De Linux kernel – een introductie

Bart De Schuymer
bdschuym@zeus.rug.ac.be

ZeusWPI
info@zeus.rug.ac.be
http://www.zeus.rug.ac.be

Licentie: GNU Free Documentation Licence
(zie *http://www.gnu.org/licenses/fdl.txt*)

Maart 2003

Inleiding

Dit document wenst enig licht te werpen op de werking van de Linuxkernel en de *Linux community*. Het is niet de bedoeling om het gebruik van het besturingssysteem naderbij te bekijken. De werking en implementatie van sommige aspecten van het besturingssysteem zullen echter wel worden besproken. Dit document schuwt de Engelstalige termen niet als er geen mooie Nederlandse vertalingen voor bestaan, of als het beter klinkt in het Engels.

De tekst is af en toe nogal technisch doordat codefragmenten uit de kernel worden besproken. Degenen die hiervoor geen interesse hebben, zouden normaal gezien toch nog iets aan dit document moeten hebben. Het is niet de bedoeling om veel te zeggen over besturingssystemen in het algemeen, daarvoor bestaan prachtige boeken waarvan er enkele in de referenties zijn terug te vinden.

De bedoeling van deze tekst is enerzijds om een kennismaking met de implementatie van het Linux-besturingssysteem te bieden, die hopelijk bepaalde inzichten bijbrengt. Anderzijds is de tekst bedoeld als wegwijzer voor de door de Linuxkernel geïntrigeerden die de kernel verder wensen te exploreren.

De Linuxkernelreeks die we zullen behandelen is de `2.5`-kernel, die de meest recente kernelreeks is. Wanneer we naar bestanden uit deze distributie verwijzen, zullen we de prefix */usr/src/linux* weglaten. We wensen ook op te merken dat deze tekst eigenlijk een "*working document*" is, hij is dus nog niet af. Alhoewel Linux beschikbaar is voor verschillende processorfamilies, zullen we ons in dit document beperken tot de Intel x86-familie van processoren.

We wensen vooraf duidelijk te stellen dat we de schijn van het bezit van

een grondige persoonlijke kennis over alle onderwerpen die in dit document aan bod komen, niet willen wekken.

We wensen verder op te merken dat via ZeusWPI [1] een cursus over de programmeertaal C kan verkregen worden. Een cursus (in wording) over het algemene gebruik van Linux is ook beschikbaar [2].

Maart, 2003.

Inhoudsopgave

1 Algemeen **1**
1.1 Ontstaan van Linux 1
1.2 Naamgeving en logo 2
1.3 GNU en GPL 2
1.4 De Linuxkernel 4
1.4.1 Nummering van kernels 5
1.4.2 Soms loopt het mis 6
1.4.3 Monolithische en microkernels 7
1.4.4 Het compileren van een kernel 8

2 Begrippen **11**
2.1 Hardware en software 11
2.1.1 Hardware 12
2.1.2 Software 14

3 De kernel codingstyle **16**
3.1 Het gebruik van accolades 17
3.2 Het `goto`-statement 18
3.3 Indentatie en alignering 21

4 SMP en locking **23**
4.1 SMP 23
4.2 Locking 24
4.2.1 Algemeen 24

4.2.2 De functie spin_trylock() . . . 25
4.2.3 Soorten locks . . . 27
4.2.4 Locking bij processen die kunnen slapen . . . 28
4.2.5 Interrupts en locking . . . 28

5 Interrupts **32**
5.1 Hardware interrupts . . . 32
5.1.1 Codepad naar do_IRQ() . . . 33
5.1.2 Initialisatie van de IDT . . . 34
5.1.3 Interruptafhandeling . . . 40
5.2 Software interrupts, system calls . . . 43
5.3 Bottom halves: tasklets en softirq's . . . 46

6 Initialisatie van de kernel **50**

7 Modules **54**
7.1 Het gebruik van modules . . . 54
7.2 Een module laden . . . 56
7.3 Werking van de module code . . . 58

8 Processen **60**
8.1 De zandbak . . . 61
8.2 Uitzicht van een programma . . . 61
8.3 Usermode-kernelmode overgang . . . 62
8.4 Virtueel geheugen, paginering en segmentatie . . . 63
8.4.1 De GDT en LDT . . . 65
8.4.2 Linux en paginering/segmentatie . . . 65
8.4.3 Uitzicht van het geheugen van een proces . . . 68
8.4.4 Memory Regions . . . 69
8.5 Bijhouden van de proceseigenschappen . . . 70
8.5.1 Locatie van de proceseigenschappen . . . 70
8.5.2 De struct task_struct . . . 73
8.5.3 Allocatie van de struct task_struct's . . . 73

8.6 Scheduling 74

9 Nawoord **76**

Hoofdstuk 1

Algemeen

1.1 Ontstaan van Linux

Linux is ontstaan in 1991 als *pet project* van de Fin Linus B. Torvalds . Hij was toen een 21-jarige “tweedekanner” informatica aan de universiteit van Helsinki. In die tijd was er een beroemd besturingssysteem Minix beschikbaar, geschreven door de Nederlandse professor Andrew Tanenbaum[4]. Dit was geschreven om informaticastudenten de werking van besturingssystemen te helpen doorgronden. De broncode van Minix was nl. beschikbaar voor iedereen. Linus had dit besturingssysteem ook, maar het voldeed niet volledig aan zijn wensen en het was ook niet gratis. Hij had zich ook net een Intel 80386 gekocht, een toen als krachtig beschouwde computer, en wilde teweten komen hoe deze werkte. Dus begon hij met het schrijven van een programma dat uit twee processen bestond, het ene proces schreef `A` op het scherm, het tweede `B`. Van in het begin deed Linux dus al aan multitasking. Dit programma begon te werken van zodra de computer werd opgestart, zonder dat een ander besturingssysteem draaide. De resultaten van zijn creaties maakte Linus via het Internet beschikbaar voor iedereen. Het Internet heeft een belangrijke rol gespeeld bij de ontwikkeling van Linux. Beeld je bvb. een ontwikkelaar in die Linux installeert op haar machine en een fout vindt. Zij verbetert deze fout en zendt de code naar Linus. Enkele dagen later kan de

aangepaste nieuwe kernel worden verdeeld via het Internet.

Een meer enthousiaste en complete beschrijving van het ontstaan van Linux kan o.a. gevonden worden op [3]. Daar vind je ook de beroemde eerste mail die Linus verstuurde over zijn creatie.

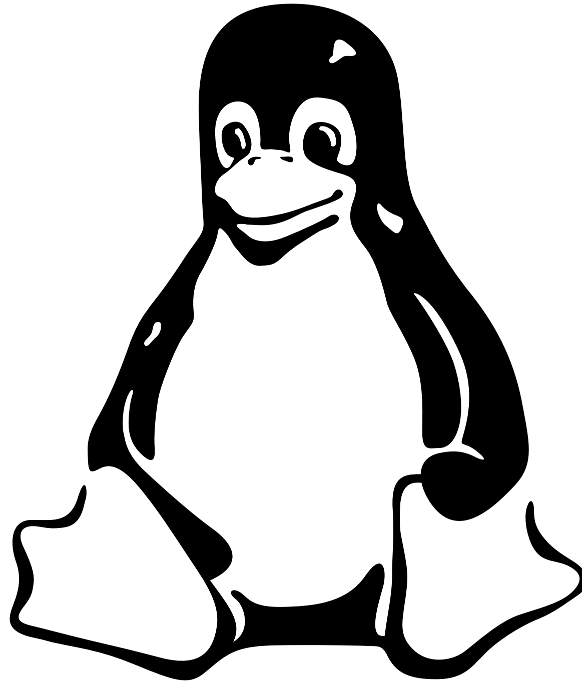
1.2 Naamgeving en logo

Het is niet moeilijk de afkomst van de naam Linux te raden: het is de samentrekking van *Linus' Unix*. Oorspronkelijk was dit maar een binnenpretje van Linus en was het niet de bedoeling het zo te noemen. De man die Linus ftp-ruimte gaf om het besturingssysteem op het Internet te verdelen, zette het bestand echter onder een subdirectory Linux.

Op een bepaald moment werd er een vraag gesteld op de Linuxkernel mailinglijst (lkml) of Linux geen nood had aan een logo. Linus kwam prompt af met het idee van een pinguïn. Hij was nl. ooit in een zoo gebeten door een pinguïn en sindsdien had hij een rare voorliefde voor die beestjes. De pinguïn moest er voldaan uitzien: alsof hij net massa's lekkere haring gegeten had of net succes had gehad bij een pinguïn-vrouwtje, zie figuur 1.1. De naam van de pinguïn is Tux, deze keer de samentrekking van *Torvalds' Unix*. Meer logo's kan je vinden op bv. [5].

1.3 GNU en GPL

Het GNU project (een recursieve arkorting die staat voor GNU*'s Not Unix*), uit te spreken als gnoe (wat ook het logo is van GNU), door Richard Stallman (RMS) opgericht, streeft naar het creëren van een volledig vrij besturingssyteem [6]. Hiervoor ontwikkelde RMS o.a. de GNU compiler gcc en het tekstverwerkingsprogramma emacs. Als besturingssysteem wil het GNU project ooit de Hurd uitbrengen. Dit besturingssysteem is echter nog altijd in ontwikkeling. De ontwikkeling van Linux, die later gestart is, heeft veel sneller vooruitgang geboekt.



Figuur 1.1: Tux, de Linux pinguïn

Tussen RMS en de *Linux kernel community* gaat het niet altijd even goed. Aangezien een systeem dat de Linuxkernel gebruikt, gebouwd is met en bestaat uit veel GNU programma's, vinden RMS en z'n aanhangers dat Linux eigenlijk GNU/Linux zou moeten heten. De overgrote meerderheid van de Linux *kernel hackers* vindt alleszins dat het niet nodig is om de Linuxkernel zelf om te dopen tot de GNU/Linux kernel. Het feit dat de Linuxkernel gecompileerd wordt met gcc rechtvaardigt die hernoeming niet, dan zouden we GNU als prefix moeten gebruiken in de naam van elk programma dat ermee gecompileerd wordt. De GNU *advocates* lijken het hiermee eens te zijn. Of een systeem dat op de Linuxkernel draait een GNU/Linux systeem moet worden genoemd, is irrelevant voor het onderwerp van dit document. We eindigen met op te merken dat enkel de Debian distributie de term GNU/Linux hanteert.

Ondanks de strubbelingen bestaat er geen twijfel dat het GNU project zeer belangrijk is voor Linux. Naast het gebruik van GNU programma's, wordt de

Linux kernel ook gedistribueerd onder een door de GNU-organisatie bedacht *copyright*: de GPL (GNU *General Public Licence*). Deze licentie houdt in dat de Linuxkernel door iedereen gratis kan worden bekomen en de broncode door iedereen kan worden bekeken. Elke verbetering aangebracht aan de broncode die niet voor strikt persoonlijk gebruik bedoeld is, moet terug publiek gemaakt worden. Daarom dat de Linuxdistributies (zoals RedHat en United Linux) altijd gratis af te halen zijn van het Internet. Deze bedrijven trachten winst te maken door het verkopen van hun expertise via service na verkoop, of via propriëtaire programma's die draaien onder Linux.

De GPL maakt bedrijven zoals Microsoft (a.k.a. *MegaSoft*, etc.) ongerust. Ze hebben geen probleem met licenties zoals deze waaronder FreeBSD wordt uitgebracht, aangezien deze licentie MS toelaat code te kopiëren in hun eigen besturingssysteem. Zo vindt de IP stack van Windows haar oorsprong in de IP stack van FreeBSD (De Linux IP stack ook, trouwens). Aangezien de GPL echter eist dat alle derivaten van de Linuxkernel terug onder de GPL moeten worden uitgebracht, kan MS deze code niet gebruiken. Anders moet alle code, of toch een deel ervan, van het Windows-besturingssysteem worden onthuld.

Nog een woordje uitleg over *free software*. Met *free* wordt hier bedoeld: "*free as in free speech, not free as in free beer*". Het is dus niet verboden geld te verdienen met *free software*, alhoewel velen die dit proberen spijtig genoeg falen.

1.4 De Linuxkernel

De Linuxkernel is een "programma" geschreven in C-code en waar nodig in assembler. De kernel wordt ontwikkeld door duizenden programmeurs, die zich graag kernelhackers laten noemen, verspreid over de volledige wereld. Men kan zich afvragen hoe dit ooit goed kan aflopen. Tussen die duizenden programmeurs zitten echter enkele sterke, belangrijke en invloedrijke kernelhackers die heel veel kennis bezitten over de kernel en die verantwo-

ordelijk zijn voor bepaalde subsystemen van de kernel. Deze personen zijn echter enkel zo belangrijk omdat ze door de Linuxgemeenschap aanvaard worden als autoriteit. Indien zo'n "leider" een beslissing neemt die ingaat tegen de algemene consensus, zal die persoon haar status verliezen en zullen haar beslissingen genegeerd worden. Dit is dus een zeer democratisch systeem. Aangezien zo'n leider constant door anderen wordt aangespoord om bepaalde beslissingen te nemen, moet zij over veel kennis beschikken en soms handelen als een verlicht despoot. Indien zij misbruik maakt van haar macht, kan die macht haar echter ontnomen worden.

Communicatie tussen de kernelhackers gebeurt via private mail, via specifieke mailing lists en soms ook (sporadisch) in levende lijve.

Op [8] vind je alle kernelbroncode en meer. Om door de kernelbroncode te navigeren via hyperlinks, wat zeer goed van pas komt tijdens een codelezing, kan je op [12] terecht. De FAQ (*Frequently Asked Questions*) over de *Linux kernel mailing list* (lkml) is te vinden op [13]. Eén van de manieren om de lkml te doorzoeken is via de relevante Google-site [9].

1.4.1 Nummering van kernels

Een bepaalde uitgave van een Linuxkernel heeft een versienummer, dat bestaat uit 3 cijfers, gescheiden door punten, bv. `2.5.49` of `2.4.20`. Het eerste cijfer wordt het *major number* genoemd en het tweede nummer wordt het *minor number* genoemd, het derde nummer geeft de *patch level* aan.

Er zijn twee soorten Linuxkernelreeksen: de stabiel geachte reeks en de reeks die dient voor nieuwe ontwikkelingen. De stabiele reeksen worden gekenmerkt door een even nummer als *minor number*, ontwikkelingskernels krijgen een oneven *minor number*. Zo is versie `2.4.20` op het moment wanneer deze zin geschreven wordt de laatste versie van de `2.4` kernel reeks. Oude stabiele reeksen, zoals de `2.2` reeks, worden nog verder onderhouden, oude ontwikkelingsreeksen niet.
Wanneer een *development kernel* stabiel wordt, zal z'n *minor number* worden verhoogd met 1. De `2.5.x` ontwikkelingsreeks is op *Halloween 2002* (30 okto-

ber) in een laatste stadium gekomen: toen werd een (vooraf aangekondigde) *feature freeze* voor de *development kernel* van kracht. Vanaf dan is het de bedoeling dat geen nieuwe *features* worden toegevoegd aan de kernel, maar dat de code stabiel wordt gemaakt. Dit goede voornemen wordt echter nog regelmatig overtreden, zo heeft Paul "Rusty" Russel de kernelmodulecode volledig herschreven na de *feature freeze* en is de oude modulecode vervangen door zijn implementatie. Na enkele maanden stabilisering zullen enkele *release candidates* worden uitgebracht, van de vorm `2.5.99-pre1`. Eenmaal de code stabiel genoeg is, wordt `2.5.99-prex` omgedoopt tot `2.6.0`.

Als men vindt dat er veel verschil is tussen de nieuwe stabiele kernelreeks en de oude, verhoogt men het *major number* en begint het *minor number* terug vanaf nul. Zo was er een discussie op de lkml tussen voorstanders van `2.6` en voorstanders van `3.0`. De voorstanders van `2.6` hebben het gehaald.

1.4.2 Soms loopt het mis

De IDE-code (*Integrated Drive Electronics*) in de kernel zorgt o.a. voor de werking van de modale hardeschijven. Kerneltesters hebben geen probleem dat hun computer regelmatig niet meer reageert zodat heropstarten noodzakelijk is. Ze vinden het meestal wel erg als de kernel hun bestandssysteem verknoeit. Daarom dat een stabiel, of bruikbaar, IDE-subsysteem voor de `2.5` ontwikkelingskernel nogal een veel gestelde wens was.

Het *maintainership* van het IDE-subsysteem was lange tijd in handen van Andre Hedrick. Andre en Linus komen echter niet goed overeen, waardoor Andre het moeilijk had om IDE-code in de door Linus gecontrolleerde `2.5` kernel te krijgen. Daarmee was er een probleem met de IDE-ontwikkeling in de `2.5` kernel. Vanaf februari 2002 begon Marcin Dalecki met het *posten* van *IDE-cleanups*, waarvan zo'n *cleanup* op 8 maart Andre Hedrick als *maintainer* van *IDE DRIVER [GENERAL]* uit het *MAINTAINERS*-bestand haalde en verving door zijn eigen naam. Linus accepteerde deze wijziging, waardoor de *hostile takeover* van het IDE-subsysteem door Marcin Dalecki een feit was. Zijn boodschap op de kernel mailing list kan je bv. vinden op [10].

