



# H5 File Systems

**HO  
GENT**

# Inhoud

5.1 Persistente opslag

5.2 Files

5.3 Directories

5.4 File systems

5.5 Partities

5.6 Booten

5.7 Voorbeelden FS

5.8 Opslag in Docker

**HO  
GENT**

## **5.1 Persistente opslag**

**HO  
GENT**

## Persistente opslag

Processen hebben nood aan persistente opslag:

- Data bewaren na afsluiten van het proces.
- Grote hoeveelheden data opslaan.
- Data delen tussen processen.

**HO  
GENT**

Zoals je geleerd hebt in het vorige hoofdstuk, halen processen hun instructies en data uit het RAM. Naast dit RAM hebben processen ook nood aan **persistente** opslag:

- Data in het RAM gaan verloren wanneer het proces afsluit. Als een proces data wil bewaren, dan moet deze data op een persistent opslagmedium worden bewaard.
- RAM is beperkt in capaciteit. Persistente opslag biedt een grotere opslagruimte aan.
- Processen werken in hun eigen (virtuele) adresruimte en zijn afgeschermd van elkaar. Via een persistent opslagmedium kunnen processen data delen met elkaar.

## Hard disk drive (HDD)

- Magnetisch opslagmedium
- Bevat draaiende schijven
- Heeft daardoor tijd (**seek time**) nodig om zich te positioneren
- Grote capaciteit
- Lage kost



**HO  
GENT**

Een **hard disk drive (HDD)** is een magnetisch opslagmedium dat een grote opslagcapaciteit biedt voor een lage prijs.

Doordat een HDD werkt met draaiende schijven, is er telkens een kleine vertraging (**seek time**) wanneer de leeskop zich moet positioneren boven de gevraagde data.

## Solid-state drive

- Bevat geen bewegende onderdelen
- Opslag gebeurt in integrated circuits (ICs)
- Sneller dan HDD
- Hogere kost per GB



**HO  
GENT**

Een **solid-state drive (SSD)** is een performanter maar duurder alternatief voor een HDD. In tegenstelling tot een HDD heeft een SSD geen draaiende onderdelen. De opslag gebeurt door circuits (ICs), waardoor de schijf robuuster is en geen seek time nodig heeft. Het nadeel is uiteraard de kostprijs: een SSD heeft een hogere prijs/GB dan een HDD.

Opgelet: In de volksmond wordt de term “harde schijf” gebruikt als algemene verwijzing naar persistente opslagmedia van het systeem. Dit kan dus zowel een HDD als een SSD zijn.

## CD/DVD

- Optisch opslagmedium.
- Beperkte capaciteit en performantie.
- Vaak read-only.
- Voornamelijk geschikt voor distributie of backup.



**HO  
GENT**

**CDs** en **DVDs** zijn optische opslagmedia. Ze hebben een beperkte capaciteit en performantie en zijn bovendien vaak read-only. Om deze redenen worden ze vooral gebruikt als distributiemedia, of voor backups.

## USB stick

- Extern opslagmedium, aangesloten via USB
- Opslag gebeurt in ICs, net zoals SSD
- Beperkte capaciteit en performantie



**HO  
GENT**

**USB sticks** maken gebruik van ICs, net zoals SSDs. Het zijn echter externe opslagmedia die worden aangesloten via USB, waardoor ze een beperkte capaciteit en performantie hebben.



## **5.2 Files**

**HO  
GENT**

## Wat is een file?

- Een fysiek opslagmedium is onderverdeeld in blokken
- Een **file** (bestand) groepeert data uit één of meer blokken
- Een file is dus een **abstracte** eenheid die de complexiteit van de fysieke opslag verbergt
- De implementatie van files gebeurt door een **file system** (bestandssysteem)

**HO  
GENT**

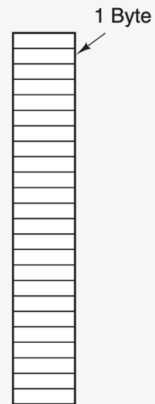
Een fysiek opslagmedium verdeelt zijn opslagruimte in **blokken**, vaak met een complexe layout. Een **file** (*bestand*) is een abstracte eenheid die data uit één of meerdere blokken groepeert en voorstelt als één geheel.

Het beheer van files en het toekennen van blokken behoort tot de taken van een **file system** (*bestandssysteem*). Op deze manier verbergt een file system de complexiteit van de fysieke opslag voor de gebruikers van het besturingssysteem.

## Voorstelling

Een file wordt voorgesteld als een opeenvolging van bytes.

Dit is echter een abstractie: in realiteit kunnen deze bytes verspreid zijn over het opslagmedium.



**HO  
GENT**

Naar de gebruikers toe wordt een file voorgesteld als een opeenvolging van bytes.

Op de vorige slide heb je geleerd dat dit een abstractie is: in realiteit kunnen de bytes van een file verspreid zijn over verschillende blokken die niet altijd aan elkaar aansluiten.

## Eigenschappen

- Naam, eventueel gevolgd door een extensie
- Huidige en maximale grootte
- Toegangsrechten
- Datum van aanmaak
- Datum van laatste aanpassing
- Verwijzing naar de datablokken
- ...

Een file heeft heel wat eigenschappen:

- Een naam, eventueel gevolgd door een extensie. Sommige besturingssystemen hechten belang aan deze extensie; bij andere is het louter een conventie.
- Een huidige en maximale grootte.
- Toegangsrechten.
- De datum van aanmaak en van laatste aanpassing.
- Een verwijzing naar de blokken waar de bytes van het bestand opgeslagen zijn.
- ...

Welke eigenschappen worden opgeslagen en hoe/waar deze worden opgeslagen hangen af van het gebruikte bestandssysteem.

## Soorten

Op UNIX geldt de regel “**everything is a file**”:

- **directories** bevatten andere bestanden of mappen
- **links** maken een bestand op meerdere plaatsen in het bestandssysteem zichtbaar
- **speciale bestanden** zijn gekoppeld aan hardware
- **sockets** zorgen voor netwerkcommunicatie
- **pipes** verbinden de output van een proces met de input van een ander proces

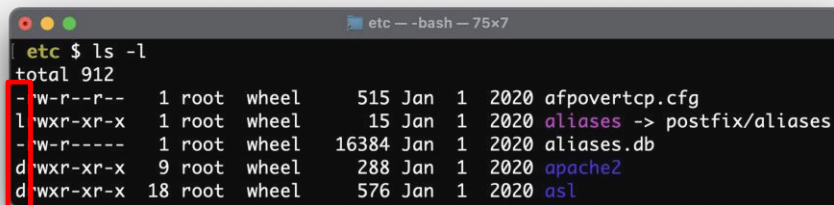
**HO  
GENT**

Op UNIX systemen (en afgeleiden zoals Linux) worden heel wat dingen voorgesteld als files. Op deze systemen zijn er dus meerdere soorten bestanden:

- Gewone bestanden bevatten data
- **Directories** (*mappen*) bevatten andere bestanden of mappen. Zij zorgen voor een hiërarchische structuur in het bestandssysteem
- **Links** maken een bestand op meerdere plaatsen in het bestandssysteem zichtbaar
- **Block of character special files** geven toegang tot hardware (bvb. printers of /dev/null)
- **Sockets** zorgen voor netwerkcommunicatie
- **Pipes** (FIFO) verbinden de output van een proces met de input van een ander proces

