IMPLEMENTERING AF EN MULTIPLEXING MIKRO-KERNE

Niels Cunningham Glimø

LYNGBY 2002
EKSAMENSPROJEKT
IMM

Trykt af IMM, DTU

Abstract

Many modern operating systems need to support two radically different types of application: Real-time applications with Quality of Service requirements and traditional applications whose resource requirements can be met in a best effort manner. As it is difficult to deal with both types of application within a single operating system, a number of proposals have been made for alternative approaches in which two or more OSs are combined. In this project, the aim is to investigate one such proposal, in which a small microkernel offers a multiplexing facility, enabling two or more OSs to access the physical hardware in the machine in an efficient manner.

The target of this project is to specify and implement such a multiplexing microkernel. The microkernel itself should only offer very basic facilities, so that an OS can handle processes, memory management and basic disk I/O. Interfaces to the (potential) OSs must be defined which allow:

- access from the OS to hardware devices;
- filtering of the view of the hardware seen by the individual OSs;
- negotiation between the individual OSs and the microkernel with respect to the amount of given resources to be made available to that OS.

In the first instance, filtering and resource negotiation in a static manner (at boot time) should be offered. If time allows, algorithms for allowing these features to be negotiated dynamically can be added. As a concrete example of an OS which can be used in conjunction with the multiplexing microkernel, it is expected that Linux will be used, and many parts of the kernel can be based on facilities which are readily available in existing Linux implementations.

Indhold

Kapitel 1

Indledning

Valget af operativsystem, som skal køres på en datamat, afhænger almindeligvis af de opgave-typer, som skal løses på den valgte platform. Dette vil oftest indebære en analyse af de faciliteter der bliver stillet til rådighed i de forskellige OS'er. Desværre er det sjældent, at et enkelt OS er i stand til at opfylde samtlige krav tilfredsstillende, hvorpå det kan være nødvendigt med multible maskiner med hvert sit OS. Dette kan være uhensigtsmæssig, og en metode til at udnytte fordelene ved hvert OS på én maskine kunne ønskes. Én måde det vil kunne gøres på, er ved at lade flere OS'er køre på den samme maskine. Maskinens arkitekturmæssige faciliteter bliver altså *multiplekset* ud til hvert *gæste-OS*, og dermed vil der kunne skiftes mellem OS'erne efterhånden som behovet opstår. Hvert OS får altså en kopi af maskin-arkitekturen, eller en virtuel maskine (VM).

Fordelene ved denne fremgangsmetode er mange. Når hvert OS bliver afviklet i et lukket og kontrolleret miljø som en udvikler frit kan stille på, opstår muligheden for nemt at teste programmer i mange slags konfigurationer. Gennem en virtuel maskine vil man nemlig kunne tænde og slukke ekstern hardware efter behov, samt analysere en VM's tilstand under udførelsen af applikationer i et OS. Et eksempel kunne være sikker vira-analysering [11]. Andre anvendelser kunne være kørsel af programmer med *Quality Of Service* (QOS) krav. Her vil almindelige programmer afvikles på et standard OS med svage tids-krav, men mere krævende applikationer vil kunne afvikles parallelt på et hårdt *real-tids* OS.

1.1 Eksisterende virtualiseringsprogrammer

Virtualisering af maskiner har været meget udbredt før fremkomsten af personlige datamater. Særligt IBM's mainframes gjorde brug af denne teknik, og har indtil nu stort set været begrænset til disse store systemer. Men inden for de sidste år er interessen for virtualiseret systemer blusset op, i takt med at hurtigere datamater har gjort det muligt.

1.1.1 Bochs

Bochs (bochs.sourceforge.net) er et *open source* IBM PC emulator påbegyndt af Kevin Lawton. I modsætning til hvad der forsøges i dette projekt, simulerer Bochs hver enkel instruktion i IA-32 arkitekturen programelt. Indtil videre op til Pentium niveau, hvilket gør den i stand til at køre de fleste operativ systemer som Linux og Windows NT. Afviklingen af OS'er og deres applikationer vil foregå meget langsommeligt pga instruktionssimuleringen.

1.1.2 WINE

WINE (www.winehq.com) er strengt taget ikke en emulator, men virker ved at implementere Win32 API kald, DirectX API kald m.m i Linux, med henblik på at kunne køre Window applikationer direkte. Denne metode kræver, at den underliggende maskine som værts-OS'et er baseret på, er den samme som applikationernes værts-OS er baseret på. Fordi alle API kaldene er implementeret i Linux vil windowsbaseret applikationer afvikles i fuld hastighed.

1.1.3 Plex86

Plex86 (www.plex86.org) er en opensource virtuel maskine ligeledes af Kevin Lawton. Denne VM udnytter et underliggende OS - i dette tilfælde Linux - til at styre de eksterne enheder, sikre beskyttelse osv. Plex86' VM monitor nøjes blot med at afbilde PC arkitekturen til gæste-OS'et. Ubeskyttede instruktioner fanges vha SBE (forklares senere). Projektets nuværende status er usikker.

1.1.4 Connectix Virtual PC

Kommercielt virtuel maskine, der ligesom Plex86 udnytter den underliggende OS til beskyttelsesfaciliteter, drivprogrammer m.m [2]. Er nået til version fem og er et meget komplet system, der er i stand til at køre flere OS'er i hver deres vindue. Understøtter Linux, alle former for Windows, DOS, OS/2 m.m. Virker ved at eksekvere et gæste-OS' systemniveau kode gennem en *Just In Time* (JIT) kompiler, for derigennem at opnå højere afviklingshastighed end gennem normal oversættelse. Almindelige applikationer bliver eksekveret direkte.

1.2 Opbygning af rapport

Denne rapport består af fire afsnit foruden indledning og konklusion. Først vil en virtuel maskine forsøges specificeret, dvs hvilke problemområder findes der og hvordan de kan løses. I dette kapitel vil der også blive opstillet krav som en maskin-arkitektur skal overholde for at understøtte en sikker virtualisering. Dernæst vil den arkitektur, som er blevet valgt i dette projekt, blive gennemgået med henblik på en mulig virtualisering. Efter det følger en gennemgang af Linux OS. Tilsidst følger der et afsnit om design og implementering af en virtualiseret maskine. Bagerst i rapporten ligger appendiks med kildekode til den implementerede kerne.

Kapitel 2

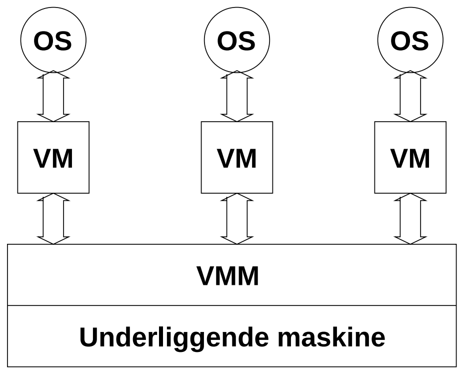
Analyse og specifikation

I dette kapitel vil de specifikke problemer vedrørende implementeringen af en virtuel maskine blive berørt. Emnerne vil blive gennemgået så vidt muligt på en platformsuafhængig måde for at give læseren en overordnet forståelse for de situationer, hvor virtualiseringen kan bryd sammen. Kapitlet vil starte med at definere betydningen af de betegnelser og forkortelser, som vil blive anvendt i løbet af rapporten. I den næste del vil begrebet en virtuel maskine forsøges defineret dels ved at beskrive forskellige former for VM, men også ved at opstille nogle krav som en arkitektur skal overholde for sikkert at kunne blive virtualiseret på sikker vis. Herefter følger en gennemgang af de nøgleområder, som skal simuleres i en virtualiseringsproces. Disse områder baserer sig primært på de faciliteter, der er stillet til rådighed i IA-32 arkitekturen for at understøtte et moderne OS. En komplet teknisk gennemgang af disse faciliteter kan fås hos [14, 7].

2.1 Analyse af krav til en VM

I bund og grund drejer virtualiseringen af en arkitektur sig om, at kopiere de faciliteter som arkitekturen stiller til rådighed i en sådan grad, at ethvert program, som kører oven på virtualiseringen, vil kunne eksekveres på korrekt vis. I denne rapport vil en sådan virtualisering blive kaldt for en *virtuel maskine* (VM). En VM er en reflektion af én maskine og dens faciliteter, dvs kørsel af ét operativ system (OS) vil være muligt. Mere interessant vil det være, hvis *flere* OS'er kunne køre på én maskine. For at dette skal kunne lade sig gøre, må maskinens faciliteter multiplekses. Et program, som gør dette muligt, vil her blive kaldt en *virtuel maskin-monitor* (VMM) [17]. En VMM vil køre oven på den underliggende maskine og dele dens resourcer ud til en mængde VM'er, der hver især eksekverer et OS (kaldet et VMOS) (se figur 2.1 for en illustration).

Mens VM-modulet blot vil være en afbildning af de faciliteter VMM'en multiplekser, vil det for en VMM dreje sig om, hvordan den kan udbyde disse faciliteter og stadig bevare den fulde kontrol. Blandt andet skal VMM'en være i stand til at adskille de forskellige VMOS'er både fra hinanden og fra den faktiske hardware. Også drivning og multipleksing af de eksterne enheder vil skulle håndteres. Hvordan det vil kunne gøres, bestemmes af selve maskinen samt det miljø, en VMM skal eksekveres i. Der vil derfor nu opstilles en række VMM implementeringstyper, hvor fordele og ulemper gennemgås.



Figur 2.1: Opbygning af et virtualiseret system.

2.1.1 Typer af virtuelle maskiner

Kørslen af et operativsystem kræver, at der bliver eksekveret maskininstruktioner på en CPU, hvoraf en vis delmængde skal simuleres i programkode. Antallet af instruktioner, der skal simuleres, kontra antallet, der kan eksekveres direkte, dikterer den type af simulator, der bliver kørt. Der kan defineres følgende hovedtyper af simulatorer [11].

1. Komplet Software-Oversætter Maskine-Monitor (**KSOMM**)
2. Hybrid Virtuel Maskine-Monitor (**HVMM**)
3. Virtuel Maskine-Monitor (**VMM**)

1. Som navnet antyder, er en KSOMM udelukkende baseret på software. I en KSOMM bliver hver enkel instruktion emuleret gennem en lille stykke programkode. Fordelen ved denne metode, er at enhver arkitektur kan implementeres på alle typer maskiner samt, at en KSOMM vil være garanteret fuld kontrol over alle VMOS'erne. Ulempen er, at hastigheden en ovenliggende operativsystem eksekveres med, vil være betydeligt forringet. Dette vil skyldes, at hver enkel simuleret instruktion kan kræve en endog meget kompleks stykke kode. Faktisk vil hastigheden være så ringe, at bare det at køre et enkelt VMOS vil være en prøvelse, de fleste nok vil være foruden!
2. En HVMM kræver i modsætning til en KSOMM, at den virtualiserede maskine og den underliggende maskine er ens. Hermed opstår muligheden for at forbedre udførelseshastigheden ved at afvikle alle instruktioner udført fra brugerniveau direkte på selve maskinen. Kode, der er udført fra systemniveau, vil derimod blive oversat som i en KSOMM, herved sikres der større kontrol af afviklingen af et VMOS. Det er denne type VM mange kommercielle VM'er er baseret på.
3. Ligesom en HVMM skal en VMM køre på den samme arkitektur som den forsøger at multiplekse. I modsætning til HVMM vil en VMM også eksekvere så mange instruktioner fra systemkode direkte på maskinen som muligt. Det vil dog være nødvendigt med en simulering af nogle instruktioner, da en vis form for sikkerhed skal oprettes. Hvilke instruktioner det vil dreje sig om, vil kræve en nøjere analyse af det system, som skal virtualiseres. Denne monitor-type vil være den hurtigste og dermed også den mest attraktive at implementere. Af den grund vil det også være denne type, dette projekt vil handle om.

Arkitekturelle krav til VMM

En VMM kører direkte ovenpå den underliggende maskine, og er i bund og grund et minimalt operativsystem med understøttelse for virtualisering. Den holder styr på al ressourceallokering af hardware samt korrekt schedulering mellem gæste-styresystemerne. For at opnå en korrekt og frem for alt sikker implementering af denne type VMM, skal den valgte maskin-arkitektur indeholde faciliteter, som kan sikre, at der kan opretholdes en konsistent tilstandsafbildning overfor hvert VMOS. Der kan af den grund opstilles en række krav til den valgte maskin-arkitektur for, at en virtualisering skal kunne lykkedes [11]:

Krav 1 Måden en CPU eksekverer en ikke-privilegeret instruktion på skal være ens på både brugerniveau og systemniveau.

Krav 2 Der skal eksistere en måde at beskytte både det underliggende system samt andre VM'er fra den aktive VM. Dette kunne være en metode til at tildele områder sit eget adresseområde og/eller en virtuel adresseringsmetode såsom paging.

Krav 3 Det skal være muligt at kunne afbryde til den underliggende VMM, når en sårbar instruktion forsøges udført. Det skal desuden også være muligt for en VMM at simulere den pågældende instruktion. Disse sårbare instruktioner er:

- **Krav 3A** Instruktioner som ændrer eller refererer til den aktive VM's eller den faktiske maskines eksekveringstilstand.
- **Krav 3B** Instruktioner som ændrer eller refererer til sårbare registre eller hukommelsesområder.
- **Krav 3C** Instruktioner som ændrer eller refererer til beskyttelsessystemet, hukommelsessystemet eller den virtuelle adresseringsmetode.
- **Krav 3D** Alle I/O instruktioner.

Krav-analyse af IA-32

Pga dens store udbredelse og (forholdsvis) nem adgang til informationer, bliver VMM'en i dette projekt forsøgt implementeret på Intel Corporations IA-32 arkitektur. IA-32 skal derfor analyseres udfra kravene stillet foroven.

Krav 1 IA-32 eksekverer systemniveau kode og brugerniveau kode på samme måde. Eneste forskel er, at priviligerede instruktioner, der udføres under brugerniveau, genererer en afbrydelse.

Krav 2 IA-32 tilbyder både segmentering og paging. Derudover er det muligt at afvikle kode på forskellige beskyttelsesniveauer.

Krav 3 IA-32 arkitekturen bruger afbrydelser til at forhindre, at privilegieret instruktioner bliver udført af upriviegieret kode. Dette giver programmøren mulighed for at behandle disse instruktioner. Desværre indholder IA-32 følsomme ubeskyttede instruktioner. Disse instruktioner vil ikke blive afbrudt selvom de bliver afviklet på brugerniveau. VMM'en vil altså *ikke* få mulighed for at simulere dem. En måde at takle disse instruktioner på vil blive givet senere i kapitlet.

Program VMM

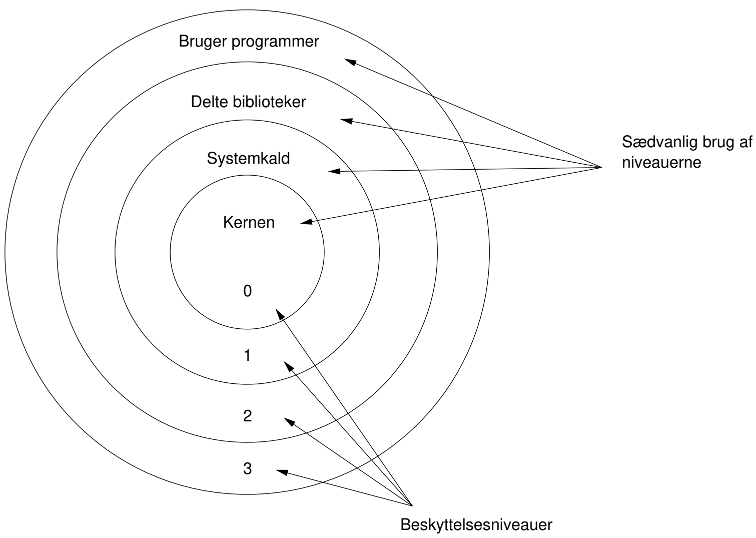
Til sidst skal det nævnes, at mikrokerne VMM, som bliver implementeret i dette projekt, ikke er den eneste måde, hvorpå en VMM kan implementeres. Den anden type VMM kører som et almindeligt applikationsprogram på et *værts-OS*. Denne form for VMM vil udelukkende håndtere selve virtualiseringsfladen og overlade værts-styresystemet til hukommelseshåndtering, process-schedulering, ressourceallokering samt I/O håndtering. De krav der er opstillet til en kerne VMM gælder også for denne type VMM, dog med den forskel, at **krav 2** er ændret til, at værts-OS'et skal kunne stille de nødvendige primitiver til rådighed. Disse inkluderer bl.a. mulighed for at sende signalet videre til VMM'en når VM'en bliver afbrudt pga. udførelsen af en følsom instruktion. Det skal også være muligt for en VMM at køre den virtuelle maskine som en under-proces.

2.2 Virtualiseringsproblemer

For på korrekt vis at kunne virtualisere en maskine, kræver det, at et OS, som kører oven på en VM, ikke er i stand til at 'opdage' de forskelle der vil være mellem maskinens sande tilstand og de virtualiserede omgivelser. Dette vil sige, at alle de steder hvor VMOS'et læser eller manipulerer enten direkte eller indirekte med systemkritiske funktioner skal identificeres. Her vil de forskellig funktioner som en moderne arkitektur tilbyder blive gennemgået med henblik på en virtualisering. Nedenstående gennemgange af løsninger er primært med Intel IA-32 arkitekturen i tankerne, selvom det såvidt muligt forsøges at holde forklaringerne platformsuafhængig.

2.2.1 Rettighedsproblemer

De fleste maskin-arkitekturer tilbyder at lette arbejdet for operativsystemer, når de skal forsøge at differentiere mellem systemkode og programkode. Dette gøres ved at tildele koden *rettighedsniveauer*. Ofte vil et operativsystem dele systemet op i flere forskellige lag alt efter funktionalitet. Et eksempel fra IA-32 arkitekturen kan ses i figur 2.2 [1], hvor beskyttelsesværdi 0 er den mest privilegeret.

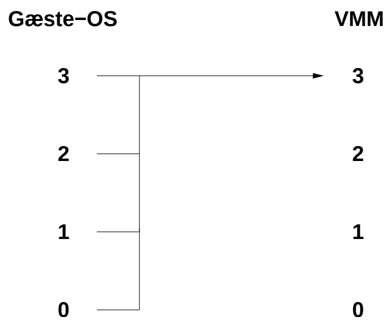


Figur 2.2: Rettighedsniveauer på Intel IA-32.

Disse rettighedsniveauer bruges, når der skal læses, skrives eller eksekveres fra et segment eller en side. Særlige følsomme instruktioner vil også være udsat for en rettighedskontrol for at sikre, at de ikke bliver udført af processer med for høj en beskyttelsesværdi. Disse rettighedskontrol er i bund og grund ret simple:

En proces befinder sig i et hukommelsessegment med en given rettighed. Hvis den forsøger på en eller anden måde at komme i kontakt med et andet segment, vil maskinen sørge for at sammenligne de to segmenters rettighedsniveauer og evt. afbryde til kernen, hvis betingelserne ikke er opfyldte. Dette vil f.eks gælde, hvis en brugerproces i et operativsystem forsøger at referere til systemhukommelse. Også paging kan have beskyttelse baseret på rettigheder og virker på en lignende måde.

Ved en virtualisering af en maskin-arkitektur er problemet, at den underliggende VMM har den højeste rettighedsniveau (værdi 0 på IA-32), mens de OS'er, der eksekverer på hver deres VM, også forventer, at de har samme høje rettighedsniveau. For at lave en korrekt implementation, er det kritisk, at alle instruktioner, der kan afsløre virtualiseringen, bliver fanget af VMM'en, når de forsøges udført af en af VMOS'erne. Dette indebærer at VMOS'erne skal eksekveres med laveste rettighedsniveau (værdi 3 på IA-32). Der sker altså en afbildning af de højeste niveauer ind til den laveste (se figur 2.3).



Figur 2.3: Rettighedsafbildning fra højste til laveste.

Det er denne afbildning, der skaber problemer. De strukturer som VMOS'et forventer at bruge - såsom descriptor-tabellerne - er konstrueret, som om VMOS'et er i besiddelse af den fulde kontrol. Alle de steder i VMOS'et, hvor rettighedsniveauet er sat for højt, skal altså fanges og modificeres når VMOS'et forsøger at bruge dem.

Rettighedsniveauet er også afgørende for hvilke instruktioner, der skal virtualiseres. Nogle af disse typer instruktioner vil blive fanget af VMM'en, da de bliver eksekveret på brugerniveau, men ikke alle. De specifikke detaljer vil afhænge af den hardware-platform, man har valgt at implementere VMM på. Nedenstående instruktionstyper kan skabe rettighedsproblemer:

- Instruktioner der læser eller undersøger et segments rettighedsniveau. En rettighedskontrol vil sandsynligvis blive foretaget.
- Instruktioner der er i stand til at skrive til segment-registre. En rettighedskontrol vil sandsynligvis blive foretaget.
- Instruktioner som kan læse værdierne i segment-registre. Et operativsystem vil kunne undersøge og opdage, at den befinder sig på et andet rettighedsniveau end forventet.
- Instruktioner der foretager spring mellem segmenter.

Virtualiseringen af disse instruktionstyper beskrives i et senere kapitel.

2.2.2 Timer problemer

Ved afviklingen af et styresystem og dertilhørende applikationer, vil der normalt eksistere mange handlinger, som på en eller anden måde er afhængig af tidens passage. Det kunne være sig pauser i et program, skiften mellem processer eller slet og ret det aktuelle klokkeslag. En arkitektur tilbyder en række funktioner/enheder for at opfylde disse krav.

- RTC (Real Time Clock)
- TSC (Time Stamp Counter)
- PIT (Programmable Interrupt Timer)

En RTC er en selvstændig ekstern enhed med eget batteri, og er i stand til periodevis eller på bestemte tidspunkter at sende afbrydelser til kernen, dermed kan den endog fungere som alarm. Et operativsystem vil dog normalt kun bruge RTC'en når det læse eller stille den aktuelle tid.

En moderne arkitektur vil ofte indeholde en register, som opdateres automatisk af maskinen hvert klokkeslag. Dette register kaldes en TSC. Fordi TSC'en tælles op så tit vil et OS være i stand til at kunne foretage meget præcise tidsmålinger ved at læse dette register.

For automatisk at kunne skifte mellem hver proces, bliver et styresystem normalt afbrudt med et vis tidsinterval (ofte 100 HZ) kaldet en *tick*. Dette gøres vha. en PIT, der som RTC er en ekstern enhed, som bliver refereret til gennem I/O kald. Et styresystem vil tælle disse ticks for hver gang der opstår en. Dette tal bruges så f.eks. til når handlinger skal fortages et bestemt tidspunkt ud i fremtiden m.m. Hvis højere præcision er påkrævet, suppleres der med indlæsninger fra TSC.

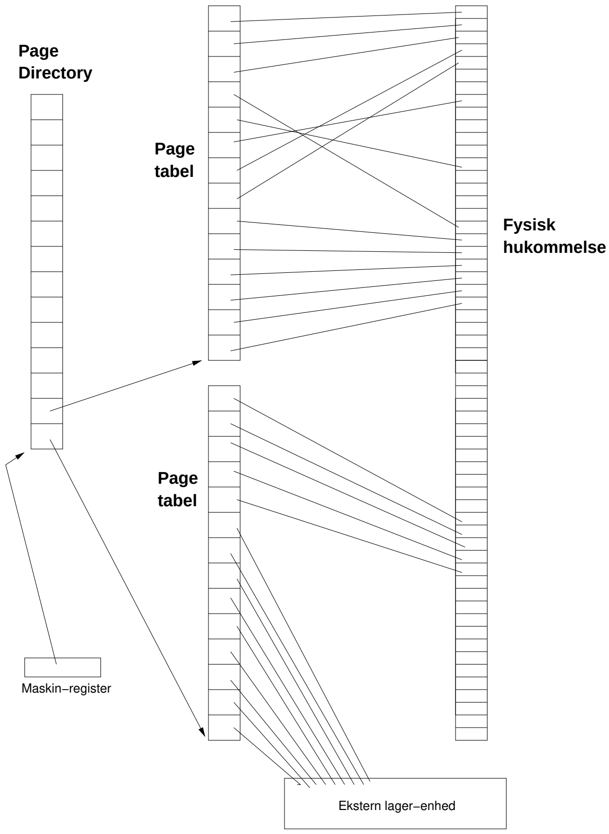
Timer virtualisering

Som sagt tæller et operativsystem antallet af ticks og bruger dem til at foretage tidsafhængige handlinger, dvs. hver gang VMM'en modtager en afbrydelse fra PIT'en sendes kontrollen videre til det valgte VMOS'. VMOS'et tæller denne tick og fortsætter eksekveringen. Da selve PIT'en er en ekstern enhed skal den emuleres på linie med de andre nødvendige enheder og er hardware afhængigt. Det samme gælder for RTC'en, mens TSC'en emuleres gennem den instruktion, som er i stand til at læse registret.

Et problem som vil opstå er, at ved kørsel af f.eks to VMOS'er, vil antallet af ticks hvert OS modtager være halveret ved almindelig sekventielt schedulering. Dette vil betyde, at hver timer-forsinkelse vil i realiteten være fordoblet. Dette burde dog ikke have nogen indflydelse for normal kørsel af ikke real-tids OS'er og applikationer.

2.2.3 Page tabeller

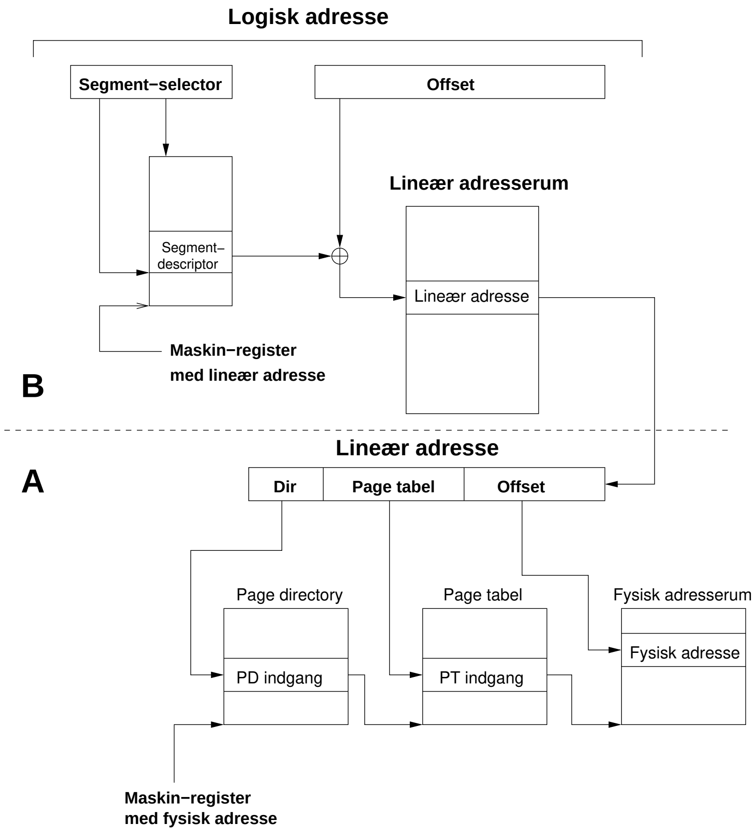
En af grundstenene i et moderne operativsystem er *paging systemet*. Paging er en maskinarkitekturmessig funktion, der gør det muligt i bund og grund at 'erstatte' en datamats fysiske hukommelsesadressering med et såkaldt virtuelt adresserum. Afbildningen vil være usynlig set ud fra en proces' synsvinkel, dvs. adresserummet vil være uændret, men i virkeligheden kan potentielt alle virtuelle adresser være afbildet over i en hvilken som helst fysisk adresse. Se figur 2.4. Denne funktionalitet gøre det muligt for f.eks. et styresystem at udnytte lagerpladsen på et eksternt diskdrev, hvis den installerede hukommelsesmængde i datamaten er mindre end det samlede fysiske adresserum. Afbildningen tillader, at en proces vil kunne referere et sted i den virtuelle hukommelse, som ikke var afbildet til egentlig installeret hukommelse. Dette kan lade sig gøre ved, at styresystemet gemmer et stykke af maskinens hukommelse på en ekstern enhed, hvorefter den virtuelle hukommelsesadresse vil kunne afbildes over i den nu frie hukommelsesblok.



Figur 2.4: Afbildning af den lineære adresserum over i den fysiske adresserum.

Paging virker ved, at den fysiske hukommelse bliver delt op i lige store stykker eller sider (deraf navnet paging). Størrelsen af siderne er maskinafhængig, men er normalt på 4 kb for en 32 bit adresserum. For at holde styr på disse sider bruges en *side-tabel*, hvor hver indgang indeholder en peger til den fysiske adresse. I en moderne datamat med 32 eller 64 bit adressering vil der - alt efter side-størrelse - ofte være mange millioner sider, hvilket vil give en uforholdsmæssig stor side-tabel. Løsningen vil tit være at dele den op i mindre tabeller, men for overskueligheden vil nedenstående forklaring nøjes med én tabel.

Den fysiske adresse til side-tabellen gemmes i et maskinregister, hvorfra den læsees automatisk af maskinen, hver gang en virtuel adresse refereres. Der sker så det, at adressen bliver delt op i flere dele alt efter hvor mange side-tabeller der er. Den mest betydende del angiver indgangen til tabel-fortegnelsen, hvori pegeren til begyndelsen af den rigtige side i den fysiske adresserum findes. Adressen til indgangen fås ved at maskinen lægger indeksværdien til adressen gemt i maskin-registret. Tilsidst, når siden er fundet, indekseres der til den rigtige fysiske adresse i siden ved at lægge den sidste del af den lineære adresse til side-begyndelsens fysiske adresse. Se figur 2.5 pkt A. En funktionel gennemgang af adressebehandlingen i paging kan ses i figur 2.6 [16].



Figur 2.5: Illustration af adressebehandling i en datamat (eksemplet er taget fra IA-32).

En vigtig del af paging, som ikke er blevet berørt endnu, er muligheden for at beskytte hukommelsesområder. Dette er af højste relevans for styresystemer, men også for at sikre korrekt og sikkert virtualisering af en maskinarkitektur. Beskyttelsessystemet udnytter, at siderne i adresseområdet er stillet sådan op, at de mindst betydende bits i starten af hver side er lig nul. Dette ved paging mekanismen i maskinen og springer dem derfor over, når adressen hentes fra side-tabellen. Herved opstår der mulighed for at bruge disse bits til andre formål, f.eks. til beskyttelse, hvilket betyder, at der afbrydes til kernen hvis betingelserne ikke er opfyldte (f.eks. forkert beskyttelsesniveau eller siden er skrivebeskyttet). Om page baseret beskyttelse er en mulighed afhænger dog af den valgte arkitektur.

Virtualisering af paging

I et virtualiseret system som i dette eksamensprojekt, vil der opstå en konflikt mellem pagingsystemet i VMM'en og pagingsystemet i VMOS'et. Da det kun er VMM'en, som kan bruge maskinens underliggende hardware, vil det være nødvendigt at kopiere pagingsystemet. En måde at gøre dette på kunne være vha. *skygge-paging* [12, 17, 15]. Skygge-paging virker ved at gemme en kopi af VMOS'ets side-tabel i hukommelsen. Disse kopier svarer til VMOS'ets, men afbilder blot VMOS'ets virtuelle adresser til de rigtige steder i den fysiske hukommelse. Teknikken kan bedst illustreres med et eksempel [17].

VMM'ens side tabel viser at side 6 i VMOS'ets allokerede hukommelse befinder sig i side 100 i den fysiske hukommelse (pkt A). VMOS'et har sin egen side-tabel for at holde styr på dens virtuelle adresserum. Ifølge tabellen befinder side 16 i VMOS'ets virtuelle adresserum sig i side 6 i dens af VMM'en allokerede hukommelsesområde (hvilket VMOS'et 'tror' er den fysiske hukommelse) (pkt. B). Hvis VMOS'et skal kunne referere hukommelsen på korrekt vis, må værts'OS'et opbygge en skygge side-tabel, der afbilder VMOS'ets virtuelle side 16 (side 6 i dens 'rigtige' hukommelse) til den fysiske hukommelses side 100 (pkt. C). Eksemplet er illustreret i figur 2.7.

Da operativsystemer (f.eks. Linux) sjældent initialiserer deres side-tabeller én gang for alle i starten, men snarer gør det løbende, vil det være nødvendigt også at oprette skygge side-tabellerne dynamisk. Denne metode udnytter at VMOS'et kører med et højt beskyttelseniveau og derfor afbryder til VMM'en, når det forsøger at gemme den fysiske startadresse til en side-tabel i det relevante maskinregister. Efter afbrydelsen har VMM'en adressen til VMOS'ets side-tabel. Denne gemmer den og opretter derefter en skygge side-tabel, hvor indholdet af hver indgang er lig nul.

