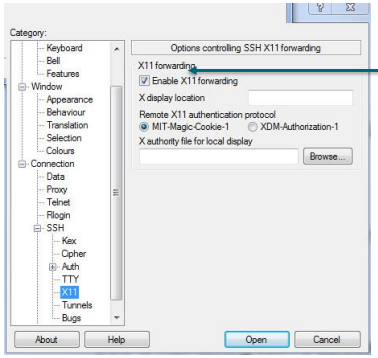
- From OS X:
  - ssh and X available – like from a Linux machine
- From Windows®:
  - Needs a ssh client: e.g. PuTTY
  - If graphics, needs a X11-emulator: e.g. Xming
- Remote desktops:
  - Needs a server running (and network connection)
  - Certain software (client + server)
  - CSC is maintaining such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

Remote Connections from Windows



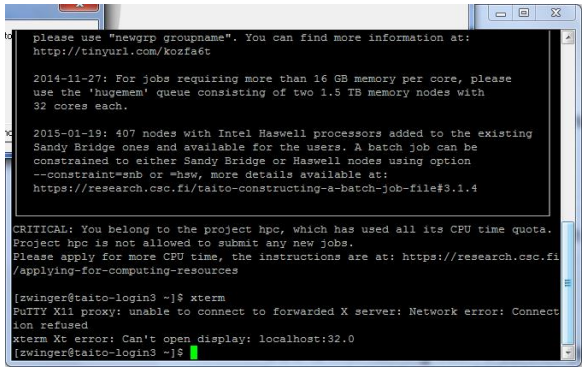
Insert host computer here

Remote Connections from Windows



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

Remote Connections from Windows

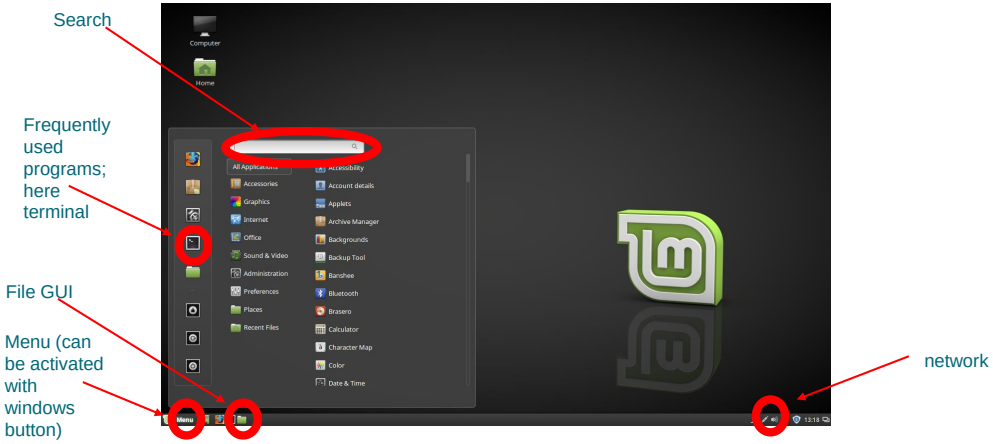


1st Utilization of Linux

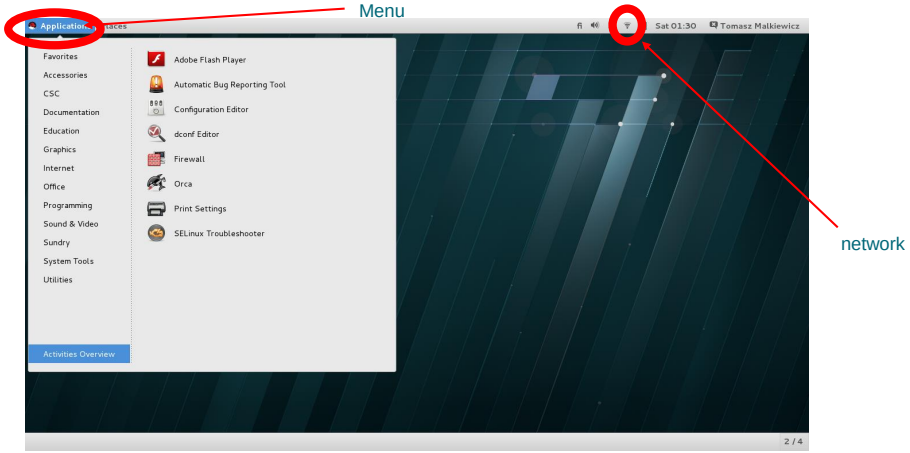
Starting

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

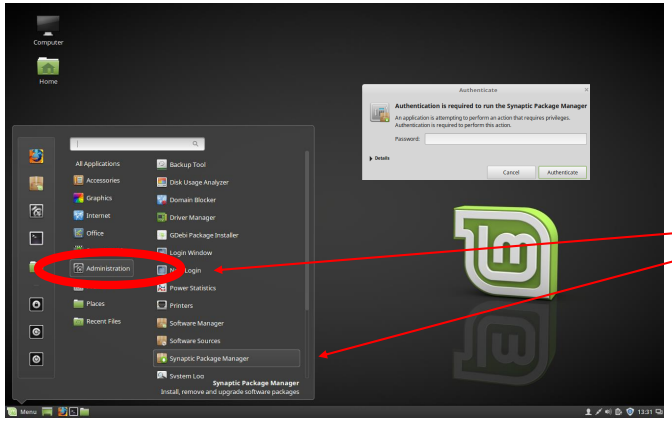
Linux Mint 18



Same thing on RHEL

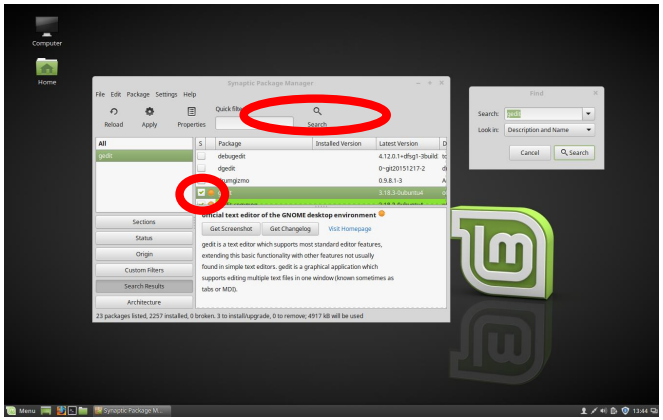


Installation of software package



Press
1) Administration
2) Synaptic …

Installation of software package



1) Press Search
2) Give “gedit”
3) Mark
4) Press apply

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

A first glimpse of the shell

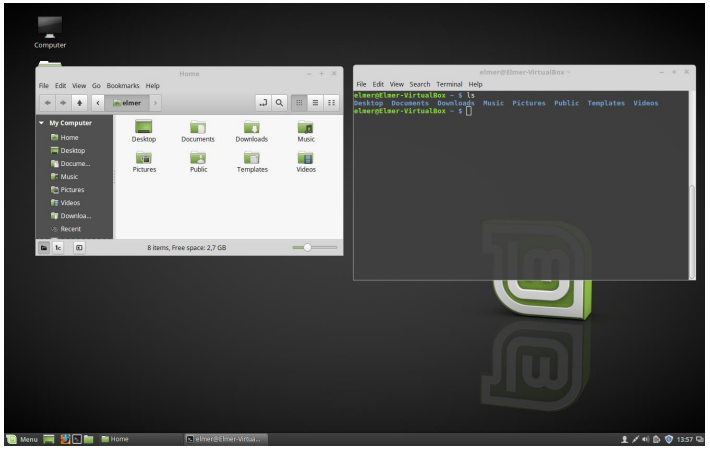
Contents

- What is a shell?
- What is a command?
- Listing of directories
- Contents of a file
- Moving around in file tree
- Directories (creating, changing into and out, removing)
- Files (creating, redirecting, re/moving)

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, …

What is a shell?



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]

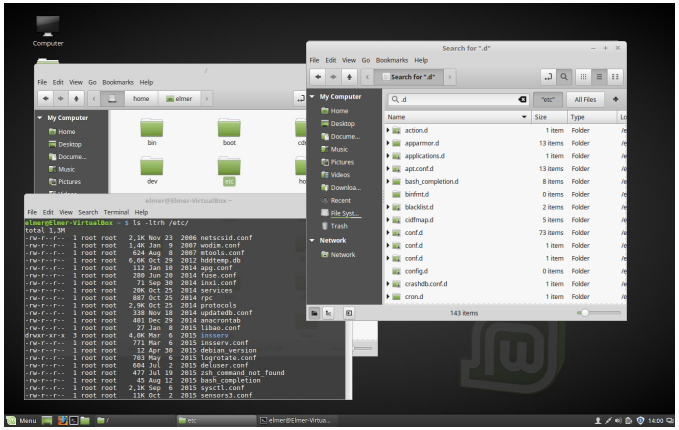
- Example:

  "$" 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



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/*.????**

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

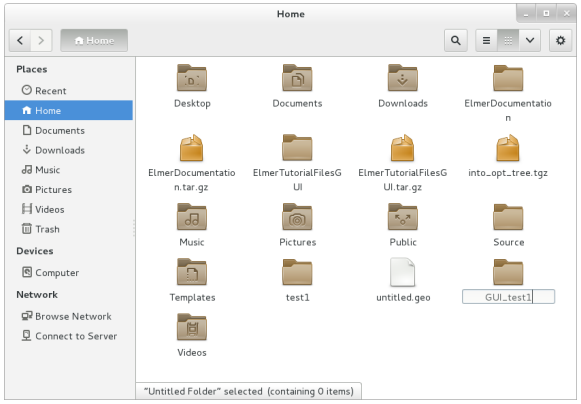
Moving around in directories