## Soorten

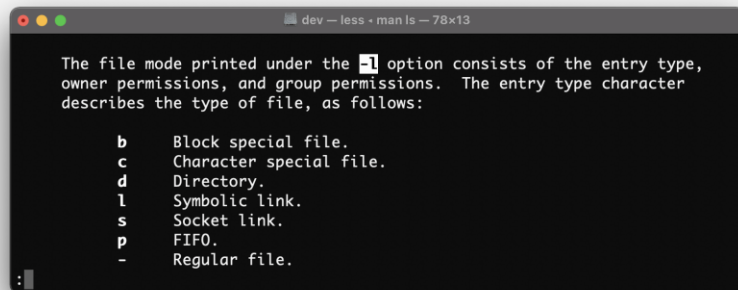
De output van `ls -l` toont het type van elk bestand:



```
etc -- -bash -- 75x7
etc $ ls -l
total 912
-rw-r--r-- 1 root wheel 515 Jan 1 2020 afpovertcp.cfg
lrwxr-xr-x 1 root wheel 15 Jan 1 2020 aliases -> postfix/aliases
-rw-r----- 1 root wheel 16384 Jan 1 2020 aliases.db
drwxr-xr-x 9 root wheel 288 Jan 1 2020 apache2
drwxr-xr-x 18 root wheel 576 Jan 1 2020 asl
```

# Soorten

Deze types zijn te vinden in de man page van **ls**:

A screenshot of a terminal window titled 'dev - less - man ls - 78x13'. The terminal displays the man page for the 'ls' command, specifically the section explaining the file mode. The text reads: 'The file mode printed under the -l option consists of the entry type, owner permissions, and group permissions. The entry type character describes the type of file, as follows:'. Below this, a list of file types is shown, each with a character and a description: 'b' for Block special file, 'c' for Character special file, 'd' for Directory, 'l' for Symbolic link, 's' for Socket link, 'p' for FIFO, and '-' for Regular file.

```
dev - less - man ls - 78x13

The file mode printed under the -l option consists of the entry type,
owner permissions, and group permissions. The entry type character
describes the type of file, as follows:

b      Block special file.
c      Character special file.
d      Directory.
l      Symbolic link.
s      Socket link.
p      FIFO.
-      Regular file.
```

## Soorten

De output van **ls -F** toont het type via een suffix:

```
etc $ ls -F
afpovertcp.cfg      openldap/
aliases@            pam.d/
aliases.db          passwd
apache2/            paths
asl/                paths.d/
asl.conf            periodic/
```

Suffix	Type
/	map
*	uitvoerbaar bestand
@	link
=	socket
	named pipe



## Bytes uitlezen

Ook de manier waarop we bestanden uitlezen kan verschillen:

- **Sequentieel**  
Bytes moeten in volgorde uitgelezen worden
- **Random access**  
Bytes kunnen in een willekeurige volgorde uitgelezen worden

**HO  
GENT**

Bestanden kunnen op twee manieren toegang geven tot hun bytes:

- Een bestand met **sequentiële toegang** dient in volgorde uitgelezen te worden. Het is niet mogelijk om bv. byte 100 uit te lezen, zonder dat bytes 1-99 reeds uitgelezen zijn (of mee uitgelezen worden).
- Een bestand met willekeurige toegang (**random access**) kan in eender welke volgorde uitgelezen worden. Bij dit soort bestanden is het wel mogelijk om bv. meteen byte 100 op te vragen.

## **5.3 Directories**

**HO  
GENT**

## Wat is een directory?

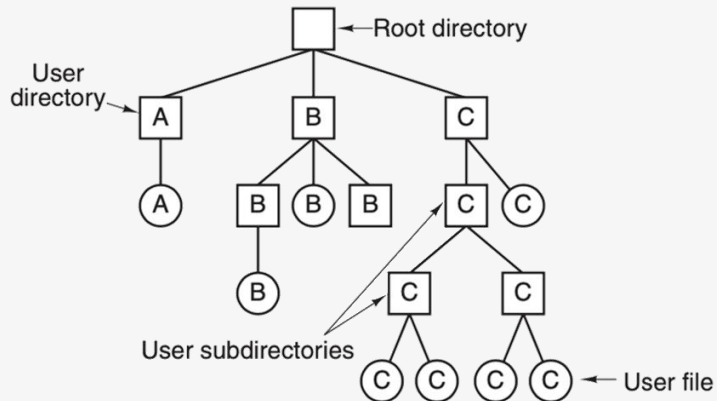
- Een **directory** (map) groepeert bestanden
- Een directory kan ook andere directories bevatten
- Dit creëert een **hiërarchische** structuur
- De implementatie van directories gebeurt door een file system
- Dit kan bv. d.m.v. een bestand

**HO  
GENT**

Een **directory** (*map*) groepeert files en kan zelf ook andere directories bevatten. Dit zorgt voor een **hiërarchische** structuur in het bestandssysteem.

De implementatie van directories is opnieuw de taak van een bestandssysteem. Een mogelijke implementatie is bv. om een directory op te slaan als een bestand met daarin de inhoudstafel van de directory.

# Hiërarchie

**HO  
GENT**

Dit voorbeeld toont een hiërarchische structuur waarbij de **root directory** (de hoofdmap) van het bestandssysteem één map per gebruiker bevat. Elke gebruikersmap bevat dan weer andere bestanden en mappen.

## Padnamen

- Het **absoluut pad** naar een bestand of map start bij de root directory en doorloopt de hiërarchie, bv:  
    /home/hogent/os (Linux of Mac)  
    C:\Users\hogent\os (Windows)
- Een **relatief pad** start vanuit een bestaande directory en kan de speciale verwijzingen . en .. gebruiken:  
    ../hogent/os (Linux of Mac)  
    ..\hogent\os (Windows)

**HO  
GENT**

Het **absoluut pad** naar een bestand of map start bij de root directory en beschrijft de weg naar dit bestand of naar deze map. Op Linux en Mac wordt een slash (/) gebruikt als scheidingsteken in een pad; op Windows is dit een backslash (\), bvb:

**/home/hogent/os**  
**C:\Users\hogent\os**

Merk op dat een absoluut pad op Windows telkens start met een letter die gekoppeld is aan een bestandssysteem. Op Linux en Mac worden geen letters gebruikt, maar worden alle bestandssystemen samengebracht tot één virtueel bestandssysteem (zie later).

Een relatief pad start vanuit een bestaande directory en kan de speciale verwijzingen . (huidige directory) en .. (parent directory) gebruiken, bvb:

**../hogent/os**  
**..\hogent\os**

## **5.4 File systems**

**HO  
GENT**

## Wat is een file system?

- Een **file system** (bestandssysteem) is een onderdeel van het besturingssysteem
- File systems beheren de fysieke opslagruimte en implementeren files en directories
- Een besturingssysteem kan meerdere file systems (tegelijk) ondersteunen

**HO  
GENT**

Een **file system** (*bestandssysteem*) is een onderdeel van een besturingssysteem. File systems beheren de fysieke opslagruimte en wijzen deze toe aan files en directories. File systems implementeren dus files en directories.

De meeste besturingssystemen ondersteunen meerdere file systems. Er kunnen ook meerdere file systems tegelijkertijd actief zijn, bv. één per opslagmedium dat aanwezig is in het systeem.

## Implementatie van files

- Files kunnen op verschillende manieren uitgewerkt worden:
  - Contiguous storage
  - Linked lists
  - File allocation table (FAT)
  - Index nodes (inodes)
- De volgende slides overlopen deze manieren.

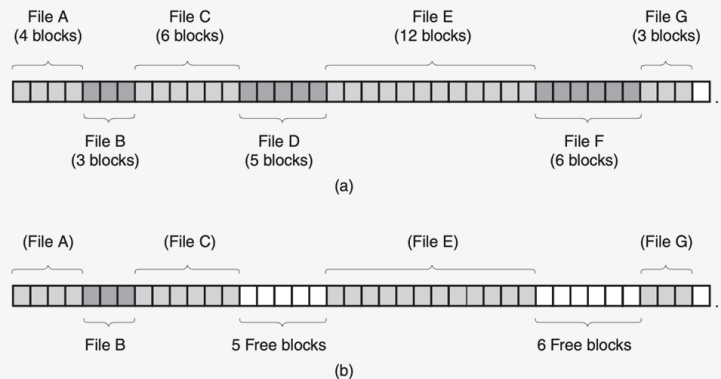
**HO  
GENT**

De implementatie van files kan op verschillende manieren gebeuren. De volgende slides overlopen:

- Contiguous storage.
- Linked lists.
- File allocation table (FAT).
- Index nodes (inodes).



