TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Maker Padari 193676

Lihtne operatsioonisüsteem

Bakalaureusetöö

Juhendaja: Peeter Ellervee
Abiprofessor

Tallinn 2022

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Maker Padari

21.11.2022

Annotatsioon

Töö eesmärk on luua põhiliste funktsionaalsustega operatsioonisüsteem. Operatsioonisüsteem täidab ülesandeid nagu mälu ja sisend- väljundsüsteemide haldus, ja andmevahetus välisseadmetega. Töötavas korras on graafika ja tekstitöötlus, numbrist lõime teisendamine, *page frame allocator, page table manager, global descriptor table,* katkestuste ja vigade käsitleja, *panic screen,* sisend- väljundsüsteemide haldus, klaviatuuri ja hiire tugi.

Lõputöö selgitab loodud operatsioonisüsteemi osade teooria, mida need vastavas operatsioonisüsteemis teevad ja viisi, kuidas need on implementeeritud. Operatsioonisüsteemi osadele, mis olid planeeritud lisamiseks, kuid ei jõudnud valmis, on antud ainult teooria ja võimalik viis, kuidas neid implementeerida.

Tuvastatud vead, mis operatsioonisüsteemis esinevad on toodud selles lõputöös esile ja antud neile selgitus ja võimalikud lahendused.

Lõputöö käib lühidalt üle eelneva läbikukkunud operatsioonisüsteemi ja annab lühiselgituse, millepärast töö selle kallal poolikuks jäeti.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 25 leheküljel, 15 peatükki, 10 joonist.

Abstract

Simple Operating System

The aim of the work is to create an operating system with basic functionalities. The operating system performs tasks such as memory managment, input-output systems and data exchange with external devices. The operating system implements graphics and text processing, number-to-string conversion, a page frame allocator, a page table manager, a global descriptor table, interrupt and error handler, panic screen, I/O system management and keyboard and mouse support. In addition to these the operating system includes all the functions required making these part work, such as bitmap functions for the page frame allocator.

The thesis explains the theory behind the parts of the created operating system, what they do in the corresponding operating system and the way they are implemented. The parts of the operating system that were planned to be added, but were not finished, are only given the theory behind the according parts and a possible way to implement them.

Identified bugs that occur in the operating system are highlighted in this thesis and given an explanation and possible solutions.

As well as going over the final program, the thesis goes over previous failed operating systems and gives an explaination as to why it failed and what could have been done to solve it.

The thesis is in estonian and contains 25 pages of text, 15 chapters, 10 figures,.

Lühendite ja mõistete sõnastik

IA	Arvutisüsteemide instituut
OS	Operatsioonisüsteem
IDE	Integrated Development Environment
I/O	*Input/Output*, Sisend- väljundsüsteem
UEFI	*Unified Exstensible Firmware Interface*
BIOS	*Basic Input Output System*
ROM	*Read-Only Memory*
ELF	*Executable and Linkable Format*
GOP	*Graphics Output Protocol*
GDT	*Global Descriptor Table*
RAM	*Random Access Memory*
ACPI	*Advanced Configuration and Power Interface*
PML4	*Page Map Level 4*
PDP	*Page Directory Pointer*
PD	*Page Directory*
PDE	*Page Directory Entry*
IDT	*Interrupt Descriptor Table*
GCC	*GNU Compiler Collection*
IDTR	*Interrupt Descriptor Table Register*
PS/2	*Personal System/2*
PCI	*Peripheral Component Interconnect*
RSDP	*Root System Description Pointer*
MCFG	*Memory mapped Configuration space access*
GUI	*Graphical User Interface*
USB	*Universal Serial Bus*
XSDT	*Extended System Descriptor Table*

Sisukord

Jooniste loetelu

1 Sissejuhatus

Lõputöö eesmärk oli luua lihtsam operatsioonisüsteem, mis suudaks täita operatsioonisüsteemi põhiülesandeid Lihtsama operatsioonisüsteemi all mõeldakse seda, et selles pole implementeeritud keerukamaid operatsioonisüsteemi funktsionaalsusi (näiteks arvutivõrkude tugi). Loodud operatsioonisüsteemis on implementeeritud lehekülgede saalimine, hiire ja klaviatuuri tugi, takistuste ja vigade käsitleja ja graafika ja tekstitöötlus. Lõputöös on selgitatud iga operatsioonisüsteemi osa teooria ja funktsioonide ja nende abifunktsioonide implementatsioon.

Operatsioonisüsteemi tüüp oli ülesande püstitamisel veel teadmata ja üks lõputöö eesmärke oli uurida eri operatsioonisüsteemi tüüpe, võimalusel implementeerida teatud operatsioonisüsteemile vastavad tunnused. Valmisaadud operatsioonisüsteem sarnaneb kõige enam personaalarvuti operatsioonisüsteemile nagu Windows, Mac OS X või Linux, ja on otsekui põhi, millest saaks sellele sarnaneva OS'i edasi arendada.

Operatsioonisüsteem on loodud Window'si peal kasutades Visual Studio Code'i IDE'd ja Ubuntu on Windows app'i. Kasutusel on GNU-EFI arendus keskkond, mis on teekide ja päiste keskkond loodud UEFI rakenduste kompileerimiseks .

2 Käivitamine ja alglaadur

Alglaadur asub failis guOS/gnu-efi/bootloader/main.c. Tuuma põhifunktsioon asub failis guOS/kernel/src/kernel.cpp. Kood on saadaval kasutades *link*'i Lisa 4 peatükis

2.1 Teooria

Algselt on protsessor välja lülitatud. Arvuti käivitamisel protessor hakkab täitma käske, mis loetakse BIOS ROM'ist või UEFI'ist. Selle operatsioonisüsteemi puhul on kasutusel UEFI ehk alglaadur on kirjutatud EFI programmeerimiskeeles. Käsud selles püsivaras käivitavad ülejäänud riistvara, püsivara kontrollib need ja otsib salvestusseadmetelt alglaaduri. Alglaadur käivitab operatsioonisüsteemi lugedes tuuma mällu.

2.2 Implementatsioon

Operatsioonisüsteemi käivitamiseks on kasutusel alglaadur, mis koosneb failist main.c ja tuum, mille põhifunktsioon on failis kernel.cpp. Alglaadur laeb tuuma mällu ja kutsub välja kernel.cpp failist põhifunktsiooni.

Alglaadur otsib tuuma ELF faili samast failisüsteemist, millest arvuti käivitub. Selleks on alglaaduris funktsioon failide laadimiseks, mis ütleb UEFI'le millisest failisüsteemist lugeda, laeb failisüsteemist nõutud faili ja kontrollib selle. Kui tuuma ELF faili ei leita siis kuvatakse ekraanile *error* sõnum

ELF päist kasutatakse, et saada informatsiooni tuuma ELF faili kohta, mida läheb vaja tuumale mälu eraldades. ELF päis loeb kui palju baite on failis ja sisestab selle suuruse päisesse. ELF fail kontrollitakse, et kas see on käivitav fail ja kas päise andmed on korrektsed. Kontrollid tehakse võrreldes ELF päises olevaid andmeid ELF spetsifikatsioonidega.

Kasutades ELF päises olevat informatsiooni laetakse tuuma ELF fail mällu. Kui fail ja ELF päises olevad andmed on korrektsed trükitakse ekraanile kiri, et tuuma laadimine mällu on õnnestunud.

Lõpetuseks alglaadur kutsub tuuma põhifunktsiooni, ehk sisenemispunkti.

Kogu käivitamis protsessi on näha plokkskeemist joonis 5 pealt. Alglaaduri töövoogu on näha joonis 7 pealt.

3 Graafika ja teksti kuvamine

3.1 Teooria

Pärast seda kui alglaadur on operatsioonisüsteemi laadinud ja tuum on mällu laetud siis laetakse GOP. *Graphics Output Protocol* on vajalik graafika kuvamiseks ekraanile. GOP initialiseerib graafikakaardi, et kuvada kaadripuhver ekraanile. Kaadripuhvrisse saab sisestada piksleid, mis kuvatakse ekraanil. Tuum saab kasutada kaadripuhvrit, et trükkida ekraanile näiteks teksti või värve.

3.2 Implementatsioon

Esiteks initsialiseeritakse GOP alglaaduris ja kontrollitakse, kas see leiti üles. Kui alglaadur leidis GOP'i siis võetakse kaadripuhvri informatsioon, mis edastatakse tuumale. Kaadripuhvri jaoks on *struct,* mis koosneb: kaadripuhvri baasaadressist ja suurusest, vertikaal ja horisontaal resolutsioonist ja viiendast muutujast *PixelsPerScanline* mida kasutatakse kui kaadripuhvris on rohkem piksleid kui on horisontaal pikkust resolutsioonis. Kaadripuhvri väljad täidetakse ja edastatakse tuumale.

