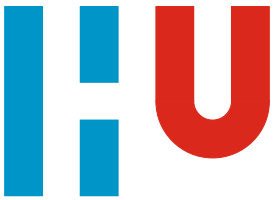
Embedded Operating Systems

Brian van der Bijl

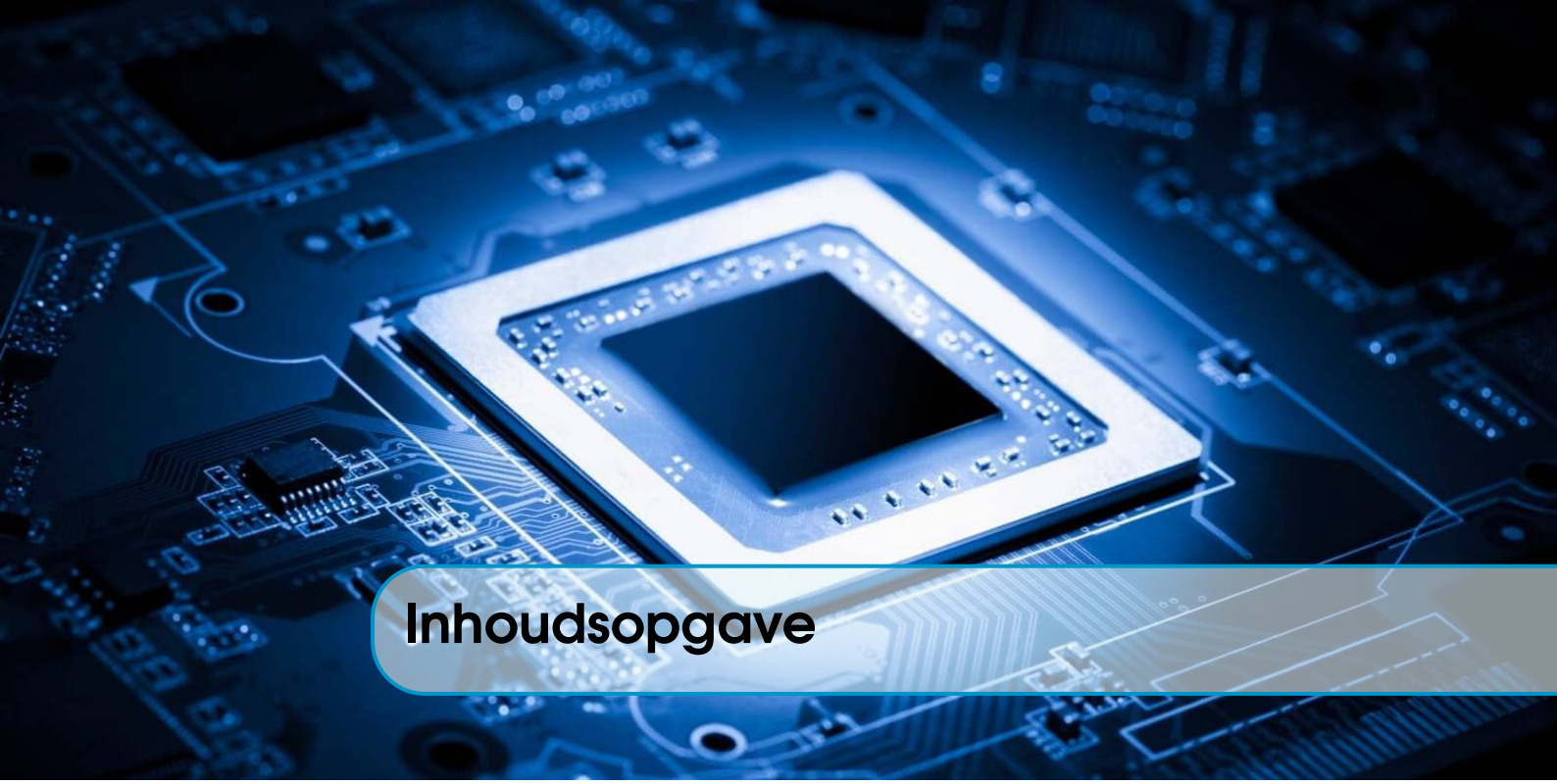


Laatste update: 11 februari 2019

This reader was verified by:

Brian van der Bijl	docent
Jorn Bunk	hogeschooldocent

First release, January 2019



Inhoudsopgave

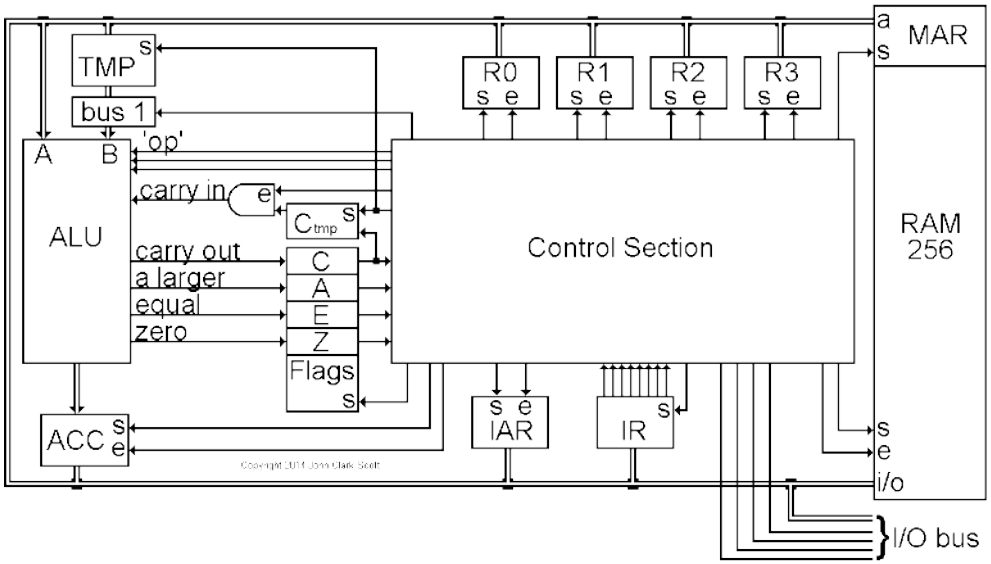
1. Wat is een OS?

Met CSN hebben we gezien hoe een computer opgebouwd kan worden uit niet veel meer dan NAND-gates. Toen we het computersystemen-gedeelte afsloten hadden we een werkende computer (zie Figuur: De CPU), die met behulp van instructies (in de vorm van binaire getallen) te programmeren was. Door een set instructies in het geheugen te zetten en de CPU naar de eerste instructie te verwijzen kon een stappenplan worden uitgevoerd, waarna het resultaat ook weer ergens in het geheugen te lezen was.

De computer die we hebben opgebouwd is echter nog niet zo makkelijk te gebruiken als bijvoorbeeld je laptop. Bij het opstarten komt deze in een werkende omgeving terecht met Windows, Linux of MacOS, en kun je zelf software schrijven met bijvoorbeeld Python. Python is net als machine-code een manier om de computer te programmeren, maar werkt een stuk makkelijker: je kan met een enkele regel programmeercode complexe bewerkingen uitvoeren, en hoeft niet alles op CPU-niveau uit te denken. In de cursus Embedded & Operating Systems gaan we kijken hoe we deze twee werelden bij elkaar kunnen brengen: de hardware die denkt in enen en nullen, en de programmeur / gebruikers die liever in menselijke termen redeneren.

1.1 Het Operating System (OS)

Om deze twee werelden bij elkaar te brengen is een systeem nodig dat enerzijds met de specifeke hardware overweg kan, en anderzijds eenzelfde basis biedt voor software (ongeacht wat de onderliggende hardware is). Dit is het Operating System, dat als laag tussen de applicaties en de hardware zit. Zonder dit OS moet je als programmeur zelf alle details van de computer-hardware managen, waardoor je telkens weer veel (foutgevoelige) code nodig hebt om simpele taken te kunnen programmeren. Denk hierbij aan het opslaan van data, het afwisselen van meerdere taken, of het weergeven van een interface op het scherm.

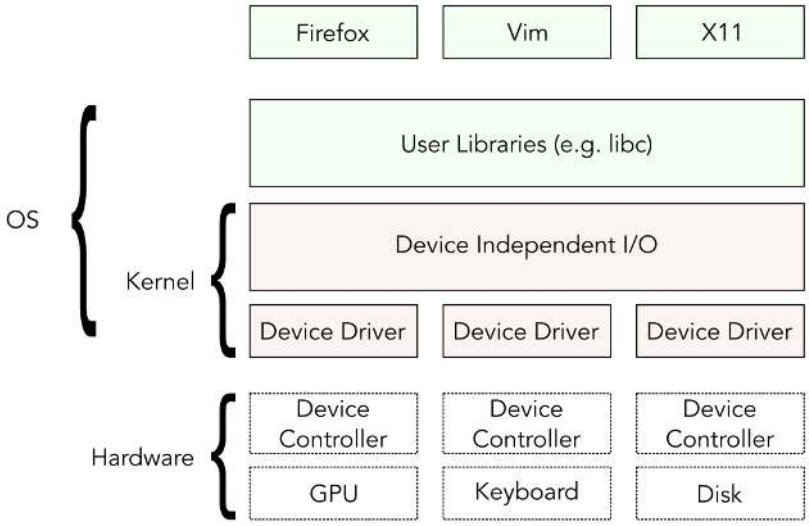


Figuur 1.1 Figuur: De CPU

Software zoals een web-browser of office-suite zijn zonder de abstractie van een OS vrijwel onmogelijk te programmeren, en als het al lukt specifiek voor één systeem gemaakt. Met het OS is er een basis waar complexe software op gemaakt kan worden, en in deze cursus maken we kennis met de principes van OS-ontwikkeling — dit is waar de TI-er zich binnen ICT onderscheidt. Het OS bestaat uit twee lagen: de kernel die direct met de hardware communiceert en de rest van het OS. Het verschil zit hem in het feit dat de kernel in kernel mode draait, en de rest van het OS net als de andere software in user mode. Dit is een verschil in welke instructies de CPU wel en niet mag uitvoeren; hier komen we later op terug. Voor nu heeft het OS twee raakvlakken: - De specifieke hardware, die door het OS aangestuurd wordt - Software, waarvoor het OS een generieke basis levert

1.2 Voorbeelden

De bekendste voorbeelden van Operating systems zijn (op de desktop) Windows, MacOS en Linux. Moderne smartphones hebben echter ook een OS, dat vanuit ons perspectief niet of nauwelijks van een desktop-OS verschilt. Android en iOS zijn de belangrijkste mobiele OSs, waarbij Android een variant van Linux en iOS een variant van macOS is. In beide gevallen is de gebruikerservaring compleet verschillend, en zal de eindgebruiker niet snel in de gaten hebben dat het om (vrijwel) hetzelfde OS gaat. Dit is omdat het daadwerkelijke OS eigenlijk vooral het gedeelte van het systeem is dat je als gebruiker niet ziet: de grafische interface is bijvoorbeeld strict genomen geen onderdeel van het OS. Wel is de gebruikersomgeving vaak aan het OS gekoppeld, en worden ze samen als bundel beschikbaar gesteld; vrijwel niemand zit er op te wachten dit handmatig te moeten installeren, en met name voor commerciële systemen (Windows en MacOS) is de gebruikersinterface alleen voor het bijbehorende OS



Figuur 1.2 Figuur: Basismodel Operating Systems

beschikbaar.

1.2.1 Windows

Microsoft Windows is nog steeds het meest gebruikte desktop OS. De meeste computers worden op het moment met Windows 10 verkocht, dat sinds 2015 de meest recente Windows versie is. De huidige versies van Windows stammen allemaal af van Windows NT uit 1993.

1.2.2 MacOS

Apple heeft een eigen besturingssysteem in de vorm van MacOS (voorheen Mac OS X). De huidige versie, Mojave / 10.14, stamt uit 2018. De eerste versie van Mac OS X (10.0) is in 2001 uitgebracht en sindsdien krijgt het systeem ruwweg jaarlijks een grote update waarbij "feature updates" (nieuwe functionaliteiten) en "under the hood" (minder zichtbare verbeteringen) updates elkaar afwisselen.

1.2.3 Linux

Linux is een gratis open-source OS dat sinds 1991 bestaat. Open source betekent dat het vrij te gebruiken en aan te passen is, en dat dit recht doormiddel van copyright beschermd wordt.In tegenstelling tot Windows en MacOS wordt Linux niet als een compleet pakket verkocht, maar is het als los besturingssysteem beschikbaar. Omdat Linux gratis te gebruiken en te verspreiden is bestaan er vele versies waarbij het OS gebundeld wordt met een of andere grafische interface en extra software. In dit vak zullen we Linux vaak gebruiken als voorbeeld en werkplatform; in les 2 komt het OS uitgebreider aan bod.

Figuur 1.3 Figuur: Windows

Figuur 1.4 Figuur: MacOS

Figuur 1.5
Figuur: Linux

1.2.4 Mobiele OSs en smart devices

Ook mobiele telefoons en smart devices (Smart TVs, Tablets, ...) maken meestal gebruik van een OS. Voor mobiele telefoons zijn de grootste twee OSs Android (een afgeleide van Linux, ontwikkeld door Google) en iOS (een afgeleide van MacOS). Voor andere apparaten worden vaak ook varianten van Android en iOS gebruikt, gespecialiseerd voor het specifieke apparaat.

1.3 Taken Operating System

Nu we wat voorbeelden gezien hebben gaan we eerst kijken naar wat een OS eigenlijk is. De beste manier om het idee van een OS te vatten is te kijken naar de taken die het uitvoert:

- Abstractie van de hardware t.b.v. user processes
- Aansturen randapparatuur / IO
- Processen laden en managen / scheduling
- Beheren system resources (e.g. memory, CPU-tijd)
- Leveren van het filesystem (indien van toepassing)

Al deze taken zullen in deze cursus bekeken worden.

1.3.1 Abstractie

De meerwaarde van het OS is dat je, in plaats van dat je software specifiek schrijft voor een bepaalde hardware-configuratie, schrijft voor een generiek OS. Nog steeds andere code nodig voor e.g. Linux vs Windows, maar dit betekent 1-3 platforms ondersteunen i.p.v. elke mogelijke hardwarecombinatie. Verdere abstractie is mogelijk (in Python maakt het nauwelijks uit of je voor Windows of Linux of Mac programmeert) maar dat is weer een laag bovenop het OS.

Figuur 1.6 Figuur: Mobiele OSs

1.3.2 Bare Metal

Niet alle computersystemen gebruiken een OS. Een systeem dat zelf direct met de hardware communiceert noemen we bare metal. Een operating system en bare metal-software hebben beide te maken met de directe hardware, en gebruiken veelal dezelfde principes om hiermee om te gaan: de principes waar je in de rest van deze cursus kennis mee zal maken. Het voornaamste verschil is dat het OS enkel een facilitaire taak heeft, namelijk het ondersteunen van de applicaties die erop gedraaid worden. Bare-metal software heeft wel een eigen taak, en is dus te beschouwen als de combinatie van een heel simpel OS en een enkele applicatie in één. Meestal hebben we het over microcontrollers als we het over bare-metal systemen hebben die zelf geen OS zijn. Natuurlijk zijn er meer verschillen waar je rekening mee moet houden als je bare-metal (aan OSs of microcontrollers) programmeert, hier zullen we later in de cursus bij stilstaan.

1.4 User Mode vs Kernel Mode

Een processor draait in verschillende modes, elk met eigen priviliges. We onderscheiden hier twee modes of "privilege levels": de kernel mode met alle privileges, en de user mode die niet direct tegen de hardware kan praten, maar van het OS afhankelijk is. User mode processen hebben daarmee ook geen directe toegang tot het geheugen. Als ze iets uit het geheugen willen halen moeten ze dit doen via het OS, dat daardoor kan voorkomen dat user-mode

processen buiten hun eigen toegewezen geheugen lezen of schrijven. OS (of specifieker de kernel) draait in kernel mode, en kan wel overal bij. Dit is van belang omdat een general-purpose computer niet zonder meer alle software kan vertrouwen. Een kwaadaardig programma kan bijvoorbeeld proberen gegevens van andere processen te lezen om achter wachtwoorden of bankgegevens te komen, of zichzelf in het opstartproces installeren. Dankzij het gebruik van privilege-levels kunnen dit soort activiteiten niet zonder meewerking van het OS plaatsvinden. Ook kan een fout in een programma voor grote schade zorgen, die dankzij privileges tot het programma zelf beperkt kunnen worden gehouden, in plaats van dat het hele systeem om zeep wordt geholpen. De huidige mode waarin de CPU draait is opgeslagen in een register binnen de CPU. Voor x86 is dit in het Code-Segment (CS) register. Dit register hield origineel bij in welk geheugen-segment de code staat die uitgevoerd wordt, waarbij de laatste twee bits het privilige level aanduidden. Alleen code die op de juiste plaats in het geheugen stond kon in kernel mode worden uitgevoerd, en het OS zorgde er tijdens het opstarten voor dat de kernel-mode segmenten overeen komen met waar de OS code te vinden was. Tegenwoordig wordt het CS register hier niet meer voor gebruikt, maar nog steeds duiden de laatste bits aan in welke mode de CPU zich bevindt. De naam “Kernel mode” houdt verband met de term “kernel”, waar de kern van het besturingssysteem mee wordt aangeduid.

1.4.1 Kernel Models

Daarmee meteen nog een belangrijk onderscheid tussen verschillende kernels: monolitisch vs microkernels. Monolitische kernels draaien alle traditionele OS taken in kernel mode, en zijn dus als een monoliet (grote steen uit één stuk). Linux en Windows zijn voorbeelden van monolithische kernels. Microkernels draaien alleen het broodnodige in de kernel, en besteden al het andere zoveel mogelijk uit aan user mode processen. Microkernels zouden veiliger zijn (al is dit een open debat) maar betalen hier wel voor met lagere performance door meer te moeten switchen tussen de verschillende processen wiens functionaliteit anders onder de kernel zou vallen.

1.5 SysCalls

Als een applicatie iets gedaan wil krijgen, heeft het voor veel taken niet de juiste privileges. Het is daarmee dus op het OS aangewezen. Een programma kan (en zal, met grote regelmaat) de kernel vragen om bijvoorbeeld (file) IO te doen, andere applicaties te starten, extra geheugen te reserveren, etc. Dit gebeurt door middel van system calls (SysCalls). Dit zijn functies die vanuit de software worden aangeroepen, en de processor onderbreken. Deze zal speciale code van het OS uitvoeren, die wel in kernel mode mag draaien. De OS-code voert het verzoek uit, en schakelt daarna weer terug naar user-mode. Hierna kan de software verder gaan waar het is gebleven. In Linux heeft iedere applicatie (of specifiek ieder proces, maar hier komen we later op terug) een eigen kopie van de SysCall table met de nummers en adressen van de system calls. Dit zorgt

ervoor dat SysCalls zo snel mogelijk kunnen worden afgehandeld.

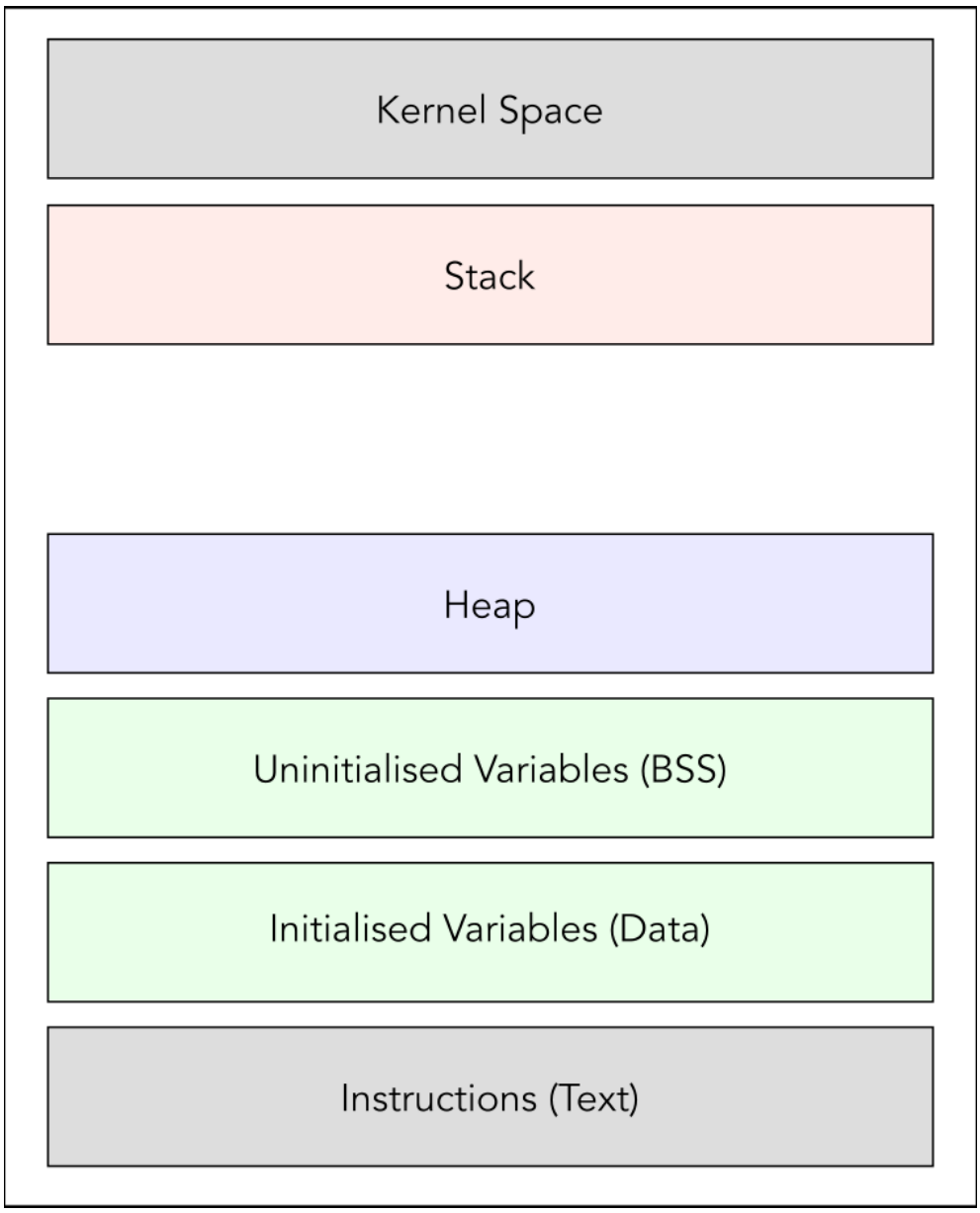
De SysCalls in een Linux systeem zijn heel anders dan die op Windows. Dit heeft ermee te maken dat de besturingssystemen een heel andere filosofie hanteren. In Linux is het bijvoorbeeld gebruikelijk om meerdere kleine (command-line) programma's aan elkaar te koppelen, en hier zijn dan ook speciale SysCalls voor. Windows heeft deze filosofische achtergrond niet (maar werkt vaak met grotere programma's die vele taken kunnen vervullen), waardoor deze SysCalls in Windows lange tijd niet aanwezig waren (inmiddels is dit concept overigens ook door Windows geleend). De meeste SysCalls bieden functionaliteit die in ieder OS gebruikt wordt, maar werken per verschillend OS net even anders. Dit is de reden dat Windows-software niet zonder meer op Linux werkt (en vice-versa). De "Portable Operating System Interface" (POSIX) en "Single Unix Specification" (SUS) zijn standaarden waar Unix-achtige systemen (en in het geval van POSIX zelfs recente Windows versies) aan voldoen, om het makkelijker te maken software op meerdere platforms te laten draaien.

1.6 Geheugensegmenten

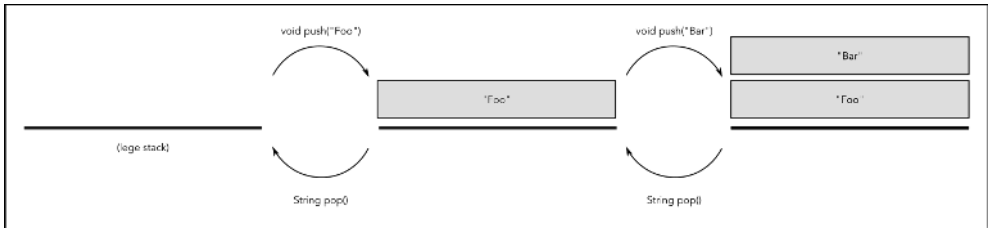
Hoewel we het inmiddels een paar keer over processen hebben gehad, leggen we voor nu de nadruk nog even op een computer met een enkele taak. Later zullen we zien hoe een computer meerdere processen kan afwisselen (of met meerdere cores, tegelijk kan draaien). Het geheugen is opgedeeld in een aantal segmenten. De eerste sectie is Text en bevat de instructies van het proces. Dit gedeelte van het geheugen is na het inladen van het proces read-only: als alle code is geladen mag deze in het kader van de veiligheid niet meer veranderd worden. Na de instructies volgen de Data en BSS segmenten. Data bevat alle globale en statische variabelen met een initiele waarde. Deze worden buiten functies gedefinieerd of zijn als static (onveranderbaar) gemarkeerd. BSS (historisch: Block Started by Symbol) bevat de statische variabelen die wel defined, maar niet geïnitialiseerd zijn. Na de BSS volgen de heap en de stack, die in de volgende sectie besproken worden. Beide kunnen hun formaat dynamisch aanpassen. De stack en heap beginnen leeg en groeien naar elkaar toe. Tot slot hebben we in een aantal OSs, waaronder Linux, een stuk kernel space; deze bevat de code van de SysCalls en een index: een tabel met voor elke SysCall het nummer van de call en het geheugenadres waar de bijbehorende code te vinden is. Doorgaans hebben SysCalls ook een naam, maar deze wordt door de computer intern niet gebruikt. Bij het gebruik van een high-level programmeertaal zoals Python heb je als programmeur vaak geen directe invloed op wat waar in het geheugen komt: dit wordt door de Python-interpreter gemanaged.

1.6.1 De Stack

Eerst zullen we even inzoomen op de Stack, en om deze te begrijpen moeten we eerst weten wat **een** stack is.



Figuur 1.7 Figuur: Geheugenmodel Proces



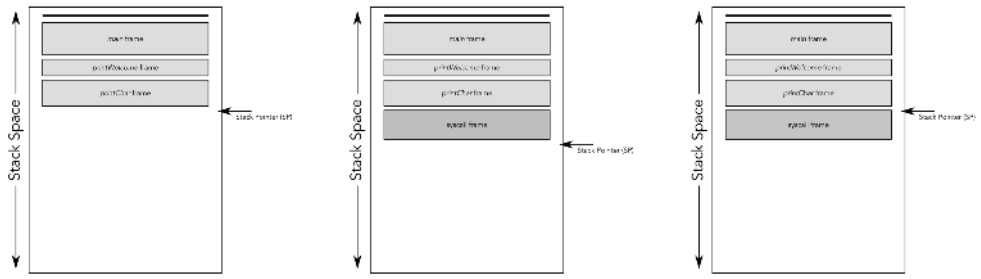
Figuur 1.8 Figuur: Stack (LIFO)

Intermezzo: Stacks

Een stack (stapel) is een datastructuur waar dynamisch informatie aan toegevoegd of uitgehaald kan worden. De stack werkt volgens het **Last in, first out** principe, en wordt daarom ook wel de LIFO genoemd. De stack begint leeg, waarna hier data aan kan worden toegevoegd. Nieuwe data wordt bovenop de stack gezet (dit heet een **push**), en alleen het bovenste element van de stack is zichtbaar. Als het bovenste element gelezen wordt, wordt het meteen van de stack verwijderd (dit een heet **pop**).

Ook in het geheugen wordt gebruik gemaakt van een stack. De stack heeft als voordeel dat je altijd de meest recente informatie bovenaan hebt. De stack wordt gebruikt om *function calls* mogelijk te maken: het aanroepen van een functie. Op het moment dat dit gebeurt wordt er een nieuw stuk van het geheugen uitgevoerd (dus niet simpelweg de volgende instructie). Dit betekent dat de oude variabelen even niet meer toegankelijk moeten zijn, en dat daar mogelijk nieuwe variabelen voor in de plaats komen. Ook moet bekend zijn waar de software gebleven was, zodat die nadat de functie beëindigd is weer op de juiste plek verder kan. Als de functie wordt aangeroepen, wordt een frame aangemaakt. Dit frame bevat de parameters van de functieaanroep, het adres van waaraf de functie werd aangeroepen (return address), lokale variabelen en soms het formaat van het frame of de vorige waarde van de stack pointer (zodat bekend is welke bytes per *pop* gelezen moeten worden, wanneer frames geen vast formaat hebben). Hierna kan met behulp van een `JUMP`-instructie (bekend van CSN) naar de functie-instructies gesprongen worden. Als een functiecall klaar is dan keert de computer terug naar de instructie direct na de functieaanroep. Deze is te vinden doordat het return adres in het frame op de stack staat opgeslagen. Zodra de computer het return adres nodig heeft wordt het frame gepopt, en wordt de stack meteen een frame kleiner.

De stack heeft een maximale grootte. Bij simpele systemen kan dit komen door een beperkte hoeveelheid geheugen, in complexere systemen met meerdere processen is er een maximum vastgesteld. Als dit maximum bereikt wordt, spreken we van een stack overflow, en wordt de executie van het programma onderbroken (met andere woorden: een crash!). Het maximumformaat van de stack is doorgaans vrij groot, een stack overflow wordt dan ook meestal door een programmeerfout veroorzaakt. Het meest simepele voorbeeld is een functie die zichzelf herhaaldelijk aanroept (recursie). Recursie is in principe geen probleem (en in sommige gevallen zelfs de meest nette oplossing) maar als een functie altijd zichzelf aan blijft roepen (en niet na een beperkt aantal keren returnt)



Figuur 1.9 Figuur: De Stack

dan hebben we een oneindige loop waarbij de stack snel volgezet wordt. Bij een beperktere recursie wordt ook een stack opgebouwd, maar op een gegeven moment zal de laatste aanroep returnen, waarna de aanroep daarvoor returnt, etc. en de stack ook weer wordt afgebroken.

1.6.2 De Heap

Waar BSS en Data voor statische data gebruikt wordt, is de heap voor dynamische data: data waarvan tijdens het programmeren en compilen nog niet bekend is hoeveel ruimte ervoor nodig is. De heap wordt tijdens het uitvoeren van de code expliciet toebedeeld (gealloceerd). - Dynamische alloceerbaar - Expliciet ruimte reserveren - Na gebruik vrijgeven

1.6.3 Memory Allocation

Met behulp van `malloc` en `free` geeft de software aan hoeveel geheugenadressen in de adresruimte gekoppeld moet worden aan daadwerkelijk heap-geheugen, zodat de software deze kan gebruiken. Bij het gebruik van `malloc` moet worden aangegeven hoeveel ruimte nodig is, maar dit kan in de software berekend worden op basis van bijvoorbeeld input. Waar dit geheugen komt te staan wordt door het OS bepaald, maar het zal altijd een aaneengesloten stuk geheugen zijn. Geheugen moet altijd teruggegeven worden om te voorkomen dat er memory leaks onstaan, en het is niet toegestaan geheugen aan te spreken dat niet gealloceerd is (dit leidt tot de error “segmentation fault”). Het niet teruggeven van gereserveerd geheugen is een van de meest voorkomende fouten bij veel latere TI vakken. Onder water wordt gebruik gemaakt van system calls waaronder `mmap`, `brk` en `munmap`. De aanroep naar `malloc` gebruikt bijvoorbeeld intern ergens een aanroep in de trant van `syscall(SYS_brk, 128*128*3)` om het besturingssysteem te vragen de ruimte vrij te maken. `SYS_brk` is een constante met de waarde `12`, het nummer van de system call.

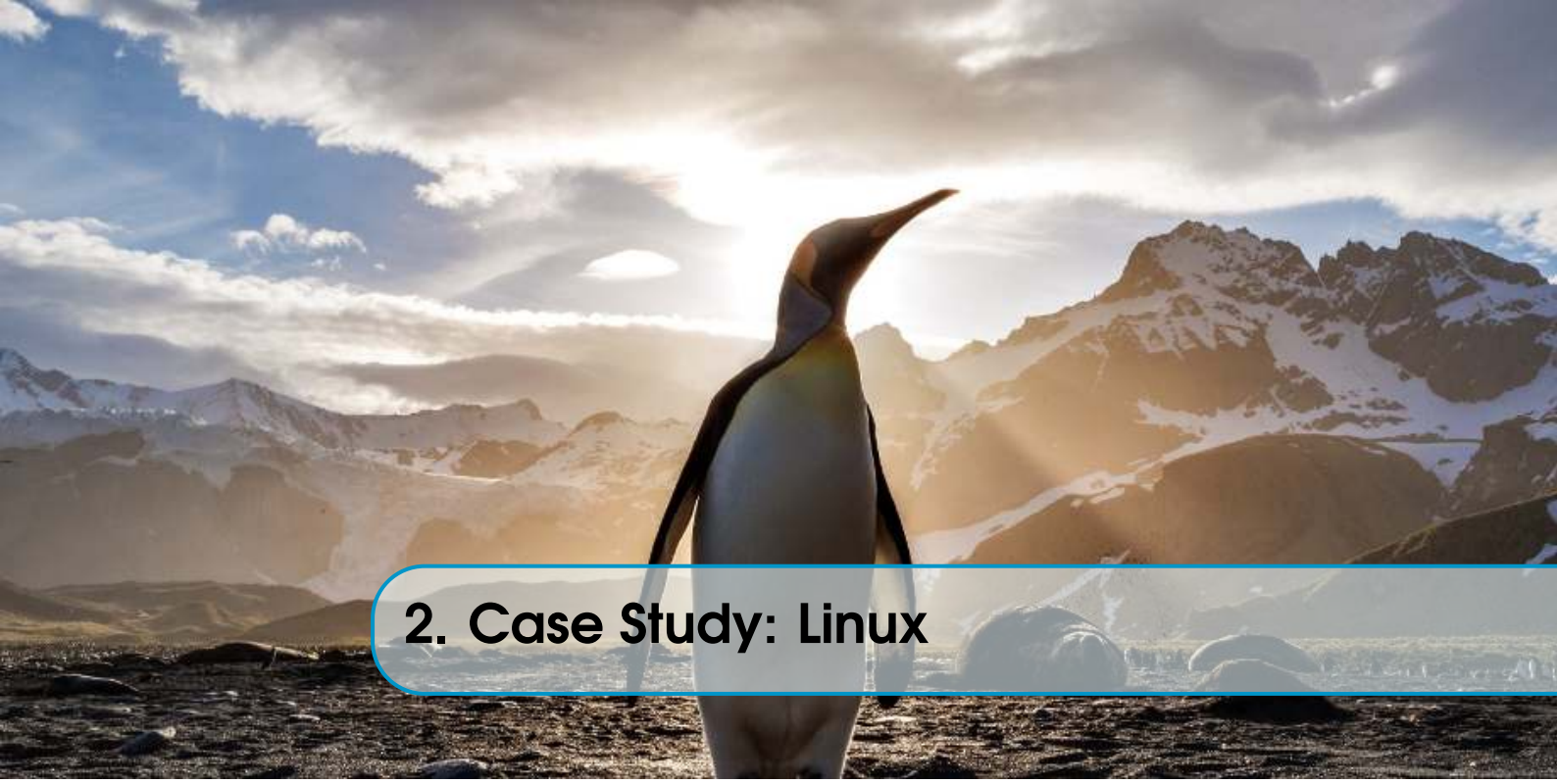
```
1 // reserveer ruimte voor 128x128 pixels met 3 bytes
     per pixel
2 int *img_data = malloc(128*128*3);
3
4 // img_data verwijst naar begin van de gereserveerde
     ruimte
5
6 free(img_data); // geef de ruimte terug aan het OS
```

```
7
8  // Onder water
9  int *img_data = syscall(SYS_brk, 128*128*3);
10 syscall(SYS_munmap, img_data, 128*128*3);
```

1.6.4 Geheugensegmenten in code

Hieronder staat een stukje C++ code waarin de verschillende geheugensegmenten gebruikt worden. Jullie kennen op dit moment waarschijnlijk nog niet zoveel C++ dat alles in het codevoorbeeld meten herkenbaar is; don't worry, hier komen geen vragen over. Dit voorbeeld is vooral ter illustratie om de verschillende segmenten wat concreter te maken.

```
1  #include <vector>
2
3  int x;                    // Globale variabele zonder
       initiele waarde staat in BSS
4  int y = 0;                // Globale variabele met
       initiele waarde staat in DATA
5
6  int main(void)            // Code staat in TEXT
7  { static int i = 10;      // Statische variabele met
       initiele waarde staat in DATA
8    static int j;           // Statische variabele zonder
         initiele waarde staat in BSS
9    int k = 42;             // Functie-variabele staat in
         de STACK
10   vector<int> v = {0,1} // Data van de vector staat in
         de HEAP
11   int *m = malloc(12);    // Ook malloc() reserveert op
         de HEAP
12   return 0;
```



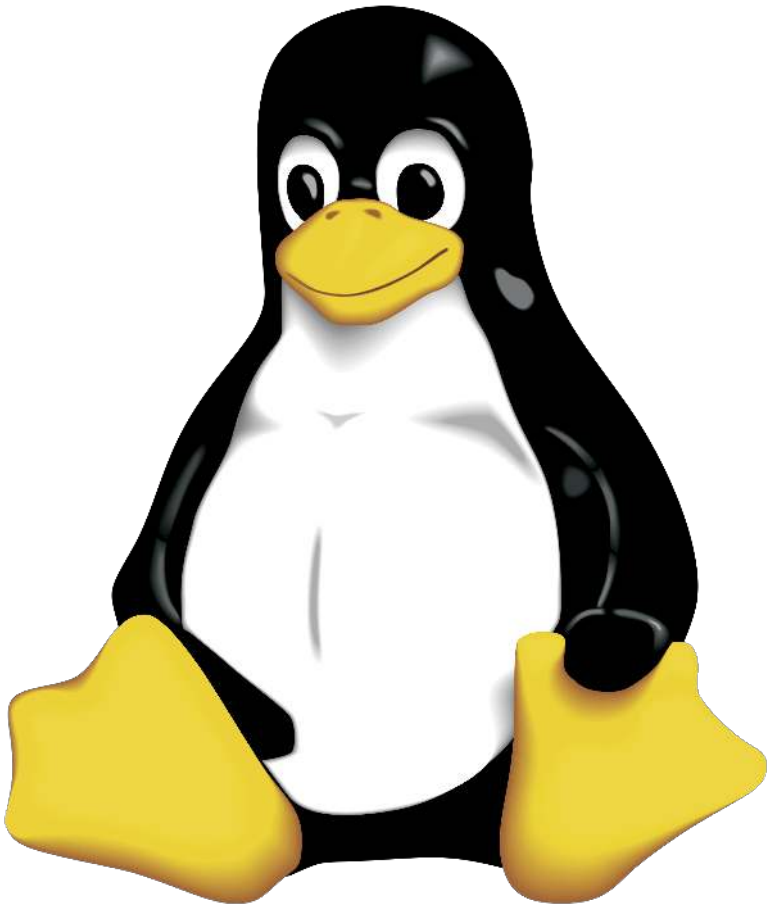
2. Case Study: Linux

Hoewel we in deze lessen de nadruk ligt op de algemene theorie achter operating systems, zullen we vaak naar specifieke voorbeelden kijken om te zien hoe alles nou echt in elkaar zit. Voor deze lessen maken we gebruik van Linux, een vrij en gratis OS gebaseerd op Unix. Daarnaast maken we ook kennis met dit besturingssysteem in de rol van programmeur / gebruiker. Per week zal één les met name de focus hebben op de algemene theorie, en zal in de andere les Linux meer centraal staan, waarbij we gaan leren werken met dit OS.

2.1 Historie

Linux is een afstammeling van de Unix traditie van besturingssystemen, die teruggaat tot Multics dat in de jaren 60 als time sharing OS voor de Multics GE-645 (Figuur Mainframe) werd ontwikkeld door MIT, AT&T Bell Labs en General Electric. Zoals te zien waren computers toen nog grote gedrochten, en was van de personal computer nog geen sprake. Een mainframe, zoals dit type computer heette, werd door hele bedrijven of afdelingen gedeeld. Programmeurs schreven code, waarna ze moesten wachten op hun beurt om de code te draaien. Als er een fout in zat, kon de programmeur opnieuw beginnen en een nieuwe beurt afwachten. Later werd het langzaam mogelijk met meerdere terminals (werkstations) de computer tegelijk te gebruiken. In deze context werd de grondslag voor Unix gelegd.

