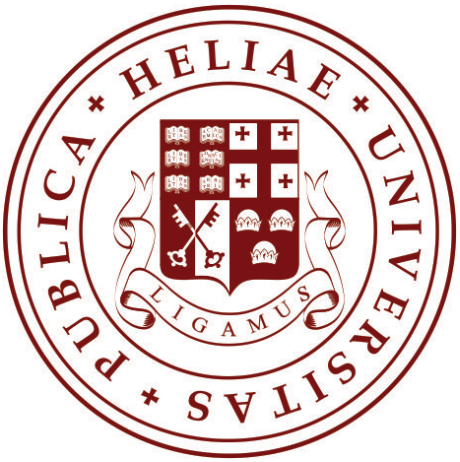

IoT with the Raspberry Pi under the LINUX OS, course script

Prof.-Dr. Ulrich Hauser-Ehninger

Ilia University Georgia, Tbilisi

2019

Abstract

The course *IoT with the Raspberry Pi under the LINUX OS* looks at the IoT in general and then focuses on the usage of the Raspberry Pi[1]. As the dominant OS (**O**perating **S**ystem) used to drive the Pi is LINUX[2], being productive with the Pi is almost impossible without fundamental knowledge of this OS. As communication is mostly done over the WWW (**W**orld **W**ide **W**eb), networking aspects and WWW-servers and -clients will be covered as far as necessary for the undestanding of the course content. Key points are:

- IoT (**I**nternet **o**f **T**hings)
- WoT (**W**eb **o**f **T**hings)
- WWW
- LINUX
- Raspberry Pi
- Sensor networks
- IoT communication
- M2M (**M**achine to (**2**) **M**achine) communication
- REST (**RE**presentational **S**tate **T**ransfer) interfaces
- network protocols
- JS (**J**ava **S**cript)
- nodeJS[3]

[1] https://www.raspberrypi.org/
[2] https://www.linuxfoundation.org/projects/linux/
[3] https://nodejs.org/en/

Contents

List of Figures v

Acronyms vi

1 Introduction to the IoT 3
- 1.1 What is the IoT and where did it come from . . . 3
- 1.2 The WoT and its relation to the IoT . . . 4
- 1.3 Applications of the IoT . . . 5
- 1.4 Explore real IoT devices . . . 6
  - 1.4.1 The human interface using a browser . . . 7
  - 1.4.2 The machine interface . . . 7
  - 1.4.3 Real time data-Event driven behaviour . . . 7
- 1.5 Networks of things . . . 8
  - 1.5.1 Topologies . . . 8
  - 1.5.2 Layers . . . 11
  - 1.5.3 Evaluation and decision . . . 11
  - 1.5.4 PANs . . . 11
  - 1.5.5 WANs . . . 11
  - 1.5.6 Examples . . . 11
  - 1.5.7 Protocols . . . 13
- 1.6 WoT architecture . . . 14
  - 1.6.1 Access . . . 14
  - 1.6.2 Find . . . 15
  - 1.6.3 Share . . . 15
  - 1.6.4 Compose . . . 16

2 Introduction to LINUX 17
- 2.1 Overview . . . 17
- 2.2 First steps . . . 18
  - 2.2.1 First Commands . . . 19
  - 2.2.2 A glimpse of the File system . . . 20
- 2.3 Basics . . . 22
  - 2.3.1 Introduction . . . 22
  - 2.3.2 History . . . 22
  - 2.3.3 Structural overview . . . 23

2.3.4 Kernel . . . 27
2.4 The file system . . . 29
2.4.1 Directories and the directory tree . . . 29
2.4.2 Links . . . 29
2.4.3 Device files . . . 29
2.4.4 Remote file systems . . . 29
2.4.5 FHS (**F**ilesystem **H**ierarchy **S**tandard) . . . 30
2.4.6 Permissions . . . 31
2.5 The shell . . . 32
2.5.1 Terminal . . . 32
2.5.2 Command line . . . 33
2.5.3 The prompt . . . 34
2.5.4 Configuration . . . 34
2.5.5 Important commands . . . 36
2.5.6 Pipes . . . 40
2.5.7 Patterns . . . 41
2.5.8 Quoting und escape-Sequenzen . . . 41
2.5.9 Stream redirection . . . 42
2.5.10 Proces control . . . 43

3 Introduction to JS **45**
3.1 Commonalities with C . . . 45
3.2 Variables and objects . . . 45
3.3 Using JSON to build objects . . . 45
3.4 Scope . . . 45
3.5 Events and listeners . . . 45
3.6 Modules . . . 45
3.6.1 Exports . . . 45
3.7 NodeJS . . . 45
3.8 JS in the Browser . . . 45

4 Introduction to the Raspberry Pi **46**
4.1 Overview . . . 47
4.2 Setting up . . . 47
4.2.1 Images . . . 47
4.2.2 Preparation . . . 47
4.2.3 Starting . . . 47
4.2.4 Improvements . . . 47

5 Building a IoT device **48**
5.1 Pi, accessing peripherals . . . 48
5.2 Access over the WWW . . . 48

6 Building a WoT device **49**
6.1 Building a WoT WWW server using NodeJS 49
6.1.1 A WWW server for M2M communication 49
6.1.2 A WWW server for the browser 49
6.2 Sending and receiving over MQTT (**M**essage **Q**ueue **T**elemetry **T**ransport) . 49
6.2.1 Synchronising data using push messages 49

References **50**

Index **51**

List of Figures

1.1 standardisation 4
1.2 Network topologies 8
1.3 The internet, partial map 9

2.1 History of UNIX 24
2.2 LINUX distributions 25
2.3 Grundaufbau von UNIX 26
2.4 Struktur des Kernel (Quelle: wikipedia) 28
2.5 Eine Shell 32
2.6 Man 37
2.7 Man, verschiedene Versionen 38

Acronyms

API **A**pplication **P**rogrammer **I**nterface. 14, 15, 26, 27

bash **b**ourne-**b**gain-**sh**ell. 33, 42, 43

BT **B**lue**t**ooth. 11, 12

CLI **C**ommand **L**ine **I**nterface. 18

CoAP **Co**nstraint **A**pplication **P**rotocol. 14

CPU **C**entral **P**rocessing **U**nit. 22, 23, 26

DoS **D**istributed **D**enial **o**f **S**ervice. 15

FHS **F**ilesystem **H**ierarchy **S**tandard. iii, 30

FTP **F**ile **T**ransfer **P**rotocol. 30

Gimp **G**nu **i**mage **m**anipulation **p**rogram. 1

GNU **G**NU's **N**ot **U**nix. 42

HAL **H**arware **A**bstraction **L**ayer. 22, 23

HCI **H**uman **C**omputer **I**nterface. 8

HTML **H**yper**t**ext **M**arkup **L**anguage. 7, 15

HTTP **H**yper**t**ext **T**ransfer **P**rotocol. 7, 14, 15, 27

HW **H**ard**w**are. 13, 16, 20, 22, 23, 29, 32

IoT **I**nternet **o**f **T**hings. i–iii, 3–8, 10, 12–15, 48

IR **I**nfra**r**ed. 6

JS **J**ava **S**cript. i, iii, 45

JSON **J**ava**S**cript **O**bject **N**otation. 7, 15

JSON-LD JavaScript Object Notation-Linked Data. 15

LoRa Long Range. 12, 13

LOS Line Of Sight. 12

LTE Long Term Evolution. 13

M2M Machine to (2) Machine. i, iv, 10, 15, 49

MIDI Musical Instrument Digital Interface. 9

MMU Memory Management Unit. 23

MQTT Message Queue Telemetry Transport. iv, 13, 14, 49

NFS Network File System. 30

OS Operating System. i, 17, 18, 22

PAN Personal Area Network. 11

RAM Random Access Memory. 27, 28

REST REpresentational State Transfer. i, 14, 15

SCI System Call Interface. 26

SFTP Secure File Transfer Protocol. 30

sh bourne-shell. 33

SMB Server Message Block. 30

ssh secure shell. 27, 47

SW Software. 1, 15, 27

UI User Interface. 18, 21

URI Unique Resource Identifier. 14

URL Unique Resource Locator. 14

VM Virtual Machine. 18

WAN Wide Area Network. 11, 12

WoT Web of Things. i, ii, iv, 3–5, 14–16, 49

WWW World Wide Web. i, iii, iv, 1, 5, 6, 14, 29, 30, 36, 41, 48, 49

Preface

The script was developed using free SW (**S**oft**w**are) as far as possible. The reasons are:

- The Author is about the opinion that open SW is important in academic education, to avoid restricting students to commercial products during this period of time.
- Students should have the chance to use SW by themselves. This is most easy if
  - SW does not cost any money.
  - the lecturer/teacher/coach uses the same SW as the student.

The script was written using LaTeX[4]. The development environment is TeXStudio[5]. Images are taken from sources under the CC license model[6]. Image processing was done with Gimp (**G**nu **i**mage **m**anipulation **p**rogram)[7]. The author tried to reference sources wherever known. This applies as well for sources under CC. If copyright infringements are found, please do not hesitate to contact the author at ulrich.hauser@htwchur.ch to correct the mistake.

Most of the IoT knowledge was taken from the book *building the web of things*[8][1]. Information about LINUX[9] is drawn from a more elaborated lecture, given by the author at FHGR[10]. The Raspberry Pi itself[11] is widely documented in the WWW.

[4] https://www.latex-project.org/
[5] https://www.texstudio.org/
[6] http://www.creativecommons.ch/
[7] https://www.gimp.org/
[8] https://webofthings.org/book/
[9] https://www.linuxfoundation.org/projects/linux/
[10] https://www.fhgr.ch/
[11] https://www.raspberrypi.org/

The script is not exhaustive. Therefore, Students are required to take notes in the lectures themselves to complete the information.

Chapter 1

Introduction to the IoT

1.1 What is the IoT and where did it come from

There is no crisp definition of the IoT. One possible way to put it is[1]:

Definition 1 *The Internet of Things is a system of physical objects that can be discovered, monitored, controlled, or interacted with by electronic devices that communicate over various networking interfaces and eventually can be connected to the wider internet.*

The *physical object* can be everything between tiniest of sensors or actuators like RFID tags to big complex control systms like cars or even a standard PC. One central aspect of IoT is that the things themselves communicate with each other and not necessarily with a human. This communication is not seamless. There are plenty of protocols[1] communicating over a plethora of physical networks and network technologies, especially in the wireless area. Up to now, there is no end to this development and new protocols crop up everywhere (see fig. 1.1[2]). One approach to address the issue of heterogeinity is the WoT, trying to connect things using established and well-known technologies.

Mark Weiser, engineer at XEROX Parc, said[1]:

> The most profound technologies are those that disappear.