- 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 ~/**

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**

Creating and (re-)moving directories



Creating/copying/(re-)moving files

- In UNIX: everything is text
  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

Creating/copying/(re-)moving files

- Copy a file: $ **cp mytest.txt othertest2.txt**
- Same **r**ecursively 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 **r**ecursively: $ **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
  - https://v4.software-carpentry.org/shell/index.html

Text Editors: Gedit, Nano, Emacs, vi

Nano Text Editor

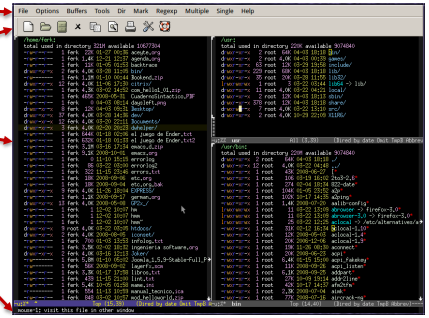
- Nano is an ncurses-based editor that focuses on simplicity.
  - ncurses (new curses) is a programming library which allows the programmer to write text-based user interfaces
  - ...which means it must be run in a terminal window.
  - Nano is a clone of the Pico text editor, the editor for the Pine email client that was very popular, back in the early '90s.
- Nano may be thought as a shell version on Gedit.
  - It is not as powerful as Windows based editors, as it does not rely on the mouse, but still has many useful features.
- Not all systems ship with Nano installed.
- To start Nano: $ **nano filename**

GNU Emacs

- GNU Emacs is an advanced, self-documenting, customizable , extensible text editor — and more.
- The features of GNU Emacs include:
  - Content-sensitive editing modes, including syntax colouring.
  - Complete built-in documentation, including a tutorial for new users.
  - Full Unicode support for nearly all human languages and their scripts.
  - Highly customizable, using Emacs Lisp code or a graphical interface.
  - A large number of extensions that add other functionality, including a project planner, mail and news reader, debugger interface, calendar, and more.