De ontwikkeling van Multics schoot niet op, waardoor Bell Labs besloot haar onderzoekers terug te trekken. Een deel daarvan, met name Ken Thompson, Dennis Ritchie en Brian Kernighan, ging bij Bell Labs verder met hun werk, maar op een kleinere schaal. Het resultaat hiervan is Unix. Unix had een eigen filosofie, waarbij alles als een file gezien werd (ook, bijvoorbeeld, devices). Hierdoor werd het makkelijker programma's aan elkaar te koppelen, met "bestanden" als koppelstukken. Dit leidde tot een filosofie van kleine, algemene programma's



Figuur 2.1 Figuur: Tux, de Linux mascotte

Figuur 2.2 Figuur: Mainframe

Figuur 2.3 Figuur: Thompson en Ritchie

die oneindig in te zetten en te combineren waren. Dennis Ritchie vatte de filosofie samen als *"What we wanted to preserve was not just a good environment in which to do programming, but a system around which a fellowship could form. We knew from experience that the essence of communal computing, as supplied by remote-access, time-shared machines, is not just to type programs into a terminal instead of a keypunch, but to encourage close communication."*

De volgende stap richting Linux was Minix, geschreven door Andrew S. Tanenbaum. Minix was een erg versimpelde Unix afgeleidde die vooral educatief bedoeld was. De hele source-code, gecombineerd met de theorie waarop deze gebaseerd was, was als boek te koop (in het Turing Lab ligt een exemplaar). Hoewel de source-code beschikbaar was, was het niet toegestaan deze te verbeteren of te verspreiden. Daarnaast was de code vanuit educatief perspectief erg goed, maar praktisch gezien vooral al snel verouderd.

2.1.1 Linux

Linus Torvalds was een student informatica bij de Universiteit van Helsinki. Gebaseerd op het boek van Tanenbaum begon hij in 1991 met zijn eigen OS Kernel. Toen dit een beetje vorm begon te krijgen heeft hij dit op usenet beschikbaar gesteld voor wie er iets mee wilde.

Ondertussen was Richard M. Stallman bezig met een eigen gratis (en vooral: open source) Unix vervanger: GNU (GNU's Not Unix). Dit was grotendeels compleet, alleen de kernel die hij ontwikkelde wilde geen kritieke massa krijgen.

Figuur 2.4 Figuur: Linus Torvalds

Figuur 2.5 Figuur: Richard Stallman (St. IGNUcius)

Omdat Linus bij zijn ontwikkeling gebruik had gemaakt van het werk dat voor GNU verzet was, was het een logische stap de twee projecten te combineren tot GNU/Linux. Hoewel we meestal gewoon Linux zeggen, is GNU/Linux de geprefereerde naam voor het hele OS.

Naast GNU heeft Stallman voor meerdere belangrijke open-source projecten gezorgd, waaronder de editor Emacs; deze wordt door gebruikers als een soort religie ervaren, the Church of Emacs (compleet met holy war tegen de andere grote Linux-editor Vim). Stallman wordt daarbij doorgaans als St. IGNUcius aangeduid.

2.2 Open Source

GNU en Linux vallen beide onder de GNU General Public License, een open source licentie. Dit houdt in dat de source code vrij beschikbaar is, mag worden aangepast en mag worden doorgegeven. Er is een grote diversiteit aan open source licenties in gebruik, en nog meer licenties die ruwweg dezelfde ideeën bevatten maar net niet aan de eisen voldoen om “open source” te mogen heten. Een deel van de open source licenties, waaronder de GPL, heeft een clausule die stelt dat hoewel je de code mag aanpassen, dat de resultaten daarvan ook onder dezelfde licentie moeten vallen. Andere open source licenties hebben deze eis niet. Merk op dat er geen verbod is op het verkopen van GPL software (of diensten daaromheen) of andere commerciële toepassingen (al zijn er licenties die dat wel hebben), zolang de source maar vrijelijk beschikbaar blijft. Later in deze les zullen we een aantal commerciële / betaalde toepassingen van Linux tegenkomen.

2.3 Linux Customiseren

Een van de grote redenen van de populariteit van Linux onder IT'ers is de keuzevrijheid die het platform biedt. Omdat alle onderdelen los van elkaar bestaan en niet (zoals bij Windows en Mac) als één compleet pakket geleverd hoeven te worden heb je erg veel invloed op hoe je besturingssysteem werkt. Daarnaast zorgt het open source model (wat ook voor de meeste pakketten geldt die je bovenop Linux gebruikt) ervoor dat er veel keuzemogelijkheid is in de configuratie van de meeste software. Als een gebruiker een optie mist in software die zij gebruikt, kan ze dit (in theorie) zelf toevoegen en teruggeven aan de community, waardoor de toevoeging mogelijk in de volgende versie van het softwarepakket zal worden opgenomen. Windows (en in mindere mate Mac) wordt vaak vergeleken met een auto waarbij je de motorkap niet open mag maken. Het werkt gewoon, maar het is niet mogelijk of toegestaan zelf je handen vuil te maken. Linux (en andere open source systemen) gaan hier dus tegenin.

2.3.1 Grafische Omgeving

Een eerste keuze die je doorgaans maakt bij het installeren van Linux is de grafische omgeving. In les 0 hebben we al gezien dat de grafische omgeving geen deel van het OS is. Hier zien we een aantal desktop environments en window managers. Desktop environments leveren een complete user experience, zoals de grafische shells bij Windows en Mac. Window managers beperken zich tot het beheren van schermen (waar staat een scherm, hoe groot is het, overlapt het met andere schermen). Hieronder een aantal voorbeelden desktop environments en window managers:

- Desktop environments
  - Gnome (Screenshot)
  - KDE (Screenshot)
  - LXDE (Screenshot)
  - Xfce (Screenshot)
- Window Managers
  - OpenBox (Screenshot)
  - xmonad (Screenshot)

Deze desktop environments en window managers draaien op hun beurt weer bovenop een windowing system, doorgaans X11 (al is Wayland bezig om hier verandering in te brengen). Het windowing system levert een display server, en iedere applicatie met GUI wordt als een client gezien. Een client is in deze context een stuk software dat van een server afhankelijk is voor (een deel van) diens functionaliteiten — in dit geval voor tekenen van de interface op het beeldscherm. Ook de Window Manager (WM), panels en andere componenten van de Desktop Environment (DE) vallen hieronder. De display server krijgt input via de kernel en zorgt ervoor dat de totale inhoud van het beeldscherm bepaald wordt, die dan weer via de kernel naar de grafische kaart wordt gestuurd. X11 is overigens zonder WM of DE te gebruiken, maar dit is wel een vrij kale bedoening.

Figuur 2.6 Figuur: X11 zonder WM

Dus, wat hebben we allemaal nodig om een grafische omgeving te creëren? Naast het windowing system zul je een window manager willen draaien. Deze zijn ruwweg in twee categoriën te verdelen: “normale” window managers met overlappende schermen en een desktop, en “tiling” window managers die de schermen zo indeelt dat het beeld gevuld is en geen enkel scherm overlapt. De laatste categorie is doorgaans wat meer minimalistisch. Naast de window manager kan je een composite manager gebruiken (voor fancy effecten zoals transparante vensters en schaduwen) en één of meerdere bars toevoegen (zoals de taakbalk van Windows of de dock van MacOS). De desktop environment combineert een WM met bars en andere functionaliteiten (al kun je ook een desktop omgeving gebruiken met bijvoorbeeld een andere WM, de combinaties zijn eindeloos). Tot slot heb je doorgaans een display manager nodig; deze levert een login scherm en maakt een sessie als de gebruiker is ingelogd. De display manager is niet per sé noodzakelijk om X11 te gebruiken, het is ook mogelijk eerst in te loggen (via de command line) en later een X sessie te starten. Hieronder een aantal voorbeelden van alle hierboven beschreven componenten:

- Windowing Systems (X11, Wayland)
- Window Managers (Mutter, Metacity, KWin, OpenBox)
  - Tiling WMs (ratpoison, xmonad, i3, awesome)
- Composite Managers (xcompmgr of built-in)
- Bars etc. (i3bar, xmobar, dzen2)
- Desktop Environments (Gnome, KDE, LXDE, Xfce)
- Display Managers (KDM, GDM, LXDM, LightDM, slim)

2.4 De Terminal

Naast een grafische omgeving zul je als power user op Linux veel tijd doorbrengen in de terminal: het tekst-scherm waar je commando's in typt en resultaten te zien krijgt. Dit kan een zogenaamde kernel-level virtuele terminal zijn (virtueel, in tegenstelling tot de fysieke terminals van de oude mainframes) zoals tijdens het opstarten op vol scherm voorbij komt, maar meestal wil je een terminal binnen X11 tot je beschikking hebben. Ook hier is weer een opeenstapeling van applicaties die je naar eigen keuze kan combineren. De zogenaamde "terminal emulator" levert het venster en tekent de terminal daarbinnen, maar doet verder niets met de interactie die je met de terminal zelf hebt: alles dat je intypt wordt naar de shell verstuurd, daar behandeld, en het resultaat wordt weer door je terminal emulator getekend. Emuleren betekent nadoen (deze term zullen we vaker tegenkomen), en een terminal emulator doet een ouderwetse Unix-terminal na. De terminal zelf is in principe een dom ding, en dat is bewust: de virtuele terminal is gebaseerd op de terminal van een mainframe, wat in essentie niet meer was dan een toetsenbord en primitief scherm (of in het begin zelfs een soort printer), en de terminal emulator is daar weer een afgeleidde van. Een kernel-level virtuele terminal bestaat niet als GUI applicatie binnnen X11, maar wordt door de kernel zelf geleverd en is te bereiken met Control-Alt-[F1-F6]. Doorgaans brengt Control-Alt-F7 je weer terug naar de grafische omgeving (ook een soort terminal, maar met GUI in plaats van text-based interface). Het is overigens ook mogelijk binnen de andere virtuele terminals een grafische omgeving op te starten. Bij sommige distributies zijn er meer dan zes kernel-level terminals beschikbaar, en schuift de GUI op naar f8 of verder — F7 is niet "speciaal" voor de GUI.

Een aantal bekende terminal emulators zijn Gnome Terminal (standaard in Gnome), Konsole (standaard in KDE), xterm (verouderd maar op elk systeem aanwezig) en rxvt (veel gebruikt met losse window managers, customiseerbaar maar wat meer minimalistisch).

2.4.1 Shells

Het echte werk gebeurt binnen de shell: dit stukje software levert een command prompt en reageert op je input door programma's of built-in functies te gebruiken. Denk aan de interactieve python sessie: dit is ook een soort van shell, alleen levert deze andere functionaliteit en is deze niet voor algemeen gebruik gemaakt. De meest-gebruikte shell is op dit moment Bash, de "Bourne Again Shell" — een uitgebreide versie van de oudere Bourne Shell. Sh is als oudste shell op alle Linux systemen geïnstalleerd. Power-users maken vaak gebruik van shells die meer te customisen zijn, zoals Csh, Zsh en Fish. Zsh is backwards compatible met Bash en Sh, de andere twee niet. Tot slot bestaan er shells die niet voor interactief gebruik bedoeld zijn, maar om systeemprocessen te opltimaliseren; een voorbeeld hiervan is Dash, de huidige standaard shell voor niet-interactieve shells in Ubuntu. Door geen gebruiksvriendelijke maar daardoor grote shell te gebruiken heeft Ubuntu de opstarttijd flink kunnen minimaliseren: hier worden erg veel (tijdelijke) shells opgestart die nu niet meer

de ongebruikte overhead van Bash hebben. Een shell prompt ziet er als volgt uit:

```
1 user@pi:~$
```

Hier kunnen commando's worden getypt die eenmailig uitgevoerd worden, of interactieve programma's worden opgestart zoals de Python interpreter:

```
1 user@pi:~$ uname -sr
2 Linux 4.17.19
3 user@pi:~$ python
4 Python 3.6.6 (default, Jun 27 2018, 05:47:41)
5 [GCC 7.3.0] on linux
6 Type "help", "copyright", "credits" or "license" for
     more information.
7 >>> _
```

2.4.2 Extra: Terminal Multiplexers

Tussen de terminal emulator en de shell kun je nog een laag toevoegen: de terminal multiplexer. Deze staat je toe je terminal te splitsen (2 of meer terminals naast of onder elkaar), tabs aan te maken binnen de terminal (soms levert de emulator dit ook, dan heb je 2 niveaus van tabs) en de sessie los te koppelen en deze bijvoorbeeld in een ander venster weer aan te koppelen. Dat laatste is bijvoorbeeld zinvol als je via een SSH (SSH is wat je gebruikt om met Putty op je Pi in te loggen: een soort remote desktop, maar dan in de terminal) verbinding van een afstand op je computer inlogt: zodra je verbinding verbroken wordt is de sessie beëindigd. Met een terminal multiplexer wordt deze losgekoppeld, en kan je in een volgende SSH verbinding (of in een terminal emulator) verder waar je gebleven was. Je kunt zelfs met meerdere terminals op dezelfde sessie werken, om bijvoorbeeld iemand mee te laten kijken of samen aan een bestand te werken.

2.4.3 Screencast

In de screencast hieronder zie je de verschillende lagen. Eerst wordt binnen de terminal emulator een nieuw tabblad aangemaakt. Later wordt `tmux` (een terminal multiplexer) gebruikt om tabs en splits te realiseren. Ieder tabblad en iedere split heeft een eigen shell, was te zien is bij het `ps -a` commando: dit geeft een lijst processen en op welke terminal deze vanuit het systeem gezien draaien. Tot slot starten we verschillende shells in elkaar (een shell is immers ook gewoon een programma). Het enige verschil dat zo te zien is, is het commando-prompt, maar natuurlijk zijn er meer verschillen tussen de verschillende shells. Voor alle beschreven lagen (terminal emulator, multiplexer en shell) bestaan binnen de Linux wereld verschillende keuzes, die elk weer naar wens te customiseren zijn.

Screencast at https://old.peikos.net/V1EOS/Images/../Images/terms.webm
Distro's Naast de grafische omgeving en de terminals zijn voor bijna ieder stuk software in Linux verschillende alternatieven beschikbaar. Omdat het

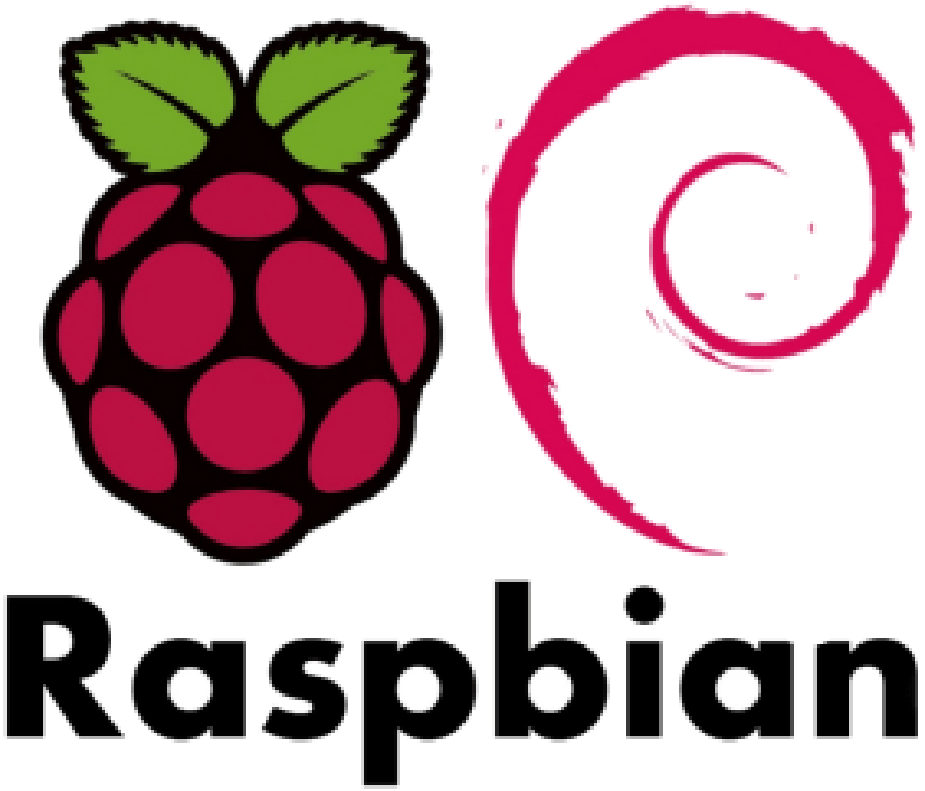
Figuur 2.7 Figuur: Enkele grote distro's

wat omslachtig is die keuze voor elk deel afzonderlijk te maken, en alles zelf te installeren, wordt Linux meestal gebruikt in de vorm van distributies of distro's. Een distributie is een verzameling software, die (meestal met een vriendelijke installer) in een keer geïnstalleerd kan worden. De maker van de distributie maakt de keuzes welke software standaard geïnstalleerd is (of biedt meerdere keuzes aan). De gebruiker kan hierna nog zelf software installeren om het systeem aan de wensen aan te passen, maar er is vast een basis. Ook zorgt dit systeem ervoor dat je met periodieke updates aan je distributie je hele systeem up-to-date kan houden. Een distro vergelijkt zich dus meer met Windows of MacOS als compleet software pakket.

2.4.4 Package Managers

De meeste distro's worden geleverd met een package manager: een systeem dat bijhoudt welke software (en welke versies) geïnstalleerd zijn, en die je kan gebruiken om zelf software aan je installatie toe te voegen of te verwijderen. De package manager houdt in de gaten of je alle benodigdheden hebt geïnstalleerd, en zal indien nodig extra software installeren om een werkend systeem op te leveren. In Debian-afgeleidden heet de package manager `apt`, en kun je software via de command line installeren. Andere distributies gebruiken afgeleiden van `rpm` zoals `yum`, of een eigen package manager. In het voorbeeld wordt `htop` geïnstalleerd, een soort Windows taakbeheer voor de Linux command-line.

Screencast at https://old.peikos.net/V1EOS/Images/../Images/apt.webm
Linux binnen TI



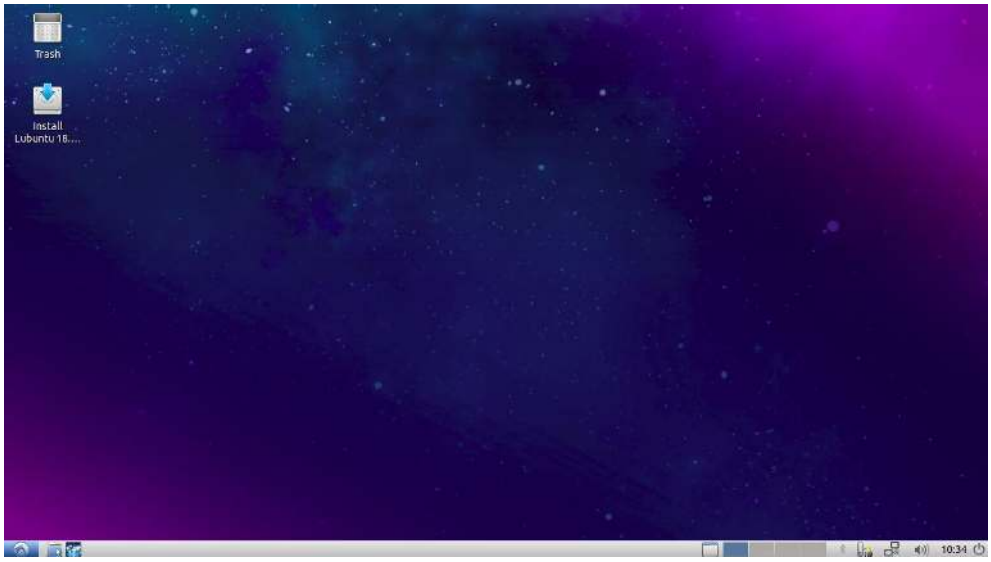
Figuur 2.8 Figuur: Raspbian

2.4.5 Raspbian op de Pi

Bekend van CSN, dit draait op de Raspberry Pi. Raspbian is een R-Pi versie van de populaire distributie Debian.

2.4.6 (L)Ubuntu VM

Voor dit vak is een VirtualBox VM beschikbaar met LUbuntu Linux (link op Canvas). Je kan deze gebruiken voor de practica, maar als je zelf een Linux systeem of VM hebt mag dit natuurlijk ook. De VM is te gebruiken als fallback, de opdrachten zijn op dit systeem getest. Je kan de image naar eigen wensen aanpassen. Let erop dat het sneller is om Linux direct op een computer of laptop te draaien: door het virtualiseren (gebruik van een VM) wordt alles een stuk langzamer. Ook is er bij de VM gekozen voor een vrij basic werkomgeving, ook weer vanwege de performance in de VM. De omgeving is dezelfde als die op de Raspberry Pi standaard geïnstalleerd staat — die is namelijk ook niet zo snel. Voel je vrij het systeem naar je wensen aan te passen, maar hou er rekening mee dat alles binnen de VM wat trager werkt als je de meest flitsende werkomgeving installeert — dan kun je beter Linux op een fysieke computer draaien.



Figuur 2.9 Figuur: Lubuntu

2.4.7 Gentoo (Les 7)

In opdracht 3 zullen we zelf een tweede Linux VM installeren: Gentoo. Gentoo is wat minder toegankelijk dan de meeste distro's: in plaats van een mooie installer moet je zelf de bestanden op de juiste plek zetten en software compileren. Dit maakt het wel een goede oefening om kennis te maken met hoe een Linux systeem in elkaar zit. Gentoo heeft de naam erg moeilijk te zijn (en dat is niet geheel onterecht), met eigenlijk alleen Linux From Scratch als overtreffende trap.

2.5 Andere open-source Unix opties

We hebben nu een hele verzameling gezien van keuzes die je kan maken om een Linux systeem te configureren. Ook Linux is deel van een groter ecosysteem, en niet de enige keuze als je een open-source Unix systeem wil. Alle Unix-achtige systemen hebben een hoge mate van compatibiliteit, waardoor software die hierboven beschreven is vaak ook voor andere, minder gebruikte Unix-varianten beschikbaar is. Andere Unices leggen de focus bijvoorbeeld op veiligheid, en er zijn versies van Unix die commercieel geproduceerd worden (MacOS heeft bijvoorbeeld als basis de open-source Darwin-kernel).

- BSD (FreeBSD, OpenBSD, ...)
- GNU Hurd
- Darwin (basis van MacOS)

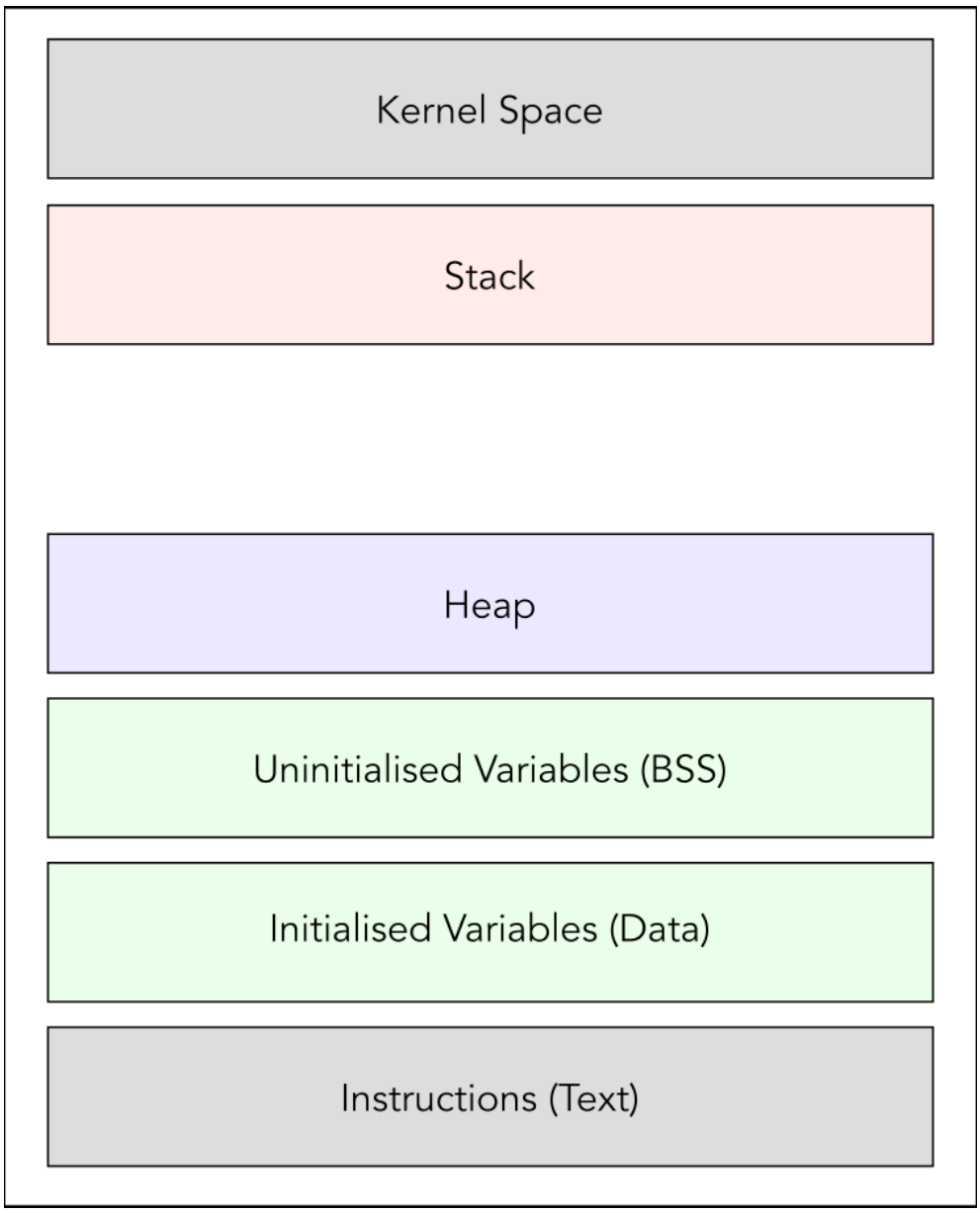
3. Processen

Vorige week zijn we onze tour door de interne werking van het OS gestopt bij het geheugenmodel van een enkele taak. Doorgaans draaien op een computer echter vele taken tegelijkertijd: ieder programma heeft op z'n minst een eigen taak (en soms veel meer, bijvoorbeeld een taak per tabblad in de Browser). Deze les gaan we kijken hoe de computer hiermee om kan gaan.

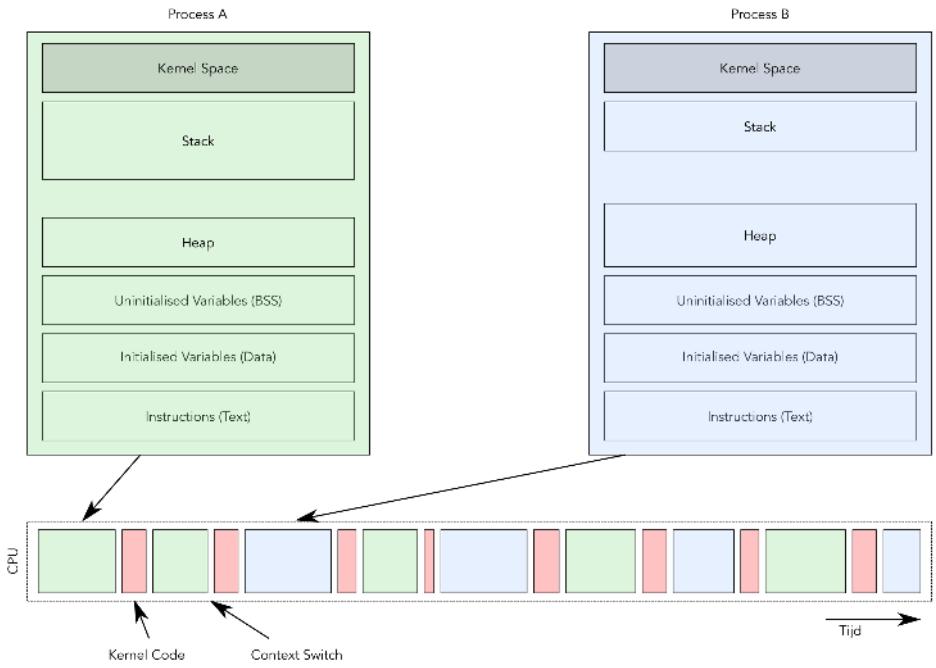
3.1 Processen

Het geheugenmodel van vorige week komt mooi overeen met hoe een proces er van binnen uitziet, alleen zijn we niet tot een enkele taak beperkt. De kernel code is in alle processen aanwezig, maar staat maar op één plek in het geheugen. De CPU wisselt waar nodig tussen beide processen, waarbij alleen het geheugen van het huidige proces in beeld is. Op het moment dat proces A draait dan worden de heap, stack, BSS, text en data van proces A gebruikt. Als er gewisseld wordt naar proces B (een zogenaamde context-switch) worden de heap, stack, BSS, text en dat van proces B gebruikt.

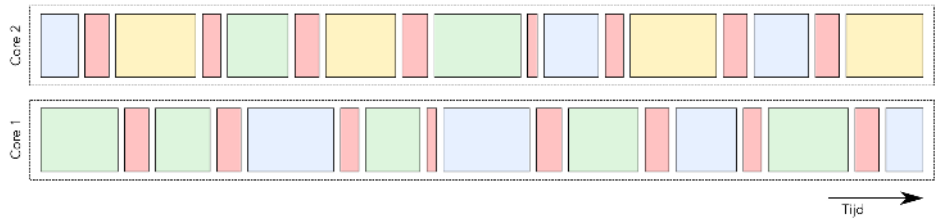
In Figuur Twee Cores zien we meerdere processen op twee cores. De rode blokjes staan voor kernel code, de gele, groene en blauwe zijn drie verschillende processen. Bij meerdere cores draait hetzelfde proces nooit op twee cores tegelijkertijd. Kernel code kan wel op meerdere cores actief zijn, waarom is dat? Het verschil tussen kernel-code en een proces is dat het proces een heel script afloopt. De CPU houdt bij, bij welke instructie het proces gebleven is. Als twee cores tegelijk hetzelfde proces zouden draaien, dan gaat dit door elkaar loopt, of worden instructies dubbel uitgevoerd. De kernel-code wordt aangeroepen voor een bepaalde kernel-functie, die zichzelf daarna weer beëindigt. Als een andere core ook iets van de kernel nodig heeft, dan kan dit prima naast elkaar gebeuren (tenzij beide kernel-functies hetzelfde geheugen aanpassen, of hetzelfde device aansturen). Zo lang de kernel dit bijhoudt kunnen botsingen voorkomen



Figuur 3.1 Figuur: Geheugenmodel Proces



Figuur 3.2 Figuur: Meerdere Processen

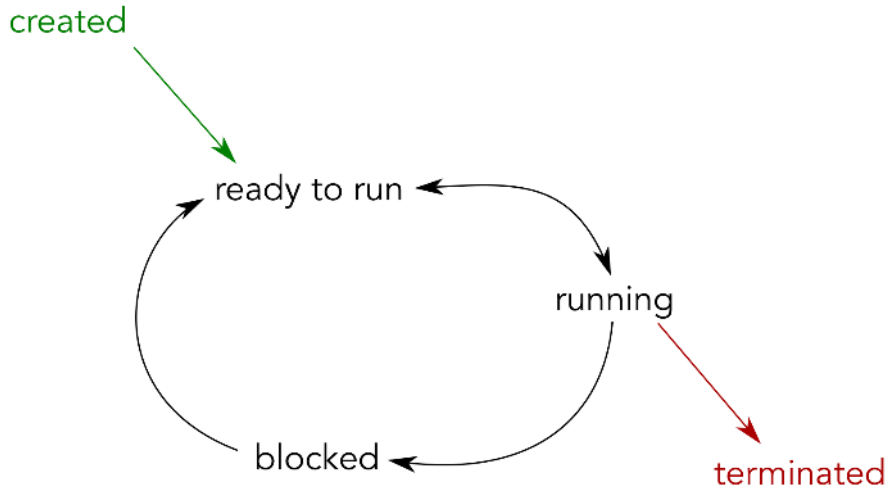


Figuur 3.3 Figuur: Twee Cores

worden, en hoeft de ene core niet op de andere te wachten om een kritieke taak uit te voeren. Dit kan, omdat de kernel het totaaloverzicht over het systeem heeft en weet welke resources in gebruik zijn.

Process State

Een proces kan zich in meerdere toestanden bevinden. Een proces dat op dit moment CPU-tijd heeft, noemen we running. Als een proces niet actief is, omdat de CPU iets anders heeft te doen, dan noemen we dit ready-to-run of soms waiting. Die laatste naam heeft echter niet de voorkeur, omdat deze ook soms voor de derde state wordt gebruikt: blocked. Als een proces deze state heeft dan kan het proces niet verder omdat het op nieuwe data moet wachten, zoals bijvoorbeeld een toetsaanslag van de gebruiker of gegevens via een trage netwerkverbinding. Bij het bereiken van een van deze gevallen (ready-to-run en blocked) vindt een context-switch plaats en zal de processor met een ander proces verder gaan. Afhankelijk van de reden van deze switch zal het OS het process opnieuw op de CPU activeren zodra het aan de beurt is, of zodra het de benodigde gegevens heeft om verder te gaan.



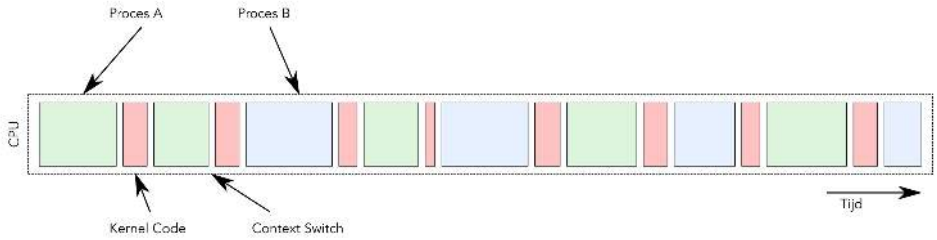
Figuur 3.4 Figuur: Process State Diagram

Threads

Een process kan meerdere subprocessen of threads bevatten en zo meerdere taken "tegelijkertijd" uitvoeren. Het afwisselen van threads gaat een stuk sneller dan het afwisselen van processen, en alle threads binnen een proces delen grotendeels dezelfde data. Iedere thread heeft een eigen stack en eigen registerwaardes, maar deelt verder alle resources. Ook de code wordt gedeeld, al is er wel voor iedere thread een eigen Program Counter (PC) of IAR (Instruction Adress Register, zoals we hem in CSN genoemd hebben). Threads worden bijvoorbeeld gebruikt om de GUI en functionaliteit van een programma te scheiden: het programma kan verder terwijl de GUI geblokeerd is omdat deze wacht op gebruikersinvoer, en de GUI blijft responsive als de andere threads lang bezig zijn.

3.1.1 Multitasking

Bij het draaien van meerdere processen (of het aanroepen van een SysCall) moet de CPU constant vrijgemaakt worden voor een nieuwe taak. De oude informatie kan echter niet worden weggegooid, want deze is later weer nodig als de bijbehorende taak weer aan de beurt is. Dit noemen we een context switch: het OS zet alle state van het lopende proces opzij en laadt de nieuwe informatie. De informatie wordt opgeslagen in een Process Control Block (PCB) of Switchframe, en bevat de inhoud van de registers (zoals de Program Counter) informatie over het proces, en informatie over welk geheugen bij het proces hoort. Dit laatste betekent dat het geheugen zelf niet verplaatst hoeft te worden, maar dat de CPU enkel de verwijzingen aan hoeft te passen.



Figuur 3.5 Figuur: Context Switch

Scheduling

Of we nu met één of meerdere cores te maken hebben, het zal niet snel genoeg zijn om alle processen tegelijk te draaien. Het OS zal dus moeten bepalen hoeveel tijd iedereen heeft en wie wanneer aan de beurt is. Hiervoor zijn twee stijlen te onderscheiden: cooperative multitasking, waarbij de context switch (zie Figuur Twee Cores) door het proces wordt geïnitieerd en dus zelf bepaalt hoe lang het de CPU vast houdt, en pre-emptive multitasking, waarbij het OS na een bepaalde tijd een proces onderbreekt om te kijken wie er nu aan de beurt is. Wat zijn de voordelen van de twee stijlen, en wat zijn de nadelen? Wanneer zou je cooperative multitasking gebruiken? En wanneer niet? Het voordeel van cooperative multitasking is dat het simpeler is: het OS hoeft alleen maar de context switch uit te voeren, en niet te bedenken wanneer deze nodig is. Ook is het in sommige kleine systemen soms zo dat twee taken elkaar afwisselen, waarbij de ene taak van de andere afhankelijk is. In dit geval heeft het geen zin taak A af te breken om tijd voor taak B te maken: taak A moet af zijn voordat taak B kan starten. In systemen met meerdere gebruikers, of processen die niets met elkaar te maken hebben, is cooperative multitasking een minder goed idee: een proces heeft vaak geen enkele reden de beurt over te geven, waardoor een enkel proces in theorie eeuwig kan blijven draaien. Dit kan zelfs per ongeluk gebeuren, als door een programmeerfout een verzoek tot context switch gemist wordt. Voor een modern desktop OS met honderden processen van allemaal verschillende programmeurs is er doorgaans onvoldoende vertrouwen om met cooperative multitasking te kunnen werken.

Pre-emptive Multitasking

We zoomen even in op pre-emptive multitasking. Hierbij ziet de context-switch er als volgt uit. De CPU interrupt wordt gedaan met behulp van een timer, zodat het lopende proces na een bepaalde tijd onderbroken wordt. De interrupt voert een simpele context-switch uit, van het proces naar de interrupt handler van het OS. Deze zal de rest van de process-state bewaren, eventuele achterstallige OS taken uitvoeren (het OS kan immers een kritieke taak voor het proces uitvoeren terwijl de timer afgaat) en de scheduler aanroepen. De scheduler heeft een lijst van alle processen. De scheduler weet welke processen belangrijker zijn en hoe lang geleden ieder proces aan de beurt is geweest. Op basis van deze factoren wordt een nieuw proces geselecteerd, waarna de state voor dat proces wordt klaargezet. Als dit gebeurt is kan het proces exact verder

waar het gebleven was. Het proces bij Pre-emptive Multitasking gaat als volgt:

1. CPU interrupt
2. Interrupt bewaart PC
3. Interrupt roept handler aan
4. Handler bewaart process state
5. Handler doet z'n werk
6. Handler roept de scheduler aan
7. Scheduler kiest een proces
8. Scheduler herstelt process state
9. Scheduler start proces

Realtime

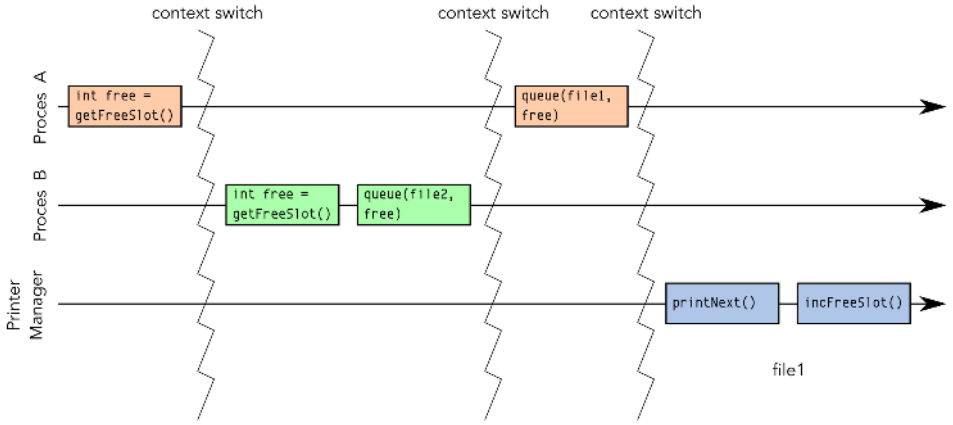
Het schedulen is het meest complex wanneer we met realtime applicaties te maken hebben. Dit zijn processen waarbij een deadline geldt die gehaald dient te worden. We vergelijken vier processen. Het verschil in hoe belangrijk de taken zijn is voor het wel of niet realtime zijn niet van belang: het gaat om het belang van de deadlines binnen de taak. Op basis van dit belang vallen de vier processen in vier verschillende categoriën in te delen.

