



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ
НАУКА У НОВОМ САДУ



Душан Лечић

Мониторинг система на програмском језику Раст

ЗАВРШНИ РАД

Основне академске студије

Нови Сад, 2026



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
21000 НОВИ САД, Трг Доситеја Обрадовића 6

Број:

ЗАДАТАК ЗА ЗАВРШНИ РАД

Датум:

(Податке уноси предметни наставник - менџор)

Студијски програм:	Софтверско инжењерство и информационе технологије		
Студент:	Душан Лечић	Број индекса:	SV 80/2021
Степен и врста студија:	Основне академске студије		
Област:	Електротехничко и рачунарско инжењерство		
Ментор:	Игор Дејановић		
НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ЗАВРШНИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:			
<ul style="list-style-type: none">- проблем – тема рада;- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;			

НАСЛОВ ЗАВРШНОГ РАДА:

Мониторинг система на програмском језику Раст

ТЕКСТ ЗАДАТКА:

Lore ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua quaeat voluptatem. Ut enim aequre doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos iridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

Руководилац студијског програма:	Ментор рада:

Примерак за: - Студента; - Ментора



КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, РБР:		
Идентификациони број, ИБР:		
Тип документације, ТД:	Монографска документација	
Тип записа, ТЗ:	Текстуални штампани материјал	
Врста рада, ВР:	Дипломски - бачелор рад	
Аутор, АУ:	Душан Лечић	
Ментор, МН:	Др Игор Дејановић, редовни професор	
Наслов рада, НР:	Мониторинг система на програмском језику Раст	
Језик публикације, ЈП:	српски/ћирилица	
Језик извода, ЈИ:	српски/енглески	
Земља публиковања, ЗП:	Република Србија	
Уже географско подручје, УГП:	Војводина	
Година, ГО:	2026	
Издавач, ИЗ:	Ауторски репринт	
Место и адреса, МА:	Нови Сад, трг Доситеја Обрадовића 6	
Физички опис рада, ФО: (поплављено/страна/цитата/табела/слика/графика/прилога)	7/56/24/2/30/0/2	
Научна област, НО:	Електротехничко и рачунарско инжењерство	
Научна дисциплина, НД:	Примењене рачунарске науке и информатика	
Предметна одредница/Кључне речи, ПО:	Мониторинг система, Модуларан дизајн, Оперативни системи, Руст, Терминал, Flux архитектура, Ratatui	
УДК		
Чува се, ЧУ:	У библиотеки Факултета техничких наука, Нови Сад	
Важна напомена, ВН:		
Извод, ИЗ:	Овај рад представља развој терминалног алатка за мониторинг система написаног у програмском језику Раст. Циљ је израда брзог, ефикасног и поузданог решења које омогућава приказ искоришћености ресурса и надзор над активним процесима. Архитектура апликације заснива се на модуларном дизајну и Flux архитектонском обрасцу, док је кориснички интерфејс реализован у терминалу помоћу Ratatui библиотеке.	
Датум прихваташа теме, ДП:		
Датум одbrane, ДО:	01.01.2025	
Чланови комисије, КО:	Председник:	Др Петар Петровић, ванредни професор
	Члан:	Др Марко Марковић, доцент
	Члан:	Потпис ментора
	Члан, ментор:	Др Игор Дејановић, редовни професор



KEY WORDS DOCUMENTATION

Accession number, ANO:		
Identification number, INO:		
Document type, DT:	Monographic publication	
Type of record, TR:	Textual printed material	
Contents code, CC:		
Author, AU:	Dušan Lečić	
Mentor, MN:	Igor Dejanović, Phd., full professor	
Title, TI:	System monitoring in Rust	
Language of text, LT:	Serbian	
Language of abstract, LA:	Serbian/English	
Country of publication, CP:	Republic of Serbia	
Locality of publication, LP:	Vojvodina	
Publication year, PY:	2026	
Publisher, PB:	Author's reprint	
Publication place, PP:	Novi Sad, Dositeja Obradovica sq. 6	
Physical description, PD: (chapters/pages/ref./tables/pictures/graphs/appendices)	7/56/24/2/30/0/2	
Scientific field, SF:	Electrical and Computer Engineering	
Scientific discipline, SD:	Applied computer science and informatics	
Subject/Key words, S/KW:	System monitor, Clean architecture, Operating systems, Rust, Terminal, Flux architecture, Ratatui	
UC		
Holding data, HD:	The Library of Faculty of Technical Sciences, Novi Sad	
Note, N:		
Abstract, AB:	This thesis presents the development of a terminal-based system monitoring tool written in the Rust programming language. The goal is to create a fast, efficient, and reliable solution for visualizing system resource usage and monitoring active processes. The application architecture follows a modular design and the Flux architectural pattern, while the terminal user interface is implemented using the Ratatui library.	
Accepted by the Scientific Board on, ASB:		
Defended on, DE:	01.01.2025	
Defended Board, DB:	President: Petar Petrović, Phd., assoc. professor	
Member:	Marko Marković, Phd., asist. professor	
Member:		
Member, Mentor:	Igor Dejanović, Phd., full professor	Mentor's sign



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
21000 НОВИ САД, Трг Доситеја Обрадовића 6

ИЗЈАВА О НЕПОСТОЈАЊУ СУКОБА ИНТЕРЕСА

Изјављујем да нисам у сукобу интереса у односу ментор – кандидат и да нисам члан породице (супружник или ванбрачни партнери, родитељ или усвојитељ, дете или усвојеник), повезано лице (кровни сродник ментора/кандидата у правој линији, односно у побочној линији закључно са другим степеном сродства, као ни физичко лице које се према другим основама и околностима може оправдано сматрати интересно повезаним са ментором или кандидатом), односно да нисам зависан/на од ментора/кандидата, да не постоје околности које би могле да утичу на моју непристрасност, нити да стичем било какве користи или погодности за себе или друго лице било позитивним или негативним исходом, као и да немам приватни интерес који утиче, може да утиче или изгледа као да утиче на однос ментор-кандидат.

У Новом Саду, дана _____

Ментор

Кандидат

Садржај

1 Увод	1
2 Стање у области	3
2.1 Традиционални алати	3
2.2 Модерни алати	4
2.3 Упоредна анализа	5
2.4 Идентификоване празнине и мотивација	5
3 Технологије	7
3.1 Раст, програмски језик	7
3.2 Tokio	8
3.3 Rataatui	9
3.4 Crossterm	10
4 Архитектура система	13
4.1 Општа архитектура	13
4.2 Модули система	13
4.3 Ток података	21
4.4 Обрасци дизајна	21
4.5 Синхронизација	21
5 Имплементација	23
5.1 Прикупљање системских метрика	23
5.2 Управљање процесима	26
5.3 Руковање грешкама	29
5.4 Асинхроно прикупљање метрика	30
5.5 Кориснички интерфејс	31
6 Пример коришћења	33
6.1 Покретање апликације	33
6.2 Навигација кроз табове	33
6.3 Мониторинг процесора	33
6.4 Мониторинг меморије	34
6.5 Управљање процесима	35
6.6 Мониторинг дискова	36
6.7 Мониторинг мреже	37
6.8 Нотификације и подешавања	37
6.9 Контроле и пречице	38
7 Закључак	41
Списак слика	43

Списак листинга	45
Списак табела	47
Списак коришћених скраћеница	49
Списак коришћених појмова	51
Биографија	53
Литература	55

Глава 1

Увод

Надгледање системских ресурса представља један од основних задатака програмера и системских администратора који раде на оптимизацији и перформансама софтвера. Способност праћења и анализе искоришћености процесора, меморије, диска и мрежних ресурса у реалном времену омогућава правовремено откривање уских грла у перформансама, дијагностиковање проблема и доношење одлука о оптимизацији система.

У овом раду приказана је имплементација системског монитора за Линукс (eng. Linux) оперативни систем, а решење је осмишљено тако да се може једноставно проширити и на друге платформе. У свету Линукс оперативних система постоји велики број алата за мониторинг система, од традиционалних као што су *top* [1] и *htop* [2], до модерних решења попут *btop* [3] и *gtop* [4]. Иако ови алати пружају основне функционалности за праћење системских ресурса, многи од њих имају одређена ограничења у погледу прегледности, перформанси или флексибилности приkaza података.

- **Функционалности ове имплементације укључују:**

- Приказ основних информација о систему, као и приказ графика искоришћености процесора и меморије у реалном времену
- Могућност управљања процесима
- Приказ метрика процесора
- Приказ метрика меморије
- Приказ метрика диска
- Приказ доступних мрежних интерфејса
- Управљање и приказ обавештења

Рад је организован у седам поглавља. У другом поглављу приказано је стање у области система за мониторинг, са освртом на тренутно најпопуларније алате и њихове предности. Треће поглавље детаљно описује технологије коришћене у овој имплементацији. Четврто поглавље представља архитектуру система са описом сваког пакета. Пето поглавље се детаљно бави имплементацијом, укључујући колекторе за прикупљање метрика, систем за управљање процесима и кориснички интерфејс. У шестом поглављу је приказан пример коришћења. Седмо поглавље сумира постигнуте резултате и даје смернице за даља унапређења.

Глава 2

Стање у области

Системи за мониторинг ресурса представљају есенцијалне алате у администрацији и оптимизацији Линукс оперативних система. Ови алати омогућавају праћење перформанси у реалном времену, идентификацију проблема и доношење одлука о алокацији ресурса. Захваљујући развоју технологија и програмских језика, последњу деценију је обележио велики број нових решења која нуде различите приступе визуелизацији и презентацији системских метрика.

2.1 Традиционални алати

2.1.1 top

Алат `top` представља један од најстаријих и најраспрострањенијих система за мониторинг [1]. Први пут се појавио као део `procps` пакета и данас долази прединсталiran на готово свим Линукс дистрибуцијама. `top` приказује листу процеса са њиховим PID-ом, корисником, искоришћењем CPU-а, меморије и другим метрикама, уз могућност интерактивног сортирања и слања сигнала процесима [5].

Главне карактеристике `top` алата су:

- Минимална потрошња ресурса и изузетно мала величина извршног фајла
- Универзална доступност на свим UNIX/Линукс системима
- Основне функционалности без напредних визуелних елемената
- Кратке, једнословне команде за интеракцију

Међутим, `top` има и значајна ограничења. Недостатак подршке за миш, немогућност скроловања, одсуство боја за истицање важних информација и мање интуитиван кориснички интерфејс чине га мање приступачним за почетнике.

2.1.2 htop

Као одговор на ограничења `top` алата, Hisham Muhammad је 2004. године започео развој `htop`-а [2], алата који је ставио фокус на организацију приказа системских метрика са графиконима за кључне CPU и меморијске метрике, хијерархијским приказом процеса и подршком за седам различитих режима боја.

`htop` пружа следеће побољшања у односу на `top`:

- Интерактивни кориснички интерфејс са подршком за миш и скроловање
- Визуелно истицање важних информација искоришћењем боја
- ASCII графикони за приказ искоришћења CPU језгара и меморије

2.1 Традиционални алаћи

- Једноставно управљање процесима преко функцијских тастера
- Могућност прегледа отворених фајлова процеса и праћења процеса

Иако има око 12,000 линија Ц кода, htop остаје релативно компактан с обзиром на понуђене функционалности.

