# Raspberry Pi Labo's Besturingssystemen

Bart Coppens

27 juli 2021

# Hoofdstuk 1

# Inleiding

De twee hoofddoelen van deze labo's zijn enerzijds om jullie kennis te leren maken met enkele fundamentele concepten in besturingssystemen, en anderzijds om wat feeling te krijgen met het programmeren van systeemcode op echte hardware. Dat laatste houdt ook in dat jullie wat meer vertrouwd raken met het lezen van technische documentatie. Een nevenbedoeling is dat jullie (hopelijk) dit ook leuk genoeg vinden om achteraf eens verder mee te experimenteren / prutsen. Dus als jullie thuis het juiste model Raspberry Pi zouden liggen hebben, en jullie je ooit zouden vervelen, kan je thuis perfect verder features toevoegen aan deze code.

Moderne besturingssystemen worden niet volledig geschreven in assembly. De meeste code is gewoon in een hoogniveauprogrammeertaal geschreven, en enkel de platform-afhankelijke codefragmenten die interageren met specifieke hardware-features die niet zomaar aangesproken kunnen worden uit die hoogniveauprogrammeertaal, wordt assembly gebruikt. Die assembly kan dan ofwel in een apart bestand zitten, of zal in sommige gevallen gewoon in inline assembly in je hoogniveaucode geschreven zijn. Aangezien we hier systeemcode schrijven, is het wel nog steeds belangrijk om goed met pointers om te kunnen gaan in de code. Onze codebase is grotendeels geschreven in moderne C++ (we maken gebruik van C++17 features op sommige plaatsen), met op een aantal plaatsen ARMv8-A assembly. Ik heb geprobeerd om de C++ code zo veel als mogelijk 'modern' te houden om zoveel mogelijk fouten reeds bij het compileren op te sporen, en om het programmeren zelf wat eenvoudiger en eenduidiger te maken dan gewone C code of C++ code in een 'oudere' stijl.

Van een aantal bestanden die belangrijk zijn voor het begrijpen van de code, en de bestanden die jullie moeten aanpassen, is de broncode volledig meegeleverd. Een aantal andere bestanden heb ik enkel de binaire code meegeleverd: voldoende om de practica te maken, maar dat zorgt er ook voor dat je achteraf eens kan proberen die functionaliteit zelf te maken. De meeste volwaardige besturingssystemen zijn cross-platform (ze kunnen gecompileerd worden voor meerdere doelplatformen), en dan wordt de code meestal gestructureerd in verschillende directories afhankelijk van of de code volledig platform-onafhankelijk is, code is die specifiek is voor een bepaalde processorarchitectuur, en zelfs code die specifiek is voor bepaalde hardware-devices. In ons geval zal onze code enkel werken voor één specifieke processor op één zeer specifiek hardware-platform met specifieke hardware-devices, daarom zitten alle bestanden voor de overzichtelijkheid in 1 directory. Voel jezelf vrij om die splitsing desnoods later zelf te maken :-)

De files die jullie zullen moeten aanpassen, kunnen jullie vinden op
`https://github.ugent.be/bcoppens/besturingssystemen-raspberrypi`.

**Let op:** dit is de eerste keer dat we dit proberen. Ga er dus van uit dat er dingen niet duidelijk zullen zijn, ontbreken in de opgave, of misschien zelfs niet werken. Het kan ook goed zijn dat er dingen niet zo goed lukken als we hopen dat het zou lukken met onze huidige uitleg. Ik zou dat dan ook uiteraard graag weten, om dit allemaal te verbeteren. Je kan al je feedback sturen naar Bart.Coppens@UGent.be, of spreek me aan op Slack!

aanpassen naar gelang de situatie in 2021

Dus dit eerste practicum is **niet verplicht**, en je hoeft ook niets in te dienen. (Maar dat mag wel, mail me gewoon de files waarvan staat dat je ze moet indienen dan.)

Als ik problemen / onduidelijkheden merk/fix tijdens de komende weken, zal ik dat communiceren via Slack om zo de andere studenten niet te storen. Ik heb op de besturingssystemen-slack een apart Slack-kanaal aangemaakt voor deze practica: #raspberry.

aanpassen naar gelang de situatie in 2021

Dit is versie 0.4 van dit document.

# Hoofdstuk 2

# De ontwikkelomgeving

In dit hoofdstuk gaan we wat dieper in op de omgeving die we zullen gebruiken, en hoe we deze gaan aanspreken.

## 2.1 De gebruikte hardware

Aangezien we gebruik gaan maken van echte hardware, moeten we een keuze maken voor een geschikt hardwareplatform. Daar zijn een aantal overwegingen te maken: kostprijs, uitbreidbaarheid, de onderliggende architectuur, documentatie, ... We hebben hierbij gekozen voor de *Raspberry Pi 4 Model B (2GiB)*. Jullie hebben er misschien thuis zelf eentje liggen, net omdat ze zo goedkoop en krachtig tegelijkertijd zijn.

Aangezien we systeemsoftware gaan ontwikkelen, volstaat het niet om te zeggen dat we zomaar een Pi gaan gebruiken. Het feit dat we gebruik gaan maken van een Pi 4(B) is belangrijk: de verschillende modellen Raspberry Pi maken onderliggend gebruik van verschillende platformen, die niet zomaar compatibel zijn.

Laten we beginnen bij het hoogste niveau. De Raspberry Pi 4 Model B maakt gebruik van een Broadcom *BCM2711* System-on-Chip (SoC). Deze chip is niet enkel de CPU, maar bevat eigenlijk een heel systeem (vandaar de naam) met een GPU, controllers voor geheugenbeheer, controllers die USB2/3 kunnen aansturen, controllers voor seriële poorten aan te sturen, ethernet, ... Vorige generaties Raspberry Pi maken gebruik van vorige generaties van de BCM-SoC, elk met iets andere configuraties, controllers, en manieren waarop deze aangestuurd moeten worden, waardoor het (op een laag niveau) aanspreken van de seriële controllers, de GPU, etc. zal verschillen tussen de verschillende generaties van Raspberry Pi. Zelfs de onderliggende CPU-architectuur verschilt sterk tussen de generaties Raspberry Pi!

De BCM2711 bevat als processor een quad-core variant van de *ARM Cortex-A72*. Deze voorziet een specifieke microarchitecturale implementatie van de 64-bit *ARMv8-A* ISA. Eerdere BCM-generaties bevatten andere processoren, sommigen daarvan implementeerden bijvoorbeeld enkel een 32-bit ISA.

De ARMv8-A architectuur is vrij flexibel, en heeft zowel een deel van de specificatie die bedoeld is voor 64-bit code, en een deel voor 32-bit code (deze is eigenlijk een voortzetting van de oudere 32-bit ISA). Deze worden respectievelijk de AArch64 en AArch32 execution states genoemd. **We gaan er voor kiezen om onze besturingssysteemcode te schrijven voor de AArch64 uitvoeringstoestand.**

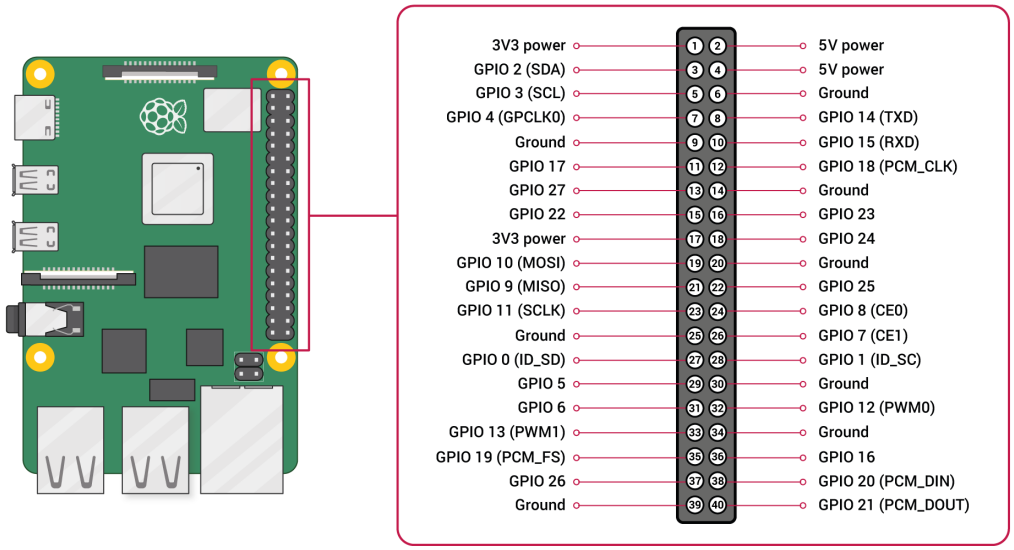
Dit wil uiteraard ook zeggen dat, om systeemsoftware zoals een besturingssysteem te schrijven, je zowel kennis moet hebben van de details van hoe de SoC werkt, van de processor, en van de ISA, en van de hardware die je daar bovenop nog wil aansturen. Deze documentatie is zeer uitvoerig. (En dan is niet eens alle relevante documentatie publiek beschikbaar, helaas!) Het is dus niet vanzelfsprekend om daar als beginner de nodige informatie in terug te vinden, en al zeker niet omdat deze zeer technisch is, en er vaak impliciet wordt van uit gegaan dat je al op de hoogte bent van bepaalde concepten.