Kasutades fondifaili psf saab tuum trükkida ekraanile teksti. Fondi laadimiseks on alglaaduris funktsioon ja kaks *struct*'i. *PSF1_HEADER* ja *PSF1_FONT*. *PSF1_HEADER* on selleks, et kontrollida fondifaili korrektsust, millist režiimi see kasutab ja kui suured on tähed baitides. *PSF1_FONT* koosneb viidast *PSF1_HEADER*'ile ja viidast glüüfide massiivile. Funktsioon, mis laeb fondi mällu kontrollib fondi korrektsust, loeb info glüüfipuhvrisse, mis hoiab kogu glüüfiinformatsiooni, ja annab kasutusele töötava fondi. Font edastatakse tuumale.

Tuum kasutab funktsiooni *PutChar* tähtede trükkimiseks ekraanile. *PutChar*'i sisenditeks on kaadripuhvri informatsioon, font, värv, täht mida trükitakse ja asukoht kuhu trükitakse. Lõngade trükkimiseks on funktsioon *Print_String*.

Funktsioon Print_String'i sisenditeks on kaadripuhvri info, font, värv ja muutuja kus hoitakse lõng. Funktsioon trükib välja iga lõngas oleva tähe kuni jõutakse lõnga lõppu. Trükkides iga uus täht liigub kursor edasi kuni ta jõuab ekraani lõppu ja liigutab kursori järgmisele reale.

4 EFI mälukaart

EFI mälu tüüpide lõngad asuvad failis guOS/kernel/src/efiMemory.cpp.

4.1 Teooria

EFI mälukaart on abi tööriist ehitatud alglaadurisse, mis kuvab RAM mälu osad mida saab operatsioonisüsteemis kasutada. Selle jaoks kasutatakse UEFI funktsiooni *BootServices->GetMemoryMap.* See funktsioon kutsutakse algselt, et saada mälukaardi suurus. Selle suuruse järgi eraldatakse puhver ja kutsutakse funktsioon uuesti saades tagasi mälukaart.

4.2 Implementatsioon

EFI mälukaart saadakse alglaaduris ja edastatakse tuumale. Alglaaduris kutsutakse mälukaart ja eraldatakse sellele mälu. Saadud info antakse tuumale, mis jätab selle meelde, et seda saaks pärast kasutada lehekülgede saalimisel. Selle jaoks on *struct EFI_MEMORY_DESCRIPTOR*, mis hoiab infot, mälu tüübi, mälu füüsiliste ja virtuaalsete aadresside viitade, lehtede arvu ja lisa informatsiooni kohta. Mälu tüüpidele on vastavad lõngad, mis aitavad neid eraldada üksteisest. Kõik eri mälu tüübid kokku liites saab kogu mälu, mis on eraldatud süsteemile. Suurem osa mälu tüüpidest ei ole kasutatud selle operatsioonisüsteemi poolt kuna neid pole vaja läinud. Kasutusel on näiteks *EfiConventionalMemory*, mis on mälu osa mida pole UEFI kasutanud ja on vabaks kasutamiseks tuuma poolt. Kõik mälu tüüpide lõngad on Lisa 2 peatüki all lisatud programmikoodis.

5 Lehekülgede saalimine

Süsteemile eraldatud mälu kokku liitmiseks olevad funktsioonid asuvad failis guOS/kernel/src/memory.cpp. *Bitmap*'i kood asub failis guOS/kernel/src/Bitmap.cpp. *Page frame allocator*'i kood on failis guOS/kernel/paging/src/PageFrameAllocator.cpp

5.1 Teooria

Lehekülgede saalimisega jagatakse arvuti mälu väiksemateks osadeks , need mälu osad on nummerdatult tabelis, mida operatsioonisüsteem saab lugeda. Kui programmil on mälu vaja siis operatsioonisüsteem saab eraldada leheküljed programmile. Uuendatud lehekülgede info sisestatakse taas tabelisse, näiteks kui on programmile eraldatud mälu või kui mälu vabastatakse.

5.2 Implementatsioon

Esiteks käidakse läbi terve mälukaart ja liidetakse kokku iga mälu tüüp, et saada terve mälu kogus, mis on süsteemile eraldatud. Selle jaoks on funktsioon, mis liigub läbi iga mälu tüübi suuruse ja liidab need kokku. Kokku liidetud mälu teisendatakse kilobaitidesse.

Kogu süsteemi mälu läheb vaja *page frame allocator*'i juures. *Page frame allocator*'it kasutatakse, et jätta meelde millised leheküljed on kasutuses ja millised on vabad. Seda on vaja, et erinevad protsessid ei kasutaks sama mälu. Lehekülgede järgimiseks on *page frame allocator'*il vaja *bitmap*'i.

Bitmap vastendab massiivi sisu üksikutele bittidele, et oleks parem mälukorraldus. Selle jaoks on *bitmap*'il klass, mis hoiab bitmap'i puhvri suurust, viita puhvri algusesse ja kaks indeks muutujat mille abil saab *bitmap*'i kasutada kui massiivi, üks lugemiseks teine sisestamiseks. *Bitmap*'i suuruse leidmiseks on vaja teada kogu süsteemi mälu kogust.

Page frame allocator'i jaoks on klass, mis koosneb funktsioonis, mis loeb EFI mälukaarti, et sealt saada informatsioon käsitleva mälu sektsiooni kohta ja *bitmap* klassi eksemplarist. Lugedes EFI mälukaarti on page frame allocatori'il vaja leida suurim vaba mälu segment, mis leitakse liikudes läbi iga mälukaardi sisendi. Suurima segmendi leidmisel ühendatakse see *bitmap*'iga. *Bitmap* initialiseeritakse andes sellele välja arvutatud suurus ja suunates *bitmap*'i puhvri viit korrektsele algaadressile. *Page frame allocator'*il on neli funktsiooni lehekülgede lukustamiseks, vabastamiseks, reservimiseks ja reserveerimise tagasi võtmiseks. Lehekülg vabastatakse kasutades *bitmap*'i indeksit, et leida üles vastav lehekülg ja sättida see vabaks. Lehekülje lukustamine töötab sarnaselt vabastamisele, vabastamise asemel lehekülg lukustatakse. Reserveerimisel kasutatakse kasutatud mälu asemel reserveeritud mälu. Reserveeritud mälu on mälu mis on kasutuses näiteks ACPI tabelite poolt. Kasutatud mälu on antud protsessidele *page frame allocator*'i poolt.

Lehekülgede nõudmiseks on funktsioon , mis liigub läbi iga indeksi *bitmap*'is ja leiab vaba koha. Leides vaba lehekülje, lukustatakse see ja tagastatakse selle koha aadress. Kui vaba kohta pole siis osa mälus salvestatakse kettale, kuid kuna ketta toetust selles operatsioonisüsteemis pole siis tagastatakse *null*.

Lehekülgede saalimise töövoogu on näha joonis 9 pealt.

6 *Page table manager*

*Page table manager'*i kood asub failis guOS/kernel/src/paging/PageMapIndexer.cpp ja guOS/kernel/src/paging/PageTableManager.cpp.

6.1 Teooria

Page table manager on operatsioonisüsteemi osa, mis hoiab vastendusi virtuaalsete ja füüsiliste aadresside vahel. Kui kirjutatakse füüsilistele aadressidele siis see aadress on samal asukohal kus ta on riistvaral. Virtuaal mälu vastandab läbi lehekülgede saalimise uue aadressi füüsilisele aadressile. Virtuaal mälu tuleb kasuks kui töötab samaaegselt kaks programmi ja on vaja määrata, millistele mälu aadressidele programmid võivad kirjutada.

6.2 Implementatsioon

Page table manager koosneb neljast eri väljast. Iga väli koosneb 512 sisendist, mis viitavad selle välja alamväljale. Esimene väli on *Page map level 4* (PML4) väli, mis koosneb 512 sisendist, mis viitavad *Page directory pointer (PDP)* väljadele. *Page directory pointer* väljad viitavad *page directory* (PD) väljadele. *Page directory* väljad viitavad *page* väljadele. *Page* väljad viitavad üksikutele lehekülgedele. Igal leheküljel on füüsiline aadress, mis teisendatakse virtuaal mällu Iga PD välja aadress viitab *page* väljale.

Page Map Indexer on *utility class*, mis muudab virtuaal mälu nõutavaks *page map*'iks. Klass koosneb funktsioonist *PageMapIndexer* ja eelenevalt nimetatud väljade indeksite muutujatest. Funktsioonis nihutatakse virtuaal aadress 12 bitti või 9 paremale, kuna *page map* on *12 bit aligned.* Väljade indeksitele antakse virtuaal aadressi väärtus, mis on mask'itud teatud väärtustega.