Dalecki's werk aan de IDE-code zorgde voor heel wat wrevel bij andere kernelhackers en kerneltesters. De veranderingen zorgden nl. voor onstabiele IDE-code, waardoor kerneltesters hun geliefde data op hun hardeschijf konden kwijtspelen. Enkel de trotse bezitters van SCSI-hardeschijven (*Small Computer System Interface*) konden de `2.5` kernel uittesten zonder een relatief grote zekerheid te hebben dat hun bestandssysteem naar de knoppen zou worden geholpen. Het probleem werd zo groot dat Linus (nadat Marcin Dalecki had gezegd dat het voor hem niet meer hoefde, hij kreeg teveel tegenkanting op de lkml) in kernel `2.5.32`, op 16 augustus 2002, het volledige `2.5` IDE-subsysteem verwijderde en verving door een *foreport* van het stabiele `2.4`-subsysteem. Deze *foreport* werd al een tijdje onderhouden door o.a. Jens Axboe.

Het resultaat van dit alles is dat de `2.5` code nu zonder officiële IDE-*maintainer* zit, aangezien het nog altijd niet botert tussen Andre Hedrick en Linus Torvalds. Het *MAINTAINERS*-bestand van de `2.5.49` kernel zegt wel dat *IDE DRIVER [GENERAL]* onderhouden wordt, maar geeft geen naam voor de *maintainer*.

Het goede nieuws is dat de IDE-code nu wel veilig is en dat de `2.5` kernel veel meer getest wordt.

1.4.3 Monolithische en microkernels

De Linuxkernel is wat men noemt een monolithische kernel. De Windows NT kernel en de Hurd kernel daarentegen zijn voorbeelden van microkernels. Er zijn al vele *flame wars* gestreden tussen aanhangers van beide types kernels. Een beroemde discussie is de Tanenbaum/Linus "*Linux is obsolete*" (Linux is voorbijgestreefd) discussie [11]. Kort komt het erop neer dat microkernels trager maar makkelijker en dus ook makkelijker bugvrij te maken zijn, terwijl monolithische kernels sneller maar ingewikkelder zijn. Microkernels zijn jonger dan monolithische kernels. De verschillende subsystemen van de microkernels communiceren met elkaar via een systeem van signalen. In de monolithische kernel gebeurt die communicatie rechtstreeks via func-

tieaanroepen. Het werken met die signalen, wat eigenlijk gewone datatypes zijn, maakt de microkernels trager, maar beschermt een subsysteem wel van eventuele fouten in andere subsystemen.

Het ontwikkelen van een nieuw subsysteem in een microkernel is minder lastig dan bij monolithische kernels, omdat de code van het subsysteem bij monolithische kernels toegang heeft tot de code en data van de volledige kernel. Bij microkernels is dit niet zo, waardoor fouten minder desastreus zijn en makkelijker te vinden zijn tijdens het debuggen.

1.4.4 Het compileren van een kernel

Oningewijden zullen waarschijnlijk denken dat het compileren van een kernel een helse taak is. Niets is minder waar, met weinig moeite kan een nieuwe, werkende kernel worden gecompileerd. Het perfect configureren van de kernel naar eigen noden en smaak zal natuurlijk wat omslachtiger zijn.

Eerst moet de broncode van de kernel worden afgehaald van het Internet, laten we de nieuwste *development kernel* van de `2.5.x` reeks afhalen:

```
$ cd /usr/src
$ ftp ftp.kernel.org
Connected to ftp.kernel.org (204.152.189.116).
220 ProFTPD [ftp.kernel.org]
Name (ftp.kernel.org:bdschuym): anonymous
Password: iets
230 Anonymous access granted, restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp > cd pub/linux/kernel/v2.5
ftp> ls *LATEST*
-rw-r--r--   1 korg  0 Feb 24 19:28 LATEST-IS-2.5.63
ftp> get linux-2.5.63.tar.bz2
ftp> exit
```

Het bestand dat begint met *LATEST* duidt aan welke kernelversie de nieuwste is. Daarna *unzippen* en *untarren*:

```
$ bzip2 -d linux-2.5.63.tar.bz2
$ tar -xf linux-2.5.63.tar
```

Nu de configuratie van de kernel:

```
$ cd linux-2.5.63
$ make menuconfig
```

Onder "*Processor type and features → Processor family*" kies je het type van je computer, bv. een "Atlon". Dit is normaalgezien al voldoende om een bruikbare kernel te compileren. Indien je wenst dat je netwerkkaart werkt, moet je deze selecteren via "*Networking support → Ethernet (10 or 100Mbit)*", bv. een "Realtek RTL-8192". Om er zeker van te zijn dat niets misloopt, kies je misschien best om die te compileren zodat ze ingebakken zit in de kernel (dus niet als module). Sluit het programma dan af en kies voor het opslaan van de configuratie. We zijn nu klaar om de kernel te compileren.

```
$ make bzImage modules modules_install
```

Dit zou zonder problemen moeten verlopen. Log nu in als **root**:

```
$ su
Password:
```

De gecompileerde kernel moet nu verplaatst worden naar je bootpartitie:

```
$ cp arch/i386/boot/bzImage /boot/2_5_63_test
```

Het bovenstaande veronderstelt dat je computer een Intel-compatibele pc is. Nu moeten we nog ervoor zorgen dat we kunnen kiezen om op te starten met deze nieuwe kernel:

```
$ pico /etc/lilo.conf
```

Hierin bevindt zich normaal al minstens 1 ingave. Kopieer die gewoon en verander de *image* naam en het *label*. De standaardingave ziet er bv. zo uit:

```
image=/boot/vmlinuz-2.4.7-10
        label=linux
        initrd=/boot/initrd-2.4.7-10.img
        read-only
        root=/dev/hda3
```

En we voegen dit toe:

```
image=/boot/2_5_63_test
        label=linux2_5_63_test
        initrd=/boot/initrd-2.4.7-10.img
        read-only
        root=/dev/hda3
```

Nu moeten we lilo zichzelf nog laten herconfigureren:

```
$ lilo
$ exit
```

Bovenstaande is in de veronderstelling dat je Linuxdistributie lilo gebruikt bij het booten. Nieuwere systemen gebruiken grub, zie *http://www.google.be* voor meer informatie hierover.

Hoofdstuk 2

Begrippen

De bedoeling van dit hoofdstuk is om de begrippen die nodig zijn om de diepere werking van de kernel te kunnen beschrijven, uit te leggen. Het is niet de bedoeling om er een cursus Besturingssystemen van te maken, daarvoor kan je (momenteel toch nog) elders terecht. Zie ook bvb. [18, 19, 25]. Dit hoofdstukje is misschien wel een beetje te beknopt. . .

2.1 Hardware en software

De uitwendige werking van een computer, zoals de gebruiker die ziet, berust op de werking van de hardware van de computer zelf, en de software die op deze computer draait.

Een computer op zich bestaat enkel uit hardware, dus harde, tastbare dingen, zoals het moederbord, het geheugen, de cpu (*central processing unit*). Deze dingen hebben elk hun functie, maar ze doen maar iets als hun gezegd wordt iets te doen. Daarvoor dient de software, deze zegt de cpu bvb. om 1 bij 1 op te tellen en het resultaat op te slaan op geheugenplaats 1777.

2.1.1 Hardware

De cpu

De cpu (*central processing unit*, of centrale verwerkingseenheid, of gewoon kort de processor) is het werkpaard van de computer. Deze wordt aangestuurd door de software. De cpu doet dus slaafs alles wat de software beveelt.

Interrupts

De hardware van de computer voorziet een mechanisme dat interrupts wordt genoemd. Via een interrupt wordt de huidige werking van de cpu, die dus bepaald wordt door de software die de computer momenteel bestuurt, tijdelijk onderbroken om de één of andere cruciale actie uit te voeren. Zo'n interrupt kan bvb. veroorzaakt worden doordat de netwerkkaart een IP-pakketje ontvangt. De bijhorende interrupt zal ervoor zorgen dat de software die de verwerking van het IP-pakketje afhandelt, wordt uitgevoerd. Nadat dit gebeurd is, wordt de onderbroken software terug uitgevoerd.

CISC en RISC

Men onderscheidt twee categorieën processoren: de CISC (*Complex Instruction Set Computer*) en RISC (*Reduced Instruction Set Computer*) architecturen. De processoren die in de pc's zitten (zoals die van Intel of AMD) zijn van het type CISC, terwijl de computers van bvb. Sun Microsystems (de Sparc processoren) van het RISC type zijn. CISC-processoren hebben een grote instructieset en weinig registers, terwijl RISC-processoren juist enkel elementaire instructies bezitten maar dan wel meer registers hebben.

Men lijkt geleidelijk aan af te stappen van CISC-processoren. Het is bvb. veel makkelijker voor de hardwaremakers om de uitvoering van RISC-instructies te parallellizeren dan de uitvoering van CISC-instructies.

Opslagmedia

Voor onze doeleinden onderscheiden we 4 opslagmedia: registers, caches, RAM-geheugen en harde schijven. Deze 4 opslagmedia zijn geordend in dalende volgorde qua snelheid en prijs, en in stijgende volgorde qua de grootte van de opslagcapaciteit.

Registers
: Elke cpu heeft zijn eigen registers, dit zijn fysieke plaatsen in de processor waarin een aantal bytes kan worden opgeslaan en gelezen. Deze schrijf- en leesacties zijn zeer snel. Het aantal registers dat een cpu heeft is zeer beperkt, dit zijn er slechts een tiental bij CISC-processoren en een honderdtal bij RISC-processoren.

Caches
: Caches zijn geheugen waarvan de alledaagse software zich niet bewust is. De cache-geheugens bevinden zich tussen het hoofdgeheugen en de processor en zijn veel sneller dan het gewone RAM-geheugen. Aangezien de processor veel sneller is dan het RAM-geheugen moet zoveel mogelijk worden vermeden dat de processor moet wachten op het voltooien van de acties van het RAM-geheugen. Dit gebeurt door de meest gebruikte of recentst gebruikte gegevens op te slaan in de cache.

RAM-geheugen
: Het RAM-geheugen (*Random Access Memory*) wordt gebruikt als tijdelijk opslagmiddel voor gegevens en software.

Harde schijven
: Deze dienen voor massa-opslag en ook als logische uitbreiding van het RAM-geheugen, zie paginering in Hoofdstuk 8. De inhoud van een harde schijf blijft bewaard, ook nadat de pc werd afgezet.

smp's

De Linuxkernel ondersteunt een speciaal soort computer: de smp (*symmetrical multi-processor*). Een smp is een computer die bestaat uit meerdere processoren (meestal een macht van twee) die een gemeenschappelijk RAM-geheugen gebruiken. Ze gebruiken dus hetzelfde besturingssysteem. Elke

processor heeft wel zijn eigen registers en cache(s). Doordat verschillende processoren aan hetzelfde geheugen kunnen, ontstaan er synchronisatieproblemen, waarover meer te vinden is in Hoofdstuk 4.

2.1.2 Software

Dit is een veel te kleine subsectie, doch, hier is ze.

De Von Neumann-architectuur

Alle hedendaagse computers bewaren zowel de software als de gegevens in hetzelfde geheugen. Dit lijkt waarschijnlijk niet meer dan logisch voor de lezer, wanneer Von Neumann dit concept bedacht was dit echter revolutionair.

Processen

Processen zijn instanties van programma's die werkzaam zijn op de computer. Zo kan een bepaald programma tegelijkertijd meermaals uitgevoerd worden. Dit zijn dan verschillende processen, maar ze voeren wel hetzelfde programma uit.

Stack

Een stack (stapel) is een zogenaamde LIFO (*Last In First Out*) datatype. Hetgeen er laatst opgezet is, wordt er het eerst afgehaald. We kunnen dit makkelijk voor ogen brengen door te denken aan een stapel papier: als we er een papier bijleggen gebeurt dit bovenaan, als we er een papier van nemen zal dit het bovenste papier zijn.

In gebruikerssoftware worden stacks via software geïmplementeerd en kan men kiezen hoe groot de data is die in één stap op een stack wordt gezet. In de hardware van de computer wordt de stack enkel per 2 of 4 bytes verhoogd of verminderd, afhankelijk van de operandgrootte. Met de assemblerinstruc-

tie `push` wordt een machinewoord op de stack gezet en met `pop` wordt een machinewoord van de stack gehaald.

De implementatie door een processor van een stack is zeer eenvoudig: een wijzer naar de top van de stack wordt bijgehouden in een register (bij de x86 is dit `esp`). Een woord toevoegen (met `push`) gebeurt door dit op het adres waarnaar verwezen wordt door `esp` te plaatsen en de waarde van `esp` met 2 of 4 te verminderen. De stack groeit dus naar beneden. Een element verwijderen (met `pop`) gebeurt door de waarde van `esp` te verhogen met 2 of 4 en het woord waarnaar `esp` verwijst te kopiëren naar het aangegeven register of geheugenadres.

In assemblercode worden stacks gebruikt om tijdelijk kopiën op te slaan van registers die zullen gebruikt worden in tussenberekeningen. Ook wanneer met de assemblerinstructie `call` een subroutine wordt aangeroepen, wordt de stack gebruikt om het `ip`-register te bewaren. Dit register bevat de *instruction pointer*: het adres van de volgend uit te voeren instructie. Door de waarde van dit register eerst te bewaren op de stack, kan de waarde van `ip` dan worden veranderd zodat code op een andere plaats in het geheugen zal worden uitgevoerd. Het terugkeren uit de subroutine gebeurt met de assemblerinstructie `ret`, die de oude waarde van `ip` terug in dit register steekt.

Hoofdstuk 3

De kernel codingstyle

Alhoewel er geen echte verplichtingen bestaan i.v.m. de gebruikte stijl van C-code, is er toch een stijl die duidelijk zichtbaar is doorheen de kernel, die door de overgrote meerderheid van de *kernel hackers* wordt gehanteerd.

De door Linus geprefereerde stijl wordt door hemzelf uitgelegd in het bestand *Documentation/CodingStyle* dat bij elke Linux kerneldistributie zit (aangename lectuur). Een afwijkende stijl is niet verboden, maar veelal zal de *maintainer* van een bepaald subsysteem van de kernel haar veto stellen tegen grove inbreuken op deze stijl.

De *kernel codingstyle* lijkt sterk op de K&R-codingstyle, ingevoerd door Brian W. Kernighan en Dennis M. Ritchie. Werkende bij AT&T Bell Laboratories ontwikkelde Ritchie, samen met Ken Thompson, het UNIX besturingssysteem voor minicomputers. Later ontwikkelde hij de programmeertaal C. Kernighan en Ritchie zijn de auteurs van o.a. wat C hackers wel eens het Oude en het Nieuwe Testament noemen: "*The C Programming Language*" resp. "*The ANSI C Progamming Language*". In deze boeken, die de C-programmeertaal beschrijven, gebruiken ze in hun code de nu beroemde K&R-stijl.

3.1 Het gebruik van accolades

Eén van de aspecten van programmeerstijl waarvoor reeds ettelijke *holy wars* gestreden zijn, is het gebruik van de accolades ({ en }). Het achtste gebod uit het document "*The Ten Commandments for C Programmers*" is hierover zeer duidelijk: "*Thou shalt make thy program's purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding*". Hierbij stelt "the One True Brace Style" de door K&R gehanteerde accoladestijl voor. Deze wordt goed omschreven door Linus Torvalds (*Documentation/CodingStyle*):

"*The other issue that always comes up in C styling is the placement of braces. Unlike the indent size, there are few technical reasons to choose one placement strategy over the other, but the preferred way, as shown to us by the prophets Kernighan and Ritchie, is to put the opening brace last on the line, and put the closing brace first, thusly:*

```
if (x is true) {
        we do y
}
```

However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
        body of function
}
```

Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are _right_ and (b) K&R are right. Besides, functions are special anyway (you can't nest them in C).

Note that the closing brace is empty on a line of its own, _except_ in the

cases where it is followed by a continuation of the same statement, i.e. a `while` *in a* `do`*-statement or an* `else` *in an* `if`*-statement, like this:*

```
do {
        body of do-loop
} while (condition);
```

and

```
if (x == y) {
        ...
} else if (x > y) {
        ...
} else {
        ...
}
```

Rationale: K&R.

Also, note that this brace-placement also minimizes the number of empty (or almost empty) lines, without any loss of readability. Thus, as the supply of new-lines on your screen is not a renewable resource (think 25-line terminal screens here), you have more empty lines to put comments on."

3.2 Het `goto`-statement

Haters van het gebruik van het `goto`-statement zullen bij een lezing van de kernelbroncode versteld staan van het veelvuldig gebruik van dit statement. De onwetenden onder hen zullen dit onmiddelijk afwijzen en de code omschrijven als spaghetticode. Doch, de iets langer nadenkende personen onder deze groep der `goto`-haters beseffen dat vele van deze *kernel hackers* zeer goede programmeurs zijn en dat er daarom een goede reden moet bestaan voor dit gebruik. Een tijdje geleden was er een interessante discussie op de lkml over het `goto`-statement. Hieruit hebben we het volgende onthouden. Het was Edsger Dijkstra (de uitvinder van de semafoor, zie Hoofdstuk 4) die als eerste

afkwam met het dogma *gotos are evil*, in zijn strijd voor het gestructureerd programmeren. Niklaus Wirth, de auteur van de programmeertalen Pascal en Modula-2, legde gestructureerd programmeren op in de twee voornoemde talen. Linus' commentaar hierover gaat als volgt: "... *Niklaus Wirth, who took the "structured programming" thing and enforced it in his languages (Pascal and Modula-2), and thus forced his evil on untold generations of poor CS students who had to learn languages that weren't actually useful for real work.*"

Rik Van Riel brengt volgend overtuigend argument aan ten gunste van het (goed) gebruik van `goto`: "*If the main flow of the code is through a bunch of hard to trace gotos and you choose to blame the tool instead of the programmer, I guess you could blame goto.*

However, the goto can also be a great tool to make the code more readable. The goto statement is, IMHO [1], *one of the more elegant ways to code exceptions into a C function; that is, dealing with error situations that don't happen very often, in such a way that the error handling code doesn't clutter up the main code path.*

As an example, you could look at fs/super.c::`do_kern_mount()`