2.2 Модерни алати

2.2.1 btop++

btop++ [3] је Ц++ варијанта популарног bashtop [6] алата од истог програмера и представља наставак развоја bashtop-а и bryutop-а [7].

Особине btop++ алата:

- Приказ статистика искоришћења процесора, меморије, дискова, мреже и процеса
- Потпуна подршка за миш, мени систем инспирисан играма, могућност филтрирања процеса и приказ у облику стабла
- Висок степен прилагођавања изгледа и понашања преко конфигурационих фајлова и тема
- Брзе перформансе захваљујући имплементацији у Ц++-у

btop++ захтева ГЦЦ (eng. GCC) 11 или новији за компајлирање и пружа статички компајлиране бинарне датотеке за различите архитектуре.

2.2.2 bottom

Bottom [8] је вишеплатформски графички систем за мониторинг процеса и система написан у Раст-у [9], са подршком за Линукс, macOS и Windows.

Карактеристике bottom алате:

- Дизајн инспирисан gotop-ом [4] са графиконима и метрикама распоређеним по областима
- Прилагодљив интерфејс са темама и опцијама за распоред вицета
- Конфигурабилност кроз ТОМЛ фајлове
- Искоришћавање Раст-овог система типова и безбедности меморије

Bottom користи око 2,400 линија Раст кода, што га чини релативно компактним у поређењу са другим решењима. Међутим, може имати стрмију криву учења због сложенијих опција конфигурације у поређењу са традиционалним алатима.

2.2.3 Други алати

Осим наведених, вредни помена су и:

- atop [10] – Напредни систем и процесни монитор који пружа детаљне информације о системским ресурсима и који је посебно користан за анализу перформанси и решавање проблема јер логује системску активност и може приказати искоришћење ресурса по појединачним процесима током времена

- nmon [11] – Алат који нуди контролу над тиме шта се приказује, са могућношћу извоза података у ЦСВ (eng. CSV) формату
- уtop [12] – Раст имплементација слична *gtop*-у (више се не одржава)

2.3 Упоредна анализа

У наставку је дат табеларни приказ поређења традиционалних и модерних решења о којима је било дискусије у претходним секцијама.

Алат	Језик	Платформе	Кључне особине
top [1]	Ц	Линукс/UNIX	Минималан, универзалан
htop [2]	Ц	Линукс/UNIX	Боје, миш, ASCII графикони
btop++ [3]	Ц++	Линукс/BSD/macOS	Модеран UI, теме, GPU подршка
bottom [8]	Раст	Линукс/macOS/Windows	Вишеплатформски, ТОМЛ config

Табела 1: Упоредни преглед алата за системски мониторинг

2.4 Идентификоване празнине и мотивација

Анализом постојећих решења могу се идентификовати следеће празнине и подручја за унапређење:

Безбедност и поузданост – Алати писани у Раст-у, попут *bottom*-а, нуде предности у виду система типова и безбедности меморије који гарантују одсуство великог броја грешака карактеристичних за системско програмирање.

Асинхронна архитектура – Традиционални алати користе синхроне приступе прикупљању података, што може довести до блокирања интерфејса.

Систем обавештења – Мали број алата пружа уграђен систем упозорења и обавештења директно у интерфејсу терминала.

Управљање процесима – Иако већина алата омогућава слање *SIGKILL* сигнала, мало њих пружа свеобухватан систем за управљање процесима са различитим типовима сигнала (*SIGTERM*, *SIGSTOP*, *SIGCONT*).

Узимајући у обзир наведене недостатке, постоји јасна потреба за модерним и безбедним алатом који комбинује предности постојећих решења, а истовремено уводи нове концепте попут асинхроне архитектуре, централизоване обраде метрика и интегрисаних системских обавештења. Ови захтеви представљају основу мотивације за развој

2.4 Идентификоване јазнице и мотивација

новог система за мониторинг, који би био лак за проширивање, стабилан у раду и прилагођен потребама савремених Линукс окружења.

Глава 3

Технологије

Развој ефикасног, поузданог и одрживог система за мониторинг захтева пажљив избор технологија које омогућавају високе перформансе, безбедност и флексибилност. Ова имплементација се базира на коришћењу модерног Раст екосистема. Овај одељак описује главне технологије коришћене у имплементацији: *Rust* (енг. *Rust*) програмски језик као основу целокупног пројекта [9], *Tokio* асинхрони *runtime* за неблокирајућу обраду података [13], *Ratatui* библиотеку за креирање корисничког интерфејса у терминалу [14], и *Crossterm* библиотеку за вишеплатформску манипулацију терминала [15]. Заједно, ове технологије чине робустан систем који омогућава прикупљање и приказивање системских метрика у реалном времену са минималним оптерећењем система.

3.1 Раст, програмски језик

Раст (енг. *Rust*) је системски програмски језик опште намене (енг. *general-purpose*) код кога се посебна пажња посвећује перформансама, безбедности типова (енг. *type safety*), безбедности меморије (енг. *memory safety*) и конкурентности [9].

Раст подржава више програмских стилова, а многе његове синтаксне и концептуалне особине као што су рад са непроменљивим подацима, коришћење функција вишег реда, алгебарских типова и подударања образца (енг. *pattern matching*), потичу из функционалних језика. Истовремено, Раст омогућава и објектно-оријентисан начин програмирања преко структуре, енумерација, трејтова и метода [16].

Познат је по изузетно строгом приступу меморијској безбедности, све референце морају бити валидне. То се постиже без коришћења традиционалног сакупљача ђубрета (енг. *garbage collector*). Уместо тога, Раст користи свој интерни позајмљивач (енг. *borrow checker*), механизам који већ током компилације прати области важења (енг. *scope*) и животни век (енг. *lifetime*) свих објеката, што спречава уобичајене грешке у руковању меморијом и појаву трке до података (енг. *data race*).

3.1.1 Власништво

Кључна особина која Раст разликује од других програмских језика је његов систем **власништва** (енг. *ownership*) [9]. Свака вредност у Раст-у има тачно једног “власника”, променљиву или структуру која контролише њен животни век. Када власник изађе из области важења, вредност се аутоматски ослобађа. Ово правило елиминише про-

блеме попут двоструког ослобађања меморије (енг. *double-free*) и коришћења меморије након ослобађања (енг. *use-after-free*).

3.1.2 Позајмљивање

Поред власништва, Раст уводи концепт **позајмљивања** (енг. *borrowing*) који омогућава привремени приступ подацима без преношења власништва. Постоје два типа позајмљивања: непроменљиво позајмљивање (енг. *immutable borrow*) означеног са &T, које дозвољава само читање, и променљиво позајмљивање (енг. *mutable borrow*) означеног са &mut T, које дозвољава и читање и писање. Компајлер гарантује да у било ком тренутку може постојати или више непроменљивих позајмљивања или тачно једно променљиво позајмљивање, али не оба истовремено.

3.1.3 Животни век

Животни век (енг. *lifetime*) је Раст-ова апстракција која описује колико дugo је референца валидна. Иако се у већини случајева животни век аутоматски закључује од стране компајлера (енг. *lifetime elision*), понекад је потребно експлицитно дефинисати животни век референци. Животни век се означава апострофом и генеричким параметром. Компајлер користи ове анотације да провери да ли су референце валидне у тренутку када се користе. Циљ је спречити ситуације где би референца могла да показује на меморију која је већ ослобођена, такође познати као висећи показивачи.

3.1.4 Висећи показивачи

Висећи показивач (енг. *dangling pointer*) је референца која показује на меморијску адресу која више није валидна, обично зато што је објекат на који је показивала ослобођен из меморије. Ово је једна од честих рањивости у језицима са ручним управљањем меморијом, као што су Ц и Ц++, која може довести до повећавања привилегија (енг. *privilege escalation*) или до цурења информација уколико се показивач користи за писање у меморију.

3.2 Tokio

Tokio [13] је асинхрони извршни оквир (енг. *async runtime*) за Раст који омогућава писање неблокирајућих апликација. Пружа инфраструктуру за извршавање асинхроних задатака (енг. *async tasks*).

3.2.1 Асинхрони модел извршавања

Раст имплементира асинхроно програмирање кроз `async/await` синтаксу [17]. Функције означене кључном речју `async` враћају објекат типа `Future` који представља вредност која ће бити доступна у будућности. Раст-ови `Future` типови су лењи (енг. *lazy*) - не извршавају се аутоматски, већ морају бити експлицитно покренути позивањем методе `poll` или извршавањем кроз оквир као што је Tokio.

Tokio користи планер (енг. *scheduler*) заснован на алгоритму крађе посла (енг. *work-stealing scheduler*) [18], који ефикасно распоређује асинхроне задатке на доступне нити. Свако језгро (енг. *core*) има свој ред (енг. *queue*) задатака који извршава. Када је ред празан, језгро проверава задатке од преосталих језгара и преузима њихов посао. Језгро ће отићи у стање спавања ако не може да пронађе задатак који би требало да се извршава.

Tokio такође пружа примитиве за синхронизацију између задатака, попут канала (енг. *channel*) и брава (енг. *mutex*), која се у овом раду користе за безбедну комуникацију између компоненти за прикупљање података и компоненти за приказ.

У овој имплементацији, Tokio се користи за неблокирајуће прикупљање метрика из различитих извора. Свака компонента за прикупљање података ради као асинхрони задатак који периодично чита системске метрике.

3.3 Ratatui

Ratatui [14] је Растворска библиотека за креирање корисничког интерфејса за терминал (енг. *Terminal User Interface*).

3.3.1 Рендеровање

Када разговарамо о развоју корисничког интерфејса, разликујемо две главне парадигме: режим тренутног рендеровања (енг. *immediate mode rendering*) и режим задржаног рендеровања (енг. *retained mode rendering*).

- **Режим тренутног рендеровања** се заснива на исцртавању корисничког интерфејса за слику (енг. *frame*) која означава промену стања апликације. Не постоје објекти који се трајно чувају у меморији.
- **Режим задржаног рендеровања** се заснива на креирању виџет (енг. *widget*) објеката који интерно чувају своје стање, а који се касније појединачно модификују у зависности од стања апликације.

Ratatui користи режим тренутног рендеровања са посредним баферима. Ово значи да апликација за сваку слику мора експлицитно рендеровати све виџете који треба да буду приказани. Иако овај приступ може изгледати неефикасно, Ratatui оптимизује рендеровање користећи двобуферски систем. Нови садржај се исцртава у позадински бафер, а затим се врши поређење (енг. *differing*) са предњим бафером који је тренутно приказан на екрану. На терминал се шаљу само ANSI escape секвенце за промене између два бафера, што значајно смањује количину података која се мора пренети.

3.3.2 Систем виџета

Ratatui пружа богат скуп уграђених виџета који се могу комбиновати за изградњу сложених интерфејса [19]:

- **Block** — основни градивни елемент који дефинише правоугаону област са границама и насловом
- **Paragraph** — приказ текстуалног садржаја са подршком за стилизовање
- **List** — вертикална листа ставки са подршком за селекцију и скроловање
- **Table** — табеларни приказ података са подршком за заглавља, сортирање и селекцију редова
- **Chart** — линијски и бар графикони за визуелизацију временских серија
- **Gauge** — индикатори напредка и искоришћености ресурса
- **Tabs** — компонента за навигацију између различитих приказа

За целокупну листу виџета, потребно је погледати документацију. Сваки виџет имплементира `Widget trait` који дефинише методу `render` за исцртавање виџета на датој правоугаоној области терминала. Виџети се могу угнездити једни у друге, што омогућава креирање сложених распореда.