Kontrollitakse iga välja olemasolu kui mõni väli on puudulik siis sellele eraldatakse mälu ja antakse sellele vajalikud väärtused. Kui väli on olemas siis võetakse PDE'st aadress ja nihutatakse see 12 bitti vasakule,, et saada loogiline aadress ja määratakse see väljale. PDE on *struct*, mis sisaldab infot sisendi aadressi ja lisaandmete kohta. Aadressi määramine ja bittide nihutamine tehakse läbi iga väljaga alustades PDP'st.

7 Global descriptor table

GDT kood asub kaustas guOS/kernel/src/gdt.

7.1 Teooria

Global descriptor table (GDT) on andmestruktuur, mille peamine ülesanne on protsessorile teada anda mälu segmentide kohta. GDT on vajalik katkestuste ja vigade käsitleja juures. Sisendid GDT tabelis on kaheksa baidi suurused ja koosnevad aadressist ja sisust. Milliseid GDT sisendeid kasutatakse sõltub, millises ruumis OS töötab, kas tuuma või kasutaja. Need ruumid või režiimid on eraldatud, et kaitsta mälu ja riistvara pahatahtliku või ekskliku tarkvara käitumise eest.

7.2 Implementatsioon

GDT koosneb *struct*'idest GDTDescriptor, GDTEntry ja GDT. GDTDescriptor annab protsessorile teada kus GDT asub ja kui suur see on. GDTEntry sisaldab GDT kõrgemaid ja madalamaid bitte. GDTEntry kirjeldab GDT segmenti, mis koosneb aladest *Base0*, *Limit0*, *Access_byte*, *Base1*, *Limit1_Flags* ja *Base2*. *Struct* GDT koosneb GDT sisenditest *KernelCode*, *KernelData*, *UserNull*, *UserCode* ja *UserData*. Esimene GDT sisend on alati tühi. *KernelCode* ja *KernelData* on kasutusel kui OS töötab tuuma ruumis ja *UserCode ja UserData* kui OS töötab kasutaja ruumis. GDT sisu on näha peatükis Lisa 3 koos inglise keelsete kommentaaridega.

Kasutades *assembly* programmeerimis keelt laetakse *GDTDescriptor* protsessorisse, et protsessor teaks, et GDT kasutada. *Assembly* kompileerimiseks on kasutuses NASM. *GDTDescriptor*'i laadimiseks vajalikku registrisse on funktsioon *LoadGDT.* Esimene väärtus, mis edastatakse GDT'le tuumast liigutatakse funktsiooni lgdt *(Load Global Descrptor Table*). Järgmisena uuendatakse koodi ja andmesegmentide registrid. Viimasena tehakse *far return* funktsioonist lgdt. *Far return*'i kasutatakse kui programm

kasutab 64 bitist *assembly*'t. Kogu *assembly* koodi on näha failis guOS/kernel/src/gdt/gdt.asm.

Kasutades *assembly*'s kirjutatud funktsiooni *LoadGDT*'d, laetakse GDT mällu failis guOS/kernel/src/KernelUtil.cpp. *Assembly* faili töövoogu on näha joonis 8 pealt.

8 Katkestuste ja vigade käsitleja

Katkestuste ja vigade käsitlejaga seotud kood asub kaustas guOS/kernel/src/Interrupts. *Panic screen*'i kood on failis guOS/kernel/src/panic.cpp.

8.1 Teooria

Katkestuste ja vigade käsitleja tegeleb väliste sisenditega ja võimalike vigadega, mis võivad esineda operatsioonisüsteemi töös. Kasutusel on GCC __attribute__((interrupt)), mis aitab luua katkestuse käsitlejaid kasutades C++'i. Katkestustel on erinevad tüübid, mida nimetatakse *trap*, *task* või *interrupt gate*'iteks. *Trap gate* on kasutuses erandite puhul, *interrupt gate*'i kasutatakse katkestusteenuse rutiini määramiseks (*Interrupt service routine*) ja *task gate* vahetab riistvaraülesandeid. *Panic screen* on vajalik, et anda kasutajale teada, millal tuum ei suuda enam oma tööd jätkata ja kuvada *debug* infot.

8.2 Implementatsioon

Katkestuste jaoks on vajalik *Interrupt descriptor table* (IDT), mis koosneb kahest *struct*'ist *IDTDescriptorEntry* ja IDTR. *IDTDescriptorEntry* sisaldab informatsiooni selle kohta, mida peab tegema kui katkestus püütakse. IDTR sisaldab informatsiooni IDT registrite kohta. IDT registrites on info IDT suuruse kohta ja IDT *offset*. *Offset* on 8 baidine IDT aadress.

IDTDescriptorEntry sisaldab kahte funktsiooni *SetOffset* ja *GetOffset* ja seitset muutujat: *offset0*, *selector*, *ist*, *type_attr*, *offset1*, *offset2* ja *ignore*. Funktsioon *SetOffset* määrab *offset*'i väärtused kasutades teatud *mask*'i ja nihutades väärtust teatud bittide kaupa paremale. Funktsioon *GetOffset* liidab loogilise võiga kõik *offset*'i väärtused ühte muutujasse ja tagastab selle. Muutuja *offset0* on *offset*'i 16 madalamat bitti, segment hoiab väärtust, mis ütleb, millisele segemdile minnakse üle kui katkestus püütakse. Muutuja *ist* on *Interrupt stack table*, mida siin OS'is ei kasutata, type_attr, mis määrab

IDT *descriptor entry* tüübi ehk kas see on *trap gate*, task *gate* või *interrupt gate*. *Offset1* ja *offset2* on ülejäänud 32 bitti pärast *offset0*'i. Muutuja ignore sisaldab bitte *IDTDescriptorEntry* lõpus, mida ei kasutata.

Lehekülje saalimisega seotud probleemide käsitlemise jaoks on *struct interrupt_frame* ja __attribute__((interrupt)) funktsioon *PageFault_Handler*, mille sisendiks on *struct interrupt_frame* frame*. Kui viga tekib trükitakse ekraanile sõnum *Page fault detected*.

Lisaks lehekülgede saalmise vigade käsitlejale on selles OS'is *DoubleFault_Handler* ja *GPFault_Handler*. *DoubleFault_Handler* kutsutakse kui on kaks käsitlematta viga samal ajal ja *GPFault_Handler* kutsutakse kui tekivad segmendi vead.

Failis guOS/kernel/src/KernelUtil.cpp on funktsioon PrepareInterrupts, mille sees luuakse IDT ja eraldatakse sellele mälu. *PageFault_Handler* funktsioonile määratakse *offset* funktsiooniga *SetOffset,* seatakse *Page fault descriptor entry* tüüp ja tunnused.

Panic screen'i jaoks kasutatakse funktsiooni *Clear*, mis puhastab ekraani ja muudab selle värvi. Funktsioon *Panic* seab ekraani punaseks funktsiooniga *Clear* ja trükib ekraanile sõnumi, mis on funktsiooni sisendiks. *Panic* funktsioon kutsutakse vigu käsitlevatest funktsioonidest nagu *PageFault_Handler*. *Panic screen*'i ekraanitõmmis on peatüki Lisa 3 all.

9 Sisend- väljundsüsteemide haldamine

Sisend- väljundsüsteemidega seotud kood on failis guOS/kernel/src/IO.cpp. Kood PIC kiibi kaardistamiseks on failis guOS/kernel/src/Interrupts/interrupts.cpp

9.1 Teooria

Kastkestusi kasutatakse PS/2 hiire ja klaviatuuri käsitlemisel. Selleks on vaja kaardistada PIC kiip. PIC kiibi kaardistamiseks on vaja funktsioone sisend-väljundsiinil.

9.2 Implementatsioon

Funktsioonid IO seadistamiseks on *outb* ja *inb. Outb* funktsioon paneb baidi IO siinile, selle sisenditeks on *port*, mis valib seadme IO siinil millega suhelda ja value, mis sisaldab väärtust, mida edastatakse. Funktsioon *inb* loeb väärtuse seadmelt millega suhtleb. Suhtlemisel vanemate seadmetega on andmevahetus aeglasem ja selle jaoks, et oodata enne järgmist saatmist on funktsioon *io_wait*.

Funktsioon PIC kiibi kaardistamiseks on *Remap_PIC*. PIC kiibi kaardistamiseks suheldakse kiibi *master* ja *slave* käsurea ja andmeliini. Kiibi kaardistamisel tuleb vältida seda, et katkestused ei põrkuks eranditega. Initialiseeritakse *master* ja *slave* PIC kiip. PIC kiipidele edastatakse nende *offset*'id, et vältida põrkumist eranditega. Määratakse, kuidas PIC kiibid koos töötavad ja millises režiimis need töötavad. Kasutades PIC kiipi luuakse katkestused klaviatuuri jaoks. Klaviatuuri sisend loetakse pordist ja peatakse katkestus edastades *master* või *slave* PIC kiibile *end of interrupt* käsk.

10 Klaviatuur

Klaviatuuriga seotud kood on kaustas guOS/kernel/src/userinput.

