**МАТЕМАТИЧКА ГИМНАЗИЈА**

**У БЕОГРАДУ**

**МАТУРСКИ РАД**

**из предмета**

**Рачунарства и информатике**

на тему:

Основе оперативних система

Ментор

Никола Тасић

Ученик

Борис Ћеранић

**Београд, Мај 2023.**

# **Увод**

Технолошка развића донеле су многе промене у живот просечног човека, само дигитални уређаји трансформисали су начине на које комуницирамо, баратамо информацијама, складиштимо драгоцена сећања у виду слика и бивамо продуктивни у друштву. Још од почетка друге половине 20-ог века тече непрекидан развој рачунара, та категорија се касније шири на мобилне телефоне, конзоле за игрице, паметне фрижидере... Данас је опус разноврсности дигиталних уређаја готово непојмљив једном човеку, они се увлаче у сваки аспект људског живота. Свим тим уређајима је потребно нешто да их управља, неки софтвер, преферабилно што стандардизованији могући. У супротном би завладао хаос, за сваки уређај од њих бесконачно би било потребно писати различиту верзију софтвера. Тај проблем решавају, или барем ублажавају оперативни системи.

# Оперативни систем

Оперативни систем је софтвер или програм који контролише операције компјутерског система и његових ресурса. Осим тога, постоји једна веома важна критерија која је заједничка за све оперативни системе, а то је, да је оперативни систем способан да учитава и извршава програме независно од хардвера (уређаја) на коме се налазе док им омогућава стандардизован интерфејс за улаз и излаз. Неке од главних функција које опепративни систем може имати су:

* Управљање меморијом, уређајима за складиштење и другим системским ресурсима
* Одржавање реда и спровођење сигурносних полиса другим програмима
* Извршавање више програма истовремено, одређивање редоследа и важности извршавања,

давајући им одговарајућу предност – multitasking

* Динамичко покретање и гашење других програма
* Обезбеђење основног корисничког интерфејса као и *API*-ова

Не подржава сваки оперативни систем све ове функције. *MS-DOS*, на пример, није могао да да извршава више програма од једном, био је *single-tasking*. Важно је такође напоменути шта оперативни системи нису:

* Комјутерски хардвер
* Нека специфична апликација као што су игрице, процесори текста и *Web browser*-и.
* Колекција алатки попут *GNU*
* Окружење за развој софтвера, додуше поједији, попут *UCSD Pascal*, поседују интегрисана окужења
* Графички корисники интерфејс (GUI) без обзира на то што већина оперативних система долази са једним

# Историјат

# Историјат оперативних система може се трагати до раних деценија 1950-их година, када су рачунари почели да се развијају. У том периоду, рачунари су били гломазни, скупи и сложени системи којима је било потребно управљати на ефикасан начин.

# Један од првих оперативних система био је General Motors Research Laboratories Operating System (GM-NAA I/O) који је развила компанија IBM за General Motors у 1956. години. Он је имао основне функционалности за управљање улазно-излазним операцијама и алокацију ресурса.

# Касније, у 1960. години, IBM је представио оперативни систем OS/360 за своје рачунаре. Овај систем је био значајан јер је пружао комплетну инфраструктуру за рад са рачунарима, укључујући систем за прекиде, улазно-излазне операције и меморијске управљачке функције.

# У наредним деценијама, оперативни системи су се развијали и усавршавали. У 1970-им годинама, Бел Лабс је развио оперативни систем UNIX који се и данас користи. UNIX је био отворени систем, што је омогућило његово широко распрострањење и употребу на различитим рачунарским платформама.

# Следећа значајна промена у оперативним системима дошла је са појавом персоналних рачунара у 1980-им годинама. Микрософт је представио MS-DOS, оперативни систем заснован на командној линији, који је постао широко коришћен на PC рачунарима.

# У 1990-им годинама, Микрософт је лансирао Windows оперативни систем који је имао графички кориснички интерфејс и омогућавао једноставније и интуитивније коришћење рачунара. Ово је довело до експлозије примене рачунара у различитим сферама, укључујући и домаће кориснике.

