Cursus besturingssystemen
Handleiding voor de practica

Versie van 19 oktober 2020

Vakgroep Elektronica en Informatiesystemen
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2020–2021

UNIVERSITEIT GENT
ELIS

Inhoudsopgave

Hoofdstuk 1

Inleiding

1.1 Doelstelling

De practica hebben als doelstelling de student iets bij te brengen over besturingssystemen, de manier waarop ze geïmplementeerd worden en de manier waarop ze door applicaties gebruikt kunnen worden.

1.2 Praktische richtlijnen

De practica gaan online door op donderdagvoormiddag van 9u30 tot 11u30. De practica worden uitgevoerd in groepjes van 2 personen. Het is niet verplicht om elk practicum in dezelfde groep te zitten.

Deelname aan een practicum is steeds verplicht, Voor enkele practica zult u een antwoordenblad in deze handleiding vinden. Dit blad moet op de dag van het practicum ingediend worden. Voor de andere practica zult u uw code in de loop van de dag moeten emailen. Alhoewel alle ingediende oplossingen verbeterd zullen worden zijn er slechts drie practica gequoteerd.

BELANGRIJK: het is de bedoeling dat de practicumhandleiding bestudeerd wordt ter voorbereiding van elk practicum. In principe vergt de voorbereiding evenveel tijd als het eigenlijke practicum.

Wie vragen heeft kan die stellen op het forum op Ufora.

1.3 De practica

Er zijn in totaal vijf practica voorzien:

- Voor twee practica zullen we gebruik maken van de Bochs-simulator. Deze simulator hebben jullie vorig jaar reeds gebruikt voor de practica van de cursus *Computerarchitectuur*. Bovendien is niet enkel de simulator maar ook de code waarvan we vertrekken ongeveer dezelfde als deze die gebruikt werd voor *Computerarchitectuur*. **Wie vorig jaar moeilijkheden had bij het gebruik van BOCHS en/of de gebruikte code frist zijn geheugen best eens op!**

- De drie resterende practica worden onder Linux uitgevoerd, en dit m.b.v. de *QEMU*-virtualisatie-omgeving.

Een overzichtje:

practicum	datum	naam	OS	gequoteerd
1	22/10	Paginering	Bochs	ja
2	29/10	Protectie en segmentatie	Bochs	nee
3	5/11	Scheduler	Linux	ja
4	12/11	Device driver	Linux	ja
5	19/11	Bestandssysteem	Linux	nee

Hoofdstuk 2

Practicum 1: Paginering (Bochs)

In dit practicum zullen we van naderbij bekijken hoe virtueel geheugen en paginering in de praktijk gebruikt kunnen worden. We zullen dit doen door op moderne Intel 80x86 processors de problemen veroorzaakt door de nog steeds aanwezige compatibiliteitsvoorzieningen met de 8086 te omzeilen.

2.1 Geheugenoverzicht van de 8086

De 8086 is een 16 bits processor. Deze processor gebruikt een gesegmenteerd geheugenmodel om voorbij de 16 bits limiet van 64 KiB adresseerbaar geheugen te geraken. Elk segment is gealigneerd op 16 bytes en is 64KiB groot.

Zowel de segmentregisters als de adresregisters zijn 16 bits breed. Een adres is steeds van de vorm `segment:verschuiving`. Aangezien elk segment slechts 16 bytes verder begint dan het vorige, is het mogelijk om de meeste elementen van het fysieke geheugen op meerdere manieren te adresseren. Adres `0:160` = `1:144` = `2:128` = ... = `10:0` = lineair adres 160.

Het hoogst adresseerbare adres op deze manier is in theorie `0xffff:0xffff` = `0xffff` * `16` + `0xffff` = `0x10FFEF`. De voorstelling van dit adres vereist echter 21 bits en de 8086 had slechts 20 adreslijnen. Het gevolg was dat dit adres door de processor als adres `0xFFEF` aanzien werd (de hoogste bit werd gewoon weggegooid).

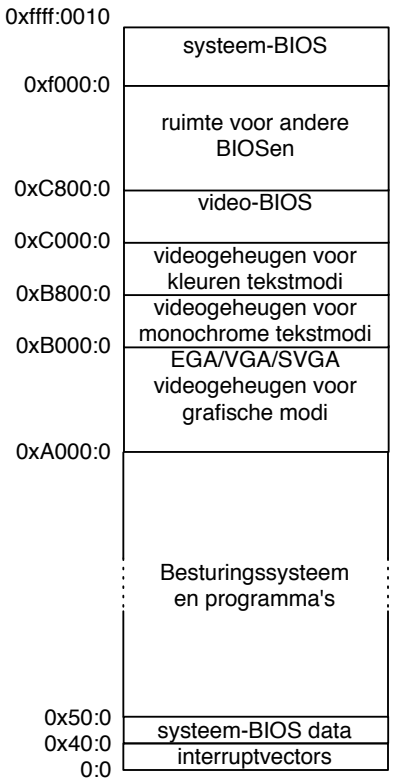
Met deze 20 adreslijnen is het mogelijk om 1MiB geheugen te adresseren. Deze adresruimte moet gedeeld worden door enerzijds de programma's die men wil uitvoeren, en anderzijds op zijn minst de systeem-BIOS, de BIOS van de videokaart en het videogeheugen. Extra randapparaten of insteekkaarten pikken hier mogelijk nog een graantje van mee.

De adresruimte werd hiervoor in verschillende delen verdeeld, zoals in figuur 2.1 aangegeven. Enkel de onderste 640 KiB werd voor systeem- en programmageheugen voorbehouden. De overige 384 KiB werd gereserveerd voor BIOSen en randapparaten.

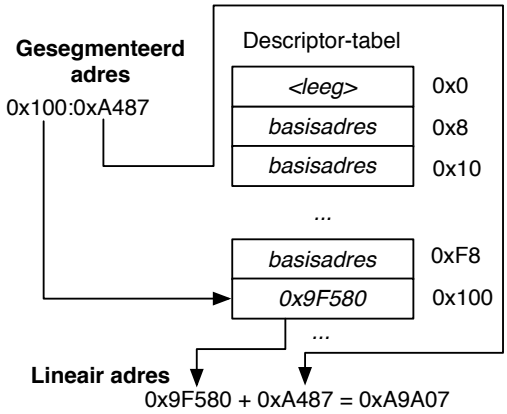
Vermits de verschillende BIOSen in ROM gebakken zijn, kan je die gebieden niet overschrijven. Verder heb je een aantal gaten in de adresruimte die al dan niet in gebruik kunnen zijn. Tot slot hangt de plaats waar het videogeheugen zich bevindt af van de huidige actieve videomode.

2.2 Bescherming

Om compatibiliteit met de 8086 te verzekeren, start de 80286 standaard in zgn. *real mode* op, waarin hij zich gewoon voordoet als een snellere 8086 (op een paar uitzonderingen na). Men blijft dus zitten met eerst 640 KiB geheugen, vervolgens 384 KiB waarin men geen programma of systeemdata mag plaatsen, en ten slotte de rest van het geheugen.



Figuur 2.1: Organisatie van de geheugenruimte van de 8086



Figuur 2.2: Descriptors in protected mode

Vermits de 80286 nog steeds met 16 bits registers werkt, is hij eveneens tot adres `0x10FFEF` beperkt (waarbij in dit geval de 21ste bit niet weggegooid wordt). Om deze en andere beperkingen op te lossen, werd in de 80286 de *protected mode* functionaliteit toegevoegd.

Wanneer men naar deze mode gaat, worden een aantal veranderingen in het geheugenmodel doorgevoerd. De belangrijkste zijn:

- De segmentregisters bevatten in deze mode zgn. *selectors*. Deze selectors worden gebruikt om *descriptortabellen* te indexeren. De elementen van deze tabellen zijn 64 bits waarden. Ze bevatten o.a. een 24 bit basisadres, dat het begin van het segment aanduidt, en een 16 bits limiet, die zegt hoe lang dit segment is.

  Op deze manier bekomt men segmentering met variabele beginadressen en segmentgrootte, zoals geïllustreerd in figuur 2.2. De 24 bits laten toe tot 16 MiB geheugen te adresseren.

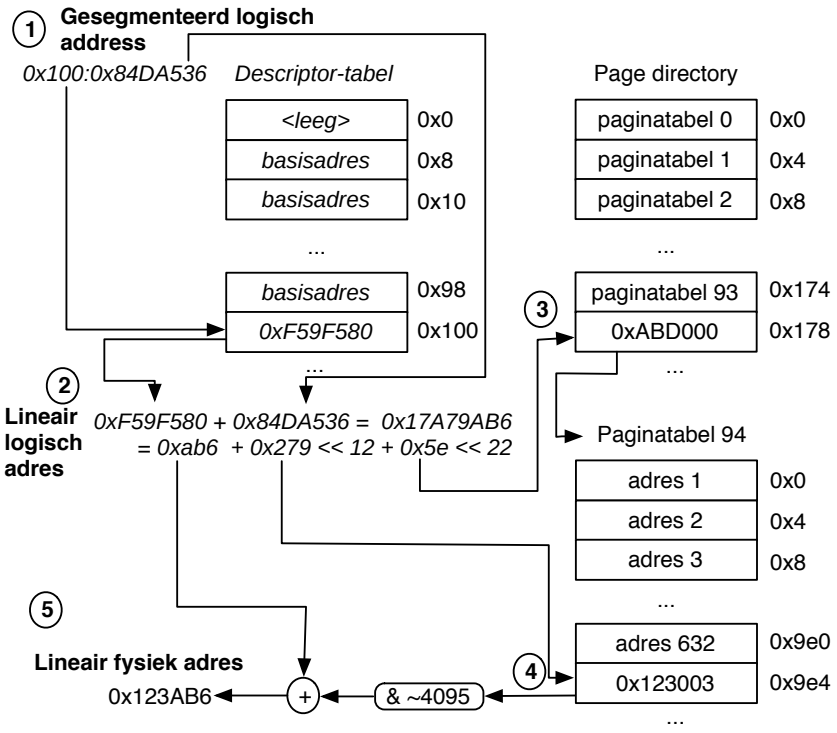
  De adressen van deze descriptortabellen moet je in speciale registers van de processor laden, en je kan ze dus eender waar in het geheugen plaatsen. Er zijn 3 dergelijke tabellen in het systeem: de globale (die door alle programma's gebruikt kan worden), de lokale (per programma) en de interrupttabel.

  De waarde van een selector wordt gewoon bij het beginadres van een descriptortabel geteld om het adres van de relevante descriptor te bekomen. Deze selectors moeten dus steeds een veelvoud van 8 zijn, aangezien elke descriptor 8 bytes groot is. De laagste 3 bits van een selector worden daarom voor andere doeleinden gebruikt: 1 bit bepaalt of de selector gebruikt moet worden om de lokale of de globale descriptortabel te indexeren, terwijl de overige twee het gevraagde privilegieniveau aanduiden (altijd 0 in ons geval).

- De mogelijkheid om beschermingen aan segmenten toe te kennen. Men kan zo b.v. de eigenschappen leesbaar, schrijfbaar en uitvoerbaar aanpassen per segment. Deze eigenschappen worden eveneens in de descriptor opgeslagen.

2.3 Paginering en logische adressering

De 80386 t.e.m. de Core i9 van vandaag starten nog steeds in dezelfde *real mode* op als de 80286, omwille van compatibiliteitsredenen. De 80386 was echter een grote stap voorwaarts: dit was de eerste processor in deze reeks die een 32 bits mode had en die paginering, en bijgevolg virtueel geheugen, ondersteunde.



Figuur 2.3: Paginering op de 80386 en hoger.

De segmentregisters blijven 16 bits in deze implementaties, maar de adresregisters werden vergroot tot 32 bits. Door tot nu toe ongebruikte bits in de descriptors te gebruiken, kon men de basisadressen vergroten tot 32 bits en de limieten tot 20 bits. Een extra vlag in de descriptors laat toe om te specificeren dat een limiet als veelvoud van 4 KiB (= 12 bits) opgegeven is, waardoor het mogelijk wordt om met 1 descriptor de hele 32 bits adresruimte te bestrijken (indien dit gewenst is).

Paginering kan enkel in *protected mode* geactiveerd worden, en werkt met 4 KiB pagina's op de 80x86 architectuur. Nadat paginering ingeschakeld is, blijft het systeem van selectors en descriptors gewoon verder werken, maar nu met logische i.p.v. met fysieke adressen.

Wanneer je nadien dus een geheugenlocatie benadert, wordt eerst via de selector de juiste descriptortabel geïndexeerd (en de speciale registers met de adressen van de descriptortabellen moeten nu de logische adressen van deze tabellen bevatten, i.p.v. de fysieke!). Vervolgens wordt m.b.v. de gegevens in de descriptor bepaald wat het lineaire logische adres is, en ten slotte wordt dit adres via het pagineringsmechanisme vertaald naar het uiteindelijke fysieke adres. Dit wordt grafisch weergegeven in figuur 2.3.

Zoals uit de figuur blijkt, gebruikt de 80x86-architectuur een 2-niveau pagineringsmechanisme. Het eerste niveau wordt de *page directory* genoemd, en bevat tot 1024 (32bits) wijzers naar paginatabellen. Elk van deze paginatabellen bevat weer 1024 elementen, maar dan met vertalingen van logische naar fysieke adressen.

Het adres van de *page directory* moet je in het controleregister `cr3` schrijven voordat je de paginering activeert. Dit is uiteraard een fysiek adres, en dat moet zo blijven nadat paginering geactiveerd is (je hebt die waarde immers nodig om logische in fysieke adressen te vertalen). De adressen van de verschillende paginatabellen in de *page directory* moeten om dezelfde reden ook steeds fysieke adressen zijn.

Telkens je iets verandert in de *page directory* of in een paginatabel, moet je dat adres opnieuw in cr3 schrijven om de *translation look-aside buffer* van de processor leeg te maken. Dat is een cache

in de processor die een aantal van de laatst gebruikte elementen van de paginatabellen bijhoudt, zodat niet voor elke geheugentoegang 2 extra geheugentoegangen nodig zijn om de gegevens in de paginatabellen op te zoeken.

De *page directory* wordt geïndexeerd met de hoogste 10 bits van het logisch adres, en de paginatabel waar dat element van de directory naar wijst met de volgende 10 bits. De laagste 12bits van het adres duiden aan welke byte in de fysieke pagina opgehaald moet worden.

Een pagina begint steeds op een veelvoud van 4 KiB, wat wil zeggen dat de 12 laagste bits van de waarden in de paginatabellen steeds 0 zouden moeten zijn. In de praktijk worden deze bits gebruikt om extra informatie over de pagina's op te slagen, maar worden ze bij gebruik voor adresberekeningen genegeerd. Voor ons zijn enkel de laagste twee bits van belang:

- bit 0: als deze bit op 1 staat, wil dit zeggen dat dit element van de paginatabel geldig is.
- bit 1: als deze bit op 1 staat, wil dit zeggen dat deze pagina zowel lees- als schrijfbaar is. Wanneer hij op nul staat, is de pagina enkel leesbaar.

2.4 Werken met Bochs

Bochs is een simulator (gepubliceerd als vrije software) voor de 80x86-architectuur. Hij simuleert zowat alle hardware tot in de kleinste details en bevat een geïntegreerde debugger. Deze laat toe code instructie per instructie uit te voeren vanaf het starten van de computer, wat op een echt systeem onmogelijk zou zijn.

Download het bestand `Practicum_2_Bochs-Paginering.zip` van de Ufora website, en unzip het. In de directory `os_paging` vind je alle nodige bestanden. Alle code staat in het bestand `paging.asm`. Deze code wordt geassembleerd tot machinecode met het programma `nasm` (de Netwide Assembler). Het bestand is zo opgesteld dat het resultaat een geldige opstartsector voor een diskette is. Vervolgens starten we Bochs met de opdracht om dat bestand als inhoud van een virtueel diskettestation te beschouwen.

Om je code te assembleren voer je `a.bat` uit, om hem uit te voeren `r.bat` en om te debuggen `d.bat`. Een overzicht van de commando's die je kan gebruiken tijdens het debuggen zijn te vinden op `http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html`. Indien je de code aanpast en opnieuw wil assembleren moet je eerst alle lopende Bochs-sessies stoppen!

Wanneer Bochs gestart is, kan je met het commando "c" zorgen dat hij gewoon alle code moet beginnen uitvoeren. Eerst is dit de BIOS, die vervolgens onze opstartcode van de virtuele floppy zal laden op adres `0x7c00`. Daarna wordt naar dit adres gesprongen en wordt onze code uitgevoerd. Als je fouten wil opsporen, kan je best op adres `0x7c00` een breakpoint plaatsen (m.b.v. het commando "b 0x7c00") en vervolgens "c" gebruiken om in één keer tot aan dat adres verder te lopen (i.p.v. instructie per instructie de hele BIOS te gaan doorlopen).

