Mika Lammi

KÄYTTÖJÄRJESTELMÄOHJELMOINTI

Tietojenkäsittelyn koulutusohjelma

2013

samk
Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

KÄYTTÖJÄRJESTELMÄOHJELMOINTI

Lammi, Mika
Satakunnan ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Tammikuu 2014
Ohjaaja: Grönholm, Jukka
Sivumäärä: 43
Liitteitä: 0

Asiasanat: käyttöjärjestelmä, bootloader, ohjelmointi

Tämän opinnäytetyön tarkoituksena oli suunnitella ja toteuttaa bootloader ja käyttöjärjestelmä nykypäiväiselle 64-bittiselle tietokoneelle. Toteutettu käyttöjärjestelmä on yleiskäyttöinen ja se sisältää tarvittavat perustoiminnot yksinkertaisten ohjelmien suorittamiseen.

Aluksi työssä esitellään projektin tavoitteita, vaadittavia työkaluja ja huomioon otettavia seikkoja. Seuraavaksi käydään läpi käyttöjärjestelmäohjelmointia x86-64-arkkitehtuuriin prosessorille ja esitellään merkittävimmät toteutetut ominaisuudet. Käsiteltyihin aihealueisiin kuuluu käyttöjärjestelmän käynnistys, muistinhallinta, keskeytykset, laitehallinta ja tiedostojärjestelmä. Lopuksi toteutusta käydään läpi jatkokehityksen kannalta.

Pääasiallisena tavoitteena oli tietokoneen toimintaperiaatteiden syvempi ymmärtäminen. Toteutettua käyttöjärjestelmää on kuitenkin yksinkertaisuudestaan huolimatta mahdollista hyödyntää myös käytännön tarpeisiin.

OPERATING SYSTEM PROGRAMMING

Lammi, Mika
Satakunnan ammattikorkeakoulu, Satakunta University of Applied Sciences
Degree Programme in Information and Communication Technology
January 2014
Supervisor: Grönholm, Jukka
Number of pages: 43
Appendices: 0

Keywords: operating system, bootloader, programming

The purpose of this thesis was to design and implement a bootloader and operating system for a modern 64-bit computer. The implementation is a general purpose operating system, and it contains required basic functionality to execute simple programs.

First, this thesis presents the project objectives, the required tools and factors to be considered. Next we will go through the programming of the operating system for x86-64 processor architecture, and introduce the most important features implemented. Treated subjects include startup, memory management, interrupts, device management, and file system. Finally, some of the post-development ideas are presented.

The main goal of the project was to gain deeper understanding of the inner workings of a computer. However, implemented operating system can be utilized for practical needs despite its simplicity.

SISÄLLYS

1 JOHDANTO

Oman käyttöjärjestelmän tekeminen on melko harvinaista, sillä käyttöjärjestelmiä on tarjolla runsaasti eri tarpeisiin. Yleisesti käytössä olevat käyttöjärjestelmät ovat hyvin monipuolisia ja soveltuvat moneen käyttötarkoitukseen. Monipuolisuus tuo mukanaan kuitenkin myös heikkouksia, kuten suuremmat laitevaatimukset.

Käyttöjärjestelmiä tehdään nykypäivänä usein harrastustoimintana oppimistarkoituksessa. Käyttöjärjestelmän suunnittelu ja ohjelmointi vaativat prosessorin ja tietokoneeseen liitettyjen laitteiden syvää tuntemusta, jota voi myöhemmin hyödyntää esimerkiksi laiteajureiden suunnittelussa. Monet sovellusohjelmoinnissa käytetyt toiminnot ovat toteutettuna järjestelmätasolla, jolloin käyttöjärjestelmän toiminnan ymmärtäminen auttaa myös tekemään tehokkaampia sovelluksia.

Oman käyttöjärjestelmän kehittäminen voi olla tarpeellista suorituskyvyn maksimoimiseksi. Yksittäiseen tarkoitukseen suunnitellut käyttöjärjestelmät ovat usein tehokkaampia, kuin moneen asiaan soveltuvat. Suurta laskentatehoa vaativa ohjelma voi olla kannattavampaa tehdä omana käyttöjärjestelmänään, mikäli etukäteen on tiedossa, ettei laskenta-aikaa ole tarkoitus käyttää muihin tehtäviin. Virtualisoinnin myötä kehitysprosessi helpottuu, eikä valmistunut käyttöjärjestelmä vaadi omaa fyysistä laitettaan, joten omia käyttöjärjestelmiä voidaan suorittaa tavallisilla kotikoneilla tai yritysten virtuaalipalvelimilla.

Kaikille prosessoreille ei välttämättä ole olemassa yleiskäyttöistä käyttöjärjestelmää, eikä sellainen useinkaan ole edes tarpeen. Varsinkin monet harraste-elektroniikassa käytetyt ohjelmoitavat mikropiirit ja yhden piirilevyn tietokoneet ohjelmoidaan sen mukaan, mitä laitteita niihin on tarkoitus liittää.

Tässä opinnäytetyössä keskitytään lähinnä omaan toteutukseen, eikä niinkään oteta kantaa olemassa oleviin käyttöjärjestelmiin. Toteutettu käyttöjärjestelmä on yleiskäyttöinen, nykypäiväiselle perustietokoneelle suunnattu käyttöjärjestelmä. Aihealueen laajuuden vuoksi toteutus sisältää vain tärkeimmät ominaisuudet ohjelmien suorittamisen mahdollistamiseksi. Työssä esitellään käyttöjärjestelmän käynnistäminen,

muistinhallinta, keskeytyksien käsittely, laitehallinta ja tiedostojärjestelmä. Ominaisuudet esitellään ennen kaikkea ohjelmoijan näkökulmasta.

2 PROJEKTIN TAVOITTEET

Käyttöjärjestelmän laitevaatimukset tulisi olla matalat, jotta sitä voidaan käyttää vanhoissa tietokoneissa. Tilavaatimuksien ollessa matalat, koko käyttöjärjestelmä mahtuu helposti kerralla muistiin, joka mahdollistaa hyvän suorituskyvyn ja yksinkertaistaa käyttöjärjestelmän rakennetta. Prosessori vaatimuksena on x86-64-arkkitehtuurin prosessori, joka on nykypäivänä yleisesti käytetty tavallisissa kotikoneissa.

Käyttöjärjestelmän ja siinä ajettavien ohjelmien on pystyttävä varaamaan ja vapauttamaan muistia dynaamisesti. Pitkäaikaisempaa tiedon varastointia varten käytetään tiedostoja, joita voidaan luoda, kirjoittaa, lukea ja poistaa. Tekstin tulostus näytölle ja syötteiden luku näppäimistöltä kuuluu myös vaadittaviin perustoimintoihin.

Lähdekoodi on oltava selkeää ja toteutetut algoritmit yksinkertaisia, jotta käyttöjärjestelmän myöhempi jatkokehitys olisi mahdollista. Monimutkaisempien algoritmien ja menetelmien avulla saatava suorituskyky ei saa mennä ymmärrettävämmän toteutuksen edelle. Tietokoneen resursseja on kuitenkin käytettävä tehokkaasti, jotta oman käyttöjärjestelmän tekemisestä olisi myös käytännön hyötyä.

3 TYÖKALUT

3.1 Ohjelmointikielet

Tyypillisin ohjelmointikielien yhdistelmä käyttöjärjestelmäohjelmoinnissa on Assembly, C ja C++. Assembly -kielen käyttö käyttöjärjestelmäohjelmoinnissa on usein

suotavaa ja joissakin tilanteissa jopa pakollista. Assembly soveltuu hyvin matalan tason ohjelmointiin, jolloin ollaan suoraan tekemissä laitteiston kanssa. Korkeamman tason kielet taas soveltuvat paremmin monimutkaisempien kokonaisuuksien toteuttamiseen, joissa laitteiston kanssa kommunikointiin käytetään erikseen Assemblyllä toteutettuja funktioita tai kirjoittamalla Assembly-osuus suoraan koodin sekaan inline-assemblynä, mikäli kääntäjä sellaista tukee.

Assemblyllä kirjotetuista funktioista ja proseduureista on mahdollista tehdä hyvin optimoituja. Optimointi tapahtuu kuitenkin hyvin usein koodin luettavuuden kustannuksella, joten siihen ei ole järkevää panostaa ennen kuin sille on oikeasti tarvetta. Korkeamman tason ohjelmointikielien kääntäjät sisältävät lähes poikkeuksetta optimoijan, jolloin kääntäjä osaa tyypillisesti tehdä tehokkaampaa koodia, kuin itse Assemblyllä kirjoitettuna. Uutta käyttöjärjestelmää ohjelmoitaessa ei kuitenkaan aina voida täysin luottaa optimoijaan, sillä optimoijat saattavat tehdä käyttöjärjestelmäkohtaisia ratkaisuja, jotka eivät olekaan uuden käyttöjärjestelmän kannalta tehokkaita tai ne eivät yksinkertaisesti toimi. Ideaalitilanteessa, käyttöjärjestelmäkehityksen ollessa tarpeeksi pitkällä, itse kääntäjä voidaan kääntää uudelle käyttöjärjestelmälle. Alkuvaiheessa koodi voidaan kääntää jollekin olemassa olevalle käyttöjärjestelmälle, kuitenkin rajoittamalla optimointia tai jättämällä se kokonaan pois.

Assemblyn heikkoutena voidaan pitää sen vaikealukuisuutta verrattuna esimerkiksi C-kieleen. Yksinkertaisistakin proseduureista saattaa tulla melko pitkiä, jolloin koodin kulkua on vaikea seurata. Luettavuutta voidaan runsaan kommentoinnin lisäksi parantaa käyttämällä makroja, jotka ovat eräänlaisia inline–funktioita. Usein toistuvat kohdat voidaan kirjoittaa kerran yhteen paikkaan makrona, jota kutsuttaessa makron sisältö kopioituu sitä kutsuvaan kohtaan.

Yksi C-kielen vahvuuksista sovellusohjelmoinnissa on kattava standardikirjasto, joka sisältää runsaasti erilaisia valmiita funktioita merkkijonon käsittelystä muistinhallintaan. Uutta käyttöjärjestelmää ohjelmoitaessa niitä ei kuitenkaan voida käyttää, sillä standardifunktiot ovat käyttöjärjestelmäkohtaisia, jonka vuoksi ne eivät ole käytettävissä ennen kuin ne on uuteen käyttöjärjestelmään toteutettu. Tästä syystä C-kielen käytön hyödyt perustuvat alkuvaiheessa lähinnä selkeämpään syntaksiin ja kompaktimpaan koodiin.

Assembly koodi on aina prosessoriarkkitehtuurikohtaista, jolloin samaa koodia ei voida käyttää kaikissa prosessoreissa. Esimerkiksi x86-64 prosessoreille kirjoitettu 64-bittinen koodi ei toimi 32-bittisissä prosessoreissa. Korkeamman tason kielissä tätä ongelmaa ei ole, sillä kielen syntaksi pysyy samana prosessorista riippumatta. Jatkokehityksen kannalta korkeamman tason kieli mahdollistaa käyttöjärjestelmän kääntämisen muille prosessoreille vähemmällä työmäärällä. Prosessorikohtaisuus mahdollistaa kuitenkin sellaisten käskyjen suorituksen, joita muuten ei olisi mahdollista käyttää (Taulukko 1).

Taulukko 1. Muutamia x86-64–prosessorin käskyjä, joiden käyttö vaatii Assemblyä (mukaillen BrokenThorn Entertainment 2008)

Käsky	**Käyttötarkoitus**
CPUID	Prosessorin ominaisuuksien tunnistus
HLT	Prosessorin pysäytys
LGDT	GDT-osoitteen asetus
LIDT	IDT-osoitteen asetus
IN	Luku portista
OUT	Kirjoitus porttiin
RDMSR	Mallikohtaisen rekisterin luku
WRMSR	Mallikohtaisen rekisterin kirjoitus

Jotta Assemblyä ja C:tä voitaisiin käyttää sekaisin, on käytettävä yhtenäistä kutsukäytäntöä (engl. calling convention). Kutsukäytäntö määrittelee mitä toimenpiteitä vaaditaan proseduurin kutsujalta ja kutsuttavalta proseduurilta. Kutsukäytännöt eroavat toisistaan mm. siinä, miten parametrit välitetään, miten paluuarvo palautetaan ja mitä rekistereitä voidaan käyttää vapaasti tallentamatta niiden alkuperäistä sisältöä. Tämän opinnäytetyön toteutuksessa on käytetty System V ABI – kutsukäytäntöä (Taulukko 2), joka on käytössä myös Linux ja Mac OS X - käyttöjärjestelmissä. (The University of Chicago - The Department of Computer Science 2009.)