# Касније, оперативни системи су се развијали и пристосавали потребама нових технологија. Данас, оперативни системи као што су Windows, macOS и Linux пружају бројне функционалности, укључујући мрежне везе, мултимедијалне могућности и подршку за различите апликације.

# Оперативни системи су проследили пут развоја рачунара и играли кључну улогу у олакшавању коришћења и управљања рачунарима. Кроз иновације и напредак у технологији, оперативни системи настављају да се развијају и у будућности ће наставити да обликују начин на који користимо рачунаре.

# Методологија и опсег

У опсег овог рада спадају, поводом ограниченог времена, само неки аспекти оперативних система. Ради дубљег и лакшег разумевања рад ће бити обрађен из перспективе прављења једног врло елементарног оперативног система. То потенцијано укључује коришћење стандарда и пракси које су се користиле у прошлости, 90-их година прошлог века, а сада само заузимају место на старим интернет форумима или чак у музејима. Новији стандарди и праксе су много компликованији, али раде на сличном или истом принципу, чиме се тематика овог рада не нарушава.

## Boris-toolchain

Сходно

## SmallerC

Smaller C је једноставан и мали single-pass C компајлер, тренутно подржава већину фукција C језика између *C89* и *C99* стандарда. *Self-host*-ован је на оперативним системима DOS, Windows и Linux, тиме може и да се покрене на њима и да компајлује програме за њих. Погодан је за писање оперативних система у различитим модовима процесора попут *Real Mode*, *Virtual 8086 Mode*, *Unreal Mode* и *32-bit Protected mode.*

Компајлер је праћен Preprocessor-ом и Linker-ом који може да произведе програмске фајл формате попут *COM* и *MZ* који користи *DOS, PE* који користи *Windows* и *ELF* који користи *Linux* оперативни систем. Подржава и стандардну C библиотеку написану за ове оперативне системе. За правилно функционисање му је потребан екстерни асемблер. Направљен је од стране *alexfru*-а и може се видети на GitHub-у путем [линка](https://github.com/alexfru/SmallerC).

Овај компајлер је поприлично згодан за писање једноставних оперативних система тиме што подржава *Unreal Mode* и поприлично је, по својој сврси, лаган и ненабуџен. Мане су додуше то што је врло слабо документиран као и недостатак људи који га користе чиме је страна подршка и помоћ страшно лимитирана.

## *NASM*

The Netwide Assembler- асемблер и дисасемблер намењен за Intel x86 архитектуру рачунара. Користи се за писање *16-Bit*, *32-Bit* и *64-Bit* програма. Оригинално написан од стране Симона Татхама. Подржава више излазних формата као *ELF* и .bin који се често користи за писање оперативних система.

Ми ћемо га већински користити индиректно преко *SmallerC*-а, а и помало директно где нам је потребан већи степен контроле поготово при првобитном покретању оперативног система. Директно написан assembly код биће примарно 16-Bit-ни.

## QEMU

QEMU је open-source емулатор машина и виртуализатор. Може се користити више сврха али се најчешће користи за емулацију система где омогућава виртуелно окружење као читав рачунар (процесор, меморија, периферни уређаји) за покретање оперативног система. Користи се помоћу обезбеђених команди за терминал. Подржава различите архитектуре емулације као ARM, MIPS, RISC-V и наравно x86.

Ми ћемо користити x86 емулацију, примарно због приступачности и брзине откривања грешака у односу да покретање оперативног система на правој машини.

## *Make*

Make је GNU алатка која је намењена за контролу генерације извршних фајлова и осталих пропратних фајлова приликом компилације и прављења програма. Може да се користи и за покретање направљених програма. Користи такозване *Makefile*-ове да би одредио редослед баратања фајлова као и који фајлови су потребни у ком тренутку.

Један *Makefile* садржи обично више правила које изгледају овако:

target: dependencies ...

commands

...

Где су *dependencies* target-и потребни да би се извршило правило, а commands команде које ће правило покренути.

(дати пример стрвари)

# x86 Архитектура – кратак потсетник

x86 је фамилија архитектура развијена од стране Интел-а базирана на 8086 микропроцесору. 8086 је настао као *16-Bit*-на екстензија *8080* *8-Bit*-ног микропроцесора. Касније, 1985. настаје *80386* са 32-bit-ном величином регистра.