```
mnt = alloc_vfsmnt(name);
if (!mnt)
        goto out;
sb = type->get_sb(type, flags, name, data);
if (IS_ERR(sb))
        goto out_mnt;
```

Do you see how the absence of the error handling cleanup code makes the normal code path easier to read?"

Rik merkt terecht op dat niemand zegt dat het gebruik van `goto` altijd goed is. Daarnaast moge het duidelijk zijn uit bovenstaand voorbeeld dat indien de code, die moet worden uitgevoerd wanneer `!mnt` waar is, groot is, de leesbaarheid vermindert. Merk op dat het `goto`-statement met labels

[1] In My Humble Opinion

werkt die door de programmeur een zeer goede omschrijvende naam kunnen krijgen. Aangezien de code die moet uitgevoerd worden indien `!mnt` geldt, het afhandelen van een niet veel voorkomende fout betreft, is het mooi dat deze code niet direct zichtbaar is tussen de code die in normale omstandigheden wordt uitgevoerd.

Een ander overtuigend argument wordt ons gegeven door Alan Cox: "*That's the trouble with programming an abstract level. People just don't understand the real world. Cache is everything, small cache footprint is king.*" Stel dat de programmacode van hierboven ergens in het geheugen zit. Indien dit geheugen kan gecachet worden, zal de programmacode veel sneller uitgevoerd worden dan indien meermaals naar het fysieke geheugen moet gegaan worden. Zoals gezegd zal bovenstaande code de meerderheid van de tijd uitgevoerd worden zonder dat één van de `goto`-statements wordt uitgevoerd. Stel nu dat we het `goto`-statement veranderen door de echte foutafhandelingscode. Dan zal er tussen de code `mnt = alloc_vfsmnt(name);` en `sb = type->get_sb(type, flags, name, data);` veel meer ruimte zijn (bezet door foutafhandelingscode). De kans wordt hierdoor groter dat deze code niet meer in 1 *cacheline* kan, waardoor misschien rechtstreeks naar het geheugen zal moeten worden gegaan om de volgend uit te voeren instructie te halen. Dit is een belangrijk verschil omdat, zoals Alan opmerkt, de processorsnelheden van de huidige computers zo snel zijn, dat de relatieve traagheid van het hoofdgeheugen een zeer grote rol begint te spelen in de performantie van het systeem. De caches zijn echter bliksemsnel, dus moet zoveel mogelijk worden gepoogd om code die waarschijnlijk zal worden uitgevoerd, reeds beschikbaar te hebben in het cachegeheugen. Zeker voor kernelcode is dit zeer belangrijk.

Het `goto`-statement is ook zeer handig in functies waar veel kan foutlopen en waarbij er dan de nodige foutafhandelingscode is. Hier is een voorbeeld in meta-taal:

```
		doe_iets;
		if (fout)
			goto foutje;
		doe_nog_iets;
		if (fout)
			goto nog_foutje;
		doe_dan_nog_iets;
		if (fout)
			goto en_nog_foutje;
		return succes;
en_nog_foutje:
		kuis_op;
nog_foutje:
		kuis_nog_op;
foutje:
		en_kuis_nog_op;
		return spijtig;
```

Zonder het `goto`-statement zou hier opkuiscode moeten herhaald worden, wat voor grotere programmacode zorgt, maar vooral: het maakt de code moeilijker onderhoudbaar. Indien `en_kuis_nog_op` moet aangepast worden zou dit, zonder het gebruik van `goto`, op drie plaatsen moeten gebeuren.

De `break`- en `continue`-statements alsook het gebruik van `return` midden in de code worden door de *kernel hackers* ook volgaarne gebruikt.

3.3 Indentatie en alignering

De meeste kernelcode gebruikt als indentatiemiddel de tabulator, waarbij wordt verondersteld dat een tab 8 spaties beslaat en waarbij er wordt gestreefd om een lijn code niet breder dan 80 karakters te maken. De getallen 8 en 80 zijn te verklaren doordat een normale console een regelbreedte heeft van 80 tekens en als tabafstand standaard 8 spaties wordt gebruikt. Tegenwoordig schrijven programmeurs hun code in een grafische omgeving, waardoor een tabulatorafstand van 8 spaties soms ongewenst is. Er is echter een manier om de indentatiediepte van de code door de lezer te laten bepalen, dankzij

het gebruik van de tabulator. We moeten hierbij onderscheid maken tussen indentatie en alignering. Indentatie dient om de nesting van codeblokken duidelijker te maken, alignering dient om bepaalde tekst mooi onder elkaar te zetten. Bekijken we volgend voorbeeld (uit *net/bridge/netfilter/ebtables.c*):

```
if (((struct ebt_entries *)e)->policy != EBT_DROP &&
   ((struct ebt_entries *)e)->policy != EBT_ACCEPT) {
	/* only RETURN from udc */
	if (i != NF_BR_NUMHOOKS ||
	   ((struct ebt_entries *)e)->policy != EBT_RETURN) {
		BUGPRINT("bad policy\n");
		return -EINVAL;
	}
}
```

Wanneer als indentatie spaties i.p.v. tabs worden gebruikt, heeft de lezer geen keuze wat indentatiediepte betreft. Indien we i.p.v. een vast aantal spaties, de tabulator gebruiken als indentatie, kan de lezer haar editor instellen naar wens en zo de indentatiediepte vastleggen op haar voorkeur. We doen dit als volgt:

```
<TAB>if (((struct ebt_entries *)e)->policy != EBT_DROP &&
<TAB>   ((struct ebt_entries *)e)->policy != EBT_ACCEPT) {
<TAB><TAB>/* only RETURN from udc */
<TAB><TAB>if (i != NF_BR_NUMHOOKS ||
<TAB><TAB>   ((struct ebt_entries *)e)->policy != EBT_RETURN) {
<TAB><TAB><TAB>BUGPRINT("bad policy\n");
<TAB><TAB><TAB>return -EINVAL;
<TAB><TAB>}
<TAB>}
```

Merk op dat we op sommige plaatsen wel nog spaties gebruiken: op de tweede en vijfde lijn zien we voorbeelden van alignering, dit zorgt ervoor dat de haakjes mooi onder elkaar blijven staan. De combinatie van alignering en de tabulator als indentatiemiddel zorgt voor code die voor iedereen goed leesbaar is.

Hoofdstuk 4

SMP en locking

In dit hoofdstukje wordt de smp-architectuur geïntroduceerd en alle *locking* die daarmee gepaard gaat. Hier en daar worden codefragmenten getoond.

4.1 SMP

Zoals reeds gezegd, staat smp voor *symmetrical multi-processor*. Een smp-machine is een computer met minstens 2 processoren (meestal een macht van 2), waarbij elke processor hetzelfde geheugen gebruikt en dus ook dezelfde besturingssysteemcode die in het geheugen zit. Wanneer Linux draait op een smp, bevindt de code van de Linuxkernel zich normaal gezien dus maar 1 keer in het geheugen. Er zijn wel speciale technieken om meerdere kernels tegelijkertijd te doen werken, bv. *User-Mode Linux* [14], wat sinds kernel `2.5.35` in de standaarddistributie zit. Het feit dat er meerdere processoren aanwezig zijn vereist dat goed omgesprongen wordt met data. Verschillende processoren kunnen nl. tegelijkertijd toegang hebben tot dezelfde gegevens, wat tot gegevenscorruptie kan leiden. De oplossing hiervoor is het gebruik van *locking*.

De Linux kernel ondersteunt smp-machines al sinds de `2.0` kernels, maar deze kernels gebruikten een *big kernel lock* om corruptie van data te voorkomen. Tussen `2.0` en `2.4` is werk verricht om de *kernel locking* efficiënter te maken.

De `2.4` kernels bevatten *fine-grained kernel locking*, wat eigenlijk gewoon betekent dat er veel *locks* zijn bijgekomen. Het *big kernel lock* is dus opgesplitst in verschillende *locks* die elk hun eigen gedeelte van de kernel beschermen. Een toename aan *locks* zorgt er natuurlijk voor dat de complexiteit van de kernelcode toeneemt, de kans op *deadlocks* [1] en gecorrumpteerde data wordt nl. veel groter. Dat is natuurlijk ook de reden waarom men smp-ondersteuning eerst simpel heeft opgelost via 1 *big kernel lock*. Complexe zaken worden soms beter in verschillende stappen bijgevoegd. Zoals iemand ooit gezegd heeft: "*premature optimization is root of all evil.*"

4.2 Locking

4.2.1 Algemeen

Wat is *locking* nu eigenlijk en waarvoor dient het? Stel dat twee mensen naar hetzelfde toilet willen gaan. Het is wenselijk dat beiden niet tegelijkertijd naar het toilet gaan, aangezien dit de aanwezige toiletinfrastructuur kan corrumperen. We moeten dus een mechanisme hebben om dit te verhinderen. In het dagelijkse leven worden hiervoor de zintuigen gebruikt (ogen en zo). Laten we dit nu vertalen naar computerprocessen (de mensen) en data (de toiletinfrastructuur). Stel dus dat twee processen tegelijkertijd data willen veranderen op dezelfde geheugenplaats. Indien dit niet gecontroleerd gebeurt, kan dit leiden tot corruptie van de data. De processen hebben dus een hulpmiddel nodig om te zien of ze mogen schrijven of niet (ze hebben geen zintuigen). Het gebruikte hulpmiddel is een *lock*. Laten we, om het simpel te houden, aannemen dat het lock de status "vrij" en "bezet" kan hebben (naar analogie met het toilet). Wanneer het proces ziet dat de status vrij is zal het vlug de status op bezet zetten en de data schrijven. Als de status op bezet staat, wacht het totdat de status op vrij komt. Er moet natuurlijk voor gezorgd worden dat twee processen niet tegelijkertijd de status op bezet

[1] een situatie waarin oneindig lang wordt gewacht op het vrijgeven van een *lock*

kunnen zetten. Om te weten hoe dit gebeurt moeten we eerst zeggen hoe zo'n *lock* in computertaal wordt omgezet. Zo'n *lock* is niets meer dan een getal, opgeslagen in het geheugen. Als dit getal op 1 staat, staat het *lock* op "vrij", anders staat het *lock* op 0 wat "bezet" betekent. Door gebruik te maken van hardware instructies die een bepaalde geheugenactie atomair [2] uitvoeren wordt ervoor gezorgd dat twee processen dit getal niet tegelijkertijd van 1 naar 0 kunnen zetten. Voor de x86 code wordt o.a. gebruik gemaakt van het assembler-voorvoegsel `lock`, dat ervoor zorgt dat het `LOCK#`-signaal van de x86 wordt geactiveerd gedurende de uitvoering van de instructie die er op volgt. Dit signaal zorgt ervoor dat de processor het exclusieve recht heeft op het gedeelde geheugen. De concrete implementatie van dit simpele "vrij/bezet"-*lock* (dat overeen komt met een *spinlock*) is als volgt. De initialisatie zet het getal op 1. Een proces probeert het *lock* op bezet te zetten door het getal met 1 te verminderen, dit gebeurt atomair. Indien de uitkomst van deze vermindering 0 is, heeft het proces het *lock* op bezet gezet, anders stond het *lock* al op bezet en zal het proces opnieuw proberen. Nadat een proces het *lock* niet meer nodig heeft, zal het proces het *lock* weer vrijgeven. Dit gebeurt door de waarde terug op 1 te zetten d.m.v. een atomaire actie. Naast dit eenvoudige *lock* bestaan er natuurlijk meer ingewikkelde varianten.

Voor een meer uitgebreide dissertatie over *locking* (en veel meer), kan je bv. terecht bij het boek [18].

4.2.2 De functie `spin_trylock()`

Bekijken we even de implementatie van het *spinlock* voor de x86-processorfamilie (zie *include/asm-i386/spinlock.h*). Een *spinlock* wordt als volgt gedefinieerd:

```
typedef struct {
  volatile unsigned int lock;
} spinlock_t;
```

[2] tijdens de geheugenbewerking kan de inhoud van de gebruikte geheugenlocatie enkel door deze instructie worden gewijzigd en/of gelezen

De *qualifier* `volatile` is nodig voor de *member* `lock` van de `struct` opdat de compiler geen ongewenste optimalisaties zou doen. De *member* `lock` kan nl. veranderd worden op manieren die de compiler niet kan weten. We bekijken de definitie van de functie `spin_trylock()` naderbij, zie *include/linux/spinlock.h*:

```
#define spin_trylock(lock) ({preempt_disable(); _raw_spin_trylock(lock) ? \
        1 : ({preempt_enable(); 0;});})
```

Eerst wordt *kernel preemption* uitgeschakeld. Indien de *kernel preemption* is ingebouwd in de Linuxkernel, kunnen *kernel threads* voortijdig worden onderbroken door de *scheduler*. De situatie die ontstaat wanneer een *kernel thread* wordt onderbroken terwijl hij een spinlock vast heeft, is geen aangename, daarom moet dit verhinderd worden. De functie `_raw_spin_trylock()` is terug architectuurafhankelijk, voor de x86-processorfamilie wordt deze gedefinieerd in *include/asm-i386/spinlock.h*:

```
static inline int _raw_spin_trylock(spinlock_t *lock)
{
  char oldval;
  __asm__ __volatile__(
    "xchgb %b0,%1"
    :"=q" (oldval), "=m" (lock->lock)
    :"0" (0) : "memory");
  return oldval > 0;
}
```

De assemblerinstructie `xchgb` wisselt de waarde van een geheugenplaats en register om op atomaire manier. De bovenstaande code is een staaltje van `gcc` *inline assembly* (zie tutorials bij [17]). De variabele `oldval` stelt een register voor dat door de compiler wordt gekozen: de (`''=q''`) *constraint* eist dat `oldval` een register voorstelt. (`''0'' (0)`) zorgt ervoor dat eerst de waarde 0 in het register corresponderende met `oldval` wordt gestoken. Daarna wordt `lock->lock` en `oldval` omgewisseld, zodat `lock->lock` zeker gelijk wordt aan nul en in `oldval` zich de oude waarde van het *lock* bevindt.

Indien de oude waarde van het *lock* gelijk was aan 0, betekent dit dat het *lock* al bezet is, anders hebben we net het *lock* zelf vast.

Een interessant Nederlandstalig *online* boek (beschikbaar in pdf-formaat) over *assembly* is te vinden op [15].

4.2.3 Soorten locks

In de kernel worden twee klasses *locks* gebruikt: de *spinlocks* en de *mutexen*. Een *spinlock* blijft in een lus spinnen totdat het proces het *lock* op bezet kan zetten. Deze *locks* zijn hierdoor zeer snel, maar nemen wel de processor in beslag terwijl ze spinnen. Herinner je dat we op een smp bezig zijn, dus de andere processoren voeren nog handelingen uit. Het vrijgeven van het *spinlock* zal dus door één van de andere processoren gebeuren. Deze *locks* worden gebruikt in situaties waar alles snel moet gaan, zoals bv. in de kernelcode die netwerkpakketten afhandelt. Er bestaat een variant van deze *spinlocks*, nl. de "read/write"-*spinlocks*. Hierbij mogen er om het even hoeveel lezers actief zijn tegelijkertijd. Als een schrijver actief is, daarentegen, mag geen enkele lezer noch andere schrijver actief zijn.

De tweede klasse bestaat uit *locks* waarbij de (kernel)processen slapen terwijl het *lock* niet vrij is, dit is de *mutex*. Wanneer zo'n proces slaapt, geeft het de processor terug vrij, waardoor ander werk kan worden verricht. *Mutexen* zijn veel trager dan *spinlocks*, omdat het slapend proces terug moet wakker gemaakt worden wanneer het *lock* is vrijgekome. Een *mutex* is eigenlijk een speciale variant van een semafoor, die door de Nederlander Dijkstra, E. werd geïntroduceerd: het is een semafoor met als beginwaarde 1. Indien de semafoor beginwaarde `n` krijgt, kan de semafoor simultaan door `n` processen vastgehouden worden. Een (algemene) semafoor wordt echter niet aanzien als een echte *lock*. Semaforen zijn echter ook beschikbaar in de kernelcode. De *mutex* wordt wel als *lock* aanzien.

Bij uniprocessoren (computers met maar 1 processor), dus bij elke computer van de modale medemens, zijn *spinlocks* overbodig. Inderdaad, indien *spinlocks* wel zouden gebruikt worden, zou de processor oneindig lang

wachten tot het *lock* wordt vrijgegeven. Daarom dat voor uniprocessoren het gebruik van *spinlocks* wordt vertaald in een *no-op* (lege operatie), die door de compiler wordt verwijderd.

4.2.4 Locking bij processen die kunnen slapen

Er kunnen vele redenen zijn waarom een proces gaat slapen: wanneer een proces op iets wacht en die wachttijd lang kan zijn, zal het slapen totdat hetgeen het op wachtte beschikbaar is. Terwijl het slaapt zal een ander proces uitgevoerd worden. Wanneer een proces in kernelmode of *user context* is en een *spinlock* vasthoudt, mag het niet slapen. Dit is omdat het lang kan duren voordat het proces terug aan bod komt en dat het *lock* daarom voor onbepaalde duur kan vastgehouden worden, waardoor een andere processor bv. constant aan het spinnen is om het *lock* op bezet te zetten. De functie `vmalloc()`, gebruikt voor het alloceren van virtueel kernelgeheugen, is zo'n functie die ervoor kan zorgen dat het proces slaapt. Het alloceren van geheugen met `vmalloc()` moet daarom altijd buiten kritieke secties die met een kernellock worden beschermd, gebeuren.

4.2.5 Interrupts en locking

Het komt veel voor in de kernelcode dat eenzelfde *lock* zowel in *user context* als in een *softirq* of *interrupt* kan worden gebruikt (zie Hoofdstuk 5). Wanneer een proces in *user context* het *lock* wenst te nemen moet het ervoor zorgen dat het niet kan onderbroken worden door een *interrupt* of *softirq* die zelf ook het *lock* wenst te nemen, anders ontstaat er nl. een *deadlock*. Inderdaad, het *lock* werd zojuist vastgenomen door het proces in *user context*, waarna een interrupt dit proces onderbrak en de interruptcode nu ook probeert het *lock* vast te nemen. Deze interruptcode zal nu oneindig lang wachten totdat het *lock* wordt vrijgegeven, aangezien enkel het proces dat in *user context* werd onderbroken het *lock* kan vrijgeven.

De oplossing is om tijdelijk de *interrupts* of de *softirq*'s uit te schakelen.

Hieronder volgen als voorbeeld de functie `write_lock()` en z'n drie varianten. De functie neemt een *write lock* vast op een *"read/write"-spinlock*, zie *include/linux/spinlock.h*. Merk op dat `write_lock()` en z'n varianten eigenlijk *preprocessor macro*'s zijn.