Organization of the Screen

- At the top is a *menu bar*.
- On a graphical display, directly below the menu bar is a *tool bar*.
- The main area is called the *window*.
  - Window is where the *buffer* — the text you are editing — is displayed.
- At the very bottom is an *echo area*.



Kinds of User Input

- Emacs is primarily designed for use with the keyboard.
  - You can use mouse and the menus as well.
- Commands are entered using *modifier keys*:
  - Control (labelled Ctrl), e.g. **C-a**, which means *Ctrl + a*
  - Meta (labelled Alt), e.g. **M-a**, which means *Alt + a*
    - You can also type Meta characters using two-character sequences starting with *Esc*. Thus, you can enter **M-a** by typing *Esc a* and **C-M-a** by typing *Esc Ctrl + a*.
  - A *key sequence* is a sequence of one or more input events that is meaningful as a unit, e.g. **C-x C-h**.

Keys and Commands

- Emacs does not assign meanings to keys directly. Instead, Emacs assigns meanings to named *commands*, and then gives keys their meanings by *binding* them to commands.
  - The command next-line does a vertical move downward. The key **C-n** has this effect because it is bound to next-line. If you rebind **C-n** to the command forward-word, **C-n** will move forward one word instead.
  - The bindings between keys and commands are recorded in tables called *keymaps*.
- This subtle distinction is irrelevant in ordinary use, but vital for Emacs customization – and for reading the help or manual. ☺

Entering Emacs

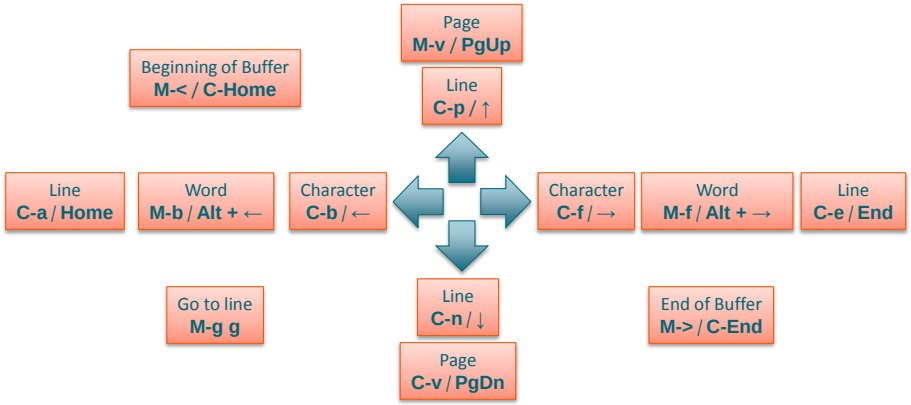
- The usual way to invoke Emacs is with the shell command **emacs**.
  - When Emacs starts up, the window displays a special buffer named '*GNU Emacs*'. This *startup screen* contains information about Emacs and links to common tasks that are useful for beginning users.
    - If the variable inhibit-startup-screen is non-nil, Emacs does not display the startup screen but a scratch buffer instead.
- Using a command line argument, you can tell Emacs to *visit* one or more files as soon as it starts up, e.g. **emacs foo.txt**.

Exiting Emacs

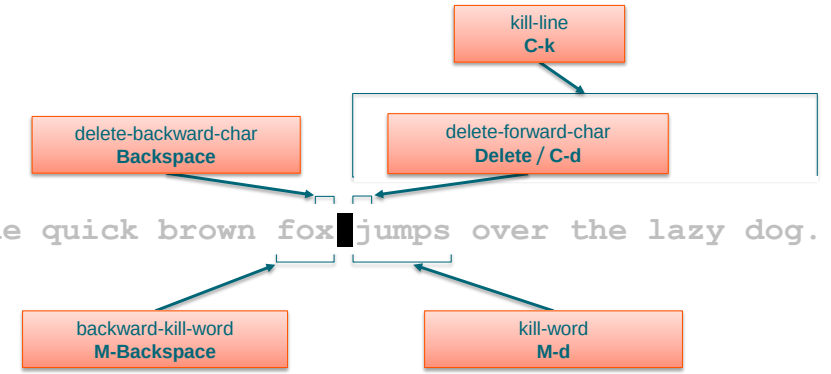
- Killing Emacs means terminating the Emacs program. To do this, type **C-x C-c** (save-buffers-kill-terminal).
  - If there are any modified file-visiting buffers when you type **C-x C-c**, Emacs first offers to save these buffers – with eight (!) different options.
- To kill Emacs without being prompted about saving, type **M-x kill-emacs**.

`y`	Save this buffer and ask about the rest of the buffers
`n`	Don't save this buffer, but ask about the rest of the buffers
`!`	Save this buffer and all the rest with no more questions
`.`	Save this buffer, then exit `save-some-buffers` without even asking about other buffers
`q`	Terminate `save-some-buffers` without any more saving
`C-r`	View the buffer that you are currently being asked about
`d`	Diff the buffer against its corresponding file, so you can see what changes you would be saving
`C-h`	Display a help message about these options

Moving Around



Erasing



Undoing Changes

- Emacs records a list of changes made in the buffer text, so you can undo recent changes. This is done using the undo command, which is bound to **C-/** (as well as **C-x u** and **C-_**).
  - Normally, this undoes the last change, moving cursor back to where it was before the change. The undo command applies only to changes in the buffer; you can't use it to undo cursor motion.
- If you repeat **C-/**, each repetition undoes another, earlier change, back to the limit of the undo information available.
  - If all recorded changes have already been undone, the undo command displays an error message and does nothing.

