

Linux Theorie

Sessie 3



PC Hardware



PC Hardware

Het x86 platform

een operating system is de brug
tussen hardware en user programma's

even een klein blikje op die hardware..



PC Hardware

Power Supply

- **ATX** : meest voorkomende vormfactor voor PC-voeding



- Typische 24 pin's connector
- Geschakelde voeding
- Meerdere spanningen:
+3.3V +5V -5V +12V -12V
- Vermogen 250W tot 2000W

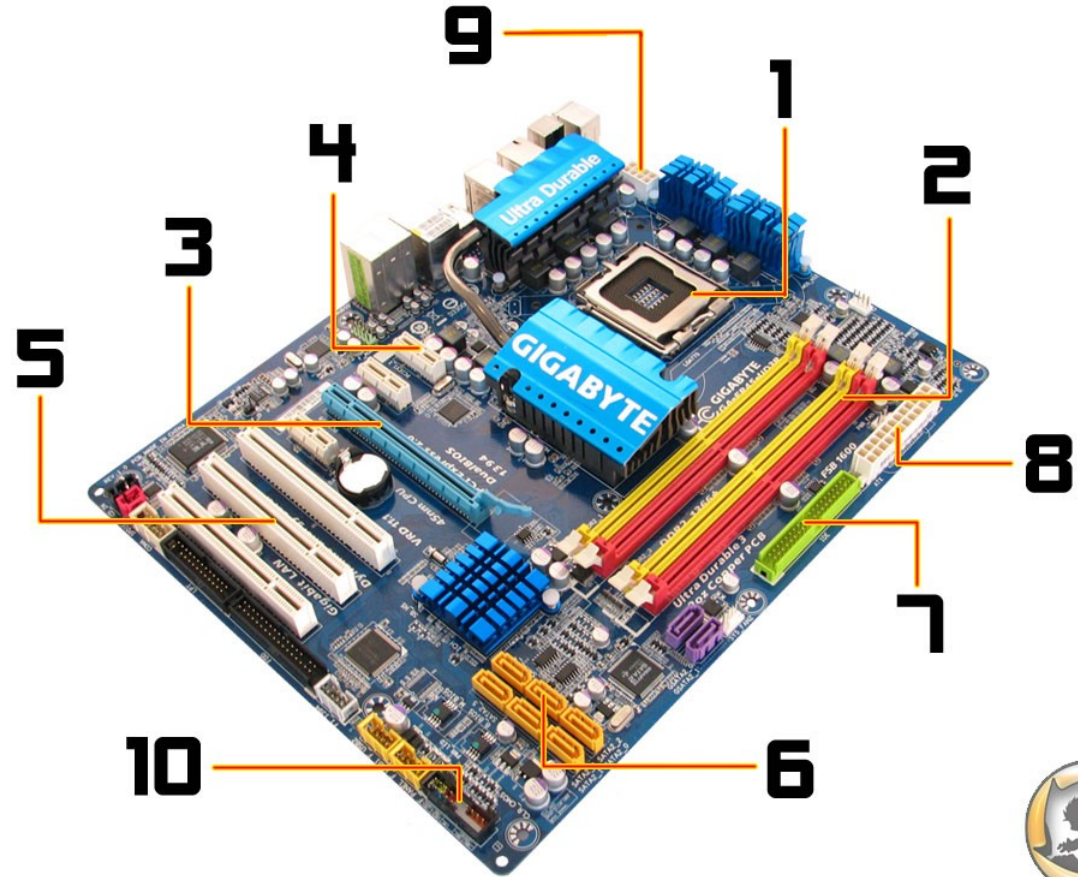
PC Hardware

motherboard

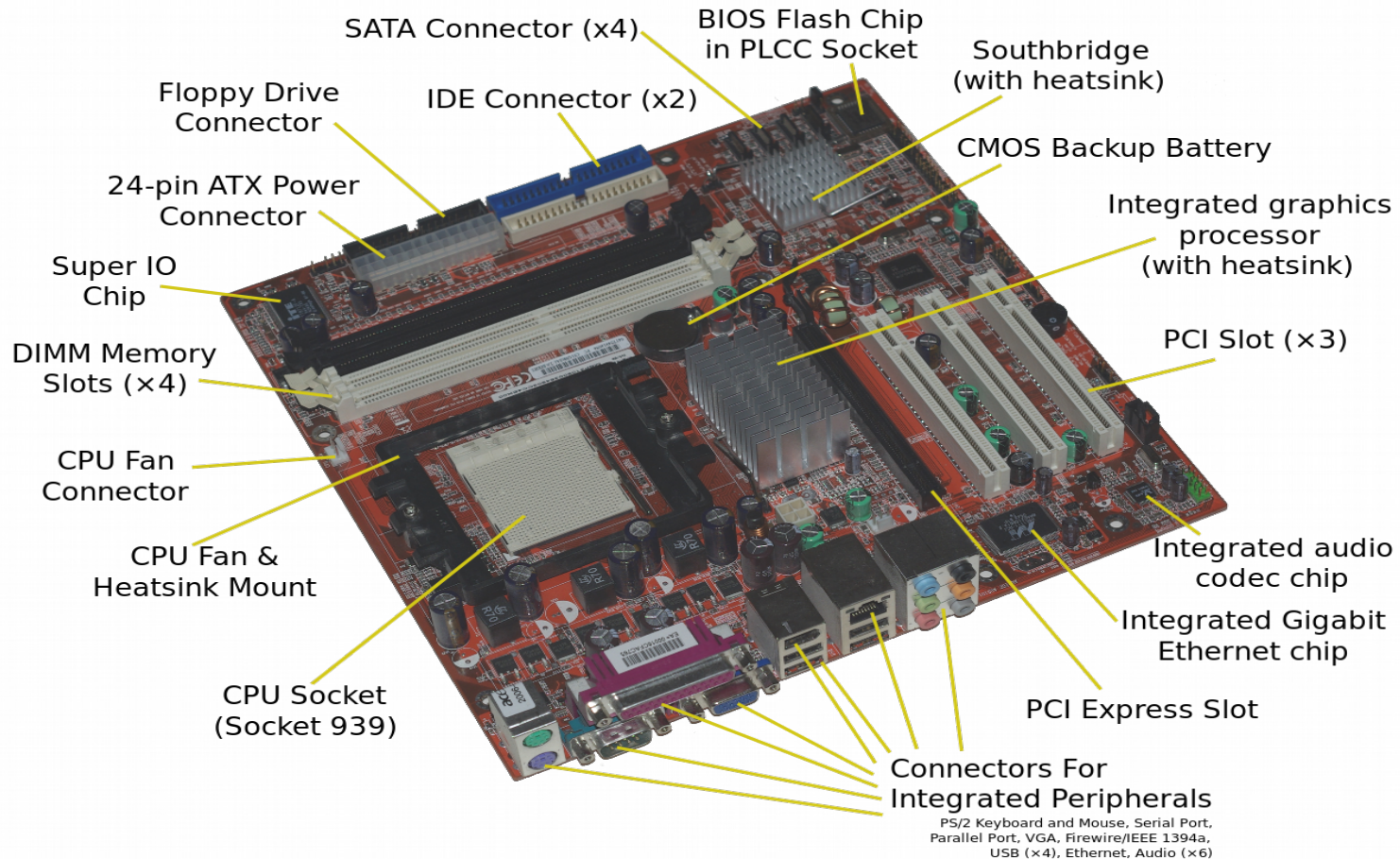


PC Hardware

1. cpu socket
2. memory slots
3. gpu slots
4. pci-e slots
5. pci slots
6. sata interfaces
7. hdd interfaces
- 8-9. power connector's
10. internal USB3.0



PC Hardware



PC Hardware

Storage interfaces

- **IDE** : “Parallel ATA” -
harddisk interface met een parallele databus
→ in de linux /dev directory zichtbaar als “hda, hdb, ..”
- **SATA** : “Serial ATA” - opvolger van IDE/ATA
harddisk interface met een seriële databus
→ in de linux /dev directory zichtbaar als “sda, sdb, ..”



PC Hardware

Storage interfaces

- **SATA :**
 - **SATA I** : 1,5 Gbit/s max snelheid
 - **SATA II** : 3 Gbit/s max snelheid
 - **SATA III** : 6 Gbit/s max snelheid (revisie 3.0)
16 Gbit/s max snelheid (revisie 3.2)
- **SCSI** : oudere seriële harddisk interface maar nog steeds gebruikt
 - in de /dev directory zichtbaar als “*sga, sgb, ..*”