Taulukko 2. Rekisterien käyttö x86-64 System V ABI –kutsukäytännössä (mukaillen The University of Chicago - The Department of Computer Science 2009)

Rekisteri	**Tallennettava**	**Käyttötarkoitus**
RAX	ei	Paluuarvo
RBX	kyllä	Yleiskäyttöinen
RCX	ei	Neljäs parametri
RDX	ei	Kolmas parametri
RSP	ei	Pino-osoitin
RBP	kyllä	Pino-osion alkuosoite
RSI	ei	Toinen parametri
RDI	ei	Ensimmäinen parametri
R8	ei	Viides parametri
R9	ei	Kuudes parametri
R10-11	ei	Yleiskäyttöinen
R12-15	kyllä	Yleiskäyttöinen

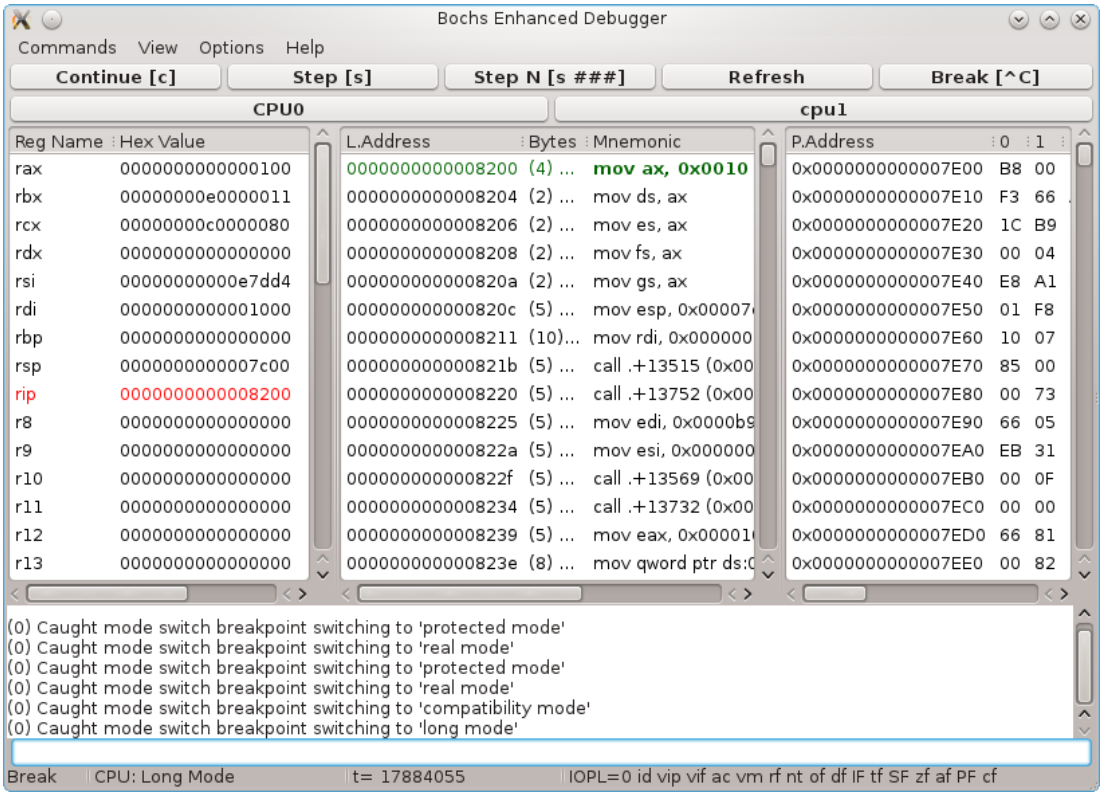
3.2 Assembler

Assemblerina projektissa toimii NASM (Netwide Assembler), joka on makroassembleri, eli se tukee makroja. Makrotuki on NASM:ssa erittäin hyödyllinen apuväline, jotta koodi pysyisi kompaktina ja helppolukuisena. Makrojen avulla voidaan toteuttaa jopa korkeamman tason kielien sisältämiä ehtolauseita ja toistorakenteita. Syntaksina NASM:ssa käytetään Intelin kehittämää syntaksia. (NASM 2013.)

3.3 Emulaattori

Emulaattori (virtuaalikone) on tärkeä työkalu käyttöjärjestelmän ohjelmoinnissa, jotta käyttöjärjestelmän testaamiseen ei tarvita fyysistä tietokonetta. Käyttöjärjestelmäohjelmointiin soveltuvassa emulaattorissa on oltava debuggaus-mahdollisuudet, jotta virheiden löytäminen olisi helpompaa tai ylipäätään mahdollista.

Projektissa käytetyllä Bochs-emulaattorilla voidaan emuloida 386, 486 ja x86-64 arkkitehtuurin prosessoreja. Se sisältää graafisen käyttöliittymän, jossa voi mm. tarkkailla prosessorin rekisterejä ja pysäyttää suorituksen haluttuun kohtaan (Kuva 1). Koodinäkymän avulla suoritettavan kohdan näkee Assembly-kielelle käännettynä. Lisäksi Bochs-sisältää virtuaalisen BIOS:n ja monia virtuaalisia laitteita CD-asemasta verkkokorttiin. (Lawton.)



Kuva 1. Bochs:n debuggaus näkymä

4 KERNEL TYYPIT

Käyttöjärjestelmän ytimen, eli kernelin, tyyppi määrittelee käyttöjärjestelmän sisältämän toiminnallisuuden ja rakenteen. Kernelit voidaan karkeasti jakaa kahteen pääryhmään: monoliittiset kernelit ja micro kernelit. Monet käyttöjärjestelmäytimet ovat kuitenkin näiden päätyyppien eriasteisia sekoituksia.

Monoliittisessa kernelissä kaikki toiminnallisuudet on sisäänrakennettu käyttöjärjestelmään yhtenä kokonaisuutena. Monoliittinen kernel on kooltaan suuri ja sen rakenne on hyvin staattinen. Monoliittisen kernelin vahvuuksiin luetaan suorituskyky ja yksinkertaisuus. (Malhar 2010.)

Micro kernelissä on toteutettuna vain välttämättömät toiminnallisuudet. Laitteidenhallinta ja erilaiset palvelut ovat omia prosessejaan, jotka toimivat erillään käyttöjärjestelmästä. Micro kernelin vahvuuksina on mm. pienempi koko ja parempi ylläpidettävyys. (Malhar 2010.)

4.1 Muistikartta

Muistissa olevien alueiden sijaintia on hyvä hahmotella jo suunnitteluvaiheessa, jotta mahdollisia virheitä on helpompi etsiä ajonaikaisesti. Tietokoneen käynnistyessä muisti ei ole tyhjillään, vaan se sisältää monia erityisiin tarkoituksiin varattuja alueita. Käyttöjärjestelmän käyttämät alueet voidaan ainakin pääpiirteisesti suunnitella varattujen alueiden mukaan (Taulukko 3).

Taulukko 3. Muistikartta

Osoite	**Sisältö**
0x000000	Käyttöjärjestelmän pino.
0x007C00	Bootloaderin ensimmäinen vaihe.
0x007E00	Bootloaderin toinen vaihe.
0x008000	Käyttöjärjestelmäkoodia.
0x040000	E820-muistikartta.
0x050000	Fyysisen muistin bittikartta.
0x070000	Osoitemuunnostaulukot.
0x080000	Varattuja alueita.
0x0B8000	Näyttömuisti ja varattuja alueita.
0x100000	Varattuja ja vapaita alueita.

5 BOOTLOADER

Bootloader on ohjelma, jonka tarkoituksena on ladata käyttöjärjestelmä muistiin. Sen ei välttämättä tarvitse olla osa käyttöjärjestelmää, vaan se voi olla täysin erillinen projekti. Monet käyttöjärjestelmät käyttävät ulkopuolisen tahon kehittämää bootloaderia, jotka ovat toiminnallisuuksiltaan laajoja ja monikäyttöisiä. Kehittyneet bootloaderit sisältävät esimerkiksi valikon, josta voi valita käynnistettävän käyttöjärjestelmän, mikäli tietokoneeseen on asennettu useita käyttöjärjestelmiä. Tämän opinnäytetyön toteutuksessa on kuitenkin tehty oma käyttöjärjestelmäkohtainen bootloader. Bootloader koostuu vähintäänkin boot sectorista, mutta se voi sisältää myös useampia vaiheita monipuolisempien toimintojen tukemiseksi.

x86-64 -arkkitehtuurin prosessori ei ole tietokoneen käynnistyessä 64-bittisessä tilassa, vaan 8086-tilassa (nk. real mode). Tässä tilassa vain osa käskyistä ja rekistereistä on käytössä. Tämän tarkoituksena on taata yhteensopivuus vanhojen 16- ja 32-bittisten prosessorien kanssa. Periaatteessa esimerkiksi MS-DOS -käyttöjärjestelmän asentaminen ja käyttö pitäisi toimia normaalisti x86-64 prosessoreissa. (Chourdakis 2009.)

5.1 Boot sector

Tietokoneen käynnistäessä boot sector, kiintolevyn ensimmäinen sektori, luetaan muistiin BIOS:n toimesta (McGuigan 2013). Boot sector on muiden sektorien tavoin kooltaan 512 -tavua, joten sen sisältämä toiminnallisuus on hyvin rajallista. Tilaa on kuitenkin riittävästi tärkeimmän tehtävän hoitamiseen: bootloaderin toisen vaiheen lukemiseen kiintolevyltä muistiin. Levyn lukemiseen voidaan käyttää BIOS:n tarjoamia keskeytyksiä (Kuva 2). Toista vaihetta siirrytään suorittamaan hyppäämällä muistiosoitteeseen, johon vaihe luettiin.

```
   ; Käynnistyksen yhteydessä dl sisältää aseman numeron
   mov   [drive], dl

   ; Tarkistetaan onko LBA tuettu
   mov   ah, 0x41           ; BIOS-tarkistusfunktion numero.
   mov   bx, 0x55AA         ; BIOS-tarkistusfunktion toinen numero.
   mov   dl, [drive]        ; Aseman numero dl-rekisteriin.
   int   0x13               ; Tarkistetaan LBA-tuki.
   jc    .no_int13ext       ; Carry asetettu, jos ei tuettu.

   ; LBA-luku
   mov   si, LBA_packet     ; Osoitetaan ds:si LBA-pakettirakenteeseen.
   mov   ah, 0x42           ; BIOS-lukufunktion numero.
   mov   dl, [drive]        ; Aseman numero dl-rekisteriin.
   int   0x13               ; Luetaan levy.
   jc    .error_reading_lba ; Carry asetettu virhetilanteessa.
   ; ......

drive      db 0
align 4
LBA_packet:
   db 16       ; Tämän paketin koko tavuina.
   db 0        ; Aina 0.
   dw 62       ; Sektorien määrä.
   dw 0x7E00   ; Kohdeosoite (segmentin sisällä).
   dw 0x0000   ; Kohdeosoite (segmentti).
   dd 266      ; Ensimmäisen luettavan sektorin numero (alempi osa).
   dd 0        ; Sektorin numero (ylempi osa).
```

Kuva 2. Levyn lukeminen boot sectorissa

5.2 Toinen vaihe

Bootloaderin toisen vaiheen tehtävä on valmistella siirtyminen varsinaiseen käyttöjärjestelmän ytimeen. Tyypillisesti tämä tarkoittaa samalla sivutuksen päälle asettamista ja siirtymistä joko 32- tai 64-bittiseen tilaan. BIOS-keskeytykset ovat vielä käytössä ennen 64-bittiseen tilaan siirtymistä, joten niitä on syytä hyödyntää tässä vaiheessa, jotta käyttöjärjestelmän ei tarvitsisi palata 8086-tilaan myöhemmin. BIOS-keskeytyksien avulla monet tiedot ovat helposti saatavilla, kun taas 64-bittisessä tilassa tietojen kerääminen voi vaatia laitteiston kanssa kommunikoimista porttien välityksellä tai vaikeasti löydettävien rakenteiden lukemista muistista.

5.2.1 Varatut muistialueet

Muistissa on valmiina useita tietorakenteita, joita tietokoneen eri laitteet tarvitsevat. Käyttöjärjestelmän on tiedettävä missä päin muistia varatut muistialueet sijaitsevat ja kuinka suuria ne ovat. Mikäli käyttöjärjestelmä kirjoittaa varatulle muistialueelle, tulokset voivat olla arvaamattomia ja ongelmat vaikeasti selvitettäviä. Tyypillinen virhe on esimerkiksi näyttömuistin päälle kirjoittaminen, jolloin ruudulle voi ilmestyä satunnaisia merkkejä eri väreissä.

