Математичка гимназија

МАТУРСКИ РАД

- из рачунарства и информатике -

32bit X86 језгро оперативног система

Ученик:
Алекса Вучковић IVц

Ментор:
Милош Арсић

Београд, јун 2022.

Садржај

1

Увод

Идеја за овај рад произишла је из екстензивног коришћења GNU/Linux система, као и жеља за разумевањем рада рачунара на најнижем нивоу.

Цео код је писан у GNU Assembler-у и C-у и може се наћи на GitHub-у на страници `https://github.com/aleksav013/mykernel`. Код, заједно са свим алатима за његово коришћење и употребу, је доступан под GPLv3 лиценцом.

Овај пројекат се константно унапређује и немогуће је одржавати синхронизованим фајлове који сачињавају оперативни систем, а који се истовремено помињу у овом раду. Из тог разлога, рад ће бити написан за верзију оперативног система 1.0.0. На страници GitHub-а ова верзија се може наћи под tag/s секцијом на страници пројекта.

Ради лакшег компајловања кода од стране читаоца, креиран је toolchain - сет алата, специфично за компајловање овог оперативног система, који се може наћи на страници `https://github.com/aleksav013/aleksa-toolchain`, такође доступан под GPLv3 лиценцом отвореног кода. Сет алата aleksa-toolchain креиран је такође и из разлога што је заправо неизбежан процес креирања cross-compiler-а - компајлера који компајлује код за други систем/архитектуру на тренутном систему/архитектури. На тај начин и други људи осим аутора могу допринети изради и побољшању овог оперативног система у будућности.

Овај пројекат има за циљ да прикаже поступак креирања једног за сада врло једноставног оперативног система, као и да поткрепи читаоце теоријом потребном за његово разумевање.

Тренутно ради:

- Boot
- VGA излаз
- GDT
- IDT
- IRQ & PIC
- PS/2 тастатура
- PIT
- Heap
- Paging
- C библиотека

Планирано у ближој будућности:

- Filesystem
- Framebuffer
- Networking
- Real User Space

О функционалностима које су имплементиране, моћи ћете да сазнате више у овом раду. Ауторов план је да предстојеће слободно време након матурирања и током студија, искористи на унапређење овог малог оперативног система, и да у му овај рад као такав буде основа за наредни.

2

X86 архитектура

X86 је фамилија архитектура CISC сета инструкција развијена од стране Интела која је базирана на 8086 микропроцесору. 8086 је настао 1978. као шеснаестобитна екстензија 8080 осмобитног микропроцесора, да би 1985. године настао 80386 са величином регистра од чак 32 бита. Сви процесори из ове фамилије носе заједнички назив управо по суфиксу 86 који се појављује у именима свих ових процесора.

У периоду од 1999. до 2003. AMD је проширио величину регистра на 64 бита и та архитектура има више назива од којих су најкоришћенији: x86_64, amd64 и x64. Велика већина уређаја данас користи x86_64 архитектуру процесора.

Језго оперативног система приказано у овом раду базирано је на X86 архитектури са регистром величине 32 бита. Конкретна верзија ове архитектуре је i386. Ова верзија добила је име по Интеловом микропроцесору под именом 80386 што означава трећу генерацију X86 архитектуре.

2.1 Регистри процесора

Постоји више врста регистара процесора[1, п. 75]. Неки од основних регистара које је потребно поменути дати су у тексту који следи. Разлог због којег су наведена и имена регистара претходних верзија X86 архитектуре је због тога што је могуће адресирати првих x битова, ако се користи име регистра за x-тобитну верзију X86 архитектуре. Наравно, ово важи само уколико је дужина регистра нашег система (32 бита), већа или једнака дужини регистра чију нотацију користимо.

Регистри опште намене:

8bit	al	bl	cl	dl	sil	dil	spl	bpl
16bit	ax	bx	cx	dx	si	di	sp	bp
32bit	eax	ebx	ecx	edx	esi	edi	esp	ebp
64bit	rax	rbx	rcx	rdx	rsi	rdi	rsp	rdp

Сегментни регистри:

cs	ds	ss	es	fs	gs

Контролни регистри:

cr0	cr2	cr3	cr4	cr8

Системски регистри (показивачи на табеле):

gdtr	ldtr	idtr

Осим поменутих, почев од шеснаестобитне X86 архитектуре, постоје и регистри ah, bh, ch, dh (h-higher) који представљају горњу половину (од 9. до 16. бита) ax, bx, cx, dx регистара редом. У даљем тексту биће приказан однос између регистара о коме је раније било речи, као и приказ и на то од ког до ког бита регистра се односи дата нотација.

63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0
						ah	al
						ax	
				eax			
rax							

Приметимо да уколико нас интересује вредност другог бајта у X86_64 архитектури, до ње можемо доћи на следећа 4 начина: ah, ax&0xFF00, eax&0x0000FF00 или rax&0x000000000000FF00.

2.2 Регистри опште намене

Регистри опште намене имају улогу у чувању операнди и показивача:

- Операнди за логичке и аритметичке операције
- Операнди за адресне калкулације
- Показивача на меморијску локацију

Регистри опште намене се могу користити произвољно према потреби. Међутим, дизајнери хардвера су увидели да постоји могућност даљих оптимизација уколико се сваком од ових регистара додели нека специфична улога у којој је мало бољи од осталих регистара опште намене.

На тај начин компајлери већину времена креирају бољи асемблерски код него људи, просто из разлога што сваки од регистара опште намене такође користе и за њихову специфичну функцију сваки пут где је то могуће.

Специфична улога регистара опште намене:

- eax - акумулатор за операнде и податке резултата
- ebx - показивач на податке у ds сегменту
- ecx - бројач за петље и операције над стринговима
- edx - показивач на У/И
- esi - показивач на податке на који показује ds регистар; почетни показивач за операције над стринговима
- edi - показивач на податке у сегменту на који показује es регистар; крајњи показивач за операције над стринговима
- esp - показивач на почетак стека
- ebp - показивач на податке у стеку

2.3 Сегментни регистри

Сегментни регистри садрже шеснаестобитне селекторе сегмента. Селектор сегмента је специјалан показивач који идентификује сегмент у меморији. Да би приступили одређеном сегменту у меморији, селектор сегмента који показује на тај сегмент мора бити доступан у одговарајућем сегментном регистру.
Специфична улога сегментних регистара:

- cs - code segment. cs регистар садржи селектор сегмента који показује на сегмент кода у коме се налазе инструкције које се извршавају.
- ds - data segment. Осим ds, сегментни регистри за сегменте података су и es, fs, као и gs.

- ss - stack segment. ss регистар садржи селектор сегмента који показује на сегмент стека где се чува стек програма који се тренутно извршава. За разлику од регистра за сегмент кода, ss регистар се може експлицитно поставити што дозвољава апликацијама да поставе више стекова и да алтернирају између њих.

2.4 Оперативни режими процесора

Оперативни режими процесора у X86 архитектури су:

- Real mode
- Protected mode (16-bit)
- Protected mode (32-bit)
- Compatibility mode (x86-64 only)
- Long mode (x86-64 only)

2.4.1 Real mode

Реални режим је стање процесора које карактерише 20-тобитна сегментирана меморија што значи да се може адресирати само мало више од једног мегабајта. Сви процесори почев од 80286 започињу у овом режиму након паљења рачунара због компатибилности. Да бисмо могли да адресирамо више меморије, потребно је да користимо сегментне регистре.

Поступак којим из реалног режима прелазимо у заштићени режим назива се далеки скок (far jump) где при постављању сегментних регистара "скачемо" из једног сегмента у други. Далеки скок најчешће подразумева промену селектора сегмента, који се налази у сегменту кода, и који показује на сегмент кода у коме се налазе инструкције које се тренутно извршавају.

2.4.2 Сегментација

Сегментација је решење којим се омогућава адресирање више меморије него што је то хардверски предвиђено. Сегментација се постиже коришћењем offset-а. Управо смо уз помоћ сегменатције у реалном режиму адресирали меморију са 20, уместо предвиђених 16 битова.

2.4.3 Protected mode

Заштићени режим је стање процесора у којем процесор има пун приступ целом опсегу меморије за разлику од реалног режима. Максимална величина RAM меморије коју архитектура i386 подржава је 4GB управо због величине регистра од 32 бита ($2^{32}B = 4GB$). Неки процесори који подржавају PAE - Physical Address Extension подржавају и адресирање меморија преко 64 бита. Касније је на овом старндарду утемељено и адресирање меморија на X86_64.

Адресирање свих 32 бита меморије постиже се коришћењем страничења (paging), које нам дозвољава да мапирамо физичку меморију на виртуелну. На тај начин приликом извршавања сваког програма може се заварати програм тако да мисли да његово извршавање креће од почека меморије. Из тог разлога значајно нам је олакшано извршавање програма јер нам сви програми постају независни од адресе физичке меморије где се они заправо налазе, док приликом креирања програма можемо фиксно поставити адресу прве инструкције.

Ово је стање процесора који наше језгро тренутно користи.