of ook als “*sda, sdb, ..*”

PC Hardware

Storage interfaces

- **M.2:** “disk”-storage zijn steeds vaker SSD's
(SSD = Solid State Drive)
SSD's werden aanvankelijk enkel gemaakt
in dezelfde vormfactor als HDD's



Zo'n SSD kan veel kleiner gemaakt worden dan
een HDD vandaar de nieuwe aansluiting M.2

M.2 werkt ook met SATA maar kent eveneens
NVMe transfer technologie die nog wat sneller is.

(NVMe = Non-Volatile Memory express)



PC Hardware

Serial bus interfaces

- **USB** : “Universal Serial Bus”
 - in de linux /dev/usb directory “hiddev0, “hiddev1”
 - met het commando **lsusb** kan men alle devices zien
 - **USB 2.0** → 480 Mbit/s max snelheid
 - **USB 3.0** → 4 Gbit/s max snelheid
 - **USB 3.2** → 5-20 Gbit/s max snelheid (USB-C)

PC Hardware

Uitbreidings connectors

- **PCI** : “Peripheral Component Interconnect”
- **PCI-e** : “PCI Express”
 - terug te vinden de `/sys/bus/pci` directory
 - met het commando **lspci** kan men alle devices zien

PC Hardware

Memory connectors

- **DIMM** : “dual-in-line memory module”
 - in de linux /dev/mem directory



- DDR3 533-800 MT/s
- DDR4 1066-1600 MT/s
- DDR5 5200-6400 MT/s

6 oktober 2020
éérste DDR5 release

(MT = *MegaTransfer*)

PC Hardware

Processor

- in de linux `/dev/mem` & `/sys/bus/cpu` directories
- met het commando **lscpu** krijgt men info over de cpu