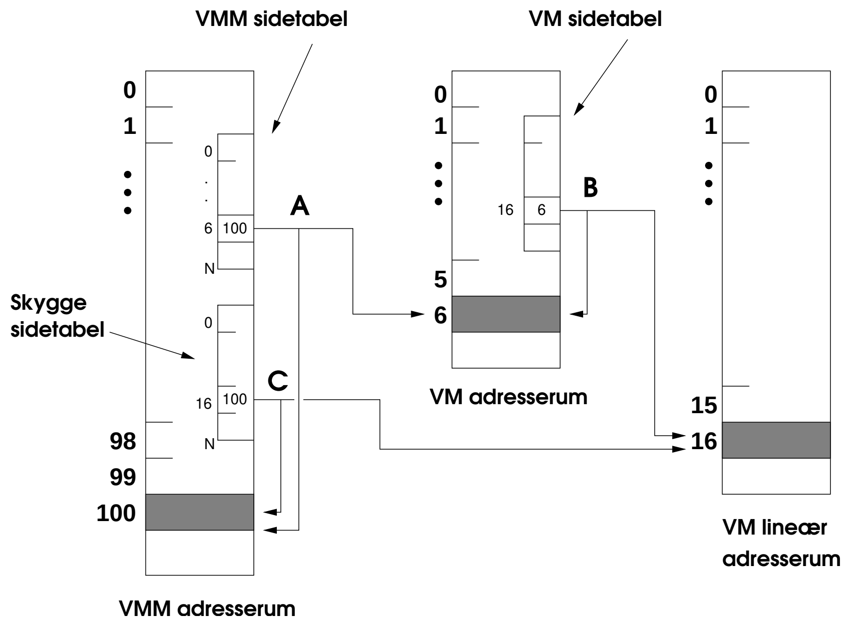
Nulstillingen af skygge tabel-fortegnelsen vil resultere i, at hver reference VMOS'et laver til sit lineære adresserum vil skabe en afbrydelse til VMM'en. VMM'en vil sende afbrydelsens fejlkode tilbage til VMOS'et, der vil konstatere at indgangen er lig nul og straks allokere en side-tabel til det relevante side-indgang. Den side-tabel, som her bliver opdateret, vil være VMOS'ets egen og altså *ikke* skygge tabellen. Dette skyldes, at VMOS'et kun kender adressen til sin egen tabel, da skygge tabellen holdes usynlig. Når VMOS'et forsøger at opdatere en side-tabel, vil der ske endnu en afbrydelse tilbage til VMM'en, da VMOS'ets tabeller er skrivebeskyttet. Her vil VMM'en sørge for at ajourføre både VMOS'ets side-tabeller samt skygge tabellerne ved at undersøge den lineære adresse VMOS'et forsøger at skrive til. Et overblik over den proces kan fås i afsnittet om paging i IA-32.

Det skal her fremhæves, at for VMOS'et er skygge side-tabellerne skjulte. Ændrer VMOS'et på et tidspunkt indholdet af sin egne side-tabeller, skal der altså afbrydes til VMM'en for at den også kan opdatere sine skygge tabeller. Dette gøres ved at bruge paging til at skrivebeskytte VMOS'ets side-tabeller.

Denne fremgangsmåde vil have det problem, at der vil blive skabt mange side afbrydelser (*page faults*) til VMM'en, mens skygge side-tabellerne bliver bygget op. Dette vil have den konsekvens, at udførelseshastigheden vil blive forringet pga. de mange afbrydelser. Det vil dog kun gælde indtil skygge tabellerne er bygget op. Yderligere forbedring vil kunne opnås ved at gemme skygge-tabellerne i hukommelsen mellem hvert VM-skift [12]. Bagsiden ved

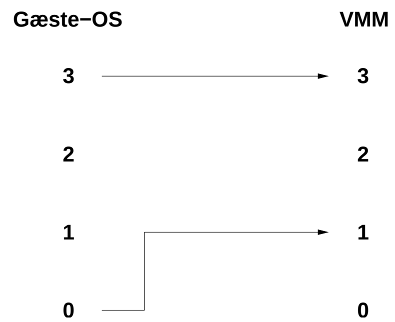
$$
\begin{array}{l}
f(la, op) \stackrel{\text{def}}{=} \text{let } p = la \text{ div } pagesize, \\
\qquad\qquad\qquad\quad b = la \text{ mod } pagesize \\
\qquad\qquad\quad \text{in if } PT[p]. = \text{invalid} \\
\qquad\qquad\qquad \text{then } fault(\text{missing\_page}) \\
\qquad\qquad\qquad \text{elsif } b > PT[p].length \\
\qquad\qquad\qquad\qquad \text{then } fault(\text{limit}) \\
\qquad\qquad\qquad\qquad \text{elsif } op \not\subseteq PT[p].operations \\
\qquad\qquad\qquad\qquad\quad \text{then } fault(\text{protection}) \\
\qquad\qquad\qquad\qquad\quad \text{else } PT[p].base + b \\
\qquad\qquad\quad \text{end}
\end{array}
$$

Figur 2.6: Adressebehandling i paging.



Figur 2.7: Illustration af brugen af skyggesider (*shadow paging*).

dette er selvfølgelig den høje mængde hukommelse, som vil det kræve at gemme skygge tabellerne. Dette vil dog være til at overskue, da selv almindelige hjemmedatamater i dag er i besiddelse af endog meget store mængder hukommelse.



Figur 2.8: Afbildning fra højeste til næst-højeste.

Et problem, som endnu ikke er blevet berørt er rettighedsproblemet. Som vist i figur 2.3 i afsnittet om rettighedsproblemer foretages der en afbildning af de laveste beskyttelsesniveauer over i det højeste beskyttelsesniveau. Under oprettelsen af skygge-tabellerne vil beskyttelsesniveauet blive ændret til det højeste niveau for at VMOS'et kan køre korrekt. Imidlertid gælder det, at da både VMOS'ets brugerkode og systemkode har samme høje niveau, vil brugerkoden have adgang til hukommelse, der normalt er beskyttet fra VMOS'ets side. For at forhindre dette findes der en metode [10], som er rettet imod Intel arkitekturen, men som kan bruges i lignende arkitekturer. I IA-32 har paging kun to beskyttelsesniveauer (bruger og system). Dette udnyttes til at sikre, at beskyttelsesniveauet i skygge-tabellens indgange er lig siderne i VMOS'ets tabel-indgange (i modsætning til før hvor de alle var sat til højeste niveau). For at dette skal virke vil det dog indebære, at VMOS'ets systemniveau kode bliver kørt på det næst-laveste beskyttelsesniveau istedet for det højeste (se figur 2.8. Herefter vil beskyttelsen for VMOS'et virke korrekt, til gengæld kan der opstå problemer med den generelle virtualisering af arkitekturen. Med reduceringen af beskyttelsesniveauet for VMOS'ets systemkode kan det ske, at visse følsomme instruktioner, som vil kunne afsløre VMM'en, nu vil få lov til at eksekvere. Det vil altså stadig være nødvendigt at emulere disse instruktioner, men da der ikke længere automatisk afbrydes til VMM'en, vil dette være problematisk. En mulig løsning gives dog i en senere sektion.

En nemmere løsning ville være at have to skygge-tabeller; en til brugerkode og en til systemkode. Indgangene i skygge side-tabellerne til brugerkoden skal blot have de samme niveauer som den rigtige tabel, og korrekt kørsel vil være sikret. Derimod skal indgangene i skygge side-tabellerne rettet mod systemkode alle have deres beskyttelsesniveauer ændret til højeste niveau for at sikre korrekt kørsel. Denne løsning vil kræve lidt mere hukommelse, men til gengæld vil man slippe for problemer med at emulere instruktioner.

2.2.4 Descriptor-tabeller

Mens paging tillader, at beskytte og afbilde hukommelse, forbliver det adresserum som den aktuelle proces kører i uændret. Tit har applikationer og styresystemer brug for flere uafhængige adresserum for bedst at kunne adskille de forskellige processer, hvilket også er nyttig under afviklingen af en VMM. Måden til at gøre dette er ved at dele hukommelsen op i blokke kaldet *segmenter*. Disse blokke kan have en vilkårlig størrelse - helt op til det fulde adresserum. Dette er i modsætning til paging, som deler hukommelsen op i blokke med en fast størrelse. Ældre arkitekturer kunne dog ofte kun have segmenter af en fast størrelse (f.eks. IA-16 fra Intel). Hvert segment bliver beskrevet i en *descriptor* og kan - som i siderne i paging - tildeles et rettighedsniveau samt stille skrive/læse/eksekvér rettigheder m.m. Præcis hvad der kan stilles varierer dog efter arkitektur. Som i paging er descriptorerne gemt i en *descriptor-tabel*, men i modsætning til en tabel-fortegnelse, er det her den *lineære* adresse til descriptor-tabellen, som gemmes i et maskin-register. Se figur 2.5 pkt B. En funktionel beskrivelse af segmenterings adresse-behandling kan ses i figur 2.9 [16].

$$
\begin{array}{llll}
f((s,b),op) \overset{\text{def}}{=} & \text{if } ST[s] = \text{invalid} & & \\
& \text{then } \mathit{fault}(\text{missing\_segment}) & & \\
& \text{elsif } b > ST[s].\mathit{length} & & \\
& & \text{then } \mathit{fault}(\text{limit}) & \\
& & \text{elsif } op \notin ST[s].\mathit{operations} & \\
& & & \text{then } \mathit{fault}(\text{protection}) \\
& & & \text{else } ST[s].\mathit{base} + b
\end{array}
$$

Figur 2.9: Adressebehandling i segmentering.

Der gælder så det, at i arkitekturer som IA-32 med stor understøttelse af segmentering, er det muligt at kombinere både paging og segmentering. Her vil paging være det ovenliggende system, således at segmenterings rettigheder og grænser vil gælde forud for pagings tilsvarende faciliteter. En funktionel beskrivelse af paging og segmentering kan ses i figur 2.10 [16].

I et styresystem vil segmentering normalt blive brugt til at holde styr på, hvad der er systemniveau kode og hvad der er brugerniveau kode. Derudover kommer evt. et antal

```
f((s,b), op) ≝ if ST[s] = invalid
               then fault(missing_segment)
               elsif b > ST[s].length
                     then fault(limit)
                     elsif op ∉ ST[s].operations
                           then fault(protection)
                           else let p = b + ST[s]. base div pagesize,
                                    b' = b + ST[s]. base mod pagesize,
                                    pt = ST[s]. ptable
                                in if pt[p] = invalid
                                   then fault(missing_page)
                                   elsif op ∉ pt[p]. operations
                                         then fault(protection)
                                         else pt[p]. base + b
                                end
```

Figur 2.10: Adressebehandling i paged segmentering.

segmenter til de forskellige stakke, der er i brug. Under udførelsen af noget kode vil man uvægerligt komme til at skulle springe fra et kodesegment til et andet. For at kunne gøre det, er det nødvendigt at fortælle spring-instruktionen, hvilket segment der skal springes til. Måden det gøres på er vha. en *selector*. En selector er en værdi, som indeholder indeksværdien til segmentet i descriptor-tabellen samt evt. ekstra information som den kaldte rettighedsniveau og den pågældende descriptor-tabel type.

Virtualisering

Ligesom ved paging vil der opstå en konflikt mellem VMM'ens brug af descriptor-tabellerne og VMOS'ets brug af descriptor-tabellerne. En lignende fremgangsmåde som pagings brug af skygge-tabeller kan også bruges til at virtualisere segment-tabellerne - dog med nogle ændringer. En variant af denne løsningsmodel beskrives delvist i [15, 10].

Konflikten mellem de to descriptor-tabeller kan løses ved at udnytte, at en descriptor-tabel kan normalt indeholde flere tusinde indgange, men at et operativsystem under normalt brug aldrig vil komme til at udnytte denne kapacitet. Det giver mulighed for at VMM'en - under oprettelsen af VMM'ens descriptor-tabel - kan lægge dens egne descriptorer højt oppe i tabellen, mens de nedre blot bliver nulstillet. Denne metode udnytter, at en maskine foretager en kontrol af descriptoren i tabel-indgangen og afbryder, hvis den ikke opfylder betingelserne. Herefter kopieres VMOS descriptoren over til den nulstillede indgang. Præcis hvor højt oppe i tabellen VMM-descriptorerne skal lægges må afhænge af erfaring, den tiltænkte brug af VMOS'erne samt hvilket OS der skal køres. Dette skyldes, at de mange indgange primært vil blive brugt til at give hver proces i VMOS'et sin egen descriptor (en såkaldt *TSS descriptor* eller proces-descriptor i IA-32), så jo højere belastning jo større fare for at overlappe VMM descriptorerne. VMOS'erne i dette projekt vil dog næppe være udsat for så stor en belastning. Problemet formindskes yderligere, da ikke alle operativsystemer behøver at bruge proces-descriptorer, og vil derfor helt kunne undgås med lidt forsigtighed. Fremgangsmåden beskrives:

1. Da VMOS'et eksekveres som applikationsniveau-kode, vil der blive afbrudt til VMM'en når adressen til VMOS'ets descriptor-tabel forsøges gemt i det relevante maskin-register. VMM'en gemmer nu adressen til descriptor-tabellen. Der springes over instruktionen og returneres til VMOS'et.
2. Den første gang den aktuelle VMOS forsøger at referere til en bestemt descriptor i descriptor-tabellen gennem f.eks. et proces-skift eller et kode-spring, konstateres der at descriptoren er nulstillet og der afbrydes til VMM'en.
   (a) Da indeks-værdien til descriptoren i tabellen er kendt gennem afbrydelsens fejlkode, og da adressen til VMOS'ets descriptor-tabel ligeledes er kendt, kan den korrekte descriptor kopieres fra VMOS'et til VMM'ens descriptor-tabel.
   (b) For at sikre efterfølgende korrekt opførsel sættes descriptorens rettighedsniveau til brugerniveau (se afsnittet om rettighedsproblemer).
   (c) Start-værdien til VMOS'ets allokeret hukommelse adderes til descriptorens start-værdi. Desuden ændres længden af segmentet, hvis det er nødvendigt for at sikre, at VMOS'ets adresserum ikke overlapper noget.
   (d) Hvis der findes en acces-bit, der fortæller om segmentet er blevet brugt, sættes denne. Dette gælder både for VMM descriptor-tabellen samt VMOS'ets tabel. Dette sikrer at begge tabeller hele tiden er synkroniseret. VMOS'et kan dog finde på at nulstille flaget i dens egen tabel, hvilket der skal tages højde for (se forneden).
3. Tilsidst returneres der fra VMM'en og VMOS'et kan nu gentage instruktionen uden afbrydelser, da descriptoren er på plads med den korrekte rettighedsniveau.

Et særligt problem ved at have flere descriptor-tabeller er, at det vil være nødvendigt at sikre, at tabellerne bliver holdt ajour. Når der bliver skiftet til et nyt segment, vil maskinen automatisk opdatere acces-flaget i descriptoren. Det vil også være nødvendigt at opdatere den aktuelle VMOS' descriptor-tabel som beskrevet i metode gennemgangen ovenfor. Desuden vil det ske, at VMOS'et vil forsøge at ændre eller tilføje en descriptor i dens tabel. Det vil derfor være nødvendigt at kunne fange handlingen når det sker. Dette vil kunne gøres vha. paging mekanismen. Når gæst-OS'ets descriptor-tabel forsøges gemt i maskin-registret afbrydes der til kernen og VMM'en kan gemme adressen. Herefter sættes den side som adressen befinder sig i til et sikkerhedsniveau mindre for at sikre, at VMOS'et hverken kan læse eller skrive til tabellen. Da descriptor-tabellen i princippet kan befinde sig hvor som helst på siden og endda overlappe to på hinanden følgende sider, vil det også være nødvendigt at beskytte den næste side. Heldigvis bliver størrelsen af descriptor-tabellen gemt i dens maskin-register under en arkitektur som IA-32. Antallet af sider som skal beskyttes kan dermed fås herfra.

2.2.5 Interrupt håndtering

En meget vigtig del af virtualiseringsprocessen er en korrekt håndtering af afbrydelser eller *interrupts*. Afbrydelser giver et styresystem mulighed for at behandle en lang række af hændelser i datamaten. Listen er lang og spænder fra basale eksekveringsfejl og beskyttelsesfejl til håndtering af eksterne enheder og schedulering af processer for at nævne nogle eksempler. På mange måder minder afbrydelses-håndtering om både paging og descriptor-behandling. Et styresystem opretter en afbrydelses-tabel og gemmer den lineære adresse til den i et maskin-register. Afbrydelses-tabellen indeholder i hver indgang en afbrydelses-descriptor, der minder om dem der findes i descriptor-tabellen. Der er dog den forskel, at afbrydelsesdescriptorerne indeholder en selector til en segment-indgang i descriptor-tabellen samt en indeks-værdi, som angiver hvor i segmentet eksekveringen skal begynde. Se figur 3.5 for et eksempel fra IA-32. Mekanismen virker ved, at der forekommer en eller anden form for hændelse, hvorefter maskinen afbryder eksekveringen af den kørende proces. Hændelsen har en værdi alt efter, hvad det er, der er sket. Denne værdi fungerer som indeks til en indgang i afbrydelses-tabellen. Dette sted, som der bliver sprunget til gennem descriptoren, indeholder styresystemets afbrydelseskode eller *interrupt handler* kode. Her bliver fejlmeddelelsen behandlet, hvorefter der vendes tilbage til processen (eller skiftes til en ny proces) og systemet kan fortsætte.

Virtualisering af afbrydelser

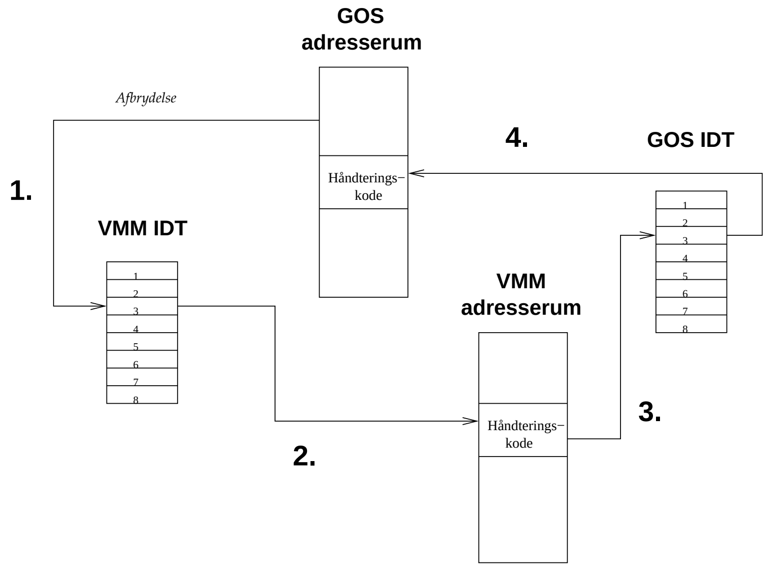
Da det ikke på forhånd kan vides, hvordan et VMOS vil behandle en afbrydelse, vil det være nødvendigt at lade VMOS'et selv sørge for at håndtere afbrydelsen. Dette kan gøres ved at initialisere alle indgange i VMM'ens afbrydelses-tabel til at pege mod VMM'en. Alle pladser i VMM'ens afbrydelses-tabel skal i princippet initialiseres, da det ikke kan vides hvilke afbrydelsesvektorer VMOS'et vil gøre brug af. I virkeligheden kan det dog indrettes alt efter, hvilket OS der skal køre på VMM'en. Det er nemlig de færreste operativsystemer, som gør brug af alle indgange.

Den underliggende maskine vil som regel have afsat et bestemt antal fejl-afbrydelser til at håndtere beskyttelsesfejl, f.eks dividér med nul m.m.. Disse vil normalt ligge samme sted i afbrydelses-tabellen, så VMM'en vil være nødt til at undersøge hvem det var, der lavede afbrydelsen; den selv eller en af VMOS'erne. Hvis det var et VMOS, skal den korrekte descriptor i VMOS'ets tabel findes v.h.a. fejlvektoren samt adressen til VMOS'ets afbrydelses-tabel. Den sidste blev gemt, da VMOS'et forsøgte at lagre den i det relevante maskin-register. Når det er gjort, kan der springes til VMOS'et efter, at fejlvektoren og andet nødvendigt information er gemt på VMOS'ets systemstak. Samme fremgangsmåde skal der bruges til at behandle VMOS'ernes systemkald; bestemme hvor afbrydelsen kom fra, finde descriptoren og springe til stedet. Et eksempel på forløbet kan ses i afsnittet om afbrydelser i IA-32 samt i figur 2.11.

Mange eksterne I/O enheder afgiver også afbrydelser, men har den forskel, at de kan programmeres af en *interrupt controller*. De fleste styresystemer, der baserer sig på Intels arkitektur vil altid placér disse på nogle bestemte pladser [8]. Håndteringen af dem vil forløbe på samme måde som de andre afbrydelser.

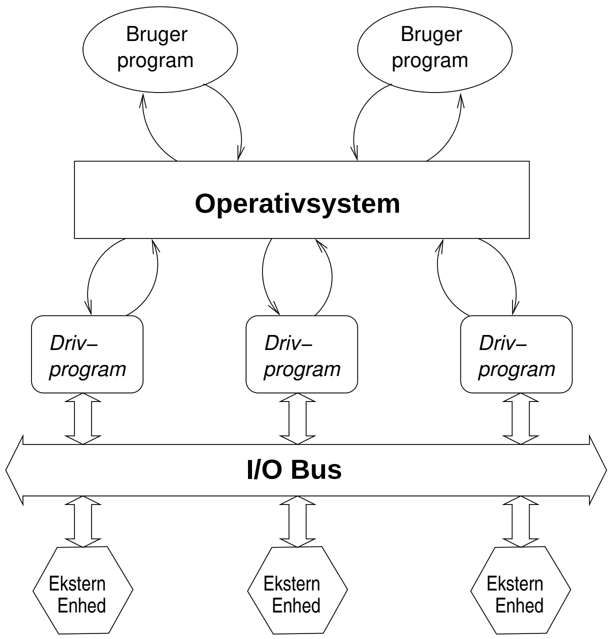
2.2.6 I/O enheder

Næsten alle maskinarkitekturer vil - foruden dens egen funktionalitet - også have tilknyttet en række eksterne enheder. Disse eksterne enheder kan være stort set alt fra lagerplads og lyd-funktionalitet til hukommelsesbusser og real-tids ure. For at drage nytte af disse eksterne enheder vil et operativsystem indeholde en samling rutiner - enten i selve kernen eller gennem en række moduler - som beskriver brugen af disse enheder, og som er i stand



Figur 2.11: Forløb over afbrydelses-håndtering.

til at kommunikere med enhederne gennem en dertil indrettet I/O-bus. Operativsystemet definerer så en standard grænseflade som en brugerproces kan bruge til at vekselvirke med enhederne gennem disse rutiner eller *drivprogrammer*. Se figur 2.12.



Figur 2.12: Vekselvirkning mellem I/O enheder og processer i et OS.

Et drivprogram kan styre en I/O enhed på to måder. Den første metode er gennem direkte referencer til et område i den fysiske adresserum, der er afbildet over på enhedens I/O adresserum. På denne måde kan processoren kommunikere med enheden gennem almindelige hent og gem instruktioner. Den anden metode er ved at sende information direkte til de porte, som er tilknyttet en enhed, gennem særlige I/O instruktioner. Gennem disse instruktioner er den givne enhed i stand til at modtage *Data Control Words* (DCW), hvilke er kommandoer som enheden forstår. DCW'er sætter enheden igang parallelt med kørslen af OS'et. Når enheden engang er færdig, skal OS'et informeres gennem en afbrydelse.

I/O Virtualisering

Da det bliver VMM'en, som kommer til at fange alle referencer til I/O adresserummet, vil det også være VMM'en, som bestemmer hvilke enheder en VMOS vil kunne 'se'. Hvad det så vil være, må afhænge både af valget af VMOS'er samt den tænkte brug af VMOS'erne. Som minimum må man dog forvente, at implementeringen af en VMM på en IBM kompatibel PC vil indebære emuleringen af følgende eksterne enheder:

- PIC (Programmable Interrupt Controller)
- DMA (Direct Memory Access) controller
- RTC (battery-backed Real Time Clock)
- PIT (Programmable Interval Timer)
- Video adaptor (minimum tekstmode)
- Keyboard controller
- Floppy eller IDE controller

En fuldstændig virtualisering af en IBM kompatibel PC vil kræve understøttelse af over 20 enheder [2].

For at emulere en ekstern enhed skal følgende gælde:

1. VMM'en skal have kendskab til alle de kommandoer og registre som hver emuleret enhed forstår.
2. Baseret på dette skal VMM'en låse for adgangen til enheden fra andre VMOS'er, indtil den aktuelle er færdig.
3. Hvis en enhed gør brug af en hukommelsesbuffer (gennem f.eks. DMA), skal hvert VMOS skal have sin egen adskilt buffer. Det er for at sikre at VMOS'erne holdes adskilt under multipleksingen af en enhed.

En virtualisering skal foregå ved, at når VMM'en modtager en afbrydelse fra VMOS'et pga. et I/O kald, skal den ud fra en tabel eller anden struktur bestemme, hvilken enhed der forsøges refereret til. Kaldet gives videre til det relevante modul, som udfører instruktionen (der ofte består af flere I/O kald) i et beskyttet miljø. På et tidspunkt vil enheden afgive en afbrydelse, som bliver fanget af VMM. Denne bliver sent videre til VMOS'et, der vil behandle den normalt.

Enheder, som er afbildet til hukommelsen, skal have de relevante sider markeret tomme, for at skabe en afbrydelse til VMM. Ligesom før vil VMM være nødt til at have en liste over hvilke adresser, der tilhører de forskellige enheder, for ordentligt at kunne behandle dem. Det vil også her være nødvendigt med seperate buffere, for at huske tilstanden mellem VMOS-skift (f.eks. vigtig mht. video adaptoren, som både skal kunne skifte mellem hvert VMOS samt VMOS'ernes egne konsoller).

2.2.7 Instruktionsbehandling

En VMM-kerne, som fungerer som et OS med fuld kontrol over den underliggende maskine, skal kunne være i stand til at skjule sin tilstedeværelse fra overliggende VMOS'er. Da disse VMOS'er eksekverer deres programkode direkte på maskinen, vil det være nødvendigt for

VMM'en at kunne fange de farlige maskin-instruktioner, som er i stand til at afsløre inkonsistenserne mellem VMOS og maskine. De fleste moderne arkitekturer er i stand til at tildele en proces et beskyttelsesniveau, som fremtvinger en afbrydelse til en kerne, når såkaldte følsomme instruktioner forsøges udført. Ved at udnytte denne funktionalitet og eksekvere VMOS'erne med et højt beskyttelsesniveau kan man fange de følsomme instruktioner. Disse følsomme instruktioner skal så emuleres, ved at indsætte værdier i instruktionens operander som VMOS'et forventer.

Ubeskyttede instruktioner

Hvis arkitekturen man har valgt at implementere en multipleksing VMM på, ikke har været tiltænkt en sådan funktion, kan man risikere, at der kan eksistere følsomme instruktioner, som ikke er omfattet af beskyttelsessystemet. Hvis der ikke bliver genereret afbrydelser under afviklingen af en følsom instruktion, kan VMM'en ikke komme til at simulere instruktionerne, hvorefter hele virtualiseringen bliver meget usikker. Det vil altså være nødvendigt at gennemkigge maskin-koden til hvert VMOS for at lokalisere de problematiske instruktioner. Uheldivis vil dette også medføre, at udførelseshastigheden vil blive væsentligt forringet.

Der findes dog en mulig løsning på problemet, der kombinerer kode-gennemsyningen med direkte eksekvering. Metoden kaldes *Scan Before Execute* (SBE) [10, 13]. Kort og godt går metoden ud på, at gennemkigge VMOS'ets maskinkode inden den bliver kørt. Hver gang en følsom ubeskyttet instruktion bliver fundet indsættes en *debug breakpoint*. Denne skaber en afbrydelse til VMM'en, der så kan simulere instruktionen. Handlingsforløbet beskrives forneden.

---

1. Notér adressen hvor skanningen begynder.
2. VMM kigger VMOS kode igennem inden eksekvering.
3. Hvis en instruktion er følsom og ubeskyttet.
   (a) Overskriv instruktionens opkode med debug breakpoint.
   (b) Indsæt i tabel adressen på instruktion samt dens opcode-værdi.
   (c) Fortsæt med pkt 2.
4. Hvis en instruktion er en spring-instruktion.
   (a) Hvis spring-instruktionen er statisk
      i. Spring til destinationsadressen, og notér adressen.
      ii. Hvis destinationen allerede er undersøgt påbegynd eksekveringen og notér slut-adressen.
      iii. Ellers fortset med pkt 2
   (b) Hvis spring-instruktionen er dynamisk
      i. Overskriv instruktionens opkode med debug breakpoint.
      ii. Indsæt i tabel adressen på instruktion samt dens opcode-værdi.
      iii. Begynd eksekvering. Notér slut-adresse.
5. Hvis kode allerede er undersøgt, så begynd eksekvering.

Til at begynde med vil der blive skabt mange afbrydelser, hvilket vil reducere afviklingshastigheden betydligt. Efterhånden som koden bliver udført, vil de mange dynamiske spring instruktioner kunne markeres som værende sikre, og dermed ekvekveres direkte. Dette kan lade sig gøre, da den oversatte kode bliver delt op i blokke. Hermed vil VMM'en - ved at analysere spring-instruktionens springområde - kunne bestemme om den kun kan nå sikker kode. Alternativt kunne hele koden oversættes på én gang, men det ville medføre, at VMOS'et skulle vente et stykke tid, inden den kunne påbegynde afviklingen.

2.2.8 16 bit problemet

En moderne datamat har normalt en ren 32-bits eller endog 64-bits instruktionsstørrelse, hvilket vil sige 32-bits eller 64-bits registre og adresser osv. På en sådan maskine vil alle instruktioner, som bliver eksekveret, have den samme størrelse fra opstart til afslutning. Der findes dog maskiner som af hensyn til bagudkompatibilitet er i stand til at skifte mellem instruktionsstørrelser. Den mest kendte af disse arkitekturer er Intel Corporations IA-32 arkitektur [5, 6, 7], der er i stand til at køre både 16- og 32-bit instruktionsstørrelser. For sådanne arkitekturer vil det normalt gælde, at den større (og som regel nyere) instruktionstilstand vil have forbedret hukommelsesstyring og beskyttelse, hvilket gør det muligt at køre et moderne operativsystem eller VMM.

Hvad angår Intel og lignende arkitekturer, så befinder de sig efter initialisering i et instruktionstilstand *instr1*. For at skifte til den anden tilstand *instr2* kræves det, at det program, som bliver kørt på maskinen, gennemfører en procedure for at få maskinen til at skifte over. For operativsystemer betyder det, at instruktionsstørrelsen under opstartssekvensen vil være anderledes end under udførelsen af den egentlig kerne. På et tidspunkt kommer der altså et skift, hvilket betyder at en korrekt fungerende VMM på en sådan arkitektur vil være nødt til at kunne håndtere begge instruktionsstørrelser. Problemet her er, at en VMM vil være nødt til at køre i den instruktionstilstand, der har de bedste beskyttelsesmuligheder for at kunne udnytte den forbedret beskyttelsesfunktionalitet. Der er to måder det vil kunne gøres på.

1. **Ren instruktionsoversætning**. *Instr1* tilstanden oversættes instruktion for instruktion indtil VMOS'et forsøger at skifte over til *instr2*, hvorefter programkørsel forløber som normalt. Det vil her kun være nødvendigt, at oversætte opstartssekvensen da et styresystem sjældent skifter tilbage igen, når det først en gang har skiftet instruktionsstørrelse. Denne metode vil være den eneste løsning for arkitekturer, der ikke tilbyder en mulighed for at køre den ene instruktionssæt oven på det andet. Ren instruktionsoversættelse er en meget langsommelig måde at virtualisere en maskine på, men da det kun vil være nødvendigt under opstartssekvensen, der jo kun bliver kørt én gang, vil det være til at overse.
2. **Almindelig kørsel af gæste-kode**. Hvis den valgte arkitektur tilbyder mulighed for at køre instruktionssæt *instr1* oven på *instr2*, enten ved at sætte en tilstandsbit eller ved hjælp af en indbygget virtualiseringsfunktionalitet, kan instruktionerne eksekveres direkte på maskinen. Fordi man stadig befinder sig i tilstand *instr2* har man mulighed for at udnytte de forbedrede beskyttelsesmuligher, og da eksekveringen foregår direkte på maskinen, kan der drages fordel af en betydelig forøget afviklingshastighed.
   Når VMOS'et på et tidspunkt forsøger at skifte over til *instr2*, vil der ske en afbrydelse til VMM'en, da VMOS'et jo kører med et højt sikkerhedsniveau. VMM'en skal nu sørge for at maskinen skifter til at køre ren *instr2*. Herefter afbrydes der tilbage til VMOS'et, der forsætter afviklingen i *instr2*.