Files

- To begin editing a file in Emacs, type C-x C-f FILENAME
  - Emacs obeys this command by visiting the file: it creates a new buffer, copies the contents of the file into the buffer, and then displays the buffer for editing.
  - To create a new file, just visit it with C-x C-f as if it already existed.
- If you alter the text, you can save the text in the buffer by typing C-x C-s (save-buffer). This copies the altered buffer contents back into the file, making them permanent.

Buffers

- Emacs can easily handle hundreds of buffers simultaneously.
  - Typically there are a few buffers at any given time as Emacs holds internal buffers such as *Messages* and *Scratch*.
- Switching buffers can be done with C-x b (switch-to-buffer) , which command reads a buffer name using the minibuffer, then makes that buffer current, and displays it in the currently-selected window.
  - C-x ← and C-x → selects previous/next buffer respectfully.
- Use C-x k (kill-buffer) to close buffers.
- To get a list of buffers hit C-x C-b (list-buffers)

Windows (Not the Microsoft Product)

- Each Emacs window displays one Emacs buffer at any time.
  - At any time, one Emacs window is the selected window; the buffer this window is displaying is the current buffer.
  - A single buffer may appear in more than one window.
- Switching between windows is done with C-x o (other-window).
- To delete a window you have (more than) two options:
  - Delete the selected window with C-x 0 (delete-window)
  - Delete all windows except the selected one C-x 1 (delete-other-windows)
- Use C-x 2 and C-x 3 to split a window into two.

Help

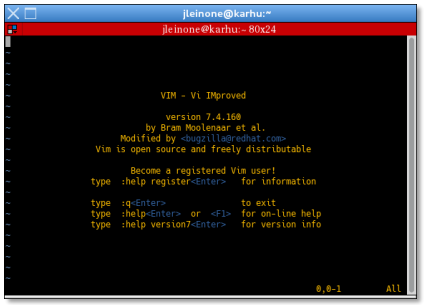
- If you forget what a key does, you can find it out by typing **C-h k** (describe-key), followed by the key of interest.
  - E.g. **C-h k C-n** tells that **C-n** runs the command next-line.
- The prefix key **C-h** stands for "help".
  - The key **F1** (usually) serves as an alias for **C-h**.
- You can visit the built-in Emacs tutorial at any time by typing **C-h t**.
  - Exit the tutorial by killing the buffer by typing **C-x k**.

vi

- vi is the *de facto* standard Unix editor; essentially all Unix and Unix-like systems come with vi (or a variant, like vim) built in.
  - System rescue environments, embedded systems and other constrained environments often include only vi.
- vi is different from many editors in that it has two main modes of operation: *command mode*, and *insert mode*. This is the cause of much of the confusion when a new user is learning vi.
  - Regarding vi's modal nature, some joke that vi has two modes – "beep repeatedly" and "break everything".

An Exercise

- Start vi by typing: **vi**
  - By default you are in *command mode*.
- Switch to the *insert mode* by pressing the **i** key.
- When you have finished entering your sample text, press **Esc** to return to the *command mode*.
- To save the file enter command **:w this-is-my-vi-test-file**



Quitting vi

- The command to quit vi is **:q**
  - You need to be in *command mode*. If you're not already in command mode, or you're not sure which mode you're in, press **Esc**.
- Sometimes, you may have made changes to a file that you do not want to save. To exit vi without saving, and ignoring any warnings about unsaved data, use a variation of the **:q** command, **:q!**
  - This will return you to the prompt, without saving any changes to the file, and with no warnings about unsaved data. Use this command carefully.

Basic Editing in vi

- To enter insert mode and insert text at current position, type **i**, or to append text after current position, type **a**.
- To delete characters while in insert mode, use **Backspace** and **Delete** keys.
- The power of editing text with vi, however, lies in using the command mode. You will find yourself switching between the insert mode and command mode constantly!

Cursor Movement

```
h   Left
j   Down
k   Up
l   Right
w   Forward one word
b   Back one word
e   End of the word
(   Beginning of current sentence
)   Beginning of next sentence
{   Beginning of current paragraph
}   Beginning of next paragraph
0   Start of current line
$   End of current line
^   First non-white character of current
    line
```

- On some systems, to move the cursor around the screen you must be in command mode and use the commands instead of the cursor keys.

  This is mainly due to historical reasons, dating back to when not all terminals had cursor keys.

Editing

a	Append after cursor
A	Append to end of current line
i	Insert before cursor
I	Insert at the beginning of line
o	Insert (open) line above cursor
O	Insert (open) line below cursor
cw	Change word
cc	Change line
C	Change from cursor to end of line
dw	Delete word
dd	Delete current line
D	Delete from cursor to end of line
p	Insert buffer at cursor
yw	Copy word to buffer
u	Undo last change

- By combining edit commands with cursor movement commands, you can quickly edit just the parts of the text you want.

  E.g. **8cw** will replace the next eight words in text with the new text you type, and **2y}P** will duplicate the next two paragraphs.

File Handling

```
:w      Write file
:w!     Write file ignoring any warnings
:w file Write file as file
:q      Quit vi
:q!     Quit even if changes are not saved
:wq     Write file and quit
:x      Quit and write file if changed
%       Display current filename
:r file Insert file after cursor
:r !cmd Run cmd and insert output after
    current line
```

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**

$ **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

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

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" &**

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"
```

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.

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.

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)

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 -'.

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.

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.

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**

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**

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**

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.

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.

Some Common Operations and Shell Modes

Operation	Shell modes
log in to a remote system via SSH: $ ssh user@host	login, interactive
execute a script remotely: $ ssh user@host 'echo $PWD'	non-login, non-interactive
execute a script remotely and request a terminal: $ ssh user@host -t 'echo $PWD'	non-login, interactive
start a new shell process: $ bash	non-login, interactive
run a script: $ bash myscript.sh	non-login, non-interactive
run an executable with #!/usr/bin/env bash 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

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.

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.

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**

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.

Listing and Creating Aliases (cont'd)

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

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 }
```

A second look at the shell

Recap: what is a shell?

