МАТЕМАТИЧКА ГИМНАЗИЈА
У БЕОГРАДУ

МАТУРСКИ РАД

из рачунарства и информатике:

# Основе оперативних система

Ученик:
Борис Ћеранић

Ментор:
Никола Тасић

Београд, Мај 2023.

# Садржај

# 1 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.

## 1.1 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 прорама 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 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 појединји, попут *UCSD Pascal*, поседују интегрисана окужења
- Графички корисници интерфејс (*GUI*) без обзира на то што већина оперативних система долази са једним

## 1.2 Историјат

Историјат оперативних система води порекло од 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 потрошачке кориснике.

Касније, оперативни системи су се развијали и прилагодили 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 подстицали 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.

# 2 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.

## 2.1 Boris-toolchain

У складу са опсегом овог рада биле су биране и алатке. Тако је одабран једноставан хоби C компајлер, најраспрострањенији асемблер приступачности ради. Због недостатка компатибилности компајлера и MacOS-а, на коме је овај рад делимично писан, било је потребно користи алатке за програмирање на даљину: SHH и Tunnels екстензије VS Code-а. Нажалост не постоји стандардан начин да се овај *toolchain* изгради.

## 2.2 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.

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 програмера који га користе чиме је страна подршка и помоћ страшно лимитирана.

## *2.3 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 шеснаестобитни.

## 2.4 QEMU

QEMU је open-source емулатор машина и виртуализатор. Може се користити разне сврхе 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.

## *2.5 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.

# 3 x86 Архитектура – кратак потсетник

x86 је фамилија архитектура развијена од стране Интел-а базирана на 8086 микропроцесору. 8086 је настао као шеснаестобитна екстензија *8080 осмобитног* микропроцесора. Касније, 1985. настаје *80386* са тридесетдвобитном 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 тридесетдвобитну x86 архитектуру. Прецизније на *i386*, ова верзија базирана је на Интел-овом *80386* микропроцесору где број 3 означава трећу генерацију архитектуре.

## 3.1 Регистри процесора

Процесор садржи више типова регистара. Неки регистри који су мањи од 32 бита 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-битне x86 архитектуре, могу се адресирати и горње половине (од 9. до 16. бита) 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* логичке операције, на пример *ah*, *ax&0xFF00, eax&0x0000FF00* и *rax&0x000000000000FF00* сви реферирају 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 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 | | NT | IOPL | | OF | DF | IF | TF | SF | ZF | | AF | | PF | | 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 функција застава, различите инструкције их различито користе.

### Сегментни регистри

Сегментни регистри су *шеснаестобитни* 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.

## 3.2 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 шеснаестобитни режим подржан од стране свих $\mathrm{x86}$ процесора. Сваки $\mathrm{x86}$ процесор се покреће у овом режиму ради компатибилности. И ако је режим у својој основи шеснаестобитни коришћење $32\text{-Bit}$-них регистара је дозвољено уз додатак одговарајућег префикса на инструкцију $(0\mathrm{x}66)$. Неке од предности овог режима су:

- $\mathrm{BIOS}$ преузима одговорност драјвера за контролу осталих уређаја и контролу прекида
- $\mathrm{BIOS}$ 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 $\mathrm{I/O}$

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

- Доступно је само $1\ \mathrm{MB}$ меморије
- Адресирање више од $64\mathrm{kB}$ меморије од једном је праћено честим променама селектора сегмента.

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

## Сегментација меморије

Сегментација меморије је решење на проблем када имамо више од $64\mathrm{kB}$ меморије и $16$-битне 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 којој желимо да приступимо на то треба да додамо одговарајући регистар за адресирање. То нам даје ефективно адресирање меморије од $20$ битова тј. имамо приступ $2\hat{}20 = 1048576\mathrm{B}{\sim}1\mathrm{MB}$ меморије. На пример адресирање $[\mathrm{ds{:}bx}]$ би у облику физичке адресе изгледало као

$$16 \cdot ds + bx = absolute\ address$$

Сегментацију користи реални режим и дозвољава следеће начине адресирања:

- $[\mathrm{bx} + \mathrm{val}]$
- $[\mathrm{si} + \mathrm{val}]$

- [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 – експлицитно 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 (шеснаестобитног 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. бит EFLAGS регистра мора да буде сетован (једнак 1).

# Unreal Mode

Нереални режим је варијанта реалног режима у коме су један или више *descriptor*-а сегмента учитани са нестандардним верностима, попут 32-битни 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.

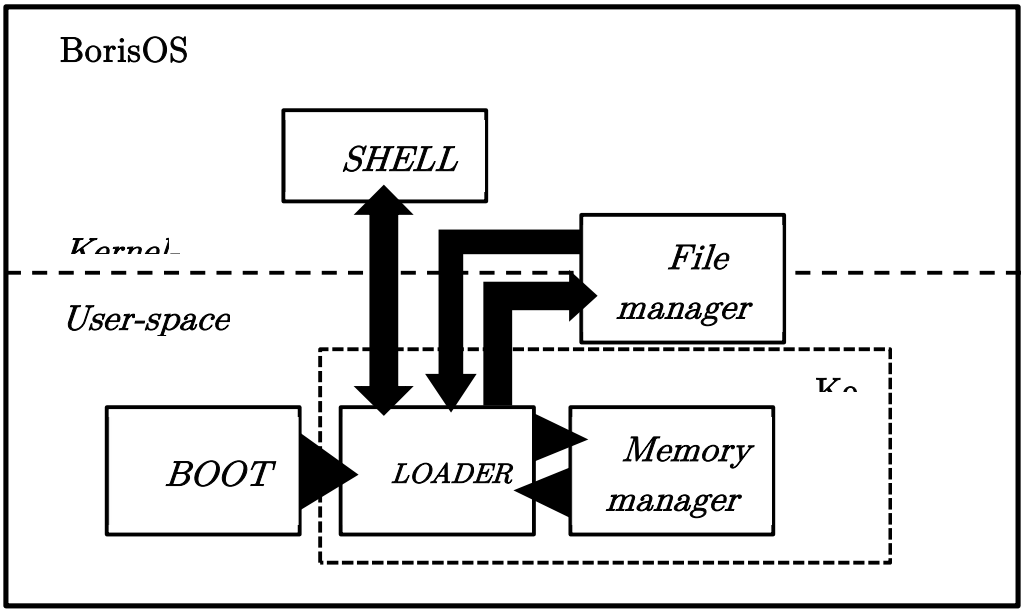
## 3.3 Прекиди

Прекиди (*Interrupts*) су знак процесору да је нешто искрсло и да је томе потребна неодложна пажња. Процесор затим престаје са оним чим је радио, чува своје стање, и посвећује пажњу извору прекида. Постоје три врсте прекида:

1. Изузеци - њих генерише сам процесор, они могу на пример обавештавати о некој грешци при рачунању
2. Хардверски - генеришу их екстерни уређаји, често од стране *chipset*-а
3. Софтверски - генерише их софтвер

# 4 Пример основа оперативног система

Оперативни систем је софтвер који решава један или више проблема претходно поменутих у раду. Иако се ти проблеми могу решити на практично бесконачно много начина, логичким следом већина оперативних система има заједничке особине. У даљем 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.