2.4.4 Compatibility mode

Режим компатибилности се покреће током преласка из заштићеног режима у дугачки режим. Режим компатибилности ради слично заштићеном режиму и користи се за омогућавање учитавања 64-битних структура пре уласка у одговарајући дугачки режим.

2.4.5 Long mode

Са временом, постало је јасно да 4GB адресибилне меморије коју нам пружа 32-битна X86 архитектура није довољно. Из тог разлога настао је long mode. Дугачки режим је заправо само екстензија 32-битног сета инструкција, али за разлику од транзиције са 16-битног на 32-битни сет инструкција, сада је уклоњен велики део инструкција. Ово не утиче на компатибилност са старијим верзијама извршних фајлова, али мења начин на који асемблер и компајлери за нови код морају да функционишу.

Креирањем ове екстензије процесора настала је и нова AMD64 архитектура, која је добила име управо по AMD компанији која је дизајнирала ову спецификацију. То је био први пут у историји да је врло важна одлука која се тиче X86 архитектуре донесена од неке друге компаније, а не самог Intel-a, али и да је Intel одлучио да прихвати технологију од страног извора.

2.5 Екстензије процесора

Екстензије процесора у X86 архитектури су:

- x86-64
- PAE/NX
- x87
- SSE
- AVX

2.5.1 x86-64

Проширење x86-64 је прилично експанзивно, примена 64-bitnog оперативног режима и 64-битно адресирање. Данас је готово немогуће пронаћи рачунар који не подржава x86-64.

2.5.2 PAE/NX

PAE/NX примењује страничне поруке на 4 нивоа и додаје прегршт својстава страницама да би се применила опсежна заштита меморије, укључујући NX, заставицу за онемогућавање извршавања на одређеној страници. Већина оперативних рачунара данас подржава PAE/NX.

2.5.3 x87

87 је проширење за floating point израчунавања. Иако је технички екстензија, подржана је на већини x86 процесора.

2.5.4 SSE

SSE је 128-битно проширење за податке. Додаје 8-16 128-битних XMM регистра опште намене и инструкција за извођење математике SIMD вектора. SSE има неколико верзија. x86-64 процесори су обавезни да подржавају најмање SSE2.

2.5.5 AVX

AVX је 256-битно проширење за податке. Обавља сврху сличну SSE и имплементира неколико 256-битних YMM регистара.

3

Boot

3.1 Редослед покретања

Од притиска тастера за паљење рачунара, па до учитавања оперативног система постоји цео један процес. Након притиска тастера, рачунар прво извршава POST (Power On Self Test) рутину, која је једна од почетних фаза BIOS-а (Basic Input Output System). У POST-у, рачунар покушава да иницијализује компоненте рачунарског система једну по једну, и проверава да ли оне испуњавају све услове за стартовање рачунара. Уколико је цео процес прошао без грешака, наставља се даље извршавање BIOS-а. BIOS сада има улогу да пронађе медијум који садржи програм који ће учитати језгро оперативног система у рам меморију рачунара. Тај програм се назива Bootloader.

3.2 Bootloader

Bootloader је програм који се налази у првих 512 бајтова медијума у MBR одељку, и његов задатак је да учита језгро оперативног система у RAM меморију и преда му даље управљање. Због комплексности данашњег софтвера, 512 бајтова је врло мало за учитавање целог bootloader-а. Из тог разлога неки bootloader-и раздвојени су у више делова, најчешће два. Први део је врло мали, стаје у 512 бајтова MBR-а и он служи само да би учитао други део у коме се налази главни код који даље предаје управљање језгру оперативног система.

Неки од најпопуларнијих bootloader-а јесу: GRUB, LILO, systemd-boot, rEFInd, као и Windows Boot Manager. За потребе овог оперативног система коришћен је GRUB о коме ће бити више речи у одељку Коришћени алати.

3.3 Multiboot спецификација

Multiboot спецификација је отворени стандард који језгрима обезбеђује уједначен начин покретања од стране bootloader-а усаглашених са овом спецификацијом. Једна од првих ствари који свако језгро уради је постављање константи дефинисаних multiboot стандардом да би било препознато од стране bootloader-а и да би му било предато даље управљање.

3.4 ELF

ELF је формат за складиштење програма или фрагмената програма на диску, креиран као резултат компајловања и линковања. ELF фајл је подељен на одељке. За програм који може да се изврши, постоје следећи одељци: одељак са текстом за код, одељак са подацима за глобалне променљиве и одељак ".rodata" (read only data) који обично садржи константне ниске. ELF фајл садржи заглавља која описују како ови одељци треба да буду ускладиштени у меморији.

Овај формат дефинисан је у System V ABI (System V Application Binary Interface)[2], који је сет спецификација који дефинише позивање функција, формат објектних фајлова, формат фајлова који могу да се изврше, динамично линковање као и многе друге.

Функције архитектуре i386, по конвенцији, повратне вредности целобројног типа чувају у eax регистру, док се вредности које не стају у 32 бита eax регистра "преливају" у edx регистар. Ова информација ће нам бити кључна када будемо позивали функција написане у C-у из Assembler-а.

4

Коришћени алати

У даљем тексту се могу видети неки од алата коришћених при креирању овог рада. Већина коришћених алата поседује GPLv3 лиценцу. GNU Public Licence је лиценца отвореног кода која дозвољава модификовање и дистрибуирање кода све док је тај код јавно доступан. Једини програми са ове листе који није креирао GNU су: QEMU виртуелна машина, git и NeoVim који користе GPLv2 лиценцу.

Оперативни систем коришћен у изради овог пројекта је Artix Linux. Artix Linux је GNU/Linux дистрибуција базирана на Arch Linux-у. Већина коришћених програма је већ компајлована и спремна за употребу и налази се у официјалним репозиторијима.

За програме који су морали бити мануелно компајловани дате су инструкције у њиховој подсекцији. Binutils и GCC су морали бити накнадно компајловани и то да не би користили стандардну библиотеку коју нам је обезбедио оперативни систем домаћин (онај на коме се компајлује овај пројекат), већ посебну библотеку креирану за ово језгро. За остале програме који су коришћени препорука је користити оне који су доступни као спремни пакети у изворима одабране диструбуције GNU/Linux-а.

4.1 Binutils

Изворни код софтвера се може наћи на страници `https://www.gnu.org/software/binutils/`, заједно са упутством за компајловање и коришћење.

Овај софтверски пакет садржи програме неопходне за израду програма од којих су најкоришћенији асемблер (as), линкер (ld), као и програм за креирање библиотека (ar).

4.1.1 Пре додавања C библиотеке

Из разлога што се не користи стандардна библиотека већ самостално написана специфично за овај пројекат, потребно је мануелно компајловати GNU Binutils. Међутим, постоји могућност коришћења већ спремног пакета који се за дистрибуције базиране на Arch Linux-у може наћи на станици `https://aur.archlinux.org/packages/i686-elf-binutils/`. Поједине дистрибуције већ имају овај пакет компајлован, али је препорука мануелно компајловати да би се избегла некомпатибилност, а и просто из разлога што ће након формирања наше C библиотеке бити неопходно компајловати овај програм за сваки систем посебно.

За оне које желе сами да компајлују binutils, дат је део инструкција који се разликује од упутства датог на званичном сајту а тиче се конфигурисања пре компилације.

```
mkdir build
cd build

../configure \
--target=i686-elf \
--with-sysroot \
--prefix=/usr \
--bindir=/usr/bin \
--libdir=/usr/lib/i686-elf \
--disable-nls \
--disable-werror

make
make install
```

4.1.2 Након додавања C библиотеке

Након додавања наше C библиотеке потребно је компајловати GNU Binutils тако да ту библиотеку и користи приликом компајловања нашег оперативног система.
Напомена: Потребно је поставити $SYSROOT на локацију која садржи библиотеку. То је могуће урадити на следећи начин:

```
export SYSROOT=/put/do/biblioteke
```

Инструкције за компајловање дате су у наставку.

```
../configure \
--target=i686-elf \
--with-sysroot=$SYSROOT \
--prefix=/usr \
--bindir=/usr/bin \
--libdir=/usr/lib/i686-elf \
--disable-nls \
--disable-werror
```

4.1.3 GNU Assembler

Иако тренутно постоје много популарније алтернативе попут NASM (Netwide Assembler) и MASM (Microsoft Assembler) који користе новију Интелову синтаксу, аутор се ипак одлучио за GASM због компатибилности са GCC компајлером. GASM користи старију AT&T синтаксу коју карактерише: обрнут поредак параметара, префикс пре имена регистара и вредности константи, а и величина параметара мора бити експлицитно дефинисана. Због тога ће можда неким читаоцима бити користан програм intel2gas који се за Arch Linux може наћи на станици `https://aur.archlinux.org/packages/intel2gas/`, а који преводи распрострањенију Интелову синтаксу у AT&T синтаксу.

GNU Assembler је коришћен за компајловање дела кода написаног у асемблеру.

4.1.4 GNU Linker

GNU Linker је коришћен за линковање, тј. "спајање" свог компајлованог кода у једну бинарну датотеку типа ELF која представља језгро (кернел). Линкер користи src/linker.ld скрипту која дефинише на који начин се пакују секције објектних фајлова, од које меморијске локације почиње учитавање језгра, као и која је почетна функција од које креће извршавање. Ова скрипта у ствари представља рецепт за креирање наведене ELF датотеке која у нашем случају чини оперативни систем.