Aan het begin van onze code is er een lus die een 15-tal keren wordt uitgevoerd. M.b.v. van het `disassemble` commando kan je zien waar die lus stopt om er na een breakpoint te plaatsen, zodat je die niet elke keer 15 maal instructie per instructie moet uitvoeren. Normaal gezien eindigt de lus steeds op adres `0x00007c48`, want de code die voor dit adres komt moet je niet aanpassen.

2.5 De betekenis van `$$` in NASM assembly code

Een assembler weet niet op welk adres de code en data uiteindelijk zal terechtkomen. De reden is dat dit afhangt van hoe het geassembleerde object nadien zal worden gebruikt. Om die reden kan je geen wiskundige berekeningen uitvoeren op symbolische adressen, zoals `KernelTaskSegment >> 16`. Aangezien we niettemin een aantal adressen moeten opsplitsen om we in de descriptors te stoppen, gebruiken we een trucje: hoewel de assembler dus het absolute adres van symbolen niet

weet, kan die wel het verschil tussen twee adressen berekenen (of meer precies: tussen twee adressen in dezelfde sectie). De reden is dat de assembler van elk stukje data (.db, .dw, .dd) en van elke instructie exact weet hoeveel bytes ze innemen. Verder weten wij, in tegenstelling tot de assembler, wel waar onze code/data geladen zal worden in het geheugen: op adres 0x7c00.

Nu, wat `$$` betreft: dit staat voor "het adres van de huidige sectie". Al onze code en data zit in één sectie, die dus zoals vermeld op adres 0x7c00 geladen wordt.

Het resultaat is dat

- de assembler kan uitdrukkingen als `KernelTaskSegment - $$` evalueren, en het resultaat is het aantal bytes vanaf het startadres van onze code/data tot aan het symbool KernelTaskSegment
- we definiëren een constante met de naam `loaded` die de waarde 0x7c00 heeft, zodat `KernelTaskSegment - $$ + loaded` gelijk is aan het effectieve adres van KernelTaskSegment tijdens de uitvoering.

Op die manier kunnen we dus het absoluut adres van een symbool berekenen en gebruiken als een constante in de assembler.

2.6 Opgaven

2.6.1 Opgave 1: Afbeelden van het videogeheugen

Aangezien het onhandig is om eerst een blok van ongeveer 640KiB te hebben, dan 384KiB waar je moet van afblijven en dan de rest van het geheugen, gebruiken moderne besturingssystemen het pagineringsmechanisme om deze artefacten uit het verleden te omzeilen. Dit practicum heeft als doel deze functionaliteit te implementeren.

De segment descriptortabellen worden reeds geïnitialiseerd zodat alle selectors met basisadres 0 beginnen en hun limiet tot het einde van de adresruimte (4GiB) reikt. Op die manier kunnen we ons toeleggen op enkel het pagineringsmechanisme.

Vraag A: Bestudeer om te beginnen de code en teken een schematisch overzicht van de *page directory* en paginatabel(len) op het moment dat paginering ingeschakeld wordt.

Zoals in Figuur 2.1 aangegeven wordt, is de locatie van het videogeheugen afhankelijk van de videomode. Een tweede deel van deze eerste opdracht is er voor zorgen dat zowel in kleuren- als in monochrome tekstmode hetzelfde logische adres gebruikt kan worden om naar het videogeheugen te schrijven.

Gebruik voor deze afbeelding ingang 1024 (gedefinieerd als de constante `VGA_PAGE_IDX`) van de paginatabel. We hebben maar één ingang nodig, aangezien steeds 2 bytes per teken nodig zijn (zowel in kleuren- als in monochrome mode). Dit komt neer op 4000 bytes voor een resolutie van 80x25 tekens.

Test je aanpassingen door bovenaan het assemblerbestand de `VIDEO_MODE` en `VGA_MEM` constanten voor de kleurenmode uit te commentariëren, en deze voor de monochrome mode te activeren. Verwijder eveneens de afbeelding van het videogeheugen op haar oorspronkelijke plaats. Vergeet de routine `printchar` ook niet aan te passen!

2.6.2 Opgave 2: De code verplaatsen

Nu paginering werkt, kunnen we dit aanwenden om onze code uit de eerste 640KiB te halen en deze na de eerste MiB te plaatsen in het fysiek geheugen, zonder dat we hiervoor enige adressen van code of data moeten aanpassen (afgezien van wat het pagineren betreft).

Maak een kopie van de eerste 48 KiB (alle geassembleerde code en data past hierin) fysiek geheugen en plaats het resultaat vanaf het begin van de eerste MiB (je kan eventueel de constante `Displacement` gebruiken, die als waarde 1 MiB heeft).

Vraag B: Beschrijf hieronder de nodige aanpassingen aan de paginatabel opdat deze kopie gebruikt wordt i.p.v. het origineel nadat paginering is ingeschakeld. Implementeer deze functionaliteit vervolgens.

Het kopiëren van een grote hoeveelheid gegevens kan je als volgt doen:

```
mov esi, BRONADRES
mov edi, DOELADRES
mov ecx, HOEVEELHEID/4
rep movsd
```

2.6.3 Opgave 3: De oorspronkelijke code wissen

Vraag C: Om zeker te zijn dat enkel de kopie gebruikt wordt nadat je de aanpassingen in stap 2 hebt uitgevoerd, gaan we nu de oorspronkelijke code en data overschrijven (dus de eerste 48 KiB van het fysiek geheugen gaan we wissen). Denk eraan dat je de code die je nog aan het uitvoeren bent niet mag overschrijven, en dat de paginatabellen zich standaard eveneens in het stuk geheugen bevinden dat je gaat overschrijven. Hoe kan je dit oplossen?

Het overschrijven van data kan je als volgt doen:

```
mov eax, 0xCCCCCCCC
mov edi, BEGINADRES
mov ecx, HOEVEELHEID/4
rep stosd
```

Waarom overschrijven we met het bitpatroon `0xCCCCCCCC`? *Hint: google naar 'int3'*.

Op het einde van het practicum mail je je code (vermeld de namen van de groepsleden bovenaan de code in commentaar!) naar michiel.ronsse@ugent.be met als onderwerp "BS_PAGING" en geef je het oplossingenblad af.

Besturingssystemen - AJ 2020-2021
Bochs-practicum 1 (paginering)

Persoon 1:

Persoon 2:

Antwoord A:

Antwoord B:

Antwoord C:

(Beschrijf uw oplossing, schrijf geen code neer!)

Hoofdstuk 3

Practicum 2: Protectie en segmentatie (Bochs)

In een vorig practicum werd paginering van naderbij bekeken. Paginering laat ons toe processen naast elkaar in het geheugen te laten bestaan zonder dat ze kennis hebben van elkaars bestaan. Een andere mogelijkheid om dit te verwezenlijken is door gebruik te maken van segmentatie. Het voordeel hiervan is dat aan de verschillende segmenten beveiligingsniveaus kunnen worden toegekend.

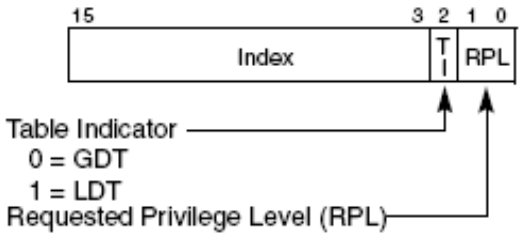
3.1 Segmenten

In de oorspronkelijke 8086 (een 16 bit processor) is het met behulp van segmentatie mogelijk om meer dan 64KiB te adresseren. Dit gebeurt door segmenten te definieren van maximaal 64KiB met telkens 16 byte tussen 2 opeenvolgende segmenten. Hierbij komt een adres van de vorm `segment:verschuiving` dan overeen met het lineaire adres `segment*16+verschuiving`.

Aangezien het maximaal adresseerbare geheugen op deze manier tot iets meer dan 1MiB (adres `0x10FFEF`) beperkt wordt, werd in de 286 een nieuwe mode (de zogenaamde *protected mode*) toegevoegd die het hiervoor beschreven geheugenmodel wijzigt. Wegens compatibiliteitsredenen starten de 286 en latere processors (tot en met de laatste generatie Core i9) echter nog steeds op in de zogenaamde *real mode* waar de beperkingen van het oude geheugenmodel gelden.

Wanneer de *protected mode* geactiveerd wordt, bevatten de 13 hoogste bits van de 6 segmentselectorregisters (CS,DS,SS,ES,FS,GS) (zie figuur 3.1) een index in de GDT (`global descriptor tabel`) of de LDT (`local descriptor table`). Hierbij wordt gekozen tussen de LDT en de GDT aan de hand van de derde laagste-orde bit van de selector.

Elk van deze beide tabellen kan maximaal 8192 segment-descriptoren bevatten, en hun beginadressen bevinden zich in twee speciale registers. De descriptoren bevatten in het algemeen de vol-



Figuur 3.1: Selector in protected mode

gende informatie (zie ook figuur 3.2).

- Base : Het beginadres van het segment
- Limit: Bepaalt de grootte van het segment (samen met de G-vlag).
- Type : Welk type descriptor (code-,data- of systeemdescriptor).
- P(resent) vlag: is het segment dat overeenkomt met de descriptor aanwezig in het geheugen?
- G(ranulatity) vlag : Bepaalt of limit de grootte van het segment in bytes of in blokken van 4KiB uitdrukt.
- DPL (Desired Privilege Level) : vereist beveiligingsniveau.
- C-vlag (Codesegment): Is het codesegment conforming of niet?
- R-vlag (Codesegment): Kan uit het codesegment gelezen worden?
- W-vlag (Datasegment): Mag in het datasegment geschreven worden?

Een adres bestaat ook nu weer uit een `selector:verschuiving` paar, maar de berekening van het bijhorend lineair adres is grondig veranderd. Uit de descriptor overeenkomstig met de segmentselector wordt het beginadres van het segment opgehaald waarbij dan de verschuiving geteld wordt om het lineair geheugenadres te bekomen (zie figuur 3.3).

Alvorens dit gebeurt wordt echter eerst gecontroleerd of het uiteindelijke adres zich wel binnen het segment bevindt (m.a.w. de opgegeven verschuiving mag niet groter zijn dan de limiet in de descriptor). Daarnaast wordt ook gecontroleerd of de opgegeven descriptor wel het goede type heeft. Zo kan men geen code in een datasegment uitvoeren, of schrijven naar een alleen-lezen-segment, etc. Tenslotte wordt ook geverifieerd of de uitgevoerde instructie genoeg privileges heeft om het segment te benaderen.

3.2 Bescherming

In tegenstelling tot sommige andere architecturen is er in de Intel x86-architectuur geen expliciet register of statusveld dat het huidige beveiligingsniveau weergeeft. Er zijn dus ook geen specifieke instructies om dit niveau te veranderen.

Iedere uitvoerende instructie heeft een beveiligingsniveau dat afhangt van zijn bijhorend codesegment. Dit huidige beveiligingsniveau (CPL: current privilege level) is gelijk aan de DPL van de bijhorende codesegmentdescriptor.

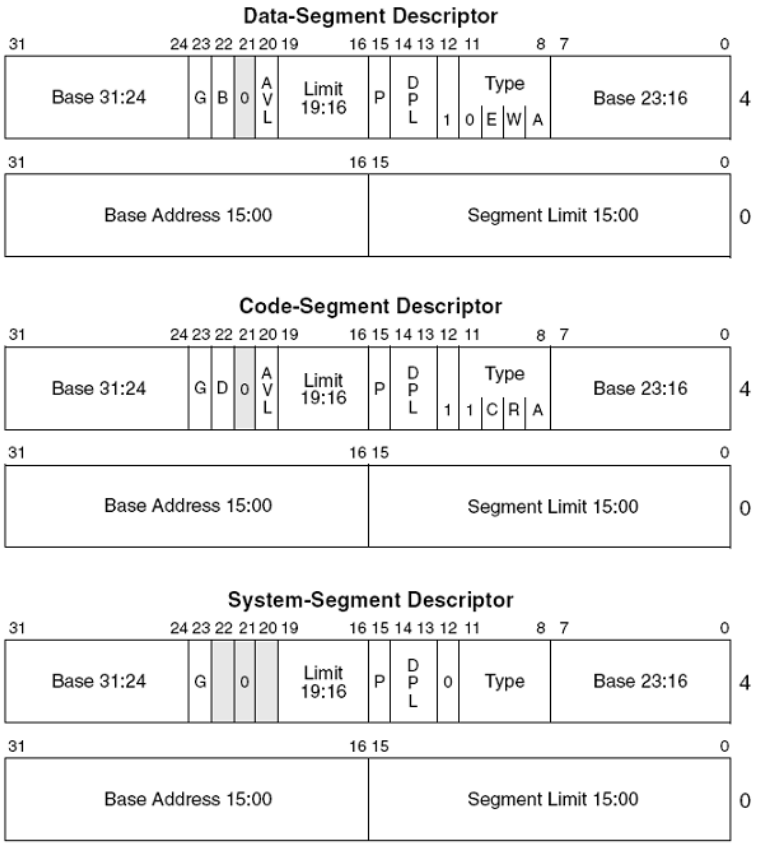
Aangezien er slechts 2 bits voorzien zijn, zijn er slecht 4 beveiligingsniveau's. Deze kunnen aanzien worden als concentrische cirkels van niveau's 0 tot 3 (zie figuur 3.4), waarbij 0 het meest geprivilegeerde niveau is en 3 het minst.

Wanneer men een toegang tot een datasegment beschouwt, geldt de regel dat men wel van binnen naar buiten kan kijken maar niet omgekeerd. Instructies met een hoger beveiligingsniveau (lagere CPL) kunnen dus steeds lezen (en/of schrijven) naar een datasegment met een lager beveiligingsniveau (hogere DPL). Het omgekeerde is ongewenst.

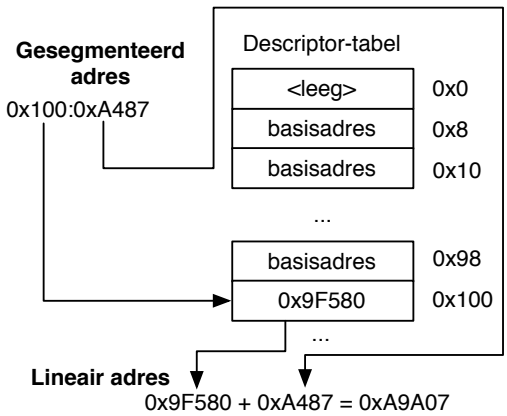
Wanneer we een toegang tot een ander codesegment beschouwen, zoals een `far jump` naar een ander codesegment, liggen de zaken iets complexer. Het is immers enkel toegestaan een ander codesegment te benaderen indien het gewenste beveiligingsniveau (DPL) van dit segment gelijk is aan het huidig beveiligingsniveau (CPL).

Nu kan het voorkomen dat een applicatie, die op het laagste beveiligingsniveau (=3) draait, een bepaalde geprivilegeerde taak wil uitvoeren. Daarom moet hij in staat zijn om code op te roepen die zich in een segment met een hoger beveiligingsniveau bevindt.

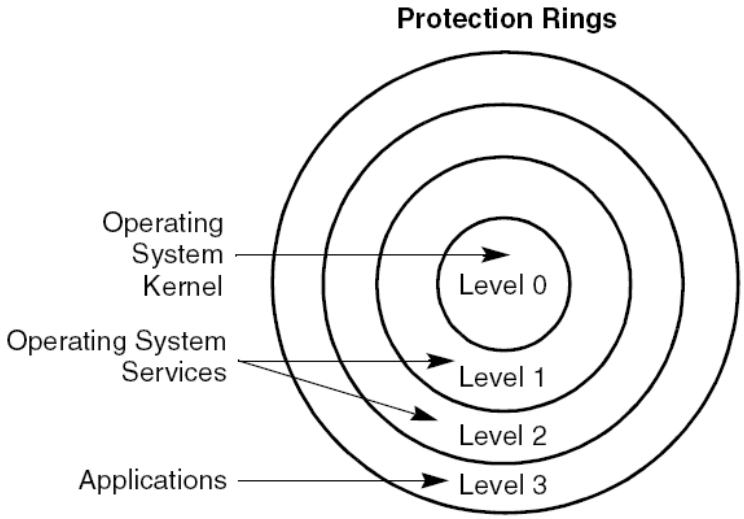
Dit kan op twee manieren gebeuren:



Figuur 3.2: Soorten Descriptors



Figuur 3.3: Descriptors in protected mode



Figuur 3.4: Beveiligingsniveau's

- Het gebruik van een conforming codesegment. Dit is een codesegment waarvan de conforming-vlag in de bijhorende descriptor op 1 staat. Wanneer men toegang wil tot zo'n segment moet het huidig beveiligingsniveau gelijk of lager zijn dan het gewenste beveiligingsniveau van het conforming codesegment. Hierbij dient wel opgemerkt dat het huidig beveiligingsniveau niet verandert, ook al is het gewenste beveiligingsniveau van het conforming codesegment groter.