- *A shell* is a program which provides the traditional, text-only user interface for Linux (and other Unix like systems).
- *Shell's* primary function is to read *commands* that are typed into a console or terminal window on the *command line* and then *execute* them.
- One can also view the default Linux shell as the REPL of the bash programming language (compare to interactive Python, R, Matlab, etc).

Command line

- Although the user can type any valid (or invalid!) bash language constructs on the command line, the basic use of command line is to *execute simple commands*, such as **ls**, list files.
- A command can be a bash (built-in) function or a program (or an alias)
- In a bit more detail: after pressing ENTER key, bash reads the command line, expands variables (explained later), and interprets the *first word* of the resulting expression as a *command*, and the *rest of the words* as the *arguments to the command* (~ functions and arguments).

$ command [arguments]

- Command 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**

Finding stuff (1)

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

  $ **find /etc -name "*.conf" -print**

  - Finds all config files in the **/etc**-tree
  - Stresses file system, avoid on supercomputer parallel file systems

- The (cached) alternative:

  $ **locate *.conf**

Finding stuff (2)

- Finding expressions inside files:
  - For instance, we want to know all files in the directory **/etc/init.d** that contain keyword "network":

    $ grep network /etc/init.d/*
  - Or recursively: **$ grep –r network /etc**
  - Getting rid of noise:

    $ grep –r network /etc 2> /dev/null
- Piping of output:
  - Instead of re-directing into files, output can be piped in a chain of commands:

    $ grep –r network /etc 2> /dev/null| grep start| less

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

- Only secure connections (no telnet, rlogin) are recommended
- Secure Shell (SSH):

  $ **ssh name@target.computer.fi –X**

  -X tunnels the graphical output, alternatively **-Y**

e.g.: $ **ssh trngXX@taito.csc.fi –Y**

Remote copying

- **scp** is like **cp,** but used for remote transfer

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

  Quotes are important here

- **rsync** works local as well as remotely and helps to keep two (remote) directories in sync:

  $ **mv lala test**

  $ **rsync –avt test/ test2**

  This syncs everything in **test** with **test2**

  Important: Do not drop trailing **/**

  - Remotely: $ **rsync –avt test user@taito.csc.fi:'$HOME'**

Remote download

- **scp** works also with remote computer as source:

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

  Here is a space

- If you know a source (=URL) on the internet[1]:
  - Usually: Open browser and download
- Elegantly from the shell:

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

[1] Be sure you can trust the contents of the source – there is malware also in UNIX!

(De-)compressing files

- Storage and copying of large files: make them smaller
- Several formats supported:
  - **gzip**: **.gz**
  - **zip**: **.zip**
  - **bzip2**: **.bz2, .bz**

(De-)compressing files

- GNU zip:
  gzip
  gunzip
- ZIP:
  zip
- Bzip
  bzip2
  bunzip2

Archives of files

- Most common: tar (tape archive)
  - Take whole sub-tree and make a single compressed file:

    $ **tar cvzf myfirsttarfile.tar.gz /etc/init.d**

    - **c** create new archive, **v** verbosity, **z** gzip simultaneously, **f** target file
- Check contents (and simultaneously gzip):

  $ **tar tvzf hello-2.7.tar.gz**
- Unpack (and simultaneously gzip):