4.2 GCC

Изворни код софтвера се може наћи на страници `https://gcc.gnu.org/`, заједно са упутством за компајловање и коришћење.

Исто као и за GNU Binutils, могуће је пронаћи већ спреман пакет у појединим дистрибуцијама, али је препорука мануелно компајловати из претходно наведених разлога. За Arch Linux овај пакет се може наћи на станици `https://aur.archlinux.org/packages/i686-elf-gcc/`.

GCC је GNU-ов сет компајлера који је коришћен за компајловање овог оперативног система.

4.2.1 Пре додавања C библиотеке

За оне које желе сами да компајлују GCC, дат је део инструкција који се разликује од упутства датог на званичном сајту а тиче се конфигурисања пре компилације.

```
mkdir build
cd build

../configure \
--target=i686-elf \
--prefix=/usr \
--disable-nls \
--disable-plugin \
--enable-languages=c,c++ \
--without-headers

make all-gcc
make all-target-libgcc

make -k check || true

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

4.2.2 Након додавања C библиотеке

Инструкције за компајловање након додавања библиотеке дате су у наставку.

```
../configure \
--target=i686-elf \
--prefix=/usr \
--with-sysroot=$SYSROOT \
--disable-nls \
--disable-plugin \
--enable-languages=c,c++
```

Компајловање GCC-а са датим опцијама обезбеђује нам коришћење дате библиотеке уместо стандардне коју нам је обезбедио оперативни систем домаћин.

4.3 GRUB

Изворни код софтвера се може наћи на страници `https://www.gnu.org/software/grub/`, заједно са упутством за компајловање и коришћење.

GRUB (GRand Unified Bootloader) је bootloader који је коришћен на овом пројекту. Тренутно коришћена верзија овог софтвера је GRUB 2.
Конфигурациони фајл grub.cfg дефинише одакле овај bootloader учитава језгро и колики је временски оквир између избора језгара односно оперативних система.

4.4 QEMU

Више информација о софтверу се може наћи на страници `https://www.qemu.org/`, заједно са упутством за коришћење.

QEMU је виртуелна машина у којој ће језгро бити тестирано и приказано зарад практичних разлога. QEMU је одабран за овај пројекат јер за разлику од других виртуелних машина поседује cli (command line interface) из кога се лако може позивати помоћу скрипти као што су Makefile-ови.

4.5 Make

Изворни код софтвера се може наћи на страници `https://www.gnu.org/software/make/`, заједно са упутством за компајловање и коришћење.

Make[3] нам омогућава да са лакоћом одржавамо и манипулишемо изворним фајловима. Могуће је све компајловати, обрисати, креирати iso фајл као и покренути QEMU виртуелну машину са само једном кључном речи у терминалу. Креирани Makefile за потребе овог пројекта биће детаљно објашњен у секцији Build system.

4.6 Мање битни алати

4.6.1 NeoVim

NeoVim је уређивач текста настао од Vim-a (Vi improved) [4]. Конфигурацијски фајлови аутора, могу се наћи на `https://github.com/aleksav013/nvim`, и имају за циљ да створе окружење погодно за рад на овом пројекту.

4.6.2 git

Креатор овог програма је Linus Torvalds, човек који је креирао Linux кернел. Изворни код софтвера се може наћи на страници `https://git.kernel.org/pub/scm/git/git.git`.

Git је програм који нам помаже да одржавамо изводне фајлове синхронизованим са репозиторијумом. Осим тога значајан је и његов систем контроле верзија - могућност да се за сваку промену (commit) види тачно који су се фајлови изменили и која је разлика између неке две верзије пројекта.

4.6.3 xorriso (libisoburn)

Изворни код софтвера се може наћи на страници `https://www.gnu.org/software/xorriso/`, заједно са упутством за компајловање и коришћење.

Служи за креирање ISO фајлова који се могу "нарезати" на CD диск или USB флеш и који се касније могу користити као оптички диск који садржи све што је неопходно за успешно подизање система.

4.6.4 GDB

Изворни код софтвера се може наћи на страници `https://www.sourceware.org/gdb/`, заједно са упутством за компајловање и коришћење.

GDB је GNU-ов debugger који служи за проналажење грешака у коду, као и активно праћење извршавања програма (у овом случају језгра). Уз помоћ овог програма установљено је шта сваки тастер на тастатури враћа након притиска, што је значајно убзало процес мапирања тастатуре.

5

Инспирација

5.1 Minix

Minix је оперативни систем базиран на UNIX-у који користи микрокернелску архитектуру. Почев од верзије 2.0 задовољава POSIX (Portable Operating System Interface) стандард. Почетне верзије Minix оперативног система креирао је Ендрју С. Таненбаум као професор Вриje универзитета у Амстердаму. Професор Таненбаум креирао је оперативни систем као пример принципа који су приказани у његовој књизи "Оперативни системи: Дизајн и Имплементација" (Operating Systems: Design and Implementation)[5]. Осим ове књиге, Таненбаум написао је и многе друге које могу бити од користи свима који желе да се баве дизајном оперативних система[6][7]. Занимљива чињеница је да је Minix најпопуларнији оперативни систем чије језгро користи микрокернелску архитектуру.

Спекулише се да сви Интелови процесори након Core2Duo серије који имају чип под називом Intel ME (Intel Managment Engine), заправо покрећу Minix оперативни систем. Ово би Minix учинило најкоришћенијим оперативним системом икада.

5.2 Linux

Већ дуже од две године користим оперативни систем са Linux језгром што је био и главни разлог зашто сам изабрао рад на ову тему. У току израде овог пројекта, пожелео сам да боље разумем како функционише израда Linux кернела и како је то "заједница" она која га покреће и одржава, а не нека велика корпорација. Стога сам одлучио да дам свој допринос и послао сам неколико закрпа (patch-ева) који су примљени и имплеметирани у новој 5.17 верзији Linux кернела. Ово искуство показало ми је колико је озбиљна организација која стоји иза целог тог пројекта и колико је битно радити на свим сферама неког пројекта као што су и израда доку-

ментације и спремност да се едукују и заинтересују млади људи којима ће овакво искуство значити у даљем професионалном усавршавању. Драго ми је што ме је овај пројекат одважио да начиним тај корак и допринесем можда највећем софтверском пројекту икада.

5.3 BSD

Најпопуларнији BSD оперативни системи су редом FreeBSD[8], OpenBSD и NetBSD. Сваки од њих има различите циљеве, али се сви слажу у једном: софтвер мора остати минималан. Сви ови оперативни системи своје системске програме и библиотеке имају написано у дупло, ако не и неколико пута мање линија кода него GNU/Linux. То их чини много лакшим за одржавање, али и много сигурнијим.

BSD оперативни системи разликују се од GNU/Linux-а по томе што су код BSD-а језгро и кориснички програми целина која је настала од стране једног тима људи и одржавају се упоредо, док је код GNU/Linux-а језгро Linux, а GNU Coreutils чини корисничке програме (user space) и њихов развој је потпуно независан. Због те чињенице, BSD је много лакше одржавати и много брже се могу имплементирати нове идеје и могућности, просто из разлога што не зависи од друге стране да се пре свега усклади, а потом и имплементира дату промену.

Такође осим горе наведених разлика, још једна веома битна разлика је то што BSD користи MIT лиценцу док GNU/Linux користи GPLv2/GPLv3 лиценцу. То значи да за BSD системе након модификације не морамо учинити код јавно доступним. Та чињеница чини BSD врло примамљивим за велике корпорације које на тај начин своја софтверска решења могу базирати на већ постојећим. Тако на пример многе конзоле покрећу управо измењену верзију FreeBSD-а, док са друге стране имамо и компаније попут Apple-а која такође користи FreeBSD, али као подлогу за своје оперативне системе за рачунаре и мобилне телефоне.

6

Build system

Систем који креира toolchain који се користи за компајловање, линковање и креирање употребљивог оперативног система у ISO формату. Осим сета алата може се видети и Makefile који памти на који се тачно начин користи тај сет алата да би се изградило језгро.

6.1 aleksa-toolchain

У склопу овог пројекта настао је и aleksa-toolchain, као и shell[9] скрипта која креира тај toolchain (сет алата) који се даље користи за компајловање језгра.
У тексту који следи може се видети које све функције скрипта садржи.