# 5 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 хард диска, USB stick-а или неког другог уређаја за складиштење. Један сектор је величине 512B. 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*-а 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. За сада нам је само битно да по спецификацији првих пар бајтова садрже инструкцију за скакање на boot.

## 5.1 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 потребним информацијама
- Трансфер контроле Kernel-у или његовом делу

Из спецификације x86 архитектуре може се закључити да boot ради у реалном режиму процесора, чиме 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 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*

```
   start:
       xor ax, ax

       xor ax,ax
       mov ds, ax
       mov ax, 0x1000
       mov ss, ax ; this can grow to 0x10000
       mov sp, 0xFFF0 ; this is arbitrary and the stack
could be whatever it wants

       cli
       push ds

       lgdt [gdtinfo]

       mov eax, cr0
       or al, 1
       mov cr0, eax
       jmp pmode
   pmode:
       mov bx, 0x10
       mov ds, bx

       and al, 0xFE
       mov cr0,eax
       jmp unreal
   unreal:
       pop ds
       sti
```

(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:

## Учитавање Kernel-а

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, услед чега настају нека нестандардна, чак и занимљива решења. Пример тога био би провера да ли се тренутни фајл који проверавамо зове "*LOADER.SYS*":

```
    ;------We check if its the loader directory entry------
-
    ;;we have it in memory at bx
    is_loader:
        cmp byte [bx],'L'
        jne no_loader_found_here
        cmp byte [bx+1],'O'
        jne no_loader_found_here
        cmp byte [bx+2],'A'
        jne no_loader_found_here
        cmp byte [bx+3],'D'
        jne no_loader_found_here
        cmp byte [bx+4],'E'
        jne no_loader_found_here
        cmp byte [bx+5],'R'
        jne no_loader_found_here
        cmp byte [bx+6],' '
        jne no_loader_found_here
        cmp byte [bx+7],' '
        jne no_loader_found_here
        cmp byte [bx+8],'S'
        jne no_loader_found_here
        cmp byte [bx+9],'Y'
        jne no_loader_found_here
        cmp byte [bx+10],'S'
        jne no_loader_found_here

        jmp found_loader

    ;-----------------------------------------------------
```

## Снабдевање Kernel-а са свим потребним информацијама

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

```
call (loader_memory_address/16):20
```

где је 20 подразумевани померај од почетка фајла за извршавање MZ фајлова када је *header* празан.

Користно је овде напоменути да су обично boot и loader спојени у boot-loader 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.

### 5.2 Подела *Kernel-space User-space*

Ради сигурности извршавање програма и меморју делимо на *Kernel-space* и *User-space.* Та подела обично почиње на нивоу хардвера имплементацијом прстења, где се у сваком тренуку рада процесора зна под којим прстеном оперира, тј. које привилегије име. Kernel-space спада у прстен 0 који има све привлиегије и готово неограничену контролу над хардвером, док User-space спада у 3. прстен и врло је ограничених могућонсти. Ту се примарно налазе програми написани од других људи који потенцијално немају најбоље намере. Због једноставности у нашем опреративном систему је подела виртуелна, а  једина разлика је како се програми третирају при покретању.

# 6 Kernel

*Kernel* је централни део оперативног система, и природна је последица потребе рачунарског система да управља ресурсима. Код једноставнијих оперативних система овај део посебно добија на значењу. Неке од важнијих функција *Kernel*-а могу бити*:*

- Менаџмент меморије и осталих периферних уређаја

- Покреће апликације и обезбеђујући им одговарајуће окружење
- Заштита програма и приступа
- Мултитаскинг

*Kernel*-и се могу поделити у неколико категорија:

## *6.1* Монолитски *Kernel*

Монолитски *Kernel*-и функционишу са целим *Kernel*-ом и драјверима у једном меморијском адресном простору тј. *kernel-space*-*у*. Овај приступ је у принципу веома ефикасан, јер је мењање контекста поприлично рачунски захтевна операција на x86 архитектури.

## *6.2* Микро *Kernel*

Микро *Kernel* тежи да што више функција и сервиса покреће у User-space-у. Тиме се доста добија на флексибилности јер *Kernel* није само један масиван програм којим је тешко баратати у смислу покретања и одржавања, као и у смислу писања и debug-овања. Као директну последицу добијамо и повећану стабилност и сигурност. Мана је што у доста случајева цена мењања контекста није вредна поменутих бенефиција.

## 6.3 Хибридни *Kernel*

За *Kernel* којим се ми бавимо је тешко рећи ког је типа, јер је превише мали и примитиван. Испољава неке карактеристике Микро *Kernel*-а тиме што раније и брже гура своје функције у User-space. Са друге стране подела User-space Kernel-space је овде врло апстрактна и виртуелна и цео оперативни систем се понаша као један програм чиме га можемо назвати и монолитским. Управо због оваквих примера постоји појам „хибридни *Kernel*“.

## 6.4 Loader

Често су термини boot и loader спојени у bootloader као једну целину чији је задатак да, слично горе наведеном, учита цео *Kernel,* претходно обезбедивши му повољно окружење. Овде ћемо узимати те појмове за одвојене. Штавише, као део овог решења на проблем оперативног система узећемо значење *Loader*-а у ширем смислу. Погодна је у овом тренутку *Kernel* поделити на две етапе: рани *Loader* и касни *Loader*. При томе треба пазити да је прелаз између ових етапа врло природан тј. не стриктно зацртан. Сходно са овим су функције раног *Loader*-а следеће:

- Системски позиви
- Учитавање остатка *Kernel*-а и оперативног система
- Учитавање системских позива као и њихово складиштење

Више о другој и трећој тачки биће објашњено у секцији Системски позиви и Дељене библиотеке.

Функције касног *Loader*-а се примарно ослањају на функције раног, утолико да су оне практично само унапређене верзије њихових претходника:

- Покретање User-space програма
- Динамично учитавање библиотека
- Складиштење меморијских места учитаних библиотека
- Давање потребних података User-space програмима

За потпуно разумевање наведених функција неопходне су информацике које се налазе у поглављима Системски позиви и дељене библиотеке и BEX формат.

# 7 Системски позиви и дељене библиотеке

## 7.1 Системски позиви

Системски позиви служе да се позове неки сервис *Kernel*-а из *User-space*-а. На пример, да се алоцира меморија или да се учита програм. Могуће методе да се имплеметирају могу бити:

- Прекиди
- Sysenter/Sysexit (Интел)
- Софтверски, far показивачима

Најчешћи начин да се имплеметирају системски позиви су прекиди, Linux користи прекид 0x80 у те сврхе. Мана овог приступа је што често, да би се писали хендлери, делови кода који обрађују прекид, морамо да користимо асемблер. То чини процес писања знатно тежим и мање флексибилнијим, али додуше бржим.