Nykyaikaisessa tietokoneessa varmin tapa löytää varatut muistialueet on käyttää BIOS keskeytystä numero 15 funktiolla 0xE820 (Kuva 3). Funktio palauttaa BIOS:sta riippuen 20 tai 24 tavuisen elementin, joka kuvaa yhtä muistialuetta. Elementti sisältää alueen alkuosoitteen, koon ja tyypin. Tyyppikenttä kuvaa onko muistialue vapaa, käytössä vai sisältääkö se tietoa, jonka käyttöjärjestelmä saa vapaasti ylikirjoittaa. Harvinaisemmissa 24 tavuisissa elementeissä on lisäksi ignore-bitti, jonka ollessa asetettuna elementti on jätettävä huomiotta. Kaikkien elementtien saamiseksi funktiota on kutsuttava monta kertaa, kunnes EBX-rekisteri on nolla tai carry-lippu on asetettu. Mikäli 0xE820-funktio ei ole tuettu, vanhemmat BIOS:t sisältävät alkeellisemmat 0xE802- ja 0xE801 funktiot, joita voidaan käyttää samaan tarkoitukseen. (Brown 2013.)

```
; Muistikartan haku
; es:di : Osoite johon muistikartta tallennetaan
; palauttaa es:di : osoite viimeiseen elementtiin
get_memory_map:
   xor  ebx, ebx     ; Nollataan ebx (aloitetaan kartan luku).
   int15_e820        ; Kutsutaan keskeytystä.
   jc   .failed      ; Jos carry-lippu on 1 tai ebx on 0
   test ebx, ebx     ; ensimmäisellä kutsulla, tapahtui virhe.
   jz   .failed
   cmp  cl, 20       ; Tarkisetaan elementin koko.
   jbe  .next        ; 20 tavuisilla elementeillä ohitetaan tarkistus.
   test byte [es:di + MMAPOFFS_EXT], 1 ; Tarkistetaan onko ignore-bitti 1.
   jnz  .ignored     ; Ohitetaan elementin kirjoitusosoitteen kasvattaminen.
.next:
   add  di, 24       ; Päivitetään kirjoitusosoitetta.
.ignored:
   int15_e820        ; Kutsutaan keskeytystä.
   jnc  .test_ignore ; Jos carry on 0, tarkistetaan ignore-bitti.
   mov  ebx, 0       ; Jos carry on 1, nollataan ebx lopetuksen merkiksi.
.test_ignore:
   cmp  cl, 20       ; Tarkisetaan elementin koko.
   jbe  .no_ignore   ; 20 tavuisilla elementeillä ohitetaan tarkistus.
   test byte [es:di + MMAPOFFS_EXT], 1 ; Tarkistetaan onko ignore-bitti 1.
   jz   .no_ignore   ; Siirrytään tarkistamaan onko elementti viimeinen.
   test ebx, ebx     ; Tarkistetaan onko viimeinen elementti ohitettava.
   jnz  .ignored     ; Ohitetaan ja siirrytään seuraavaan.
   jmp  .lastignored ; Viimeinen ohitettiin, siirrytään korjaamaan osoite.
.no_ignore:
   test ebx, ebx     ; Tarkistetaan onko ebx 0.
   jnz  .next        ; Jos ei, elementtejä on vielä lisää.
   jmp  .done        ; Haku on valmis.
.lastignored:
   sub  di, 24      ; Korjataan es:di mikäli viimeinen elementti ohitettiin.
.done:
   clc               ; Nollataan carry (= ei virhettä).
   ret
.failed:
   stc               ; Asetetaan carry virhetilanteessa.
   ret

%macro int15_e820 0
   mov  ecx, 24          ; Elementin maksimikoko = 24.
   mov  eax, 0x0000E820 ; BIOS-keskeytyksen funktio.
   mov  edx, 0x534D4150 ; Parametrina 'SMAP'.
   int  0x15             ; Kutsutaan BIOS-keskeytystä.
%endmacro
```

Kuva 3. E820-muistikartan haku

5.2.2 Virtuaalimuistin sivutusmekanismi

Muistissa olevia alueita kutsutaan sivuiksi. x86-64 arkkitehtuurin prosessoreissa sivujen kooksi voidaan valita 4 kilotavua, 2 megatavua tai 1 gigatavu (AMD 2012a, 130). Sivujen koko määrittää kuinka tehokkaasti järjestelmän muisti voidaan hyödyntää ja miten tarkasti se voidaan jakaa virtuaalisiin alueisiin. Suurella sivukoolla osa muistista voi mennä hukkaan tilanteessa, jossa varattava muistin määrä on merkittävästi sivukokoa pienempi.

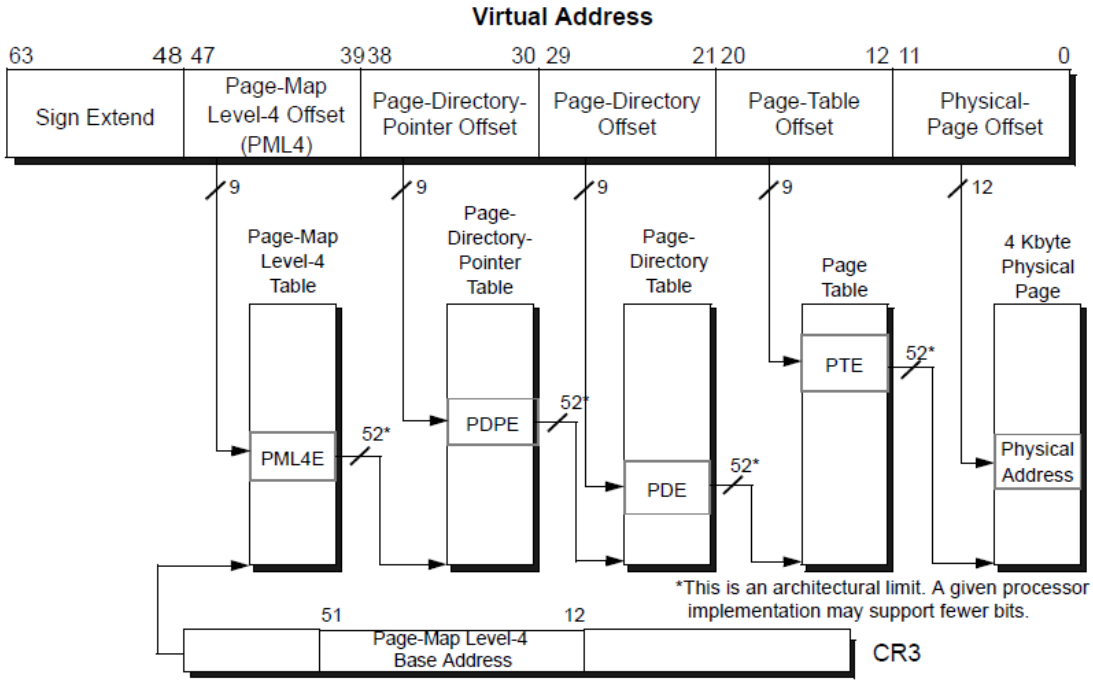
Jotta x86-64-arkkitehtuurin prosessori voisi siirtyä 64-bittiseen tilaan, on sivutettu virtuaalimuisti asetettava päälle. Käyttöjärjestelmä ei voi tehdä viittauksia suoraan fyysiseen muistiin, vaan sen on asetettava virtuaalisia sivuja osoittamaan fyysisiin muistialueisiin. Sivutusmekanismin ansiosta järjestelmä voi näennäisesti sisältää enemmän muistia, kuin oikeasti laitteessa on saatavilla. Useampi virtuaaliosoite voi viitata samaan fyysiseen osoitteeseen, jolloin samaan muistialueeseen pääsee käsiksi monesta eri osoitteesta.

Virtuaaliosoitteet ovat x86-64-arkkitehtuurin prosessoreissa 64-bittisiä, josta tyypillisesti ensimmäiset 48–bittiä ovat merkitseviä, vaikka tietokone ei 256 teratavua fyysistä muistia sisältäisikään. Merkitsevien bittien määrä vaihtelee hieman prosessorimallien välillä, mutta tällä ei juurikaan ole käytännön merkitystä osoiteavaruuden laajuuden vuoksi. Ei-merkitsevät bitit on aina asetettava samaksi, kuin eniten merkitsevä bitti, jotta niitä ei voisi käyttää omiin tarkoituksiin rikkoen yhteensopivuuden laitteiden kanssa, jotka sisältävät eri määrän merkitseviä bittejä.

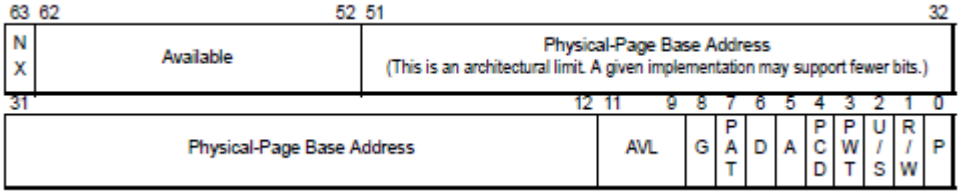
Sivutetun muistin hyödyt tulevat esiin etenkin moniajoa tukevissa käyttöjärjestelmissä. Eri prosessit vaativat usein yksityisen muistialueen, johon muut prosessit eivät saa päästä käsiksi. Suoritettava prosessi vaihtuu jatkuvasti, jolloin muistialueen talteen kirjoittaminen esimerkiksi kiintolevylle olisi hyvin raskas operaatio. Sivutetussa muistissa vain virtuaaliosoitteita muutetaan siten, että kulloinkin suoritettavana oleva prosessi näkee vain sille tarkoitetut sivut. Sivutettu muisti tehostaa myös jaettujen resurssien käyttöä. Jaettu resurssi voidaan lukea kerran muistiin ja jakaa monen prosessin käyttöön asettamalla osa prosessin sivuista osoittamaan yhteiseen fyysiseen muistialueeseen. Näin samaa resurssia voidaan käyttää useasta paikasta ilman, että muisti sisältäisi useita kopioita tarvittavasta resurssista. Resurssin osoite voi myöskin vaihdella eri prosessien välillä.

Käytettäessä neljän kilotavun kokoisia sivuja, virtuaaliosoite koostuu viidestä kentästä: PML4 (Page-Map Level-4), PDP (Page-Directory-Pointer), Page-Directory (PD), PT (Page-Table) ja sivun sisäisestä fyysisestä osoittimesta (Kuva 4). Nämä kentät ovat indeksejä tauluihin, joiden avulla prosessori muuntaa virtuaaliset osoitteet fyysisiksi osoitteeksi. PML4-taulun osoite on tallennettuna prosessorin CR3-rekisteriin. PML4-taulu sisältää osoitteita PDP-tauluihin, joka taas sisältää osoitteita PD-

tauluihin. PD-elementit viittaavat PT-tauluihin, jotka sisältävät sivujen fyysisiä osoitteita (Kuva 5), joihin lisätään virtuaaliosoitteen sivun sisäinen osoite lopullisen fyysisen osoitteen aikaansaamiseksi. (AMD 2012a, 131.)