[1] https://en.wikipedia.org/wiki/List_of_automation_protocols
[2] https://xkcd.com/927/

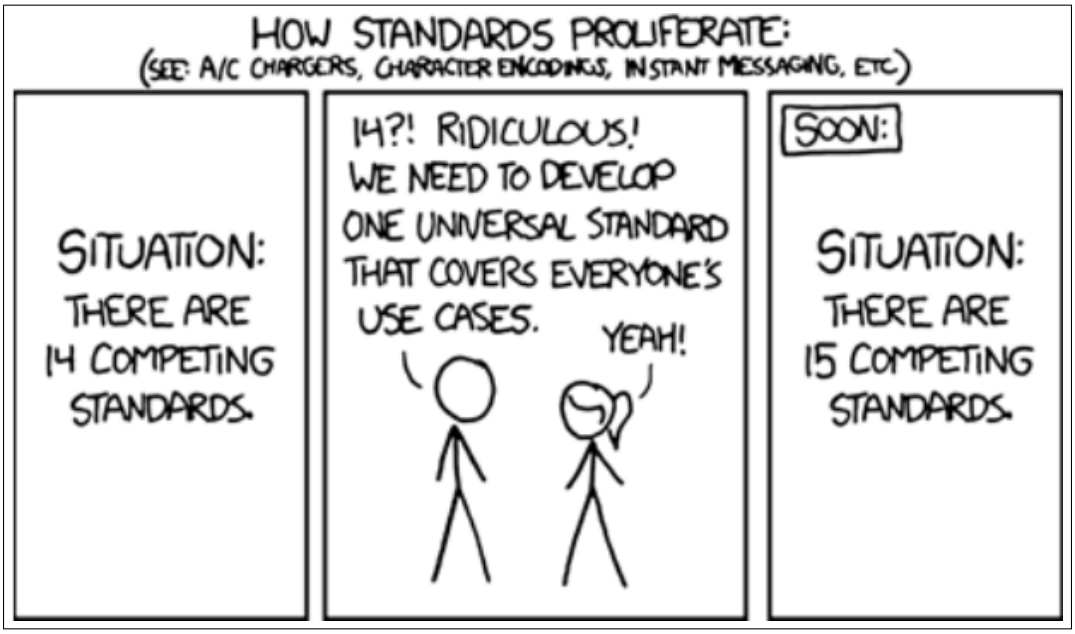

Figure 1.1: standardisation

This is, what the IoT is all about.

The term itself came up in 1999. Things developed away from the public until 2013 the term became very prominent in the WEB 2.0 movement. Forecasts back in 2014 estimated the number of IoT devices to be between 25 and 50 billion ba 2020. Today it looks like the number will be closer to the upper limit or even higher. Connected to this development is a growing demand in energy and other resources not to be underestimated.

1.2 The WoT and its relation to the IoT

As mentioned above, one of the challenges in the IoT-area is the number of technologies, protocols and 'standards' evolving around the subject. The companies developing IoT-devices developed new protocols or wireless technologies as the need arose. Huge players tried to establish their protocols as new standards. But keeping control over a proprietary protocol never worked well in the internet domain. The standards are more or less all open standards, developed largely not by industry but by academics. Why not use established standards

in the IoT-world as well? This is essentially what the WoT tries to accomplish. The main challenge is that IoTdevices have quite different requirements compared to standard WWW-devices. The main important ones are related to energy issues and issues of data rate and data amount. A sensor node at a remote location needs to consume as little energy as possible, transfer data at the lowest possible cost concerning energy and bandwidth aspects to be able to live on battery as long as possible or even draw all energy necessary from the sources available locally. On the other hand, sensors, but even more so actuators, need to be able to receive data, perhaps with quite low response times. This leads to receivers with as little energy consumption as possible as well as being receptive for as much a percentage of the time as posible.

The solution is to use well-known protocols where possible and combine them with technologies fullfilling the restricive requirements of the IoT. This is –in essence– abstracting the demands of the IoT behind the upper layers of the communication model.

1.3 Applications of the IoT

It is rather irrational to have a section about this topic as each day new applications for the IoT arise. But still, a small list of the most prominet areas shall be given.

- Monitoring of parameters in distributed contexts like
  - biology
  - geology
  - energy
  - manufacturing
  - meteorology
  - community (traffic, people commute etc.)
  - ...

- control in distributed contexts like
  - home automation
  - manufactoring
  - marketing
  - logistics and delivery
  - access technologies (entrance systems etc.)
  - ...

PLenty of information can be fount in the WWW[3,4,5,6,7,8,9,10][1].

1.4 Explore real IoT devices

Some IoT devices are directly visible in the WWW[11]. This is a website set up by the authors of the book this part of the lecture is based on. The devices accessible are (all sitting on a raspberry pi):

- an IR (Infrared) sensor
- a temperature sensor
- a camera
- a LCD display

[3] http://www.wired.com/2013/02/budweiser-red-light/
[4] http://www.altomagazine.com/newsdetails/travel/hotels/dom-prignon-at-the-press-of-a-button-431
[5] http://theinspirationroom.com/daily/2012/evian-smart-drop/
[6] http://postscapes.com/internet-of-things-investment
[7] http://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes
[8] http://d3s.disi.unitn.it/projects/torreaquila
[9] http://www.cs.berkeley.edu/~culler/papers/wsna02.pdf
[10] http://www.mdpi.com/1424-8220/9/6/4728/htm
[11] http://devices.webofthings.io/

1.4.1 The human interface using a browser

The devices can be accessed via a web browser. The values can be explored, the LCD can be set. The display can be photographed with the camera. If the respective ports are open, the image can be downloaded and will be displayed.

1.4.2 The machine interface

A browser will normally send a HTTP (**H**yper**t**ext **T**ransfer **P**rotocol) header information to accept *text/html*. This indicates to the server that it is supposed to return something in HTML (**H**yper**t**ext **M**arkup **L**anguage)-code. If this information is replaced by *application /json*, the server (if programmed accordingly) should respond with information, coded by JSON (**J**ava**S**cript **O**bject **N**otation). To create the correct resquest, extra toolingf is needed. There are browser plugins or applications like postman[12,13] which allow to generate a wide variety of requests and then show the responses.

1.4.3 Real time data-Event driven behaviour

Exploring a device is done by sending requests and evaluating the responses. In the IoT it would often be desirable to have an inversion of the control flow. This means that the device would start the communication by itself, because something was registered by some sensors or something reached a certain state. Normal HTTP did not support this, but the standard HTML5 introduced *websockets*[14] and uses HTTP upgrading[15] to allow communication to be initiated by the server itself.

Using this technology, real-time event driving is possible. It allows to very quickly after a certain condition is met, the associated server issue information which –in the client– causes an event to be triggered. The client can react to this event directly.

[12] https://www.getpostman.com/
[13] https://insomnia.rest/
[14] https://en.wikipedia.org/wiki/WebSocket
[15] https://en.wikipedia.org/wiki/HTTP/1.1_Upgrade_header

Without communication inversion, the client needs to use *polling*. Polling is a concept where a client requests information from a server in regular intervals to get to know about the state of the server. This, obviously, need much more data to be transferred and is less responsive compared to event-driven behaviour.

The thing explored above does not use this technology by the time of the writing of this script.

1.5 Networks of things

The IoT is a network of things, communicating with each other and communicating with humans over HCI (**H**uman **C**omputer **I**nterface)'s. If one wants to understand the IoT, a basic knowledge of networks and their properties is necessary.

1.5.1 Topologies

A *topology* of connections is the way in which the partners are connected to each other. An overview is given in Figure 1.2[16]. In fig. 1.3, a part of the internet is visualised as a map[17].