## Contiguous storage



**HO  
GENT**

Bij **contiguous storage** (*samenhangende opslag*) wordt elk bestand in één of meer **aansluitende blokken** opgeslagen.

Dit is een eenvoudige methode met een **goede leessnelheid** en met ondersteuning voor **random access**. Er zijn echter ook belangrijke nadelen:

- Als een bestand groeit, dan moet het mogelijk verplaatst worden naar een grotere vrije ruimte.
- Als een bestand wordt verwijderd, en de vrije ruimte wordt ingenomen door een kleiner bestand, dan ontstaat er **fragmentatie** (= meer en meer kleine ruimtes die niet opgevuld geraken). Een systeem dat deze techniek gebruikt moet dus regelmatig gedefragmenteerd worden om de kleine vrije ruimtes terug samen te voegen. Een visuele uitleg van fragmentatie vind je op <https://www.youtube.com/watch?v=BKsVM89ZhRk>.

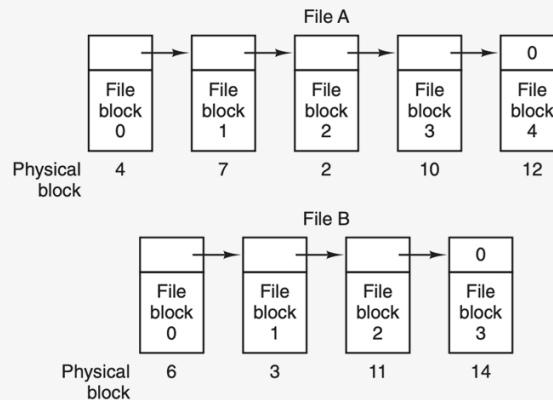
Deze combinaties van voor- en nadelen maakt contiguous storage voornamelijk geschikt voor read-only of write-once media, zoals CDs en DVDs.

In de figuur wordt uit (a) bestanden D en F verwijderd, (b) toont de opslag na het verwijderen.

## Contiguous storage

- 👍 Eenvoudig
- 👍 Goede leessnelheid
- 👍 Ondersteunt random access
- 👎 Bestanden die groeien moeten worden verplaatst
- 👎 Leidt tot fragmentatie

## Linked lists






**HO  
GENT**

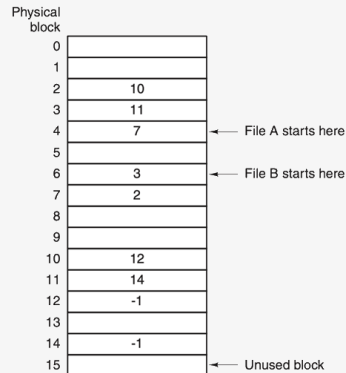
Bij **linked lists** (*gelinkte lijsten*) hoeven de datablokken van een bestand niet aan te sluiten. Elk blok bevat namelijk een **verwijzing** (link) naar het volgende blok, dat zich eender waar mag bevinden.

Deze techniek lost de problematiek rond fragmentatie op, maar heeft een **slechte performantie**. Doordat de datablokken verspreid staan, daalt de leesperformantie, i.h.b. op opslagmedia met een seek time. Bovendien ondersteunt deze techniek **geen random access**, omdat de blokken in volgorde moeten worden uitgelezen.

## Linked lists

-  Geen fragmentatie
-  Slechte leessnelheid
-  Ondersteunt geen random access

## File allocation table (FAT)



**HO  
GENT**

Een **file allocation table (FAT)** (*toewijzingstabel*) is een verbetering van de linked list techniek. De datablokken van elk bestand vormen nog steeds een linked list, maar alle verwijzingen tussen de blokken worden samengebracht in één tabel.

Door deze tabel in het RAM te bewaren, is het mogelijk snel een lijst te vinden van alle blokken van een bestand, zonder de blokken zelf te moeten uitlezen. Dit verhoogt de **leessnelheid** en maakt **random access** mogelijk.

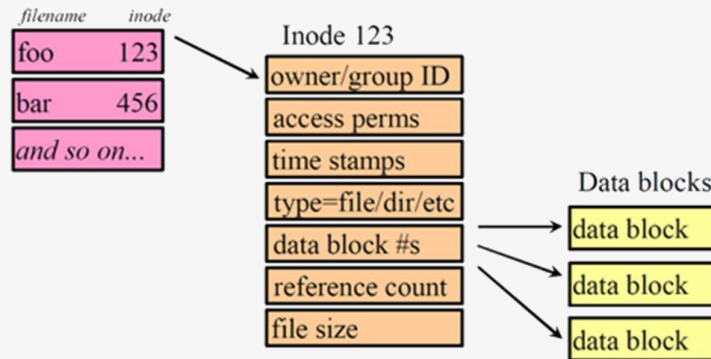
Het nadeel van deze techniek is dat de FAT een **hoog RAM-verbruik** kan hebben. Neem bvb. een harde schijf van 1TB, onderverdeeld in blokken van 4KB. Deze schijf heeft dan  $1\text{TB} / 4\text{KB} = 256\text{M}$  blokken. Indien elke verwijzing 32 bits in beslag neemt, dan is de grootte van de FAT  $256\text{M} \times 4\text{B} = 1\text{GB}$ .

## File allocation table (FAT)

- 👍 Betere leessnelheid dan linked lists
- 👍 Ondersteunt random access
- 👎 FAT kan erg veel RAM in beslag nemen

**HO  
GENT**

## Index nodes (inodes)



**HO  
GENT**

Een **index node (inode)** is een datastructuur die zowel de metadata van een bestand als verwijzingen naar de datablokken bevat. Een bestandssysteem dat inodes gebruikt, hoeft enkel de inodes van de geopende bestanden in het RAM te bewaren. Deze techniek combineert dus een **goede leessnelheid** met een **beperkt RAM-verbruik**.

## Index nodes (inodes)

- 👍 Goede leessnelheid
- 👍 Ondersteunt random access
- 👍 Beperkt RAM-verbruik

**HO  
GENT**



## Implementatie van directories

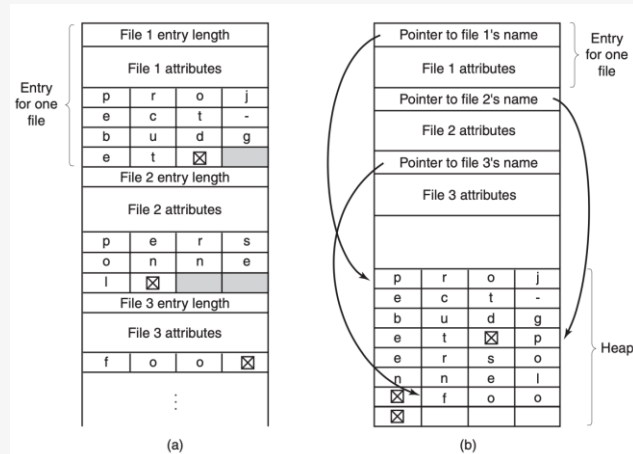
- Een directory kan opgeslagen worden in een bestand
- Dit bestand bevat één entry per file of subdirectory
- Een entry bevat o.a. een verwijzing naar het eerste datablok, of de inode, van de file of subdirectory

**HO  
GENT**

Een directory kan opgeslagen worden in een bestand. Dit bestand bevat dan een **directory entry** voor elke file of subdirectory.