У периоду од 1999. до 2003. *AMD* развија архитектуру са величином регистра од *64-Bit*-а, она се често води под називом x86\_64 или amd64.

Ми ћемо се фокусирати на *32-Bit*-ну x86 архитектуру. Прецизније на *i386*, oва верзија базирана је на Интел-овом *80386* микропроцесору где број 3 означава трећу генерацију архитектуре.

## Регистри процесора

Процесор садржи више типова регистара. Неки регистри који су мањи од 32-Bit-а могу се такође адресирати. Они су остатак претходних верзија архитектуре, упркос томе могу често могу бити корисни.

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 | rbp |

EFLAGS регистар

Сегментни регистри:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| cs | Ds | ss | es | fs | gs |

Контролни регистри:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| cr0 | cr2 | cr3 | cr4 | cr8 |

Системски регистри (показивачи на табеле):

|  |  |  |
| --- | --- | --- |
| gdtr | ldtr | idtr |

Почев од 16-Bit-не x86 архитектуре, могу се адресирати и горње половине (од 9. до 16. Bit-а) регистара ax, bx, cx и dx редом под именом ah, bh, ch и dh. У наредној табели је приказана однос поменутих регистара као и које битове они адресирају:

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

У адресирању делова регистара могу помоћи такође *bitwise* логичкеоперације, на пример *ah*, *ax&0xFF00,* *eax&0x0000FF00* и *rax&0x000000000000FF00* се сви односе на исто.

### Регистри опште намене

Регистри опште намене имају улогу да чувају операнде и показиваче:

* Операнде за логичке и аритметичке операције
* Операције калкулацију адресирања
* Показиваче на локацију у меморији

Регистри опште намене се, као што назив су сугерише, могу користити поизвољно према потреби. Међутим, дизајнери хардвера су увидели могућност даљих оптимизација у томе да сваком регистри додадтно доделе неку специфичну улогу.

* *eax* – акумулатор за операнде и резултате неких операција
* *ebx* – показивач на податке у *ds* сегменту
* *ecx* – бројач петље и операције над стринговима
* *edx* – показивач на улаз и излаз
* *esi* – показивач на податке у сегменту ds регистра, изворни показивач за операције над стринговима
* *edi* – показивач на податке у сегменту es регистра, крајњи показивач за операције над стринговима
* *esp* – показивач на врх стека
* *ebp* – показивач на део стека, често одређује тренутни stack-frame

### EFLAGS регистар

EFLAGS регистар

Садржи заставице о тренутном стању процесора као и резултате неких операција

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Име | ID | VIP | VIF | AC | VM | RF | R | NT | IOPL | | OF | DF | IF | TF | SF | ZF | R | AF | R | CF | R | CF |

* CF – Carry flag
* PF – Parity flag
* AF – Auxiliary flag
* ZF – Zero flag
* SF – Sign flag
* TF – Trap flag
* IF – Interrupt enable flag
* DF – Direction flag
* OF – Overflow flag
* IOPL – I/O privilege level flag
* NT – Nested task flag
* RF – Resume flag
* VM – Virtual 8086 mode flag
* AC – Alignment check
* VIF – Virtual interrupt flag
* VIP – Virtual interrupt pending flag
* ID – CPUID instruction flag

Битови који нису приказани или они који су обележени са R су резервисани. Наведене су само неке од неких функција застава, различите инструкције их различито користе.

### Сегментни регистри

Сегментни регистри су *16-Bit*-ни показивачи на сегмент у меморији. Да би смо радили на неком делу меморије, у сегментним ретистрима морамо да имамо одговарајући сегментни селектор(који показује на тај део меморије.

* *cs* – *code segment* садржи селектор сегмента који показује на сегмент у коме је код тј. инструкције које се извршавају
* *ds* – *data segment* садржи селектор сегмента података, сличну функцију врше и *es*, *fs* као и *gs*
* *ss* – *stack segment*, показује на сегмент у коме је тренутни стек, будући се овај регистар може мењати подржано је да програми баратају са више стекова и да мењају између њих.