```
main()
{
    download
    extract
    patch_gnu
    build_binutils
    install_headers
    build_gcc
    setup_compiler
}
```

Скрипта прво скида изворни код binutils-а и GCC-а, екстрактује из архиве, а затим је patch-ује (примењује скуп измена). Након тога коришћењем GCC компајлера на host систему, компајлује се посебан GCC компајлер који служи за компајловање изворног кода намењеног језгру овог оперативног система. Након тога он се инсталира на пут /opt/aleksa/usr/bin/, где се налази нови $SYSROOT који користи Makefile за изградњу језгра.

6.2 Makefile

Makefile је фајл који контролише понашање make команде у терминалу. Сваки Makefile садржи неки сет правила који дефинише процес који се покреће укуцавањем команде make <ime_pravila>. У тексту који следи може се видети који је то сет правила која су дефинисана за овај пројекат.

Makefile:

```
# RULES
.PHONY: all clean compile install_headers iso run run-iso
all: compile

clean:
	$(RM) $(BUILD_DIR) $(ISO_DIR) $(ISO)

$(BINARY): $(OBJ)
	$(CC) -T $(LINKER) -o $(BINARY) $(CFLAGS) -nostdlib -lgcc $(OBJ)

install_headers:
	./scripts/install_headers.sh

compile:
	$(MAKE) install_headers
	@$(MAKE) --directory $(SOURCE_DIR)
	$(MAKE) $(BINARY)

$(ISO): $(OBJ) $(GRUB_CFG)
ifeq ($(shell ./scripts/is_multiboot.sh), 0)
	$(MKDIR) $(ISO_DIR)/boot/grub
	$(CP) $(BINARY) $(ISO_DIR)/boot/$(TARGET).bin
	$(CP) $(GRUB_CFG) $(ISO_DIR)/boot/grub/grub.cfg
	grub-mkrescue -o $(ISO) $(ISO_DIR)
else
	@echo "$(BINARY) is not x86 multiboot compatible"
	$(RM) $(BINARY)
endif

iso: compile
	$(MAKE) $(ISO)

run: compile
	$(QEMU) -kernel $(BINARY) $(QEMU_DEBUG)

run-iso: iso
	$(QEMU) -cdrom $(ISO)
```

Осим наведеног сета правила, у Makefile-у можемо дефинисати неке променљиве које би нам омогућиле краћи запис, избегавање понављања константи, као и значајно лакше мењање истих. Оне функционишу на сличан начин као и претпроцесорске директиве у C-у. Приметимо да испред неких линија у којима додељујемо вредности променљивима имамо и кључну реч export. То значи да ће се вредност дате променљиве, пренети и у рекурзивни позив Makefile-а.

```
# HOST ARCH
ARCH=i686-aleksa-


# GCC CROSS COMPILER
export CC=$(ARCH)gcc
export AS=$(ARCH)as

WARNINGS := -Wall -Wextra -pedantic -Wshadow -Wpointer-arith -Wcast-align \
            -Wwrite-strings -Wmissing-prototypes -Wmissing-declarations \
            -Wredundant-decls -Wnested-externs -Winline -Wno-long-long \
            -Wconversion -Wstrict-prototypes

export CFLAGS=-std=gnu99 -O3 $(WARNINGS) -ffreestanding -fstack-protector-all


# TOOLS
export MKDIR=mkdir -p
export RM=rm -rf
export CP=cp -r
QEMU=qemu-system-i386
```

```
# DIRS

## SOURCE
SOURCE_DIR=src

## BUILD
ISO_DIR=isodir
BUILD_DIR=${CURDIR}/build
export AS_OBJECT_DIR=$(BUILD_DIR)/as
export C_OBJECT_DIR=$(BUILD_DIR)/c

## GCC USR
GCC_USR_DIR=$(shell $(CC) -print-file-name=)
```

Можемо видети да је ово комплекснија форма Makefile-а јер садржи процес рекурзивног позивања $(MAKE) команде. То значи да овај пројекат садржи више различитих Makefile-ова за сваки део пројекта, што је и стандардна процедура при великим пројектима као што је овај. Конкретно код овог језгра постоје још два Makefile-а, а тичу се кода написаног у C-у, као и кода написаног у GNU Assembler-у за X86 архитектуру.

src/as/Makefile:

```
.PHONY: all

all: $(AS_OBJECTS)

$(AS_OBJECT_DIR)/%.o: %.s Makefile
    $(AS) $< -o $@
```

src/c/Makefile:

```
.PHONY: all

all: $(C_OBJECTS)

$(C_OBJECT_DIR)/%.o: %.c Makefile
    $(CC) $(CFLAGS) -MD -MP -c $< -o $@

-include $(C_OBJECTS:.o=.d)
```

На тај начин би са лакоћом пројекат могао бити проширен на више различитих архитектура процесора, што је и ауторов план у будућности. Библиотеке, интерактивни shell, као и драјвер за тастатуру написан за ово језго се не би мењали, док би због специфичности друге архитектуре биле промењене компоненте задужене за почетну иницијализацију и контолисање хардвера на најнижем нивоу.

За сваки пројекат могуће је аутоматизовати део посла креирањем Makefile, па тако на пример и овај матурски рад који је искуцан у LaTeX-у има свој Makefile:

```
NAME=kernel
TEX_FILE=$(NAME).tex
BIB_FILE=$(NAME).bib
PDF_FILE=$(NAME).pdf

.PHONY: all clean
all: $(PDF_FILE)

$(PDF_FILE): $(TEX_FILE) $(BIB_FILE)
    pdflatex $(TEX_FILE)
    bibtex $(NAME)
    pdflatex $(TEX_FILE)

clean:
    git clean -dfx
    rm -f $(PDF_FILE)
```

7

Језгро оперативног система

Рад је првобитно био замишљен као поступно излагање настајања овог оперативног система, али се касније аутор одлучио да ипак изложи само тренутну верзију рада, с обрзиром на то да би рад био непотребно дужи. Језгро је, као што је већ речено написано у асемблеру и C програмском језику[10][11].

7.1 Почетак

src/as/boot.s:

У првом делу постављамо промењиве на вредности које су одређене multiboot стандардом да би bootloader препознао наше језгро.

```
.set ALIGN,    1<<0
.set MEMINFO,  1<<1
.set FLAGS,    ALIGN | MEMINFO
.set MAGIC,    0x1BADB002
.set CHECKSUM, -(MAGIC + FLAGS)
```

Након тога постављамо првих 512 бајтова на претходно поменуте вредности, али тако да за сваку промењиву остављамо 32 бита простора.

```
.section .multiboot
.align 4
.long MAGIC
.long FLAGS
.long CHECKSUM
```

Постављамо промењиве које чувају адресе на којима почињу сегменти кода и података, редом.

```
.set CODE_SEGMENT, 0x08
.set DATA_SEGMENT, 0x10
```

Дефинишемо секцију .bss у којој креирамо стек и додељујемо му 16 килобајта.

```
.section .bss
.align 16
stack_bottom:
.skip 16384
stack_top:
```

Дефинишемо почетну функцију _start позивајући функцију за иницијализацију gdt табеле и "скачемо" на сегмент кода. Овај поступак има назив "far jump" јер скачемо ван текућег сегмента.

```
.section .text
.global _start
.type _start, @function
_start:
    call init_gdt_table
    ljmp $CODE_SEGMENT, $code
```

У сегменту кода постављамо сегментне регистре на адресу сегмента података. Затим постављамо esp регистар на почетак стека који смо иницијализовали у .bss секцији и предајемо управљање kernel_main функцији.

```
code:
    movw $DATA_SEGMENT, %ax
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %fs
    movw %ax, %gs
    movw %ax, %ss
    movl $stack_top, %esp
    cli
    call _init
    call kernel_main
    hlt
```

Постављамо величину функције _start што нам касније може бити корисно при debug-овању.

```
.size _start, . - _start
```

7.2 Испис на екран - VGA

src/c/vga.c:

Коришћене библиотеке:

```
#include<source/vga.h>
#include<types.h>
#include<source/string.h>
#include<asm.h>
```

Приметимо да у C-у користимо uintX_t променљиве. То је због тога што нам је у оваквом окружењу врло битно да пазимо на величину коју заузимају променљиве.

```
size_t terminal_row;
size_t terminal_column;
uint8_t terminal_color;
uint16_t* terminal_buffer;
```

4 значајнија бита означавају боју позадине, док остала 4 бита означавају боју карактера.

```
void set_color(enum vga_color fg, enum vga_color bg)
{
    terminal_color = (uint8_t)(fg|bg<<4);
}
```

```
static inline uint16_t vga_entry(char uc, uint8_t color);
static inline uint16_t vga_entry(char uc, uint8_t color)
{
    return (uint16_t)(uc|color<<8);
}
```

На VGA излаз исписујемо карактере тако што почев од адресе 0xB80000 пишемо шеснаестобитне вредности које се преводе у карактере и њихову боју. 8 значајнијих битова одређују боју карактера док преосталих 8 битова означавају карактер.

```
void terminal_initialize(void)
{
    terminal_row=0;
    terminal_column=0;
    set_color(VGA_COLOR_LIGHT_GREY, VGA_COLOR_BLACK);
    terminal_buffer=(uint16_t*) 0xB8000;
    for(size_t y=0;y<VGA_HEIGHT;y++)
    {
        for(size_t x=0;x<VGA_WIDTH;x++)
        {
            const size_t index=y*VGA_WIDTH+x;
            terminal_buffer[index]=vga_entry(' ', terminal_color);
        }
    }
}
```

Функција која исписује одређени карактер на екрану.

```
void terminal_putentryat(char c, uint8_t color, size_t x, size_t y)
{
    const size_t index=y*VGA_WIDTH+x;
    terminal_buffer[index]=vga_entry(c, color);
}
```

Функција која помера све до сада исписано за један ред на горе и ослобађа нови ред за испис када понестане места на екрану.