Интелови процесори од Pentium II подржавају специјалне инструкције за мењање контекста из извршавање системских позива, **sysenter** и **sysexit**. Овај метод имплеметације је доста комплексан и спада ван опсега овог рада.

Једноставан начин да се имплементирају системски позиви јесте као обичне C функције које неко може да позове ***far*** показивачем. Овај начин је додатно олакшан тиме што имамо глобални стек и тиме што немамо реалну поделу на User-space и Kernel-space.

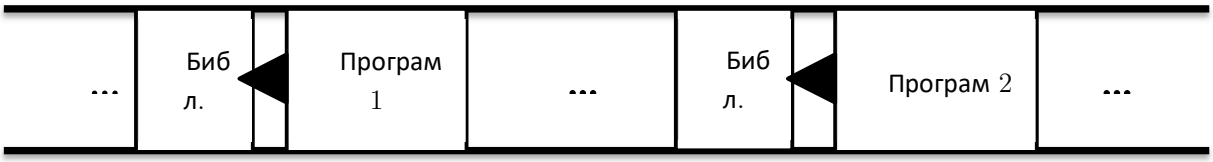
## 7.2 Дељене библиотеке

Концепт дељених библиотека тј. динамичког link-овања је поприлично једноставан и биће овде описан. Већини програма су потребне неке библиотеке, готово написан код да обавља неку функцију. Често се у пракси

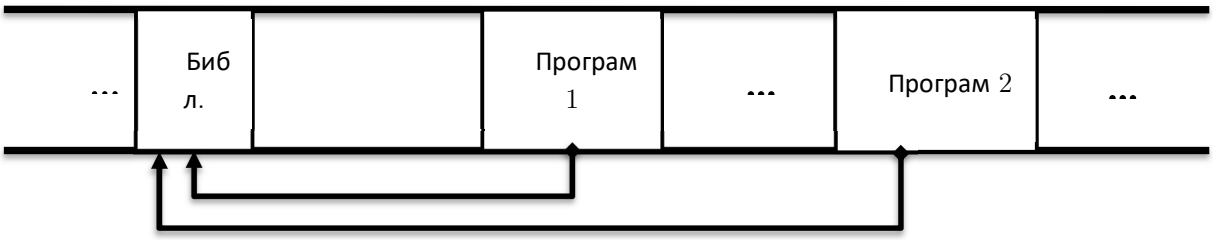
деси да је различитим програмима потребна једна те иста библиотека. Најједноставније решење је да се за сваки програм учита засебна инстанца те библиотеке чиме је она **статички link-ована** заједно са остаком програма.

Мало компликованије а и занимљивије решење јесте да приметимо да програми могу да деле библиотеке тј. да више програма може да користи једну инстанцу једне библиотеке које је само једном учитана у меморију тек када је била потребна неком програму. Овакво решење се зове **динамичко link-овање**. Сликовити приказ меморије може овде бити од помоћи:

*Статично link-овање*



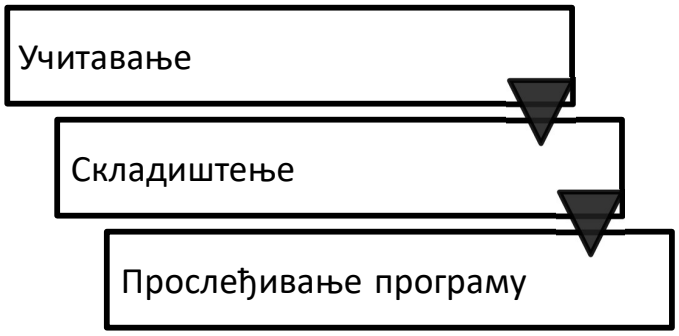

*Динамичко link-овање*




Овај приступ постаје компликованији чињеницом да немамо попуну контролу над *linker*-ом пошто користимо онај који је дошао уз компајлер, а прављење скроз новог *linker*-а спада ван опсега овог рада.

## 7.3 Тренутна имплементација

Тренутна имплементација ова два сервиса је суштински спојена у једно. Оба су обичне C функције које се могу позивати преко *far* показивача. То је урађено на следећи начин:




## Учитавање дељених библиотека и функција

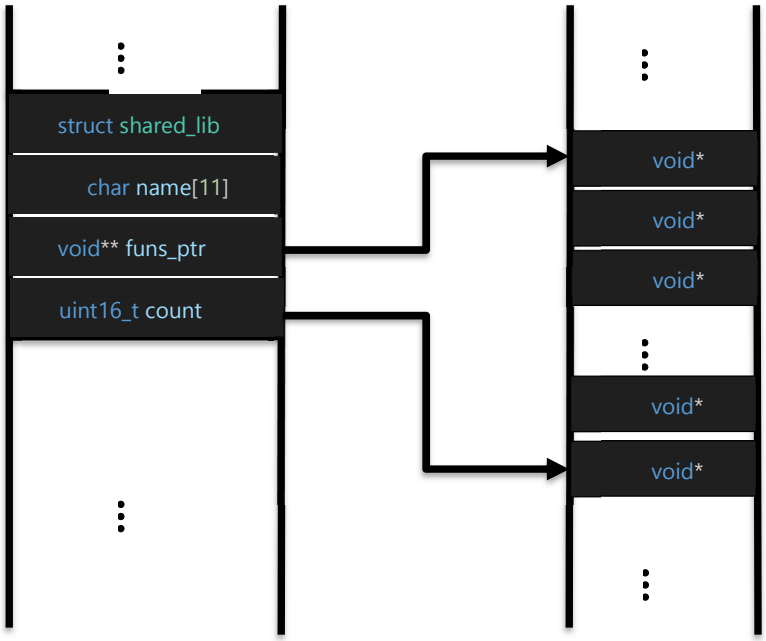
*Loader* у овој фази, као што назив сугерише, учитава потребне податке о програму да би се он користио као библиотека. Он ће, након учитавања програма наћи мапу програма која по спецификацији мора имати исти назив као програм осим што има екстензију *.MAP* и мора се налазити у истом фолдеру као и програм. Више о фолдерима биће у секцији *File system*. Битно је да се офсети који се налазе у мапи приликом учитавања додају на апсолутну адресу где је учитана библиотека, чиме добијамо апсолутну адресу функција тј. *far* показиваче.