```
#define write_lock(lock) \
do { \
  preempt_disable(); \
  _raw_write_lock(lock); \
} while(0)

#define write_lock_irqsave(lock, flags) \
do { \
  local_irq_save(flags); \
  preempt_disable(); \
  _raw_write_lock(lock); \
} while (0)

#define write_lock_irq(lock) \
do { \
  local_irq_disable(); \
  preempt_disable(); \
  _raw_write_lock(lock); \
} while (0)

#define write_lock_bh(lock) \
do { \
  local_bh_disable(); \
  preempt_disable(); \
  _raw_write_lock(lock); \
} while (0)
```

De functie `_raw_write_lock()` is een architectuurspecifieke *inline* functie die het *write lock* vastneemt. De nog niet behandelde functies uit de bovenstaande code bevinden zich in *include/asm-i386/system.h* en *include/asm-i386/softirq.h*:

```
#define local_irq_save(x) __asm__ __volatile__("pushfl ; popl %0 ; cli":"=g"
(x):/* no input */ :"memory")
#define local_irq_disable() __asm__ __volatile__("cli": : :"memory")
#define local_bh_disable() \
    do { preempt_count() += SOFTIRQ_OFFSET; barrier(); } while (0)
```

De macro `local_irq_save` doet het volgende: eerst wordt de waarde van het vlaggenregister van de processor op de stack *gepusht*, dan wordt deze waarde terug *gepopt* in de variabele `x` en tenslotte wordt de `cli` assembler instructie uitgevoerd. De `cli` instructie schakelt de interrupts uit voor de specifieke processor. De macro `write_unlock_irqrestore` kan dan achteraf worden gebruikt om de interrupts terug aan te zetten en het vlaggenregister te herstellen naar de waarde voor we het lock namen (die opgeslaan is in de variabele `x`).

De macro `local_irq_disable` schakelt enkel de interrupts uit.

De macro `local_bh_disable` verhoogt de `preempt_count` van het proces die deze macro oproept met `SOFTIRQ_OFFSET`, wat eigenlijk gewoon als resultaat heeft dat minstens 1 bit van het *softirqmask* van de `preempt_count` op 1 staat, waardoor de macro `in_softirq` zeker een waarde verschillend van nul teruggeeft (zie *include/asm-i386/hardirq.h*):

```
#define SOFTIRQ_BITS 8
#define SOFTIRQ_SHIFT (PREEMPT_SHIFT + PREEMPT_BITS)
#define __MASK(x) ((1UL << (x))-1)
#define SOFTIRQ_MASK (__MASK(SOFTIRQ_BITS) <<
SOFTIRQ_SHIFT)
#define in_softirq() (softirq_count())
#define softirq_count() (preempt_count() & SOFTIRQ_MASK)
```

Hierboven gebeurt leuke binaire arithmetiek die je misschien eens van naderbij moet bestuderen (bemerk de `-1` in de macro `_MASK`)...

De definitie van `barrier` is te vinden in *include/asm-i386/hardirq.h*:

```
#define barrier() __asm__ __volatile__("": : :"memory")
```

Dit zegt gewoon dat er moet gewacht worden tot de schrijfacties naar het geheugen voltooid zijn vooraleer aan de volgende instructie te beginnen. Inderdaad, de *softirq*'s zijn pas echt uitgeschakeld als de optelling werkelijk naar het geheugen is geschreven.

Hoofdstuk 5

Interrupts

Interrupts zijn een voorbeeld van dingen waarmee een applicatieprogrammeur zelden mee in contact komt, terwijl ze alomtegenwoordig zijn voor de systeemprogrammeur. In dit hoofdstukje zal het afhandelen van *interrupts* door de kernel worden beschreven, daarnaast zullen enkele begrippen die nauw samenhangen met *interrupts* van naderbij worden bekeken.

5.1 Hardware interrupts

Elke processor voorziet het mechanisme van *hardware interrupts*. Deze zorgen ervoor dat het communiceren met randapparaten, zoals de harde schijf en de netwerkkaart efficiënt kan gebeuren.

Een *interrupt* zorgt ervoor dat de taak die een processor op het moment van de *interrupt* aan het doen is, tijdelijk wordt opgeschort voor het afhandelen van de *interrupt*. Zowel een gebruikersprogramma als de kernel zelf kan door een *interrupt* onderbroken worden. Via de x86 assemblerinstructies `sti` en `cli` kan de kernel wel tijdelijk de *interrupts* uitschakelen, zie verder.

Zonder in detail te gaan is een *interrupt* op hardwareniveau niets anders dan een signaal dat via de daartoe bestemde processorpinnen binnenkomt in de processor. De processor merkt dit en zal zijn huidige taak opschorten en een softwareroutine die zich op een specifiek adres bevindt in het geheugen

uitvoeren. Deze onderbreking gebeurt dus autonoom door de processorhardware, de enige manier om zo'n onderbreking te voorkomen is door de *interrupts* tijdelijk uit te schakelen.

Waarmee applicatieprogrammeurs wel te maken krijgen zijn zogenaamde excepties (*exceptions*). Wanneer een deling door nul gebeurt, wordt de huidige taak ook opgeschort door de processor en wordt een routine uitgevoerd die zegt wat er moet gebeuren bij deze fout. Deze routine kan door de applicatie zelf worden gedefinieerd. In wezen verschillen excepties en *interrupts* niet veel van elkaar, behalve dat excepties zogenaamd synchroon zijn en *interrupts* asynchroon. Door het programma terug te laten lopen en dezelfde invoer te geven kan een exceptie gereproduceerd worden, terwijl *interrupts* onvoorspelbaar zijn. Een programma kan ook moedwillig een exceptie genereren wanneer het met de kernel wenst te communiceren. Op de x86-processors gebeurt dit met de assemblerinstructie `int 0x80`. Dit wordt de *software interrupt* genoemd.

5.1.1 Codepad naar `do_IRQ()`

Wanneer een interrupt binnenkomt op een x86-processor wordt deze afgehandeld in de functie `do_IRQ()` die te vinden is in *arch/i386/kernel/irq.c*. Hier bekijken we wat er precies gedaan wordt opdat `do_IRQ()` een *interrupt* kan afhandelen. In de volgende sectie bekijken we dan deze *low-level*-code van dichterbij.

Interrupts worden door de x86-processorhardware afgehandeld via een IDT (*interrupt descriptor table*). Deze tabel bevat voor elke mogelijke interruptwaarde een *entry*, in totaal 256. Het bestand *arch/i386/kernel/head.S* bevat de assemblercode die de geheugenplaats voorziet voor deze IDT, deze IDT initialiseert met standaardwaarden en de processor de plaats van de IDT in het geheugen duidelijk maakt. De excepties, *interrupts* en de *software interrupt* worden afgehandeld via deze IDT. De IDT tabelentries voor de excepties en *software interrupt* worden later ingevuld met bruikbare waarden, dit gebeurt in de functie `trap_init()` in het bestand *arch/i386/kernel/traps.c*.

De entries voor de *interrupts* krijgen een goede initiële waarde in de functie `init_IRQ()` van *arch/i386/kernel/i8259.c* (de i8259 is hardware van Intel, het is een *programmable interrupt controller*). Een normaal systeem kan hoogstens 16 *interrupts* hebben, terwijl een systeem met de *Intel APIC interrupt controller* (een wat meer uit de kluiten gewassen *controller*) er hoogstens 224 kan hebben. Merk op dat de initialisatie van de IDT in `init_IRQ()` er nog niet voor zorgt dat bv. een *driver* voor een netwerkkaart is geregistreerd voor de *interrupts* behorende bij deze netwerkkaart. Merk ook op dat indien er meer randapparaten aanwezig zijn dan interruptwaarden, er interruptwaarden zullen moeten worden gedeeld door verschillende randapparaten.

5.1.2 Initialisatie van de IDT

Het opstellen van de IDT gebeurt in *arch/i386/kernel/head.S* (dit bestand bevat de *low-level* opstartcode voor de x86). Hier volgen enkele extracten.

```
/*
 * The IDT and GDT 'descriptors' are a strange 48-bit object
 * only used by the lidt and lgdt instructions. They are not
 * like usual segment descriptors - they consist of a 16-bit
 * segment size, and 32-bit linear address value:
 */
.globl idt_descr
.globl cpu_gdt_descr
        ALIGN
        .word 0 # 32-bit align idt_desc.address
idt_descr:
        .word IDT_ENTRIES*8-1 # idt contains 256 entries
        .long idt_table
```

De bovenstaande assemblercode zorgt voor de creatie van 256 IDT entries. In wat volgt knippen we enkele codefragmenten, die relevant zijn voor onze discussie over *interrupts*, uit de opstartcode van de x86. In de initialisatiecode gebeurt natuurlijk vanalles (sequentieel), op een bepaald moment gebeurt volgende oproep:

```
/*
 * start system 32-bit setup. We need to re-do some of the things done
 * in 16-bit mode for the "real" operations.
 */
        call setup_idt
```

In de initialisatiecode wordt naar de subroutine `setup_idt` gesprongen. Deze subroutine initialiseert de IDT zodat alle interrupts zullen afgehandeld worden met de `ignore_int` subroutine. Deze subroutine doet eigenlijk niets, behalve het proberen uitschrijven van "*Unknown interrupt*" naar de console en/of het logsysteem. De functie `printk()` is een C-functie, die dienst doet als de `printf()` van de gewone userspace programma's. De functie `printf()` bestaat niet voor de kernel. De subroutine `setup_idt` wordt hieronder weergegeven.

Na de aanroep van die subroutine volgt wat later de volgende assemblerinstructie:

```
lidt idt_descr
```

De `lidt` assemblerinstructie wordt uitgevoerd enkele instructies voordat een aanroep van `start_kernel()` (*init/main.c*) of `initialize_secondary()` (*arch/i386/kernel/smpboot.c*) gebeurt (zie Hoofdstuk 6). Deze instructie laadt het IDT register met het adres van de IDT tabel, we zien dat dit inderdaad `idt_descr` is. Het IDT register wordt door de hardware gebruikt als adres van de IDT tabel, het interruptnummer wordt als index in de IDT tabel gebruikt.

De twee subroutines `setup_idt` en `ignore_int` bevinden zich wat lager in het bestand *arch/i386/kernel/head.S*. We tonen ze hieronder.

```
/*
 * setup_idt
 *
 * sets up a idt with 256 entries pointing to
 * ignore_int, interrupt gates. It doesn't actually load
 * idt - that can be done only after paging has been enabled
 * and the kernel moved to PAGE_OFFSET. Interrupts
 * are enabled elsewhere, when we can be relatively
 * sure everything is ok.
 */
setup_idt:
        lea ignore_int,%edx
        movl $(__KERNEL_CS << 16),%eax
        movw %dx,%ax /* selector = 0x0010 = cs */
        movw $0x8E00,%dx /* interrupt gate - dpl=0, present */
        lea idt_table,%edi
        mov $256,%ecx
rp_sidt:
        movl %eax,(%edi)
        movl %edx,4(%edi)
        addl $8,%edi
        dec %ecx
        jne rp_sidt
        ret
```

Een *interrupt descriptor* bestaat uit 8 bytes. Het naslaan van [20] geeft ons het uitzicht van zo'n *descriptor* en dit stemt inderdaad overeen met de assemblercode die hierboven staat. De eerste 2 bytes beslaan de *segment selector* (zie Hoofdstuk 8. Voor interruptafhandeling wordt deze op `__KERNEL_CS` gezet(zie *include/asm-i386/segment.h*). Dit zorgt ervoor dat in de GDT (zie sectie 8.4.1) wordt gekeken om het logische adres te vertalen. De volgende 2 bytes bepalen de laatste 2 bytes van de offset, dit wordt gedaan door de derde assemblerinstructie. Het register `eax` zal dus de eerste 8 bytes bevatten. De 2 volgende bytes bevatten de eerste 2 bytes van de offset. De laatste 2 bytes geven eigenschappen aan de *interrupt descriptor*. De waarde `0x08E00` komt overeen met deze eigenschappen: P-bit (Present bit) op 1 (dit is altijd zo in Linux), de DPL (*Descriptor Privilege Level*, waarden 0-3) wordt op 0 gezet, wat de interruptafhandelingscode het grootste privilege geeft, de *interrupt descriptor* beschrijft een *interrupt gate* (negeer dit gerust). Het register `edx`

bevat deze laatste 8 bytes. Nadat `eax` en `edx` goed zijn ingevuld, worden de 256 IDT entries hiermee geïnitialiseerd. Dit komt erop neer dat wanneer later de interrupts worden aangeschakeld, de subroutine `ignore_int` zal worden uitgevoerd wanneer een *interrupt* wordt ontvangen (zolang geen echte *interrupt handler* is geregistreerd voor die *interrupt*).

```
        /* This is the default interrupt "handler":-) */
int_msg:
        .asciz "Unknown interrupt\n"
        ALIGN
ignore_int:
        cld
        pushl %eax
        pushl %ecx
        pushl %edx
        pushl %es
        pushl %ds
        movl $(__KERNEL_DS),%eax
        movl %eax,%ds
        movl %eax,%es
        pushl $int_msg
        call printk
        popl %eax
        popl %ds
        popl %es
        popl %edx
        popl %ecx
        popl %eax
        iret
```

We zien hier dat eerst de nodige registers worden bewaard op de stack, daarna wordt het kerneldatasegment (zie Hoofdstuk 8) in de registers `ds` (het datasegment register) en `es` geladen, een pointer naar de string `int_msg` (die luidt "Unknown interrupt\n") wordt op de stack gezet en de kernelfunctie `printk()` wordt opgeroepen om de boodschap weer te geven.

Later in de initialisatie van de kernel wordt de IDT tabel ingevuld met bruikbare data. Voor de excepties en de *software interrupt* gebeurt dit in *arch/i386/kernel/traps.c* in de functie `trap_init()`, waarvan hier een deel

wordt getoond:

```
void __init trap_init(void)
{
        /* snip */
        set_trap_gate(0,&divide_error);
        set_trap_gate(1,&debug);
        /* enzovoort */
        set_system_gate(SYSCALL_VECTOR,&system_call);
        /* snip */
}
```

Naslag van een boek over de x86 leert ons dat interrupt waarde 0 staat voor de deelfout exception. De subroutine `divide_error` is gedefinieerd in *arch/i386/kernel/entry.S*. De waarde van `SYSCALL_VECTOR` is niet toevallig gedefinieerd als `0x80` (de *software interrupt*). Na deze initialisatie worden alle excepties en de *software interrupt* reeds goed afgehandeld. De entries van de externe *interrupts* worden in *arch/i386/kernel/i8259.c* geïnitialiseerd op bruikbare waarden, in de functie `init_IRQ()`, waarvan hieronder een extract staat:

```
void __init init_IRQ(void)
{
  int i;
  /*
   * Cover the whole vector space, no vector can escape
   * us. (some of these will be overridden and become
   * 'special' SMP interrupts)
   */
  for (i = 0; i < NR_IRQS; i++) {
    int vector = FIRST_EXTERNAL_VECTOR + i;
    if (vector != SYSCALL_VECTOR)
      set_intr_gate(vector, interrupt[i]);
  }
```

De waarde voor `FIRST_EXTERNAL_VECTOR` is gedefinieerd als `0x20`, dus 32. Naslag van een boek over de x86 leert ons dat interruptnummers 32-255 beschikbaar zijn voor externe *interrupts*. `NR_IRQS` is als 16 of 224 gedefinieerd,

bij een alledaags systeem als 16. De functie `set_intr_gate()` bevindt zich in *arch/i386/kernel/traps.c* en vult de IDT tabel op plaats `vector` in met de pointer naar `interrupt`. Het (wederom) naslaan van een boek over de x86 leert ons dat *interrupts* door *interrupt-gates* er automatisch voor zorgen (via de hardware) dat de interruptvlag wordt uitgezet, zodat de *interrupts* tijdelijk zijn uitgeschakeld. Deze zal dus later terug moeten worden aangezet. De array `interrupt` is de interrupttabel, die te vinden is in *arch/i386/kernel/entry.S* wordt hieronder ingevuld:

```
/*
 * Build the entry stubs and pointer table with
 * some assembler magic.
 */
.data
ENTRY(interrupt)
.text

vector=0
ENTRY(irq_entries_start)
.rept NR_IRQS
        ALIGN
1: pushl $vector-256
        jmp common_interrupt
.data
        .long 1b
.text
vector=vector+1
.endr
```

info gas leert ons dat `.rept` *gas magic* is om iets een aantal maal te doen bij het compileren. De bovenstaande code maakt een array van telkens twee instructies aan. De eerste instructie op index `vector` zet de waarde `vector-256` op de stapel, waarvoor dit dient zien we straks. De tweede instructie springt naar de subroutine `common_interrupt`, die we zo meteen bekijken. Het is deze subroutine die de functie `do_IRQ()` uitvoert. De addressen door `init_IRQ()` meegegeven aan `set_intr_gate()` (zie boven) wijzen in bovenstaande array. Zo zal de entry op index 32 van de IDT wijzen

naar de code op index 0 in de interruptarray hierboven.

5.1.3 Interruptafhandeling

We bekijken nu de subroutine `common_interrupt` (in *arch/i386/kernel/entry.S*) en het begin van de functie `do_IRQ()` (in *arch/i386/kernel/irq.c*), beginnende met laatstgenoemde:

```
asmlinkage unsigned int do_IRQ(struct pt_regs regs)
```

De headerfile *include/asm-i386/linkage.h* definieert `asmlinkage` als volgt (`CPP_ASMLINKAGE` is gedefinieerd in *include/linux/linkage.h*):

```
#define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))

#ifdef __cplusplus
#define CPP_ASMLINKAGE extern "C"
#else
#define CPP_ASMLINKAGE
#endif
```

De `extern "C"` duidt gewoon aan dat de functie een C functie is, dit is nodig indien de compiler een C++ compiler is, aangezien een C++ functieaanroep anders wordt geïmplementeerd dan de aanroep van een C functie. Deze `extern "C"` is een zogenaamde *linkage directive* en is standaard C++ taal. Waarom iemand ooit de Linux kernel met een C++ compiler zou willen compileren is ons niet bekend.

Interessanter is `__attribute__((regparm(0)))`. Dit is pure *gcc magic* (dus incompatibel met andere compilers), waarover meer te vinden is via `info gcc`. Met het *keyword* `attribute` kunnen speciale attributen worden gespecifieerd. In het geval hierboven wordt `regparm(0)` als attribuut gegeven, dit zorgt ervoor dat geen enkel argument van de functie via de registers wordt meegegeven: de compiler zorgt ervoor dat alle argumenten op de stack worden gezet. Anders zou de compiler kunnen opteren voor optimalisatie via registers. Waarom dit attribuut nodig is voor de functie `do_IRQ()` wordt

duidelijk als we naar wat code uit *arch/i386/kernel/entry.S* kijken:

```
#define SAVE_ALL \
  cld; \
  pushl %es; \
  pushl %ds; \
  pushl %eax; \
  pushl %ebp; \
  pushl %edi; \
  pushl %esi; \
  pushl %edx; \
  pushl %ecx; \
  pushl %ebx; \
  movl $(__USER_DS), %edx; \
  movl %edx, %ds; \
  movl %edx, %es;
<snip>
common_interrupt:
  SAVE_ALL
  call do_IRQ
  jmp ret_from_intr
```

We zien dat vooraleer `do_IRQ()` wordt aangeroepen (via de assembler instructie `call`), de macro `SAVE_ALL` wordt uitgevoerd. Deze macro zet via `pushl` assembler instructies een heleboel registers op de stack (wat de `movl` en `cld` instructies precies doen negeren we hier). Een lichtje begint te branden wanneer we naar de definitie van `struct pt_regs` (het datatype van het argument van de functie `do_IRQ()`) kijken, die te vinden is in *include/asm-i386/ptrace.h*:

```
/* this struct defines the way the registers are stored on the
   stack during a system call. */
struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int xds;
        int xes;
        long orig_eax;
        long eip;
        int xcs;
        long eflags;
        long esp;
        int xss;
};
```

We zien dat het laatste register dat op de stack werd gezet de eerste member van de `struct pt_regs` is. Wat we hier onrechtstreeks kunnen uit afleiden is dat de stack naar beneden groeit. Stacks groeien meestal naar beneden, en als we de beschrijving van de `pushl` assemblerinstructie van de x86 erop naslaan zien we dat deze assemblerinstructie de stackpointer (het register `esp`) inderdaad verlaagt.

Eventjes wat meer uitleg over de members van de `struct pt_regs`. De laatste 5 members worden door de x86 processorhardware op de stack gezet wanneer een *interrupt* toekomt. De member `orig_eax` is de waarde `(vector - 256)` die we tegenkwamen in het codefragment dat naar de subroutine `common_interrupt` springt. De members hiervoor zijn net deze die door de macro `SAVE_ALL` op de stack worden gezet.

Aangezien alle elementen van de `struct pt_regs` van het argument van `do_IRQ()` op de stapel worden gezet, moet ervoor gezorgd worden dat de C compiler geen optimalisatie via registers doet. De C compiler weet nl. niet dat de assembler code in *arch/i386/kernel/entry.S* de argumenten op de

stack zet. Vandaar dus `asmlinkage`.

Nu bekijken we het nut van de waarde `(vector - 256)` die op de stapel wordt gezet, dit komt overeen met de member `orig_eax` van het argument van `do_IRQ()` (zie hiervoor). Het begin van `do_IRQ()` ziet er zo uit:

```
asmlinkage unsigned int do_IRQ(struct pt_regs regs)
{
<snip>
  int irq = regs.orig_eax & 0xff; /* high bits used in ret_from_ code */
<snip>
}
```

Hier zien we een staaltje van bitarithmetiek: de waarde `(vector - 256)` wordt omgezet via een logische `and` met `0xff` naar het interruptnummer met waarde `vector`. Zoals de code aangeeft worden de meer significante bits elders gebruikt, waarschijnlijk daarom dat de initialisatie `(vector - 256)` gebruikt.

Een *device driver* zal zich registreren op een *interrupt* via de functie `request_irq()` in *arch/i386/kernel/irq.c*, deze functie zet de juiste waarden in de array `irq_desc_t irq_desc[NR_IRQS]`. Deze array wordt in `do_IRQ()` gebruikt om de *handler* voor de *interrupt* te vinden.

5.2 Software interrupts, system calls

De termen *software interrupt* en *system call* betekenen hetzelfde en worden hier door elkaar gebruikt. Wanneer een gebruikersprogramma een functionaliteit van de kernel wenst te gebruiken, wordt dit gedaan via een *software interrupt*. Applicatieprogrammeurs moeten deze *software interrupts* normaal nooit zelf gebruiken, het zijn de bibliotheekfuncties die dit voor ze doen. De broncode van de GNU C bibliotheek kan worden bekomen via [7] (bv. *glibc-2.3.tar.bz2*), deze bibliotheek bevat de broncode voor functies zoals `printf()` en `setuid()`. We bekijken even de (een beetje aangepaste) implementatie van de functie `setuid()`, samen met het gebruik van de *software interrupt*.

De twee relevante glibc-bestanden zijn *sysdeps/unix/sysv/linux/i386/setuid.c* en *sysdeps/unix/sysv/linux/i386/sysdep.h*.

```
/* Define a macro which expands inline into the wrapper code for a system
  call. */
#undef INLINE_SYSCALL
#define INLINE_SYSCALL(name, nr, args...) \
 ({ \
  unsigned int resultvar; \
  asm volatile ( \
  LOADARGS_##nr \
  "movl %1, %%eax\n\t" \
  "int $0x80\n\t" \
  RESTOREARGS_##nr \
  : "=a" (resultvar) \
  : "i" (__NR_##name) ASMFMT_##nr(args) : "memory", "cc"); \
  if (resultvar >= 0xfffff001) \
   { \
  __set_errno (-resultvar); \
  resultvar = 0xffffffff; \
   } \
  (int) resultvar; })

int setuid (uid_t uid)
{
 return INLINE_SYSCALL (setuid32, 1, uid);
}
```

De macro `INLINE_SYSCALL` zet de argumenten op de stack, het nummer van de functie wordt in het register `eax` gestoken en de *software interrupt* wordt aangeroepen. In het alomvertrouwde bestand *arch/i386/kernel/entry.S* vinden we de kernelafhandelingscode terug voor de *system call* (zie ook het stuk over `trap_init()` eerder in dit hoofdstuk):

```
ENTRY(system_call)
  pushl %eax # save orig_eax
  SAVE_ALL
  GET_THREAD_INFO(%ebp)
  cmpl $(NR_syscalls), %eax
  jae syscall_badsys
          # system call tracing in operation
  testb $_TIF_SYSCALL_TRACE,TI_FLAGS(%ebp)
  jnz syscall_trace_entry
syscall_call:
  call *sys_call_table(,%eax,4)
  movl %eax,EAX(%esp) # store the return value
<snip>
.data
ENTRY(sys_call_table)
  .long sys_restart_syscall /* 0 - old "setup()" system call, used for restarting */
  .long sys_exit
  .long sys_fork
  .long sys_read
  .long sys_write
<snip>
```

De meerderheid van deze code negeren we hier. Wat we zien is dat de functie behorende bij het nummer dat zich in het `eax`-register bevindt wordt uitgevoerd, dit door het adres van de functie te nemen dat zich op index `eax` in de `sys_call_table` array bevindt. Deze array wordt in hetzelfde bestand gedefinieerd.

De headerfile *include/asm-i386/unistd.h* definieert `__NR_setuid32` als 213. De entry op index 213 van de array `sys_call_table` verwijst naar de functie `sys_setuid()`, die gedefinieerd is in *kernel/sys.c.*

Een foutboodschap van kernelfuncties die een *system call* afhandelen is altijd een negatieve waarde, meerbepaald de tegengestelde waarde van de echte errorwaarde, zo is één van de mogelijke returnwaarden van `sys_setuid()` de waarde `-EPERM`, wat aanduidt dat het gebruikersprogramma niet de juiste permissies heeft om zijn id te veranderen. Als we terugkijken naar de macro `INLINE_SYSCALL`, zien we dat indien deze returnwaarde tussen -4095 en -1

ligt, de variabele `errno` (zie `man errno`) het tegengestelde van deze waarde krijgt en er -1 wordt teruggegeven. Anders wordt de waarde zelf teruggegeven. De macro `INLINE_SYSCALL` geeft dus een waarde terug, dit komt omdat een codeblok (iets dat tussen { en } staat) altijd een waarde teruggeeft in C, nl. de waarde van het laatst uitgevoerde statement. In bovenstaande geval is dit `(int) resultvar`.

5.3 Bottom halves: tasklets en softirq's

Deze sectie bevat gegevens gehaald uit het recente document getiteld "*I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers*" en gepubliceerd door Matthew Wilcox, te vinden op [28]. In de `2.4` kernel heeft een *bottom halve* een ietwat andere betekenis dan in de `2.5` kernel. *Bottom halves* zijn in de `2.4` kernel een restrictieve vorm van *tasklets*, maar in de `2.5` kernel zijn de *bottom halves* verwijderd en wordt deze term nu gebruikt als verzamelnaam voor *tasklet* en *softirq*. In wat volgt bekijken we de implementatie van de *softirq*.

Wanneer data beschikbaar wordt via een randapparaat en dit gesignaleerd wordt met een *interrupt*, moet er soms veel werk verricht worden. Wanneer een IP-pakket (een klein pakketje data verzonden over een netwerk zoals het Internet) binnenkomt op een netwerkkaart, kunnen er potentieel duizenden lijnen C-code worden uitgevoerd voor dit pakket. Indien dit alles zou gebeuren terwijl de *interrupts* zijn uitgeschakeld voor de cpu die deze *interrupt* afhandelt, zouden *interrupts* kunnen verloren gaan, zouden andere randapparaten kunnen verhongeren en zou de responstijd van de computer te traag worden. De oplossing hiervoor is de *interrupt handler* enkel het meest broodnodige werk te laten doen en het niet hoogdringende werk te laten doen in een *bottom half*. Via de macro `__cpu_raise_softirq` (*include/linux/interrupt.h*) wordt aangeduid dat er werk te doen is voor een *softirq*:

```
/* de eerste 2 lijnen komen uit include/linux/irq_cpustat.h */
#define __IRQ_STAT(cpu, member) (irq_stat[cpu].member)
#define softirq_pending(cpu) __IRQ_STAT((cpu), __softirq_pending)
#define __cpu_raise_softirq(cpu, nr) \
  do { softirq_pending(cpu) |= 1UL << (nr); } while (0)
```

Hierin is `irq_stat` van het type `irq_cpustat_t`, wat een `struct` is die o.a. de *member* `__softirq_pending` bevat. Wat we zien is dat de macro `__cpu_raise_softirq` de bit van deze *member* voor de gegeven cpu op 1 zet.

Op specifieke plaatsen in de kernel wordt gekeken of `softirq_pending(cpu) != 0` en indien zo wordt de functie `do_softirq()` uitgevoerd (*kernel/softirq.c*). Deze functie bekijkt de bits van de hiervoor bekeken *member* `__softirq_pending` één voor één en indien een bit gelijk is aan 1 wordt de bijhorende *softirq* uitgevoerd.

De *inline* functie `__netif_rx_schedule()` (*include/linux/netdevice.h*) is zo'n functie die een softirq soms aanzet. Deze functie wordt soms opgeroepen tijdens de afhandeling van een *interrupt* van een netwerkkaart. Wanneer er genoeg netwerkpakketten binnengekomen zijn, m.a.w. wanneer de wachtlijn voor de pakketten vol zit, zal deze functie de *softirq* activeren, waardoor de netwerksoftirq de verdere afhandeling van deze pakketten zal verzorgen. Deze logica bevindt zich in de functie `netif_rx()` (*net/core/dev.c*), die de pakketten krijgt van de *device drivers* van de netwerkkaart in kwestie. In de interrupt handler worden de pakketten dus enkel *gequeued* (in een wachtlijn gestoken). Deze wachtlijn wordt dan later door de netwerksoftirq verwerkt. De verwerking van inkomende netwerkpakketten gebeurt door de functie `net_rx_action()` (*net/core/dev.c*), die bij de algemene *softirq*-code is geregistreerd via de initialisatiefunctie (bemerk het `__init` *tag*, zie Hoofdstuk 6) `net_dev_init()` (*net/core/dev.c*) van de algemene netwerkcode:

```
static int __init net_dev_init(void)
{
<snip>
  open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
  open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
<snip>
```

We bekijken nu de algemene *softirq*-code van dichterbij. De *softirq*-afhandelingsfunctie heet `do_softirq()` (*kernel/softirq.c*):

```
static struct softirq_action softirq_vec[32] __cacheline_aligned_in_smp;
<snip>
asmlinkage void do_softirq()
{
<snip>
  pending = softirq_pending(cpu);
<snip>
    h = softirq_vec;
    do {
      if (pending & 1)
        h->action(h);
      h++;
      pending >>= 1;
    } while (pending);
<snip>
}
```

Aangezien de functie door assemblerroutines kan opgeroepen worden heeft ze het *tag* `asmlinkage` (zie eerder). We zien dat het maximaal aantal verschillende softirq's *hardcoded* is op 32. We zien dat de code alle mogelijke *softirq*'s afloops en kijkt of de *softirq* moet uitgevoerd worden en deze indien nodig dan uitvoert.

Hierboven hebben we gezien dat de netwerksoftirq's worden geregistreerd via de functie `open_softirq()` (*kernel/softirq.c*):

```
void open_softirq(int nr, void (*action)(struct softirq_action*), void *data)
{
  softirq_vec[nr].data = data;
  softirq_vec[nr].action = action;
}
```

We zien dat deze functie niets meer doet dan een element van de *softirq*-array invullen met de juiste waarden.

Een plaats waar `do_softirq()` wordt opgeroepen is bv. op het einde van de functie `do_IRQ()`, die we al eerder tegenkwamen.

Hoofdstuk 6

Initialisatie van de kernel

Zie het "*Linux Kernel 2.4 internals*" -document (en de broncode natuurlijk) voor meer details [26].

Wanneer we een computer aanzetten, moet de hardware natuurlijk de eerste stappen zetten. Het *boot proces* is architectuurafhankelijk, we beperken ons tot de IBM PC architectuur. Door het oude design en de vereiste achterwaartse compatibiliteit wordt het besturingssysteem door de PC *firmware* op een ouderwetse manier *geboot*. Dit proces kan opgesplitst worden in zes logische stappen:

- het BIOS selecteert een *boot device*,
- het BIOS laadt de bootsector van het *boot device*,
- de bootsector laadt de setup en decompressieroutines en het gecomprimeerde *kernel image*,
- de kernel wordt gedecomprimeerd in "*protected mode*",
- de *low-level* initialisatie gebeurt door assemblercode,
- de *high-level* initialisatie in C code begint.

We bekijken de *high-level* initialisatie van dichterbij, deze begint met *arch/i386/kernel/head.S*, wat eigenlijk nog in assembler is. Elke cpu zal de

code in dit bestand uitvoeren. De hardware heeft de eerste cpu (de BP, *boot processor*) opgestart, deze zal het grootste deel van het werk doen. De eventuele andere cpu's zijn nu nog niet actief (dit wordt verhinderd door de hardware van de smp-machine). De code weet of ze bezig is met iets uit te voeren voor de BP of niet, door te kijken naar het `bx`-register, dat op nul staat voor de BP en op 1 voor een andere cpu. Herinner je dat elke cpu zijn eigen registers heeft. Het spreekt voor zich dat bepaalde initialisatiecode enkel door de BP zal worden uitgevoerd.

De BP doet allerlei *low-level* initialisatie, zoals de paginatabellen initialiseren, paginering aanzetten (zie Hoofdstuk 8), de *interrupt discriptor table* (zie Hoofdstuk 5) initialiseren (dit alles in assembler), en voert uiteindelijk *init/main.c*::`start_kernel()` uit: de eerst uitgevoerde C-functie. De andere cpu's zullen `initialize_secondary()` uitvoeren, wat de *idle thread* van die cpu zal opstarten (zie straks). Wanneer de BP `start_kernel()` begint uit te voeren, is er nog altijd geen andere cpu actief.