Kuva 4. Virtuaalisen osoitteen muunnos fyysiseksi osoitteeksi (AMD 2012a, 132)



Kuva 5. Page Table -elementti (AMD 2012a, 133)

Yksinkertaisimmillaan sivutustaulut voidaan rakentaa siten, että virtuaaliosoitteet vastaavat täysin fyysisiä osoitteita, jolloin muistinkäsittely helpottuu. Tästä tekniikasta käytetään nimitystä identity-mapping (Kuva 6). Käyttöjärjestelmälle on myös annettava mahdollisuus muuttaa taulujen sisältöä myöhemmässä vaiheessa, esimerkiksi muistinvarauksen yhteydessä. Tämä onnistuu asettamalla osa taulujen elementeistä osoittamaan itse tauluihin, jotta niihin voidaan myöhemmin viitata.

```
    ; Asetetaan es:di osoittamaan PML4-tauluun.
    mov   ax, PML4_BASE >> 4  ; PML4-taulun segmentti.
    mov   es, ax              ; Asetetaan segmentti-rekisteri.
    xor   di, di              ; Nollataan indeksi.

    ; PML4-, PDP-, PD-elementtien luonti (1 kpl / taulu)
    mov   eax, PDP_BASE | PAGE_PRESENT | PAGE_WRITE ; PDP:n osoite ja liput.
    mov   [es:di], eax          ; Talletetaan elementti PML4-tauluun.
    mov   eax, PD_BASE | PAGE_PRESENT | PAGE_WRITE ; PD:n osoite ja liput.
    mov   [es:di + 0x1000], eax ; Talletetaan elementti PDP-tauluun.
    mov   eax, PT_BASE | PAGE_PRESENT | PAGE_WRITE ; PT:n osoite ja liput.
    mov   [es:di + 0x2000], eax ; Talletetaan elementti PD-tauluun.

    ; PT-elementtien luonti (512 kpl).
    mov   ax, PT_BASE >> 4    ; PT-taulun segmentti.
    mov   es, ax              ; Asetetaan segmentti-rekisteri.
    xor   di, di              ; Nollataan indeksi.
    mov   eax, dword PAGE_PRESENT | PAGE_WRITE ; Fyysisten sivujen liput.
.loop_pt:
    mov   [es:di], eax        ; Talletetaan elementti PT-tauluun.
    add   eax, 0x1000         ; Kasvatetaan fyysistä osoitetta.
    add   di, 8               ; Seuraavan PT-elementin kirjoitusosoite.
    cmp   eax, 0x00200000     ; Tarkistetaan onko osoite yli kahden megatavun.
    jb    .loop_pt            ; Jos ei, jatketaan elementtien kirjoittamista.
```

Kuva 6. Kahden ensimmäisen megatavun identity-mapping

5.2.3 64-bittiseen tilaan siirtyminen

Kun sivutustaulut on rakennettu ja käyttöjärjestelmää helpottavat tiedot on kerätty, voidaan siirtyä 64-bittiseen tilaan (nk. long mode) ja alkaa suorittamaan varsinaista käyttöjärjestelmä koodia. Prosessorin tilan vaihtamiselle on olemassa tarkat määritykset (AMD 2012a, 437). Virtuaalimuistin sivutus, suojattu tila ja long mode asetetaan päälle kirjoittamalla prosessorin CR0-, CR3- ja CR4-ohjausrekistereihin (Kuva 7). Samalla on ladattava GDT (Global Descriptor Table), joka sisältää valitsimet eri oikeustasoille (AMD 2012a, 73). GDT:n avulla käyttöjärjestelmä voi suorittaa ohjelmia rajoitetuin oikeuksin esimerkiksi rajoittamalla joidenkin muistialueiden käyttöä tai estämällä osan keskeytyksistä.

```
mov    eax, CR4_PAE | CR4_PGE ; PAE- (Physical Address Extensions) ja
mov    cr4, eax               ; PGE- (Page Global Enable) liput CR4-rekisteriin.
mov    eax, PML4_BASE         ; PML4-osoite.
mov    cr3, eax               ; Asetetaan PML4-osoite CR3-rekisteriin.
mov    ecx, 0xC0000080 ; Mallikohtaisen rekisterin valitsin (EFER MSR).
rdmsr                  ; Luetaan mallikohtainen EFER-rekisteri.
or     eax, EFER_LME   ; Asetetaan rekisterin LME-bitti (Long Mode Enable).
wrmsr                  ; Kirjoitetaan uusi arvo rekisteriin.
mov    ebx, cr0        ; Siirretään CR0-rekisteri ebx-rekisteriin.
or     ebx, CR0_PG | CR0_PE ; Asetetaan liput PG (Paging) ja
mov    cr0, ebx        ; PE (Protection Enabled) CR0-rekisteriin.
lgdt  [GDT.ptr] ; Ladataan GDT, joka sisältää määritykset eri oikeustasoille
; Siirrytään järjestelmätason 64-bittiseen koodiin:
jmp    GDT.code_0:long_mode_entry

ALIGN 4
GDT:                   ; Global Descriptor Table.
.null: equ $ - GDT     ; 1. descriptor on tyhjä (ns. null-descriptor).
   dq 0
.code_0: equ $ - GDT   ; Järjestelmätason koodi-descriptor.
   dGDT64_CODE 0, 1, 1, 0, 0
.data_0: equ $ - GDT   ; Järjestelmätason data-descriptor.
   dGDT64_DATA 1
.code_3: equ $ - GDT   ; Käyttäjätason koodi-descriptor.
   dGDT64_CODE 0, 1, 1, 3, 0
.data_3: equ $ - GDT   ; Käyttäjätason data-descriptor.
   dGDT64_DATA 1
```

Kuva 7. 64-bittiseen tilaan siirtyminen

6 MUISTINHALLINTA

Muistinhallinta on yksi käyttöjärjestelmän tärkeimmistä tehtävistä. Muistinhallinta sisältää sekä fyysisen muistin, että virtuaalimuistin käytön hallintaa. Tavoitteena on mahdollistaa muistin dynaaminen varaus ja vapautus. Dynaamisuudella tarkoitetaan sitä, että ohjelma voi varata tarvittavan määrän muistia ajon aikaisesti, välittämättä sen lopullisesta osoitteesta. Muistinhallinta varmistaa myös asianmukaisen käsittelyn tilanteessa, jossa muisti loppuu kesken.

6.1 Fyysinen muisti

Varattujen fyysisten muistialueiden seuraamiseen on kehitetty erilaisia algoritmeja. Eri algoritmit eroavat toisistaan lähinnä yksinkertaisuuden, nopeuden ja vaadittavan muistimäärän osalta.

Tyypillisin ratkaisu on käyttää bittikarttaan pohjautuvaa algoritmia. Muistiin varataan alue, jossa jokainen fyysisen muistin osio on kuvattuna yhdellä bitillä. Bitin ollessa yksi, alue on jo varattu muuhun käyttöön. Jos bitti on nolla, muistialue on vapaa ja se voidaan varata uuteen käyttöön asettamalla se ykköseksi. Vapaan alueen hakemisen nopeuttamiseksi useita bittejä voidaan tarkistaa kerralla (Kuva 8). Yksittäisen bitin kuvaaman osion suuruus on vakio koko bittikartassa. Osion suuruudeksi valitaan tyypillisesti korkeintaan muutama kilotavu, sillä kerralla varattavat muistimäärät ovat yleensä vähäisiä. Mitä pienempiä alueita bitit kuvaavat, sitä tarkemmin muistia voidaan hyödyntää, mutta itse bittikartan koko kasvaa vieden enemmän muistia. Algoritmin hyötyihin lukeutuu yksinkertaisuus ja hyvä suorituskyky.

```
; Yksittäisen nollan hakeminen bittikartasta
;  rdi : bittikartan osoite
;  rsi : bittikartan koko
;  palauttaa rax : löydetyn bitin järjestys numero (-1 jos ei löydy)
bitmap_getfree:
  xor  rax, rax    ; Nollataan paluuarvo.
  xor  rcx, rcx    ; Nollataan rcx.
  not  rcx         ; Asetetaan kaikki rcx:n bitit ykköseksi.
  add  rsi, rdi    ; Bittikartan raja (bittikartan osoite + koko).
  jmp  .first      ; Aloitetaan etsiminen.
.next:
  add  rdi, 8      ; Päivitetään tarkistettava osoite.
  cmp  rdi, rsi    ; Tarkistetaan onko osoite bittikartan ulkopuolella.
  jae  .limit_reached ; Jos on, lopetetaan etsintä.
  add  rax, 64     ; Päivitetään paluuarvoa.
.first:
  mov  rdx, [rdi] ; Luetaan 64 bittiä.
  cmp  rdx, rcx    ; Tarkistetaan onko kaikki 64 bittiä 1.
  je   .next       ; Jos on, siirrytään suoraan seuraaviin tavuihin.
.bitcheck:
  test dl, 1       ; Tarkistetaan bitti.
  jz   .found      ; Lopetetaan etsintä, jos bitti on nolla.
  shr  rdx, 1      ; Luetaan seuraava bitti.
  inc  rax         ; Päivitetään paluuarvoa.
  jmp  .bitcheck   ; Siirrytään tarkistamaan seuraava bitti.
.found:
  ret              ; rax sisältää paluuarvon.
.limit_reached:
  xor  rax, rax    ; Nollataan paluuarvo.
  not  rax         ; Käännetään bitit (= -1).
  ret
```

Kuva 8. Nollan hakeminen bittikartasta

Bittikarttaan pohjautuvaa algoritmia voidaan tehostaa toteuttamalla puumainen rakenne, jossa useampaa bittiä kuvaa yksi ylemmän tason bitti. Näin bittikartan läpikäyminen nopeutuu tilanteissa, joissa etsitään suuria vapaita muistialueita ja monta peräkkäistä aluetta on varattuna. Puurakenteen mukanaan tuomat uudet bitit vievät

tilaa, joten käytettävän muistimäärän vaihto nopeuteen ei välttämättä ole hyvä ratkaisu järjestelmissä, joissa muistia on hyvin rajoitetusti.

Bittikarttapohjaisen toteutuksen lisäksi toinen suosittu tapa on käyttää pino- tai listapohjaista ratkaisua. Vapaat muistialueet sisällytetään taulukkoon, josta selviää alueen osoite ja koko. Muistia varattaessa taulukosta luetaan elementtejä, kunnes riittävän suuri alue löytyy. Sopivan elementin löytyessä sen kokoa pienennetään varattavan muistin määrällä ja osoite päivitetään vastaamaan seuraavaa vapaata osoitetta. Kun muistia vapautetaan, vierekkäisiä alueita kuvaavat elementit pyritään yhdistämään tilan säästämiseksi ja seuraavan haun nopeuttamiseksi. Pino- ja listapohjaisten algoritmien etuina ovat vähäinen tilan tarve ja hyvä suorituskyky, mikäli muistialueiden pirstaloituminen on vähäistä.

6.2 Virtuaalimuisti

Virtuaalimuistin hallinta perustuu PML4-, PDP-, PD- ja PT-osoitemuunnostaulukoiden manipulointiin. Virtuaalisia sivuja voidaan asettaa viittamaan fyysisiin muistialueisiin kirjoittamalla näihin tauluihin (Kuva 9). Samalla kun uusia virtuaaliosoitteita määritellään, osoitemuunnostaulukoille on tarvittaessa varattava lisää tilaa. Esimerkiksi ensimmäisen PD-taulun elementin osoittama PT-taulu määrittelee virtuaaliosoitteet 0x1FF000 asti, joten jos halutaan määrittää fyysinen alue osoitteelle 0x200000, on luotava toinen PT-taulu ja merkittävä tämän osoite PD-tauluun.

```
;  rdi : virtuaaliosoite
;  rsi : fyysinen osoite ja liput (PT-elementti)
map_page:
  push  rsi                   ; Talletetaan fyysinen osoite pinoon.
  mov   rsi, cr3              ; Luetaan CR3.
  mov   rax, CR3_PML4ADDRMASK ; Peite PML4 osoitteelle.
  and   rsi, rax              ; Luetaan PML4:n osoite.
  mov   cl, 39                ; Bittien shiftaus-määrä indeksin saamiseksi.
.next:
  mov   rax, rdi              ; Virtuaaliosoite.
  shr   rax, cl               ; Shiftataan bittejä.
  and   rax, 0x1FF            ; Indeksi taulun sisällä.
  lea   rax, [rsi+rax*8]      ; Elementin osoite (rax).
  mov   rdx, [rax]            ; Elementti (rdx).
  test  dl, PAGE_PRESENT      ; Testataan onko sivu olemassa.
  jnz   .is_present           ; Jos on, ohitetaan muistinvaraus.
  mpush rax, rcx, rsi, rdi    ; Talletetaan rekisterit pinoon.
  mov   rdi, [SYSVAR.physmem_bm_addr] ; Fyysisen muistibittikartan osoite.
  mov   rsi, [SYSVAR.physmem_bm_size] ; Bittikartan koko.
  call  bitmap_getfree        ; Haetaan vapaa muistialue.
  mov   rsi, rax              ; Talletetaan bitin numero (rsi).
  shl   rax, 12               ; Muunnetaan bitin numero osoitteeksi.
  push  rax                   ; Talletetaan osoite pinoon.
  mov   rdi, [SYSVAR.physmem_bm_addr] ; Fyysisen muistibittikartan osoite.
  call  bitmap_setbit         ; Asetetaan alue varatuksi.
  pop   rdx                   ; Palautetaan alueen osoite pinosta.
  mpop  rax, rcx, rsi, rdi    ; Palautetaan rekisterit pinosta.
  or    rdx, PAGE_PRESENT     ; Asetetaan present-bitti.
  mov   [rax], rdx            ; Päivitetään taulun elementti.
.is_present:
  and   dx, 1111100000000000b ; Peite seuraavan taulun osoitteelle.
  mov   rsi, rdx              ; Taulun osoite (rsi).
  sub   cl, 9                 ; Shiftataan seuraavan taulun indeksiin.
  cmp   cl, 12                ; Tarkistetaan ollaanko jo PT-tasolla.
  jg    .next                 ; Jos ei, siirrytään seuraavaan tauluun.
  mov   rax, rdi              ; Virtuaaliosoite (rax).
  shr   rax, cl               ; Shiftataan indeksin saamiseksi.
  and   rax, 0x1FF            ; Peite indeksille.
  lea   rax, [rsi+rax*8]      ; Lasketaan PT-elementin osoite.
  pop   rsi                   ; Palautetaan fyysinen osoite pinosta.
  or    sil, PAGE_PRESENT     ; Asetetaan present-bitti.
  mov   [rax], rsi            ; Päivitetään PT-elementtiä.
  ret
```

