

IZRADA JEDNOSTAVNOG OPERATIVNOG SUSTAVA

Abramović, Zdravko

Undergraduate thesis / Završni rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Split / Sveučilište u Splitu**

Permanent link / Trajna poveznica: <https://um.nsk.hr/um:nbn:hr:228:000312>

Rights / Prava: [In copyright](#)/[Zaštićeno autorskim pravom.](#)

Download date / Datum preuzimanja: **2024-12-04**



Repository / Repozitorij:

[Repository of University Department of Professional Studies](#)



UNIVERSITY OF SPLIT



SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informatička tehnologija

ZDRAVKO ABRAMOVIĆ

ZAVRŠNI RAD

**IZRADA JEDNOSTAVNOG OPERATIVNOG
SUSTAVA**

Split, rujan 2021.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informatička tehnologija

Predmet: Operacijski sustavi

ZAVRŠNI RAD

Kandidat: Zdravko Abramović

Naslov rada: Izrada jednostavnog operativnog sustava

Mentor: Ljiljana Despalatović, viši predavač

Split, rujan 2021.

Sadržaj

Sažetak	4
Summary	5
1 Uvod	6
2 Implementacija.....	7
2.1 Alati.....	7
2.1.1 NASM.....	7
2.1.2 gcc.....	7
2.1.3 Bash	7
2.1.4 QEMU.....	7
2.1.5 Vim.....	7
2.1.6 GIMP	8
2.2 Potrebna predznanja	8
2.2.1 BIOS.....	8
2.2.2 ISA	8
2.2.3 Asembler.....	9
2.2.4 <i>Bootloader</i>	9
2.2.5 Jezgra	9
2.2.6 Pogonski programi.....	10
2.3 <i>Bootloader</i>	10
2.4 Učitavanje kôda s diska u radnu memoriju	15
2.5 Prebacivanje u <i>Protected Mode</i>	18
2.6 Pogonski program za tekstualnu manipulaciju ekranom	25
2.7 Jezgra	34
2.8 <i>Cross-compiler</i>	35
2.9 <i>Makefile</i>	39
3 Zaključak	41
4 Literatura.....	42

Sažetak

Cilj ovog završnog rada bio je razviti funkcionalan operacijski sustav (engl. *operating system*) „od nule”. Iako je poprilično jednostavan, usporedivši ga s modernim operacijskim sustavima široke uporabe, pruža uvid u osnove njihovog funkcioniranja i razvoja.

Prvi je korak bio napisati njemu pripadajući pokretač operacijskog sustava (engl. *bootloader*), u ovom slučaju u MBR (engl. *Master Boot Record*) formatu. Zatim taj *bootloader* učitava narednih „x” sektora sa diska (nakon prvog, koji je sâm *bootloader*). Učitavanje tih sektora odnosi se na učitavanje kôda jezgre (engl. *kernel*) koji se nalazi u njima.

Zadano procesor kôd počinje izvršavati u tzv. *Real Mode*-u, odnosno interpretira instrukcije kao 16-bitne. Kako bi imao proširen set mogućnosti potrebno ga je prebaciti u 32-bitni zaštićeni način rada (engl. *Protected Mode*, u daljnjem tekstu „PM”). U PM-u procesor dobiva mogućnost adresiranja, odnosno korištenja, mnogo više memorije nego što je imao u 16-bitnom načinu, a također se otvara i mogućnost zaštite memorije. Prebacivanje u PM podrazumijeva određene pripreme koje su objašnjene u glavnom dijelu rada.

Nakon svega toga počinje se izvršavati kôd operacijskog sustava, za potrebe ovog rada napisan u programskom jeziku C. Video memorija koristi se u tekstualnom načinu rada koji je jednostavniji jer se pikselima (engl. *pixel – picture element*) ne pristupa direktno.

Za pokretanje kôda koristio se emulator QEMU (engl. *Quick Emulator*), ali treba imati u vidu da ga je moguće pokrenuti i preko nekog tipa stvarnog diska (npr. CD-ROM-a, tvrdog diska, memorijskog štapića itd.) direktno na bilo kojem 32 ili 64-bitnom računalu x86 arhitekture. Točnije rečeno, na procesoru koji implementira x86 set instrukcija (engl. *instruction set*). U praksi praktički na bilo kojem današnjem stolnom (engl. *desktop*) računalu.

Ključne riječi: *assembler, bootloader, jezgra, low-level, operacijski sustav*

Summary

Simple operating system

The goal of this undergraduate thesis was to develop functional operating system (OS) from scratch. Even though it's pretty simple, comparing it to the modern widespread operating systems, it provides insight into the basics of their functioning and development.

The first step was writing it's appurtenant bootloader, in this case in MBR (Master Boot Record) format. Then that bootloader loads subsequent „x” number of sectors from the disk (after the first one, which is the bootloader itself). Loading of sectors is about loading the kernel code which resides inside them.

By default, CPU (Central Processing Unit) starts to execute code in the so-called Real Mode, i.e. it interprets code as a set of 16-bit instructions. To widen its options, it's necessary to make a switch to 32-bit Protected Mode (hereinafter PM). Being in PM, CPU gains access to much more memory than it had on disposition in the 16-bit mode. Also, it opens up a way to implement memory protection. Switch to PM implies certain setup steps which are explained in the main part.

After all that the OS code, for this thesis' needs written in programming language C, starts to execute. Video memory is used in text mode which is simpler because pixels (picture elements) are not accessed directly.

Code was executed with QEMU (Quick Emulator), but it must be noted that it's possible to execute it using some type of real-life disk (e.g. CD-ROM, hard disk, USB stick etc.) directly on any 32 or 64-bit computer which implements x86 Instruction Set; virtually on any modern desktop computer.

Keywords: *assembler, bootloader, kernel, low-level, operating system*

1 Uvod

Velik broj ljudi svakodnevno se služi računalima. Glavna motivacija za odabir ove teme za završni rad je želja da se bolje upozna oprema koja je za današnji način života postala neophodna, imajući u vidu sve znatizeljne, ali prvenstveno programere kojima je računalo sredstvo rada, a zapravo nisu baš upoznati s njim. Zato je ovaj rad zamišljen kao koherentan i sažet prikaz procesa koji se događaju od uključivanja računala do točke u kojoj se ono počinje koristiti; uz korištenje jednostavnih pogonskih programa (engl. *driver*), ručno napisanih i bez pomoći biblioteka. Kôd je preuzet iz knjige *Writing a Simple Operating System from Scratch* i modificiran [1]. Važno je napomenuti da i ovaj rad ostavlja dosta toga skrivenog, jer se svaki detalj za sebe može detaljno objasniti. Npr. kako funkcionira elektronika, BIOS (engl. *Basic Input/Output System*), kompajler (engl. *compiler*) itd.

Rad je podijeljen na četiri dijela. Prvi dio je uvod. Drugi dio, implementacija, podijeljen je na devet potpoglavlja. U prvom se potpoglavlju navode alati korišteni za izradu rada. U drugom se ukratko opisuju pojmovi koje je potrebno okvirno razumijeti za razumijevanje samog rada. U trećem se prikazuje način kodiranja *bootloadera* i njegov odnos s BIOS-om. U četvrtom se obavlja učitavanje (ostalog) kôda s diska u memoriju. U petom prebacivanje u zaštićeni način (engl. *Protected Mode*). U šestom je opisan pogonski program za ispis teksta na ekran. U sedmom jezgra koja koristi taj pogonski program. U osmom potpoglavlju objašnjen je *cross-compiler* koji je bio nužan za ispravno funkcioniranje kôda. I naposljetku u devetom *Makefile* koji je značajno ubrzao razvoj sustava.

Nakon implementacije slijede kratak zaključak i korištena literatura.

2 Implementacija

2.1 Alati

Glavni alati koji su se koristili u ovom radu su: NASM, gcc, bash, QEMU, Vim i GIMP. U nastavku će se svaki od njih ukratko opisati.

2.1.1 NASM

NASM (engl. *Netwide Assembler*) je assembler (engl. *assembler*) koji se koristio za pretvaranje asemblerskog kôda u izvršne (engl. *executable*) ili objektne (engl. *object*) datoteke.