Kapitel 3

Faciliteter til rådighed i Intel IA-32 arkitekturen

I kapitel to blev der opstillet en række krav som en maskinarkitektur skal overholde for, at den vil være i stand til at kunne virtualiseres. I kapitel 3 blev de følsomme maskin-mæssige faciliteter i en korrekt fungerende VMM belyst. Begge dele blev gjort så vidt muligt arkitektur-uafhængigt. I dette kapitel vil de emner, som blev berørt i de to nævnte kapitler, beskrives konkret med henblik på Intel Corporations IA-32. En fuldstændig beskrivelse kan findes i [5, 6, 7].

3.1 Følsomme registre

Under gennemgangen i dette kapitel vil der forekomme referencer til flere forskellige registre. Disse registre tilhører IA-32 eksekveringsmiljø og vil skulle virtualiseres ved at give hver VM en skygge-version. Registrene det drejer sig om listes forneden.

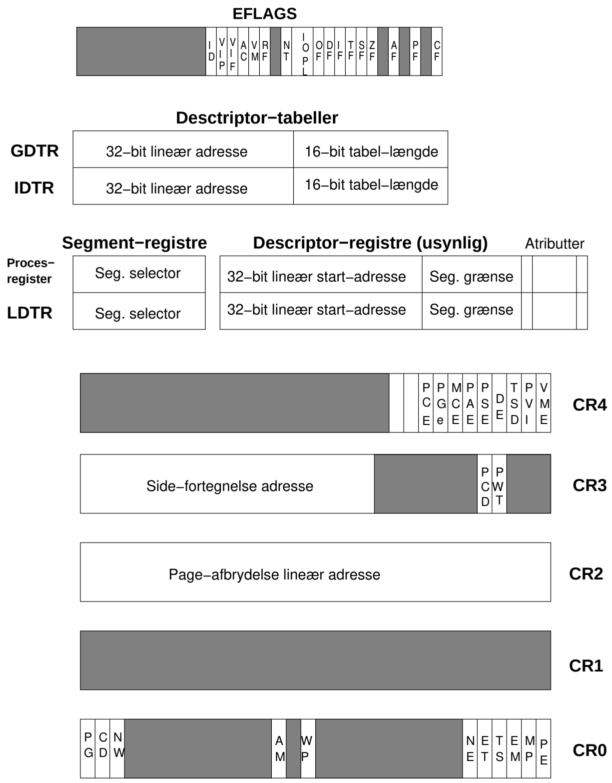
- Kontrol registre: CR0, CR1, CR2, CR3, CR4, CR5
- Tabel registre: IDTR, GDTR, LDTR
- Flag register: EFLAGS
- Segment registre: CS, SS, DS, ES, FS, GS
- Generelle registre: EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP
- Instruktionspeger register: EIP
- Proces register: TR

Af disse registre vil det kun være de generelle registre, som ikke behøver en skygge udgave. De vil stadigvæk være nødvendig at gemme under hvert kontekstskift. En figur over de vigigste kan ses i 3.1. For en nærmere forklaring af de forskellige flag henvises der til Intels manualer.

3.2 Processor tilstande

Intels IA-32 arkitektur har fire forskellige tilstande:

1. *Real mode*



Figur 3.1: Vigtige registre i IA-32.

2. *Protected mode*
3. *Virtual-8086 mode*
4. *System management mode*

Den første tilstand - real mode - er til for at sikre bagudkompatibilitet med ældre programmer. Instruktionsstørrelsen er 16-bit og der findes ingen beskyttelsesfunktionalitet overhoved. Protected mode er en 32-bit tilstand og tilbyder moderne beskyttelsesmekanismer. Denne tilstand er den 'officielle' tilstand. Virtual-8086 er en særlig tilstand, som tillader OS'er at eksekvere gamle programmer skrevet til 8086 CPU'en. System management mode bruges til f.eks at køre specielle programmer, undersøge energi faciliteter m.m. De to sidste tilstande behandles ikke i denne rapport.

3.3 Real mode

IA-32 arkitekturen er en todelt arkitektur, som giver den muligheden for at kunne afvikle ældre programmer. Disse programmer var egentlig udviklet til ældre 16-bit Intel processorer, men pga. deres meget store udbredelse, valgte man at beholde muligheden for at kunne køre disse programmer, da Intel med 80386 processoren skiftede til IA-32. *Real mode* eller IA-16 som denne tilstand kaldes er den tilstand som IA-32 arkitekturen starter op i efter initialisering. I denne tilstand mangler alle former for beskyttelse, da real mode kun var tiltænkt én-proces operativsystemer som MS-DOS fra Microsoft, dvs. et MS-DOS program har fuld adgang til alle I/O enheder og hukommelsesområder.

3.3.1 Kørsel af 16-bit kode

For på korrekt vis at virtualisere IA-32 arkitekturen vil det være nødvendigt, at være istand til at afvikle real mode programkode, og samtidig beholde de beskyttelsesfaciliteter, der findes i *protected mode*, som er 32-bit tilstanden. Der gælder det, at opcode-værdien for de instruktioner, som findes i real mode, er det samme som i protected mode (se næste sektion). Dette indebærer, at for at afvikle 16-bit instruktioner mens processoren befinder sig i 32-bit tilstanden, kan man gøre to ting:

1. Tilføje et præfiks foran instruktionens opkode-værdi samt evt. foran en adresse i dens operand(er). Denne metode er fin, hvis der blot skal eksekveres enkelte 16-bit instruktioner, men ikke hvis der skal afvikles et helt program.
2. Type-betegnelsen for det segment, som programkoden befinder sig i, indeholder et flag `D`, der beskriver om segmentet er 16 eller 32-bit. Hvis `D` er nulstillet er segmentet sat som værende 16-bit og al koden bliver afviklet derefter. På samme måde kan en 16-bit stak defineres ved at nulstille `B`-flaget i type-betegnelsen på et data segment. Det vil dog kun være de første 64kb af data-segmentet som vil kunne refereres af et 16-bit program.

Under kørslen af 16-bit kode segmentet vil der ske afbrydelser til VMM'en enten som led i den generelle VM-schedulering eller gennem udførelsen af beskyttede handlinger. Der kommer altså til at eksistere skift mellem 16 og 32-bit kode-segmenter. Skiftet til VMM'en vil ske gennem en interrupt-gate eller en trap-gate (beskrevet i et senere afsnit), og her vil `D`-flaget være sat (32-bit). Skiftet tilbage til 16-bit segmentet foretages gennem en `IRET` instruktion, og her vil det være segmentets `D`-flag, som bestemte.

3.3.2 Tilstandsskift

På et tidspunkt skal operativsystemet til at skifte til 32-bit tilstanden. IA-32 arkitekturen forventer en række handlinger som hver skal behandles [7].

1. Sluk for afbrydelser vha. `CLI` instrukionen. `CLI` er en følsom beskyttet instruktion og der afbrydes derfor til VMM'en. Denne behandler `CLI` på normal virtualiseret vis.
2. Adressen til `GDT`-strukturen lægges i `GDTR` via `LGDT` instruktionen, der er en beskyttet funktion. Der afbrydes til VMM, som emulerer instruktionen.
3. Udføre en `MOV CR0` instruktion, som sætter `PE`-flaget (Protected mode Enabled) samt evt. `PG`-flaget (Paging Enabled). `MOV CR0` er ligeledes en beskyttet handling, så der afbrydes til VMM. VMM'en emulerer instruktionen og, da VMOS'et nu tror, at den er i protected mode, sætter VMM'en `D`-flaget i VMOS'et kode-segment samt `B`-flaget i dens data-segment.
4. Straks efter at `PE` er blevet sat, skal der springes til næste linie via en lang `JMP` eller `CALL` instruktion. Da der er begge tale om lange kald til samme segment og dermed samme beskyttelsesniveau, kan disse instruktioner få lov til at blive eksekveret uden forstyrrelse. Instruktionerne skifter maskinens eksekveringsretning (flow of execution) og henter segmentet ind i `CS`-registret. På en ikke-virtualiseret maskine med paging skal `MOV CR0` og `JMP` eller `CALL` instruktionerne kaldes fra en identitetsafbildet side (den virtuelle og fysiske adresse er ens). Da VMM'en alligevel både har paging og protected mode slået til, kan der ses bort fra dette krav.

5. Hvis operativsystemet bruger en `LDT`-tabel skal instruktionen `LLDT` eksekveres. Der afbrydes og instruktionen emuleres.
6. `LTR` instruktionen udføres for at gemme task registret i `TR`. Der afbrydes og VMM behandler instruktionen.
7. Segment registrene `DS`, `SS`, `ES`, `FS` og `GS` skal opdateres enten gennem normal vis (`MOV`, `POP` osv.) eller via et spring eller kald til en ny task.
8. Gem `IDT` i `IDTR` vha. `LIDT`. Instruktionen fanges og behandles i VMM'en.
9. Genstarte afbrydelser med `STI`. Denne fanges og behandles i VMM'en.

Operativsystemet vil herefter normalt ikke vende tilbage til real mode.

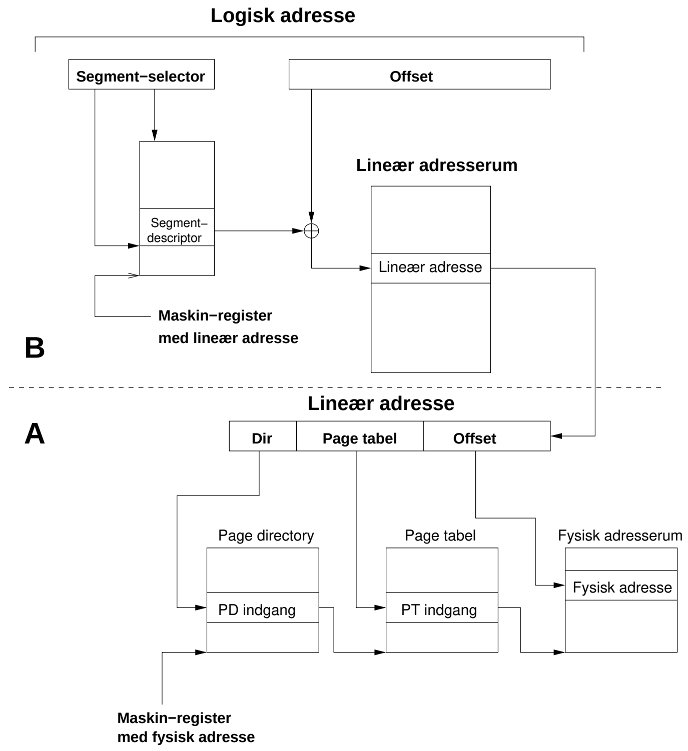
3.4 Paging

Paging funktionaliteten i IA-32 fungerer stort set som forklaret i kapitel 3. Der er dog visse konkrete forskelle. Den første er, at siderne, som hukommelsen er delt op i, kan være på enten 4kb eller 4Mb. 2Mb sider er også muligt, hvis man udnytter IA-32 udvidet 36-bit adresseringsrum, men 4kb er den mest almindelige og er også den størrelse VMM'en bruger. Den anden forskel er, at man med 4kb sider ikke har en enkel side-tabel, men derimod en to-lags tabel-struktur. Det første lag består af én tabel kaldet en *page directory* eller side-fortegnelse, hvis fysiske adresse er gemt i `CR3`. Den har 1024 indgange, som peger på hver sin *page tabel* eller side-tabel. Disse side-tabeller har ligeledes 1024 indgange, som indeholder adresserne til begyndelsen af hver side i den fysiske adresserum. Maskinen modtager en 32-bit lineær adresse fra segment mekanismen, som skal afbildes mod det fysiske adresserum. Dette gøres ved, at dele adressen op i tre dele, hvor de 10 mest betydende bit angiver indgangen til side-fortegnelsen. De næste 10 bit angiver indgangen til side-tabellen, mens de sidste 12 angiver indekset til den fundne side. Ovenstående illustreres i figur 3.2(A) [7], der er en gengivelse fra kapitel 2.

Både side-fortegnelsen og side-tabellerne skal placeres sådan i hukommelsen at de stilles op ved side grænserne, hvilket bevirker, at de 12 nederste bit (22 for 4Mb sider) i deres start-adresse er lig nul. Dette bliver udnyttet til at gemme informationer i de pladser (se figur 3.3.

Særligt *accessed* og *dirty* flagene kræver særlig behandling. Accessed flaget angiver, at en side eller side-fortegnelsesindgang har været udsat for en handling, mens dirty flaget viser, at en side er blevet ændret siden den bliv hentet/oprettet. Disse flag bliver sat automatisk af Intel CPU'en, men ikke nulstillet igen. Da et operativsystem bruger flagene til at vide hvilken information, der skal kopieres ned på den eksterne lagerplads, er det vigtigt at disse flag holdes synkroniseret mellem skygge-tabellerne og VMOS'et egne tabeller. Et VMOS vil konsultere sine egne tabeller for at bestemme deres status, så det er nærliggende at stille beskyttelsesniveauet for de sider, hvor gæste-tabellerne findes, til system-niveau og ikke bare skrivebeskyttet som foreslået i analyse sektionen. Dette vil skabe en afbrydelse til VMM'en som vil kunne holde flagene ajour.

Som nævnt i problemanalysen vil der opstå konflikter mellem paging systemerne i VMM'en og VMOS'erne. Løsningen på dette vil være at bruge skygge-paging (se 2.7). Helt konkret vil metoden være følgende på en IA-32 maskine (der antages her, at VMM'en har gemt adressen til VMOS'ets egne page-tabeller, samt har markeret siderne de befinder sig i som værende systemniveau sider):



Figur 3.2: Paging mekanismen i IA-32.

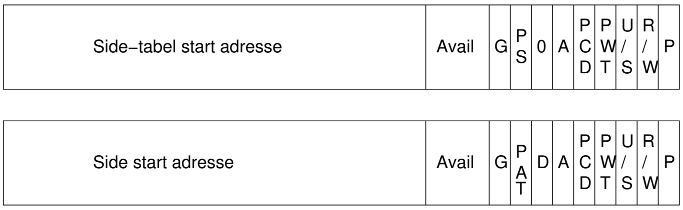
---

1. En side fejl (*page fault*) modtages.
   (a) Hvis fejlen skyldes, at VMOS'et har forsøgt at initialisere eller ændre en indgang i VMOS'ets egen side-fortegnelse eller en af dets side-tabeller:
      - Den nye descriptor fra VMOS'et hentes.
      - Indsættes i den rigtige plads i VMOS'ets tabel-struktur.
      - Kopieres over til skygge-indgangen, hvor beskyttelsesniveauet ændres og accessed og evt. dirty flaget sættes.
      - Der springes over fejl-instruktionen, hvorefter der returneres tilbage til VMOS'et, som vil forsætte.

   (b) Hvis fejlen skyldes, at en indgang i skygge side-fortegnelsen eller en af skygge side-tabellerne er nulstillet:
      - VMOS'ets egen tilsvarende indgang undersøges.
      - Hvis den er også er nul, springes der til VMOS'ets fejl-handler.
        – Denne opretter en ny tabel og forsøger at gemme dets adresse i indgangen.
        – Der fejles, og VMM'en behandler den nye fejl.
      - Hvis den ikke er nul, kopieres indgangen over til skygge-indgangen. Beskyttelsesniveauet ændres og accessed og evt. dirty flaget sættes. Hvis skygge-indgangen er en side-tabel ændres adressen til siden således, at den peger på den faktiske fysiske adresse.

   (c) Fejlen er en ordinær page fejl.
      - Der springes til VMOS'ets fejl-handler.

Hvis et VMOS har separate paging-tabeller for hver proces, som bliver kørt, vil VMM'en ligeledes være nødt til at skifte skygge-tabellerne ud, hver gang en ny adresse forsøges gemt



Avail	**Availiable for system programmer**
G	**Global page**
PAT	**Page Tabel Attribute Index**
PS	**Page size**
D	**Dirty**
A	**Accessed**
PDC	**Cache Disabled**
PWT	**Write−Through**
U/S	**User/Supervisor**
R/W	**Read/Write**
P	**Present**

Figur 3.3: Tabel-indgange.

i `CR3`. Det kan gøres på to måder, enten ved helt at kassere de gamle skygge-tabeller, eller ved at gemme dem i hukommelsen. Den første metode er den nemmeste men indebærer også, at tabellerne begynder fra nul, hver gang VMOS'et skifter mellem processer. Dette vil indebære mange afbrydelser og stærkt reducere udførelseshastigheden. Det vil den anden metode forbedre meget, da VMM'en blot skifter til den skygge-tabel, som er tilknyttet den aktuelle proces. Da indgangene beholdes, vil eksekveringen af gæste-processen forløbe mere flydende.

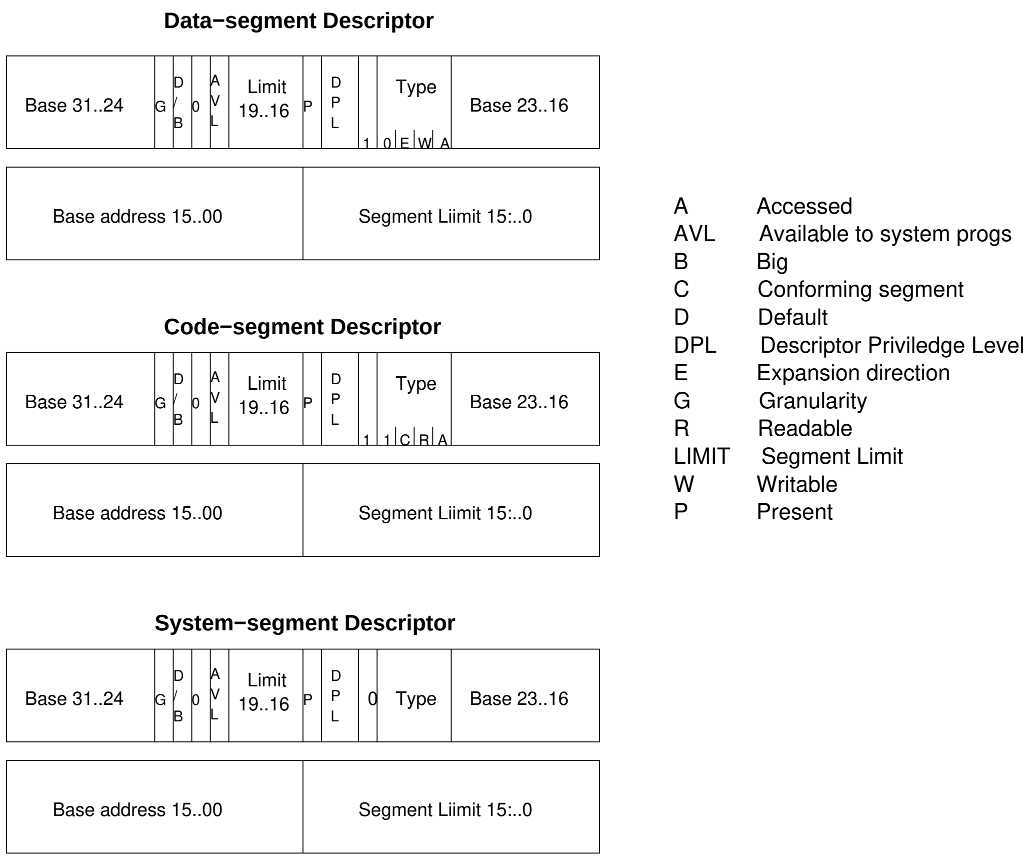
3.4.1 Detaljer vedrørende paging

Da gentagne referencer til lineære adresser i paging systemet kræver en del tid, bliver de sidste/oftest brugte lineære adresser gemt i noget midlertidig lagerplads ved navn *Translation Lookaside Buffers* (TLB). At adresserne er gemt heri vil medføre, at hver gang en page-tabel indgang bliver ændret (evt pga VMOS) skal det pågældende midlertidige lager slettes. Dette gøres ved hjælp af `INVLPG`, som er en beskyttet funktion. For at slette hele bufferen skal page-tabel adressen i `CR3` overskrives, hvilket betyder, at ved hver ny skygge-tabel bliver TLB automatisk slettet. En korrekt implemetering af skygge-tabeller skal derfor huske at tømme TLB for hver gang der sker en ændring i en af indgangene.

3.5 Segmentering

Segmentering i IA-32 er baseret på descriptorer, der udgør indgangene i descriptor-tabellerne `GDT` og `LDT`. Adresserne til disse tabeller gemmes i henholdsvis registrene `GDTR` og `LDTR` vha. instruktionerne `LGDT` og `LLDT`. Maskinen holder styr på de forskellige descriptorer i segment-tabellerne vha. 16-bit selectorer. Det er disse, som bliver gemt i segment-registrerne `CS`, `SS` osv. Et overblik kan dannes vha. figur 3.4.

Virtualiseringsprocessen beskrives i forneden.



Figur 3.4: Illustration over descriptor-tabellerne.

---

1. Til at begynde med har VMM'en opstillet sin egen GDT (VMM_GDT), hvor VMM descriptorerne er gemt helt ned i enden af tabellen og de andre er nulstillet. VMM'en vil her allerede have allokeret et segment til hvert af VMOS'erne (G_DESC). Disse bruges til at sikre, at VMOS'erne ikke forsøger at referere ud over sit allokeret område.
2. Når VMOS'et forsøger at gemme sin GDT i GDTR afbrydes der til kernen, og adressen gemmes (GOS_GDT). Da størrelsen af GOS_GDT er kendt (se figur 3.4) kan de sider GOS_GDT befinder sig i skrivebeskyttes.
3. De descriptorer, der er initialiseret kopieres over i deres respektive indgange i VMM_GDT. Beskyttelsesniveauerne ændres til højeste niveau (DPL=3) og start værdien sættes til G_DESC.
4. Eksekveringen af VMOS'et kan nu forløbe uden afbrydelser, indtil der forsøges tilføjet noget i GOS_GDT.
   (a) Der afbrydes til VMM.
   (b) Hvis en VMM descriptor i VMM_GDT forsøges overskrevet, fejles der.
   (c) Ellers opdateres både VMM_GDT og GOS_GDT.
   (d) Eventuelle flag-uregelmæssigheder korrigeres.
   (e) Der springes over instruktionen og returneres.

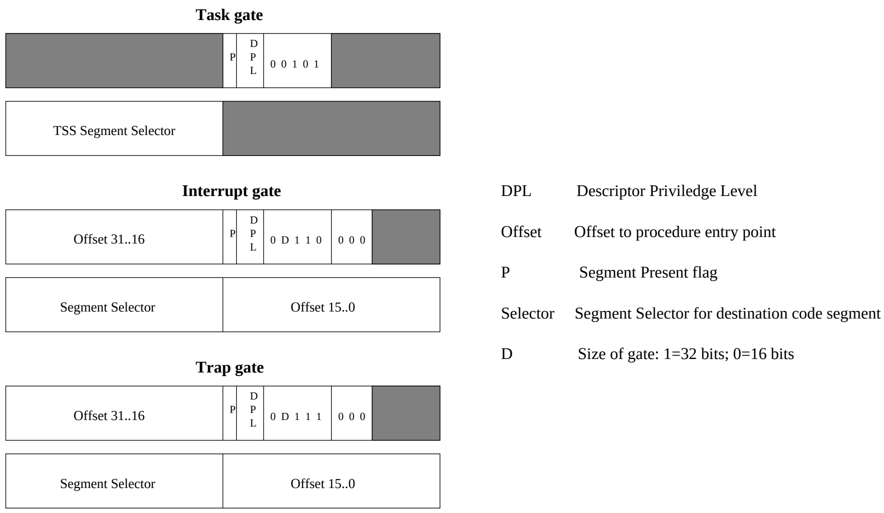
Ligesom med paging har segmentering et accessed flag (`A`) og et present flag (`P`). Disse kan bruges til at implementere en form for paging bare med segmenter. Dette giver dog kun mening, hvis en paging mekanisme ikke findes. Ellers holdes `A`-flaget synkroniseret på samme måde som i paging.

3.6 Interrupts

Interrupts eller afbrydelser på IA-32 er baseret på en descriptor tabel ligesom segmenteringssystemet. Tabellen hedder IDT og kan indeholde følgende typer descriptorer:

- *Task gate*
- *Interrupt gate*
- *Trap gate*

Descriptorerne kan ses på figur 3.5.



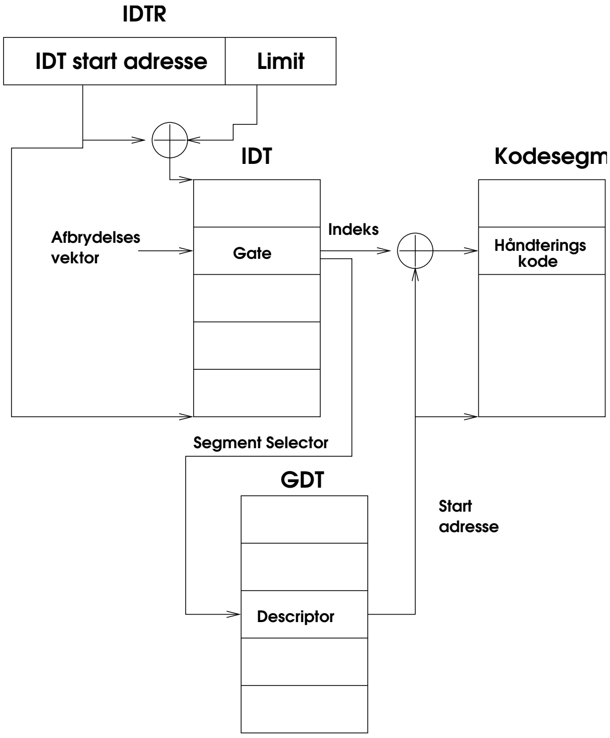
Figur 3.5: Illustration over descriptor-tabellerne.

En task gate tillader proces-skift at ske automatisk gennem en interrupt eller en programafbrydelse (*software interrupt*). En task gate indeholder en selector til en TSS descriptor i GDT'en som der skal skiftes til, hvis der sker en afbrydelse.

En interrupt gate og en trap opfører sig stort set ens. Maskinen læser selectoren og indeksværdien gemt i gaten og springer dertil. En interrupt gate sætter samtidig `IF` flaget i `EFLAGS` registret, mens en trap gate ikke gør det.

Afbrydelser fungerer ved, at hver afbrydelse er tildelt en vektor, som angiver hvilken descriptor-indgang i IDT'en, der skal læses fra. Denne descriptor indeholder en selector i GDT'en, som sammen med indeksen i descriptoren angiver, hvor koden, der skal håndtere afbrydelsen findes. Se figur 3.6.

Intel IA-32 arkitekturen tillader i alt 256 (0-255) forskellige afbrydelser. Det giver derfor ingen mening, at have en større IDT. Af disse 256 afbrydelser er de første 32 (0-31) reserveret af Intel, mens resten fra 32 og opefter er til fri afbenyttelse. I skrivende stund er der defineret 19 *exceptions* (undtagelser), der netop optager de første 19 pladser. På disse pladser befinder CPU'ens fejl, fælder og aborter, mens de eksterne afbrydelser fra I/O enhederne skal placeres i det frie område. De eksterne afbrydelser kaldes IRQ'er (Interrupt ReQuest), og for dem gælder det, at IRQ n som standard bliver programmeret



Figur 3.6: Illustration over afbrydelses-håndtering.

til at ligge på plads `n + 32` [9]. At afbrydelserne har fast definerede pladser er en hjælp til at virtualisere IDT-tabellen i VMM-kernen. Metoden er baseret på forslaget fra analyse sektionen og forklares forneden.

---

1. VMM'en opretter sin afbrydelses-tabel og programmerer interrupt kontrol-chippen. Hver descriptor i IDT'en peger på et kode stykke, der gemmer afbrydelses-vektoren inden der springes videre til den egentlige håndteringskode.
2. VMOS'et forsøger at gøre det samme, men der afbrydes til VMM, som gemmer adressen til gæste-IDT'et (GOS_IDT) samt pladserne for IRQ'erne i en afbildningstabel, hvis de er anderledes end forventet.
3. Der kommer nu en afbrydelse og vektoren gemmes.
   (a) Hvis afbrydelsen er en fejl, fælde eller abort, kigges der på GOS_IDT[vektor] for at finde den rigtige sted i VMOS'et springe til.
   (b) Hvis afbrydelsen er en IRQ, sammenlignes der først med afbildningstabellen inden der springes til GOS_IDT[vektor].
   (c) Hvis afbrydelsen er et software interrupt, der springes der til GOS_IDT[vektor].
   (d) Ellers abort.
4. VMOS'ets håndteringskode bruger den tilstands information, som blev gemt inden der blev sprunget til VMOS'et, til at returnere tilbage til den proces, som var udsat for afbrydelsen.

3.7 Håndtering af I/O enheder

IA-32 arkitekturen tillader brugen af I/O på to måder. (1) Gennem kald til et seperat I/O adresserum, eller (2) gennem brug af hukommelsesafbildning. I/O adresserummet har en længde på 16-bit, dvs. der findes 64K 8-bit I/O porte fra 0h til FFFFh. Der gælder det, at to på hinanden følgende 8-bit porte kan behandles som én 16-bit port, mens fire kan behandles som én 32-bit port. For at referere disse porte stiller IA-32 fire instruktioner til rådighed.

- `IN`
- `OUT`
- `INS`
- `OUTS`

`IN` og `OUT` flytter en enkel 8, 16 eller 32-bit dataværdi mellem portene og `EAX` registret. `INS` og `OUTS` bruges sammen en af gentagelsesinstruktionerne som `REP` til at flytte en data blok til og fra en I/O port.

For at virtualisere I/O portene kræver det, at VMM'en er i stand til at fange instruktionerne når de bliver kaldt. Heldigvis tilbyder IA-32 at sætte portenes rettighedsniveau. Dette gøres i `EFLAGS`-registrets IOPL indgang. Hvis en proces forsøger at udføre en I/O instruktion, sammenligner CPU'en CPL med IOPL. Hvis CPL er numerisk højere end IOPL (CPL > IOPL), afbrydes der til VMM. For komplethedens skyld skal det nævnes, at det er muligt for en proces med en CPL > IOPL at referere en I/O port. Dette kan lade sig gøre gennem I/O bit mappen, der ligger øverst i TSS-strukturen. Her gælder der, at hvis CPL > IOPL og den pågældende bit til den kaldte I/O port *ikke* er sat, vil handlingen kunne udføres alligevel.

3.8 Time Stamp Counter

IA-32 arkitekturens Time Stamp Counter (TSC) består af en 64-bit register, som kan læses gennem instruktionen `RDTSC`. Denne instruktion kan indstilles til at være en priviligeret handling, ved at stille `TSD`-flaget i kontrolregister `CR4`. Dette vil være nødvendigt for at sikre, at en VMOS' brugerkode ikke vil være i stand til at eksekvere instruktionen, hvis VMOS'et forventer at `CR4`[`TSD`] er nulstillet. VMM'bliver altså nødt til at undersøge VMOS'ets egen kopi af `CR4`, samt det virtualiseret rettighedsniveau af VMOS-koden.

Som nævnt i analyse-sektionen er de resterende timer-faciliteter externe enheder. Emuleringen af dem er hardware afhængig og håndteres gennem I/O systemet.

3.9 Følsomme beskyttede instruktioner

IA-32 indeholder 16 beskyttede instruktioner, som alle fremtvinger en afbrydelse, hvis de bliver kaldt fra en proces, hvis CPL>0.

Se appendiks for en liste over instruktionerne med opcode.

LGDT, LLDT, LIDT, LTR

Instruktionerne her bruges til at gemme den lineære adresse til de forskellige descriptor-tabeller samt TSS-tabellen i deres respektive maskin-registre. VMM'en skal altså blot gemme adressen i operanden i en variabel, der tilknyttet den kørende VMOS.

MOV (kontrol registre)

Disse instruktioner læser og skriver indholdet af kontrol-registrene. VMM'en skal gemme værdierne i de virtualiserede registrer for at skjule registrenes faktiske indhold.

LMSW

Load Machine Status Word. Flytter en 16-bit operand over i de 16 første bit af `CR0`. Eksisterer kun for bagudkompatibilitet med Intel 80286. Alle nyere maskiner bruger `MOV`. Behandles af VMM som med `MOV`.

CLTS

Clear Task-Switched Flag in CR0. Nulstiller `TS`-flaget i `CR0`. VMM'en skal blot sørge for at opdatere den virtuelle `CR0`.