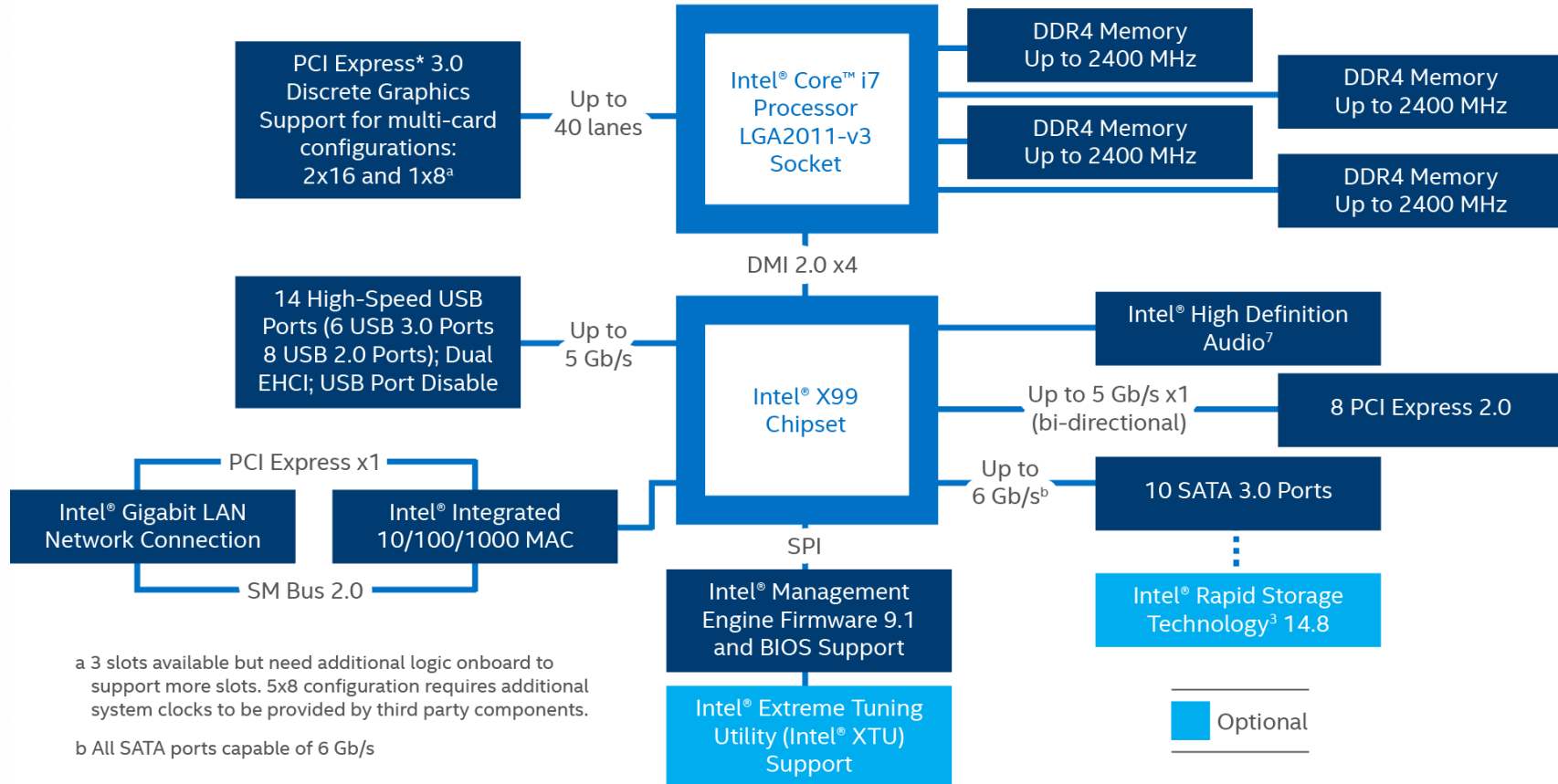
- (32) – 64 bit's
- Data bus
- Address bus
- Control bus

PC Hardware

BIOS Flash chip

- **BIOS** = “Basic Input Output System”
- Bevat de enige software die default in een PC te vinden is.
- Dit is software die typisch volgende dingen kan:
 - een grafische kaart in basic VGA mode aansturen
 - het keyboard kan lezen (geen layout keuze)
 - het lezen van een opstart medium kan initiëren
- De software in de BIOS-chip noemt men ook **de firmware**
(*alle bios is firmware niet alle firmware is bios...*)