2.1.2 gcc

gcc (eng. *GNU Compiler Collection*) je kompajler koji se koristio za kompajliranje kôda napisanog u programskom jeziku C.

2.1.3 Bash

Bash (engl. *Bourne Again Shell*) je ljuska (engl. *shell*) i naredbeni jezik (engl. *command language*) koji se koristio za izvršavanje naredbi.

2.1.4 QEMU

Za pokretanje kôda koristio se QEMU (engl. *Quick Emulator*). Kôd se može pokretati i u stvarnoj okolini, ali je preko emulatora to puno brže i praktičnije.

2.1.5 Vim

Za pisanje kôda koristio se uređivač teksta Vim (engl. *Vi Improved*).

2.1.6 GIMP

Za uređivanje slikovnih prikaza koristio se uređivač slika GIMP (engl. *GNU Image Manipulation Program*).

2.2 Potrebna predznanja

Kako bi se razumio ovaj rad potrebno je nešto znanja o ISA (engl. *Instruction Set Architecture*), asembleru, BIOS-u, *bootloaderu*, jezgri (engl. *kernel*) i pogonskim programima (engl. *driver*).

2.2.1 BIOS

BIOS (engl. *Basic Input Output System*) je program koji je utisnut u čip (engl. *chip*). Ostaje zapisan i nakon što računalo ostane bez električne energije, za razliku od npr. radne memorije (engl. RAM – *Random Access Memory*).

Kada se računalo pokrene procesor prvo poziva BIOS. BIOS prvo izvodi POST (engl. *Power-on self-test*) kojim provjerava je li sve u redu s hardverom računala (engl. *hardware*). Zatim određenim redoslijedom traži na kojem se disku nalazi ”*boot sector*” s kojeg bi se trebao pokrenuti OS.

Općenito mu je glavna svrha „posredovanje” između OS-a i hardvera.

Iako je u novije vrijeme BIOS zamijenjen UEFI-jem (engl. *Unified Extensible Firmware Interface*) mnoga starija računala funkcioniraju pomoću njega, a i novija ga, još uvijek, uglavnom podržavaju zbog kompatibilnosti [2].

2.2.2 ISA

ISA specificira kako se svaka instrukcija interpretira. Ista instrukcija uvijek mora dati isti rezultat.

ISA je zapravo sučelje. Zato npr. Intelov i AMD-ov procesor mogu imati različitu implementaciju, ali oboje mogu implementirati istu ISA-u, odnosno mogu izvršavati isti strojni kôd.

Kôd napisan za ovaj rad može se izvršiti na bilo kojem računalu koje implementira instrukcijski set x86 (engl. *x86 Instruction Set*).

2.2.3 Asembler

Asembler (engl. *assembly*) je programski jezik najniže razine. Prevodi se direktno u strojni kôd, a čovjeku je razumljiv. Svaki se viši programski jezik naposljetku prevodi u asembler.

Prednosti asemblera su brzina i mogućnost fine kontrole. Razumijevanje funkcioniranja računala praktički nije moguće bez poznavanja asemblera.

Svaka ISA ima svoju asemblersku sintaksu. Asembler (engl. *assembler*) u drugom smislu riječi označava program koji pretvara kôd napisan u asemblerskom jeziku u strojni kôd. U ovom se radu koristio asembler NASM koji je pogodan za x86 ISA-u za koju je kôd napisan.

2.2.4 Bootloader

Bootloader je mali program koji pokreće operacijski sustav. Nalazi se u prvom sektoru "bootabilnog" diska. Mora se nalaziti tu jer ga BIOS tu „očekuje”. Veličine je jednog sektora, obično 512 bajta.

2.2.5 Jezgra

Pojam jezgra, kao što i ime kaže, odnosi se na osnovni dio operacijskog sustava. Jezgra je spona koja povezuje hardver i programsku podršku (engl. *software*). Ona upravlja pogonskim programima, obavlja zadaće koje korisničkim programima nisu dopuštene itd.

Najjednostavniji i najbrži tip jezgre je monolitna. Kod takvog tipa usluge (engl. *services*) jezgre i korisničke usluge dijele adresni prostor. Mana monolitne jezgre je što se u slučaju da se sruši samo jedna usluga ruši cijeli sustav. Ovaj rad koristi monolitnu jezgru, ali moguće ju je u budućnosti pretvoriti u neki drugi tip.

2.2.6 Pogonski programi

Pogonski programi služe za upravljanje hardverom. Tako postoje pogonski programi za grafičku karticu, zvučnu karticu, tipkovnicu itd. Kako bi se napisao pogonski program za određeni uređaj potrebno je točno poznavati njegovu arhitekturu. Npr. u kojim unutarnjim registrima sprema koje podatke, kojim je sabirnicama (engl. *bus*) spojen, itd.

Pogonski programi omogućuju jezgri lakše upravljanje hardverom oslobodivši je njegovih fizičkih karakteristika. Služe kao sučelje prema hardveru.

2.3 Bootloader

Za pokretanje *bootloadera* zadužen je BIOS. BIOS *bootloader* traži u prvom sektoru diska (tvrdog diska, memorijskog štapića, CD-a i sl.) znanom kao "*boot sector*", čija je adresa, koristeći CHS adresiranje: (0, 0, 0) (*Cylinder 0, Head 0, Sector 0*) [3].

BIOS "*bootabilni*" disk prepoznaje po zadnja dva bajta njegovog prvog sektora, tzv. čarobnom broju (engl. *magic number*) 0xAA55. Ako se u ta dva bajta ne nalazi čarobni broj, BIOS zaključuje da se na promatranom disku ne nalazi *bootloader*.

Bootloader se može napisati direktno u strojnom kôdu, ali jednostavnije je to učiniti u assembleru. Ispod su prikazane zadnje dvije linije kôda *bootloadera*.

\$ - trenutna linija kôda,
\$\$ - prva linija kôda

<code>times 510-(\$\$\$) db 0 ; ubacuju se nule kako bi carobni broj bio na pravom mjestu</code>	Ispis 1: Dno bootloadera
--------------------------------------------------------------------------------------------------	--------------------------

Ako u prvom sektoru ostane mjesto ostatak se nadopuni nulama kako bi čarobni broj bio na pravom mjestu. U tu se svrhu koristi instrukcija `times`.

Instrukcijom `dw` (engl. *define word*) definiraju se dva bajta koja označavaju čarobni broj.

x86 arhitektura koristi tzv. *little-endian* način zapisivanja riječi (engl. *word*) što znači da se prvo zapisuje niži bajt.

Unosom naredbe:

```
od -t x1 -A n boot_sect.bin #za hex prikaz po bajtovima
```

može se vidjeti kako je čarobni broj `0xAA55` zapisan u obrnutom smjeru.

```
razul@Ahrenaion...
razul@Ahrenaion:~/TheSys/boot$ od -t x1 -A n boot_sect.bin
88 16 ec 7c bd 00 90 89 ec bb ed 7c e8 08 00 e8
b6 00 e8 69 00 eb fe 50 53 b4 0e 80 3f 00 74 09
8a 07 cd 10 83 c3 01 eb f2 5b 58 c3 50 51 52 b4
02 88 f0 b5 00 b6 00 b1 02 cd 13 72 08 5a 38 c6
75 03 59 58 c3 bb 4d 7c e8 cc ff eb fe 44 69 73
6b 20 72 65 61 64 20 65 72 72 6f 72 21 0d 0a 00
00 00 00 00 00 00 00 00 ff ff 00 00 00 9a cf 00
ff ff 00 00 00 92 cf 00 17 00 60 7c 00 00 fa 0f
01 16 78 7c 0f 20 c0 66 83 c8 01 0f 22 c0 ea 93
7c 08 00 66 b8 10 00 8e d8 8e d0 8e c0 8e e0 8e
e8 bd 00 00 09 00 89 ec e8 2e 00 00 00 60 ba 00
80 0b 00 b4 0f 8a 03 3c 00 74 0b 66 89 02 83 c3
01 83 c2 02 eb ef 61 c3 bb 3e 7d e8 49 ff bb 00
10 b6 0f 8a 16 ec 7c e8 52 ff c3 bb 0e 7d 00 00
e8 c8 ff ff ff e8 16 93 ff ff eb fe 00 53 74 61
72 74 65 64 20 69 6e 20 31 36 2d 62 69 74 20 52
65 61 6c 20 4d 6f 64 65 2e 2e 2e 0d 0a 00 53 75
63 63 65 73 73 66 75 6c 6c 79 20 6c 61 6e 64 65
64 20 69 6e 20 33 32 2d 62 69 74 20 50 72 6f 74
65 63 74 65 64 20 4d 6f 64 65 2e 2e 2e 00 4c 6f
61 64 69 6e 67 20 6b 65 72 6e 65 6c 20 69 6e 74
6f 20 6d 65 6d 6f 72 79 2e 2e 2e 0d 0a 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
razul@Ahrenaion:~/TheSys/boot$
```

Slika 1: *Bootloader* - strojni kôd

Naredba od zadano izbacuje oktalni zapis, ali se pomoću parametara može odabrati heksadecimalni zapis po bajtovima.