MOV (debug registre)

Læser og skriver til og fra debug registrene. Håndteres som de andre `MOV` instruktioner.

INVD

Invalidate Internal Caches (uden writeback). Sletter de midlertidige hukommelsesområder *uden* at kopiere dem til hovedlageret først. Tvivlsom om den bliver brugt af de fleste styresystemer, en virtualisering kunne evt udføre `WBINVD` i stedet.

WBINVD

Write Back and Invalidate Cache. Samme som `INVD` her bliver data lageret i cachen blot kopieret tilbage til hovedhukommelsen først.

INVLPG

Invalidate TLB Entry. Tømmer TLB indgangen til adressen gemt i operanden. VMM skal konvertere denne adresse til den rigtige i skygge-tabellen inden den eksekverer instruktionen.

HLT

Halt. Standser CPU'en. VMM'en skal blot fjerne det aktuelle VMOS fra scheduleringskøen, og først starte det igen, hvis det modtager en afbrydelse.

RDMSR

Read from Model-Specific Register. Læser fra den 64-bit model-specifikt register, som er angivet i `ECX` og gemmer den i `EDX:EAX`. Sjældent brugt og vil nok godt kunne undlades at blive implementeret, de fleste operativsystemer også er designet til at kunne køre på ældre 32-bit Intel processorer.

WRMSR

Write to Model-Specific Register. Modsat af `RDMSR` og behandles på samme måde.

RDPMC

Read Performance Monitoring Counter. Læser 40-bit registeret ind i `EDX:EAX`. En nyere arkitekturbaseret udvidelse som ikke behøver blive understøttet af VMM'en i første omgang.

RDTSC

Read Time Stamp Counter. Læser 64-bit TSC registeret ind i `EDX:EAX`. VMM'en skal blot sikre sig om VMOS'ets kopi af `CR4` registret tillader brugerniveau-kørsel af instruktionen, ellers skal den ikke emuleres.

3.10 Følsomme ikke-beskyttede instruktioner

Det største problem ved IA-32 arkitekturen er, at den ikke fra starten har været beregnet på at blive multiplekset. Dette giver udslag i, at der eksisterer en del ubeskyttede instruktioner, som er i stand til at læse den underliggende maskines tilstand. Et VMOS, som eksekverer en af disse instruktioner vil kunne risikere at få data tilbage som den ikke forventer, da VMM'en ingen naturlig muligheder har for at fange instruktionerne. For at bestemme hvilke instruktioner der er tale om, skal IA-32 instruktionsættet analyseres. Resultatet er at der findes 14 følsomme, ubeskyttede instruktioner som bryder kravene 3B og 3C [11].

3.10.1 Følsomme register instruktioner

SIDT, SGDT og SLDT

Disse instruktioner læser værdierne gemt i registrene `IDTR`, `GDTR` og `LDTR`. `IDTR` og `GDTR` gemmer en 6-byte værdi et et lagerområde, mens `LDTR` gemmer en 16-bit selector i enten en register eller lagerområde. En simulering af disse instruktioner vil kræve, at en VMM har sine egne skygge-registre.

SMSW

Store Machine Status Word. Henter de første 16 bit fra `CR0` og gemmer dem i enten en register eller lagerområde. `SMSW` er til for at sikre bagudkompatibilitet med Intel 80286. Nyere processorer fra 80386 og opefter forventes at bruge `MOV` instruktionerne. `SMSW` kan derfor med rimelig sandsynlighed ignoreres. Ellers skal VMM'en blot sørge for, at hvert VMOS har en skygge-`CR0`.

PUSHF og POPF

Pusher eller popper `EFLAGS` registret til og fra stakken. En push bevirker, at den rigtige `EFLAGS` kan aflæses, men en pop betyder, at den kan ændres. Dette er farligt, da `EFLAGS` indeholder en mængde information vedrørende maskinens tilstand. En emulering kræver, at hvert VMOS har en skygge-`EFLAGS` register.

3.10.2 Referencer til beskyttelsessystemet

LAR, LSL, VERR, VERW

Load Access Rights Byte, Load Segment Limit, Verify a Segment for Reading or Writing. `LAR` henter beskyttelsesniveauet fra en descriptor angivet i den anden operand og sætter Z-flaget i `EFLAGS`. `LSL` henter segment størrelsen fra descriptoren angivet i den anden operand og sætter ZF-flaget i `EFLAGS`. `VERR` og `VERW` undersøger om det segment givet i operanden er henholdsvis læsbar eller skrivbar mht den aktuelle beskyttelsesniveau (CPL). Sætter ZF-flaget alt efter resultat.

Alle fire instruktioner udfører den samme sikkerhedskontrol; $(\text{CPL}>\text{DPL}) \vee (\text{RPL}>\text{DPL})$. Dette betyder at et VMOS, som kører med CPL=3 vil kun kunne bruge instruktionerne på segmenter med samme beskyttelsesniveau. En virtualisering vil være nødvendig, da skygge-paging mekanismen afslører egenskaber ved descriptorerne som kunne være anderledes end de forventede.

POP

Hvis der forsøges at poppe til et segment-register (`CS` er ikke muligt), vil maskinen udføre en kontrol mellem RPL, CPL og DPL. Hvis enten RPL's eller CPL's privilegieniveau er lavere end DPL $(\text{RPL}>\text{DPL}) \wedge (\text{CPL}>\text{DPL})$, fejles der til kernen. En simulation vil indebære, at selectoren som forsøges gemt i registret skal have sin RPL-værdi ændret til RPL=3, *inden* VMM'en selv eksekverer instruktionen samt at alle DPL-værdierne ligeledes er sat til tre i skygge-tabellerne.

PUSH

Et problem kan opstå, hvis der forsøges at pushe en selector indeholdt i et segmentregister ned på stakken. Dette giver et VMOS mulighed for at undersøge værdierne. En korrekt simulering af `PUSH` indebærer, at VMM må ændre selectorens informationer til de af VMOS forventede værdier.

STR

Store Task Register. `STR` instruktionen gemmer segment-selectoren i `TR` i operanden. Da selectoren indeholder RPL-værdien for den aktuelle VMOS, vil et sådant kunne opdage, at RPL$\neq$0. En simulering af `STR` vil indebære, at RPL-værdien bliver ændret til det forventede inden den gemmes i operanden.

MOV (segment registre)

Som før omtalt indeholder segment registrene en selector, som angiver RPL-værdien. Et VMOS kan aflæse disse værdier ved at flytte en selector over til en anden register eller hukommelsesplads. En korrekt simulering af disse `MOV` instruktioner vil kræve, at selectoren får den forventede RPL inden den gemmes i operanden.

3.11 Diverse instruktioner

Udover de ovenover omtalte instruktioner, findes der også en række andre instruktioner, som der også skal tages stilling til. Disse er:

LDS, LES, LFS, LGS & LSS

Disse instruktioner modtager en adresse bestående af et selector:offset par som den ene operand. Denne adresse forsøges gemt i instruktionens segment-register samt destinationsregistret, der er det andet operand. Instruktionerne vil fejle, hvis deres RPL, CPL og DPL værdier ikke stemmer overens, men sålænge VMM'en sørger for, at descriptorerne i skyggetabellen stemmer overens med VMOS'ets, burde instruktionerne kunne eksekveres uden problemer.

CALL, JMP, INT n, RET og IRET

`CALL` instruktionen kan foretage fire slags spring:

- Et nært kald.
- Et langt kald til samme privilegieniveau.
- Et langt kald til forskelligt privilegieniveau.
- Proces-skift.

Et nært kald og et langt kald til samme privilegieniveau, vil ikke indebære problemer for et CPL=0 VMOS, der bliver kørt med CPL=3. De to sidste slags kald er til gengæld problemfyldte, da de indebærer en undersøgelse af RPL, CPL og DPL ved skift mellem segmenter eller ved skift til en ny proces. `JMP` fungerer på samme måde, men gemmer blot ikke returneringsinformation. Ligeledes med `INT n`, som også kan foretage privilegie-skift, når der afbrydes til håndteringsrutinerne. `RET` opererer med modsat effekt, da der her returneres til en retur-adresse placeret i stakken. Hvis der sker et segment-skift, bliver privilegieværdierne undersøgt, for se om det kan tillades. Desuden bliver segment-registrene `DS`, `ES`, `FS` og `GS` nulstillet, hvis CPL>DPL i hvert segment-register, da de ikke vil kunne refereres til af den nye CPL. `IRET` fungerer som `RET` bortset fra, at den returnerer fra en afbrydelse. Alt i alt vil alle disse spring-instruktioner kunne fungere på korrekt vis, sålænge privilegieniveauerne i skygge-tabellerne alle er afbildet til niveau tre for de descriptorer, der vil blive brugt af det pågældende VMOS.

CLI og STI

`CLI` og `STI` henholdsvis stopper og starter for genereringen af afbrydelser. Følgende rettighedskontrol foretages: CPL$\leq$IOPL. Da IOPL-flaget i `EFLAGS` vil være sat lig nul, er instruktionerne beskyttede og kan simuleres af VMM. Simuleringen gøres ved ikke at springe til VMOS'ets håndteringskode for afbrydelser, men istedet blot at lade eksekveringen af VMOS fortsætte fra det sted afbrydelsen skete.

IN, INS, OUT og OUTS

Disse instruktioner læser og skriver til og fra I/O portene som forklaret i sektionen om I/O enheder. CPU'en undersøger om CPL$\leq$IOPL er opfyldt inden en af instruktionerne udføres, hvis det ikke tilfældet undersøges I/O bitmappen i TSS-strukturen, og hvis den pågældene bit ikke er sat tillades handlingen. `EFLAGS` skal altså være nul for at beskytte instruktionerne mod udførelse. Simuleringen forgår som forklaret tidligere.

Jcc, LOOPx

Disse instruktioner har ingen sikkerhedsmæssige aspekter, da de kun kan bruges indenfor nære kald, men hvis VMOS-koden skal virtualiseres som forklaret i analyse-sektionen, skal der indsættes en software afbrydelse foran disse instruktioner.

INT3

En debugger facilitet. Instruktionens opkode fylder kun én byte og kan derfor indsættes i en instruktionsopkodes første byte. `INT3` genererer en adbrydelse #3, som kan opsnappes af VMM, der så behandler den rigtige instruktion. Den rigtige instruktion findes vha en tabel, der opbygges efterhånden som `INT3` instruktionerne indsættes i koden.

Kapitel 4

Analyse af et operativsystem

I dette kapitel gennemgås de forskellige faser i et operativsystem med henblik på kørsel i et virtualiseret miljø. Specifikt vil det indebære en analyse af, hvordan et OS kommunikerer med den underliggende maskine. Meningen er at danne et overblik over hvilke faciliteter, det vil være nødvendigt at simulere, men også for at identificere dele af et maskin-arkitektur, der ikke bliver brugt. Analysen vil tage udgangspunkt i et virkeligt styresystem; nemlig Linux kernen version 2.2.X til IA-32, men vil ellers forsøge at holde det arkitektur uafhængigt hvor muligt.

4.1 Faser i et operativsystem

En samlet færdiggjort datamat består ikke kun af den elektronik, som udgør størstedelen af maskinen, men ofte også af en mindre mængde programkode. Denne programkode er lagret i et vedvarende lagermedie som *flash ROM* og er den første programkode, som bliver eksekveret under opstart af datamaten (hos en IBM kompatibel PC kaldes den BIOS (Basic Input/Output System). Her initialiserer og nulstiller den de interne registre og eksterne enheder. ROM-programmet tilbyder dog også funktionalitet til et ovenliggende operativsystem efter at dette er startet op. Denne funktionalitet kunne være styring af de eksterne enheder eller standard biblioteket til et programmeringssprog. I hvert fald vil det være nødvendigt også at undersøge, hvilke dele af ROM-programmet et OS bruger for at kunne definere et minimums funtionalitetskrav til en virtualisering af en datamat.

Opstartssekvens

Opstart af en maskin-arkitektur foregår ved, at CPU'en forsøger at eksekvere kode, som findes i en forudbestemt fysisk adresse. Det vil normalt være her ROM-programmet er afbildet. ROM-programmet indeholder basale drivprogrammer, hvilket sætter det i stand til at hente det første stykke af det OS som skal køres. Denne første del af et operativsystem kaldes for en *boot loader* eller en *bootstrap* og er skrevet i assembler. En bootstrap er et meget lille program skrevet i maskinkode, og som udelukkende skal læse resten af operativsystemet ind i et sted i hukommelsen for dernæst at springe derhen.

Initialisering

Efter bootstrap programmet kommer næste del af programmet, som oftest også vil være et assembler program pga krav om, at data-strukturer skal stilles op ved adresse grænser (*alignment*). Her vil de forskellige enheder og registre blive gjort klar til OS'et. Hvis der af plads hensyn er komprimeret dele af OS-koden, vil det ligeledes være her dekomprimeringen vil blive foretaget. Når alt er klar springes der til hovedløkken.

4.1.1 Linux på IA-32

Hos IA-32 er den adresse som CPU'en springer til efter opstart lig med 0xFFFFFFF0. Her befinder BIOS rutinerne sig, der straks henter det første 512 byte segment i en floppy, HD eller CD-ROM. Segmentet indeholder en bootstrap til Linux kernen og placeres i adresse 0x00007C00, hvorfra den eksekveres.

Man kunne forstille sig, at der normalt kun ville være tale om ét bootstrap program, men på grund af IA-32's to-delt arkitektur, bliver bootstrap-begrebet i denne rapport udvidet til gælde for al 16-bit initialiseringskode. Dette skyldes den store mængde initialisering, som skal foregå, før CPU'en kan skifte fra 16-bit til 32-bit tilstanden. Af den grund skal både den næste del af bootstrap sekvensen samt selve Linux kernen hentes ind i RAM. Da et lille selvstændigt program på under 512 byte vil have svært ved at indeholde alle de nødvendige drivprogrammer til at initialisere OS'et, udnyttes BIOS' indbyggede drivprogrammer. Disse aktiveres gennem `INT n` kald til bestemte afbrydelser, der er afbildet ind i BIOS. Et kig gennem assemblerkoden for bootstrap programmet afslører at to BIOS funktioner bliver brugt: `INT 10` (Disk Input/Output) samt `INT 13` (Video Display).

Disk-rutinerne udnyttes til at hente den næste del af Linux kernen ind til adresse 0x00090200. Denne del klargør systemet til et tilstandsskift til Protected Mode. Et kig gennem kildekoden afslører, at seks BIOS rutiner bliver brugt.

- `INT 10` Disk Input/Output
- `INT 11` Equipment Determination
- `INT 13` Video Display
- `INT 15` System Services
- `INT 16` Keyboard Input
- `INT 1A` Read and Set Time

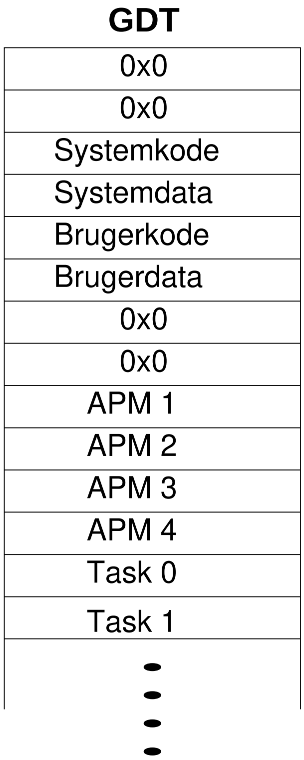
Disse BIOS afbrydelser udnyttes til at klargøre de externe enheder.

- Mængden af hukommelse i systemet
- Tastatur kontrolleren
- Videokortet
- Mus
- Disk kontroller
- Advanced Power Management (APM)

Alle informationerne fra disse enheder lægges ind på den plads, hvor den første del af bootstrap-programmet lå, dvs 0x90000 - 0x901FF. En midlertidig GDT og IDT oprettes, hvorefter datamatens PIC omprogrammeres, så afbrydelserne stemmer overens med det forventede i Protected Mode. Nu bliver CPU'en skiftet til Protected Mode og der springes til den tredige del af opstartsprocessen.

Initialisering

I denne del af initialiseringsfasen bliver den egentlige Linux kerne dekomprimeret og flyttet til adresse 0x00100000, hvis den ikke allerede er blevet det under den to-delte bootstrap. Det er nu de endelige initialiseringer af de forskellige page- og descriptor-tabeller sker. GDT'en initialiseres indeholdende otte descriptorer; en code og data segment hver til system- og brugerkode, samt fire descirptorer til evt APM funktionalitet. Derudover er der skabt plads til fire nulstillede indgange, en default LDT-descriptor (som ellers ikke bliver brugt) og en default TSS-descriptor. Resten af pladsen er fyldt op med et antal TSS-descriptorer. I alt $2 \times 4 + 4 +$ `NR_TASKS`, hvor `NR_TASKS` $= 512$. Se figur 4.1.



Figur 4.1: .

Den første indgang i side-fortegnelsen, der angiver de første 4 MB af den virtuelle hukommelse, bliver først identitetsafbildet. Senere nulstilles den og resten initialiseres efterhånden som behovet opstår. IDT'en initialiseres med 256 indgange, der alle er fyldt med en descriptor, som peger hen imod `ignore_int()`. Senere fyldes den op med standard værdier som forklaret i kapitel 4, med `INT 0x80` som eneste tilføjelse. Denne bliver brugt til at fange systemkald med. Tilsidst bliver parametrerne hentet fra BIOS kopieret over til den første side i hukommelsen, `LGDT` og `LIDT` udført og RTC indstillet.

4.2 Opsummering af kontaktfladen mellem VMM og OS

Under antagelse af Linux' brug af IA-32 er typisk for moderne IA-32 operativsystemer, kan det ses, at hvis kørsel af f.eks Linux på en virtualiseret IA-32 VM skal kunne lade sig gøre, vil det kræve lidt mere end blot hardware emulering. ROM-programmet er nemlig også en facilitet som er et krav til at sikre korrekt kørsel af IA-32 operativsystemer. Der er tre måder dette vil kunne lykkedes på.

1. Skrivning af et komplet ROM-program. Denne løsning vil være den mest korrekte, men også den absolut sværeste at implementere. Denne metode vil indebære, at det komplette ROM-program bliver afbildet det rigtige sted i VMM'ens fysiske adresserum. VMM'en vil så sørge for, at hvert VM vil begynde eksekveringen fra dette sted, hvorefter ROM-programmet overtager og henter OS'et ind. Problemet er selve ROM-programmet, som normalt er et proprietær program. En løsning kunne være en open source version...
2. Simulering af ROM-program afbrydelser. Denne metode vil være mest brugbar hos IA-32, hvor BIOS'en tilbyder en masse funktionalitet gennem afbrydelser mens i Real Mode. For at bestemme implementeringskravet, vil denne metode indebære en analyse af de operativsystemer, som skal køre oven på hver deres VM. F.eks vil kørsel af Linux indebære, at i alt seks BIOS funktioner og deres underfunktioner vil skulle simuleres. Det vil helt præcis dreje sig om:
   - `INT 10` Disk Input/Output
     - 0x03 : Read cursor position
     - 0x0E : Write Teletype
     - 0x0F : Get Current Video Mode
     - 0x12 : Check EGA/VGA (Select Alternative Screen Rutine)
     - 0x1A : Check for EGA/VGA discrimination (Read/Write Display Combination Code)
     - 0x4F : Additional SVGA BIOS Functions
   - `INT 11` Equipment Determination
   - `INT 13` Video Display
     - 0x00 : Reset Disk System
     - 0x02 : Read Disk Sectors
     - 0x08 : Get Disk Drive Parameters
     - 0x15 : Get Disk Type
   - `INT 15` System Services
     - 0x53 : APM Check
     - 0x87 : Make extended memory accessible (shift to PM and back)
     - 0x88 : Determine Extended Memory Size
     - 0xC0 : Examine externel databus
     - 0xE8 : Find top of conventional memory
   - `INT 16` Keyboard Input
     - 0x03 : Determine Status of Mouse Buttons and Position of Mouse Cursor
   - `INT 1A` Read and Set Time
     - 0x02 : Handle Time and Date for Real-Time Clock Services

   Denne metode vil være mindre krævende end den første, men vil have den ulempe, at de software afbrydelser, som bliver kaldt fra OS'et, vil ligge oven i de standard defineret afbrydelser maskinen bruger. VMM'en vil altså blive nødt til at undersøge afbrydelserne for at se, hvor de kom fra.
3. Den sidste metode vil være helt at springe Real Mode delen over. Her vil VMM'en selv hente hele kernen ind (minus Real Mode koden) og placere den i adresse 0x100000. Under afviklingen af 16-bit koden er der en mængde informationer, der

bliver gemt i hukommelsesområdet 0x00090000 - 0x000901FF. VMM'en vil selv være nødt til at sørge for, at de korrekte informationer befinder sig der når eksekveringen af OS'et begynder.

Kapitel 5

Design af VMM

5.1 Implementering på Intel IA-32

Dels pga. dens store udbredelse og dels pga. alle de værktøjer, som findes til platformen er Intels IA-32 arkitektur valgt som virtualiseringsplatform. Som udviklingsværktøjer er valget faldet på open-source assembleren NASM v0.98bf samt open-source kompileren DJGPP v2.03, som er en DOS konvertering af gcc v3.10 kendt fra UNIX. For at eksekvere VMM-kernen blev et værktøj ved navn BING Bootloader v0.40 brugt.

BING er ligeledes et DOS-program, og er skrevet til at hjælpe med OS udvikling. Den eksekveres fra DOS og skifter tilstand fra Real Mode til Protected Mode. Når det er sket kan man vælge hvilket udviklingskerne man vil eksekvere, hvorefter ens kerne tager over.

At MS-DOS blev valgt som udviklingsmiljø skyldes primært tre ting. For det første så vil en implementering af en VMM-kerne betyde, at test af kernen kun vil kunne gøres ved genstart af maskinen. MS-DOS hurtige opstartstid var en stor fordel her. Fordi DOS er et Real Mode operativsystem, vil et DOS-program have fuld kontrol over den underliggende maskine. Dermed kan et bootloader program som BING bruges, som skifter CPU-tilstand. Da der desuden fandtes glimrende udviklingsværktøjer således, at det var muligt både at udvikle og starte VMM-kernen uden at skifte miljø, blev valget let.

5.2 Strukturer

De strukturer, som bliver brugt i VMM-kernen bliver beskrevet i denne sektion.

5.2.1 regs_t

```
typedef struct {
    unsigned long edi, esi, ebp, esp, ebx, edx, ecx, eax;
    unsigned long ds, es, fs, gs;
    unsigned long which_int, err_code;
    unsigned long eip, cs, eflags, user_esp, user_ss;
} regs_t;
```

Denne struktur indeholder de vigtigste informationer og strukturer til brug under et proces-skift. De fleste struktur-værdier er selvindlysende, bortset fra `which_int` og `err_code`. Den første indeholder værdien af den afbrydelse, som forårsagede skiftet, mens den anden indeholder en evt. afbrydelsesfejlkode. `User_esp` og `user_ss` vil kun indeholde værdier, hvis et privilegie-skift samtidig med proces-skiftet. Defineret i `krnl.h`.

5.2.2 task_t

```
typedef struct _task {
    unsigned long krnl_esp;
    unsigned *page_dir;
    unsigned long size;
    void *task_mem;
    void *kstack_mem;
    struct _console *vc;
    enum
    {
        TS_RUNNABLE = 1, TS_BLOCKED = 2, TS_ZOMBIE = 3
    } status;
    unsigned long timeout;
    struct _task *next, *prev;
} task_t;
```

Indeholder værdier med relevans til hvert proces. Defineret i `krnl.h`.

krnl_esp Indeholder pegeren til stakkens nuværende placering.
page_dir Indeholder adressen til processens side-fortegnelse.
size Størrelsen af det til processen allokeret hukommelse.
task_mem Peger til begyndelsen af det allokeret hukommelse.
kstack_mem Peger til begyndelsen af stakkens hukommelse.
vc Peger til processen konsol-kode (vil ikke blive brugt, er kun med for test).
status Processens tilstand.
timeout Længden en proces skal blokeres.
next, prev Pegere til andre processer.

5.2.3 _page_use_count

```
#define MAX 32768
static unsigned char _page_use_count[MAX];
```

For hurtigt at kunne bestemme om en side er i brug eller ej, er der erklæret denne tabel i `paging.c`. Hver indgang er enten 0 eller 1 alt efter om siden er blevet allokeret eller ej.

5.2.4 floppy_t

```
typedef struct floppy {
    unsigned short base;
```

```
    volatile char state;
    fdc_format_t format;
    unsigned char head;
    unsigned char cylinder;
    unsigned char sector;
    unsigned char status[7];
    dma_block *dma;
} floppy_t;
```

Indeholder statusværdier og databytes for floppydrevet. Defineret i `floppy.h`.

base Indeholder start I/O adressen til kontrolleren. De egentlige porte fås ved at addere offset-værdier til `base`.
state Floppy-drevet globale tilstand. Bruges i `floppy_irq` for at bestemme hvilken kommando drevet har udført.
format Diskettens type. Bruges ikke endnu.
head Placeringen af drevets hoved.
cylinder Drevets aktuelle cylinder-værdi.
sector Drevets aktuelle sektor-værdi.
status Drevet returnerer information efter hvert kommando. Denne information bliver gemt her.
dma Information om DMA bufferen drevet sender data til.

5.2.5 dma_block

```
typedef struct {
    char page;
    unsigned short offset;
    unsigned short length;
} dma_block;
```

Indeholder informationer vedrørende DMA bufferen.

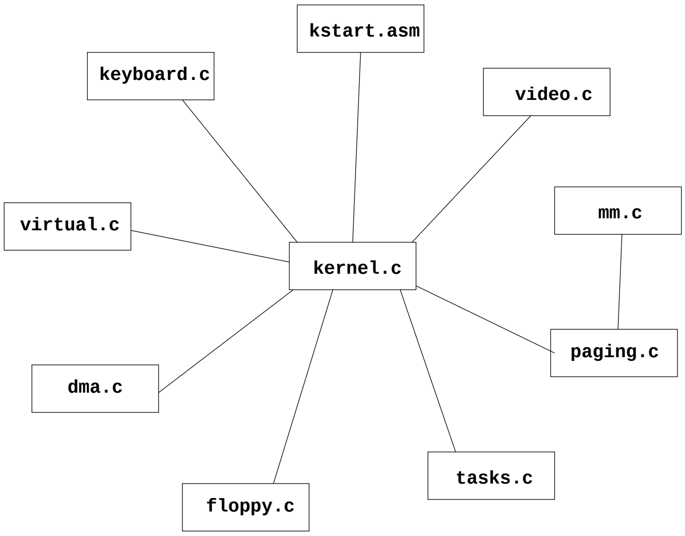
page 16-bit segmentet i den første MB af hukommelsen
offset Placeringen i segmentet
length Mængden af information som skal sendes.

5.2.6 _code, _data, _bss & _end

I link-skriptet bliver der defineret fire variabler. `_code`, `_data`, `_bss` og `_end`. Disse bliver brugt til at lokalisere de forskellige afsnit af kildekoden.

5.3 Moduler

VMM-kernen består af 10 moduler som vist i figur 5.1



Figur 5.1: Modulerne i VMM-kernen

5.3.1 kstart.asm

Dette modul er stedet, hvor kernens assemblerkode er samlet. Det er i kstart.asm, at kernen påbegynder eksekveringen. Her bliver system-tabellerne erklæret og initialiseret.

GDT

GDT-tabellen bliver initialiseret med 8 descriptorer.

1. NULL descriptor. Den første indgang i GDT tabellen er defineret fra Intels side til at være nulstillet.
2. Lineær data descriptor. Definerer et segmentet, som fylder hele adresserummet på 4GB. På denne måde skabes der et fladt adresserum således, at alle adresser kan refereres.
3. Første TSS-descriptor. Under IA-32 skal en multitasking protected mode kerne have mindst ét TSS-segment. Selvom denne kerne ikke bruger TSS-baseret proces-skift, bliver kerne-niveau stakken automatisk gemt i TSS'en under et proces-skift.
4. Anden TSS-descriptor. Bliver ikke brugt. Inkluderet pga test.
5. System-niveau kode-segment. Indeholder selve kernens maskinkode.
6. System-niveau data-segment. Indeholder kernens stak.
7. Bruger-niveau kode-segment. Indeholder VMOS'ets maskinkode.
8. Bruger-niveau data-segment. Indeholder VMOS'ets stak.

IDT-tabellen initialiseres med 49 system-niveau afbrydelsesindgange *interrupt gates*, hvoraf den sidste er en bruger-niveau indgang for systemkald. Denne bruges dog ikke og er kun inkluderet pga test. De resterende 48 indgange er delt op følgende måde: 0-31 IA-32 reserveret afbrydelser f.eks dividér med nul og stak fejl. 32-47 indeholder de eksterne afbrydelser (IRQ), der programmeres gennem PIC-chippen.

To TSS-segmenter `tss0` og `tss1` erklæres, hvor det kun er `tss0`, der bliver brugt. Dette segment er stort set nulstillet bortset fra `ss0` indgangen som skal bruges til kerne-stakken. Desuden bliver alle I/O bits sat til 1 (=beskyttet).

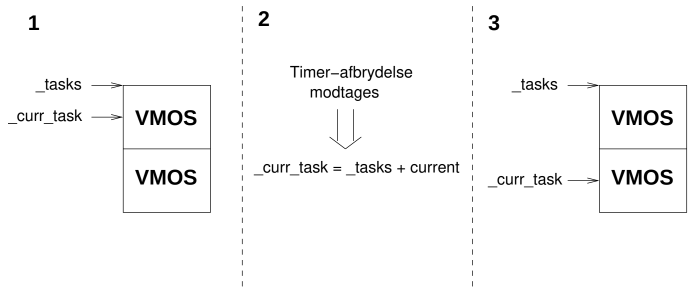
To sæt page-tabeller, hvor der bliver reserveret tabeller til 124MB til kernen og 16MB til VMOS'et.

Derudover bliver der reserveret plads til data fra BING bootloaderen. Fem variabler indeholdende forskellig information, dette bliver dog heller ikke brugt.

Foruden al initialiseringskoden indeholder `kstart.asm` og en række små hjælpefucktioner i assembler. Disse er funktioner til at hente kontrol registre eller skabe en reset.

5.3.2 kernel.c

Dette modul er som navnet antyder hovedmodulet. Det er herfra alle de andre moduler bliver kaldt. Under afviklingen af VMOS'et vil VMM'en være hel usynlig. Den vil faktisk udelukkende køre når et VMOS afgiver kontrollen pga en afbrydelse. Når en aådan bliver modtaget, vil den blive fanget af IDT'en i `kstart.asm`. Der springes nu hen til en assembler rutine, der gemmer kontekst information inden `fault` i `kernel.c` bliver kaldt. `Fault` modtager som parameter en struktur, der indeholder netop kontekst informationen. Den henter også fra en global peger den aktuelle proces-struktur. Disse bruger den til at se hvilken fejl afbrydelsen skyldes samt til at behandle den, hvilket den gør vha en stor switch case. Se figur 5.2



Figur 5.2: Afbrydelsesbehandling i VMM-kernen

Derudover ligger `main` funktionen her, men den bliver kun kørt én gang for at programmere de forskellige enheder.

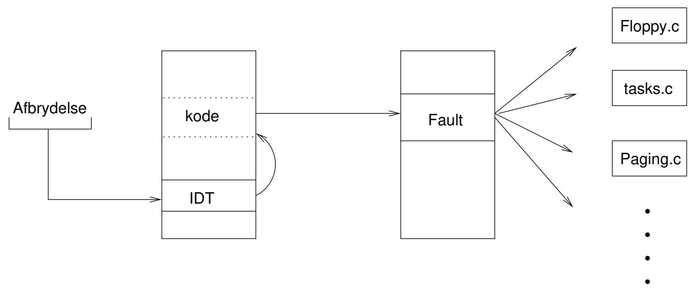
5.3.3 tasks.c

`Tasks.c` håndterer timer-afbrydelser samt schedulerer VMOS'erne. Ved modtagelsen af en timer-afbrydelse i `fault`-funktionen, springes der til `timer-irq`. Denne funktion kalder scheduleringsfunktionen `schedule`, som vælger den næste VMOS i listen. Se figur 5.3.

`Task.c` indeholder desuden to funktioner `set_descriptor_base` og `set_descriptor_limit`. Disse bliver brugt til at stille start-adressen samt størrelsen af segmenter.