```
void movescreen(void)
{
    terminal_row--;
    for(size_t i=0;i<VGA_HEIGHT;i++) for(size_t j=0;j<VGA_WIDTH;j++)
        terminal_buffer[i*VGA_WIDTH+j]=terminal_buffer[(i+1)*VGA_WIDTH+j];
}
```

Функција која поставља бројаче колоне и реда на следеће, углавном празно, поље на екрану.

```
void next_field(void)
{
    if(++terminal_column==VGA_WIDTH) terminal_column=0,terminal_row++;
}
```

Функција која поставља бројаче колоне и реда на претходно поље на екрану и користи се приликом брисања претходно исписаних карактера.

```
void previous_field(void)
{
    if(terminal_column) terminal_column--;
    else terminal_row--,terminal_column=VGA_WIDTH-1;
}
```

Функција која исписује један карактер на екрану. Функција проверава да ли је потребно исписати нови ред уместо карактера '\n', као и да ли је потребно ослободити нови ред уколико се екран попунио.

```
void terminal_putchar(char c)
{
    if(c=='\n') terminal_column=0,terminal_row++;
    else
    {
        terminal_putentryat(c, terminal_color, terminal_column, terminal_row);
        next_field();
    }
    if (terminal_row==VGA_HEIGHT) movescreen();
}
```

Функција која исписује низ карактера на екран.

```
void terminal_writestring(const char* data)
{
    for(int i=0;data[i]!='\0';i++) terminal_putchar(data[i]);
}
```

Функција која исписује целобројну вредност на екран тако што је прво претвори у низ карактера а затим искористи претходну функцију.

```
void terminal_writeint(uint32_t num)
{
    char string[100];
    for(int i=0;i<100;i++) string[i]='\0';
    char *str=string;
    itos(num,str);
    terminal_writestring(str);
}
```

Функција која исписује реалан број на екран тако што је прво претвори у низ карактера а затим искористи функцију за испис низа карактера.

```
void terminal_writefloat(double num)
{
    char string[100];
    for(int i=0;i<100;i++) string[i]='\0';
    char *str=string;
    ftos(num,str);
    terminal_writestring(str);
}
```

Функција која брише све до сада исписане карактере са екрана и поставља бројаче колоне и реда на почетну позицију.

```
void clear(void)
{
    for(size_t i=0;i<VGA_HEIGHT;i++) for(size_t j=0;j<VGA_WIDTH;j++)
        terminal_putchar(' ');
    terminal_column=0;
    terminal_row=0;
}
```

7.3 Global Desctiptor Table

Глобална табела дескриптора је структура у којој се чувају информације о сегментима меморије.

src/include/source/gdt.h:

Користимо #ifndef и #define директиве да би се осигурали да се овај фајл укључује (include-yje) само једном у току целог рада оперативног система. Ове директиве

делују као осигурач који прескаче све декларације уколико су оне већ дефинисане под тим именом.

```
#ifndef SOURCE_GDT_H
#define SOURCE_GDT_H
```

Аутор је креирао библиотеку types.h да би одједном додао све компајлерске библиотеке које дефинишу врсте променљивих као што су int, uint16_t, bool,..., као и константе као што су true, false, UINT16_MAX.... На тај начин у једном реду се додају све библиотеке задужене за то, док се читљивост рада повећава.

```
#include<types.h>
```

Приметимо __attribute__((packed)), на крају дефиниције структуре. То нам означава да се неће остављати места у меморији између променљивих унутар структуре, већ ће се "паковати" једна до друге у меморији.

```
struct gdt_entry
{
    uint16_t limit;
    uint16_t base1;
    uint8_t base2;
    uint8_t access;
    uint8_t limit_flags;
    uint8_t base3;
} __attribute__((packed));
```

Формат који рачунар прихвата за табелу свих дефиниција сегмената меморије.

```
struct gdt_pointer
{
    uint16_t size;
    uint32_t offset;
} __attribute__((packed));
```

Функције које су дефинисане у c/gdt.c, а тичу се иницијализације глобалне табеле дескриптора.

```
void init_gdt_entry(size_t num, uint16_t limit, uint32_t base, uint8_t access,
        uint8_t limit_flags);
void init_gdt_table(void);
```

Директива за затварање која се налази на крају фајла и којом се завршава сегмент кода предвиђен за декларације.

```
#endif
```

src/c/gdt.c:

У source/ фолдеру се налазе прототипи за функције истоименог C изворног фајла.

```
#include<source/gdt.h>
#include<types.h>
```

Функција написана у Assembler-у која учитава табелу дескриптора у предвиђени регистар.

```
extern void load_gdt(struct gdt_pointer *gdtp);
```

Табела сегмената и показивач на њу који се прослеђује претходно поменутој функцији.

```
struct gdt_entry gdt[5];
struct gdt_pointer gdtp;
```

Функција која на основу параметара иницијализује један глобални сегмент меморије.

```
void init_gdt_entry(size_t num, uint16_t limit, uint32_t base, uint8_t access,
        uint8_t limit_flags)
{
    gdt[num].limit=limit;
    gdt[num].base1=(base & 0xffff);
    gdt[num].base2=(uint8_t)((base & 0xff0000) >> 16);
    gdt[num].access=access;
    gdt[num].limit_flags=limit_flags;
    gdt[num].base3=(uint8_t)((base & 0xff000000) >> 24);
}
```

Главна функција која поставља величину показивача који се прослеђује,

```
void init_gdt_table(void)
{
    gdtp.size=sizeof(gdt)-1;
    gdtp.offset=(uint32_t)&gdt;
```

иницијализује све сегменте меморије који ће се користити,

```
    init_gdt_entry(0,0,0,0,0);                  // null segment
    init_gdt_entry(1,0xffff,0,0x9a,0xcf);       // code segment
    init_gdt_entry(2,0xffff,0,0x92,0xcf);       // data segment
    init_gdt_entry(3,0xffff,0,0xfa,0xcf);       // user mode code segment
    init_gdt_entry(4,0xffff,0,0xf2,0xcf);       // user mode data segment
```

као и позива функцију којој прослеђује показивач на глобалну табелу сегмената меморије.

```
    load_gdt(&gdtp);
}
```

7.4 Interrupt Desctiptor Table

Глобална табела прекида је структура која нам говори која функција заправо реагује када процесор добије захтев за прекид (interrupt request). Тада процесор паузира оно

што тренутно ради и предаје управљање одређеној функцији.

Функција је дужна да дојави процесору да је ће она преузети (handle-овати) тај прекид и уколико је прекид фаталан (дељење нулом, triple fault,...) функција има задатак да уклони новонастали проблем и обезбеди оперативном систему опоравак од грешке.

src/include/source/idt.h:

Директиве које се даље понављају у сваком header фајлу. Даље позивање директива ће бити изостављено, ради суштине.

```
#ifndef SOURCE_IDT_H
#define SOURCE_IDT_H
```

Константа коју даље користимо при дефинисању прекида.

```
#define INTERRUPT_GATE_32 0x8E
```

Сегменти у меморији о којима је у прошлој секцији било речи.

```
#define KERNEL_CODE 0x08
#define KERNEL_DATA 0x10
```

Константе које чувају вредности адреса преко којих даље комуницирамо са PIC (Programmable Interrupt Controller)-ом.

```
#define PIC1_COMMAND_PORT 0x20
#define PIC1_DATA_PORT 0x21
#define PIC2_COMMAND_PORT 0xA0
#define PIC2_DATA_PORT 0xA1
```

Структура у којој рачунар прихвата глобалну табелу прекида.

```
struct idt_entry
{
    uint16_t offset1;
    uint16_t selector;
    uint8_t zero;
    uint8_t type_attr;
    uint16_t offset2;
} __attribute__((packed));
```

Показивач на табелу прекида која се прослеђује функцији написаној у Assembler-у, а која учитава табелу прекида у одговарајући регистар.

```
struct idt_pointer
{
    uint16_t size;
    uint32_t offset;
} __attribute__((packed));
```

Прототипи функција који су коришћени у следећем поглављу.

```
void init_idt_entry(size_t num, uint32_t offset, uint16_t selector, uint8_t
        type_attr);
void add_idt_entry(size_t num,uint32_t offset);
void init_pic(void);
void init_idt_table(void);
```

Директиве које се даље понављају у сваком header фајлу. Даље позивање директива ће бити изостављено, ради суштине.

```
#endif
```

7.5 IRQ и PIC

src/c/idt.c:

Коришћене библиотеке:

```
#include<source/idt.h>
#include<types.h>
#include<source/irq.h>
#include<asm.h>
```

Асемблерска функција која учитава показивач на табелу прекида у предвиђени регистар.

```
extern void load_idt(struct idt_pointer *idtp);
```

Табела прекида и показивач на њу који се прослеђује претходно поменутој функцији.

```
struct idt_entry idt[256];
struct idt_pointer idtp;
```

Функција која на основу пружених параметара дефинише прекид у табели.