Treba

```
; Boot sektor koji CPU stavlja u 32-bitni Protected Mode i prepusta kontrolu jezgri
[org 0x7c00] ; BIOS boot sektor ucitava na ovo mjesto u memoriji
KERNEL_OFFSET equ 0x1000 ; mjesto gdje ce se kasnije učitati jezgra; konstanta
mov [BOOT_DRIVE], dl
mov bp, 0x9000 ; postavljanje stacka, 0x7c00 + 10*512
mov sp, bp
mov bx, MSG_REAL_MODE
call print_string
call load_kernel ; učitavanje jezgre s diska
MSG_REAL_MODE db "Started in 16-bit Real Mode...", 0xD, 0xA, 0
```

Ispis 2: Početak *bootloadera*

napomenuti da se "endianness" odnosi samo na skup bajtova. Svaki bajt posebno uvijek je "big-endian".

Sljedeći ispis prikazuje prvi, gornji dio *bootloadera*.

Direktiva [org 0x7c00] označava ishodište, odnosno znači da svako korištenje adrese u kôdu, bilo u obliku labele (engl. *label*) ili u obliku heksadecimalnog zapisa, podrazumijeva pomak u odnosu na to ishodište. Ishodište je točno ta adresa u (RAM) memoriji zato što BIOS učitava *bootloader* točno na to mjesto. Kada se ne bi koristila ova direktiva, bilo bi potrebno na svaku adresu u kôdu pribrojiti broj 0x7c00, što bi ga nepotrebno zakompliciralo, odnosno smanjilo bi čitkost i povećalo vjerojatnost pogreške.

Instrukcija equ labeli s lijeve strane dodjeljuje vrijednost s desne. Koristi se za konstante.

Instrukcija mov, u Intelovoj sintaksi, dodjeljuje vrijednosti s lijeve strane onu s desne. Operanti mogu biti registri, mjesta u memoriji i konstante.

Funkcije se pozivaju instrukcijom call koja uz to pohranjuje iduću instrukciju na *stack*.

Instrukcija db (engl. *define byte*) koristi se za deklaraciju statičnih dijelova koda. Ono što joj prethodi je (opcionalna) labela kako bi se poslije ta „varijabla” mogla koristiti. To je u osnovi pokazivač na mjesto u memoriji na kojem se nalazi danā vrijednost [4].

Prvo što se na ekranu ispisuje, ne računajući ono što ispisuje emulator, poruka je da se kôd izvršava u 16-bitnom načinu rada.

Heksadecimalne vrijednosti 0xD i 0xA označavaju povratak kursora na prvi stupac (engl. CR – *Carriage Return*) i prelazak u novi red (engl. LF - *Line Feed*) uobičajeno korištene u kombinaciji i poznate kao CR-LF.

```
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
Started in 16-bit Real Mode...
Loading kernel into memory...
_
```

Slika 2: Prikaz prvog ispisa na ekranu

Ta je poruka ispisana preko `print_string` funkcije koja koristi BIOS-ovu "*Scrolling tele-type routine*" funkciju, koja služi za ispis jednog znaka i pomak kursora za jedno mjesto.

Ova rutina da se ah registar vrijednost 0x0e (engl. brojem 0x10. ekran na mjestu znak čija se American for Information vrijednost registru.

Određeno je da odnosno adresa znaka, nalazi u pozivatelj ove znati da bi je mogao koristiti.

```

load_kernel ispis stringa pomocu BIOS-a u Real Mode-u
mov bx, MSG_LOAD_KERNEL
path_printing
; push ax, ipoh(15) sektora (registra B00) funkcija mijenjati na stack
; push bx(0x0000:0x1000) sa dl-a(0)(boot disk, kernel kod)
mov cx, KERNEL_OFFSET, ah=0x0e --> scrolling teletype BIOS routine
mov dh, 15
mov dl, [BOOT_DRIVE]
call is_byte [bx], 0 ; nula oznacava kraj stringa
; je return
ret
mov al, byte [bx] ; ukoliko znak nije nula, ispisuje se na ekran
KERNEL_OFFSET equ 0x1000 ; mjesto gdje ce se ucitati jezgra; konstanta
MSG_LOAD_KERNEL db "Loading kernel into memory...", 0xD, 0xA, 0
; Ispis 4: Učitavanje sektora s diska u radnu memoriju

return:
pop bx ; povratak koristenih registara u prvobitno stanje
pop ax
ret ; izlazak iz funkcije

Ispis 3: Funkcija za ispis stringa pomocu BIOS-a

```

poziva se tako postavi na i aktivira prekid interrupt) s Tada se na kursora ispiše ASCII (engl. Standard Code Interchange) nalazi u al

se string, prvog njegovog registru bx i to bi funkcije trebao

2.4 Učitavanje kôda s diska u radnu memoriju

Iduće što se događa je poziv labela load_kernel.

Kako bi korisnik znao što se događa, poziva se `print_string` funkcija kao i ranije. Zatim se priprema teren za funkciju `disk_load`.

Funkcija je napisana tako da prima tri parametra.

U `dl` registar stavlja se vrijednost koja označava disk s kojeg funkcija čita sektore. Prilikom pokretanja računala BIOS tu vrijednost spremi u taj isti `dl` registar, ali nitko ne garantira da se njegova vrijednost u nekom trenutku neće promijeniti. Zato je ona odmah spremljena u `BOOT_DRIVE` globalnu varijablu. Točnije, spremljena je na lokaciju koju labela označava. Stoga je potrebno staviti uglate zagrade (`[]`) oko nje, kako bi se dohvatila vrijednost koja se nalazi na toj adresi, a ne samā adresa.

U `dh` registar stavlja se broj sektora koji će se učitati s diska, u ovom slučaju 15. Bolje ih je staviti malo više da se povećavanjem kôda u budućnosti ne bi izašlo van okvira, što bi prouzročilo greške, a uzrok možda ne bi bio očit.

I konačno u `bx` registar postavlja se pomak u odnosu na segmentni registar `es`. Segmentni registar i pomak u kombinaciji određuju adresu u radnoj memoriji na koju će se kôd učitati.

Adresiranje u 16-bitnom načinu rada vrši se tako da se jedan od segmentnih registara, u ovom slučaju *extra segment* `es`, pomnoži sa 4, odnosno pomakne jednu heksadecimalnu znamenku ulijevo, te se tom broju pribroji pomak, ovdje u registru `bx`.

U **ispisu 5** prikazana je `disk_load` funkcija.

Ovdje se ova *read-* Za njeno potrebno je u staviti pozvati prekid 0x13. Svi registre koje nalaže.