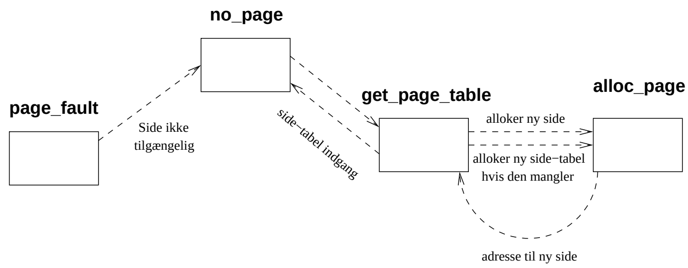
5.3.4 paging.c

Her ligger paging-rutinerne. Efter modtagelsen af en page-fejl kaldes `page_fault` i `paging.c`. `Page_fault` undersøger fejlkoden og kalder `no_page`, hvis siden ikke er tilgængelig eller



Figur 5.3: VMM-kernens scheduleringsproces.

returneres der med en fejl-værdi. Fordi VMM-kernen ikke pager til et eksternt lager vil en *side ikke tilgængelig* fejl betyde, at siden ikke er blevet initialiseret endnu. `No_page` allokerer derfor en ny side-tabel og/eller en ny side vha `get_page_table`. Forløbet illustreres i figur 5.4



Figur 5.4: Side-allokeringsmekanismen.

`Alloc_this_page` forsøger ligesom `alloc_page` at finde en ikke-allokeret side, men modtager blot som argument en adresse til den side, som skal allokeres. Hvis det ikke lykkedes returneres der med en nul-peger.

Hvis der i `page_fault` sker en beskyttelsesovertrædelse, returneres der med en fejlkode. Fra `fault`-funktionen kaldes så `hndl_prot_fault`, der analyserer instruktionen, som var skyld i fejlen. Hvis fejlen f.eks skyldes at VMOS'et har forsøgt at skrive til skærmen, behandles dette i `grafx_handler`. Hvis fejlen var noget andet returneres der fra `page_fault` med en fejlkode.

For at initialisere paging-mekanismen bliver `init_paging` kaldt fra hovedfunktionen `main` i `kernel.c`. Først bliver de områder i hukommelsen som enten er brugte eller ikke findes markeret som værende i brug i `_page_use_count`. Disse områder er VMM-kernen, området mellem 640KB og 1024KB, hvor bl.a videohukommelsen er afbildet, samt den del af adresserummet, som ikke findes fysisk. Når det er gjort bliver de i `kstart.asm` oprettet page-tabeller identitets-afbildet og deres beskyttelsesniveauer sat.

5.3.5 mm.c

Består af to funktioner nemlig `kmalloc` og `kfree`. `Kmalloc` er meget simpel og virker ved at betragte hukommelsen som én stor blok. Funktionen modtager som argument størrelsen af den blok, som skal allokeres, hvilket den runder op til nærmeste antal hele sider. Ved hjælp af `alloc_this_page` forsøger den at reservere det nødvendige antal sider. Disse sider skal ligge som en samlet blok og funktionen fejler, hvis det ikke kan lade sig gøre. Hver gang `kmalloc` kaldes bliver den nye blok lagt lige efter den tidligere. Slutadressen af den sidste blok bliver lagret i den statiske variabel `brk`.

Funktionen `kfree` er ikke implementeret pga. tidskrav.

5.3.6 virtual.c

Dette modul indeholder koden til selve virtualiseringsprocessen. Fra `main` bliver funktionen `init_testcode` kaldt. Her bliver processens `task_t` og `regs_t` strukturer erklæret og initialiseret. Da der her er tale om kode til test brug og ikke et egentligt VMOS, bliver der kun allokeret 16MB hukommelse, koden bliver hentet ind til adresse 0x100000, descriptor-værdierne bliver sat og testkoden kopieret fra den midlertidig lager. Tilsidst bliver testkodens egen page-tabeller initialiseret i `init_guest_paging`.

`Init_guest_paging` minder meget om `init_paging`. Den første del af processens lineære adresse bliver identitetsafbildet således, at descriptor-tabellerne ikke pludselig ændrer adresser under skiftet fra kerne page-tabellerne til proces page-tabellerne. Bagefter bliver processens allokeret hukommelse afbildet ind i siderne og beskyttelsesniveauerne sat.

Tilsidst indholder `virtual.c` funktionerne `instr_handler` og `check_mov`. Disse funktioner undersøger og behandler de instruktioner, som genererer en beskyttelsesafbrydelse.

5.3.7 DMA.c

Her ligger driv-rutinerne til DMA chippen. `Pause_DMA`, `start_DMA`, `stop_DMA` og `unpause_DMA` gør som deres navne antyder og følger de specifikationer givet i [4]. `Alloc_DMA` allokerer en DMA-buffer i <640MB området.

5.3.8 floppy.c

Her ligger al kode vedrørende diskdrevet. `get_byte`, `send_byte`, `recalibrate`, `seek` og `read_sector` følger de specifikationer givet i [3] og gennem granskning af kildekode givet i Möbius OS (www.themoebius.org.uk).Fra `main` bliver der kaldt `init_floppy`, som opretter en `floppy_t` struktur og klargør drevet.

Modtages en floppy-afbrydelse bliver funktionen `floppy_irq` kaldt fra `fault`. Denne henter information fra enheden alt efter drevets status. I `load_testcode` og `load_kernel` bliver henholdsvis testkode og Linux kernen hentet ind i hukommelsen.

5.3.9 keyboard.c & video.c

Disse moduler er lånt open-source kode og tilhører ikke VM-kernen. `Video.c` modulet bliver kun brugt i VMM-kernen til at printe test-data ud til skærmen med vha. `kprintf`, mens `keyboard.c`, er kun med for at skifte konsol i `video.c`.

Kapitel 6

Konklusion

Desværre må det konkluderes, at projektet ikke nåede at blive færdig-implementeret. Et af problemerne ved at arbejde på så lavt et niveau er, at det kan være problematisk at lokalisere tilstrækkeligt med teknisk dokumentation vedrørende de mange forskellige enheder, som findes i en arkitektur som en standard PC. I betragtning af PC'ens udbredelse forekommer dette overraskende! Det skyldes dog nok, at det ikke længere er så udbredt at udvikle så tæt på maskinen. Et andet problem, som var særdeles aktuelt i løbet af projektet, var undertegnedes manglende erfaring med denne udviklingsform. De almindelige hjælpemidler, som man normalt tager for givet, er stort set ikke-eksisterende. Dette betyder, at det kræver en langt større intuitiv fornemmelse for de mekanismer som ens programkode sætter igang, hvilket er noget som kun opnås gennem erfaring.

Med de foreslået teknikker mener undertegnet dog at have dækket de områder, der skal til, for at implementere en korrekt fungerende VMM. Forslag er blevet givet til håndtering af de mest problematiske områder som descriptor-tabellerne og paging-systemet. Dertil kommer en gennemgang af de instruktioner som vil volde problemer på en IA-32. Forslaget om SBE (Scan Before Execute) til at håndtere de følsomme ubeskyttede instruktioner kan dog måske være problematisk at implementere. Dertil kommer, at den sandsynligvis meget store effektivitetstab pga de mange afbrydelser, hvilket sammen med de andre VMOS'er og deres paging systemer måske kan gå så vidt at producere *thrashing* af paging-systemet. I sidste kan det være at de kommercielle produkters brug af JIT-teknologi både vil give mere kontrol samt god udførelseshastighed.

6.1 Mangler

Foruden det faktum, at metoderne foreslået i denne rapport ikke er blevet afprøvet, vil der være nogle steder, hvor virtualiseringen vil bryde sammen. Først og fremmest vil SBE teknikken foreslået i denne rapport skabe problemer for visse typer applikationer, som læser eller skriver til eksekverende kode. Da SBE indsætter breakpoint instruktioner, vil et program, der undersøger den ændrede kode opdage disse fremmede instruktioner. Omvendt vil et program, som selv modificerer kode (f.eks en debugger) kunne ødelægge den skannede kode og gør den usikker. [10, 13] foreslår en måde, hvor man kan udnytte tekniske egenskaber omkring TLB (Translation Lookaside Buffer) bufferne til at mærke kode som værende *execute only*, hvilket ellers ikke er muligt i IA-32.

Ellers drejer problemerne sig om manglende faciliteter, såsom understøttelse af flydende kommatal, MMX/SSE instruktioner, SMM (System Management Mode) og MSR (Model-Specific Registers).

19. August 2002
c958255
Niels Cunningham Glimø
IMM
DTU
Lyngby

Litteratur

[1] Albert S. Woodhill Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*, chapter 4.6, pages 343–356. Prentice-Hall International, Inc., 1997.

[2] Connectix Corp. The technology of virtual machines. Technical report, Connectix Corp, 2001. http://www.connectex.com/.

[3] Intel Corporation. 8259a programmable interrupt controller datasheet. Technical Report 231468-003, Intel Corporation, December 1988.

[4] Intel Corporation. 8237a programmable dma controller datasheet. Technical Report 231 466-005, Intel Coporation, September 1993.

[5] Intel Corporation. Ia-32 intel architecture software developer's manual, vol. 1: Basic architecture. Technical Report 245470, Intel Corporation, 2001.

[6] Intel Corporation. Ia-32 intel architecture software developer's manual, vol. 2: Instruction set reference. Technical Report 245471, Intel Corporation, 2001.

[7] Intel Corporation. Ia-32 intel architecture software developer's manual, vol. 3: System programming guide. Technical Report 245472, Intel Corporation, 2001.

[8] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel*, chapter 4, pages 104–105. O'Reilly & Associates, Inc., 1st edition, 2001.

[9] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., 1st edition, January 2001.

[10] Kevin Lawton et. al. Running multiple operating systems concurrently on an ia-32 pc using virtualization techniques. http://www.plex86.org/research/paper.txt, November 1999.

[11] Cynthia E. Irvine John S. Robin. Analysis of the intel pentium's ability to support a secure virtual machine monitor. *Proceedings of the 9th USENIX Security Symposium*, pages 129–144, August 2000.

[12] Paul T. Robinson Judith S. Hall. Virtualizing the vax architecture. *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 19(3):380–389, April 1991.

[13] Kevin Lawton. The plex86 internals guide. ftp://plex86.org/docs/, November 2001.

[14] Tom Shanley. *Protected Mode Software Architecture*. The PC System Architecture Series. Addison-Wesley Publishing Company, 1st edition, February 1996.

[15] Jonathan Shapiro. Ia-32 emulation. http://www.eros-os.org/design-notes/IA32-Emulation.html, March 2002.

[16] Robin Sharp. High performance operating systems. Course notes for High Performance Operating Systems course at DTU, Lyngby, Denmark, August 2001.

[17] John J. Donovan Stuart E. Madnick. *Operating Systems*, chapter 9-5, pages 549–563. McGraw-Hill Computer Science Series. McGraw-Hill, Inc., 1974.

Bilag A

kstart.asm

```
%macro  EXPORT  1
        GLOBAL %1
        %1:
        GLOBAL _%1
        _%1:
%endmacro

%macro  IMPORT 1
%ifdef UNDERBARS
        EXTERN _%1              ; GCC for DOS (DJGPP; COFF)
        %define %1 _%1
%else
        EXTERN %1               ; GCC with Linux (ELF)
%endif
%endmacro

; IMPORTS
; from linker script:
IMPORT _code
IMPORT _data
IMPORT _bss
IMPORT _end

DS_MAGIC equ 3544DA2Ah

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; pmode entry point
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

[SECTION .text]
[BITS 32]
; all data segment accesses must be relative to ESI until we activate paging
; virt_adr + virt_to_phys = virt_adr + (phys - virt) = phys_adr
; phys_adr - virt_to_phys = phys_adr - (phys - virt) = virt_adr

EXPORT entry
; where did the loader put us?
        call where_am_i
where_am_i:
        pop esi                         ; ESI=physical adr (where we are)
        sub esi,where_am_i              ; subtract virtual adr (where we want to be)
```

```
                                        ; now ESI=virt-to-phys
        mov [_virt_to_phys + esi],esi   ; Save it

; check if data segment linked/located/loaded properly
    mov eax,[esi + ds_magic]
    cmp eax,DS_MAGIC
    je ds_ok

; display a blinking white-on-blue 'D' if bad data segment
    mov word [0B8000h],9F44h
    jmp short $
ds_ok:

        ; Initialise paging
        call init_paging
        jnc paging_ok

; display a blinking white-on-blue 'P' if error setting up page tables
        mov word [0B8000h],9F50h
        jmp short $

paging_ok:
        lea eax,[esi + task0_page_dir]
        mov cr3,eax

        mov eax,cr0
;        or eax,080000000h  ; xxx - make sure you have a 486+
;        or eax,080010000h  ; set the 486 WP (page write protect) bit
        or eax,0C0010000h   ; set the 486 CE (cache enable) and WP bits
        mov cr0,eax
        jmp short paging_enabled    ; near, EIP-relative JMP; to physical address

paging_enabled:
        mov ebx,virtual_addresses_enabled
        jmp ebx             ; near, absolute JMP; to virtual address

virtual_addresses_enabled:
; Now that paging is enabled, there is no need to initialise the base
; addresses in the GDT selectors (virtual addresses always start from zero).
; TSS stil needs to be set however :-)

; loader GDT is gone (or not what we want); so set up a new one.
; Note that all data segment addressing is relative to ESI

;        mov eax,esi
;        shr eax,16

;        add [gdt2 + 2 + esi],si
;        adc [gdt2 + 4 + esi],al
;        adc [gdt2 + 7 + esi],ah

;        add [gdt3 + 2 + esi],si
;        adc [gdt3 + 4 + esi],al
;        adc [gdt3 + 7 + esi],ah

;        add [gdt_ptr + 2 + esi],esi     ; gdt_ptr = LINEAR adr of GDT
;        add [idt_ptr + 2 + esi],esi     ; idt_ptr = LINEAR adr of IDT
```

```
        lea eax,[tss0]          ; Fix up kernel TSS descriptor
        mov [gdt1 + 2],ax
        shr eax,16
        mov [gdt1 + 4],al
        mov [gdt1 + 7],ah

        lea eax, [tss1]         ; Fix up guest TSS descriptor
        mov [gdt2 + 2], ax
        shr eax, 16
        mov [gdt2 + 4], al
        mov [gdt2 + 7], ah

;        lgdt [gdt_ptr + esi]

        lgdt [gdt_ptr]                      ; stop using bootloader GDT; load new GDT

        mov ax,SYS_DATA_SEL
        mov ds,eax
        mov ss,eax
        mov es,eax
        mov fs,eax
        mov gs,eax

; Now that the kernel code selector has been initialised;
; perform a far jump to load new selector
        jmp SYS_CODE_SEL:pmode
pmode:

        mov esp,stack           ; set up pmode stack

; Since new selector has been loaded with base address relative to kernel code,
; the esi register does not need to be added (old selector probably had base at 0x0).
;        mov [_virt_to_phys],esi

; LTR is illegal in real mode; can do it now
        mov ax,TSS0_SEL
        ltr ax

; set up interrupt handlers
        mov ecx,(idt_end - idt) >> 3    ; number of exception handlers
        mov ebx,idt
        mov edx,isr0
do_idt:
        mov eax,edx             ; EAX=offset of entry point
        mov [ebx],ax            ; set low 16 bits of gate offset
        shr eax,16
        mov [ebx + 6],ax        ; set high 16 bits of gate offset
        add ebx,8               ; 8 bytes/interrupt gate
        add edx,(isr1 - isr0)   ; 9 bytes/stub (push+push+32-bit jmp)
        loop do_idt

        lidt [idt_ptr]

; the loader pokes boot-time information into the first 4K of the BSS
; and zeroes the rest, so we don't have to
;IMPORT edata
;IMPORT end
```

```
;       xor eax,eax
;       mov edi,edata           ; BSS starts at 'edata'
;       mov ecx,end
;       sub ecx,edi             ; 'end' - 'edata' bytes
;       cld
;       rep stosb


IMPORT main
        call main               ; call C code
        jmp $                   ; freeze


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:         map_page
; action:       maps one page into the page tables
; in:           EAX=page virtual address
;           EBX=page directory physical address
;           DX=page privilege
;           EDI=page physical address
; out (error):      CY=1 if a necessary page table is not installed
; out (success):    CY=0
; modifies:     (nothing)
; notes:        all address translation must be turned off when
;           this function is called: no paging,
;           and base address of data segment == 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

map_page:
    push ebx
    push eax

; get index-into-page-directory from page virtual address
        shr eax,22

; add index to page directory physical address
        shl eax,2
        add ebx,eax
    pop eax
    push eax

; get page directory entry (PDE)
        mov ebx,[ebx]
mov byte [0B8668h], 'J'

; convert PDE to page table physical address
; error if no page table installed
        and ebx,0FFFFF000h
        stc
        je map_page_err
mov byte [0B8672h], 'K'
; get index-into-page-table from page virtual address
; Note: the 'and' instruction clears the carry
        shr eax,12
        and eax,000003FFh
        shl eax,2

; add index to page table physical address, forming physical
```

```
; address of page table entry (PTE)
        add ebx,eax

; create PTE from page physical address and privilege
        mov eax,edi
        and eax,0FFFFF000h
        or ax,dx
; store PTE
        mov [ebx],eax
map_page_err:
    pop eax
    pop ebx
    ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:         map_pages
; action:       maps a range of pages with contiguous virtual
;           and physical addresses into the page tables
; in:           EAX=first page virtual address
;           EBX=page directory physical address
;           ECX=range size, in bytes
;           DX=page privilege
;           EDI=first page physical address
; out (error):      CY=1 if a necessary page table is not installed
; out (success):    CY=0
; modifies:     (nothing)
; notes:        all address translation must be turned off when
;           this function is called: no paging,
;           and base address of data segment == 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

map_pages:
    pusha
        and eax,0FFFFF000h
        and ebx,0FFFFF000h
        and ecx,0FFFFF000h
        je map_pages_2
map_pages_1:
        call map_page
        jc map_pages_2
        add edi,4096
        add eax,4096
        sub ecx,4096
        jne map_pages_1
map_pages_2:
    popa
    ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Print ebx on screen
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
print_num:
mov eax, ebx
mov ecx, 30h
and eax, 0F0000000h
shr eax, 28
add eax, ecx
mov byte [0B8650h], al
```

```
mov eax, ebx
and eax, 0F000000h
shr eax, 24
add eax, ecx
mov byte [0B8652h], al
mov eax, ebx
and eax, 00F00000h
shr eax, 20
add eax, ecx
mov byte [0B8654h], al
mov eax, ebx
and eax, 000F0000h
shr eax, 16
add eax, ecx
mov byte [0B8656h], al
mov eax, ebx
and eax, 0000F000h
shr eax, 12
add eax, ecx
mov byte [0B8658h], al
mov eax, ebx
and eax, 00000F00h
shr eax, 8
add eax, ecx
mov byte [0B865Ah], al
mov eax, ebx
and eax, 000000F0h
shr eax, 4
add eax, ecx
mov byte [0B865Ch], al
mov eax, ebx
and eax, 0000000Fh
add eax, ecx
mov byte [0B865Eh], al
ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:         init_paging
; action:
; in:           ESI=kernel virt-to-phys
; out (error):      CY=1 if a necessary page table is not installed
; out (success):    CY=0
; modifies:     (nothing)
; notes:        all address translation must be turned off when
;           this function is called: no paging,
;           and base address of data segment == 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

init_paging:
    pusha

; get physical address of page dir to EBX
        lea ebx,[esi + task0_page_dir]

; SET UP PAGE DIRECTORY:
; make an entry for conv_mem_page_table
        lea eax,[esi + conv_mem_page_table]
        or al,7     ; ring 3, writable, present
```

```
        mov [ebx + 0],eax

; get physical address of kernel to EDI, convert to page dir index
        mov edi,_code
        add edi,esi
        shr edi,22
        shl edi,2

; make entries for im_kernel_page_tables. If im_kernel_page_table_1
; maps the same memory as conv_mem_page_table, then use conv_mem_page_table
        xor eax,eax
        or eax,[ebx + edi]
        jne init_paging_1

        lea eax,[esi + im_kernel_page_table_1]
        or al,7     ; ring 3, writable, present
        mov [ebx + edi],eax
init_paging_1:
        lea eax,[esi + im_kernel_page_table_2]
        or al,7     ; ring 3, writable, present
        mov [ebx + edi + 4],eax

; get virtual address of kernel to EDI, convert to page dir index
        mov edi,_code
        shr edi,22
        shl edi,2

; make entries for kernel_page_tables
        xor eax,eax
        or eax,[ebx + edi]
        lea eax,[esi + conv_mem_page_table]
        jne init_paging_2
        lea eax,[esi + kernel_page_table_1]
init_paging_2:
        or al,7     ; ring 3, writable, present
        mov [ebx + edi],eax

        lea eax,[esi + kernel_page_table_2]
        or al,7     ; ring 3, writable, present
        mov [ebx + edi + 4],eax

; make an entry for page dir itself
; xxx - maybe the 4092 should be a named constant, not a magic value
        lea eax,[esi + task0_page_dir]
        or al,7     ; ring 3, writable, present
        mov [ebx + 4092],eax

; SET UP PAGE TABLES
; identity-map conventional RAM read-write
; this includes the BIOS data segment in page 0 and the RDSK file
        mov dx,7            ; privilege = ring 3, writable, present
        xor eax,eax         ; virtual address = 0
        mov edi,eax         ; physical address is the same
; The BING bootloader seems to store too small a value
; into _conv_mem (ca. 468kb), so hardcode correct value
;        mov ecx, [esi + _conv_mem]
        mov ecx, 000A0000h
        call map_pages
```

```
        jc near init_paging_err

; identity-map video memory (framebuffer) read-write
        mov dx,3            ; privilege = ring 0, writable, present
        mov eax,0A0000h     ; virtual address
        mov edi,eax         ; physical address is the same
        mov ecx,20000h      ; 0xA0000 to 0xBFFFF
        call map_pages
mov byte [0B8004h], 'x'
        jc near init_paging_err
mov byte [0B8002h], 'c'

; identity-map BIOS ROMs read-only
; xxx - what I'd really like to do here is:
; 1. find location and size of video BIOS ROM and map it read-only
; 2. find location and size of motherboard BIOS ROM and map it read-only
; 3. leave other memory unmapped, mapping it later if necessary
;   e.g. for Ethernet card with shared memory
        mov dx,1            ; privilege = ring 0, read-only, present
        mov eax,0C0000h     ; virtual address
        mov edi,eax         ; physical address is the same
        mov ecx,40000h      ; 0xC0000 to 0xFFFFF
        call map_pages
        jc near init_paging_err
mov byte [0B8002h],'d'

; identity-map kernel code (and possibly rodata) read-only
        mov dx,1            ; privilege = ring 0, read-only, present
        mov edi,_code
        add edi,esi         ; physical address
        mov eax,edi         ; virtual address is the same
        mov ecx,_data
        sub ecx,_code
        call map_pages
;        jc init_paging_err
; xxx - fails here
mov byte [0B8002h],'e'

; identity-map kernel data and BSS read-write
        mov dx,3            ; privilege = ring 0, writable, present
        mov edi,_data
        add edi,esi         ; physical address
        mov eax,edi         ; virtual address is the same
        mov ecx,_end
        sub ecx,_data
        call map_pages
        jc init_paging_err
mov byte [0B8002h],'f'

; map kernel code (and possibly rodata) read-only
        mov dx,1            ; privilege = ring 0, read-only, present
        mov eax,_code       ; virtual address
        mov edi,eax
        add edi,esi         ; physical address
        mov ecx,_data
        sub ecx,_code
        call map_pages
        jc init_paging_err
```

```
mov byte [0B8002h],'g'

; map kernel data and BSS read-write
        mov dx,3            ; privilege = ring 0, writable, present
        mov eax,_data       ; virtual address
        mov edi,eax
        add edi,esi         ; physical address
        mov ecx,_end
        sub ecx,_data
        call map_pages
jc init_paging_err
mov byte [0B8002h],'h'
init_paging_err:
    popa
    ret


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:         get_page_fault_address
; action:       gets virtual address of latest page fault
;                   from register CR2
; in:           (nothing)
; out:          EAX=page fault address
; modifies:     EAX
; notes:        C prototype: unsigned get_page_fault_address(void);
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

EXPORT get_page_fault_address
    mov eax,cr2
    ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:         get_cr3
; action:       gets stuff from register CR3
; in:           (nothing)
; out:          EAX=page fault address
; modifies:     EAX
; notes:        C prototype: unsigned get_cr3(void);
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

EXPORT get_cr3
    mov eax, cr3
    ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:         get_cr0
; action:       gets stuff from register CR0
; in:           (nothing)
; out:          EAX=page fault address
; modifies:     EAX
; notes:        C prototype: unsigned get_cr0(void);
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

EXPORT get_cr0
    mov eax, cr0
    ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; Reboot by causing a deliberate triple fault
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

EXPORT reboot
        cli
        lidt [bogus_idt_ptr]
        int 0
        jmp $

bogus_idt_ptr:
        dw 0    ; IDT limit
        dd 0    ; IDT linear address

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Halt CPU until interrupt occurs.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

EXPORT halt
        hlt

EXPORT do_nothing
        ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; interrupt/exception handlers
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

fault1:
        push gs                 ; push segment registers
        push fs
        push es
        push ds
        pusha                   ; push EDI,ESI,EBP,ESP,EBX,EDX,ECX,EAX
                mov ax,SYS_DATA_SEL ; put known-good values in seg regs
                mov ds,eax
                mov es,eax
                mov fs,eax
                mov gs,eax
IMPORT fault
                call fault

; fault() returns old value of _curr_task
; save previous krnl_esp
                mov ebx,eax             ; Get old _curr_task
                mov [ebx + 0],esp       ; Save old ESP

; load new krnl_esp
IMPORT _curr_task
                mov esi,[_curr_task]    ; Get new _curr_task
                mov esp,[esi + 0]       ; Load new ESP

; May be needed at some point...
; Change to new page directory table if necessary
                mov eax,[esi + 4]       ; Get new CR3
                cmp eax,[ebx + 4]       ; Changing CR3?
                je no_switch
;                mov [tss1 + 28], eax    ; Insert new CR3 into TSS
;                mov cr3,eax             ; Insert new CR3
```

```
                mov ebx, cr0
                and ebx, 7FFFFFFFh
                mov cr0, ebx            ; Disable paging
                mov cr3, eax             ; Insert new CR3
                mov ebx, cr0
                or  ebx, 80000000h
                mov cr0, ebx            ; Reenable paging
        jmp short paging_enabled1    ; near, EIP-relative JMP; to physical address

paging_enabled1:
        mov ebx,virtual_addresses_enabled1
        jmp ebx             ; near, absolute JMP; to virtual address

virtual_addresses_enabled1:

no_switch:
                lea eax,[esp + 76]
                mov [tss0_esp0],eax

        popa                    ; pop GP registers
        pop ds                  ; pop segment registers
        pop es
        pop fs
        pop gs
        add esp,8               ; drop exception number and error code

        iret

; The stack is always 32 bits wide in 32-bit pmode therefor
; the push will zero-extend (sign-extend?) the byte to 32 bits
; The extra push when there is no error code, is to ensure everything matches regs_t
isr0:
        push byte 0
        push byte 0
        jmp fault1              ; zero divide (fault)
isr1:
        push byte 0
        push byte 1
        jmp fault1              ; debug/single step
isr2:
        push byte 0
        push byte 2
        jmp fault1              ; non-maskable interrupt (trap)
isr3:
        push byte 0
        push byte 3
        jmp fault1              ; INT3 (trap)
isr4:
        push byte 0
        push byte 4
        jmp fault1              ; INTO (trap)
isr5:
        push byte 0
        push byte 5
        jmp fault1              ; BOUND (fault)
isr6:
        push byte 0
```

```
        push byte 6
        jmp fault1              ; invalid opcode (fault)
isr7:
        push byte 0
        push byte 7
        jmp fault1              ; coprocessor not available (fault)
isr8:
        nop
        nop
        push byte 8
        jmp fault1              ; double fault (abort w/ error code)
isr9:
        push byte 0
        push byte 9
        jmp fault1              ; coproc segment overrun (abort; 386/486SX only)
isr0A:
        nop
        nop
        push byte 0Ah
        jmp fault1              ; bad TSS (fault w/ error code)
isr0B:
        nop
        nop
        push byte 0Bh
        jmp fault1              ; segment not present (fault w/ error code)
isr0C:
        nop
        nop
        push byte 0Ch
        jmp fault1              ; stack fault (fault w/ error code)
isr0D:
        nop
        nop
        push byte 0Dh
        jmp fault1              ; GPF (fault w/ error code)
isr0E:
        nop
        nop
        push byte 0Eh
        jmp fault1              ; page fault
isr0F:
        push byte 0
        push byte 0Fh
        jmp fault1              ; reserved
isr10:
        push byte 0
        push byte 10h
        jmp fault1              ; FP exception/coprocessor error (trap)
isr11:
        push byte 0
        push byte 11h
        jmp fault1              ; alignment check (trap; 486+ only)
isr12:
        push byte 0
        push byte 12h
        jmp fault1              ; machine check (Pentium+ only)
isr13:
        push byte 0
```

```
        push byte 13h
        jmp fault1
isr14:
        push byte 0
        push byte 14h
        jmp fault1
isr15:
        push byte 0
        push byte 15h
        jmp fault1
isr16:
        push byte 0
        push byte 16h
        jmp fault1
isr17:
        push byte 0
        push byte 17h
        jmp fault1
isr18:
        push byte 0
        push byte 18h
        jmp fault1
isr19:
        push byte 0
        push byte 19h
        jmp fault1
isr1A:
        push byte 0
        push byte 1Ah
        jmp fault1
isr1B:
        push byte 0
        push byte 1Bh
        jmp fault1
isr1C:
        push byte 0
        push byte 1Ch
        jmp fault1
isr1D:
        push byte 0
        push byte 1Dh
        jmp fault1
isr1E:
        push byte 0
        push byte 1Eh
        jmp fault1
isr1F:
        push byte 0
        push byte 1Fh
        jmp fault1

; isr20 through isr2F are hardware interrupts. The 8259 programmable
; interrupt controller (PIC) chips must be reprogrammed to make these work.
isr20:
        push byte 0
        push byte 20h
        jmp fault1              ; IRQ 0/timer interrupt
isr21:
```

```
        push byte 0
        push byte 21h
        jmp fault1              ; IRQ 1/keyboard interrupt
isr22:
        push byte 0
        push byte 22h
        jmp fault1
isr23:
        push byte 0
        push byte 23h
        jmp fault1
isr24:
        push byte 0
        push byte 24h
        jmp fault1
isr25:
        push byte 0
        push byte 25h
        jmp fault1
isr26:
        push byte 0
        push byte 26h
        jmp fault1              ; IRQ 6/floppy interrupt
isr27:
        push byte 0
        push byte 27h
        jmp fault1
isr28:
        push byte 0
        push byte 28h
        jmp fault1              ; IRQ 8/real-time clock interrupt
isr29:
        push byte 0
        push byte 29h
        jmp fault1
isr2A:
        push byte 0
        push byte 2Ah
        jmp fault1
isr2B:
        push byte 0
        push byte 2Bh
        jmp fault1
isr2C:
        push byte 0
        push byte 2Ch
        jmp fault1
isr2D:
        push byte 0
        push byte 2Dh
        jmp fault1              ; IRQ 13/math coprocessor interrupt
isr2E:
        push byte 0
        push byte 2Eh
        jmp fault1              ; IRQ 14/primary ATA ("IDE") drive interrupt
isr2F:
        push byte 0
        push byte 2Fh
```

```
        jmp fault1              ; IRQ 15/secondary ATA drive interrupt

; syscall software interrupt
isr30:
        push byte 0
        push byte 30h
        jmp fault1

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

[SECTION .data]

ds_magic:
        dd DS_MAGIC

EXPORT _virt_to_phys
        dd 0

; 32 ring 0 interrupt gates
idt:
        times 48        dw 0, SYS_CODE_SEL, 8E00h, 0    ; 48 = 30h
;        times 26        dw 0, SYS_CODE_SEL, 8E00h, 0
; One ring 3 task gate for timer interrupts (INT 20h)
; As the offset bytes are not used in a task gate,
; it doesn't matter what I put in there
;        dw 0                ; offset 15:0
;        dw TSS0_SEL         ; selector
;        db 0                ; (always 0 for interrupt gates)
;        db 0E5h             ; present,ring 3,'386 task gate
;        dw 0                ; offset 31:16

;        times 21        dw 0, SYS_CODE_SEL, 8E00h, 0

; one ring 3 interrupt gate for syscalls (INT 30h)
        dw 0                ; offset 15:0
        dw SYS_CODE_SEL     ; selector
        db 0                ; (always 0 for interrupt gates)
        db 0EEh             ; present,ring 3,'386 interrupt gate
        dw 0                ; offset 31:16

idt_end:

EXPORT idt_ptr
        dw idt_end - idt - 1    ; IDT limit
        dd idt                  ; linear adr of IDT (set above)

; Even if you don't use TSS-based task-switching, you
; need one TSS to hold the kernel (ring 0) stack pointer.
; I don't have to fill in the structure, as the CPU
; will do it for me during a task-switch (I think...)
tss0:
        dw TSS1_SEL, 0          ; back link
tss0_esp0:
        dd 0                    ; ESP0
        dw SYS_DATA_SEL, 0      ; SS0, reserved

        dd 0                    ; ESP1
```

```
        dw 0, 0                 ; SS1, reserved

        dd 0                    ; ESP2
        dw 0, 0                 ; SS2, reserved

        dd 0                    ; CR3   (byte 1C (28) from tss)
tss0_eip:
        dd 0, 0                 ; EIP, EFLAGS
        dd 0, 0, 0, 0           ; EAX, ECX, EDX, EBX
        dd 0, 0, 0, 0           ; ESP, EBP, ESI, EDI
        dw 0, 0                 ; ES, reserved
        dw 0, 0                 ; CS, reserved
        dw 0, 0                 ; SS, reserved
        dw 0, 0                 ; DS, reserved
        dw 0, 0                 ; FS, reserved
        dw 0, 0                 ; GS, reserved
        dw 0, 0                 ; LDT, reserved
        dw 0, 104               ; debug, IO permission bitmap base
;        times 128 db 0          ; IO permission bitmap up to 1024 (0 = OK, 1 = forbidden)
        times 128 db 255        ; IO permission bitmap up to 1024 (0 = OK, 1 = forbidden)

; TSS for guest code
; This I have to initialise somewhere before use
EXPORT tss1
        dw TSS1_SEL, 0          ; back link
tss1_esp0:
        dd 0                    ; ESP0
        dw SYS_DATA_SEL, 0      ; SS0, reserved

        dd 0                    ; ESP1
        dw 0, 0                 ; SS1, reserved

        dd 0                    ; ESP2
        dw 0, 0                 ; SS2, reserved

        dd 0                    ; CR3   (byte 1C (28) from tss1)
        dd 0, 0                 ; EIP, EFLAGS
        dd 0, 0, 0, 0           ; EAX, ECX, EDX, EBX
        dd 0, 0, 0, 0           ; ESP, EBP, ESI, EDI
        dw 0, 0                 ; ES, reserved
        dw 0, 0                 ; CS, reserved
        dw 0, 0                 ; SS, reserved
        dw 0, 0                 ; DS, reserved
        dw 0, 0                 ; FS, reserved
        dw 0, 0                 ; GS, reserved
        dw 0, 0                 ; LDT, reserved
        dw 0, 104               ; debug, IO permission bitmap base
        times 128 db 0          ; IO permission bitmap up to 1024 (0 = OK, 1 = forbidden)
;        times 128 db 255        ; IO permission bitmap up to 1024 (0 = OK, 1 = forbidden)

; Null descriptor. gdt_ptr could be put here to save a few
; bytes, but that can be confusing.
EXPORT gdt
        dw 0            ; limit 15:0
        dw 0            ; base 15:0
        db 0            ; base 23:16
        db 0            ; type
        db 0            ; limit 19:16, flags
```

```
        db 0            ; base 31:24

; Linear data segment descriptor
LINEAR_SEL      equ     $-gdt           ; 08
        dw 0FFFFh       ; limit 0FFFFFh (1 meg or 4 gig)
        dw 0            ; base for this one is always 0
        db 0
        db 92h          ; present,ring 0,data,expand-up,writable
        db 0CFh         ; page-granular (4 gig limit), 32-bit
        db 0

; Descriptor for task-state segment (TSS)
TSS0_SEL        equ     $-gdt           ; 10
gdt1:
        dw 232
        dw 0
        db 0
        db 89h          ; ring 0 available 32-bit TSS
        db 0
        db 0

TSS1_SEL        equ     $-gdt           ; 18
gdt2:
        dw 232
        dw 0
        db 0
        db 0E9h         ; ring 3 available 32-bit TSS
        db 0
        db 0

; Ring 0 kernel code segment descriptor
; !!! - Compare initialisation code with COSMOS
SYS_CODE_SEL    equ     $-gdt           ; 20
gdt3:
        dw 0FFFFh
        dw 0            ; (base gets set above)
        db 0
        db 9Ah          ; present,ring 0,code,non-conforming,readable
        db 0CFh
        db 0

; Ring 0 kernel data segment descriptor
SYS_DATA_SEL    equ     $-gdt           ; 28
gdt4:
        dw 0FFFFh
        dw 0            ; (base gets set above)
        db 0
        db 92h          ; present,ring 0,data,expand-up,writable
        db 0CFh
        db 0

; Ring 3 user code segment descriptor
; The first line is the selector
; The first two bits set are the RPL (Requestor Piviledge Level)
; The third bit is the TI (Table Indicator), TI=1 means that the entry is in the LDT
USER_CODE_SEL   equ     ($-gdt) | 3     ; 30 (33)
EXPORT _gdt5
        dw 0FFFFh
```

```
        dw 0            ; (base gets set later)
        db 0
        db 0FAh         ; present,ring 3,code,non-conforming,readable
        db 0CFh
        db 0

; Ring 3 user data segment descriptor
USER_DATA_SEL   equ     ($-gdt) | 3     ; 38 (3B)
EXPORT _gdt6
        dw 0FFFFh
        dw 0            ; (base gets set later)
        db 0
        db 0F2h         ; present,ring 3,data,expand-up,writable
        db 0CFh
        db 0

gdt_end:

EXPORT gdt_ptr
        dw gdt_end - gdt - 1            ; GDT limit
        dd gdt                          ; linear adr of GDT (set above)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

[SECTION .bss]

; Cosmos-compatible boot data in the BSS
; /------- START OF 4K BOOT DATA
EXPORT _conv_mem                ; conventional memory size (bytes)
        resd 1

EXPORT _ext_mem                 ; extended memory size (bytes)
        resd 1

EXPORT _init_ramdisk_adr        ; where the initial RAM disk is loaded
        resd 1

EXPORT _init_ramdisk_size       ; size of initial RAM disk
        resd 1

EXPORT _krnl_cmd_line           ; kernel command line
        times (1024-4) resd 1   ; padding to 4K
; \------- END OF 4K BOOT DATA

; *** WARNING ***: page tables must be aligned to page (4K) boundaries:
; 1. make sure the linker script aligns the BSS to a page boundary
; 2. make sure the bootloader loads the kernel to a page boundary
; 3. make sure that anything placed before the page tables in the BSS
;    (i.e. the boot data above) is a multiple of 4K in size.

EXPORT task0_page_dir           ; initial page directory
    times 1024 resd 1
EXPORT conv_mem_page_table      ; page table to identity-map <= 4 meg RAM
    times 1024 resd 1

;im_kernel_page_table_1:         ; page tables to identity-map kernel
;    times 1024 resd 1
```

```
;im_kernel_page_table_2:
;    times 1024 resd 1

kernel_page_table_1:            ; page tables to map kernel
    times 1024 resd 1
EXPORT kernel_page_table_2
    times 1024 resd 1

; Create page tables for extended memory (120MB)
EXPORT ext_mem_page_table
    times 1024 resd 30

; Create initial page directory for guest OS or test program
EXPORT task1_page_dir
    times 1024 resd 1

; Create page tables for guest OS or test program (16MB)
EXPORT vir_page_table
    times 1024 resd 4

; task #0 kernel stack
;    resd 1024
;stack:

;EXP _kvirt_to_phys
;    resd 1

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Remember to correct this at some point (ha!)
;[SECTION .dbss bss]

im_kernel_page_table_1:     ; page tables to identity-map kernel
    times 1024 resd 1
im_kernel_page_table_2:
    times 1024 resd 1

    resd 1024
EXPORT stack
```