Kuva 9. Sivun osoittaminen fyysiseen alueeseen

Muistia varattaessa tarvittava määrä virtuaalisia muistisivuja asetetaan osoittamaan vapaisiin fyysisiin muistialueisiin ja alueet on merkittävä bittikarttaan (Kuva 10). Vapauttaessa sivut merkitään pois käytöstä, jotta ne voidaan myöhemmin käyttää uudelleen. Yksinkertaisimmillaan tämä tarkoittaa osoitemuunnostaulukoiden elementtien sisältämän present-bitin asettamista nollaksi.

```
/**
 * Muistin varaus.
 * minVirtual = vähimmäisvirtuaaliosoite
 * numPages = sivujen määrä
 */
void* allocate_pages(qword minVirtual, qword numPages) {
   void *pBitmap = physmemmap_get_ptr(); // muistibittikartan osoite
   qword bmsize = physmemmap_get_size(); // bittikartan koko

   /* Haetaan peräkkäisiä vapaita sivuja ja fyysisiä alueita */
   qword virtual = virtmem_getnotpresent_ext(minVirtual, numPages);
   qword pagenum = bitmap_getfree_ext(pBitmap, bmsize, 0, numPages);

   void *ret = (void *)virtual; // paluuarvona varattu osoite

   /* Asetetaan sivut osoittamaan fyysisiin alueisiin */
   qword c = 0;
   while (c < numPages)
   {
      map_page(virtual, pagenum * 0x1000);
      bitmap_setbit(pBitmap, pagenum); // merkintä bittikarttaan
      virtual += 0x1000;               // seuraavan sivun osoite
      pagenum++;                       // seuraava sivu
      c++;
   }
   return ret; // palautetaan sivujen virtuaaliosoite
}
```

Kuva 10. Muistin varaus

Monet käyttöjärjestelmät tukevat ns. swap-osiota tai tiedostoa. Niissä järjestelmissä present-bitti merkitsee ainoastaan onko sivu muistissa, eikä onko sen osoittama alue varattuna. Muistiosoite voi olla varattu bitin ollessa nolla, mutta varsinainen data on tallennettuna väliaikaisesti levylle muistin vapauttamiseksi muuhun tarkoitukseen. Osoitemuunnostaulukoiden elementit sisältävät kuitenkin käyttöjärjestelmän käyttöön varattuja bittejä, joita on mahdollista käyttää varattujen osoitteiden seurantaan samalla periaatteella.

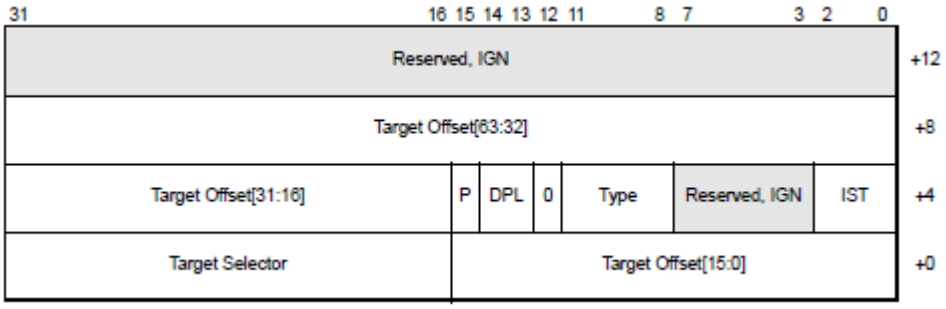
7 KESKEYTYKSET

Keskeytys (engl. interrupt) tapahtuu tilanteissa, joissa vaaditaan prosessorin välitöntä huomiota. Keskeytyksen tapahtuessa prosessori siirtyy automaattiesti suorittamaan keskeytyskäsittelijää. Käyttöjärjestelmän on määriteltävä kaikki keskeytyskäsittelijät IDT-tauluun (Interrupt Descriptor Table) ennen keskeytysten sallimista. Keskeytyksen mahdolliset lähteet voidaan jakaa kolmeen pääryhmään: laitteistokeskeytykset

(engl. hardware interrupts), ohjelmalliset keskeytykset (engl. software interrupts) ja poikkeukset (engl. exceptions). (Intel 2012, 55.)

7.1 IDT

IDT (Interrupt Descriptor Table) on taulukko, joka sisältää joukon Interrupt Gate Descriptoreja (IGD). IGD pitää sisällään tarvittavat tiedot keskeytyskäsittelijään siirtymiseen (Kuva 11). Elementit ovat 16 tavuisia ja niitä on 256 kappaletta. Elementin indeksi on yhtä kuin keskeytyksen numero, jota käytetään mm. keskeytystä kutsuttaessa. (Intel 2012, 171.)



Kuva 11. 64-bittinen Interrupt Gate Descriptor (AMD 2012a, 93)

IGD:n sisältämä 3-bittinen Interrupt-Stack-Table-kenttä (IST) on eräänlainen stack-valitsin, jonka avulla keskeytyksen tapahtuessa pino-osoite–rekisteriin (RSP) voidaan valita uusi arvo. Kentän ansiosta eri keskeytykset voivat käyttää eri pinoja, joka helpottaa varsinkin samaan aikaan tapahtuvien keskeytysten käsittelyä. Present-bitti (P) kertoo onko kyseisen elementin kuvaama keskeytys käytössä, eli onko keskeytykselle olemassa keskeytyskäsittelijää. Descriptor Privilege Level (DPL) -kenttä valitsee vaadittavan oikeustason, jolla keskeytys voidaan suorittaa. Vain käyttöjärjestelmän käyttöön tarkoitetuissa keskeytyksissä DPL on 0, tavallisille ohjelmille suunnatuissa se taas on 3. Selector-kenttä valitsee vastaavasti tason, jota käytetään keskeytyksen suorittamiseen. Target Offset -kenttä sisältää keskeytyksen käsittelijän muistiosoitteen. Prosessorin suoritus siirtyy automaattisesti tähän osoitteeseen keskeytyksen tapahtuessa. (AMD 2012a, 93.)

IDT:n sisältämiä elementtejä ei tavallisesti tarvitse myöhemmin muuttaa, joten ne voidaan luoda jo koodin kääntämisvaiheessa esimerkiksi makrojen avulla (Kuva 12). IDT:n sijainti on käyttöjärjestelmän kehittäjän määriteltävissä antamalla taulukon osoite parametrina LIDT-käskylle (AMD 2012b, 347).

```
%macro dIGD64 6
  dw %1 & 0xFFFF                ; osoitteen bitit 0..15
  dw %2 & 0xFFFF                ; valitsin (selector)
  db %6 & 0x07                  ; pinovalitsin (IST)
  db %3 | (%4 << 5) | (%5 << 7) ; tyyppi, käyttöoikeus ja present-bitti
  dw (%1 >> 16) & 0xFFFF        ; osoitteen bitit 16..31
  dd (%1 >> 32) & 0xFFFFFFFF    ; osoitteen bitit 32..63
  dd 0
%endmacro

IDT:
  dIGD64 addr(int_00), GDT.code_0, SSD_TYPE_INTERRUPT_GATE64, 0, 1, 1
  dIGD64 addr(int_01), GDT.code_0, SSD_TYPE_INTERRUPT_GATE64, 0, 1, 1
  dIGD64 addr(int_02), GDT.code_0, SSD_TYPE_INTERRUPT_GATE64, 0, 1, 1
   ; .....
```

Kuva 12. IDT:n luonti

7.2 APIC

APIC (Advanced Programmable Interrupt Controller) on ohjelmoitava keskeytysohjain. Se on tietokoneen osa, joka vastaa keskeytyssignaalien välittämisestä prosessorien ja laitteiden välillä. APIC sisältää omia rekistereitä, joita voidaan käsitellä valmiiksi varatulla muistialueella. Rekistereihin kirjoittamalla käyttöjärjestelmä voi vaikuttaa keskeytyksien tapahtumiseen, esimerkiksi uudelleen määrittämällä signalointiin käytettävät IRQ (Interrupt Request) – väylät, estämällä osan keskeytyksistä tai vaihtamalla keskeytysten välistä prioriteettia. (Intel 2010.)

7.3 Laitteistokeskeytykset

Laitteistokeskeytys tapahtuu, kun jokin tietokoneeseen liitetyistä laitteista lähettää prosessorille keskeytyssignaalin. Signaalien ansiosta laitteiden tilaa ei tarvitse jatkuvasti kysellä (engl. poll). Laitteen tilan tarkistaminen aiheuttaa prosessorille ylimääräistä odotusaikaa, jonka voisi paremmin käyttää muiden tehtävien suorittamiseen. Moniajokäyttöjärjestelmissä tämä voi tarkoittaa esimerkiksi muiden prosessien jatkamista kunnes laite antaa vastauksen. Vastaavasti keskeytyskäsittelijän on ilmoitet-

tava laitteelle, että signaali on käsitelty. Tavallisesti tämä tapahtuu kirjoittamalla APIC:n End of Interrupt –rekisteriin (Kuva 13). (Dandamudi 2003.)

```
; IRQ1-keskeytyskäsittelijä (Näppäimistö)
int_E4:
   cli                  ; Estetään uudet keskeytykset.
   call handle_keyboard ; Kutsutaan näppäimistökäsittelijää.

   ; Kirjoitetaan APIC:n EOI-rekisteriin (End Of Interrupt).
   xor eax, eax         ; Nollataan kirjoitettava arvo.
   mov [APIC_MAPPED_ADDR + APIC_REG_EOI], eax ; Signaali APIC:lle.
   sti                  ; Sallitaan uudet keskeytykset.
   iretq                ; Palataan keskeytyskäsittelijästä.
```

Kuva 13. Laitteistokeskeytyksen käsittelijä

7.4 Ohjelmalliset keskeytykset

Ohjelmallisia keskeytyksiä voidaan pitää eräänlaisina käyttöjärjestelmän tarjoamina palveluina. Ohjelmat voivat aiheuttaa keskeytyksen käyttämällä INT-käskyä, jolloin prosessorin suoritus siirtyy haluttuun keskeytyskäsittelijään (Kuva 14). Keskeytyskäsittelijä voi esimerkiksi suorittaa toimenpiteitä, jotka vaativat käyttöjärjestelmätason oikeuksia. Tavallisella ohjelmalla ei tyypillisesti ole esimerkiksi oikeuksia suoraan muokata muistinhallintajärjestelmän ylläpitämiä rakenteita, vaan muistin käsittely hoidetaan epäsuorasti käyttöjärjestelmän tarjoamien muistinhallintapalveluiden kautta.

```
; Palvelukeskeytys (rax = funktion numero)
int_80:
  cmp  rax, 0x10               ; Tarkistetaan, ettei numero ole liian suuri.
  jge  .error                  ; Jos on, tulostetaan virheviesti.
  shl  rax, 3                  ; Kerrotaan 8:lla.
  add  rax, addr(service_table) ; Lisätään funktiotaulun osoite.
  call [rax]                   ; Kutsutaan taulussa olevaa funktiota.
  iretq                        ; Paluu keskeytyskäsittelijästä.
.error:
  mov  rdi, .msg_error         ; Parametri tulostukselle.
  call println                 ; Tulosetaan virheviesti.
  iretq                        ; Paluu keskeytyskäsittelijästä.
.msg_error: db 'INT 0x80 Invalid function',0

service_table:
  dq addr(print)               ; Funktio 0x00.
  dq addr(println)             ; Funktio 0x01.
  dq addr(print_hex)           ; Funktio 0x02.
  dq addr(clear_screen)        ; Funktio 0x03.
  ; .....
```

Kuva 14. Ohjelmallisen keskeytyksen käsittelijä

7.5 Poikkeukset

Poikkeukset ovat keskeytyksiä, jotka tapahtuvat erilaisissa virhetilanteissa. Virhetilanteiden aiheuttamista keskeytyksistä ei välttämättä ole mielekästä tai edes mahdollista palata takaisin tavalliseen suoritukseen. Poikkeusten käsittelyssä käyttöjärjestelmän on kuitenkin pyrittävä palautumaan tilanteesta mahdollisimman pienin vahingoin. Yksinkertaisimmillaan tämä voi tarkoittaa sitä, että käyttäjälle tulostetaan virheviestin lisäksi tietoja virheen aiheuttaman koodin sijainnista, jonka jälkeen siirrytään tilaan, josta käyttäjä voi jatkaa järjestelmän käyttöä ilman tietokoneen uudelleen käynnistämistä.