Bitno je kod odabira koristiti CHS adresiranja. način zastario, za koji je ga koristi. danas koristi LBA (engl. *Addressing*) LBA adrese jednostavno jednog oblika u

U slučaju pogreške BIOS (engl. *Carry*

```

disk_load: ; s diska DL učitaj DH sektora na adresu ES:BX
push ax
push cx
push dx ; broj sektora koji ce se pokusati učitati
mov ah, 0x02 ; BIOS read-sector funkcija
mov al, dh ; učitavanje DH sektora
mov ch, 0x00 ; odabir cilindra 0
mov dh, 0x00 ; odabir glave 0
mov cl, 0x02 ; odabir DRUGOG sektora, sektora 2

int 0x13 ; BIOS prekid za učitavanje

jc disk_error ; ako je CF=1 dogodila se greska
; ako zatrazeni i stvarni broj ucitanih sektora nisu isti dogodila se greska
pop dx
cmp dh, al
jne disk_error

pop cx
pop ax
ret

; u 'ah' je error kod, a u 'dl' disk na kojem se ona dogodila
disk_error:
mov bx, DISK_ERROR_MSG
call print_string
jmp $

DISK_ERROR_MSG db "Disk read error!", 0xD, 0xA, 0

```

Ispis 5: Funkcija za učitavanje s diska u memoriju

koristi BIOS-*sector* funkcija. korištenje ah registar vrijednost 2 i pod brojem parametri idu u procedura

naglasiti da se sektorā mora način Iako je taj kao i tvrdi disk razvijen, BIOS Općenito se novija linearna *Linear Block* shema. CHS i mogu se pretvarati iz drugi.

generalne postavlja CF (*Flag*)

posebnog *Flags* registra. CF u slobodnom prijevodu znači „zastavica prijenosa” i inače se koristi kod računskih operacija kad rezultat ne može stati u prostor koji mu je namijenjen, odnosno nedostaje mu još jedna znamenka.

Ako CF nije postavljen potrebno je provjeriti odgovara li broj sektora koji se trebao učitati stvarnom broju učitanih sektora. U tu se svrhu broj sektora koji dobiven kao parametar pohranjuje na *stack* i kasnije uspoređuje sa stvarnim brojem, kojeg BIOS po završetku čitanja stavlja u *al* registar. U slučaju da nisu isti dogodila se greška.

2.5 Prebacivanje u *Protected Mode*

Nakon učitavanja jezgre u memoriju slijedi prebacivanje procesora u 32-bitni način rada. To se obavlja preko *switch_to_pm* funkcije. Ali prije poziva te funkcije potrebno je napraviti GDT (engl. *Global Descriptor Table*). Tablica izgleda dosta komplicirano, ali treba joj pristupiti korak po korak.

U 16-bitnom načinu rada segmentiranje memorije funkcionira na način da se vrijednost korištenog segmentnog registra množi sa 16 (jedna heksadecimalna znamenka ulijevo) i tom broju pribroji se pomak. Raspon adresa koja se dobiva na taj način je

$$(0xFFFF * 16) + 0xFFFF = 0xFFFF0 + 0xFFFF = 0x10FFEF$$

odnosno nešto više od 1 MB.

U 32-bitnom se načinu segmentni registar više ne množi sa 16 nego označava indeks određenog segmentnog deskriptora (engl. *Segment Descriptor – SD*) u GDT. SD opisuje baznu adresu sa 32 bita, granicu segmenta (engl. *Segment Limit*) (koja određuje njegovu veličinu) sa 20 bitova, i razne zastavice koje definiraju lepezu mogućnosti.

$$\text{Sada je moguće adresirati } 0xFFFFFFFF + 0xFFFF = 0x1000FFFE$$

odnosno nešto više od 4 GB memorije.

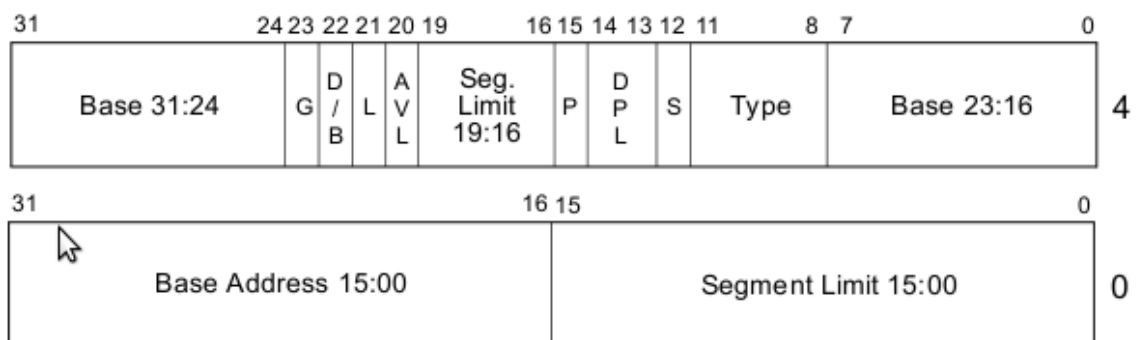
Tablica se sastoji od 8-bajtnih deskriptora. Mora započeti obaveznim *null*-deskriptorom (engl. *null*). To je niz od 64 nule, tj. svih 8 bajtova koji opisuju deskriptor.

Najjednostavnija konfiguracija GDT ima samo dva segmenta i ime joj je osnovni ravni model (engl. *basic flat model*). Naziva se „ravnim” jer podrazumijeva baznu adresu 0x0 [5].

To u principu znači da granica segmenta određuje adresu. Jedan se segment definiira za kôd, a drugi za podatke.

Treba naglasiti da se segmenti preklapaju i ne postoji prava zaštita memorije, ali to je moguće, a i jednostavnije, napraviti naknadno u kôdu jezgre koji se može napisati u višem programskom jeziku. Za potrebe ovog rada u programskom jeziku C.

Na **slici 3** prikazana je shema GDT konstruirane po osnovnom ravnom modelu.



- L — 64-bit code segment (IA-32e mode only)
- AVL — Available for use by system software
- BASE — Segment base address
- D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- DPL — Descriptor privilege level
- G — Granularity
- LIMIT — Segment Limit
- P — Segment present
- S — Descriptor type (0 = system; 1 = code or data)
- TYPE — Segment type

Slika 3: GDT - osnovni ravni model

Prvo što se može primijetiti je da su bitovi bazne adrese i granice segmenta razbacani, što smanjuje preglednost.

Implementacija tablice detaljnije je opisana u kôdu u **ispisu 6**.

```

gdt_start:

gdt_null: ; obavezni null-deskriptor
dd 0x0 ; 2(2*16) = 64 bita[8 bajtova] - duzina svakog GDT deskriptora
dd 0x0

gdt_code: ; kod-segment deskriptor
; baza=0x0, limit=0xffff
dw 0xffff ; limit (bitovi 0-15)
dw 0x0 ; baza (bitovi 0-15)
db 0x0 ; baza (bitovi 16-23)

; prisutan: 1 = prisutan u memoriji, koristi se za virtualnu memoriju
; prava: 00 = najveca prava
; tip: 1 = kod ili data segment
; 1st flags: 1(prisutan) 00(prava) 1(tip deskriptora)
; kod: 1 = kod segment
; conforming: 0 = zasticen, kod u segmentu s manjim pravima ne moze zvati
; kod u ovom segmentu; zastita memorije
; readable: 1 = nije samo izvrsiv, dopusteno citanje konstanti iz koda
; accessed: 0 = CPU ga postavlja kad pristupi segmentu, za debug i sl.
; type-flags: 1(kod) 0(conforming) 1(readable) 0(accessed)
db 0b10011010 ; 1st flags(1001), type-flags(1010) ; ili db 10011010b

; granularnost: 1 = limit se mnozi sa 4096(16*16*16),
; odnosno pomice 3 hex znamenke ulijevo, max limit je 4GB
; 32-bit zadano: 1 = zadana rijec postavlja se na 32-bit
; 64-bit seg: 0 = ne koristi se na 32-bitnom procesoru
; AVL: 0 = nekoristeno (za osobnu upotrebu)
; 2nd flags: 1(granularnost) 1(32-bit zadano) 0(64-bit seg) 0(AVL)
db 0b11001111 ; 2nd flags(1100), limit(bitovi 16-19)

db 0x0 ; baza (bitovi 24-31)

gdt_data: ; data-segment deskriptor
; sve kao kod kod-segmenta, osim type-flagova
dw 0xffff
dw 0x0
db 0x0
; expand down: 0 = seg. se moze prosiriti prema dolje
; writable: 1 = uz citanje, dopusteno i pisanje po data segmentu
; type-flags: 0(data) 0(expand down) 1(writable) 0(accessed)
db 0b10010010
db 0b11001111
db 0x0

gdt_end: ; da assembler moze izracunati velicinu GDT-a za deskriptor (ispod)

gdt_descriptor:
dw gdt_end - gdt_start - 1 ; velicina, uvijek 1 manje od stvarne
dd gdt_start ; pocetna adresa GDT-a