- Hard realtime
- Firm realtime
- Soft realtime
- Not realtime

Als eerste voorbeeld nemen we een computer die een maanlander bestuurt. Dit is een voorbeeld van een *hard realtime* systeem: Het missen van een enkele deadline kan leiden tot een crash, waarmee het hele resultaat in een klap (letterlijk) waardeloos is geworden. Het tweede voorbeeld is Spotify. Hier spreken we van *firm realtime*: het missen van een deadline is acceptabel, zo lang het niet te vaak gebeurt. Vanaf een bepaald punt is het resultaat waardeloos: als Spotify te veel frames mist is het resultaat niet meer als muziek te herkennen. De sensor-controller van een weerstation is een voorbeeld *soft realtime*: de deadlines mogen zo af en toe gemist worden, en hoe vaker de deadline niet gehaald wordt, des te groter is het waardeverlies. In dit geval zal de voorspelling op minder data gebaseerd zijn, en de waarde neemt af wanneer meer metingen worden gemist. In tegenstelling tot bij firm realtime is het resultaat hier echter niet in een keer waardeloos: de waarde daalt geleidelijk. Tot slot hebben we een lange berekening met weerdata. Dit is (doorgaans) een voorbeeld van *not realtime*. Er is geen deadline, en hoewel het vervelend is als dit te lang duurt, maakt een vertraging het resultaat niet meteeen onbruikbaar (al ligt dit er natuurlijk wel aan waar de berekening toe dient).

Realtime Scheduling Strategiën

Afhankelijk van het type realtime en de lengte van de deadlines kan de CPU verschillende strategiën gebruiken voor het schedulen van taken. De belangrijkste vormen zijn Rate Monotonic Scheduling (RMS) en Earliest Deadline First (EDF). Bij RMS wordt de prioriteit van een taak gebaseerd op hoe lang deze taak nodig heeft. Kortere taken hebben een hogere prioriteit en kunnen vaker



Figuur 3.6 Figuur: Shared Memory

aan de beurt komen. Bij EDF wordt gekeken naar de deadline van een proces: wanneer moet het proces resultaten opleveren. Processen die minder speling hebben komen eerder aan de beurt.

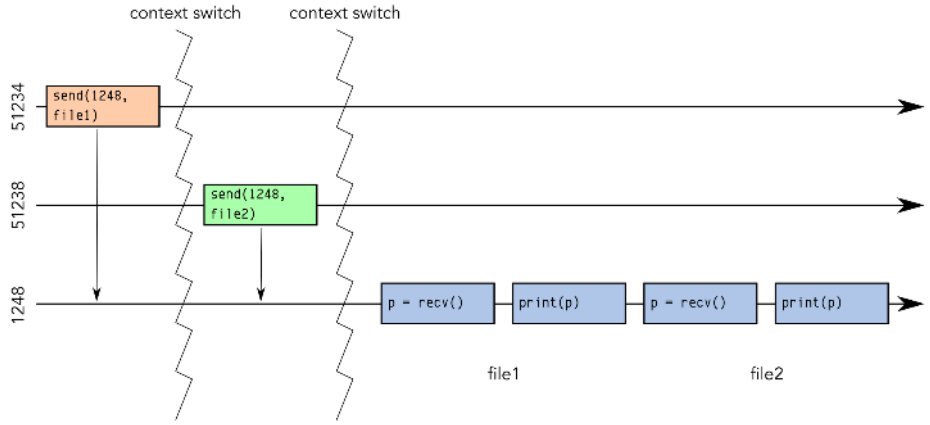
3.1.2 Inter Process Communication

Vaak moeten meerdere processen samenwerken om een doel te bereiken. Als je een mail binnenkrijgt dan zal dit bijvoorbeeld door je mailprogramma worden binnengehaald. Ondertussen zorgt een ander proces ervoor dat je een notificatie in je scherm krijgt, en is er mogelijk nog een aparte notificatielijst, of een widget op het bureaublad, of... Met andere woorden: dezelfde informatie is vaak in meerdere processen nodig. We hebben eerder gezien dat ieder proces zijn eigen geheugen heeft, die niet voor andere processen toegankelijk is. Om toch info te kunnen delen kennen we twee belangrijke varianten. Shared Memory en Message Passing.

Shared Memory

In eerste instantie werd voor dit soort problemen vooral gebruik gemaakt van shared memory: stukken geheugen die door meerdere processen te lezen en schrijven zijn. Dit kan echter makkelijk misgaan; stel bijvoorbeeld het volgende scenario voor: Twee processen willen beide een bestand afdrukken. Proces A is aan de beurt, en zoekt de eerste vrije plaats in de wachtrij. Op dit moment gebeurt er een context switch, en is Proces B is aan de beurt. Ook deze zoekt de eerste vrije plaats, en zet op dat geheugenadres (hetzelfde adres als A had gevonden) een verwijzing naar een te printen bestand. Na de volgende context switch is A weer aan de beurt, die op hetzelfde adres een ander bestand zet. Het originele bestand is overschreven, zonder dat A of B dit gemerkt heeft.

Gelukkkig zijn hier oplossingen voor, maar die voegen een hoop complexiteit toe en zijn vaak lastig toe te passen, zeker in grotere systemen. Voorbeelden hiervan zijn mutexes (Mutual Exclusion) en Semaphores (vlaggen die aangeven dat een stuk geheugen in gebruik is). Elk heeft voor en nadelen, maar hier gaan we deze cursus verder niet op in.



Figuur 3.7 Figuur: Message Passing

Alternatief: Message Passing

Een alternatief voor shared memory is message passing: beide processen communiceren via een centraal proces naar een printer manager proces, dat alleenrechten heeft op het betreffende stuk geheugen. Zowel de printer manager als de processen communiceren via system calls. Proces A en B gebruiken de `send` system call met het proces-ID (PID) van de print manager (1248), waarbij een message wordt meegegeven. Zodra de printer manager aan de beurt is kan deze met de `recv` system call berichten lezen. In Figuur Message Passing wordt geen PID meegegeven aan `recv`, door dit wel te doen kan de eerste message van een specifieke verzender geselecteerd worden. Zowel `send` als `recv` kunnen *synchroon* (blocking) en *asynchroon* (non-blocking) aangeroepen worden. By een synchrone `send` wacht het proces tot de hele boodschap succesvol verzonden is, en kan het programma tot dat moment niet verder naar de volgende instructie. Het resultaat van de `send` (is het versturen gelukt?) is direct als return-value beschikbaar. Bij een asynchrone call gaat het programma gelijk verder, en moet het programma zelf aan het OS vragen of de boodschap verstuurd is. Het versturen wordt door een aparte thread gedaan, die zichzelf beëindigt zodra het bericht verstuurd is. Voor `recv` wordt bij synchroon gebruik gewacht tot er een bericht is: de software kan niet verder als er geen bericht af te handelen is, en zal dus blokkeren tot er iets binnenkomt. Bij en asynchrone `recv` kan het resultaat een message zijn, als er eentje klaar staat, zo niet krijgt de software een `null`-waarde (met andere woorden: niets). Voor het versturen van messages wordt in de meest simpele vorm een buffer gebruikt: een vaste hoeveelheid geheugen die door het OS voor dit doel gereserveerd is. Als deze buffer vol zit en er wordt een message verstuurd, dan zal de verzender blokkeren tot er plaats is; bij een asynchrone `send` is de verzender een aparte thread voor het versturen van de message, niet de main thread (die kan gewoon verder). Bij `recv` wordt juist geblokkeerd als de buffer leeg is. In simpele systemen kan ook zonder buffer gewerkt worden. In dit geval is alleen synchrone communicatie mogelijk: de verzender moet het bericht versturen en de ontvanger moet het bericht ontvangen voordat de processen verder kunnen.

Beide variaties zijn in de meeste moderne operating systems aanwezig, en voor elk van beide types bestaan er verschillende vormen.

- Shared Files
- Sockets / Unix Domain Socket
- Message Queue
- Pipes

Shared memory wordt bijvoorbeeld soms via bestanden toegepast. Voor message passing zijn onder andere Unix Domain Sockets aanwezig: een variatie op de sockets die met netwerkcommunicatie gebruikt worden, maar dan speciaal voor binnen het OS. Met het CSN miniproject hebben jullie mogelijk al sockets gebruikt om twee Pi's aan elkaar te verbinden. Een ander message-passing mechanisme is een pipe, een directe verbinding tussen twee processen die ook als bestand aan te spreken is. Tot slot worden Message Queues of Mailboxes gebruikt om makkelijk een enkel proces (een server) met meerdere client-processen te laten communiceren.

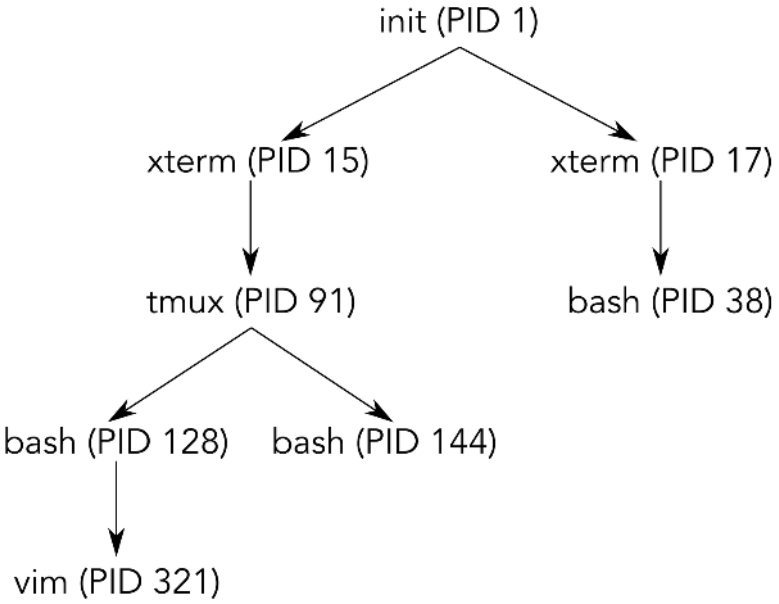
3.1.3 Processen in Linux

Om met meerdere processen te kunnen werken is het van belang te weten hoe deze georganiseerd zijn. We straks kijken naar hoe nieuwe processen in het leven geroepen kunnen worden, waarbij ook de communicatie tussen het nieuwe proces en gerelateerde processen opgezet moet worden. In Linux zijn processen in een tree georganiseerd: ieder proces heeft een parent, met uitzondering van het *init* proces dat de root van de tree vormt. Dit proces wordt als eerste gestart en heeft PID 1. Alle processen stammen van init af. Als een proces met levende kinderen doodgaat dan zou dit tot een split in de process-tree leiden, daarom worden deze "wezen" door init geadopteerd.

Process Files

Ieder proces heeft een eigen Process ID (PID): een getal dat het OS gebruikt om naar een proces te refereren. Voor ieder proces is in het Linux filesysteem een map met virtuele bestanden aanwezig. Dit is deel van de Unix filosofie: alles is een bestand. De bestanden bevatten informatie zoals de programmacode, de argumenten waarmee het bestand gestart is, en de stack van het proces. We gaan hier later meer van zien zodra we het Linux filesysteem gaan bekijken, maar dit is vast een voorbeeld waar we dit principe in actie zien. Op andere besturingssystemen is deze informatie ook in het geheugen aanwezig, maar is er geen bestand-achtige interface om dit makkelijk in te zien zoals op Linux.

- `/proc/PID` process directory
  - `/proc/PID/exe` (bevat een link naar de executable)
  - `/proc/PID/cmdline` (een tekstbestand met daarin het aangeroepen commando)
  - `/proc/PID/environ` (een overzicht van de environment, zoals variabelen die de shell heeft meegegeven)
  - `/proc/PID/stack` (een weergave van de call stack)
  - ...



Figuur 3.8 Figuur: Process Tree

Processen Starten en Stoppen

Het opstarten van een nieuw proces gaat in Linux een beetje merkwaardig. Als gebruiker merk je hier niets van: je klikt op een applicatie-icoon of typt een commando in. Vanuit het perspectief van het OS ziet dit er anders uit: er moet het een en ander gebeuren. Met `fork()` kan een proces zichzelf geheel dupliceren, waarna je twee vrijwel identieke processen hebt die verder gaan na de `fork` call. Het enige verschil is de return-waarde van de `fork`: voor het parent proces (denk terug aan de tree) zal dit de Process ID (PID) van de kopie zijn, maar voor het kind is deze 0. Dit klinkt wat omslachtig, maar het valt mee: beide processen delen hetzelfde geheugen totdat er ergens iets veranderd wordt, pas dan wordt er een kopie gemaakt van het geheugendeel in kwestie. Het geheugen is als copy-on-write (CoW) gemarkeerd, en op het moment dat een van beide processen probeert te schrijven volgt een interrupt, waarna de kernel ervoor zorgt dat het benodigde geheugen gekopieerd wordt.

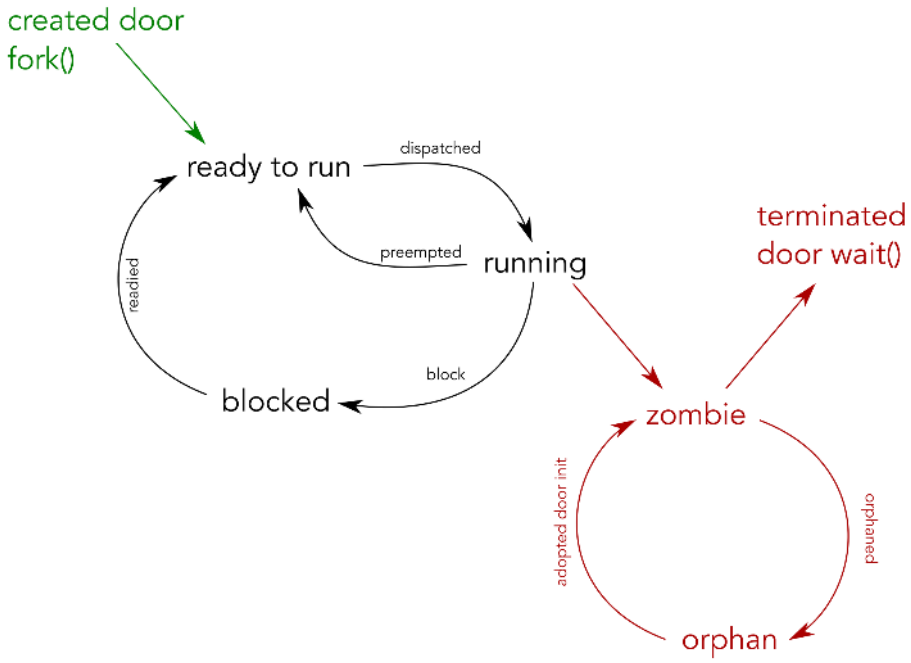
```
1  if (syscall(SYS_fork) == 0)
2  {
3    // This is the child
4    // Do child stuff
5  }
6  else
7  {
8    // This is the parent
9    // Do parent stuff
10 }
```

Hoewel het mogelijk is beide geforkte processen met dezelfde code door te laten draaien, is dit doorgaans niet hoe `fork` gebruikt wordt. In plaats daarvan zal een van beide processen, doorgaans het kind, een `execve` system call doen om een andere executable uit te voeren. Het gehele geadreseerde geheugen wordt weggegooid, en de code uit de executable wordt ingeladen als text segment. De data en bss worden geïnitialiseerd, en het proces heeft een lege heap en stack. Het programma begint met de `main()` van de executable. Het parent proces kan de originele code hervatten, maar zal in veel gevallen wachten tot het kind doodgaat. Hoewel dit wat tragisch klinkt, is het in Unix een groter probleem als de parent komt te overlijden terwijl de child nog leeft: in dat geval hebben we een orphan (weesproces), dat door het init process geadopteerd moet worden. Om op het overlijden van een proces te wachten wordt de `wait` system call gebruikt. Hierbij wordt de PID van het kind meegestuurd zodat het OS weet op welk proces er gewacht wordt. Als de parent besluit niet op het kind te wachten, en dit komt te overlijden, dan zal het kind niet direct verwijderd worden. Het proces is welliswaar beëindigd, maar komt nog wel in de procestabel voor tot de parent het overlijden heeft geregistreerd door de exit-code te lezen. Tot die tijd wordt dit kind een zombie-proces genoemd. Dit is niet wenselijk, omdat de tabel die het OS gebruikt om de processen bij te houden een maximale grootte heeft, en zombies blijven in de tabel tot ze zijn opgeruimd. Om te voorkomen dat de tabel volloopt en er geen nieuwe `fork` gedaan kan worden is het van belang dat overleden processen worden opgeruimd. Het bijgewerkte diagram van process states staat in Figuur Process State Diagram.

```
1  int pid = fork();
2  if (pid == 0)
3  {
4    // Execute another process
5    char *args[] = {"/bin/ls","-r", "-t", "-l", (char *)
         0 };
6    execv("/bin/ls", args);
7  }
8  else
9  {
10   // Wait for child to die
11   int exit_status;
12   wait(&exit_status);
13 }
```

Een proces dat klaar is zal zichzelf beëindigen met behulp van de `SYS_exit` call. Deze krijgt een argument mee van het type `int`, die vertelt waarom het proces gestopt is. Dit wordt de exit-code genoemd. Een exit-code van `0` betekent dat alles goed is gegaan. Een ander getal duidt op een error, waarbij het specifieke getal gebruikt kan worden om de foutmelding aan te geven. Hiervoor is verder geen conventie, de betekenis van de overige exit-codes verschilt van programma tot programma.

```
1  syscall(SYS_exit, 0);
```



Figuur 3.9 Figuur: Process State Diagram

```
2  // of
3  _exit(0);
```

4. De Bash Command Line

Deze les gaan we de Linux command line verkennen. We maken kennis met Bash, de meest gangbare shell voor Linux. Ook komen we een diversiteit aan programma's tegen die we vanaf de command line kunnen gebruiken. Het is in Bash niet altijd duidelijk welk commando een "built-in" functie van Bash is, en welk command een programma is: omdat Bash gebruikt wordt om programma's aan elkaar te knopen is deze grens behoorlijk vervaagd. Deze kennis is meestal ook niet nodig om Bash te gebruiken, maar wel iets om in het achterhoofd te houden.

4.1 Je weg vinden op de command-line

Als je een nieuwe terminal start, heeft je shell standaard een working directory: het punt op het filesysteem waar je je bevindt. Bestanden die je maakt worden in deze map gezet, en bestanden die je zoekt kunnen relatief vanaf deze map benaderd worden. Standaard zul je beginnen in je *home* directory, de map die je als gebruiker zelf beheert en waar je eigen bestanden (en instellingen) te vinden zijn. Het commando `cd` (change directory) wordt gebruikt om van map te wisselen, en `pwd` (present working directory) vertelt waar je je nu bevindt. Met `ls` (list) kun je de inhoud van de huidige map te zien krijgen. Een aantal paden zijn belangrijk om te weten: de bovenste directory, de root directory, wordt met een `/` (slash) aangeduid. Vanaf hier worden directory-namen door een `/` gescheiden om een pad uit te drukken. De home directory van de gebruiker `jon` bevindt zich bijvoorbeeld op `/home/jon` (de map `jon`, binnen de map `home`, binnen de root). `~` wordt gebruikt als synoniem voor de home directory van de huidige gebruiker. Voor Jon is `cd ~` dus hetzelfde als `cd /home/jon`. `.` verwijst naar de huidige map: `cd .` doet niets. Dit kan voor nu een beetje overbodig lijken, maar later zullen we zien hoe de `.` soms korter is dan de alternatieven. Daarnaast hebben we `..`, die naar de parent van de huidige directory verwijst.

Vanaf Jon's home zal `cd ..` dus naar `/home` verwijzen. Deze kun je ook middenin een pad gebruiken, de map omhoog wordt dan gerekend vanaf waar in het pad deze wordt gebruikt. `cd /home/jon/Documents/../Music` is bijvoorbeeld hetzelfde als `cd /home/jon/Music`. `cd` zonder argument gaat terug naar de home directory, en `cd -` gaat terug naar de vorige directory.

```
[jon@Snow:~]$ cd Documents

[jon@Snow:~/Documents]$ pwd
/home/jon/Documents

[jon@Snow:~/Documents]$ cd ..

[jon@Snow:~]$ cd Folder

[jon@Snow:~/Folder]$ ls
AnotherFile  File

[jon@Snow:~/Folder]$ cd ~

[jon@Snow:~]$ cd -
/home/jon/Folder

[jon@Snow:~/Folder]$ cd .

[jon@Snow:~/Folder]$ cd /

[jon@Snow:/]$ ls
bin/  boot/  dev/  etc/  home/  media/  mnt/  proc/
   root/  run/  sys/  tmp/  usr/  var/

[jon@Snow:/]$ cd

[jon@Snow:~]$ pwd
/home/jon
```

De volgende commando's manupileren bestanden. Met `touch` wordt een bestand aangemaakt (als deze nog niet bestond) of wordt de "laatst aangepast" datum naar nu gezet. Het bestand wordt gewijzigd, maar er verandert niets aan. Met `cp` kun je een bestand kopiëren, en met `mv` kun je het verplaatsen. `rm` verwijdert een bestand.

```
[jon@Snow:~/Folder]$ ls
AnotherFile  File

[jon@Snow:~/Folder]$ touch File3

[jon@Snow:~/Folder]$ ls
AnotherFile  File  File3

```

```
[jon@Snow:~/Folder]$ cp File File2

[jon@Snow:~/Folder]$ ls
AnotherFile  File  File2  File3

[jon@Snow:~/Folder]$ mv AnotherFile File1

[jon@Snow:~/Folder]$ ls
File  File1  File2  File3

[jon@Snow:~/Folder]$ rm File3

[jon@Snow:~/Folder]$ ls
File  File1  File2
```

De meeste commando's voor bestanden werken niet direct op directories. `mv` mag, maar `cp` en `rm` werken niet. Voor het verwijderen van een *lege* map wordt `rmdir` gebruikt, maar dit mag niet als er een bestand in de map zit. Dit is omdat Linux je wil beschermen geen verkeerde handeling uit te voeren: als je aangeeft dat je een bestand wil verwijderen, maar je geeft een map op, dan is de kans groot dat je de verkeerde naam hebt meegegeven. Als je toch een map (met alle inhoud) wilt kopiëren of verwijderen, kun je `cp -r` en `rm -r` gebruiken. De toevoeging `-r` staat voor *recursief* (dat betekent in dit geval: met alles wat eronder hangt). Gebruik bijvoorbeeld `rm -r Music` als je de map Music met alle inhoud wil verwijderen. `-r` is het eerste voorbeeld van een flag.

```
[jon@Snow:~/Folder]$ ls
File  File1  File2

[jon@Snow:~/Folder]$ mkdir Subfolder

[jon@Snow:~/Folder]$ touch Subfolder/File

[jon@Snow:~/Folder]$ mkdir EmptyFolder

[jon@Snow:~/Folder]$ ls
EmptyFolder/  File  File1  File2  Subfolder/

[jon@Snow:~/Folder]$ mkdir EmptyFolder2

[jon@Snow:~/Folder]$ ls
EmptyFolder/  EmptyFolder2/  File  File1  File2
   Subfolder/

[jon@Snow:~/Folder]$ rmdir EmptyFolder

[jon@Snow:~/Folder]$ rmdir Subfolder
rmdir: failed to remove 'Subfolder/': Directory not
   empty

```

```
[jon@Snow:~/Folder]$ cp Subfolder Backup
cp: -r not specified; omitting directory 'Subfolder'

[jon@Snow:~/Folder]$ cp -r Subfolder Backup

[jon@Snow:~/Folder]$ ls
Backup/  EmptyFolder2/  File  File1  File2  Subfolder/

[jon@Snow:~/Folder]$ rm -r Subfolder

[jon@Snow:~/Folder]$ rm -r EmptyFolder2

[jon@Snow:~/Folder]$ ls
Backup/  File  File1  File2
```

4.1.1 Flags / Command Line Arguments

De meeste commando's accepteren command line argumenten. Allereerst zijn er verplichte argumenten. Waar `pwd` bijvoorbeeld geen argumenten nodig heeft, heeft `touch` enkel zin als meegegeven wordt welk bestand gemaakt of geüpdatet moet worden. `cp` heeft zelfs twee argumenten nodig: een bron en een doel. Soms kan een commando een optioneel pad meekrijgen: `ls` werkt in de huidige working directory, `ls /home` leest de inhoud van `/home`. Al deze argumenten zijn positie-bepaald. Het eerste argument bij `cp` is altijd de bron, de tweede het doel. Naast deze positiegebonden argumenten hebben we ook flags, die met één of twee streepjes (dashes) kunnen beginnen. Flags komen doorgaans overeen met booleaanse waardes: wel of niet. Als de flag met één dash begint, dan is het doorgaans een enkele letter, en kun je deze combineren. `-r -f` is hetzelfde als `-rf`; het maakt niet uit of je `rm -r -f Music` gebruikt of het kortere `rm -rf Music`. Ook de volgorde maakt meestal niet uit. Flags met een dubbele dash bestaan uit meerdere letters en zijn niet samen te voegen. Tot slot kunnen deze flags soms een argument erbij nemen. Bij enkele letters volgt het argument meestal na een (optionele) spatie, bij de dubbele dash wordt meestal een `=` gebruikt om de waarde van de flag te scheiden.

```
[jon@Snow:~/Folder]$ ls
File  File1  File2  Folder1/  Folder2/ Folder3/

[jon@Snow:~/Folder]$ rm -r Folder1    # Recursive om
   directory te deleten

[jon@Snow:~/Folder]$ rm -f File2      # Force remove

[jon@Snow:~/Folder]$ rm -r -f Folder2 # Force
   recursive remove

[jon@Snow:~/Folder]$ rm -rf Folder3   # Zelfde als -r
   -f

```

```
12 [jon@Snow:~/Folder]$ rm --force File1 # Zelfde als -f
13
14 [jon@Snow:~/Folder]$ ls
15 File
```

Sommige commando's hebben erg veel opties om de gewenste functionaliteit te selecteren. De meeste commando's hebben een `-h` en/of `--help` optie om wat tips te geven over het gebruik. Voor het hele verhaal kun je `man` gebruiken, oftewel de manual.

```
1 [jon@Snow:~]$ rm --help
2 # Print een (relatief) korte handleiding
3
4 [jon@Snow:~]$ man rm
5 # Scroll door de uitgebreide handleiding
```

4.1.2 IO en Pipes

Om een tree te krijgen van de inhoud van een map en alle submappen kun je het commando `find` gebruiken. Dit geeft echter vrij veel uitvoer, het zou fijn zijn als we hier hapklare brokken van konden maken. Het commando `less` kan gebruikt worden om een bestand te openen en er doorheen te kunnen scrollen. Kunnen we dit gebruiken om de uitvoer van `find` te pagineren? In Linux kun je twee commando's aan elkaar koppelen: uitvoer van het ene programma wordt als invoer voor het tweede gebruikt. Hiervoor gebruiken we het `|` symbool: de pipe. Denk aan een pijpleiding die de commando's met elkaar verbindt.

```
1 [jon@Snow:~]$ find /home/jon
2 # Print de hele mappenstructuur; veel output
3
4 [jon@Snow:~]$ less file
5 # Lees een bestand door er doorheen te scrollen
6
7 [jon@Snow:~]$ find /home/jon | less
8 # Combineer de tools: scroll door de uitvoer van find
```

`SYS_pipe`

Pipes zijn een fundamenteel onderdeel van de Unix filosofie om eenvoudige programma's te combineren voor complex gedrag. De pipe is dermate belangrijk dat hiervoor een speciale system call bestaat: `SYS_pipe`. Hieronder zien we de stappen die de shell doorloopt bij het uitvoeren van een `find | less`.

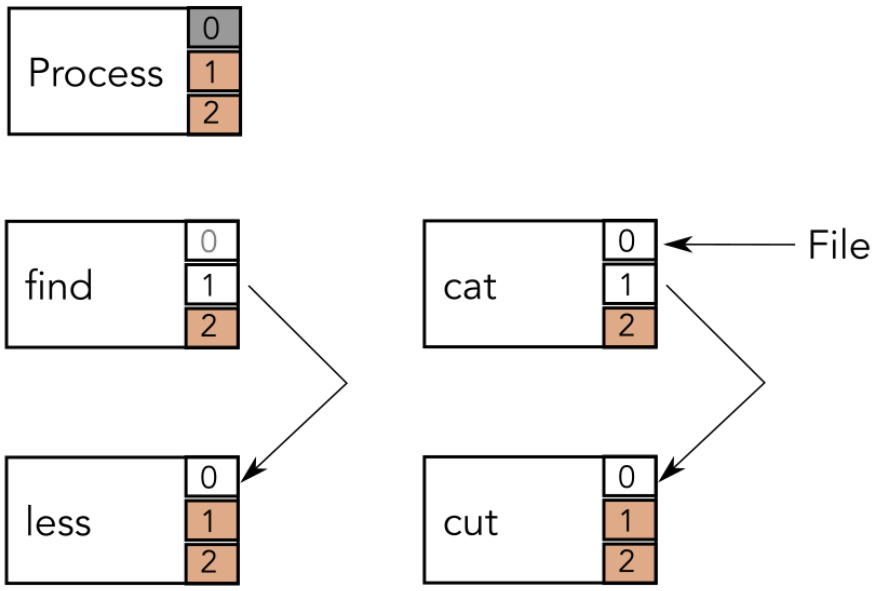
1. Shell maakt een pipe met `pipe()`
2. Shell `fork`t twee keer
3. Child 1 stuurt uitvoer naar pipe
4. Child 2 leest invoer uit pipe
5. Child 1 roept `exec()` aan met `find`
6. Child 2 roept `exec()` aan met `less`
7. Parent wacht tot beide children `exit`en

STDIN en STDOUT

Het sturen van de uitvoer noemen we het *redirecten* van STDOUT (standard out), en het lezen van de invoer is het *redirecten* van STDIN (standard in). Dit hoeft niet met een pipe te gebeuren, maar kan ook van/naar een bestand. STDOUT is een file descriptor (nummer 1) die in ieder proces aanwezig is, en die gebruikt wordt om alles naartoe te schrijven dat het programma uitvoert (denk aan prints). STDIN is ook een file descriptor (nummer 0), maar deze wordt gelezen voor invoer. Normaal is dit de gebruikersinvoer vanaf de command line, maar dit kan ook prima een bestand of pipe zijn. Daarnaast is er vaak een derde file descriptor, STDERR (standard error, nummer 2), voor foutmeldingen (ook dit is standaard de command line, maar het is hiermee dus mogelijk te splitsen tussen uitvoer en excepties). Het gebruik van file descriptors voor de invoer/uitvoer is weer een voorbeeld van de UNIX-filosofie dat alles een bestand is. Een descriptor is de datastructuur die UNIX intern gebruikt om iets te kunnen benaderen; een file descriptor verwijst dus naar een bestand (en hoe dit gelezen/geschreven kan worden). Waar STDIN en STDOUT precies aan verbonden zijn, is te vinden in de environment informatie die bij een proces hoort. Standaard zijn dit voor processen die je in de shell start de invoer en de uitvoer van de terminal, en als je een child process start worden deze overgeërfd. Tijdens executie van het proces kunnen deze descriptors altijd worden gesloten of ergens anders aan verbonden worden. Bij processen die door de kernel gestart worden, zoals init is de STDOUT doorgaans aan het kernel log gekoppeld. STDIN kan aan een buffer of (als het proces geen input verwacht) aan een dummy file gekoppeld zijn: een lege file descriptor die niet aan een daadwerkelijk bestand op de schijf gekoppeld is.

IO Redirection

Net als dat we vanaf de shell twee commando's aan elkaar kunnen pipen, kunnen we ook een file descriptor meegeven om als STDIN, STDOUT of STDERR te gebruiken. Dit kan respectievelijk met <, > en 2>. Je kunt bijvoorbeeld find /home/jon > filetree gebruiken om de uitvoer van find /home/jon in het bestand filetree te schrijven. Bij find /home/jon 2> errors.log wordt niet de uitvoer verbonden, maar eventuele foutmeldingen via STDERR. Als het bestand waarnaar je wilt uitvoeren al bestaat, wordt het als je > gebruikt overschreven, maar je kunt, >> en 2>> gebruiken om in plaats daarvan achteraan het bestand toe te voegen. Tot slot kun je << MARKER of <<- MARKER (een zogenaamde "heredoc") gebruiken om STDIN vanaf de command line mee te geven, tot aan de eerste regel met enkel MARKER. Dit maakt het makkelijk meerdere regels tekst mee te geven. De toegevoegde waarde van de - is dat deze een tab aan het begin van een regel negeert en mogelijk leesbaarder is. Op zichzelf wordt < niet zo veel gebruikt, omdat de meeste programma's standaard een meegegeven bestandsnaam als STDIN interpreteren (dus less < file is hetzelfde als less file), maar bij sommige programma's kan dit nodig zijn. Een voorbeeld is psql, een database waarbij je een bestand met instructies mee kan geven met psql databasename < sqlfile. Als je < niet gebruikt en vanaf



Figuur 4.1 Figuur: File Descriptors van een Proces

de command line invoert kun je `Control-d` gebruiken om je invoer te beëindigen, in plaats van de explicete `MARKER` die je bij `<<` gebruikt. `wc` staat trouwens voor *word count*, waarbij `wc -l` het aantal regels telt, en `wc -c` het aantal karakters.

```
1  [jon@Snow:~]$ find /home/jon > filetree 2> errors.log
2  # Geen output; alles staat in bestanden filetree en
       errorslog
3
4  [jon@Snow:~]$ wc -l filetree >> filetree
5  # Het totaal aantal regels in filetree wordt onderaan
       filetree toegevoegd
6
7  [jon@Snow:~]$ wc -c << EOF
8  > Lorem Ipsum
9  > Dolor Sit Amet
10 > EOF
11 27
12
13 [jon@Snow:~]$ psql databasename < sqlfile
14 # Laadt een database uit een file
```

Hier zien we hoe redirection er vanuit het shell proces uit ziet. Als voorbeeld gebruiken we het commando `command a1 a2 < in_file > out_file 2> /dev/null`: `/dev/null` is een speciaal virtueel device, waar altijd naar geschreven kan worden. `/dev/null` gedraagt zich als een zwart gat: alles dat ernaar geschreven wordt verdwijnt. In dit geval onderdrukken we dus error messages door alle `STDERR` uitvoer naar de prullenbak te verwijzen. Zowel `< in_file` als `> out_file`

en 2> /dev/null hebben invloed op het commando command a1 a2, en worden niet onderling met elkaar verbonden anders dan via het command a1 a2-proces. Merk op dat bestanden in Linux lang niet altijd een extensie hebben (zoals .txt of .doc). Linux bepaalt meestal aan de hand van de inhoud met wat voor type bestand het te maken heeft. In dit geval zijn in_file en out_file waarschijnlijk tekstbestanden.

1. De shell forkt
2. Child sluit STDIN en opent in_file als STDIN
3. Child sluit STDOUT en opent out_file als STDOUT
4. Child sluit STDERR, opent /dev/null als STDERR
5. Child roept exec() aan met command en geeft a1 a2 als argumenten mee
6. Parent wacht tot het kind exit

4.1.3 Meer Commando's met Pipes

Nu we het bestaan van pipes kennen, zijn hier nog wat commando's die op zichzelf niet altijd even zinvol zijn, maar als deel van een pipeline goed werken. Om uit bestand (of uitvoer) regels te filteren die een bepaalde string bevatten kun je grep gebruiken. sort geeft de invoer gesorteerd terug, en cut snijdt per regel een substring uit. In het onderste voorbeeld wordt ls -l gebruikt (een directory listing met uitgebreide info). We gebruiken cut -c14- om karakter 14 tot eind te bewaren, waardoor de eigenaar van het bestand vooraan komt te staan. Met sort kunnen we hierop sorteren.

```
[jon@Snow:~/Folder]$ ls -la
total 0
drwxr-xr-x 1 jon users   44 Jan 28 10:10 .
drwx------ 1 jon users 6142 Jan 28 10:05 ..
-rw-r--r-- 1 root   users    0 Jan 28 09:46 File
-rw-r--r-- 1 jon users    0 Jan 28 10:08 File2
-rw-r--r-- 1 jon users    0 Jan 28 10:08 File2018
-rw-r--r-- 1 jon users    0 Jan 28 10:08 File3

[jon@Snow:~/Folder]$ ls -la | grep 2018
-rw-r--r-- 1 jon users 0 Jan 28 10:08 File2018

[jon@Snow:~/Folder]$ ls -la | cut -c14- | sort

jon users    0 Jan 28 10:08 File2
jon users    0 Jan 28 10:08 File2018
jon users    0 Jan 28 10:08 File3
jon users   44 Jan 28 10:10 .
jon users 6142 Jan 28 10:05 ..
root   users    0 Jan 28 09:46 File
```

4.1.4 Pipelines en Command-lists

Meerdere commands gescheiden door pipes vormen samen een pipeline. De commando's worden parallel uitgevoerd. Je kunt ook meerdere commands na

elkaar uit laten voeren, dit kan met de puntkomma en wordt een command-list genoemd. Het is zelfs mogelijk meerdere pipelines in een command list te zetten.

```
[jon@Snow:~]$ foo | bar
# De uitvoer van foo gaat naar bar

[jon@Snow:~]$ foo ; bar
# bar wordt na foo uitgevoerd

[jon@Snow:~]$ foo | bar ; baz | quux
# Eerst worden foo en bar uitgevoerd, waarbij de
    uitvoer van foo naar bar gaat
# Als dit klaar is worden baz en quux gestart, en gaat
     de uitvoer van baz naar quux
```

Commands in een command list worden na elkaar uitgevoerd, ongeacht of er ergens iets mis gaat. Soms wil je het tweede command enkel uitvoeren als het eerste command geslaagd is, of juist alleen als er een fout is geweest. Zoals we weten heeft ieder command een exit status, `0` voor succes of iets anders voor een fout. Door `&&` tussen commands te zetten, in plaats van `;` wordt het tweede commando alleen uitgevoerd als het eerste geslaagd is. De dubbele pipe `||` werkt precies andersom: het tweede command wordt alleen uitgevoerd als de eerste mislukte. We noemen deze operators *en* en *of*. "Doe a en b" kan alleen maar als *a* gelukt is, zo niet zal de shell het meteen opgeven. "Doe a of b" levert de shell een alternatief, probeer eerst *a*, en als dat niet lukt, doe dan *b*. ::: end study

```
[jon@Snow:~]$ foo || bar
# bar wordt uitgevoerd als foo faalt

[jon@Snow:~]$ foo && bar
# bar wordt uitgevoerd als foo gelukt is
```

4.1.5 Procesmanagement

Als je wilt weten welke processen er op dit moment actief zijn kun je het commando `ps` gebruiken. Normaal geeft dit alleen een lijst met je eigen processen in terminals, maar met `ps -A` krijg je alle processen te zien. De standaarduitvoer geeft (afhankelijk van je distributie) in ieder geval de PID, terminal en het command. Vaak wordt ook de effectieve tijd die het proces gedraaid heeft meegegeven (effectieve tijd telt alleen de tijd dat het proces daadwerkelijk actief was). Daarnaast heeft `ps` nog heel veel opties om processen te selecteren en aan te geven welke informatie je wel en niet wil hebben, zie daarvoor de `man` pagina. `ps` geeft een eenmalige uitvoer van de processenlijst op dat moment; dit is handig als je dit met andere commands in een pipeline wil combineren. Soms wil je echter een dynamische lijst die zichzelf update, zoals het taakbeheer in Windows. Hier is het commando `top` voor, of de verbeterde versie `htop`

(meestal niet standaard geïnstalleerd). Tot slot kan je een proces vanaf de command line beëindigen (op voorwaarde dat je zelf de eigenaar van het proces bent). Hiervoor gebruik je `kill` met de PID van het proces, of `killall` met de naam van het proces. De laatste heet killall, omdat het goed kan zijn dat je meerdere instanties van dezelfde executable open hebt staan. `killall firefox` sluit bijvoorbeeld alle Firefox vensters, niet een specifieke.

```
1  [jon@Snow:~]$ ps
2  # geef de processen binnen de huidige shell
3
4  [jon@Snow:~]$ ps -A
5  # geef alle processen
6
7  [jon@Snow:~]$ top
8  # interactieve "task manager"
9
10 [jon@Snow:~]$ kill
11 # stop een proces met een gegeven PID
12
13 [jon@Snow:~]$ killall
14 # stop elk proces met een gegeven naam
```

4.1.6 Signals

`kill` en `killall` zullen standaard een `SIGTERM` signaal naar een proces sturen: een vriendelijk doch dringend verzoek om op te houden te bestaan. Dit betekent dat het proces de gelegenheid heeft achter zich op te ruimen, de gebruiker te vragen of zij de wijzigingen op wil slaan, of het verzoek te negeren. Er bestaan echter meerdere signalen die je kan sturen, de belangrijksten staan in de tabel op deze slide. Om bijvoorbeeld een `SIGKILL` signaal te versturen (om een proces direct en *with extreme prejudice* om zeep te helpen) kan dit met `kill -SIGKILL`. `SIGSTOP` pauzeert een proces, dat met `SIGCONT` weer hervat kan worden. In de tussentijd is het proces bevroren, en zal het niet gescheduled worden. Een GUI applicatie zal wel een scherm behouden, maar nergens meer op reageren tot het een `SIGCONT` ontvangt. De tabel met beschikbare signalen is groot, en bevat ook signalen die normaal alleen voor intern gebruik zijn: `SIGSEGV` wordt bijvoorbeeld door de kernel gestuurd als een proces buiten zijn geheugenruimte opereert (een Segmentation Fault), en `SIGFPE` wordt onder andere door de CPU gestuurd naar een proces dat door 0 probeert te delen. De hele tabel is in de Linux documentatie of op internet te vinden. De getallen zijn de interne waardes die het OS herkent, de namen zijn enkel om het de gebruiker makkelijker te maken.