Elk file system kiest zelf welke informatie er wordt opgeslagen in een directory entry, maar deze informatie omvat zeker een verwijzing naar het eerste datablok (bij gebruik van contiguous storage, linked lists, of FAT) of naar de inode (bij gebruik van inodes).

# Implementatie van directories



**HO  
GENT**

Een belangrijke keuze die moet worden gemaakt is hoe/waar de naam van elke file of subdirectory wordt opgeslagen. In Figuur (a) wordt de naam opgeslagen als deel van de entry. In Figuur (b) worden de namen apart opgeslagen in een heap.

Versie (a) heeft als nadeel dat directory entries een variabele grootte hebben, en er dus fragmentatie kan ontstaan binnen het directory-bestand. Versie (b) heeft dan weer als nadeel dat er een heap moet beheerd worden.

## Implementatie van links

- De meeste file systems ondersteunen **links**
- Bij een **hard link** worden datablokken of inodes gedeeld
- Een **soft link** heeft een eigen datablok of inode

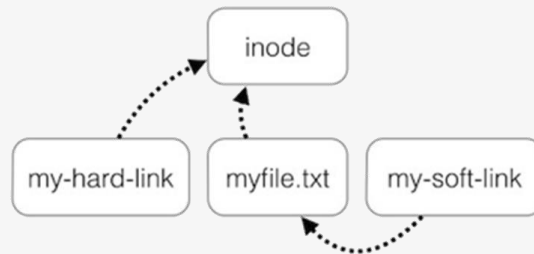
**HO  
GENT**

Een **link** is een bestand dat verwijst naar - of gekoppeld is aan - een ander bestand. Links bestaan in twee vormen:

- Een **hard link** is een koppeling die wordt gecreëerd door dezelfde datablokken of inodes in meerdere directory entries in te schrijven. Deze entries blijven onafhankelijk maar delen wel dezelfde data.
- Een **soft link** (ook gekend als symbolic link of snelkoppeling) heeft een eigen datablok of inode. Dit soort link is een bestand waarvan de inhoud bestaat uit een verwijzing naar een ander bestand.

Soft links zijn flexibeler dan hard links omdat ze ook werken tussen verschillende bestandssystemen, terwijl hard links enkel mogelijk zijn binnen hetzelfde bestandssysteem. Een nadeel van soft links is dan weer dat ze kunnen leiden tot **dangling pointers**. Een dangling pointer is een soft link die verwijst naar een bestand dat niet meer bestaat.

## Implementatie van links



**HO  
GENT**

In dit voorbeeld delen **my-hard-link** en **myfile.txt** dezelfde inode. **my-soft-link** verwijst rechtstreeks naar het bestand **myfile.txt**.

## Journaling

- Een crash tijdens een schrijfbewerking kan leiden tot corrupte of verloren data
- Het bijhouden van een journal (logboek van bewerkingen) kan hier tegen beschermen

**HO  
GENT**

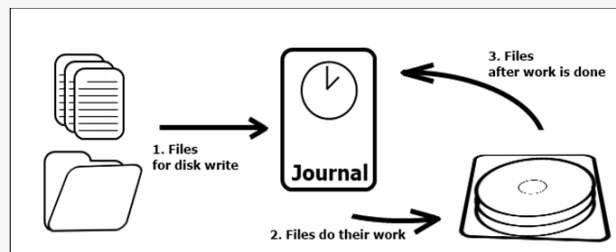
Wanneer het bestandssysteem of besturingssysteem crasht tijdens een schrijfbewerking, dan kan er data corrupt worden of verloren gaan. Een schrijfbewerking kan namelijk uit verschillende stappen bestaan, die allemaal moeten uitgevoerd worden. Zo komt het verwijderen van een file neer op:

1. Verwijder de directory entry voor deze file.
2. Verwijder de inode voor deze file.
3. Markeer de datablokken van deze file als vrije ruimte.

Het onderbreken van deze stappen kan het bestandssysteem in een ongeldige toestand brengen.

Door het bijhouden van een **journal** (logboek van bewerkingen) kan het bestandssysteem zichzelf herstellen na een crash.

## Journaling



**HO  
GENT**

Een bestandssysteem met journaling werkt als volgt:

1. Elke uit te voeren bewerking wordt eerst neergeschreven in het logboek.
2. Vervolgens wordt de bewerking zelf uitgevoerd.
3. Tenslotte wordt in het logboek de bewerking gemarkeerd als voltooid.

Wanneer het bestandssysteem of besturingssysteem crasht tijdens de uitvoering van de bewerking, dan kan het bestandssysteem deze crash detecteren omdat er een nog-niet-voltooid bewerking in het logboek staat. Vervolgens kan het bestandssysteem de onvoltooide bewerking alsnog voltooien op basis van de informatie in het logboek.

## Virtual file systems

- Op **Windows** krijgt elk bestandssysteem een **drive letter** toegewezen (C, D, ...)
- Linux en Mac hebben een **virtual file system** dat alle bestandssystemen presenteert als één hiërarchische structuur

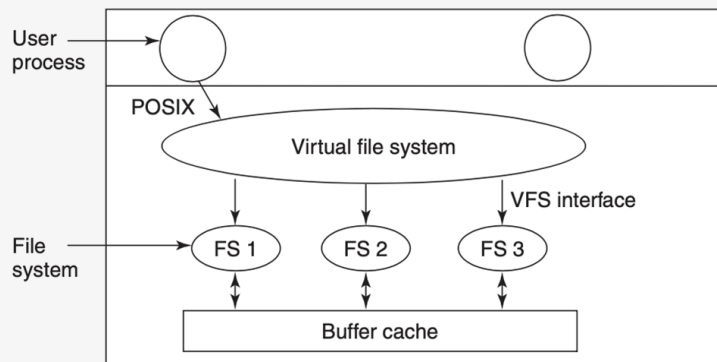
**HO  
GENT**

Zoals je al weet, is er een belangrijk verschil in hoe Windows, Linux, en Mac omgaan met bestandssystemen.

Op Windows krijgt elk bestandssysteem een **drive letter** toegewezen. Zo spreek je bvb. over het bestand **C:\Users\hogent\file.txt**, dat zich bevindt op “de C schijf” (het bestandssysteem van deze schijf).

Op Linux en Mac daarentegen is er een **virtual file system**. Deze systemen brengen alle actieve bestandssystemen samen onder één hiërarchische structuur. Voor de gebruiker lijkt het alsof er slechts één bestandssysteem is.

## Virtual file systems



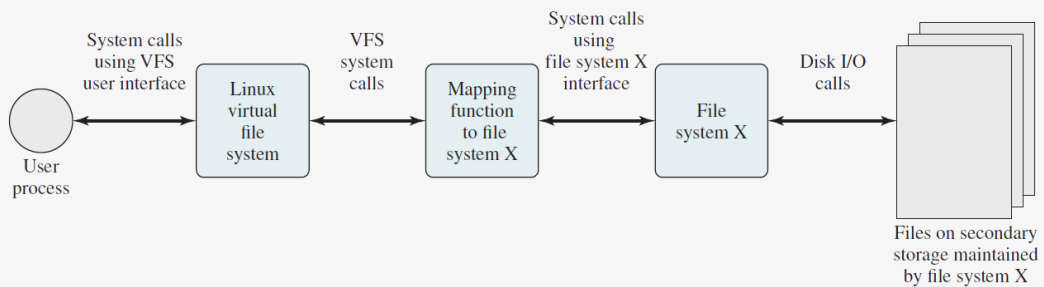
Deze afbeelding toont aan hoe een virtual file system werkt:

- Processen gebruiken de algemene functies van het besturingssysteem om het virtual file system aan te spreken. Deze functies (op de figuur aangeduid als POSIX) zijn dus niet gebonden aan een specifiek file system.
- Het virtual file systeem vertaalt deze algemene functieoproepen naar oproepen voor de drivers van elk file system.