**Vandaar ook dat we lichtjes geannoteerde versies van deze documenten voorzien, waarbij we in de opgave uitdrukkelijk zullen zeggen welke secties/figuren van welk document je wanneer moet lezen!**

## 2.2 Praktisch: aansluiten van de Pi

Het idee is dat we jullie in groepjes een Pi 4 meegeven, en dan dus ook zelf de seriële kabel op jullie laptop moeten kunnen aansluiten. We hebben ter voorbereiding reeds onze eigen bordjes in elkaar gestoken. Het lijkt echter wel nog steeds nuttig om in deze sectie kort te beschrijven hoe je dit kan doen als je thuis zelf een eigen Raspberry (het juiste model) zou liggen hebben.

Op de Raspberry Pi 4 model B is een dubbele rij pinnen voorzien. De functionaliteit van deze pinnen verschilt van pin tot pin: sommigen zijn doorverbonden met de elektrische grond, sommigen met 3.3V, sommigen met 5V, en sommigen met I/O pinnen van de BCM2711 (die dan daarop geconfigureerd kunnen worden). Je mag dus niet zomaar elke kabel gelijk waar aansluiten. Daarom dus ook dat het belangrijk is om een uniform referentiekader te hebben in hoe we deze pinnen nummeren. We gebruiken hiervoor de standaard-nummering die je op de Raspberry-website kan vinden[1], en die hieronder ook te zien is:




Op ons bordje is de *fan* reeds aangesloten als volgt: de zwarte ground-kabel op pin 6 (Ground), en de rode kabel op pin 1 (3V3 power). Onze *seriële kabel* sluiten we dan als volgt aan: de zwarte grond-kabel op pin 14 (Ground), de witte kabel op pin 8 (GPIO 12/TXD), en de groene kabel op pin 10 (GPIO 15/RXD). De rode kabel wordt **niet** aangesloten!

Het is de bedoeling om de aan/uit-knop op de stroomkabel zelf te gebruiken om het systeem te herstarten. Zodanig moet je niet elke keer je stroomkabel uittrekken/insteken als je code blijft hangen. Voor het tweede labo kan je optioneel een scherm aan je Raspberry Pi hangen, hiervoor kan je een microHDMI-kabel in de linker-HDMI-poort steken (dit is de poort die het dichtst bij de stroom-USB-C-aansluiting zit).

## 2.3 Praktisch: de ontwikkelomgeving

We gaan proberen werken op een uniforme ontwikkelomgeving. De compiler zelf zal draaien op Linux. We kunnen niet zomaar gelijk welke compiler gebruiken, we hebben er een nodig die AArch64-code kan genereren. (Een compiler die code kan genereren voor een ander type platform dan het platform waarop de compiler zelf uitvoert, is een *cross-compiler*.) We maken gebruik van de `g++` (versie 8.3) compiler. Om het onderscheid te maken met de gewone `g++`-compiler, heeft de cross-compiler een andere bestandsnaam. Al onze cross-tools hebben `aarch64-linux-gnu-` als prefix, dus: `aarch64-linux-gnu-g++` is de naam van onze cross-`g++`.

[1] Op `https://www.raspberrypi.org/documentation/usage/gpio/README.md`, waarvan die figuur ook komt.

Omdat jullie niet allemaal Linux draaien[2], hebben we een Linux-gebaseerde VirtualBox image ter beschikking gesteld, die je kan downloaden van
`https://users.elis.ugent.be/~bcoppens/BesturingssystemenRaspberry.vdi`. Het idee is dat je die (in principe) enkel zou moeten gebruiken voor de bestanden te compileren en dan de bootloader te starten. Om dit te doen, ga je de VM best als volgt configureren:

- Maak een gedeelde map aan tussen je host-computer en de VM. Je kan die in de VM mounten door een (permanente) auto-mount share toe te voegen aan je VM met de naam *shared*, die zou dan normaalgezien in de VM zelf beschikbaar zijn onder `/media/sf_shared/`. Je kan dan buiten je VM de code editeren (je mag dat natuurlijk ook altijd in de VM zelf doen!), en dan via de gedeelde map de code in de VM zelf compilen & booten.

mando ge-

- Je moet de seriële poort delen tussen je host-computer en de VM. Je doet dit in de Settings van je VM, bij *Serial ports*, *Port 1*, *Enable Serial Port* aanvinken, en dan de optie *Host Device* kiezen, en dan bij path geef je de file in voor de juiste seriële poort. Dat doe je als volgt:
  - *Op een Linux-host:* Linux gaat die convertor normaalgezien automatisch herkennen. Het device daarvan zal `/dev/ttyUSB0` zijn (er van uitgaande dat je enkel onze serieel-naar-USB convertor hebt ingeplugd als seriële poort).
  - *Op een Windows-host:* Installeer eerst zelf de drivers van `http://www.silabs.com/Support%20Documents/Software/CP210x_Windows_Drivers.zip`. Dan plug je je device in, en bepaal je de COM-poort waarop dat device zit. Ga daarvoor naar apparaatbeheer, ga naar Serial devices, normaal zou je er daar exact 1 moeten zien: de onze. Tussen de haakjes gaat dan normaal het COM-device vermeld staan waarop dit gemapt is, *COM3* bijvoorbeeld. Dat laatste is wat je moet invullen in VirtualBox.
  - *Op OSX:* Heb ik eigenlijk geen idee van :-) De kans bestaat dat je een driver moet installeren, en het zou kunnen dat die devices `/dev/cu.SLAB_USBtoUART` heten daar, maar dat zal je zelf eens moeten opzoeken vrees ik. (Maar laat het me zeker weten, dan pas ik dit tekstje aan voor volgend jaar!)

- Als je toch de code op je eigen Linux-machine zou proberen in plaats van in de VM, moet je het device aanpassen in de `bootloader.py` naar het device op je eigen machine, zoals bijvoorbeeld `/dev/ttyUSB0`.

Eens je de VM geconfigureerd hebt en de seriële adapter in je USB-poort zit, kan je de VM opstarten.
**Log in als gebruiker `root` met als wachtwoord `student`**
Als je een azerty-toetsenbord hebt, en je wil dit ook in de VM gebruiken, dien je `apt-get update && apt-get install console-data` uit te voeren, en daar dan azerty te selecteren.
Door in de directory met de broncode te gaan staan in de console in je VM, en dan `make` uit te voeren, zullen je bestanden gecompileerd / geassembleerd / gelinkt worden. Let altijd op dat dit succesvol is, en dat er dus geen fouten in je code zitten!

## 2.4 Praktisch: booten van een kernel

Eens alle broncodebestanden gecompileerd zijn, worden deze samengevoegd tot 1 bestand. Het finale bestand (`kernel.bin`) kan dan worden ingeladen in het geheugen van de Raspberry Pi, en als we dan de instructiewijzer laten wijzen naar het juiste adres in dat ingeladen stuk geheugen, zal deze kernelcode uitvoeren.
Maar hoe krijgen we die `kernel.bin` dan effectief geboot? Op de Raspberry Pi zit firmware die daarvoor zal zorgen. Als je in de Pi een (FAT-geformatteerd) micro-SD-kaartje steekt, zal die firmware informatie lezen van dat SD-kaartje. De firmware heeft een aantal standaardinstellingen (zoals de naam van het bestand waar je kernel inzit), die je zelf kan instellen door deze waarden in het bestand `config.txt` te schrijven[3]. Een aantal van de waarden die voor ons van belang zijn, en die wij

[2] Voor degenen die wél Linux draaien: Debian stable heeft de 8.3-versie van `g++` die we gebruiken, daar kan je dus makkelijk als volgt de benodigde tools installeren: `apt-get install g++-aarch64-linux-gnu python3-serial`
[3] De meeste van de instellingen kan je terugvinden op `https://www.raspberrypi.org/documentation/configuration/config-txt/`

dus al hebben ingesteld, zijn de naam van de kernel, het adres waarop deze geladen moet worden, het feit dat we de seriële poort willen gebruiken, het feit dat we een AArch64-kernel willen booten, en de juiste configuratie-optie voor de adressen van de devices (meer details over dat laatste zal je later in dit document zien).

In normale omstandigheden ga je je kernel image (met bijhordende configuratiebestand) gewoon op je SD-kaartje willen zetten en daarvan booten. In dit geval gaan we echter een kernel ontwikkelen, en als we elke keer we een probleem merken en iets moeten aanpassen eerst het kaartje uit de Raspberry Pi moeten halen, in een SD-kaartlezer steken, nieuwe kernel overzetten, kaartje weer uit de lezer halen en de Raspberry Pi steken, ... dan verliezen we veel tijd (om nog maar te zwijgen van het feit dat dat best gefoefel is om dat kaartje er uit te halen). Daarom hebben wij een kaarje voorzien met een custom *bootloader*. Diens enige taak is om jouw zelfgeschreven kernel image via de seriële poort op de Raspberry Pi in te lezen, op de juiste plaats in het geheugen te laden, en dan die vers-ingeladen kernel uit te voeren.