## Складиштење дељених библиотека и функција

Складиштење локација библиотека ради се уз помоћ два низа. Један садржи структуре које су састава:

- Име *.MAP* фајла
- Показивач на почетак одговарајућих функција
- Број функција на који се показује

Други низ само садржи показиваче на функције. Битно је напоменути да су ово низови са фиксном величином, тиме смо у датом тренутку лимитирани да пратимо 64 **библиотека** и 256 **функција.**




## Прослеђивање програму

По BEX спецификацији која је већ описана *Loader* ће ишчитати библиотеке потребне програму. У зависности да ли је програм у *Kernel-space*-у или у *User-space*-у дешава се следеће:

1. У *Kernel-space*-у *Loader* има приступ само сопственим примитивним функцијама учитавања фајлова. Тиме може да учитава само из *root* директоријума. Ако не нађе потребне библиотеке односно програме и мапе, он ће пријавити грешку и неће учитати жељени *Kernel-space* програм
2. У *User-space*-у *Loader* већ има приступ напреднијим функцијама учитавања из *File manager*-а, сходно томе он ће сваку затражену библиотеку потражити у PATH директоријуму, где је за сада PATH само сачуван у коду *Loader*-а. Ако је нађе он ће пробати да је учита и покрене. Покретање програма успева само ако су успешна учитавања свих библиотека. **Важно је напоменути да превенција застоја и кружног учитавања није имплементирана, а при томе се мора водити рачуна о потребним библиотекама.**

Поводом већ поменуте лимитације коришћења готовог *linker*-а имплементација је захтевала креативна решења. Програму се прослеђује низ показивача који је састављен од показивача на низ функција из прве табеле. Приметимо да не морамо да прослеђујемо величину низа програму јер је она иста као и број тражених библиотека, који програм очигледно зна. Ово је такође низ фискне величине чиме смо ограничени на 128 **библиотека које можемо да проследимо,** то је веће од броја библиотека које тренутно можемо да пратимо али је тако због последица имплементације.

*Код за формирање низа код Kernel-space програма*

```
    void*** lib_store = &temp_sector; //where in the temp sector to put
the funcs
        for(char *lib_search= &(mz->lib_name); *lib_search;
lib_search+=16){//per BEX spec
            struct shared_lib* sh = get_shared_lib(lib_search);
            if(sh==NULL){
                return 2; // LIB NOT FOUND
            }
            *lib_store = sh->funs_ptr;
            lib_store++;

        }
```

Програм добија тако конструисан низ на стеку и статички *link*-ован код попут следећег би га дочекао:

*Иницијализатор link-овања, пример*

```
extern void start_program();

extern void init_MEMMNG(void** funcs);

void __start__(void*** libs){
    init_MEMMNG(libs[0]);
    start_program();
}
```

Као што видимо он прослеђује header фајлу који је статички *link*-ован заједно са програмом попут овог:

*Пример header фајла библиотеке*

```
#define func_num 3

typedef void* (*malloc_ptr)(size_t size);
malloc_ptr malloc;

typedef void (*dalloc_ptr)(uint32_t begin, size_t size);
dalloc_ptr dalloc;

typedef void (*memcpy_ptr)(void* src, void *dest, size_t n);
memcpy_ptr memcpy;

typedef void (*print_mem_ptr)();
print_mem_ptr print_mem;

void init_MEMMNG(void** funcs){
    malloc = (malloc_ptr)(funcs[0]);
    dalloc = (dalloc_ptr)(funcs[1]);
    memcpy = (memcpy_ptr)(funcs[2]);
    print_mem = (print_mem_ptr)(funcs[3]);
}
```

Овим приступом смо заобишли поменуте лимитације и одржали једноставност употребе библиотека приликом писања кода.

## 7.4 BEX формат

BEX је формат извршних фајлова и скраћеница је за Boris EXectutable. Састоји се од врло примитивне екстензије MZ извршног формата. Битно је имати у глави да ово поглавље не може бити скроз јасно док се не прочита одељак Системски позиви и дељене библиотеке.

# MZ извршни формат

МЗ овог назива представља потпис Микрософтовог инжењера Марка Збиковског. Дизајнирана је као преместиви извршни формат који би радио под реалним режимом процесора. Оригинално је направљен за MS-DOS и имао .EXE фајл екстензију. Разни формати попут NE, LE и PE који користи Windows имају уграђен у себи овај формат. Спецификација структуре фајла дата је табелом:

| Офсет | | Поље | Величина | Опис |
|---|---|---|---|---|
| 0 | 0x00 | Signature | реч | 0x5A4D (ASCII за слова 'M' и 'Z') |
| 2 | 0x02 | Extra bytes | реч | Број бајтова у последњем сектору. |
| 4 | 0x04 | Pages | реч | Број целих сектора |
| 6 | 0x06 | Relocation items | реч | Број уноса у таблици премештања |
| 8 | 0x08 | Header size | реч | Количина параграфа (дужина 16 бајтова) који заузима header |
| 10 | 0x0A | Minimum allocation | реч | Број параграфа који су неопходни да се алоцирају да би програм могао да се покрене. |
| 12 | 0x0C | Maximum allocation | реч | Број параграфа које пгограм тражи да се алоцирају |
| 14 | 0x0E | Initial SS | реч | Преместива вредност ss регистра, иницјални стек програма |
| 16 | 0x10 | Initial SP | реч | Преместива вредност sp регистра, иницјални стек програма |
| 18 | 0x12 | Checksum | реч | Када се ова реч дода на све остале речи фајла, резултат треба бити нула |
| 20 | 0x14 | Initial IP | реч | Иницијална вредност ip регистра |
| 22 | 0x16 | Initial CS | реч | Иницијална вредност cs регистра |
| 24 | 0x18 | Relocation table | реч | Апсолутни офсет таблице премештања |

| | | | | |
|---|---|---|---|---|
| 26 | 0x1A | Overlay | реч | Вредност која се користи за overlay management |
| 28 | 0x1C | Overlay information | N/A | Фајлови понекад садрже екстра информације potrebne за overlay managment |

Ради једноставности ми ћемо употребљавати само део горе наведених података, примарно за одређивање почека извршавања програма. Сходно тиме офсет почетка извршавања програма се може добити формулом:

$$16 * Header\ size\ +\ 16\ *\ Initial\ CS\ +\ Initial\ CS$$

То додато на апсолутну адресу на којој је програм учитан даје апсолутну адресу почетка извршавања.

Претходно је поменуто да је ово преместиви извршни формат. То значи да адреса на којој се програм учитава у меморију није фиксна, већ се може динамички мењати по потреби оперативног система или корисника.

У x86 архитектури да би програм могао да функционише свака инструкција која мења ток извршавања кода, попут jump, call и ret инструкција, мора да има апсолутну адресу на коју "скаче". То није тако једноставно када програм има могућност да се налази на различитим местима у меморији.

Решење на претходно поменути проблем су таблице премештаја( *relocation table*). Оне нам показују где се све налазе инструкције чије адресе на које "скачу" морамо да мењамо. То је обично приказано у релативном офсету у односу на почетак програма. Уобичајено је да део оперативног система који учитава програм, врло вероватно *Kernel,* пре извршавања али после учитавања програма у меморију погледа таблицу премештања и промени адресе, тј. дода апсолутну адресу почетка програма.