SIGKILL(9)	Proces beëindigen, kan niet afgevangen worden
SIGUSR1(10)	User defined, verschilt per programma
SIGTERM(15)	Standaard signaal, “sterf, alsjeblieft”
SIGCONT(18)	Hervat een gestopt (gepauseerd) proces
SIGSTOP(19)	Stop tot je een SIGCONT krijgt

4.1.7 Fore- en Background jobs

Doorgaans zal een shell bij het uitvoeren van een command een `fork()` en een `exec()` uitvoeren, waarna de parent `wait()` tot het kind klaar is. Met behulp van de enkele ampersand (`&`) kunnen we een proces op de achtergrond starten: de shell wacht niet en geeft meteen een nieuw prompt aan de gebruiker. Het nieuwe proces is nog steeds een kind van de shell, en als de shell beëindigd wordt voor het kind klaar is wordt dit ook getermineerd. Met `Control-z` kun je het huidige foreground proces (het proces waarop de shell aan het wachten is) een `SIGSTOP` sturen, waarna de shell weer op de gebruiker kan reageren. Met `fg` krijgt het proces een `SIGCONT` en hervat het de claim op de shell: het wordt naar de voorgrond gestuurd. Je kunt echter ook `bg` gebruiken om het proces op de achtergrond door te laten gaan, net alsof het met een `&` gestart was. `jobs` geeft een lijst met actieve achtergrond processen. Als je een shell met actieve achtergrondprocessen probeert te beëindigen krijg je meestal een waarschuwing: als de terminal sluit stoppen de processen ook. Om dit te voorkomen kun je het proces met `disown` onterven: het is niet langer een kind van de shell waar het gestart is, maar wordt onder het *init* proces gehangen.

```
1  [jon@Snow:~]$ python3
2  Python 3.6.6 (default, Jun 27 2018, 05:47:41)
3  [GCC 7.3.0] on linux
4  Type "help", "copyright", "credits" or "license" for
       more information.
5  >>> antwoord = 42
6  >>>
7  [1]+  Stopped                 python3
                       # Python gepauzeerd ("Stopped")
8
9  [jon@Snow:~]$ ls Folder
                                     # Terug in Bash
10 File File2 File2018 File3
11
12 [jon@Snow:~]$ jobs
13 [1]+  Stopped                 python3
14
15 [jon@Snow:~]$ fg
                                              # Terug in
       Python, waar we gebleven waren
16 python3
                                                      #
       Bash vertelt nog welk proces er draait
17 print(antwoord)
                                               # Geen >>> (
       die staat hierboven al)
18 42
19 >>>

       # Na een instructie wel weer >>>
20
```

```
21 [jon@Snow:~]$ du -chd1  > filesizes
                        # Bereken grootte van elke
   subfolder
22 ^Z

   # Dit duurt te lang...
23 [1]+  Stopped                 du -chd1 > filesizes
         # Gepauzeerd
24
25 [jon@Snow:~]$ bg
                                          # Proces gaat
    op achtergrond door
26 [1]+ du -chd1 > filesizes &
27
28 [jon@Snow:~]$ jobs
                                        # Running in
   joblist
29 [1]+  Running                 du -chd1 > filesizes &
     # Merk op: er is een & toegevoegd
30
31 [jon@Snow:~]$ disown
32
33 [jon@Snow:~]$ jobs
                                        # Geen uitvoer,
    job is onterfd
34
35 [jon@Snow:~]$ du -chd1  > filesizes &
                      # Met & -> Start op achtergrond
36 [1] 20251
                                                # Job
   1, PID 20251
```

4.2 Werken met tekst

We hebben inmiddels een aantal symbolen gezien met speciale betekenis in Bash. De hele lijst is de karakter `# ' " \ $ ‘ * ~ ? < > ( )! & | ;`, spatie en enter. Wat nou als we deze karakters als karakter willen gebruiken, bijvoorbeeld omdat ze in een filename voorkomen? We kunnen de backslash gebruiken om een karakter te “escapen”: bash negeert de speciale betekenis. Een bestandsnaam met een spatie kan je bijvoorbeeld met `bestandsnaam\ met\ spatie` gebruiken (zonder spatie wordt dit als drie losse bestandsnamen beschouwd). Ook kunnen we quotes gebruiken; binnen enkele quotes wordt elk speciaal karakter genegeerd en als karakter gelezen. Bij dubbele quotes geldt dit alleen voor `$ ‘ \ ! *` en `@`. Ons eerste voorbeeld had dus ook bijvoorbeeld als `'bestandsnaam met spatie'` ingevoerd kunnen worden. Variabelen beginnen in Bash met een `$`. De variabele `$HOME` bevat bijvoorbeeld het pad van de home directory. Dubbele quotes escapen de meeste karakters, maar variabelen worden wel geëvalueerd. Het commando `echo` kan gebruikt worden om de waarde van een variabele te

laten zien. `echo` geeft de invoer ongewijzigd terug, maar zal wel variabelen evalueren. `echo $HOME` (of `echo ~`) geeft bijvoorbeeld het pad van de home directory als uitvoer.

We hebben `less` gezien om een bestand te lezen. De uitvoer is echter interactief, wat het ongeschikt maakt om de uitvoer met een pipe door te sturen naar een ander programma. Hiervoor kunnen we `cat` gebruiken: dit leest een meegegeven bestand (of `STDIN`) en stuurt de inhoud naar `STDOUT`. Het lijkt hiermee een beetje op `echo`, maar er zijn belangrijke verschillen. `echo` zal zijn argument returnen, maar doet niets met `STDIN`. Probeer de commando's op deze slide, en kijk wat het verschil is. Wat doet elke pipeline?

```
1  [jon@Snow:~/Folder]$ echo  $HOME              # Echo de
       inhoud van een variabele
2  /home/jon
3
4  [jon@Snow:~/Folder]$ echo "Hallo Wereld"      # Echo
       een string
5  Hallo Wereld
6
7  [jon@Snow:~/Folder]$ cat                      #
       Interactieve echo
8  Hallo Wereld!                                 # Dit
       typt de gebruiker
9  Hallo Wereld!                                 # Dit
       stuurt cat terug
10 ^D                                            #
       Beeindig met Control-D
11
12 [jon@Snow:~/Folder]$ cat File2018             # Lees de
       inhoud van het bestand
13 Hallo uit 2018!
14
15 [jon@Snow:~/Folder]$ echo File2018            # Nu
       wordt de filename als string gezien
16 File2018
17
18 [jon@Snow:~/Folder]$ echo File2018 | cat      # cat
       geeft het resultaat van echo door
19 File2018
20
21 [jon@Snow:~/Folder]$ echo File2018 | echo     # echo
       doet niets met invoer vanuit pipes
```

4.3 root-rechten

Het kan voorkomen dat je probeert een proces te doden of een bestand of map te lezen, en dat Linux dit niet toe laat. Je hebt te maken met een rechtenprobleem. Meestal is dit op te lossen door tijdelijk root (super-user of in Windows termen "administrator") rechten aan te vragen. Hiervoor zijn de commando's `su` en

`sudo`. De eerste geeft je doorgaans superuser rechten door een nieuwe shell als de root-gebruiker te openen, al kun je met `su -c` een enkel commando als root uitvoeren. `sudo` laat je standaard een enkel commando uitvoeren, maar dit mag ook een shell zijn met `sudo bash`. Beide kun je ook gebruiken om als een andere gebruiker dan root te werken: `sudo -u jon` en `su jon` geven je de rechten van Jon. Het belangrijke verschil in de werking is dat `su` het wachtwoord van de doelgebruiker verwacht: het root-wachtwoord voor gewoon `su` of Jon's wachtwoord voor `su jon`. Bij `sudo` gebruik je standaard je eigen wachtwoord, en wordt in een bestand gekeken of je het recht hebt om als de gekozen gebruiker te werken. Het `sudo` commando is erg configureerbaar, je kunt bijvoorbeeld ook alsnog het root-wachtwoord eisen voor root rechten.

```
1  [jon@Snow:~/Folder]$ ls RootFolder
                              # Deze map is niet leesbaar
2  ls: cannot open directory 'RootFolder': Permission
       denied
3
4  [jon@Snow:~/Folder]$ cd RootFolder
                              # Maar mag wel geopend
       worden
5
6  [jon@Snow:~/Folder/RootFolder]$ ls
7  ls: cannot open directory '.': Permission denied
                 # De inhoud is nog steeds onleesbaar
8
9  [jon@Snow:~/Folder/RootFolder]$ sudo ls
                           # Met sudo als root proberen
10 [sudo] password for jon:
                                          # Eigen
       wachtwoord
11 HiddenFile HiddenFolder/
                                          # De mapinhoud
12
13 [jon@Snow:~/Folder/RootFolder]$ su -c ls
                           # Met su als root proberen
14 Password:
                                                        #
       Wachtwoord van root
15 HiddenFile HiddenFolder/
16
17 [jon@Snow:~/Folder/RootFolder]$ sudo bash
                         # Open een shell als root met sudo
18 # Watchwoord niet opnieuw nodig voor aantal minuten
19
20 [root@Snow:/home/jon/Folder/RootFolder]# ls
                        # ls in de root shell
21 HiddenFile HiddenFolder/
22
23 [root@Snow:/home/jon/Folder/RootFolder]# exit
                      # Terug naar de vorige shell
```

```
24
25 [jon@Snow:~/Folder/RootFolder]$ su
                                # Open een shell als root
   met su
26 Password:
                                                          #
   Wachtwoord (root) wel nodig
27
28 [root@Snow:/home/jon/Folder/RootFolder]# ls
                     # ls in de root shell
29 HiddenFile HiddenFolder/
30
31 [root@Snow:/home/jon/Folder/RootFolder]# exit
                   # Terug naar de vorige shell
```

4.4 Lifehacks

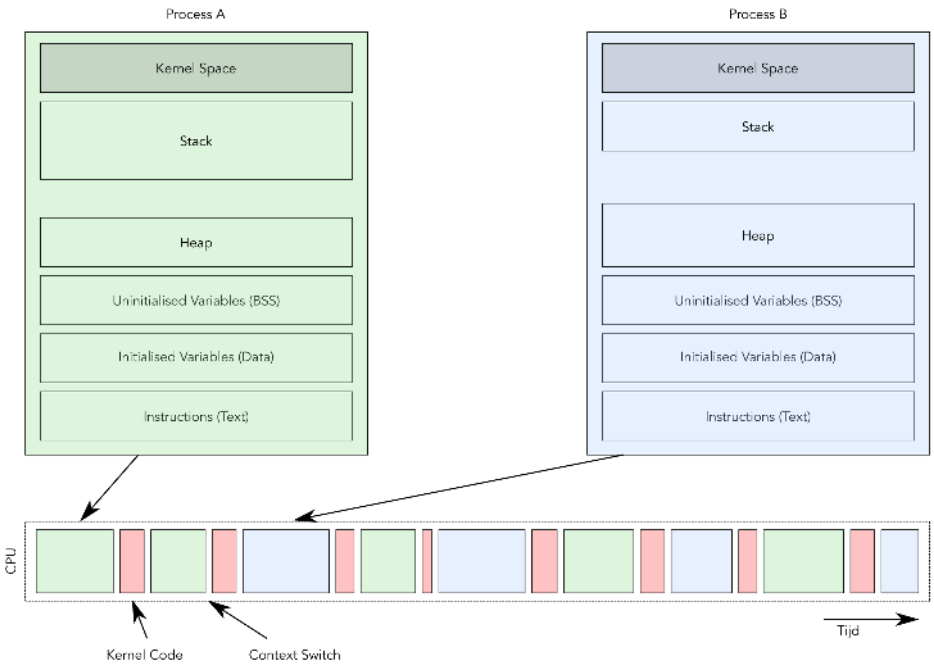
Lange commando' of bestandsnamen kan Bash voor je aanvullen. Dit gebeurt met de Tab toets en heet tab-completion. `*` en `?` kunnen in een bestandsnaam gebruikt worden als soort van wildcards. `*` zal zoveel characters matchen als nodig is (inclusief 0): `File*.png` matcht bijvoorbeeld `File.png`, maar ook `FileFooBarUnicornLasers.png`. `?` werkt op eenzelfde manier, maar zal altijd exact één karakter matchen. Als je veel tussen dezelfde mappen heen en weer beweegt kan `cd -` niet meer voldoende zijn. In dit geval kan het handig zijn een stack (remember, les 0?) van paden te gebruiken. Je kunt `pushd` als alternatief voor `cd` gebruiken om meteen je huidige werkmap op een stack te zetten, en `popd` gebruiken om terug te keren naar de laatste map (het bovenste element van de stack). Het commando `dirs` toont de opgebouwde directory stack. Met de pijltjes naar boven en beneden kan je door je command-history lopen. Het vorige command nogmaals uitvoeren kan door een keer op pijltje naar boven te drukken, en dan op enter. `Control-p` en `Control-n` staan respectievelijk voor previous en next, en kunnen in plaats van boven en beneden gebruikt worden. `Control-r` staat je toe te zoeken in je command history, en met `history` kan je de history bekijken en eventueel entries verwijderen. Om een programma te stoppen kun je `Control-c` gebruiken, en in sommige gevallen `Control-d`. Die laatste stopt bijvoorbeeld je huidige shell. Waar `Control-c` als een letterlijk stop commando geïnterpreteerd wordt, is `Control-d` een end-of-file (EOF) karakter. Omdat je shell de interactie met de gebruiker als de `STDIN` file ziet, kun je deze met een EOF beëindigen en stopt de shell. `Control-z` stuurt een `SIGSTOP` en is ook handig om programma's te onderbreken. Tot slot nog een paar toetsenbordcombinaties die als alternatief gebruikt kunnen worden voor toetsen die mogelijk verder weg zitten: `Control-a` is Home en `Control-e` is End. `Control-j` komt overeen met Enter. `Control-l` komt niet met een toets overeen, maar met het commando `clear`, voor als je de terminal leeg wil maken.

5. Memory Management

Vorige week zijn we onze tour door de interne werking van het OS gestopt bij het geheugenmodel van een enkele taak. Doorgaans draaien op een computer echter vele taken tegelijkertijd: ieder programma heeft op z'n minst een eigen taak (en soms veel meer, bijvoorbeeld een taak per tabblad in de Browser). Deze les gaan we kijken hoe de computer hiermee om kan gaan.

5.1 Wat is geheugen?

Een computer heeft verschillende vormen van geheugen. De registers hebben we in CSN gezien: deze slaan kleine hoeveelheden data op (in CSN 8 bits, in moderne computers doorgaans 64 bits) en kunnen direct door de processor aangesproken worden voor berekeningen. RAM-geheugen is een stuk langzamer: informatie moet worden opgezocht en dan in een register worden gezet om ermee te kunnen werken. Tussen deze twee lagen zit de cache. Deze heeft meerdere levels, tegenwoordig doorgaans 3. De level 1 cache is vrij klein en is erg snel aan te spreken, de level 2 cache is wat groter maar minder snel, en level 3 het langzaamst (maar nog steeds veel sneller dan het RAM) en het grootst (maar nog steeds veel kleiner dan RAM). Level 1 en 2 cache zit in moderne CPUs per core, terwijl level 3 cache door alle cores gedeeld wordt. Van oudsher zat level 1 cache op de CPU en level 2 cache op het moederbord, maar tegenwoordig zit alles tot en met level 3 in de CPU. Je kunt de cache van een CPU zien als een kleine hoeveelheid super-snel geheugen. De cache wordt gebruikt voor data die recent nodig is geweest, en die mogelijk binnenkort opnieuw nodig zal zijn. Gegevens die door de processor gebruikt worden gaan via de cache, waar een kopie bewaard wordt. Als de CPU dezelfde gegevens opnieuw opvraagt, dan wordt dit uit het snelst mogelijke geheugen opgeleverd. De verschillende lagen geheugen kunnen worden vergeleken met fysieke opslagruimte: de registers zijn je handen, die de dingen bevatten waar je nu mee bezig bent. Cache komt

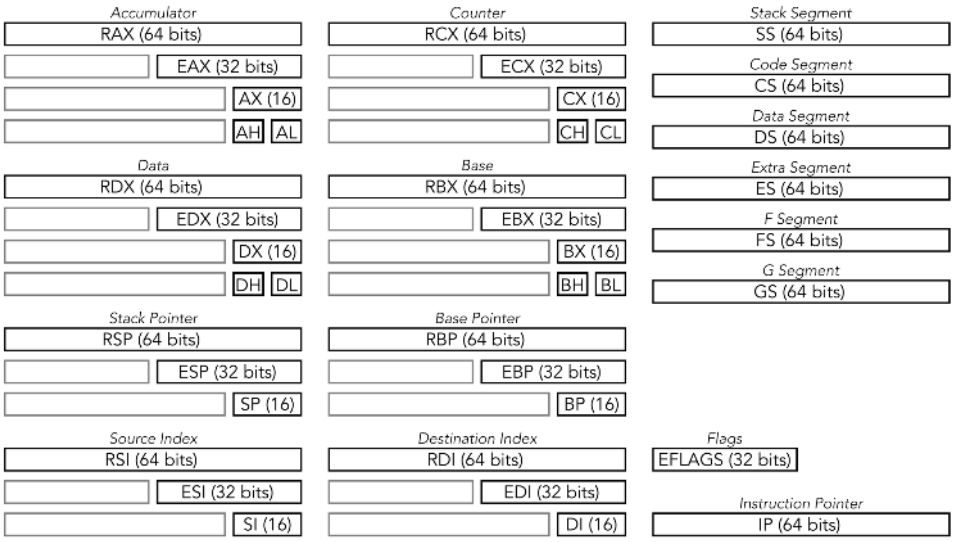


Figuur 5.1 Figuur: Meerdere Processen

overeen met je bureau: level 1 is recht voor je neus, level 3 is op de hoek. Je kunt objecten hier niet direct gebruiken, maar je kunt ze erg snel oppakken. Het RAM geheugen is in deze vergelijking de kast naast je bureau, nog steeds redelijk bij de hand. Tot slot hebben we de non-volatile (niet vluchtig) storage: de harde schijf en/of solid state disk. Beide zijn erg veel groter en minder snel dan het geheugen, maar hebben het voordeel dat ze informatie vasthouden zelfs als de computer wordt uitgezet. In de vergelijking hierboven is de harde schijf een garage waar je dingen voor langere tijd in opslaat. Hoewel dit allemaal vormen van geheugen zijn wordt met de term "geheugen" doorgaans "werkgeheugen" of RAM bedoeld. Let hierbij op!

5.1.1 Registers

Het register hebben we met CSN gezien. Registers bevatten waardes waar de CPU direct bij moet kunnen. We zijn een aantal "general purpose" registers tegengekomen: registers 0 tot 3, het instruction register (IR), instruction address register (IAR) en het memory address register (MAR). Ook de flags die we hebben gezien worden doorgaans in een register opgeslagen; voor de simpele CPU van CSN waren dit er 4 (dus dat past makkelijk in een byte), bij moderne Intel-compatible CPUs zijn dit er 32. Totaal bevat een dergelijke CPU 8 general purpose registers, 6 segment registers (meestal niet meer gebruikt, dit was om aan te geven welk geheugensegment waar in het geheugen begon), een flag register en de instruction pointer. De architectuur is ooit met 16 bits begonnen, en vertoont wat backwards compatibility. Zo isieder general purpose register niet alleen in 64 bits, maar oo in 32 of 16 bits aan te spreken, waarbij je telkens



Figuur 5.2 Figuur: 64 bits x86 registers

de laagste bits krijgt. Daarnaast hebben de accumulator, counter, data en base registers ook twee 8-bits identifiers, voor bits 0-7 en 8-15.

5.1.2 Cache

Tussen de registers en het RAM geheugen zit de cache. Deze wordt op dezelfde manier aangesproken als het geheugen, maar kan recent gebruikte waardes sneller leveren. Cachen verhoogt de snelheid van een computer doordat veel data vaker wordt opgevraagd. Recent gebruikte of berekende data wordt in de cache gezet, en zodra de cache vol zit wordt oudere data verwijderd. Mogelijk is de data nog wel in een andere cache aanwezig, die verder van de CPU verwijderd is. Zolang de data nog één van de caches aanwezig is hoeft het niet opnieuw uit het geheugen gehaald te woreden. De data is dan relatief lang niet nodig geweest, en de kans is groot dat deze niet snel weer gebruikt zal worden. Waar bij normaal geheugen een index bepaalt hoe een stuk geheugen geaddresseerd kan worden, heeft een cache twee indices: de index binnen de cache en een verwijzing naar de geheugenplaats waar de data een kopie van is. De eerste byte in het cache geheugen verwijst naar byte `0x02` in het geheugen.

Memory

Index	Data
0x00	01011101
0x01	10100110
0x02	10000101
0x03	11100110

Cache Memory

Index	Tag	Data
0x00	0x02	10000101
0x01	0x00	01011101

Als de gevraagde gegevens in de cache worden gevonden spreken we over een *cache hit*, en als de gegevens er niet zijn (en dus elders vandaan worden gehaald) dan hebben we een *cache miss*. Het precentage van de requests dat in een hit resulteert noemen we de *hit ratio*. Hoe hoger deze ratio is, hoe sneller de computer functioneert.

bits per adres. In moderne computers is dit meestal 64 bits, waarbij ieder geheugenadres 1 byte bevat. Het totaal adresseerbaar geheugen konden we berekenen: $2^64 \times 1$ bytes. Maar de computer van CSN was nog vrij eenvoudig: die hoefde nog niet met meerdere processen om te kunnen gaan. In de code kan worden verwezen naar specifieke geheugenadressen, waar code of data te vinden is. Maar hoe voorkomen we dat de ruimte waar naar verwezen wordt aan een ander proces toebehoort? Om deze reden werken computers met relatieve adressen. Een verwijzing naar een functie wordt bij het compileren door de computer vertaalt naar een specifiek geheugenadres, maar in plaats van dat dit bij 0 begint is dit gerekend vanaf een bepaalde geheugeneenheid. Het proces is zich hier niet van bewust, en denkt dat het daadwerkelijk naar een vastgesteld adres schrijft.

5.2.1 Segmentation en de MMU

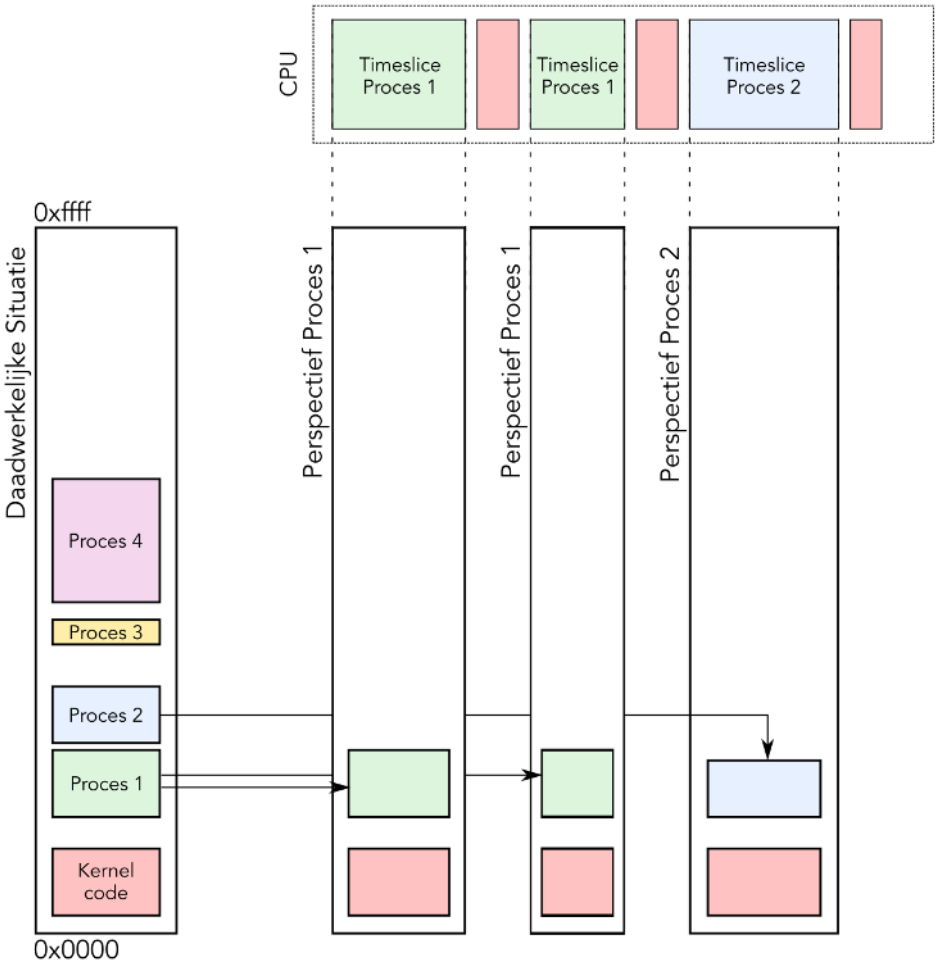
Deze vertaling wordt gedaan door de Memory Management Unit (MMU). Deze weet waar elk proces in het geheugen staat, en telt het relatieve adres hierbij op. Waar in de machine-instructies voor een functie-aanroep een adres als `0x00004203` staat, rekent de MMU het daadwerkelijke adres uit. Als de geheugenruimte van het proces bijvoorbeeld op `0xdb130000` begint, dan is het daadwerkelijke geheugenadres dus `0xdb134203`. Addressen van syscalls hoeven niet vertaald te worden, maar zijn voor ieder proces hetzelfde. Op deze manier wordt dit (read-only) stuk van het geheugen door alle processen gedeeld.

5.2.2 Paging

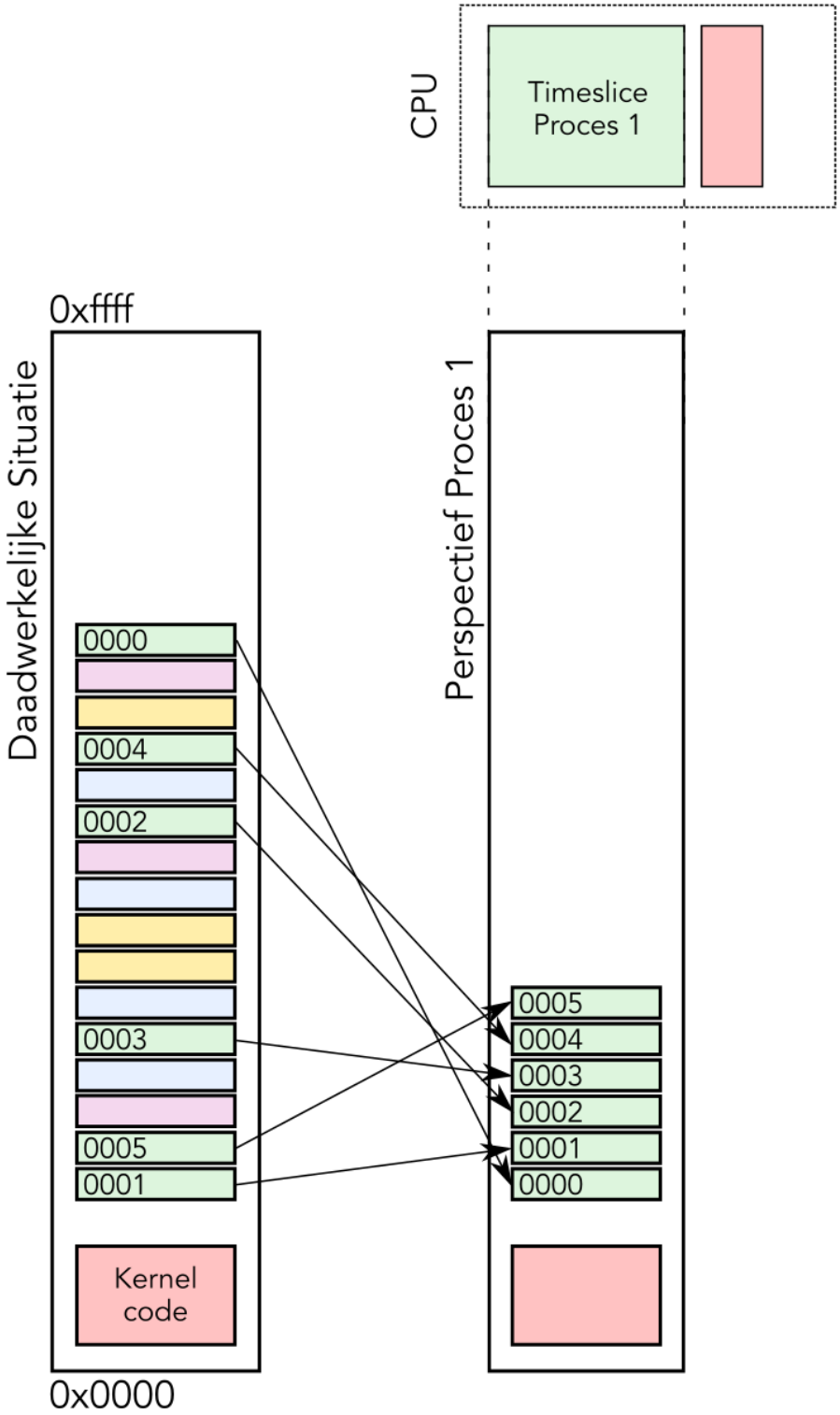
Bij segmentation worden processen als hele blokken geheugen beschouwd, en op die manier vertaald door de MMU. Tegenwoordig zien we doorgaans paging als alternatief. Hierbij wordt met blokken van een vast formaat gewerkt, en kan een proces meerdere pages hebben. De pages zijn ergens in het geheugen aanwezig, maar niet per sé bij elkaar of in de juiste volgorde. Voor het proces lijkt het echter nog steeds alsof het een aaneengesloten stuk geheugen heeft. Een voordeel van deze aanpak is dat als een klein proces beëindigd wordt, de vrijgekomen page (of pages) direct voor een ander proces inzetbaar is. Ook als een nieuw proces te groot is om de ruimte te vullen kan de vrijgekomen ruimte worden gebruikt voor een of meerdere pages. Dit in tegenstelling tot segmentation, waarbij naar verloop van tijd het geheugen vol zit met grote blokken geheugen, met hiertussen onbruikbaar kleine stukken vrij geheugen (dit noemen we *external fragmentation*). Een nadeel is echter dat de pagesize niet flexibel is. Als een proces ook maar één byte meer nodig heeft dan de pagesize toelaat, dan is meteen een volledige nieuwe pagina gevuld. Dit noemen we *internal fragmentation*.

5.2.3 Physical en Virtual Memory

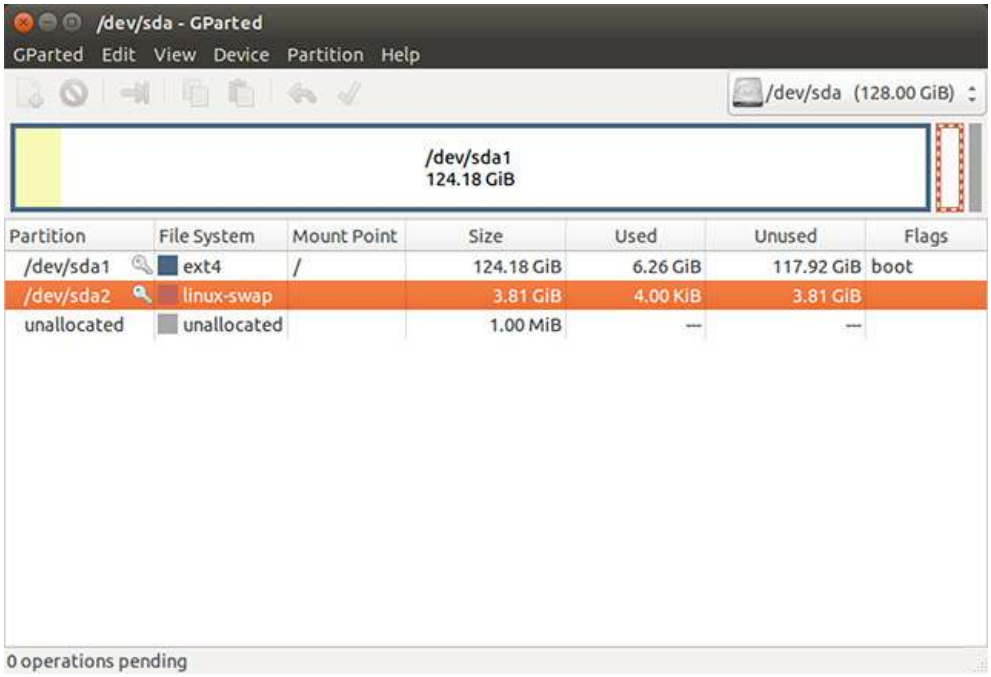
Tot nu toe hebben we telkens het werkgeheugen (het geheugen van het OS en de processen) en het RAM als synoniem beschouwd: het werkgeheugen staat altijd in het RAM. Dit is echter niet per sé nodig: nu we pages hebben kunnen



Figuur 5.3 Figuur: Vertaling adressen door MMU



Figuur 5.4
Figuur: Paging

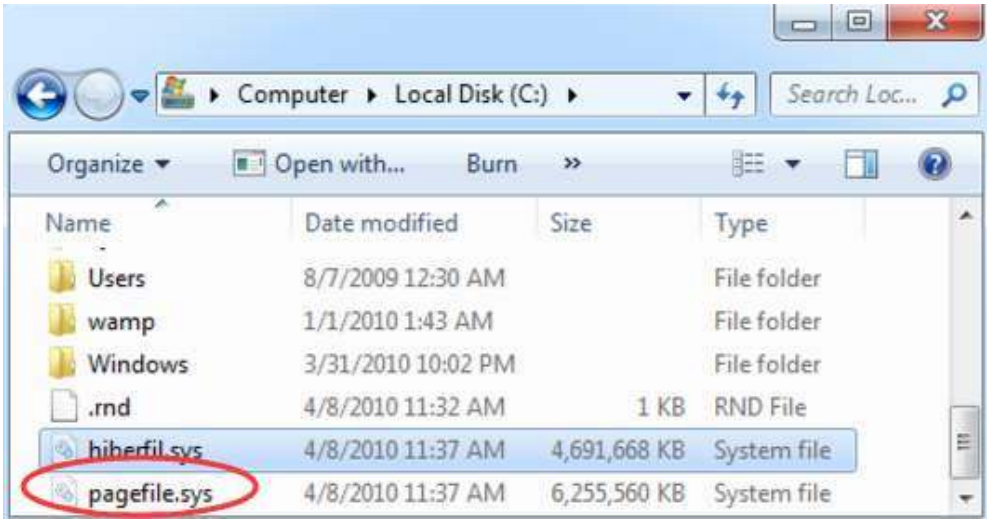


Figuur 5.5 Figuur: Swap-partition in Linux

we die ook op andere plaatsten op slaan. Als het geheugen vol zit kunnen we bijvoorbeeld de oudste (langst niet gebruikte) pagina's uit het geheugen verwijderen en in plaats daarvan op de harde schijf bewaren. Dit is een stuk langzamer te benaderen dan het vanuit het RAM zou zijn, maar dat is nog altijd beter dan een systeem dat onbruikbaar wordt omdat het geen geheugen meer heeft. Een stuk geheugen dat lang ongebruikt is heeft een hoge kans dat het voorlopig nog niet nodig is, waardoor het verlies aan snelheid minder uitmaakt. Als de pagina toch nodig blijkt zal het OS het systeem kort pauzeren om ruimte vrij te maken en de pagina naar het geheugen te verplaatsen. Geheugen op de harde schijf noemen we virtueel geheugen.

In Windows is voor dit virtuele geheugen een speciaal bestand aanwezig, de zogenaamde swapfile of pagefile. Linux (en andere Unices) gebruiken hiervoor doorgaans een aparte partitie, de swap partitie. Deze heeft als nadeel dat deze niet ergens anders voor te gebruiken is, maar is wel sneller omdat het geheugen direct op de schijf wordt opgeslagen, in plaats van dat hier nog een filesystem tussen zit. Omdat Linux echter makkelijk een filesystem in een file kan zetten, kun je in geval van nood ook snel een swapfile aanmaken.

Een laatste voordeel van het paging systeem is dat het mogelijk is op meerdere plaatsen naar eenzelfde stuk geheugen te verwijzen. Dit is hoe in de meeste paging-systemen met de syscall table (en rest van de kernel space) wordt omgegaan: de pages die dit deel van het geheugen bevatten worden bij elk proces vooraan gezet. Een ander voorbeeld hiervan is de `SYS_fork` system-call. Deze maakt een exacte kopie van het draaiende proces, wat ontzettend inefficiënt zou zijn als al het geheugen ook daadwerkelijk gekopieerd zou moeten worden. In plaats daarvan worden dezelfde pages vanuit beide



Figuur 5.6 Figuur: Swap-file in Windows

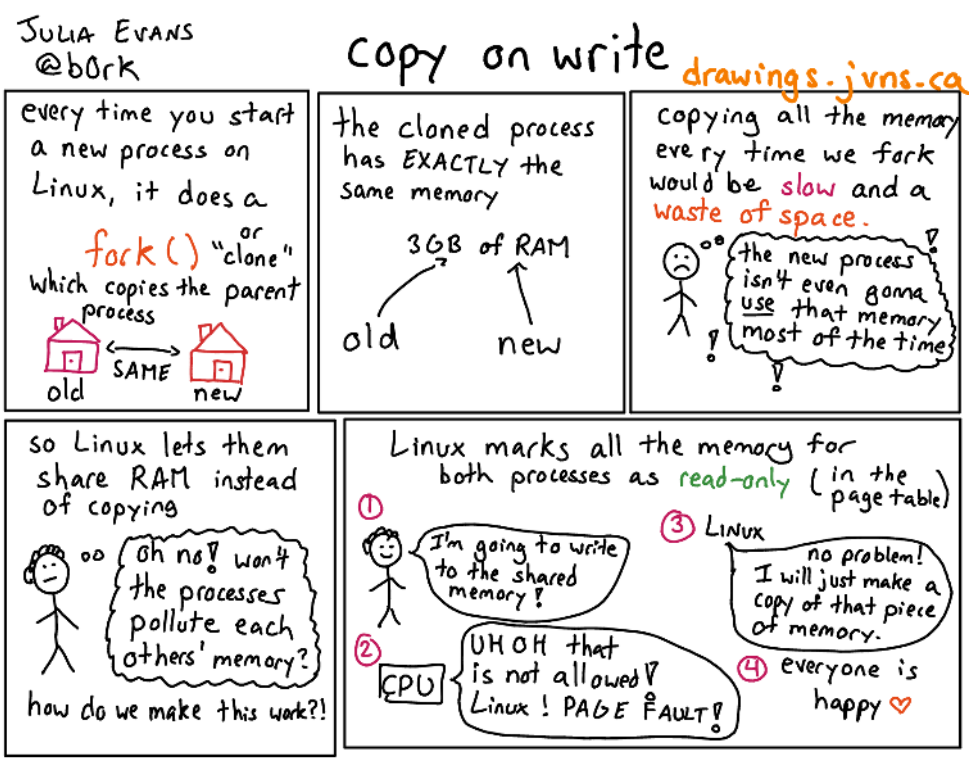
processen aangeroepen, en pas als er iets in een page verandert wordt deze gekopieerd en aangepast. Veel pages bevatten echter programma-code en andere read-only data, dus zullen nooit gekopieerd hoeven worden. Hierom wordt Copy-on-Write toegepast: zolang de data hetzelfde is voor beide processen wordt naar hetzelfde stuk geheugen verwezen, en zodra een proces wijzigingen aanbrengt wordt de relevante pagina gekopieerd zodat beide processen hun eigen versie kunnen behouden.