```

Sada slijedi
u PM,
ispisu 7.

```
[bits 16]

switch_to_pm:

cli ; clear interrupts; moraju se ugasiti dok se ne podesi IVT za PM

lgdt [gdt_descriptor] ; učitava se GDT koja definira PM segmente (kod i data)

; prvi bit cr0 kontrolnog registra postavlja se na 1, ne mijenjajući ostale
; ne može se mijenjati direktno nego se mora koristiti GPR
mov eax, cr0
or eax, 0x1
mov cr0, eax

; far jump (na novi segment) na 32-bitni kod
; prisiljava CPU da dovrši instrukcije započete u 16-bitnom modu
jmp CODE_SEG:init_pm

[bits 32]
; inicijalizacija registara i stacka za PM način rada
init_pm:
mov ax, DATA_SEG
mov ds, ax
mov ss, ax
mov es, ax
mov fs, ax
mov gs, ax

mov ebp, 0x90000 ; (1152*512)
mov esp, ebp

call BEGIN_PM
```

prebacivanje
prikazano u

Ispis 7: Prebacivanje u PM

Instrukcija `cli` korištenje `clear` Prekidi se dok se ne IVT (engl. *Vector Table*) prikladna za instrukcije.

```
[bits 32] ; 32-bitne instrukcije
; nakon inicijalizacije PM-a, i promjene na njega, kod se nastavlja ovdje
BEGIN_PM:
mov ebx, MSG_PROT_MODE ; 32-bitni reg.
call print_string_pm ; a print funkcija

call KERNEL_OFFSET ; nakon svega, skok na mjesto gdje je učitana jezgra
MSG_PROT_MODE db "Successfully landed in 32-bit Protected Mode...", 0

Ispis 8: Bootloader predaje kontrolu jezgri
```

onemogućava prekida (engl. *interrupts*). moraju ugasiti podesi nova *Interrupt* tablica 32-bitne

Instrukcija `lgdt` učitava GDT za 32-bitni *Protected Mode*.

U segmentne se registre ne mogu direktno učitati adrese, broj prvo treba ubaciti u registar opće namjene (engl. *GPR – General Purpose Register*) i onda ga iz njega kopirati u segmentni. Na taj se način mijenja najmanje značajan bit (engl. *Least Significant Bit – LSB*) kontrolnog registra `cr0`, koji je potrebno postaviti na jedinicu. To označava da je procesor sada u PM-u (0 označava RM).

Zatim je potrebno izvršiti daleki skok (engl. *far jump*) kako bi procesor izvršio *pipeline* 16-bitnih instrukcija do kraja. Nije bitna udaljenost u memoriji nego da se skoči na način da se specificira bazna adresa i pomak.

Naposlijetku se inicijaliziraju registri opće namjene, postavlja *stack* i poziva labela `BEGIN_PM`. Sada se kôd zapravo već izvršava u PM načinu.

Ispis 8 prikazuje labelu `BEGIN_PM`.

Direktiva `[bits 32]` označava da se instrukcije nakon nje interpretiraju kao 32-bitne.

Zatim se poziva `print_string_pm` funkcija koja ispisuje poruku kako bi korisnik bio obaviješten da je procesor uspješno ušao u PM. Ta se funkcija razlikuje od funkcije `print_string` jer je napisana za izvršavanje u 32-bitnom PM načinu.


```

[bits 32]
; konstante
VIDEO_MEMORY equ 0xb8000 ; pocetak tekstualne video memorije
WHITE_ON_BLACK equ 0x0f ; atributi za ispis

print_string_pm: ; null-terminated, na adresi EBX
    push eax;
    push ebx;
    push edx;
    mov edx, VIDEO_MEMORY ; EDX se postavlja na pocetak video memorije
    mov ah, WHITE_ON_BLACK ; atributi u AH

print_string_pm_loop:
    mov al, [ebx] ; ASCII znak za ispis, na adresi EBX

    cmp al, 0 ; ako se doslo do kraja stringa izlazi se iz funkcije
    je print_string_pm_done

    mov [edx], ax ; u video memoriju se zapisuje ASCII vrijednost znaka uz pripadajuce vizualne atribute

    add ebx, 1 ; ide se na iduci znak stringa
    add edx, 2 ; ide se na iduci znak-atribut par video memorije

    jmp print_string_pm_loop

print_string_pm_done:
    pop edx;
    pop ebx;
    pop eax;
    ret

```

Ispis 9: 32-bitna print_string funkcija

Naposlijetku se skače na mjesto u memoriji gdje je ranije učitana jezgra. Jezgra samo pokreće pogonski program za bazičnu manipulaciju tekstem na ekranu.

```

// pocetak video memorije, VGA text mode
#define VIDEO_ADDRESS 0xb8000

#define MAX_ROWS 25
#define MAX_COLS 80

//atributni bajt za zadanu colour scheme
#define WHITE_ON_BLACK 0x0f

//screen device I/O ports
#define REG_SCREEN_CTRL 0x3D4
#define REG_SCREEN_DATA 0x3D5

int get_screen_offset(int row, int col);
int get_cursor();
void set_cursor(int offset);
int handle_scrolling(int cursor_offset);
void print_char(char character, int row, int col, char attribute_byte);
void print_at(char* message, int row, int col);
void print(char* message);
void clear_screen();

Ispis 10: Header file funkcija za manipulaciju ekranom

```

2.6 Pogonski program za tekstualnu manipulaciju ekranom

U **ispisu 10** prikazan je *header file* u kojem se nalaze globalne konstante i deklaracije funkcija za manipulaciju ekranom koje ih koriste.

```
#include "screen.h"
#include "../kernel/low_level.h"
#include "../kernel/util.h"

int get_screen_offset(int row, int col) {
    return (2 * (row * MAX_COLS + col));
}
Ispis 11: Linearni pomak (od 0xb000) za pristup ćeliji
```

U nastavku će se pokazati implementacija svake od ovih funkcija.

`#include` direktive nalaze se na vrhu dokumenta i podrazumijevat će se da uključene dokumente mogu koristiti i sve ostale funkcije koje se bave ekranom.

Prva, i jedna od jednostavnijih, je funkcija `get_screen_offset()`.

Pomak se računa tako da se broj redaka pomnoži s najvećim mogućim stupcem (na taj se način dođe na početak odabranog reda) i tom se broju pribroji broj stupaca.

Sada je izračunat logički pomak, ali budući da svaka ćelija zauzima dva bajta (jedan za ASCII vrijednost znaka koji se ispisuje, a drugi za attribute ćelije – kao što su boja znaka, boja pozadine itd.) potrebno je još taj broj pomnožiti sa dva. **Ispis 11** prikazuje njenu implementaciju.

Iduća na redu je `get_cursor()`. Da potrebno je prvo pomoćne funkcije `port_byte_out()` i `port_byte_in()`. Ove dvije tzv. *inline* unutarlinijski programski jezik. Općenito pojam

```

//C Ova funkcija je za postavljanje bajta na odabrani port
//d'a (data) u odabranom portu mora ići u edx
//h'al; (port) čita bajt iz odabranog portu EDI (1 bajt) u AL
//ca (default) > na odabranom portu postavlja vrijednost atrAg.-a u var. RESULT
//void port_byte_out(unsigned short port, unsigned char data) {
unsigned char op; op = data; outb(op, port);
unsigned char result;
Ispis 12: Funkcija za postavljanje vrijednosti na port
return result;
}

```

funkcija `int` bi se ona shvatila promotriti dvije jezgre: `port_byte_in()`. funkcije koriste *assembly* (hrv. asembler) koji C dopušta. *inline* označava

da se čitava funkcija ubacuje na mjesto na kojem je pozvana. Prednost toga je što se funkcija ubiti uopće ne poziva u klasičnom smislu, nego se program samo nastavlja izvršavati na njoj. Zato je *inline assembly* brz i često korišten u sistemskom programiranju [6].