Een virtual file system verbergt dus de verschillen tussen de file systems. Processen zien slechts één bestandsstructuur, die op een uniforme manier kan worden bewerkt.



## Virtual file systems



**HO  
GENT**

Deze afbeelding toont nogmaals aan hoe een virtual file system werkt:

- Processen gebruiken de algemene functies van het besturingssysteem om het virtual file system aan te spreken. Deze functies (op de figuur: pijl tussen User process en Linux VFS) zijn dus niet gebonden aan een specifiek file system.
- Het virtual file systeem vertaalt deze algemene functieoproepen naar oproepen voor de drivers van elk file system.

Een virtual file system verbergt dus de verschillen tussen de file systems. Processen zien slechts één bestandsstructuur, die op een uniforme manier kan worden bewerkt.

## Virtual file systems

- Het inladen van een bestandssysteem binnen de VFS-hiërarchie heet een **mount** bewerking
- Het uitladen van een bestandssysteem heet een **unmount**
- Bij het mounten geef je aan in welke map van de VFS-hiërarchie je de root directory van het bestandssysteem wil inladen

**HO  
GENT**

Twee belangrijke bewerkingen bij een virtual file system zijn:

- **mount:** het inladen van de root directory van een bestandssysteem in een directory van de virtuele hiërarchie. Zo kan je bvb. kiezen om de root directory van een USB-stick in te laden in de map **/media/usb**. Vanaf dan verwijst deze map naar het bestandssysteem van de USB-stick.
- **unmount:** de omgekeerde bewerking. Een reeds ingeladen bestandssysteem terug loskoppelen van het virtuele bestandssysteem.

## **5.5 Partities**

**HO  
GENT**

## Partities

- Een fysiek opslagmedium kan zijn capaciteit onderverdelen in **partities**
- Elke partitie heeft een eigen bestandssysteem
- Het opslagmedium voorziet dan ook een **partitietabel**, bvb:
  - Master Boot Record (MBR)
  - GUID Partition Table (GPT)
- Deze tabel bevat informatie over de partities en hun bestandssystemen

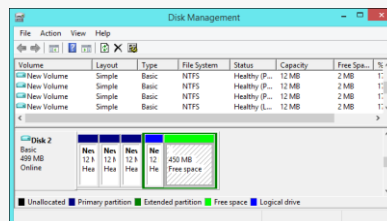
**HO  
GENT**

Een fysiek opslagmedium (zoals een harde schijf) kan zijn capaciteit onderverdelen in **partities**. Elke partitie heeft dan een eigen bestandssysteem.

Een opslagmedium met partities moet ook een **partitietabel** voorzien. Deze tabel beschrijft de partities en hun bestandssystemen. Twee systemen die hiervoor worden gebruikt zijn een **Master Boot Record (MBR)** of een **GUID Partition Table (GPT)**.

## MBR vs GPT

- MBR: 1983 (DOS 2.0), belangrijke beperkingen:
  - Maximum grootte schijf: 2TB
  - Maximum 4 primaire partities
    - Workaround: Extended partitie maken die logische partities bevat
- GPT (1998): opvolger voor MBR
  - Elke partitie heeft (unieke) GUID
  - Ondersteuning voor schijven > 2TB
  - Theoretisch onbeperkt aantal partities (Windows max. 128)
  - Boot support voor Windows enkel op 64-bit systemen met UEFI.
  - Boot support voor Linux ook op BIOS firmware.



**HO  
GENT**

We kunnen dus een schijf partitioneren met behulp van de Master Boot Record (MBR) of een GUID Partition Table (GPT). Elke methode heeft zijn voor- en nadelen.

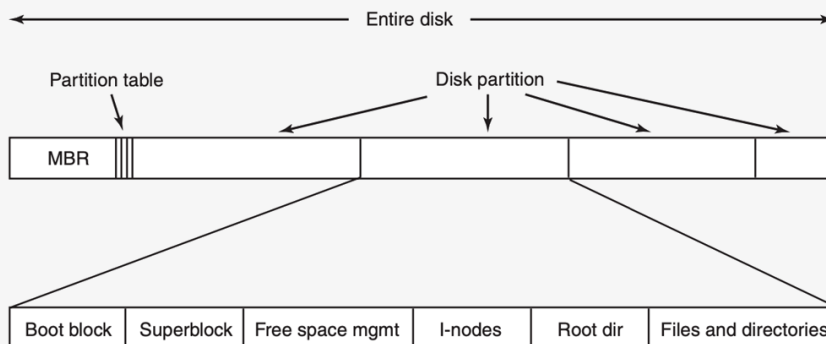
MBR is de oude manier van werken, en bestaat al sinds 1983 (geïntroduceerd als deel van PC DOS 2.0). Bij MBR wordt een speciale boot sector gebruikt aan het begin van de schijf om de partitietabel op te slaan. MBR heeft echter 2 belangrijke beperkingen: je kan MBR enkel gebruiken bij schijven van maximum 2 TB in grootte (terwijl er vandaag al veel schijven van 4, 6, 8 of meer TB te verkrijgen zijn), en je kan maximum 4 (primaire) partities hebben op 1 schijf. Dit laatste kan je wel omzeilen door gebruik te maken van extended partities, dit zijn een soort van virtuele partitie waarin je meerdere logische partities kan maken – zie ook de screenshot op de slide – maar dit maakt het schijfbeheer uiteraard onnodig complex.

GPT is de opvolger van MBR, en is een onderdeel van UEFI. GPT biedt ondersteuning voor schijven groter dan 2TB, en binnen GPT kan je in theorie een bijna onbeperkt aantal partities hebben (maar in Windows is het aantal partities wel beperkt tot 128). Bij GPT krijgt elke partitie een unieke ID, de Globally Unique Identifier (GUID), die opgeslagen wordt in de GUID partition table (GPT). Als je dus een schijf hebt van meer dan 2TB ben je verplicht om GPT te gebruiken, maar een kleine beperking is dat

Windows enkel bootable is vanop een schijf met GPT voor 64-bit systemen die gebruikmaken van UEFI (opvolger van BIOS) als interface tussen de hardware en het besturingssysteem. Het is wel mogelijk om Linux te starten vanop een schijf met GPT in combinatie met BIOS firmware.

## 5.5 Partities

# MBR



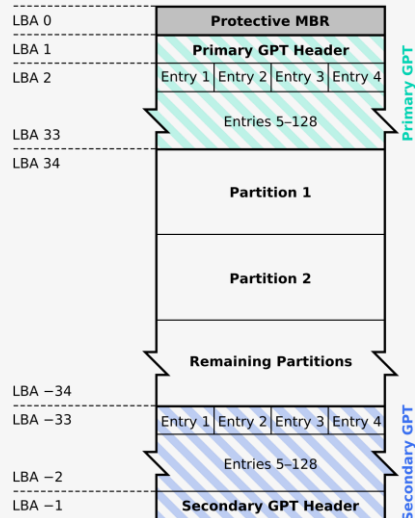
**HO  
GENT**

Dit voorbeeld toont een schijf die onderverdeeld is in partities. Aan het begin van de schijf zie je de Master Boot Record met de partitietabel. Daarna volgen de verschillende partities.

Een Master Boot Record bevat naast de partitietabel ook een zogenaamde **boot loader**, die verantwoordelijk is voor het opstarten van het systeem. Deze boot loader gaat op zoek naar een partitie met een besturingssysteem en schakelt dan door naar de code in het **boot block** van deze partitie.

# GPT

## GUID Partition Table Scheme



HO  
GENT

Dit voorbeeld maakt gebruik van een GUID Partition Table (GPT). Deze GPT is onderdeel van de UEFI standaard, wat het verouderde BIOS vervangt. GPT is achterwaarts compatibel met MBR en kan ook worden gebruikt in combinatie met een BIOS. UEFI is dus niet verplicht, behalve als je Windows wil booten vanop een schijf met GPT.