Op je PC/laptop zal je dan een commando moeten uitvoeren om je vers-gecompileerde kernel naar die bootloader te sturen. Dat doe je met het meegeleverde `bootloader.py` scriptje. Dit script wacht eerst op een teken van leven van de bootloader op de seriële poort, en zal dan de data doorsturen. Je gaat dus best als volgt te werk: check dat je USB-naar-serieel adapter aangesloten is, check dat de settings in je VirtualBox deze correct delen met de VM, check dat het SD-kaartje met de bootloader in de Raspberry Pi zit, start de `bootloader.py` op je host computer door `python3 ./bootloader.py` uit te voeren, en druk dan op de *drukknop* om stroom te leveren aan de Raspberry Pi, die dan de bootloader zal uitvoeren, die dan de kernel zal laden. Dat ziet er in de uitvoer van `bootloader.py` dan zo uit (het kan even duren eer de kernel overgezet is, de seriële connectie is niet zo snel):

```
$ python3 ./bootloader.py
Using serial device /dev/ttyUSB0 (you will need to change this if you're running Linux
natively rather than in the VM)
Waiting for bootloader to initialize...
Initialising SDRAM 'Micron' 16Gb x2 total-size: 32 Gbit 3200
[...]
Bootloader initialized!
Sending kernel with size 276676 and checksum 1052349
Kernel size confirmed. Sending kernel
Wrote kernel
Done!
```

# Hoofdstuk 3

# Labo 1: Simpel virtueel geheugen

In dit eerste labo gaan we kennis maken met het opzetten van virtueel geheugen dat verwijst naar fysiek geheugen.

In een eerste stap gaan we een *identische mapping* opzetten die een virtueel adres zal vertalen in een fysiek adres dat exact dezelfde waarde heeft als het te vertalen virtueel adres. Dit wil dus zeggen dat bijvoorbeeld virtueel adres 0 op fysiek adres 0 gemapt zal worden, virtueel adres 0x10000 op fysiek adres 0x10000, etc.

Echter, uit C en C++ ga je allicht onthouden hebben dat een pointer naar het adres 0 een ongeldige pointer is. (De waarde 0 kan je herkennen in de typische naamgevingen `NULL` en `nullptr`, etc.) Meer nog, als je een nullptr zou proberen dereferencen, zal je programma crashen. In ons geval is fysiek adres 0 echter een perfect geldig adres. En dus zal een access naar virtueel adres 0 omgezet worden in een access naar fysiek adres 0, en dat is een perfect geldige access voor de hardware. Daarom zal je in het tweede deel van dit labo er voor zorgen dat een toegang naar virtueel adres 0 ongeldig is, maar dat fysiek adres 0 nog steeds bereikbaar is via een ander virtueel adres (het zou immers suboptimaal zijn om die fysiek beschikbare ruimte kwijt te spelen).

## 3.1 Inleiding tot AArch64

Alvorens we in detail kunnen begrijpen hoe virtueel geheugen op de AArch64 architectuur werkt, is het belangrijk om een aantal achterliggende concepten en termen van deze van deze architectuur te kennen.

De AArch64 heeft *31 algemeen bruikbare 64-bit registers*, genaamd `x0` tot en met `x30`. Deze registers overlappen met de 32-bit registers `w0` tot en met `w30`, waarbij een leesoperatie de onderste 32 bit zal lezen uit het corresponderende x-register, en een schrijfoperatie de onderste 32-bit van het corresponderende x-register zal vervangen (en de hoogste bits op 0 zal zetten). Er is bovendien een 64-bit stack-register, een 64-bit program counter-register, en een 64-bit zero register, die je afhankelijk van de specifieke instructie zal kunnen aanspreken. Op deze registers kan je allerhande bewerkingen uitvoeren met verschillende instructies die allen beschreven staan in de manual.

Alle instructies zijn 4 bytes groot. Dat wil dan ook zeggen dat je niet zomaar gelijk welke 64-bit waarde in één keer naar een register kan schrijven. Vele instructies hebben verschillende varianten: eentje waarbij een van alle operands een 32-bit of 64-bit register zijn, en eentje waarbij één van de operands een simpele constante waarde is (een *immediate*). De architectuur heeft een aantal truukjes waarbij je ook specifieke grote waarden toch kan encoderen als een immediate in een 32-bit instructie. Dat zal uiteraard in niet alle gevallen lukken. In die situaties kan men altijd de 64-bit waarde in het geheugen schrijven, en een load-instructie uitvoeren van het adres waar die 64-bit waarde staat. De assembler voorziet een speciale syntax om dit automatisch te doen: `ldr x0, =123456` (dit is de *Load Register* instructie) zal voorzien dat de constante 123456 in je binaire bestand geplaatst wordt, en dat de load-operatie naar `x0` verwijst naar het adres waarop die constante te vinden is.

Procedure-oproepen gebeuren iets anders dan je misschien gewend bent van de Intel architecturen. Daar waar je allicht uit het vak Computerarchitectuur je wel zal herinneren dat in de Intel architecturen de CPU het terugkeeradres van de call-instructie automatisch op de stapel plaatst, wordt het terugkeeradres op AArch64 in het `x30`-register geplaatst! Dit `x30`-register dat dan het terugkeeradres bevat, wordt ook wel eens het *link register* genoemd. Indien men `x30` gebruikt om er een terugkeeradres in te schrijven / van te lezen, kan je dit in de assembly ook schrijven als het `lr`-register, de assembler vertaalt dit automatisch

naar `x30`. De instructies die zo'n functie-oproep verzorgen op AArch64 zijn `bl` (*Branch with Link*, waar je een functieoproep doet naar een label/offset) en `blr`-instructies (*Branch with Link to Register*, waar je een functieoproep doet naar een functie die zich bevindt op het adres dat als operandregister meegegeven wordt). Om terug te keren uit een functie wordt dan de `ret`-instructie gebruikt, die zal springen naar het adres dat in het link register zit.

Naast de gewone registers zijn er ook *systeemregisters*. Deze registers worden onder andere gebruikt om specifieke informatie over de processor (modelnummer, hoeveel cores er aanwezig zijn, instellingen van virtueel geheugen, en veel veel meer) te lezen of aan te passen. In tegenstelling tot de 31 algemeen bruikbare registers, kan je deze registers enkel lezen en schrijven met specifieke systeemregister-instructies (zoals `mrs` om de waarde van een systeemregister te kopiëren naar een algemeen bruikbaar register, en `msr` om de waarde van een algemeen register te schrijven naar een systeemregister).

De *Arm® Architecture Reference Manual* bevat in deel *C: The AArch64 Instruction Set* een volledig overzicht van alle instructies. (Je zal dit normaal niet nodig hebben.)

Zoals je intussen allicht uit de lessen van het vak besturingssystemen al weet, hebben de meeste ISA's een concept van een privilegeniveau. Op de Intel architecturen heb je enerzijds de verschillende privilegeringen, het verschil tussen user mode/supervisor mode (waarbij je virtueel geheugen van de besturingssysteemkernel kan afschermen van gebruikersapplicaties, en omgekeerd), eventueel een hypervisor, etc. Gelijkaardige concepten bestaan uiteraard ook bij Arm architecturen. De AArch64-architectuur definieert hierbij vier verschillende privilegeniveau's, en CPU-manufacturers kunnen kiezen welke ze hiervan implementeren. De Cortex-A72 implementeert alle vier deze niveau's. Deze niveau's worden *Exception Levels* (ELs) genoemd: *EL0* tot en met *EL3*. Het idee is dat applicatiecode in EL0 draait, een besturingssysteem in EL1, een hypervisor in EL2, en een secure monitor in EL3. We gaan ons beperken tot EL0 en EL1.

Heel wat systeeminstellingen verschillen tussen de verschillende exception levels. Daarom zal je bij de meeste systeemregisters een suffix zien die het exception level aangeeft. Zo is `VBAR_EL1` het systeemregister dat het *Vector Base Address* bevat voor exceptions die in het besturingssysteem op EL1 afgehandeld worden, `VBAR_EL2` is het Vector Base Address voor de exceptions voor de hypervisor op EL2, etc. Opgelet, de precieze betekenis van de specifieke bits in sommige van deze systeemregister kan verschillen tussen de exception levels, lees dus altijd goed de manual als je die registers uitleest of aanpast!

Als er een exception opgeworpen wordt, zal er een exception handler op het juiste EL opgeroepen worden. Daarbij worden er door de processor een aantal extra systeemregisters ingesteld, die je toelaten om te achterhalen wat de oorzaak van de exception is. Eerst en vooral is er het *Exception Syndrome Register* `ESR_EL1`, dat informatie bevat over het type exception/de exception class (instruction abort, data abort, alignment fault, ...). Als je dit nodig zou hebben, kan je vanaf pagina D13-2919 van de Arm manual achterhalen welk bitpatroon welke exception voorstelt. Als er een probleem was met een geheugenaccess, ben je allicht ook geïnteresseerd in het geheugenadres dat niet geaccesst kon worden. Dit geheugenadres zit in het *Fault Address Register* `FAR_EL1`. Tot slot kan je ook achterhalen wat het adres was van de instructie die de fault veroorzaakte: net zoals we bij gewone functie-oproepen een link register hebben om het terugkeeradres in te bewaren, hebben we voor exceptions een *Exception Link Register* `ELR_EL1`. In het geval van faults bevat dit het adres van de instructie die de fault veroorzaakte. (Er is uiteraard ook een instructie `eret` die gebruikt wordt om *terug te keren uit een exception handler*. Deze instructie zal de waarde uit `ELR_EL1` gebruiken om de uitvoering op verder te zetten na de exception handler. Nuttig om weten is dat je dit register zowel kan uitlezen als kan aanpassen, zodat je kernel kan instellen naar welke codelocatie op EL0 je wil terugkeren na je exception handler op EL1.)