3.4 Crossterm

Crossterm [15] је Раст библиотека за манипулацију терминалом која омогућава писање вишеплатформских текстуалних интерфејса. Подржава UNIX и Windows терминале.

3.4.1 Апстракција функција терминала

Crossterm пружа API за рад са терминалом који апстрахије разлике између различитих оперативних система и емулатора терминала. На UNIX системима, Crossterm користи ANSI escape секвенце за контролу терминала, док на Windows-у користи Windows Console API функције. Ова апстракција омогућава да исти код ради на свим подржаним платформама без измена.

Главне области функционалности које Crossterm пружа су:

Управљање курсором — Crossterm омогућава померање курсора на произвољне позиције, сакривање и приказивање курсора, чување и враћање позиције курсора, као и контролу трептања курсора.

Стилизовање текста — Подршка за боје текста и позадине, као и атрибуте текста попут подебљавања, италика, подвлачења, и других ефеката.

Управљање терминалом — Чиšћење екрана или појединачних линија, улазак у сирови режим (енг. *raw mode*) за читање сваког притиска тастера појединачно, и добијање димензија терминала.

Обрада догађаја — Читање догађаја са тастатуре и миша у блокирајућем или неблокирајућем режиму, са подршком за детектовање модификаторских тастера (`Ctrl`, `Alt`, `Shift`).

3.4.2 Интеграција са Ratatui библиотеком

Иако Ratatui пружа апстракцију вишег нивоа за креирање корисничких интерфејса, он се ослања на библиотеке нижег нивоа попут Crossterm-а за стварну интеракцију са терминалом [15]. Ratatui подржава више *backend*-а (Crossterm, Termion [20], Termwiz [21]), али је Crossterm подразумевани избор због своје вишеплатформске подршке и богатог скупа функција. Ова архитектура омогућава Ratatui-ју да се фокусира на логику приказа и распореда виџета, док Crossterm брине о детаљима као што су слање ANSI секвенци, управљање баферима и комуникација са оперативним системом.

3.4 Crossterm

Глава 4

Архитектура система

Архитектура је заснована на модуларном приступу који раздваја одговорности и омогућава лако проширивање и одржавање. Систем је организован у неколико независних модула који међусобно комуницирају, при чему сваки модул има јасну сврху и одговорност.

4.1 Општа архитектура

Имплементација следи архитектуру засновану на слојевима где се компоненте организују у хијерархијске нивое према њиховој апстракцији и одговорности. Ова архитектура промовише раздвајање одговорности и омогућава да промене у једном слоју имају минималан утицај на друге слојеве.

У наставку је дат сажет опис свих главних слојева:

Слој домена (`oxyd-domain`) — Дефинише основне типове података, *trait*-ове и апстракције које користе остали модули [16].

Слој прикупљача (`oxyd-collectors`) — Садржи имплементације за прикупљање различитих типова метрика из оперативног система. Сваки прикупљач чита податке из /`rgos` фајл система [22] и трансформише их у доменске типове.

Слој управљања процесима (`oxyd-process-manager`) — Пружа функционалност за управљање системским процесима, укључујући слање сигнала [5], терминацију и суспенђовање процеса.

Слој језгра (`oxyd-core`) — Оркестрира рад свих компоненти, управља животним циклусом прикупљача, и координира ток података између различитих делова система.

Слој корисничког интерфејса (`oxyd-tui`) — Имплементира терминалски кориснички интерфејс користећи `Ratatui` [14] и `Crossterm` [15] библиотеке. Одговоран је за приказ података и обраду корисничких улаза.

4.2 Модули система

4.2.1 `oxyd-domain`

Модул `oxyd-domain` представља срце система и дефинише све кључне типове података и интерфејсе. Овај модул нема зависности према другим модулима, што га чини стабилном основом за целокупан систем.

- Главни типови дефинисани у овом модулу укључују:
 - **SystemMetrics** — Структура која садржи све прикупљене метрике система у једном тренутку (CPU, меморија, мрежа, диск).

```
pub struct SystemMetrics {
    pub timestamp: DateTime<Utc>,
    pub system_info: SystemInfo,
    pub cpu: CpuMetrics,
    pub memory: MemoryInfo,
    pub disks: Vec<DiskMetrics>,
    pub network: NetworkMetrics,
    pub processes: ProcessMetrics,
}
```

Листинг 1: Приказ SystemMetrics структуре

- **Process** — Детаљне информације о појединачном процесу (PID, име, CPU употреба, меморија, стање).

```
pub struct Process {
    pub pid: u32,
    pub ppid: Option<u32>,
    pub name: String,
    pub command: String,
    pub arguments: Vec<String>,
    pub executable_path: Option<String>,
    pub working_dir: Option<String>,
    pub state: ProcessState,
    pub user: String,
    pub group: String,
    pub priority: i32,
    pub nice: i32,
    pub threads: u32,
    pub start_time: DateTime<Utc>,
    pub cpu_usage_percent: f64,
    pub memory_usage_bytes: u64,
    pub memory_usage_percent: f64,
    pub virtual_memory_bytes: u64,
    pub disk_write_bytes: u64,
    pub disk_read_bytes: u64,
    pub open_files: u32,
    pub open_connections: u32,
}
```

Листинг 2: Приказ Process структуре

- **CpuMetrics** — Метрике CPU-а укључујући укупну искоришћеност, искоришћеност по језгрима, и фреквенције.

```
pub struct CpuMetrics {  
    pub overall_usage_percent: f32,  
    pub cores: Vec<CpuCore>,  
    pub load_average: LoadAverage,  
    pub context_switches: u64,  
    pub interrupts: u64,  
}  
}
```

Листинг 3: Приказ CpuMetrics структуре

- **MemoryInfo** — Информације о RAM и Swap меморији (укупна, коришћена, доступна, кеширана).

```
pub struct MemoryInfo {  
    pub total_bytes: u64,  
    pub used_bytes: u64,  
    pub free_bytes: u64,  
    pub available_bytes: u64,  
    pub cached_bytes: u64,  
    pub buffers_bytes: u64,  
    pub swap_total_bytes: u64,  
    pub swap_used_bytes: u64,  
    pub swap_free_bytes: u64,  
    pub usage_percent: f32,  
    pub swap_usage_percent: f32,  
}  
}
```

Листинг 4: Приказ MemoryInfo структуре

- **NetworkMetrics** — Статистика мрежних интерфејса (пренети и примљени бајтови, пакети, грешке).

```
pub struct NetworkMetrics {  
    pub interfaces: Vec<NetworkInterface>,  
    pub stats: Vec<NetworkStats>,  
    pub total_bytes_sent: u64,  
    pub total_bytes_received: u64,  
    pub active_connections: Vec<NetworkConnection>,  
}  
}
```

Листинг 5: Приказ NetworkMetrics структуре

- **DiskMetrics** — Метрике дисковних уређаја (искоришћеност простора, I/O операције).

```
pub struct DiskMetrics {  
    pub info: DiskInfo,  
    pub io_stats: DiskIoStats,  
}  
}
```

Листинг 6: Приказ DiskMetrics структуре

4.2 Mogući siscīem

- Поред структура података, модул дефинише и trait-ове [9]:
 - ▶ **Collector** - Trait који служи за постављање основних метода колектора.

```
#[async_trait]
pub trait Collector: Send + Sync {
    // Јединствени идентификатор за колектор
    fn id(&self) -> &str;

    // Прикупља метрике система
    async fn collect(&self) -> Result<SystemMetrics, CollectorError>;

    // Провера да ли је колектор доступан на овом систему
    fn is_available(&self) -> bool;

    // Дефинише интервал очитавања метрика за колектор
    fn interval_ms(&self) -> u64 {
        1000
    }
}
```

Листинг 7: Приказ Collector trait-а из oxyd-domain модула

- **ProcessManager** - Trait који служи за постављање основних метода менаџера процеса.

```

#[async_trait]
pub trait ProcessManager: Send + Sync {
    // Приказује све активне процесе.
    async fn list_processes(&self) ->
        Result<Vec<u32>, ProcessError>;

    // Проналази детаљне информације о једном процесу
    // на основу идентификатора.
    async fn get_process(&self, pid: u32) ->
        Result<Process, ProcessError>;

    // Шаље сигнал за насиљан крај процеса.
    async fn kill_process(&self, pid: u32) ->
        Result<Process, ProcessError>;

    // Шаље сигнал прослеђен као параметар процесу.
    async fn send_signal(&self, pid: u32, signal: ProcessSignal) ->
        Result<ProcessActionResult, ProcessError>;

    // Поставља приоритет процеса.
    async fn send_priority(&self, pid: u32, priority: i32) ->
        Result<ProcessActionResult, ProcessError>;

    // Суспендује процес.
    async fn suspend_process(&self, pid: u32) ->
        Result<ProcessActionResult, ProcessError>;

    // Наставља процес.
    async fn continue_process(&self, pid: u32) ->
        Result<ProcessActionResult, ProcessError>;
}

}

```

Листинг 8: Приказ ProcessManager trait-а из oxyd-domain модула

4.2.2 oxyd.Collectors

Модул oxyd.Collectors садржи имплементације прикупљача за различите типове метрика. Сваки прикупљач имплементира Collector trait и специјализован је за читање одређеног типа података из /proc фајл система [22].

CpuCollector — Чита /proc/stat фајл и рашчлањује линије које садрже CPU статистике. Прати време проведено у различитим стањима (user, system, idle, iowait) за сваки CPU и израчунава проценат искоришћености.

MemoryCollector — Чита /proc/meminfo фајл који садржи детаљне информације о меморији. Парсира вредности као што су MemTotal, MemFree, MemAvailable, Buffers, Cached, SwapTotal, SwapFree.

NetworkCollector — Чита /proc/net/dev који садржи статистику за све мрежне интерфејсе. За сваки интерфејс прати примљене и послате бајтове, пакете, грешке и одбачене пакете.

DiskCollector — Комбинује информације из /proc/diskstats за I/O статистику и df команду за искоришћеност простора на монтираним фајл системима.

ProcessCollector — Итерира кроз /proc/[pid]/ директоријуме за све процесе и чита информације из фајлова као што су stat, status, cmdline, io.

Сви прикупљачи раде асинхроно користећи Tokio runtime [13], што омогућава паралелно прикупљање различитих метрика без блокирања.

```
#[async_trait]
impl Collector for CpuCollector {
    fn id(&self) -> &str {
        "cpu"
    }

    async fn collect(&self) -> Result<SystemMetrics, CollectorError> {
        let current_stats = self.read_cpu_stats().await?;
        let load_avg = self.read_load_average().await?;

        let mut previous_lock = self.previous_stats.lock().await;
        ...
    }

    fn is_available(&self) -> bool {
        std::path::Path::new("/proc/stat").exists()
    }
}
```

Листинг 9: Део кода једног од колектора

4.2.3 oxyd-process-manager

Модул oxyd-process-manager имплементира функционалност за управљање процесима [5].