GPT plaatst de boot loader op een speciale **EFI System Partition**. Uiteraard kan er via de achterwaartse compatibiliteit met MBR nog steeds gebruik worden gemaakt van boot blocks.



## **5.6 Booten**

**HO  
GENT**

## Booten

= opstarten van de computer en inladen operating system

- Nood aan apart programma opgeslagen in een bootable partitie  
= **bootloader**
- Instellen volgorde bootable partities in de **BIOS**
- Bootloader is meestal OS afhankelijk
  - Linux: grub, lilo, rEFInd, ...
  - Mac: BootX
  - Windows: Windows Boot manager

**HO  
GENT**

Een besturingssysteem op een computersysteem start niet vanzelf op. Het proces om de computer op te starten en een besturingssysteem te laden, wordt booten genoemd. Om een besturingssysteem te laden bij het opstarten van de computer, hebben we een apart programma nodig, namelijk de bootloader. De bootloader bevindt zich in een opstartbare (of bootable) partitie van het secundaire geheugen, zoals een HDD of SSD.

Als er meerdere bootable partities aanwezig zijn, kunnen we in de BIOS instellen in welke volgorde de partities worden overlopen. Bij het booten wordt de eerst gevonden bootloader op een bootable partitie uitgevoerd.

Een bootloader is meestal afhankelijk van het besturingssysteem en specifiek ontworpen om alleen dat type besturingssysteem te laden. Enkele voorbeelden van bekende bootloaders per besturingssysteem zijn:

- Linux: grub, lilo, rEFInd, ...
- macOS: BootX
- Windows: Windows Boot manager



## Bootloader

- Werking bootloader:
  1. Uitpakken gecomprimeerde bestanden op bootpartitie
  2. Inladen kernel in het geheugen
  3. Inladen root file system in het geheugen
  4. Controle doorgeven aan kernel om OS verder in te laden en te starten
- Elke kernel update OS => updaten files bootpartitie  
Zet je computer dus nooit uit tijdens OS update!
- Corrupte bootpartitie en/of files = onmogelijk om OS te laden

**HO  
GENT**

Maar hoe werkt een bootloader nu precies?

In grote lijnen voert de bootloader tijdens het booten vier stappen uit:

1. Het uitpakken van gecomprimeerde bestanden op de bootpartitie. De opstartbare bestanden bevinden zich niet rechtstreeks op de partitie van het besturingssysteem zelf, maar in een aparte afgeschermd partitie: de bootpartitie. De kernel en het root file system worden gecomprimeerd om plaats te besparen omdat de bootpartitie standaard klein is (bijvoorbeeld de Windows bootpartitie is standaard maar 128 MB).
2. Het inladen van de kernel in het geheugen.
3. Het inladen van het root file system in het geheugen.
4. Het doorgeven van de controle aan de kernel om het besturingssysteem verder te laden en te starten.

Telkens wanneer het besturingssysteem de kernel bijwerkt naar een nieuwe versie, worden er nieuwe bootbestanden gegenereerd. Zet daarom nooit de computer uit tijdens het uitvoeren van een systeemupdate. Hierdoor kunnen de bootbestanden beschadigd raken, waardoor het besturingssysteem niet meer opstart.

## Multi-Boot

- Meerdere OS op eenzelfde computer
  - OS in juist volgorde installeren want voorgaande bootloader wordt overschreven
  - B.v. bij Linux en Windows samen
    - Eerst Windows installeren
    - Dan Linux installeren
    - Linux bootloader laten doorverwijzen Windows Boot Manager
- Chain loaden bootloaders

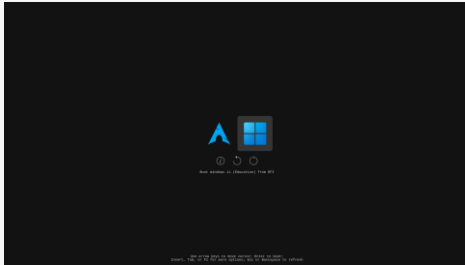
**HO  
GENT**

Heel wat mensen hebben soms nood aan een extra OS op hun computer (bv. voor OS specifieke software) en installeren daarom meer dan één OS op hun computersysteem, dit wordt multi-boot genoemd. Om ervoor te zorgen dat je elk OS afzonderlijk zonder problemen kan starten, heb je dus een bootloader nodig die elk OS kan starten. Aangezien bij de installatie van een ander OS meestal de huidige bootloader zal overschreven worden, is het van groot belang om elk OS in de juiste volgorde te installeren.

Wil je graag Linux en Windows samen op een toestel installeren dan is het aan te raden om eerst Windows te installeren en pas nadien Linux. Na installatie van beide kan je de linux bootloader (b.v. rEFInd) laten door verwijzen naar de bootloader van Windows. Dit principe heet **chain loaden** van bootloaders. Windows heeft nl. geen boot menu en zal dus steeds meteen Windows opstarten i.p.v. de keuze te geven (GRUB kan dit wel).

## Voorbeeld boot loader (Multi-boot)

### rEFInd Boot Loader



```
menuentry "Arch Linux" {
    volume "Arch Linux"
    loader /EFI/ARCH/vmlinuz-linux
    initrd /EFI/ARCH/initramfs-linux.img
    options "root=UUID=6287d387-be27-4e98-b453-6fb0ea3154fa rw
add_efi_memmap initrd=\EFI\ARCH\intel-ucode.img"
    submenuentry "Boot using fallback initramfs" {
        initrd /EFI/ARCH/initramfs-linux-fallback.img
    }
}

menuentry "Windows 11 (Education)" {
    loader \EFI\Microsoft\Boot\bootmgfw.efi
}
```

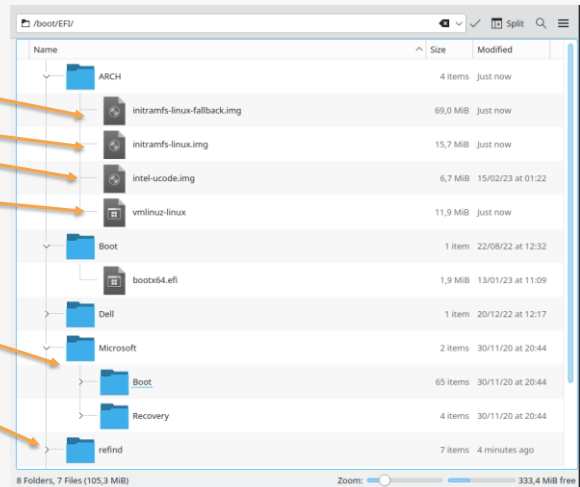
**HO  
GENT**

Voorbeeld van Multi-boot Arch Linux en Windows 11 Met rEFInd als bootloader.

Arch Linux kan rechtstreeks worden gestart en als extra optie kun je ook "fallback" starten als er een probleem is bij het normale opstarten. Dit is een back-up optie om toch je systeem te kunnen starten. Om Windows 11 te kunnen starten, moeten we zijn bootloader chain loaden.

## Voorbeeld bootpartitie (Multi-boot)

- Arch Linux Boot files
  - Root file system (Back-up)
  - Root file system
  - Intel CPU patch (Optioneel)
  - Kernel
- Windows Boot Files
- Bootloader (rEFInd)



## **5.7 Voorbeelden FS**

**HO  
GENT**



## Voorbeelden FS

- Windows:
  - NTFS
  - FAT32
  - exFAT
- Mac:
  - HFS+
  - APFS
- Linux:
  - ext4
  - ZFS
  - Btrfs
- Andere:
  - ISO 9660 / UDF

**HO  
GENT**

De volgende slides bespreken enkele voorbeelden van courante file systems op Windows, Mac, en Linux.