[16] https://simple.wikipedia.org/wiki/Network_topology

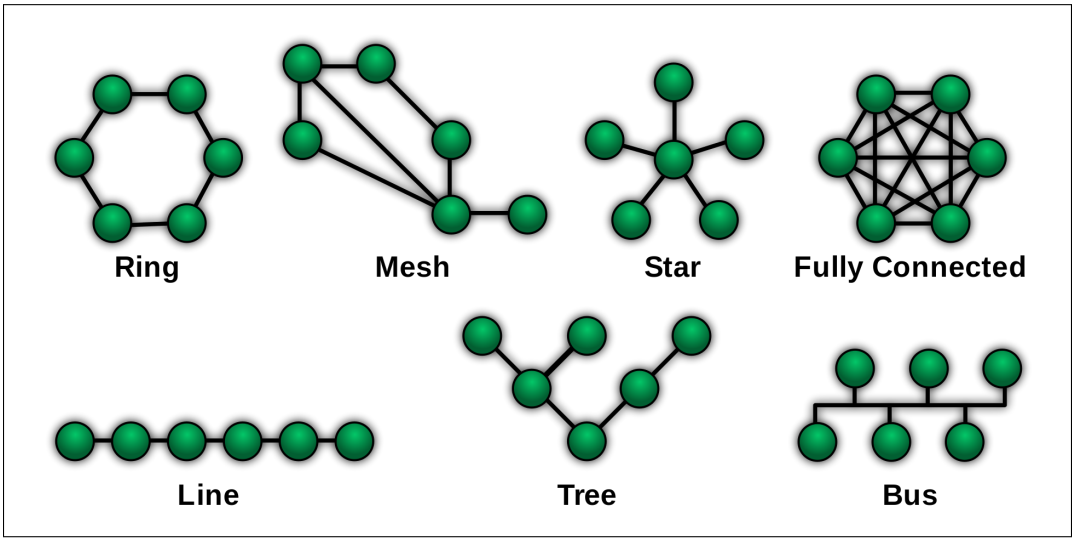

Figure 1.2: Network topologies

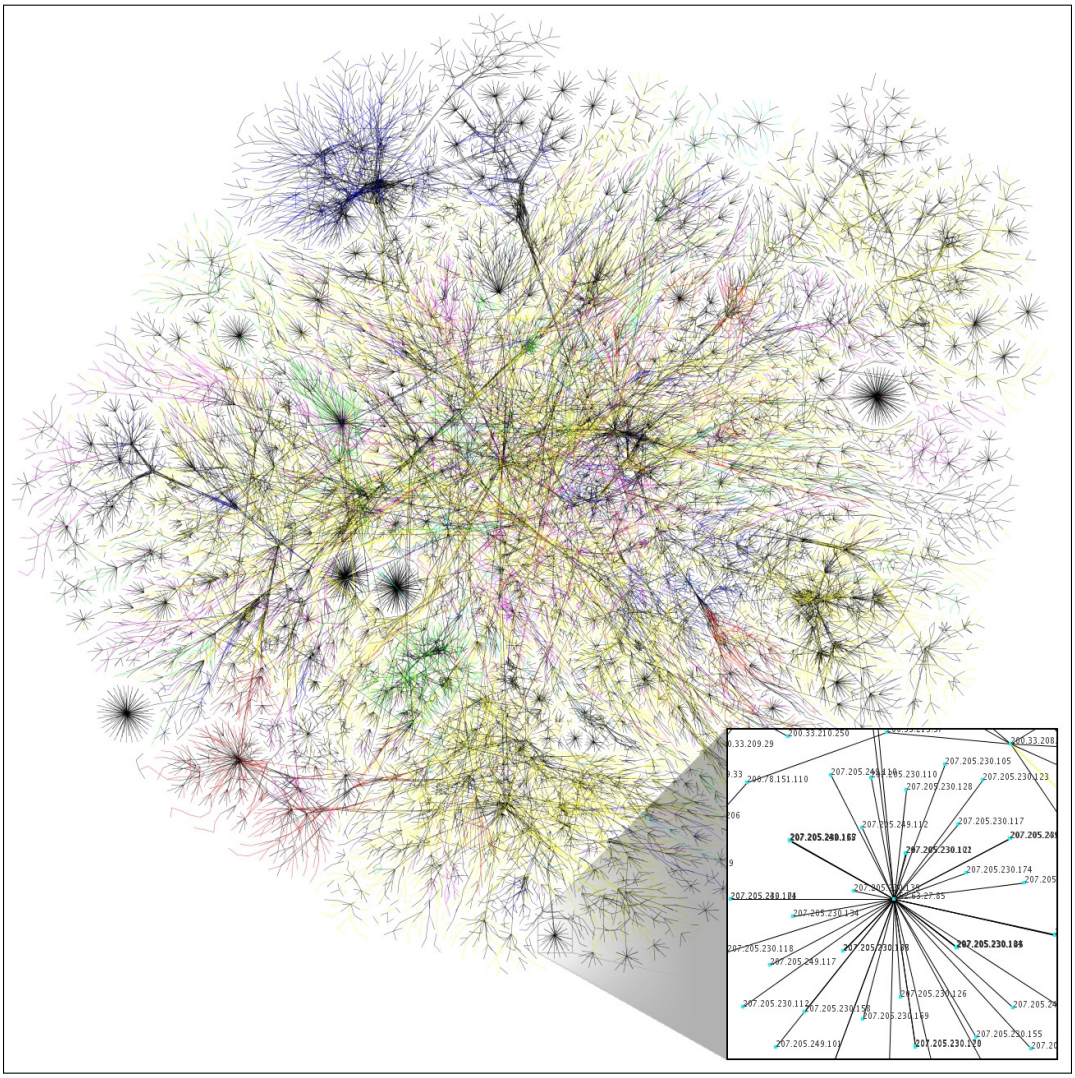

Figure 1.3: The internet, partial map

1.5.1.1 Ring

For electrical reasons, the ring used to be one of the most prominent network topologies for a long time[18]. Networking, including the internet, started out mostly as token ring networks.

1.5.1.2 Linear

Linear networks are quite basic. Often, they are unidirectional as well. Often, tis is called *daisy chaining* as well. MIDI (**M**usical **I**nstrument **D**igital **I**nterface), used for music since

[17] https://en.wikipedia.org/wiki/Network_topology#/media/File:Internet_map_1024.jpg
[18] https://en.wikipedia.org/wiki/Ring_network

the 1980s is such a network.

1.5.1.3 Star

The most common network topology these days is the star of starts configuration. Ethernet clients are connected to a single gateway which itself connects to a switch or another gateway and so on, building *stars*[19] of devices, consisting of stars of devices, therefore named *star of stars*.

It is strictly hierarchical and allows hierarchical addressing over several levels of address spaces. If this network is strictly star or star of stars, only one way of communication between two partners is possible. As the failure of a single device could jeopardise a connection, redundency is normally built in the system. Gateway between two systems are implemented twice or more. Then –if one gateway or switch fails or is overloaded– another device can take over the necessary data transfers. But still the logic topology would be a star of stars.

1.5.1.4 Tree

Logically, a star of star network can be looked at as being a tree. The internet is one single gigantic tree.

1.5.1.5 Bus

Measurement systems are often arranged this way. Although today often connected to ethernet, *IEEE-488*[20] (often called HP-IB) used to be such a bus. As the measurement devices often talked to each other (M2M), it can be looked at as predecessors of IoT devices.

1.5.1.6 Mesh

IoT devices often use radio links of short range to save energy. This leads to the problem that, in larger areas, nodes may not be able to connect to the central data link device. Then,

[19] https://en.wikipedia.org/wiki/Star_network
[20] https://en.wikipedia.org/wiki/IEEE-488

a mesh network is a big advantage as nodes can use other nodes as links or relays towards the gateway. This way, a huge network can operate on quite small radio ranges. ZigBee[21] is such a network, but BT (**B**lue**t**ooth) in its last revisions is able to build meshes as well.

1.5.2 Layers

Networks are layered. The lowest layer normally is the physical medium. The higher the layer the mor abstract its working. The highest layer is the application using the network. A reference for networks is the OSI model[22].

1.5.3 Evaluation and decision

1.5.4 PANs

PAN (**P**ersonal **A**rea **N**etwork)s are local networks bridging distances from only mm or cm up to some m or dozens of meters. They are hugely use in home automation, logistics. Most of them are used in urban areas where distances are small and data needs to be kept local if possible.

1.5.5 WANs

WAN (**W**ide **A**rea **N**etwork)s span wide areas up some dozens of km. They are used for sensor networks, often in rural areas to be able to monitor conditions across large distances.

1.5.6 Examples

1.5.6.1 Bluetooth

BT is a typical network protocol for short distances. In its beginnings it was intended to reduce the cable chaos around computers. Printers, mice and other devices were supposed to

[21] https://zigbee.org/
[22] https://en.wikipedia.org/wiki/OSI_model

communicate through this interface. Data rates where quite low these days. Although data rates through the newest standards are much higher, still is not a connection type intended for the big data transfer.

Since version 4.0, BT supports low energy modes. As well starting with version 4.0, mesh networking is supported. Both aspects increased the for IoT applications to use BT significantly.

1.5.6.2 ZigBee

ZigBee is used in the same area as BT but was intended for IoT from the start. It was designed using low energy and mesh networking from the start. Ther are three main frequency bands, 915MHz and 868MHz, depending on regulatory limitation of the area of use and 2.5GHz worldwide. The intended range is 10-100m although outside with LOS (**L**ine **O**f **S**ight), up to 1500m can be obtained. There are proprietary ZigBee long range versions for much higher distances as well[23].

1.5.6.3 LoRa

LoRa (**Lo**ng **Ra**nge)[24] WAN is a technology to bridge large distances (typically 10 to 15km). It allows devices to live on single batteries for a number of years as the power consumption is very low. On the other hand, data rates are low, typically 10's of bytes per second up to some kBaud. This is ideal for sensor networks, transmitting just small bits of data with a large time inbetween transmissions. There are a lot of applications like mintoring in the IoT area where this technology is ideal.

[23] https://www.digi.com/products/embedded-systems/rf-modules/sub-1-ghz-modules/xbee-pro-900hp

[24] https://en.wikipedia.org/wiki/LoRa

1.5.6.4 SigFox

Technically, although being quite different to LoRa, *SigFox* has the same target application, long distance, low bandwidth, low energy. The business model is very different to LoRa. SigFox is proprietary, LoRa is open. Being direct competition SigFox seems to loose ground in recent times.

1.5.6.5 Cellular technologies

These technologies need support by the commercial mobile communications supplier as they partly use the technology provided there. and LTE (**L**ong **T**erm **E**volution)-M are examples. Together with 5G mobile communications big changes in IoT data transfer are to be expected during the coming years.

1.5.7 Protocols

The HW (**H**ard**w**are) technology is not everything needed to supply energy efficient data communication. The protocols defining the data exchange have a huge influence as well. It is important for energy efficiency to allow the link to be quiet for a long time. With WiFi e.g. this is not the case.

1.5.7.1 MQTT

MQTT is a service based on the publish/subscribe pattern. A message delivery entity sends messages to a broker. 'Customers' can subscribe to listen to certain message types. Whenever a fitting message is issued to the broker it will inform the listening entities. MQTT knows different levels for the quality of service to give the guarantee (or not) that messages are delivered correctly.

1.5.7.2 COAP

CoAP (**Co**nstraint **A**pplication **P**rotocol) –in contrast to MQTT– is placed on top of UDP to get rid of some amount of data traffic due to packet acknowledgement and keeping open connections. It is a request/response protocol and is based on REST [1]. It therefore genuinely supports HTTP with REST.

The very small memory footprint in usage and the connectionless design help in its applicability in th IoT.

1.6 WoT architecture

We now leave the physical and logical structures of the networks and focus on the architecture of the WoT. There are four stages of using the connected things in the WoT:

- The *access* of devices through the technologies covered in section 1.5. The central aspect hier is how the API (**A**pplication **P**rogrammer **I**nterface)'s are designed to allow easy access of devices using well-known ways.
- Things need to provide means to lay open what they have to offer in a machine-readable way to make them *find*able.
- Things need to *share* data in a controlled, safe and secure way
- Tools can use the data found in the WoT to *compose* meaningful information from multiple sources of data.

1.6.1 Access

The WWW knows many techniques to access devices which are part of the internet. URL (**U**nique **R**esource **L**ocator)'s/URI (**U**nique **R**esource **I**dentifier)'s allow identification of devices and services on devices. *Websockets* are the base to use bidirectionally induced communication in web browsers. MQTT and CoAP help with lightweight protocols in

restricted environments. With HTTP, an application layer protocol to statelessly transfer data from one entity to another is at one's disposal. It helps with M2M communication as well through header attributes. REST API's support M2M communication representing states in a stateless surrounding. This is obtained by transmitting all information necessary for a process in each HTTP-message. HTML as the language of web-pages and JSON as a data format to transport structured data sets from small chunks of configuration data up to whole databases are languages for access by humand and by machines.

1.6.2 Find

To access devices the problems are mainly of syntactical nature. To find desired services, semantic information becomes more important. Marking parts of web pages (possibly using `schema.org`), linking to other data to get machine readable linked data, enhancing web pages with JSON-LD (**J**ava**S**cript **O**bject **N**otation-**L**inked **D**ata) to create the semantic web are all tools to help give data meaning. The same techniques can be used in the IoT to tell machines what given sensors or actors can deliver or do.

1.6.3 Share

The IoT would be meaningless without sharing data traveling through the internet. But for prober publushing of data, proper security means need to be considered. It is thought as one of the big threats, not only across the IoT devices, but across the internet and possibly even beyond it, that IoT devices might be (and have been) captured using malignant SW to do unintended things. Cameras have been captured in 2016[25],[26] to start DoS (**D**istributed **D**enial **o**f **S**ervice) attacks as they were so easily captured, available in large numbers and connected to the internet. The awareness about IoT security is still much too low, although it rises. More and more IoT devices come with security chips installed already or processors

[25] https://www.csoonline.com
[26] https://krebsonsecurity.com/2016/10

with security HW layers builtin[27].

1.6.4 Compose

Once meaningful data has been obtained, it can be used in conjunction to other data to crate new and meaningful knowledge. This is the huge market gaining momentum these years. It deals with mashups[28] (a word stemming from music[29]), industry 4.0 production, monitoring[30] and much more like smart cities etc., affecting all societies at all levels.