PC Hardware



Linux Boot Process

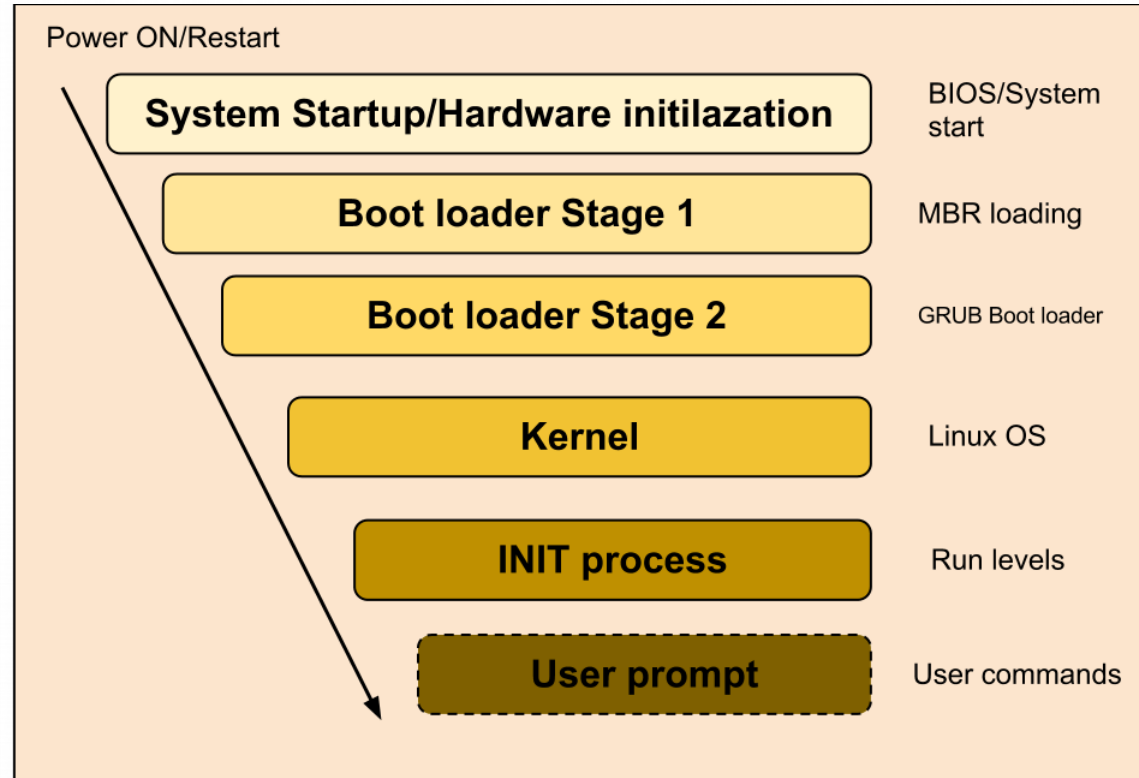
Processes & System calls

Terminal & Shell



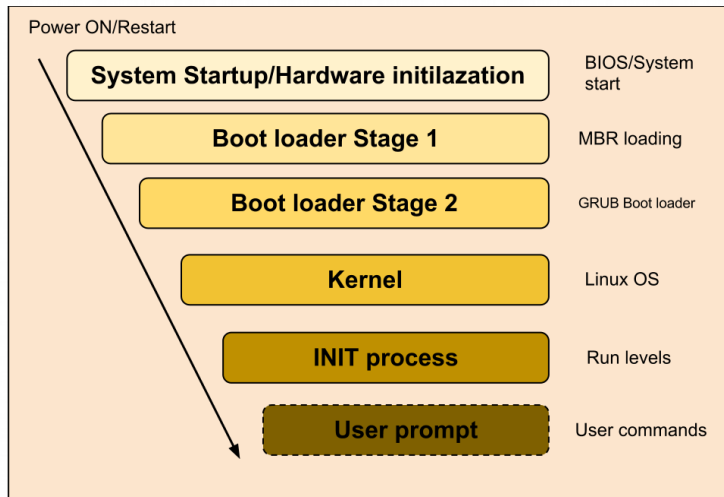
Linux Boot Process

6 stappen



Linux Boot Process

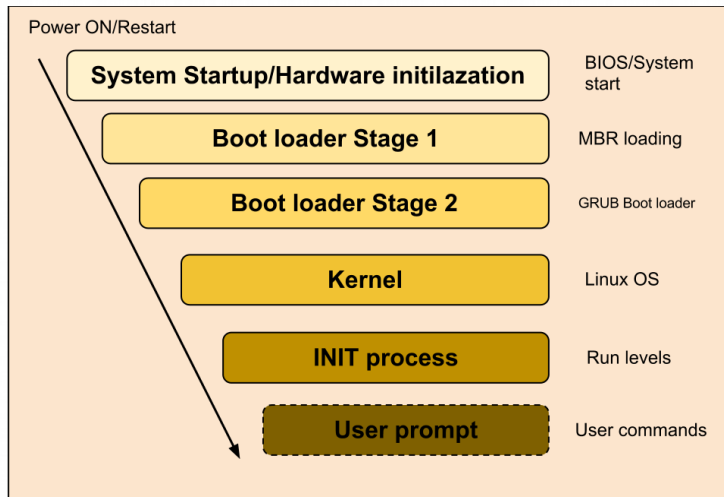
Bios system start



- Test de hardware op fouten
- Zoekt de *MBR boot-loader* op diverse media volgens vooraf bepaalde volgorde
- Eens gevonden laad de BIOS deze MBR boot-loader in het geheugen
- BIOS start dit programma en geeft hiermee de controle over

Linux Boot Process

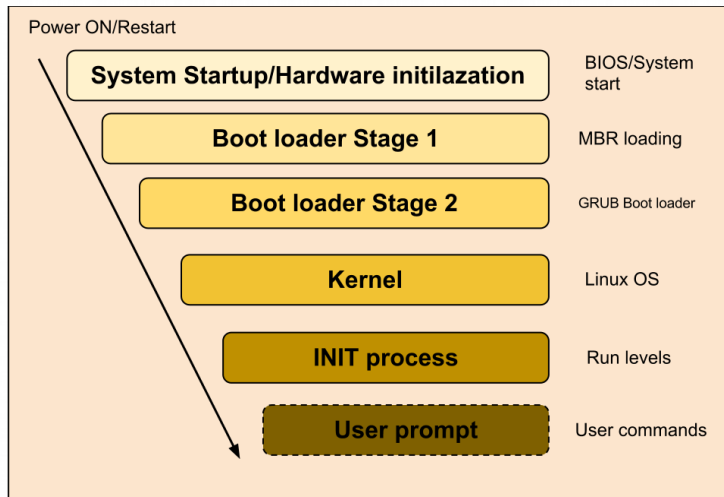
MBR boot loader



- MBR = master boot record
- Staat op de 1^e sector van de bootable disk (*)
- MBR bevat de 'primary boot loader' en de partitie-tabel info
- MBR laad op zijn beurt de 'primary boot loader' in het geheugen
- Bij linux is dit GRUB of LILO (oud)
- GRUB wordt gestart

Linux Boot Process

GRUB boot loader



- GRUB = “grand unified bootloader”
- Start het ‘splash screen’
 - hier kan je kiezen tussen verschillende kernels of zelfs een ander OS.