## NTFS

- New Technology File System
- Standaard file system op **Windows**
  - Ook bruikbaar op Mac en Linux via de NTFS-3G driver
- Gebruikt journaling
- Vervangt oudere file systems zoals FAT32

**HO  
GENT**

**NTFS** (*New Technology File System*) is het standaard file system op Windows. Dit file system gebruikt journaling en vervangt het oudere FAT32 file system.

NTFS is een propriëtair file system van Microsoft, met beperkte ondersteuning op macOS en Linux. Op deze platformen gebruik je best de open source **NTFS-3G** driver, die het mogelijk maakt om NTFS partities te lezen en te schrijven.

## FAT32

- File Allocation Table
- Ontwikkeld door Microsoft maar werkt op verschillende OS
- Opvolger FAT12 en FAT16
- 32 bit voor adressering clusters ( $0 - 2^{32}-1$ )
  - Maximum bestandsgrootte: 4GB
  - Maximum grootte bestandssysteem: 2TB

**HO  
GENT**

**FAT32** is een bestandssysteem dat gebruikmaakt van een 32 bits File Allocation Table. FAT32 is ontwikkeld door Microsoft als opvolger voor FAT12 en FAT16. Het nummer verwijst naar het aantal bits dat gebruikt wordt voor de adressering: bij FAT32 worden 32 bits gebruikt om de clusters te adresseren. Hierdoor heeft FAT32 wel enkele beperkingen:

- De maximum grootte van het bestandssysteem is 2TB, maar tegenwoordig bestaan er al harde schijven van 4, 8 of 16TB
- De maximum grootte voor een bestand is 4GB, terwijl bestanden van >4GB tegenwoordig vrij courant zijn

Door deze beperkingen wordt FAT32 tegenwoordig nog weinig gebruikt. De opvolger is **exFAT** (zie volgende slide).

## exFAT

- Extensible File Allocation Table
- Ontwikkeld door Microsoft
  - Goede ondersteuning op Mac en Linux
- Gericht op USB-sticks en SD-kaarten
  - Vervangt FAT32 voor deze toepassingen
- Maakt gebruik van een file allocation table
- Gebruikt geen journaling

**HO  
GENT**

Ook **exFAT** (*Extensible File Allocation Table*) is een propriëtair file system van Microsoft. Zoals je uit de naam kan afleiden, gebruikt dit file system een file allocation table. In tegenstelling tot NTFS gebruikt exFAT geen journaling — dit om schijfruimte te besparen. exFAT is namelijk gericht op verwijderbare media zoals USB-sticks en SD-kaarten.

exFAT wordt goed ondersteund op Mac en Linux, wat het uitermate geschikt maakt voor bestandsuitwisseling.

## HFS+

- Hierarchical File System Plus
- Was het standaard file system op Mac tot enkele jaren terug
  - Beperkte ondersteuning op Windows en Linux
- Gebruikt journaling
- Vervangen door APFS en niet langer aan te raden

**HO  
GENT**

**HFS+** (*Hierarchical File System Plus, of MacOS Extended*) was het standaard file system op macOS tot enkele jaren terug. Dit file system heeft erg lang dienst gedaan maar is ondertussen volledig vervangen door het nieuwere APFS.

## APFS

- Apple File System
- Standaard file system op **Mac**
  - Beperkte ondersteuning op Windows en Linux
- Sterke focus op SSDs en encryptie
- Gebruikt GPT partitionering met containers en volumes

**HO  
GENT**

**APFS** (*Apple File System*) is het nieuwste file system voor macOS, iOS, en aanverwanten. Dit file system heeft een grote focus op SSDs en encryptie, en gebruikt GPT containers en volumes i.p.v. klassieke partities.

APFS is hoofdzakelijk een Apple-product en heeft slechts een beperkte ondersteuning op Windows en Linux, via third-party (commerciële) drivers.

## ext4

- Fourth extended file system
- Standaard file system op vele Linux-distributies
  - Beperkte ondersteuning op Windows en Mac
- Gebruikt journaling

HO  
GENT

**ext4** (*fourth extended file system*) is het standaard file system op vele Linux-distributies. Dit file system gebruikt journaling en is achterwaarts compatibel met zijn voorgangers **ext2** en **ext3**.

ext4 wordt bijna uitsluitend gebruikt op Linux en heeft slechts beperkte ondersteuning op Windows en Mac, via third-party (commerciële) drivers.

## ZFS

- Zettabyte File System
- Ontwikkeld door Sun Microsystems voor Solaris
- Populair op Linux en FreeBSD
- Heel geavanceerd: volume management, snapshots maken, klonen, integriteitscontrole, caching, ...
- Niet zo flexibel als andere file systems
- Heel sterk tegen bitrot en data corruptie

**HO  
GENT**

**ZFS** (*Zettabyte File System*) is een file system oorspronkelijk ontwikkeld door Sun Microsystems voor hun Solaris besturingssysteem.

ZFS is uitermate krachtig en complex, en voornamelijk gericht op intensieve servertoepassingen. Het is daarom erg populair op Linux en FreeBSD.



## Btrfs

- B-tree file system / Butter FS / Better file system
- Ontwikkeld door Oracle
- Antwoord op ZFS
- Standaard file system op **Fedora**
- Sterk tegen bitrot en datacorruptie

**HO  
GENT**

**Btrfs** (*B-tree File System of Butter FS of Better File System*) is een file system ontwikkeld door Oracle, als antwoord op ZFS. Net als ZFS is Btrfs een geavanceerd file system, met een pak meer features dan ext4.

Btrfs is sinds kort ook het standaard file system op Fedora.

## ISO 9660 / UDF

- File systems voor **optische schijven** (resp. CD en DVD)
- Vooral gericht op write-once media
- Ondersteund op Windows, Mac, en Linux

**HO  
GENT**

**ISO 9660** en **UDF** (*Universal Disk Format*) zijn file systems voor optische schijven (resp. CD en DVD). Deze file systems verschillen heel erg van de vorige omdat ze gericht zijn op read-only of write-once media voor distributie of backup.

Beide file systems zijn uiteraard ondersteund op alle gangbare platformen.

## **5.8 Opslag in Docker**

**HO  
GENT**

## VM vs container

- Een virtuele machine bevat één of meerdere **virtuele disks**
  - Typisch een bestand, opgeslagen op de fysieke schijf van de host  
bv. in Virtualbox bestand in VDI-formaat met extensie .vdi
  - Deze virtuele disk stelt een virtueel blokapparaat voor
  - Een virtuele disk heeft eigen partitietabel, en elke partitie heeft eigen file system
- Een docker container heeft geen virtuele disk
  - Bestanden in Docker container worden opgeslagen in **writable container layer**
  - Beheer van deze layer gebeurt via een **storage driver**

**HO  
GENT**

### VM vs container

Zoals we reeds gezien hebben in H2, bevat een virtuele machine één of meerdere virtuele disks. Dit zijn bestanden die opgeslagen worden op de fysieke harde schijf van de host. In VirtualBox zijn dit bijvoorbeeld bestanden in VDI-formaat (met extensie .vdi) en default worden deze opgeslagen in de folder `C:\users\<username>\VirtualBox VMs` (wanneer je VirtualBox gebruikt in Windows). Hyper-V gebruikt het VHDX-formaat en VMWare gebruikt typisch het VMDK-formaat voor de opslag van de virtuele harde schijf.

Een virtuele harde schijf (virtual disk) stelt een (virtueel) blokapparaat voor. Een virtuele harde schijf heeft dus zijn eigen partitietabel, en elke partitie heeft zijn eigen file system. Eén .vdi-bestand kan dus meerdere partities bevatten (zie ook Labo 8).