[27] e.g. https://www.arm.com/solutions/security
[28] https://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)
[29] https://en.wikipedia.org/wiki/Mashup_(music)
[30] https://en.wikipedia.org/wiki/Industrial_internet_of_things

Chapter 2

Introduction to LINUX

2.1 Overview

LINUX –in general– is an OS with multiuser, multitasking and multiprocessing support. It even supports computer clusters like Beowulf[1] It started in 1991 as a project by Linus Thorvalds as a free alternativ to MINIX[2], which itself is an alternative to UNIX[3], all systems striving to be POSIX[4]-compliant OS's. The term LINUX is an acronym. Two following main exist:

- Linus Thorvald's UNIX
- Linus Thorvald's MINIX
- LINUX is not UNIX

Modern versions consist of more than 20 million lines of code and is published under the GNU General Public License[5]. Further details will be covered in section 2.3.2.

The first contact to LINUX in this lecture is supposed to be a practical one without having too much of a theoretical background obstructing one from having the first frustrations.

[1] https://en.wikipedia.org/wiki/Beowulf_cluster
[2] https://en.wikipedia.org/wiki/MINIX
[3] https://en.wikipedia.org/wiki/Unix
[4] https://en.wikipedia.org/wiki/POSIX
[5] https://en.wikipedia.org/wiki/GNU_General_Public_License

The author had his first contact with LINUX 1997 through a CD grabbed from a jumble sale stand in a book shop. Just two weeks later the system was up and running with a text console on a 80486 PC. Just to illustrate how far the system comfort progressed since then: A normal user with not more OS knowledge as a standard windows user now can set up a dual boot system with LINUX (.e.g. UBUNTU[6]) running in parallel to windows, with a comfortable graphical UI (**U**ser **I**nterface) in about an hours time.

To be able to use the same system with each student, VM (**V**irtual **M**achine)'s[7,8] are awsome tools to enable experiences without the danger of damaging a whole system and again and again spending hours of installing again and again. VirtualBox by Oracle will be the tool of choice in this script although every other VM able to host a LINUX system will be as sufficient as running on a native LINUX system will be.

2.2 First steps

LINUX-OS is started by running it from a VM or directly.

After the booting process, the user is asked to log in with a name and a password.

If the LINUX is non-graphical, the user now will be greeted by a console, the primary tool to interact with the system in this lecture. On systems with graphical UI, a terminal needs to bi started through the standard menu, which looks different in different distributions. The console cannot do more that accepting commands and printing answers. This limitation to entering text to make the system do certain things might be felt as a restriction at first, especially for users of windows or mac, used to just clicking around. After one gets used to it and one knows the most important commands, their interaction and interconnection, many of the everyday tasks can be accomplished much faster through a CLI (**C**ommand **L**ine **I**nterface).

The following command need to be issued exactly like given to get the same results. So

[6] https://ubuntu.com/
[7] https://www.virtualbox.org/
[8] https://wiki.qemu.org

playing around now is not a good idea.

2.2.1 First Commands

cd

cd / (keyed in and finished by hitting *Enter*) leads to the *root directory*. What is that and how do I know? LINUX does not know drive names. LINUX knows just a single gigantic directory tree starting with `/`. The *working directory* is the directory the actual commands are executed in and on. It normally is shown in the console *prompt*, the information preceding the blinking cursor. Before entering the command given above, the prompt showed `~` at the position of the working directory. This indicates the home directory, the place where each user will store all personal files.

ls

Now issuing cd home again changes into another directory.

ls shows the content of the working directory in a short and readable way. As we are in the directory `/home`, we should be given the names of the user accounts on this machine. Apart from these accounts there is a priviledged account, called `root`-account to administer the system.

pwd

cd takes us back to the home directory. But which one is that? pwd shows us, that it is a subdirectory of `/home`.

touch

touch somefile.txt will generate a file named `somefile.txt`. We can see this, issuing one more `ls`.

find

find /home > files.txt searches for all files in `/home` and pushes the result into a file named `files.txt`. cat files.txt will show the content of this file.

cat

top

top results in a well populated screen This command gives quite a lot of information about what is going on in the system. The number of tasks running, system utilisation, information about the most time consuming tasks are all shown and updated regularly. Even without performing high pressure tasks, quite a lot is going on here. q leaves this screen again.

2.2.2 A glimpse of the File system

As we have now seen the very first commands, it is time to look at the file system. Its role is quite different to the role in windows, where the file system is ther mostly to hold files. As mentioned above, LINUX knows a single directory tree. LINUX –from the user's point of view– doesn't know printers, network interfaces, graphics cards, USB nor any other HW. LINUX just knows files, even for the computer HW and the processes running at the moment. Generally speaking, a file in LINUX is *everything you can write to or read from.* proc

cd /proc, followed by `ls`, e.g. leads us to a very interesting directory. Many subdirectories with numbers as names are present. Each of this directory represents one of the running processes in the system. This is called a virtual directory (as there can be virtual files as well). These directories cannot be found on disk. They are there, created by the system in memory, when the associated information is asked for. Other files in the directory `/proc` are related to other system information. cat meminfo for example shows how the memory of the machine is used.

cd takes us 'home' again.

The following 'monster' command searches all files in the system starting with the letter `p` and looks for the content `notroot` inside the files. The result is stored in a file called `notroot.txt`.

```
find / -iname p* | xargs grep notroot >notroot.txt
```

This takes a while as the directory tree has hundreds of thousands of files. But now, the console is blocked by the running process. No prompt is visible any more.

Ctrl-C stops the command execution. this is called *break*.

The next trial:

```
find / -iname p* | xargs grep notroot >notroot.txt &
```

Now the promt is there again. But why does something write into the console, although we can write into the console ourselves? The ampersand (`&`) tells the command to run in the 'background'. It will run in parallel to everything what the console itself does. Therefore the

console is accessible to the user to issue new commands, but the command in the background will write to the console as soon as there is something to write. As the command issued above is supposed to write to a file anyway, it is not necessary to block the console. The problem is, that error message still are directed to the console not to clutter the normal output of the command.

ps shows the commands running in the console at the moment. This shows, that all the commands `find`, `xargs` and `grep` are active at the same time. `ps` as actually running command is listed as well..

We conclude that there is no problem with having multiple commands running i one console simultaneously.

A last example can show how powerful the concept behind the LINUX directory tree is: Having a graphical UI, Ctrl-AltF2 leads to a text console, without a graphical UI it is done by Alt-F2. After logging in, one can get back to the old console using Alt-F7 on most graphical UI's, Alt-F1 on text console systems.

ls -l >/dev/tty2 now seems to do nothing, but going back to the other console as before shows, that th output was channeled to the other console although `/dev/tty2` is supposed to be a file. There are two new concept here:

- There are several parallel consoles, even if the system is non-graphical. Not tested up to here but still possible: On the different consoles, different users can be logged in.
- In essence `/dev/tty2` is a file representing the console. Writing to the file is writing to the console, reading from the file is reading information that was typed on that console.

These first steps should have given a little teaser of LINUX. Some more basic understanding is stimulated in the next section.

2.3 Basics

2.3.1 Introduction

Users of computer system would not like if they had to interact with a system differently when different HW would be mounted. Buying a new arddisk and because of that having to access files differently would be inacceptible. To solve this problem (among others) *operating systens* exist.

- An OS does HW abstraction to obstruct the details from the user. this is done by the HAL (**H**arware **A**bstraction **L**ayer).
- An OS manages resources like memory, hard disks, CPU (**C**entral **P**rocessing **U**nit) time etc. for the users. If a user program needs such a resource it asks for it in programmatical way. The OS the deals with the request. Generally, a means to deal with a request is called a *service*.

Therefore an OS can be described as a collection of services for the user/programmer to keep things HW independent. The OS 'sits' inbetween the HW and the user.

2.3.2 History

1. 1969: Start of the UNIX-development from conclusions from the MULTICs project by Ken Thompsen.
   (a) no closed Shop (punched card, given to an operator, no interaction between user and computer)
   (b) hierarchical file system
   (c) Multiuser
   (d) Multiprocessing
   (e) portabel

Teaming up with Dennis Ritchie, Rudd Canady, Brian Kernighan

2. 1970: Name UNIX

3. 1972: New implemention in C (up to there too much assembly language)

4. later: Diversion into multiple branches (see fig. 2.1[9])

5. 1991: LINUX 0.99

6. 1994: LINUX 1.0

Today, several distributions are present, deveoped with different aims or due to disagreement over development issues. It is a very fractured landscape as can be seen in fig. 2.2[10]. The full image is of such detail that the names cannot be distinguished any more.

2.3.3 Structural overview

2.3.3.1 composition

In fig. 2.3, the basic laout of UNIX is shown. The device drivers –strongly dependent on the HW– are closely bound to the kernel and manage the data exchange between kernel and HW. The therefore form the HAL.

Most of the kernel is written in C. Some of the code very close to the HW needs to be written in assembly language and therefore will be dapted to fit the needs of the respective HW. This is mostly limited to the CPU and the peripherls close to it, but in modern computer systems there are a lot of components with processing power which needs to be programmed. Apart from CPU's these are for example MMU (**M**emory **M**anagement **U**nit)'s, graphic controllers, hard disk controllers, bus controllers for peripherals, bridges and sound controllers,

[9] https://de.wikipedia.org/wiki/Unix#/media/Datei:Unix_history-simple.svg
[10] http://futurist.se