Bilag B

kernel.c

```
//////////////////////////////////////////////////////////////////////////////
// KERNEL
//
// EXPORTS:
// void panic(const char *fmt, ...);
// void kprintf(const char *fmt, ...);
// void fault(volatile regs_t regs);
// void *kmalloc(unsigned long size);
// void kfree(void *buf);
// int main(void);
//////////////////////////////////////////////////////////////////////////////
#include <_null.h> /* NULL */
#include <_sys.h> /* SYSCALL_INT, SYS_YIELD, SYS_WRITE */
#include <_printf.h> /* do_printf() */
#include <stdarg.h> /* va_list, va_start(), va_end() */
#include <string.h> /* movedata() */
#include <krnl.h> /* console_t */
#include <x86.h> /* disable() */
#include <DMA.h> /* Necessary for floppy.h */
#include <floppy.h> /* load_kernel() */

/* IMPORTS */

/* From start4.asm */
void halt(void);
unsigned long get_page_fault_address(void);
unsigned long get_cr0(void);
unsigned long get_cr3(void);
extern unsigned long _init_ramdisk_adr;
extern unsigned long _conv_mem, _ext_mem;
extern unsigned long stack;

/* From video/console code */
extern console_t _vc[];
extern volatile console_t *_curr_vc;
void sys_putch(console_t *con, unsigned char c);
void init_console(void);

/* From keyboard code */
void kbd_irq(void);
void init_kbd(void);
```

```
int sys_getch(console_t *con);

/* From paging code */
int init_paging(void);
int page_fault(unsigned err_code);
int map_page(unsigned virt, unsigned phys, unsigned priv);
unsigned alloc_this_page(unsigned long addr);
int hndl_prot_fault(regs_t *regs, unsigned long addr, task_t *this_task);

/* From floppy controller code */
void floppy_irq(void);
void init_floppy(void);
void load_testcode(void);

/* From ELF/COFF loader code */
int run(task_t *task, unsigned char *image);

/* From memory manager code */
void* kmalloc(unsigned long size);

/* From task handling code */
int init_tasks(void);
void timer_irq(unsigned long eip);
void schedule(void);
extern volatile task_t *_curr_task;

/* From virtualisation code */
unsigned long instr_handler(regs_t regs, unsigned int instr,
                            unsigned int byte2, unsigned int byte3);
int init_linux(void);
int init_testcode(void);
void init_guest_paging(task_t *task);

/* Import from start4.asm */
//extern segdesc_t gdt[];

/*****************************************************************************
*****************************************************************************/
static int kprintf_help(char c, void **ptr)
{
    ptr++;
    ptr--;
    sys_putch(_curr_vc, c);
    return 0;
}
/*****************************************************************************
*****************************************************************************/
void kprintf(const char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    (void)do_printf(fmt, args, kprintf_help, NULL);
    va_end(args);
}
/*****************************************************************************
*****************************************************************************/
/*static*/
```

```
void panic(const char *fmt, ...)
{
    va_list args;

    disable();
    va_start(args, fmt);
    (void)do_printf(fmt, args, kprintf_help, NULL);
    halt();
} /* panic */
/*****************************************************************************
*****************************************************************************/
static int sys_write(void *bufp, unsigned len)
{
    char *buf = (char *)bufp;
    unsigned pos;

    for(pos = 0; pos < len; pos++)
    {
        sys_putch(_curr_task->vc, *buf);
        buf++;
    }
    return len;
}
/*****************************************************************************
*****************************************************************************/
static void syscall(volatile regs_t *regs)
{
    unsigned long tvirt_to_kvirt;

    tvirt_to_kvirt = (unsigned long)_curr_task->task_mem -
        _curr_task->lva;
    switch(regs->eax)
    {
        case SYS_WRITE:
            regs->eax = sys_write((unsigned char *)regs->ebx +
                tvirt_to_kvirt, regs->ecx);
            break;
        case SYS_YIELD:
            schedule();
            break;
        case SYS_GETCH:
            regs->eax = sys_getch(_curr_task->vc);
            break;
        default:
            panic("\nIllegal syscall 0x%X", regs->eax);
            break;
    }
}

/*****************************************************************************
*****************************************************************************/
/* Get TSS from kstart.asm */
extern tss_t tss1;

task_t* fault(volatile regs_t regs)
{
    unsigned tmp, i;
    unsigned int instr, byte2, byte3;
```

```
    unsigned *page_dir, *page_table, addr;


    static const char *msg[] =
    {
        "divide error", "debug exception", "NMI", "INT3",
        "INTO", "BOUND exception", "invalid opcode", "no coprocessor",
        "double fault", "coprocessor segment overrun",
            "bad TSS", "segment not present",
        "stack fault", "GPF", "page fault", "??",
        "coprocessor error", "alignment check", "??", "??",
        "??", "??", "??", "??",
        "??", "??", "??", "??",
        "??", "??", "??", "??"
    };
    task_t *ret_val;
/* return old value of _curr_task to asm code,
so it can determine if a task-switch occurred */
    ret_val = _curr_task;

//kprintf("fault: %X\n", regs.which_int);
//kprintf("EIP:   %X\n", regs.eip);
//kprintf("Instr: %X\n", peekb(regs.cs, regs.eip));
    switch(regs.which_int)
    {
/* timer (IRQ 0) */
        case 0x20:
            pokeb(LINEAR_SEL, 0xB8000L, peekb(LINEAR_SEL, 0xB8000L) + 1);

            timer_irq(regs.eip);
//if (chup==8) panic("\nStopping system...\n"); chup++;
//            schedule();
/* reset 8259 programmable interrupt controller (PIC) chip */
//            outportb(0x20, 0x20);
/*
kprintf("Delaying...");
DELAY
kprintf("...after delay\n");
*/
            break;
/* Page fault (int 0Eh = 14) */
        case 0x0E:
/*            kprintf("Page fault detected!\n");
            kprintf("Error code...     #%u\n", regs.err_code);
*/
            if ((tmp=page_fault(regs.err_code))) {
                page_dir = (unsigned *)PAGE_DIR_VA;
                addr = get_page_fault_address();
                page_table = (unsigned *)(page_dir[addr>>22] & 0xFFFFF000);
                instr = peekb(regs.cs, regs.eip);
                byte2 = peekb(regs.cs, regs.eip+1);
                byte3 = peekb(regs.cs, regs.eip+2);

                /* Handle access to priviledged memory */
                if (tmp == 1) {
                    if (!hndl_prot_fault(&regs, addr, ret_val))
                        break;
```

```
                }

                kprintf("Page fault detected!\n");
                kprintf("Error code...     #%u\n", regs.err_code);
                kprintf("PF linear address... #%X\n", addr);
                kprintf("page_dir[%d]:      %X\n", addr>>22, page_dir[addr>>22]);
                kprintf("page_table[%d]:    %X\n", ((addr >> 12) & 0x3FF),
                                                   page_table[((addr >> 12) & 0x3FF)]);
                kprintf("EFLAGS:            %8X\n", regs.eflags);
                kprintf("CR0:               %8X\n", get_cr0());
                kprintf("CR3:               %8X", get_cr3());
kprintf("    |    EDI:             %8X\n", regs.edi);
                kprintf("EIP:               %8X", regs.eip);
kprintf("    |    ESI:             %8X\n", regs.esi);
                kprintf("CS:                %8X", regs.cs);
kprintf("    |    EBX:             %8X\n", regs.ebx);
                kprintf("Instr:             %8X", instr);
kprintf("    |    EDX:             %8X\n", regs.edx);
                kprintf("2nd byte:          %8X", byte2);
kprintf("    |    ECX:             %8X\n", regs.ecx);
                kprintf("3rd byte:          %8X", byte3);
kprintf("    |    EAX:             %8X\n", regs.eax);
                panic("System halted...");
            }
            break;
/* keyboard (IRQ 1) */
        case 0x21:
            kbd_irq();
            outportb(0x20, 0x20);
            break;
/* Floppy disk (IRQ 6) */
        case 0x26:
            floppy_irq();
            /* Send EOI to PIC 1 (resets IRQ) */
            outportb(0x20, 0x20);
            break;
/* GPF (int 0Dh = 13) */
       case 0x0D:
            kprintf("General Protection Fault... #%u\n", regs.which_int);
            kprintf("Error code... #%u\n", regs.err_code);

/* EIP is a pointer to the instruction being executed, it does not contain the instruction */
instr = peekb(regs.cs, regs.eip);
byte2 = peekb(regs.cs, regs.eip+1);
byte3 = peekb(regs.cs, regs.eip+2);
kprintf("EIP:      %X\n", regs.eip);
kprintf("CS:       %X\n", regs.cs);
kprintf("EFLAGS:   %X\n", regs.eflags);
kprintf("CR0:      %X\n", get_cr0());
kprintf("CR3:      %X\n", get_cr3());
kprintf("Instr:    %X\n", instr);
kprintf("2nd byte: %X\n", byte2);
kprintf("3rd byte: %X\n", byte3);

    /* Call instruction handler */
    tmp = instr_handler(regs, instr, byte2, byte3);
    kprintf("tmp: %x\n", tmp);
    if (tmp) {
```

```
        /* Should never happen */
        if (tmp == (unsigned)-1) {
            kprintf("Oh, oh!\n");
            panic("System halted");
        }
        regs.eip += tmp;
        break;
    }
            panic("\nSystem halted");
            break;
/* IOE (int 06h = 06) */
       case 0x06:
            kprintf("Invalid Opcode Exception... #%u\n", regs.which_int);
            kprintf("Error code... #%u\n", regs.err_code);
instr = peekb(regs.cs, regs.eip);
byte2 = peekb(regs.cs, regs.eip+1);
byte3 = peekb(regs.cs, regs.eip+2);
kprintf("EIP:      %X\n", regs.eip);
kprintf("CS:       %X\n", regs.cs);
kprintf("EFLAGS:   %X\n", regs.eflags);
kprintf("Instr:    %X\n", instr);
kprintf("2nd byte: %X\n", byte2);
kprintf("3rd byte: %X\n", byte3);

            panic("\nSystem halted");
            break;

/* int 30h (syscall) */
        case SYSCALL_INT:
            syscall(&regs);
            break;
/* anything else */
        default:
            kprintf("\nUnhandled kernel exception #%u ",
                regs.which_int);
            if(regs.which_int <= sizeof(msg) / sizeof(char *))
                kprintf("(%s) ", msg[regs.which_int]);
            /* Dump instruction opcode etc. */
            kprintf("\nEIP:    %X\n", regs.eip);
            kprintf("CS:     %X\n", regs.cs);
            kprintf("EFLAGS: %X\n", regs.eflags);
            kprintf("Inst:   %X\n", peekb(regs.cs, regs.eip));
            panic("\nSystem halted");
            break;
    }

    return ret_val;
}
/*****************************************************************************
*****************************************************************************/
static void init_8259s(void)
{
    outportb(0x20, 0x11); /* ICW1 */
    outportb(0xA0, 0x11);

    outportb(0x21, 0x20); /* ICW2: route IRQs 0...7 to INTs 20h...27h */
    outportb(0xA1, 0x28); /* ...IRQs 8...15 to INTs 28h...2Fh */
```

```
    outportb(0x21, 0x04); /* ICW3 */
    outportb(0xA1, 0x02);

    outportb(0x21, 0x01); /* ICW4 */
    outportb(0xA1, 0x01);

    /* enable IRQ1 (keyboard) and IRQ6 (floppy disk) */
    outportb(0x21, ~0x42);
    outportb(0xA1, ~0x00);
}

static void init_8253(void)
{
/* I can remember the TV color burst frequency, but not the PC
peripheral clock. Fortunately, they are related: */
    static const unsigned short foo = (3579545L / 3) / HZ;

/* reprogram the 8253 timer chip to run at 'HZ', instead of 18 Hz */
    outportb(0x43, 0x36);   /* channel 0, LSB/MSB, mode 3, binary */
    outportb(0x40, foo & 0xFF); /* LSB */
    outportb(0x40, foo >> 8);   /* MSB */
}

/*****************************************************************************
for MinGW32
*****************************************************************************/
#ifdef __WIN32__
int __main(void) { return 0; }
#endif
/*****************************************************************************
*****************************************************************************/
int main(void)
{
    int temp, i;

    init_console();
    kprintf("Welcome to OSD\n"
        "press F1, F2, etc. to select virtual console\n");
    init_kbd();
    /* Program interrupts */
    init_8259s();

    init_paging();

    enable();
    /* Load Linux kernel from disk */
//    load_kernel();
    load_testcode();
    disable();

    /* Attempting to initialise Linux bootloader */
//    temp = init_linux();
//    temp = init_tasks();
    temp = init_testcode();

    if(temp != 0)
        kprintf("sorry, did not find any tasks to run\n");
/* enabling timer interrupt starts the scheduler */
```

```
    else {
        /* Starting timer (IRQ 0) */
        outportb(0x21, (inportb(0x21) & 0xFE));
        /* Reprogam timer */
        init_8253();
kprintf("Enabling timer interrupts\n");
        enable();
    }

/* freeze */
    while(1)
        halt();
    return 0;
}
```

Bilag C

tasks.c

```
/*****************************************************************************
TASKS

EXPORTS:
int init_tasks(void);
void timer_irq(unsigned long eip);
void schedule(void);
void set_descriptor_base(unsigned char *desc, unsigned long base);
void set_descriptor_limit(unsigned char *desc, unsigned long limit);
volatile task_t *_curr_task;
task_t _tasks[];
*****************************************************************************/
#include <string.h>
#include <krnl.h>
#include <x86.h>

/* IMPORTS */

/* Import from assembly code */
extern unsigned long _virt_to_phys;
extern unsigned char _gdt5[], _gdt6[];

/* From video code */
extern console_t _vc[MAX_VC];

/* now we use the task_t struct defined in os.h */
volatile task_t *_curr_task;
task_t _tasks[MAX_TASK];

int init_tasks(void)
{
    unsigned short num_objs, t, o;
    task_t *task;

    kprintf("init_tasks:\n");
/* MUST do this: */
    _curr_task = _tasks + 0;
/* the number of os's is currently one + main screen */
    num_objs = 2;
    t = 1;
    for(o = 1; o < num_objs; o++)
```

```
    {
/* don't run more tasks than we have virtual consoles */
        if(t >= MAX_VC)
            break;
        kprintf("obj %02u -> task %02u,  ", o, t);
        task = _tasks + t;
/* set other task stuff */
        task->status = TS_RUNNABLE;
        task->vc = _vc + t;
        t++;
    }
    kprintf("\n");
/* return nonzero if no tasks were loaded */
    return (t == 1) ? -1 : 0;
} /* init_tasks */

void wake_up(wait_queue_t *queue)
{
    task_t *task, *next;

/* make sure queue is not empty */
    task = queue->head;
    if(task == NULL)
        return;
/* mark head task in queue runnable */
    task->status = TS_RUNNABLE;
/* remove head task from queue */
    next = task->next;
    queue->head = next;
    if(next != NULL)
        next->prev = NULL;
    else
        queue->tail = NULL;
} /* wake_up */


int sleep_on(wait_queue_t *queue, unsigned long *timeout)
{
    int ret_val = 0;
    task_t *prev;

/* mark task blocked */
    _curr_task = (task_t *)_curr_task;
    _curr_task->status = TS_BLOCKED;
/* splice into wait queue at queue->tail */
    prev = queue->tail;
    queue->tail = _curr_task;
    if(prev == NULL)
    {
        queue->head = _curr_task;
        _curr_task->prev = NULL;
    }
    else
    {
        _curr_task->prev = prev;
        prev->next = _curr_task;
    }
    _curr_task->next = NULL;
```

```
/* set the timeout, if there is one */
    if(timeout != NULL)
        _curr_task->timeout = *timeout;
    else
        _curr_task->timeout = 0;
/* "fake" a timer interrupt. timer_irq() must detect this */
    __asm__ __volatile__("int $0x20");
/* now: why did we return? */
    if(timeout != NULL)
    {
        *timeout = _curr_task->timeout;
/* there was a timeout, so timer_irq() awoke us. Return -1 */
        if(*timeout == 0)
            ret_val = -1;
    }
/* someone called wake_up(), making us TS_RUNNABLE again. Return 0 */
    return ret_val;
} /* sleep_on */
/*****************************************************************************
*****************************************************************************/
/*static*/
void set_descriptor_base(unsigned char *desc, unsigned long base)
{
    *(unsigned short *)(desc + 2) = base;
    base >>= 16;
    *(unsigned char *)(desc + 4) = base;
    base >>= 8;
    *(unsigned char *)(desc + 7) = base;
}
/*****************************************************************************
*****************************************************************************/
/*static*/
void set_descriptor_limit(unsigned char *desc, unsigned long limit)
{
    unsigned char temp;

    *(unsigned short *)(desc + 0) = limit;
    limit >>= 16;
    temp = *(unsigned char *)(desc + 6);
    temp = (temp & 0xF0) | (limit & 0x0F);
    *(unsigned char *)(desc + 6) = temp;
}
/*****************************************************************************
*****************************************************************************/
/* static */
void schedule(void)
{
    static unsigned current;
    unsigned long tvirt_to_kvirt, tvirt_to_phys;

    do
    {
        current++;
        if(current >= MAX_TASK)
            current = 0;
        _curr_task = _tasks + current;
    } while(_curr_task->status != TS_RUNNABLE);
/* */
```

```
    tvirt_to_kvirt = (unsigned long)_curr_task->task_mem -
        _curr_task->lva;
    tvirt_to_phys = tvirt_to_kvirt + _virt_to_phys;
    set_descriptor_base(_gdt5, tvirt_to_phys);
    set_descriptor_base(_gdt6, tvirt_to_phys);
    set_descriptor_limit(_gdt5, _curr_task->size - 1);
    set_descriptor_limit(_gdt6, _curr_task->size - 1);
}

static void schedule2(void)
{
    static unsigned current;
    unsigned long tvirt_to_phys;
    regs_t *regs;

    do
    {
        current++;
        if(current >= MAX_TASK)
            current = 0;
        _curr_task = _tasks + current;
    } while(_curr_task->status != TS_RUNNABLE);

    tvirt_to_phys = (unsigned long)_curr_task->task_mem + _virt_to_phys;
    set_descriptor_base(_gdt5, tvirt_to_phys);
    set_descriptor_base(_gdt6, tvirt_to_phys);
    set_descriptor_limit(_gdt5, _curr_task->size - 1);
    set_descriptor_limit(_gdt6, _curr_task->size - 1);
} /* schedule2 */

void timer_irq(unsigned long eip)
{
//    unsigned long new_time;
    unsigned char *instr;
//    unsigned short i;

/* was it a real timer interrupt? */
//  if(inportb(0x20) & 0x01)
    instr = (unsigned char *)eip - 2;
    if(*instr != 0xCD) /* 0xCD = 11001101 = 'Int' instruction (software int) */
    {
/* decrement timeouts for tasks that have them *
        for(i = 0; i < MAX_TASK; i++)
        {
            new_time = _tasks[i].timeout;
            if(new_time == 0)
                continue;
* number of microseconds per timer IRQ *
            new_time -= (1000000L / HZ);
            if(new_time > _tasks[i].timeout)
                new_time = 0;   * underflow *
            _tasks[i].timeout = new_time;
* timeout? wake up task *
            if(new_time == 0 && _tasks[i].status == TS_BLOCKED)
                _tasks[i].status = TS_RUNNABLE;
        }
        outportb(0x20, 0x20); * reset 8259 chip */
    }
```

```
    outportb(0x20, 0x20);

    /* run the scheduler, whether the interrupt was hardware (IRQ 0) or software (INT 20h) */
    schedule2();
} /* timer_irq */
/*****************************************************************************
*****************************************************************************/
```