## 3.2 De fysieke geheugenlayout

Alvorens we virtueel geheugen kunnen gebruiken, moeten we natuurlijk weten hoe de fysieke adresruimte in elkaar zit. Dit klinkt misschien als een simpel iets waar je bytes in je RAM accesst, maar dat is het niet. In de fysieke adresruimte kan je niet alleen lezen/schrijven naar RAM, maar in deze ruimte zitten ook *devices*. Als je bijvoorbeeld de seriële poort wil configureren, zal je dit doen door te schrijven naar een specifiek adres, dat dan door de SoC gemapt wordt op configuratieregisters van het seriële device. Aangezien we onze code in Labo 1 tekst over het seriële device stuurt, en in Labo 2 iets op het scherm zal schrijven, is het dus belangrijk dat we weten waar precies in de fysieke adresruimte het RAM zich bevindt, en waar de devices zich bevinden.

Dus, hoe en waar wordt die fysieke adresruimte bepaald? Dat wordt bepaald door de BCM2711 SoC. In het bijzonder, het is eigenlijk de VideoCore GPU/VPU die achter de schermen alle devices beheert en met elkaar laat communiceren en dus ook bepaalt op welke fysieke adressen welke devices zitten. (Het is overigens ook die VideoCore die de ARM processor zal opstarten.)

**Lees daarom nu Chapter 1 van `bcm2711--annotated.pdf` om een zicht te krijgen van hoe de fysieke adresruimte op de BCM2711 er uitziet.**

**OPGELET!! Die annotaties zijn niet zichtbaar in de Ufora-PDF-viewer of de standaard PDF-viewers van browsers. Je zal de PDFs best downloaden en in een externe viewer bekijken!**

Zoals ik in de annotaties in die PDF reeds aangaf, gaan we er in onze setup voor kiezen om de VideoCore de *Low Peripheral Mode* te laten gebruiken. We hebben dit gedaan door in de `config.txt` op het micro-SD-kaartje `arm_peri_high=0` mee te geven.

## 3.3 Virtueel geheugen op AArch64

Elke architectuur heeft wat zijn eigen manier om virtueel geheugen op te zetten en te configureren. De verschillende architecturen hebben ook (meer of minder) verschillende features en configuratiemogelijkheden. AArch64 in ARMv8-A is vrij flexibel en krachtig wat virtueel geheugen betreft: het heeft veel features, parameters en instellingen die je kan gebruiken bij het opzetten van virtueel geheugen. Bovendien zijn er in de verschillende instructiesetuitbreidingen bovenop ARMv8-A die nog extra features en instellingen toevoegen. Dit maakt het helaas ook vrij complex om aan de hand van de officiële Arm Architecture Reference Manual alles te begrijpen, die meteen alles tegelijkertijd en door elkaar, op een vrij laag niveau uitlegt. Die Manual gaat bovendien wat voorbij aan het uitleggen van hoe alles met elkaar interageert in een beknopt high-level overzicht. Gelukkig heeft Arm ook een apart document, *ARMv8-A Address Translation* dat weliswaar technisch minder gedetailleerd is, maar wel een beknopt en duidelijk overzicht geeft van de mogelijkheden.

**Lees daarom nu de tekst van `armv8_a_address translation--annotated.pdf`. In de annotaties staat, naast extra informatie en duiding, ook informatie over de keuzes die we zullen maken voor bepaalde parameters in deze labo's. Bovendien staat ook duidelijk aangegeven welke secties/paragrafen je mag *overslaan* bij het lezen. Je moet dus alles lezen, behalve secties die expliciet aangegeven of geschrapt staan als niet-te-lezen.**

Aan de hand van dat document heb je een hoogniveau inzicht van hoe virtueel geheugen en adresvertaling in elkaar zitten op AArch64. We hebben dus een adresvertaling die uit verschillende *levels* bestaat. Een entry op een specifiek level kan ofwel *invalid* zijn, ofwel een *block entry* zijn die verwijst naar een hele geheugenregio en waarvan de grootte afhangt van het level van de adresvertaling waarop die entry zich bevindt, ofwel is het een *table entry* die verwijst naar een tabel op het volgende level van de adresvertaling, ofwel is het (op level 3) een *page entry* die verwijst naar 1 enkele pagina (waarvan de grootte op voorhand vast ligt). Maar hoe groot is nu de range van een block entry op level 1, en hoe groot is die op level 2? En welke bits uit het virtueel adres worden gebruikt om te indexeren? Dat hangt af van de paginagrootte. Je kan die waarden in principe eenvoudig zelf berekenen. Maar het is allicht handiger om dit gewoon te bekijken in de manual. **Open daarom `armv8_arm--annotated.pdf`, en bekijk Table D5-19 op pagina D5-2549**, die deze informatie geeft voor onze paginagrootte van 4KiB.

Bovendien hebben we geleerd uit de high-level overview dat de adresvertaling op ARM, afhankelijk van de specifieke configuratie, soms op level 0 zal starten, soms op level 1, ... Dit is omdat je het systeem op voorhand kan configureren om aan te geven dat je virtuele adressen slechts een beperkt deel van de adresruimte in beslag zullen nemen. Dus stel je voor dat je de virtuele adresruimte zodanig beperkt hebt dat je eigenlijk volstaat met één gevulde paginatabel van 512 entries. Dan zou het een verspilling zijn om dan voor elke adresvertaling eerst de hoogste-orde bits te moeten gebruiken om de tabel op 0 te indexeren (die slechts 1 geldig element bevat, dat wijst naar de level 1 tabel), om dan vervolgens in de tabel op level 1 opnieuw slechts 1 geldige entry te hebben, etc. Om dat te voorkomen, gaat de architectuur er van uit dat de adresvertaling start op het eerste level dat effectief nuttig is. Maar hoe zit dat dan concreet? In de code die we reeds voorzien, hebben we de waarde van `TCR_EL1.T0SZ` zo geconfigureerd dat we onze adressen (voor `TTBR0`) zullen limiteren tot adressen met een numerieke waarde kleiner dan 64GiB. We doen dit door de `T0SZ` in te stellen op de waarde 28 (en 4KiB pagina's te kiezen). Je zou

dan opnieuw kunnen beredeneren op welk level de adresvertaling moet starten, maar opnieuw heeft de manual hier een handig overzicht van. **Open daarom `armv8_arm--annotated.pdf`, en bekijk Table D5-12 op pagina D5-2536**, die deze informatie geeft voor onze paginagrootte van 4KiB. Hieruit kan je achterhalen op welk level de adresvertaling zal starten, gegeven de door ons geconfigureerde waarde van `T0SZ`. Tot slot kan je **op de volgende pagina nog eens kijken naar Figuur D5-7**, die nog eens schematisch aangeeft hoe groot de regio's zijn die beschreven worden door één enkele block entry op een specifiek level.

## 3.4 Een kernel in C++ schrijven en moderne C++ features

Zoals reeds gezegd gaan we onze kernel in C++ schrijven, en niet, zoals bijvoorbeeld de Linux kernel, in C. Dat klinkt misschien een beetje vreemd, maar dat is het zeker niet. Moderne C++ heeft heel wat features die ons gaan kunnen helpen om makkelijker correcte code te schrijven.

Je zou misschien denken dat de meest voordehandliggende reden om voor C++ te kiezen de zeer uitgebreide C++ standard library is. Die gaan we helaas niet zomaar kunnen gebruiken, omdat we hier stand-alone kernelcode aan het schrijven zijn. Dus net zoals we als we onze kernel in C hadden geschreven, niet zomaar gebruik kunnen maken van een aantal standaardfuncties zoals `malloc` en `printf`, gaan we in C++ bovendien niet zomaar gebruik kunnen maken van `new` en alle code die achter de schermen geheugenbeheer nodig heeft. In het algemeen kunnen we geen gebruik maken van features die gebruik maken van `libc` of `libstdc++`. Voor wat specifieke functionaliteit die we toch nodig hebben uit die bibliotheken (zoals bijvoorbeeld `memset`), hebben we zelf vervangingen voorzien die we dan op de gepaste plaatsen beschrijven.