Linux Boot Process

GRUB boot loader

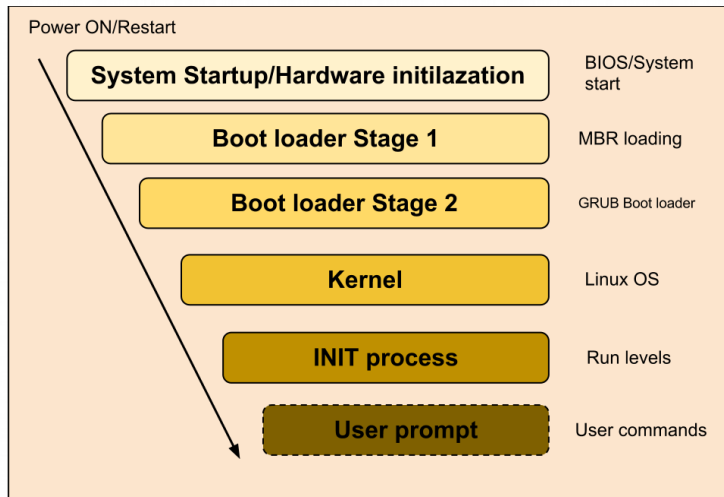
GRUB2
“splash screen”

```
Ubuntu, with Linux 2.6.32-21-generic
Ubuntu, with Linux 2.6.32-21-generic (recovery mode)
Ubuntu, with Linux 2.6.32-19-generic
Ubuntu, with Linux 2.6.32-19-generic (recovery mode)
Memory test (memtest86+)
Memory test (memtest86+, serial console 115200)
```

Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, 'e' to edit the commands
before booting or 'c' for a command-line.

Linux Boot Process

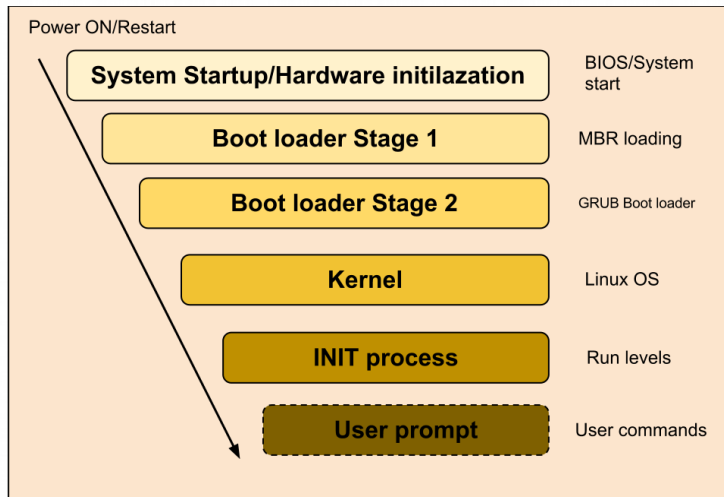
GRUB boot loader



- GRUB = “grand unified bootloader”
- Start het ‘splash screen’
 - hier kan je kiezen tussen verschillende kernels of zelfs een ander OS.
- GRUB heeft een config file in */boot/grub/grub.conf*
- GRUB laad nu de kernel uit de */boot* directory
- GRUB start de linux kernel

Linux Boot Process

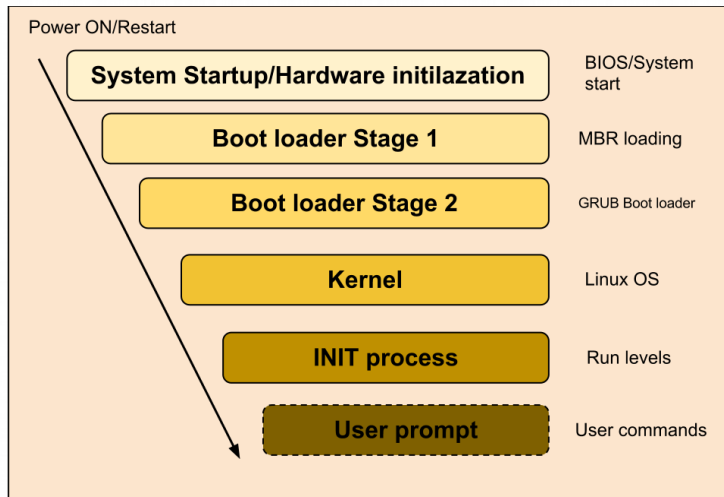
Linux Kernel



- mount het root file system
(dit gaat op een speciale manier want het mount commando is onderdeel van het file systeem...)
- De Kernel start nu het init programma uit /sbin/init
→ *init.d* of *upstart* of *systemd*
Het volledige FS wordt gemount
- Init neemt over..

Linux Boot Process

Init process



- Init bepaald het “run level” uit `/etc/inittab` of het “target level” bij systemd in `/lib/systemd/system`
- Init gebruikt deze info om de desbetreffende programma's te laden en te runnen
- Init is het éérste process dat gestart wordt door de kernel `pid=1`