Bilag D

paging.c

```
/*****************************************************************************
EXPORTS:
int init_paging(void);
int map_page(unsigned virt, unsigned phys, unsigned priv);
void discard_mem(void);
unsigned alloc_page(void);
int free_page(unsigned phys_adr);
void *kbrk(int incr);
void free_task_pages(unsigned *page_dir);
int page_fault(unsigned err_code);
*****************************************************************************/
#include <string.h>
#include <krnl.h>
#include <x86.h>

/* IMPORTS
from kernel linker script file */
extern unsigned char _code[], _d_code[], _data[];
extern unsigned char _d_data[], _bss[], _d_bss[], _end[];

/* from KERNEL.C */
extern task_t *_curr_task;

/* from KSTART.ASM */
extern unsigned _conv_mem, _ext_mem, ext_mem_page_table;
extern unsigned _virt_to_phys, _init_ramdisk_adr;//, _init_ramdisk_size;

/* From VIDEO.C */
extern unsigned long _vga_fb_adr;


unsigned get_page_fault_address(void);
unsigned get_page_dir(void);
void set_page_dir(unsigned cr3);

/* the 12 bits at the bottom of a page directory/table entry: */
#define PRIV_PRESENT    0x001
#define PRIV_WRITABLE   0x002
#define PRIV_USER       0x004
/* b3, b4 used for cache control on 486+ */
//efine PRIV_ACCESSED   0x020
```

```
//efine PRIV_DIRTY      0x040
/* b7, b8 reserved */
//efine PRIV_USER2      0x200       /* user-defined */
//efine PRIV_USER4      0x400
//efine PRIV_USER8      0x800
//efine PRIV_COW        PRIV_USER2  /* copy-on-write */
//efine PRIV_ALL        0xFFF

/*   8192 =  32 meg RAM exactly */
/*  32768 = 128 meg RAM exactly */
/* 131072 = 512 meg RAM exactly */
#define MAX 32768

static unsigned char _page_use_count[MAX];

int init_paging(void) {
    unsigned start, end, i, j;
    unsigned *page_dir, dir_ent, *page_table, ptr;

/* mark page 0 in use */
//    _page_use_count[0] = (unsigned char)-1;
    _page_use_count[0] = 0x0;
/* mark conventional memory used by RDSK file in use */
/* !!! Not sure whether this is necessary */
/* !!! Remember to remove ramdisk stuff from makefile */
    start = _init_ramdisk_adr / PAGE_SIZE;
    end = (_init_ramdisk_adr + 0x11000) / PAGE_SIZE;
    for(i = start; i < end; i++)
        _page_use_count[i] = (unsigned char)-1;
/* mark non-existent conventional memory and adapter memory in use */
//    start = _conv_mem / PAGE_SIZE;
    start = 0xA0000 / PAGE_SIZE; /* 0xA0000 = 640KB */
    end = 0x100000L / PAGE_SIZE;
    for(i = start; i < end; i++)
        _page_use_count[i] = (unsigned char)-1;
/* mark kernel physical memory in use */
    start = ((unsigned)_code + _virt_to_phys) / PAGE_SIZE;
    end = ((unsigned)_end + _virt_to_phys) / PAGE_SIZE;
    for(i = start; i < end; i++)
        _page_use_count[i] = (unsigned char)-1;
/* mark non-existent extended memory in use */
//    start = (0x100000L + _ext_mem) / PAGE_SIZE;
    start = (0x100000L + 0x6000000) / PAGE_SIZE; /* 0x6000000 = 96MB */
    end = MAX;
    for(i = start; i < end; i++)
        _page_use_count[i] = (unsigned char)-1;

    /* Initialise next 120MB of memory */
    page_dir = (unsigned *)PAGE_DIR_VA;
    dir_ent = (unsigned)&ext_mem_page_table;
    for (i=2; i<32; i++) {
        /* Set page info */
        dir_ent |= 0x07;
        /* Insert page entry */
        page_dir[i] = dir_ent;
        /* Next page entry */
        dir_ent += 0x1000;
    }
```

```
    /* Identity map linear address space */
    /* First two indices are mapped in kstart.asm */
    for (i=2; i<32; i++) {
        page_table = (unsigned *)page_dir[i];
        /* ptr contains the physical 4KB-aligned address to be mapped */
        /* Starts at 0x0 because of identity mapping (0x400000 = 4MB) */
        ptr = (unsigned)(0x0 + i*0x400000);
        /* 0x400 = 1KB, 0x1000 = 4KB */
        for (j=0x0; j<0x400; j++)  {
            /* Set page address */
            page_table[j] = ptr + j*0x1000;
            /* Set page info (user, r/w, present) */
            page_table[j] |= 0x07;
        }
    }

    /* Take a peek *
    page_table = (unsigned *)PAGE_TABLES_VA;
    kprintf("page_dir:          %X\n", page_dir);
    kprintf("page_dir[0]:       %X\n", page_dir[0]);
    kprintf("page_table:        %X\n", page_table);
    kprintf("page_table[0]:     %X\n", page_table[0]);
    kprintf("page_table[1]:     %X\n", page_table[1]);
    kprintf("page_table[2]:     %X\n", page_table[2]);
    kprintf("page_table[150]:   %X\n", page_table[150]);
    kprintf("page_table[588]:   %X\n", page_table[588]);
    kprintf("page_table[589]:   %X\n", page_table[589]);
    kprintf("page_table[590]:   %X\n", page_table[590]);
    kprintf("page_table[591]:   %X\n", page_table[591]);
    page_table = (unsigned *)0x118000;
    kprintf("A check:           %X\n", page_table[3]);
for(;;);
*/
    return 0;
} /* init_paging */
/******************************************************************************
******************************************************************************/
int map_page(unsigned virt, unsigned phys, unsigned priv)
{
    unsigned *page_dir, d, dir_ent, dt, *page_tables;

    /* Get pointer to page directory table */
    page_dir = (unsigned *)PAGE_DIR_VA;
    d = virt >> 22;
    dir_ent = page_dir[d];

    if((dir_ent & PRIV_PRESENT) == 0) {
        kprintf("map_page(): missing page table while mapping 0x%X -> 0x%X\n", virt, phys);
        return -1;
    }

    /* Get pointer to relevant page table */
    page_tables = (unsigned *)PAGE_TABLES_VA;
    dt = virt >> 12;
/* store mapping */
    phys &= -PAGE_SIZE;
    priv &= (PAGE_SIZE - 1);
```

```
    page_tables[dt] = phys | priv;
    return 0;
}
/*****************************************************************************
*****************************************************************************/
static int unmap_mem(unsigned virt, unsigned len)
{
    unsigned tvirt;

    for(tvirt = virt; tvirt < virt + len; tvirt += PAGE_SIZE)
    {
        if(map_page(tvirt, 0, /* priv== */0) != 0)
            return -1;
    }
    return 0;
}
/*****************************************************************************
allocates one page (4K) of memory
returns PHYSICAL address or NULL if out of memory
*****************************************************************************/
unsigned alloc_page(void)
{
    unsigned i;

/* skip page 0; start with 1 */
    for(i = 1; i < MAX; i++)
    {
        if(_page_use_count[i] == 0)
        {
            _page_use_count[i] = 1;
            return i * PAGE_SIZE;
        }
    }

    return NULL;
}
/*****************************************************************************
allocates one given page (4K) of memory
returns PHYSICAL address or NULL if out of memory
*****************************************************************************/
unsigned alloc_this_page(unsigned long addr)
{
    unsigned index;

    /* Convert addr into page index */
    index = addr / PAGE_SIZE;
    /* Check page */
    if(_page_use_count[index] == 0) {
        _page_use_count[index] = 1;
        return addr;
    }

//    for (index=0; index<600; index++) {
//        if (_page_use_count[index]==0)
//            kprintf("0");
//        else
//           kprintf("1");
//    }
```

```
//    kprintf("\n");
//    for(;;);

    return NULL;
} /* alloc_this_page */
/*****************************************************************************
*****************************************************************************/
int free_page(unsigned phys_adr)
{
    unsigned i;

    i = phys_adr >> 12;
    if(i >= MAX)
    {
        kprintf("free_page: bad page address 0x%X\n", phys_adr);
        return -1;
    }
    if(_page_use_count[i] == 0)
    {
        kprintf("free_page: trying to free already-free page 0x%X\n",
            phys_adr);
        return -1;
    }
    (_page_use_count[i])--;
    return 0;
}
/*****************************************************************************
*****************************************************************************/
void free_pages(unsigned start_phys_adr, unsigned size)
{
    for(; size != 0; size -= PAGE_SIZE)
    {
        if(free_page(start_phys_adr) != 0)
            break;
        start_phys_adr += PAGE_SIZE;
    }
}
/*****************************************************************************
*****************************************************************************/
/*unsigned alloc_page(void) {
    unsigned i;

* skip page 0; start with 1 *
    for(i = 1; i < MAX; i++) {
        if(_page_use_count[i] == 0) {
            _page_use_count[i] = 1;
            return i * PAGE_SIZE;
        }
    }
    return NULL;
} * alloc_page */

/*unsigned alloc_pages(unsigned size) {
    for (; size > 0; size -= PAGE_SIZE) {
        if (alloc_page() == 0)
            break;
        start_phys_adr += PAGE_SIZE;
    }
```

```
    return NULL;
} * alloc_pages */
/*****************************************************************************
converts fault_adr to index-into-page-directory,
checks if page table installed there,
installs one if necessary
returns VIRTUAL address of page table
*****************************************************************************/
static unsigned* get_page_table(unsigned *dir_virt, unsigned fault_adr)
{
    unsigned tab_phys, *tab_virt, de;

/* get page directory entry (PDE)
The PDE contains the physical address of the page table */
    de = (fault_adr >> 22) & 0x3FF;
    tab_phys = dir_virt[de];
/* the directory entry number can also be converted into the
VIRTUAL address of the page table */
    tab_virt = (unsigned *)(PAGE_TABLES_VA | (de << 12));
/* if no page table here, create one */
    if(tab_phys == 0) /* ### - maybe ((tab_phys & PRIV_PRESENT) == 0) */
    {
        tab_phys = alloc_page();
        if(tab_phys == NULL)
        {
            kprintf("get_page_table: out of memory\n");
            return NULL;
        }
/* graft new page table into page directory
For the underlying pages to be PRIV_USER, the page table must be PRIV_USER
### - set PRIV_ACCESSED and PRIV_DIRTY as well? to prevent TLB miss? */
        dir_virt[de] = tab_phys | PRIV_PRESENT |
            PRIV_WRITABLE | PRIV_USER;
        memset(tab_virt, 0, PAGE_SIZE);
    }
    return tab_virt;
}
/*****************************************************************************
We need no invalidates here. As the source code to Linux says,
"no need to invalidate: a not-present page shouldn't be cached"
*****************************************************************************/
static int no_page(unsigned fault_adr, unsigned err_code)
{
    unsigned new_page_phys = 0, *dir_virt, *tab_virt, priv, te;

    priv = PRIV_PRESENT;
    if(err_code & 0x02)
        priv |= PRIV_WRITABLE;
    priv |= PRIV_USER;
/* alloc page and check for out-of-memory */
    new_page_phys = alloc_this_page(fault_adr);

    if(new_page_phys == 0)
    {
        kprintf("Out of memory at ");
        kprintf("fault_adr: %X\n");
        return -1;
    }
```

```
    dir_virt = (unsigned *)PAGE_DIR_VA;
/* get page directory entry (PDE),
which contains the physical address of the page table */
    tab_virt = get_page_table(dir_virt, fault_adr);
    if(tab_virt == NULL)
    {
        kprintf("Oops, missing page table\n");
        return -1;
    }
/* set page table entry (PTE) to
PHYSICAL address of new page and correct privilege */
    te = (fault_adr >> 12) & 0x3FF;
/* even if the page is read-only, make it writable so we can load it */
//    tab_virt[te] = new_page_phys | (PRIV_USER | PRIV_WRITABLE | PRIV_PRESENT);

/* NOW make the page read-only, if necessary */
    tab_virt[te] = new_page_phys | priv;
    kprintf("\n");
    return 0;
}
/*****************************************************************************
*****************************************************************************/
int page_fault(unsigned err_code)
{
    unsigned fault_adr;
    unsigned *page_dir, *page_table;

    fault_adr = get_page_fault_address();
/* PAGE NOT PRESENT (write or read) */
    if((err_code & 1) == 0) {
        kprintf("No page! ");
        return no_page(fault_adr, err_code);
    }
/* PRIVILEGE VIOLATION */
/* let fault() kill the task */
    if(err_code & 4)
    {
        kprintf("page_fault: attempt to access privileged "
            "memory at 0x%X\n", fault_adr);
        return 1;
    }
/* WRITE TO READ-ONLY PAGE */
/* let fault() kill the task or panic *
    else if(err_code & 2) */
    return -1;
}
/*****************************************************************************
*****************************************************************************/
void grafx_handler(console_t *vir_con, unsigned long mem_addr, unsigned char value) {
    /* Is protected memory video RAM? */
    if (mem_addr>0xA0000 && mem_addr<0xC0000) {
        mem_addr = mem_addr - 0xA0000;
        vir_con->vid_mem[mem_addr] = value;
        pokew(LINEAR_SEL, vir_con->fb_adr + mem_addr, value);
    }
} /* grafx_handler */

int hndl_prot_fault(regs_t *regs, unsigned long addr, task_t *this_task) {
```

```
    unsigned long mem_addr = 0;
    unsigned char instr[8], value;
    console_t *vir_con = this_task->vc;
    int i;

    /* Fetch instructions */
    for (i=0; i<8; i++)
        instr[i] = peekb(regs->cs, regs->eip + i);

    switch (instr[0]) {
        /* mov [mem], al (5 bytes) */
        case 0xA2:
        /* mov [mem], eax (5 bytes) */
        case 0xA3:
            mem_addr = (instr[4]<<24) + (instr[3]<<16) + (instr[2]<<8) + instr[1];
            value = (unsigned char)regs->eax;
            grafx_handler(vir_con, mem_addr, value);
            regs->eip += 5;
            break;
        /* mov r/m8, imm8 */
        case 0xC6:
            /* Check ModR/M byte */
            switch (instr[1]) {
                case 0x00:  /* eax */
                case 0x01:  /* ecx */
                case 0x02:  /* edx */
                case 0x03:  /* ebx */
                    goto LABEL0;
                case 0x04:
                    kprintf("Unhandled ModR/M. Contains SIB byte\n");
                    kprintf("ModR/M:    %X\n", instr[1]);
                    kprintf("SIB:       %X\n", instr[2]);
                    return 3;
                case 0x05:
                    mem_addr = (instr[5]<<24) + (instr[4]<<16) + (instr[3]<<8) + instr[2];
                    value = instr[6];
                    grafx_handler(vir_con, mem_addr, value);
                    regs->eip += 7;
                    break;
                case 0x06:  /* esi */
                case 0x07:  /* edi */
LABEL0:
                    mem_addr = ((unsigned long *)regs)[7-instr[1]];
                    value = instr[2];
                    grafx_handler(vir_con, mem_addr, value);
                    regs->eip += 3;
                    break;
                default:
                    kprintf("Unknown ModR/M value in MOV instruction\n");
                    kprintf("ModR/M:    %X\n", instr[1]);
                    return 2;
            }
            break;
        default:
            kprintf("Unhandled instruction in hndl_prot_fault\n");
            return 1;
    }
```

```
    return 0;
} /* hndl_prot_fault */
```

Bilag E

mm.c

```
/*****************************************************************************
MEMORY MANAGER

EXPORTS:
void* kmalloc(unsigned long size);
*****************************************************************************/
#include <string.h>
#include <krnl.h>

/* From linker script file */
extern unsigned char end[];

/* From paging code */
unsigned alloc_this_page(unsigned long addr);
int map_page(unsigned virt, unsigned phys, unsigned priv);

void* kmalloc(unsigned long size)
{
    static void *brk;
    void *ret_val;
    unsigned virt, phys, err;

    if (brk == NULL) {
        brk = end;
//        kprintf("Initialised kernel heap\n");
    }
    ret_val = brk;

    /* Round size up to nearest multiple of PAGE_SIZE */
    if(size > 0)
        size += (PAGE_SIZE - 1);
    size &= -PAGE_SIZE;

    /* Not necessary at this point to be able to reduce allocated memory *
    if (size < 0)
        return brk;*/

    /* alloc_page returns the physical page address. */
    /* I am assuming it is the same as the address currently contained within brk. */
    /* !!! - Remember to check */
    for (virt=(unsigned)brk; virt<(unsigned)brk + size; virt += PAGE_SIZE) {
```

```
        phys = alloc_this_page(virt);

        if(phys == NULL)
        {
            kprintf("kmalloc - Error: out of memory, or memory fragmented\n");
            kprintf("          phys.............. #%X\n", phys);
            kprintf("          virt.............. #%X\n", virt);
            kprintf("          brk............... #%X\n", brk);
            kprintf("          ret_val........... #%X\n", ret_val);
            kprintf("          Difference........ #%X\n", virt-(unsigned)ret_val);
            return ret_val;
        }

        /* Create page map. Present, writable and supervisor level */
        err = map_page(virt, phys, 0x03);
        if(err)
        {
            kprintf("kbrk() - error\n");
            return ret_val;
        }
    }

    brk = (unsigned char *)brk + size;
    return ret_val;
} /* kmalloc */
/*****************************************************************************
*****************************************************************************/
void kfree(void *buf)
{
    buf++;
    buf--;
}
/*****************************************************************************
*****************************************************************************/
```

Bilag F

virtual.c

```
/****************************************************************************
VIRTUALISER

EXPORTS:
unsigned long instr_handler(regs_t regs, unsigned int instr, unsigned int byte2,
                            unsigned int byte3);
int init_linux(void);
****************************************************************************/
#include <string.h>
#include <krnl.h>
#include <x86.h>

/* IMPORTS */

/* Import from assembly code */
extern unsigned long _virt_to_phys;
extern unsigned char _gdt5[], _gdt6[];
unsigned long get_page_fault_address(void);
unsigned long get_cr3(void);

/* Import from task code */
extern volatile task_t *_curr_task;
extern task_t _tasks[MAX_TASK];
extern void set_descriptor_base(unsigned char *desc, unsigned long base);
extern void set_descriptor_limit(unsigned char *desc, unsigned long limit);

/* From VIDEO.C */
extern console_t _vc[MAX_VC];

/* Declare segment descriptor type */
typedef struct Seg_Desc {
    unsigned short lowlimit;
    unsigned short lowbase;
    unsigned int   midbase : 8;
    unsigned int   type : 4;        /* Segment type */
    unsigned int   S : 1;           /* Descriptor type (0=system, 1=code or data) */
    unsigned int   DPL : 2;         /* Descriptor Privilege Level */
    unsigned int   P : 1;           /* Segment present in memory (0=no, 1=yes) */
    unsigned int   highlimit : 4;
    unsigned int   AVL : 1;         /* Available for use by system software (not used) */
    unsigned int   X : 1;           /* Not used always zero */
```

```
    unsigned int   DB : 1;          /* Default operation size (0=16-bit, 1=32-bit) */
    unsigned int   G : 1;           /* Granularity defines meaning of */
                                    /* limit value (0=bytes, 1=pages) */
    unsigned int   highbase : 8;
} segdesc_t;

/* Create GDT */
segdesc_t GDT[7];

/* Create guest OS TSS (add [] later) */
tss_t guest_tss;

/*********************************************************************
**********************************************************************/
void check_mov(regs_t regs, unsigned int byte2) {
//    unsigned char byte2;
    unsigned int cmd1, cmd2;

//    byte2 = peekb(regs.cs, regs.eip+1);
    kprintf("2nd byte: %X\n", byte2);

    /* Is data being copied from register? */

    if ((byte2 & 0xC0) == 0xC0) { /* C0h = 11000000 */
        cmd1 = byte2 & 0x38; /* 38h = 00111000 */
        cmd2 = byte2 & 0x07; /* 07h = 00000111 */

        switch (cmd1) {
            case 0x00:  // ES
                if (cmd2 == 0x00) {
                    guest_tss.es = regs.eax;
                }
                else if (cmd2 == 0x01) {
                    guest_tss.es = regs.ecx;
                }
                else if (cmd2 == 0x02) {
                    guest_tss.es = regs.edx;
                }
                else if (cmd2 == 0x03) {
                    guest_tss.es = regs.ebx;
                }
                else {
                    kprintf("Unhandled case!\n");
                    panic("System halted...\n");
                };
                break;
            case 0x18:  // DS, 18h = 00011000
kprintf("Should be here... for DS\n");
                if (cmd2 == 0x00) {
kprintf("And then here... for EAX\n");
                    guest_tss.ds = regs.eax;
                }
                else if (cmd2 == 0x01) {
                    guest_tss.ds = regs.ecx;
                }
                else if (cmd2 == 0x02) {
                    guest_tss.ds = regs.edx;
                }
```

```
                else if (cmd2 == 0x03) {
                    guest_tss.ds = regs.ebx;
                }
                else {
                    kprintf("Unhandled case!\n");
                    panic("System halted...\n");
                };              break;
            case 0x20:  // FS, 20h = 00100000
                if (cmd2 == 0x00) {
                    guest_tss.fs = regs.eax;
                }
                else if (cmd2 == 0x01) {
                    guest_tss.fs = regs.ecx;
                }
                else if (cmd2 == 0x02) {
                    guest_tss.fs = regs.edx;
                }
                else if (cmd2 == 0x03) {
                    guest_tss.fs = regs.ebx;
                }
                else {
                    kprintf("Unhandled case!\n");
                    panic("System halted...\n");
                };              break;
            case 0x28:  // GS, 28h = 00101000
                if (cmd2 == 0x00) {
                    guest_tss.gs = regs.eax;
                }
                else if (cmd2 == 0x01) {
                    guest_tss.gs = regs.ecx;
                }
                else if (cmd2 == 0x02) {
                    guest_tss.gs = regs.edx;
                }
                else if (cmd2 == 0x03) {
                    guest_tss.gs = regs.ebx;
                }
                else {
                    kprintf("Unhandled case!\n");
                    panic("System halted...\n");
                };
                break;
            default:
                kprintf("Unhandled case!\n");
                panic("System halted...\n");
        };
    }
    /* Not from register */
    else {
        kprintf("Mod not set!\n");
        panic("System halted...\n");
    }
} /* check_mov */

void check_mov2(regs_t regs) {
    unsigned int byte2;
    unsigned int cmd1, cmd2;
```

```
    byte2 = peekb(regs.cs, regs.eip+1);
    kprintf("2nd byte: %X\n", byte2);

    /* Is data being copied from register? */

    if ((byte2 && 0xC0) == 0xC0) {
        cmd1 = byte2 & 0x38; /* 38h = 00111000 */
        cmd2 = byte2 & 0x07; /* 07h = 00000111 */
        switch (cmd1) {
            case 0x00:  /* ES */
                switch (cmd2) {
                    case 0x00:  /* EAX */
                        guest_tss.es = regs.eax;
                        break;
                    case 0x01:  /* ECX */
                        guest_tss.es = regs.ecx;
                        break;
                    case 0x02:  /* EDX */
                        guest_tss.es = regs.edx;
                        break;
                    case 0x03:  /* EBX */
                        guest_tss.es = regs.ebx;
                        break;
                    default:
                        kprintf("Unhandled case!\n");
                        panic("System halted...\n");
                };
                break;
            case 0x18:  /* DS */
kprintf("Should be here...\n");
                switch (cmd2) {
                    case 0x00:  /* EAX */
kprintf("And here...\n");
                        guest_tss.ds = regs.eax;
                        break;
                    case 0x01:  /* ECX */
                        guest_tss.ds = regs.ecx;
                        break;
                    case 0x02:  /* EDX */
                        guest_tss.ds = regs.edx;
                        break;
                    case 0x03:  /* EBX */
                        guest_tss.ds = regs.ebx;
                        break;
                    default:
                        kprintf("Unhandled case!\n");
                        panic("System halted...\n");
                };
                break;
            case 0x20:  /* FS */
                switch (cmd2) {
                    case 0x00:  /* EAX */
                        guest_tss.fs = regs.eax;
                        break;
                    case 0x01:  /* ECX */
                        guest_tss.fs = regs.ecx;
                        break;
                    case 0x02:  /* EDX */
```

```
                        guest_tss.fs = regs.edx;
                        break;
                    case 0x03:  /* EBX */
                        guest_tss.fs = regs.ebx;
                        break;
                    default:
                        kprintf("Unhandled case!\n");
                        panic("System halted...\n");
                };
                break;
            case 0x28:  /* GS */
                switch (cmd2) {
                    case 0x00:  /* EAX */
                        guest_tss.gs = regs.eax;
                        break;
                    case 0x01:  /* ECX */
                        guest_tss.gs = regs.ecx;
                        break;
                    case 0x02:  /* EDX */
                        guest_tss.gs = regs.edx;
                        break;
                    case 0x03:  /* EBX */
                        guest_tss.gs = regs.ebx;
                        break;
                    default:
                        kprintf("Unhandled case!\n");
                        panic("System halted...\n");
                };
                break;
            default:
                kprintf("Unhandled case!\n");
                panic("System halted...\n");
        };
    }
    /* Not from register */
    else {
        kprintf("Unhandled case!\n");
        panic("System halted...\n");
    }
} /* check_mov */

unsigned long instr_handler(regs_t regs, unsigned int instr, unsigned int byte2,
                            unsigned int byte3) {
    unsigned long ptr;

    switch (instr) {
        case 0xFA:  /* cli (1 byte) */
            return 1;
            break;
        case 0x8E:  /* Mov reg/mem to seg reg (2 bytes) */
kprintf("We aught to be here...\n");
//            byte2 = peekb(regs.cs, regs.eip+1);
            check_mov(regs, byte2);
            return 2;
            break;
        case 0x0F:  /* LSS, LFS or LGS */
//            byte2 = peekb(regs.cs, regs.eip+1);
            switch (byte2) {
```

```
                case 0xB2:  /* LSS */
                    if (byte3 == 0x25) {
                        ptr = peekl(regs.cs, regs.eip+3);
                        guest_tss.ss = peekw(regs.cs, ptr+4);
                        guest_tss.esp0 = peekl(regs.cs, ptr);
                        regs.esp = guest_tss.esp0;
                    }
                    return 7;
                    break;
                case 0xB4:  /* LFS */
                    kprintf("[0F]:[B4] = LFS\n");
                    panic("System halted...\n");
                    break;
                case 0xB5:  /* LGS */
                    kprintf("[0F]:[B5] = LGS\n");
                    panic("System halted...\n");
                    break;
                default:
                    kprintf("Unhandled case!\n");
                    panic("System halted...\n");
                    break;
            };
            break;
        case 0xE6:  /* out imm8, AL */
            outportb(byte2, (unsigned char)regs.eax);
            return 2;
            break;
        case 0xEE:  /* out DX, AL */
            outportb((unsigned short)regs.edx, (unsigned char)regs.eax);
            return 1;
            break;
        case 0xEF:  /* outw */
            kprintf("Unhandled case!\n");
            panic("System halted...\n");
            break;
        case 0xF4:  /* hlt */
            panic("Halting");
            break;
        default:
            return 0;
            break;
    };
    /* Should never be called */
    return 0xFFFFFFFF;
} /* instr_handler */
/******************************************************************************
******************************************************************************/
extern unsigned task0_page_dir, task1_page_dir;
extern unsigned vir_page_table;
extern unsigned idt_ptr, gdt_ptr;
extern unsigned conv_mem_page_table, kernel_page_table_2;

void init_guest_paging(task_t *task) {
    unsigned *page_dir, dir_ent, *page_table, ptr;
    int i, j;
    unsigned pde, pte;

kprintf("In init_guest_paging...\n");
```

```
    /* Get start of guest page table directory */
    page_dir = &task1_page_dir;
    /* Get guest page table entries */
    dir_ent = (unsigned)&vir_page_table;

    /* Initialise 16MB of memory (4 indices) */
    for (i=0; i<4; i++) {
        /* Set page info (user, r/w, present) */
        dir_ent |= 0x07;
        /* Insert page entry */
        page_dir[i] = dir_ent;
        /* Next page entry */
        dir_ent += 0x1000;
    }

    /* Create entry for page directory itself */
    page_dir[1023] = (unsigned)&task1_page_dir;
    page_dir[1023] |= 0x07;

    /* Identity map first 16MB of linear address space */
    page_table = &vir_page_table;
    for (i=0x0; i<0x04; i++) {
        page_table = (unsigned *)(page_dir[i] & 0xFFFFF000);
        ptr = i * 0x400000;
        for (j=0x0; j<0x400; j++)  {
            /* Set page address */
            page_table[j] = ptr + j*0x1000;
            /* Set page info (user, r/w, present) */
            page_table[j] |= 0x07;
        }
    }

    /* Map test code */
    pde = ((unsigned)task->task_mem + 0x100000) >> 22;
    pte = (((unsigned)task->task_mem + 0x100000) >> 12) & 0x3FF;
    page_table = (unsigned *)(page_dir[pde] & 0xFFFFF000);
    page_table[pte] = 0x4100000;
    page_table[pte] |= 0x07;

    /* Protect VGA memory */
    pde = ((unsigned)task->task_mem + 0xA0000) >> 22;
    pte = (((unsigned)task->task_mem + 0xA0000) >> 12) & 0x3FF;
    page_table = (unsigned *)(page_dir[pde] & 0xFFFFF000);
    for (i=0x0; i<0x20; i++)
        /* Change info to (supervisor, r/w, present) */
        page_table[pte+i] &= 0xFFFFFFFB;

    /* Update task structure with new page table */
    task->page_dir = &task1_page_dir;

/* Take a little peek *
pde = (((unsigned)task->task_mem) + 0x10000) >> 22;
pte = ((((unsigned)task->task_mem) + 0x10000) >> 12) & 0x3FF;
page_table = (unsigned *)(page_dir[0] & 0xFFFFF000);
kprintf("cr2:                 %X\n", get_page_fault_address());
kprintf("cr3:                 %X\n", get_cr3());
kprintf("page_table:          %X\n", &page_table);
```

```
kprintf("&conv_mem_page_table:  %X\n", &conv_mem_page_table);
kprintf("&kernel_page_table_2:  %X\n", &kernel_page_table_2);
kprintf("task1_page_dir:        %X\n", task1_page_dir);
kprintf("&task1_page_dir:       %X\n", &task1_page_dir);
kprintf("page_dir[%3d]:         %X\n", pde, page_dir[pde]);
kprintf("page_table[%3d]:       %X\n", 0, page_table[0]);
kprintf("page_table[%3d]:       %X\n", 1, page_table[1]);
kprintf("page_table[%3d]:       %X\n", pte, page_table[pte]);
kprintf("page_table[%3d]:       %X\n", 0x10, page_table[0x10]);
kprintf("page_table[%3d]:       %X\n", ((unsigned)task->task_mem >> 12)
                                       & 0x3FF, page_table[((unsigned)task->task_mem >> 12)
                                       & 0x3FF]);
for(;;); */


} /* init_guest_paging */

extern unsigned tss1[];

void init_tss(task_t *task, regs_t *regs) {
    tss1[1] = task->krnl_esp;
    tss1[7] = (unsigned)task->page_dir;
    tss1[8] = regs->eip;
    tss1[9] = regs->eflags;
    tss1[14] = regs->user_esp;
    tss1[18] = regs->es;
    tss1[19] = regs->cs;
    tss1[20] = regs->user_ss;
    tss1[21] = regs->ds;
    tss1[22] = regs->fs;
    tss1[23] = regs->gs;
} /* init_tss */

int init_testcode(void) {
    unsigned long size;
    task_t *task;
    regs_t *regs;

    task = _tasks + 1;
    /* Alloc kernel stack */
    size = KRNL_STACK_SIZE;
    task->kstack_mem = kmalloc(size);

    if(task->kstack_mem == NULL) {
        kprintf("Init_testcode: Couldn't alloc %lu bytes for stack\n", size);
        return -1;
    }
    task->krnl_esp = (unsigned long)task->kstack_mem + size - sizeof(regs_t);

    /* Alloc virtual video memory (128kb) */
    task->vc = _vc + 1;
    task->vc->vid_mem = kmalloc(0x20000);

    /* Push initial task regs on kernel stack */
    regs = (regs_t *)task->krnl_esp;
    regs->ds = regs->es = regs->fs = regs->gs =
        regs->user_ss = USER_DATA_SEL;
    regs->cs = USER_CODE_SEL;
```

```
    /* Start at 0x100000 = 1MB */
    regs->eip = (unsigned long)task->task_mem + 0x100000;
//    regs->eip = (unsigned long)task->task_mem;
    /* Enable interrupts */
    regs->eflags = 0x200;
    /* convert highest to size, including user stack */
    task->size = size = 0x01000000;     /* 0x1000000 = 16777216 */
    regs->user_esp = 0x0 + size;

    /* alloc task memory */
    task->task_mem = kmalloc(size);

    if(task->task_mem == NULL)
    {
        kprintf("Init_testcode: couldn't alloc %lu bytes for task\n", size);
        kfree(task->kstack_mem);
        return -1;
    }

    /* Identity map low 1st MB of memory */
//    map_low(task->task_mem, size);

    /* Initialise rest of task structure */
    task->page_dir = &task1_page_dir;
    task->lva = 0x0;
    task->exit_code = 0x0;
    task->timeout = 0x0;
    task->next = task->prev = 0x0;
    task->status = TS_RUNNABLE;
    task->tss = TSS1_SEL;

    /* Set ring 3 segment descriptors */
    set_descriptor_base(_gdt5, (unsigned long)task->task_mem + _virt_to_phys);
    set_descriptor_base(_gdt6, (unsigned long)task->task_mem + _virt_to_phys);
    set_descriptor_limit(_gdt5, task->size - 1);
    set_descriptor_limit(_gdt6, task->size - 1);

    /* Copy 512 byte 32bit test code to allocated memory */
//    movedata(LINEAR_SEL, 0x010000, LINEAR_SEL, (unsigned long)task->task_mem +
                                                    _virt_to_phys + 0x00010000, 0x200);
    movedata(LINEAR_SEL, 0x010000, LINEAR_SEL, 0x04100000, 0x200);
    /* !!! - Remember to free pages after use */

    /* MUST do this: */
    _curr_task = _tasks + 0;

    /* Setup TSS for guest task */
    /* init_tss(task, regs); */

    /* Setup paging for guest code */
    init_guest_paging(task);

    return 0;
} /* init_testcode */

/* Linux bootloader expects to be placed at 0x00007C00, so copy code there */
int init_linux(void) {
    unsigned long size;
```

```
    task_t *task;
    regs_t *regs;

    task = _tasks + 1;
    /* alloc kernel stack */
    size = KRNL_STACK_SIZE;
    task->kstack_mem = kmalloc(size);

    if(task->kstack_mem == NULL)
    {
        kprintf("run: couldn't alloc %lu bytes for stack\n", size);
        return -1;
    }
    task->krnl_esp = (unsigned long)task->kstack_mem +
        size - sizeof(regs_t);

    /* push initial task regs on kernel stack */
    regs = (regs_t *)task->krnl_esp;
    regs->ds = regs->es = regs->fs = regs->gs =
        regs->user_ss = USER_DATA_SEL;
    regs->cs = USER_CODE_SEL;
    regs->eip = 0x00100000;
    regs->eflags = 0x200;     /* Enable interrupts */
    /* convert highest to size, including user stack */
    task->size = size = 0x01000000;     /* 0x1000000 = 16777216 */
    regs->user_esp = 0x0 + size;

    /* alloc task memory */
    task->task_mem = kmalloc(size);

    if(task->task_mem == NULL)
    {
        kprintf("run: couldn't alloc %lu bytes for task\n", size);
        kfree(task->kstack_mem);
        return -1;
    }

    /* Identity map low 1st Mb of memory */
//    map_low(task->task_mem, size);

    /* Initialise rest of task structure */
    task->lva = 0x0;
    task->vc = 0x0;
    task->exit_code = 0x0;
    task->timeout = 0x0;
    task->next = task->prev = 0x0;
    task->status = TS_RUNNABLE;

    /* Set ring 3 segment descriptors */
    set_descriptor_base(_gdt5, (unsigned long)task->task_mem + _virt_to_phys);
    set_descriptor_base(_gdt6, (unsigned long)task->task_mem + _virt_to_phys);
    set_descriptor_limit(_gdt5, task->size - 1);
    set_descriptor_limit(_gdt6, task->size - 1);

    /* Copy 523Kb 32bit kernel code to allocated memory (0x82D41 = 535873) */
    movedata(LINEAR_SEL, 0x02000A00, LINEAR_SEL, (unsigned long)task->task_mem
                                            + _virt_to_phys + 0x00100000, 0x82D41);
    /* !!! - Remember to free pages after use */
```

```
    /* MUST do this: */
    _curr_task = _tasks + 0;

    return 0;
} /* init_linux */
```