  $ **tar xvzf hello-2.7.tar.gz**

More tools

top, ps, head, tail, wc, which, time, sort, uniq, cut, paste, sed, awk

Use case: write 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
# How to count the number of lines in files
echo "Counting the number of lines in files $@"
wc –l "$@"
```

- Count the lines: **$ bash notes.bash notes.bash**

Introduction to Linux Security

What is Security all about actually?

- Security is a set of appropriate procedures (controls) to protect your resources (your data, your account, your services and your reputation) against **risks**
- The main aspects of security are
  - Confidentiality (don't let others access or forward your confidential data, such as passwords, personal data, business secrets)
  - Integrity (don't let others change your data without permission, beware of malware and hackers)
  - Availability (keep your data and services available for yourself and those who should have access to it)

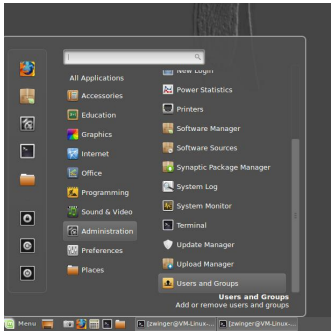
How to protect yourself against risks?

- Compromised account
  - Use only good passwords (hard to guess, easy to remember)
  - 8 chars min, (large alphabet, no dictionary worlds), use password managers (such as KeePass)
  - Be careful with public systems and services (never recycle passwords)
  - User keys instead of passwords (but protect your keys too!)
- System compromise
  - Patch your own system regularly, keep firewall (iptables, ufw) on, use only necessary services and allow only minimal incoming connections
- Denial of Service
  - System getting overwhelmed (and unusable) due to large number of external requests
  - Offer only the necessary services to others
- Surveillance
  - Don't store any confidential information on international cloud services
- Bad user and system administration
  - Beware of forgotten test accounts, patch your system regularly

Security Risks and Compliance

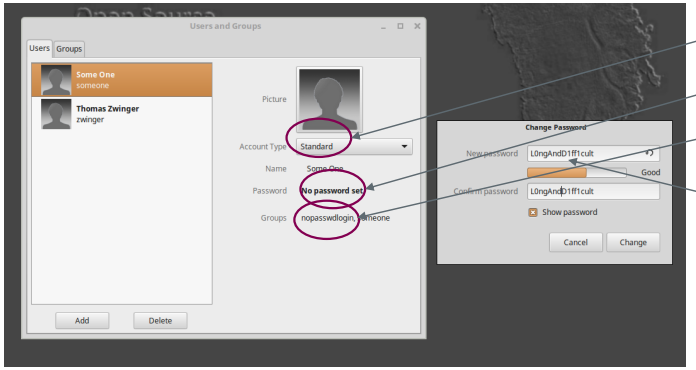
- Typical risks for Linux users:
  - Compromised account (#1!)
  - System compromise and spying
  - Loss of data
  - Surveillance
  - Infrastructure related issues/ Downtime
  - Bad user and system administration
  - Legal issues
- You must comply with laws and Terms of Use:
  - Do not endanger other users or the Infrastructure
  - Protect personal data and other confidential information
  - As a User, you are responsible to protect your account
  - As an administrator (e.g., on your own PC) you are responsible for all accounts