Linux Boot Process

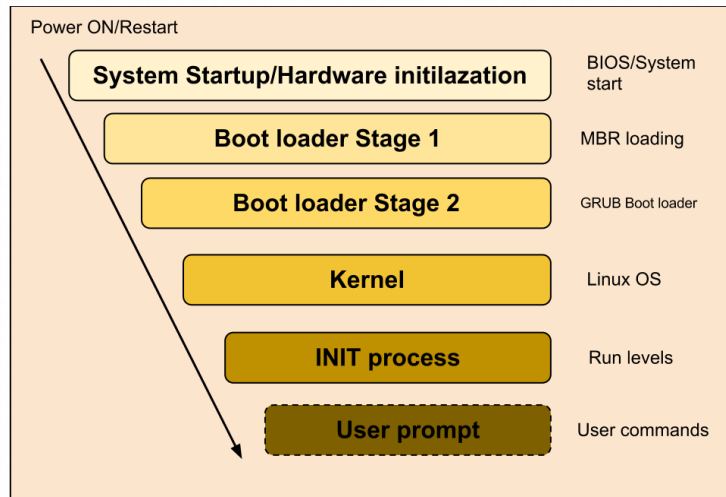
Runlevel	Scripts Directory (Red Hat/Fedora Core)	State
0	/etc/rc.d/rc0.d/	shutdown/halt system
1	/etc/rc.d/rc1.d/	Single user mode
2	/etc/rc.d/rc2.d/	Multiuser with no network services exported
3	/etc/rc.d/rc3.d/	Default text/console only start. Full multiuser
4	/etc/rc.d/rc4.d/	Reserved for local use. Also X-windows (Slackware/BSD)
5	/etc/rc.d/rc5.d/	XDM X-windows GUI mode (Redhat/System V)
6	/etc/rc.d/rc6.d/	Reboot
s or S		Single user/Maintenance mode (Slackware)
M		Multiuser mode (Slackware)

Linux Boot Process

Runlevel	Target Units	Description
0	<code>runlevel0.target</code> , <code>poweroff.target</code>	Shut down and power off the system.
1	<code>runlevel1.target</code> , <code>rescue.target</code>	Set up a rescue shell.
2	<code>runlevel2.target</code> , <code>multi-user.target</code>	Set up a non-graphical multi-user system.
3	<code>runlevel3.target</code> , <code>multi-user.target</code>	Set up a non-graphical multi-user system.
4	<code>runlevel4.target</code> , <code>multi-user.target</code>	Set up a non-graphical multi-user system.
5	<code>runlevel5.target</code> , <code>graphical.target</code>	Set up a graphical multi-user system.
6	<code>runlevel6.target</code> , <code>reboot.target</code>	Shut down and reboot the system.

Linux Boot Process

User commands



- Afhankelijk van de geladen programma's krijgt de user een GUI te zien of een terminal etc..
- Het systeem is volledig opgestart.

Linux Boot Process

Samenvatting:

- BIOS → BIOS system start → laad MBR boot loader
- MBR boot loader → laad 'primary boot loader' van master-boot-record
- Primary boot loader (GRUB)
 - laad en toont splash-screen
 - laad gekozen kernel uit /boot
 - start kernel
- Kernel
 - mount root file system
 - start init
- Init (systemd)
 - laad programma's van gekozen run-level
- Run-level gestart, klaar voor de user

Processes & System calls



Linux Processes & Linux System calls

Wat is een linux process?

- Een proces voert een bepaalde taak uit binnen het systeem
- *Proces = "Programma-code in uitvoering"*
- Processen worden aangeduid met uniek referentie nummer

PID = **P**roces **I**dentification number

- De kernel creëert, beheert en vernietigt processen in het systeem



Linux Processes & Linux System calls

Wat is een system call?

- Wanneer een programma-code in uitvoering beroep doet op de functies van de kernel → system-call
- Weergegeven als functies in de c-programmeertaal.
- System call's → bvb c-functies die hardware aanspreken
- Uitvoering altijd indirect:
 - het zijn vragen aan de kernel
 - kernel beslist en voort uit (of niet → later)



Linux Processes & Linux System calls

Voorbeelden van system calls:

- `sys_read`

```
ssize_t sys_read(unsigned int fd, char * buf, size_t count)
```

Action: read from a file descriptor

Linux Processes & Linux System calls

Voorbeelden van system calls:

- `sys_write`

`ssize_t sys_write(unsigned int fd, const char * buf, size_t count)`

Action: write to a file descriptor

Linux Processes & Linux System calls

Voorbeelden van system calls:

- `sys_chdir`

```
int sys_chdir(const char * filename)
```

Action: change working directory

Linux Processes & Linux System calls

Voorbeelden van system calls:

- `sys_lchown`

```
int sys_lchown(const char * filename, uid_t user, gid_t group)
```

Action: change ownership of a file

Linux Processes & Linux System calls

System calls m.b.t creatie v/e proces

- Fork()
 - Exec()
 - Wait()
 - _Exit()