Bilag G

dma.c

```
////////////////////////////////////////////////////////////////////////////////
// DMA CONTROLLER
//
// EXPORTS:
// None
////////////////////////////////////////////////////////////////////////////////
#include <x86.h> /* disable(), enable(), outportb() */
#include <krnl.h> /* kmalloc() */
#include <DMA.h>

/* Quick-access registers and ports for each DMA channel. */
unsigned char MaskReg[8]   = { 0x0A, 0x0A, 0x0A, 0x0A, 0xD4, 0xD4, 0xD4, 0xD4 };
unsigned char ModeReg[8]   = { 0x0B, 0x0B, 0x0B, 0x0B, 0xD6, 0xD6, 0xD6, 0xD6 };
unsigned char ClearReg[8]  = { 0x0C, 0x0C, 0x0C, 0x0C, 0xD8, 0xD8, 0xD8, 0xD8 };

unsigned char PagePort[8]  = { 0x87, 0x83, 0x81, 0x82, 0x8F, 0x8B, 0x89, 0x8A };
unsigned char AddrPort[8]  = { 0x00, 0x02, 0x04, 0x06, 0xC0, 0xC4, 0xC8, 0xCC };
unsigned char CountPort[8] = { 0x01, 0x03, 0x05, 0x07, 0xC2, 0xC6, 0xCA, 0xCE };

/* Allocate DMA compatible low memory (first 640Kb) */
/* Hardcoded to be second 64Kb segment at [1000]:[0000] */
dma_block* alloc_DMA(unsigned short length) {
    dma_block *dma;

    /* Allocate memory for structure */
    dma = (dma_block *)kmalloc(sizeof(dma_block));

    dma->page = 0x0001;
    dma->offset = 0x0000;
    dma->length = length-1;     /* 0=1 byte transfered, 511=512 bytes transfered */

    return(dma);
} /* alloc_DMA */

void start_DMA(unsigned char DMA_channel, dma_block *dma, unsigned char mode)
{
/*    unsigned short page = 0x0001;
    unsigned short offset = 0x0000; */

/* Allocate DMA compatible low memory (first 640Kb) */
/* Hardcoded to be second 64Kb segment at [1000]:[0000] */
```

```
    /* First, make sure our 'mode' is using the DMA channel specified. */
    mode |= DMA_channel;

    /* Don't let anyone else mess up what we're doing. */
    disable();

    /* Set up the DMA channel so we can use it.  This tells the DMA */
    /* that we're going to be using this channel.  (It's masked) */
    outportb(MaskReg[DMA_channel], 0x04 | DMA_channel);

    /* Clear any data transfers that are currently executing. */
    outportb(ClearReg[DMA_channel], 0x00);

    /* Send the specified mode to the DMA. */
    outportb(ModeReg[DMA_channel], mode);

    /* Send the offset address.  The first byte is the low base offset, the */
    /* second byte is the high offset. */
    outportb(AddrPort[DMA_channel], LOW_BYTE((dma->offset + dma->used)));
    outportb(AddrPort[DMA_channel], HI_BYTE((dma->offset + dma->used)));

    /* Send the physical page that the data lies on. */
    outportb(PagePort[DMA_channel], dma->page);

    /* Send the length of the data.  Again, low byte first. */
    outportb(CountPort[DMA_channel], LOW_BYTE(dma->length));
    outportb(CountPort[DMA_channel], HI_BYTE(dma->length));

    /* Ok, we're done.  Enable the DMA channel (clear the mask). */
    outportb(MaskReg[DMA_channel], DMA_channel);

    /* Re-enable interrupts before we leave. */
    enable();
} /* start_DMA */

void pause_DMA(unsigned char DMA_channel)
{
    /* All we have to do is mask the DMA channel's bit on. */
    outportb(MaskReg[DMA_channel], 0x04 | DMA_channel);
} /* pause_DMA */

void unpause_DMA(unsigned char DMA_channel)
{
    /* Simply clear the mask, and the DMA continues where it left off. */
    outportb(MaskReg[DMA_channel], DMA_channel);
} /* unpause_DMA */

void stop_DMA(unsigned char DMA_channel)
{
    /* We need to set the mask bit for this channel, and then clear the */
    /* selected channel.  Then we can clear the mask. */
    outportb(MaskReg[DMA_channel], 0x04 | DMA_channel);

    /* Send the clear command. */
    outportb(ClearReg[DMA_channel], 0x00);

    /* And clear the mask. */
```

```
    outportb(MaskReg[DMA_channel], DMA_channel);
} /* stop_DMA */
```

Bilag H

floppy.c

```
////////////////////////////////////////////////////////////////////////////////
// FLOPPY CONTROLLER
//
// EXPORTS:
// void floppy_irq(void);
// void init_floppy(void);
////////////////////////////////////////////////////////////////////////////////
#include <x86.h>
#include <krnl.h>
#include <string.h> /* movedata() */
#include <DMA.h>
#include <floppy.h>

/* declare global floppy structure */
floppy_t *fdc;

void send_byte(floppy_t *fdc, unsigned char byte)
{
    volatile int msr;
    int tmo;

    for (tmo = 0;tmo < 128;tmo++) {
        msr = inportb(fdc->base + MAIN_STATUS_REG);
        if ((msr & 0xC0) == 0x80) {
            outportb(fdc->base + DATA_REG, byte);
            return;
        }
        inportb(0x80);   /* delay */
    }
} /* send_byte */

unsigned char get_byte(floppy_t *fdc)
{
    volatile int msr;
    int tmo;

    for (tmo = 0;tmo < 128;tmo++) {
        msr = inportb(fdc->base + MAIN_STATUS_REG);
        if ((msr & 0xC0) == 0xC0) {
            return inportb(fdc->base + DATA_REG);
        }
```

```
        inportb(0x80);   /* delay */
    }

    return (unsigned char) -1;  /* read timeout */
} /* get_byte */

void floppy_irq(void) {

//kprintf("In floppy_irq...\n");
    /* Disable interrupts */
    disable();

    switch (fdc->state) {
        case RESET:
            send_byte(fdc, SENSE_CMD);
            fdc->st0 = get_byte(fdc);
            fdc->cylinder = get_byte(fdc);
            fdc->state = CLEAR;
            break;
        case SENSE_INT:
            send_byte(fdc, SENSE_CMD);
            fdc->st0 = get_byte(fdc);
            fdc->cylinder = get_byte(fdc);
            fdc->state = CLEAR;
            break;
        case ISSUE_INT:
            fdc->status[0] = get_byte(fdc);
            fdc->status[1] = get_byte(fdc);
            fdc->status[2] = get_byte(fdc);
            fdc->status[3] = get_byte(fdc);
            fdc->status[4] = get_byte(fdc);
            fdc->status[5] = get_byte(fdc);
            fdc->status[6] = get_byte(fdc);
            fdc->state = CLEAR;
            break;
        default:
            kprintf("Something else...?\n");
            fdc->state = CLEAR;
    }
    /* Enable interrupts */
    enable();
} /* floppy_irq */

void recalibrate(void) {

//    kprintf("In recalibrate...\n");

    /* Send Recalibrate command */
    fdc->state = SENSE_INT;
    send_byte(fdc, RECAL_CMD);
    send_byte(fdc, 0x0);

    /* Wait until Sense Interrupt command has been issued */
    while (fdc->state != CLEAR);

    if ((fdc->st0 & 0xE0) != 0x20) { /* 0xE0 = 11100000, 0x20 = 00100000 */
        kprintf("Fault in recalibrate operation!\n");
        kprintf("    st0 value.......... #%x\n", fdc->st0);
```

```
        kprintf("    Drive at cylinder... #%x\n", fdc->cylinder);
    }
} /* recalibrate */

void seek(unsigned char head, unsigned char cylinder) {

//    kprintf("In seek...\n");

    /* Sending Seek command to controller */
    fdc->state = SENSE_INT;
    send_byte(fdc, SEEK_CMD);
    send_byte(fdc, ((0x0 & head) << 2));
    send_byte(fdc, cylinder);

    /* Wait until Sense Interrupt command has been issued */
    while (fdc->state != CLEAR);

    if ((fdc->st0 & 0xE0) != 0x20) { /* 0xE0 = 11100000, 0x20 = 00100000 */
        kprintf("Fault in seek operation!\n");
        kprintf("    st0 value........... #%x\n", fdc->st0);
        kprintf("    Drive at cylinder... #%x\n", fdc->cylinder);
        kprintf("    Should be at........ #%x\n", cylinder);
    }
} /* seek */

void read_sector(unsigned char head, unsigned char cylinder, unsigned char sector) {

//    kprintf("In read_sector...\n");

    /* Initialise DMA controller? */
    /* ... */

    /* Expect interrupt */
    fdc->state = ISSUE_INT;

    /* Send Read command */
    send_byte(fdc, READ_CMD);       /* MT, MFM */
    send_byte(fdc, head << 2);     /* Head, drive A */
    send_byte(fdc, cylinder);
    send_byte(fdc, head);
    send_byte(fdc, sector);
    send_byte(fdc, 0x02);           /* Sector size 2 = 512b */
    send_byte(fdc, 0x12);           /* Max sector nr = 18 */
    send_byte(fdc, 0x1B);           /* Length of GAP3 = 27 */
    send_byte(fdc, 0xFF);           /* Not used */

    /* Wait for interrupt */
    while (fdc->state != CLEAR);

    if ((fdc->status[0] & 0xFB)) { /* 0xFB=11111011, where 0 = floppy head */
        kprintf("Fault in read_sector!\n");
        kprintf("    st0................. #%x\n", fdc->status[0]);
        kprintf("    st1................. #%x\n", fdc->status[1]);
        kprintf("    st2................. #%x\n", fdc->status[2]);
        kprintf("    Cylinder............ #%x\n", fdc->status[3]);
        kprintf("    Head................ #%x\n", fdc->status[4]);
        kprintf("    Sector.............. #%x\n", fdc->status[5]);
        kprintf("    Sector size......... #%x\n", fdc->status[6]);
```

```
        return;
    }

    fdc->cylinder = fdc->status[3];
    fdc->head = fdc->status[4];
    fdc->sector = fdc->status[5];

    /* All clear. Update dma_block structure */
    /* Remember to check for page boundary + length */
    fdc->dma->used += fdc->dma->length+1;

} /* read_sector */

void init_floppy(void) {

//    kprintf("In init_floppy...\n");

    /* Allocate memory for floppy structure */
    fdc = (floppy_t *)kmalloc(sizeof(floppy_t));

    fdc->state = CLEAR;
    fdc->spin_end = 0;
    fdc->base = PRIM_BASE_ADDRESS;

    /* Allocate memory for DMA??? */
    /* ... maybe during Read Sector command? */

    /* Reset floppy controller */
    fdc->state = RESET;
    outportb(fdc->base + DIGITAL_OUTPUT_REG, 0x00);

    /* Enable motor A, DMA/IRQ, controller and select drive A */
    outportb(fdc->base + DIGITAL_OUTPUT_REG, DRIVE_ON);
    /* Wait until drive is ready */
    while ((inportb(fdc->base + MAIN_STATUS_REG) & 0x80) == 0);

    /* Program data rate (500 kbit/s) */
    outportb(fdc->base + CONFIG_CONTROL_REG, 0x0);
    while ((inportb(fdc->base + MAIN_STATUS_REG) & 0x80) == 0);

    /* Wait for interrupt to clear status flag after reset */
    while (fdc->state != CLEAR);

    /* Issue Configure commmand??? */
    /* ... default values acceptable */

    /* Specify drive timings (got these off the BIOS) */
    send_byte(fdc, SPECIFY_CMD);
    send_byte(fdc, 0xDF);       /* SRT = 3ms, HUT = 240ms */
    send_byte(fdc, 0x02);       /* HLT = 16ms, ND = 0 */
} /* init_floppy */

void load_testcode(void) {
    dma_block *dma_mem;

    kprintf("Loading test code");
    /* 0x4800 = 18432 bytes = 2 tracks = 1 cylinder = 36 sectors */
    /* 0x2800 = 10240 bytes = 64KB - 3*2 tracks = space remaining in DMA buffer */
```

```
    /* 0x2000 = 8192 bytes */
    /* 0x200  = 512 bytes = 1 sector */
    dma_mem = alloc_DMA(0x200);
    /* For start_DMA third operand: */
    /* 0x46 = 01000110 (single, increment, no-auto, write, channel 2) */
    /* 0x06 = 00000110 (demand, increment, no-auto, write, channel 2) */
    start_DMA(0x02, dma_mem, 0x46);
    init_floppy();
    fdc->dma = dma_mem;
    fdc->head = 0x0;
    fdc->cylinder = 0x0;
    fdc->sector = 0x01;

    /* Floppy controller commands */
    recalibrate();
    /* Read first sector on disk (boot sector) */
    read_sector(fdc->head, fdc->cylinder, fdc->sector);

    /* Stop drive, execute reset */
    fdc->state = RESET;
    outportb(fdc->base + DIGITAL_OUTPUT_REG, 0x00);
    while (fdc->state != CLEAR);

    kprintf("!\n");
} /* load_testcode */

void load_kernel(void) {
    dma_block *dma_mem;
    int clndr;
    unsigned short dma_left = 0xFFFF;
    unsigned long tmp_ptr = 0x2000000;
    bool full = true;

    kprintf("Loading Linux kernel");
    /* 0x4800 = 18432 bytes = 2 tracks = 1 cylinder = 36 sectors */
    /* 0x2800 = 10240 bytes = 64KB - 3*2 tracks = space remaining in DMA buffer */
    /* 0x2000 = 8192 bytes */
    /* 0x200  = 512 bytes = 1 sector */
    dma_mem = alloc_DMA(0x4800);
    /* For start_DMA third operand: */
    /* 0x46 = 01000110 (single, increment, no-auto, write, channel 2) */
    /* 0x06 = 00000110 (demand, increment, no-auto, write, channel 2) */
    start_DMA(0x02, dma_mem, 0x46);
    init_floppy();
    fdc->dma = dma_mem;
    fdc->head = 0x0;
    fdc->cylinder = 0x0;
    fdc->sector = 0x01;

    /* Floppy controller commands */
    recalibrate();
    /* Read first sector on disk (boot sector) */

    for (clndr=0; clndr<41; clndr++) {
        /* Is DMA buffer full? (Greater than 64KB) */
        if (fdc->dma->used >= 0x10000) {
            /* Copy linux kernel to 32MB */
            movedata(LINEAR_SEL, 0x00010000, LINEAR_SEL, tmp_ptr, 0x10000);
```

```
            /* Reallocate DMA buffer */
            fdc->dma->length = 0x4800 - fdc->dma->length -2;
            fdc->dma->used = 0x0;
            dma_left = 0xFFFF;
            tmp_ptr += 0x10000;
            /* Next read won't be a full cylinder */
            full = false;
        }
        if (dma_left < 0x4800) {
            fdc->dma->length = dma_left-1;
            /* dma_mem->used is unchanged */
        }
        if (clndr > 0) {
            start_DMA(0x02, fdc->dma, 0x46);
        }
        seek(0x0, fdc->cylinder);
        read_sector(fdc->head, fdc->cylinder, fdc->sector);
        /* Calculate space left in the DMA buffer */
        dma_left = 0x10000 - fdc->dma->used;
        /* Are we ready to read a full cylinder? */
        if (!full) {
            fdc->dma->length = 0x4800-1;
            full = true;
        }
        kprintf(".");
    }

    /* Stop drive, execute reset */
    fdc->state = RESET;
    outportb(fdc->base + DIGITAL_OUTPUT_REG, 0x00);
    while (fdc->state != CLEAR);

    kprintf("\n");
} /* load_kernel */
```

Bilag I

keyboard.c

```
///////////////////////////////////////////////////////////////////////////////
// KEYBOARD
//
// EXPORTS:
// void kbd_irq(void);
// void init_kbd(void);
// int sys_getch(console_t *con);
///////////////////////////////////////////////////////////////////////////////
#include <conio.h> /* KEY_nnn */
#include <krnl.h> /* true, false, queue_t, console_t, kprintf() */
#include <x86.h> /* inportb(), outportb() */

/* IMPORTS
from start4.asm */
void reboot(void);
void do_nothing(void);

/* from video/console code */
extern console_t _vc[];
extern volatile console_t *_curr_vc;
void select_vc(unsigned which_con);

/* "raw" set 1 scancodes from PC keyboard. Keyboard info here:
http://www.execpc.com/~geezer/os/kbd.txt
http://www.execpc.com/~geezer/osd/kbd */
#define RAW_LEFT_CTRL   0x1D
#define RAW_LEFT_SHIFT  0x2A
#define RAW_CAPS_LOCK   0x3A
#define RAW_LEFT_ALT    0x38
#define RAW_RIGHT_ALT   0x38    /* same as left */
#define RAW_RIGHT_CTRL  0x1D    /* same as left */
#define RAW_RIGHT_SHIFT 0x36
#define RAW_SCROLL_LOCK 0x46
#define RAW_NUM_LOCK    0x45

#define KBD_BUF_SIZE    512
/*****************************************************************************
    name:   inq
    action: tries to add data (a byte) to queue
    returns:-1 if queue full
        0  if success
```

```
*************************************************************************/
static int inq(queue_t *queue, unsigned char data)
{
    unsigned temp;

    temp = queue->inptr + 1;
    if(temp >= queue->size)
        temp = 0;
    if(temp == queue->outptr)
        return -1;  /* full */
    queue->data[queue->inptr] = data;
    queue->inptr = temp;
    queue->non_empty = true;
    return 0;
}
/*************************************************************************
    name:   deq
    action: tries to get byte from queue
    returns:-1 if queue empty
        0  if success (data set to value read from queue)
*************************************************************************/
static int deq(queue_t *queue, unsigned char *data)
{
    if(!queue->non_empty)
        return -1;  /* empty */
    *data = queue->data[queue->outptr++];
    if(queue->outptr >= queue->size)
        queue->outptr = 0;
    if(queue->outptr == queue->inptr)
        queue->non_empty = false;
    return 0;
}
/*************************************************************************
    name:   write_kbd
    action: writes data to 8048 keyboard MCU (adr=0x60) or
        8042 keyboard controller (adr=0x64)
*************************************************************************/
static void write_kbd(unsigned adr, unsigned char data)
{
    unsigned long timeout;
    unsigned char stat;

/* Linux code didn't have a timeout here... */
    for(timeout = 500000L; timeout != 0; timeout--)
    {
        stat = inportb(0x64);
/* loop until 8042 input buffer empty */
        if((stat & 0x02) == 0)
            break;
    }
    if(timeout != 0)
        outportb(adr, data);
/* xxx - else? */ }
/*************************************************************************
    name:   convert
    action: converts raw scancodes
    returns:0 if nothing to return, else 8-bit "ASCII" value
*************************************************************************/
```

```
static unsigned short convert(console_t *con, unsigned char code)
{
    static const unsigned char map[] =
    {
/* 00 */0,  0x1B,   '1',    '2',    '3',    '4',    '5',    '6',
/* 08 */'7',    '8',    '9',    '0',    '-',    '=',    '\b',   '\t',
/* 10 */'q',    'w',    'e',    'r',    't',    'y',    'u',    'i',
/* 1Dh is left Ctrl */
/* 18 */'o',    'p',    '[',    ']',    '\n',   0,  'a',    's',
/* 20 */'d',    'f',    'g',    'h',    'j',    'k',    'l',    ';',
/* 2Ah is left Shift */
/* 28 */'\'',   '`',    0,  '\\',   'z',    'x',    'c',    'v',
/* 36h is right Shift */
/* 30 */'b',    'n',    'm',    ',',    '.',    '/',    0,  0,
/* 38h is left Alt, 3Ah is Caps Lock */
/* 38 */0,  ' ',    0,  KEY_F1, KEY_F2, KEY_F3, KEY_F4, KEY_F5,
/* 45h is Num Lock, 46h is Scroll Lock */
/* 40 */KEY_F6, KEY_F7, KEY_F8, KEY_F9, KEY_F10,0,  0,  KEY_HOME,
/* 48 */KEY_UP, KEY_PGUP,'-',   KEY_LFT,'5',    KEY_RT, '+',    KEY_END,
/* 50 */KEY_DN, KEY_PGDN,KEY_INS,KEY_DEL,0, 0,  KEY_F11,KEY_F12
    };
    static unsigned short kbd_status;
    unsigned short temp;

/* check for break code (i.e. a key is released) */
    if(code >= 0x80)
    {
        con->saw_break_code = true;
        code &= 0x7F;
    }
/* the only break codes we're interested in are Shift, Ctrl, Alt */
    if(con->saw_break_code)
    {
        if(code == RAW_LEFT_ALT || code == RAW_RIGHT_ALT)
            kbd_status &= ~KBD_META_ALT;
        else if(code == RAW_LEFT_CTRL || code == RAW_RIGHT_CTRL)
            kbd_status &= ~KBD_META_CTRL;
        else if(code == RAW_LEFT_SHIFT || code == RAW_RIGHT_SHIFT)
            kbd_status &= ~KBD_META_SHIFT;
        con->saw_break_code = false;
        return 0;
    }
/* it's a make code: check the "meta" keys, as above */
    if(code == RAW_LEFT_ALT || code == RAW_RIGHT_ALT)
    {
        kbd_status |= KBD_META_ALT;
        return 0;
    }
    if(code == RAW_LEFT_CTRL || code == RAW_RIGHT_CTRL)
    {
        kbd_status |= KBD_META_CTRL;
        return 0;
    }
    if(code == RAW_LEFT_SHIFT || code == RAW_RIGHT_SHIFT)
    {
        kbd_status |= KBD_META_SHIFT;
        return 0;
    }
```

```
/* Scroll Lock, Num Lock, and Caps Lock set the LEDs. These keys
have on-off (toggle or XOR) action, instead of momentary action */
    if(code == RAW_SCROLL_LOCK)
    {
        kbd_status ^= KBD_META_SCRL;
        goto LEDS;
    }
    if(code == RAW_NUM_LOCK)
    {
        kbd_status ^= KBD_META_NUM;
        goto LEDS;
    }
    if(code == RAW_CAPS_LOCK)
    {
        kbd_status ^= KBD_META_CAPS;
LEDS:       write_kbd(0x60, 0xED);  /* "set LEDs" command */
        temp = 0;
        if(kbd_status & KBD_META_SCRL)
            temp |= 1;
        if(kbd_status & KBD_META_NUM)
            temp |= 2;
        if(kbd_status & KBD_META_CAPS)
            temp |= 4;
        write_kbd(0x60, temp);  /* bottom 3 bits set LEDs */
        return 0;
    }
/* ignore invalid scan codes */
    if(code >= sizeof(map) / sizeof(map[0]))
        return 0;
/* convert raw scancode in code to unshifted ASCII in temp */
    temp = map[code];
/* defective keyboard? non-US keyboard? more than 104 keys? */
    if(temp == 0)
        return temp;
/* handle the three-finger salute */
    if((kbd_status & KBD_META_CTRL) && (kbd_status & KBD_META_ALT) &&
        (temp == KEY_DEL))
    {
        kprintf("\n""\x1B[42;37;1m""*** rebooting!");
        reboot();
    }
/* I really don't know what to do yet with Alt, Ctrl, etc. -- punt */
    return temp;
}
/*****************************************************************************
*****************************************************************************/
void kbd_irq(void)
{
    unsigned short temp;

/* get scancode from port 0x60 */
    temp = inportb(0x60);
/* convert scancode to pseudo-ASCII */
    temp = convert(_curr_vc, temp);
/* if it's F1, F2 etc. switch to the appropriate virtual console */
    switch(temp)
    {
        case KEY_F1:
```

```
            temp = 0;
            goto SWITCH_VC;
        case KEY_F2:
            temp = 1;
            goto SWITCH_VC;
        case KEY_F3:
            temp = 2;
            goto SWITCH_VC;
        case KEY_F4:
            temp = 3;
            goto SWITCH_VC;
        case KEY_F5:
            temp = 4;
            goto SWITCH_VC;
        case KEY_F6:
            temp = 5;
            goto SWITCH_VC;
        case KEY_F7:
            temp = 6;
            goto SWITCH_VC;
        case KEY_F8:
            temp = 7;
            goto SWITCH_VC;
        case KEY_F9:
            temp = 8;
            goto SWITCH_VC;
        case KEY_F10:
            temp = 9;
            goto SWITCH_VC;
        case KEY_F11:
            temp = 10;
            goto SWITCH_VC;
        case KEY_F12:
            temp = 11;
SWITCH_VC:
            select_vc(temp);
            break;
/* convert() ate it */
        case 0:
            break;
        default:
            if(inq(&_curr_vc->keystrokes, temp) != 0)
                temp++;
                temp--;
                /* full queue, beep or something */;
            break;
    }
    outportb(0x20, 0x20); /* reset 8259 interrupt controller chip */
}
/*****************************************************************************
*****************************************************************************/
void init_kbd(void)
{
    static unsigned char buffers[KBD_BUF_SIZE * MAX_VC];
    int temp;

    for(temp = 0; temp < MAX_VC; temp++)
    {
```

```
        _vc[temp].keystrokes.data = buffers + KBD_BUF_SIZE * temp;
        _vc[temp].keystrokes.size = KBD_BUF_SIZE;
    }
    kprintf("init_kbd: %u buffers, %u bytes each\n",
        MAX_VC, KBD_BUF_SIZE);
}
/*****************************************************************************
this routine spins until a byte arrives in the queue (slow!)
*****************************************************************************/
int sys_getch(console_t *con)
{
    unsigned char ret_val;

    while(deq(&con->keystrokes, &ret_val) != 0)
    {
/* open an interrupt "window" to let timer or keyboard interrupts occur */
            enable();
            do_nothing();
            disable();
    }
    return ret_val;
}
```

Bilag J

video.c

```
///////////////////////////////////////////////////////////////////////////////
// VIRTUAL CONSOLES
//
// EXPORTS:
// unsigned long _vga_fb_adr;
// console_t _vc[];
// volatile console_t *_curr_vc;
// void select_vc(unsigned which_con);
// void sys_putch(console_t *con, unsigned char c);
// void init_console(void);
////////////////////////////////////////////////////////////////////////////////
#include <string.h> /* movedata() */
#include <stdio.h> /* sprintf() */
#include <ctype.h> /* isdigit() */
#include <krnl.h> /* LINEAR_SEL, MAX_VC, console_t */
#include <x86.h> /* fmemsetw(), pokew() */

#define VGA_MISC_READ   0x3CC

console_t _vc[MAX_VC];
volatile console_t *_curr_vc;

/*static*/ unsigned long _vga_fb_adr;
static unsigned short _crtc_io_adr;
static unsigned char _vc_width = 80, _vc_height = 25;
/*****************************************************************************
*****************************************************************************/
static void scroll(console_t *con)
{
    unsigned short blank, temp;

    blank = 0x20 | ((unsigned short)con->attrib << 8);
/* scroll up */
    if(con->csr_y >= _vc_height)
    {
        temp = con->csr_y - _vc_height + 1;
        movedata(LINEAR_SEL, con->fb_adr + temp * _vc_width * 2,
            LINEAR_SEL, con->fb_adr,
            (_vc_height - temp) * _vc_width * 2);
/* blank the bottom line of the screen */
        fmemsetw(LINEAR_SEL,
```

```
            con->fb_adr + (_vc_height - temp) * _vc_width * 2,
            blank, _vc_width);
        con->csr_y = _vc_height - 1;
    }
}
/*****************************************************************************
*****************************************************************************/
static void set_attrib(unsigned char att, console_t *con)
{
    static const char ansi_to_vga[] =
    {
        0, 4, 2, 6, 1, 5, 3, 7
    };
    unsigned char new_att;

    new_att = con->attrib;
    if(att == 0)
        new_att &= ~0x08;       /* bold off */
    else if(att == 1)
        new_att |= 0x08;        /* bold on */
    else if(att >= 30 && att <= 37)
    {
        att = ansi_to_vga[att - 30];
        new_att = (new_att & ~0x07) | att;/* fg color */
    }
    else if(att >= 40 && att <= 47)
    {
        att = ansi_to_vga[att - 40] << 4;
        new_att = (new_att & ~0x70) | att;/* bg color */
    }
    con->attrib = new_att;
}
/*****************************************************************************
*****************************************************************************/
static void move_csr(console_t *con)
{
    unsigned long temp;
    unsigned short off;
    unsigned flags;

    temp = (con->csr_y * _vc_width + con->csr_x) * 2;
    temp = con->fb_adr + temp - _vga_fb_adr;
    off = temp;
    flags = critb();
/* extra shift because even/odd text mode uses word clocking */
    outportb(_crtc_io_adr + 0, 14);
    outportb(_crtc_io_adr + 1, off >> 9);
    outportb(_crtc_io_adr + 0, 15);
    outportb(_crtc_io_adr + 1, off >> 1);
    crite(flags);
}
/*****************************************************************************
*****************************************************************************/
void select_vc(unsigned which_con)
{
    unsigned long temp;
    unsigned short off;
    unsigned flags;
```

```
    if(which_con >= MAX_VC)
        return;
    _curr_vc = _vc + which_con;
    temp = _curr_vc->fb_adr - _vga_fb_adr;
    off = temp;
    flags = critb();
/* extra shift because even/odd text mode uses word clocking */
    outportb(_crtc_io_adr + 0, 12);
    outportb(_crtc_io_adr + 1, off >> 9);
    outportb(_crtc_io_adr + 0, 13);
    outportb(_crtc_io_adr + 1, off >> 1);
    move_csr(_curr_vc);
    crite(flags);
}
/*****************************************************************************
*****************************************************************************/
void sys_putch(console_t *con, unsigned char c)
{
    unsigned short att;

    att = (unsigned)con->attrib << 8;
/* state machine to handle the escape sequences
ESC */
    if(con->esc == 1)
    {
        if(c == '[')
        {
            con->esc++;
            con->esc1 = 0;
            return;
        }
        /* else fall-through: zero esc and print c */
    }
/* ESC[ */
    else if(con->esc == 2)
    {
        if(isdigit(c))
        {
            con->esc1 = con->esc1 * 10 + c - '0';
            return;
        }
        else if(c == ';')
        {
            con->esc++;
            con->esc2 = 0;
            return;
        }
/* ESC[2J -- clear screen */
        else if(c == 'J')
        {
            if(con->esc1 == 2)
            {
                fmemsetw(LINEAR_SEL, con->fb_adr,
                    ' ' | att,
                    _vc_height * _vc_width);
                con->csr_x=con->csr_y = 0;
            }
```

```
        }
/* ESC[num1m -- set attribute num1 */
        else if(c == 'm')
            set_attrib(con->esc1, con);
        con->esc = 0;   /* anything else with one numeric arg */
        return;
    }
/* ESC[num1; */
    else if(con->esc == 3)
    {
        if(isdigit(c))
        {
            con->esc2 = con->esc2 * 10 + c - '0';
            return;
        }
        else if(c == ';')
        {
            con->esc++; /* ESC[num1;num2; */
            con->esc3 = 0;
            return;
        }
/* ESC[num1;num2H -- move cursor to num1,num2 */
        else if(c == 'H')
        {
            if(con->esc2 < _vc_width)
                con->csr_x = con->esc2;
            if(con->esc1 < _vc_height)
                con->csr_y = con->esc1;
        }
/* ESC[num1;num2m -- set attributes num1,num2 */
        else if(c == 'm')
        {
            set_attrib(con->esc1, con);
            set_attrib(con->esc2, con);
        }
        con->esc = 0;
        return;
    }
/* ESC[num1;num2;num3 */
    else if(con->esc == 4)
    {
        if(isdigit(c))
        {
            con->esc3 = con->esc3 * 10 + c - '0';
            return;
        }
/* ESC[num1;num2;num3m -- set attributes num1,num2,num3 */
        else if(c == 'm')
        {
            set_attrib(con->esc1, con);
            set_attrib(con->esc2, con);
            set_attrib(con->esc3, con);
        }
        con->esc = 0;
        return;
    }
    con->esc = 0;
```

```
/* escape character */
    if(c == 0x1B)
    {
        con->esc = 1;
        return;
    }
/* backspace */
    if(c == 0x08)
    {
        if(con->csr_x != 0)
            con->csr_x--;
    }
/* tab */
    else if(c == 0x09)
        con->csr_x = (con->csr_x + 8) & ~(8 - 1);
/* carriage return */
    else if(c == '\r')  /* 0x0D */
        con->csr_x = 0;
/* line feed */
//  else if(c == '\n')  /* 0x0A */
//      con->csr_y++;
/* CR/LF */
    else if(c == '\n')  /* ### - 0x0A again */
    {
        con->csr_x = 0;
        con->csr_y++;
    }
/* printable ASCII */
    else if(c >= ' ')
    {
        unsigned long where;

        where = con->fb_adr + 2 *
            (con->csr_y * _vc_width + con->csr_x);
        pokew(LINEAR_SEL, where, c | att);
        con->csr_x++;
    }
    if(con->csr_x >= _vc_width)
    {
        con->csr_x = 0;
        con->csr_y++;
    }
    scroll(con);
/* move cursor only if the VC we're writing is the current VC */
    if(_curr_vc == con)
        move_csr(con);
}
/*****************************************************************************
*****************************************************************************/
static void puts(console_t *con, char *str)
{
    while(*str != '\0')
    {
        sys_putch(con, *str);
        str++;
    }
}
/*****************************************************************************
```

```
*****************************************************************************/
void init_console(void)
{
    unsigned short temp;
    console_t *con;

/* code to detect mono/color emulation
cobbled from info in Finn Thoegersen's VGADOC4
Mono is UNTESTED. */
    if((inportb(VGA_MISC_READ) & 0x01) != 0)
    {
        _crtc_io_adr = 0x3D4;   /* color */
        _vga_fb_adr = 0xB8000L;
    }
    else
    {
        _crtc_io_adr = 0x3B4;   /* mono */
        _vga_fb_adr = 0xB0000L;
    }
/* init VCs (different foreground color for each) */
    _curr_vc = con = _vc + 0;
    for(temp = 0; temp < MAX_VC; temp++)
    {
        char buf[64];

        con->attrib = temp + 1;
        con->fb_adr = _vga_fb_adr + _vc_width *
            _vc_height * 2 * temp;
/* ESC[2J is an ANSI-like escape to clear the screen */
        sprintf(buf, "\x1B[2J""this is VC#%u (of %u)\n",
            temp + 1, MAX_VC);
        puts(con, buf);
        con++;
    }
    select_vc(0);
}
```