```
void init_idt_entry(size_t num, uint32_t offset, uint16_t selector, uint8_t
        type_attr)
{
    idt[num].offset1=(uint16_t)(offset & 0xffff);
    idt[num].selector=selector;
    idt[num].zero=0;
    idt[num].type_attr=type_attr;
    idt[num].offset2=(uint16_t)((offset & 0xffff0000)>>16);
}
```

Функција која додаје прекид на табели у низ који садржи све прекиде.

```
void add_idt_entry(size_t num, uint32_t offset)
{
    init_idt_entry(num, offset ,KERNEL_CODE,INTERRUPT_GATE_32);
}
```

Функција која иницијализује контролер прекида. То је један од најбитнијих чипова који дефинише X86 архитектуру. Без њега ова архитектура не би била вођена принципима прекида.

```
void init_pic(void)
{
    ioport_out(PIC1_COMMAND_PORT, 0x11);
    ioport_out(PIC2_COMMAND_PORT, 0x11);
    ioport_out(PIC1_DATA_PORT, 0x20);
    ioport_out(PIC2_DATA_PORT, 0x28);
    ioport_out(PIC1_DATA_PORT, 0x04);
    ioport_out(PIC2_DATA_PORT, 0x02);
    ioport_out(PIC1_DATA_PORT, 0x01);
    ioport_out(PIC2_DATA_PORT, 0x01);
    ioport_out(PIC1_DATA_PORT, 0xff);
    ioport_out(PIC2_DATA_PORT, 0xff);
    ioport_out(PIC1_DATA_PORT, 0xFC);
}
```

Функција која дефинише све прекиде које ћемо користити уз помоћ претходно поменутих функција.

```
void init_idt_table(void)
{
    init_pic();
    add_idt_entry(0,(uint32_t)irq0);
    add_idt_entry(1,(uint32_t)irq1);
    add_idt_entry(2,(uint32_t)irq2);
    add_idt_entry(30,(uint32_t)irq30);
    add_idt_entry(31,(uint32_t)irq31);
    add_idt_entry(32,(uint32_t)timer_irq);
    add_idt_entry(33,(uint32_t)keyboard_irq);
```

Ова функција све податке о прекидима, складиштеним у низ, прослеђује посебно формираној структури показивача на табелу прекида,

```
    idtp.size=sizeof(struct idt_entry)*256-1;
    idtp.offset=(uint32_t)&idt;
```

који се затим прослеђује асемблерској функцији која учитава ту табелу прекида.

```
    load_idt(&idtp);
}
```

7.6 Тастатура

src/c/keyboard.c:

Коришћене библиотеке:

```
#include<source/keyboard.h>
#include<types.h>
#include<asm.h>
#include<source/stdio.h>
#include<source/keymap.h>
#include<source/vga.h>
#include<source/tty.h>
```

Бафер који чува претходно позване команде ради имплементације историје команда, као и бројач бафера на тренутну команду.

```
char buffer[BUFFER_LOG][BUFFER_SIZE];
size_t buffer_size[BUFFER_LOG];
size_t buffer_current=0;
size_t buffer_all=0;
size_t buffer_index=0;
```

Табела која преводи кодове које враћа прекид, у карактере које функција исписује. Такође, чува се и информација да ли је дати карактер тренутно притиснут, што нам може послужити приликом мапирања притиска више карактера одједном.

```
char charcode[256];
char shift_charcode[256];
bool ispressed[128];
```

Функција за иницијализацију тастатуре. Ова функција поставља језик тастатуре и мапира кодове које враћа handler interrupt request-a, на карактере који се исписују на екрану.

```
void init_keyboard()
{
    us_en(charcode);
    us_en_shift(shift_charcode);
}
```

Функција која користи претходно дефинисане функције у драјверу за испис на екран, како би обрисала последњи исписан карактер и вратила се једно поље уназад.

```
void deletelast()
{
    previous_field();
    printf(" ");
    previous_field();
}
```

Функција која брише последњи исписан карактер притиском на backspace, притом пазећи да не испадне ван оквира који је могуће обрисати.

```
void backspace()
{
    if(buffer_index<=0) return;
```

Такође, функција ажурира бафер који чува до сада позиване команде.

```
    deletelast();
    buffer[buffer_current][--buffer_index]='\0';
    return;
}
```

Функција која дефинише понашање језгра након притиска enter тастера на тастатури. Приметимо да се извршава команда која је у том тренутку у баферу, а након тога се дати бафер чува у историји бафера.

```
void enter()
{
    printf("\n");
    if(buffer_index>0)
    {
        tty(buffer[buffer_current]);
        buffer_size[buffer_current]=buffer_index;
        if(buffer_current==buffer_all) buffer_current=(++buffer_all);
        else
        {
            for(size_t i=0;i<BUFFER_SIZE;i++) buffer[buffer_all][i]='\0';
            buffer_current=buffer_all;
        }
        buffer_index=0;
    }
    prompt();
    return;
}
```

Функција која одређује понашање језгра након притиска space тастера на тастатури.

```
void space()
{
    buffer[buffer_current][buffer_index++]=' ';
    printf(" ");
}
```

Функција која одређује понашање језгра након притиска тастера са стрелицом окренутом ка горе, на тастатури. Након притиска тастера, у тренутни бафер се учитава претходно извршена команда. На тај начин корисник може допунити или изменити претходну команду.

```
void keyup()
{
    if(buffer_current>0)
    {
        buffer_size[buffer_current]=buffer_index;
        for(size_t i=0;i<buffer_index;i++) deletelast();
        buffer_current--;
        buffer_index=buffer_size[buffer_current];
        printf("%s",buffer[buffer_current]);
    }
}
```

Функција која одређује понашање језгра након притиска тастера са стрелицом окренутом ка доле, на тастатури.

```
void keydown()
{
    if(buffer_current<buffer_all)
    {
        buffer_size[buffer_current]=buffer_index;
        for(size_t i=0;i<buffer_index;i++) deletelast();
        buffer_current++;
        buffer_index=buffer_size[buffer_current];
        printf("%s",buffer[buffer_current]);
    }
}
```

Функција која је постављена у табели прекида да одговори на прекиде који се тичу тастатуре. Приметимо ioport_in и ioport_out фунцкије. Помоћу њих ми комуницирамо са PIC контролером и одговарамо на дате прекиде, као и примамо информације од истих.

```
void keyboard_handler()
{
    ioport_out(PIC1_COMMAND_PORT, 0x20);
    uint8_t status = ioport_in(KEYBOARD_STATUS_PORT);
```

Уколико је прекид изазван притиском неког тастера на тастатури он се учитава помоћу PIC контролера, а затим се уз помоћ преходно постављеног низа одређује који је карактер притиснут на тастатури.

```
    if (status & 0x1)
    {
        uint8_t keycode = ioport_in(KEYBOARD_DATA_PORT);
        if(keycode<0x80)
        {
            char c=charcode[keycode];
            ispressed[keycode]=1;
            // printf("%d ",&keycode);
```

Функција узима код који враћа прекид и претвара га у карактер или извршава неку од претходно наведених функција у зависности од тога који карактер је притиснут

на тастатури.

```
switch(keycode)
{
    case 0x0E:
        backspace();
        break;
    case 0x1C:
        enter();
        break;
    case 0x39:
        space();
        break;
    case 72:
        keyup();
        break;
    case 80:
        keydown();
        break;
    case 75:
        keyleft();
        break;
    case 77:
        keyright();
        break;
```

У коду који следи се може видети како се на пример нека комбинација карактера може искористити за дефинисање пречица на тастатури.

На пример уколико корисник држи shift тастер на тастатури док притиска друге тастере, исписиваће се велика слова, или секундарни карактери, који су предвиђени за тај тастер. Исто тако, уколико корисник држи ctrl тастер и притисне L на тастатури, позваће се функција која "очисти" екран.

```
default:
    if(c!=' ')
    {
        if(ispressed[lctrl]||ispressed[rctrl])
        {
            if(c=='l')
            {
                clear();
                prompt();
                printf("%s",buffer[buffer_current]);
                return;
            }
        }
        if(ispressed[lshift]||ispressed[rshift])
        {
            c=shift_charcode[keycode];
        }
```

```
                        if(buffer_index<BUFFER_SIZE)
                        {
                            buffer[buffer_current][buffer_index++]=c;
                            printf("%c",c);
                        }
                    }
                    break;
            }
```

```
        }
        else
        {
            ispressed[keycode-0x80]=0;
        }
    }
}
```

7.7 PIT - Programmable Interval Timer

PIT (Programmable Interval Timer) је програмабилни тајмер који се може подесити тако да на тачно дефинисани период времена шаље захтев за прекидом. На тај начин може се мерити колико времена је протекло од почетка паљења рачунара или колико се тачно времена утрошило на извршавање неког програма.

src/c/timer.c:

Коришћене библиотеке:

```
#include<source/timer.h>
#include<types.h>
#include<asm.h>
#include<source/stdio.h>
```

Променљиве које чувају број откуцаја тајмера као и време протекло од паљења рачунара. Приметимо константу која дефинише колика је заправо учесталост овог тајмера.

```
uint32_t tick=0;
const uint32_t TICKS_PER_SECOND=50;
extern uint32_t time;
uint32_t time=0;
```

Функција која handle-ује прекид који тајмер упућује језгру. Ова фунцкија има задатак да ажурира време протекло од паљења рачунара.