5.3 Non-volatile Storage

Met virtual memory op de harde schijf hebben we meteen een mooi bruggetje naar de laatste soort geheugen in ons lijstje van het begin. Onder non-volatile geheugen verstaan we harde schijven en SSDs, maar bijvoorbeeld ook USB-sticks, CD-ROMS, Floppy's, etc. Een overeenkomst tussen al deze voorbeelden (en de meeste moderne voorbeelden van non-volatile storage) is dat we hier niet over denken alsof het een groot blok data is, maar als een filestystem met daarop bestanden (doorgaans ingedeeld in directories). In les 6 worden filesystems uitvoerig behandeld.

5.3.1 ROM Geheugen

Een laatste vorm van geheugen die we willen behandelen is het zogenaamde ROM: Read-only memory. Dit is geheugen dat eenmalig geschreven wordt, en daarna alleen maar wordt gelezen. Er bestaan vormen van geheugen die gewist en herschreven kunnen worden, maar ook in dat geval zal het herschrijven slechts erg sporadisch gebeuren. Qua terminologie is ROM de meest simpele vorm: een geheugen dat met inhoud geleverd wordt, en die onveranderbaar is. Denk aan bijvoobeeld een cartridge voor een oude game-console. Programmable ROM wordt leeg gekocht, en kan eenmalig geschreven worden. Hiermee konden mensen bijvoorbeeld hun eigen cartridges maken. De volgende ontwikkeling was Erasable PROM, dat met behulp van UV licht gewist kon worden, en later



Figuur 5.7 Figuur: Forking met Copy-on-Write [Source]

Electronic Erasable PROM. Dat laatste wordt nog steeds gebruikt, bijvoorbeeld voor de firmware van een computer. Bij het opstarten zal een computer eerst inhoud van een stukje EEPROM op het moederbord lezen, waarin de instructies staan die nodig zijn om het systeem verder op te starten. Bij een update kan deze firmware "geflashed" worden (bijvoorbeeld als ergens een fout is ontdekt), maar doorgaans zal dit geheugen alleen gelezen worden. De laatste ontwikkeling op het gebied van deze ROM-chips was uiteindelijk het Flash geheugen: non-volatile, maar wel op bit-niveau aan te passen. Dit is de technologie die aan de basis van SSDs en USB sticks staat.

6. Bash Scripting

Vorige week hebben we gezien hoe we ons kunnen redden op de Bash command line. Vandaag zullen we nog een aantal meer geavanceerde opties zien, maar zullen we vooral de focus verleggen. Waar we de shell vorige les interactief gebruikt, gaan we nu aan de gang met shellscripts: voorgeprogrammeerde programma's om te doorlopen. In plaats van dat `STDIN` direct door ons als gebruiker gevuld wordt, wordt hiervoor een bestand ingelezen. Dit verschil maakt voor de shell in principe niets uit, in beide gevallen is er een lijst met commando's. In het geval van een interactieve moet er zo af en toe gewacht worden op de gebruiker, terwijl bij een script op de harde schijf of het geheugen gewacht moet worden.

6.1 Bash-bestanden

Als we een serie Bash commands in een bestand onder elkaar zetten, hebben we in principe al een simpel script dat we met `bash scriptfile` kunnen uitvoeren. Een scriptfile mag elke extensie hebben, vaak wordt de `.sh` uitgang gebruikt. Ook komt het vaak voor dat er geen extensie gebruikt wordt, om als bona fide programma over te komen. Dat laatste heeft vooral zin als we het programma direct met de naam (e.g. `scriptfile`) aan kunnen roepen. Gelukkig is dat makkelijk te doen, al moeten we hier wel twee stappen voor uitvoeren: we moeten Linux vertellen wat voor code het bestand bevat, en dat het uitgevoerd mag worden. Dat eerste doen we met het commando `chmod +x scriptfile`. We zullen later verder naar `chmod` kijken, zodra het rechtensysteem aan bod komt, maar voor nu kunnen we dit lezen als "voeg de eXecutable bit to voer het bestand `scriptfile`". Daarnaast moet Linux kunnen zien dat het bestand Bash code bevat, en niet Python of C++ of nog iets anders. Het moet weten welke *interpreter* gebruikt moet worden. In Windows wordt doorgaans de extensie gebruikt om een bestandtype te herkennen, maar omdat in Linux de koppeling

tussen bestand en bestandsnaam wat losser is (zoals we in Les 7 zullen zien) kijkt Linux liever naar de inhoud van het bestand. Op basis van de eerste paar bytes zijn de meeste bestandtypes te herkennen. Je kunt dit uitproberen met de tool `file`, die vertelt wat voor bestandstype Linux denkt dat bij het bestand hoort. Scriptfiles zijn bestanden met tekstdata die te herkennen zijn aan een verwijzing naar de juiste interpreter op de eerste regel: deze moet met `#!` beginnen, gevolgd door het pad naar de gewenste interpreter. Als je script met `#!/bin/bash` begint, zal `file` (en daarmee Linux) het correct als een Bash script identificeren. Deze regel noemen we de "shebang". Als we beide stappen hebben doorlopen, uitvoerbaar maken en de shebang toevoegen, kunnen we het bestand uitvoeren met `./scriptfile` – toch alweer iets korter dan `bash scriptfile`. **Noot:** In plaats van `#!/bin/bash` wordt tegenwoordig liever `#!/usr/bin/env bash` gebruikt (al kom je de oudere vorm nog vaker tegen). Bij de nieuwe variant wordt het programma `env` gebruikt om de juiste interpreter ergens in het `PATH` te vinden. Programma's staan niet bij alle Linux distributies op dezelfde plek, en soms is een programma niet op systeemniveau maar binnen de home directory van een gebruiker geïnstalleerd. Door `env` te gebruiken kan bash gevonden worden zo lang je `bash` op een command line kan aanroepen.

6.1.1 `PATH` en `which`

Hoewel `./scriptname` al wat beter is dan `bash scriptname`, is het nog steeds opvallend anders dan de meeste andere commands. Dit heeft te maken met de `$PATH` variabele. Deze variabele bevat een lijst met mappen, en alleen programma's of scripts in één van deze mappen kunnen zonder pad worden uitgevoerd. Zo niet, dan moet iedere aanroep met `./`, `../`, `/`, of `~/` beginnen. De huidige map zit niet in het `$PATH`, om te voorkomen dat per ongeluk een kwaadaardig script of programma uitgevoerd kan worden. Stel bijvoorbeeld dat je een bestand met de naam `cd` gedownload hebt, en probeert iets vanuit je Downloads-map te kopiëren; het zou niet handig zijn als het gedownloadde bestand zomaar wordt uitgevoerd, want hier kan alles in staan. De waarde van `$PATH` is met `echo $PATH` te lezen, of met `env` (die alle environment variabelen print). Als we ons script in een directory zetten die in het `$PATH` voorkomt, zal het van overal aan te roepen zijn. Ook kunnen we en map aan ons pad toevoegen. Om bijvoorbeeld de map `~/bin` toe te voegen, gebruik je `PATH=~/bin:$PATH`. De `:` is een scheidingsteken dat we gebruiken om de toegevoegde map met het oude `$PATH` te combineren. We zetten de map vooraan, wat betekent dat dit de eerste map is waar de shell gaat zoeken. We kunnen bijvoorbeeld een eigen versie van `mv` maken, die ergens een logbestand bijhoudt; als we deze nu in `~/bin` zetten dan verwijst het commando `mv` naar `~/bin/mv` en niet meer naar `/bin/mv`. Tot slot kunnen we de variabele alleen voor een subprocess aanpassen door `env PATH=~/bin` voor een commando te zetten, bijvoorbeeld `env PATH=~/bin bash` om een nieuwe Bash instantie te openen met enkel `~/bin` in het `$PATH`.

```
1 PATH=~/bin:$PATH
```

```
2 export PATH=~/bin:$PATH
3 env PATH=~/bin command
```

6.1.2 .bashrc

De wijziging aan `$PATH` die we gedaan hebben geldt alleen voor het huidige proces. Een andere shell zal de aanpassing niet overnemen, zelfs niet als we die vanaf de shell met het nieuwe `$PATH` aanroepen. Door `export PATH=~/bin:$PATH` te gebruiken zal de variabele wel doorgegeven worden aan subprocessen. Een onfhankelijk geopende shell weet nog steeds niet van ons nieuwe `$PATH` af, hiervoor moeten we de `export PATH=~/bin:$PATH` in elke nieuwe shell draaien. Hiervoor is het bestand `~/.bashrc`. De code in dit bestand wordt in iedere nieuwe Bash shell uitgevoerd. Door onze aanpassingen daar neer te zetten, hebben die voor elke (nieuwe) shell effect. Naast `export`s van variabelen wordt `.bashrc` ook veel gebruikt voor het `alias` commando. Hiermee kun je eenvoudig nieuwe commando's aanmaken, zo lang deze gebaseerd zijn op bestaande commando's. We zullen straks zien hoe we ook complexere functies kunnen toevoegen, maar in veel simpele gevallen is een `alias` voldoende. Je kunt een alias voor een bestaand commando maken, zoals `sl` voor `ls` als je dit vaak verkeerd typt. Daarnaast kun je een bestaand commando hiermee standaard opties geven, bijvoorbeeld `alias ls='ls -la'` om `ls` altijd met de `-la`-opties aan te roepen. Naast `.bashrc` hebben we `.bash_profile`. Beide vervullen eenzelfde taak, maar `.bash_profile` wordt voor login shells gebruikt (in de praktijk alleen als je in een kernel level virtual terminal), `.bashrc` voor non-login shells. Dit kun je gebruiken om bijvoorbeeld extra informatie te printen bij een login-shell die je niet in iedere nieuwe terminal nodig hebt. Meestal wordt `.bash_profile` ingesteld om ook `.bashrc` te laden, zodat deze alle algemene informatie (voor alle shells) bevat en `.bash_profile` gebruikt kan worden voor specifieke code voor login shells. Hiervoor gebruiken we onderstaand code-voorbeeld. Aan het einde van deze les zou je de code ook moeten kunnen begrijpen.

`.bash_profile`

```
1 if [ -f ~/.bashrc ]; then
2    source ~/.bashrc
3 fi
```

6.1.3 Subshells

De `.bash_profile` en `.bashrc` worden standaard bij het opstarten door Bash gelezen. In dit geval wordt alle code in de (net aangemaakte maar) bestaande shell uitgevoerd. Dit is anders dan wanneer je een Bash script aanroept, hiervoor wordt namelijk een nieuw Bash proces aangemaakt. Dit betekent ook dat als je een script hebt waar variabelen in worden ingesteld, deze alleen in de subshell worden uitgevoerd en weer verdwenen zijn zodra het script beëindigd is. Als het zetten van variabelen de taak van het script is, is dit niet wat je wilt. Gelukkig kun je ook bestaande code uitvoeren in de huidige shell. Dit gebeurt met

`. filename` of `source filename` (`source` is een alias voor `.`). Scripts die bedoeld zijn om te `source`n en niet in een subshell uitgevoerd mogen worden kunnen de shebang `#!/bin/false` krijgen: Bash probeert de code met `false` uit te voeren, een programma dat zichzelf altijd direct met een non-zero return value beëindigd (en dus een fout simuleert). Het script wordt dus niet uitgevoerd, en de gebruiker krijgt een melding dat er iets niet naar behoren is gegaan.

6.2 Control flow en arrays

Nu we weten hoe we een shell script schrijven en kunnen uitvoeren, is het tijd om naar wat code te gaan kijken. We kunnen natuurlijk wat commando's achter elkaar zetten en het gezegend vinden, maar als we echt wat willen programmeren dan moeten we keuzes kunnen maken. We hebben al gezien hoe we `&&` en `||` kunnen gebruiken om de exit status van een proces te gebruiken om te bepalen om wel of niet de volgende instructie uit te voeren (als A dan B) maar complexere keuzes zijn minder eenvoudig (als A dan B anders C). Hiervoor hebben we in Bash een `if-then-else` constructie. Deze heeft de vorm `if commandoA; then commandoB; else commandoC; fi`. De puntkomma is in Bash equivalent aan een nieuwe regel, je kunt dit dus ook op meerdere regels doen zonder de puntkomma's nodig te hebben. Het format in het voorbeeld kom je vaak tegen, maar is niet de enige manier. Het inspringen is niet noodzakelijk, maar wel leesbaar. Het eerste commando, `commandoA` bepaalt welk pad gekozen wordt. Net als bij `&&` en `||` gebeurt dit op basis van de exit-code: `0` is `true`, elk ander getal is `false` (dit is dus precies anders dan in de meeste talen, waaronder Python). `then` en `else` geven aan waar de volgende code-blokken beginnen, en `fi` (`if` achterstevoren) beëindigd het `if`-statement.

```
1 # Download een bestand
2 if wget http://test.me/audio.mp3; then
3   # Speel een audiobestand
4   mpg123 audio.mp3
5 else
6   echo "Failure"
7 fi
```

Vaak willen we testen of een bestand bestaat: bij een CSV bestand willen we bijvoorbeeld een nieuwe rij toevoegen, maar als het nog niet bestaat moeten we het maken met de header row. Hiervoor hebben we het `test` commando beschikbaar. Met `test -e` testen we of iets bestaat, met `test -f` of het ook nog een bestand is (en geen map) en met `test -d` of het een directory is. Daarnaast zijn er nog vele andere opties om bestanden te testen, hiervoor moet je de `man`-page raadplegen. Het `test` commando kan ook gebruikt worden met strings (`test -n $STRING` test of een string niet leeg is, `test $A = $B` test of twee strings gelijk zijn, ...) en getallen (`test $A -eq $B` test of de getallen gelijk zijn, ...) — ook hiervoor moet je in de `man`-page zijn. Tot slot kun je in één aanroep naar `test` meerdere expressies combineren. `test <TEST1> -a <TEST2>` is de *and* en `test <TEST1> -o <TEST2>` is de *or*. Je kunt een uitroepteken gebruiken

voor *not*, en haakjes om testen te combineren. Een alternatief voor `test` is de blokhaakjes-notatie. Onderwater gebeurt hetzelfde, maar de code kan iets beter leesbaar zijn. `[` is een shorthand voor `test`, met de extra voorwaarde dat het laatste argument `]` is. Zowel bij `test` als bij `[..]` geldt dat je op moet letten met de spatiëring. Iedere operator is een commando of argument, dus moet door een spatie gescheiden zijn. `[-d '/tmp']` werkt niet, dit moet `[ -d '/tmp']` zijn. Hetzelfde geldt bijvoorbeeld voor het uitroepteken. De haakjes zijn een command op zich, geen onderdeel van de `if`-syntax. Bij twijfel: zorg dat overal een spatie tussen staat.

```
1  if test -e audio.mp3; then
2    # Speel een audiobestand
3    mpg123 audio.mp3
4  else
5    echo "Failure"
6  fi
7
8  if [ -e audio.mp3 ]; then
9    # Speel een audiobestand
10   mpg123 audio.mp3
11 else
12   echo "Failure"
13 fi
```

Meerdere keuzes zijn ook mogelijk: de `case`-constructie vergelijkt een variabele met meerdere opties. Neem bijvoorbeeld een `.bashrc`-bestand dat op meerdere systemen gebruikt wordt. Sommige code is algemeen, andere is misschien wat meer systeem-specifiek. We kunnen in dit geval de `case` constructie gebruiken om op de `$HOSTNAME` variabele te splitsen (deze bevat de hostname, die per systeem anders zou moeten zijn). De code hieronder bevat specifieke code voor de Raspberry Pi (genaamd *Pi*), de laptop (genaamd *Voyager*) en de pc (genaamd *Defiant*). De constructie begint met `case $VARIABELE in`, gevolgd door een aantal opties. Iedere optie begint met de te matchen string, gevolgd door een sluit-haakje: `Pi)`. De regels hierna worden alleen op de Pi uitgevoerd, tot aan de dubbele puntkomma `;;`. Je kunt `*)` gebruiken voor de `else`, en de hele constructie wordt met `esac` beëindigd (`case` achterstevoren).

```
1  case $HOSTNAME in
2    Pi)
3    # Code voor de Raspberry Pi
4    ;;
5    Voyager)
6    # Code voor de laptop
7    ;;
8    Defiant)
9    # Code voor de PC
10   ;;
11   *)
12   # Code voor als $HOSTNAME iets anders blijkt te zijn
```

```
13    ;;
14  esac
```

6.2.1 Loops

Naast keuzes willen we ook kunnen loopen: code een aantal keer uitvoeren, of herhalen tot een bepaald punt bereikt is. Hiervoor kennen we de `for` en `while` (en `until`) loops. De `for`-loop heeft weer twee varianten. Met de eerste variant wordt over een array gelopen, bijvoorbeeld om elk element te printen. Daarnaast ondersteunt Bash de C-style `for` loop, waarbij een variabele aangemaakt wordt die iedere keer opgehoogd wordt tot een bepaalde waarde wordt bereikt.

```
1  declare -a ARR=('Christopher' 'David' 'Matt' 'Peter' '
       Jodie')
2
3  for word in $ARR; do
4    echo $word
5  done
6
7  for ((x=1; x<=10; x++)); do
8    echo $x
9  done
```

In het vorige voorbeeld zagen we een array (ook wel lijst). Een lijst kan worden aangemaakt met `declare -a ARRAY`. Lege arrays worden aangemaakt met lege haakjes `()`. Een waarde binnen de array kan worden aangepast met `ARRAY[INDEX] = Foo`, waarbij we `INDEX` natuurlijk 0 is voor het eerste element. Om een waarde op te zoeken gebruiken we `${ARRAY[0]}`; de accolades worden gebruikt om aan te geven dat we de eerste waarde van `ARRAY` willen opzoeken, niet een variabele genaamd `ARRAY` met nog wat tekst erachteraan: `echo $ARRAY[0]` geeft (met de Array van hierboven) `Christopher[0]` terug. `$ARRAY` evalueert naar het eerste element in de lijst, en `[0]` wordt als deel van de argumenten van `echo` beschouwd. `$ARRAY` is dus hetzelfde als `${ARRAY[0]}`, en `ARRAY=John` vervangt niet de hele array door John, maar alleen het eerste element. Je kunt in Bash ook arrays maken die niet met een integer, maar met een string geïndexeerd zijn (vergelijkbaar met de dict in Python). Dit heet een associative array, en wordt aangemaakt met `declare -A ARRAY`. Als je `$ARRAY` vraagt voor een associative array krijg je het element met index "`0`" als dit bestaat (anders een lege string). Tot slot kun je arrays uitbreiden met `+=`, zoals hieronder voorgedaan voor associative arrays.

```
1  declare -A ARRAY=([A]=Jim [B]=John [C]=Rachel [D]=Jean
       -Luc)
2  ARRAY=Jim
3  echo ${ARRAY[0]} # Jim
4  ARRAY+=([E]=Jean-Luc)
```

Accolades bij variabelen

De accolades die we nodig hadden om elementen uit de array te halen zijn niet specifiek voor arrays: deze mogen overal gebruikt worden wanneer er onduidelijkheid kan zijn of een teken bij de variabele-naam hoort of niet. Meestal kan een spatie gebruikt worden om het einde van de variabelenaam aan te geven, maar deze wordt dan ook geprint; als dit niet wenselijk is bieden de accolades uitkomst.

```
1 FOO="foo"
2 echo $FOOd # lege output
3 echo $FOO d # foo d
4 echo ${FOO}d # food
```

While

Naast de `for`-loop ondersteunt Bash ook `while`- en `until` loops. Het verschil is dat bij `while` de loop herhaalt zo lang de stopconditie waar is, terwijl `until` doorgaat zo lang de stopconditie (nog) niet waar is. Net als met het `if`-statement is de keuze gebaseerd op de return value van het command (of de command list) voor het `do`-statement. Daarnaast ondersteunen loops `break` (stop de loop) en `continue` (begin direct aan de volgende ronde door de loop), al wordt het aangeraden deze alleen te gebruiken wanneer strikt noodzakelijk omdat dit de leesbaarheid van de code niet helpt. Beide commands hebben een optioneel argument voor het werken met genestte loops: `break 2` stopt de binnenste twee loops, en gaat daarbuiten verder. De return-value van de loop zelf is die van het laatste commando dat is uitgevoerd na `do`, of `0` als er niets is uitgevoerd.

```
1 while <commands>; do
2   <commands>
3 done
4
5 until <commands>; do
6   <commands>
7 done
```

6.3 Functies

Vorige week hebben we kennis gemaakt met de command-list. Om van een command-list weer één object te maken, kunnen we twee vormen gebruiken: gewone haakjes en accolades. Bij accolades moet het sluithaakje op een eigen regel staan (of voorafgegaan worden door een `;`); voor gewone haakjes is dit niet zo. Het verschil is waar de commands worden uitgevoerd: commands tussen accolades gebeuren binnen de huidige shell, terwijl bij haakjes een subshell geopend wordt: een apart proces. Dit houdt in dat wijzigingen aan de environment (bijvoorbeeld variabelen) de subshell niet overleven.

- `( foo; bar && baz )`
- `{ foo; bar && baz; }`

```
cd
echo $PWD # /home/username

( cd /; echo $PWD ) # /
echo $PWD # /home/username

{ cd /; echo $PWD } # /
echo $PWD # /
```

Nu we de control structures kennen rest ons nog één belangrijke constructie om echte code te schrijven: de functie. De syntax hiervoor in Bash is `<naam> ()<samengesteld commando> <io redirections>`. Samengesteld commands die we tot nu toe hebben gezien zijn een command-list tussen accolades of haakjes of een `for`, `select`, `if`, `while` of `until` constructie. Naast de "standaard" syntax kom je ook vaak vormen met het codewoord `function` tegen, al dan niet met de haakjes. Hieronder staan alle mogelijke combinaties. De haakjes zijn altijd leeg, argumenten worden bij het aanroepen van de functie in variabelen `$1`, `$2`, etc. Dit werkt hetzelfde als voor scriptbestanden. `$0` bevat in dat geval de naam van het script, of als die er niet is de naam van de shell. Na de body van de functie kunnen we nog IO redirections toepassen om de uitvoer bijvoorbeeld naar een bestand te schrijven. Deze redirections worden uitgevoerd als de functie wordt aangeroepen (bij het definiëren van de functie gebeurt niets). Je kunt hier ook functie-argumenten in gebruiken om een filename te maken.

```
filetype_count() { find -name "*.$1" | wc -l; }
filetype_count "jpg" # 31167

filetype_count() { find -name "*.$1" | wc -l; } > $2
filetype_count "jpg" "jpgcount"
cat jpgcount # 31167

# Equivalente syntax
function filetype_count() { find -name "*.$1" | wc -l;
    }
function filetype_count { find -name "*.$1" | wc -l; }
filetype_count() { find -name "*.$1" | wc -l; }
```

6.4 Expansie en substitutie

6.4.1 Pathname expansion

Omdat Linux alles als een file ziet, zullen we vaak met bestandsnamen te maken hebben. We hebben de vorige les kennis gemaakt met globs (`*` en `?`), oftewel pathname-expansion. De `*` matcht iedere string, `?` een enkel karakter (niet meer, niet minder). In sommige gevallen wil je een karakter matchen met `?`, maar dit eigenlijk beperken tot een bepaalde set mogelijkheden. Hiervoor kun je de blokhaken gebruiken, met daarbinnen alle opties (of een range). `file[0-9]` zal `file0` matchen, `file1`, etc. tot `file9`. Daarnaast kun je accolades en komma's

gebruiken om met een set specifieke strings filenames te maken. De output van `echo file{1,2,3,4,5}` is *file1 file2 file3 file4 file5*. Ook hier kan je ranges gebruiken: het voorbeeld had ook als `echo file` mogelijkheden. Hiervoor kun je de blokhaken gebruiken, met daarbinnen alle opties (of een range). `file [0-9]` zal `file0` matchen, `file1`, etc. tot `file9`. Daarnaast kun je accolades en komma's gebruiken om met een set specifieke strings filenames te maken. De output van `echo file{1,2,3,4,5}` is *file1 file2 file3 file4 file5*. Ook hier kan je ranges gebruiken: het voorbeeld had ook als `echo file{1..5}` gegeven kunnen worden.

```
ls file.*                   # matcht file met elke
    uitgang
ls file?.zip                # matcht filea.zip, file1.
    zip, fileP.zip, etc
ls file[0-9].zip            # matcht alleen file0.zip
    tot file9.zip
ls file[NZ].zip             # matcht alleen fileN.zip
    en fileZ.zip
ls file.{zip,7z,tbz,rar}    # expand naar een de string
     "file.zip file.7z file.tbz file.rar"
ls file{1..5}               # ranges ook toegestaan:
    ‘{1..5}‘
```

6.4.2 Command Substitution

Soms is het handig de uitvoer van een commando in een variabele te kunnen zetten, of als argument voor een ander commando te gebruiken. In veel gevallen is dit met pipes op te lossen, maar dat is niet altijd het geval. Gelukkig heeft Bash command substitution: een commando verpakt in `$( .. )` of ‘..‘ wordt door Bash uitgevoerd, waarna de uitvoer in de plaats van het commando komt. Beide notities zijn bruikbaar, al moet je met de haakjes-notatie uitkijken dat er geen verwarring ontstaat met haakjes die een andere betekenis hebben. De haakjes-notatie heeft wel als voordeel dat deze zich makkelijker laat nesten: met backticks moeten de binnenste backticks met een \ gequote worden. De spaties bij de haakjes-notatie zijn niet verplicht, en vooral om de constructie `$((COMMAND))` te voorkomen teneinde het command in een subshell uit te voeren: de dubbele haakjes zijn namelijk ergens anders voor gereserveerd. In dit geval meot je dus `$( (COMMAND))` gebruiken. - voer command uit en ga verder alsof de output er stond - zet datum in `$DATE` - print de datum (datum naar echo dat input doorgeeft als output, naar nog een echo) - zelfde met backticks (let op de escaping) - dubbele haken (voor subshell binnen `$(..)`) moet met extra spaties, anders arithmetic expansion ::: end pres

```
$( COMMANDS )
‘COMMANDS‘

DATE=$(date)                    # Zet de datum in $DATE
echo $(echo $(date))            # Print de datum (
```

```
    dubbele echo doet niets, alleen voor voorbeeld
    nesting)
6 echo `echo \`date\``          # Hetzelfde met
    backticks: let op de escaping!
7 FOUT=$((date)) # 0            # Dit mag niet, $(())
    wordt voor arithmetic expansion gebruikt
8 BETER=$( (date) ) # de datum  # Een subshell starten
    binnen command substition met extra spaties
```

6.4.3 Arithmetic Expansion en Evaluation

De reden dat we geen dubbele haakjes kunnen gebruiken bij command substitution is dat deze notatie gebruikt wordt voor arithmetic expansion (oftewel: rekenen). Arithmetic expansion geeft je de mogelijkheid een som op te geven, waarvan het antwoord op de plaats van de som komt te staan. De notatie hiervoor is `$((..))`. Naast expansions hebben we ook arithmetic evaluations. Deze gebruiken we voor expressies waar het antwoord geen getal is, maar een *true* of *false*. Deze kunnen we gebruiken in bijvoorbeeld een `if`-statement of een `while`- of `until`-loop. De notatie is `((..))`. De meeste gebruikelijke rekenkundige operatoren, vergelijkingen en zelfs bitwise OR, AND en shifts zijn ondersteund; de gehele lijst is te vinden in de `man`-page van Bash.

```
1  echo $((6*7)) # 42
2
3  read -p "Geef een getal: " getal
4  echo $(($getal**2))
5
6  if (($getal**2 > 100)); then
7    echo "Het kwadraat van het getal was groter dan 100"
8  else
9    echo "Het kwadraat van het getal was kleiner dan 100
       "
10 fi
```

6.5 Lezen van de command-line

Het commando `read` kan gebruikt worden om te lezen van de command-line (of, met `-u` uit een file). Standaard wordt een hele regel ingelezen, zodat je met een `while`-loop een hele file in kan lezen (allen als je een file leest met `read -u`, of door `STDIN` aan te passen). Het resultaat wordt opgeslagen in variabelen waarvan de naam als argument moet worden meegegeven. `read foo` slaat de hele input op in `$foo`. Als er meerdere variabelen mee worden gegeven wordt het eerste woord in de eerste variabele opgeslagen, het tweede woord in de tweede, etc. De laatste variabele krijgt de rest van de input als er teveel woorden ingevoerd zijn. Als je geen variabelenaam meegeeft dan wordt `$REPLY` gebruikt. Standaardopties voor `read` zijn `-p <PROMPT>` om eerst een prompt te printen, `-n <NUMMER>` om een gegeven aantal karakters te lezen, `-a <ARRAY>` om een array

te gebruiken in plaats van normale variabelen, `-t <TIMEOUT>` voor een maximale wachttijd, en `-s` voor secure input (bash laat niet zien wat/dat er getypt wordt, zoals bij `sudo` het password-prompt).

```
1 read var                                                    #
     lees command line en zet in $var
2 read foo bar baz                                            #
     lees command line en zet woord 1 in $foo, woord 2
     in $bar en rest in $baz
3 read -a ARRAY                                               #
     lees alles naar een array $ARRAY
4 read -n 1 -r                                                #
     lees 1 karakter in raw mode (-> lees een enkele
     toetsaanslag)
5 read -s -t 15 -p "Say friend and enter: " password  #
     lees binnen 15 sec een password (onzichtbaar typen)
      met een prompt
```

6.5.1 Process Substitution

Een laatste notatie die je in scripts tegen kan komen is die van process substitution. Hier wordt een pipe tussen twee processen vervangen door tijdelijk bestand, zodat een subshell voorkomen kan worden (en dus variabelen niet "kwijtraken" zodra de subshell verdwijnt). De syntax hiervoor is `<(COMMAND)`, waarbij de uitvoer van `COMMAND` in een tijdelijk bestand gezet wordt,dat automatisch verwijderd wordt zodra de inhoud niet meer nodig is. Deze notatie is als alternatief voor pipes te gebruiken, en werkt met tijdelijke bestanden in `/dev/fd`. Hier wordt een bestand aangemaakt dat automatisch weer verwijderd wordt. In tegenstelling tot met een pipe wordt hier geen subshell aangemaakt, waardoor eventuele variabelen die aangepast worden bewaard blijven (en niet verdwijnen als de pipe wordt gesloten). De notatie voor process-substitution is zeldzaam, en hoeven jullie niet toe te kunen passen. Wel is het van belang dit te herkenne, omdat het soms in scripts voorkomt die jullie moeten kunnen interpreteren.

```
1 uname | cat     # Linux
2 cat <(uname)    # Linux
3
4 echo <(find)    # /dev/fd/63
5
6 cat < <(find)   # Linux
7 echo < <(find) #
```

6.6 Troubleshooting

6.6.1 Precedentie

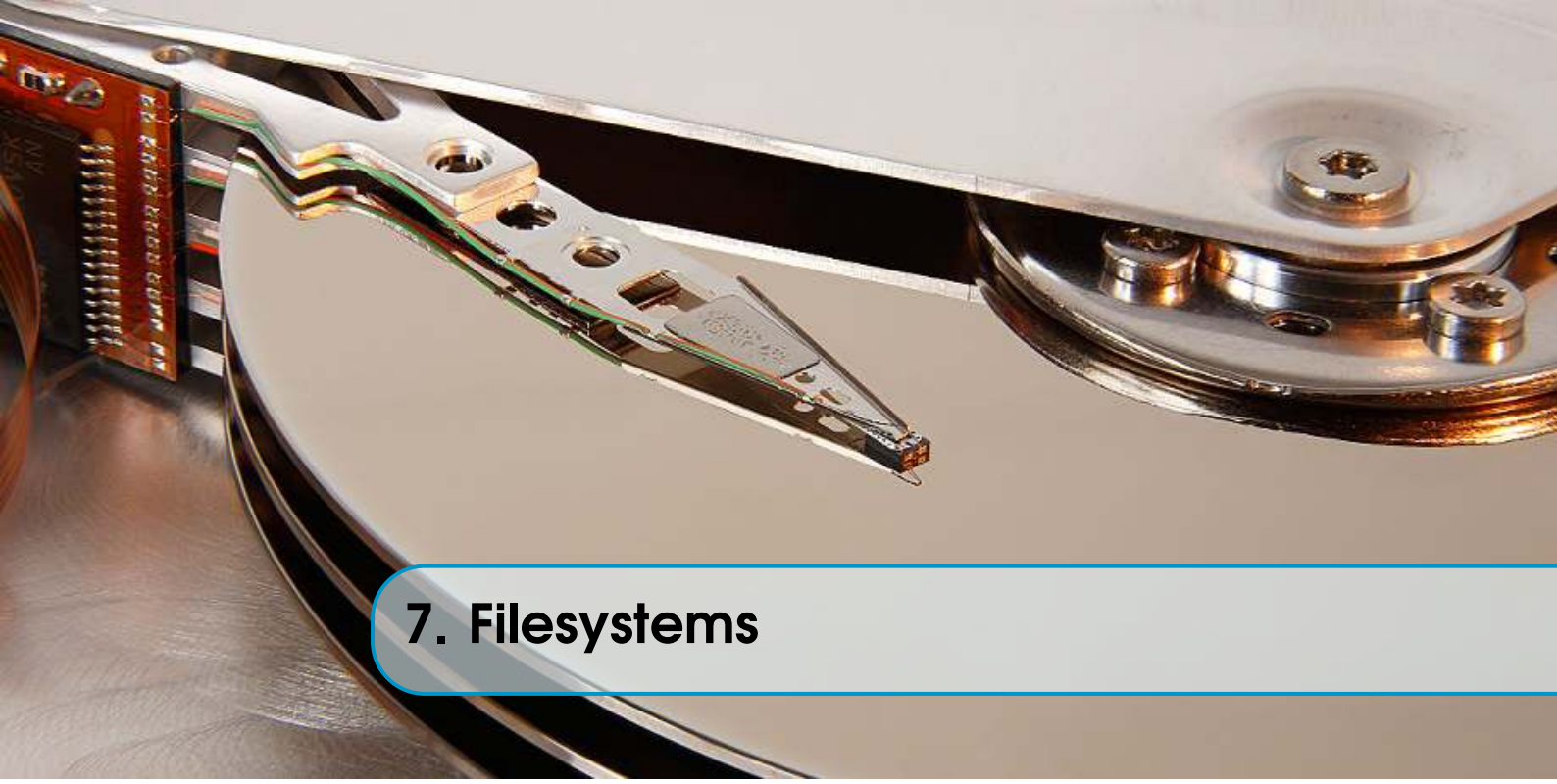
We hebben er inmiddels een hele tour van de Bash-shell opzitten. We hebben commando's gezien, die programma's kunnen zijn (e.g. `time`) of built-ins (e.g.

`cd`). Daarnaast hebben we scripts en functies... Wat nou als we een functie `cd` maken, welke `cd` kiest Bash dan? De keuze die Bash zal maken is bepaald door de precedentie van de verschillende soorten commands. Functies gaan altijd voor, dus bij het invoeren van een command kijkt Bash eerst of er een functie gedefiniëerd is. Zo niet, dan wordt naar built-ins gekeken. Als die er ook niet is, dan wordt het `$PATH` doorzocht. De eerste map waar een script of programma met dezelfde naam als het gevraagde commando te vinden is, wordt gekozen.

6.6.2 Profiling

Soms wil je weten hoe lang je script gedraaid heeft, of hoe lang een commando binnen je script duurt. Linux heeft hiervoor het `time` commando. Deze geeft drie tijden terug: hoe lang het duurde tot het proces daadwerkelijk klaar was, hoeveel tijd het proces aan de beurt is geweest in user mode, en hoe lang de kernel voor het proces bezig is geweest.

```
1 time ./myScript
2 # real    0m4.743s
3 # user    0m0.957s
4 # sys     0m1.948s
```



7. Filesystems

Deze les gaan we kijken naar harde schijven. We weten dat we deze kunnen gebruiken om data voor langere tijd (zelfs na het uitzetten van de computer) te kunnen bewaren, maar hoe gaat dit in z'n werk? Een harde schijf (of tegenwoordig meestal SSD) is een dom apparaat, dat geen weet heeft van bestanden of mappen. Toch ziet het er voor de gebruiker zo uit, dankzij het OS en het filesystem (bestandsysteem). Daarnaast zullen we kijken naar de standaard mappenstructuur in Linux, en alles dat daarbij komt kijken.

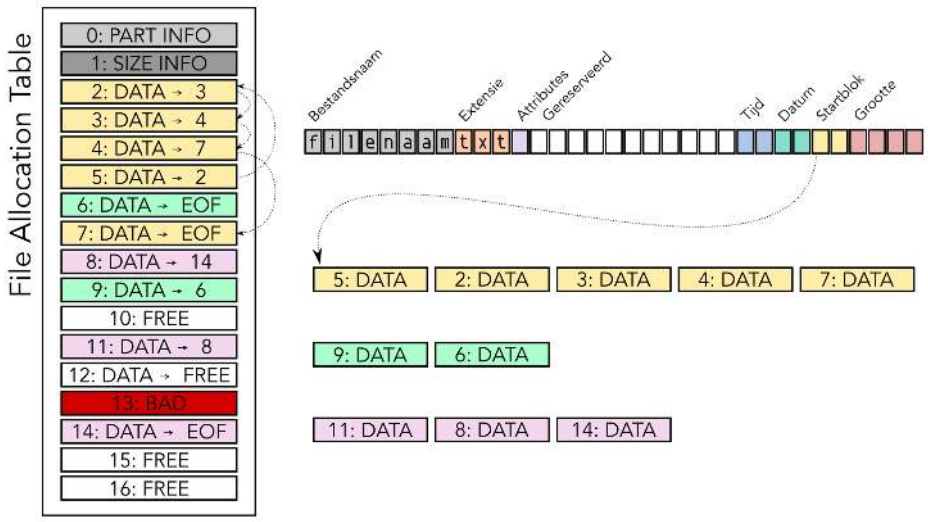
7.1 Filesystems

Vanuit de CPU gezien is de harde schijf niet veel meer dan een IO device: het zit op de IO bus, en er kan data naartoe of vanaf. De disk slaat deze data op met behulp van magnetisme (let op: we hebben het hier niet over SSDs; die zijn in les 4 aan bod gekomen) en is daarmee non-volatile. De manier waarop de disk informatie bewaart is gebaseerd op disks, tracks en sectors. Een harde schijf bestaat doorgaans uit meerdere daadwerkelijke schijven met elk twee kanten. Elke kant heeft een eigen head, een klein stukje electronica dat data kan lezen of schrijven. Een schijfoppervlak is onderverdeeld in tracks: dunne ringen waar de kop naartoe bewogen kan worden. Elke ring bevat tot slot meerdere sectoren, waar de daadwerkelijke data aan gekoppeld is. Het is alleen mogelijk hele blocks (de inhoud van een sector, vaak enkele kilobytes) te lezen of te schrijven. Deze worden in hun geheel naar het geheugen gekopieerd als er iets met de data moet worden gedaan. Om te lezen en te schrijven moet de hardeschijf-controller dus een read-write bit krijgen, plus getallen die de head, track en sector aanduiden. Dit is een heel andere wereld dan de nette geheugenadressen die de computer gewend is, en nog verder verwijderd dan de bekende mappenstructuur die je denkt op de harde schijf te hebben. De laag hiertussen noemen we het filesystem.

Er bestaan verschillende filesystems, elk met eigen voor- en nadelen. Een aantal concepten komt echter bij alle filesystems terug: data wordt verpakt in files (bestanden), en deze files worden ingedeeld in directories (ook wel mappen of folders). Elke file of directory heeft een naam (soms meerdere namen), en door deze paden aan elkaar te plakken hebben we paden waarmee we een bestand kunnen terugvinden. Veel filesystems (met name op Linux en andere unices) ondersteunen daarnaast ook verschillende vormen van links. Conceptueel is een link een snelkoppeling (zoals in Windows), maar waar een snelkoppelling een bestand is met een verwijzing naar een bestand of programmma, is een link op filesystem-niveau geïmplementeerd. Links zorgen ervoor dat één bestand meerdere namen kan hebben.