Waarom kiezen we dan wél voor C++? Een van de problemen met C is dat het zeer makkelijk is om per ongeluk fouten te maken in het typesysteem van de taal, waarbij het achteraf moeilijk is, zowel voor mensen als voor compilers, om te achterhalen of een bepaalde actie per ongeluk fout is, of toch juist is. Dan gaat het zowel over automatische conversies van types waar je precisie verliest, en expliciete casts waar je arbitraire types in elkaar kan omzetten. Dit zijn krachtige tools als je ze correct gebruikt, maar zo makkelijk om fouten tegen te maken. Beschouw bijvoorbeeld het volgende voorbeeld in C/C++:

```
void f(uint64_t acht_bytes, void const* buffer) {
  int value = acht_bytes;
  int* typed_buffer = (int*) buffer;

  value += *typed_buffer;
  *typed_buffer = value;
}
```

Deze zal compileren met zowel je C als C++-compiler. Maar daarin staan een aantal op zijn minst dubieuze praktijken, waren die wel de bedoeling? Bijvoorbeeld, op onze 64-bit AArch compiler is een int 32 bit groot. Die eerste regel zal bij de initialisatie van `value` een impliciete conversie doen van een 64-bit `uint64_t` naar een 32-bit `int`. Bij de tweede lijn casten we het type niet enkel van `void*` naar `int*`, bovendien is de `const` weg. Is dat een vergetelheid van de programmeur, of was dat de bedoeling? Een beetje later schrijven we dan ook naar die buffer, maar misschien was de programmeur dan al vergeten dat die buffer eigenlijk `const` was... Dit soort problemen kan ook gaandeweg optreden bij het refactoren of aanpassen van code, wie weet was `acht_bytes` oorspronkelijk een `uint32_t` en is dat type later aangepast omdat dat te klein bleek te zijn.

In C++ kan dat allemaal een pak explicieter:

```
void f(uint64_t acht_bytes, void const* buffer) {
  int value_ { acht_bytes }; // Geeft expliciete warning dat dit een downcast is!
  int value { static_cast<int>( acht_bytes ) }; // Geeft duidelijk onze bedoeling aan
  int* typed_buffer_ { static_cast<int*>( buffer ) }; // Error: const is weg!
  int* typed_buffer { const_cast<int*>( static_cast<const int*>( buffer ) ) }; // Ok!
  // ...
}
```

Hier geeft de compiler duidelijker en explicietere warnings of errors als we iets gevaarlijk doen, en geven we als programmeur duidelijk aan dat het echt wel onze bedoeling was om die `const` daar weg te halen. Dit laat ons ook toe om in een oogopslag te zien waar de (gevaarlijke) casts zich bevinden in onze code.

Zeker op het niveau van een besturingssysteemcode moeten we alle hulpmiddelen grijpen die ons helpen bij het schrijven van correcte, bugvrije code. Want bugs zijn een pak moeilijker op te sporen... Vandaar dus dat we de C++-compiler hebben geconfigureerd opdat deze alvast geen C-stijl casts meer zal toelaten.

Gerelateerd hieraan is het feit dat we graag een expliciet onderscheid willen maken tussen variabelen/-terugkeerwaarden die wijzen naar een fysiek adres, en die die wijzen naar een virtueel adres. We zouden C-gewijze een of ander type kunnen `typedef`fen, maar het probleem is dat dit onderliggend hetzelfde type blijft, en we dus impliciet (en per ongeluk) van het ene type naar het andere type kunnen casten. Er zijn een aantal mogelijke oplossingen hiervoor, in dit geval hebben we er voor geopteerd om het verschil zeer expliciet te maken door voor beiden een apart type te definiëren, `phys_addr` en `virt_addr`. Je kan deze via `to_underlying_cast` omzetten naar een `uint64_t` om er bitmanipulaties op te kunnen doen, en met `pa_to_va` kan je een virtueel adres bekomen dat wijst naar het opgegeven fysiek adres.

Tot slot, voor degenen die er meer over zouden willen opzoeken/leren, in onze C++ code maken we dus gebruik van een aantal nuttige features zoals expliciete C++ casts, braced initialisation, `auto` als type specifier, scoped enums, opaque enum declarations, `constexpr`, `nullptr`, en binary literals[1].

## 3.5 Opgave 1: Een identische mapping

Nu weten we eindelijk genoeg om met de eigenlijke opgave van dit labo te starten.

Onze code doet eigenlijk al vanalles voor jou: het overschakelen naar EL1, er voor zorgen dat onze quadcore-processor niet 4 keer dezelfde code door elkaar uitvoert maar dat er slechts één core onze code uitvoert, het registreren van exception handlers, het opzetten van communicatie via de seriële poort, *en het instellen van de parameters voor het gebruik van virtueel geheugen.* Het enige wat nog niet gebeurd is, en wat je met andere woorden zelf zal moeten doen, is het invullen van de tabellen voor virtueel geheugen, waarna je onze code kan oproepen die virtueel geheugen activeert.

Zoals in de inleiding reeds gezegd werd, gaan we de code deze week uiteindelijk aanpassen zodat we NULL pointer dereferences kunnen opvangen in onze code. We gaan dus echter klein beginnen met een identische mapping. Om het wat beperkt te maken, gaan we in deze eerste opgave beginnen met enkel een mapping te voorzien voor de code en data van onze kernel enerzijds, en het device memory anderzijds.

We gaan dit doen door de code in `mm.cc` aan te passen.

**Vul de code in voor de functies `block`, en `lower_attributes`.** (De `upper_attributes` mogen 0 zijn in deze labo's.)

Als je debugoutput wil printen naar de seriële uitvoer kan je `printk` gebruiken, wat een kernel-side implementatie is van `printf`. Je kan die dus ook een format string meegeven die de extra argumenten dan mooi zal formateren en doorsturen naar je seriële console.

Eens dat gedaan is, kunnen we beginnen aan de `init_mm` functie. De resultaatwaarde van `lower_attributes` moet als argument aan de `block`-functie meegegeven worden. Die `block`-functie geeft een (64-bit) descriptorwaarde terug die een *geldig* block-entry vertegenwoordigt die een blok virtueel geheugen zal vertalen naar het fysieke adres dat opgegeven wordt als parameter. Kijk goed in de manual welke bits op welke posities aanwezig moeten zijn! De posities van de bits kan je helaas niet duidelijk zien in `armv8_a_address translation--annotated.pdf`, maar dit **staat expliciet in `armv8_arm--annotated.pdf` onder Figure D5-15 op pagina D5-2566. De lower attributes vind je op pagina's D5-2572 tot en met D5-2574 en D5-2581.** Ik heb hier ook duidelijk aangegeven welke bits een vaste waarde moeten hebben die wij je opgeven, en over welke bits je zal moeten nadenken hoe je ze instelt (al dan niet afhankelijk van de argumenten van de functie.)

Als je deze of de volgende stukken code later zou willen debuggen, is het handig om te weten dat in het bestand `kernel.list` een overzicht zit van de assembly-code van de hele kernel, inclusief de finale geheugenadressen van zowel de functies als van globale variabelen.

[1] Een andere nuttige moderne C++ constructie die we in onze codebase toevallig niet gebruiken maar die zeker nuttig om te kennen is, is de range-based `for` waarbij je makkelijk kan itereren over elementen uit een array: `for(auto element: collection) { use(element); }`

De functies `kalloc_single_frame` en `kalloc_single_page` kan je gebruiken als een soort kernel-malloc. De functie `kalloc_single_frame` zal een pointer naar een vrij frame (dus fysiek geheugen) teruggeven. (Dit geheugen wordt voor de netheid alvast bij het voor het teruggeven vol met 0-bytes geinitialiseerd.) De functie `kalloc_single_page` zal een pointer naar een vrije pagina (virtueel geheugen) teruggeven. De functie `kalloc_single_page` zal dus, om een nieuwe virtuele geheugenpagina te alloceren, achter de schermen via `kalloc_single_frame` eerst een frame fysiek geheugen alloceren, en zal er dan voor moet zorgen dat het gealloceerde frame dan ergens in de virtuele adresruimte gemapt wordt, alvorens die pointer terug te geven. In dit geval zijn beide implementaties super-triviaal (check gerust eens de implementaties!): frames worden gewoon achter elkaar in het fysieke geheugen gezet, zonder de mogelijkheid om ze ooit terug vrij te geven (dat wordt ook wel een *bump-allocator* genoemd), en de mapping van de net gealloceerde frame naar een pagina gebeurt door impliciet gebruik te maken van de identische mapping tussen fysieke en virtuele adressen die we gaan opzetten. (Als je dus ooit verdere features zou willen toevoegen, zal een betere implementatie van deze functies allicht nodig zijn.)

Nu, zoals je uit de sectie over C++ weet, gebruiken we opake types om virtuele en fysieke adressen voor te stellen. De bovenvermelde functies hebben dan ook als return type respectievelijk `phys_addr` en `virt_addr`. Je kan die uiteraard zelf casten naar het juiste type pointer (denk eens na over wanneer je dat mag doen/wanneer je dan die kan gebruiken/waarom...), maar wij hebben voor jullie al de volgende hulpfunctie voorzien: `kalloc_single_frame_as_array_of_type<T>()`. Deze functie zal een frame alloceren, en dit achter de schermen omzetten naar een pointer van het type `T*`, waarbij `T` het template argument is[2].

Die gealloceerde frames kan je gebruiken om je paginatabellen in op te slaan. Het is dus het makkelijkste om zo'n frame te alloceren met `kalloc_single_frame_as_array_of_type<uint64_t>()`, zodat je die meteen kan gebruiken als array van het juiste type.

**Schrijf de code om de paginatabel aan te maken. Deze paginatabel moet dan de volgende data bevatten:**

- Een block dat de adressen 0-1GiB (virtueel) vertaalt naar adressen 0-1GiB (fysiek), voor de code en data van de kernel.
- Een block dat de adressen 3-4GiB (virtueel) vertaalt naar adressen 3-4GiB (fysiek), voor de devices.
- De andere entries zijn invalid. (Denk eens na waarom je dit wil. Denk dan eens na of je hier veel werk voor zou moeten doen: waarom (niet)?)