```
void timer_handler(void)
{
    tick++;
    if(tick==TICKS_PER_SECOND)
    {
        tick=0;
        time++;
    }
```

```
    ioport_out(0x20, 0x20);
    ioport_out(0xa0,0x20);
}
```

Наше језгро иницијализује тајмер тако да има 20 откуцаја у секунди. PIT чип састоји се од осцилатора који ради на фреквенцији од 1.193182 MHz. Уколико ту вредоност поделимо са бројем жељених откуцаја у секунди, добићемо фреквенцију на којој чип треба да ради да би генерисао прекиде у жељеном временском интервалу.

```
void init_timer(uint32_t frequency)
{
    uint32_t divisor = 1193180 / frequency;
    ioport_out(0x43, 0x36);
```

```
    uint8_t l = (uint8_t)(divisor & 0xFF);
    uint8_t h = (uint8_t)( (divisor>>8) & 0xFF );
```

```
    ioport_out(0x40, l);
    ioport_out(0x40, h);
}
```

7.8 Heap

Heap за ово језгро користи алгоритам[12] битмапе да би на најефикаснији начин сачувало податке о тренутно коришћеним блоковима. Хип нам служи да бисмо динамички алоцирали меморију, тј. да бисмо могли да користимо глобалне променљиве, као и променљиве чији животни век мора бити дуготрајнији од функције у којој је та променљива настала.

src/c/heap.c:

```
#include<source/heap.h>
#include<types.h>
```

```
void k_heapBMInit(KHEAPBM *heap)
{
    heap->fblock = 0;
}
```

```
int k_heapBMAddBlock(KHEAPBM *heap, uintptr_t addr, uint32_t size, uint32_t
        bsize)
{
    KHEAPBLOCKBM        *b;
    uint32_t            bcnt;
    uint32_t            x;
    uint8_t             *bm;
```

```
    b = (KHEAPBLOCKBM*)addr;
    b->size = size - sizeof(KHEAPBLOCKBM);
    b->bsize = bsize;
```

```
    b->next = heap->fblock;
    heap->fblock = b;
```

```
    bcnt = b->size / b->bsize;
    bm = (uint8_t*)&b[1];
```

```
    /* clear bitmap */
    for (x = 0; x < bcnt; ++x) {
        bm[x] = 0;
    }
```

```
    /* reserve room for bitmap */
    bcnt = (bcnt / bsize) * bsize < bcnt ? bcnt / bsize + 1 : bcnt / bsize;
    for (x = 0; x < bcnt; ++x) {
        bm[x] = 5;
    }
```

```
    b->lfb = bcnt - 1;
    b->used = bcnt;
    return 1;
}
```

```
static uint8_t k_heapBMGetNID(uint8_t a, uint8_t b);
static uint8_t k_heapBMGetNID(uint8_t a, uint8_t b)
{
    uint8_t         c;
    for (c=a+1;c==b||c==0;++c);
    return c;
}
```

```
void *k_heapBMAlloc(KHEAPBM *heap, uint32_t size)
{
    KHEAPBLOCKBM    *b;
    uint8_t         *bm;
    uint32_t        bcnt;
    uint32_t        x, y, z;
    uint32_t        bneed;
    uint8_t         nid;
```

```
    /* iterate blocks */
    for (b = heap->fblock; b; b = b->next)
    {
        /* check if block has enough room */
        if (b->size - (b->used * b->bsize) >= size)
        {
```

```
            bcnt = b->size / b->bsize;
            bneed = (size / b->bsize) * b->bsize < size ? size / b->bsize + 1 :
                size / b->bsize;
            bm = (uint8_t*)&b[1];
```

```
            for (x = (b->lfb + 1 >= bcnt ? 0 : b->lfb + 1); x != b->lfb; ++x)
            {
                /* just wrap around */
                if (x >= bcnt) x = 0;
```

```
                if (bm[x] == 0)
                {
                    /* count free blocks */
                    for (y = 0; bm[x + y] == 0 && y < bneed && (x + y) < bcnt;
                            ++y);
```

```
                    /* we have enough, now allocate them */
                    if (y == bneed)
                    {
                        /* find ID that does not match left or right */
                        nid = k_heapBMGetNID(bm[x - 1], bm[x + y]);
```

```
                        /* allocate by setting id */
                        for (z = 0; z < y; ++z) bm[x + z] = nid;
```

```
                        /* optimization */
                        b->lfb = (x + bneed) - 2;
```

```
                        /* count used blocks NOT bytes */
                        b->used += y;
```

```
                        return (void*)(x * b->bsize + (uintptr_t)&b[1]);
                    }
```

```
                    /* x will be incremented by one ONCE more in our FOR loop
                     * */
                    x += (y - 1);
                    continue;
                }
            }
        }
    }
    return 0;
}
```

```
void k_heapBMFree(KHEAPBM *heap, void *ptr)
{
    KHEAPBLOCKBM    *b;
    uintptr_t       ptroff;
    uint32_t        bi, x;
    uint8_t         *bm;
    uint8_t         id;
    uint32_t        max;
```

```
    for (b = heap->fblock; b; b = b->next)
    {
        if ((uintptr_t)ptr > (uintptr_t)b && (uintptr_t)ptr < (uintptr_t)b +
                sizeof(KHEAPBLOCKBM) + b->size) {
            /* found block */
            ptroff = (uintptr_t)ptr - (uintptr_t)&b[1];
/* get offset to get
                                                        block */
            /* block offset in BM */
            bi = ptroff / b->bsize;
            /* .. */
            bm = (uint8_t*)&b[1];
            /* clear allocation */
            id = bm[bi];
            /* oddly.. GCC did not optimize this */
            max = b->size / b->bsize;
            for (x = bi; bm[x] == id && x < max; ++x) bm[x] = 0;
            /* update free block count */
            b->used -= x - bi;
            return;
        }
    }
```

```
    /* this error needs to be raised or reported somehow */
    return;
}
```

```
KHEAPBM kheap;
```

```
void kheapinit()
{
    k_heapBMInit(&kheap);
}
```

```
int kheapaddblock(uintptr_t addr,uint32_t size,uint32_t bsize)
{
    return k_heapBMAddBlock(&kheap,addr,size,bsize);
}
```

```
void *kmalloc(uint32_t size)
{
    return k_heapBMAlloc(&kheap,size);
}
```

```
void kfree(void *ptr)
{
    k_heapBMFree(&kheap, ptr);
}
```

7.9 Paging

Страничење је систем који дозвољава сваком процесу да користи цео простор виртуелне меморије, без да је толико меморије за сваки процес физички доступно. 32-битни процесори подржавају 32-битне вирутелне адресе и 4GB виртуелне меморије.

Поред овога, страничење има и предност заштите на нивоу странице. У овом систему, кориснички процеси могу да виде и измене само податке који се налазе на сопственом простору адресе, обезбеђујући изолацију на хардверском нивоу између процеса.

src/c/paging.c:

Коришћене библиотеке:

```
#include<source/paging.h>
#include<types.h>
```

Асемблерске функције које учитавају табелу која одређује која виртуелна адреса се односи на коју физичку, а затим укључују страничење.

```
extern void loadPageDirectory(uint32_t*);
extern void enablePaging(void);
```

Низ у коме се чувају директоријуми страница. Сваки директоријум може да садржи 1024 табеле страница.

```
uint32_t page_directory[1024] __attribute__((aligned(4096)));
```

Функција која поставља вредности свих директоријума страница који се не користе на константу која одређује да ли је директоријум у употреби и да ли уопште садржи табеле страница.

```
void set_pd(void)
{
    //set each entry to not present
    for(size_t i=0;i<1024;i++)
    {
        // This sets the following flags to the pages:
        //   Supervisor: Only kernel-mode can access them
        //   Write Enabled: It can be both read from and written to
        //   Not Present: The page table is not present
        page_directory[i] = 0x00000002;
    }
}
```

Низ у коме се чувају табеле страница.

```
uint32_t page_table[1024][1024] __attribute__((aligned(4096)));
```

Функција која поставља вредности једне табеле страница на одређени опсег у физичкој меморији додељен датом процесу.

```
void set_pt(size_t num,uint32_t address)
{
    // holds the physical address where we want to start mapping these pages
    // to.
    // in this case, we want to map these pages to the very beginning of
    // memory.
```

```
    //we will fill all 1024 entries in the table, mapping 4 megabytes
    for(size_t i=0;i<1024;i++)
    {
        // As the address is page aligned, it will always leave 12 bits zeroed.
        // Those bits are used by the attributes ;)
        page_table[num][i] = (address + i * 0x1000) | 3;
        // attributes: supervisor level, read/write, present.
    }
```

```
    page_directory[num] = ((uint32_t)page_table[num]) | 3;
    // attributes: supervisor level, read/write, present
}
```

Главна функција која мапира свих 4GB физичке меморије на виртуелну меморију са истом адресом, а затим укључује страничење.