- Het gebruik van een callgate. Hierbij is het mogelijk van code met een lager beveiligingsniveau te springen naar code met een hoger niveau, m.a.w. het huidig beveiligingsniveau zal wel veranderen eens in het nieuwe segment aangekomen.

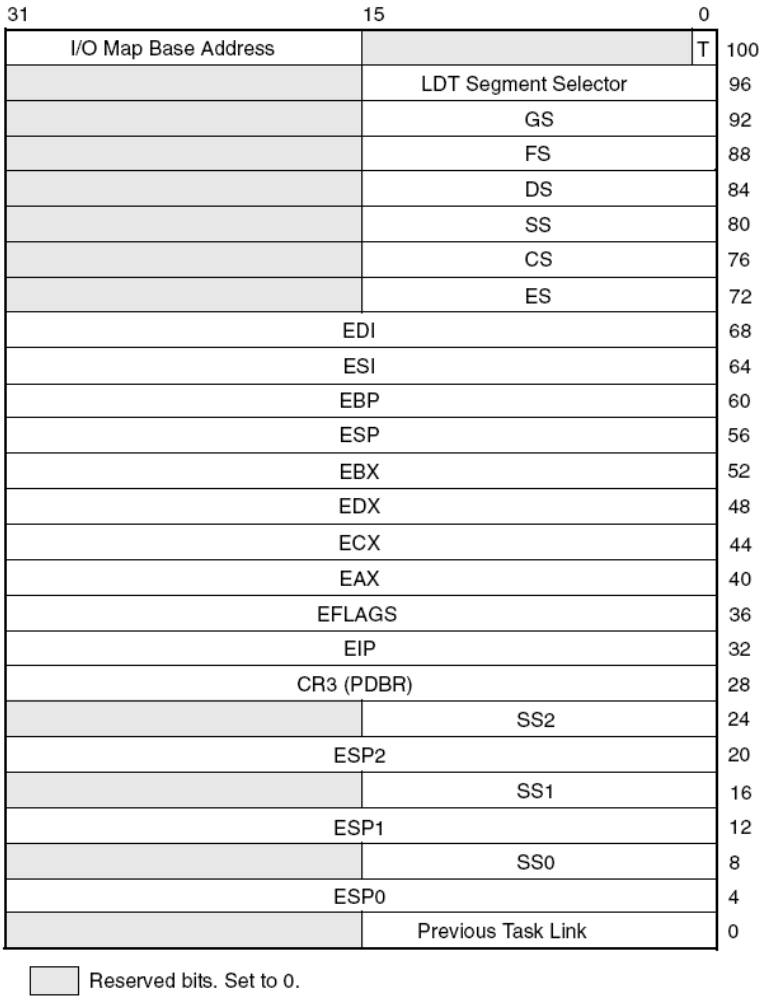
Zoals misschien reeds is opgevallen, is er nog geen mogelijkheid vermeld waarmee van code met een hoger beveiligingsniveau naar code met een lager niveau kan gesprongen worden. Dit is enkel mogelijk mits gebruik van de taakwissel-mogelijkheden van de processor.

Een opmerking dient wel gemaakt te worden over de 2 laagste orde bits van het selectorregister. Deze bepalen het gevraagde beveiligingsniveau (Requested Privilege Level), dit laat een nog fijnkorreligere controle toe van de beveiliging. Meestal wordt de RPL echter gelijk gesteld aan de DPL van de descriptor waarnaar verwezen wordt.

3.3 Taakwisseling

Een andere functionaliteit die vanaf de 386 ingebouwd zit in de Intel x86-processoren is de hardwarematige ondersteuning van multitasking. Dit laat toe om met één instructie van taak te wisselen (deze instructie neemt dan wel de nodige tijd in beslag).

Hiervoor is plaats in het geheugen nodig om de toestand van de taken in het geheugen te bewaren. Deze wordt geregeld aan de hand van een nieuw soort segment, het taaktoestandssegment (TSS: Task State Segment). Dit segment moet minstens 104 bytes groot zijn. De eerste 104 bytes zijn gereserveerd omdat de processor hier automatisch de registers voor algemeen gebruik, de segmentselectors, de programmateller, de wijzer naar de paginadirectory etc. opslaat (zie figuur 3.5). De rest van het segment kan gebruikt worden door het besturingssysteem om extra informatie horend bij de uitvoering van de taak op te slaan.



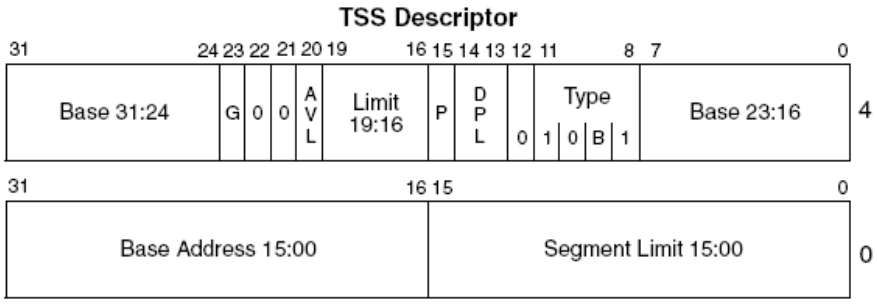
Figuur 3.5: TSS-segment

De eerste 104 bytes bevatten echter ook drie extra stapelsegmentselectors en stapelwijzers. Deze worden gebruikt indien een taak tijdelijk van beveiligingsniveau verandert, door bijvoorbeeld hoger gepriviligeerde code aan te roepen met een callgate.

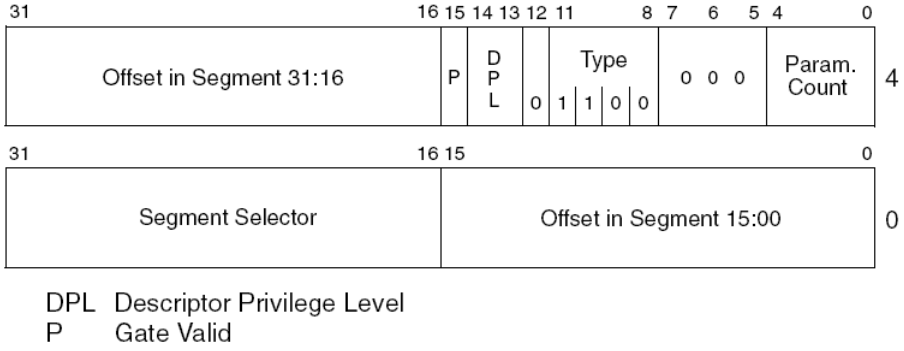
Bij dit taaktoestandssegment hoort natuurlijk ook een eigen type descriptor, de TSS-descriptor (figuur 3.6). Deze heeft dezelfde beveiligingsvoorwaarden als een codesegmentdescriptor, m.a.w. enkel hoger gepriviligeerde code mag een taakwissel veroorzaken. Wanneer een far call of far jump naar een ander taaktoestandssegment gebeurt, wordt niet gewoon van codesegment gewisseld zoals bij een far call/jump naar een ander codesegment. Eerst wordt de toestand van de huidige taak opgeslagen in het huidige TSS, vervolgens wordt de toestand van de nieuwe taak uit het segment waar de opgegeven TSS-selector naar verwijst, opgehaald en wordt verdergegaan met de uitvoering van de nieuwe taak (met nieuwe selectors en een nieuwe instructiewijzer).

3.4 Veranderen van beveiligingsniveau: Callgate

Zoals reeds eerder vermeld, is de aangewezen manier om hoger gepriviligeerde code uit te voeren het gebruik van een callgate. (Het is ook mogelijk een interruptgate of trapgate te gebruiken; de



Figuur 3.6: TSS Descriptor



Figuur 3.7: Call Gate Descriptor

werking hiervan is echter analoog zodat enkel de callgate besproken wordt.)

Een callgate is een speciale descriptor (figuur 3.7) die een soort indirecte sprong toestaat naar een hoger geprivilegeerde functie.

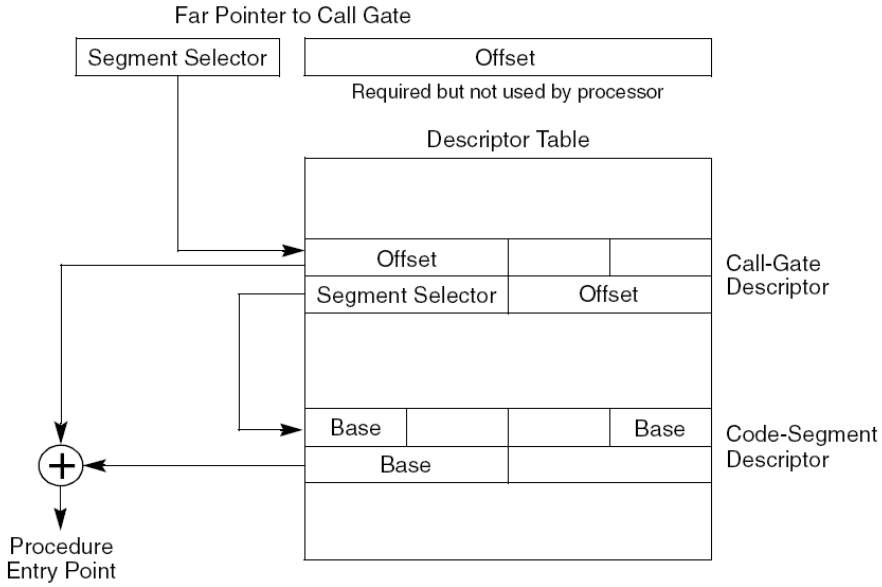
De belangrijkste velden zijn :

- Segment Selector: Deze selector wijst naar het segment waar de geprivilegeerde code zich bevindt.
- Offset: De offset binnen dat segment waar de functie zich bevindt.
- Param. Count: het aantal argumenten op de stapel voor de geprivilegeerde functie.
- DPL: Het minimale huidig beveiligingsniveau dat men moet hebben om de callgate te mogen gebruiken.

Het is dus mogelijk de callgate een lager beveiligingsniveau toe te kennen dan het beveiligingsniveau van de hoger geprivilegeerde functie. Dit vormt geen probleem aangezien bij een far call/jump naar een callgate gebruik wordt gemaakt van de offset in de callgate en niet deze van de far call/jump (zie figuur 3.8). Bij deze far call/jump dient wel een offset opgegeven te worden, hij wordt echter niet gebruikt.

Wanneer ten gevolge van het gebruik van een callgate het huidig beveiligingsniveau verandert, moet ook van stapel gewijzigd worden. Dit heeft 2 redenen:

- Het is mogelijk dat indien een hoger geprivilegeerde functie de stapel van een lager geprivilegeerde taak gebruikt, er gegevens kunnen lekken van een hoger naar een lager geprivilegeerd niveau.



Figuur 3.8: Gebruik van Call Gate

- Daarnaast is het ook mogelijk dat er geen stapelruimte meer vrij is op de stapel van de taak die de callgate gebruikt heeft.

Deze stapelwijziging is mogelijk dankzij de 3 extra stapelsegmentselectors en stapelwijzers die zich in het taaktoestandssegment bevinden. Deze 3 extra stapels zijn er voor de priviligeniveau's 0 tot 2. De huidige stapel wordt bewaard door eerst op de nieuwe stapel de selector en stapelwijzer van de huidige stapel bij te houden. Vervolgens wordt het noodzakelijk aantal argumenten gekopieerd van de huidige naar de nieuwe stapel en tenslotte wordt het terugkeeradres op de nieuwe stapel gezet. (Dit wordt allemaal wat duidelijker gemaakt in figuur 3.9).

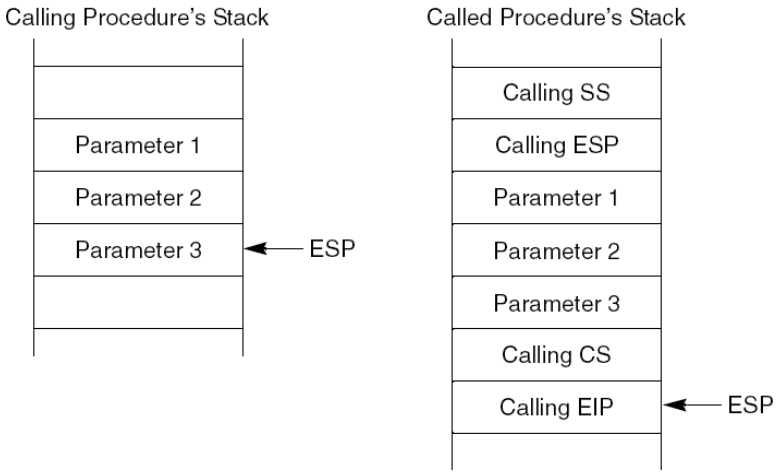
3.5 Opgave

Download het bestand `Practicum_2_Bochs-protectie.zip` van de Ufora website, en unzip het. Alle code staat in het bestand `protect.asm`. Zie de uitleg bij Practicum 1 over het gebruik van Bochs.

Vraag A: Analyse van de taakwisseling

De bijgevoegde code zal bekend voorkomen. Ze werd echter aangepast om gebruik te maken van taakwisseling en segmentbescherming. Merk hierbij op dat het videogeheugen een eigen segment gekregen heeft. Bestudeer de code en geef schematisch weer:

- De plaats van de segmenten in het geheugen en hun beveiligingsniveau (hierbij is het interessant eerst de geheugenas te tekenen met ernaast de segmenten, om onduidelijkheden te voorkomen). Als u van sommige segmenten enkel een symbolisch beginadres kent mag u dit gebruiken in de figuur.
- De waarde van belangrijkste velden (nl. de segment selectors en stapelwijzers) in de taaksegmenten net voor de eerste taakwisseling. Gebruik hiervoor een aparte figuur.



Figuur 3.9: Wisselen van Stapel

Vraag B: Gebruik van een conforming codesegment

Aangezien het niet aangeraden is dat gebruikersniveautoepassingen rechtstreekse toegang hebben tot het videogeheugen, gaan we nu de printchar routine als een kernelfunctie implementeren. Een eenvoudige manier om dit te doen is het codesegment dat de kernelfunctionaliteit bevat veranderen in een conforming segment. Hiervoor dient u volgende stappen te volgen :

- Pas de reeds aanwezige kernelfunctie aan zodat de goede argumenten van de stapel gebruikt worden.
- Pas de descriptor van het codesegment aan zodat het een conforming segment wordt.
- Pas de spiraal-routine aan zodat de kernelfunctie gebruikt wordt in plaats van de gebruikersroutine.

Test uw oplossing en beschrijf *in detail* de gedane wijzigingen.

Nu zorgen we ervoor dat enkel kerneltaken nog toegang hebben tot het videogeheugen. Hoe doet u dit? Werkt uw oplossing nog? Indien niet, waarom niet?

Vraag C: De callgate

Nu wordt de callgate geïmplementeerd. U moet dus een nieuwe descriptor en bijhorende selector in de GDT maken. Hierbij kan u uitgaan van de reeds gedefinieerde descriptors en selectors. Om de velden juist in te vullen is het gebruik van de gedefinieerde constanten aan te raden. Vergeet ook niet om het kernelcodesegment terug niet-conforming te maken.

Eens de callgate geïmplementeerd is, dient u de spiraalroutine weer aan te passen om gebruik te maken van de callgate. Test uw oplossing met het beveiligingsniveau van het videosegment maximaal.

Wanneer uw oplossing werkt mag u de inhoud van uw callgate met de nodige commentaar (cfr. de reeds in de code aanwezige selectors) neerschrijven op het oplossingenblad.

Besturingssystemen - AJ 2020-2021
Practicum Protectie en Segmentatie)

Persoon 1:

Persoon 2:

Antwoord A:

Antwoord B:

Antwoord C:

Hoofdstuk 4

Linux-practica: inleiding

4.1 Platform

Voor de drie UNIX-practica werd er gekozen voor Linux als besturingssysteem en dit omdat de broncode ervan vrij beschikbaar is en omdat Linux makkelijk uitbreidbaar is met nieuwe device drivers, bestandssystemen, e.d.m.

Om de practica vlot te laten verlopen wordt er een minimale kennis van UNIX-commando's vereist. Wie deze niet heeft vindt een korte uitleg van de commando's die we zullen gebruiken in Appendix A.

De Linux-practica worden uitgevoerd onder een virtuele machine: QEMU. De virtuele machine en de code vind je in qemu.zip op Ufora. Pak de zip uit en start de virtuele machine onder Windows uit met `begin.bat` . Voor Linux of MacOS moet je qemu eerst nog installeren, zie `https://www.qemu.org`.