7.1.1 FAT

FAT (File Allocation Table) is een van de oudste en meest eenvoudige filesystems. Origineel is FAT ontwikkeld voor floppy disks, maar doordat het bestandssysteem door vroege versies van Windows gebruikt werd heeft het de floppy met flink wat jaren overleefd (zij het met de nodige aanpassingen om het systeem te moderniseren). De basis van FAT is een datastructuur die de File Allocation Table heet. De schijf is opgedeeld in *clusters* van een vast formaat, en ieder bestand heeft de vorm van een *chain* of *linked list* van clusters. Dit betekent dat ieder cluster naast data ook een verwijzing bevat naar het volgende cluster: waar het bestand verder gaat. Een bestand wordt afgesloten met een *EOF* (end of file) marker in plaats van een verwijzing naar een volgend cluster. In eerste instantie worden clusters meestal netjes achter elkaar gezet, maar dit hoeft niet. Als een bestaand bestand wordt uitgebreid kan het zijn dat de ruimte erna niet meer vrij is. Door het gebruik van verwijzingen kan op een vrij deel van de schijf verder geschreven worden, en hoeft enkel een verwijzing aangebracht te worden waar het bestand verder gaat. Het gevolg hiervan is wel dat bestanden na verloop van tijd over de hele schijf verspreid staan, wat een vertraging met zich mee brengt omdat de harde schijf veel over moet springen. Dit fenomeen wordt *fragmentatie* genoemd. Naast de gewone clusters met data bevat FAT ook een aantal speciale clusters voor metadata (FAT ID, formaat, ...), directories en clusters die als beschadigd zijn gemarkeerd (en dus niet meer worden gebruikt). Directory tables bevatten per bestand een directory entry. Hierin staat de filenaam, die traditioneel uit 8 karakters + 3 karakters extensie (`filenaam.txt`) bestaat. Naast de bestandsnaam worden attributes bewaard, zoals of een bestand verborgen of alleen-lezen is, en zijn er bytes gereserveerd voor tijd/datum van de laatste aanpassing en de grootte van het bestand. Tot slot bevat een entry een verwijzing naar het eerste cluster waar de data van het bestand te vinden is. In recentere versies worden langere bestandsnamen ook ondersteund, maar ieder bestand heeft ook nog steeds zogenaamde *8+3* bestandsnaam die intern gebruikt wordt: het bestand `bestandmetlangenaam.txt` heeft intern een naam als `bestan~1.txt`. De lange bestandsnaam wordt vervolgens in een of meerdere "fake entries" opgeslagen.



Figuur 7.1 Figuur: Opbouw van het FAT filesystem

7.1.2 NTFS

FAT was aanvankelijk 8-bits en ontwikkeld voor gebruik op floppies, en hoewel Microsoft verschillende verbeterde versies heeft uitgebracht (FAT12, FAT16, FAT32, ExFAT) is het bestandssysteem op een gegeven moment ingeruild voor het nieuwere NTFS (New Technology FileSystem). NTFS biedt beter ondersteuning voor metadata, heeft van nature langere bestandsnamen en ondersteunt bestand groter dan 2^32 bytes (4 GiB), de bovengrens van FAT32. Delen van de werking van NTFS zijn echter niet vrijgegeven, waardoor het bestandssysteem eigenlijk alleen op Windows gebruikt wordt. Mac en Linux kunnen het systeem lezen, maar in de meeste gevallen niet schrijven. FAT-afstammelingen en het voor CD-ROMS ontwikkelde UDF (Universal Disk Format) zijn nog steeds de standaard voor uitwisselbare media die op meerdere besturingssystemen gebruikt moeten worden.

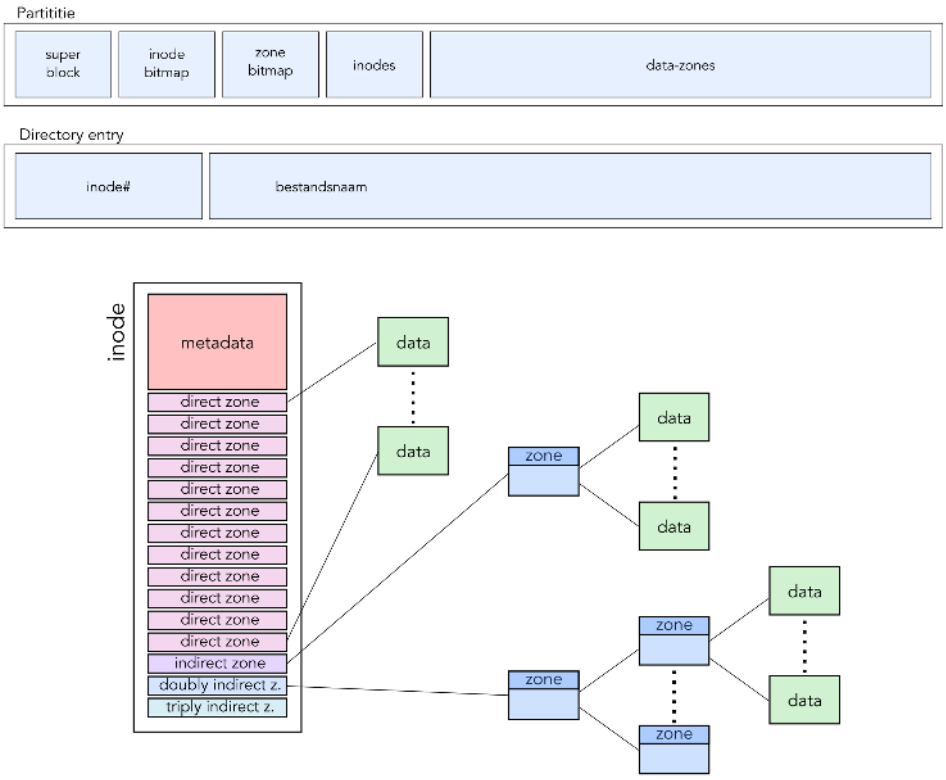
7.1.3 inodes

Unices gebruiken een ander model voor het structureren van filesystems: het hele filesystem is in twee delen opgedeeld: inodes en data. Een inode is een datastructuur die alle metadata van een bestand bevat: rechten, eigenaar en groep (meer hierover later), formaat, een aantal timestamps en het aantal links naar de file. Wat een inode niet bevat is de bestandsnaam / het pad. Deze wordt in een directory bewaard: een tabel van bestandsnamen en bijbehorende de bijbehorende inode. Meerdere bestandsnamen kunnen naar dezelfde inode verwijzen (dit noemen we hardlinks) en de gebruiker kan naar wens links toevoegen en verwijderen. Zodra een inode nergens meer gelinkt wordt, kan deze worden verwijderd (vandaar dat het aantal links wordt bijgehouden). Het maken van een link kan met het programma `ln`, dat hetzelfde werkt als `cp`, maar zonder de data daadwerkelijk te kopiëren: een extra verwijzing is genoeg. Links

verwijderen gaat met `rm`: zo lang er meer links naar een bestand zijn wordt het niet verwijderd, maar alleen de verwijzing. Een inode bevat de verwijzingen naar de daadwerkelijke data-blocks die bij het bestand horen. Een inode in de meeste moderne filesystems bevat 15 pointers (verwijzingen), waarvan de eerste 12 direct naar een block met data verwijzen. De volgende pointer is "singly indirect", en verwijst indien nodig naar een block vol pointers. De volgende pointer is "doubly indirect", en verwijst naar een block vol met singly linked pointers. De laatste pointer is "triply indirect", nog een laag dieper dus. Iedere laag pointers geeft een exponentiële groei aan het aantal koppelbare data blocks. Desalniettemin is het aantal pointers beperkt, en daarmee ook de maximale bestandsgrootte. Wat dit is hangt af van de versie van het bestandssysteem en de grootte van de datablocks, maar voor Ext4 (de laatste versie) met 4KiB blocks is dit 16 TiB — ruim voldoende dus. Een directory entry kan ook naar een subdirectory verwijzen: directories zijn ook gewoon inodes, die met een directory-table verbonden zijn in plaats van bestands-data. De mappen `.` en `..` die we eerder hebben gezien zijn de eerste entries die in iedere directory aanwezig zijn. Tot slot kan een directory entry niet naar een inode verwijzen, maar naar een pad op de harde schijf; dit noemen we een soft-link. Omdat een soft-link niet naar een bestand maar naar een pad verwijst, en het bestand waarnaar gelinkt wordt niet van de link afweet, kan een soft-link verwijzen naar een pad dat niet meer bestaat: het bronbestand is verwijderd of verplaatst. Dit noemen we een dode link, en is een nadeel van soft-links ten opzichte van hard links. In sommige gevallen is het juist wenselijk naar de locatie te linken in plaats van het bestand. Als een bestand vervangen wordt door een nieuwere versie maar op dezelfde plek staat, zal een soft-link ook automatisch naar het nieuwe bestand verwijzen. Het verschilt per situatie welke link dus het beste is. Een Unix partitie is opgebouwd uit een superblock met informatie over de partitie, een inode en zone bitmap met voor iedere inode / zone een bitje dat aangeeft of de inode/zone nog vrij is. Daarna komen de inodes en zones zelf. De aantallen inodes en datazones staan bij het formateren vast, en kunnen niet aangepast worden.

7.2 Partities

Vaak is het wenselijk een harde schijf onder te verdelen in meerdere logische units. Deze onderverdeling noemen we partities, en worden door het OS als aparte schijven behandeld die toevallig op hetzelfde fysieke medium staan. Hierdoor is het dus mogelijk meerdere bestandsystemen met verschillen filesystems naast elkaar te gebruiken. Met name in Linux is het partitioneren van de harde schijf erg gebruikelijk: `boot` staat bijvoorbeeld vaak op een aparte partitie, met een ander filesystem. Omdat de BIOS meestal niet met complexe bestandssystemen om kan gaan is dit een van de opties om het systeem te starten. De boot-partitie bevat de informatie die nodig is om de rest van de schijf te lezen. Daarnaast wordt vaak een speciale partitie voor swap-ruimte gereserveerd. Meerdere partities komen ook vaak voor bij externe harde schijven, waarbij een



Figuur 7.2 Figuur: Opbouw filesystem met inodes

kleine partitie wordt gebruikt om software te leveren die de rest van de schijf (beter) bruikbaar maakt. De partitieindeling staat aan het begin van de schijf opgeslagen. Bij oudere systemen gebeurt dit in het zogenaamde Master Boot Record (MBR), maar tegenwoordig wordt vaak het modernere GUID Partition Table (GPT) gebruikt. MBR heeft een maximum van 4 primary partitions (waarvan je er een kunt op offeren om een paar extra logical partitions toe te kunnen voegen). Naast dit gebrek aan flexibiliteit gebruikt MBR ook nog 32-bits om de startpunten van partities mee aan te kunnen geven, waardoor disks groter dan 2.2 TB niet volledig te gebruiken zijn. Bij dergelijke grote disks verlies je ook flexibiliteit in de verdeling van je ruimte, omdat dit maximum alleen met een optimale indeling te behalen valt. GPT is ontwikkeld om deze tekortkomingen op te lossen. GPT maakt gebruik van zogenaamde GUIDs om partities mee te labelen: een lange random string die uniek is op een harde schijf. De string is zelfs dermate lang dat de kans dat een andere schijf wel eenzelfde GUID heeft, te verwaarlozen is. Dankzij deze identifiers kun je zoveel partities maken als je wilt, zonder logical partitions nodig te hebben. Het OS kan het aantal partities nog wel beperken, zoals 128 als bovengrens voor Windows. Daarnaast heeft GPT een veel groter maximum qua diskgrootte: ergens in de orde van tientallen zettabyte (10^21 bytes; een terabyte is 10^12 bytes).

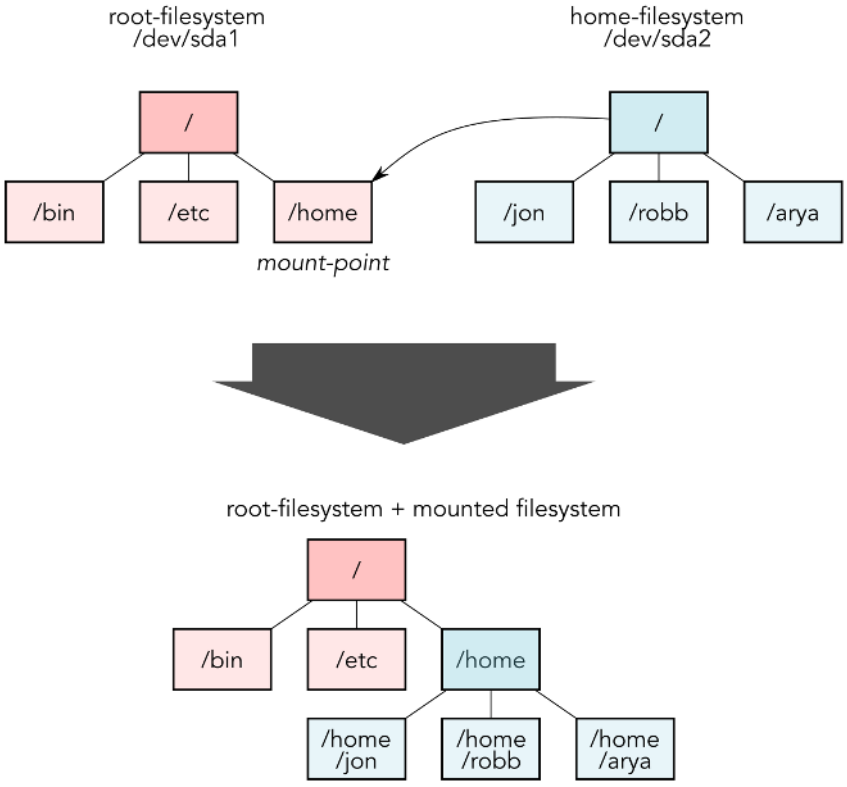
7.2.1 Mounting

Waar Windows voor iedere schijf of partitie een eigen root heeft (`C:\`, `D:\`, ...) wordt bij Linux altijd een root voor het hele systeem gehanteerd. Extra schijven of partities worden *gemount*, wat betekent dat een (doorgaans) lege map wordt gebruikt om de inhoud van de extra partitie onder te zetten. Bestanden en directories die in de root-map van partitie 2 stonden, staan nu in de map die als "mount-point" gebruikt is, en alle paden relatief vanaf de root van partitie 2 zijn nu relatief vanaf het mountpoint op partitie 1 te benaderen. Als de map in kwestie niet leeg was, is de inhoud onbenaderbaar totdat de mount ongedaan gemaakt is. Omdat "unmount" erg veel typen is, is het commando hiervoor `umount`. Hardlinks kunnen alleen binnen een partitie gebruikt worden, en niet over een mount-point heen. Iedere partitie heeft immers dezelfde inode-nummers voor andere bestanden in gebruik, en het is niet mogelijk aan te geven welke partitie een link naar zou moeten verwijzen. Soft-links daarentegen werken wel over partities heen, al worden dit dode links als de partitie waar het verwezen bestand op staat niet gemount is. Als de partitie later alsnog wordt gemount, werken de links weer.

```
1 sudo mount /dev/sda2 /home
2 sudo umount /home # of sudo umount /dev/sda2
```

7.2.2 inodes inzien met `ls -i`

Met de `-i` flag kun je `ls` vragen de inode-nummers te printen. Hieronder is de `ls -ila` van een root-partitie te zien. De inode-nummers staan in de eerste



Figuur 7.3 Figuur: Mounten van een partitie

kolom. De root partitie heeft op de meeste systemen inode 2. Omdat root geen parent heeft, verwijst .. naar de root-map zelf, en is dus ook 2. inodes 0 en 1 worden gebruikt voor partitieinformatie, maar soms ook toegekend voor virtuele mappen. In de listing valt op dat boot ook inode 2 heeft. Dit komt omdat deze gemount is, en dus de root van een eigen partitie.

```
1        2 drwxr-xr-x   1 root root   112 Sep  7 14:27 ./
2        2 drwxr-xr-x   1 root root   112 Sep  7 14:27
             ../
3   509707 drwxr-xr-x   1 root root     4 Oct  3 14:37
         bin/
4   510435 drwxr-xr-x   4 root root  4096 Jan  1  1970
         boot/
5     1025 drwxr-xr-x  20 root root  3720 Oct 18 18:17
           dev/
6      258 drwxr-xr-x   1 root root  1118 Oct 19 12:44
           etc/
7        2 drwxr-xr-x   1 root root    20 Aug  7 10:34
             home/
8  2259030 drwxr-xr-x   1 root root    12 Sep  7 14:27
         media/
9   510827 drwxr-xr-x   1 root root     0 Aug  6 15:43
         mnt/
10       1 dr-xr-xr-x 292 root root     0 Oct  3 14:37
             proc/
11  509700 drwx------   1 root root   198 Oct 19 12:44
         root/
12    3085 drwxr-xr-x  21 root root   680 Oct  5 18:42
           run/
13       1 dr-xr-xr-x  13 root root     0 Oct  3 14:37
             sys/
14  510245 drwxrwxrwt   1 root root 16318 Oct 19 17:19
         tmp/
15  509874 drwxr-xr-x   1 root root     6 Aug  6 15:33
         usr/
16  509710 drwxr-xr-x   1 root root    82 Oct  3 14:37
         var/
```

7.2.3 RAID en LVM

RAID (Redundant Array of Independent Disks) is een opslag-virtualisatie technologie die met name in de serverwereld gebruikt wordt. De koppeling tussen fysieke media en logische partities wordt hierbij verder losgelaten: een partitie kan zich op meerdere schijven bevinden, zowel parallel (alle data is op elke schijf aanwezig) als serieel (data staat over schijven verdeeld). Parallelle data-opslag zorgt voor redundancy: een schijf kan falen, maar er is altijd een tweede of derde schijf met dezelfde data. Seriële data-opslag combineert schijven tot een groter geheel. Beide methoden geven ook een snelheidswinsts: omdat delen van een bestand op andere schijven staan kunnen deze tegelijk worden

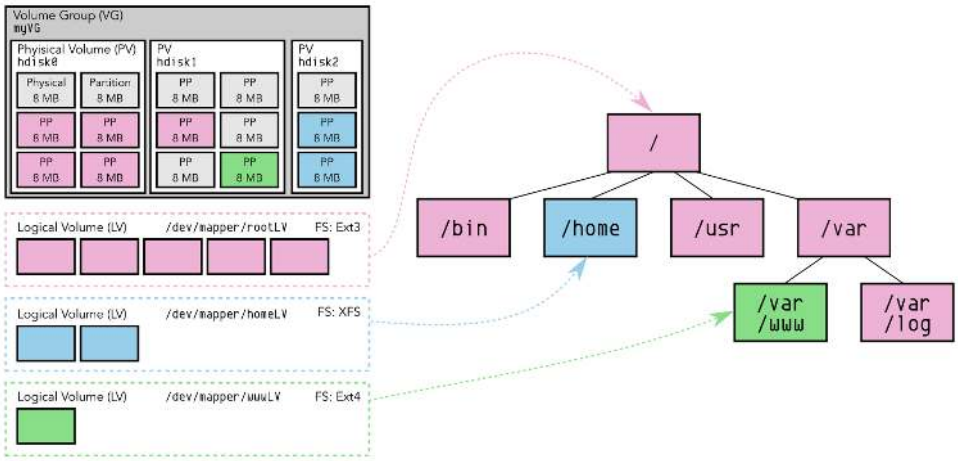
gelezen. Voor RAID bestaan verschillende levels, waarbij parallel en serieel vaak worden gecombineerd: RAID 0 gebruikt *striping* (seriële segmentering van data over schijven), RAID 1 gebruikt *data mirroring* (parallel). Hogere RAID-levels gebruiken verschillende technieken om data-integriteit te bevorderen. Verschillende RAID-levels kunnen worden gecombineerd, zoals RAID 0+1 en RAID 1+0 (beide combineren RAID 0 en 1, maar in een andere volgorde). RAID kan op software-niveau geïmplementeerd worden, maar is meestal hardwarematig in servers aanwezig. RAID en LVM functioneren als een soort van laag onder het bestandssysteem: vanuit het bestandssysteem lijkt het alsof er op een normale harde schijf (of partitie) gewerkt wordt, terwijl hier eigenlijk nog een extra laag tussen zit. Daadwerkelijke harde schijven (physical volumes) worden bij RAID gegroepeerd in pools, waarbinnen virtuele partities (logical volumes) worden aangemaakt — dit is wat de gebruiker als de schijven ziet. Een logical volume kan over meerdere physical volumes verspreid staan, en daarmee groter zijn dan anders mogelijk was geweest.

7.2.4 LVM

LVM (Logical Volume Manager) is iets gebruikelijker in consumenten-systemen, en is geheel als kernel-niveau software geïmplementeerd. Ook bij LVM wordt de koppeling tussen fysieke media en partities zoveel mogelijk losgelaten, maar bij LVM is het doel met name het combineren van schijven voor opslagruimte. Daarnaast heeft LVM het voordeel dat partities makkelijker aangepast kunnen worden, bijvoorbeeld als deze groter of kleiner dan gewenst is. Bij LVM worden verschillende Physical Volumes (PVs) gecombineerd tot Volume Groups (VGs). Binnen deze VG worden Logical Volumes (LVs) aangemaakt, die als virtuele schijven benaderd worden en een filesystem kunnen hebben. Daarnaast maakt LVM het mogelijk snapshots (momentopnames van de toestand van het systeem op een bepaald moment, als een backup) van LVs te maken, en meerdere PVs met een enkel wachtwoord te encrypten. Concreet nemen we als voorbeeld een computer met twee harde schijven: een harde schijf van 1 TiB en een van 4 TiB. Dit zijn in LVM twee PVs. Iedere harde schijf wordt opgedeeld in meerdere kleine PPs, bijvoorbeeld van 8 MiB. Dit betekent dat de eerste schijf 131072 partities bevat, en de tweede 524288 voor een totaal van 655360. Hiervan kunnen bijvoorbeeld 200000 PPs toegewezen zijn aan de root LV, oftewel 1600000 MiB of 1562.5 GiB. De overige 455360 PPs vormen samen de home LV, met totaal 3642880 MiB oftewel 3557.5 GiB.

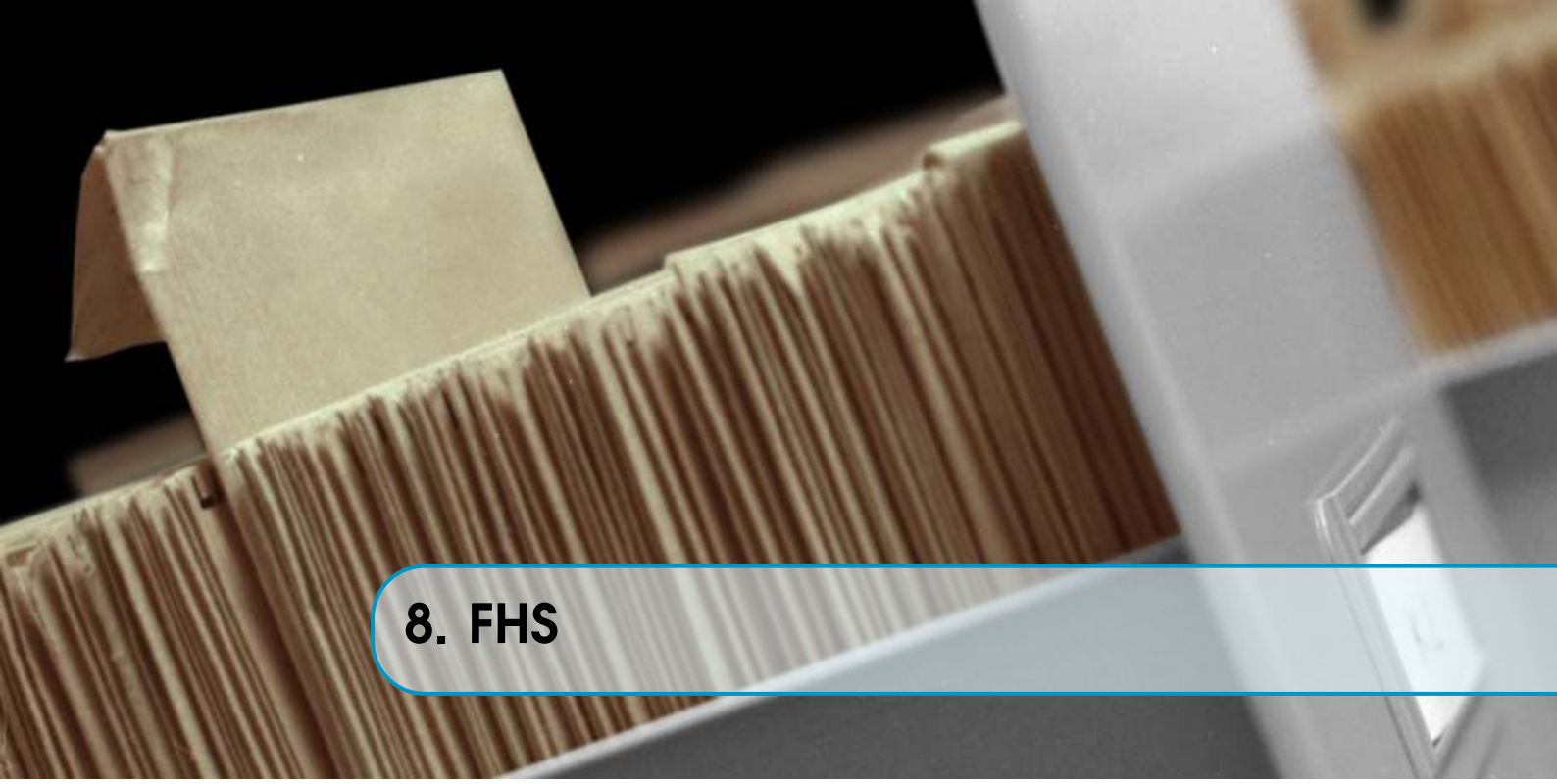
7.2.5 The Next Generation

Om een filesystem power-safe te maken wordt in moderne filesystems gebruik gemaakt van Copy-on-Write. Nieuwe data kan niet zonder meer oude data overschrijven, maar in plaats daarvan wordt de nieuwe situatie elders op de disk opgebouwd. Ongewijzigde data wordt niet gekopieerd, maar verwijst naar de bestaande data; op het moment dat er wel iets verandert, wordt alleen dat deel gekopieerd. Zodra de vernieuwde situatie klaar is, wordt deze “live” gemaakt en kunnen blokken die niet meer nodig zijn van de oude situatie worden verwijderd.



Figuur 7.4 Figuur: Opbouw LVM

Copy-on-Write wordt daarnaast ook gebruikt voor snapshots, waarbij de oude data als snapshot blijft bestaan en naast de nieuwe situatie op te roepen is. Filesystems als ZFS (Sun) en Btrfs (B-Tree FS, Oracle) maken gebruik van deze nieuwe ideeën. ZFS ondersteunt ook een eigen vorm van RAID en logical volume management. Hiermee verhogen zij de veiligheid, performance en capaciteit ten opzichte van de oudere, simpelere filesystems.



8. FHS

We weten nu hoe een filesystem eruit ziet, maar hoe worden de directories ingedeeld? Onder Windows is er een vrij standaard indeling met mappen als *Program Files* en *Users* maar hoe zit dat onder Linux?

8.1 De FHS

De indeling van een Linux bestandssysteem is in grote mate gestandaardiseerd. Waar dit in Windows van natuure het geval is (iedere versie van Windows wordt immers door hetzelfde bedrijf uitgebracht), is het voor Linux wenselijk dat verschillende distro-makers dezelfde structuur hanteren. Hiervoor is een standaard indeling afgesproken, waarbij de belangrijkste mappen de volgende zijn:

- `bin` - binaries (executables)
- `boot` - boot informatie
- `dev` - devices (virtueel)
- `etc` - system-level configuratie
- `home` - user-level
- `lib` - libraies
- `media` - opvolger van `mnt`
- `mnt` - designated mount-point
- `proc` - processen en systeem-informatie (virtueel)
- `root` - home-directory van de systeembeheerder
- `run` - run-time variabele data
- `sbin` - binaries voor alleen root
- `sys` - info over devices, drivers en kernel-features (virtueel)
- `tmp` - tijdelijke map, wordt bij reboot geleegd
- `usr` - secondary hierarchy, bevat mappen voor multi-user
- `var` - variabele data, logs, tijdelijke bestanden

8.1.1 /dev

In Linux (en Unix in het algemeen) wordt alles als bestand gezien; dit geldt ook voor fysieke devices. Dergelijke *device files* of *special files* werken als interface naar een device driver, en zijn te vinden in de map `/dev`. Bij de uitleg over het mounten van filesystems hebben we daar al een voorbeeld van gezien: de partitie die we willen mounten wordt aangeduid met bijvoorbeeld `/dev/sda1`: de eerste partitie op de eerste (SATA) harde schijf. Met `/dev/sda` kan de schijf als geheel aangeduid worden, bijvoorbeeld om de partitie-structuur aan te passen. Ook bij de afbeelding over LVM hebben we entries in `/dev` gezien voor Logical Volumes. Device files hoeven geen fysiek device te representeren: ook virtuele devices hebben een bestand. Deze devices worden *pseudo-devices* of *pseudo-device-files* genoemd, waarbij pseudo een prefix uit het Grieks is dat "niet echt" betekent (denk bijvoorbeeld aan "pseudonym": schuilnaam). Pseudo-devices worden bijvoorbeeld gebruikt door terminal emulators om een terminal op je desktop te krijgen. Bij het openen van een terminal wordt een paar aangemaakt van een pseudo-terminal master en een pseudo-terminal slave. Dit doet de terminal emulator door het pseudo-device `/dev/ptmx` te openen, dat altijd een file-descriptor voor een master teruggeeft en een slave aanmaakt in `/dev/pts`. Pseudo terminal slaves zijn pseudo-devices die doen alsof ze een daadwerkelijke hardware-terminal zijn. Deze zijn elk gekoppeld aan een pseudo-terminal master, ook een virtueel device. Voor iedere terminal op je desktop is dus een dergelijk paar aanwezig. De master en slave zijn met elkaar verbonden, waarbij data die naar de master gestuurd wordt als input voor de slave gebruikt wordt, en andersom. De terminal emulator gebruikt de master om de slave aan te sturen, waarbij de slave het daadwerkelijke werk verricht. Pseudo-terminal slaves (`/dev/pts/*`) hebben elk een entry in de `/dev`-directory. Hetzelfde geldt voor virtual consoles (`/dev/ttyN`) die in de kernel geïmplementeerd zijn en je kan berijken met control, alt en een functietoets. F7 is daarbij op de meeste Linux-systemen de grafische omgeving. `/dev/tty` zonder nummer verwijst naar de terminal die het huidige proces beheert. Met behulp van `cat` en `echo` kunnen we dit bestand lezen en schrijven: `cat /dev/tty` geeft alle invoer terug (tot je cat beëindigt met `C-c` of `C-d`), en `echo foo > /dev/tty` print informatie naar de huidige terminal (net als gewoon `echo` zou doen). Als je de device-file van een terminal weet (hier kom je achter met het `tty` commando) kun je ook naar een andere terminal `echo`en, of de invoer met `cat` afvangen (al leidt dit vooral tot ongedefinieerd gedrag).

Pseudo Devices

De `/dev`-directory bevat een aantal files die vooral handig zijn om als invoer of uitvoer van processen te dienen. Het bestand `/dev/null` kan altijd naar geschreven worden, en de data die erin terecht komt verdwijnt. Dit is bijvoobeeld handig om de `STDOUT` of `STDERR` van een proces naartoe te schrijven als deze niet nodig is. `/dev/zero` en `/dev/random` zijn data generators: `zero` geeft een constante stroom `NULL`-karakters (ASCII waarde 0), en `/dev/random` genereert random data. Beide kunnen bijvoorbeeld gebruikt worden met `dd` (byte level

kopiëren van data) om een file of partitie leeg te maken of met random data te vullen. In dit geval gebruiken we `/dev/urandom` in plaats van `random`, dit is een variant die sneller maar minder secuur is. Voor het vernietigen van data op een harde schijf volstaat `urandom`, maar voor het genereren van wachtwoorden is `random` beter: deze wacht tot voldoende random data beschikbaar is, ook als hiervoor geblocked moet worden.

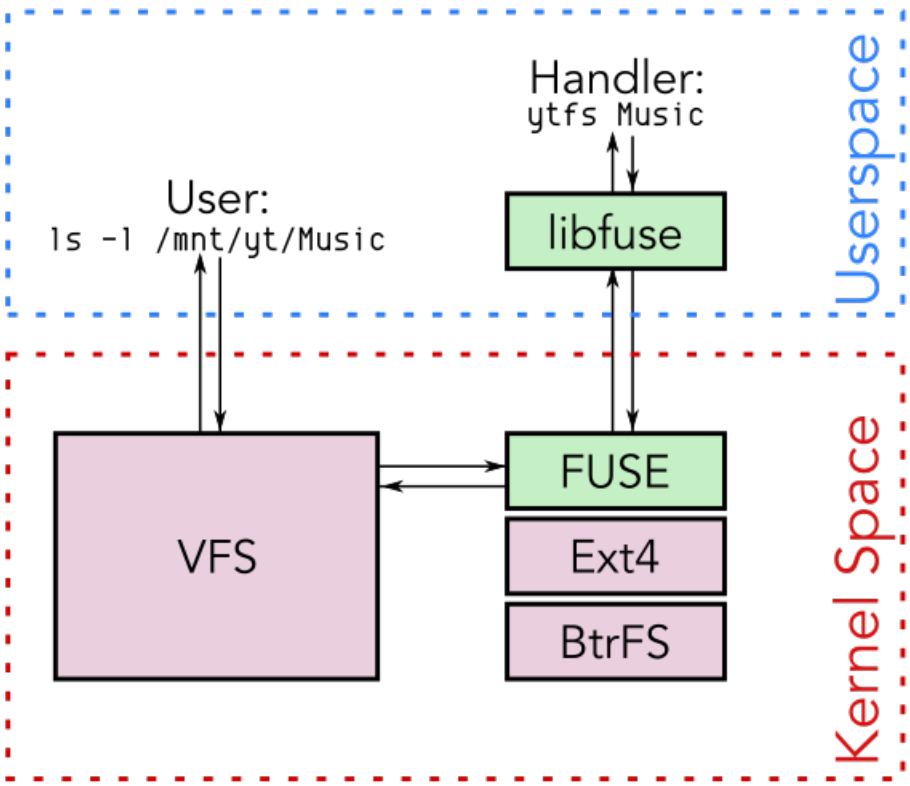
```
1 find . -name "*.svg" 2> /dev/null           # verberg
     error messages
2 dd count=512 if=/dev/zero of=zerofile       # vul een
     file voor 512 blocks met NULL-karakters
3 dd count=512 if=/dev/urandom of=randomfile  # vul een
     file voor 512 blocks met random data
```

8.1.2 `/proc` en `/sys`

Naast devices heeft ook ieder proces zijn eigen file in Linux. In les 2 hebben we deze al voorbij zien komen: per proces is binnen `/proc` een directory aanwezig waarin alle informatie over dit proces te vinden is. Daarnaast bevat `/proc` ook andere virtuele bestanden, zoals `/proc/cpuinfo` en `/proc/meminfo` met informatie over de CPU en geheugen van de computer. Vroeger werd `/proc` voor nog meer gebruikt virtuele bestanden gebruikt, waaronder informatie over de ACPI (Advanced Configuration and Power Interface) zoals de toestand van de batterij van een laptop. Deze informatie is tegenwoordig naar `/sys` verplaatst, die verschillende kernel-level datastructuren beschikbaar maakt. In deze map is diverse systeem- en hardware-informatie op een gestructureerde manier te vinden. Ook kun je in sommige gevallen naar een bestand in `/sys` schrijven om het systeem te verzoeken iets te doen, zoals met het voorbeeld hieronder dat gebruikt kan worden als je een nieuwe harde schijf aan een systeem toevoegt en het systeem deze niet automatisch detecteert. Beide mappen (en `/dev`) zijn niet op de schijf, maar in het RAM aanwezig.

```
1 echo $(( $(cat /sys/class/power_supply/BAT0/charge_now
      ) \          # Bereken batterij-percentage m.b.v.
     virtuele bestanden in /sys
2         / $(cat /sys/class/power_supply/BAT0/
             charge_full) \          # Hier wordt gebruik
              gemaakt van arithmetic expansion
3         * 100 ))%
4
5 # Forceer rescan harde schijven door data in /sys te
     schrijven
6 echo "- - -" > /sys/class/scsi_host/host0/scan
7 fdisk -l

     # Lijst na scan
8 tail -f /var/log/message
                                                # Bekijk
      kernel messages
```



Figuur 8.1 Figuur: Virtual Filesystem Architectuur

8.1.3 Virtual Filesystems en FUSE

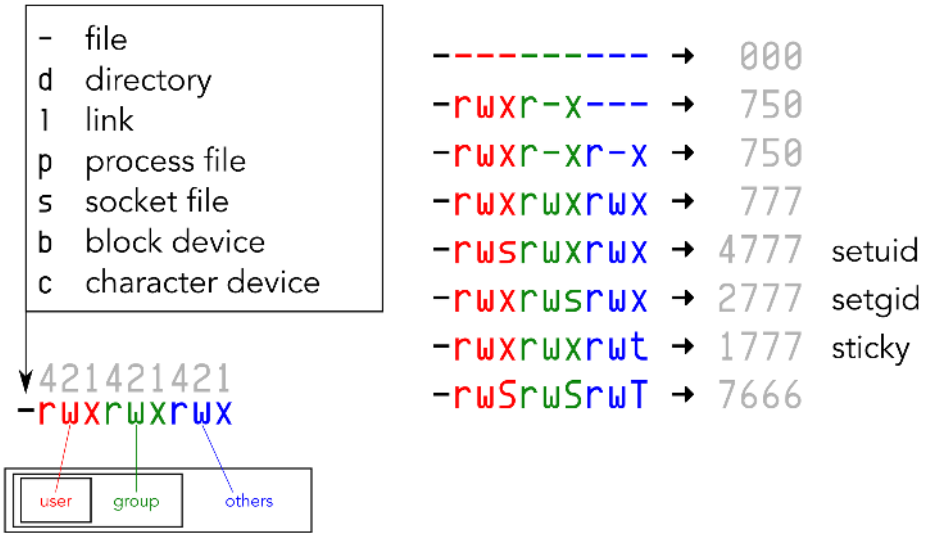
Naast de standaardmappen die we zojuist hebben gezien, wordt in Linux vaker gebruik gemaakt van virtuele filesystems. Een goed voorbeeld hiervan is Filesystems in Userspace (FUSE), waarbij het mogelijk is in user space bestandssystemen aan te koppelen. Dit wordt gebruikt voor "vreemde" bestandssystemen zoals NTFS, maar ook om dingen die geen bestandssysteem zijn wel als bestandssysteem te benaderen. Voorbeelden hiervan zijn WikipediaFS, GMailFS en ytfs (YouTube filesystem).

8.2 Gebruikers en rechten

Ieder bestand op een Linux systeem heeft een eigenaar, een groep en een verzameling rechten. De root-gebruiker is de systeembeheerder en heeft overal toegang toe. Groepen zijn labels die een gebruiker wel of niet kan hebben; als een gebruiker het label `wheel` heeft, dan maakt deze gebruiker deel uit van deze groep. De `wheel` groep is gereserveerd voor gebruikers die root-rechten kunnen aanvragen. Voor veel specifieke rechten zijn groepen, zoals het direct aanspreken van USB-apparaten, het gebruiken van audio en zelfs het spelen van games. De

meeste systemen zijn tegenwoordig Single-user, waarbij de hoofdgebruiker in de meeste (zo niet alle) groepen zit. De groepen (en in mindere mate, gebruikers) stammen nog uit de tijd van de mainframes, maar worden nog veel gebruikt om achtergrond-processen beperkte rechten te geven. Iedere gebruiker heeft een standaard-groep, die meestal dezelfde naam heeft als de gebruiker.