User administration



- You can add users form the command line (tedious)
- Most systems have administrative graphical user interfaces for doing that job
- Dangerous sometimes if default settings there are not tight enough

User administration



Think, if you really need Administrator group for each user

Always set a password

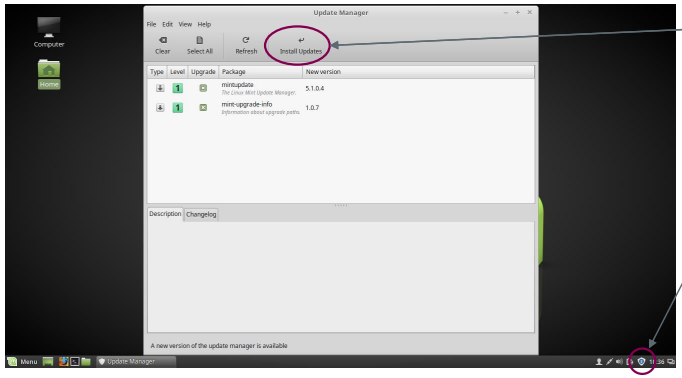
Better remove

Make password long (enough) and difficult to guess, yet easy to remember

Patch and secure your own computer!

- Do not run any unnecessary services
  - Email, WWW, SMB, telnet
- Do not keep test accounts with bad passwords
  - Systems are continuously scanned by intruders
- Install patches regularly:
  - Even a virtual machine is a potential target for intruders
  - Debian: apt-get update && apt-get upgrade
  - RHEL/Centos: yum update
  - GUI and scheduled updates

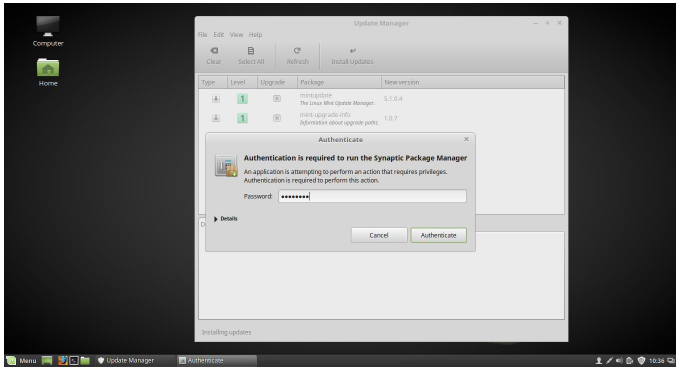
Patch and secure your own computer!



Click to load list of to-be-updated packages

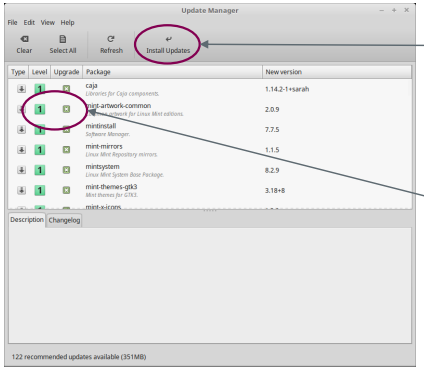
Indicates that updates are waiting

Patch and secure your own computer!



You need your password

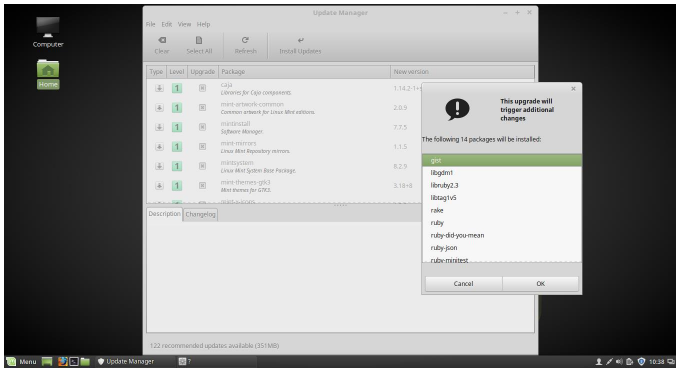
Patch and secure your own computer!



Click to launch update of tagged packages

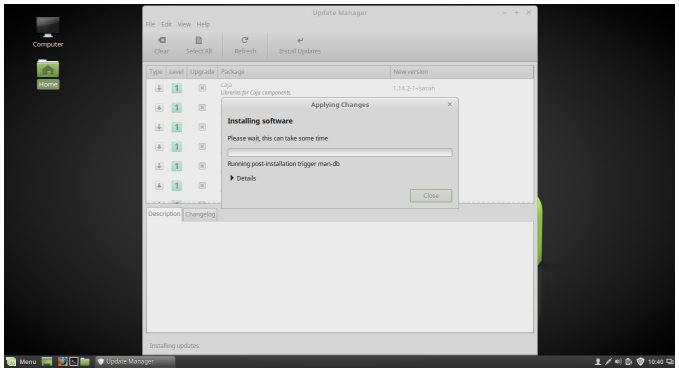
Number indicates importance of update (1 = lowest);
Untag to skip upgrade

Patch and secure your own computer!



Warns, if additional packages due to new dependencies have to be uploaded

Patch and secure your own computer!

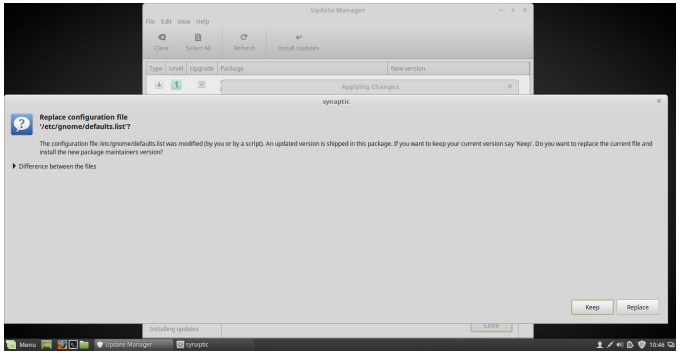


Shows progress (first download, then installation)

If new kernel-modules have to be compiled, this can last a little bit longer

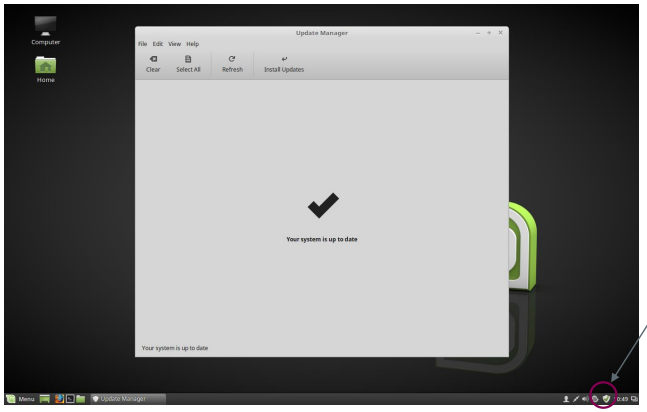
Better not to interrupt now, although error handling is quite good

Patch and secure your own computer!



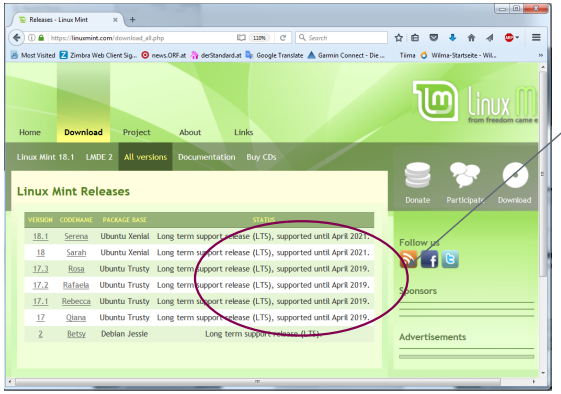
Sometimes you will be asked to overwrite system files (here for default applications)

Patch and secure your own computer!



Indicates that updates are in order

How long are Linux distributions supported?

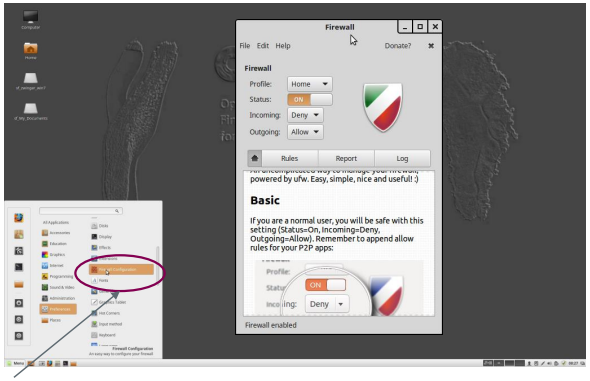


LTS = Long Term Support
After the dates no patches will be distributed

Enable a Firewall

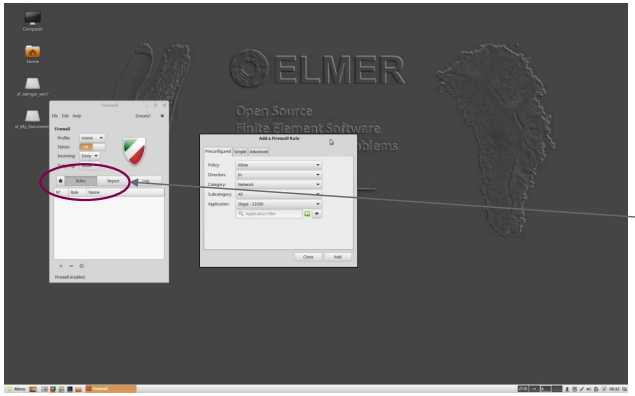
- What is a firewall?
  - A program (process) that handles network traffic with respect to security (filtering, blocking)
- Enable local firewall
  - Iptables, yum (Redhat)
    - ufw enable
    - ufw allow ssh, ufw default deny incoming