Enkele opmerkingen hierbij:

- Wij hebben al 2 `MAIR_EL1` entries ingevuld: index 0 kan je gebruiken voor device memory, index 1 kan je gebruiken voor gewone code.
- We willen dat we de geheugenlocaties zowel kunnen lezen als schrijven, stel dus in de lower attributes de AP-bits correct in.
- Hierbij is echter een **belangrijke opmerking!** Op pagina D5-2584 van de Arm manual zie je terloops vermeld staan dat '*For a translation regime that applies to EL0 and a higher Exception level, if the value of the AP[2:1] bits is 0b01, permitting write access from EL0, then the PXN field is treated as if it has the value 1, regardless of its actual value.*' Dit klinkt misschien wat irrelevant, is echter zéér belangrijk voor ons! PXN staat immers voor *Privileged Execute-Never*, met andere woorden: als deze vlag aanstaat voor een geheugenregio, mag de code in die geheugenregio nooit op een hoog privilegeniveau (zoals EL1) uitgevoerd worden. Wat deze quote dus wil zeggen, is dat als je deze regio's schrijfbaar zou maken vanuit EL0, dat de processor dan *impliciet* zal doen alsof de code helemaal niét uitvoerbaar is (vanuit EL1), ook al configureer je de PXN bit expliciet anders (zoals wij dus doen door de upper attributes op 0 te zetten)! Aangezien onze kernelcode in EL1 draait is het dus een groot probleem als onze pagina schrijfbaar zou zijn door code op EL0! (Dan zal onze code immers niet meer uitvoeren.) Dus je zal er voor moeten zorgen dat

[2]Merk op dat wat we hier *eigenlijk* als return type willen een *array van Ts* is met de expliciete lengte `page_size/sizeof(uint64_t)`, in plaats van een simpele pointer naar `T`. Eens onze compiler ondersteuning zou hebben voor C++20 zouden we een `std::span` kunnen gebruiken als return type om die informatie netjes terug te geven naar de oproepende functie.

je bij het instellen van de permissies de juiste permissie-bits instelt. Een volledig overzicht van wanneer geheugen leesbaar/schrijfbaar/uitvoerbaar is door de verschillende ELs kan je vinden op *Table D5-33 op pagina D5-2586* van de ARM Reference Manual.

- Vergeet de AF-flag niet :-)

Roep tot slot eerst de functie `switch_page_tables` op met je hoogste-niveau paginatabel, en roep dan de functie `enable_mmu` op.

Als het niet crasht is dat een goed teken dat het werkt :-)

**Dien deze code apart in als `mm_vraag1.cc`**

## 3.6 Opgave 2: De 0-pagina hermappen

In opgave 1 had je dus een block mapping die virtueel adres 0 vertaalt naar fysiek adres 0. Je gaat de code van opgave nu uitbreiden als volgt:

- Zorg er voor de **pagina waartoe virtueel adres 0 behoort**, vertaald wordt naar een ***invalid page*.**
- Zorg er voor de **fysieke pagina op adres 0** nog altijd **in de virtuele adresruimte beschikbaar is op adres 15GiB.**
- Hiertoe zal je ook de functies `page` **en** `table` **moeten invullen en gebruiken.**

Hoe een pagina-entry er uit ziet, **staat expliciet in `armv8_arm--annotated.pdf` onder Figure D5-17 op pagina D5-2569.**

Of dit alles werkt, kan je makkelijk als volgt testen:

- **Voor het inschakelen van de MMU, print je de waarde op adres 0.**
- **Na het inschakelen van de MMU, print je de waarde op adres 15GiB.**
- **Na het printen van de waarde op adres 15GiB, print je de waarde op adres 0.**

Hierbij zal je moeten gebruik maken van de `read32`-functie die een 32-bit waarde uit het geheugen zal lezen van het gegeven adres. (Er is reeds code voorzien die bij een synchrone exceptie op EL1 de waarden van `ESR_EL1`, `FAR_EL1` en `ELR_EL1` schrijft.)

(Een oproep naar `read32(p)` is functioneel equivalent aan `*(uint32_t*) p`. Echter, als je daar de waarde 0 als zou gebruiken, gaat de compiler beslissen dat dat die code *uiteraard* incorrect is, want je probeert een `nullptr` te dereferencen, en zal de compiler dus code genereren die *altijd* een exception zal opwerken, zonder zelfs nog maar te proberen om die pointer te dereferencen. Om dat te voorkomen dat de compiler onze code als foutief aanziet, hebben we dus de `read32` apart in assembly geschreven.)

Denk eerst na wat je verwacht te zien bij elk van die stappen, en vergelijk dat dan met de effectieve output.

**Dien deze code (inclusief de hierboven beschreven 3 tests, die allen het juiste gedrag moeten vertonen!) in als `mm_vraag2.cc`**

# Hoofdstuk 4

# Labo 2: Code afschermen door user-level code en system-level code te splitsen

In het vorige labo hebben we code geschreven die volledig in EL1 draait, en toegang heeft tot het de volledige geheugenruimte van devices. Dat is natuurlijk niet echt ideaal, in de meeste besturingssystemen verwacht je eigenlijk een sterke isolatie tussen de code die toegang heeft tot device memory en andere gepriviligeerde operaties kan afhandelen enerzijds, en de applicatiecode die dat allemaal niet kan anderzijds. Daartussen is dan een interface gedefinieerd waarmee de applicatiecode aan het besturingssysteem kan vragen om bepaalde gepriviligeerde acties te ondernemen. In dit labo gaan we daarom dus applicatiecode en kernel code afscheiden van elkaar door ze respectievelijk in EL0 en EL1 uit te voeren, en een simpele system call interface te gebruiken. We gaan onze applicatie ook (optioneel) iets op je scherm laten tekenen via deze system call interface.

## 4.1 System calls en exceptions in AArch64

Zoals je uit het vorige labo al gemerkt hebt, kunnen er exceptions optreden in de code, die dan afgehandeld worden door speciaal daarvoor geregistreerde code. De code die opgeroepen wordt voor de verschillende types exceptions die voor EL1 relevant zijn, staan in een tabel waarvan een pointer in `VBAR_EL1` wordt bewaard. In deze tabel wordt in de eerste plaats al een onderscheid gemaakt tussen exceptions die veroorzaakt worden door code in EL0, en exceptions in EL1. Bovendien is er een verschil tussen synchrone excepties (zoals door een ongeldige geheugentoegang zoals we die zagen in het vorige labo), excepties veroorzaakt door externe interrupts (IRQs), ... Wij hebben al de juiste functies in de juiste plaatsen in de tabel ingevuld.

Als de CPU door een exception van exception level verandert, wil je natuurlijk niet dat je registers overschreven worden. Je zou die registers dus misschien zomaar op de stack willen pushen. Je kan er echter niet altijd zomaar die push-instructies uitvoeren: wat als de stack pointer van EL0 zou wijzen naar een ongeldig adres (of erger, naar kernel code)? Dan zouden we zomaar daarheen beginnen schrijven. Daarom heeft de CPU een aparte stack pointer per exception level. Je kan met een configuratie-bit aan/uit te zetten in het *Saved Program Status Register* `SPSR_EL1` aangeven of een exception handler de stack pointer van EL0 mag blijven gebruiken, of dat er geswitcht moet worden naar de stack pointer van EL1. (Dit register bevat de bewaarde statusbits—zero flag, cary flag, ...— van voor een exceptie, maar bevat onder andere ook of een exception van EL0 of van EL1 kwam.) De mode waarbij er geswitcht wordt naar de EL1 stack pointer wordt EL1h genoemd; die waar niet geswitcht wordt van stack wordt EL1t genoemd; ook die informatie wordt in `SPSR_EL1` bewaard. (Dit staat in meer detail in de Arm Architecture Reference Manual op pagina's D1-2277, D1-2278, en C5-422.)

Een systeemoproep doe je met behulp van de `svc`-instructie (*SuperVisor Call*). Deze heeft een integer-operand dat we voor deze labo's niet zullen gebruiken, en dus als 0 zullen invullen. Wat bovendien belangrijk is, is dat system calls ook als een synchrone exceptie behandeld worden! Zo'n system call zal er dus voor zorgen dat er een specifieke exception class ingevuld wordt in het exception syndrome register. Het exception link register zal worden ingesteld op het adres van de instructie nét ná de `svc`-instructie, zodat, eens de system call afgehandeld is, de EL1-code een `eret`-instructie kan uitvoeren, waarna de CPU de code volgende op de system call verder kan uitvoeren op EL0.

Wij hebben gekozen voor EL1h. De functie `handler_synchronous_after_storing_registers` in `vectors.S` is verantwoordelijk voor het bewaren van alle registers in een speciaal daarvoor voorzien stukje van het geheugen dat we tijdelijk instellen als `SP_EL1`. Eens we de EL0-registers hierin bewaard hebben, switchen we de EL1-stack terug naar een pointer die we effectief kunnen gebruiken als stack. De code om terug te keren van een system call bevindt zich in `return_from_syscall`, en zal de waarde in het `x0`-register overschrijven met de terugkeerwaarde van de system call, alvorens `SP_EL1` terug in te stellen op het stukje geheugen om registers te bewaren en dan de andere registers te herstellen, en zal tenslotte terug te keren naar EL0 via `eret`.