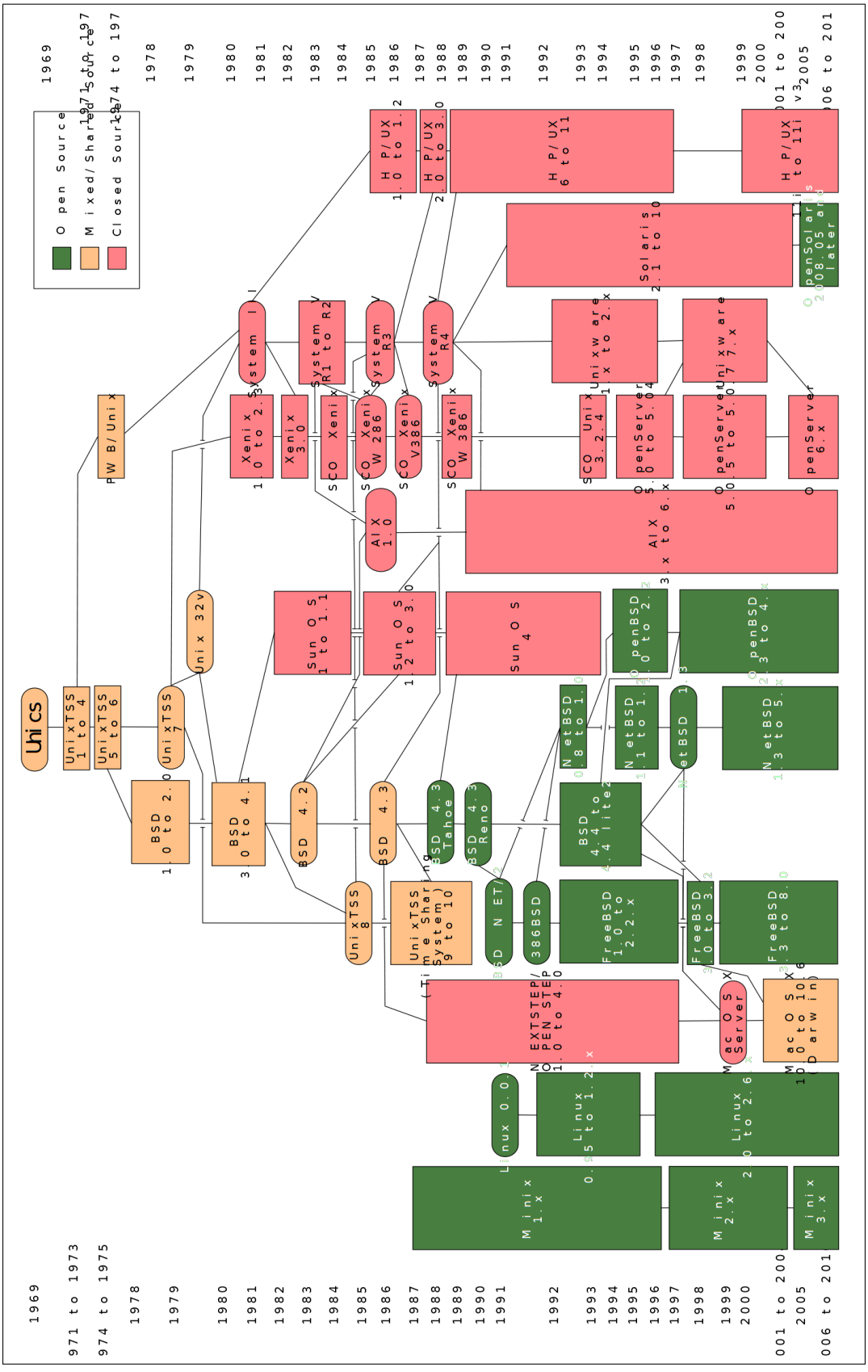

Figure 2.1: History of UNIX

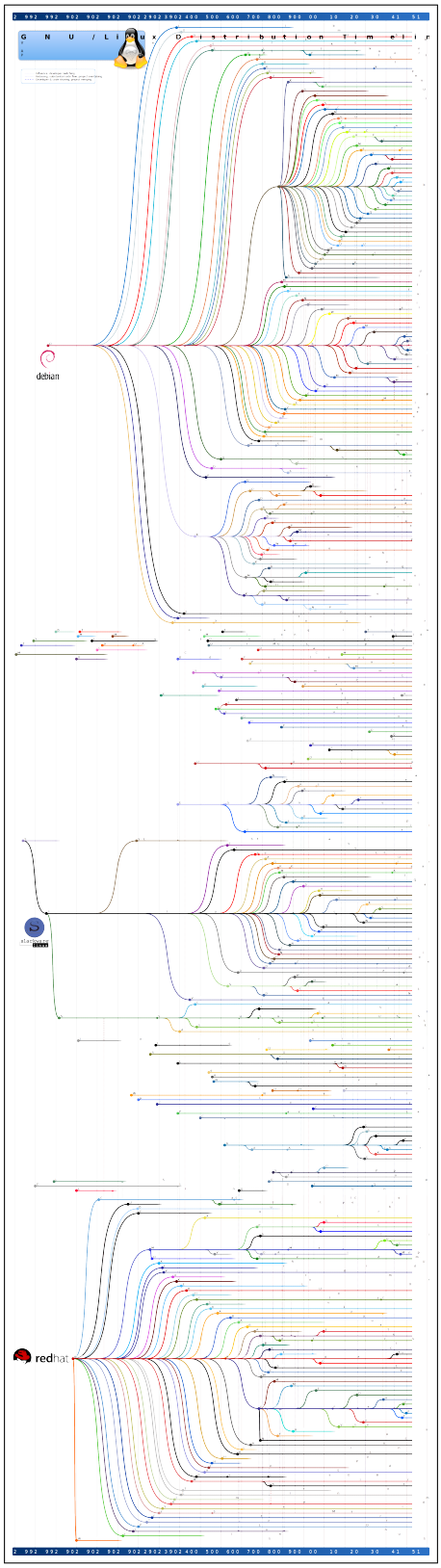

Figure 2.2: LINUX distributions

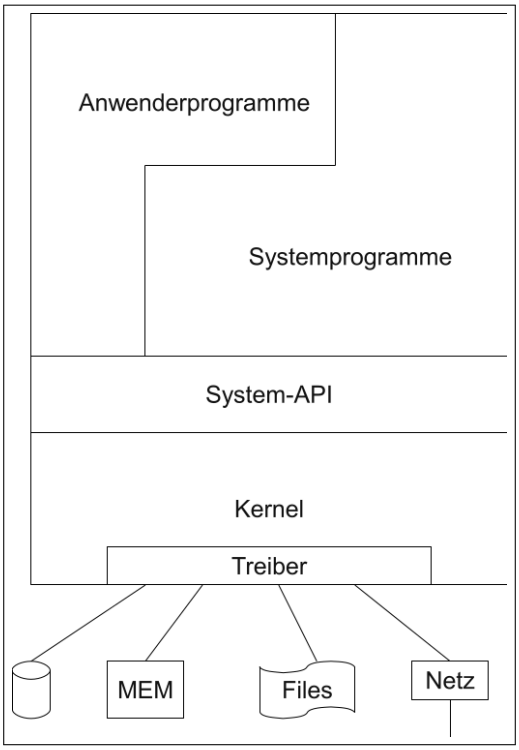

Figure 2.3: Grundaufbau von UNIX

The higher level services provided to application programmers are called API and in LINUX the SCI (**S**ystem **C**all **I**nterface).

2.3.3.2 Multitasking

UNIX can do several things at a time. This i called *multitasking*. But there is a big difference between *able to do at the same time* and *able to do at* nearly *the same time*. In systems with only a single CPU with a single core, only one command can be processed at a time. Nevertheless is UNIX able to give the user the impression that things happen at the same time by switching very cleverly and fast between jobs. Therefore, it is sufficient to say that UNIX is able to logically support multitasking. If multiple CPU's and/or multiple cores are active, things really habppen at the same time. But it does not make a difference to the user in most cases.

UNIX uses different priorities to deal with jobs according to their importance. There

needs to be abalnce between very fast switching to satisfy the expectancy of the users to have quick system responses and slow switching to have high performance in computing-intensive processes. The distribution of computing time is called *scheduling*[2, p.90ff]. Scheduling is an extremely complex subject and is solved in LINUX in a very modern way.

The kernel manage memory and other resources as well. A program simply cannot write everywher in memory. It is not just an issue about memory integrity but about safety as well. If applications could read everywhere it would be easy e.g. to write SW to spy on other users in the same system. Applications have to request memory, which e.g. could be done in C using `malloc`, which in UNIX systems is a call to the system API.

2.3.3.3 Multiuser

UNIX supports multiple users. Obviously it does not make much sense to have multiple users to sit in front of a single keyboard and a single screen. Therefore, most of the users will be logged in remotely using communication lines and protocols like telnet, ssh (**s**ecure **sh**ell) or HTTP. UNIX manages the integrity between users.

2.3.4 Kernel

fig. 2.4 shows the complexity of the system by showing the connection between the different parts of the kernel. Although it looks very dense and complicated, it can be seen, that the connections are mostly vertical and that the structure is 2-dimensional. Horizontally above each other there are layers of increasing abstraction, verticall next to each other different aspects of system control.

2.3.4.1 Memory management

Resources are limited and expensive. Therefore, memory management is a difficult task, striving to distribute the memory allocation in a way that everybode feels like having unlimited memory with no access time at all. This is not possible. Once, RAM (**R**andom

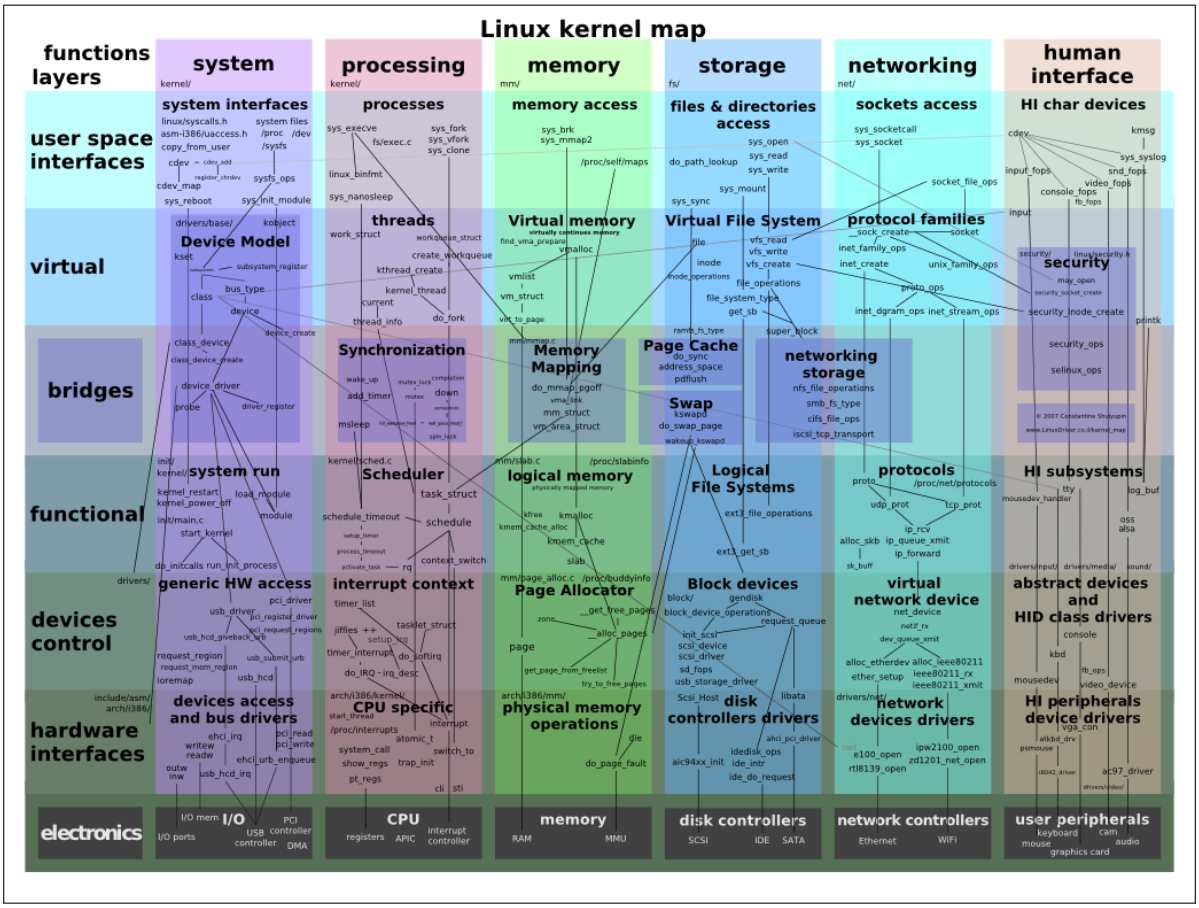

Figure 2.4: Struktur des Kernel (Quelle: wikipedia)

Access Memory) comes to an end, fast memory will be outsourced by writing the content to files. When memory which is not directly accessible is requested, the system needs to again load the neccessary areas to RAM before. This can take up to seconds instead of ns or even sorter access times in direct access. A good strategy for this memory *swapping* helps to improve the user experience significantly.