```
void set_paging(void)
{
    set_pd();
    for(size_t i=0;i<1024;i++) set_pt(i,0x00400000 * i); // all 4GB mapped
    loadPageDirectory(page_directory);
    enablePaging();
}
```

7.10 Минимална C biblioteka

Због већ до сада велике опширности рада, овде ће бити приказана само минимална C библиотека потребна за компајловање језгра оперативног система. Приметимо да неки од фајлова морају само постојати, тј. могу бити и празни. Ова библиотека осим постојања одређених header фајлова подразумева и прототипе функција и структура без којих се неће компајловати.
src/include/errno.h:

src/include/stdio.h:

```
#ifndef _STDIO_H
#define _STDIO_H

#include <stdarg.h>
#include <stddef.h>

#define SEEK_SET 0
typedef struct { int unused; } FILE;

extern FILE* stderr;
#define stderr stderr
int fclose(FILE*);
int fflush(FILE*);
FILE* fopen(const char*, const char*);
int fprintf(FILE*, const char*, ...);
size_t fread(void*, size_t, size_t, FILE*);
int fseek(FILE*, long, int);
long ftell(FILE*);
size_t fwrite(const void*, size_t, size_t, FILE*);
void setbuf(FILE*, char*);
int vfprintf(FILE*, const char*, va_list);

#endif
```

src/include/stdlib.h:

```
#ifndef _STDLIB_H
#define _STDLIB_H

void abort(void);
int atexit(void (*)(void));
int atoi(const char*);
void free(void*);
char* getenv(const char*);
void* malloc(size_t);

#endif
```

src/include/string.h:

```
#ifndef _STRING_H
#define _STRING_H

#include <stddef.h>

void* memcpy(void*, const void*, size_t);
void* memset(void*, int, size_t);
char* strcpy(char*, const char*);
size_t strlen(const char*);

#endif
```

src/include/time.h:

```
```

src/include/unistd.h:

```
#ifndef _UNISTD_H
#define _UNISTD_H

#include <sys/types.h>

int execv(const char*, char* const[]);
int execve(const char*, char* const[], char* const[]);
int execvp(const char*, char* const[]);
pid_t fork(void);

#endif
```

src/include/sys/types.h:

```
#ifndef _SYS_TYPES_H
#define _SYS_TYPES_H

typedef int pid_t;

#endif
```

7.11 Помоћна C biblioteka

src/include/types.h:

```
#ifndef SOURCE_TYPES_H
#define SOURCE_TYPES_H

#include<stdbool.h>
#include<stddef.h>
#include<stdint.h>

#endif
```

src/include/asm.h:

```
#ifndef SOURCE_ASM_H
#define SOURCE_ASM_H

#include<types.h>

#define PIC1_COMMAND_PORT 0x20
#define PIC1_DATA_PORT 0x21
#define PIC2_COMMAND_PORT 0xA0
#define PIC2_DATA_PORT 0xA1


extern uint8_t ioport_in(uint8_t port);
extern void ioport_out(uint8_t port, int data);


#endif
```

7.12 Главна функција

Главна функција којој се предаје управљање у src/as/boot.s асемблерском фајлу који смо имали прилике да видимо на почетку. Функција обједињује све функцоналности које смо до сада имплементирали.
src/c/kernel.c:

```
#include<source/kernel.h>
#include<source/paging.h>
#include<source/idt.h>
#include<source/timer.h>
#include<source/keyboard.h>
#include<source/heap.h>
#include<source/vga.h>
#include<source/tty.h>


void kernel_main(void)
{
    set_paging();
    init_idt_table();
    init_timer(50);
    init_keyboard();
    kheapinit();
    kheapaddblock(0x00200000, 0x00100000, 16);

    terminal_initialize();
    prompt();

    while(1) __asm__("hlt\n\t");
}
```

8

Закључак

Данас, када постоје многобројни виши програмски језици, креирање оперативног система је врло тежак изазов за који се врло мало људи опредељује. Међутим, за мене, ово је једно врло охрабрујуће искуство са којим тек неколицина може да се похвали. Желео бих да у будућности примарно програмирам у нижим програмским језицима као што су Assembler и C, просто из разлога што сматрам да је то једини начин да се на дуже стазе извојева победа по питању перформанси у односу на неке апстрактније програмске језике. Осећај који доноси чињеница да се мој код извршава директно на хардверу без било каквог посредника је мени од непроценљивог значаја.

Да бих одважио читаоце, као и све друге које ова тема занима, да се и сами опробају у изради оперативног система, све изворне фајлове овог пројекта објавио сам под GPLv3 лиценцом, као и све своје досадашње пројекте, на страници `https://github.com/aleksav013`. Сви моји пројекти су open source што значи да код може послужити свакоме ко жели да сазна нешто више.

У току израде суочио сам се са многобројним проблемима. Најтежи део је био испловљавање и започињање ове експедиције. Море информација и стандарда који су морали бити задовољени само да би се систем покренуо, а затим и исписао неку корисну информацију, а камоли нешто више. Најтеже је било то што није било могуће додати било какву функционалност без да се у потпуности разуме свака линија кода. Предуслов за сваку, макар и најситнију измену, био је исцрпно и до детаља ишчитати документацију за предвиђену област, ако та документација уопште и постоји.

Овај пројекат је био сјајан показатељ колико је заправо комплексна израда језгра оперативног система који треба да представља мост између хардвера и софтвера. Драго ми је што сам одабрао овако тежак пројекат за матурски рад из разлога што ми је то помогло да пробијем баријеру и уложим пуно труда да бих заправо разумео како раде оперативни системи и колико је софистициран њихов дизајн.

Изабрао сам ову тему из разлога што истовремено повезује Linux и програмирање у C програмском језику, моје тренутно две омиљене ствари, на предлог мог ментора. Искористио бих ову прилику да се захвалим професору Милошу Арсићу, као и професору Филипу Хаџићу на сјајним идејама и усмеравању приликом израде овог матурског рада, као и свим другарима, професорима и породици који су ме подржали.

Литература

[1] Intel. *Intel® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[2] Intel. *System V Application Binary Interface AMD64 Architecture Processor Supplement*. URL: `https://www.intel.com/content/dam/develop/external/us/en/documents/mpx-linux64-abi.pdf`.

[3] Robert Mecklenburg. *Managing Projects with GNU Make*. 3rd ed. Nutshell Handbooks. O'Reilly Media, 2004. ISBN: 0596006101,9780596006105.

[4] Arnold Robbins, Elbert Hannah, and Linda Lamb. *Learning the Vi and Vim Editors*. 7th ed. O'Reilly Media, 2008. ISBN: 9780596529833,059652983X.

[5] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. 3rd ed. Pearson, 2006. ISBN: 0131429388, 9780131429383.

[6] Andrew S. Tanenbaum. *Modern Operating Systems*. 3rd ed. Pearson Prentice Hall, 2008. ISBN: 0136006639,9780136006633.

[7] Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. 6th ed. Prentice Hall, 2012. ISBN: 0132916525,9780132916523.

[8] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. 2nd ed. Addison-Wesley Professional, 2014. ISBN: 0321968972,9780321968975.

[9] Bill Rosenblatt and Cameron Newham. *Learning the bash shell*. 3rd ed. Nutshell handbook. O'Reilly, 2005. ISBN: 9780596009656,0596009658.

[10] Samuel P. Harbison and Guy L. Steele. *C: A Reference Manual*. 5th ed. Prentice Hall, 2002. ISBN: 013089592X,9780130895929.

[11] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language*. 2nd ed. Prentice Hall, 1988. ISBN: 0131103628,9780131103627.

[12] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. 1st ed. McGraw-Hill Science/Engineering/Math, 2006. ISBN: 0073523402,9780073523408.

[13] *OsDev Wiki*. URL: `https://wiki.osdev.org/Expanded_Main_Page`.

[14] David Salomon. *Assemblers and loaders*. Ellis Horwood series in computers and their applications. Ellis Horwood, 1992. ISBN: 9780130525642,0130525642.

[15] Alfred V. Aho et al. *Compilers - Principles, Techniques, and Tools*. 2nd ed. Pearson/Addison Wesley, 2006. ISBN: 0321486811.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th ed. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2011. ISBN: 012383872X,9780123838728.

[17] Bryant Randal E. and O'Hallaron David Richard. *Computer Systems: A Programmer's Perspective*. 2nd ed. Pearson, 2010. ISBN: 0136108040,9780136108047.

[18] Benjamin Lunt. *USB: The Universal Serial Bus (FYSOS: Operating System Design Book 8)*. 2013. ISBN: 1717425364.

[19] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating system concepts*. 9th ed. Wiley, 2012. ISBN: 9781118063330,1118063333.

[20] William Stallings. *Operating systems: internals and design principles*. 7th ed. Prentice Hall, 2011. ISBN: 013230998X, 9780132309981, 0273751506, 9780273751502, 1299318266, 9781299318267.

[21] Remzi H. Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Paperback. Createspace Independent Publishing Platform, 2018. ISBN: 198508659X,9781985086593.

[22] Stephen A. Rago and W. Richard Stevens. *Advanced Programming in the UNIX Environment*. 3rd ed. Addison-Wesley professional computing series. Addison-Wesley Professional, 2013. ISBN: 0321637739,9780321637734.