```
asmlinkage void __init start_kernel(void)
```

In de functie `start_kernel()` worden allerhande initialisatiefuncties opgeroepen. De *interrupts* worden ook aangezet. Bijna op het einde volgt een oproep van *kernel/sched.c*::`init_idle()`, waardoor de code die nog volgt de *idle thread* van de BP wordt. Als laatste wordt `rest_init()` uitgevoerd, deze maakt een nieuwe *kernel thread* aan, de *init thread*. Deze *init thread* begint de uitvoering met *init/main.c*:`init()`. Aangezien *scheduling* al geïnitialiseerd is en de *interrupts* al aan staan, zullen deze twee threads al beide kunnen lopen op de BP. De *idle thread* voert hierna de functie *arch/i386/kernel/process.c*::`cpu_idle()` uit, deze functie keert nooit terug. `cpu_idle()` doet standaard werkelijk niets, behalve kijken of de computer moet *gehalt* worden en de *scheduler* aanroepen (zie Hoofdstuk 8). De schedulingprioriteit van deze *idle thread* is zodanig laag dat de *scheduler* deze enkel zal laten uitvoeren als er niets anders moet gedaan worden. Doordat er altijd een thread uitgevoerd wordt, maakt dit de schedulingcode veel eenvoudiger dan wanneer er met het speciale geval

dat er geen werk is zou moeten rekening worden gehouden.

Nu dus de *init thread*. Hierin worden de `initcalls` opgeroepen waarna een `free_initmem()` volgt. De `initcalls` zijn alle functies die een `__init`-*tag* hebben. Bij compilatie wordt een array met pointers naar die functies aangemaakt. Nu worden deze functies één voor één opgeroepen. Nadat dit gebeurd is zullen deze functies nooit meer gebruikt worden, daarom verwijdert `free_initmem()` het geheugen dat door deze functies werd gebruikt. Bij de compilatie wordt, via *gcc magic*, nl. gezorgd dat al deze functies in eenzelfde aaneensluitend geheugengebied liggen, waardoor dit dus makkelijk kan vrijgemaakt worden nadien. Merk op dat dit een mooie toepassing is van het Von Neumann principe.

De eventuele andere cpu's worden gestart door de *init thread*. Het codepad dat deze andere cpu's volgen is hetzelfde als het pad dat door de BP is doorlopen, behalve dan dat sommige dingen niet meer moeten gedaan worden. Deze cpu's voeren dus ook uiteindelijk `cpu_idle()` uit, elke cpu heeft dus zijn eigen *idle thread*. Terwijl de eventuele andere cpu's hun opstartcode aan het uitvoeren zijn, zal de *init thread* het programma init proberen te vinden vanop de harde schijf (zie *init/main.c*):

```
execve(/sbin/init",argv_init,envp_init);
execve(/etc/init",argv_init,envp_init);
execve(/bin/init",argv_init,envp_init);
execve(/bin/sh",argv_sh,envp_init);
panic("No init found.  Try passing init= option to kernel.");
```

Indien de functie `execve()` succesvol wordt uitgevoerd, zal ze nooit terugkeren. `execve()` zorgt er in dit geval voor dat het programma, bv. /sbin/init, in de *init thread* wordt uitgevoerd. De functie `panic()` wordt dus uitgevoerd indien geen enkel geldig init-programma wordt gevonden. Dit zorgt voor het verschijnen van de alom gevreesde *kernel panic* op het scherm.

Normaal gezien wordt dus het *init proces* geladen van de schijf en uitgevoerd. De Linuxkernel is nu volledig *up and running*, de eventuele andere cpu's zijn met hun initialisatie bezig (of hebben al gedaan). Het *init proces*

doet alle initialisatie van de Linux omgeving. Dit proces bestaat dus uit een normaal programma, als je de broncode ervan bekijkt zul je een `main()` ontdekken. Zie bv. *simpleinit.c*, dat je kan vinden in de *pub/linux/utils/util-linux* directory van *ftp.kernel.org* (gebruik als login `anonymous`). Dit is dus een zeer eenvoudig concept: alle echte *high-level* initialisatie van de Linux (of GNU/Linux zoals door sommigen wordt geprefereerd) omgeving wordt gedaan via het *init proces* en alle andere processen die ooit op de Linuxbox zullen lopen zijn kinderen van dit proces.

Dit proces is dus de ouder van alle andere normale processen, zoals bv. een `bash` shell. Daarom dat dit proces "*the grim reaper of innocent orphaned children*" (vertaald klinkt dit als "de Magere Hein der onschuldige weeskindjes") wordt genoemd (zie de commentaar bij *init/main.c*::`init()`). Het zal nl. alle kinderen die geen ouder meer hebben verwijderen. Een *parent process* kan nl. beslissen dat het niet wenst te weten wat er met een kindproces bij het creëren van een *child process* nl. aangeven dat het niet wenst te weten wat er met dat nieuwe proces gebeurt. Dan wordt dit proces als het ware een weeskindje en zal het *init proces* zich ontfermen over dit proces als het gestopt wordt. Een ouderproces kan ook beëindigd worden zonder dat het de statuscode van het kindproces heeft bekeken, dan zal dit kindproces ook door het *init proces* worden verwijderd.

Als dit alles gebeurd is, hebben we dus het *init proces*, een *idle proces* per cpu en daarnaast nog wat *kernel threads* (zoals `ksoftirq`), interrupts zijn ingeschakeld, de kernel is volledig opgestart. Het *init proces* heeft zelf al verschillende processen opgestart en uiteindelijk wordt een shell of xwindows opgestart, waarna de gebruiker dan nieuwe processen kan opstarten.

Hoofdstuk 7

Modules

De module-implementatie van Linux is in de `2.5`-serie herschreven door Paul "Rusty" Russel. Deze implementatie zullen we hier wat van dichterbij bekijken. We beginnen met een korte uitleg over Linuxmodules.

7.1 Het gebruik van modules

Zoals reeds eerder gezegd is de Linuxkernel een zgn. monolithische kernel: eigenlijk gewoon 1 groot programma. Dankzij de Linuxkernel modulecode kan de kernel *on the fly* worden uitgebreid door een *kernel module* te laden, wat een groot nadeel van het monolithische karakter van de kernel wegneemt. De kernel is echter nog altijd monolithisch, aangezien de geladen modules toegang hebben tot de volledige kernelcode -en data.

Dankzij modules kunnen we de kernelgrootte verminderen, zonder te rebooten, door ongebruikte modules te verwijderen. Van de modulefunctionaliteit wordt gretig gebruik gemaakt door de makers van zgn. Linuxdistributies, zoals Red Hat: zij kunnen hun distributies verschepen door de vele ondersteunde drivers voor hardware te compileren als module en de installatie- of opstartsoftware dan te laten bepalen welke hardware aanwezig is en de corresponderende modules in de kernel te laden. Zonder modules zouden de kernels moeten verscheept worden met al deze drivers erin gecompileerd, wat

de kernel onnodig groot zou maken. De *kernel hackers* die als module compileerbare code schrijven of aanpassen kunnen ook handig gebruik maken van de eigenschappen van modules. Tijdens het debuggen kan het gebeuren dat de kernelcode een niet-fatale fout bevat. Met een beetje geluk kan de *kernel hacker* veranderingen aanbrengen aan de code van haar module, de foutieve module uit de kernel laden en de nieuw gecompileerde code in de kernel laden. Dit alles dus zonder het heropstarten van de computer. Zonder modulefunctionaliteit heeft de kernelhacker geen enkel ander alternatief dan eerst de kernel te hercompileren en te rebooten met de nieuwe kernel, wat duidelijk veel meer tijd in beslag neemt. De *kernel hackers* die zich bezighouden met bv. de *low-level* assemblercode zijn echt wel te beklagen, aangezien zij dus wel telkens moeten rebooten om aangepaste code te kunnen testen.

Kernelmodules worden onder de directory */lib/modules/x.y.uv/kernel* geplaatst, waarbij *x.y.uv* bv. *2.5.59* is en dus de kernelversie voorstelt. Je kan dus met verschillende kernelversies booten en er toch voor zorgen dat de juiste modules in de kernel worden geladen (met `modprobe`). De kernelversie die op een bepaald moment op je computer draait kun je zien door bv. `cat /proc/version` te doen. De bestandsnaam van een kernelmodule eindigt op de `.ko` extensie.

De mogelijkheid van modules heeft als gevolg dat *closed source* modules ook in de kernel kunnen worden geladen. Zo was er begin januari 2003 een lange discussie op de lkml over de *closed source* videokaart driver van Nvidia. Het bleek helemaal niet duidelijk te zijn of *closed source* modules wel toegelaten zijn. Zoals eerder gezegd wordt de Linux kernel uitgebracht onder de GPL licentie. Deze licentie stipuleert dat elk afgeleid werk van de Linuxkernel ook onder de GPL moet worden uitgebracht, afgeleide werken mogen dus geen *closed source* zijn. De vraag rees dus of kernelmodules als afgeleide werken van de kernel moeten worden aanzien. Het verdict is dat zolang kernelmodules enkel de functies uit de *kernel API* gebruiken, ze niet aanzien worden als afgeleid werk. Een module moet duidelijk maken onder welke licentie ze is uitgebracht via de macro `MODULE_LICENSE`, bv. `MODULE_LICENSE("GPL")`. Wanneer een gebruiker een probleem met de kernel heeft dat resulteert in

een *kernel oops*, zal de *oops*-boodschap de string *Tainted* bevatten indien de kernel *non-gpl* code bevat. Zo'n gebruikersprobleem zal dan waarschijnlijk worden genegeerd door de *kernel hackers*. Met de *kernel API* worden alle headerfiles en alle vanuit een module beschikbare variabelen en oproepbare functies bedoeld. Een kernelfunctie is beschikbaar voor modules als de functie geëxporteerd wordt, dit gebeurt via de macro `EXPORT_SYMBOL`. Kernelcode kan eisen dat enkel modules die uitgebracht zijn onder de GPL de door de code geëxporteerde symbolen kunnen gebruiken. Dit gebeurt door het symbool te exporteren met de macro `EXPORT_SYMBOL_GPL`.

7.2 Een module laden

De herschreven kernelmodulecode door Rusty noodzaakt nieuwe versies van de modutils gebruikersprogramma's. Deze kunnen bekomen worden via de pagina's van Rusty:
http://www.kernel.org/pub/linux/kernel/people/rusty/modules/.
Het programma insmod probeert een module in de kernel te laden. Dit is een zeer kort programma, maar 110 lijntjes code. We bekijken het even (de code komt uit *module-init-tools-0.6*):

```
int main(int argc, char *argv[ ])
{
<snip>
  fd = open(filename, O_RDONLY);
  if (fd < 0) {
    fprintf(stderr, "Can't open `%s':%s\n",
      filename, strerror(errno));
    exit(1);
  }
  fstat(fd, &st);
  len = st.st_size;
  map = mmap(NULL, len, PROT_READ, MAP_SHARED, fd, 0);
  if (map == MAP_FAILED) {
    fprintf(stderr, "Can't map `%s':  %s\n",
      filename, strerror(errno));
    exit(1);
  }

  ret = syscall(__NR_init_module, map, len, options);
<snip>
}
```

Het programma probeert de zich op de harde schijf bevindende module (dit is dus een gewoon bestand) te openen en mapt dit bestand op het geheugen (met de bibliotheekfunctie `mmap`). Vervolgens wordt een *system call* (`syscall`) opgeroepen met als *system call* nummer dat voor `init_module()`. We komen nu dus op het terrein van de kernel. De waarde voor de constante `__NR_init_module` wordt gedefinieerd in *include/asm-i386/unistd.h*:

```
<snip>
#define __NR_init_module 128
<snip>
```

Het bestand *arch/i386/kernel/entry.S* (zie Hoofdstuk 5) bevat op positie 128 in de array `sys_call_table`:

```
ENTRY(sys_call_table)
<snip>
        .long sys_init_module
<snip>
```

De functie `sys_init_module()` bevindt zich in *kernel/module.c*:

```
/* This is where the real work happens */
asmlinkage long
sys_init_module(void *umod, unsigned long len, const char *uargs)
```

We verwijzen naar sectie 5.2 om te weten hoe de *system call* van het programma insmod eindigt in het uitvoeren van bovenstaande kernelfunctie.

7.3 Werking van de module code

De implementatie van de generische modulecode is niet echt bijster interessant. We zullen ons hier beperken tot de principes.

Via een mutex (zie Hoofdstuk 4) wordt ervoor gezord dat het laden en/of verwijderen van modules gesynchroniseerd wordt: op elk moment kan maar 1 module worden verwijderd/geladen. Bemerk dat dit een versimpeling is van de realiteit, de details van de code maken natuurlijk dat enkel de kritieke secties beschermd zijn, om de performantie te verbeteren. De generische code houdt een lijst bij van alle geladen modules. Per module houdt ze een array bij van alle door de module geëxporteerde symbolen. Deze symbolen worden gespecifieerd in het bestand dat de module bevat. Ook de door de kernel geëxporteerde symbolen die door de module gebruikt worden, staan in dit bestand. Een module zal enkel geladen worden indien al deze symbolen aanwezig zijn in de kernel. Zo'n symbool kan een functienaam zijn (zoals `printk()`) of een variabele, zoals bv. de functiepointer `br_should_route_hook` (*net/bridge/br.c*). Zoals eerder aangehaald, dient de macro `EXPORT_SYMBOL` gebruikt te worden om symbolen beschikbaar te maken voor modules.

De grootste moeilijkheid bij modules wordt veroorzaakt doordat modules verwijderbaar zijn. Wanneer een module verwijderd wordt, moet ervoor worden gezorgd dat de code van die module niet kan worden uitgevoerd. Met het verwijderen van een module wordt bedoeld dat de code, dus de instructies, samen met de data van die module worden verwijderd uit het kernelgeheugen. Indien er nog ergens een pointer in de andere kerneldata aanwezig is die verwijst naar dit deel van het geheugen, nadat de module is verwijderd, hebben we te maken met een *dangling pointer* en hebben we dus een *kernel bug* gevonden.

Wanneer een eerste module een geëxporteerd symbool van een tweede module gebruikt, wordt door generieke modulecode gezorgd dat de tweede module niet kan verwijderd worden zolang de eerste module niet verwijderd is. Wanneer de gebruiker het programma lsmod uitvoert, zal er weergegeven worden dat de tweede module wordt gebruikt door de eerste module. Situaties komen echter voor waarin de generieke modulecode niet zelf kan uitvissen dat het verwijderen van een module gevaarlijk is. Daarom zijn de functies `try_module_get()` en `module_put()` voorzien. Code die ervoor zorgt dat instructies uit een andere module worden uitgevoerd, moet erover waken dat deze module niet kan verwijderd worden terwijl die instructies worden uitgevoerd. Indien nodig moet de code dus eerst via `try_module_get()` de *usage count* van de andere module – eventueel tijdelijk – verhogen. Via lsmod zullen we dan zien dat het cijfer onder "Used" van de module zal verhoogd zijn.

Kernelmodules zijn gecompileerd in het ELF-formaat (zie sectie 8.2),wat het standaardformaat is van door recente versies van gcc gecompileerde C-bestanden. Het maakt het dynamisch laden van *executables* mogelijk en zorgt ook voor de mogelijkheid om met extern gedefinieerde symbolen te werken. Met het commando nm kunnen we bv. uitvissen naar welke symbolen een bepaalde module referenties heeft. Andere leuke commando's zijn `ldd` (*list dynamic dependencies*), objdump (eigenschappen van gecompileerde objectcode weergeven) en file (probeert het type van het bestand te achterhalen).

Hoofdstuk 8

Processen

Ruwweg is een proces computerjargon voor een entiteit in het computergeheugen die een bepaalde taak verricht. De entiteit die ervoor zorgt dat je een login-scherm krijgt wanneer je inlogt op eduserv is een voorbeeld van zo'n proces. De enige taak van een besturingssysteem is eigenlijk het de processen gemakkelijk maken en processen t.o.v. elkaar beschermen. Een besturingssysteem zorgt ervoor dat verschillende processen "gelijktijdig" de computer kunnen benutten, terwijl de processen zich (meestal) niets hoeven aan te trekken van de andere actieve processen. Het gebruik van een besturingssysteem heeft als nadeel dat de programma's trager werken doordat het besturingssysteem regelmatig dingen doet. Een besturingssysteem biedt echter een enorme flexibiliteit aan de gebruiker, zodat deze performantiekost niet opweegt tegen alle voordelen.

Hetgeen waarmee de gebruiker rechtstreeks in contact komt zijn de processen. Ook de grote meerderheid van de programmeurs hoeven enkel notie te hebben van processen. Er is een subtiel verschil tussen het concept programma en het concept proces. Applicatieprogrammeurs schrijven programma's, een programma kunnen we aanzien als hetgeen opgeslagen is op de schijf. Op elk moment kunnen meerdere instanties van dit programma actief zijn als proces in het besturingssysteem. Er is dus maar 1 programma, terwijl er meerdere processen kunnen zijn die dit programma uitvoeren.

8.1 De zandbak

Vooraleer we de structuur van programma's van dichterbij bekijken, geven we even het grote verschil tussen een C-programma en Java-applicatie. Een Java-applicatie draait in een zogenaamde zandbak, gecreëerd door de Java Virtual Machine. Een proces dat een C-programma uitvoert kunnen we beschouwen als iets dat uitgevoerd wordt in een zandbak gecreëerd door het besturingssysteem. Bij Java applicaties is het de *Java Virtual Machine* (JVM) die uitgevoerd wordt in de door het besturingssysteem gecreëerde zandbak. De *Java Virtual Machine* wordt dan ook typisch geschreven in C. Wanneer het uitvoeren van bepaalde Java-code een zogenaamde *segmentation fault* geeft, heb je eigenlijk een bug in de JVM gevonden. Een segmentatiefout wordt nl. ontdekt door het besturingssysteem en het is dus het programma dat in de zandbak van het besturingssysteem draait dat de fout maakt. Voor de rest van het verhaal houden we ons enkel met processen bezig.

8.2 Uitzicht van een programma

Gecompileerde C-programma's kunnen in verschillende formaten opgeslaan worden op de harde schijf. Een belangrijk formaat is het ELF-formaat (*Executable and Linking Format*), dat het standaardformaat is van `gcc`. De standaardwebsite waar de specificatie van dit formaat wordt onderhouden is ons onbekend, maar je vindt de specificatie makkelijk via *google*, zie ook [29]. Een ELF-bestand begint altijd met de magische string `\177ELF`, het "*magic number*" van het bestand genoemd. Een Linuxsysteem bevat normaal gezien het bestand */usr/share/misc/magic*, dat door het commando `ls` wordt gebruikt om bv. sommige bestanden eventueel in een ander kleurtje weer te geven. De lijn voor ELF uit dat bestand ziet er zo uit:

```
0       string          \177ELF         ELF
```

Zie man magic voor meer informatie. Het kernelbestand *include/linux/elf.h* bevat deze entry:

```
#define ELFMAG0 0x7f /* EI_MAG */
#define ELFMAG1 'E'
#define ELFMAG2 'L'
#define ELFMAG3 'F'
#define ELFMAG "\177ELF"
#define SELFMAG 4
```

Merk op dat `\177` de octale notatie is voor het decimale getal 127 (hexadecimaal wordt dat `0x7f`). De kernelfunctie die verantwoordelijk is voor het uitvoerbaar maken van een ELF-bestand zal eerst kijken of het bestand dit *magic number* bevat, dit gebeurt in de functie `load_elf_binary()` (*fs/binfmt_elf.c*):

```
if (memcmp(elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
        goto out;
```

Bemerk het gebruik van het prachtige `goto`-statement :)

Vooraleer we de geheugenstructuur van een proces kunnen naderbij beschouwen, moeten de concepten virtueel geheugen en paginering ingevoerd worden. Een veel uitgebreidere behandeling van deze concepten is bv. te vinden in [19]. Een handig hulpmiddel kan ook een naslagwerk van de architectuur zelf zijn, zoals bv. [21] (je neemt misschien beter een recenter boek).

8.3 Usermode-kernelmode overgang

Dit is een overgang van de uitvoering van code van een gebruikersprogramma naar de uitvoering van kernelcode of omgekeerd. Zoals we zometeen zullen zien, gaat zo'n overgang gepaard met een verandering van gebruikte segmenten.

8.4 Virtueel geheugen, paginering en segmentatie

Het onderwerp van deze sectie en subsecties is zonder twijfel één van de ingewikkeldste die men tegenkomt in de beschrijving van een besturingssysteem. De tekst in deze sectie is een samenraapsel van informatie afkomstig uit [19, 21, 23, 24] en andere bronnen. Te weten komen hoe de geheugenstructuur van een gebruikersproces in Linux er precies uitziet, is geen sinecure. Het document "*Linux kernel Hacker's Guide*" [22] was redelijk verhelderend.

Virtueel geheugen heeft betrekking op systemen waarin de totale omvang van programma- en datagebieden het fysieke geheugen te boven kan gaan. Het besturingssysteem houdt alleen de in gebruik zijnde delen van het programma en de data in het fysieke geheugen. De rest wordt op de schijf bewaard totdat er behoefte aan is. Voordat de computer een geheugenlocatie benadert, bepaalt een eenheid voor geheugenbeheer (de MMU, *memory management unit*) of die locatie zich ook werkelijk in het geheugen bevindt. Indien dit niet zo is, moet de data eerst van de schijf worden gehaald. De MMU moet de logische adressen waarnaar de programma's verwijzen, omzetten in de fysieke adressen waar de data werkelijk staat. De x86 heeft zijn eigen MMU aan boord, die bestaat uit de segmenteringseenheid en de pagineringseenheid.

De adressen waarmee programmeurs te maken krijgen (zoals bij het gebruik van pointers), noemen we de virtuele adressen. Daarnaast hebben we de fysieke adresruimte, die de werkelijke adressen in het geheugen voorstellen. Via de hardware worden virtuele adressen vertaald naar fysieke adressen. De virtuele adresruimte kan veel groter zijn dan de fysieke adresruimte doordat gebruik wordt gemaakt van de harde schijf. Een door een programma geadresseerd (virtueel) adres hoeft nl. niet noodzakelijk in het fysieke geheugen aanwezig te zijn. Indien het niet aanwezig is, zal het van de schijf in het geheugen worden geladen, waarbij eventueel eerst de data uit dit geheugendeel naar de schijf moet worden geschreven. Dit schrijven naar en van de

schijf gebeurt per pagina. Een pagina is een stuk geheugen die een bepaalde grootte heeft, in de x86 ligt die grootte vast op 4 Kilobyte. Het fysieke geheugen met adressen 0 t.e.m. 4095 komt bv. overeen met de eerste pagina van het geheugen. Dankzij virtueel geheugen kan de programmeur haar programma's schrijven zonder rekening te moeten houden met de fysieke grootte van het geheugen van het systeem. Twee instanties van hetzelfde programma (twee processen) kunnen hierdoor ook met dezelfde virtuele geheugenadressen werken: het virtuele geheugen van beide processen wordt op een andere manier gemapt op het fysiek beschikbare geheugen. Zonder virtueel geheugen zouden alle geheugenadressen moeten worden aangepast wanneer een programma op een andere plaats in het geheugen wordt gezet.

Naast de combinatie virtueel geheugen/paginering biedt de x86 ook nog segmentatie. Zonder segmentatie heeft een programma slechts één eendimensionale geheugenruimte. Het kan soms handig zijn om verschillende aparte eendimensionale geheugenruimtes (allen virtueel geheugen) te hebben. De ene ruimte bevat dan bv. de programma-instructies, terwijl de andere ruimte de data van het programma bevat. Dit wordt mogelijk gemaakt dankzij segmentatie. Het ELF-formaat dat hierboven werd besproken, laat bv. toe om expliciet te specifiëren welke segmenten een programma heeft en wat er zich in die segmenten bevindt. Terwijl paginering volledig transparant is voor de programmeur, moet de programmeur bij het gebruik van segmentatie expliciet aanduiden welk segment wordt bedoeld, als er een geheugenadres wordt opgegeven. Dit segmentnummer wordt opgeslaan in één van de segmentregisters en wordt gebruikt als index in de LDT. De drie belangrijkste segmentregisters zijn `ss` (het stack segment register), `cs` (het codesegment register) en `ds` (het datasegment register). Linux maakt echter weinig gebruik van segmentering. Het is nl. eenvoudiger indien de segmenten van alle processen er ongeveer gelijk uitzien. Sommige door Linux ondersteunde processoren van Sun hebben trouwens geen segmentatiemogelijkheden.

We kunnen drie soorten adressen onderscheiden. De logische adressen bestaan uit een *segment selector* (die zich in zo'n segmentregister bevindt)

samen met een offset. Via de LDT (of GDT) wordt dit logisch adres omgezet naar een lineair adres. Aangezien de x86 een 32-bitmachine is, heeft zo'n lineair adres een bereik van 2^{32} bytes, wat dus 4 Gigabyte is. Via de paginering wordt dit lineair adres dan vertaald naar een fysiek adres, waarmee de applicatieprogrammeur nooit te maken krijgt.

8.4.1 De GDT en LDT

De GDT (*Global Descriptor Table*) en LDT (*Local Descriptor Table*) bevatten de data voor de vertaling van een adres relatief t.o.v. een segment naar een lineair (fysiek of virtueel) adres. In Linux zal de vertaling altijd naar een virtueel (lineair) adres zijn. Via de segmentregisters wordt duidelijk gemaakt aan de processor relatief t.o.v. welk segment we bezig zijn. Deze registers bevatten een getal dat de index voorstelt van een entry in de GDT of LDT. Of het getal wijst in de GDT of de LDT is afhankelijk van de tweede minst significante [1] bit van dit getal. De layout van de GDT in Linux kan gevonden worden in *include/asm-i386/segment.h*.

8.4.2 Linux en paginering/segmentatie

Zoals gezegd, maakt Linux weinig gebruik van segmentatie. Segmentatie wordt vooral gebruikt om onderscheid tussen kernelgeheugen en gebruikersgeheugen te maken. Standaard zal het zo zijn dat het datasegment, codesegment en stacksegment van een proces allen beginadres 0 hebben en een grootte van 3 GigaByte (zie straks), ze overlappen dus.

Linux maakt wel veelvuldig gebruik van paginering. Paginering op de x86 werkt alsvolgt: elk proces heeft een paginadirectory, met grootte 1 pagina (dus 4 KiloByte). De paginadirectory is een array van 32-bits paginaspecificatoren die verwijzen naar paginatabellen en bevat dus maximaal 1024 zo'n verwijzingen. Een paginatabel heeft ook een grootte van 1 pagina en kan

[1] hoe groter de macht van 2 is die de bit (als deel van een byte) voorstelt in de computer, hoe significanter de bit

maximaal 1024 pagina's adresseren. Eén paginadirectory kan dus maximaal $1024^2 = 2^{20}$ pagina's adresseren. Aangezien de grootte van een pagina gelijk is aan 2^{12}, kan één pagina dus de volledige lineaire adresruimte van de x86 adresseren ($2^{20} * 2^{12} = 2^{32}$), want de x86 is een 32-bitmatchine. Een element van een paginatabel bevat, naast een fysieke paginaselector van 20 bits nog 12 bits die de eigenschappen aangeven. Aangezien de paginagrootte 2^{12} is, zijn er maar 20 bits nodig om naar alle mogelijke fysieke pagina's te kunnen verwijzen. Die 12 bits bevatten o.a. een *present-bit*, die aangeeft of de pagina in het geheugen aanwezig is of niet; een *dirty-bit*, die aangeeft of er geschreven is naar die pagina; een *accessed-bit*, die aangeeft of van de pagina gelezen is of ernaar geschreven is. Wanneer een pagina gebruikt wordt waarvan de *present-bit* uit staat, wordt een "pagina-niet-aanwezig"-exceptie gegenereerd, Linux kan dan deze pagina in het geheugen laden. De *accessed-bit* en *dirty-bit* kunnen door het besturingssysteem gebruikt worden om te kiezen welke pagina er moet vervangen worden, wanneer het fysieke geheugen schaars wordt.

Elk proces in Linux heeft zijn eigen paginadirectory. Elk proces heeft beschikking over de adressen `0-0xBFFFFFFF`, de segmenten van het proces zullen als basisadres 0 hebben en als maximale lengte `0xC0000000` (3 Gigabyte). De reden waarom processen geen hogere adressen kunnen gebruiken, komt door een mechanisme door Linux gebruikt om overgangen van *user context* naar *kernel context* (bv. via een *system call*) sneller te maken. Wanneer overgegaan wordt naar *kernel context*, wordt via de IDT (zie sectie 5.1.2) het kernelsegment gebruikt i.p.v. het gebruikerssegment. Er wordt echter dezelfde paginadirectory gebruikt. Alle lineaire adressen vanaf `0xC0000000` verwijzen naar kernelgeheugen. Een gebruikersprogramma kan niet aan dit geheugen omdat zijn segmenten een maximale grootte hebben van `0xC0000000` bytes, de hardware zal ervoor zorgen dat indien het gebruikersprogramma adressen met grotere waarde adresseert, er een segmentatiefout wordt gegenereerd. De paginatabellen van het gebruikersprogramma zullen voor deze hoge adressen wel naar het kernelgeheugen wijzen, maar deze

kunnen dus niet door het gebruikersprogramma aangeraakt worden. Dankzij dit mechanisme kan de kernelcode, die altijd in de context van een proces wordt uitgevoerd, de paginatabellen van het lopend proces gebruiken. Moest dit niet het geval zijn, zou bij elke overgang de pagina-directory moeten herladen worden.

De kernelsegmenten beginnen op offset `0xC0000000`, zodat een logisch kerneladres, bv. `0x10`, correspondeert met het lineaire adres `0xC0000010`. In het kernelsegment wordt het volledige fysieke geheugen gemapt op de overeenkomstige logische adressen, zodat logisch kerneladres `0x10` na de vertaling naar het lineaire adres zal worden gemapt op het fysische adres `0x10`. De logische adressen die groter zijn dan het beschikbare fysieke geheugen worden bv. gebruikt om kernelgeheugen te alloceren dat in de virtuele adresruimte aaneensluitend is, maar niet noodzakelijk in de fysieke adresruimte. Deze techniek zorgt er trouwens voor dat het totaal aan nuttig fysiek geheugen op een standaard Linuxkernel beperkt is tot een 950 MB, zoals volgend codefragment uit *include/asm-i386/page.h* ons aangeeft:

```
/*
 * This handles the memory map.. We could make this a config
 * option, but too many people screw it up, and too few need
 * it.
 *
 * A __PAGE_OFFSET of 0xC0000000 means that the kernel has
 * a virtual address space of one gigabyte, which limits the
 * amount of physical memory you can use to about 950MB.
 *
 * If you want more physical memory than this then see the CONFIG_HIGHMEM4G
 * and CONFIG_HIGHMEM64G options in the kernel configuration.
 */

#define __PAGE_OFFSET (0xC0000000)

/*
 * This much address space is reserved for vmalloc() and iomap()
 * as well as fixmap mappings.
 */
#define __VMALLOC_RESERVE (128 << 20)
```

Als je machientje dus meer dan 1 GB geheugen heeft, zet je best "*High Memory Support*" aan bij het compileren. Dit is dus omdat in *kernel context*

enkel de lineaire adressen `0xC0000000-0xFFFFFFFF` kunnen worden gebruikt.

8.4.3 Uitzicht van het geheugen van een proces

We vatten nog eens enkele dingen samen en vullen hier en daar aan. Elk gebruikersproces heeft een LDT die een codesegment en data-stack segment bevat. Deze segmenten gaan van 0 tot 3 GB (dus een grootte van `0xC0000000`). Hierdoor zijn voor een gebruikersproces de lineaire en logische adressen dezelfde (doordat de offset 0 is). De adressen vanaf 3 GB tot 4 GB bevatten, zoals gezegd, het kernelgeheugen. Deze paginatabellen zijn dezelfde voor elk proces.

De gebruikersstack zit bovenaan het datasegment van de gebruiker en groeit naar beneden. De kernelstack komt aan bod in sectie 8.5. Gebruikerspagina's, dit zijn de pagina's die zich onder de 3 GB in de gebruikerspaginatabel bevinden, kunnen worden geswapped.

De LDT voor het *idle proces* (zie Hoofdstuk 6) ziet er zo uit (komt uit [22]):