Главне функције овог модула су:

- Слање SIGTERM сигнала за љубазно гашење процеса

- Слање SIGKILL сигнала за насиљно гашење
- Слање SIGSTOP сигнала за суспендовање процеса
- Слање SIGCONT сигнала за наставак суспендованог процеса
- Провера постојања и валидности процеса

```

#[async_trait]
impl ProcessManager for LinuxProcessManager {
    async fn list_processes(&self) -> Result<Vec<u32>, ProcessError> {
        let mut pids = Vec::new();
        let mut entries = fs::read_dir("/proc").await
            .map_err(|e| ProcessError::ListFailed(format!("Failed to
read /proc: {}", e)))?;

        while let Some(entry) = entries.next_entry().await
            .map_err(|e| ProcessError::ListFailed(format!("Failed to
read entry: {}", e)))? {
            let file_name = entry.file_name();
            let file_name_str = file_name.to_string_lossy();

            if let Ok(pid) = file_name_str.parse::<u32>() {
                let stat_path = format!("/proc/{}/stat", pid);
                if Path::new(&stat_path).exists() {
                    pids.push(pid);
                }
            }
        }

        Ok(pids)
    }
}

```

Листинг 10: Приказ функције за добављање процеса

4.2.4 oxyd-core

Модул oxyd-core представља централну компоненту која оркестрира рад целокупног система. Овај модул је одговоран за:

- Иницијализацију свих прикупљача
- Периодично покретање прикупљања метрика
- Агрегацију података из различитих извора
- Пружање API-ја за остале компоненте

Engine — Главна структура која управља свим прикупљачима. Периодично позива `collect()` методу на сваком прикупљачу и складиши резултате. Користи `broadcast` канал из Tokio библиотеке за дистрибуцију метрика.

```
pub struct Engine {
    collectors: Arc<RwLock<Vec<Box<dyn Collector>>>,
    process_manager: Arc<dyn ProcessManager>,
    metrics_tx: broadcast::Sender<SystemMetrics>,
    config: Config,
    running: Arc<RwLock<bool>>,
}
```

Листинг 11: Приказ структуре Engine

4.2.5 oxyd-tui

Модул oxyd-tui имплементира терминалски кориснички интерфејс користећи Ratatui [14] и Crossterm [15]. Архитектура је заснована на Flux обрасцу [23], који користи једносмеран ток података и предвидљиво управљање стањем апликације.

4.2.5.1 Flux архитектура

Flux је образац креиран од стране Facebook-а за развој корисничких интерфејса. Основна идеја је једносмеран ток података (енг. *unidirectional data flow*) где акције (енг. *actions*) покрећу промене стања, а стање се затим одражава у приказу.

oxyd-tui модул имплементира следеће Flux компоненте:

Actions — Представљају све могуће догађаје и намере у апликацији. Ово укључује корисничке интеракције (притиске тастера, кликове миша), системске догађаје (ажурирања метрика, промене величине терминала), и интерне догађаје (таймаути, грешке).

Dispatcher — Централна тачка за рутирање акција. Прима акције из различитих извора и прослеђује их одговарајућим handler-има који ажурирају стање.

Store — Чува комплетно стање апликације. У Oxyd-у, ово је AppState структура која садржи тренутни таб, метрике, селекције, филтере и сва друга релевантна стања.

View — Рендерује кориснички интерфејс на основу тренутног стања. Користи Ratatui вицете [19] за приказ података и не мења стање директно.

4.2.5.2 Предности Flux архитектуре

Коришћење Flux обрасца у oxyd-tui модулу пружа неколико кључних предности:

Предвидљиво стање — Једносмеран ток података чини промењиво стање лакшим за праћење и дебаговање. Свака промена стања се дешава кроз акције које се могу лако логовати и пратити.

Развојене одговорности — View слој је потпуно одвојен од логике управљања стањем. View само рендерује и генерише акције, док Dispatcher брине о ажурирању стања.

Тестабилност — Свака компонента се може независно тестирати. Dispatcher логика се тестира слањем акција и проверавањем резултујућег стања, док се View тестира провером да исправно рендерује дато стање.

Проширивост — Додавање нових функција захтева само дефинисање нових акција и њихових handler-а, без промене постојећег кода.

4.3 Ток података

Ток података кроз Oxyd систем прати јасан образац од прикупљања до приказа:

- Прикупљање** — Engine периодично активира све регистроване колекторе. Сваки колектор асинхроно чита свој извор података из /rgos фајл система.
- Агрегација** — Подаци из свих прикупљача се комбинују у јединствену SystemMetrics структуру која представља комплетан приказ стања система у датом тренутку.
- Дистрибуција** — Engine емитује нове метрике преко broadcast канала на који су претплаћене компоненте корисничког интерфејса.
- Приказ** — TUI компонента прима ажурирање, ажурира своје интерно стање, и рендерује нови приказ.

Овај приступ обезбеђује да кориснички интерфејс остаје ажуран јер прикупљање података ради у позадини и не блокира нит корисничког интерфејса. Коришћење broadcast канала омогућава да више компоненти UI-ја могу независно реаговати на иста ажурирања.

4.4 Обрасци дизајна

У имплементацији Oxyd-а коришћено је неколико добро познатих образаца дизајна:

Observer — Broadcast канали имплементирају observer образац где TUI компоненте могу да се претплате на ажурирања метрика.

Command — Обрада корисничких улаза користи command образац где сваки input производи Action enum који се затим извршава.

Dependency Injection — Сви модули примају своје зависности кроз конструкторе или методе, што олакшава тестирање и флексибилност.

4.5 Синхронизација

За синхронизацију се користе Tokio примитиви [18]:

4.5 Синхронизација

Mutex и RwLock — За заштиту дељених структура података.

Channels — За комуникацију између задатака.

Arc — За дељење података између задатака.

Раст-ов систем власништва и позајмљивања [9] гарантује да не може доћи до data race-ова, што чини конкурентни код безбедним по дизајну.

Глава 5

Имплементација

Ово поглавље детаљно описује кључне аспекте имплементације, са фокусом на конкретна техничка решења, алгоритме и изазове. Приказани су најважнији делови кода са објашњењима имплементационих одлука и решењима специфичних проблема.

5.1 Прикупљање системских метрика

Сви колектори читају податке из Линукс /proc псеудо фајл система [22] који пружа интерфејс ка kernel подацима. Сваки колектор имплементира Collector trait [9] и ради асинхроно користећи Tokio runtime [13].

5.1.1 Читање из /proc фајл система

Linux /proc фајл систем пружа текстуални интерфејс ка kernel подацима. Иако изгледају као обични фајлови, подаци се генеришу динамички приликом читања. Читање се обавља асинхроно користећи tokio::fs модул:

```
use tokio::fs;

async fn read_cpu_stats(&self) -> Result<Vec<CpuStates>, CollectorError> {
    let content = fs::read_to_string("/proc/stat")
        .await
        .map_err(|e| CollectorError::AccessError("/proc/stat".
            to_string(), e.to_string()))?;

    let mut stats: Vec<CpuStates> = Vec::new();

    for line in content.lines() {
        if line.starts_with("cpu") && line.chars().nth(3).map_or(false,
|c| c.is_whitespace() || c.is_numeric()) {
            let parts: Vec<&str> = line.split_whitespace().collect();
            if parts.len() > 0 {
                stats.push(CpuStates {
                    user: parts[1].parse().unwrap_or(0),
                    nice: parts[2].parse().unwrap_or(0),
                    system: parts[3].parse().unwrap_or(0),
                    idle: parts[4].parse().unwrap_or(0),
                    iowait: parts[5].parse().unwrap_or(0),
                    irq: parts[6].parse().unwrap_or(0),
                    softirq: parts[7].parse().unwrap_or(0),
                    steal: parts.get(8).and_then(|s|
s.parse().ok()).unwrap_or(0),
                    guest: parts.get(9).and_then(|s|
s.parse().ok()).unwrap_or(0),
                    guest_nice: parts.get(10).and_then(|s|
s.parse().ok()).unwrap_or(0),
                })
            }
        }
    }

    Ok(stats)
}
```

Листинг 12: Асинхроно читање из /proc файл система

Овај приступ користи асинхрони I/O [17] што омогућава да друге операције настављају док се чека на читање фајла. Грешке се мапирају у специјализоване типове грешака који садрже контекст о томе који фајл је узроковао проблем.

5.1.2 CPU метрике

CPU метрике се прикупљају читањем /proc/stat фајла који садржи агрегиране статистике за све CPU-ове у систему, као и појединачне статистике за сваки CPU. Формат је следећи:

```
cputime 74608 2520 24433 1117073 6176 4054 0 0 0 0  
cpu0 37784 1260 12062 558377 3089 2027 0 0 0 0  
cpu1 36824 1260 12371 558696 3087 2027 0 0 0 0
```

Свака линија садржи 10 бројева који представљају време (у јединицама од 1/100 секунде) проведено у различитим стањима: user, nice, system, idle, iowait, irq, softirq, steal, guest, guest_nice.

CPU искоришћеност се не може одредити из једног мерења већ захтева два узастопна мерења и рачунање разлике. Алгоритам је следећи:

```
async fn calculate_usage(&self, current: &CpuStates,  
previous: &CpuStates) -> f32 {  
  
    let prev_total = previous.user + previous.nice +  
                    previous.system + previous.idle  
                    + previous.iowait + previous.irq +  
                    previous.softirq + previous.steal;  
  
    let curr_total = current.user + current.nice  
                    + current.system + current.idle  
                    + current.iowait + current.irq +  
                    current.softirq + current.steal;  
  
    let prev_idle = previous.idle + previous.iowait;  
    let curr_idle = current.idle + current.iowait;  
  
    let total_diff = curr_total.saturating_sub(prev_total);  
    let idle_diff = curr_idle.saturating_sub(prev_idle);  
  
    if total_diff == 0 {  
        return 0.0;  
    }  
  
    let usage_diff = total_diff.saturating_sub(idle_diff);  
    (usage_diff as f32 / total_diff as f32) * 100.0  
}
```

Листинг 13: Рачунање искоришћености процесора

5.1.3 Метрике меморије

Информације о меморији се налазе у `/proc/meminfo` фајлу који садржи преко 40 различитих метрика. За ову имплементацију су релевантне следеће:

MemTotal:	16384000 kB
MemFree:	2048000 kB
MemAvailable:	8192000 kB
Buffers:	512000 kB
Cached:	4096000 kB
SwapTotal:	8192000 kB
SwapFree:	8000000 kB

Разлика између `MemFree` и `MemAvailable` је важна. `MemFree` показује потпуно некоришћену меморију, док `MemAvailable` укључује и меморију која се користи за cache али се може лако ослободити. За кориснике је `MemAvailable` релевантнија метрика јер показује колико меморије је заиста доступно за нове апликације.

5.1.4 Информације о процесима

Прикупљање информација о процесима је најкомплекснија операција јер захтева читање више фајлова за сваки процес. За сваки процес се читају следећи фајлови из `/proc/[pid]/` директоријума:

- `stat` — основне статистике (PID, име, стање, CPU време)
- `status` — детаљне информације (меморија, UID, GID)
- `cmdline` — комплетна командна линија
- `io` — I/O статистике (опционо, захтева root)

Прикупљање информација о процесима је I/O интензивна операција јер за 100 процеса треба прочитати најмање 300 фајлова. Асинхрони приступ омогућава да се читања обављају паралелно, што значајно убрзава овај процес.