Ми на жалост нећемо користити таблицу премештања по MZ спецификацији, а алтернатива је представљена у следећој секцији.

## Лимитације *linker*-а

Због одабраних алатки, посебно *linker*-а који долази уз компајлер који користимо, морамо занемарити таблицу премештања коју нам нуди MZ спецификација. Уместо ње, морамо да користимо таблицу премештања везану за сам компајлер. Она је читљива само пре фазе link-овања програма. Због тога ,као део BEX екстензије, link-ујемо мали део кода који чита ту таблицу и по њој, после, за време извршавања програма промени све потребне адресе.

## Формат екстензије

Екстензија садржи следеће:

- Библиотеке и системске позиве на које се програм ослања
- Код за учитавање библиотека
- Код за премештање адреса

Почетак екстензије су специфицира програму потребне библиотеке или системске позиве низом који се састоји од неког броја параграфа од $16$ бајтова да би било у сагласности са $\mathrm{MZ}$ спецификацијом. Низ је $\mathrm{null\text{-}terminated}$ што значи да се чита док се на наиђе на параграф који је састављен искључиво од нула. Од тих $16$ бајтова ми користимо само првих $11$, који складиште стандаризовано име мапе програма који нам служи као библиотека или део оперативног система са системским позивима.

Код за учитавање библиотека узима са стека, који је глобалан показивач на низ локација потребних библиотека. Дужина низа је позната јер је она иста као и број тражених библиотека(у супротном се програм не учитава). Чланови тог низа прослеђују различитим $\mathrm{header}$ фајловима одговарајућих библиотека, чиме је омогућено да су библиотеке интегрисане тако да се приликом писања $\mathrm{C}$ кода ни не примети да нису директно $\mathrm{link}$-ован. На крају учитавања пребацује контролу почетку програма тј. функцији типа *void* означеном *start_program*

Код за премештање адреса обавља задатак описан у делу Лимитације $\mathrm{Linker}$-а. И ако је налази у меморије после кода за учитавање библиотека, извршава се први. То је логично јер за правилно функционисање било ког дела програма је потребно да све адресе за ”скакање” буду на месту. Овај код се извршава не дирајући стек, а након завршетка преправки адреса прослеђује контролу коду из претходног параграфа.

Потенцијално корисно је да се формат приказе кроз пример:

| Офсет | Садржај |
|---|---|
| 0x00 | MEMMNG MAP |
| 0x10 | FILEMNG MAP |
| 0x20 | Null |
| 0x30 | `asm("db '</MAP NAME/>'");`<br>`asm("times 5 db 0");`<br>`asm("times 16 db 0"); //null terminated`<br><br>`extern void _start_program();`<br><br>`//extern void init_</FILE NAME WITHOUT EXT/>(void* funcs);`<br><br>`void __start__(void** libs){`<br>`  //call externs above`<br>`  _start_program();`<br>`}`<br>(горњи асемблер код је остављен примера ради али је у суштини оно што приказују поља горе) |
| ... | ...<br>`.relo_data_loop:`<br>`  cmp  esi, edx`<br>`  jae  .relo_data_done`<br><br>`  lea  edi, [ebx + esi] ; edi = physical address of a relocation table element`<br><br>`  ror  edi, 4`<br>`  mov  ds, di`<br>`  shr  edi, 28`<br>...<br>(ово је само део кода, читав код не би могао практично стати) |

# MAP пропратни фајл

Овај тип фајла носи екстензију .MAP и стоје уз библиотеку или било који фајл чије функције требају другом фајлу. Као што име сугерише овај фајл садржи мапу програма уз који долази, а са њим дели и име тј. њихово име се разликује само у екстензији.

Мапа програма је врло једноставна. Састоји се прво од једне речи која нам означава број N мапираних функција, при чему нису све функције мапиране, а потом наредних N речи означава офсете мапираних функција у

односу на почетак програма. Редослед по коме су функције је унапред одређен од онога који конструише мапу и не мора одговарати редоследу по коме су функције написане у C фајлу.

Тренутно се овај фајл мора ручно конструисати. Кратак програм који на C конструише мапу је у плану.

Формат фајла се може приказати табелом:

| Офсет | Садржај, пример |
|---|---|
| 0x00 | N |
| 0x02 | 0x1337 |
| 0x04 | 0xDEAD |
| ... | ... |
| N-1 | 0xBEEF |

## 7.5 Менаџер меморије

Менаџмент меморије је, као што је већ поменуто, једна од најважнијих задатака *Kernel*-а. Имплементација алоцирана меморије може да се уради на разне начине:

- Bitmap-ом
- Buddy Allocation System који користи Linux
- Равним низом

Оптимална имплементација зависи од ситуације и њено одређивање захтева барем:

- Апсолутно познавање хардвера, брзине писања и читања меморије и уређаја за складиштење
- Статистички анализу и математичко моделовање правилности у приступању меморији
- Дубоко разумевања процеса кеширања

Оптимална имплементација спада ван опсега овог рада, па ћемо ми остати на најједноставнијој имплементацији. Такође сложенији оперативни системи деле алоцирање меморије на више нивоа. На пример на оперативног система који даје меморију програмима и на ниво програма који раздељује меморију својим деловима која му је дата од претходног нивоа.

То је низ структура које описују почетак и дужину алоцираног сегмента меморије (**сегменти поменути овде нису исти као и сегменти из одељка о сегментацији меморије**).

Тражење слободне меморије неке одређене величине се врши функцијом ***malloc која тражи прву слободну рупу између алоцираних сегмената које је исте или веће величине него затражена.***

*Функција malloc*

```
/// @brief Allocates memory, the user is responsible for deallocating it
/// @param size the size of memory to be allocated
/// @return pointer to the location of available memory
void* malloc(size_t size){
    for(int i = 0;memory[i].len && i < max_memory_sectors;i++){
        size_t cur_available = memory[i+1].begin-memory[i].begin-memory[i].len;
        if(cur_available>size){ //we have found the available space
            void* res =(void*) (memory[i].begin+memory[i].len);
            memory[i].len+=size;
            return res;
        }
        else if(cur_available==size && memory[i+1].len){ //we have found the available space but we need to
merge the segments
            void* res =(void*) (memory[i].begin+memory[i].len);
            memory[i].len+=size;
            memory[i].len+=memory[i+1].len;
            for(int j = i+1;memory[j].len & j < max_memory_sectors;j++){//we shift the rest of the array one to
the left
                memory[j] = memory[j+1];

            }
            return res;
        }
    }
    return NULL; //no memory slot found
}
```

*Дealоцирање меморије врши се функцијом **dalloc*** (***free*** у другим оперативним системима) која само брише или измешта алоциране сегменте

## 7.6 IPC – **међипроцеска комуникација**

Потреба процеса, програма, да комуницирају једно са другим и сарађују је порпилично очигледна, не може нити би требало да може један процес све да зна и да све задатке може да обавља. Процеси могу комуницирати на разне начине:

- Прослеђивање порука
- Позивање удаљених процедура *RPC*
- Семафор
- Дељена меморија

Разлике између ових приступа је тешко видети из перспективе овако примитивног оперативног система. Али наш оперативни систем користи пар ових метода у упрошћеној форми.