Als je virtuele PC 'hangt' moet je QEMU herstarten. Doe dit door op de sluitknop 'x' in de rechterbovenhoek van QEMU te klikken en herstart m.b.v. `begin.bat`

Linux start op in een omgeving met vier teksterminals (gebruik `Alt-Fx`, met x=1..4 om te schakelen tussen de schermen). Inloggen kan als `root`, een paswoord is niet nodig. `Alt-F5` toont foutboodschappen van de kernel of applicaties (zelfde informatie die `dmesg` of `/var/log/syslog` toont). U kunt Linux stoppen d.m.v. het `halt`-commando.

4.2 Overzicht van het Linux-besturingssysteem

Het Linux-besturingssysteem bestaat uit een *kernel* en een aantal hulpprogramma's. De kernel is het eigenlijke hart van het besturingssysteem en zorgt voor een uniforme interface naar de hardware en voor procesbeheer, geheugenbeheer, ... De hulpprogramma's stellen de gebruiker in staat om te communiceren met de kernel in een gebruiksvriendelijke manier, bv. *shells* om programma's te starten, de X-omgeving met bijhorende *window managers* om grafische in- en uitvoer mogelijk te maken, *file managers* en programma's als `cp, mv, ...` om bestanden te manipuleren. De kernel zelf is het belangrijkste onderdeel van het besturingssysteem en in het vervolg van deze tekst en in de practica zullen we enkel de kernel bestuderen.

4.2.1 De kernel

De kernel is een stuk code dat steeds in het geheugen aanwezig is en dat zorgt voor procesbeheer (starten van nieuwe processen, schedulen van processen, ...), geheugenbeheer, bestandsbeheer (m.b.v. bestandssystemen), en hardwarebeheer (m.b.v. device drivers). Bovendien zorgt het voor een uniforme toegang tot de diverse hardware zoals harde schijven, diskettes, CD's, ... Daartoe heeft de

kernel twee interfaces: een interface naar de applicaties (processen) en een interface naar de hardware. Figuur 4.1 toont een schematisch overzicht van de Linux-kernel waarop we de twee interfaces en de belangrijkste onderdelen van de kernel onderscheiden. Volgende tabel[1] toont het aantal lijnen code per onderdeel:

	C-code	assembler
centrale kernel	50918	8791
device drivers	856731	615
bestandssystemen	124574	0
netwerk	148674	0
totaal	1180897	9406

Processen en hardware kunnen de kernel vragen om een functie uit te voeren; beiden maken daartoe gebruik van een *interrupt*. Processen doen dit via de *system calls interface* door interrupt 128 op te roepen terwijl de hardware dit doet door een IRQ te lanceren (bv. de systeemklok, de toetsenbordcontroller, ...). Het bestand `/proc/interrupts` bevat steeds de laatste informatie over het aantal opgetreden hardware-interrupts. Daarnaast bestaat nog de mogelijkheid dat de kernel een functie uitvoert omdat er iets 'fout' gaat met een proces, zoals een deling door nul, een segmentatiefout, enz. Deze fouten zorgen ook voor het optreden van een interrupt, in dit geval spreekt men trouwens meestal over *excepties*. Merk op dat dit de enige mogelijkheden zijn om een kernelfunctie uit te voeren.

De kernel bevindt zich steeds in het geheugen en dit tussen de (virtuele) adressen 3GiB en 4GiB[2]. De ruimte tussen 0GiB en 3GiB wordt op elke moment gebruikt door exact één proces[3] (zie Figuur 4.2).

4.2.2 Kernelfuncties

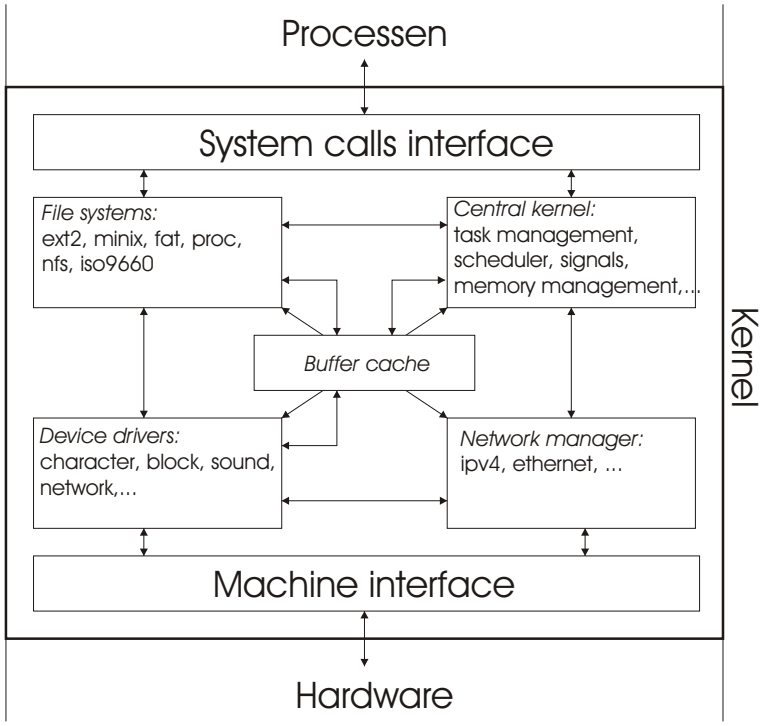
Zoals reeds vermeld kan een proces een kernelfunctie laten uitvoeren, en dit door het oproepen van interrupt 128. Door register `%eax` een bepaalde waarde te geven kiezen we voor een bepaalde functie, terwijl `%ebx`, `%ecx` en `%edx` de argumenten voor de functie zijn (maximum drie argumenten dus!). De lijst van alle kernelfuncties (190 in totaal) is te vinden in `/usr/include/asm/unistd.h`. Het begin van dit bestand ziet er als volgt uit:

```
1:  #define __NR_exit                1
2:  #define __NR_fork                2
3:  #define __NR_read                3
4:  #define __NR_write               4
5:  #define __NR_open                5
6:  #define __NR_close               6
7:  ...
```

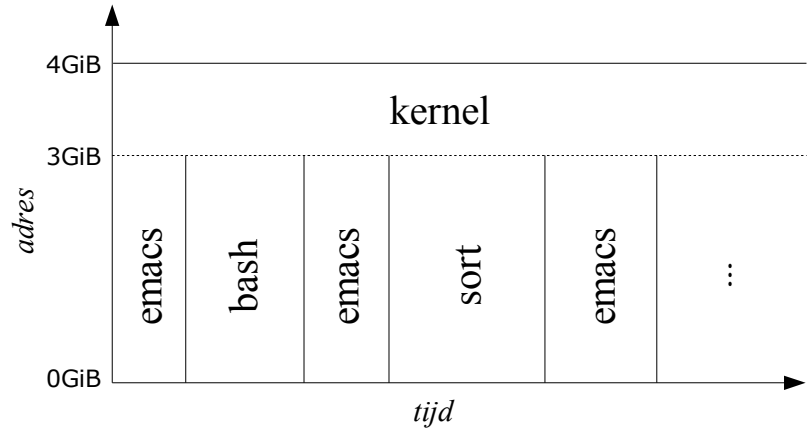
We merken bv. dat er één kernelfunctie is om uit bestanden te lezen: `read` met functienummer 3. Alle functies uit de C-bibliotheek die we kunnen gebruiken om te lezen (`read()`, `fread()`, `gets()`, `getchar()`, `fgetc()`, `fgets()`, `fscanf()`, ...) maken dus gebruik van deze systeemoproep. Merk op dat alle systeemoproepen ook rechtstreeks oproepbaar zijn vanuit een applicatie, daartoe worden elke systeemoproep ingekapseld in een klein proceduurtje (in dit geval `read()`) dat niet meer doet dan de argumenten kopiëren van de stapel naar de registers. Om een en ander te verduidelijken volgen we de weg die afgelegd wordt indien we volgend stukje code uitvoeren (zie ook Figuur 4.3):

```
8:  #include <stdio.h>
```

[1] Alle gegevens in deze handleiding hebben betrekking op kernel 2.4.7.
[2] We gebruiken een 32-bits processor.
[3] Meerdere processen bij multiprocessoren.



Figuur 4.1: De belangrijkste onderdelen van de Linux-kernel.



Figuur 4.2: De kernel bevindt zich steeds tussen de (virtuele) adressen 3GiB en 4GiB terwijl de onderste 3GiB door de verschillende processen (na elkaar!) gebruikt wordt.

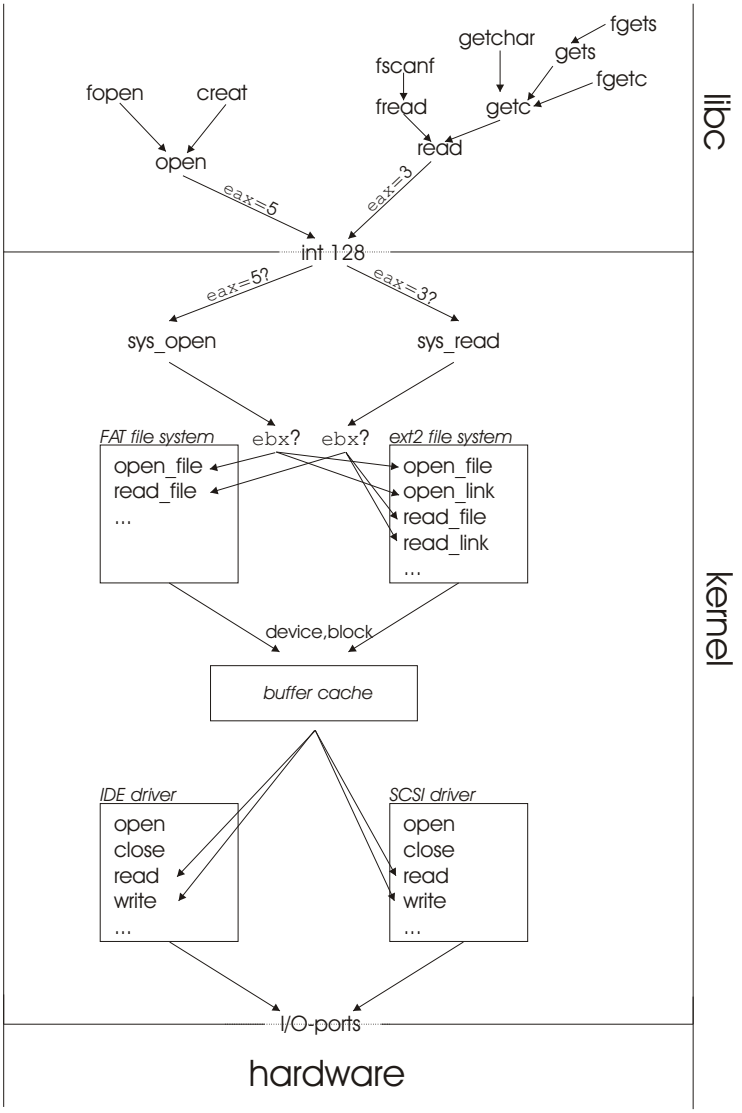
```
9:
10:  main(){
11:          int kar=getchar();
12:  }
```

Dit stukje code leest één karakter van standaardinvoer (normaal gezien het toetsenbord). De functie `getchar()` is een C-functie die (via een omweg via `getc()`) gebruik maakt van de `read()`-functie. Deze functie is niet meer dan een wrapper die de drie argumenten van $read()$ (fd, buf en count) van de stapel naar registers kopieert (regels 15, 16 en 17) en die nadien systeemoproep 3 uitvoert door interrupt 128 uit te voeren met argument %eax=3 (regels 18 en 19):

```
13:  00000000 <__libc_read>:
14:      pushl  %ebx
15:      movl   0x10(%esp,1),%edx
16:      movl   0xc(%esp,1),%ecx
17:      movl   0x8(%esp,1),%ebx
18:      movl   $0x3,%eax
19:      int    $0x80
20:      popl   %ebx
21:      cmpl   $0xfffff001,%eax
22:      jae    1c <__libc_read+$0x1c>
23:      ret
```

Het oproepen van interrupt 128 heeft tot gevolg dat de processor springt naar de interrupthandler. Het adres van deze handler wordt opgezocht in een tabel en dit adres wordt door de kernel ingevuld bij het booten (gebeurt in de functie `trap_init()` gedefinieerd in `/usr/src/linux/arch/i386/kernel/traps.c`):

```
24:  set_system_gate(128, &system_call);
```



Figuur 4.3: De weg die wordt afgelegd indien `getchar()` wordt uitgevoerd.

De interrupthandler voor interrupt 128 heet dus `system_call` en wordt gedefinieerd in het bestand `/usr/src/linux/arch/i386/kernel/entry.S` en ziet er als volgt uit (vanaf dit moment 'zitten' we dus in de kernel):

```
25:  ENTRY(system_call)
26:          pushl %eax
27:          SAVE_ALL
28:          GET_CURRENT(%ebx)
29:          cmpl $(NR_syscalls),%eax
30:          jae badsys
31:          testb $0x20,flags(%ebx)
32:          jne tracesys
33:          call *SYMBOL_NAME(sys_call_table)(,%eax,4)
34:          movl %eax,EAX(%esp)
35:      ...
```

We merken dat `system_call` o.a. test of het nummer van de systeemoproep niet te groot is, waarna het adres van de systeemroutine die de systeemoproep moet afhandelen opgezocht wordt in de tabel `sys_call_table`, gebruik makende van `%eax` als index (het nummer van de systeemoproep). Nadien wordt er gesprongen naar de routine die zich op dit adres bevindt. Het begin van `sys_call_table` ziet er als volgt uit:

```
36:  .data
37:  ENTRY(sys_call_table)
38:          .long SYMBOL_NAME(sys_ni_syscall)
39:          .long SYMBOL_NAME(sys_exit)
40:          .long SYMBOL_NAME(sys_fork)
41:          .long SYMBOL_NAME(sys_read)
42:          .long SYMBOL_NAME(sys_write)
43:          .long SYMBOL_NAME(sys_open)
44:      ...
```

De eigenlijke routine die de leesoperatie zal uitvoeren is dus `sys_read()`. Deze routine is verassend klein:

```
45:  ssize_t sys_read(unsigned int fd, char * buf, size_t count){
46:          ssize_t ret;
47:          struct file * file;
48:          ssize_t (*read)(struct file *, char *, size_t, loff_t *);
49:
50:          lock_kernel();
51:
52:          ret = -EBADF;
53:          file = fget(fd);
54:          if (!file)
55:                  goto bad_file;
56:          if (!(file->f_mode & FMODE_READ))
57:                  goto out;
58:          ret = locks_verify_area(FLOCK_VERIFY_READ,
59:                                  file->f_dentry->d_inode,
60:                                  file, file->f_pos, count);
61:          if (ret)
62:                  goto out;
63:          ret = -EINVAL;
64:          if (!file->f_op || !(read = file->f_op->read))
65:                  goto out;
66:          ret = read(file, buf, count, &file->f_pos);
67:  out:
```

```
68:             fput(file);
69:     bad_file:
70:             unlock_kernel();
71:             return ret;
72:     }
```

We merken dat de functie een aantal testen uitvoert: is het bestand wel geopend, mag het bestand gelezen worden en is het bestand niet gelocked. Op basis van `fd (=%ebx)` wordt de uit te voeren leesfunctie bepaald (regel 64). Het invullen van het adres van de juiste functie gebeurt tijdens het openen van het bestand. Merk op dat het op deze manier niet alleen mogelijk is om tekiezen tussen leesfuncties die zich bevinden in diverse bestandssystemen (afhankelijk van het bestandssysteem dat het bestand bevat, bv. FAT of ext2), maar ook tussen diverse leesfuncties van één bestandssysteem. Dit laatste wordt gebruikt indien een bestandssysteem meerdere soorten bestanden kan bevatten (gewone bestanden, links, devices, pipes, doors, ...) die elk een specifieke leesfunctie hebben. Tenslotte wordt deze functie opgeroepen (regel 66), waarbij deze functie de testen die `sys_read` reeds uitgevoerd heeft niet meer moeten uitgevoeren (bv. testen of het bestand wel gelezen mag worden door de gebruiker). Deze functie weet op welk device het bestandssysteem zich bevindt (opgegeven bij het 'mounten', zie verder) en geeft een leesopdracht door aan de buffercache die uiteindelijk via de device driver de data opvraagt aan de hardware.

De meeste kernelcode bevindt zich in een bestand dat bij het booten geladen wordt (`/vmlinuz`). Linux laat echter toe om kernelcode dynamisch te laden. Dit maakt het mogelijk om device drivers dynamisch in de kernel te laden. In dit geval wordt de geladen module gelinkt met de reeds aanwezige kernelcode, omdat de module functionaliteit aangeboden door de kernel zal gebruiken (bv. `strlen()`), en worden de bijkomende functies die de module aanbiedt ter beschikking gesteld via een sprongtabel. Een module `test.o` wordt geladen m.b.v. van het commando `insmod test.o`.