Dosada je bila korištena Intelova asemblerska sintaksa. Mogla se iskoristiti i sada, ali za promjenu je odabrana AT&T sintaksa. Najupadljivije razlike su što kod nje treba koristiti znak za postotak (%) za pristup registru i redoslijed operanata je obrnut.

Funkcija `port_byte_out()` prikazana u **ispisu 12** postavlja odabranu vrijednost na odabrani port.

Funkcija `port_byte_in()` prikazana u **ispisu 13** služi za čitanje vrijednosti s odabranog porta.

Sada se
promotriti

```
// preko kontrolnog registra odabire se interni registar
// neki od njih su:
// reg. 14: visi bajt pomaka kursora
// reg. 15: nizi bajt pomaka kursora
// kad je int. reg. odabran moze se pročitati ili zapisati bajt u podatkovni reg.
int get_cursor() {
    // na port 0x3D4(u VGA kontrolni reg.) zapise se broj 14 (odabere se int. reg. 14)
    // odabravši int.reg. pod. reg.(port 0x3D5) se auto. azurira na njegovu vrijednost
    // data reg. služi kao povratna vrijednost, slicno kao eax u assembleru
    port_byte_out(REG_SCREEN_CTRL, 14);

    // sa porta 0x3D5 pročitaj bajt, i postavi ga kao visi bajt pomaka
    int offset = (port_byte_in(REG_SCREEN_DATA) << 8);

    // na port 0x3D4(u kontrolni reg.) zapisi broj 15 (odaberi interni reg. 15)
    port_byte_out(REG_SCREEN_CTRL, 15);

    // sa porta 0x3D5 pročitaj bajt, i postavi ga kao nizi bajt pomaka
    offset += port_byte_in(REG_SCREEN_DATA);

    // pomak koji VGA hardver javlja odnosi se na broj znakova
    // *2 zbog atributnog bajta koji ga prati
    return (2 * offset);
}
```

Ispis 14: Funkcija koja računa lokaciju kursora

može
funkcija

get_cursor() (ispis 14).

VGA kontrolnom registru pristupa se preko porta 0x3D4. Na taj se port zapiše vrijednost 14 i time se odabire interni registar 14 u kojem se nalazi viši bajt pomaka kursora.

Posljedično se automatski vrijednost odabranog internog registra (taj viši bajt) kopira u podatkovni registar koji se nalazi na portu 0x3D5.

Tada se ta vrijednost čita iz podatkovnog registra i sprema u varijablu offset. Budući da je to viši bajt pomaka potrebno je pomaknuti ga osam mjesta ulijevo.

Na isti
način
donji

```
// [0] ili [par] je ASCII znak, [nepar] je atribut za [nepar-1]
void set_cursor(int offset) {
    // pretvorba iz pomaka za celiju (znak, atribut) u pomak za znak (VGA nacin)
    offset /= 2;

    // preko kontrolnog registra odabere se interni registar 14
    // u kojem se nalazi visi bajt pomaka kursora
    port_byte_out(REG_SCREEN_CTRL, 14);

    // posto je odabran interni reg. 14, mijenjajuci podatkovni reg. mijenja se njegova vrijednost
    // azurira se vrijednost viseg bajta pomaka kursora
    // pomak je int = 16bit = 2B
    // unsigned char = 1B, kao i registri
    port_byte_out(REG_SCREEN_DATA, (unsigned char)(offset >> 8));

    // preko kontrolnog registra odabere se int. reg. 15
    // u kojem se nalazi nizi bajt pomaka kursora
    port_byte_out(REG_SCREEN_CTRL, 15);

    // azurira se vrijednost nizeg bajta pomaka kursora
    port_byte_out(REG_SCREEN_DATA, (unsigned char) offset);
}
```

se
čita
bajt

Ispis 15: Funkcija za postavljanje kursora

pomaka. Pošto je **donji** on se sada jednostavno pribroji vrijednosti gornjeg.

Na kraju je potrebno ukupan pomak pomnožiti s dva kako bi se uzeo u obzir i atributni bajt koji prati svaki ASCII znak.

Iduća na redu je funkcija `set_cursor()` koja postavlja kursor na zadano mjesto.

Prvo se realni pomak pretvara u znakovni pomak, koji VGA računa.

Zatim se, kao i kod `get_cursor()` funkcije, preko kontrolnog registra odabire interni registar broj 14.

Sada se u podatkovni registar ubacuje gornji bajt pomaka. Pošto je odabran registar 14, njegova se vrijednost automatski usklađuje s vrijednošću podatkovnog.

Isto se učini i s donjim bajtom s registrom 15.

Iduća će se promatrati funkcija `print_char()` koja ispisuje odabrani znak na birano mjesto na ekranu. Prikazana je u **ispisu 16**.

Funkcija
četiri
znak,
stupac i
(koji nije
Ako su
stupac
(pozitivni)
funkcija za
pozicije za
slučaju da
se trenutna
kursora.
Ako je
znak za
redak se
za jedan, a
stavlja na
suprotnom
ispisuje, u
popratnim
U slučaju
pomak
ekrana

```
// ispis znaka na ekranu na poziciji (redak, stupac) ili na mjestu kursora
void print_char(char character, int row, int col, char attribute_byte) {
    // byte(char) pokazivac na pocetak video memorije (0xb8000)
    unsigned char* vidmem = (unsigned char*) VIDEO_ADDRESS;

    // zadani atribut
    if(!attribute_byte)
        attribute_byte = WHITE_ON_BLACK;

    int offset;
    if(row >= 0 && col >= 0)
        offset = get_screen_offset(row, col);
    else
        offset = get_cursor();

    // u slucaju nove linije, postaviti pomak na prvi(nulti) stupac iduceg reda
    if(character == '\n') {
        // int rows = (offset - 2 * col) / (2 * MAX_COLS);
        // (-2 * col) u brojniku suvisno, ne mijenja rezultat (2*col < 2*MAX_COLS)
        // samo bi vratilo pomak na pocetak retka
        int rows = offset / (2 * MAX_COLS);
        offset = get_screen_offset(++rows, 0);
    }
    else {
        *(vidmem + offset++) = character;
        *(vidmem + offset++) = attribute_byte;

        // offset ide na iduci znak-atribut par
        // offset += 2;
    }

    // za slucaj da se doslo do dna ekrana
    offset = handle_scrolling(offset);

    set_cursor(offset);
}
```

prima
parametra:
redak,
atribut
obavezan).
redak i
validni
poziva se
izračun
ispis, a u
nisu uzima
pozicija
primljen
novu liniju
povećava
stupac
nulu. U
se znak
skladu s
atributom.
da je
prešao dno
poziva se

funkcija `handle_scrolling()` koja će se promotriti iduća.

Zatim se kursor pomiče na iduću ćeliju.

Funkcija `handle_scrolling()` testira je li pomak, odnosno kursor, prešao dno ekrana. U slučaju da jest, prebacuje sve linije za jedno mjesto gore i čisti zadnju. Implementirana je tako da se najgornja linija zauvijek gubi i tu ima prostora za poboljšanje. Prikazana je u **ispisu 17**.

```

void memory_copy(char* source, char* dest, int num_of_bytes) {
// ukoliko je potrebno, skrolanje ekrana i pomak kursora
// zasad brise najstariju liniju
for(int i = 0; i < num_of_bytes; ++i)
    *(dest + i) = *(source + i);
}

int handle_scrolling(int cursor_offset) {
// ako kursor nije "van ekrana" sve je u redu
if(cursor_offset < (2 * MAX_COLS * MAX_ROWS))
    return cursor_offset;

// svaki red ide za jedan gore (-1)
int i = 1;
for(; i < MAX_ROWS; ++i)
    memory_copy((char*)(VIDEO_ADDRESS + get_screen_offset(i, 0)),
                (char*)(VIDEO_ADDRESS + get_screen_offset(i - 1, 0)),
                2 * MAX_COLS);

// praznjenje zadnje linije
char* last_line = (char*)(VIDEO_ADDRESS + get_screen_offset(MAX_ROWS - 1, 0));
for(i = 0; i < 2 * MAX_COLS; ++i) {
    *(last_line + i++) = ' '; // ASCII znak
    *(last_line + i) = WHITE_ON_BLACK; // bijelo na crno, 0x0f, atribut
}

// da pomak bude na prvoj celiji zadnjeg vidljivog retka
cursor_offset -= (2 * MAX_COLS);

return cursor_offset;
}

```

Ispis 17: Funkcija za skrolanje ekrana

Funkcija prvo provjerava je li kursor uopće van ekrana i u slučaju da nije jednostavno vraća pomak. Ako jest, svaka se linija kopira u onu poviše nje, čime se prva gubi. Zadnja se linija prazni i kursor postavlja na njenu prvu ćeliju.