- Наиме дељене библиотеке и системски позиви су један облик позивања удаљених процедура. Овај метод је доста поједностављен недостатком асинхроности и постојањем глобалног стека.
- Интерфејс са *file manager*-ом је урађен преко дељене меморије, али је због монолитске природе овог оперативног система сва меморија дељена

# 8 User-space

## 8.1 File system

Дугорочно складиштење информација је један од главних задатака сваког рачунарског система. Тај задатак по дефиницији поменутој на почетку пада на оперативни систем. Као природно решење на тај проблем поставила се подела података на фајлове који су категоризовани у фолдере, са којима смо сви већ упознати. Такво решење назива се *File system*. Ми ћемо овде обрадити ранију верзију таквог система чија се мало унапређена верзија и даље користи у рачунарима данашњице.

## 8.2 FAT16 file system

Оригинално направљен 1977. за MS-DOS и касније коришћен као примарни *file system Windows 9x* оперативних система. Изумели су га Бил Гејтс и Марк МекДоналд.

Сам концепт је врло једноставан. Састоји се од практично две структуре података. Једна је повезана листа смештена у таблицу, која одређује који

делови диска(уређаја за складиштење података) је заузета и који је редослед читања тих делова.

Другу структуру чине директоријуми (фолдери и фајлови) који подвлачењем једни под друго, фолдер у фолдеру, формирају стабло, где је *root* директоријум корен, а фајлови листови. Предност овог система чини врло мало заузимање меморије подразумевајући релативно компетентну имплентацију.

# Header у boot sector-у

Као што је претходно поменуто специфичности система стоје у једном делу *boot sector*-а, што је врло логично јер су информације како су подаци распоређени нешто што је потребно оперативном систему при самом почетку дизања. Први део информација чини BPB - BIOS Parameter Block.

## *BIOS Parameter Block*

Ово је блок података који је најбоље приказан табелом:

| Офсет (hex) | Величина у бајтовима | Садржај |
|---|---|---|
| **0x00** | 3 | Прва три бајта овог блока гласе*: EB 3C 90*. Дисасембловани они се преводе на *JMP SHORT 3C NOP*. Ово служи само да би прескочили блок података и прешли на део *boot sector*-а који садржи дање инструкције. Овај код је неопходан чак и за партиције диска које нису намењене за подизање система, јер доста оперативних система проверава валидност и постојање file system-а преко овог кода, притом се адреса на коју се ”скаче” мења на бесконачни циклус тј. EB FE 90. |
| **0x03** | 8 | OEM идентификатор. Првих 8 бајтова је верзија *DOS*-а која се користи. Званична Микрософтова спецификација каже да у суштини ово поље нема велико значење али упркос томе препоручују да се стави *MSWIN4.1,* јер неки драјвери проверавају ово поље као доказ валидности file system-а, што се у пракси показује тачним. |
| **0x0B** | 2 | Број бајтова по сектору диска, обично 512 |

| | | |
|---|---|---|
| **0x0D** | 1 | Број бајтова по кластеру(шта је кластер биће касније описано) |
| **0x0E** | 2 | Број резервисаних сектора, *boot sector* мора бити укључен у ову вредност |
| **0x10** | 1 | Број FAT-ова овај број је често 2 ради редундансе, ми ћемо овде ставити 1 |
| **0x11** | 2 | Максималан број уноса у *root* директоријуму |
| **0x13** | 2 | Мали укупан број сектора у партицији, ако број не може да стане у вредност једне речи, користи се велики број сектора у партијци. |
| **0x15** | 1 | Овај бајт означава media descriptor type |
| **0x16** | 2 | Број сектора који заузима један *FAT* |
| **0x18** | 2 | Број сектора по траци, коришћено за *floppy* дискове, нама непотребно |
| **0x1A** | 2 | Број глава или страна диска, исто као и претходно поље нама непотребно |
| **0x1C** | 4 | Број сакривених сектора, тј. почетак партиције |
| **0x20** | 4 | Велики број сектора, користи се када мали није довољан |

Поља на офсетима $24, 26$ представљају геометрију диска. Ми се ослањамо на $\mathrm{BIOS}$-ове драјвере са интеракцијом са диском па нам то није потребно.

## *Media descriptor type*

Користи се за одређивање каквим уређајем за складиштење управљамо. За наше сврхе то је *Fixed disk* са једном страном, а тиме нам је потребна вредност $0\mathrm{xF8}$.

## *Extended Boot Record*

Следећи блок података назива се Extended Boot Record. Налази се одмах после претходног и у себи садржи такође код за подизање система. Ово је исто прикладно приказати табелом:

| Офсет (hex) | Величина у бајтовима | Садржај |
|---|---|---|
| **0x024** | 1 | Број диска, вредност која стоји овде треба да буде идентична вредности коју враћа BISO прекид 0x13, или оној која стоји у DL регистру при учитавању *boot sector*-а. Преведено ова вредност је 0x00 за *floppy* дискове, а 0x80 за хард дискове. |
| **0x025** | 1 | Заставе за *Windows NT, у супротном резервисано*. |
| **0x026** | 1 | Потпис, мора бити 0x28 или 0x29, у пракси више драјвера признаје 0x29 |
| **0x027** | 4 | VolumeID - серијални број, користи се за праћење партиција између компјутера, нама није од неке користи |
| **0x02B** | 11 | Обележје имена партиције, празан део је попуњен *space* карактерима |
| **0x036** | 8 | System identifier string - Ово поље је стринг репрезентације типа FAT фајл система, празан део је исто попуњен *space* карактерима. Спецификација каже да се никад не треба ослањати на садржај овог поља. |
| **0x03E** | 448 | Код. |
| **0x1FE** | 2 | Потпис партиције која може да се подигне 0xAA55. |