10.1 Teooria

See peatükk selgitab, kuidas PS/2 standardiga klaviatuur suhtleb OS'iga. Läbi virtualiseerimistarkvara on võimalik kasutada USB klaviatuuri ilma, et oleks vaja sellele eraldi funktsionaalsust luua. PS/2 klaviatuurid on sünkroonsed, kahesuunalised ja kasutavad IO porte suhtlemiseks riistvaraga. Kõik käsud täidetakse ühe kaupa ja andmed edastatakse klaviatuuri ja klaviatuurist.

Igal klaviatuuri klahvil on vastav 16 bitine väärtus ehk *scan code*, mis suhtleb teatud sisendiga riistvaras. Klahvidel nagu *caps lock* on eri funktsionaalsused.

10.2 Implementatsioon

Kasutades funktsioon KeyboardInt_Handler luuakse läbi PIC kiibi katkestus klaviatuuri kasutamiseks.

Klaviatuur kasutab US-QWERTY *scan code* komplekti, et tõlgendada klaviatuurilt edastatud *scan code* loetavasse teksti. Funktsioon Translate koosneb sisenditest *scancode* ja isLeftShiftPressed | isRightShiftPressed. *Scancode* on klaviatuurilt saadud klahvivajutus, mis tõlgendatakse täheks kasutades ascii tabelit ja loogiline või tehe *isLeftShiftPressed* | *isRightShiftPressed* on kasutusel kui hoitakse all *shift* klahvi ja trükitakse ekraanile suur täht.

Backspace, *shift* ja *enter* jaoks on *switch statement*, mis loeb *scancode* muutujat. *Shift*'i vajutades jäetakse vajutus klahvile meelde kuni klahv vabastatakse, et saaks *shift*'i all hoides kirjutada suurte tähtedega. *Enter* jaoks on funktsioon *Next*, mis lükkab kursori järgmisele reale. *Backspace* jaoks on funktsioon *ClearChar*. *ClearChar* kontrollib, et kursor ei läheks *framebuffer*'ist välja ja siis liigutab kursori tagasi ja kustutab tähe.

11 Hiir

Hiirega seotud kood on failides guOS/kernel/src/Interrupts/interrupt.cpp ja guOS/kernel/src/userinputs/mouse.cpp.

11.1 Teooria

Kasutusel on PS/2 standardiga hiir, mis suhtleb PS/2 kontrolleriga kasutades jadasidet. Kui hiir on lähtestatud, hakkab hiir saatma 3 kuni 4 baidiseid pakette, et edastada hiire liikumine ja hiirenupu vajutamise/vabastamise sündmused. Esimene bait sisaldab informatsiooni parema, vasaku ja keskmise hiirenupu kohta, lisaks info x'i ja y'i *overflow* ja *sign* bittide kohta. Baidid 2 ja 3 koosnevad infost hiire x ja y liikumise kohta. Neljas bait on info hiire lisanuppude kohta, mida selles OS'is ei kasutata.

11.2 Implementatsioon

Hiir töötab hiirele loodud katkestuse käsitlejaga nimega *MouseInt_Handle*r. Esmalt initialiseeritakse PS/2 hiir funktsiooniga *InitPS2Mouse*, mis lülitab lisaseadme sisse ja ootab kuni on vastus tagasi tulnud, et seadet saab kasutada. Hiirele käskude edastamiseks on funktsioon *MouseWrite*, millega seadistatakse hiir vaikeseadeid kasutama. Hiire seadmete kontrollimiseks on funktsioon *MouseRead*.

Saades 3 kuni 4 baiti informatsioon hiirelt kasutades funktsiooni *HandlePS2Mouse,* liidetakse need baidid üheks paketiks. See pakett sisaldab andmeid hiire x ja y asukoha kohta ja vasaku/parema hiirenupu kohta. Pakett loetakse funktsiooni *ProcessMousePacket* poolt, mis loeb välja teatud bitid ja muudab hiire asukohta või loeb nupuvahetusi.

Paketist loetakse vasaku, parema ja keskmise hiirenupu klikke funktsioonis *ProcessMousePacket*. Selles operatsioonisüsteemis hiire vasak nupp trükib hiire kursori asukohale valge a tähe ja parem nupp trükib rohelise a tähe.

Hiire kursor joonistatakse kasutades massiivi, mis ütleb hiire joonistamis funktsiionile *DrawOverlayMouseCursor*, millised pikslid kuvada, milliseid mitte. Kursorit joonistades pannakse paika piirangud, et ei saaks kursorit kuvada ekraanist väljaspool ja siis joonistatakse kursor asukohale kus hiir parasjagu asub. Hiire kursori asukoht jätakse meelde ja kustutakse eelmine kursor, mis ekraanil kuvati, et funktsioon pidevalt uut kursoril ekraanile ei trükiks. Hiire kursorit on näha peatükk Lisa 3 joonis 3 pealt.

12 PCI

PCI'ga seotud kood on failis guOS/kernel/src/acpi.cpp ja guOS/kernel/src/PCI.cpp.

12.1 Teooria

PCI tuleb kasuks kui kasutatakse PS/2 seadmete asemel USB klaviatuuri ja hiirt.USB seadmete kasutamiseks on vaja ühendust USB draiveritega, et seda teha on vaja ühendada PCI siiniga, et leida sellel olevad seaded.

UEFI'lt saadakse RSDP, mida kasutatakse MCFG tabeli leidmisel, et saada teavet PCI kohta. Liigutakse läbi kõik PCI siinid, seadmed ja funktsioonid, et luua ühendus kõigi PCI seadmetega.

12.2 Implementatsioon

PCI siiniga ühenduse loomiseks on vaja leida *root system description pointer* ja MCFG tabel UEFI alglaadurist. RSDP saadakse UEFI protokolliga *configuration* tabelist, mis kuulub süsteemi tabeli alla. Neid tabeleid kasutatakse, et UEFI'd kasutav programm pääseks süsteemi konfiguratsiooniteabele ligi. *Configuration* tabelist otsitakse *ACPI 2.0 grid system description pointer.* ACPI tabelist otsitakse RSDP signatuur kasutades lõnga võrdlemis funktsiooni. RSDP edastatakse tuumale kus sellega edasi tegeletakse. Kasutades RSDP'd on kergem leida teisi süsteemi tabeleid ja riistvara mis on saadaval.

Liikudes läbi RSDP leitakse MCFG ja XSDT tabelid, et nende sisu saaks trükkida ekraanile. Kasutades funktsiooni *FindTable* leitakse MCFG tabel. Funktsioon *FindTable* liigub läbi kõikide tabelite XSDT's ja otsib MCFG signatuuri.

MCFG tabelit saab kasutada, et saada infot PCI siini kohta. Selle jaoks on funktsioon *EnumeratePCI*, mille sisendiks on MCFG tabel. PCI siini seadmete kätte saamiseks on vaja funktsioone *EnumerateBus*, *EnumerateDevice* ja *EnumerateFunction*. EnumeratePCI funktsioon loeb üle kui palju on MCFG tabelis sissekandeid, et saada

ACPI seadme *config*. ACPI seadme *config*'i jaoks on *struct DeviceConfig*, mis koosneb muutujatest *BaseAddress*, *PCISegGroup*, *StartBus*, *EndBus* ja *Reserved*. Liikudes läbi iga seadme siini kutsutakse funktsioon *EnumerateBus*. Funktsioonis *EnumerateBus* kaardistatakse mälu, mida teatud aadressiga siin kasutab. Struct PCIDeviceHeader hoiab muutujaid, mis sisaldavad infot ja ID'si seadmete kohta. Kasutades infot PCI seadmete kohta kasutatakse funktsiooni *EnumerateDevice*, et liikuda läbi iga seadme siinil. Funktsioon *EnumerateFunction* on selleks, et liikuda läbi iga funktsiooni seadmes. *EnumeratePCI* kutsutakse tuumast.

Seadmete ja nende tootjate ID'd trükitakse ekraanile ja neid on nähe peatüki Lisa 3 joonis 3 pealt. Ekraani vasakus nurgas esimeses reas on 8086, mis on Intel'i tootja ID, 29c0 on *Express DRAM Controller*'i ID. Järgmisel real olev 1234 on QEMU virtualiseerimistarkvara graafika *controller*'i ID.

13 Eelmine läbikukkunud operatsioonisüsteem

Eelmise operatsioonisüsteemi kood on saadaval *githubis*. *Link* on peatükk Lisa 4 all. OS'i funktsioonid asuvad kasutades 64bit_kernel/src/intf ja 64bit_kernel/src/impl/x86_64, tuuma põhifunktsioon on failis 64bit_kernel/src/impl/kernel/main.c ja alglaaduri poolt loetav kood on kaustas 64bit_kernel/src/impl/x86_64/boot. Kood on inglise keeles kommenteeritud.