- Most Linux systems have a graphical user interface for setting up their built-in firewall

Enable a Firewall



GUI access to Firewall settings

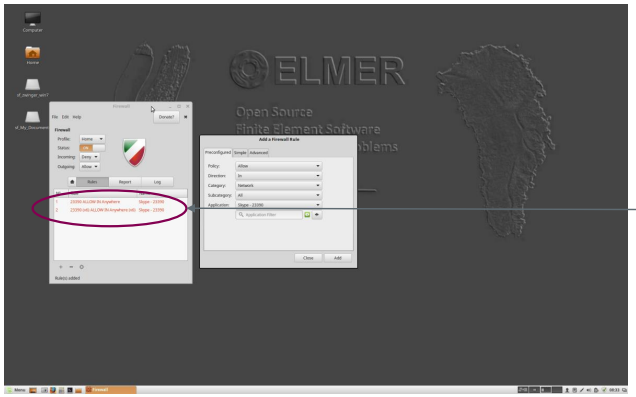
Enable a Firewall



By default (almost) all incoming ports are closed

Changing rules for incoming traffic

Enable a Firewall



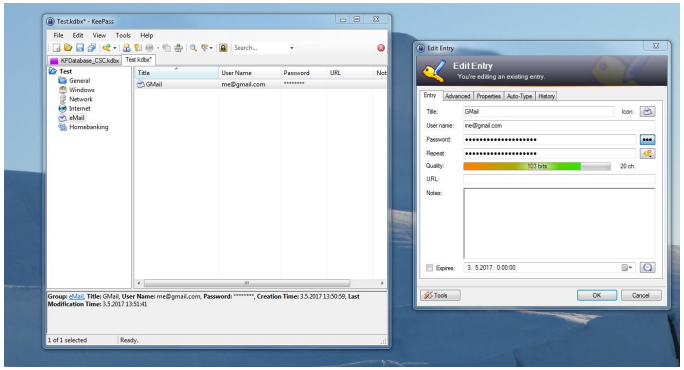
By default (almost) all incoming ports are closed

Reports the ports that are open

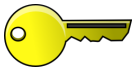
About Passwords

- Do not use the same password for different platforms/services
  - Once hacked, you loose it all
- Keyword Management: Keepass (or cross-platform KeepassX) is a handy tool for managing passwords
- You have to remember a single master password (or even better create a key)
- You can create extremely complicated passwords for services

About Passwords

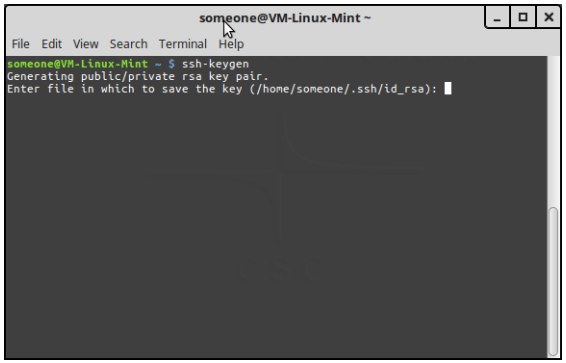


SSH keys



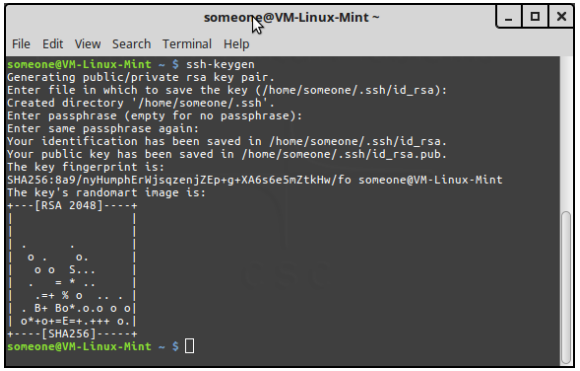
- Keys are a nice way to circumvent the hassle of remembering/typing long and complicated passphrases
- You win twice:
  1. Higher convenience (no typing)
  2. Increased security
- There are key-gen programs
- Here, we demonstrate how it is done from the command line

SSH keys



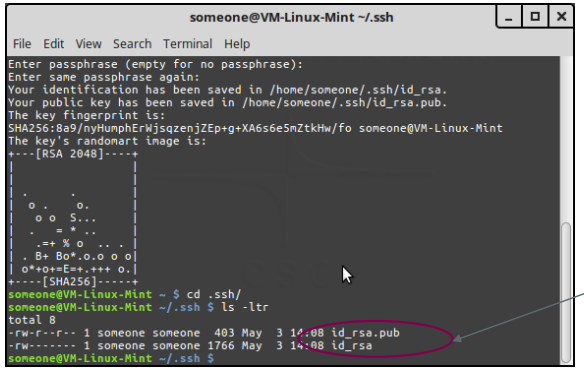
- `ssh-keygen` command
- Usually OK to go with default destination, except you want to be sure not to overwrite something (but you should be asked)

SSH keys



- You have to type a key-passphrase
- Generally, you would have this one distinguished from your system passphrase
- This passphrase will be requested by the system the first time you use ssh-key to connect

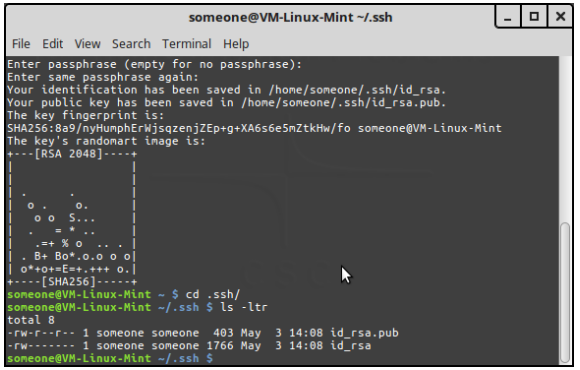
SSH keys



Two files

- `Id_rsa` is your private key, which under no circumstance you should reveal to outside world
- `Id_rsa.pub` is your public key

SSH keys



In order to be able to use the key-pair for connecting to another system, you have to place the public key in the file `~/.ssh/authorized_keys`

on the remote computer (e.g., on taito)

The login then should work without any password inquiry

Encrypt your data

- Use native encryption on your workstation
  - Improves basic protection
  - Usually you can choose upon installation
- Encrypt confidential email with PGP/GnuPG
  - Can be a little bit difficult to implement for non-technical people
  - No centralized key-management
  - Plug-ins for email clients
- Encrypting cloud content
  - Some solutions available