`insmod` voert de volgende stappen uit bij het laden van een module (zie Figuur 4.4):

1. de module wordt in kernel space geladen (ergens tussen 3GiB en 4GiB);
2. de module wordt gelinkt met de kernel;
3. de functie `init_module()`, gedefinieerd in de module, wordt uitgevoerd. Het is de bedoeling dat deze functie de functionaliteit van de module registreert in de kernel. In dit voorbeeld betreft het een device driver die zijn functionaliteit registreert d.m.v. `register_chrdev(...)`. Deze functie geeft 0 terug bij succes, anders een negatieve foutcode.

4.3 Ontwikkelen en testen van kernelcode

4.3.1 Schrijven en compileren van code

Elk C-programma dat kernelcode bevat moet starten met volgende lijnen:

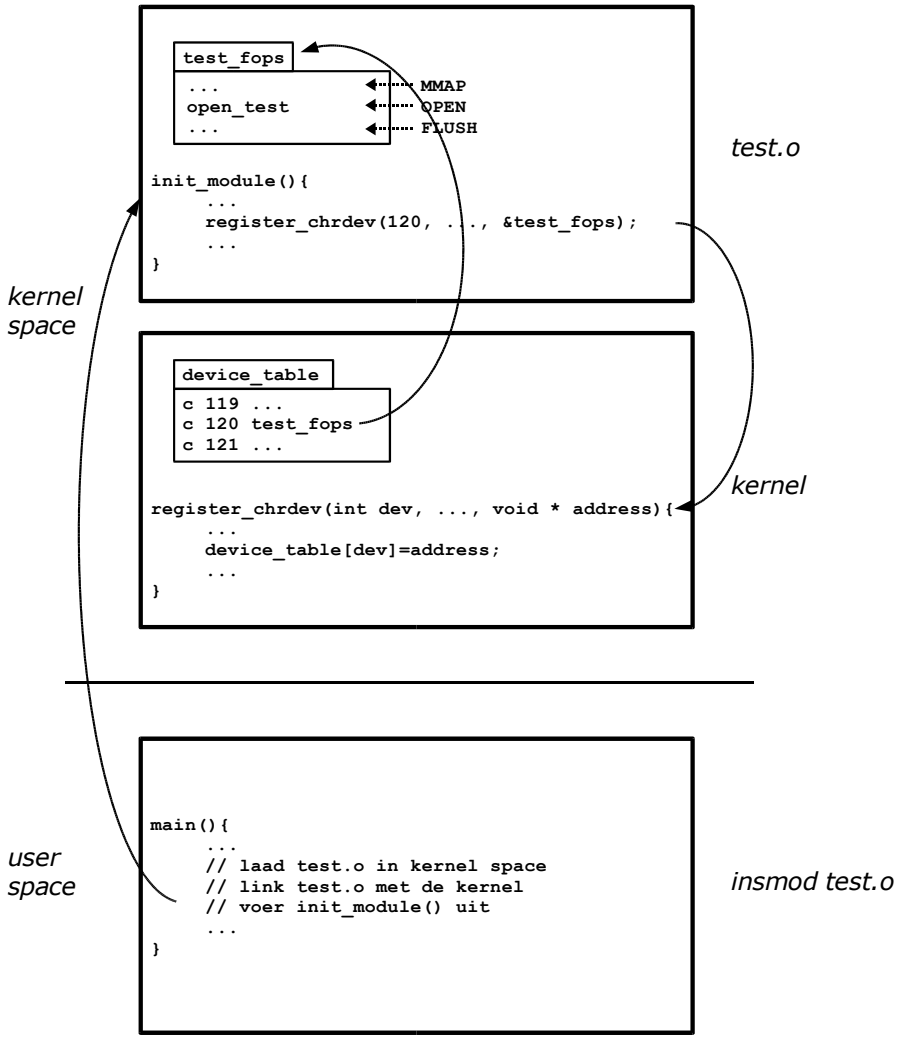
```
73:   #define __KERNEL__
74:   #include <linux/kernel.h>
75:   #include <linux/errno.h>
```

Maakt de code bovendien geen deel uit van de kernel, maar is het een zogenaamde module die dynamisch geladen kan worden, dan moeten volgende regels hieraan toegevoegd worden:

```
76:   #define MODULE
77:   #include <linux/module.h>
```

Het compileren van kernelcode *moet* als volgt gebeuren:

```
78:   gcc -O -c programma.c
```



Figuur 4.4: Het installeren van een module m.b.v. `insmod`.

De optimalisatievlag `-O` is verplicht omdat inline-functies anders niet geëxpandeerd worden.

Kernelcode kan enkel gebruik maken van functies die aangeboden worden door de kernel; dit is een beperkt aantal functies uit de C-bibliotheek zoals `strlen()`, `memset()`, ... In- en uitvoer m.b.v. `printf()` is in de kernel niet mogelijk daar de kernel geen standaard invoer, standaard uitvoer, ... heeft. Ook het openen van bestanden door de kernel is niet mogelijk. De enige manier waarop de kernel in- en uitvoer kan doen is door het rechtsreeks communiceren met een bestandssysteem of device driver; dit is echter niet eenvoudig: de kernel is geen proces dat een eigenaar heeft, wat betekent dat het bestand ook geen eigenaar zal hebben...

Ondanks de afwezigheid van `printf` kunnen we toch statusboodschappen op het scherm laten verschijnen. De kernel biedt daartoe de `printk()`-functie aan. Deze functie kent dezelfde argumenten als `printf()` (maar geen vlottende komma getallen!) en 'schrijft' de boodschap naar een circulaire buffer in het geheugen. Deze buffer wordt automatisch uitgelezen door `syslogd` die ze (in de PC-klas) op het 5^e tekstscherm zet. De boodschappen kunnen ook uitgelezen worden m.b.v. het `dmesg`-commando of via het `/var/log/syslog`-bestand.

Het oproepen van functies die aangeboden worden door o.a. bestandssystemen en device drivers gebeurt m.b.v. sprongtabellen. Dit maakt het dynamisch laden van code van bv. een nieuw bestandssysteem zeer eenvoudig.

Naast deze twee soorten functies (een subset van `libc()` en de functionaliteit aangeboden door de verschillende onderdelen van de kernel) kent de kernel nog een aantal belangrijke functies die specifiek zijn voor een kernel. De belangrijkste twee zijn `copy_to_user()` en `copy_from_user()`. `copy_to_user(void *naar, const void *van, unsigned long n)` kopieert `n` bytes vanaf adres `van` in *kernel* space naar adres `naar` in *user* space. `copy_from_user` doet de omgekeerde operatie. Beide functies geven het aantal niet gekopieerde bytes terug.

Hoofdstuk 5

Practicum 3: scheduler (Linux)

5.1 Doelstelling

Dit practicum heeft als doelstelling inzichten te verwerven over schedulers. Daartoe wordt de Linux-scheduler bestudeerd en worden een aantal alternatieve schedulers geïmplementeerd en getest.

5.2 Inleiding

Een belangrijk onderdeel van een kernel is de scheduler (taakverdeler). Deze zorgt er voor dat de verschillende processen om beurt uitgevoerd worden en dat elk proces eerlijk behandeld wordt, m.a.w. het monopoliseren van de processor door één proces wordt bestreden. Daartoe houdt Linux alle uitvoerbare processen bij in een `run_queue`. Dit is een dubbel gelinkte circulaire lijst met één startknoop (`init_task`) (zie Figuur 5.1). Het proces in uitvoering wordt steeds aangeduid door `current`.

De lijst bestaat uit een aantal `task_structs` waarin informatie over één uitvoerbaar proces wordt bijgehouden (zie `/usr/include/linux/sched.h`). De voornaamste elementen van deze structuur zijn: `pid` (procesnummer), `comm` (procesnaam), `state` (de toestand van het proces), `next_task` en `prev_task` (wijzers naar het volgende en het vorige proces), `counter` (lengte van het timeslice) en `policy, nice` en `rt_priority`. Deze laatste drie velden worden door de scheduler gebruikt om het volgende uit te voeren proces te kiezen.

Linux kent één speciaal proces: proces `swapper` met nummer 0 en prioriteit -1000 (zie verder). Dit is een kernelproces (met laagste *goodness*, zie later) dat (conceptueel) enkel

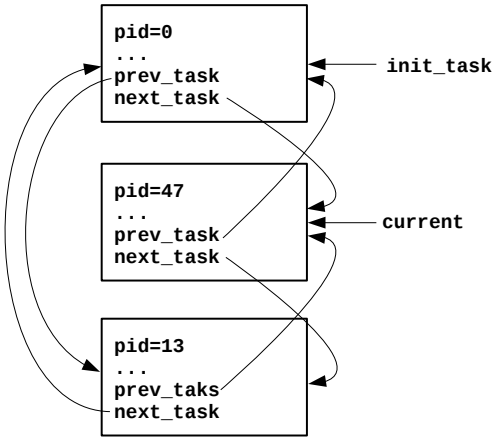
```
79:   while (1);
```

uitvoert. Dit proces wordt uitgevoerd indien er geen andere uitvoerbare processen zijn. Het feit dat er steeds minstens één uitvoerbaar proces is maakt het schedulen heel wat eenvoudiger.

Processen kunnen uit de `run_queue` verdwijnen doordat ze eindigen of doordat ze een I/O-operatie uitvoeren waardoor ze blokkeren. Processen worden op de lijst geplaatst indien het nieuwe processen zijn, of indien ze gedeblokkeerd worden (I/O-operatie afgelopen). Deze processen worden d.m.v. `add_to_runqueue()` bovenaan de lijst geplaatst, maar wel na proces 0.

De scheduler zelf (`schedule()` in `/usr/src/linux/kernel/sched.c`) wordt opgeroepen indien

- het lopende proces de `sched_yield()`-operatie uitvoert. Op deze manier kan een proces de processor vrijwillig afstaan;
- het lopende proces een blokkerende I/O-operatie (d.m.v. een systeemoproep) start. In dit geval verdwijnt het proces dus van de `run_queue`.



Figuur 5.1: De `run_queue`

- een interruptroutine hiertoe beslist. Zo'n routine wordt o.a. opgeroepen indien een I/O-operatie afgelopen is. In dit geval *kan* het gedeblokkeerde proces weer op de `run_queue` geplaatst worden. Niet alle processen worden echter onderbroken door een interrupthandler (zie verder). De belangrijkste interrupthandler is `do_timer();` deze wordt periodisch opgeroepen (elke 10ms (=1 jiffie)). Deze routine moet ervoor zorgen dat de processor niet gemonopoliseerd wordt door één proces.

De velden `counter, policy, nice` en `rt_priority` staan in voor de werking van de scheduler. `Policy` (`SCHED_FIFO`, `SCHED_RR` of `SCHED_OTHER`) bepaalt de klasse waarin het proces zich bevindt. `Nice` is een parameter voor `SCHED_OTHER` terwijl `rt_priority` dit is voor de andere klassen. `Counter` wordt gebruikt door de timerroutine (`do_timer`). Deze routine doet (sterk vereenvoudigd) het volgende

```
80:  jiffies++;
81:  current->counter--;
82:  if (current->counter==0)
83:      schedule();
```

Elke 10ms wordt `counter` dus verlaagd en de scheduler wordt opgeroepen indien `counter` negatief wordt (*preemptive multitasking*). Dit veld (ook wel tijdsquotum genoemd) bepaalt dus hoelang het proces ononderbroken mag lopen (in de afwezigheid van interrupts).

Het indelen van een proces in een bepaalde klasse en het opgeven van de prioriteitsvelden gebeurt door bibliotheekfuncties (en kan dus gebeuren door een applicatie zoals `nice` of `top`). Voor `SCHED_OTHER` mag de parameter tussen -19 en 19 liggen, voor de andere klassen tussen 1 en 99. Indien niets wordt opgegeven komt het proces in de `SCHED_OTHER`-klasse met parameter 0.

De code van de scheduler ziet er (lichtjes vereenvoudigd en herschreven voor monoprocessoren) als volgt uit:

```
84:  asmlinkage void schedule(void) {
85:
86:  need_resched_back:
87:      prev = current;
```

```
88:       this_cpu = prev->processor;
89:       if (prev->policy == SCHED_RR)
90:           goto move_rr_last;
91:   move_rr_back:
92:       switch (prev->state) {
93:           case TASK_INTERRUPTIBLE:
94:               if (signal_pending(prev)) {
95:                   prev->state = TASK_RUNNING;
96:                   break;
97:               }
98:           default:
99:               del_from_runqueue(prev);
100:          case TASK_RUNNING:;
101:      }
102:      prev->need_resched = 0;
103:  repeat_schedule:
104:      next = idle_task(this_cpu);
105:      c = -1000;
106:  still_running_back:
107:      list_for_each(tmp, &runqueue_head) {
108:          p = list_entry(tmp, struct task_struct, run_list);
109:          if (can_schedule(p, this_cpu)) {
110:              int weight = goodness(p, this_cpu, prev);
111:              if (weight > c)
112:                  c = weight, next = p;
113:          }
114:      }
115:      if (!c)
116:          goto recalculate;
117:      if (prev == next) {
118:          prev->policy &= ~SCHED_YIELD;
119:          goto same_process;
120:      }
121:      kstat.context_swtch++;
122:      prepare_to_switch();
123:      switch_mm(oldmm, mm, next, this_cpu);
124:      switch_to(prev, next, prev);
125:      __schedule_tail(prev);
126:  same_process:
127:      reacquire_kernel_lock(current);
128:      if (current->need_resched)
129:          goto need_resched_back;
130:
131:      return;
132:  recalculate:
133:      for_each_task(p)
134:        p->counter = NICE_TO_TICKS(p->nice);
135:      goto repeat_schedule;
136:  move_rr_last:
137:      if (!prev->counter) {
138:          prev->counter = NICE_TO_TICKS(prev->nice);
139:          move_last_runqueue(prev);
140:      }
141:      goto move_rr_back;
142:  }
```

Waarom maakt deze functie overvloedig gebruik van `goto`, iets wat normaal gezien zo veel mo-

gelijk vermeden wordt (*spaghetticode*)?

Zoals we merken maakt de scheduler gebruik van de `goodness()`-functie om het volgende proces te bepalen; het proces met de hoogste waarde wint. Deze waarde wordt als volgt bepaald:

```
143: static inline int goodness(struct task_struct * p,
144:                   int this_cpu, struct task_struct *prev){
145:     int weight;
146:
147:     weight = -1;
148:     if (p->policy & SCHED_YIELD)
149:         goto out;
150:
151:     if (p->policy == SCHED_OTHER) {
152:         weight = p->counter;
153:         if (!weight)
154:             goto out;
155:
156:         if (p->mm == this_mm || !p->mm)
157:             weight += 1;
158:         weight += 20 - p->nice;
159:         goto out;
160:     }
161:     weight = 1000 + p->rt_priority;
162: out:
163:     return weight;
164: }
```

Hierbij dient vermeld te worden dat `mm`=*memory map*: dit verwijst naar een structuur waar o.a. de paginatabellen (voor de vertaling van logische naar fysische adressen) worden bijgehouden.

Zoals we reeds vermeld hebben wordt een nieuw of gedeblokkeerd proces bovenaan de lijst geplaatst.

5.3 Opgave

De opgave bestaat uit twee delen. In het eerste deel zal de Linux-scheduler bestudeerd worden, terwijl tijdens het tweede deel alternatieve schedulers zullen geïmplementeerd worden.

Om het bestuderen/implementeren van schedulers wat eenvoudiger te maken is het aangewezen om over een aantal testapplicaties te kunnen beschikken. Hiervoor kan je gebruik maken van `load1`, `load2`, `load3`, `...`.Dit zijn identieke, CPU-gebonden applicaties die $\pm 20s$ rekenen. De applicatie is beschikbaar onder verschillende namen om ze van elkaar te kunnen onderscheiden indien je ze samen, maar met verschillende prioriteit, start.

Indien nodig kun je zelf een aantal bijkomende kleine applicaties schrijven, bv. om het gebruik van `sched_yield()` te testen.