Tyypillinen poikkeuskeskeytys tapahtuu jaettaessa nollalla, jonka käsittelyn jälkeen ei voida palata keskeytyksen aiheuttaneeseen koodiin. Keskeytyksen tapahtuessa paluuosoite on tallennettuna pinoon. IRETQ-käsky poistaa paluuosoitteen pinosta ja siirtyy suorittamaan sen osoittamaa koodia (AMD 2012b, 337). Paluuosoitetta voidaan muuttaa poistamalla se pinosta ja lisäämällä uusi osoite sen tilalle ennen IRETQ-käskyn suorittamista (Kuva 15).

```
; #DE Divide-By-Zero-Error
int_00:
  mov rdi, .msg       ; Virheviesti parametriksi.
  call println        ; Tulostetaan virheviesti.
  pop rax             ; Poistetaan paluuosoite pinosta.
  mov rax, addr(system_reinit) ; Uusi paluuosoite.
  push rax            ; Järjestelmän nollausfunktion osoite pinoon.
  iretq               ; Poistutaan keskeytyskäsittelijästä nollausfunktioon.

.msg: db 'INT 0x00 Divide-By-Zero-Error',0
```

Kuva 15. Poikkeuksen käsittelijä

8 LAITEHALLINTA

Tietokoneeseen liitettyjen laitteiden hallinta on pääasiassa laiteajureiden tehtävä. Laiteajurit ovat tyypillisesti omia ohjelmiaan, erillään käyttöjärjestelmästä. Käyttöjärjestelmän on kuitenkin pystyttävä yksinkertaisiin perustoimenpiteisiin, jotta esimerkiksi myöhempi laiteajureiden lataaminen olisi mahdollista.

8.1 Kiintolevy

Kiintolevyn hallintaan kuuluu yksinkertaisimmillaan tiedon luku ja kirjoittaminen (Kuva 16). Näiden välttämättömien perustoiminnallisuuksien lisäksi, kiintolevy saattaa sisältää kehittyneempiä toimintoja, kuten esimerkiksi virransäästö ja välimuistin käsittely. Kiintolevyn ohjaus tapahtuu kommunikoimalla porttien kautta, käyttäen standardoituja protokollia (INCITS 2006).

Useimmat kiintolevyt tukevat ainakin kahta eri tapaa ilmaista muistin fyysisiä osoitteita. Vanhanaikainen tapa on käyttää ns. CHS-osoitetta (Cylinder-Head-Sector). CHS-osoite perustuu kiintolevyn sylinteri-, lukupää- ja sektorinumeroihin, jonka vuoksi tätä tapaa käytettäessä kiintolevyn fyysinen rakenne on oltava tiedossa. Nykypäivänä käytetään tyypillisesti LBA-osoitetta (Linear-Block-Address), jonka avulla kiintolevyn sektoreihin viitataan suoraan järjestysnumerolla, jolloin vältytään kiintolevykohtaisilta osoitemuunnoksilta. (Straub 2011.)

```
; Sektorien luku LBA-tilassa
;  rdi : Osoite johon data luetaan
;  rsi : LBA-osoite
;  dx  : Sektorien lukumäärä
ata_lba48_read:
  ata_lba48_rw ATA_CMD_READ_SECTORS_EXT ; Lukemisen valmistelut.
  rep  insw                ; Luetaan data portista [rdi].
  ret

; Sektorien kirjoitus LBA-tilassa
;  rdi : Kirjoitettavan datan osoite
;  rsi : LBA-osoite
;  dx  : Sektorien lukumäärä
ata_lba48_write:
  ata_lba48_rw ATA_CMD_WRITE_SECTORS_EXT ; Kirjoittamisen valmistelut.
  mov  rsi, rdi            ; Siirretään muistialueen osoite rsi-rekisteriin.
  rep  outsw               ; Kirjoitetaan data porttiin [rsi].
  ret

%macro ata_lba48_rw 1
  mov  rax, 0xFFFFFFFFFFFF ; Peite 48-bittiselle luvulle.
  and  rsi, rax            ; Varmistetaan LBA-osoitteen bittisyys.
  xor  rcx, rcx            ; Tyhjennetään rcx.
  mov  cx, dx              ; Talletetaan sektorien lukumäärä.
  out_byte ATA_PORT_DRIVE, 0x40 ; Valitaan master-asema
  out_byte ATA_PORT_NUM_SECTORS, ch ;Sektorien määrän eniten merkitsevä osa.
  ata_set_lba48 ATA_PORT_LBA4, rsi, 24 ; Bitit 24..31 LBA-osoitteesta.
  ata_set_lba48 ATA_PORT_LBA5, rsi, 32 ; Bitit 32..39 LBA-osoitteesta.
  ata_set_lba48 ATA_PORT_LBA6, rsi, 40 ; Bitit 40..47 LBA-osoitteesta.
  out_byte ATA_PORT_NUM_SECTORS, cl;Sektorien määrän vähiten merkitsevä osa.
  ata_set_lba48 ATA_PORT_LBA1, rsi, 0  ; Bitit 0..7 LBA-osoitteesta.
  ata_set_lba48 ATA_PORT_LBA2, rsi, 16 ; Bitit 16..23 LBA-osoitteesta.
  ata_set_lba48 ATA_PORT_LBA3, rsi, 8  ; Bitit 8..15 LBA-osoitteesta.
  out_byte ATA_PORT_COMMAND, %1        ; Valitaan käsky.
.wait:
  in   al, dx              ; Luetaan levyn tila.
  test al, 8               ; Tarkistetaan onko levy valmis.
  jz   .wait               ; Odotetaan, jollei ole.
  mov  rax, 256            ; 256 x 16 bittistä arvoa = 1 sektori.
  mul  cx                  ; Kerrotaan 256 sektorien määrällä.
  mov  rcx, rax            ; Asetetaan tulos laskuriin.
  mov  dx, ATA_PORT_DATA   ; Valitaan data-portti.
%endmacro

%macro ata_set_lba48 3
   mov  rax, %2            ; LBA-osoite.
   shr  rax, %3            ; Otetaan osoitteesta valittu osa.
   out_byte %1, al         ; Kirjoitetaan osoitteen osa porttiin.
%endmacro

%macro out_byte 2
   mov  dx, %1             ; Valitaan portti.
   mov  al, %2             ; Kirjoitettava arvo.
   out  dx, al             ; Kirjoitetaan porttiin.
%endmacro
```

Kuva 16. Kiintolevyn luku ja kirjoitus

8.2 Näyttö

Näytön yksinkertainen käsittely on yleensä tarpeen jo käyttöjärjestelmän käynnistyksen alkuvaiheessa. Erilaisten tietojen tulostaminen on olennaista etenkin testausvai-

heessa ja mahdollisissa virhetilanteissa. Näyttö on käyttöjärjestelmän käynnistysvaiheessa VGA-tekstitilassa, jossa käytettävänä on vilkkuva kursori, 16 väriä ja 80x25–merkkialue (Neal 1998). Tämä tila on varsin riittävä yksinkertaisille käyttöjärjestelmille, jotka eivät sisällä graafista käyttöliittymää. Monimutkaisempi näytön käsittely tapahtuu tyypillisesti näytönohjainkohtaisten ajureiden toimesta.

VGA-tekstitilassa näytön merkkejä ja värejä manipuloidaan kirjoittamalla tarkoitukseen varatulle muistialueelle (Kuva 17). Muistialue sijaitsee fyysisessä osoitteessa 0xB8000 ja se sisältää 2000 16-bittistä elementtiä. Jokainen elementti sisältää tiedon näytöllä näkyvästä merkistä, merkin väristä ja taustaväristä. (Neal 1998.)

```
; Tekstin tulostus
;  rdi : null-päätteisen merkkijonon osoite
print:
  mov  ah, 0x07      ; Asetetaan väri (harmaa teksti, musta pohja).
  mov  rsi, [SYSVAR.print_addr] ; Haetaan tulostusosoite.
.next:
  mov  al, byte[rdi] ; Luetaan merkki merkkijonosta.
  test al, al        ; Testataan onko merkki null.
  jz   .end          ; Siirrytään loppuun merkin ollessa null.
  mov  [rsi], ax     ; Siirretään merkki ja väri tulostusosoitteeseen.
  add  rsi, 2        ; Lisätään tulostusosoitteeseen 2.
  inc  rdi           ; Lisätään merkkijono-osoitinta.
  jmp  .next         ; Siirrytään seuraavaan merkkiin.
.end:
  mov [SYSVAR.print_addr], rsi ; Päivitetään tulostusosoite.
  jmp  cursor_pos_update       ; Siirrytään päivittämään kursorin sijainti.

; Ruudun tyhjennys
clear_screen:
  mov  rdi, VGA_CHARBUF_BASE    ; VGA-tekstialueen osoite.
  mov  rcx, 500                 ; Laskuri = 500.
  mov  rax, 0x0720072007200720  ; Välilyöntejä mustalla taustalla.
  rep  stosq                   ; Kopioidaan 8 tavua välilyöntejä 500 kertaa.
  mov  rdi, to_text_addr(0, 24) ; Uusi tulostusosoite.
  mov  [SYSVAR.print_addr], rdi ; Päivitetään tulostusosoite.
  jmp  cursor_pos_update        ; Siirrytään päivittämään kursorin sijainti.
```

Kuva 17. Tekstin tulostus ja ruudun tyhjennys

Kursoria ohjataan VGA-rekistereillä, joita voidaan lukea ja kirjoittaa porttien välityksellä (Kuva 18). Rekisterit mahdollistavat kursorin sijainnin, muodon ja vilkkumisnopeuden muuttamisen. (Neal 1998)

```
; Kursorin sijainnin päivitys
cursor_pos_update:
  mov  rcx, [SYSVAR.print_addr] ; Luetaan tulostusosoite.
  sub  rcx, VGA_CHARBUF_BASE    ; Vähennetään VGA-tekstialueen alkuosoite.
  shr  rcx, 1                   ; Jaetaan kahdella väritiedon vuoksi.
  write_vga_reg 14, ch          ; Kirjoitetaan kursorin sijainnin osat
  write_vga_reg 15, cl          ; eri rekistereihin.
  ret

%macro write_vga_reg 2
  mov  dx, CRTC_ADR  ; Rekisterin valinta portti.
  mov  al, %1        ; Valitaan VGA-rekisteri.
  out  dx, al        ; Kirjoitetaan rekisterin numero porttiin.
  inc  dx            ; Seuraava portti.
  mov  al, %2        ; Rekisteriin kirjoitettava arvo.
  out  dx, al        ; Kirjoitetaan VGA-rekisteriin.
%endmacro
```

Kuva 18. Kursorin sijainnin muuttaminen VGA-rekistereillä

8.3 Näppäimistö

Näppäimistön nappien painaminen aiheuttaa laitteistokeskeytyksen, joten painalluksien käsittelyyn voidaan siirtyä suoraan keskeytyskäsittelijästä. Käsittelijässä painetun näppäimen scan-koodi saadaan selville lukemalla se tarkoitukseen varatusta portista. Portista luettu scan-koodi riippuu näppäimistössä käytössä olevan scan-koodijoukon (engl. scan-code set) mukaan, jolloin koodia vastaava kirjain on luettava merkistökohtaisesta muunnostaulukosta (BrokenThorn Entertainment 2008). Käytössä olevaa scan-koodijoukkoa voidaan tarvittaessa vaihtaa kommunikoimalla näppäimistön kanssa porttien välityksellä. Näppäimistöjen tuki erilaisille scan-koodijoukoille kuitenkin vaihtelee, mutta yleisimmin oletuksena käytössä oleva joukko on riittävä perusnäppäinten käsittelyyn (Kuva 19).

```
handle_keyboard:
.readkey:
  xor   eax, eax              ; Nollataan eax.
  in    al, 0x64              ; Luetaan näppäimistön tilarekisteri.
  test  al, 1                 ; 1. bitti kertoo onko puskurissa painalluksia.
  jz    .end                  ; Jos ei ole, siirrytään loppuun.
  in    al, 0x60              ; Luetaan painettu näppäin.
  test  al, 10000000b         ; Tarkistetaan onko kyseessä erikoisnäppäin.
  jnz   .readkey              ; Jos on, luetaan seuraava näppäin.
.check_enter:
  cmp   al, 0x1C              ; Tarkistetaan onko näppäin enter.
  jne   .check_backspace      ; Jos ei, tarkistetaan backspace.
  call  cmdline_handle_enter  ; Enterin jatkokäsittely.
  jmp   .readkey              ; Luetaan seuraava näppäin.
.check_backspace:
  cmp   al, 0x0E              ; Tarkistetaan onko näppäin backspace.
  jne   .character            ; Jos ei, se on tavallinen merkkinäppäin.
  call  cmdline_handle_backspace ; Backspacen jatkokäsittely.
  jmp   .readkey              ; Luetaan seuraava näppäin.
.character:
  mov   dil, [scancode_list + eax] ; Luetaan kirjain muunnostaulukosta.
  test  dil, dil              ; Tarkistetaan löytyykö näppäimelle kirjainta.
  jz    .readkey              ; Jos ei, luetaan seuraava näppäin.
  call  cmdline_handle_char   ; Kirjaimen jatkokäsittely.
  jmp   .readkey              ; Luetaan seuraava näppäin.
.end:
  ret
```