Docker werkt echter iets anders: een docker container heeft geen virtuele disks, maar bestanden in een Docker container worden opgeslagen in een writable container layer. Het beheer van deze layer gebeurt aan de hand van een storage driver. Op de volgende slides gaan we iets dieper in op de werking van deze layers.

## Storage driver en layers

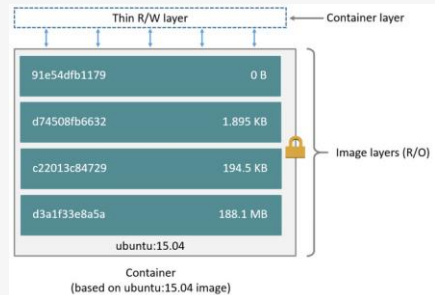
Een docker container bestaat typisch uit:

- Meerdere **image layers** (readonly)
- Eén schrijfbare **container layer** (writeable)

De onderste layer is de basis image, en elke layer houdt wijzigingen bij ten opzichte van de onderliggende layer.

Bij aanmaken van een nieuwe container maak je een nieuwe writable layer, vaak de "**container layer**" genoemd. Dit is de enige layer waarin de container wijzigingen kan wegschrijven.

De **storage driver** is verantwoordelijk voor het beheer van deze layers.



**HO  
GENT**

### Storage driver en layer

Containers in Docker gebruiken dus geen virtuele harde schijven, maar maken gebruik van meerdere layers (lagen), en enkel de bovenste layer is schrijfbaar. Concreet bestaat een docker container uit:

- Meerdere image layers, deze zijn readonly en worden aangemaakt tijdens het bouwen van een image
- Eén schrijfbare container layer

De onderste image layer is de basis-image van de container (bv: een ubuntu:15-04 image), en elke layer daarboven houdt de wijzigingen bij ten opzichte van de onderliggende layer. In de volgende slide zullen we de werking van deze image layers illustreren aan de hand van een klein voorbeeld.

Bij het aanmaken van een container (op basis van een image) maak je een nieuwe layer aan bovenop de image layers, deze container wordt vaak de **container layer** genoemd. Dit is de enige layer waarin de container (tijdens de uitvoering) wijzigingen kan wegschrijven, alle andere image layers zijn readonly.

De storage driver is binnen docker verantwoordelijk voor het beheer van de verschillende layers.

## Voorbeeld image layers

- Het **FROM** statement maakt een layer aan op basis van de `ubuntu:18.04` image
- Het **COPY** commando kopieert enkele bestanden naar de container, en slaat dit op in een nieuwe layer
- Het eerste **RUN** commando compileert de applicatie en schrijft het resultaat naar een nieuwe layer
- Het tweede **RUN** commando verwijdert tijdelijke bestanden, en schrijft het resultaat naar een nieuwe layer
- De **CMD** instructie zegt wat de container moet doen bij opstarten, dit voegt geen nieuwe image layer toe (maar wijzigt de metadata van de image)

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```



4 image layers!

**HO  
GENT**

### Voorbeeld image layers

Als voorbeeld nemen we de volgende Dockerfile:

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

Als we deze Dockerfile builden maken we een image aan, die uiteindelijk zal bestaan uit 4 image layers:

- Het eerste **FROM** statement maakt een nieuwe image layer aan op basis van de `ubuntu:18.04` image.
- Het **COPY** commando kopieert enkele bestanden naar de container, en slaat dit op in een nieuwe image layer.
- Het eerste **RUN** commando compileert de applicatie (via een `make` commando), en schrijft het resultaat naar een nieuwe image layer.

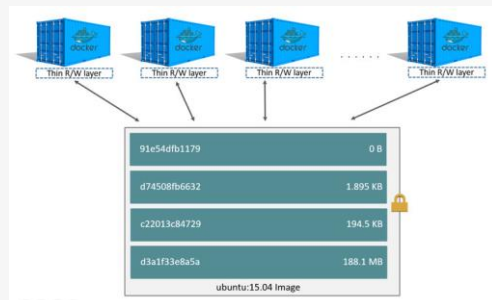
- Het tweede RUN commando verwijdert enkele tijdelijke bestanden, en schrijft het resultaat naar een nieuwe image layer.

De CMD instructie op de laatste regel zegt wat de container moet doen bij opstarten. Dit maakt geen nieuwe image layer aan, maar wijzigt enkel de metadata van de image.



## Container Layer

- Bij aanmaken van een container maak je een **writable layer** bovenop de image layers
  - Deze layer noemen we de **container layer** (= thin R/W layer)
  - Alle **wijzigingen** van **data** tijdens **uitvoeren** van de container worden bijgehouden in deze container layer
- Bij verwijderen container wordt de container layer verwijderd
  - Onderliggende image layers worden niet verwijderd!
- Meerdere containers kunnen dezelfde onderliggende image (layers) gebruiken
  - Elke container houdt wijzigingen ten opzichte van image bij in eigen container layer



### Container Layer

Bij het aanmaken van een container op basis van een gecompileerde image, maak je een writable layer aan bovenop de image layers. Deze layer noemen we de **container layer** of **Thin R/W layer**. Als tijdens de uitvoering van de container data gewijzigd wordt (= aanmaken, wijzigen of verwijderen van mappen en bestanden) wordt dit bijgehouden in deze container layer. De container layer is de enige layer die de container kan aanpassen, de image layers zijn readonly – de container kan deze enkel lezen en dus niet aanpassen.

Wanneer we een container verwijderen, dan wordt de container layer verwijderd. De onderliggende image layers worden echter **niet verwijderd**, het kan immers zijn dat andere containers dezelfde image gebruiken. Enkel bij verwijderen van de image zullen de image layers verwijderd worden.

Meerdere containers kunnen dus dezelfde onderliggende image layers gebruiken, en elke container houdt wijzigingen ten opzichte van de image bij in hun eigen container layer (zie ook figuur rechtsonder).

## Persistente opslag in Docker

- Aangezien de container layer verwijderd wordt bij verwijderen container, is dit niet aangeraden voor persistente opslag (bv. bewaren data in databank).
- De layers worden welliswaar “ergens” opgeslagen op het fysieke filesystem van de Linux host, maar het is niet de bedoeling om bestanden en mappen in deze layers manueel te wijzigen vanaf de host.
- Voor persistente opslag is het dus beter om gebruik te maken van **volumes** en **bind mounts** (zie H2.3 - Docker).
  - Volumes zijn nuttig om data te delen tussen verschillende containers
  - Bind mounts zijn nuttig als je vanaf de host ook toegang wil tot de data

**HO  
GENT**

### Persistente opslag in Docker

Aangezien de container layer dus verwijderd wordt bij het verwijderen van een container, is dit geen optimale oplossing voor persistente opslag. Als je bijvoorbeeld een container gebruikt voor het hosten van een databank, kan het handig zijn om nog aan de records (data) van de databank te kunnen wanneer de container niet meer bestaat.

De verschillende layers worden welliswaar “ergens” opgeslagen op het fysieke filesystem van de Linux host (hoe en waar is afhankelijk van de gebruikte storage driver), maar het is eigenlijk niet de bedoeling om manueel bestanden en mappen in deze layers te wijzigen vanaf de host. Dit zou immers de correcte werking van de storage driver in gevaar kunnen brengen, en containers (of images) corrupt maken.

Voor persistente opslag is het dus beter om gebruik te maken van **volumes** en **bind mounts**. Deze kwamen reeds aan bod in H2.3 – Docker. Even snel opfrissen:

- Volumes zijn vooral nuttig als je data wil delen tussen verschillende containers.
- Bind mounts zijn nuttig als je vanaf de host ook toegang wil tot de data. Bind mounts zijn eigenlijk een soort volumes die gemount worden op het virtual file

system van de Linux host waarop Docker geïnstalleerd is.

**HO  
GENT**