## Оперативни режими процесора

Неки од оперативних режима процесора у x86 архитектуруи су:

* Real mode
* Protected mode
* Virtual 8086 mode
* Unreal mode

### Real Mode

Реални режим је једноставни 16-Bit-ни режим подржан од стране свих x86 процесора. Сваки x86 процесор се покреће у овом режиму ради компатибилности. И ако је режим у својој основи 16-Bit-ни коришћење 32-Bit-них регистара је дозвољено уз додатак одговарајућег префикса на инструкцију (0x66). Неке од предности овог режима су:

* BIOS преузима одговорност драјвера за контролу осталих уређаја и контролу прекида
* BIOS функције обезбеђују напредну колекцију *low-level API функција*
* Приступ меморији је бржи и једноставнији поводом недостатка *descriptor* таблица
* Нема сигурносних ограничења процесора на меморију и хардверски I/O

Овај режим, из перспективе конструкције оперативног система, има и мане:

* Доступно је само 1 MB меморије
* Адресирање више од 64kB меморије од једном је праћено честим променама селектора сегмента.

Овај режим ћемо користити само за покретање оперативног система, део BOOT, чиме ћемо што пре прећи у Unreal mode.

### Сегментација меморије

Сегментација меморије је решење на проблем када имамо више од 64kB меморије и 16-Bit-не регистре да њима адресирамо меморију. То се ради увођењем поменутих сегментних регистара где се у рачунању физичке адресе они множе са 16. Да би добили адресу коју којој желимо да приступимо на то треба да додамо одговарајући регистар за адресирање. То нам даје ефективно адресирање меморије од 20 Bit-ова тј. имамо приступ 2^20 = (nesto) B. На пример адресирање [ds:bx] би у облику физичке адресеизгледало као

(стави овде формулу)

Сегментацију користи реални режим и дозвољава следеће начине адресирања:

* [bx + val]
* [si + val]
* [di + val]
* [bp + val]
* [bx + si + val]
* [bx + di + val]
* [bp + si + val]
* [bp + di + val]
* [val]

Различите инструкције ће уз ове начине адресирања имплицирати различите селекторе сегмента:

mov [si], ax – ће подразумевати ds селектор сегмента тј. *ds:si*

mov es:[si], ax – екплицитно користи es селектор сегмента

cmpsb – инструкција ће поредити бајтове на *ds:si и* es:di и складиштити одговоарајући резултат у EFLAGS регистар

### Protected Mode

Заштићени режим је главни оперативи режим модерних Интел процесора још од 80286 (16-Bit-ног процесора). Максимална доступна меморија је 2^32 B тј. 4GB.

Мане из перспективе конструкције једноставног оперативног система овог режима су:

* Потреба за писање драјвера за сваки уређај који се користи
* Строга контрола меморије и I/O уређаја од стране процесора као и дељење програма по привилегијама на прстење

### Virtual 8086 Mode

Виртуелни 8086 режим је под-режим заштићеног режима где процесор емулира реални режим док је у заштићеном режиму. Сходно тиме се користи за покретање програма који подржавају само реални режим. Да би процесор био у овом режиму 17. Bit EFLAGS регистра мора да буде сетован(једнак 1).

### Unreal Mode

Нереални режим је варијанта реалног режима у коме су један или више *descpripor-*асегмента учитани са нестандардним верностима, попут 32-Bit-ни лимити који дозвољавају приступ целој меморији. Користи се у 80286 и каснијим x86 процесорима.

Компајлер који ми користимо подржава нереални режим, када SmallerC компајлује за нереални режим доста ствари попут far поинтера и far call-ова су олакшани. Из наше перспективе овај режим спаја предности реалног и заштићеног режима:

* BIOS функције стоје на располагању тако да је писање драјвера за већину једноставних ствари непотребно
* На располагању нам стоји цела меморија која може линеарно да се адресира за податке, а и код са употребом пар трикова
* Једноставније је

У нашем случају мењаћемо лимите само ds, es, fs и gs сегмената тако да буду максималне могуће веиличине тј. 4GB(2^32), док cs i ss, сегменти за код и стек, остају 64kB.