Kuva 19. Näppäimistökäsittelijä

9 TIEDOSTOJÄRJESTELMÄ

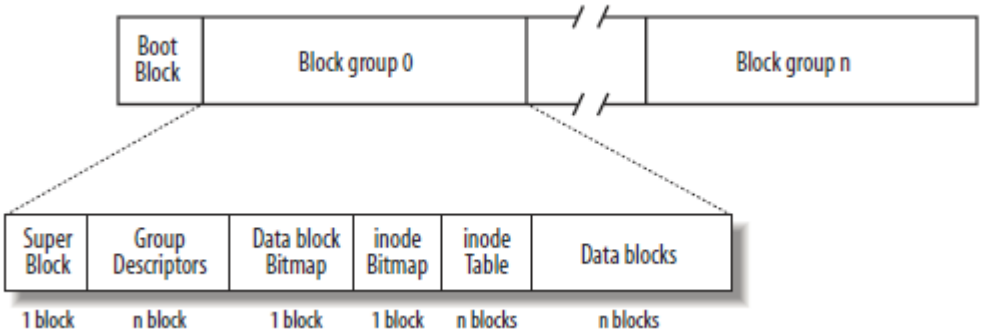
Tiedostojärjestelmä on rakenne, joka mahdollistaa tiedostojen käsittelyn ja tiedostoon liittyvien metatietojen säilyttämisen. Useimmat tiedostojärjestelmät sisältävät myös jonkinlaisen mekanismin tiedostojen säilyttämiseen eri hakemistoissa. Tyypillisimpiin metatietoihin kuuluu mm. luontipäivämäärä ja käyttöoikeudet. Monimutkaisemmat tiedostojärjestelmät sisältävät valmiudet tiedostojen pakkaukseen, salaukseen ja muutosten seurantaan. Muutoksien seurannassa käytetään erillistä lokirakennetta, jonka avulla järjestelmä voidaan myöhemmin palauttaa aiempaan tilaan.

Uutta käyttöjärjestelmää toteuttaessa tiedostojärjestelmäksi on hyvä valita yleisesti tunnettu ja standardoitu tiedostojärjestelmä, jolloin esimerkiksi tiedostojen siirto järjestelmästä toiseen olisi mahdollista käyttäen olemassa olevia työkaluja. Tyypillisiä valintoja pienimuotoisiin käyttöjärjestelmiin on Microsoftin kehittämä FAT tai Linu-

xeissa yleisesti tuettu Ext2. Tämän opinnäytetyön käyttöjärjestelmässä on toteutettuna Ext2 tiedostojärjestelmän hallinta.

9.1 Ext2 rakenne

Ext2-tiedostojärjestelmä on rakenteeltaan suhteellisen yksinkertainen, mutta suorituskyvyn kannalta hyvin optimaalinen. Levy on jaettuna block-ryhmiin (engl. block group), jotka sisältävät mm. tiedostoja kuvaavia inode-rakenteita ja varsinaista tiedostodataa (Kuva 20). Ideaalitilanteessa samoihin aikoihin käsiteltävät tiedostot sijaitsevat samassa ryhmässä, jolloin tiedostojen käsittely nopeutuu, sillä esimerkiksi kiintolevyltä lukeminen on tehokkaampaa tiedostojen sijaitessa fyysisesti lähempänä toisiaan. (Bovet & Cesati 2005, 738.)



Kuva 20: Ext2 levyn rakenne (Bovet & Cesati 2005, 741)

9.1.1 Superblock

Superblock sisältää koko tiedostojärjestelmää koskevat tiedot (Kuva 21). Siinä on määriteltynä montako inodea ja data-blockia kuhunkin ryhmään kuuluu. Kenttien avulla voidaan myöhemmin laskea esimerkiksi missä ryhmässä tarvittava inode sijaitsee. Se sisältää tiedot myös siitä montako inodea ja data-blockia tiedostojärjestelmässä on yhteensä ja montako niistä on vapaina. Blockien koko ei myöskään ole ennalta määritelty, vaan se lasketaan s_log_block_size–kentän avulla. (Poirier 2011.)

```
typedef struct {
   dword s_inodes_count;       // inodejen määrä
   dword s_blocks_count;       // blockien määrä
   dword s_r_blocks_count;     // varattujen blockien määrä
   dword s_free_blocks_count; // vapaiden blockien määrä
   dword s_free_inodes_count; // vapaiden inodejen määrä
   dword s_first_data_block;  // ensimmäisen data-blockin numero
   dword s_log_block_size;    // blockien koko (1kt << s_log_block_size)
   dword s_log_frag_size;
   dword s_blocks_per_group;  // blockien määrä per ryhmä
   dword s_frags_per_group;
   dword s_inodes_per_group;  // inodejen määrä per ryhmä
   dword s_mtime;
   dword s_wtime;
   word  s_mnt_count;
   word  s_max_mnt_count;
   word  s_magic;              // tarkistusluku
   word  s_state;              // tila (1 = ok, 2 = virheitä)
   word  s_errors;             // virhekoodi
   word  s_minor_rev_level;    // versionumero
   dword s_lastcheck;
   dword s_checkinterval;
   dword s_creator_os;
   dword s_rev_level;
   word  s_def_resuid;
   word  s_def_resgid;
} __attribute__((packed, aligned(1))) Superblock;
```

Kuva 21. Superblock (mukaillen Bovet & Cesati 2005, 742)

Superblock sijaitsee levyllä aina heti ensimmäisen kilotavun jälkeen, joten se kuuluu block-ryhmään 0. Blockien koosta riippuen se voi sisältyä ensimmäiseen blockiin boot sectorin kanssa, mutta yhden kilotavun block-koolla se vie aina koko toisen blockin. Superblock on tiedostojärjestelmän kannalta olennaisin rakenne, joten se voi myös olla varmuuskopioituna muiden block-ryhmien alkuun. (Poirier 2011.)

9.1.2 Group descriptors

Group descriptors -alue löytyy aina block-ryhmästä 0, ja se sisältää yhden elementin (Block Group Descriptor) jokaista tiedostojärjestelmässä olevaa block-ryhmää kohden. Alueesta voi Superblockin tavoin olla kopioita muissakin ryhmissä. Block Group Descriptorin sisältämät kentät määrittävät mm. missä blockissa inode-taulu, inode-bittikartta ja block-bittikartta sijaitsevat (Kuva 22). (Poirier 2011.)

```
typedef struct  {
   dword bg_block_bitmap;      // block-bittikartan blocknumero
   dword bg_inode_bitmap;      // inode-bittikartan blocknumero
   dword bg_inode_table;       // inode-taulun blocknumero
   word  bg_free_blocks_count; // vapaiden blockien määrä
   word  bg_free_inodes_count; // vapaiden inodejen määrä
   word  bg_used_dirs_count;   // kansio-inodejen määrä
   word  bg_pad;
   byte  bg_reserved[12];
} __attribute__((packed, aligned(1))) BlockGroupDescriptor;
```

Kuva 22. Block Group Descriptor (mukaillen Bovet & Cesati 2005, 744)

Käytetyistä inodeista ja blockeista pidetään kirjaa bittikarttojen avulla, joissa jokainen bitti kuvaa yhtä inodea tai blockia. Kaikki bittikarttojen käsittelyyn vaadittavat tiedot selviävät Superblockin ja Block Group Descriptorin sisältämistä kentistä. Esimerkiksi bittikarttojen koko riippuu siitä, montako inodea ja blockia kuhunkin ryhmään on määritelty Superblockissa. (Poirier 2011.)

9.1.3 Inode

Inode sisältää yksittäisen tiedostojärjestelmän elementin käsittelyyn tarvittavat tiedot. Elementti voi olla esimerkiksi tavallinen tiedosto tai kansio. Inodesta selviää elementin tyypin lisäksi mm. sen käyttämän datan koko, käyttöoikeudet, luontipäivämäärä ja muokkauspäivämäärä. (Bovet & Cesati 2005, 745.)

```
typedef struct  {
   word  i_mode;        // tyyppi ja käyttöoikeudet
   word  i_uid;
   dword i_size;        // tiedoston koko
   dword i_atime;       // käyttöaika
   dword i_ctime;       // luontiaika
   dword i_mtime;       // muokkausaika
   dword i_dtime;       // poistoaika
   word  i_gid;
   word  i_links_count;
   dword i_blocks;      // käytettyjen 512-tavuisten osien määrä
   dword i_flags;
   dword i_osd1;
   dword i_block[15];   // datablockien numerot
   dword i_generation;
   dword i_file_acl;
   dword i_dir_acl;
   dword i_faddr;
   byte  i_osd2[12];
} __attribute__((packed, aligned(1))) Inode;
```

Kuva 23. Inode (mukaillen Bovet & Cesati 2005, 745)

Inodeen liittyvä data löydetään i_block-taulukkokentän avulla. Taulukon ensimmäiset 12 elementtiä ovat suoria viittauksia, jotka sisältävät varsinaisten data-blockien numeroita. Kolmastoista elementti on epäsuora viittaus eli se ilmaisee blockin, joka sisältää jälleen suoria viittauksia data-blockeihin. Neljästoista elementti on kaksinkertaisesti epäsuora viittaus, joten sen kuvaama block sisältää viittauksia epäsuoria viittauksia sisältäviin blockeihin. Viimeinen elementti on vastaavalla periaatteella toimiva kolminkertainen epäsuora viittaus, joka mahdollistaa jo hyvin massiivisten tiedostojen tallentamisen pienelläkin block-koolla. Tämän järjestelmän ansiosta Ext2-tiedostojärjestelmä on tehokkain yleisimmässä käyttötapauksessa, eli pienien tiedostojen käsittelyssä. Epäsuorat viittaukset mahdollistavat kuitenkin suurienkin tiedostojen käsittelyn. (Poirier 2011.)