Voor de andere gebruikers geldt echter het rechten-systeem. Een bestand kan leesbaar, schrijfbaar en uitvoerbaar zijn, en deze eigenschappen worden op drie niveaus bijgehouden: wat de eigenaar mag, wat gebruikers in de juiste groep mogen, en wat de rest van de wereld mag. Ieder bestand op een Linux systeem heeft een eigenaar, een groep en een verzameling rechten. Een bestand kan leesbaar, schrijfbaar en uitvoerbaar zijn, en deze eigenschappen worden op drie niveaus bijgehouden: wat de eigenaar mag, wat gebruikers in de juiste groep mogen, en wat de rest van de wereld mag. De root-gebruiker mag altijd alles met ieder bestand. De rechten van een bestand worden bijgehouden met octale getallen, waarbij een bit op een bepaalde positie `1` is als de gebruiker/groep/wereld het recht heeft, en `0` als dat niet zo is. In Figuur Permission Bits is de volgorde van de bits te zien. Ieder octaal getal (groep van 3 bits) staat voor een niveau (user, group, others), en de bits binnen het getal staan voor de rechten. De eerste bit in ieder niveau is `Read`: voor een bestand betekent dit dat de inhoud van het bestand gelezen mag worden, voor een directory dat de bestandslijst mag worden opgevraagd. De tweede bit in ieder niveau is `Write`: voor een bestand betekent dit dat het bestand mag worden gewijzigd of verwijderd, voor een directory dat bestanden en subdirectories mogen worden aangemaakt. De laatste bit in ieder niveau is `eXecute`: voor een bestand betekent dit dat het uitgevoerd mag worden (bijvoorbeeld bij scripts), bij directories dat de map als werkmap gebruikt mag worden en je ernaar kunt `cd`en. Deze bits worden doorgaans als getal (e.g. `755`) of symbolisch (e.g. `-rwxr-xr-x`) weergegeven. Ieder bestand heeft naast deze drie rechten-niveaus ook nog een drietal andere bits: `Setuid`, `Setgid` en `sTicky`. De `setuid`-bit heeft alleen maar zin als de user execute bit gezet is: in dit geval kan het programma of script door iedereen worden uitgevoerd, en wordt dit als de eigenaar van de executable gedaan. Een voorbeeld is het `passwd` commando, dat waarmee een gebruiker haar wachtwoord kan aanpassen. Hiervoor moet in een bestand geschreven worden dat alleen door root te gebruiken is; dankzij de `setuid` bit krijgt de gebruiker bij het uitvoeren van dit programma de rechten van root, zodat het bestand kan worden aangepast. Het is van belang dat hier zorgvuldig mee wordt omgegaan. De `setgid`-bit werkt hetzelfde, maar dan voor de group in plaats van de user. Deze bit heeft enkel zin als het bestand voor de group uitvoerbaar is. In tegenstelling tot `setuid` heeft deze bit ook invloed op mappen: een gebruiker die een bestand in de map met `setgid`-bit aanmaakt, doet dit niet met haar standaard-group, maar met de group van de map. De `sticky`-bit, tot slot, werkt enkel op directories. Als deze gezet is, zijn alle bestanden binnen de map alleen door de eigenaar aan te passen. Standaard is de inhoud van de bestanden veilig, maar kan iemand met de juiste rechten wel bestanden uit een map verwijderen (zelfs zonder rechten voor het bestand zelf). Meestal wordt dit



Figuur 8.2 Figuur: Permission Bits

voorkomen door de rechten van de map ook te beperken, maar voor sommige mappen is dit niet wenselijk. `/tmp` moet bijvoorbeeld voor iedere gebruiker te schrijven zijn. In de symbolische notatie (`-rwxrwxrwx`) wordt `setuid` aangeduid met een `s` op de plaats van de `x` bij de gebruiker. Omdat deze bit geen zin heeft als het bestand niet user-executable is, impliceert de `s` dat ook de `x` gezet is. In het geval dat `setuid` wel gezet is, maar `x` bij de gebruiker niet wordt een `S` getoond als waarschuwing. `setgid` heeft dezelfde notatie, maar nu bij de group `x`, en `sTicky` gebruikt de `t` bij de `x` van others. Numeriek worden `setuid`, `setgid` en `sticky` gecombineerd tot een vierde octale getal, dat vooraan wordt geplaatst. De bitwaardes zijn respectievelijk 4, 2 en 1.

8.2.1 Eigenaar, groep en rechten aanpassen

Om de eigenaar van een bestand aan te passen, kun je `chown` gebruiken: `chown user file`. Met behulp van de `-R` flag kun je dit recursief op een map en alles daaronder toepassen. `chgrp` doet hetzelfde voor groepen, en als je beide wilt veranderen kan dit met `chown user:group file`. De rechten van een bestand aanpassen gaat met `chmod`. Doorgaans wordt hiervoor de numerieke notatie gebruikt, maar je kunt ook een short-hand als `u+x` gebruiken om voor de gebruiker de `eXecutable` bit te toe te kennen. `u`, `g` en `o` staan hierbij respectievelijk voor de user, groep en others. Deze eerste letter is optioneel, als deze niet is gegeven worden de rechten zowel voor user als group als others gezet. De `+` wordt gebruikt om rechten toe te kennen, de `-` om rechten te ontnemen. De laatste letter is doorgaans `r`, `w` of `x`, maar kan ook `s` of `t` zijn om `setuid`, `setgid` en `sticky` aan te passen. Om `setgid` voor een bestand te zetten gebruik je dus bijvoorbeeld `chmod g+s file`, en voor het `sticky` maken van een map `chmod o+t`.

- `chown user:group file` (user en group met een command)

- `chmod -R group directory` (recursief)
- `chmod 755 file` (numerieke waarde)
- `chmod u+x` (zet execute voor user)
- `chmod +x` (zet execute voor user, group en others)
- `chmod o+t` (zet sticky bit)

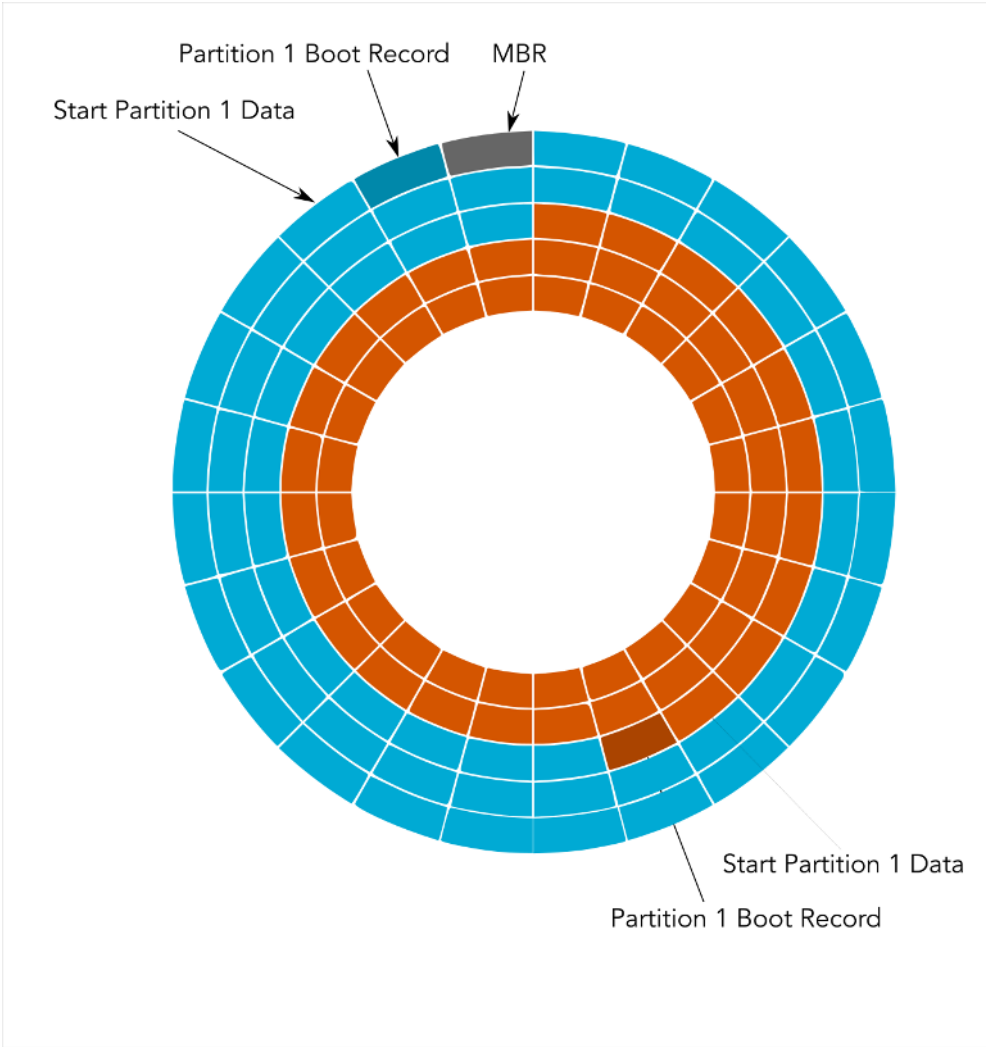
9. Bootstrapping

We hebben inmiddels de meeste taken van een OS gezien. Wat echter nog niet is besproken, is hoe we een OS moeten starten: hoe maken we van onze programmeercode een executable, en hoe zorgen we dat deze start als de computer wordt aangezet? Deze les gaan we daar naar kijken.

9.1 Where to start?

Zodra je de computer aan zet, wordt eerst een stuk (EEP)ROM vanaf het moederbord geladen. Dit is bij oudere computers de BIOS (Basic Input/Output System), maar tegenwoordig vaak UEFI (Unified Extensible Firmware Interface). Beide zullen bij het aanzetten van de PC de aanwezige hardware inventariseren en testen, en vervolgens op zoek gaan naar een OS. Hiertoe kan de gebruiker een boot-order instellen, welke de firmware van boven naar beneden afgaat. Deze zijn op device-niveau vastgesteld: een harde schijf of USB stick heeft een positie in deze rangorde, van partities is nog geen sprake. In het geval van de BIOS wordt bij de geselecteerde schijf (de eerste in de boot-order, of de volgende als de schijf niet bootable was) het Master Boot Record (MBR) gelezen. Hier staat aangegeven welke partities er aanwezig zijn, waar deze beginnen en eindigen, welke partitie de boot-partitie is, en code om de boot-sector (ook wel Volume Boot Record of VBR) van deze partitie te lezen. Bij UEFI wordt gebruik gemaakt van een speciaal gemarkeerde EFI boot-partitie, de ESP (EFI System Partition). Deze bevat een of meerdere EFI bootloaders, doorgaans een enkele per OS. De ESP is bevat voor backwards compatibility een boot-sector (zoals bij MBR), maar kan in de praktijk veel grotere binaries bevatten om complexe taken uit te voeren, zoals het booten van bijzondere filesystems of een netwerk.

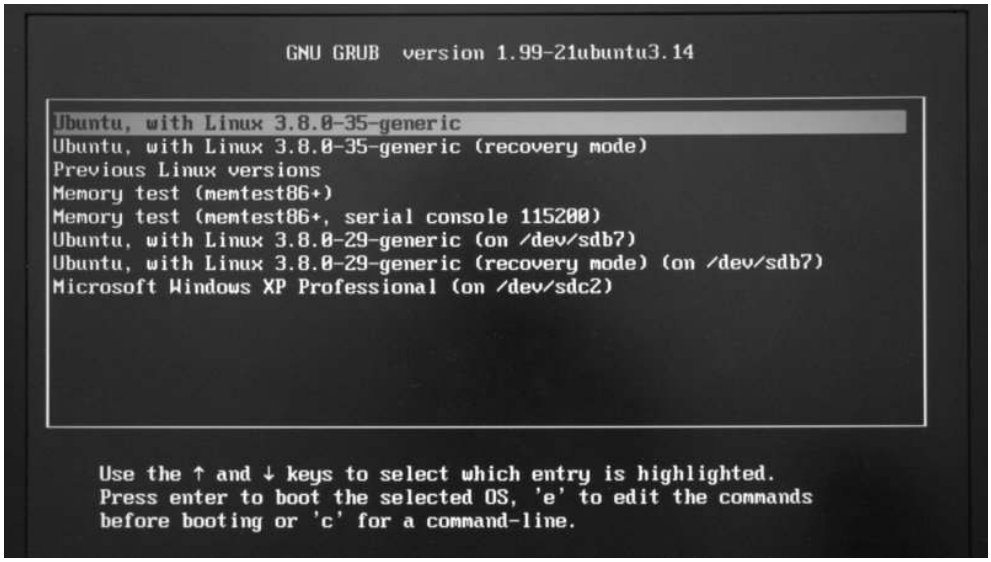
De boot-sector bevat instructies die nodig zijn om een programma op te starten. Doorgaans is dit programma het OS, maar als een eenvoudig systeem een enkele applicatie moet draaien en geen OS functionaliteit nodig heeft



Figuur 9.1 Figuur: Bootsectoren

kan deze ook in de plaats van het OS staan. Zodra de BIOS een boot-device geselecteerd heeft wordt de eerste sector (512 bytes) naar het geheugen geplaatst (bij IBM-compatible PCs naar het geheugenadres `0x7c00`). Deze sector bevat de instructies die nodig zijn om de volgende fase van het boot-proces te laden, en eindigt altijd op de hexadecimale code `0x55aa`: dit *boot sector signature* geeft aan dat de instructies als boot-instructie bedoeld zijn, en veilig kunnen worden uitgevoerd. De boot-sector is doorgaans helemaal aan het begin van de harde schijf te vinden, zoals te zien op de afbeelding. Voor de SSD ziet dit er natuurlijk anders uit, maar ook hier vormt de boot-sector het begin van de schijfruimte.

Omdat de eerste sector (van 512 bytes) ook nog eens informatie over de partitie-indeling van een schijf kan moeten bevatten, en daarmee erg weinig ruimte overblijft om een heel besturingssyteem mee te kunnen laden, wordt het boot-proces vaak in 2 trappen gedaan. De first-stage bootloader bevat alles dat nodig is om de second-stage bootloader te laden, die vervolgens in staat is om



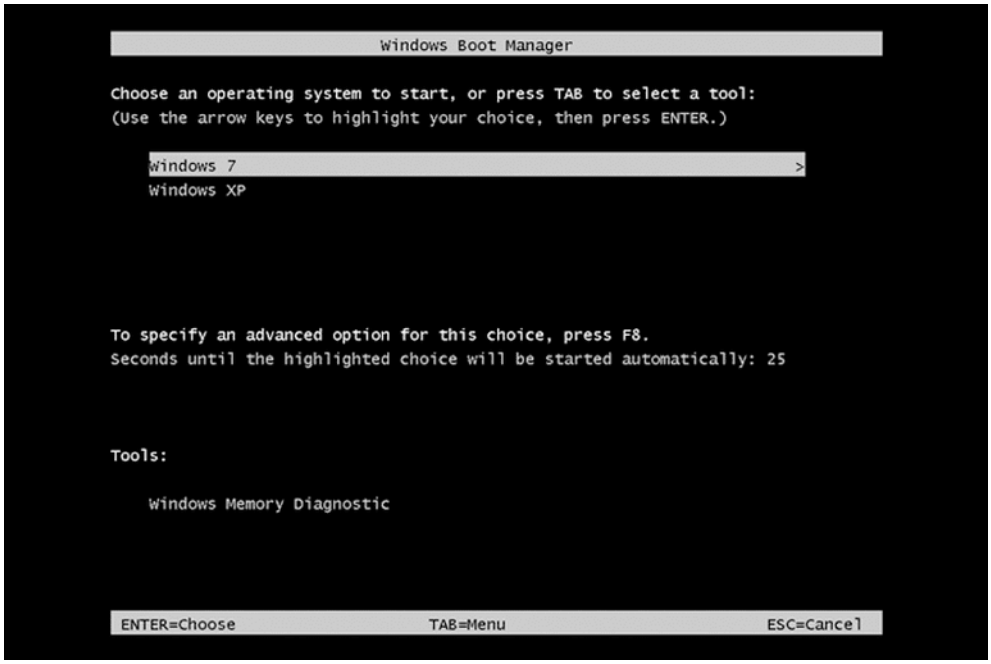
Figuur 9.2 Figuur: Grub, een Linux bootloader

een OS te laden en de uitvoering ervan te starten. Tijdens de installatie van de Gentoo VM ben je al een van deze bootloaders tegengekomen: vermoedelijk was dat Grub, al zijn LiLo (Linux Loader) en Syslinux ook mogelijkheden. Ook op Windows wordt een second-stage bootloader gebruikt, doorgaans BOOTMGR of voor oudere versies NTLDR.

9.2 Het boot-proces in code

Hieronder zien we een voorbeeld van een stuk Assembly dat geboot kan worden en een boodschap print. Het startadres (`0x7c00`) en de signature (`0xaa55`) maken dat deze code bootable is. Daarnaast moet de code natuurlijk in de boot-sector staan, dus de eerste sector van de schijf of partitie. Het schrijven of lezen van deze code is geen tentamenstof, maar het is wel van belang te weten wat het nut van deze code is. De code is met behulp van commentaar toegelicht.

```
1  bits 16
2  org 0x7c00              ; begin op dit adres
3  boot:
4      mov si,welcome      ; si bevat het adres van
           welcome
5      mov ah,0x0e         ; nodig voor interupt 0x10:
           schrijf karakter
6  .loop:
7      lodsb               ; laad eerste karakter in al
8      or al,al            ; als al 0 is:
9      jz halt             ;   spring naar halt
10     int 0x10            ; BIOS interrupt 0x10 (video
           services)
11     jmp .loop           ; loop
12 halt:
```



Figuur 9.3 Figuur: BOOTMGR, de Windows bootloader

```
13      cli                     ; clear interrupt flag
14      hlt                     ; halt execution
15  welcome: db "V1E-OS loading...",0
16  times 510 - ($-$$) db 0 ; vul tot byte 510
17  dw 0xaa55               ; laatste twee bytes:
        signature
```

Als we deze code assembleren levert dat de onderstaande binaire code op. Merk op dat de binary 512 bytes is (en opgevuld met null bytes om daar te komen) en dat de laatste bytes de boot signature `0x55aa` zijn.

```
1  00000000: be10 7cb4 0eac 08c0 7404 cd10 ebf7 faf4
       ..|.....t.......
2  00000010: 5631 452d 4f53 206c 6f61 6469 6e67 2e2e  V1E
       -OS loading..
3  00000020: 2e00 0000 0000 0000 0000 0000 0000 0000
       ................
4  00000030: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
5  00000040: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
6  00000050: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
7  00000060: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
8  00000070: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
9  00000080: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
```

```
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000120: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000130: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000160: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000170: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000180: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000190: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  ..............U.
```

9.2.1 BIOS-calls / Interrupts

Zoals software gebruik maakt van system calls om iets aan de kernel te vragen, moet het OS de hardware aankijken om bepaalde taken uit te voeren. Dit

gebeurt met BIOS calls, ookwel interrupts. Deze worden aangeroepen met de `int` instructie (niet te verwarren met het type `int` uit C++!). Deze krijgt een argument dat aangeeft welke categorie interrupt wordt aangeroepen. De specifieke functie wordt met behulp van het AH (en soms BH) register (zie Les 4) gecommuniceerd, en parameters en return-values staan ook in (functie-specifieke) registers. In onze voorbeelden gebruiken we de volgende interrupts:

- `int 0x10` voor video services
- `int 0x13` voor low-level disk services
- `int 0x15` voor system services

9.2.2 32-bits mode

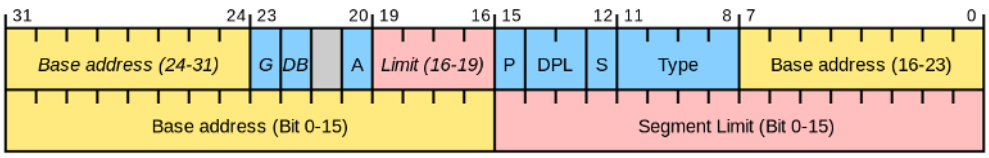
Om complexere code te kunnen draaien is de eerste stap het overschakelen naar 32- of 64-bits mode. De x86-64 processor architectuur start namelijk standaard nog steeds in 16-bits mode. Daarnaast zijn we aanvankelijk beperkt tot 1 MB geheugen, maar met de juiste interrupt kunnen we ons complete geheugen aanspreekbaar maken. Voor de overstap naar 32-bits moet een GDT (Global Descriptor Table) beschikbaar zijn waarin staat hoe het geheugen verdeeld is.

```
1  bits 16
2  org 0x7c00                ; begin op dit adres
3  boot:
4      mov ax, 0x2401        ; function code
5      int 0x15              ; enable >1MB geheugen
6      cli
7      lgdt [gdt_pointer]    ; laad GDT tabel
8      mov eax, cr0
9      or eax,0x1            ; protected mode bit 1
10     mov cr0, eax
11     jmp CODE_SEG:boot2
12 bits 32
13 boot2:
14     ; <32-bits instructies>
15     cli                   ; clear interrupt flag
16     hlt                   ; halt execution
17 times 510 - ($-$$) db 0 ; vul tot byte 510
18 dw 0xaa55                 ; laatste twee bytes:
       signature
```

9.2.3 De General Descriptor Table

De Global Descriptor Table vertelt de CPU waar de verschillende geheugensegmenten te vinden zijn: de offsets waarop deze beginnen en de grootte van de segmenten. De MMU (les 4) gebruikt deze tabel om het geheugen op de juiste manier onder te verdelen.

De GDT tabel wordt in de binary opgebouwd uit letterlijke bytewaardes. In assembly worden deze met data-commando's gegeven: `db` voor data byte, `dw` voor data woord, etc. Doordat de tabel een vast formaat heeft weet de CPU wat waar staat en kan de MMU deze data gebruiken om het geheugen klaar te



Figuur 9.4 Figuur: Formaat van een segment descriptor

maken voor een OS.

```
1  ; offset 0x0
2  .null descriptor:
3      dq 0
4
5  ; offset 0x8
6  .code:                ; cs register verwijst hiernaar
7      dw 0xffff         ; segment limit eerste 0-15 bits
8      dw 0              ; base eerste 0-15 bits
9      db 0              ; base 16-23 bits
10     db 0x9a           ; access byte
11     db 11001111b      ; eerste 4 bits (flags) rest (
           limit 4 laatste bits)(limit is 20 bits breeds)
12     db 0              ; base 24-31 bits
13
14 ; offset 0x10
15 .data:                ; ds, ss, es, fs, en gs verwijzen
       hiernaar
16     dw 0xffff         ; segment limit eerste 0-15 bits
17     dw 0              ; base eerste 0-15 bits
18     db 0              ; base 16-23 bits
19     db 0x92           ; access byte
20     db 11001111b      ; eerste 4 bits (flags) rest (
           limit 4 laatste bits)(limit is 20 bits breeds)
21     db 0              ; base 24-31 bits
```

9.2.4 Meer dan 512 bytes

De volgende beperking die we moeten oplossen is het feit dat we nog steeds in een enkele sector van 512 bytes zitten te werken. Met behulp van interrupts kunnen we de volgende sector(en) van de harde schijf opvragen. In dit voorbeeld wordt tweemaal padding (opvulling) met null-bytes gedaan, om ervoor te zorgen dat een deel van de code in het nieuwe stuk geheugen staat. Omdat het een vrij kort voorbeeld is, hebben we de extra sector niet nodig (alles past in 512 bytes), maar op deze manier wordt het extra geheugen ook daadwerkelijk gebruikt.

```
1  mov ah, 0x2          ; functie: lees sectoren
2  mov al, 1            ; aantal sectoren
3  mov ch, 0            ; welke track
4  mov dh, 0            ; welke head
5  mov cl, 2            ; welke sector
6  mov dl, [disk]       ; welke disk
```

```
7  mov bx, copy_target ; target pointer
8  int 0x13            ; do the thing
```

```
1  ; <code met daarin jump naar label>
2  times 510 - ($-$$) db 0
3  dw 0xaa55
4  label:
5    ; <code>
6  times 1024 - ($-$$) db 0
```

Na assemblage ziet onze code er als volgt uit. Merk op dat de lege stukken (alleen maar nullen) weg zijn gelaten. Een deel van de code is door het padden (opvullen met nullen) in de tweede sector terecht gekomen. De boot-signature `0x55aa` bevindt zich nog steeds aan het einde van de eerste sector.

```
1  00000000: b801 24cd 15b8 0300 cd10 8816 617c b402  ..$
       .........a|..
2  00000010: b001 b500 b600 b102 8a16 617c bb00 7ecd
       ..........a|..~.
3  00000020: 13fa 0f01 165b 7c0f 20c0 6683 c801 0f22
       .....[|. .f...."
4  00000030: c0b8 1000 8ed8 8ec0 8ee0 8ee8 8ed0 ea22
       ..............."
5  00000040: 7e08 0000 0000 0000 0000 00ff ff00 0000
       ~...............
6  00000050: 9acf 00ff ff00 0000 92cf 0018 0043 7c00
       ............C|.
7  00000060: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
8  00000070: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
9  00000080: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
10 00000090: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
11 000000a0: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
12 000000b0: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
13 000000c0: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
14 000000d0: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
15 000000e0: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
16 000000f0: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
17 00000100: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
18 00000110: 0000 0000 0000 0000 0000 0000 0000 0000
       ................
```

```
00000120: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000130: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000160: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000170: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000180: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000190: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  ..............U.
00000200: 4865 6c6c 6f20 6d6f 7265 2074 6861 6e20  Hello more than
00000210: 3531 3220 6279 7465 7320 776f 726c 6421  512 bytes world!
00000220: 2100 be00 7e00 00bb 0080 0b00 ac08 c074  !...~..........t
00000230: 0d0d 000f 0000 6689 0383 c302 ebee faf4  ......f.........
00000240: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000250: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000260: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000270: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000280: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000290: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000002a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000002b0: 0000 0000 0000 0000 0000 0000 0000 0000
```

```
        ................
000002c0: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
000002d0: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
000002e0: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
000002f0: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
00000300: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
00000310: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
00000320: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
00000330: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
00000340: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
00000350: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
00000360: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
00000370: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
00000380: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
00000390: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
000003a0: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
000003b0: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
000003c0: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
000003d0: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
000003e0: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
000003f0: 0000 0000 0000 0000 0000 0000 0000 0000
        ................
```

9.2.5 Klaar voor C++

Inmiddels zijn we zo ver dat we in C++ verder kunnen. Nadat we in 32 bits mode draaien en voldoende geheugen kunnen bereiken kunnen we externe subroutines aanroepen, bijvoorbeeld `kmain` (die we later zullen zien). De assembler controlleert niet of deze functie daadwerkelijk bestaat, we moeten er zelf voor zorgen dat onze assembly code en de C++ goed aan elkaar gelinkt worden.

```
1  mov ah, 0x2                ; functie: lees sectoren
2  mov al, 6                  ; 6 sectoren = 3kb
3  mov ch, 0
4  mov dh, 0
5  mov cl, 2
6  mov dl, [disk]
7  mov bx, copy_target
8  int 0x13
9  ; <GDT code>
10 mov esp,kernel_stack_top   ; stack pointer naar label
       na code
11 extern kmain               ; maak externe functie kmain
       beschikbaar
12 call kmain                 ; roep C++ functie kmain aan
13 cli
14 hlt
15 section .bss
16 align 4
17 kernel_stack_bottom: equ $
18     resb 16384             ; reserveer 16 KB stack
          ruimte in het bss segment
19 kernel_stack_top:
```

9.2.6 Cross-compilen en Linken

Tot zover hebben we enkel met assembly gewerkt, die toch al platform-specifiek is. Nu we met C++ aan de gang gaan, zullen we moeten cross compilen. Dit is nodig omdat we welliswaar op dezelfde processor-architectuur zitten, maar wel in een ander OS werken (ons eigen OS). Linux gebruikt standaard vrij complexe binaire bestanden voor executables (de ELF binaries die we eerder deze les zagen), maar hier kan onze bootloader nog niet mee overweg. Met behulp van een cross-compiler kunnen we een object-file genereren die wel in het juiste binaire formaat is. Om deze te combineren met onze assembly kickstarter moeten we de binaire bestanden aan elkaar linken. Hieronder is een linker-script te zien dat ervoor zorgt dat we een bootable image krijgen.

```
1  ENTRY(boot)
2  OUTPUT_FORMAT("binary")
3  SECTIONS {
4      . = 0x7c00;
5      .text :
6      {
7          *(.boot)
8          *(.text)
9      }
10     .rodata :
11     {
12         *(.rodata)
13     }
```

```
14      .data :
15      {
16          *(.data)
17      }
18      .bss :
19      {
20          *(.bss)
21      }
22  }
```

9.2.7 Het leven zonder glibc

Hoewel we inmiddels in C++ verder kunnen, en ons leven alweer een stuk makkelijker is dan met alleen assembly, is het nog lang niet zo eenvoudig als je van je al compleet werkende computer gewend bent. Alle libraries waar je mee hebt leren werken zijn nog niet op ons nieuwe platform beschikbaar, en zullen in low-level C++ opnieuw geïmplementeerd moeten worden. De C Library (in Linux *glibc*), die samen met de kernel de meeste high-level functionaliteit levert, is nog niet aanspreekbaar: glibc heeft de kernel nodig om functionaliteit te bieden. Zelfs de system calls die we aan het begin van deze cursus gezien hebben, moeten nog worden opgebouwd. De kernel levert een lijst system-codes, die in dit low-level C++ (of in het geval van Linux: C) zijn geprogrammeerd, en hierbovenop wordt de C library gebouwd om een meer vriendelijke interface te leveren. De `kmain` functie die hieronder te zien is werkt welliswaar al met een loop in plaats van de handmatig jumps van assembly, maar het schrijven naar het scherm gebeurt nog altijd door handmatig waardes in geheugenadressen te schrijven. De variabele `vga` is een verwijzing naar het geheugenadres waar de inhoud van het scherm te vinden is, en met een loop schrijven we hier handmatig karakters en kleur-waardes naartoe. De naam `kmain` is arbitrair maar gebruikelijk. Bij normale userspace programma's wordt `main` gebruikt als entrypoint, omdat dat is wat `glibc` verwacht. De kernel heeft deze luxe dus nog niet, en het entrypoint wordt vastgesteld in de bootloader. In ons laatste voorbeeld assembly werd `call kmain` gebruikt om de stap naar C++ te maken, maar iedere andere naam had ook gewerkt. Bij Linux is het entrypoint van de kernel `start_kernel()`. Van hieruit worden alle OS functionaliteiten gestart, waarna de kernel `init` als eerste proces start, dat met behulp van `fork` en `exec` alle andere processen in userspace start.

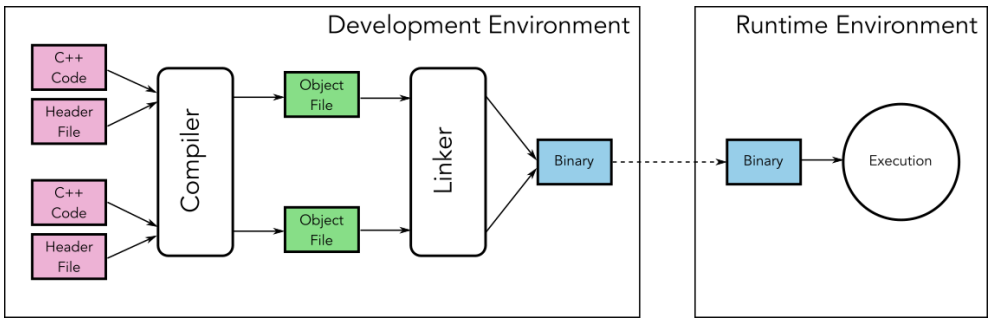
```
1  extern "C" void kmain()
2  { const short color = 0x0F00;
3    const char* hello = "Welcome to V1E-OS!";
4    short* vga = (short*)0xb8000;
5    for (int i = 0; i<16;++i)
6      vga[i+80] = color | hello[i]; }
```

10. Compilation

Aan het einde van deze les gaan we zien wat er nodig is om een eigen first-stage bootloader operationeel te krijgen. Om dit te kunnen volgen missen we echter nog wat kennis over het compilen van code, dus hier zullen we eerst even naar kijken.

10.1 Compilation

Het eerste onderscheid dat we moeten maken, is dat tussen compilation van code en interpretation van scripts. Dit onderscheid bepaalt hoe een programma van de programmeur naar de eindgebruiker gaat. Python, waar je in het eerste semester kennis mee hebt gemaakt, is een voorbeeld van een interpreted language: de code wordt als een tekst-bestand verspreid, en de Python-interpreter is nodig om het script uit te voeren. Wat de interpreter doet, is regel voor regel door het script heen lopen, en deze on-the-fly naar machine-code vertalen. Dit is heel anders dan bijvoorbeeld C++, waar je een compiler gebruikt om de hele source in een keer naar machine-code te vertalen. Dit resulteert in een binair bestand, dat vervolgens in het geheugen van een computer kan worden geladen om deze uit te voeren. We zullen eerst nauwkeurig kijken naar compilation (en de andere stappen die hierbij komen kijken), waarna we ook de eigenschappen van interpretation en JIT (Just-In-Time) compilation kort zullen behandelen. Het is belangrijk om hierbij te onthouden dat hoewel de meeste talen *doorgaans* compiled of interpreted gebruit worden, dit geen noodzakelijke eigenschap van de taal zelf is: Python, bijvoorbeeld, kan zowel interpreted als JIT gebruikt worden. De programmeertaal dicteert de syntax: wat is een geldig programma en hoe wordt deze naar machine-code vertaald. Of dit in een keer van te voren, of on-the-fly gebeurt hangt vaak wel samen met de programmeertaal, maar dit is zeker niet een één-op-één-relatie. Bij compilation wordt de source-code door de compiler vertaalt naar machine-code, met als resultaat een executable.



Figuur 10.1 Figuur: Compilatie van C++ code

Deze executable kan verspreid worden, zonder dat de source-code meegedeeld hoeft te worden. Bij open-source software is dit logischerwijs wel het geval, maar van veel closed-source software is de broncode een (soms zwaar verdedigd) bedrijfsgeheim. Het voordeel van compiled code is dat deze vele malen sneller is. Allereerst is het vertalen al gebeurd, wat een hoop tijd scheelt. Daarnaast kan compiled code verder geoptimaliseerd worden, waar doorgaans de meeste winst behaald wordt. Hier staat tegenover dat het ontwikkelen wel iets trager gaat: het compilen van een fors programma kan minuten tot uren duren, waardoor het lang duurt voor er feedback is of de nieuwe code werkt, waarna je voor een kleine aanpassing opnieuw moet compilen. De compiler zal de meeste syntax- en typefouten gelukkig snel vinden, waardoor het proces vroeg beëindigd kan worden, maar valide code die niet het juiste resultaat levert wordt vaak niet door de compiler gevonden. Een ander nadeel is dat de binary executables die een compiler genereert platform- en CPU specifiek zijn. Een programma dat op Linux is gecompileerd zal niet zonder meer werken op een Windows systeem. Hetzelfde geldt voor processor-architecturen: een programma dat is gecompileerd voor een Intel x86-64 CPU (gebruikelijk voor PCs en laptops) werkt niet op een ARM computer (waaronder de meeste tablets, smartphones en sommige netbooks). Hoewel er "compatibiliteitslagen" bestaan om bijvoorbeeld Windows-binaries op Linux te draaien, werkt dit lang niet altijd 100% en kan dit de performance behoorlijk beïnvloeden. In de praktijk betekent dit dat cross-platform software voor verschillende OS en CPU combinaties gecompileerd moet worden.

10.1.1 Optimisation

Tijdens het compileer-proces kan de compiler je code proberen te optimaliseren. Bij C++ zijn hier bijvoorbeeld meerdere flags voor die je aan de compiler meegeeft: `-O` voert de meest gangbare optimalisatie-technieken uit. Met de flag `-O2` kun je nog wat aggressiever optimaliseren, en `-O3` gaat nog verder. Optimaliseren kost tijd tijdens het compilatie-proces, en kan in uitzonderlijke gevallen problemen in het leven roepen. Daarnaast wordt de uiteindelijke lijst instructies moeilijker te interpreteren. Op https://godbolt.org kun je experimenteren met verschillende optimalisatie-flags. Je C++-code wordt hier live naar assembly vertaald (als tussenstap naar hexadecimale machine-code) zodat je kan zien wat er gebeurt. Bij compiler-options kun je flags zoals `-O`

invullen.

10.1.2 Linking

De meeste programma's bestaan uit verschillende source-bestanden, die elk individueel gecompileerd worden. Het resultaat is machinecode, maar nog niet een uitvoerbaar bestand. Iedere source-file levert een eigen *object*-file op, die de vertaalde machine-instructies van dat bestand bevat. Als hierin verwezen wordt naar subroutines uit een ander bestand, zal de compiler deze niet kunnen vinden. De compiler gaat er voor het gemak vanuit dat de subroutines in een ander bestand staan. Om uit verschillende object-files een uitvoerbaar bestand te maken is een *linker* nodig. Deze kijkt naar de verschillende object-files, en verifieert dat alle verwezen subroutines kloppen. Vervolgens wordt alle benodigde machine-code in een enkel uitvoerbaar bestand gezet, en zorgt de linker ervoor dat de adressen van alle subroutines kloppen. Het voordeel van deze manier van werken is (naast het feit dat de compiler en linker eenvoudiger te houden zijn als ze een enkele taak hebben) dat bij het opnieuw compileren alleen nieuwe object files gemaakt hoeven te worden voor de source-files waarin iets veranderd is. De ongewijzigde bestanden geven exact dezelfde object files, en de linker heeft er geen probleem mee als een deel van de object files al wat ouder is. Een belangrijk onderscheid dat tijdens het linken gemaakt wordt is het dynamic vs static linken van gebruikte subroutines. Bij static linken wordt de machine-code van de subroutine in de executable opgenomen, terwijl bij dynamic linken een verwijzing naar een gedeelde bibliotheek wordt verwezen.

10.1.3 Makefiles

Doorgaans wordt het linken automatisch bij het compilen meegenomen: bij het gebruik van de `-o` flag roept de compiler de linker aan om direct een executable binary te generen. Soms moet er echter handmatig gelinkt worden, en wordt `ld` handmatig aangeroepen. Hierbij kan een *linker script* worden meegegeven om te specificeren hoe de verschillende object-files gecombineerd moeten worden. Bij complexere projecten moeten vaak meerdere bestanden gecompileerd en aan elkaar gelinkt worden. Daarnaast is het soms nodig een preprocessor te draaien, of zijn er andere taken die telkens in de juiste volgorde moeten worden uitgevoerd. Dit handmatige werk wordt meestal met een script opgelost, de zogenaamde *Makefile*. Dit is een bestandsformaat dat speciaal gemaakt is om gestructureerd "recepten" op te bouwen om bijvoorbeeld code te compileren. Ook andere command-line tools die bestanden als input en output hebben kunnen met een makefile gecombineerd worden (deze reader, bijvoorbeeld, worden met behulp van een makefile omgezet van Markdown naar HTML en PDF). Ook taken als het installeren van de software (alle bestanden op de juiste plek in het bestandssysteem van de gebruiker zetten), het genereren van documentatie en het opruimen van de build-directory wordt vaak met een (apart recept binnen dezelfde) makefile gedaan. De recepten worden aangeroepen met het `make` commando, met als argument het gevraagde recept.

Hieronder is een simpel voorbeeld van een makefile te zien, gevolgd door een

meer algemene. In het eerste geval gebruiken we macros voor de compiler en de compile flags, die letterlijk vervangen kunnen worden. Het recept om `main` te maken is afhankelijk van `main.cc` en `functions.cc`, en als deze aanwezig zijn kan de executable gemaakt worden door het recept te volgen. Als de executable aanwezig is, en de afhankelijkheden zijn niet gewijzigd, dan zal `make` ook niet opnieuw compilen. Deze makefile is echter vrij naief: we beheren de call naar de compiler handmatig, en alles wordt in één commando gedaan. Als een enkel bestand aangepast wordt, zal alles opnieuw gecompileerd worden.