Om de schedule-operaties te kunnen observeren werd de `schedule()`-functie lichtjes gewijzigd. Indien er een functie `scheduling()` bestaat wordt deze door `schedule()` opgeroepen, met drie argumenten: de `task_struct` van het vorige proces dat werd uitgevoerd, een getal waarvan de absolute waarde het aantal taken op de `run_queue` weergeeft en de `goodness` van de verkozen taak[1]. De module `scheduler.c` die je vindt in de `pract/scheduler`-directory op de PC maakt gebruik van deze mogelijkheid om een `scheduling()` functie te definiëren:

```
165: extern void (*scheduling)(struct task_struct *, signed, signed);
166:
167:
168: void show_info(struct task_struct * last, signed aantal, signed c){
```

[1] De naam van de verkozen taak is dus pas bekend bij de volgende oproep van `scheduling()`

```
169:          unsigned long long timestamp;
170:          unsigned diff;
171:          static unsigned long long lasttimestamp;
172:
173:          timestamp=cyclecounter();
174:          diff=timestamp-lasttimestamp;
175:
176:          if (last->pid)
177:                  printk("LAST: %4d [%8.8s], %10d cyc. [%5dms]"
178:                         " TOT: %2d NEW: %d\n", last->pid, last->comm,
179:                         diff, diff/450000, aantal, c);
180:
181:          lasttimestamp=timestamp;
182:  }
183:
184:  int init_module(void){
185:          scheduling   = show_info;
186:          return 0;
187:  }
188:
189:  void cleanup_module(void){
190:          scheduling   = NULL;
191:  }
```

De functie `show_info` wordt dus bij elke schedule-operatie opgeroepen en toont informatie over het *vorige* proces dat uitgevoerd werd. Deze module kan dus gebruikt worden om informatie te verzamelen. Aangezien `printk` de informatie niet enkel op scherm zet, maar ook naar een bestand wegschrijft (en naar het 5e tekstscherm) is het ten sterkste aangeraden om deze laatste optie af te zetten. Doe dit met het volgende commando: `killall klogd` (merk op dat dit commando na elke herboot opnieuw moet uitgevoerd worden!).

Na compileren (`gcc -c -O scheduler.c`) en laden (`insmod scheduler.o`) verschijnt er bij elke schedule-operatie een regel op het scherm.

Je kan bovendien gebruik maken van het commando `i` om de module te laden en van `r` om ze terug te verwijderen uit het geheugen (deze commando's moet je natuurlijk afsluiten met een `enter`!). De informatie zal namelijk voorbijvliegen over het scherm, wat het (blind) typen van `rmmod scheduler` er niet eenvoudiger op maakt.

Om de testprogramma's in een bepaalde klasse en met een bepaalde parameter te starten kan je gebruik maken van het `run`-commando dat je ook kan vinden in de `pract/scheduler`-directory. Dit commando verwacht drie parameters: de naam van het uit te voeren programma (het programma wordt gestart na 1 seconde), de klasse (0, 1 of 2 voor respectievelijk `SCHED_OTHER`, `SCHED_FIFO` en `SCHED_RR`) en de prioriteitsparameter van de klasse (tussen -19 en 19 (voor `SCHED_OTHER`) of tussen 1 en 99 (voor de andere klassen)).[2] Om bv. twee versies van `load` tesamen te starten, eentje in de `SCHED_RR`-klasse met parameter 20 en eentje in de `SCHED_OTHER` met parameter -10 terwijl men ondertussen de activiteiten van de scheduler bekijkt gebruiken we het volgende commando:

```
192:  run load1 2 20 & run load2 0 -10 & i
```

5.3.1 Analyse van de Linux-scheduler

In het eerste gedeelte van het practicum moet men de bestaande Linux-scheduler bestuderen. Doe dit door het bestuderen van de code van de scheduler en door het uitvoeren van volgende testen. Beschouw steeds de situatie waarbij er geen interrupts buiten de timer-interrupt optreden. Vergeet niet om `klogd` te stoppen m.b.v. `killall klogd`!

[2] Merk op dat `cmd` hetzelfde is als `run cmd 0 0 .`

Opgave: zorg dat de functie `show_info` de verlopen tijd niet enkel in processorcycli toont, maar ook in milliseconden (wijzig daartoe het argument van `printk` dat nu 450000 is). De klokfrequentie van de processor staat in `/proc/cpuinfo` en kun je bv. als volgt lezen: `cat /proc/cpuinfo`.

Vraag A: (1pt) voer `load1 & i` uit, wacht een paar seconden en voer dan `r` uit. Wat gebeurt er tussen het tikken van `r` en het moment waarop de uitvoer stopt?

Vraag B: (1pt) Bepaal de schedulevolgorde voor `load1` en `load2` na uitvoeren van `run load1 0 19 & run load2 0 -19 & i`. Welk proces zal het eerst eindigen?

Vraag C: (1pt) Idem, maar nu voor `run load1 0 0 & run load2 0 0 & i` (wat hetzelfde is als `load1 & load2 & i`). Wordt er fair gescheduled? Wat valt er op en hoe komt dit?

Vraag D: (1pt) rangschik de klassen + argument volgens prioriteit (enkel de hoogste en laagste prioriteit van elke klasse). Vb.:

$$\text{fifo}(1) < \text{rr}(99) < \text{other}(-19) < \text{fifo}(99) < \text{other}(19) < \text{rr}(1)$$

Dit zou bv. betekenen dat van twee identieke programma's, gestart op hetzelfde ogenblik, maar de ene met `SCHED_FIFO`-prioriteit van 1 en de andere met `SCHED_RR`-prioriteit van 99, het proces met prioriteit 99 eerst klaar is.

Vraag E: (1pt) Wat gebeurt er indien er interrupts optreden voor de drie klassen? Test dit door bv. de muis te bewegen; de veroorzaakte hardware-interrupt zal ervoor zorgen dat het muisproces (`gpm`) op de `run_queue` geplaatst worden.

5.3.2 Implementatie van andere schedulers

In het tweede deel worden een aantal alternatieve schedulers geïmplementeerd en vergeleken met elkaar en met de originele Linux-scheduler. Deze alternatieve schedulers maken geen gebruik van prioriteiten en worden daarom enkel vergeleken met `SCHED_OTHER` met prioriteit 0.

We kunnen alternatieve schedulers implementeren door het wijzigen van de `goodness()`-functie. Het adres van de alternatieve functie kunnen we via `extern_goodness()` doorgeven. De module bevat dan volgende code:

```
193: extern int (*extern_goodness)(struct task_struct *,
194:                int, struct task_struct *);
195:
196: int our_goodness(struct task_struct * p,
197:                  int cpu,
198:                  struct task_struct * prev){
199:     ...
200:     return ...;
201: }
202:
203: int init_module(void){
204:     scheduling   = show_info;
205:     extern_goodness = our_goodness;
206:     return 0;
207: }
208:
209: void cleanup_module(void){
210:     scheduling   = NULL;
211:     extern_goodness = NULL;
212: }
```

Vraag F: (5pt.) Gevraagd wordt volgende schedulers te implementeren (stijgende moeilijkheidsgraad maar allen implementeerbaar in een paar lijnen). Merk op dat we via de `goodness()`-functie niet enkel de keuze van het volgende proces kunnen beïnvloeden maar dat we ook het tijdsquotum van dit proces kunnen instellen (`p->counter=...`). Belangrijk: aangezien we de 'normale' Linux-scheduler niet meer gebruiken, hebben de verschillende klassen en prioriteiten geen betekenis meer. Het is dus niet meer nodig om de processen m.b.v. `run` te starten. Let op voor het speciale proces 0: zorg ervoor dat dit proces niet wordt uitgevoerd! Goodness moet steeds een getal groter dan 0 teruggeven (uitgezonderd voor proces 0).

1. **random**.
2. **LIFO**: voorrang voor nieuwe processen.
3. **FIFO**: voorrang voor oude processen.
4. **I/O-driven**: een herstart proces (bv. na het wachten op het verwerken van I/O) krijgt voorrang. Dit proces zal bovenaan de runqueue geplaatst worden.
5. **RR**: ROUND-ROBIN

Als voorbeeld volgt hier de code voor de randomscheduler:

```
213: int our_goodness(struct task_struct * p,
214:                  int cpu,
215:                  struct task_struct * prev){
216:     int prio=1;
217:     p->counter=20;            // stel lengte timeslice in
218:     if (p->pid==0) return 0;  // proces 0 willen we nooit schedulen
219:
220:     prio=random();
221:
222:     return prio;
223: }
```

Hierbij mag men veronderstellen (niet 100% correct) dat een nieuwer proces een hoger procesnummer heeft. Proces `a` is dus ouder (d.w.z. vroeger gestart) dan proces `b` indien `a->pid < b->pid`. Op Ufora vind je bij de documenten wat uitleg over de takenlijst in Linux; deze uitleg kan je helpen bij het implementeren van de ROUND-ROBIN-scheduler.

Doe volgende test voor deze 5 schedulers:

```
224: ./load1 & i
225: r
```

Hoe lang duurt het voordat de modules uit het geheugen geladen wordt (m.b.v het `r`-commando) en de uitvoer op het scherm dus stopt? Antwoord in de zin van 'nooit', 'onmiddellijk', 'na het beëindigen van load1', ... En waarom?

Besturingssystemen - AJ 2020-2021
Linux-practicum 1 (scheduler)

Persoon 1:

Persoon 2:

Antwoord A:

Antwoord B:

Antwoord C:

Antwoord D:

Antwoord E:

SCHED_OTHER:

SCHED_FIFO:

SCHED_RR:

Antwoord F:

type	wachtperiode waarom?
random	
LIFO	
FIFO	
I/O-driven	
RR	

Hoofdstuk 6

Practicum 4: device driver (Linux)

6.1 Doelstelling

Dit practicum heeft als doelstelling de student iets bij te leren over device drivers en de interactie ervan met de kernel en applicaties. Daartoe wordt een device driver voor de luidspreker van de PC geschreven en worden er verschillende testen uitgevoerd.

6.2 Inleiding

Net zoals in alle UNIX-versies worden hardware devices voorgesteld als 'speciale' bestanden (bv. `/dev/tty0`). Deze speciale bestanden worden aangemaakt met `mknod <file> <type> <major-nummer> <minor-nummer>`. De meest gebruikte types zijn 'c' voor karakter devices en 'b' voor blok devices. Alle communicatie tussen een applicatie en het device verloopt dan via de device driver, die de semantiek van de bestandsoperaties (`open(), close(),...`) voor het device implementeert. De functies die een device (driver) daartoe aanbiedt aan de kernel worden gedefinieerd in een lijst van wijzers naar functies. In de Linux-kernel wordt de lijst (voor karakter devices) als volgt gedefinieerd:

```
226: struct file_operations {
227:     loff_t (*llseek) (struct file *, loff_t, int);
228:     ssize_t (*read) (struct file *, char *, size_t, loff_t *);
229:     ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
230:     int (*readdir) (struct file *, void *, filldir_t);
231:     unsigned int (*poll) (struct file *, struct poll_table_struct *);
232:     int (*ioctl) (struct inode *, struct file *,
233:                   unsigned int, unsigned long);
234:     int (*mmap) (struct file *, struct vm_area_struct *);
235:     int (*open) (struct inode *, struct file *);
236:     int (*flush) (struct file *);
237:     int (*release) (struct inode *, struct file *);
238:     int (*fsync) (struct file *, struct dentry *);
239:     int (*fasync) (int, struct file *, int);
240:     int (*check_media_change) (kdev_t dev);
241:     int (*revalidate) (kdev_t dev);
242:     int (*lock) (struct file *, int, struct file_lock *);
243: };
```

Ondersteunt een device een bepaald type operatie niet, dan wordt NULL als wijzer gebruikt. De kernel zal dan de operatie afhandelen (meestal door het teruggeven van `-ENOSYS`, wat betekent dat het apparaat de operatie niet ondersteunt). Deze 15 functies zijn dus alle device-operaties die we kunnen uitvoeren. De belangrijkste zijn: `open, close, read` en `write` en `ioctl`. Deze laatste laat toe om

bepaalde instellingen van een I/O-apparaat in te stellen. Wensen we bv. (m.b.v. een geluidskaart) geluid op te nemen dan zullen we eerst via een `ioctl`-oproep de kaart configureren (mono/stereo, samplefrequentie, samplegrootte, ...) en nadien de gesampelde data lezen via `read`.

Een device driver maakt zichzelf bekend aan de kernel door het uitvoeren van `register_chrdev()`: bv.

```
244: register_chrdev(120, 'test', &test_operaties);
```

vertelt aan de kernel dat `test_operaties` de lijst van functies is die gebruikt moet worden voor het device met major-nummer 120, en dat het device luistert naar de naam *test*.[1] De lijst van device drivers die aanwezig zijn in de kernel kan bekeken worden in het bestand `/proc/devices`.

Het openen, door een applicatie, van een device verloopt dan bv. als volgt (figuur 6.1):

1. de applicatie gebruikt een functie uit de C-bibliotheek, en geeft o.a. de naam van het device als parameter mee: `fopen('/dev/test', ...);`
2. de functie uit de C-bibliotheek (statisch of dynamisch gelinkt) zal de systeemoproep *open* uitvoeren;
3. de kernel (die de systeemoproep afhandelt) vraagt aan het bestandssysteem informatie over het bestand;
4. het bestandssysteem antwoordt dat `/dev/test` een character device met major-nummer 120 en minor-nummer 0 is;
5. de kernel zoekt de tabel met de bestandsoperaties voor dit specifieke device.
6. de kernel roept de 8^e functie (`open()`) uit de tabel op, en geeft o.a. als argument door dat het minor-nummer 0 betreft.

Een speciale eigenschap van device drivers is dat de functionaliteit die ze aanbieden wordt doorgegeven m.b.v. een tabel met wijzers naar functies. Aangezien de functies m.b.v. deze tabel worden opgeroepen (d.m.v. hun positie in de tabel) en niet door een gewone functieoproep (d.m.v. hun naam), is het niet nodig om de kernel statisch te linken met de device driver. Dit maakt het dus mogelijk om device drivers dynamisch in de kernel te laden.

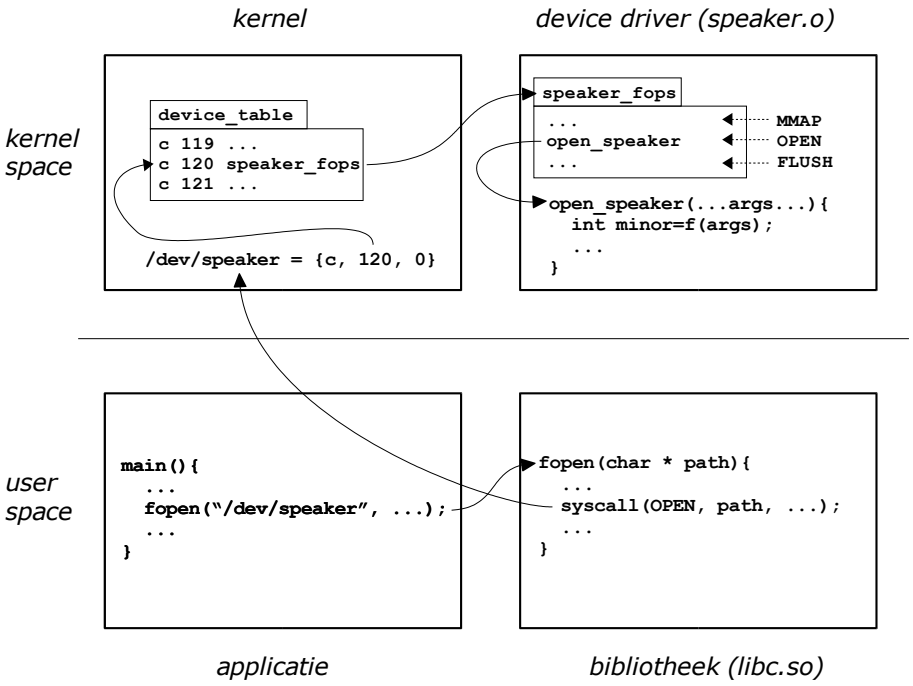
De kernel biedt de device drivers een aantal hulpfuncties aan. De belangrijkste zijn:

- de macros `MOD_INC_USE_COUNT` en `MOD_DEC_USE_COUNT` helpen de kernel bij het beheer van de modules. Deze twee macros verhogen en verlagen een teller die wordt bijgehouden in de kernel. Het verwijderen van de module (m.b.v. `rmmod`) is enkel mogelijk indien de teller op nul staat. De teller kan opgevraagd worden m.b.v. het `lsmod` programma of de `MOD_IN_USE` macro.
- `add_timer(struct timer_list * timer)` voegt een timer die werkt als een hardware-interrupt toe aan de kernel. Hierbij is `timer.function` de functie die op moment `timer.expires` moet uitgevoerd worden. De tijd wordt gemeten in *jiffies*: er zijn 100 jiffies per seconde. De huidige tijd wordt bijgehouden in de variabele `jiffies`. Vóór het eerste gebruik van de timer moet de timer geïnitialiseerd worden (`init_timer(struct timer_list * timer)`). Vb.:

  ```
  245:     struct timer_list timer;
  246:     init_timer(&timer);
  247:     timer.function = functie;
  248:     timer.expires  = jiffies + 100/2;
  249:     add_timer(&timer);
  ```

  zorgt ervoor dat de functie `functie()` na 1/2 seconde wordt uitgevoerd. Je kan een timer terug verwijderen m.b.v. `del_timer(struct timer_list * timer)`.

[1] Merk op dat de kernel geen weet heeft van minor-nummers. Het minor nummer wordt echter wel als argument doorgegeven aan de driver functies.



Figuur 6.1: Het oproepen van een device driver functie.