Kansioiden data-blockit sisältävät joukon rakenteita, jotka kuvaavat kansion sisältämiä tiedostoja ja alikansioita. Ne sisältävät tiedostonimen, nimen pituuden, tyypin, rakenteen koon ja kansioelementtiin liittyvän inoden numeron (Kuva 24). Tyyppikenttä on käytännössä sama, kuin inodessa määritelty, joten se kuvaa onko kyseessä esimerkiksi tavallinen tiedosto vai kansio. Alkuperäisessä Ext2-määrittelyssä tyyppikenttää ei ollut, vaan tiedostonimelle varattu kenttä oli suurempi. Useimmat toteutukset kuitenkin tukivat enintään 255 merkkisiä tiedostonimiä, joten tilalle keksittiin parempaa käyttöä. Tyyppikentän ollessa mukana kansiorakenteessa, varsinaista inodea ei tarvitse lukea levyltä sen tyypin selvittämiseksi. Rakenteen koko – kentän avulla selviää, mistä kohtaa seuraava elementti alkaa, joten sitä käytetään elementtien läpikäymiseen. (Poirier 2011.)

```
typedef struct {
   dword inode;     // inode-numero
   word  rec_len;   // tämän elementin koko
   byte  name_len;  // nimen pituus
   byte  file_type; // tiedostotyyppi
   byte  name[255]; // tiedoston nimi
} __attribute__((packed, aligned(1))) DirectoryEntry;
```

Kuva 24. Kansioelementti (mukaillen Bovet & Cesati 2005, 749)

9.2 Tiedostojen käsittely

Tiedostojen ja kansioiden luonti alkaa tarkistamalla, että järjestelmässä on vapaita inodeja, jonka jälkeen vapaa inode etsitään valitun block-ryhmän inode-bittikartasta. Saatu inode-numero muutetaan osoitteeksi, johon uusi inode voidaan kirjoittaa (Kuva 25). Inoden kenttien täyttämisen jälkeen, halutusta kansiosta on haettava tyhjä tila uudelle kansioelementille, johon tiedoston nimi ja inode-numero kirjoitetaan. Kansion käyttämiä data-blockeja on mahdollisesti varattava lisää ja merkittävä kansion inoden i_block-taulukkoon. Tiedostojen poisto voidaan hoitaa yksinkertaisesti poistamalla käytetty inode ja kansioelementti. Vapautetut alueet merkitään bittikarttoihin.

```
/**
 * Inoden haku numeron perusteella.
 * inodeNumber = inoden numero
 */
Inode* inode_num_to_ptr(dword inodeNumber) {
   // Inoden numero alkaa ykkösestä, joten vähennetään 1
   dword ii = inodeNumber - 1;

   /* Lasketaan block-ryhmän numero ja indeksi ryhmän sisällä */
   dword blockGroup = ii / pSuperBlock->s_inodes_per_group;
   dword indexInGroup = ii % pSuperBlock->s_inodes_per_group;

   /* Luetaan BlockGroupDescriptor ja sen sisältämä inode-taulu */
   BlockGroupDescriptor* pBlockGroup = get_block_group(blockGroup);
   Inode* pInodeTable = get_inode_table(pBlockGroup);

   // Palautetaan inoden osoite
   return (Inode*)
          ((qword)pInodeTable + indexInGroup * pSuperblock->s_inode_size);
}
```

Kuva 25. Inoden haku numeron perusteella

Tiedoston kirjoituksessa vapaiden data-blockien määrä tarkistetaan ja etsitään tiedostoa kuvaava kansioelementti. Elementin inode-numeron perusteella löydetään tiedostoa kuvaava inode. Mikäli kirjoitettavan datan määrä ylittää tiedostolle varattujen blockien koon, lisätilaa etsitään käyttämällä block-ryhmän block-bittikarttaa. Bittikartan osoittamaan tilaan kirjoitetaan varsinainen tiedostodata ja inoden i_block-taulukkoon merkitään kirjoitetut data-blockit. Tiedostojen lukemisessa taulukon osoittamat blockit luetaan muistiin (Kuva 26).

```
/**
 * Tiedoston luku puskuriin.
 * fileName = tiedoston nimi
 * offset = lukemisen aloituskohta tiedoston sisällä
 * size = luettava tavumäärä
 * pBuffer = puskurin osoite
 */
qword file_read(char *fileName, qword offset, qword size, void *pBuffer) {
   if (size == 0) return 0;

   // Haetaan kansioelementti
   DirectoryEntry *pDirEntry = alloc_page(0);
   byte found = directory_get_entry(pCurrentDirInode, pDirEntry, fileName);

   qword bytesRead;
   if (found != 0)
   {
      // Haetaan inode
      Inode *pInode = inode_num_to_ptr(pDirEntry->inode);

      // Luetaan inoden datablockit
      qword bytesRead = inode_read_data(pInode, offset, size, pBuffer);
   }
   else
      bytesRead = 0;

   free_page(pDirEntry); // vapautetaan kansioelementti
   return bytesRead;     // palautetaan luettujen tavujen määrä
}
```

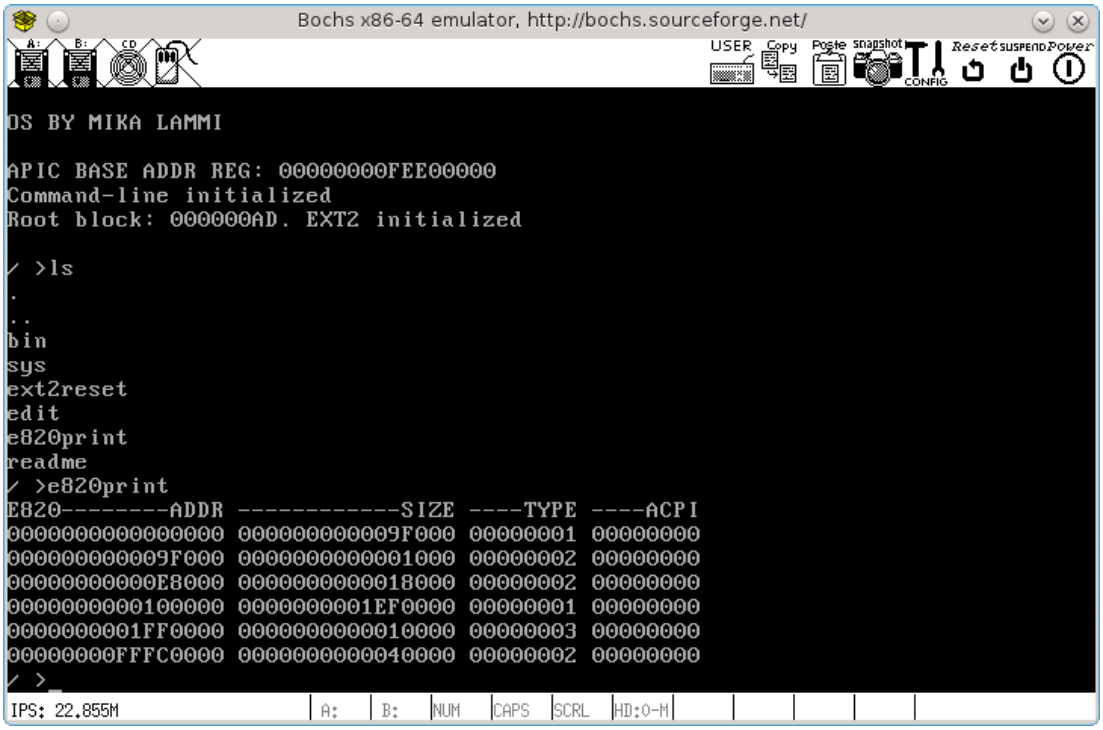
Kuva 26. Tiedoston luku

10 OHJELMIEN SUORITUS

Monissa nykypäivän käyttöjärjestelmissä ohjelmatiedostoilla on monimutkainen rakenne, jossa on eri tarkoituksiin suunnattuja alueita esimerkiksi metatiedoille, koodille ja datalle. Tämän opinnäytetyön toteutuksessa ohjelmien rakennetta ei ole mitenkään määritelty, vaan ohjelmat ovat suoritettavaa koodia sisältäviä binääritiedostoja.

Käyttöjärjestelmällä on oltava jonkinlainen käyttöliittymä, jonka kautta käyttäjä voi suorittaa ohjelmia. Ohjelmia käynnistetään komentoriviltä (Kuva 27), johon kirjoitetaan ohjelmatiedoston nimi, jonka perään ohjelmalle voi antaa merkkijonotyyppisiä parametreja välilyönnein eroteltuina. Näppäimenpainallukset välittyvät komentoriville suoraan näppäimistökäsittelijältä. Enterin painalluksesta käyttöjärjestelmä etsii ohjelmatiedoston nykyisestä hakemistosta, lukee sen muistiin ja alkaa suorittamaan tiedostoa sen ensimmäisestä tavusta alkaen. Muistin varaus ja vapautus, tiedostojen

käsittely, näytölle tulostaminen ja syötteiden luku tapahtuu käyttämällä käyttöjärjestelmän tarjoamia palveluita keskeytyksen kautta.



Kuva 27. Komentorivi ja ohjelmien käynnistys

11 LOPUKSI

Toteutettu käyttöjärjestelmä sisältää vaaditut perustoiminnot yksinkertaisten ohjelmien suorittamiseen. Projektina se on kuitenkin jatkuva, joten uusia ominaisuuksia on tarkoitus tulla vielä lisää. Varsinkin yhteensopivuuksia olemassa olevien ohjelmien ja käyttöjärjestelmien kanssa on tehtävä lisää, jotta aivan kaikkea ei tarvitsisi tehdä itse.

Kernel on muutettava sellaiseksi, että sen lataamiseen voidaan käyttää muitakin bootloadereita. Mikäli tämän hetkinen bootloader ja käyttöjärjestelmä asennettaisi fyysiselle laitteelle, tietokoneeseen ei voisi asentaa muita käyttöjärjestelmiä. Bootloader toteutettiin lähinnä oppimistarkoituksessa, eikä sitä ole tarkoitus kehittää eteenpäin.

Suoritettavat ohjelmat tarvitsevat yhtenäisen formaatin. Etenkin laiteajureina pitäisi pystyä käyttämään esimerkiksi Linuxissa käytettyjä ajureita, sille ohjaimien tekeminen kaikille eri laitteille olisi mahdoton urakka yhdelle ihmiselle. Kommunikointi lähiverkossa muiden tietokoneiden kanssa on tarkoitus mahdollistaa hyödyntämällä jo olemassa olevia laiteajureita.

Tavallista moniajoa käyttöjärjestelmä ei todennäköisesti tule sisältämään. Useiden prosessorien ja prosessoriytimien hyödyntäminen on kuitenkin tarkoitus toteuttaa, jolloin eri ytimissä voitaisiin suorittaa ohjelmia samanaikaisesti. Tällä ominaisuudella on kuitenkin merkittävä vaikutus lähes jokaiseen käyttöjärjestelmän osaan, joten se olisi kannattanut ottaa huomioon heti kehityksen alkuvaiheessa.

Lopullisena tavoitteena käyttöjärjestelmästä on tarkoitus tulla eräänlainen palvelin, jolle voidaan antaa suurta laskentatehoa vaativia tehtäviä. Suoritettava tehtävä saa käyttöönsä kaikki tietokoneen resurssit, jolloin laskentatehoa ja muistia ei kulu minkään ylimääräisen prosessin suorittamiseen.

LÄHTEET

AMD. 2012a. AMD64 Architecture Programmer's Manual Volume 2: System Programming.

AMD. 2012b. AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions.

Bovet, D. P. & Cesati, M. 2005. Understanding the Linux Kernel, 3rd Edition. O'Reilly Media.

BrokenThorn Entertainment. 2008. Operating System Development Series. Viitattu 5.11.2013. http://www.brokenthorn.com/Resources/

Brown R. 2013. Ralf Brown's Interrupt List. Viitattu 16.10.2013. http://www.ctyme.com/rbrown.htm

Chourdakis, M. 2009. The Real, Protected, Long mode assembly tutorial for PCs. Viitattu 18.10.2013. http://www.codeproject.com/Articles/45788/The-Real-Protected-Long-mode-assembly-tutorial-for

Dandamudi, S. 2003. Interrupts. Viitattu 24.11.2013. http://www.scs.carleton.ca/sivarama/org_book/org_book_web/slides/chap_1_versions/ch20_1.pdf

INCITS. 2006. ATA/ATAPI Command Set (ATA8-ACS). Viitattu 27.12.2013. http://www.t13.org/documents/UploadedDocuments/docs2008/D1699r6a-ATA8-ACS.pdf

Intel. 2010. Intel 64 Architecture x2APIC Specification. Viitattu 12.12.2013. http://www.intel.com/Assets/en_US/PDF/manual/318148.pdf

Intel. 2012. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3. Viitattu 10.12.2013. http://www.intel.co.za/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf

Lawton, K., Denney, B., Guarneri, D., Ruppert, V. & Bothamy, C. Bochs User Manual. Viitattu 17.10.2013. http://bochs.sourceforge.net/doc/docbook/user/index.html

Malhar, V. 2010. Viitattu 24.11.2013. http://malhar2010.blogspot.fi/2010/11/types-of-kernel.html

McGuigan, B. 2013. Viitattu 25.16.2013. http://www.wisegeek.com/what-is-a-boot-sector.htm

NASM. 2013. The Netwide Assembler: NASM. Viitattu 16.10.2013. http://www.nasm.us/doc/nasmdoc0.html

Neal , J. D. 1998. Hardware Level VGA and SVGA Video Programming Information. Viitattu 19.11.2013. http://www.osdever.net/FreeVGA/home.htm

Poirier, D. 2011. The Second Extended File System. Viitattu 26.12.2013. http://www.nongnu.org/ext2-doc/ext2.html

Straub, U. 2011. LBA and CHS format, LBA mapping. Viitattu 21.12.2013. http://www.boot-us.com/gloss11.htm

The University of Chicago - The Department of Computer Science. 2009. x86-64 Instructions and ABI. Viitattu 24.10.2013. http://www.classes.cs.uchicago.edu/archive/2009/spring/22620-1/docs/handout-03.pdf