5.2 Управљање процесима

Модул за управљање процесима омогућава слање различитих сигнала процесима [5] и пружа заштиту од случајног гашења критичних системских процеса. За разлику од директног коришћења `libc::kill()` системског позива, имплементација користи `kill` команду преко Tokio Command API-ја [13], што пружа бољу изолацију и безбедност.

```
use tokio::process::Command;

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum ProcessSignal {
    Kill,           // SIGKILL - принудно гашење
    Terminate,      // SIGTERM - љубазно гашење
    Stop,           // SIGSTOP - паузирање процеса
    Continue,       // SIGCONT - наставак паузираног процеса
    Interrupt,     // SIGINT - прекид
    Quit,           // SIGQUIT - излазак из процеса
    Custom(i32),   // Произвољан сигнал број
}
```

Листинг 14: Дефиниција типова сигнала за процесе

5.2.1 Имплементација слања сигнала

Главна метода за слање сигнала користи `kill` команду преко Tokio Command API-ја. Овај приступ има неколико предности у односу на директно коришћење `libc::kill()`:

1. **Избегава несигуран код** — Не захтева unsafe блокове.
2. **Боља изолација** — Користи постојећу `kill` команду која је добро тестирана.
3. **Флексибилност** — Лакше подржава различите опције и сигнале.
4. **Безбедност** — Kernel проверава дозволе независно од нашег кода.

У наставку следи приказ конкретне методе за слање сигнала одређеном процесу:

```
async fn send_signal(&self, pid: u32, signal: ProcessSignal) ->
Result<ProcessActionResult, ProcessError> {
    let signal_name = match signal {
        ProcessSignal::Kill => "SIGKILL",
        ProcessSignal::Terminate => "SIGTERM",
        ProcessSignal::Stop => "SIGSTOP",
        ProcessSignal::Continue => "SIGCONT",
        ProcessSignal::Interrupt => "SIGINT",
        ProcessSignal::Quit => "SIGQUIT",
        ProcessSignal::Hangup => "SIGHUP",
        ProcessSignal::Custom(_) => "CUSTOM",
    };

    let process = self.get_process(pid).await?;
    if self.protected_processes.contains(&process.name) {
        return Err(ProcessError::PermissionDenied(pid));
    }
    let output = tokio::process::Command::new("kill")
        .arg(format!("-{}", signal_name))
        .arg(pid.to_string())
        .output()
        .await
        .map_err(|e| ProcessError::ActionFailed(
            "kill".to_string(),
            pid,
            e.to_string()
        ))?;
    if !output.status.success() {
        let stderr = String::from_utf8_lossy(&output.stderr);
        return Err(ProcessError::ActionFailed(
            "kill".to_string(),
            pid,
            stderr.to_string()
        ));
    }
}

Ok(ProcessActionResult {
    pid,
    action: ProcessAction::Terminate,
    success: true,
    message: Some(format!("Sent {} to process {}", signal_name, pid)),
    timestamp: Utc::now(),
})
}
```

Листинг 15: Метода за слање сигнала одређеном процесу

5.3 Руковање грешкама

Иако је могуће ручно имплементирати типове за грешке, `thiserror` [24] библиотека значајно поједностављује овај процес. Ова библиотека пружа процедурални макро који автоматски генерише имплементације потребних trait-ова за custom error типове.

`thiserror` пружа следеће могућности:

error(...) атрибут — Дефинише поруку грешке са подршком за форматирање. Бројеви у витичастим заградама ({0}, {1}) се односе на поља enum варијанте.

from атрибут — Аутоматски генерише From trait имплементацију, што омогућава коришћење ? оператора за конверзију из једног типа грешке у други.

Ова имплементација користи хијерархијску структуру грешака где `OxydError` представља главни error тип који обједињује све остale специјализоване типове грешака. Овај приступ пружа:

1. **Модуларност** — Сваки модул дефинише своје специфичне грешке
2. **Композицију** — Главни еггр тип може конвертовати све специјализоване грешке
3. **Детаљност** — Свака грешка садржи контекст специфичан за свој домен
4. **Пропагацију** — Аутоматска конверзија омогућава лако пропагирање грешака кроз слојеве

```
#[derive(Error, Debug)]
pub enum OxydError {
    #[error("Collector error: {0}")]
    Collector(#[from] CollectorError),

    #[error("Process manager error: {0}")]
    ProcessManager(#[from] ProcessError),

    #[error("Plugin error: {0}")]
    PluginError(#[from] PluginError),

    #[error("Configuration error: {0}")]
    ConfigError(#[from] ConfigError),

    #[error("IO error: {0}")]
    IOError(#[from] std::io::Error),

    #[error("Unknown error: {0}")]
    Unknown(String),
}
```

Листинг 16: Приказ имплементације главног коренског типа за грешке

Затим сваки следећи подтип детаљније описује грешку која може настати у том слоју имплементације.

```
#[derive(Error, Debug)]
pub enum CollectorError {
    #[error("Failed to read system information: {0}")]
    SystemInfoError(String),

    #[error("Failed to access {0}: {1}")]
    AccessError(String, String),

    #[error("Parse error for {0}: {1}")]
    ParseError(String, String),

    #[error("Collector {0} not available on this system")]
    NotAvailable(String),
    #[error("Timeout while collecting {0}")]
    Timeout(String),
}
```

Листинг 17: Пример једне имплементације типа грешке

5.4 Асинхроно прикупљање метрика

Имплементација користи `tokio::join!` [13] за асинхроно прикупљање свих метрика. Прикупљање метрика је приказано у наредном листингу:

```

async fn collect(&self) -> Result<SystemMetrics, CollectorError> {
    let (cpu_result, memory_result, process_result,
        network_result, disk_result)
        = tokio::join!(
            self.cpu_collector.collect(),
            self.memory_collector.collect(),
            self.process_collector.collect(),
            self.network_collector.collect(),
            self.disk_collector.collect()
        );

    let system_info = self.get_system_info().await;

    let cpu_metrics = cpu_result?.cpu;
    let memory_metrics = memory_result?.memory;
    let process_metrics = process_result?.processes;
    let network_metrics = network_result?.network;
    let disk_metrics = disk_result?.disks;

    Ok(SystemMetrics {
        timestamp: Utc::now(),
        system_info,
        cpu: cpu_metrics,
        memory: memory_metrics,
        disks: disk_metrics,
        network: network_metrics,
        processes: process_metrics,
    })
}

```

Листинг 18: Прикупљање свих метрика система

5.5 Кориснички интерфејс

Кориснички интерфејс је подељен у табове, где сваки таб представља појединачну метрику која се прикупља [14]. Сваки таб се посебно имплементира са својим вице-тима [19] и опцијама. На крају, неопходно их је повезати како би се омогућило њихово смењивање.

```
pub enum Tab {
    Overview,
    Cpu,
    Memory,
    Processes,
    Network,
    Disk,
    Notifications,
    Settings,
}
impl Tab {
    pub fn next(&self) -> Self {
        match self {
            Tab::Overview => Tab::Cpu,
            Tab::Cpu => Tab::Memory,
            Tab::Memory => Tab::Processes,
            Tab::Processes => Tab::Network,
            Tab::Network => Tab::Disk,
            Tab::Disk => Tab::Notifications,
            Tab::Notifications => Tab::Settings,
            Tab::Settings => Tab::Overview,
        }
    }
    pub fn previous(&self) -> Self {
        match self {
            Tab::Overview => Tab::Settings,
            Tab::Cpu => Tab::Overview,
            Tab::Memory => Tab::Cpu,
            Tab::Processes => Tab::Memory,
            Tab::Network => Tab::Processes,
            Tab::Disk => Tab::Network,
            Tab::Notifications => Tab::Disk,
            Tab::Settings => Tab::Notifications,
        }
    }
    pub fn title(&self) -> &str {
        match self {
            Tab::Overview => "Overview",
            Tab::Cpu => "CPU",
            ...
        }
    }
}
```

Листинг 19: Дефиниција таба и њихово повезивање

Глава 6

Пример коришћења

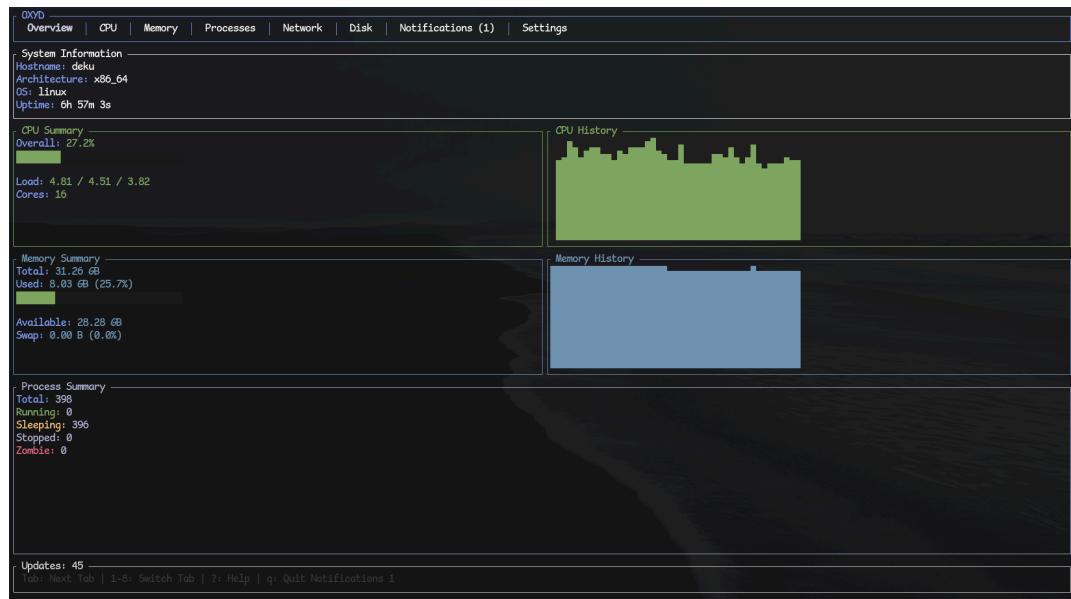
У овом поглављу ће бити приказани сви имплементирани екрани корисничког интерфејса.

6.1 Покретање апликације

Апликација се покреће из терминала без додатних аргумента:

```
$ oxyd
```

При покретању, апликација иницијализује све колекторе и приказује почетни таб са најосновнијим метрикама:



Слика 1: Почетна страница.

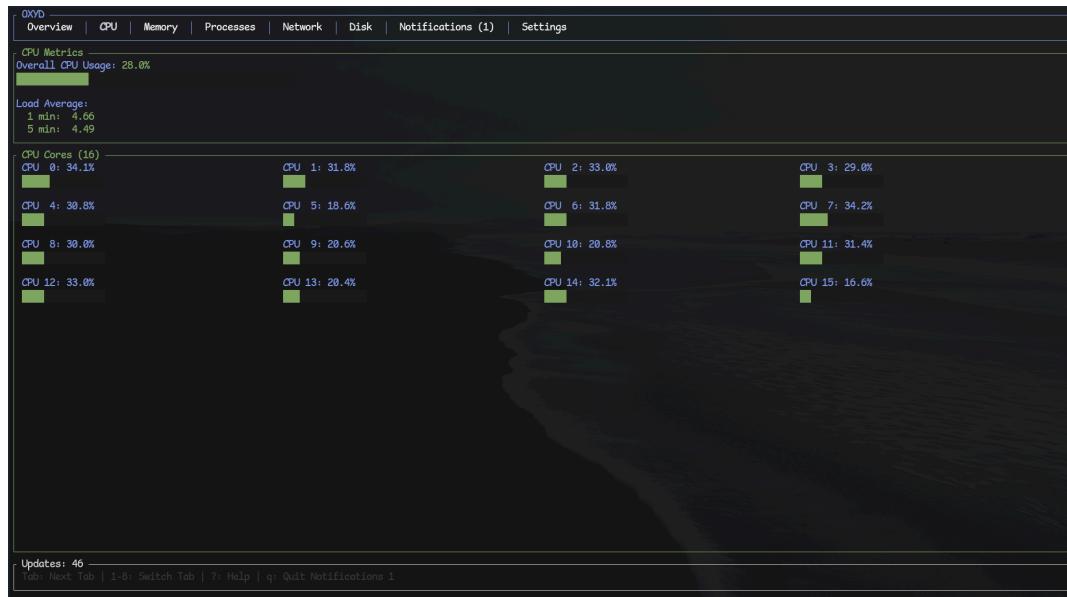
6.2 Навигација кроз табове

Имплементација користи навигацију која омогућава брз приступ различитим категоријама метрика. Тренутно активан таб је истакнут другом бојом у header-у.