6.3 Opgave

OPGELET: voor dit practicum moet je een oor-, kop- of hoofdtelefoon meebrengen!
Tijdens het practicum starten we met het raamwerk van een device driver voor de luidspreker in een PC. Het device krijgt als naam `/dev/speaker` en 120 als *major number*. Bij het schrijven naar het device worden de gelezen bytes als volgt geïnterpreteerd:

c,d,e,f,g,a,b	De noten do, re, mi, fa, sol, la en si.
C,D,F,G,A	Idem, maar de gekruiste versies, t.t.z. c#, d#, f#, g# en a#.
1,2,4,8,6	Stelt de lengte van de volgende noten in (1/1, 1/2, 1/4, 1/8 of 1/16)
,	Een rust (even lang als de vorige noot).
.	De vorige noot wordt 50% langer gemaakt.
<	Ga naar een lager octaaf.
>	Ga naar een hoger octaaf.
_	Een marker.
^ gevolgd door een cijfer N	Spring N keer naar de laatste marker _

.

Initieel starten we met noten met lengte 1/1 en in het octaaf dat overeenkomt met de middelste toetsen op een piano (met A=440Hz). Bedoeling is dat er muziek uit de luidspreker komt als we noten naar het device sturen. Voorbeeld: `echo "8<<<_cCdDefFgGaAb>^7" > /dev/speaker` speelt de volledige toonladder. Merk op dat je aanhalingstekens moet gebruiken als je > of < in de af te spelen string gebruikt. Een aantal muziekstukjes vind je in de `pract/speaker`-directory, `cat jacob > /dev/speaker` of `cp jacob /dev/speaker` speelt bv. *Broeder Jacob*.

De device driver (ook te vinden in de `pract/speaker`-directory) bevat reeds heel wat code. Gevraagd wordt de code aan te vullen om tot een werkende device driver te komen. De code om de ingelezen karakterstroom te interpreteren en om de luidspreker te programmeren is reeds aanwezig. De functie `next_noot()` interpreteert de karakterstroom en bepaalt de volgende af te spelen noot: de waarde die teruggeven wordt is ofwel

- 0: alle noten zijn gespeeld
- 1: er volgt een rust
- > 1: de 'divisor' (`div`) van de af te spelen noot

Deze functie stelt ook drie globale variabelen in: `base` bepaalt het octaaf en `teller` en `noemer` bepalen de lengte van de volgende noten (de volgende noot moet dus gedurende $100 \times$teller/noemer jiffies afgespeeld worden). De verwerking van al deze gegevens gebeurt in `silence()` en `play_next_note()`.

Wat er nog moet gebeuren is het sturen van de juiste noot naar de hardware (dit gebeurt met de `play_note()`-oproep in de functie `play_next_note()`). Tussen twee noten pauzeren we gedurende een tijdsduur die $1/8^e$ van de lengte van de vorige noot bedraagt, dit moet gebeuren in de functie `silence()`. Tijdens het laden van de speaker-module wordt de functie `init_module()` uitgevoerd, en het is de bedoeling dat jij ervoor zorgt dat `play_next_note()` en `silence()` op het juiste moment worden opgeroepen. Doe dit m.b.v. timers: `silence()` zal de luidspreker uitschakelen en `play_next_note()` schedulen voor uitvoering m.b.v. een timer, terwijl `play_next_note()` de luidspreker inschakelt en `silence()` zal schedulen. Voor het afspelen van de eerste noot zal je `write_speaker()` moeten aanpassen: dit is nl. de functie die wordt opgeroepen als er naar `/dev/speaker` geschreven wordt.

Een belangrijke test voor de correcte werking van je device driver is:

```
cp bach /dev/speaker
rmmod speaker
```

Hierbij wordt het `rmmod`-commando uitgevoerd terwijl de muziek nog aan het spelen is. Wat kan hier het probleem zijn? Dit probleem kan je op twee manieren oplossen; implementeer ze (en zet één van de twee in commentaar in je code).

Ter herinnnering: compileren gebeurt met `cc -O -c speaker.c`, laden met `insmod speaker.o` en terug verwijderen uit het geheugen met `rmmod speaker`.

Op het einde van het practicum mail je het bestand `hdb.qcow2` naar michiel.ronsse@ugent.be met 'BS - device driver' als onderwerp. Vermeld de namen van de groepsleden bovenaan de code in commentaar en stop de virtuele machine voor je het bestand verstuurt!

Hoofdstuk 7

Practicum 5: bestandssysteem (Linux)

7.1 Doelstelling

Dit practicum heeft als doelstelling de student iets bij te leren over bestandssystemen en de interactie ervan met de kernel en applicaties. Daartoe wordt een bestandssysteem voor TAR-bestanden (TAR=*Tape ARchive*) ontwikkeld.

7.2 Inleiding

In het vorige practicum hebben we gezien hoe we de kernel d.m.v. een device driver een uniforme toegang kan bieden tot een hardware-apparaat. Het is dus mogelijk om data van elk apparaat te lezen door de `read()`-routine van de device driver aan te spreken. Om bv. de eerste 1024 bytes van de floppy te lezen kunnen we volgend commando uitvoeren:

250:
```
dd if=/dev/fd0 bs=1024 count=1
```

Het is duidelijk dat het gebruik van een opslagmedium als één grote verzameling van bytes niet eenvoudig is. Daarom zullen we normaal gezien een bestandssysteem op het opslagmedium gebruiken. Een bestandssysteem zorgt ervoor dat we op een gemakkelijke manier bestanden kunnen gebruiken op het opslagmedium. Daartoe zal het bestandssysteem een deel van het opslagmedium gebruiken voor zijn interne boekhouding (directory-informatie, FAT-tabellen, ...). In Linux koppelen we een opslagmedium (via zijn device-driver) aan een bepaalde (reeds bestaande) directory via het commando

251:
```
mount -t <type bestandssysteem> <device> <directory>
```

Hierbij vertellen we aan Linux het type bestandssysteem dat te vinden is op het device. Een MS-DOS floppy 'mounten' we bv. als volgt:

252:
```
mount -t fat /dev/fd0 /floppy
```

Wensen we dan `autoexec.bat` op de floppy te lezen (`cat /floppy/autoexec.bat`) dan zal deze leesopdracht door de kernel doorgegeven worden aan de code voor het FAT-bestandssysteem, die de data zal lezen via de device driver voor de floppy (die gekoppeld is aan het device `/dev/fd0`).

Het is belangrijk om op te merken dat er geen vaste koppeling is tussen een device en een bestandssysteem (zoals wel het geval is onder Windows: een floppy is steeds FAT, een CD steeds ISO9660, ...).

Het is niet noodzakelijk dat een bestandssysteem een opslagmedium gebruikt. Het /proc-bestandssysteem is bv. een volledig virtueel bestandssysteem: de bestanden (bv. `/proc/interrupts`) in deze directory worden on-the-fly (tijdens het lezen) gecreëerd.

7.2.1 Formaat van een TAR-bestand

Het TAR-bestandssformaat werd lang geleden ontwikkeld om backups te maken op tape (*Tape ARchive*), maar wordt nu nog steeds gebruikt, voornamelijk in de UNIX-wereld, maar dan in combinatie met een zipper (wat `.tar.gz`, `.tgz`, `.tar.bz2` of `.tbz` als extensies voor een TAR-bestand oplevert). Een TAR-bestand bestaat uit een aantal blokken van 512 bytes: per gearchiveerd bestand is er een *header* die één blok groot is, gevolgd door het eigenlijke bestand, dat een geheel aantal blokken in beslag neemt. Zowel de header als het laatste blok van een bestand zullen niet volledig gebruikt worden: het blok wordt dan aangevuld met binaire nullen. De header bevat de bestandsinformatie en heeft volgende velden:

lengte in bytes	veldnaam	betekenis
100	name	name of file
8	mode	file mode
8	uid	owner user ID
8	gid	owner group ID
12	size	length of file in bytes
12	mtime	modify time of file
8	chksum	checksum for header
1	typeflag	type of file
100	linkname	name of linked file
6	magic	USTAR indicator
2	version	USTAR version
32	uname	owner user name
32	gname	owner group name
8	devmajor	device major number
8	devminor	device minor number
155	prefix	prefix for file name

De headerinformatie wordt opgeslagen in afdrukbare ASCII om de porteerbaarheid te vergroten. Ook de getallen zijn dus in ASCII opgeslagen, met de bijkomende moeilijkheid dat ze in octale vorm zijn opgeslagen. Het einde van het TAR-bestand wordt aangeduid door een header die enkel uit nullen bestaat.

TAR-bestanden worden aangemaakt en uitgepakt m.b.v. het `tar`-commando. Bv.:

```
253: [root@localhost vmware-tools]# ll
254: total 48
255: -r-xr-xr-x    1 root     root        16916 Nov 22  2001 vmware-guestd
256: -rwxr-xr-x    1 root     root         5092 Nov 22  2001 dualconf.vm
257: -rwxr-xr-x    1 root     root         3178 Nov 22  2001 dualconf.org
258: -r-xr-xr-x    1 root     root         3620 Nov 22  2001 checkvm
259: -rw-r--r--    1 root     root         1384 Nov 22  2001 tools_log
260: drwxr-xr-x   31 root     root         4096 Nov 30 13:47 ..
261: drwxr-xr-x    2 root     root         4096 Nov 30 13:52 .
262: [root@localhost vmware-tools]# tar cf all.tar *
263: [root@localhost vmware-tools]# ll all.tar
264: -rw-r--r--    1 root     root        40960 Nov 30 13:52 all.tar
265: [root@localhost vmware-tools]# tar tvf all.tar
266: -r-xr-xr-x root/root      3620 2001-11-22 09:22:35 checkvm
```

```
267: -rwxr-xr-x root/root      3178 2001-11-22 09:22:35 dualconf.org
268: -rwxr-xr-x root/root      5092 2001-11-22 09:22:35 dualconf.vm
269: -rw-r--r-- root/root      1384 2001-11-22 09:22:39 tools_log
270: -rw-r--r-- root/root         0 2004-11-30 13:52:16 typescript
271: -r-xr-xr-x root/root     16916 2001-11-22 09:22:35 vmware-guestd
272: [root@localhost vmware-tools]# od -c -Ad -v all.tar |less
273: 0000000   c   h   e   c   k   v   m  \0  \0  \0  \0  \0  \0  \0  \0  \0
274: 0000016  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
275: 0000032  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
276: 0000048  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
277: 0000064  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
278: 0000080  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
279: 0000096  \0  \0  \0  \0   0   1   0   0   5   5   5  \0   0   0   0   0
280: 0000112   0   0   0  \0   0   0   0   0   0   0   0  \0   0   0   0   0
281: 0000128   0   0   0   7   0   4   4  \0   0   7   3   7   7   1   3   2
282: 0000144   7   1   3  \0   0   1   1   1   4   0  \0       0  \0  \0  \0
283: 0000160  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
284: 0000176  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
285: 0000192  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
286: 0000208  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
287: 0000224  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
288: 0000240  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
289: 0000256  \0   u   s   t   a   r          \0   r   o   o   t  \0  \0  \0
290: 0000272  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
291: 0000288  \0  \0  \0  \0  \0  \0  \0  \0  \0   r   o   o   t  \0  \0  \0
292: 0000304  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
293: 0000320  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
294: 0000336  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
295: 0000352  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
296: 0000368  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
297: 0000384  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
298: 0000400  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
299: 0000416  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
300: 0000432  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
301: 0000448  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
302: 0000464  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
303: 0000480  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
304: 0000496  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
305: 0000512 177   E   L   F 001 001 001  \0  \0  \0  \0  \0  \0  \0  \0  \0
306: 0000528 002  \0 003  \0 001  \0  \0  \0   Ì 204 004  \b   4  \0  \0  \0
```

In regel 262 maken we het TAR-bestand `all.tar` aan, en dit wordt 40960 bytes groot (=80 blokken), terwijl de bestanden samen maar 30190 bytes groot zijn (interne fragmentatie!). Het commando op regel 272 toont de binaire inhoud van het TAR-bestand. Bytes worden voorgesteld door hun ASCII-teken, waar dit niet mogelijk is wordt de waarde van de byte zelf getoond, voorafgegaan door een \. Het ASCII-teken "0"(binaire waarde 48) wordt dus voorgesteld door "0", terwijl de binaire "0"getoond wordt als "\0". Het TAR-bestand begint met de header voor het bestand `checkvm`: bytes 0-99 tonen bv. de naam van het bestand (aangevuld met binaire nullen) terwijl de lengte van het bestand te vinden is op bytes 124-135. Deze lengte wordt net als de andere getallen in de header voorgesteld door een nul-getermineerde string van de octale voorstelling: in dit geval moeten we de string "00000007044\0"dus interpreteren als het octale getal 7044, wat overeenkomt met 3620 in het decimale stelsel. Het eigenlijke bestand begint in het volgende blok dat begint op adres 512.

Bedoeling van dit practicum is om een driver te maken die het mogelijk maakt om een TAR-bestand te 'mounten': het zal dus mogelijk zijn om een TAR-bestand te benaderen alsof het een directory is. I.p.v. het `tar`-commando te gebruiken om bestanden te lezen kunnen we dan op de gewone manier de bestanden lezen. Vb.:

```
[root@localhost tmp]# ll
total 48
drwxr-xr-x   17 root     root         4096 Nov 29 18:54 ..
-rw-r--r--    1 root     root        40960 Nov 30 13:52 all.tar
drwxrwxrwt    2 root     root         4096 Nov 30 17:07 .
[root@localhost tmp]# insmod tarfs.o
[root@localhost tmp]# mount -o loop -t tarfs all.tar /mnt/tar
[root@localhost tmp]# ll /mnt/tar
mtotal 4
drwxr-xr-x    5 root     root         4096 Nov 29 18:55 ..
dr-xr-xr-x    1 500      500             1 Nov 30 17:07 .
-r--r--r--    1 500      500         16916 Nov 30 17:07 vmware-guestd
-r--r--r--    1 500      500             0 Nov 30 17:07 typescript
-r--r--r--    1 500      500          1384 Nov 30 17:07 tools_log
-r--r--r--    1 500      500          5092 Nov 30 17:07 mdualconf.vm
-r--r--r--    1 500      500          3178 Nov 30 17:07 mdualconf.org
```

Het bestandssysteem dat we zullen ontwikkelen zal zeer eenvoudig zijn: we ondersteunen TAR-bestanden met maximaal 100 *normale* bestanden, dus geen directories, pijpen, links, enz. De directories '.' en '..' die steeds aanwezig moeten zijn zullen we moeten emuleren daar ze niet noodzakelijk in een TAR-bestand aanwezig zijn. Bovendien moeten we de bestanden enkel kunnen lezen.

7.2.2 Werking van de Linux-bestandssystemen.

Linux groepeert alle bestandssystemen[1] via een 'Virtual File System' (VFS, zie figuur 7.1). Dit VFS vangt de opdrachten van processen op en roept, na vertaling van de argumenten, de eigenlijke procedure (in één van de echte bestandssystemen) op. Indien we bv.

```
cat /mnt/tar/tools_log
```

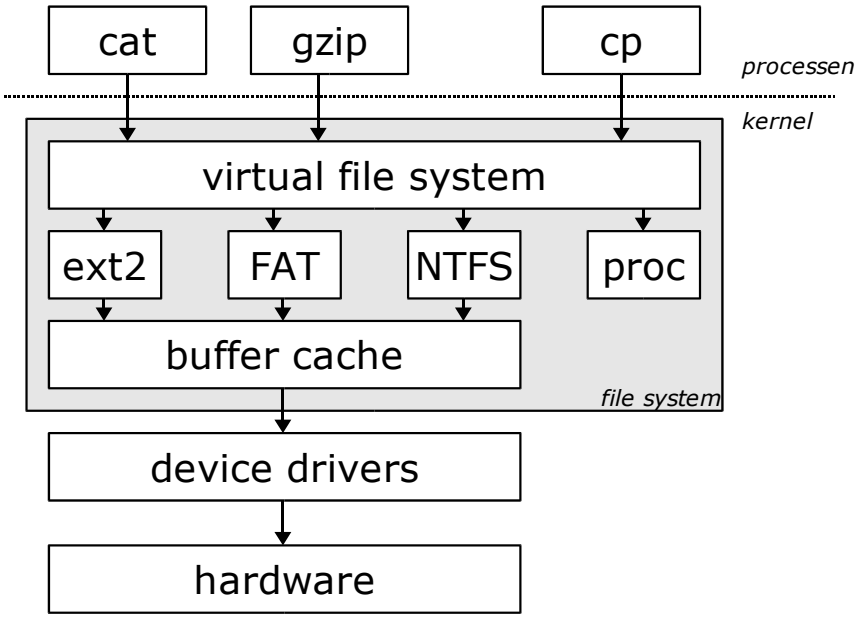
uitvoeren, zal het VFS het bestandssysteem bepalen dat dit bestand bevat en zal het de aanvraag om het bestand `tools_log` te openen doorgeven (dus zonder de mount-locatie!) aan het bestandssysteem.