Operatsioonisüsteem on kirjutatud peamiselt C keeles kasutades *Assembly*'t alglaaduriga suheldes. Kompileerimine ja *build*'imine on tehtud *Docker* programmi abil. *Assembly* kompileerimiseks on kasutuses NASM.

Operatsioonisüsteemil oli 64-bitise tuumaga, kasutas x86 arhitektuuri. Loodud oli lihtne stack ja lehekülgede saalimine, trükkimine ekraanile kasutades C programmeerimiskeelt ja osaliselt oli implementeeritud klaviatuuri funktsionaalsus ja mäluga töötamise funktsioonid. Alglaadur töötab *multiboot2*'ga.

64-Bitise tuuma jaoks on vaja protsessor kõige pealt lülitada *long mode*'i. Enne seda tuleb kontrollida, kas protsessor *long mode*'i toetab. Kontroll tehakse vaadates protsessori ID'd. Kui protsessor long mode'i ei toeta siis trükitakse ekraanile *error* kood. *Long mode*'i lülitamiseks on vaja lehekülgede saalimist. Lehekülgede saalimisel oli igale lehekülje tabelile eraldatud 4 kilobaiti mälu.

Operatsioonisüsteem kasutas *stack*'i, mis sisaldas funktsioonikutseid, mida lahendati programmi töö käigus. *Stack*'i jaoks oli eraldatud 16 kilobaiti mälu. Protessor kasutab esb registrit, et teada millist *stack frame*'i hetkel loetakse.

Selle operatsioonisüsteemi jätsin pooleli kuna ei osanud lahendada probleemi, mis tekkis ekraanile trükkimisel. Trükkides ekraanile klaviatuurilt rohkem kui 3 tähte tegi OS restardi. Probleem oli tõenäoliselt selles, kuidas OS mälu haldas kasutades stack'i . Võimalik oli, et toimus puhvri ületäitumine või oli probleem lehekülgede saalimisel Kuna mul puudusid oskused ja teadmised *assembly* keeles, et testida kas viga on puhvri

ületäitumises, jätkasin tööd alustades uue OS’iga, millel on detailsem juhend, eriti mälu haldamise teemadel.

14 Edasi liikumine

Järgnevates alampeatükkides on toodud esile kõik operatsioonisüsteemi osad, mida saaks täiendada või osad mida juurde lisada. Juurde on lisatud võimalikud viisid, kuidas neid implementeerida.

14.1 Mitmete protsesside käsitlemine

Mitmete protesesside käsitlemine toimub *context switch*'idega. *Context switch* toimub kui tuum kutsutakse rakenduse poolt välja. *Context switch* jätab meelde vana oleku ja hakkab uut olekut hankima.

Alustades operatsioonisüsteemi loomist soovitas juhendaja selgitada välja millist operatsioonisüsteemi ehitada ja kuidas see protsesse käsitleb. Üks soovitustest oli luua reaalaja operatsioonisüsteem kasutades *round-robin* ressursijaotus algoritmi kus igale protsessile antakse võrdselt aega ja iga protsess on sama prioriteediga. Lisaks sellele uurisin multitegum ressursijaotust, mis on keerukam kui *round-robin* meetod. Multitegum süsteemid jagavad saadaolevat protsessori aega automaatselt mitme ülesande vahel.

14.2 Lehekülgede saalimine

Lehekülgede saalimine on implementeeritud praeguses operatsioonisüsteemis poolikult. Täiendada võiks lehekülgede nõudmist. Praegu kui leheküljed saavad otsa siis tagastatakse *NULL,* kuid valmis süsteemis suhtleb page frame allocator kõvakettaga ja salvestab lehekülgi sinna kui neid puudu jääb. Seda ei implementeeritud kuna puudub kõvakettaga suhtlemis funktsionaalsus.

14.3 Kõvakettalt lugemine

Kasutades *Advanced Host Controller Interface* (AHCI) draiverit on võimalik lugeda kõvakettalt. AHCI draiveriga on võimalik seadistada SATA seadmeid, mille alla kuuluvad kettaseadmed ja prodikordistajad.

14.4 Vigade käsitleja

Panic screen peab lisaks error sõnumile kuvama ka ekraanile debug info, mida praegune OS ei trüki. Debug info saab trükkida ekraanile näiteks *General Protection* vigade puhul. *General Protection* vigade puhul salvestatud käsu viit (*saved instruction pointer*) osutab juhisele, mis põhjustas erandi, mille saab trükkida ekraanile *debug* eesmärkidel.

14.5 Hiir

Hiir trükib ekraanile vasakut nuppu klikkides valge a tähe ja parem nupp rohelise a tähe. Peale selle muud funktsionaalsust hiirel pole ja on ainult visuaalne kuna pole midagi mida hiir saaks mõjutada näiteks GUI.

15 Kokkuvõte

Viimane esitatud versioon operatsioonisüsteemist sarnaneb kõige enam personaalarvuti OS'iga. Funktsionaalsuse ja visuaali poolt on see OS alus sellisele operatsioonisüsteemile, millest võiks saada personaalarvuti OS. Kuigi on muid erinevaid OS'i tüüpe, mida oleks võinud implementeerida, kulges töö personaalarvuti funktsionaalsuste poole.

Operatsioonisüsteemi ehitamine on väga aega nõudev töö. Lisaks on vaja väga hästi tunda programmeerimiskeeli ja kuidas arvuti madalamal tasemel töötab. Kuigi ma tegin palju uurimistööd operatsioonisüsteemide teemal enne tööle asumist jäi ikkagi palju teadmistest tihti puudu eriti implementeerides eri osi, mis oli peamine põhjus, miks esimene OS nurjus. Neli kuud septembrist detsembrini on väga vähe aega operatsioonisüsteemi loomiseks, kuid lõpptulemus on töötavate funktsionaalsustega OS'i alus, mida on võimalik edasi arendada.

Kasutatud kirjandus

[1] CodePulse: Write Your Own 64-bit Operating System, https://youtu.be/FkrpUaGThTQ, vaadatud 3.1.2023.

[2] Poncho: OSDev – S2, https://www.youtube.com/playlist?list=PLxN4E629pPnJxCQCLy7E0SQY_zuumOVyZ, vaadatud 3.1.2023.

[3] Ed Nutting: FlingOS: Creating an OS, https://www.youtube.com/playlist?list=PLKbvCgwMcH7BX6Z8Bk1EuFwDa0WGkMnrz, vaadatud 3.1.2023.

[4] IKnow: How to make an operating system from scratch, https://www.youtube.com/watch?v=rr-9w2gITDM&t=0s, vaadatud 3.1.2023.

[5] OSDev wiki, https://wiki.osdev.org/Main_Page, vaadatud 3.1.2023

[6] Vikipeedia: Operatsioonisüsteem, https://et.wikipedia.org/wiki/Operatsioonis%C3%Bcsteem, vaadatud 3.1.2023

[7] Wikipedia: Operating System, https://en.wikipedia.org/wiki/Operating_system, vaadatud 3.1.2023

[8] Vaibhav Kandwal: UEFI vs. BIOS: What's the Difference?, https://www.freecodecamp.org/news/uefi-vs-bios/, vaadatud 3.1.2023

Lisa 1– Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Maker Padari

1 Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose Lihtne operatsioonisüsteem, mille juhendaja on Peeter Ellervee

1.1 reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

1.2 üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2 Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.