Programs do not 'see' anything of swapping. They just see a nearly infinite amount of memory, called *virtual memory*.

2.4 The file system

2.4.1 Directories and the directory tree

The main difference, users 'see' between windows and UNIX file systems it the difference between `/` and `\`, the symbols to seperate directory levels from each other. Actually, the UNIX way is superior to the windows way as `\` is the *escape* symbol in many contexts.

2.4.2 Links

In LINUX, files can be indicated by referencing other files. this helps as often files should be visible or accessible from multiple points in the file tree. These links in UNIX can be treated in nearly exactly the same way as the files they refer to. Most commands make no distinction between accessing a file or accessing a link to a file.

For more information on links, there is much material on the WWW[11].

2.4.3 Device files

Hard drives and partitions in LINUX are erepresented as files, visible in the directory `/dev`, as well as other HW-components like printers, consoles, serial ports etc. and need to be specially treated when used. The dangerous aspects of these devices are not accessible to normal system users to protect the system. these devices are generally named *device*s.

Thee are some special and interesting devices. Writing to `/dev/null` lets the output just vanish into nothingness. Reading from `/dev/zero` or from `/dev/random` yields the expected output.

2.4.4 Remote file systems

File systems on rmote machines can be included into the file tree in many ways. They then appear as if they were local and can be accessed by all standard ways of accessing files. A

[11] https://www.linux.com/tutorials/understanding-linux-links/

small selection:

- NFS (**N**etwork **F**ile **S**ystem)
- FTP (**F**ile **T**ransfer **P**rotocol)
- SFTP (**S**ecure **F**ile **T**ransfer **P**rotocol)
- Samba (SMB (**S**erver **M**essage **B**lock))

2.4.5 FHS

The structure of the directory tree is standardised in the FHS. It can be referred to at the WWW[12]. Not all LINUX distributions adhere to this standard, but most of them do.

The mot important directories will be covered later in the course as the need arises. Up to now, `/home` and `/proc` were touched briefly. Most interesting in the scope of this course are `/etc`, `/var` or `/srv`.

Table 2.1 istaken from version 2.3.

[12] http://refspecs.linuxfoundation.org/fhs.shtml

Table 2.1: Root-directory according to Filesystem Hierarchy Standard, Version 2.3

Verzeichnis	Bedeutung
bin	Essential command binaries
boot	Static files of the boot loader
dev	Device files
etc	Host-specific system configuration
lib	Essential shared libraries and kernel modules
media	Mount point for removeable media
mnt	Mount point for mounting a filesystem temporarily
opt	Add-on application software packages
sbin	Essential system binaries
srv	Data for services provided by this system
tmp	Temporary files
usr	Secondary hierarchy
var	Variable data

2.4.6 Permissions

In a multiuser environment it is compulsory to have permissions on different regions of the system. The system is much simpler, compared to windows systems, but still sufficient to implement secure multiuser environments.

There are three different aspects to each file or directory.

- *owner*: the file belongs to this user.
- *group*: the group the file is associated to. Users can belong to groups. Groups –and therefore users, belonging to such groups– have well-defined permissions
- everybode else or the *world*: everybody else (not owner or group member)

Permissions are '*read*', '*write*' und '*execute*', the shortcuts being 'r' (), 'w' () and 'x' (). ls-l lists a directory and the permissions of the entries., e.g.:

```
-rwxrwxr-x 1 hauseru users    870 10. Dez 11:32 abstract.tex~
drwxrwxr-x 2 hauseru users   4096 14. Feb 11:10 figures
```

The first entry belongs to the user `hauseru` and to the group `users`. The permissions rad from left to right

- owner: `rwx`, read, write and execute
- group: `rwx`, read, write and execute
- world: `rx`, read and execute

The second enry is a directory, indicated by the `d` at the begining of the line. With directories, permissions work slightly differently compared to files. Further information can be found at linux.com[13]. chmod

chmod[14] is the command to change permissions.

[13] https://www.linux.com/tutorials/understanding-linux-file-permissions/
[14] https://en.wikipedia.org/wiki/Chmod

2.5 The shell

A shell in UNIX has its windows counterpart in command.com. It is called a *command line* or *console* (see fig. 2.5). First steps in the console were done in section 2.2. It is important to note that a terminal for communication (e.g. a serial line) and a terminal for interaction are to be distinguished from each other.

2.5.1 Terminal

Historically, a terminal is a piece of HW with a keyboard and a display that can be connected to a computer system using a serial communication line. It was only used to display charcters sent by the conputer to the user and, accordingly, transfer characters keyed in by the user to the computer. The standards for communication were written down, amongst others, as ANSI escape codes, ANSI X3.64 und ISO/IEC 6429.

The shell today communicates to the computer in the same way as in the 70's of the last century. this has the advantage of always working inthe same way, no matter if communication is done locally or around the globe.

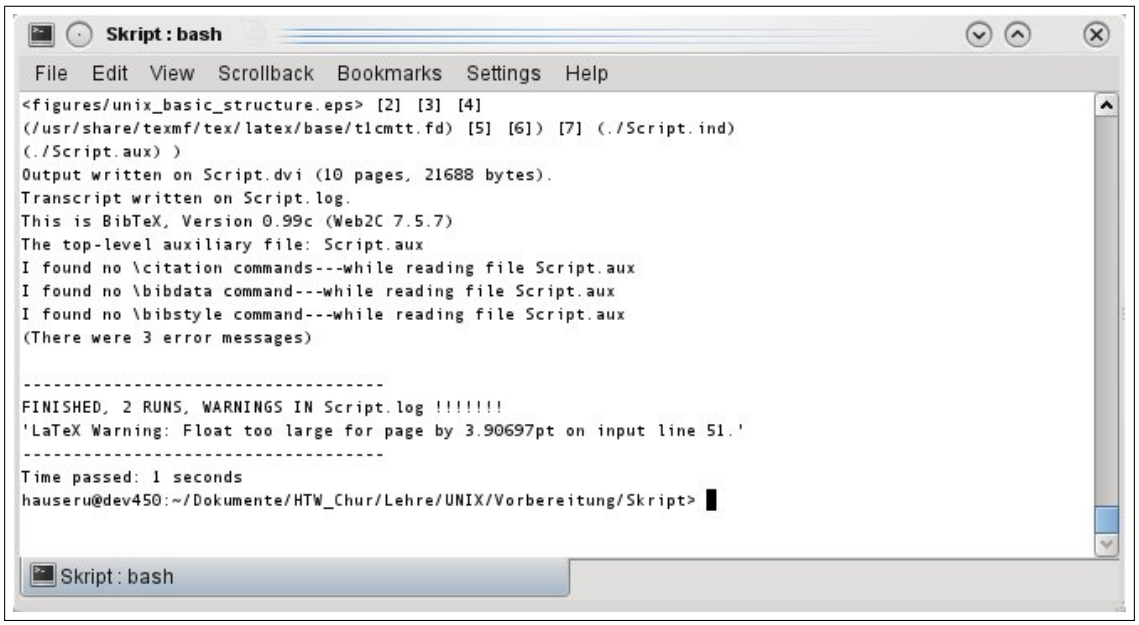

Figure 2.5: Eine Shell

2.5.2 Command line

A *command line* is a program to receive and analyse instructions given by a user and to issue the appropriate commands to the addressed computer. Data resulting from the operations, normally will be presented to the user back on the same display.

UNIX/LINUX knows a whole bunch of command line programs. They share a common base, but have quite distinct differences when it comes to details. The most common used shells are sh (bourne-**sh**ell) and its succesor, bash (**b**ourne-**b**gain-**sh**ell).

The commands known by a shell are divided into

- *internal* commands: Commands directly known by the shell program itself
- *external* commands: Commands where the executable is a file that needs to be called
- *alias*-commands: Commands, created by a definition during shell configuration, mostly creating shortkeys for complicated command expressions. Alias commands are merely renaming operations for commands.

type

type command will reveal the type of a command.

2.5.2.1 Entering commands

Entering commands is supported by the intelligence of modern shells. Normally, a system is configured in a way that for external commands, no path information needs to be given. A *path* is a way through the directory tree which leeds to a file. The `tab` key supports the entry of commands tremendously. Hitting `tab` at nearly any time will give useful information about possible entries or –if no alternativ is given– will complete the given expression automatically. Using `tab` becomes second nature quite quickly.

UNIX does not need special extensions for a file to be executable. If the permission (seesection 2.4.6) are set accordingly, a file simply is executed. If that is not possible, an error will be issued. Several types of files can be executed.

- executables themselves
- scripts for a shell
- files which need to be executed by an executable like PHP-Scripts etc.

LINUX –if configured properly– will know how to execute the necessary files.

Commands can be edited comfortably with →, ←, even in combination with `Ctrl`. Depending on the system configuation, the mouse can be used as well for pointing, marking and then appending by using the middle mouse button.

2.5.2.1.1 History The command line records and remembers typed command lines. this is called the *command history*. With ↑ ↓ the history can be walked back and forth. `Ctrl-r` allows the search for commands issued earlier by giving patterns. This is called *reverse intelligent search*. After proposing a command with Ctrl-r it can be edited normally before issuing it. `Ctrl-g` leaves this search mode.

2.5.3 The prompt

The *prompt* is the collection of characters in front of the blinking curser prior to entering a command. It can (and normally will) give essential information about the system. It can be configured widely[15].

In normal configurations it will at least give the name of the machine, the logged in user and the working directory, but much more information is possible.

2.5.4 Configuration

The shell configuration is divided into several aspects.

[15] https://linuxconfig.org/bash-prompt-basics

2.5.4.1 Variables

Shell variables control a lot of a shell's behaviour and can control a lot of application's behaviour as well. A shell variable consists of a name and a value. To define a shell variable its name is keyed in, followed by a `=` and its value.

```
MYVARIABLE='I am a variable'
```

Important variables will be covered seperately

2.5.4.1.1 `PATH` *PATH* controls the directories and their order in which the shell will search for executables if no explicit path is given. It should not include `.` for security reasons. Therefore, to call executables in the working directory, the executable needs to be prefixed with `./`.

2.5.4.1.2 `USER` *USER* holds the name of the logged-in user.

2.5.4.1.3 `HOSTNAME` *HOSTNAME* holds the name of the machine, the shell runs in.

2.5.4.1.4 `HISTSIZE` *HISTSIZE* controls how many commands are recorded for the history.

2.5.4.2 Configuration files

When a shell starts it calls on several files for configuration. The order of processing can be looked up in the `man`-page of the shell (see**??** 2.5.5.1.1). With these configuration files the behaviour of the shell can be widely influenced. These files are normally setup by the administrator and are only locally to be adjusted by system users.