We kunnen een nieuw type bestandssysteem als volgt registreren in het VFS:

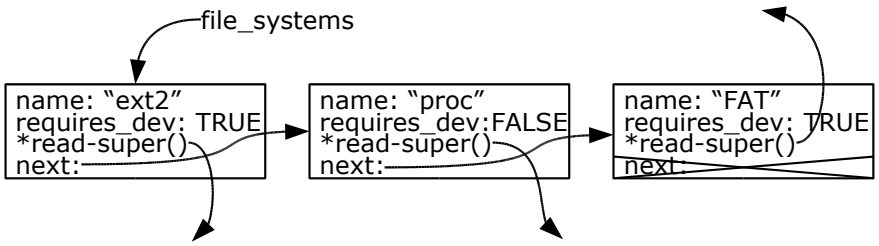
```
static struct file_system_type ext2_fs_type = {
  "ext2",
  FS_REQUIRES_DEV,
  ext2fs_mount,
  NULL
};

register_filesystem(&ext2_fs_type);
```

Hierbij geven we aan het VFS volgende zaken door: de naam van het bestandssysteem, of het bestandssysteem al dan niet een device nodig heeft (is bv. niet nodig voor `/proc`), de procedure om een nieuw bestandssysteem van dit type te mounten, en een lege wijzer (wordt door VFS ingevuld en wijst naar het volgende type bestandssysteem). Een VFS met drie types bestandssystemen ziet er dus uit zoals weergegeven in figuur 7.2. Voeren we bv. het commando

[1] Met bestandssysteem bedoelen we zowel de data zoals die te vinden is op een opslagmedium als de code die in de kernel aanwezig is om deze data te interpreteren. De context zal steeds duidelijk maken wat we in feite bedoelen.



Figuur 7.1: De gelaagde structuur van het VFS.



Figuur 7.2: De manier waarop drivers voor bestandssystemen gelinkt zijn.

```
mount -t ext2 /dev/hda2 /home
```

uit, dan zal het VFS de lijst van bestandssystemen aflopen tot de juiste naam (ext2) gevonden wordt, en de corresponderende `read_super()` uitvoeren. Deze functie moet dan zorgen voor het eigenlijke mounten van de schijf (o.a. testen of het device het juiste type bestandssysteem bevat en dit bestandssysteem consistent is). Voor elk gemount bestandssysteem wordt er door de kernel informatie in een 'superblock' bijgehouden. Bij het mounten vullen we deze informatie aan met een lijst van superblock-operaties:

```
struct super_operations {
   void (*read_inode) (struct inode *);
   void (*write_inode) (struct inode *);
   void (*put_inode) (struct inode *);
   void (*delete_inode) (struct inode *);
   int  (*notify_change) (struct dentry *, struct iattr *);
   void (*put_super) (struct super_block *);
   void (*write_super) (struct super_block *);
   int  (*statfs) (struct super_block *, struct statfs *, int);
   int  (*remount_fs) (struct super_block *, int *, char *);
   void (*clear_inode) (struct inode *);
   void (*umount_begin) (struct super_block *);
};
```

De belangrijkste functies zijn:

read_inode die zorgt voor het opvragen van informatie over een bestand (inode genaamd in de kernel). Elk bestand van het bestandssysteem heeft een uniek inodenummer en een inodestructuur. Deze functie moet zorgen voor het invullen van deze structuur. Deze structuur bevat onder andere informatie over de eigenaar, grootte, mode, creatietijdstip, enz. van het bestand. Het belangrijkste is echter een wijzer naar een lijst van inode-operaties (zie verder).

write_inode die informatie over een inode wegschrijft.

put_super die een bestandssysteem unmount.

statfs die informatie over het totale bestandssysteem teruggeeft (o.a. het aantal bestanden en de totale grootte).

Aangezien we de bestanden in een TAR-bestand enkel willen kunnen lezen gebruiken wij enkel de functies `read_inode`, `put_super` en `statfs`.

Zoals gezegd kent de functie `read_inode` een lijst van inode-operaties toe aan elk bestand (of inode). Deze operaties zijn:

```
struct inode_operations {
  struct file_operations * default_file_ops;
  int (*create) (struct inode *,struct dentry *,int);
  int (*lookup) (struct inode *,struct dentry *);
  int (*link) (struct dentry *,struct inode *,struct dentry *);
  int (*unlink) (struct inode *,struct dentry *);
  int (*symlink) (struct inode *,struct dentry *,const char *);
  int (*mkdir) (struct inode *,struct dentry *,int);
  int (*rmdir) (struct inode *,struct dentry *);
  int (*mknod) (struct inode *,struct dentry *,int,int);
  int (*rename) (struct inode *, struct dentry *
                 struct inode *, struct dentry *);
  int (*readlink) (struct dentry *, char *,int);
  struct dentry * (*follow_link) (struct dentry *, struct dentry *,
```

```
360:                                    unsigned int);
361:    int (*readpage) (struct file *, struct page *);
362:    int (*writepage) (struct file *, struct page *);
363:    int (*bmap) (struct inode *,int);
364:    void (*truncate) (struct inode *);
365:    int (*permission) (struct inode *, int);
366:    int (*smap) (struct inode *,int);
367:    int (*updatepage) (struct file *, struct page *,
368:                       unsigned long, unsigned int, int);
369:    int (*revalidate) (struct dentry *);
370:  };
```

met `default_file_ops` een wijzer naar

```
371:  struct file_operations {
372:    loff_t (*llseek) (struct file *, loff_t, int);
373:    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
374:    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
375:    int (*readdir) (struct file *, void *, filldir_t);
376:    unsigned int (*poll) (struct file *, struct poll_table_struct *);
377:    int (*ioctl) (struct inode *, struct file *, unsigned int,
378:                  unsigned long);
379:    int (*mmap) (struct file *, struct vm_area_struct *);
380:    int (*open) (struct inode *, struct file *);
381:    int (*flush) (struct file *);
382:    int (*release) (struct inode *, struct file *);
383:    int (*fsync) (struct file *, struct dentry *);
384:    int (*fasync) (int, struct file *, int);
385:    int (*check_media_change) (kdev_t dev);
386:    int (*revalidate) (kdev_t dev);
387:    int (*lock) (struct file *, int, struct file_lock *);
388:  };
```

Het toekennen van deze operaties gebeurt dus door de `read_inode` operatie en deze wordt normaal gezien door het VFS uitgevoerd bij het openen van een bestand. We kunnen dus tijdens het openen van een bestand kiezen welke operaties we zullen gebruiken om bewerkingen op dit bestand uit te voeren. Normaal gezien volgt de keuze van operaties uit het type bestand (gewoon bestand, directory, link, pijp, ...). Ons bestandssysteem kent echter slechts twee types bestanden: directories ('.' en '..') en gewone bestanden (maximaal 100). Voor de directories implementeren we enkel `lookup` (krijgt als argument een bestandsnaam en geeft een inodenummer terug) en `readdir` (geeft alle bestandsnamen van een directory terug). Voor de bestanden moeten we enkel `file_read` (lezen van een bestand) implementeren. Wij zullen inodenummer 0 gebruiken voor '.', nummer 1 voor '..' en volgende nummers voor de eigenlijke bestanden.

Onze driver kan het TAR-bestand enkel als *block device* (in dit geval met blokken van 512 bytes) benaderen en de kernel kent daar één functie voor: met

```
389:      struct buffer_head * bh = bread(device, bloknummer, blokgrootte);
390:      char * data = bh->data;
```

vragen we het blok `bloknummer` met grootte `blokgrootte` aan de device driver van `device`. We krijgen een structuur terug waarvan enkel het `data`-veld voor ons van belang is; dit zal een wijzer zijn naar de ingelezen data. Met `brelse(bh)` geven we een `bh` terug vrij (op dat moment wordt de `data`-wijzer natuurlijk ongeldig!).

7.3 Opgave

Het is de bedoeling om een werkende driver voor een TAR-bestandssysteem te implementeren. Daartoe kan je starten van de gedeeltelijke implementatie `tarfs.c` die je vind in de `pract/tarfs`-directory.

Er ontbreken nog twee belangrijke onderdelen in deze implementatie:

- Het inlezen van de directory-informatie die aanwezig is in het TAR-bestand.
- Het eigenlijke lezen van data uit bestanden.

7.3.1 Inlezen van directory-informatie

Het inlezen van de informatie over de bestanden die aanwezig zijn in een TAR-bestand gebeurt tijdens het mounten (in de functie `tar_mount`). Bedoeling is dat het TAR-bestand sequentieel gescand wordt en dat er voor elk gevonden bestand op zijn minst de naam, het startadres en de lengte van het bestand wordt bijgehouden. U beschikt over de hulpfunctie

```
391:    long octal2long(char *)
```

om een octale string om te zetten in een getal.

Laat u leiden door het in de code aanwezige commentaar. Je moet enkel code toevoegen tussen `/* BEGIN VAN UW CODE */` en `/* EINDE VAN UW CODE */` (staat tweemaal in de code). Om te testen maak je TAR-bestanden aan, en vergeet ook de speciale gevallen niet (bv. een leeg bestand opnemen in een TAR-bestand. Een leeg bestand maak je als volgt aan: `touch leeg`).

Als alles correct geïmplementeerd is kun je een TAR-bestand mounten op een nog aan te maken *mount point* (bv. `mkdir /mnt/tar`) en moet je de directory-informatie kunnen lezen. Het eigenlijke lezen van een bestand zal echter nog niet lukken.

7.3.2 Lezen van bestanden

Om bestanden te kunnen lezen moet er één functie aangepast worden: `copy_from_tar`. De moeilijkheid hierbij is dat de argumenten hierbij aangeven welke bytes we willen lezen (nl. van `start` tot `stop`) en we enkel blokken van 512 bytes kunnen lezen uit het TAR-bestand.

Deze blokken van 512 bytes zullen we terug lezen m.b.v. een `bread()`-oproep, en om (delen van) deze data te kopiëren naar de applicatie moeten we gebruik maken van `copy_to_user(user_address, kernel_address, length)`. Hierbij is `user_address` het adres in de (datasectie van de) applicatie waarnaar er `length` bytes vanaf `kernel_address` moeten gekopieerd worden.

Ter herinnnering: compileren gebeurt met `cc -O -c tarfs.c`, laden met `insmod tarfs.o`, terug verwijderen uit het geheugen met `rmmod tarfs`, mounten van een TAR-bestand met `mount -o loop -t tarfs test.tar /mnt/tar` en unmounten van een TAR-bestand met `umount /mnt/tar`.

Op het einde van het practicum mail je het bestand `hdb.qcow2` naar michiel.ronsse@ugent.be met 'BS - FS' als onderwerp. Vermeld de namen van de groepsleden bovenaan de code in commentaar en stop de virtuele machine voor je het bestand verstuurt!

Appendix A: Unix-commando's

Deze appendix bevat een zeer korte beschrijving van de commando's die je nodig zal hebben. Voor meer informatie verwijzen we naar de zogenaamde man-pages, opvraagbaar met het `man` commando (bv. `man bash`).

bash (Bourne-Again SHell) is de shell die gebruikt wordt om commando's door te geven aan de kernel. De shell interpreteert de commandolijn, breekt die eventueel in stukken (bv. bij pipes: `sort file | uniq | lp`), voert eventuele redirecties uit (bv. `sortfile > file2`), en zorgt er dan voor dat de programma's gestart worden. Een programma krijgt drie open bestanden van de shell: standaard-invoer (0), standaard-uitvoer (1) en error-uitvoer (2). M.b.v. het schieidingsteken ";" kunnen we een aantal programma's na elkaar starten (`prog1 ; prog2 ; prog2`) terwijl we "&" gebruiken om een aantal programma's parallel te starten (`prog1 & prog2 & prog3`).

echo *string* stuurt *string* naar standaard uitvoer (meestal het scherm). Door omleiding van de uitvoer kunnen we naar een bestand schrijven: `echo hallo > a`.

cat concateneert verschillende bestanden en print de uitvoer naar het scherm (vb. `cat a b c`). Door het omleiden van de uitvoer kunnen we aldus bestanden kopiëren: `cat a > b` is equivalent met `cp a b`. Worden er geen argumenten opgegeven, dan wordt er gelezen van/naar standaardinvoer (m.a.w. in de pijplijn `... | cat | ...` kopieert `cat` zijn standaardinvoer naar standaarduitvoer).

dd is een 'data duplicator': `dd if=file1 of=file2 bs=1024 count=2 skip=16` leest 2 blokken van 1024 bytes beginnende vanaf blok 17 van `file1` en schrijft die naar `file2`. We mogen `if` en/of `of` weglaten om te lezen/schrijven van standaard invoer/uitvoer.

strace wordt gebruikt om de systeemoproepen en signalen van een programma te tonen. `strace echo Hallo` bv. toont de systeemoproepen die uitgevoerd worden door `echo Hallo`. De uitvoer bestaat uit een aantal lijnen (één lijn per systeemoproep) en bestaat uit een linker- en rechtergedeelte (gescheiden door het =-teken). Een lijn als

```
open("/lib/libc.so.6", O_RDONLY)        = 3
```

betekent bv. dat de applicatie het bestand `/lib/libc.so.6` geopend heeft en dat dit bestandsnummer 3 is.

Merk op dat dit programma zich op de grens tussen user en kernel space plaatst en de overgangen van user naar kernel space toont.

insmod, rmmod en lsmod worden gebruikt om modules te manipuleren. Een module wordt geladen met `insmod` *`module`* en terugverwijderd met `rmmod` *`module`*. De lijst van geladen modules wordt opgevraagd met `lsmod`.

sleep `time` wordt gebruikt om een proces gedurende *time* seconden te laten slapen.

mknod gebruiken we om devices aan te maken.

mkdir gebruiken we om een directory aan te maken.

ls staat voor 'list' en toont de inhoud van een directory. Gebruik `ls -al` om meer informatie over alle bestanden in een directory te krijgen. Gebruik `cd` om van directory te wisselen.

gcc is de GNU C-compiler.

/proc is een virtuele directory die allerlei systeeminformatie bevat. De virtuele bestanden `devices`, `ioports`, `interrupts` en `filesystems` geven bv. respectievelijk informatie over de device drivers, poorten, interrupts en bestandssystemen die op dat moment door de kernel gekend of in gebruik zijn.

mount gebruiken we om een bestandssysteem te 'mounten' op een bepaalde plaats. Vanaf dat ogenblik kunnen de bestanden van het bestandssysteem gebruikt worden. Bv. `mount -t fat /dev/fd0 /mnt/floppy` zorgt ervoor dat de data die zich bevindt op de eerste diskette (`/dev/fd0`, beter bekend als `A:` onder Windows) geïnterpreteerd wordt als een FAT-bestandssysteem (t.t.z. het normale Windows-bestandssysteem) en dat alle bestanden op de diskette bereikbaar zijn onder de directory `/mnt/floppy`. Dit zeer belangrijke commando zorgt dus voor de link tussen een device driver (`/dev/fd0`) en het bestandssysteem (FAT) dat het bevat.

time kunnen we voor een commando plaatsen om de uitvoeringstijd ervan te bepalen.

pico is een simpele teksteditor.

Bibliografie

[BBD+96] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, 1996.

[BC00] Daniel Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly, October 2000.

[Bow98] Ivan Bowman. Conceptual architecture of the linux kernel. http://www.grad.math.uwaterloo.ca/~itbowman/CS746G/a1/, January 1998.

[Bru00] Herman Bruyninckx. Real time and embedded howto. Technical report, K.U.Leuven, August, 2000.

[BST98] Ivan Bowman, Saheem Siddiqi, and Meyer Tanuan. Concrete architecture of the linux kernel. http://plg.uwaterloo.ca/~itbowman/CS746G/a2/, February 1998.

[CDM98] Rémy Card, Éric Dumas, and Franck Mével. *The Linux Kernel Book*. John Wiley & Sons, 1998.

[DB98] Koen De Bosschere. *Inleiding tot de Besturingssystemen*. ELIS, 1998.

[ET88] Janet Egan and T. Teixeira. *Writing a Unix Device Driver*. John Wiley & Sons, 1988.

[Goo99] Richard Gooch. Overview of the virtual file system. http://www.atnf.csiro.au/~rgooch/linux/docs/vfs.txt, July 1999.

[Joh95] Michael Johnson. Writing linux device drivers. In *Spring DECUS'95*, Washington, D.C., 1995.

[lin] The linux documentation project. http://www.linuxdoc.org/.

[RA00] Russ Ross and Deborah Abel. An evaluation of the linux scheduler for single user workstations. http://www.people.fas.harvard.edu/~ross/cs261/paper.html, January 2000.

[Rub98] Alessandro Rubini. *Linux Device Drivers*. O'Reilly, February 1998.