6.3 Мониторинг процесора

CPU таб приказује детаљне информације о искоришћености процесора.

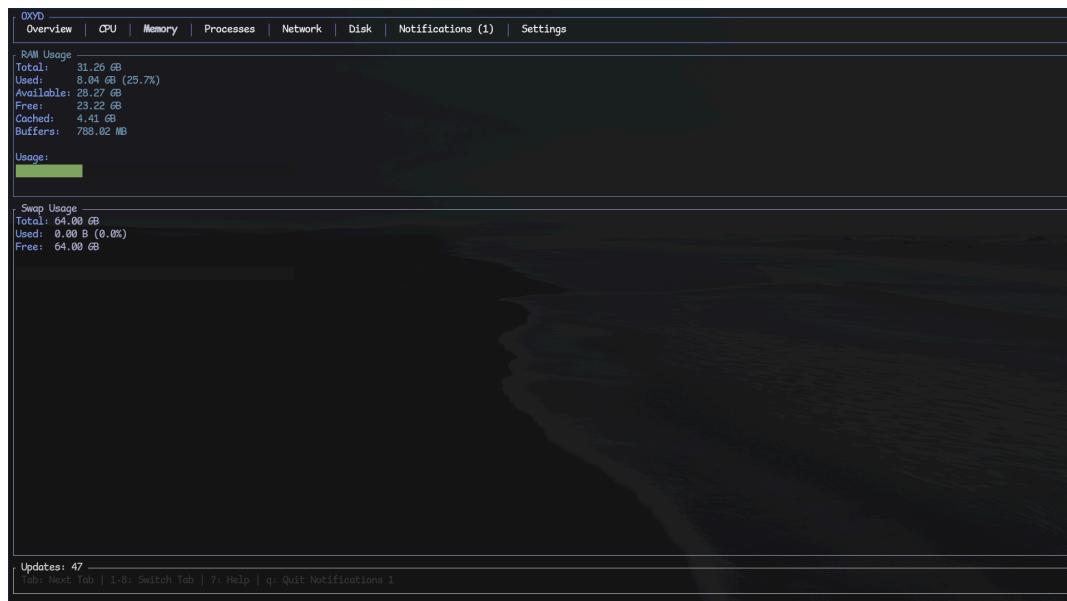
6.3 Мониторинг процесора



Слика 2: Приказ странице за увид у искоришћење процесора.

6.4 Мониторинг меморије

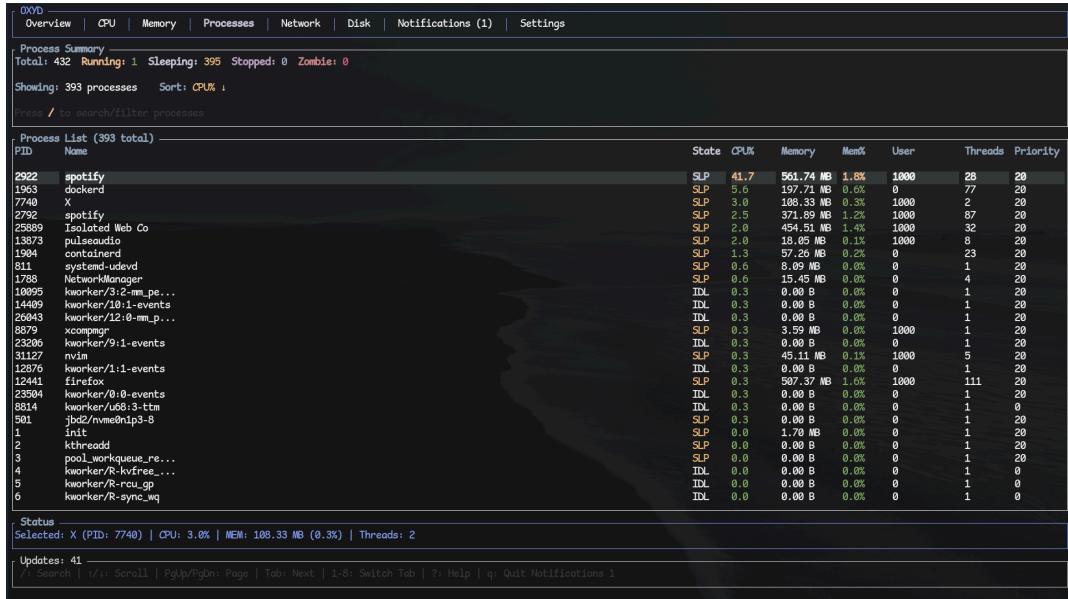
Memory таб приказује детаљне информације о коришћењу RAM меморије и swap простора.



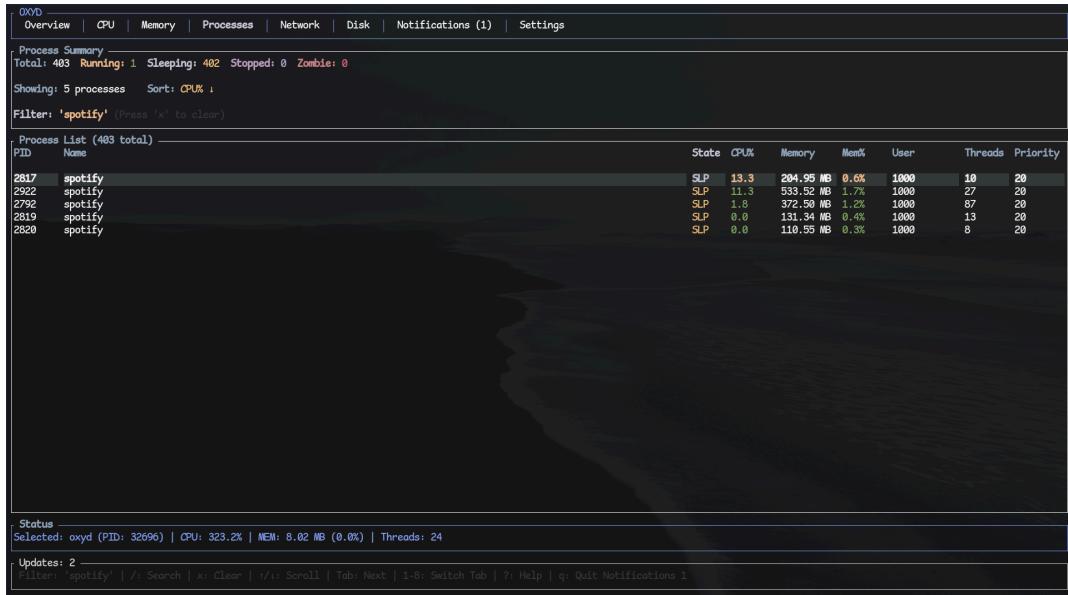
Слика 3: Приказ странице за увид у меморију.

6.5 Управљање процесима

Таб са процесима пружа детаљан увид у стање свих процеса оперативног система. Одавде је могуће филтрирање и руковање свим процесима.

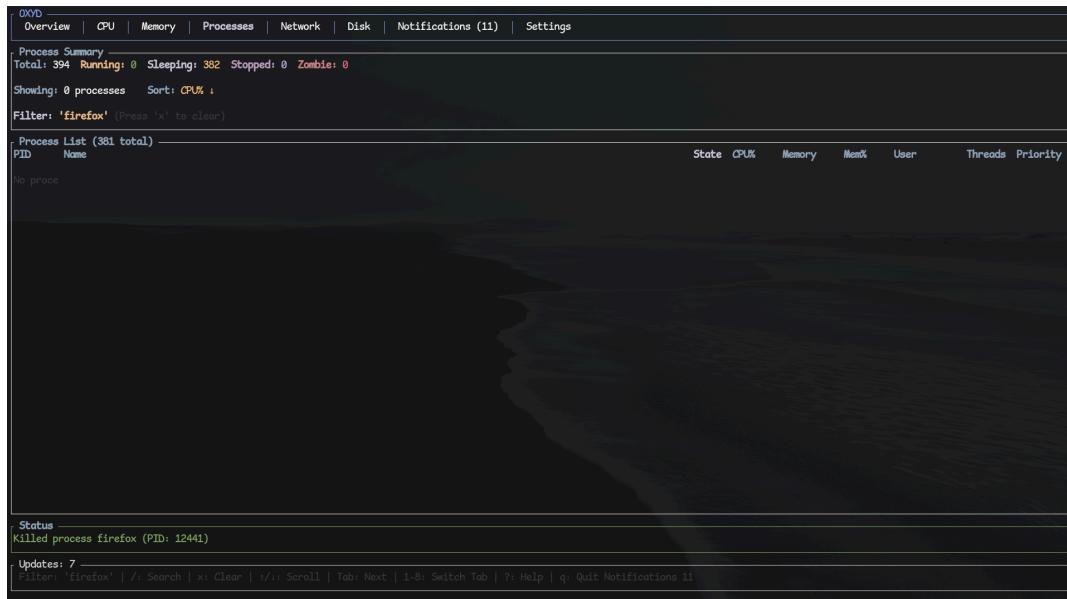


Слика 4: Приказ свих процеса.



Слика 5: Филтрирање процеса.

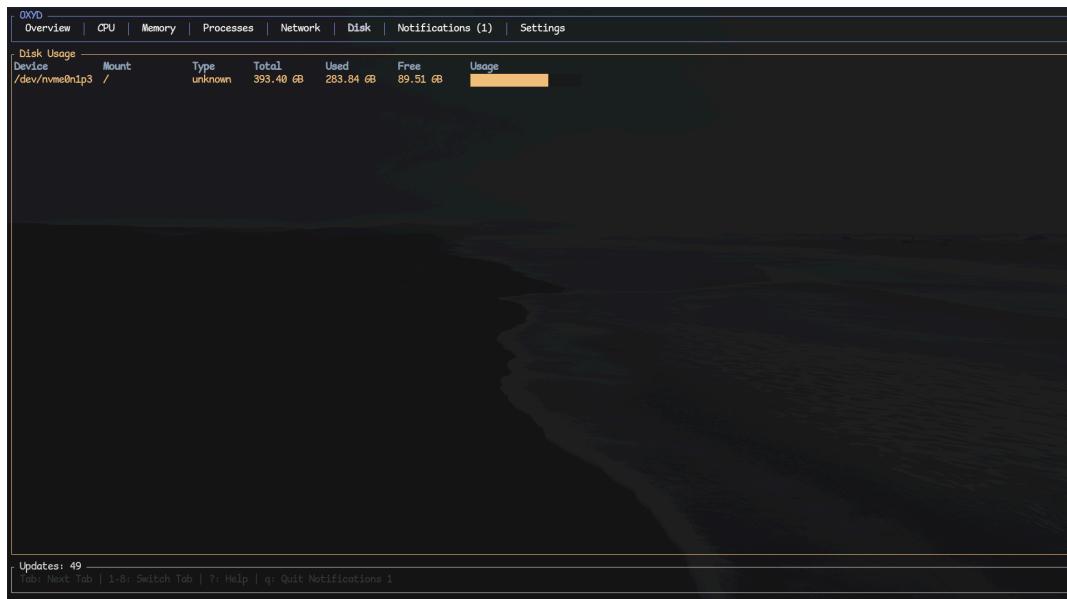
6.5 Управљање процесима



Слика 6: Слање сигнала end процесу.