Linux Processes & Linux System calls

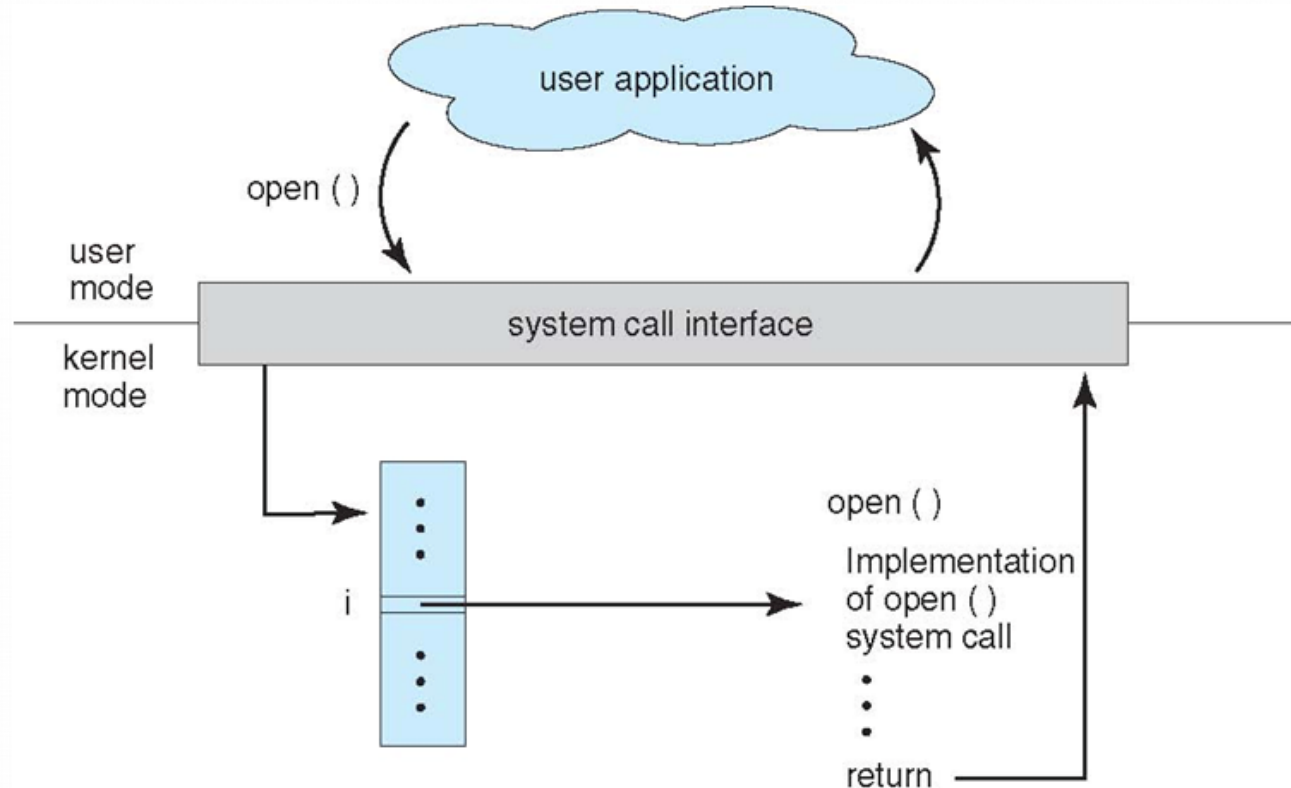
Kernel & User space

- Bij het opstarten splits de kernel het geheugen in twee belangrijke delen op:
 - Kernel space → dit is waar de kernel zich bevindt alsook het geheugen dat de kernel gebruikt
 - User space → dit is waar de user processen runnen
- **Merk op:** 'user' is hier ruimer dan de fysieke gebruiker !
 - bvb apache webserver → user: "www-data"
 - bvb geluidsprogramma pulseaudio → user: "pulse"



Linux Processes & Linux System calls

Kernel & User space



Linux Processes & Linux System calls

init: het éérste proces

- Tijdens het booten start de kernel het init proces.
- Dit is het éérste proces in 'user-space'
- De kernel kent dit proces **PID = 1** toe
- Init start bij het booten vervolgens alle programma's vereist voor het gekozen runlevel
- Al deze alsook alle volgende processen vertrekken vanuit dit éérste init proces
- Hoe.. ?
 - *door middel van “forking”*



Forking



Forking

wat?

- De system call `fork()` creëert een exacte kopie van het aanroepende programma in memory
- De kernel start deze kopie één instructie na de fork call
- Ook het origineel gaat verder met uitvoering
- Het origineel noemen we het “**parent process**”
- De kopie noemen we het “**child process**”

Forking

wat?

- De kernel kent elk child proces eigen PID nummer toe
- Fork() returned het PID nummer van het child proces aan het parent proces
- Fork() returned 0 (nul) aan het child proces

Forking

voorbeeld

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```


Forking

voorbeeld

parent

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

Forking

voorbeeld

parent

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```



child

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

Forking

voorbeeld

parent

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

child

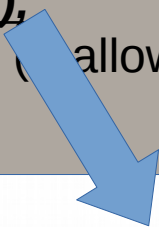
```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

Forking

voorbeeld

parent

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

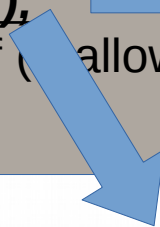


child

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

child

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```



parent

child

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

Forking

voorbeeld

parent

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

child

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

child

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

child

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```



Forking

voorbeeld

parent

helloworld

```
int main() {  
    fork();  
    fork();  
    printf ("helloworld");  
}
```

child

helloworld

```
int main() {  
    fork();  
    fork();  
    printf ("helloworld");  
}
```

child

helloworld

```
int main() {  
    fork();  
    fork();  
    printf ("helloworld");  
}
```

child

helloworld

```
int main() {  
    fork();  
    fork();  
    printf ("helloworld");  
}
```



Forking

Parent **PID=278**

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

voorbeeld

child **PID=344**

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

child: **PID=425**

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

child: **PID=536**

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```



Forking

Parent PID=278

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

child PID=344

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

voorbeeld

return-waarde
int x = fork();

child: PID=425

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```

child: PID=536

```
int main() {  
    fork();  
    fork();  
    printf ("halloworld");  
}
```



Forking

Parent PID=278

```
int main() {  
    fork();  
    fork();  
    printf ("Hello world");  
}
```

child PID=344

```
int main() {  
    fork();  
    fork();  
    printf ("Hello world");  
}
```

voorbeeld

return-waarde
int x = fork();

child: PID=425

```
int main() {  
    fork();  
    fork();  
    printf ("Hello world");  
}
```

child: PID=536

```
int main() {  
    fork();  
    fork();  
    printf ("Hello world");  
}
```



Forking



Forking

returnwaarde

Hoe kan het programma weten of het parent of child is?