Za kopiranje linija koristi se pomoćna funkcija jezgre `memory_copy()` prikazana u **ispisu 18**.

Funkcija preslikava odabrani broj bajtova iz izvora u odredište.

Iduća pogonskog ekran zove služi za ispis počevši od mjesta. u ispisu 19.

U slučaju da vrijednosti negativne samo iterirati dok se ne kraja niza. na trenutnu poziciju Ako su pozitivne

ručno pomicati kursor.

Funkcija je poprilično jednostavna jer zapravo funkcija `print_char()` u pozadini obavlja glavninu posla.

Zbog jednostavnosti korištenja, napisana je i *wrapper* funkcija `print()` koja ispisuje poruku na mjestu kursora i u zadanom formatu (**ispis 20**).

```
// ispis stringa na ekran na trenutnu poziciju kursora
// postavljanje kursora preko print_char funkcije
void print(char* message) {
void print_at(char* message, int row, int col) {
}
int i = 0;
if(row >= 0 && col >= 0) {
    set_cursor(get_screen_offset(row, col));

    while(*(message + i) != 0) {
        print_char(*(message + i++), row, col, WHITE_ON_BLACK);

        if(++col == 80) {
            col = 0;
            ++row;
        }
    }
} // ako se koristi trenutna pozicija kursora print_char obavlja iteraciju
else
    while(*(message + i) != 0)
        print_char(*(message + i++), row, col, WHITE_ON_BLACK);
}

Ispis 20: Jednostavna funkcija za ispis
Ispis 19: Funkcija za ispis niza znakova
```

funkcija programa za se `print_at()` i niza znakova odabranog Prikazana je

su retka i stupca dovoljno je znak po znak dođe do Uvijek se ide (iduću) kursora. vrijednosti potrebno je

Posljednja funkcija je `clear_screen()` koja, pobriše cijeli ekran

```
void clear_screen() {  
    int row = 0;  
    int col = 0;  
  
    for(row = 0; row < MAX_ROWS; ++row)  
        for(col = 0; col < MAX_COLS; ++col)  
            print_char(' ', row, col, WHITE_ON_BLACK);  
  
    set_cursor(0); // nema potrebe zvati get_offset  
}
```

Ispis 21: Funkcija za brisanje ekrana

koja se bavi ekranom kao što joj i ime kaže, (ispis 21).

Funkcija jednostavno iterira po cijeloj vidljivoj tekstualnoj video memoriji i „ispisuje” prazan znak na svakom mjestu.

Na kraju vraća kursor na prvu ćeliju.

2.7 Jezgra

Jezgra koristi pogonski program za ispis teksta na ekran. Na **slici 4** prikazan je primjer ispisa poruke iz jezgre.

```
#include "../drivers/screen.h"
void main() {
char* m = "Welcome to TheSys OS!\n";
Successfully landed in 32-bit Protected Mode...
15 0
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00
Booting from Hard Disk...
Boot failed: could not read the boot disk
Booting from Floppy...
Started in 16-bit Real Mode...
Loading kernel into memory...
Welcome to TheSys OS!
Feel free to build upon it . . . . .
-
```

Slika 4: Primjer ispisa na ekran iz jezgre

Fluorescentno označeni dio ispisan je iz jezgre. Kôd jezgre u ovom slučaju izgledao bi ovako (ispis 22):

Uz ovu datoteku, jezgra sadržava tri pomoćne funkcije `port_byte_out()` (ispis 12), `port_byte_in()` (ispis 13) i `memory_copy()` (ispis 18) promotrene ranije.

2.8 Cross-compiler

Sve ove funkcije logički izgledaju u redu, ali nisu funkcionirale kako treba dok nije napravljen tzv. *cross-compiler*.

Cross-compiler je potreban jer bi obični kompajler kôd kompajlirao za platformu na kojoj se razvija, u ovom slučaju Ubuntu 20.04. Ali budući da se razvija samostalni operacijski sustav potrebno je to uzeti u obzir.

Kôd je kompajliran tako da bude lišen biblioteka ili ikakvih mogućnosti koje pruža OS domaćin na kojem se sustav izrađuje.

Specificirano je jedino da bi se trebao moći izvršiti na računalima koja podržavaju šestu generaciju Intelove x86 arhitekture, popularno znanu kao *i686*. Kad OS uđe u dalji stadij razvoja moguće je, i poželjno, napraviti *cross-compiler* za točno taj OS.

U literaturi su detaljno navedeni koraci koje je potrebno izvršiti kako bi se dobio *cross-compiler* [7].

Preporučljivo je skinuti najnoviju verziju kompajlera i ovisnosti (engl. *dependencies*) jer su u pravilu najpouzdaniji i najbrži. S napomenom da treba paziti jesu li međusobno kompatibilni.

Ispod se nalaze naredbe koje su upotrijebljene za izradu konkretnog *cross-compiler*a koji je korišten za ovaj rad.

Treba naglasiti da bi osoba neiskusna s Linuxom mogla imati poteškoća s instalacijom kompajlera.

Instalacija ovisnosti

Najbolje je prvo ažurirati menadžer paketa (engl. *packet manager*), u ovom slučaju APT (*Advanced Packaging Tool*), a zatim pomoću njega instalirati sve potrebne ovisnosti.

```
sudo apt update
```

```
sudo apt install build-essential
```

```
sudo apt install manpages-dev
```

```
sudo apt install bison
sudo apt install flex
sudo apt install libgmp3-dev
sudo apt install libmpc-dev
sudo apt install libmpfr-dev
sudo apt install texinfo
```

Skidanje izvornog kôda (engl. *downloading source code*)

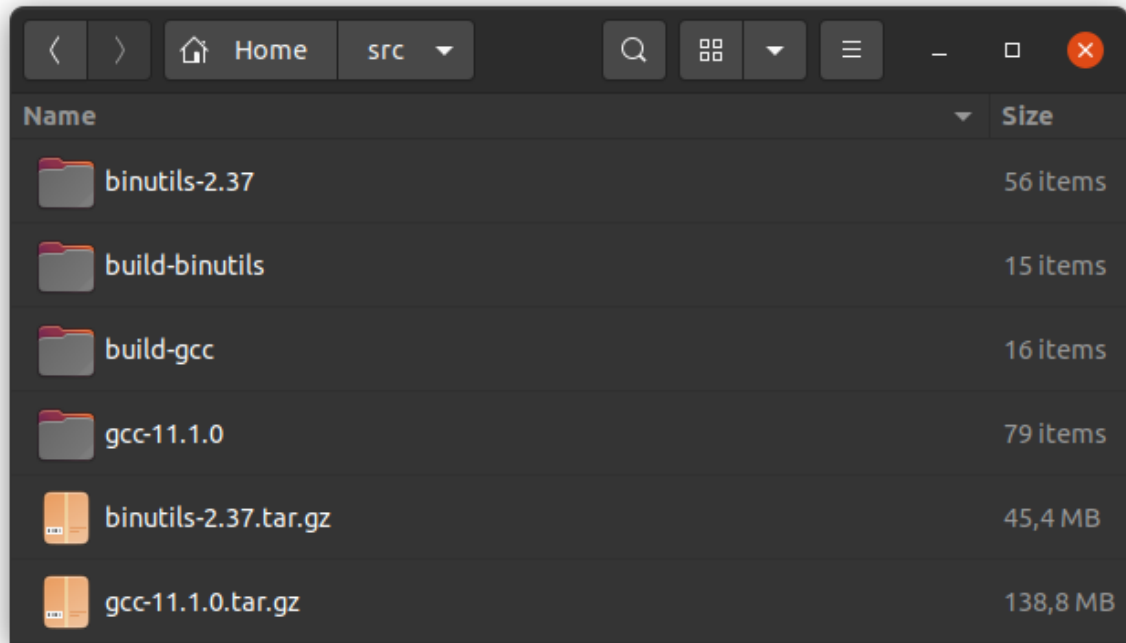
Link za skidanje *binutils*a nalazi se u literaturi pod brojem [8].