## *Имплементација*

Најједноставнији начин да се имплементира *file system* а да се притом испоштује спецификација јесте да се у асемблер коду ручно дефинишу ове вредности, што би изгледало овако:

```
;;=========================Variables for the FAT16 file system==========
%assign jump_point 0x3E
%define OEM_ID 'BSD  4.4'
%assign bytes_per_sector 512
;in code we assume this is always 1
%assign sectors_per_clutser 1
%assign reserved_sectors 1
%assign num_of_FATs 1
;each directory is 32
;;please make it divisible by 16
%assign num_of_root_dir 512
%assign small_sector_num 0xFFFF
%assign media_descriptor 0xF8
;;this should be calcualed according to small_sector_num
%assign sectors_per_FAT 61
%assign sectors_per_track 0
%assign num_of_heads 1
;;start of volume basically
%assign hidden_sectors 0
%assign large_sector_num 0
%assign drive_num 0x80
%assign reserved 0
%assign extended_boot_sig 0x29
;;idk what this is MacOS put it there
%assign volume_serial_num 0x93811B4
;should be the same as the root dir
%define volume_label 'BORISOSVOL '
%define file_system_type 'FAT16   '
;;-------------------------------------
;;Calculated starting points, in sectors
%assign volume_start hidden_sectors
%assign FAT_start volume_start + reserved_sectors
%assign root_dir FAT_start+num_of_FATs*sectors_per_FAT
%assign root_dir_size (num_of_root_dir*32)/bytes_per_sector
%assign data_start root_dir+root_dir_size
;;-----------------------------------------------------
;;assign where to put the sectors
%assign search_sector_address 0x7E00
%assign disk_address_packet_struct 0x8000
;;-----------------------------------------------------
;;we assign where the loader will be, needs to be 16 bytes alligned
%assign loader_memory_address 0x500
```

Где би даљи код користио асемблерске директиве *db, dw* и *dd,* на пример.

```
;jump point
db 0xEB
db jump_point
db 0x90
```

или

```
db volume_label
```

# File Allocation Table

*File Allocation Table* је таблица речи која се састоји од једног или више везаних листи, овде названих ланаца. Обратимо пажњу на то да је ово таблица речи и да један унос заузима 2 бајта, а тиме ја офсет N-тог уноса 2*N. Вредности поља у таблици престављају кластере, логичку поделу диска, и имају следеће значење:

- 0x0000 значи да је кластер слободан тј. не садржи никакве податке
- 0x0001 није валидна вредност јер је та позиција у табели резервисана
- 0x0002-0xFFF6 показивач на следећи кластер, тј. следећи унос у табели
- 0xFFF7 значи да је један или више сектора у кластеру корумпиран тј. кластер је лош
- >= 0xFFF8 Крај ланца

Конкретни параметри таблице које смо ми користили могу се видети у прошлом поглављу. Осим њих вредности резервисаних поља тј. прва два су:

- 0x0000 - за наше сврхе, први бајт је *media descriptor*, док је други само 0xFF
- 0x0002 - 0xFFFF, ознака крај ланца

Згодно за објашњење функционисање овакве табеле је пример:

| ОФСЕТ | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 | +8 | +9 | +A | +B | +C | +D | +E | +F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +0000 | F8 | FF | FF | FF | 03 | 00 | 04 | 00 | 05 | 00 | 06 | 00 | 07 | 00 | 08 | 00 |
| +0010 | FF | FF | 0A | 00 | 14 | 00 | 0C | 00 | 0D | 00 | 0E | 00 | 0F | 00 | 10 | 00 |
| +0020 | 11 | 00 | FF | FF | 00 | 00 | FF | FF | 15 | 00 | 16 | 00 | 19 | 00 | F7 | FF |
| +0030 | F7 | FF | 1A | 00 | FF | FF | 00 | 00 | 00 | 00 | F7 | FF | 00 | 00 | 00 | 00 |

- Други сектор резервисан за крај ланца
- Први директоријум у једном комаду.
- Други директоријум који је распрострањен
- Трећи ланац тј. директоријум у једном комаду
- Лоши кластери

## Имплементација руковања $FAT$-а

Оптимална имплементација носи са собом већ поменуте захтеве сличне оним за менаџмент меморије. Сходно са тим ми ћемо тежити једноставности. Потребне су нам три функције:

- Погледај један унос у таблици - $\mathrm{FAT\_lookup}$
- Нађи следећи слободан кластер - $\mathrm{FAT\_free}$
- Измени један унос - $\mathrm{FAT\_edit}$

При томе да ћемо у датом тренутку кеширати само један сектор таблице. Имплементације функција су следеће:

*Имплементација функције $FAT\_lookup$*

```
uint16_t FAT_lookup(uint16_t cluster){
    int needed_sector = FAT_start+(cluster*2)/bytes_per_sector;
    if(needed_sector == cached_FAT_sector){ //the sector we need is cached
        return FAT_cache[(cluster%(bytes_per_sector/2))];//TODO: check this
calc
    }
    else{ //we need to load a new sector
        load_sector(needed_sector,(void*) &FAT_cache);
        cached_FAT_sector = needed_sector;
        return FAT_cache[(cluster%(bytes_per_sector/2))];//TODO: check this
calc
    }
}
```

*Имплементација функције* $FAT\_free$

```
uint16_t FAT_free(){
    for(uint16_t i =2;i<sectors_per_FAT*(bytes_per_sector/2);i++){
        If(!FAT_lookup(i)){return i;}
    }
    return 0;
}
```

*Имплементација функције* $FAT\_edit$

```
int FAT_edit(uint16_t cluster,uint16_t val){//this is really in-
efficient but i don't have time nor energy for something bet-
ter
    FAT_lookup(cluster); //it caches the cluster
    FAT_cache[(cluster%(bytes_per_sector/2))] = val;
    write_sector(FAT_start+(cluster*2)/bytes_per_sec-
tor,&FAT_cache);
    return 0; //TODO: Diagnostics
}
```

# Формат директоријума

У *FAT16 file system*-у су сви фолдери и фајлови категорисани као директоријуми. То су у суштини стуктуре које садрже све податке које логички припадају неком фајлу или долдеру. Следећих 32 бајта чине податке једног директоријума:

| ОФСЕТ(HEX) | ВЕИЛЧИНА | САДРЖАЈ |
|---|---|---|
| 00H | 8 бајтова | Име фајла |
| 08H | 3 бајта | Екстензија |
| 0BH | 1 бајт | Атрибутни бајт |
| 0CH | 1 бајт | Резервисано за Windows NT |
| 0DH | 1 бајт | Печат тренутка креацјие фајла у милисекундама |
| 0EH | 2 бајта | Време креације |
| 10H | 2 бајта | Датум креације |
| 12H | 2 бајта | Датум последњег приступа |
| 14H | 2 бајта | Резервисано |
| 16H | 2 бајта | Датум последњих измена |
| 18H | 2 бајта | Време последњих измена |
| 1AH | 2 бајта | Почетни кластер |
| 1CH | 4 бајта | Величина фајла у бајтовима |

## *Име фајла и екстензија*

Име фајла је дугачко 8 бајтова, при чему краћа имена морају да се попуне са празним простором (ASCII 0x20). Ектензија је дугачка 3 бајта и подлаже истим правилима попуњавања као и име. У суштини, по спецификацији дозвољени карактери за име фајла су велика слове енглеске абецеде, цифре од 0 до 9 и симболи #, $, %, &, ', (, ), -, @.