```
if ( fork() == 0 ) {  
    // Child  
} else {  
    // parent  
}
```

Forking

starten ander programma

- Fork() wordt aangeroepen → creatie Child process
- Executie van parent gaat verder
- if proces == Child → volgend commando: **Exec() system call**
- Exec() system call :
 - wist de bestaande programma code in het child process
 - laad de nieuwe programma code in het child process
 - start de uitvoering van deze nieuwe code

Forking

Waarom deze manier van starten?

- Door Forking erven child processes dezelfde environment parameters
- Dit mechanisme heeft bewezen efficiënt te zijn
- Door Forking is het heel eenvoudig om een tweede versie van hetzelfde programma te starten



Forking

Beëindigen programma

- Bij het eindigen van een programma wordt de **_Exit()** system call aangeroepen
- “_” om verschil aan te duiden met C-functie exit()
- _Exit() returned 0 bij normaal programma einde
- Error-code bij crash, out-of- memory etc

Forking

Forking met blocking

- Fork() wordt aangeroepen
- if proces == parent
 - volgend commando: Wait() system call
- Wait() system call :
 - blokkeert verdere uitvoering van de parent
 - dit totdat het child proces **_Exit()** aanroept

Forking

Forking door init

Nu begrijpen we exact wat init doet bij boot:

- 1) *Init Fork()’t zichzelf → creatie child*
- 2) *In het child proces wordt Exec() aangeroepen*
- 3) *Exec() vervangt de code en start dit programma*
- 4) *terug naar 1) en start het volgende programma tot alle programma’s van dit runlevel gestart zijn*

Forking

Forking door init

Init blijft verantwoordelijk tot shutdown:

- *Telkens een programma start is dit door het **Fork()** en **Exec()** mechanisme met aan de Bron de “parent van alles” PID=1*
- *Bij shutdown → alle processen **_exit()** en dit tot init geen children meer heeft*

Linux Processes

Status

Linux processen kunnen zich in verschillende stadia bevinden

1) Running het proces loopt of is klaar om te starten

(klaar om te start = het wacht om toegewezen te worden
aan een CPU door de kernel)



Linux Processes

Status

2) Waiting het proces wacht op een event of op een resource.

→ 2 soorten: *interruptible & uninterruptible*:

- Interruptible waiting processes:
kan onderbroken worden door signals
(signal = software interrupt)
- uninterruptible waiting processes:
wacht op de juiste hardware condities en kan onder geen enkel beding onderbroken worden.

Linux Processes

Status

3) Stopped het proces is gestopt.

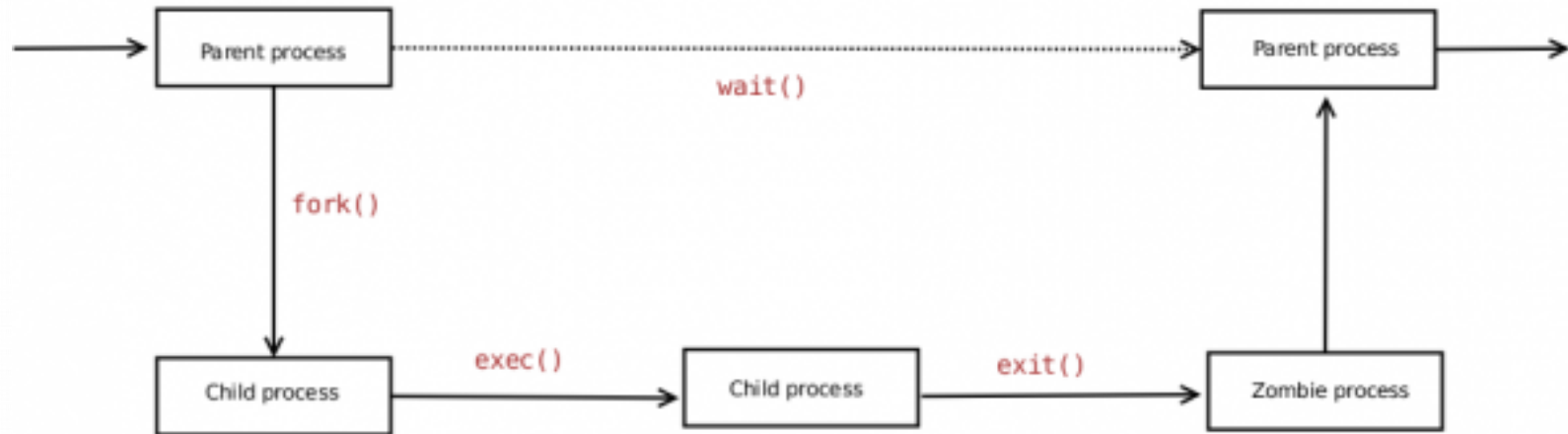
- Meestal na het ontvangen van een signal.
- Een process dat ge-debugged wordt kan eveneens “Stopped” zijn

4) Zombie is een gestopt proces dat voor een of andere reden nog steeds een task struct data structuur heeft in de task vector. Het is zoals het klinkt een dood proces.

Linux Processes

Status

Linux Processes Life Cycle



Linux Processes

Commando's

- **ps** commando → geeft een statische lijst van actuele processen
- **top** commando → doet hetzelfde maar een dynamische lijst
- **kill** commando → laat de admin toe om zelf een process vroegtijdig stop te zetten.
- **pstree** commando → iets grafische weergaven
- **pgrep** → toont de processen van een bepaald programma
bvb pgrep firefox

Linux Processes

Demo Commando's