2.5.5 Important commands

The most important commands are given here. There is a wealth of further information available on the WWW[16].

Reading about the main commands is strongly recommended.

Commands consist of the command name, options, other parameters to be involved and a return type which reports on the command success. Successful operation normally will return 0.

The return value can be accessed by echo $?. $?

```
ls --xxxx
ls: unrecognized option '--xxxx'
Try 'ls --help' for more information.
echo $?
2
```

Above is a possible communication between shell and user. `2`, given in the last line is a return value indicating an error.

2.5.5.1 Basics, navigation and directories

2.5.5.1.1 `man` how to get help All commands should have a `man`-page associated, callable by man command. In fig. 2.6 the manpage to `ls` is given. man

Sometimes –when calling up the man page– the user is requested to isuue a number. This number refers to a *section*. The section divide the information into several aspects. Section 1 e.g. refers to shell commands, 3 are so called subroutines but effectively describe a huge number of C system calls. More information is given under `man man`.

2.5.5.1.2 `apropos` This command supports `man`. It searches the summaries of the man pages for keywords. `apropos directory` will pint all names of commands who deal with

[16] https://manpages.debian.org/

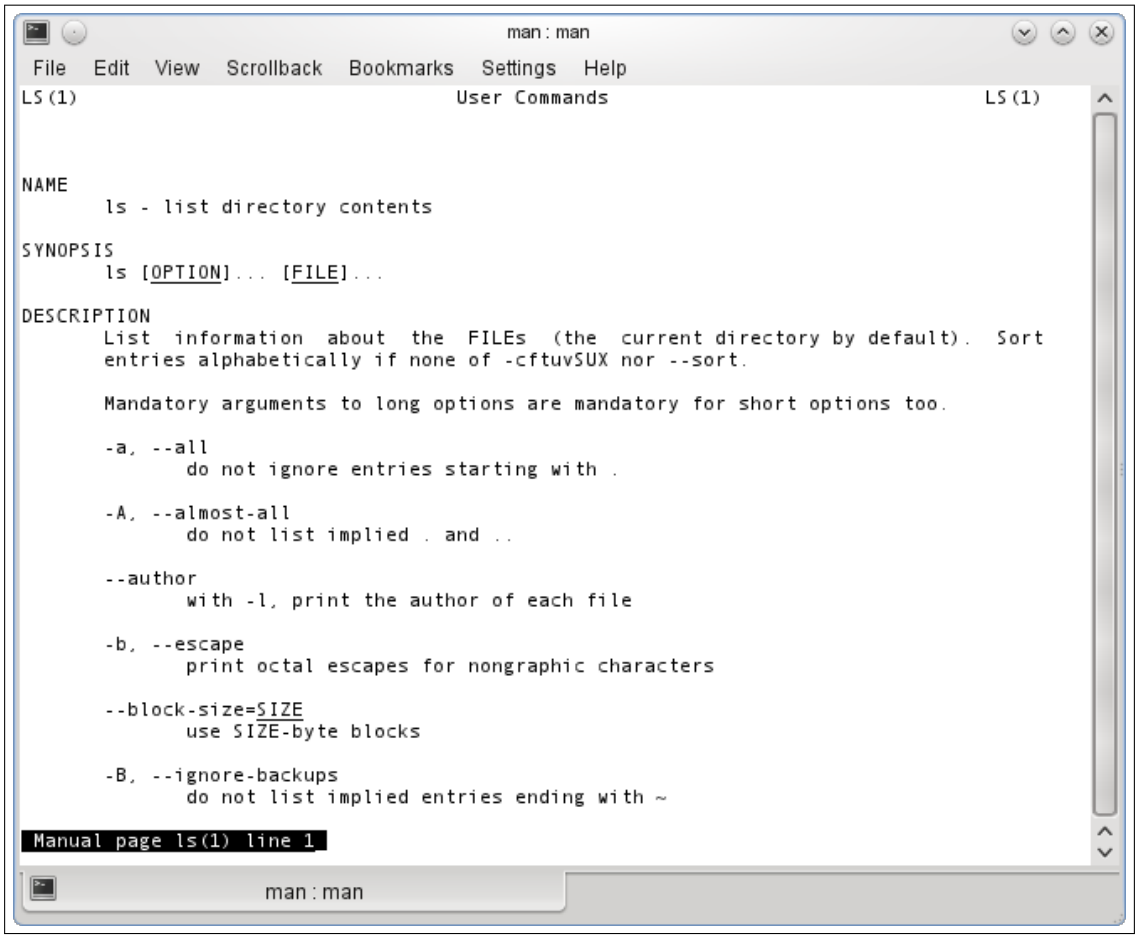

Figure 2.6: Man

directories. By using this command, comands asociated to some subject can be found.

2.5.5.1.3 `ls` `ls` shows directory content.

2.5.5.1.4 `pwd` This command shows the working directory.

2.5.5.1.5 `cd` This command changes the working directory.

2.5.5.1.6 `pushd`, `popd` **und** `dirs` These commands allow to change directories using a history stack.

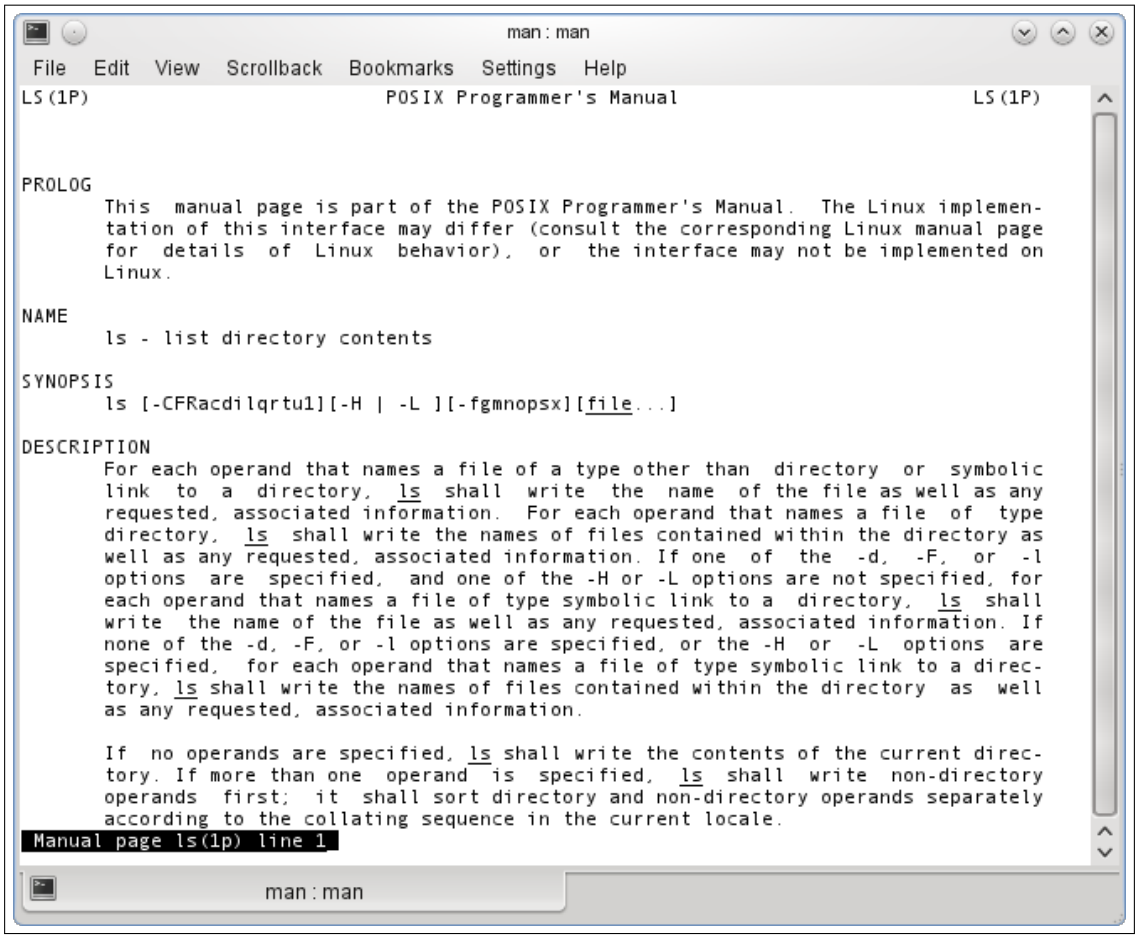

Figure 2.7: Man, verschiedene Versionen

2.5.5.2 Console

2.5.5.2.1 `clear` Clears the console.

2.5.5.2.2 `reset` Initialises the console.

2.5.5.2.3 `tput` Helps to control advanced activities on the console display. Uses *terminfo*

E.g. `tput cup 0 0` moves the cursor to position (0|0) of the terminal.

2.5.5.3 Information and search

2.5.5.3.1 `date` gives the date.

2.5.5.3.2 `echo` prints text to the console.

2.5.5.3.3 `du` Informs about disk usage. Interesting option: `-h`.

2.5.5.3.4 `find` Very versatile command for file searching.

2.5.5.3.5 `grep` Searches for character patterns, extremely flexible to use[17,18].

2.5.5.4 Account and system

2.5.5.4.1 `groups` Shows the groups a user belongs to.

2.5.5.4.2 `passwd` Change a password.

2.5.5.4.3 `shutdown` shutdown the machine. On many systems this is only allowed to be done by the administrator.

2.5.5.4.4 `mount` Connect to other file systems and include them into the directory tree.

2.5.5.5 Files

2.5.5.5.1 `cat` print some file content.

2.5.5.5.2 `less` File pager. This command can be used to show, search and navigate big files page by page.

2.5.5.5.3 `tail` Prints the end of a file.

2.5.5.5.4 `cmp`, `diff` Tools to compare files.

2.5.5.5.5 `sort` sort a file according to given aspects.

[17] https://en.wikipedia.org/wiki/Grep
[18] https://en.wikipedia.org/wiki/Regular_expression

2.5.5.5.6 `touch` Create a file or update its time stamp.

2.5.5.5.7 `chmod` Change file permissions.

2.5.5.5.8 `chown` Change file owner.

2.5.5.5.9 `chgrp` Change group of file.

2.5.5.5.10 `cp` Copy files.

2.5.5.5.11 `mv` Move files.

2.5.5.5.12 `rm` Remove files.

2.5.5.5.13 `rmdir` Remove directories.

2.5.5.5.14 `ar`, `compress`, `gzip`, `tar`, `unzip`, `zip` Deal with archives.

2.5.6 Pipes

Commands can be used in chains, chaining the output of one command directly to the input of another command. the notation is
`cmd1 | cmd2 | cmd3`. The symbol `|` therefore is often named the *pipe* symbol. Symbolically ther is a pipe created, chanelling the output from one command to the next. With *piping* very flexible command combinations can be created.