6.6 Мониторинг дискова

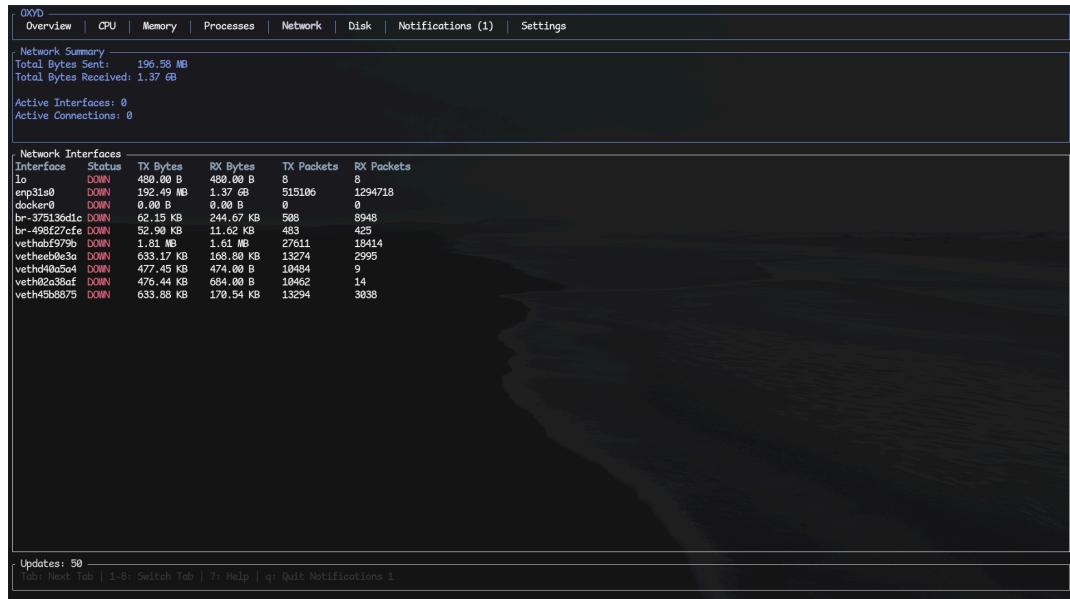
Disk таб приказује информације о искоришћености простора на диску.



Слика 7: Приказ странице за увид у стање диска.

6.7 Мониторинг мреже

Network таб приказује информације о тренутно доступним мрежним интерфејсима.

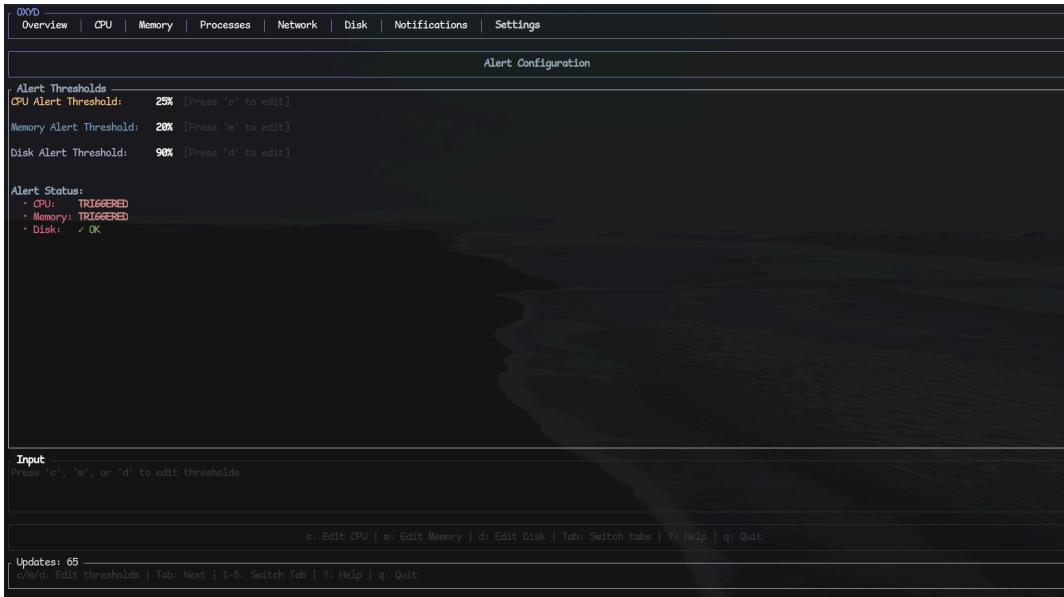


Слика 8: Приказ странице за увид у мрежне интерфејсе.

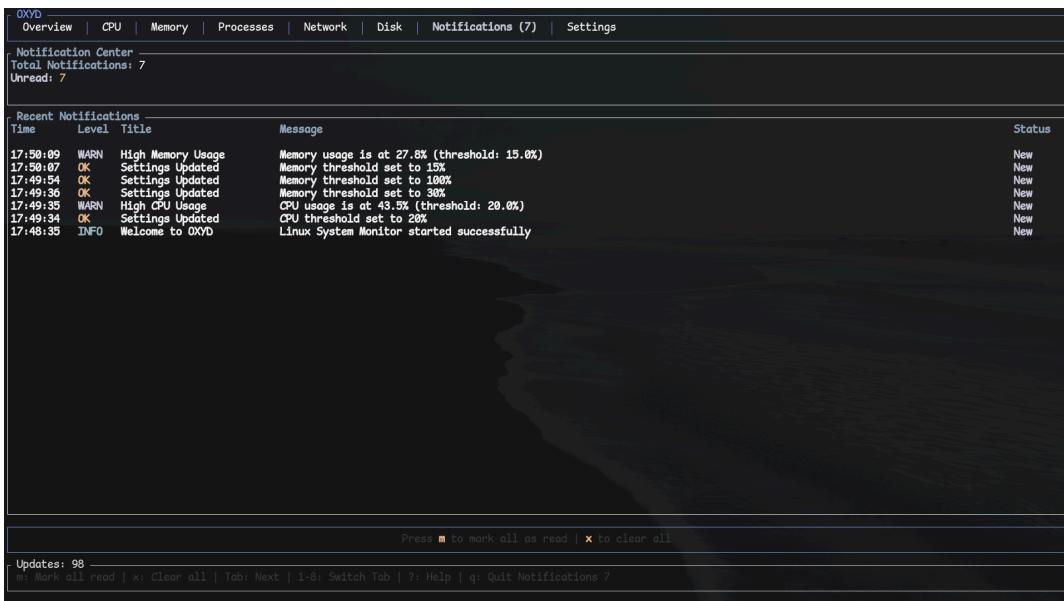
6.8 Нотификације и подешавања

Имплементација подржава подешавање и слање обавештења за процесор, меморију или диск уколико метрика за њихово искоришћење пређе одређени проценат.

6.8 Ноћификације и њодешавања



Слика 9: Приказ странице за подешавања нотификација.

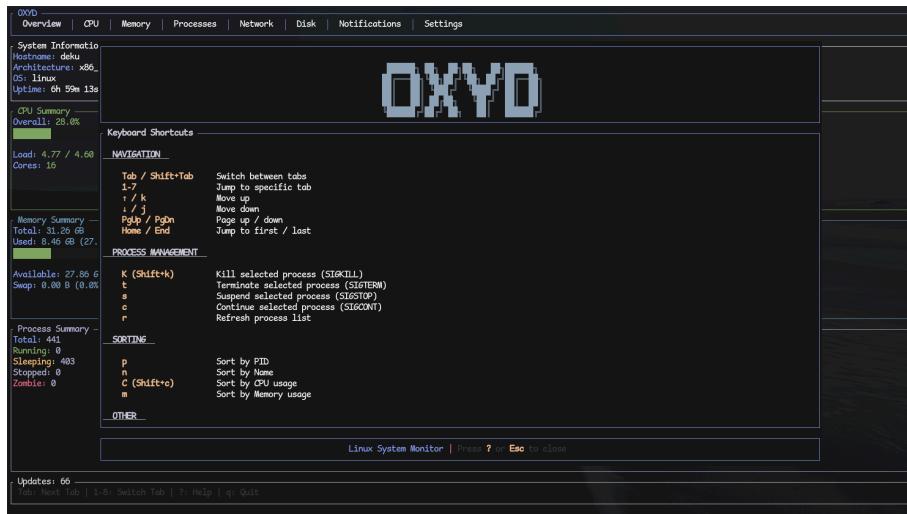


Слика 10: Приказ странице за увид у нотификације.

6.9 Контроле и пречице

Имплементација пружа интуитиван систем пречица са тастатуре који омогућава брзу навигацију и управљање свим аспектима апликације без потребе за мишем.

Притиском на тастер ?, отвара се специјални прозор са свим могућим опцијама за интеракцију са корисничким интерфејсом. Овај екран за помоћ је увек доступан из било ког таба и пружа брз преглед свих доступних команда.



Слика 11: Приказ екрана за помоћ са свим доступним пречицама и акцијама

Списак свих акција је приказан у наредној табели. Пречице су организоване по категоријама за лакшу навигацију и коришћење.

Тастер	Акција
Навигација између табова	
1 - 8	Директан прелазак на таб по броју (1 = Overview, 2 = CPU, итд.)
Tab	Прелазак на следећи таб (десно)
Shift+Tab	Прелазак на претходни таб (лево)
Навигација кроз листе	
↑ / k	Помери селекцију на горњи елемент
↓ / j	Помери селекцију на доњи елемент
Управљање процесима	
Shift+k	Угаси селектовани процес (SIGKILL - принудно гашење)
Филтрирање и претрага	
/	Активирај режим филтрирања процеса
Esc	Изађи из режима филтрирања и врати се на пун приказ
Сортирање	
p	Сортирај процесе по PID-у
n	Сортирај процесе по имену
c	Сортирај процесе по CPU искоришћености
m	Сортирај процесе по коришћењу меморије
Опште контроле	
?	Прикажи екран за помоћ са свим пречицама
q	Изађи из апликације
Ctrl+C	Принудни излазак из апликације

Табела 2: Комплетан преглед свих доступних пречица у Oxyd апликацији

Глава 7

Закључак

Овај рад је представио модеран систем за мониторинг Линукс система развијен у програмском језику Раст. Основни циљ био је креирање ефикасног, безбедног и кориснички пријатног алата који комбинује предности традиционалних мониторинг решења са могућностима модерних технологија.