```
* LDT[1] = user code, base=0xc0000000, size = 640K
* LDT[2] = user data, base=0xc0000000, size = 640K
```

Aangezien het *idle proces* een kernelproces is, was het te verwachten dat de LDT voor dit proces op de GDT van de kernel zou lijken (basisadres voor de code- en datasegmenten gelijk aan `0xC0000000`). Andere kernelthreads zullen ook een LDT structuur hebben gelijkaardig met bovenstaande. Gebruikersprocessen zullen standaard `base = 0x0, limit = TASK_SIZE = 0xc0000000` hebben. Er dient wel opgemerkt te worden dat gebruikersprogramma's hun LDT kunnen veranderen via *system calls* (mits goedkeuring van de kernel).

Een proces erft de paginatabellen van zijn ouder, tijdens een oproep van de functie `fork()`. De paginatabelentries worden als *read-only* gemarkeerd. Wanneer een proces dan naar die geheugenruimte probeert te schrijven, zal de pagina eerst worden gekopieerd en worden beide pagina's als read-write gemarkeerd. Dit heet *copy-on-write*. Een groot voordeel van deze werkwijze

is de performantiewinst wanneer een proces een nieuw proces wenst aan te maken (dat dan een kindproces wordt). Onder Linux gebeurt dit door eerst een `fork()` te doen en daarna een `execve()` in het kindproces. Indien bij de `fork()` alles werd gekopieerd naar apart geheugen voor het kindproces, zou dit totaal nutteloos zijn geweest aangezien dat kindproces een nieuw programma zal uitvoeren waardoor de gekopieerde data niet gebruikt wordt door het kindproces.

8.4.4 Memory Regions

De adresruimte van een proces bestaat uit verschillende *memory regions*, elk zulke "regio" bestaat uit een start- en eindadres, de toeganspermissies en het object dat ermee geassocieerd is (bv. een uitvoerbaar bestand dat de code van het uitgevoerde proces bevat). Met `cat /proc/id/maps` kunnen we de *memory regions* van het proces met id `id` bekijken (id's van draaiende processen bekomen we via `ps`). Zo'n *memory region* kan bv. de `libc` C-bibliotheek zijn, deze wordt door alle processen die deze bibliotheek gebruiken gedeeld. Dit is mogelijk dankzij het eerder besproken ELF-formaat. Wanneer twee processen hetzelfde programma uitvoeren, zal de programmacode ook in gedeeld geheugen staan. Een *memory region* is dus een deel van de logische adresruimte van het proces, die bepaalde eigenschappen heeft; dit is geen segment.

We grijpen hier even de kans om duidelijk te stellen dat een *kernel hacker* niet beschikt over de C-bibliotheek, die dus wel in een *memory region* beschikbaar is voor gebruikersprocessen die (gecompileerde) C-code uitvoeren. De functie `printf()` bestaat bv. gewoon niet in de kernel. Het zou natuurlijk zonde zijn dat handige functies zoals `strcmp()` niet beschikbaar zijn voor *kernel hackers*, deze functies worden echter expliciet gedefinieerd door kernelcode zelf, zie bv. *include/asm-i386/string.h*.

8.5 Bijhouden van de proceseigenschappen

8.5.1 Locatie van de proceseigenschappen

Een proces heeft allerhande eigenschappen, die door de kernel bijgehouden worden. Deze informatie bevindt zich in een `struct thread_info` en een `struct task_struct` (zie de bestanden *include/asm-i386/thread_info.h* en *include/linux/sched.h*). De `struct thread_info` is een kleine `struct` die in 1 *cache line* kan, we tonen hier een paar *members*:

```
struct thread_info {
        struct task_struct *task; /* main task structure */
        __u32 cpu; /* current CPU */
        __s32 preempt_count; /* 0 => preemptable, <0 => BUG */
}
```

De member `cpu` bevat het nummer van de processor waarop het proces momenteel wordt uitgevoerd. Aangezien kernelcode regelmatig dit `cpu`-nummer bekijkt, is het niet onlogisch dat dit nummer zich in de `struct thread_info` bevindt (die veel kans heeft om gecached te zijn). De `struct task_struct` bevat het overgrote deel van de eigenschappen van een proces. We bekijken eerst eens van naderbij waar de `struct thread_info`'s zich bevinden in het kernelgeheugen.

Het bestand *kernel/fork.c* bevat de kernelcode voor het creëren van nieuwe processen. Geheugen voor de `struct thread_info` wordt hierbij gealloceerd (`dup_task_struct()`) via de macro `alloc_thread_info` (*include/asm-i386/thread_info.h*):

```
#define alloc_thread_info() ((struct thread_info *) \
__get_free_pages(GFP_KERNEL,1))
#define THREAD_SIZE (2*PAGE_SIZE)
```

Deze macro zorgt dat er 2 geheugenpagina's worden gereserveerd. De grootte van een pagina is gedefinieerd in *include/asm-i386/page.h*:

```
#define PAGE_SHIFT 12
#define PAGE_SIZE (1UL << PAGE_SHIFT)
#define PAGE_MASK (~(PAGE_SIZE-1))
```

Als we er een boek over de x86-processor op naslaan, lezen we dat de paginagrootte vastligt op 4 KB, waarbij 1 KB 2^{10} bytes groot is. Dit feit zien we dus ook terug in de bovenstaande definitie van `PAGE_SIZE`, want de definitie van die macro stelt `PAGE_SIZE` = 2^{12} = `4 KB`. Twee pagina's reserveren enkel voor de `struct thread_info` is natuurlijk nonsens. Op het begin van dit geheugen zal de `struct thread_info` worden geplaatst, terwijl de rest van het gealloceerde geheugen zal dienst doen als de kernelstack voor dat proces. Deze stack groeit naar beneden. Indien het gebruikte geheugen op de kernelstack dus groter wordt dan ongeveer 4 KB, zullen we met gecorrumpeerd geheugen zitten, het eerste wat overschreven zal worden is de `struct thread_info` behorende bij het proces. Het gebruik van recursieve aanroepen van functies is dan ook uit den boze in de kernelcode. Wanneer een proces een *system call* doet, zal kernelcode uitgevoerd worden om die call af te handelen. Om veiligheidsredenen gebruikt die kernelcode niet de stack die het gebruikersprogramma gebruikt, maar wel een aparte kernelstack die voor het proces in kernelgeheugen is gereserveerd. Het proces kan ook *herscheduled* worden terwijl deze kernelcode wordt uitgevoerd, dit kan bv. gebeuren wanneer de kernelcode `copy_from_user()` uitvoert. Deze functie kopieert data vanuit userspacegeheugen (gebruikersgeheugen) naar kernelgeheugen en deze functie kan het proces doen slapen. Daarom moet er per proces een kernelstack zijn, zodat wanneer het proces terug wordt uitgevoerd, deze kernelstack er identiek uitziet zoals hij was net voordat het proces begon te slapen.

De *inline* functie `current_thread_info()` (*include/asm-i386/thread_info.h*) wordt gebruikt om een pointer te verkrijgen naar de `struct thread_info` van het huidige proces:

```
/* how to get the thread information struct from C */
static inline struct thread_info *current_thread_info(void)
{
          struct thread_info *ti;
          __asm__("andl %%esp,%0; " : "=r"(ti) : "0"(~8191UL));
          return ti;
}
```

Wanneer iemand deze functie voor het eerst bekijkt, is het alsof hier magie gebeurt. Maar, met hetgeen we hierboven al gezegd hebben kunnen we snappen wat hier gebeurt. Het register `esp` wordt altijd gebruikt om de stackpointer te bewaren. De *logische and* wordt genomen van de stackpointer (die dus naar de kernelstack wijst) samen met de constante `~8191UL`. Een kleine rekensom: `8192 = 0x00002000 =` 2^{13}`=2*PAGE_SIZE=THREAD_SIZE`. Een klein voorbeeldje in hexadecimale notatie, stel dat `esp=0xcba0fdd0`:

```
  0xcba0fdd0 & ~(0x00002000 - 1)
= 0xcba0fdd0 & ~(0x0001fff)
= 0xcba0fdd0 & 0xffffe000
= 0xcba0e000
```

We vinden dus dat het beginadres van de bijhorende `struct thread_info` gelijk is aan `0xcba0e000`.

De data van het actieve proces is toegankelijk voor kernelcode via `current`, wat kan gebruikt worden alsof het een pointer naar de `struct task_struct` van het huidige proces is. In feite is dit echter een *preprocessor macro*. We bekijken `current` even van dichterbij, de relevante bestanden zijn *include/asm-i386/current.h* en *include/asm-i386/thread_info.h*.

De macro `current` is gedefinieerd als de inline functie `get_current()`, die een pointer naar een `struct task_struct` teruggeeft:

```
#define current get_current()
static inline struct task_struct * get_current(void)
{
  return current_thread_info()→task;
}
```

Indien het proces een bestand wil openen, heeft de kernelcode via `current` toegang tot de `task_struct` van het proces en kan zo de privileges van het proces bepalen en beslissen of het proces het bestand wel mag openen.

8.5.2 De `struct task_struct`

Zoals gezegd bevat de `struct task_struct` (*include/linux/sched.h*) het overgrote deel van de informatie over een proces. Die `struct` heeft meer dan 60 members, waarvan sommige pointers zijn naar nog andere `struct`'s. Een opsomming of bespreking van al deze members zou nogal langdradig zijn en de meerderheid ervan is ons toch onbekend. We beperken ons tot een kleine opsomming van wat in deze `struct` allemaal wordt bewaard: het pid (*process id*) van het proces waarbij deze `struct` hoort, een pointer naar de ouder van het proces, een lijst van de kinderen van het proces, informatie over hoe het virtueel geheugen van het proces eruit ziet (paginadirectory, *memory regions*, ...), de bestanden die geopend zijn door het proces, informatie over de IPC (*inter-process communicatie*), de signalen die het proces ontvangen heeft...

8.5.3 Allocatie van de `struct task_struct`'s

Om de performantie te verhogen wordt zo'n `struct`, wanneer het bijhorende proces niet meer bestaat, gerecycleerd om te gebruiken als `struct task_struct` voor een nieuw proces. Dit is beter dan telkens het geheugen voor deze `struct` vrijgeven wanneer het proces verwijderd wordt. Geheugen kan trouwens maar per pagina worden gealloceerd. Tijdens de initialisatie van de kernel wordt een *software cache* (*slab*) gecreëerd voor het alloceren van deze `struct task_struct`'s. Wanneer zo'n `struct` wordt hergebruikt, wordt ook het geheugen voor de kernelstack en de `struct thread_info` hergebruikt. De initialisatie van deze cache gebeurt in `fork_init()` (*kernel/fork.c*):

```
/* Each cache has a short per-cpu head array, most allocs
 * and frees go into that array, and if that array overflows, then 1/2
 * of the entries in the array are given back into the global cache.
 * The head array is strictly LIFO and should improve the cache hit rates.
 * On SMP, it additionally reduces the spinlock operations. */
void __init fork_init(unsigned long mempages)
  /* create a slab on which task_structs can be allocated */
  task_struct_cachep = kmem_cache_create("task_struct",
    sizeof(struct task_struct),0, SLAB_HWCACHE_ALIGN, NULL, NULL);
```

Zo'n cache bestaat uit een

```
struct array_cache {
  unsigned int avail;
  unsigned int limit;
  unsigned int batchcount;
  unsigned int touched;
};
```

gevolgd door een aantal *void pointers*, die dus naar gealloceerd geheugen wijzen. De member `avail` duidt de positie van de eerste pointer aan die niet in gebruik is (LIFO).

8.6 Scheduling

Een *scheduler* verdeelt de processortijd onder de verschillende processen. Het is duidelijk dat op eender welk moment slechts één proces actief kan zijn op een bepaalde cpu. Het is de taak van de *scheduler* om alle processen voldoende aan bod te laten, zodat het voor de gebruiker lijkt alsof alles tegelijkertijd gebeurt. De functie die dit implementeert heet niet toevallig `schedule()` en bevindt zich in *kernel/sched.c*. Deze functie wordt op vele plaatsen in de kernel opgeroepen. Dit gebeurt enerzijds op specifieke plaatsen in de generieke kernelcode, bv. na een *system call*, anderzijds gebeurt dit ook in kernelcode die in *user context* wordt uitgevoerd. Zoals we in Hoofdstuk 4 hebben gezien, kan een proces slapen indien het een semafoor probeert te bezetten die al bezet is. De code om een semafoor neer te brengen

(zie *arch/i386/kernel/semaphore.c*::`__down()`) roept dan ook `schedule()` op indien nodig.

De schedulinginformatie per proces wordt bijgehouden in de bijhorende `struct task_struct` van het proces. Wanneer de *scheduler* een ander proces kiest, gebeurt er een zgn. *context switch*. Deze houdt in dat het *memory management* systeem van de processor de waarden voor het nieuwe proces krijgt. Zo zullen de paginatabellen van het nieuwe proces gebruikt worden. Daarnaast worden ook de registerwaarden bewaard van het oude proces en de bewaarde registerwaarden van het nieuwe proces in de registers van de processor geladen. Ook het `IP`-register (*instruction pointer*) krijgt terug de oude waarde, waardoor de uitvoering van het huidige proces gewoon doorgaat waar het eerder werd gescheduled.

Scheduling is een nogal ingewikkeld deel van de Linuxkernel. Om meer te weten te komen over *scheduling* in Linux `2.5` kan je beginnen met het lezen van *Documentation/sched-design.txt*.

Hoofdstuk 9

Nawoord

De hoeveelheid broncode in een distributie van de Linuxkernel is groot, zeer groot. Het zou een helse taak zijn om al deze code te moeten bekijken. Gelukkig is dit helemaal niet nodig, ook niet als je zelf wat in de kernelcode wenst te *hacken*. Het is zelfs niet nodig om een goede kennis te hebben van de sleutelonderdelen van de Linuxkernel, zoals *scheduling*. Slechts weinigen weten alles af van deze sleutelonderdelen. Wanneer je zelf aan de kernel wilt sleutelen, of er meer over wenst bij te leren, moet je er een bepaald subsysteem uit nemen dat je sterk interesseert. Dit kan een onderdeel zijn van het bestandssysteem (bv. het Linux Ext2 bestandssysteem), een onderdeel van het netwerksubsysteem (zoals IP routing, bridging, firewalling), een bepaalde *device driver*, een onderdeel van het geluidssubsysteem, misschien ben je wel moedig genoeg om de *scheduling* te doorgronden ...

De Linuxkernel is zeer modulair opgebouwd. Veelal is het genoeg om te weten welke functie je precies moet uitvoeren, zonder enige verdere details ervan te weten. In de kernel gebeurt alles via registratie. Er is generieke code aanwezig voor de subsystemen van de kernel, die de erbij geregistreerde functies/modules voorziet van data. Wanneer we bv. IP pakketten wensen te filteren via een zelfgeschreven module, is het voldoende om de functie die filtert te registreren bij het netfilter systeem. Dit systeem zorgt er dan voor dat de gewenste IP pakketten aan de functie worden doorgegeven voor

controle. De moduleschrijver hoeft zich dus niets aan te trekken van hoe dat IP pakket ooit ontvangen werd door de *device driver* van de netwerkkaart, hoe dit dan via de generieke netwerkcode werd doorgegeven aan de IP-code van de kernel, hoe dit pakket eventueel werd gedefragmenteerd uit verschillende UDP-fragmenten, hoe het pakket ooit de bestemming zal bereiken, enz.

De leercurve is dus wel groter dan ze is voor normale gebruikersprogramma's, maar het leerproces kan toch geleidelijk aan gebeuren, terwijl er toch al met kernelcode kan ge{prutst,speeld} worden. Als beloning krijg je een gevoel van vrijheid en ook wel macht: het besturingssysteem ligt in je handen.

Wanneer je broncode doorworstelt, is gebruik van de juiste hulpmiddelen noodzakelijk, grep en find zijn je vrienden. De *Linux Cross-Reference* website [12] is ook zeer handig om uit te vissen waar wat precies gedefinieerd is en op welke plaatsen het wordt gebruikt. De enige manier om alles echt te doorgronden is door zelf je handen vuil te maken en kernelcode te ontwikkelen. Dan zul je innige contacten mogen beleven met de felgevreesde *kernel panic*. Niets zo leuk dan C-code debuggen in de kernel :)

Voor *kernel newbies* bestaat de site [30]. Als je jezelf nuttig wenst te maken voor de Linuxkernelgemeenschap, zonder dat je beschikt over jaren ervaring, kan je bij [31] terecht voor suggesties.

Voor diegenen die het zien zitten om de kernelcode te gaan verkennen: hoera, sterkte en veel leesplezier!

Bibliografie

[1] http://www.zeus.rug.ac.be

[2] http://www.nongnu.org/glms/

[3] http://ragib.hypermart.net/linux/

[4] http://www.cs.vu.nl/˜ast/

[5] http://www.isc.tamu.edu/˜lewing/linux/

[6] http://www.gnu.org

[7] http://ftp.gnu.org/gnu/glibc/

[8] http://www.kernel.org

[9] http://groups.google.com/groups?hl=en&lr=&ie=UTF-8&group=linux.kernel

[10] http://lwn.net/Articles/8106/

[11] http://www.oreilly.com/catalog/opensources/book/appa.html

[12] http://lxr.linux.no/

[13] http://www.kernel.org/pub/linux/docs/lkml/

[14] http://user-mode-linux.sourceforge.net/

[15] http://www.posix.nl/linuxassembly/

[16] http://webster.cs.ucr.edu/Page_asm/ArtOfAsm.html

[17] http://linuxassembly.org/

[18] Operating Systems, William Stallings, Academic Service

[19] Gestructureerde computerarchitectuur, Andrew S. Tanenbaum, Academic Service

[20] Handboek voor 80386 Programmeurs, Intel, Kluwer

[21] De Intel 80386, Lance A. Laventhal, Academic Service

[22] http://victoria.uci.agh.edu.pl/doc/khg/khg.html

[23] The Linux Kernel Book, Rémy Card et al., John Wiley & Sons Ltd.

[24] Linux Kernel Internals, Beck M. et al., Addison-Wesley

[25] Unix Systems for Modern Architectures, Curt Schimmel, Addison-Wesley

[26] http://www.tldp.org/LDP/lki/

[27] http://billgatliff.com/articles/emb-linux/interrupts.pdf

[28] ftp://ftp.uk.linux.org/pub/linux/willy/lca/

[29] http://www.tldp.org/HOWTO/GCC-HOWTO/

[30] http://www.kernelnewbies.org/

[31] http://kernel-janitor.sourceforge.net/