Link za skidanje gcc-a nalazi se u literaturi pod brojem [9].

Kôd je skinut u „tar.gz” formatu i raspakiran u `/$HOME/src` direktorij. Direktorij bi sada trebao izgledati ovako, izuzevši nenumerirane direktorije koji se pojave **nakon** instalacije. Komprimirane datoteke mogu se izbrisati.

Za izradu ovog rada koristio se gcc 11.1.0 i binutils 2.37 .

Sljedeći direktoriji napravljeni su u \$HOME(~,/home/ime_korisnika) direktoriju:



Slika 5: *Source* direktorij za gcc i binutils

Korištene su naredbe:

```
mkdir opt  
mkdir opt/cross  
cd opt/cross/
```

Naredba `export` pretvara običnu varijablu u varijablu okruženja/okoline (engl. *environment variable*) [10].

Budući *cross*-kompajler nalazit će se u direktoriju koji je postavljen kao `PREFIX`. Prefiks se stavlja u `PATH` varijablu (varijablu putanje) za trenutnu sesiju ljuske (engl. *shell*) kako bi kompajler detektirao *binutils*.

Kao TARGET odabire se arhitektura za koju je OS namijenjen; odnosno za koju *cross-compiler* kompajlira kôd.

```
export PREFIX="$HOME/opt/cross"  
export TARGET=i686-elf  
export PATH="$PREFIX/bin:$PATH"
```

U src direktoriju pravi se build-binutils direktorij vidljiv na **slici 5** ranije.

```
cd $HOME/src/  
mkdir build-binutils
```

Builda se izvan izvornog direktorija što se smatra dobrom praksom (za gcc je čak obavezno).

```
cd build-binutils/  
../binutils-2.37/configure --target=$TARGET --prefix="$PREFIX" --with-sysroot --disable-nls --disable-werror  
make  
make install
```

Opcija `--disable-werror` onemogućava da se upozorenja (engl. *warnings*) tretiraju kao greške (engl. *errors*).

Vrši se provjera je li `$PREFIX/bin` direktorij u putanji (PATH).

```
which -- $TARGET-gcc || echo $TARGET-as is not in the PATH
```

Ako jest ide se na idući korak, instalaciju gcc-a, koristeći sljedeće naredbe:

```
cd $HOME/src  
mkdir build-gcc  
cd build-gcc/  
../gcc-11.1.0/configure --target=$TARGET --prefix="$PREFIX" --disable-nls --enable-languages=c,c++ --without-headers
```

Opcija `--without-headers` označava da gcc ne smije koristiti **nijednu** C biblioteku.

```
make all-gcc
make all-target-libgcc
make install-gcc
make install-target-libgcc
```

Time je *cross-compiler* završen.

Sljedeću je naredbu potrebno zapisati u `~/.bashrc` datoteku kako bi trenutni korisnik imao mogućnost korištenja naredbi iz `~/opt/cross/bin` direktorija bez da mu navodi putanju [10].

```
export PATH="$HOME/opt/cross/bin:$PATH"
```

To je moguće napraviti preko Vim-a ili bilo kojeg drugog uređivača teksta (engl. *text editor*).

Npr. `vim ~/.bashrc`

2.9 Makefile

Pisanje *makefilea* nije neophodno, ali je uvelike olakšalo razvoj ovog sustava. Iako savladavanje njegove izrade zahtijeva nešto vremena, to vrijeme je zanemarivo u usporedbi s onim uštedenim.

Ispod je prikazana zadnja verzija *makefilea* korištenog u ovom radu.

```

# auto. generate list of sources using wildcards
C_SOURCES = $(wildcard kernel/*.c drivers/*.c)
HEADERS = $(wildcard kernel/*.h drivers/*.h)

# list of obj. files TO build
OBJ = ${C_SOURCES:.c=.o}

# default make target
all: os-image

run: all
    qemu-system-i386 -drive format=raw,file=os-image,index=0,if=floppy

# disk image that the computer loads
os-image: boot/boot_sect.bin kernel.bin
    cat $^ > $@

# $^ - all target's dependency files
# entry - skace na main() jezgre
# kernel - kompajlirana C-jezgra
kernel.bin: kernel/kernel_entry.o ${OBJ}
    i686-elf-ld -o $@ -Ttext 0x1000 $^ --oformat binary

# $< - 1st dependency, $@ - target file
# generic rule for compiling C code to an object file
# for simplicity, C files depend on all header files
# -Wall - all Warnings
%.o: %.c ${HEADERS}
    i686-elf-gcc -Wall -ffreestanding -c $< -o $@

# assemble the kernel entry
%.o: %.asm
    nasm $< -f elf -o $@

%.bin: %.asm
    nasm $< -f bin -o $@

# obrise sve izgenerirane datoteke    42
clean:
    rm -fr *.bin *.dis *.o os-image *.map
    rm -fr kernel/*.o boot/*.bin drivers/*.o

# disasemlanje jezgre; moze biti korisno za debug
kernel.dis: kernel.bin

```

3 Zaključak

Ovaj rad prikazao je proces razvijanja operacijskog sustava od početka. Prvenstvena mu je svrha razumijevanje osnovnih principa njegovog funkcioniranja. Počevši od *bootloadera* i završivši u jezgri. Moguće ga je koristiti u edukativne svrhe u smislu samog upoznavanja s materijalom, ali i nadogradnje. Npr. funkcija `handle_scrolling()` može se implementirati tako da pamti i linije koje su van ekrana, odnosno da podržava i skrolanje unatrag. Funkcija `print_char()` mogla bi obavijestiti korisnika ukoliko pokuša pisati van ekrana i slično. Procesor se može prebaciti u 64-bitni *Long mode*. U samu jezgru mogu se dodati još raznorazne mogućnosti, a naposljetku čak i pisati korisnički programi za razvijeni OS.

4 Literatura

- [1] Blundell, N.: Writing a Simple Operating System from Scratch, School of Computer Science, University of Birmingham, Ujedinjeno Kraljevstvo, 2010.
- [2] Lutkevich B.: BIOS (basic input/output system),
<https://whatis.techtarget.com/definition/BIOS-basic-input-output-system> (posjećeno 19.08.2021.)
- [3] MiniTool[®] Software Ltd.: [Disk Basic Knowledge] What Is Cylinder-head-sector? [Help], <https://www.partitionwizard.com/help/what-is-chs.html> (posjećeno 26.07.2021.)
- [4] Evans D.: x86 Assembly Guide,
<https://www.cs.virginia.edu/~evans/cs216/guides/x86.html> (posjećeno 06.08.2021.)
- [5] OSDev.org: GDT Tutorial,
https://wiki.osdev.org/GDT_Tutorial#From_flat.2C_protected_mode (posjećeno 07.08.2021.)
- [6] Code Project: Using Inline Assembly in C/C++,
<https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C> (posjećeno 08.07.2021.)
- [7] OSDev.org: GCC Cross-Compiler, https://wiki.osdev.org/GCC_Cross-Compiler (posjećeno 28.07.2021.)
- [8] <https://ftp.gnu.org/gnu/binutils/?C=M;O=D> (posjećeno 30.07.2021.)
- [9] <https://ftp.gnu.org/gnu/gcc/> (posjećeno 30.07.2021.)
- [10] Linuxize: How to Set and List Environment Variables in Linux,
<https://linuxize.com/post/how-to-set-and-list-environment-variables-in-linux/> (posjećeno 30.07.2021.)