# **Пример основа оперативног система**

Оперативни систем је софтвер који решава један или више проблема, ти проблеми и потребе су већ претходно поменуте у раду. Консеквентно се тај проблем може решити на практично бесконачно много начина, опет логичким следом већина оперативних система има заједничке особине. У дањем раду биће изложен један начин решавања тог проблема, као и сличности тог решења са постојећим познатијим оперативним системима. Преглед функционисања BorisOS-а се налази на следећем графику. Корисно је поменути да је разумевање овог графика ”на прву руку” није обавезно и охрабрено је враћање на њега.

# MBR – boot

Када се рачунар покрене између осталог покреће се и BIOS. Шта BIOS тачно ради спада ван опсега овог рада, али за нас је важно да он учитава први сектор, један сектор је 512B, хард диска, USB stick-а или неког другог уређаја за складиштење. Тај сектор обично називамо *MBR***-** *Master Boot**Record*, он уобичајено садржи информације о партицијама уређаја за складиштење. Ради једноставности ми ћемо прескочити могућност више партиција на место *MBR*-a ставити само *boot sector.*

*Boot sector* се по правилу учитава на меморијску адресу 0x7C00 и предаје му се контрола.

*Boot sector* иначе само треба да садржи инструкције за дање покретање система, али по спецификацији *FAT16 file system* заузима део овог сектора, о њему ће бити више речено касније, за сада нам је само битно да по спецификацији првих пар бајтова садрже инструкцију за скакање на boot.

## Boot

По горе написаном контрола је предата boot-у. Он има следеће функције:

* Обезбеђење повољног окружења за Kernel
* Учитавање Kernel-а или његовог дела (и свега њему потребног) у меморију
* Снабдевање Kernel-а са свим информацијама потребним
* Трансфер контроле Kernel-у

Из спецификације x86 архитектуре може се закључити да boot ради у реалном режиму процесора, тиме има приступ BIOS-овим функцијама и ресурсима што знатно олакшава његову израду.

### Обезбеђење повољног окружења за Kernel

У нашем случају прво ћемо подесити стек и одмах ући у нереалан режим. Стек ћемо ради избегавања преклапања са другим деловима оперативног система и програма ставити у други сегмент тј. у интервал [0x10000,0xFFFF]. Пошто стек у x86 архитектури расте ”на доле” почетна вредности sp и ss су 0xFFF0 (стандардна је пракса да стек буде усклађен по 4 или 16 бајтова) и 0x1000. Овиме је стек има своју максималну могућу дужину да режим коме ћемо радити, нереални.

Даље да би се пребацили у нереални режим морамо прво да се пребацимо у заштићени. Пре тога морамо учитати *Global Descriptor Table* чији опис спада ван опуса овог рада и да привремено онемогућимо прекиде. Када у *Global Descriptor Table-у* променимо лимите враћамо се у ”реални режим” који је заправо сада нереални. Горе поменуто извршава следећи код:

(стави код за ово у текстбокс)

### Учитавање Kernel-a

За правилно објашњење овог дела потребно нам да знамо начин *функционисања FAT16 file system*-а. За сада је само битно да boot тражи фајл под називом ”*LOADER.SYS*” и да га учитава на адресу 0x500.

Важно је напоменути да је 512 бајтова врло мало простора па форсира нека нестандардна, чак и занимљива решења. Пример тога био би провера да ли се тренутни фајл који проверавамо зове ”*LOADER.SYS*”:

(убаци део комичног кода овде у текстбокс )

### Снабдевање Kernel-а са свим информацијама потребним

Због ограниченог времена и једноставности оперативног система ова функција није имплементирана, али би иначе требало да садржи податке о подизању и о диску на коме се оперативни систем налази.

### Трансфер контроле Kernel-у

Ово се у нашем случају ради far call-ом са фиксним параметрима

(текстбокс кал инструкције). Где је 20 подразумевани померај од почетка фајла за извршавање MZ фајлова када је *header* празан.

Лепо је овде напоменути да су обично boot и loader спојени у bootloader који учитава цео Kernel. У нашем случају boot учитава loader који има више функција од учитавања остатка Kernel-а, што ће бити описано у даљем раду.

# Kernel

## Loader

## Менаџер меморије

## IPC – међипроцеска комуникација

# User-space

## File system

## Shell