Први бајт имена може садржати додатне информације и подлаже додатним правилима:

1. Вредност 0x00 се чита као престани претрагу, тј. крај директоријума
2. Вредност 0x05 треба замеинити са 0xE5
3. Не сме имати вредност 0x20 - празан простор
4. Вредност 0xE5 се интерпретира као да је место које запоседа директоријум слободно и да се на његово место може ставити други

## *Атрибутни бајт*

Третира се као низ заставица:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Резервисано | A | D | V | S | H | R | 0000 |

- R - Read only, не сме се модификовати нити брисати
- H - Hidden, сакивен је од користника
- S - System, системски фајл, користник и већина програма не требају да га дирају
- V - Volume name, Када је ова застава сетована, директорија не показује на почетак фајла, него ни на шта. У целом систему мора постојати тачно

један унос са овом заставом. Нема екстензију па име може да буде дугачко 11 бајтова, при томе да подлаже истим рестрикцијама
- D - Directory, ова заставица означава да директоријум не показује на фајл на фајл него на још једну таблицу директоријума, тј. фолдер. Важно је напоменути да је величина фолдера у уносу директорије увек 0.
- A - Archive flag, користи се од сране *backup* алата

### *Резервисано за Windows NT*

Овај бајт користи Windows NT. Он подеси тај бајт на 0 и више га никад не дира. У које сврхе га користи није познато.

### *Одреднице времена креације, читања и модификације*

Тренутна имплемнтација *file manager*-а не подржава временске одреднице па их ни нећемо описати овде.

### *Почетни кластер*

Реч која показује на почетни кластер података датог уноса директоријма. Ако је ово фолдер, показивач показује на почетак таблице директоријума састављене од под-директоријума

## Организација у фајлове и фолдере

Већина описа *FAT16 File system*-а нема овај одељак, који је аргументативно кључан за разумевање функционисања.

### *Root фолдер*

*Root dir* је специјалан фолдер који се налази у већ предодређеном месту на диску (то се може видети у секцији BIOS Parameter Block). Он има предефинисан максималан број уноса, обично па и код нас је то 512. У пракси се испоставља да је због те предефинисаности много лакше наћи потребне фајлове у њима што може бити корисно *bootloader*-има који имају ограничен простор за код или немају приступ напреднијим функцијама које би баратале *File system*-ом.

### *Фолдер*

Као што је већ поменуто фајлови и фолдери чине структуру података стабло. Да би се остварило стабло:

1. Сваки родитељ мора имати показиваче на своју децу

2. (Скоро) свако дете мора имати показивач на родитеља, фајловима не треба то јер ми гледамо њихов почетни кластер само са намером читања истих
3. Прва ставка је описана у предходним секцијама. Друга ставка је урађена на следећи начин.

На почетку таблице директорија у фолдеру налазе се два уноса:

1. Носи назив „ .. “ (без празног места испред) и не садржи никакве податке осим показивача на кластер родитељског над-фолдера у пољу почетног кластера.
2. Носи назив „ . “ и врло је сличан првом осим што садржи показивач на почетни кластер тренутног фолдера, што се у пракси може испоставити корисно при имплементацији.

Овим је омогућено релативно несметано кретање по стаблу. **Врло битно** за напоменути је да фолдери такође могу бити ланци у $FAT$-у, под условом да је таблица директоријума дужа од једног кластера.

### *Израчунавање неких битних адреса*

```
;;Calculated starting points, in sectors
%assign volume_start hidden_sectors
%assign FAT_start volume_start + reserved_sectors
%assign root_dir FAT_start+num_of_FATs*sectors_per_FAT
%assign root_dir_size (num_of_root_dir*32)/bytes_per_sector
%assign data_start root_dir+root_dir_size
```

## Детаљи имплементације File system-а

- Сва наша интеракција са диском базирана је на BIOS-овим прекидма
- Имплементација се доста ослања на динамичко алоцирање меморије и интерагује са другим програмима преко исте

## 8.3 Shell

*Shell* је *user-space* програм који обезбеђује јеснтоставан интерфејс са Kernel-ом оперативног система. Најједноставнији *Shell* обавља скедеће функције:

- Покретање других програма
- Омогућава улаз за извршавање других програма
- Обезбеђује излаз за извршене програме

Те функције обично испуњава у виду текстуалног интерфејса. Нажалост у нашем опреативном систему се *Shell* идаље пише, при томе да још није функционалан.

# 9 Мотивација

Мотивација за овај рад била је чисте истраживачке природе. Овакав пројекат је поприлично озбиљан и захтева време. Такође је била радозналост: колико су оперативни системи компликовани, и колико један човек, изводећи логичке закључке може да се приближи прављењу једног опреативног ситема. Додуше овај рад је само један грана фасцинације аутора са технологијом, а поготово *low-level* програмирањем.

# 10 Закључак

Оперативни системи су подцењен део наше свакодневнице, олакшавају па често и омогућавају директно или индиректно разне аспекте модерног живота. Они су по дефиницији у скоро сваком дигиталном уређају.

Из перспективе конструкције једног оперативног система није погрешно рећи да је то једна од најкомплекснијих области информатике, једним делом због широкоопусности коју је овај рад само дотакао, а другим делом практично погледано: за програмирање других ствари, у *high-level* језицима на располагању стоје бесконачне алатке за помоћ и исправке грешака попут *Garbage collector*-а или врло софистицираних *debugger*-а. Програмирање оперативних система учи да такве ствари не узимамо „здраво за готово“. Врло битна тачка је што програмирање оперативних система може прогамеру разјаснити мистерју која лежи између онога што се дешава на нивоу хардвера и високих апсракција са којима се ми срећемо у модерним оперативним сиссетмима попут прозора или класа у објектном оријентисаном програмирању.

У вези са BorisOS-*ом* специфично, то је пројекат који очигледно није завршен поводом временских ограничења, али је упркос томе био изузетно значајан за упознавање са технологијом, и као такав залужује да се настави рад на њему. У скоријем плану за развој били би очигледнији недостаци попут *Shell*-а и непоштовања стандардне C библиотеке. Али, у све практичне сврхе, овај пројекат нема границу.

# 11 Литература

Brown, Ralf. „Ralf Brown's interrupt list.“ . *ctyme.* <http://www.ctyme.com/rbrown.htm>.

„Design of the FAT file system.“ . *Wikipedia.* <https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system>.

„FAT16 File system.“ . *Maverick-OS.* <http://www.maverick-os.dk/FileSystemFormats/FAT16_FileSystem.html>.

Forever Young Software. „GitHub.“ . *FYSOS.* <https://github.com/fysnet/FYSOS/tree/master>.

INTEL. „Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4.“ . *ntel.com.* <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.

OSDEV. „wiki.osdev.org.“ . <wiki.osdev.org>.