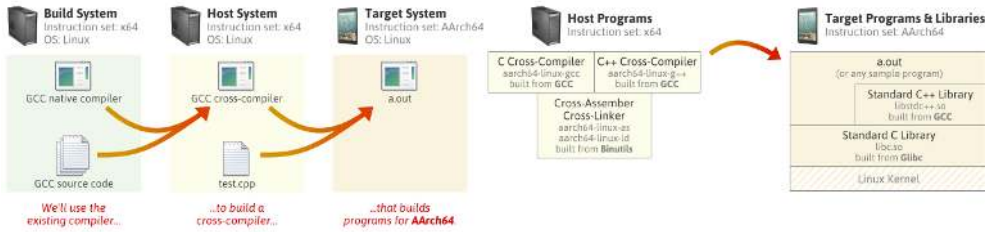
```
1  CXX=g++
2  CXXFLAGS=-I.
3
4  main: main.cc functions.cc
5       $(CXX) -o main main.cc functions.cc $(CXXFLAGS)
```

De tweede makefile is flink veralgemeniseerd. We definiëren een recept dat vertelt hoe een `.o` bestand van een `.cc` bestand gemaakt kan worden. De procent-tekens gelden als wildcards, waarbij De speciale variabele `$@` verwijst naar het doelbestand van het recept, `$<` naar het bronbestand. Het recept is verder afhankelijk van de header-files, zodat een aanpassing in de header ook voldoende is om hercompilatie te triggeren. Een tweede recept, `main`, is afhankelijk van de object files en wordt gebruikt om `g++` te vragen deze te linken. In dit voorbeeld zal een aanpassing aan een enkel bestand ertoe leiden dat alleen dat object opnieuw gecompiled wordt, waarna de binary opnieuw gelinkt wordt.

```
1  CXX=g++
2  CXXFLAGS=-I.
3  DEPS = functions.hh
4  OBJ = main.o functions.o
5
6  %.o: %.cc $(DEPS)
7      $(CXX) -c -o $@ $< $(CXXFLAGS)
8
9  main: $(OBJ)
10     $(CXX) -o $@ $^ $(CXXFLAGS)
```

10.1.4 ELF Binaries

De binaries die we voor onze bootloader zullen maken zijn niet zomaar uitwisselbaar met binaries die we op ons eigen systeem zullen draaien: ze gebruiken een ander binair formaat. Bootable binaries zijn zo simpel mogelijk: ze bevatten informatie over wat waar in de binary te vinden is, en verder enkel data en code. Applicaties op een modern besturingssysteem zijn vaak complexer: het Windows `.exe` formaat en de Linux ELF (Executable and Linkable Format) binaries bevatten flink wat extra structuur. Het ELF formaat wordt daarnaast ook voor gedeelde bibliotheken (op Windows: `.dll`) en core dumps (een dump van het geheugen in het bestand, om mee te debuggen) gebruikt. Ieder ELF



Figuur 10.2 Figuur: Cross Compilation

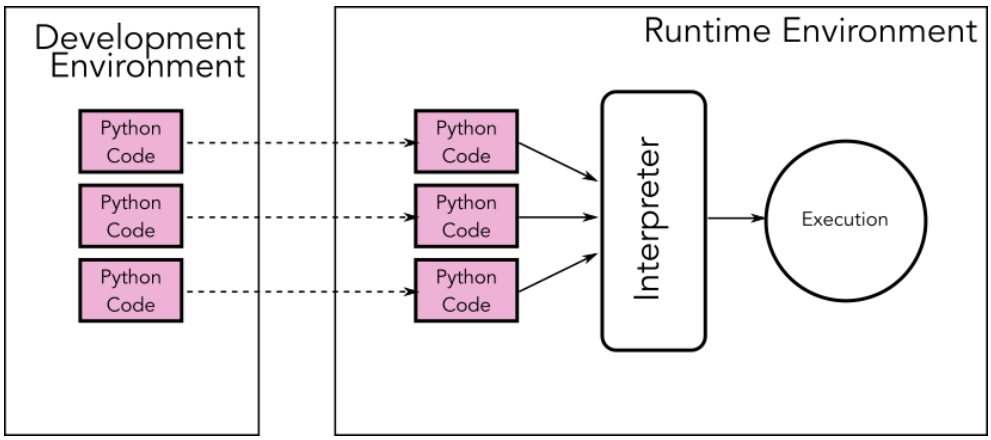
bestand begint met een header, waarin informatie over de binary te vinden is, zoals het platform waarvoor deze gecompileerd is, waar de verschillende segmenten zich bevinden, en uit welke bronbestanden het bestand opgebouwd is. Met het `readelf` commando is deze informatie uit te lezen (`readelf -a filename` geeft alle informatie weer).

10.1.5 Cross compilation

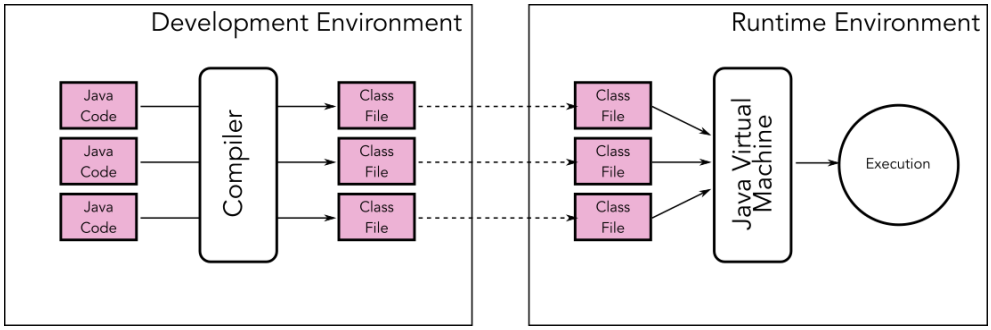
Tot nu toe hebben we bij compilation gezien dat de compiler code omzet in binary executables voor hetzelfde platform als waar de compiler op draait. Bij cross compilation is dit niet het geval: de compiler, draaiend op platform A, maakt executables voor platform B. Dit is bijvoorbeeld nodig bij microcontrollers, waar het doelplatform niet de capaciteit heeft om een compiler te draaien. Ook is cross compilation handig om voor meerder platformen te ontwikkelen, of om gebruik te maken van een server farm. Een platform is hier de combinatie van een processor-architectuur en een OS. In het compilen van OSs is cross-compilation een gegeven, tenzij het OS ver genoeg ontwikkeld is dat het zelf een compiler kan draaien. Bij cross-compilation komen extra uitdagingen kijken, omdat de meeste compilers ervan uit gaan dat binary tools en libraries voor het doel-platform op het build-platform aanwezig zijn. GCC heeft bijvoorbeeld de gecompileerde *binutils* nodig, die dus eerst gecompileerd moeten worden voor dezelfde target en in de PATH beschikbaar moeten zijn. In het vak CPSE1 zullen jullie zelf met cross-compilation moeten werken: van de computer naar de Arduino.

10.1.6 Interpretation

Tegenover compilation vinden we interpretation. Interpretation heeft globaal dezelfde voordelen als compilation nadelen heeft, en vice versa. Interpreted scripts zijn van nature redelijk cross-platform (de code zelf moet dit natuurlijk ook zijn, een Python script dat direct van Windows system-calls gebruikt maakt werkt niet op Linux) en makkelijk / snel te debuggen. Daar staat tegenover dat de scripts zelf een stuk minder snel uitvoeren, en dat het lastig is de source-code verborgen te houden. Voor dat laatste zijn er tools om je code zo onleesbaar mogelijk te maken, maar dit is nog altijd makkelijker ongedaan te maken dan het decompilen van een binary.



Figuur 10.3 Figuur: Interpratie van Python-code



Figuur 10.4 Figuur: JIT compilatie van Java-code

10.1.7 JIT Compilation

Om de voordelen van compilation en interpretation zoveel mogelijk te combineren kan gebruik worden gemaakt van Just-In-Time (JIT) compilation. Hierbij wordt de code door een compiler vertaald naar een *Intermediate Language* ofwel *bytecode*, die op een lichtgewicht virtuele machine draait. De code wordt als het ware "half" gecompiled, en het resultaat is cross-platform zolang de virtuele machine aanwezig is. Bytecode is vergelijkbaar met assembly: een complex programma is vertaald naar simpele instructies die een-voor-een afgehandeld kunnen worden. Het voornaamste verschil is dat de bytecode niet specifiek voor een CPU is, maar voor een virtuele machine. JIT-compiled software is sneller en beter te optimaliseren dan interpreted code, maar nog altijd minder snel dan compiled code. Net als bij compilation moet tijdens het programmeren herhaaldelijk gecompileerd worden. Omdat de bytecode een soort van assembly voor een virtuele machine is, blijft ook de source code standaard verborgen. JIT wordt vooral geassocieerd met talen als Java (op de JVM, de Java Virtual Machine) en C# (op de .NET virtual machine), maar ook talen als Python ondersteunen hun eigen vorm van bytecode. Daarnaast is het voor veel scriptingtalen (zoals Python en Ruby) mogelijk om te compilen naar .NET of de JVM.

10.1.8 Assembly

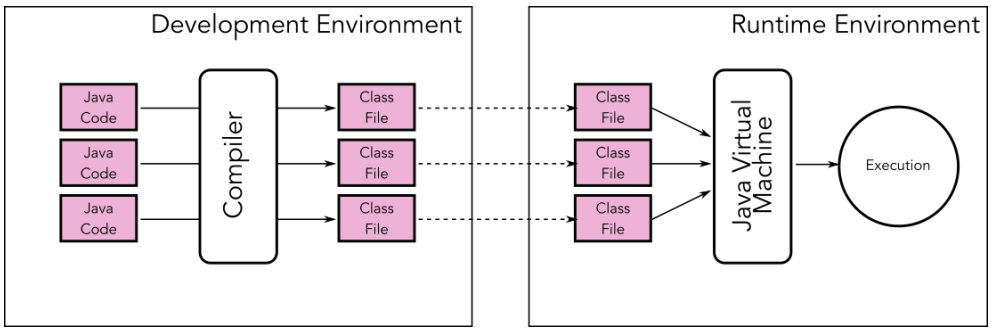
Als laatste bekijken we ook nog even het principe van Assembly. Dit is niet zozeer een programmeertaal die naar machinecode vertaald hoeft te worden, maar een letterlijke 1-op-1 codering van de machinecode. Waar een loop in een programmeertaal naar een hele serie instructies met jumps vertaalt zal worden, worden deze in assembly uitgeschreven. Dit geeft complete controle over de resulterende machinecode, maar is erg ongebruiksvriendelijk. Om deze reden wordt assembly eigenlijk niet meer met de hand geprogrammeerd (compilers leveren nog wel vaak assembly als optionele tussenstap), maar hier zijn uitzonderingen op. Het programmeren van de bootloader is hier één van: omdat de juiste bits op de juiste adressen moeten staan: denk aan de offset `0x7c00` waar de BIOS naar bootable code zoekt, en de signature `0xaa55` die exact 510 bytes daarna moeten komen. Een compiler zal de code interpreteren en een zo efficient mogelijke binary proberen te maken, en is zich niet van deze regels bewust. Natuurlijk is het mogelijk een compiler te schrijven speciaal voor deze eigenaardigheden, maar dit is meer werk dan dat het bespaart: allereerst is de boot code heel beperkt en kunnen we vrij snel verder in een makkelijkere taal zoals C++. Daarnaast is er weinig variatie in de boot-code voor een gegeven platform en kan de code met miscchien een kleine aanpassing vrijwel 1-op-1 worden overgenomen van een bestaande bootloader. In de volgende secties zullen we wat assembly-fragmenten voorbij zien komen. Deze zijn misschien een beetje afschrikwekkend, maar schrik hier niet van. In principe hoef je voor dit vak geen assembly te kunnen, de voorbeelden laten vooral zien hoe de eisen die we hebben gezien in code te vertalen zijn, en dat ook dit fundamentele deel van het systeem uiteindelijk ook maar programmeercode is die op afspraken met de hardware gebaseerd is. Om een bootbaar OS te ontwikkelen moeten de gegeven voorbeelden (op de juiste manier gecombineerd) volstaan om verder in C++ aan de slag te kunnen.

11. Virtualisatie

De vorige les hebben we al een beetje kennis gemaakt met virtualisatie, in de vorm van de JVM en .NET virtual machine. In deze les kijken we verder naar virtuele machines, en hoe deze in het OS geïntergreerd kunnen zijn. De overkoepelende eigenchap van virtual machines is dat deze een systeem emuleren in software. Dit kan een PC zijn die een andere PC in een venster draait (om meerdere OSs tegelijk te draaien), een ander type hardware (emulatie) of een type machine dat fysiek niet bestaat. Daarnaast zullen we kijken naar microcontrollers: embedded systemen zonder OS. Het aansturen van de hardware is grotendeels hetzelfde, maar in dit geval combineert de embedded software deze taak met de daadwerkelijke functionaliteit die het device moet leveren.

11.1 Bytecode Virtual Machines

De JVM en .NET virtual machines behoren tot de laatste categorie: de machine die gevirtualiseerd wordt is niet een daadwerkelijke fysieke machine, maar een construct dat specifiek als virtuele machine ontworpen is. De machine dient als interpreter voor code die al gedeeltelijk gecompileerd is, met als voornaamste doel portability: als de VM op een systeem draait, kan alle software die op de VM draait gebruikt worden. Ook WebAssembly, een low-level assembly achtige subset van JavaScript, kan als een bytecode virtual machine beschouwd worden. Code in C++ (of welke andere taal dan ook, zo lang er een compiler bestaat) kan tot WebAssembly gecompileerd worden, waarna een browser kan dienen als virtual machine / JIT compiler. Sommige talen nemen de intergratie met de bijbehorende VM nog een stapje verder. Pharo, een dialect van SmallTalk, wordt zowel ontwikkeld als uitgevoerd binnen de virtuele machine. De VM ondersteunt een grafische interface, waarbinnen code (zowel van de gebruiker als van het systeem zelf) geinspecteerd, aangepast en uitgevoerd kan worden.



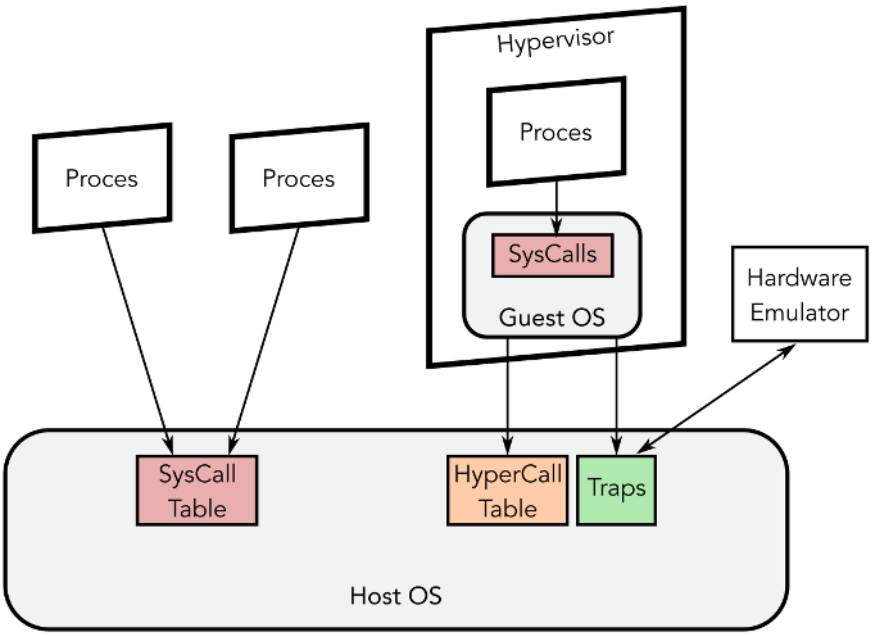
Figuur 11.1 Figuur: JIT compilatie van Java-code

11.2 Hardware Virtualisation

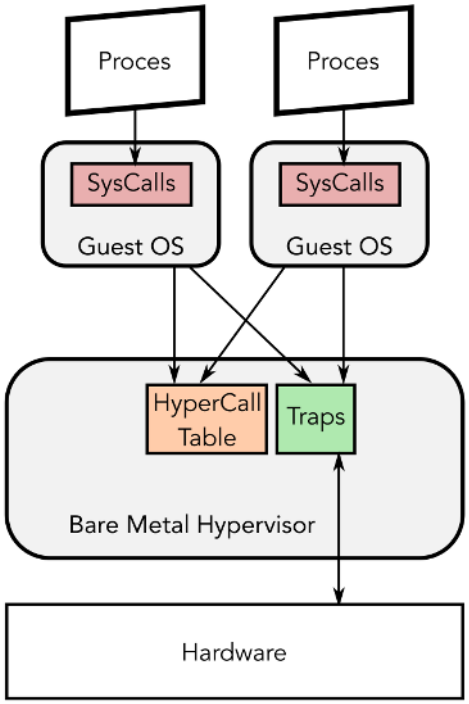
De meest klassieke vorm van virtual machines wordt doorgaans als *platform virtualisation* of *hardware virtualisation* aangeduid. De computer draait een besturingssysteem, met daarbinnen een applicatie, de *hypervisor*, die een fysieke computer nabootst. Vaak is dit een beperkte kopie van de *host computer*, de computer waarop de software draait. De *guest computer* kan een kopie van hetzelfde besturingssysteem draaien, bijvoorbeeld om een afgeschermde omgeving te leveren, of een ander OS draaien, bijvoorbeeld Linux op Windows of vice-versa. Als de architectuur van de *guest computer* afwijkt van die van de *host computer* wordt dit doorgaans *emuleren* genoemd. Een *emulator* kan bijvoorbeeld een spelcomputer of mobiel platform nabootsen, zodat software hiervoor op de computer ontwikkeld en direct getest kan worden. Emulatoren worden ook ingezet om software voor verouderde hardware te kunnen blijven gebruiken, zoals bijvoorbeeld een Gameboy Advance emulator op je computer of smartphone.

11.2.1 Bare-Metal Hypervisors

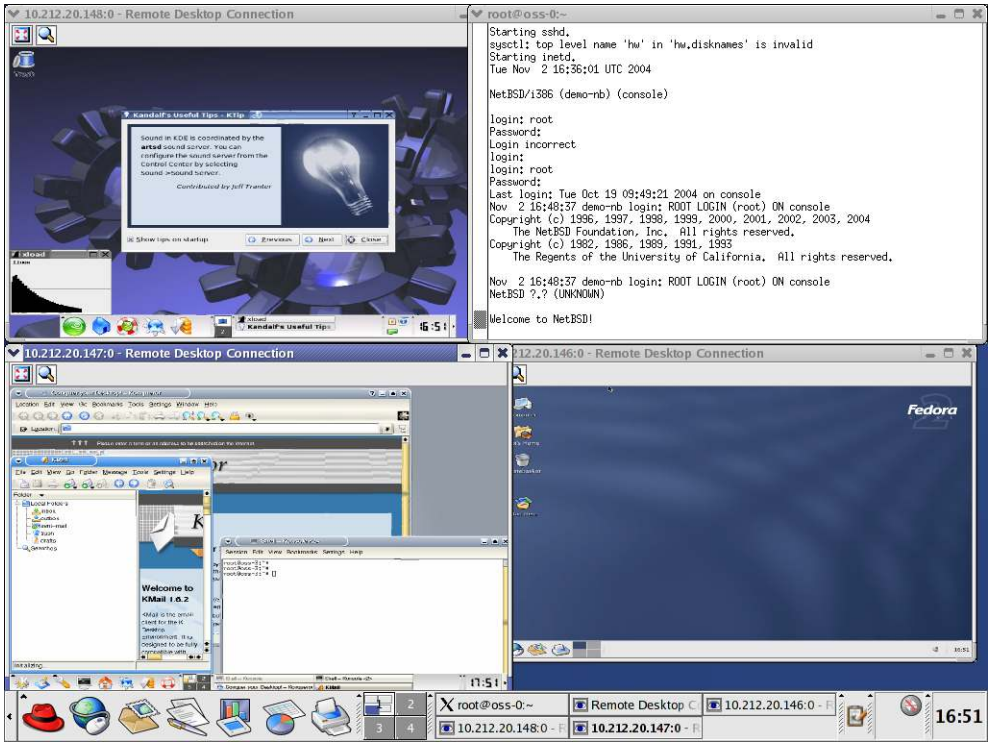
Hoewel de meeste consumenten-hypervisors als applicatie binnen een OS draaien (zogenaamde *hosted hypervisors*, waarbij het OS de *host* levert) is dit niet noodzakelijk. Bij bare-metal hypervisors draait de hypervisor direct op de hardware — in plaats van een OS. Binnen de hypervisor kunnen dan verschillende besturingssytemen draaien. Bij een hosted hypervisor is vaak sprake van een flinke performance-hit voor de guest OSs: de software binnen de guest is erg ver van de hardware verwijderd, waardoor er veel overhead is en vertaling plaatsvindt. Bij bare-metal hypervisors wordt is dit niet (of nauwelijks) het geval. Alle OSs die binnen een bare-metal hypervisor draaien zijn gelijkwaardig, en draaien zo dicht mogelijk op de hardware als mogelijk (met alleen een minimale hypervisor ertussen). Een open-source voorbeeld van een bare-metal hypervisor is Xen: dit kan op de plaats van het OS geïnstalleerd worden, waarbinnen je vervolgens Linux en Windows naast elkaar kan draaien. De afname in performance is voor elk OS gelijk, en veel kleiner dan die zou zijn binnen een traditionele VM. Zo kun je bijvoorbeeld Linux voor werk of studie gebruiken, en daarnaast een Windows installatie om op te gamen.



Figuur 11.2 Figuur: Virtualisatie



Figuur 11.3 Figuur: Bare-Metal Virtualisatie



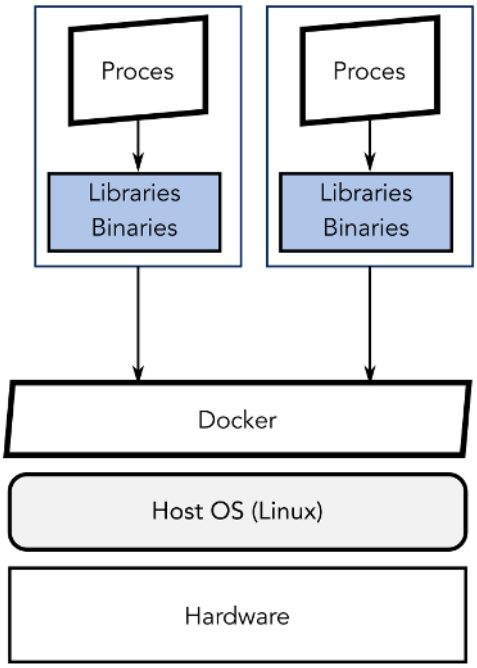
Figuur 11.4 Figuur: Xen, een bare-metal hypervisor

11.2.2 Bare-Metal Hypervisors

11.3 Virtualisatie op OS-level

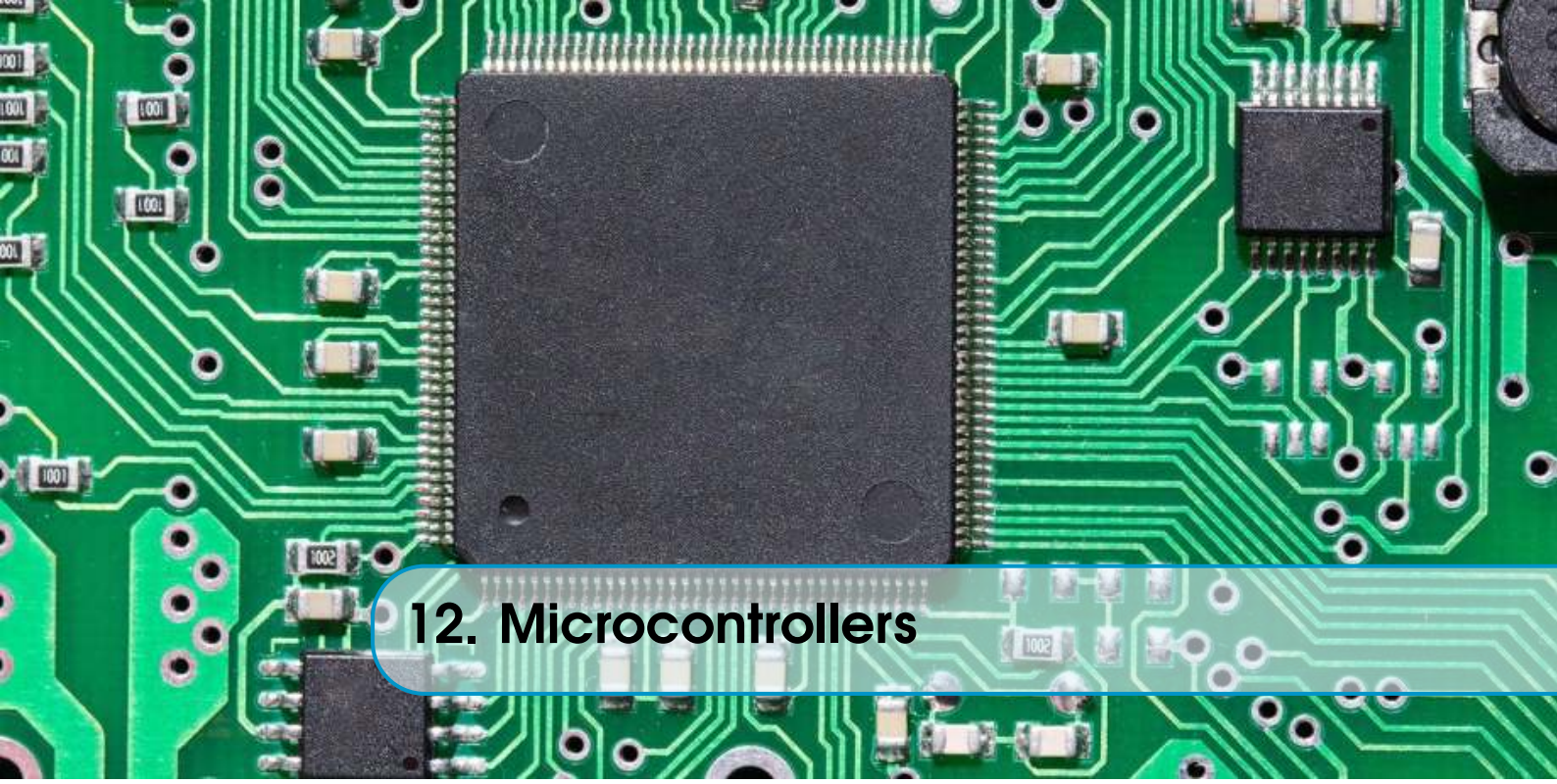
Bij hypervisor-virtualisatie wordt een compleet hardware-systeem gevirtualiseerd. Virtualisatie kan daarnaast ook op OS-level plaatsvinden. Het systeem draait dan een enkele kernel, maar hierop draaien meerdere geisoleerde userspaces: *containers*. Binnen Linux is Docker het meest gebruikte platform voor het draaien van containers. Voor een proces binnen een container lijkt het alsof deze zich op een normale computer bevindt. Wat het proces van het systeem kan zien is echter beperkt tot de inhoud van de container. Meestal wordt een container voor een enkel proces (server, compiler, etc.) gebruikt, dat binnen de container compleet geïsoleerd draait. De verschillende containers werken allemaal op dezelfde kernel, maar kunnen op userspace-niveau flink van elkaar afwijken. Dit vergroot de veiligheid aanzienlijk: verschillende processen zijn zich niet eens van elkaars bestaan bewust, en kunnen dus ook geen kwaadaardige acties op elkaar proberen uit te voeren. Het is daarbij prima mogelijk op een Ubuntu syteem een container te draaien die gebaseerd is op Fedora, of welke andere distro dan ook, zodat voor iedere applicatie de beste omgeving kan worden gekozen. Doorgaans wordt hiervoor een zo licht mogelijke disto gebruikt, zoals Alpine Linux. Daarnaast bestaan er distributies die speciaal ontwikkeld zijn om containers te hosten, waarbij de host zo licht mogelijk wordt gehouden en alle software binnen containers draait. Voorbeelden hiervan zijn CoreOS en RancherOS.

Het eerder genoemde Docker is een goed stuk gereedschap in de toolkit



Figuur 11.5
Figuur: Containers

van ieder (technisch) informaticus. Voor programmeurs kun je met Docker een out-of-the-box-werkende ontwikkel-omgeving leveren: dependencies kunnen in een container aan worden geboden, en geven geen conflicten met reeds geïnstalleerde bibliotheken of programma's. De stappen die nodig zijn om de ontwikkelomgeving op te zetten worden gespecificeerd in een `Dockerfile`, die gebruikt kan worden om een image te realiseren. Dit image kan een daemon-proces bevatten (een proces dat op de achtergrond draait) of een enkele taak uitvoeren en zichzelf afsluiten: denk hierbij bijvoorbeeld aan het (cross)-compilen van een project. Docker files en images kunnen worden geüpload naar een Docker repository, waarvandaan anderen je image kunnen downloaden. Door een Dockerfile te leveren en een openbaar repository te leveren kun je drempel om bij te dragen aan of gebruik te maken van een project behoorlijk verlagen. Andersom kan je werk besparen niet zelf een ontwikkelomgeving op te zetten (mogelijk op een niet-ondersteunde distro of OS) maar van een Docker image gebruik te maken. Docker-containers met daarin een daemon worden veelal gebruikt in de server-wereld: op het moment dat de docker-image werkt op het test-systeem, dan zal deze zonder moeite op een server te deployen moeten zijn.



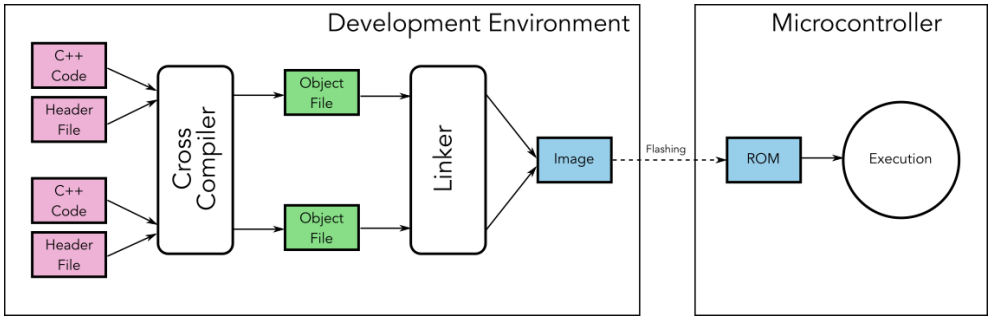
12. Microcontrollers

Hoewel de focus tot nu toe gelegen heeft op OSs en applicaties die op een OS draaien (hosted applications), ligt de focus van de TI-er vaak op een ander domein: de microcontroller. Hier wordt net als bij (het programmeren van) een OS bare-metal gewerkt. De software heeft directe toegang tot de hardware, en er zijn niet zomaar abstracties aanwezig om het leven van de ontwikkelaar te vereenvoudigen. In deze sectie zullen we zien hoe het programmeren van een microcontroller verschilt ten opzichte van OS development en het schrijven van hosted applicaties.

12.1 Development Pipeline voor Microcontrollers

Het eerste verschil is de manier waarop de development pipeline in elkaar steekt. Omdat een bare-metal microcontroller geen OS heeft, draait hier geen tekst-editor of compiler op waardoor deze niet direct geprogrammeerd kan worden. In plaats hiervan moeten we gebruik maken van een development-systeem om de code te schrijven, en deze met een cross-compiler omzetten naar een binary voor de micro-controller. Dit binary-bestand kan dan in de microcontroller worden ingeladen door deze naar het EEPROM of Flash-geheugen te schrijven.

Een andere manier waarop een bare-metal microcontroller van een computer met OS verschilt is de manier waarop het geheugen wordt gebruikt. De computers waar we tot zover naar hebben gekeken maken gebruik van een combinatie van RAM geheugen en persistent storage (harde schijf of SSD). Alle code en bijbehorende data, zowel van het OS als van de applicaties, is ergens op de persistent storage te vinden. Bij het booten wordt de OS code hiervandaan naar het RAM gekopieerd, en bij het starten van een programma komt ook deze code ergens in het RAM te staan. Alle memory-segmenten die we hebben gezien binnen een proces (of het OS) zijn in de binary file aanwezig en worden 1-op-1 in het geheugen overgenomen. Bij een bare-metal microcontroller werkt

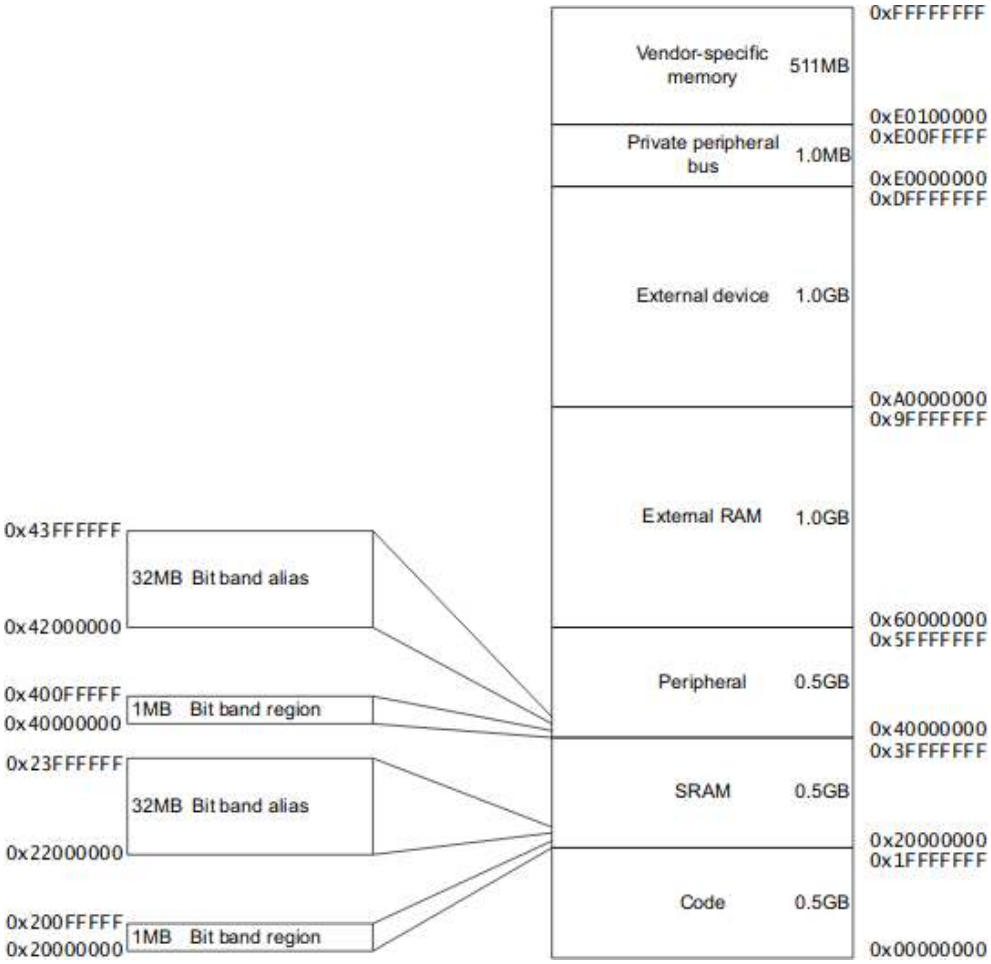


Figuur 12.1 Figuur: Compilation voor Microcontrollers

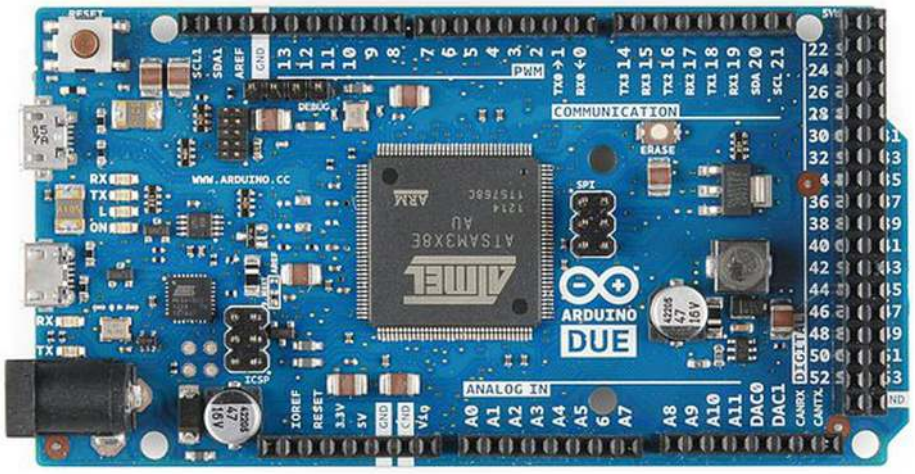
dit anders: alle code en data is bij het flashen in het ROM geheugen opgeslagen. Bij het uitvoeren wordt dit niet naar RAM geheugen verplaatst, maar wordt alles zoveel mogelijk direct uit het ROM gelezen. Variabele data, dus data die kan veranderen, heeft een initiële waarde in het ROM staan, maar wordt naar het RAM gekopieerd om de data aan te kunnen passen. In Figuur Memory Map van de Arduino Due zien we een voorbeeld van een memory-map van een microcontroller: de Arduino Due. In latere vakken komt het programmeren van microcontrollers in C++ aan bod, waarbij de Arduino Due gebruikt zal worden. Zoals op de afbeelding is te zien, wordt al het geheugen in een enkele map geplaatst, ongeacht wat voor geheugen het is. Ook externe devices (peripherals) zijn in de memory-map vertegenwoordigd. De totale memorymap van de Arduino Due is 4GB (het maximaal adresseerbare geheugen met 32 bits en 1 byte per adres), maar dat betekent niet dat de Arduino Due ook zoveel geheugen heeft. Een deel van de adressen is niet standaard in gebruik, of verwijst naar devices die geen deel uitmaken van wat we normaal het "geheugen" zouden noemen.

Tot slot is het bij een bare-metal microcontroller niet mogelijk op de BIOS te vertrouwen om bepaalde taken uit te voeren: de BIOS is op zichzelf al een complex stuk software (of beter gezegd, firmware) dat niet op een standaard microcontroller aanwezig zal zijn. Meer nog dan bij het schrijven van een OS worden peripherals (randapparatuur, alles dat zich buiten de CPU en het geheugen bevindt) aangestuurd door direct naar geheugenposities te schrijven en direct uit geheugenposities te lezen: alle peripherals zijn direct in de memory map aanwezig. In code voor operating systems gebeurt dit ook wel (zoals te zien in ons C++ voorbeeld vorige week, waarbij naar een geheugenrange geschreven werd om tekst te printen), maar bij microcontrollers is dit nog meer gebruikelijk.

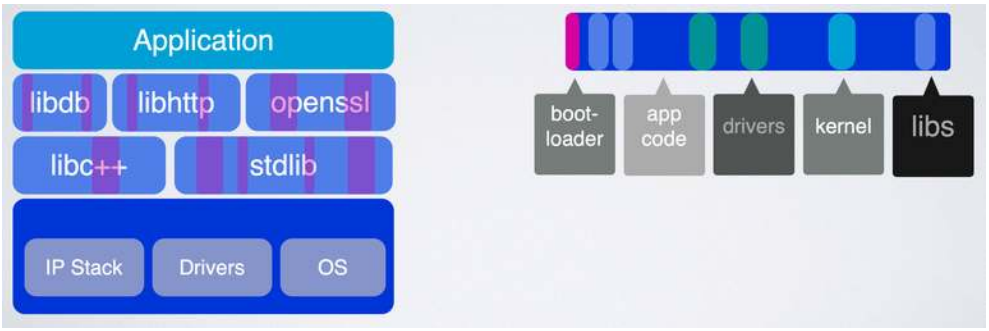
```
1 // Maak een GPIO pin high op een Arduino Uno
2 PORTB |= 0x01 << 12;
3
4 // Maak een GPIO pin high op een Arduino Due
5 PIOB->PIO_SODR = 0x01 << 27;
6
7 void make_pin_high()
8 { PORTB |= 0x01 << 12; }
```



Figuur 12.2 Figuur: Memory Map van de Adruino Due



Figuur 12.3 Figuur: De Arduino Due



Figuur 12.4 Figuur: Opbouw IncludeOS

```
9
10 void make_pin_high()
11 { PIOB->PIO_SODR = 0x01 << 27; }
12
13 make_pin_high();
```

12.2 Bibliotheken voor MC programmeren

12.2.1 Microcontroller Libraries: Hardware Abstraction Layer

Hoewel bare-metal programmeren op een microcontroller een hoop meer handwerk vereist, kunnen we nog steeds gebruik maken van de eerste tool die een programmeur voor handen heeft: compositie. Door een probleem in kleinere subproblemen op te delen kan herbruikbare code gemaakt worden, die in de vorm van libraries te delen is tussen verschillende projecten. Hiermee wordt een abstractielaag toegevoegd tussen de hardware en de applicatie: de Hardware Abstraction Layer (HAL). Jullie hebben al kennis gemaakt met `RPi.GPIO` om pinnen op de Pi aan te spreken, en mogelijk ook met *Wiring* dat programmeren voor de Arduino's mogelijk maakt. Later in de opleiding zullen jullie met Hwlib te maken krijgen: een HAL-library die op de HU ontwikkeld is om het programmeren van microcontrollers toegankelijker te maken. Ook de BIOS is niet veel anders dan een hardware abstraction layer.

12.2.2 Library Operating Systems en Unikernels

Een tussenvorm tussen een bare-metal microcontroller en een operating system is een zogenaamd Library OS. Hierbij worden de gebruikelijke diensten die een OS levert aangeboden in de vorm van libraries. Dit maakt het mogelijk complexe functies te gebruiken alsof de applicatie op een OS draait, maar in werklelijkheid is deze functionaliteit in de applicatie opgenomen. Het resultaat is een binary die zowel de applicatie-code als (de relevante delen van) de OS code bevat: een zogenaamde unikernel. Een unikernel heeft een enkele adresruimte, dus geen onderscheid tussen kernel-space en user-space, en kan in het geheel op microcontrollers ingezet worden. Een voorbeeld van een library OS is IncludeOS.