Het grootste deel van de low-level code is dus al geschreven voor exceptions af te handelen. We moeten echter nog het onderscheid maken tussen een system call, en andere exceptions.

**Pas dus `handler_synchronous` in `exceptions.cc` aan zodat, indien het `ESR_EL1`-register een exception code bevat die hoort bij een system call, de `handle_syscall`-functie opgeroepen wordt met de correcte argumenten.** (Je zal hiervoor dus de juiste exception class moeten opzoeken in de manual.)

## 4.2 Code uitvoeren op een lager privilegeniveau

In de attributes van het virtueel geheugen kunnen we aangeven of pagina's leesbaar/schrijfbaar zijn door EL1 en/of EL0. We willen dus dat de meeste pagina's enkel door de systeemcode op EL1 toegankelijk is, en dat enkel de pagina's van onze (test)applicatie door EL0 toegankelijk is.

**Verifieer dan ook eerst dat de attributes die je gebruikt hebt in labo 1 voor de mappings van de kernel-code, kernel-data, en het device memory in `init_mm` enkel lezen en schrijven vanuit EL1 toestaat! (En pas dat desnoods aan.)**

In een normaal besturingssysteem gaan we onze applicaties van een bestandssysteem inlezen, de headers van die applicatie-binary correct parsen om alles goed te kunnen initialiseren, om tot slot de uitvoering op EL0 te starten bij het entry point van deze applicatie. In ons zeer simpel besturingssysteem hebben we geeneens een bestandssysteem, laat staan dat we een device driver hebben die we kunnen gebruiken om code van disk te lezen! We gaan het dan ook een pak simpeler houden, en onze applicatiecode gewoon deel laten uitmaken van onze `kernel.bin` image die we via de seriële poort inladen op de Raspberry Pi.

Al onze applicatiecode zit in `user_process.cc`. Via wat vuil gepruts in het linker-script `link_os.ld` dat we aan de linker meegeven om onze finale `kernel.bin` te maken, zal alle code en data op een aparte plek gezet worden in onze image. In het bijzonder zal de code starten op het adres van `__begin_user_shared`. Dit zal meteen gevolgd worden door de data die onze applicatiecode nodig heeft. Het eindadres zal zich bevinden op `__end_user_unshared`[1].

Om onze applicatiecode nu uit te kunnen voeren, moeten we drie dingen regelen: bepaalde pagina's markeren als leesbaar/schrijfbaar voor EL0, onze applicatie klaar zetten om uitgevoerd te worden door EL0, en dan de effectieve switch doen.

*Pagina's leesbaar/schrijfbaar maken voor EL0.* Hiertoe gaan we *bestaande* page table entries aanpassen om ze accessible te maken voor EL0. Om dit te doen, moeten we dus wat code in `mm.cc` aanvullen. **Schrijf eerst code om een page table entry op te zoeken: `get_level3_page_table_entry_for`, wat als argument het adres van een pagina krijgt, en als terugkeerwaarde een pointer naar de bijhorende 64-bit page table entry heeft.** Je zal dus zelf een page table walk moeten doen startende van `TTBR0`, en dan de verschillende tussenliggende tabellen te indexeren aan de hand van de bits van het adres. Je mag er hierbij van uitgaan dat er inderdaad een bijhorende level 3 entry te vinden is. Je kan verifiëren dat deze code werkt door de tussenliggende waarden van de tabellen uit te printen, en in `init_mm` dezelfde pointers uit te printen; net als de waarde van de finale page table entry.

**Vervolgens pas je `make_userspace` aan zodat de aangeleverde pagina inderdaad leesbaar/-schrijfbaar wordt op EL0.** De inline assembly die op het einde van deze functie staat, zorgt er voor dat de TLB geflusht wordt, en zorgt er voor de processor geen verdere code gaat uitvoeren tot die TLB-veranderingen correct doorgevoerd zijn.

[1] Het onderscheid tussen shared en unshared kan van pas komen als je ooit meerdere processen zou willen hebben die dezelfde read-only code-pagina's delen, maar afzonderlijke private data-pagina's hebben.

Ter herinnering: in de ARM Architecture Reference Manual heb je al gelezen welke bits je moet aanzetten hiervoor. Een overzicht van alle combinaties van permission bits en hun effect vind je op *Table D5-33 op pagina D5-2586.*

*De applicatie klaarzetten.* Nu gaan we die permissies inderdaad correct instellen voor onze applicatie. In de `initialize_and_go_to_our_single_userspace`-functie in `process.cc` hebben we alvast pointers voorzien die wijzen naar het begin van de applicatiecode en net voorbij het einde van de applicatie-data. **Pas de permissies aan van de pagina's die bij de applicatie horen.** Vervolgens moet onze applicatie ook een eigen stack krijgen. **Hiervoor zal je dus geheugen moeten alloceren, en dit geheugen ook de juiste permissies voor EL0 moeten geven.**

*Overschakelen naar het uitvoeren van de applicatie op EL0.* Hiervoor gaan we wat inline assembly schrijven in onze `initialize_and_go_to_our_single_userspace`-functie. **Pas dus de inline assembly-code aan zodat ze de volgende dingen doet:**

1. Stel de EL0 stackpointer `SP_EL0` in met de net-aangemaakte stack voor onze applicatie. **Opgelet!** stacks groeien nog steeds van hoge naar lage adressen, en het is belangrijk dat de stack wijst naar een adres dat een veelvoud van 16 is.

2. Stel het exception link register `ELR_EL1` in naar het entry point van onze applicatie.

3. Stel het program status register in zodanig dat de CPU weet dat onze applicatie in EL0 moet draaien (met andere woorden dat er naar EL0 moet teruggekeerd worden bij het uitvoeren van `eret`). Hierbij volstaat het om de waarde 0 te schrijven naar `SPSR_EL1`. Dit gaat uiteraard ook andere statusbits op 0 initialiseren die we hier gaan negeren. (Je kan gerust opzoeken in de Arm Architecture Reference Manual wat de betekenis is van alle bits op 0 te zetten in `SPSR_EL1` in AArch64 modus, en waarom dit er onder andere voor zorgt dat we hiermee terugkeren naar EL0, maar dat is niet zo belangrijk.)

4. Ten slotte moeten we de stack pointer van EL1 aanpassen naar het stukje geheugen dat we voorzien hebben om bij een exception, de EL0-registers naar te kopiëren.
   Herinnering: de `initialize_and_go_to_our_single_userspace`-functie voert uit in EL1, je kan dus hiervoor gewoon naar het algemene `sp`-register schrijven.

5. Nu zijn we klaar om 'terug te keren' naar onze EL0-applicatie vanuit EL1, met de `eret`-instructie.

Dit kan in 5 assembly-instructies :-) Ter herinnering, je kan waarden kopiëren van algemene registers naar een systeemregister met de `msr`-instructie. Systeemregisters kan je in deze herkennen aan hun `_EL0` of `_EL1` suffix.

We maken gebruik van de extended gcc syntax voor inline assembly. Hierbij kunnen we makkelijk verwijzen naar lokale C++-variabelen vanuit onze assembly, door achteraan de inline assembly een mapping mee te geven waar we zeggen welke variabelen we in de assembly gaan gebruiken, en hoe we daarnaar in de assembly kunnen verwijzen. (Afhankelijk van op welke positie ten opzichte van de meerdere dubbelpunten geven we mee dat die variabele gelezen en/of geschreven zal worden door onze assembly.) Door daar `"r"` bij te vermelden, geven we aan de compiler mee dat deze lokale variabelen in een register moeten zitten. Als we dan in onze inline assembly `%[naam]` gebruiken, zal de compiler dit automatisch vervangen door het algemene register waarin die variabele bewaard wordt. Je kan een aantal voorbeelden van die syntax zien in `mm.cc`[2].

## 4.3 Systeemoproepen toevoegen, afscheiding tussen privilegeniveau's controlleren

Nu is alles klaar om vanuit onze applicatie enkele system calls te doen! Hiervoor moeten we dus zowel aan de applicatiekant als aan de kernelkant een overeenkomende interface hebben aan de hand van gedeelde system call numbers. We hebben er al voor gezorgd dat in de kernel `handle_syscall` opgeroepen wordt, nu gaan we daar iets mee doen!

[2]De volledige beschrijving van die syntax kan je vinden op `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`, maar normaalgezien ga je voldoende hebben aan de voorbeelden in onze code eens te bekijken.

Aan de kernelkant moeten we kijken wat het syscallnummer is, de bijhorende code uitvoeren (gebruik makende van de argumenten die uit userspace komen als dat relevant is), en zal dan uiteindelijk met `return_from_syscall(terugkeerwaarde)` de uitvoering in EL0 verder zetten. Alle argumenten zijn van het type `uint64_t`, net als de terugkeerwaarde, zodat zowel getallen als 64-bit pointers kunnen worden doorgegeven tussen de kernel en de applicatie.

**Pas `handle_syscall` aan zodat `Syscall::print` als syscall-nummer de `printk`-functie zal oproepen met als pointer de string-pointer die uit userspace komt.** (De kernel heeft lees/-schrijfrechten op alle pagina's die door EL0 lees/schrijfbaar zijn, de pointer kan dus gewoon door `printk` gebruikt worden.)