3 Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

03.01.2023

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 – Koodi näited

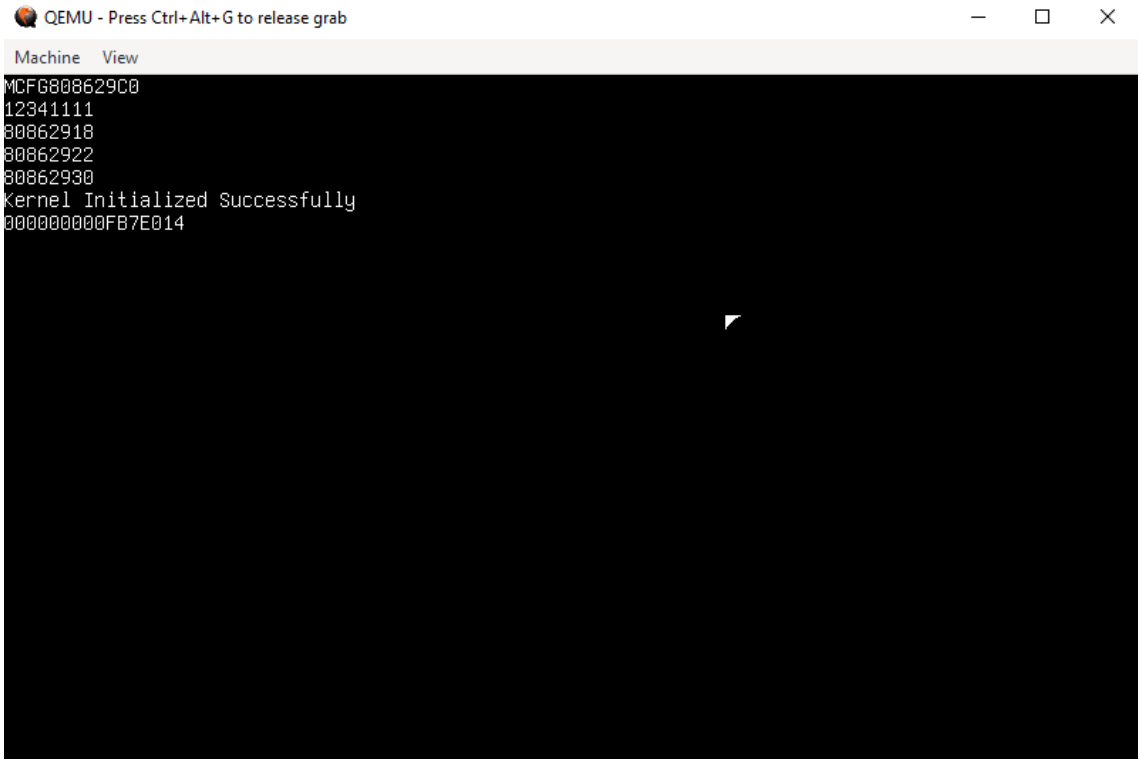
```
#include "efiMemory.h"
/*
EFI memory is used to find out which sections of memory are availabe for the
kernel to use
*/


/* Memory type strings, adding all these together will equal to total memory
allocated for the system */
const char* EFI_MEMORY_TYPE_STRINGS[] {

    "EfiReservedMemoryType",
    "EfiLoaderCode",
    "EfiLoaderData",
    "EfiBootServicesCode",
    "EfiBootServicesData",
    "EfiRuntimeServicesCode",
    "EfiRuntimeServicesData",

    // Memory section UEFI has not used that is free to use in the kernel
    "EfiConventionalMemory",
    "EfiUnusableMemory",

    // Memory that can be used after getting info from ACPI tables
    "EfiACPIReclaimMemory",
    "EfiACPIMemoryNVS",
    "EfiMemoryMappedIO",
    "EfiMemoryMappedIOPortSpace",
    "EfiPalCode",

};
```

Joonis 1. efiMemory.cpp sisu, mälu tüüpide lõngad koos inglise keelsete kommentaaridega

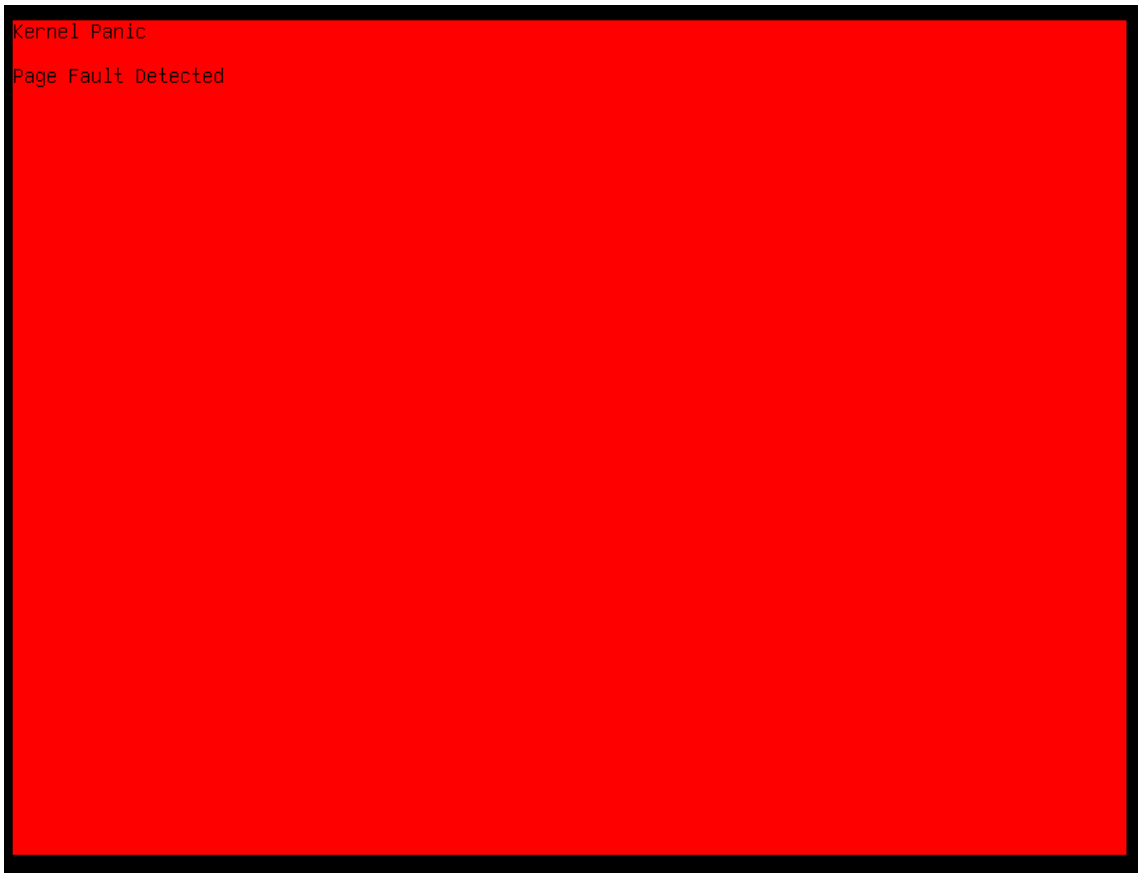
```
#include "gdt.h"
/*
Global Descriptor Table is a data struct used by the CPU to know specific
things about memory segments
*/


__attribute__((aligned(0x1000)))
GDT DefaultGDT = {
    {0, 0, 0, 0x00, 0x00, 0}, // null
    {0, 0, 0, 0x9a, 0xa0, 0}, // kernel code segment
    {0, 0, 0, 0x92, 0xa0, 0}, // kernel data segment
    {0, 0, 0, 0x00, 0x00, 0}, // user null
    {0, 0, 0, 0x92, 0xa0, 0}, // user data segment
};
```

Joonis 2. gdt.cpp sisu inglise keelsete kommentaaridega

Lisa 3 – Ekraanitõmmised



Joonis 3. Operatsioonisüsteem QEMU virtualiseerimistarkvaras, näha on kursor ja PCI testid



Joonis 4. *Panic screen*

Lisa 4 – Koodi link

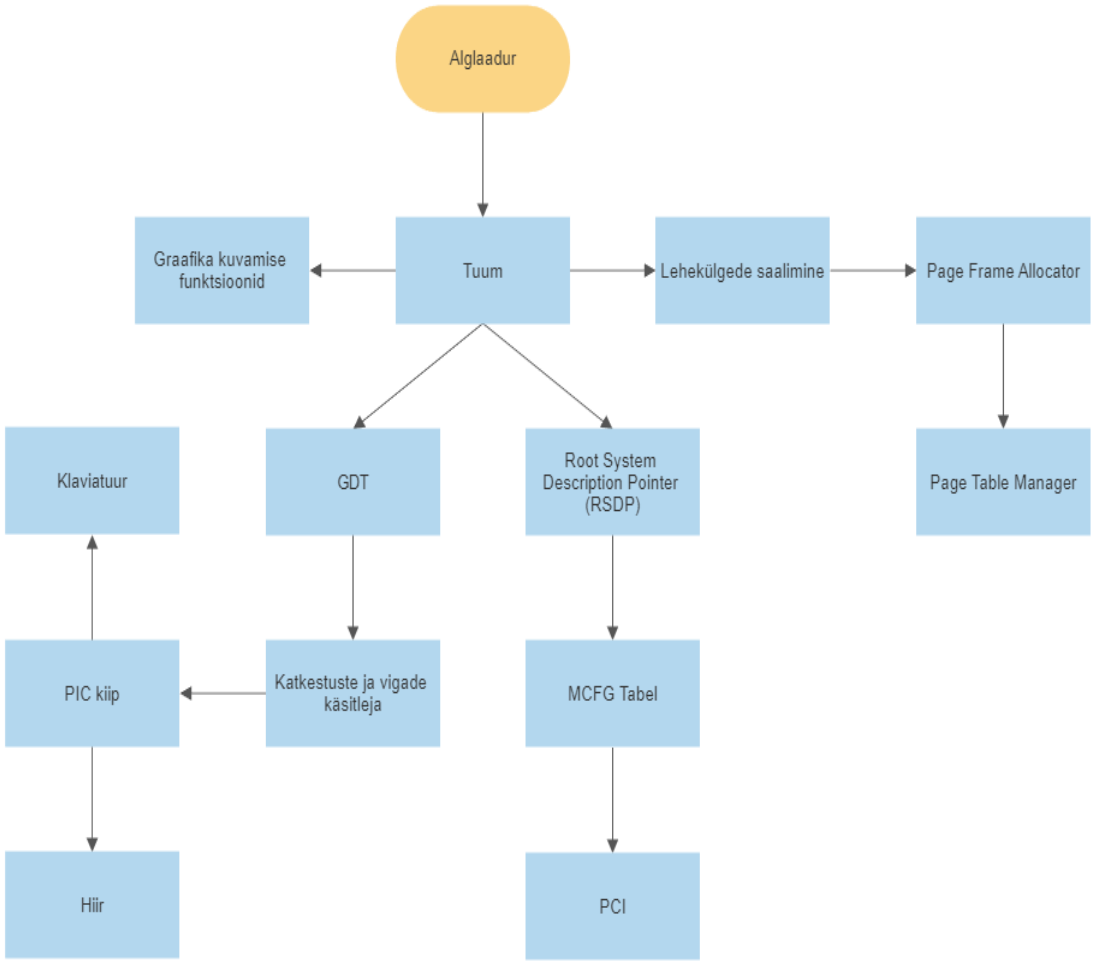
Operatsioonisüsteem kood Github'is:

https://github.com/makerpadari2/guOS

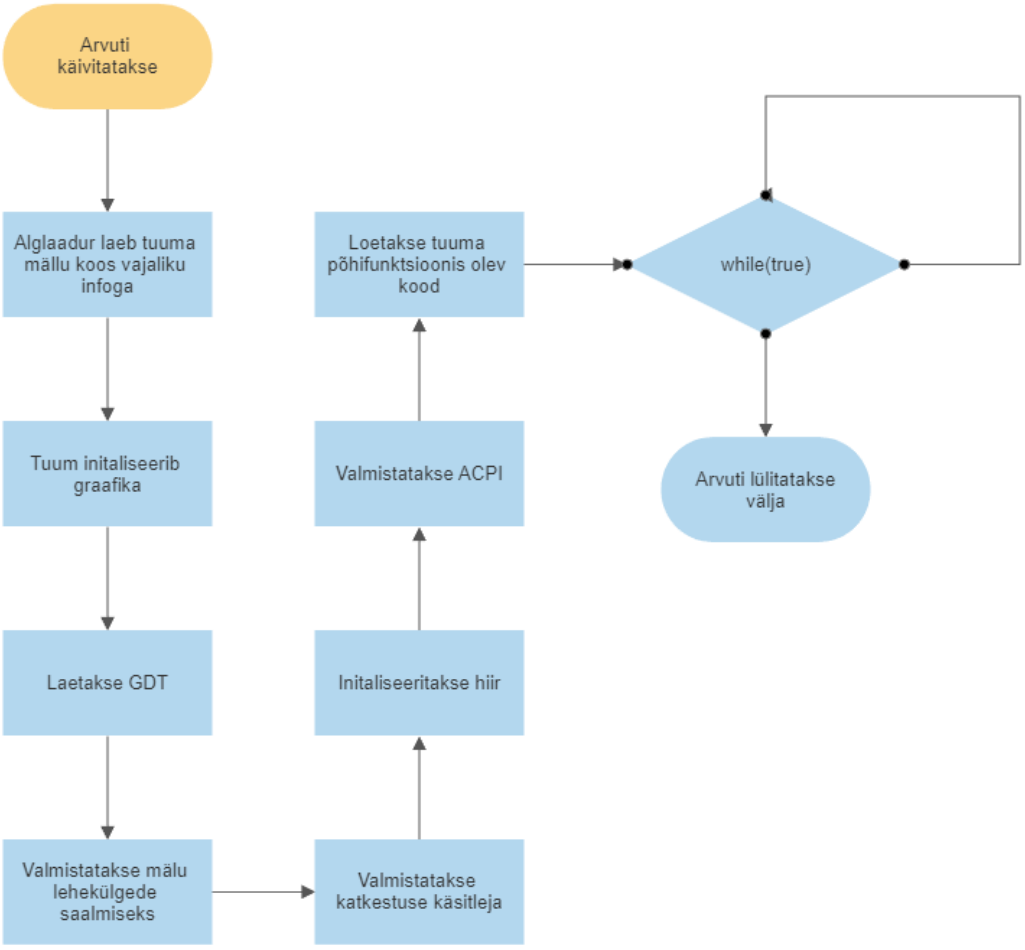
Vanem operatsioonisüsteem:

https://github.com/makerpadari2/guOS0.1/tree/main/64bit kernel

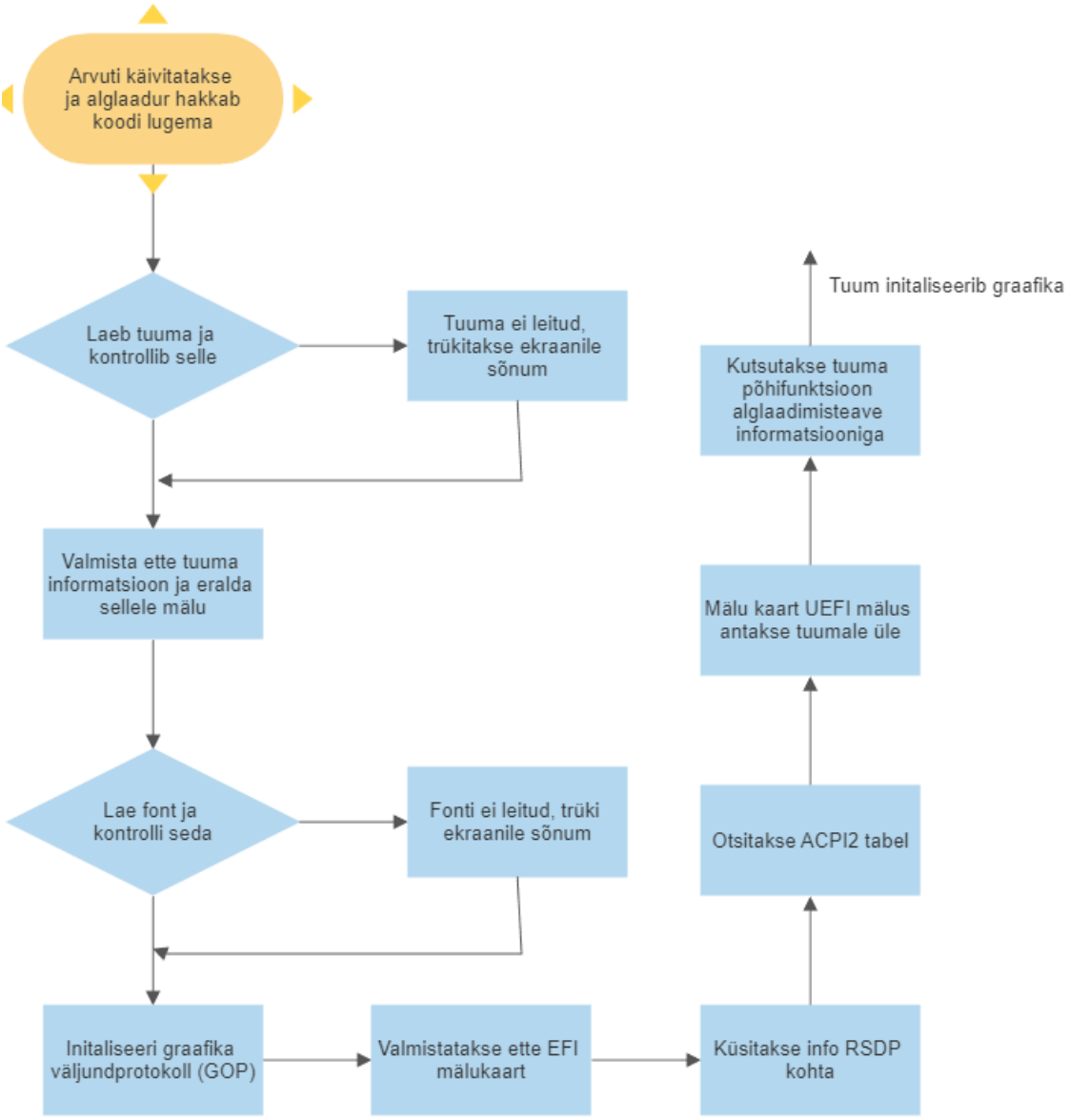
Lisa 5– Plokkskeemid ja vooskeemid



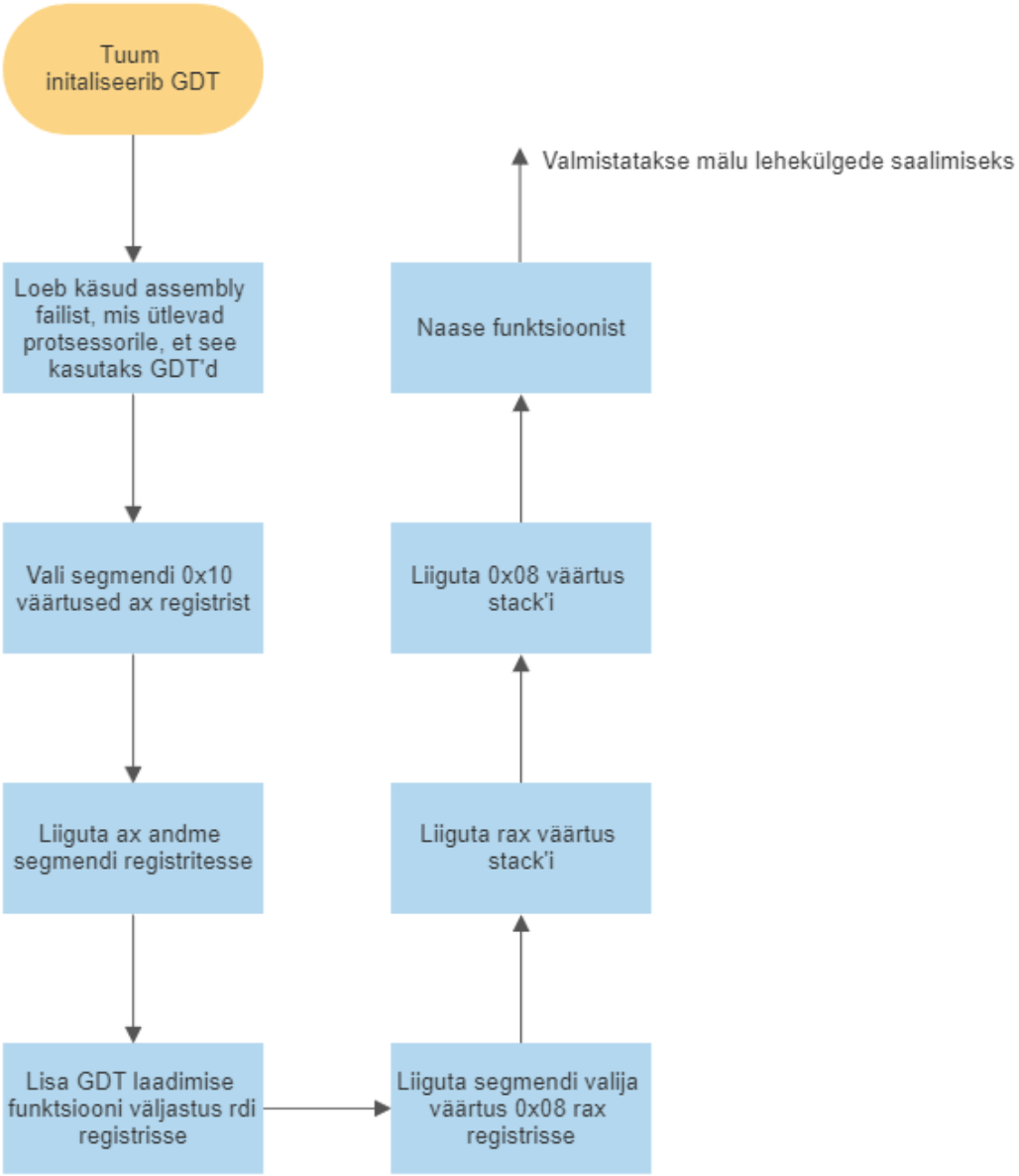
Joonis 5. Operatsioonisüsteemi plokkskeem



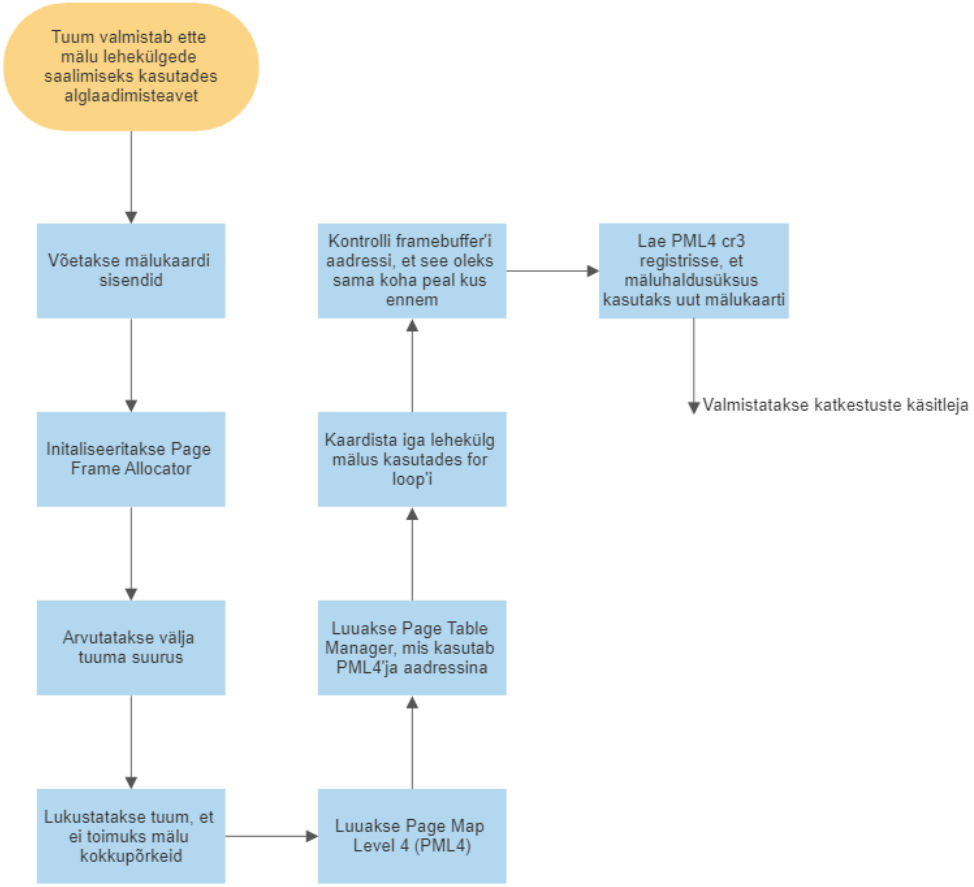
Joonis 6. Käivitamisprotsess



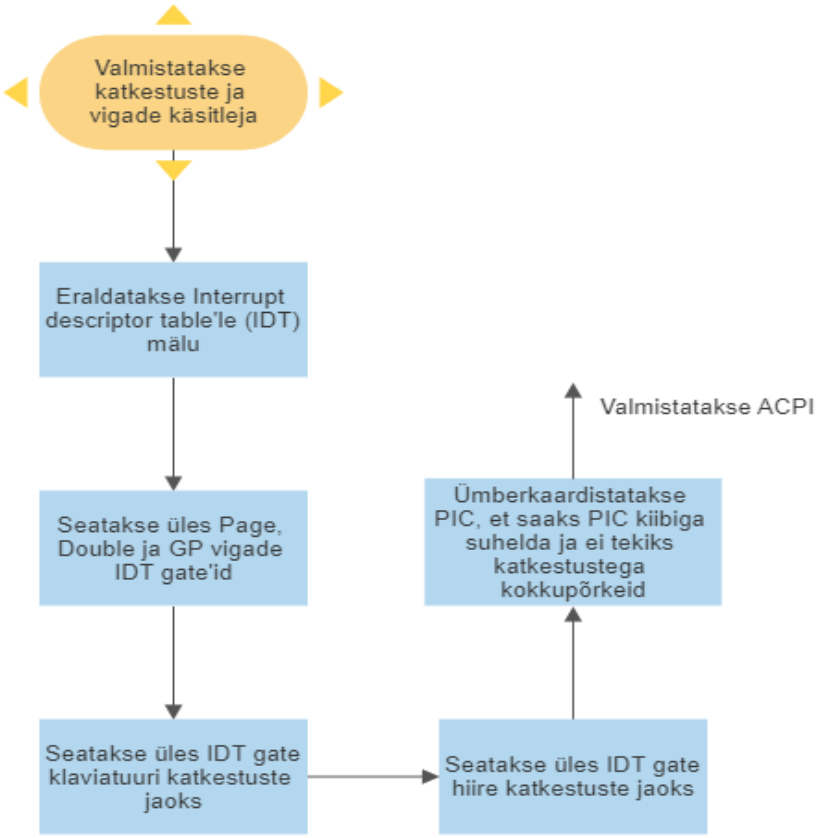
Joonis 7. Alglaaduri töövoog



Joonis 8. GDT initialiseerimine, gdt.asm töövoog



Joonis 9. Lehekülgede saalimise vooskeem



Joonis 10. Katkestuste ja vigade käsitleja vooskeem