Имплементација је показала да Раст програмски језик представља одличан избор за развој системских алата. Систем власништва елиминисао је целу класу грешака везаних за управљање меморијом, док је Tokio омогућио ефикасну конкурентност без блокирајућих операција. Модуларна архитектура организована у јасно одвојене слојеве (domain, collectors, process manager, core, tui) олакшава одржавање и проширивање система.

Као пројекат отвореног кода, пружа солидну базу за даљи развој у неколико праваца. Будући циљеви укључују додавање подршке за конфигурационе фајлове и побољшано филтрирање процеса. Такође, у плану је и мултиплатформска подршка (Windows, macOS), мониторинг удаљеног рачунара и побољшани систем алармирања.

Списак слика

Слика 1	Почетна страница.	33
Слика 2	Приказ странице за увид у искоришћење процесора.	34
Слика 3	Приказ странице за увид у меморију.	34
Слика 4	Приказ свих процеса.	35
Слика 5	Филтрирање процеса.	35
Слика 6	Слање сигнала end процесу.	36
Слика 7	Приказ странице за увид у стање диска.	36
Слика 8	Приказ странице за увид у мрежне интерфејсе.	37
Слика 9	Приказ странице за подешавања нотификација.	38
Слика 10	Приказ странице за увид у нотификације.	38
Слика 11	Приказ екрана за помоћ са свим доступним пречицама и акцијама	39

Списак листинга

Листинг 1	Приказ SystemMetrics структуре	14
Листинг 2	Приказ Process структуре	14
Листинг 3	Приказ CpuMetrics структуре	15
Листинг 4	Приказ MemoryInfo структуре	15
Листинг 5	Приказ NetworkMetrics структуре	15
Листинг 6	Приказ DiskMetrics структуре	15
Листинг 7	Приказ Collector trait-а из oxyd-domain модула	16
Листинг 8	Приказ ProcessManager trait-а из oxyd-domain модула	17
Листинг 9	Део кода једног од колектора	18
Листинг 10	Приказ функције за добављање процеса	19
Листинг 11	Приказ структуре Engine	20
Листинг 12	Асинхроно читање из /proc файл система	24
Листинг 13	Рачунање искоришћености процесора	25
Листинг 14	Дефиниција типова сигнала за процесе	27
Листинг 15	Метода за слање сигнала одређеном процесу	28
Листинг 16	Приказ имплементације главног коренског типа за грешке	29
Листинг 17	Пример једне имплементације типа грешке	30
Листинг 18	Прикупљање свих метрика система	31
Листинг 19	Дефиниција таба и њихово повезивање	32

Списак табела

Табела 1 Упоредни преглед алата за системски мониторинг	5
Табела 2 Комплетан преглед свих доступних пречица у Oxyd апликацији	40

Списак коришћених скраћеница

Скраћеница	Опис
ANSI	American National Standards Institute (Амерички национални институт за стандарде)
API	Application Programming Interface (апликациони програмски интерфејс)
ASCII	American Standard Code for Information Interchange (амерички стандардни код за размену информација)
BSD	Berkeley Software Distribution (Беркли софтверска дистрибуција)
CPU	Central Processing Unit (централна процесорска јединица)
CSV	Comma-Separated Values (вредности раздвојене зарезом)
GCC	GNU Compiler Collection (ГНУ колекција компајлера)
GID	Group Identifier (идентификатор групе)
GPU	Graphics Processing Unit (графичка процесорска јединица)
I/O	Input/Output (улауз/излауз)
macOS	Macintosh Operating System (Мекинтош оперативни систем)
PID	Process Identifier (идентификатор процеса)
RAM	Random Access Memory (меморија са директним приступом)
SIGCONT	Signal Continue (сигнал за наставак процеса)
SIGINT	Signal Interrupt (сигнал за прекид процеса)
SIGKILL	Signal Kill (сигнал за принудно гашење процеса)
SIGQUIT	Signal Quit (сигнал за излазак из процеса)
SIGSTOP	Signal Stop (сигнал за заустављање процеса)
SIGTERM	Signal Terminate (сигнал за љубазно гашење процеса)
SIGHUP	Signal Hangup (сигнал за прекид везе)

Скраћеница	Опис
TOML	Tom's Obvious Minimal Language (Томов очигледан минималан језик)
TUI	Terminal User Interface (терминалски кориснички интерфејс)
UI	User Interface (кориснички интерфејс)
UID	User Identifier (идентификатор корисника)
UNIX	Uniplexed Information Computing System (оперативни систем)

Списак коришћених појмова

Појам	Објашњење
Асинхрони рад	Рад који се изводи независно од главног тока извршавања, омогућавајући наставак других операција без чекања на његов завршетак.
Borrow checker	Механизам у Раст компјултеру који проверава правила власништва и позајмљивања током компилације, спречавајући грешке у управљању меморијом.
Broadcast канал	Комуникациони канал који омогућава слање порука од једног пошиљаоца ка више прималаца истовремено.
Cross-platform	Способност софтвера да се извршава на више различитих оперативних система из истог кода.
Dangling pointer	Висећи показивач — референца која показује на меморијску адресу која више није валидна.
Data race	Трка до података — ситуација у конкурентном програмирању када две или више нити истовремено приступају истом меморијском простору, од којих барем једна врши писање.
Double-free	Грешка двоструког ослобађања меморије, када се исти меморијски блок ослободи два пута.
Flux	Архитектурни образац за развој корисничких интерфејса који користи једносмеран ток података.
Future	Тип у Раству који представља вредност која ће бити доступна у будућности, основа асинхроног програмирања.
Garbage collector	Сакупљач ђубрета — аутоматски механизам за ослобађање меморије коју програм више не користи.
Immediate mode rendering	Режим рендеровања у коме се кориснички интерфејс исцртава изнова за сваку слику без чувања стања вицета.
Kernel	Језгро оперативног система које управља хардверским ресурсима и пружа основне сервисе програмима.
Lifetime	Животни век — Раствова апстракција која описује колико дugo је нека референца валидна.

Појам	Објашњење
Mutex	Механизам узајамног искључивања који обезбеђује да само једна нит у датом тренутку приступа заштићеном ресурсу.
Observer обра- зец	Образац дизајна у коме објекат одржава листу зависних посматрача и аутомат- ски их обавештава о променама стања.
Ownership	Власништво — систем у Расту где свака вредност има тачно једног власника који контролише њен животни век.
/proc фајл си- стем	Виртуелни фајл систем у Линуксу који пружа текстуални интерфејс ка подацима кернела о процесима и систему.
Raw mode	Сирови режим рада терминала у коме се сваки притисак тастера чита поједи- начно без баферовања и обраде.
Retained mode rendering	Режим рендеровања у коме се виџети чувају као објекти у меморији и поједи- начно модификују.
Runtime	Извршно окружење које пружа инфраструктуру за извршавање програма, у контексту Tokio-а — асинхрони извршни оквир.
RwLock	Брава за читање и писање која дозвољава вишеструке истовремене читаоце или једног писца.
Trait	Растов механизам за дефинисање заједничког понашања, сличан интерфејсима у другим програмским језицима.
Use-after-free	Грешка коришћења меморије након њеног ослобађања, што може довести до непредвидивог понашања програма.
Widget	Виџет — компонента корисничког интерфејса која приказује податке или омо- гућава интеракцију.
Work-stealing scheduler	Планер заснован на краји посла — алгоритам где неактивне нити преузимају задатке од заузетих нити.

Биографија

Душан Лечић је рођен 27. јула 2002. у Чачку. У родном граду је завршио Основну школу „Владислав Петковић Дис“ 2017. као носилац Вукове дипломе, а потом је уписао Техничку школу у Чачку, смер Електротехничар информационих технологија - оглед. Године 2021. уписује Факултет техничких наука Универзитета у Новом Саду, смер Софтверско инжењерство и информационе технологије. Од раног доба гајио је велико интересовање за технологију и рачунарство, што га је довело до активног рада на бројним пројектима, а омиљене области су му системско програмирање, дистрибуирани системи и безбедност. Посвећен је коришћењу Линукс оперативног система и Вим (енг. *Vim*) едитора као примарних алата у свакодневном раду и развоју софтвера. Поред постигнутих успеха током школовања, у слободно време бави се фотографијом, читањем, кошарком, билијаром и склапањем механичких тастатура.

Литература

- [1] GNU Project, „procps - /proc file system utilities“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://gitlab.com/procps-ng/procps>
- [2] H. Muhammad, „htop - an interactive process viewer“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://htop.dev/>
- [3] J. P. Liljenberg, „btop++ - Resource monitor that shows usage and stats“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://github.com/aristocratos/btop>
- [4] C. Bassi, „gotop - A terminal based graphical activity monitor“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://github.com/cjbassi/gotop>
- [5] IEEE, „POSIX.1-2017 - Signal Concepts“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на https://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html
- [6] J. P. Liljenberg, „bashtop - Linux/OSX/FreeBSD resource monitor“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://github.com/aristocratos/bashtop>
- [7] J. P. Liljenberg, „bryutop - Linux/OSX/FreeBSD resource monitor“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://github.com/aristocratos/bryutop>
- [8] C. Tsang, „bottom - A customizable cross-platform graphical process/system monitor“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://github.com/ClementTsang/bottom>
- [9] S. Klabnik и C. Nichols, *The Rust Programming Language*. No Starch Press, 2023. [На Интернету]. Доступно на <https://doc.rust-lang.org/book/>
- [10] Atop Contributors, „atop - Advanced System and Process Monitor“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://www.atoptool.nl/>
- [11] N. Griffiths, „nmon - Nigel's performance Monitor for Linux“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://nmon.sourceforge.io/>
- [12] C. Bassi, „ytop - A TUI system monitor written in Rust“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://github.com/cjbassi/ytop>

-
- [13] Tokio Contributors, „Tokio - An asynchronous Rust runtime“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://tokio.rs/>
 - [14] Ratatui Contributors, „Ratatui - A Rust library for cooking up Terminal User Interfaces“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://ratatui.rs/>
 - [15] Crossterm Contributors, „Crossterm - Cross-platform terminal manipulation library“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://github.com/crossterm-rs/crossterm>
 - [16] Rust Foundation, „The Rust Reference“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://doc.rust-lang.org/reference/>
 - [17] Rust Foundation, „Asynchronous Programming in Rust“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://rust-lang.github.io/async-book/>
 - [18] Tokio Contributors, „Tokio Tutorial“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://tokio.rs/tokio/tutorial>
 - [19] Ratatui Contributors, „Ratatui Widget Documentation“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://docs.rs/ratatui/latest/ratatui/widgets/index.html>
 - [20] Termion Contributors, „Termion - A bindless library for manipulating terminals“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://github.com/redox-os/termion>
 - [21] Termwiz Contributors, „Termwiz - Terminal Wizardry“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://github.com/wez/wezterm/tree/main/termwiz>
 - [22] Linux Kernel Organization, „The /proc Filesystem“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://docs.kernel.org/filesystems/proc.html>
 - [23] Facebook, „Flux - Application Architecture for Building User Interfaces“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://facebookarchive.github.io/flux/>
 - [24] D. Tolnay, „thiserror - derive(Error) for struct and enum error types“. Приступљено: 21. Фебруар 2025. [На Интернету]. Доступно на <https://github.com/dtolnay/thiserror>