Nu moet ook onze applicatie die syscall gebruiken natuurlijk. **Vul hiervoor de `print`-functie in `user_process.cc` aan om dit te doen.** Hiervoor kan je de `do_syscall`-functie uit `syscalls.h` gebruiken.

Merk op dat de applicatiecode geen functienamen kan definiëren die óók gedefinieerd worden in de kernelcode. Dit komt door de manier waarop onze applicatie en onze kernel samen in 1 bestand samengevoegd worden. Als je dit toch zou doen, ga je foutmeldingen krijgen over dubbele definities.

**Roep nu vanuit de `user_main` de print-functie op, en kijk of alles nu werkt!**

Let hierbij op: als je functie gedaan is, dan gaat die 'terugkeren' naar de oproepende functie. Maar nu heeft de kernel die applicatie zelf gestart, en is er na onze `user_main` niets meer om naar terug te keren. Dus ofwel laat je de `user_main` gewoon terugkeren en zal het systeem dan een exception triggeren na `user_main` (omdat die probeert code uit te voeren op zijn link register, wat we geïnitialiseerd hebben als 0), ofwel zal je `user_main` een oneindige lus laten uitvoeren.

Tot slot gaan we eens kijken of onze applicatiecode nu inderdaad niet meer aan het device memory kan. We gaan hiervoor dus gewoon proberen schrijven naar een EL1-only adres vanuit onze applicatiecode. Het leukste is om een kerneladres te 'lekken' via een system call. **Implementeer hiervoor de code van de system call `get_device_pointer`, die vanuit de kernel een device pointer zal teruggeven als resultaat van de system call.** (Zo heb je ook eens een system call geïmplementeerd die informatie teruggeeft naar de applicatie.) Een voorbeeld van een pointer die je kan lekken is `GPIO_BASE`.

**Ten slotte ga je die system call vanuit de applicatiecode oproepen, en de resulterende pointer proberen te accessen.** Wat zie je gebeuren? Klopt dit met wat je verwacht had?

**Dien de aangepaste files in: `exceptions.cc`, `user_process.cc`, `process.cc`, en `mm.cc`**

## 4.4 Tekenen op een scherm (optioneel)

Er is nog één set system calls die we kunnen implementeren gebaseerd op onze aangeleverde code, namelijk die voor te kunnen tekenen op een scherm! Wij hebben al de code voorzien die in de kernel een framebuffer opzet, en die toestaat om pixels op die framebuffer te tekenen. Je moet eerst de framebuffer initializeren (daarvoor zal je dan de `init_framebuffer`-syscall toevoegen die `framebuffer_init` functie oproept). Vervolgens kan je de `put_pixel`-syscall gebruiken om de `put_pixel`-functie op te roepen. Als je dan je Raspberry Pi boot met een HDMI-kabel die op een extern scherm aangesloten is, zou je dan daarop moeten kunnen tekenen vanuit je userspace code. Ik heb in `user_process.cc` al een `draw_lines`-functie voorzien die je kan gebruiken om dat te testen. Merk op dat de code op dit punt de vooraf vastgelegde resolutie van 640x480 aanvraagt en je dus beperkt bent tot dat om iets te tekenen. (Je kan eventueel ook nog eens de pointer naar de framebuffer lekken naar de applicatie met de `get_device_pointer`-functie om te verifiëren of ook die correct is afgeschermd van de applicatiecode.)

# Hoofdstuk 5

# Where do we go from here?

Hierbij zijn we klaar met de labo's op de Raspberry Pi. Maar laat dat je niet tegenhouden om eens na te denken over extra features! Hier zijn alvast enkele ideetjes om deze code uit te breiden, sommigen gaan heel simpel zijn, voor anderen ga je misschien wat meer (lees)werk hebben (ik heb deze uitbreidingen zelf ook nog niet geprobeerd wel, caveat emptor :-)) :

1. *W^X* (Write Xor eXecutable). Op dit punt kan je uitvoerbare code zomaar overschrijven. Dat wil zeggen dat het zou kunnen gebeuren dat er een pointer die je gebruikt om data naar te schrijven, per ongeluk wijst naar een pagina met uitvoerbare code, waarna je die uitvoerbare code zomaar overschrijft en ze dus corrupt maakt. Dat is vrij vervelend als dat gebeurt: debuggen wordt lastig omdat je code die uitgevoerd wordt niet meer overeenkomt met de code die je gecompileerd hebt. Bovendien is het ook een beveiliginsrisico bij bijvoorbeeld buffer overflows: niet alleen kunnen aanvallers code zomaar overschrijven met hun eigen code, ze kunnen ook een lokale buffer invullen met code in, en als dan een terugkeeradres (of andere functiepointer) overschreven kan worden door een aanvaller, kan die dat terugkeeradres overschreven worden met een pointer naar die zelf-ingevulde buffer. Wanneer de uitvoering van het programma dan naar die functiepointer/dat terugkeeradres gaat, wordt dus code uitgevoerd die door de aanvaller zelf geschreven werd. Door te verbieden dat geheugenpagina's tegelijkertijd beschrijfbaar en uitvoerbaar is, zal zo'n aanval niet meer mogelijk zijn. Op ARM heet deze feature *WXN* (*Write implies eXecute Never*).

2. Uit het document over virtueel geheugen op ARM weet je dat we nu enkel de onderste helft van de virtuele geheugenruimte kunnen aanspreken via `TTBR0`, en dat je om de bovenste helft aan te spreken, `TTBR1` zal moeten gebruiken. Normaalgezien wordt dan de applicatiecode in de onderste helft van het virtuele geheugen gemapt, en wordt de bovenste helft (en dus `TTBR1`) gebruikt voor de kernel. Zo kan je later ook switchen tussen verschillende applicaties die elk hun eigen page tables hebben die dan via `TTBR0` zal geselecteerd worden, terwijl al die applicaties dezelfde `TTBR1` en bijhorende page tables kunnen blijven gebruiken voor de kernel. (Hierbij map je dan best meteen ook de devices in de bovenste helft van de fysieke adresruimte natuurlijk.)

3. Op onze Raspberry Pi zit een quadcore CPU, en heeft dus 4 cores. Wij gebruiken echter enkel de eerste voor onze kernel. Onze bootloadercode laat op de 3 andere cores gewoon een infinite loop uitvoeren. Het kan interessant zijn om te kijken om op die andere cores ook iets nuttigs uit te voeren. (Daarvoor zal je dus zowel de kernel als de bootloadercode moeten aanpassen.)

4. Wij hebben nu een `printk`-functie die allerhande argumenten kan formateren en printen naar de seriële poort. In userspace hebben we echter nu enkel een `print`-functie zonder argumenten. Idealiter hebben we een `printf` in userspace die al een string aanmaakt en die dan print. Dus je zou je kunnen baseren op de `printk`-functie om een gelijkaardige userspace-functionaliteit te maken.

5. Je kan eens proberen kijken naar externe interrupts en daar iets mee te doen.

6. Nu printen we de output van onze printk gewoon naar de seriële poort, maar nu we een framebuffer hebben, kunnen we in principe die ook naar het scherm laten schrijven. Wat je daar typisch hebt, is dat je een grote tabel hebt die voor elke letter de grafische voorstelling ervan heeft, waarna je

dan je output letter per letter naar de framebuffer kan kopiëren op de juiste locatie (jij wordt dan natuurlijk ook verantwoordelijk voor het bijhouden van welke letter je waar op het scherm tekent).

7. Wij hebben al een framebuffer-implementatie voorzien. Misschien kan je die zelf eens opnieuw maken (en uitbreiden zodat de resolutie ook wat hoger is). De makkelijkste (en voor zover ik weet enige deftig publiek 'gedocumenteerde') manier om er eentje op te zetten is om te vragen aan de firmware op de GPU om zo'n framebuffer op te zetten door daarmee te communiceren via diens *mailbox* interface. De documentatie voor hoe je via mailboxes communiceert vind je op de firmware-wiki van Raspberry Pi: `https://github.com/raspberrypi/firmware/wiki/Accessing-mailboxes`. (`MAIL_BASE` in de documentatie is daar relatief ten opzichte van het begin van de geheugenruimte voor peripherals, dus die bevindt zich in de praktijk op legacy-adres `0x7e00b880`.) Eens je weet hoe je kan communiceren met de firmware op conceptueel niveau, moet je natuurlijk weten welke specifieke berichten je kan sturen naar de firmware. Een (helaas onvolledig, maar voldoende voor deze doeleinden) overzicht van de berichten die je kan sturen naar de firmware vind je op `https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface`.

8. Je kan de bump-allocator vervangen door een iets meer gesofisticeerde allocator, die ook pagina's terug kan vrijgeven.

9. Onze userspace applicatie moet nu eindigen in een oneindige lus, als je niet wil dat je een exception triggert. Je kan er voor zorgen dat (zoals bij echte operating systems) je eigenlijk extra functionaltieit toevoegt voor/na de main-functie, die na het terugkeren van de main-functie een (nieuw toe te voegen) `exit` system call zal oproepen.

10. Mooooooooooooooo, verzin zelf iets, the sky is the limit! :-)