The concept of pipes change the complete system as the basic commands turn into little tools which can be combined to powerful processing chains, performing amazing things on a single line of command.

A special command in this context is xargs. This command takes the input and puts it at the end of the following command. This is often used to find a collection of files with one command and the do manipulations or searches on this collection of files.

xargs

```
find . -iname "*.txt" -type f | xargs grep -i boss
```

The command line above will search for all files with extension `txt` and look in all of them for the word `boss`, ignoring lower or upper case notaion.

Such an *anonymous pipe* ist unidirectional and is removed after the associated processes are finished.

There are named pipes. These must be explicitly created in the file system. They have no exiry time. To create a named pipe mkfifo thepipe is used to create a pipe named `thepipe`. mkfifo It will be indicated (using `ls -l`) as follows:

```
prw-r--r-- 1 hauseru users      0 28. Feb 11:55 thepipe
```

As UNIX is able to run processes at the same time, input and output of independent processes can now be coupled using such a pipe. Synchronisation is automatically managed.

2.5.7 Patterns

In ?? 2.5.5.3.5, the command `grep` is shown. This is command generally able to look for all kinds of patterns. The search for patterns is a very general subject in itself as it is necessary to search for things very often. The WWW produces a wealth of information if the search expressions 'regular expressiosns' or 'regex' are issued. Wikipedia[19]gives a good start.

2.5.8 Quoting und escape-Sequenzen

Quite often, the need to identify groups of words as a single text arises. The use od several types of quotation marks is very confusing at first. Evene experienced users sometimes run into difficulties when using quotes.

```
echo Hello, Don
echo "Hello, Don"
echo 'Hello, Don'
```

[19] https://en.wikipedia.org/wiki/Regular_expression

will all print the same string, `Hello, Don`. Many characters, however, have a special meaning in the bash (as in languages like C). This fact leads to new challenges. E.g., how can we print " itself? To accomplish tasks like this *escape* ()\) is there.

`echo Hello, \"Don\"` will print as `Hello, "Don"`.

The combination `\"` tells the system *to take the* " *literally* and not as an indication of the end of a string of words. In the same way, `echo Hello, \'Don\'` will yield `Hello, 'Don'`.

`name="Bruno";echo $name` yields `Bruno`. What happens here? First, a variable named `text` is defined, bearing the content `Bruno`. Then, the content of the variable is printed. Here `$` has a special meaning, starting the name of a variable.

Variables (section 2.5.4.1) play an important role in quoting.

`name="Bruno";echo "$name"` yields `Bruno` again. But

`name="Bruno";echo '$name'` yields `'$name'`.

The single quotes seem not to lead to a variable evaluation, but to a literal printing. It is often not really easy to predict the behaviour without a really acute knowledge of escaping:

`name="Bruno";echo -e "1:\\t$name"` yields `1: Bruno`,

`name="Bruno";echo -e '1:\\t$name'` yields `1:\t$nam`.

The option `-e` at `echo` is necessary to trigger the evaluation of escape sequences like `\t` themselves[20]. Even the GNU (**G**NU's **N**ot **U**nix)GNU bash manual-page (`http://www.gnu.org/software/bash/manual/bashref.html#Single-Quotes`) is quite involved to be understood correctly. The quickest option often is a good starting guess followed by strategically clever trials and tests.

2.5.9 Stream redirection

Commands do not necessarily need to print to the console. > filename will redirect the > output to a file. This first was used in section 2.2.1 without explanation. The target can be any kind of file, e.g. a device or a pipe.

[20] `https://en.wikipedia.org/wiki/Escape_sequence`

The output of `ls >/dev/null` discards the output.

Die error output is not redirected this way.

`cat xyz >/dev/null` will generate an error on the screen, if `xyz` does not exist. The reason is that output is divided into several channels. Giving no number uses channel 1 as the standard output channel. this channel is often named `stdout`, as this is the name for this channel in C. (`stderr`) has number 2. `cat xyz 1>/dev/null 2>error` discards the normal output and diverts `stderr` into the assigned file.

If `stdout` and `stderr` are to be redirected into the same target, this is indicated by `2>&1`. `cat xyz 1>/dev/null 2>&1` redirects `stdout` to `null` and *then* `stderr` to where `stdout` used to direct. The order need to be taken into account. `cat xyz 2>&1 1>/dev/null` redirects `stderr` to where `stdout` points to (still at the console) and then `stdout` to `null`.

Input can be redirected as well using `<`. `grep verb <Script.tex` is the same as `cat Script.tex | grep verb` (see section 2.5.6 for pipes).

For output to appear on the console as well as in a file e.g. ls | tee file can be used. It is derived from a piece of water pipe formed like a `T` to divert the flow into two directions at the same time.

tee

2.5.10 Proces control

UNIX is able to run processes concurrently. This can be used by tha bash as well as by programming languages like C. In a shell, a command can be topped by using CTRL-Z. Using bg (background) afterwords will restart the process, but in the background. This is visualised by the reappearing prompt. Now, new commands can be issued. Both commands now share the same output channel, the console. They will write on the console at the same time, normally resulting in a chaotic output. the command fg (foreground) pulls the process back to the foreground. A process can be started in background directly by appending `&` at the end of a command.

bg

fg

The following commands are helpful when working with processes.

ps

ps lists all processes running in shell the command is un on

pstree

pstree lists all processes in a tree structure

top

top shows the most important processas and some more system information. the command can be stopped with `q`.

Each process has a number which identifies the process. Using `pstree` shows, that all processes can be traced back to a general parent process called *init*. This process is the one created first when a LINUX system is started. All other processes are started from there as a cascade.

`ps -f` shows some additional information:

```
UID        PID  PPID  C STIME TTY          TIME CMD
hauseru  10043 10041  0 08:58 pts/1    00:00:00 /bin/bash
hauseru  20142 10043  0 12:55 pts/1    00:00:00 ps -f
```

Each process is allocated to a user (UID), knows its ID (PID), the ID (PPID) of its parents its (dynamic) priority (C) the starting time (STIME) and to which terminal it is associated to (TTY). Additionally, the processor time used (TIME) and the full command line itself (CMD) are known by the process.

Processes can be influenced by commands. nice influences the processor time, a process is allocated. kill sends a signal to a process. These signals can be caught (trapped) by a process and can stimulate whatever reaction the programmer wants to have. Often, this command is used to kill a process (Signal 9 is the standard there and is issued bc `Ctrl-C`), hence the command name `kill`.

nice

kill

Chapter 3

Introduction to JS

3.1 Commonalities with C

3.2 Variables and objects

3.3 Using JSON to build objects

3.4 Scope

3.5 Events and listeners

3.6 Modules

3.6.1 Exports

3.7 NodeJS

3.8 JS in the Browser

Chapter 4

Introduction to the Raspberry Pi

4.1 Overview

4.2 Setting up

4.2.1 Images

4.2.1.1 Standard

4.2.1.2 Headless

4.2.2 Preparation

4.2.3 Starting

4.2.3.1 Contacting via ssh

4.2.4 Improvements

4.2.4.1 SFTP

4.2.4.2 X-Server

4.2.4.3 Mounting

Chapter 5

Building a IoT device

5.1 Pi, accessing peripherals

5.2 Access over the WWW

Chapter 6

Building a WoT device

6.1 Building a WoT WWW server using NodeJS

6.1.1 A WWW server for M2M communication

6.1.2 A WWW server for the browser

6.2 Sending and receiving over MQTT

6.2.1 Synchronising data using push messages

Hier muss alles rein Ã¼ber MQTT brokerage und pushing mit WebSockets!!!

Bibliography

[1] Dominique Guinard and Vlad Trifa. *Building the Web of Things: With Examples in Node.Js and Raspberry Pi*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2016.

[2] Phillip A. Laplate. *Real-Time Systems Design and Analysis*. IEEE Press, 2004.

Index

/dev, 29

/dev/random, 29

/dev/zero, 29

`>`, 42

`$?`, 36

`bg`, 43

`cat`, 19

`cd`, 19

`chmod`, 31

`fg`, 43

`find`, 19

`kill`, 44

`ls`, 19

`man`, 36

`mkfifo`, 41

`nice`, 44

`proc`, 20

`pstree`, 44

`ps`, 43

`pwd`, 19

`tee`, 43

`top`, 19, 44

`touch`, 19

`type`, 33

`xargs`, 40

command

- `ps`, 43

account

- root, 19
- user, 19

alias, 33

anonymous pipe, 41

ANSI

- escape code, 32
- X3.64, 32

archive, 40

assembly language, 23

bash, 33

break, 20

BT, 11

closed Shop, 22

command

- `>`, 42
- `$?`, 36
- `bg`, 43

`cat`, 19

`cd`, 19

`chmod`, 31

`fg`, 43

`find`, 19

`kill`, 44

`ls`, 19

`man`, 36

`mkfifo`, 41

`nice`, 44

`proc`, 20

`pstree`, 44

`ps`, 43

`pwd`, 19

`tee`, 43

`top`, 19, 44

`touch`, 19

`type`, 33

`xargs`, 40

alias, 33

external, 33

history, 34

internal, 33

command history, 34

command line, 32, 33

compare, 39

compression, 40

computer

cluster, 17

concurrency, 43

console, 32

content, 39

copy, 40

daisy chaining, 9

database, 15

delete, 40

device, 29

device drivers, 23

directory

root, 19

working, 19

error output, 43

escape, 29, 42

execute, 31

external, 33

files, 29

Filesystem Hierarchy Standard, 30

GNU, 42

group, 31, 40

HAL, 22

hard drive, 29

HISTSIZE, 35

HOSTNAME, 35

HP-IB, 10
HTTP, 27
http upgrading, 7

IEEE-488, 10
init, 44
internal, 33
ISO/IEC
    6429, 32

kernel, 27

linked data, 15
LINUX, 23
LTE-M, 13

memory
    swapping, 28
    virtual, 28
memory management, 27
move, 40
MULTICs, 22
Multiprocessing, 22
multiprocessing, 17
multitasking, 17, 26
Multiuser, 22
multiuser, 17

navigate, 39
NB-IOT, 13

operating systens, 22
owner, 31, 40

pager, 39
PAN, 11
partition, 29
password, 39
PATH, 35
path, 33
permission
    execute, 31
    read, 31
    write, 31
permissions, 40
pipe, 40
piping, 40
polling, 8
prompt, 19, 34

read, 31
redirection, 42
regex, 41
regular expressiosns, 41
remove, 40
reverse intelligent search, 34
root acount, 19
root directory, 19

scheduling, 27

search, 39, 41
section, 36
semantic web, 15
service, 22
sh, 33
show, 39
SigFox, 13
signal, 44
SSH, 27
star
    ofstars, 10
star of stars, 10
stars, 10
stderr, 43
stdout, 43
structured data, 15
swapping, 28

telnet, 27
terminfo, 38
time stamp, 40
topology, 8
trap, 44

UNIX, 23
USER, 35
user accounts, 19

virtual memory, 28

WAN, 11
websocket, 7
Websockets, 14
working directory, 19
world, 31
write, 31

ZigBee, 12