Математичка гимназија Београд

Матурски рад

Информатика и рачунарство

Модерна микрокернелска архитектура за ARMv8-базиране паметне уређаје

Ученик:
Давид Давидовић

Ментори:
Станка Матковић
Нина Алимпић

28. maj 2015

Сажетак

Овај рад је својеврсна теоретска расправа о архитектуралним и имплементационим одлукама у развоју модерног кернела који би опслуживао данашње паметне уређаје (мобилне телефоне и таблет рачунаре) базиране на ARMv8 архитектури и AArch64 инструкционом сету, уз примере из праксе постојећих и широко коришћених, као и истраживачких и *proof-of-concept* решења, из којих се могу извући закључци о детаљима развоја ових система. Иако је сам рад ограничен на конкретну процесорску архитектуру, већина разматраних решења се може без губитка општости имплементирати и на другим процесорима.

Испитивање овог проблема мотивисано је чињеницом да све већи број људи широм света витално зависи од својих мобилних телефона како би комуницирали, радили и разонодили се, те је у интересу целе индустрије да они буду што је могуће више стабилни, безбедни и брзи—а те три ставке омогућује већим делом и OS кернел који их покреће.

У раду ће бити истражен развој микрокернела који пружа основне сервисе IPC-а, синхронизације, контроле приступа хардверу и *process scheduling*, између осталог, док се драјверски код и остале интегралне компоненте OS кернела делегирају серверима који раде одвојено и изолирано, са пажљивом контролом приступа системским ресурсима. Биће истражено повећање перформанси избегавањем хардверских промена контекста (енгл. *context switches*) који су досад били највећи "камен спотицања" приликом имплементације микрокернелских архитектура у пракси, при чему неће бити компромитовани безбедност и стабилност система. Закључно, биће дат осврт на даље могуће правце истраживања који нису могли бити обрађени унутар овог рада, као и начини на које би кернел који прати идеје изложене овде могао да се користи у реалним, практичним ситуацијама.

Садржај

1 Увод

Пре упуштања у дубину материје, корисно је рећи неколико речи о разлозима због којих је овај рад настао, о архитектури о којој се примарно говори, као и упознати се са неким битнијим ранијим пројектима у овој области, како истраживачких тако и практично употребљивих. Треба се, такође, осврнути и на неке стручне термине и акрониме чије значење није из контекста сасвим јасно, као и на неколико других детаља без чијег спомињања рад може бити некомплетан. У овом одељку биће посвећена пажња овим темама, како би се поставио солидан темељ за расправу која ће бити представљена у наставку.

Референце на постојећи рад ће, на нивоу свих одељака у овом документу, често водити не ка научним радовима, чланцима у магазинима и сл, већ на Интернет ресурсе попут докумената у којима су информално описани разни приступи, методологије и запажања, постове на *mailing* листама и сличним изворима доступним само на Интернету. Разлог овоме је тенденција програмера, посебно "хакера"[1], да се опредељују за брзу имплементацију својих идеја, уз кратак период итерације, што доводи до тога да њихове идеје на крају буду само необавезно записане у минималној форми. У случају да се тиме не губи на садржају, цитирани су боље структуирани и објашњени радови, углавном у штампаном облику, али у неким ситуацијама не постоји рад који илуструје неки конкретан концепт (или је тај рад инфериорнији од других извора), и у том случају се директно цитирају URL адресе које воде до релевантних ресурса.

Са претходним ставовима на уму, овај одељак биће више информалан и мање технички оријентисан од наредних; биће обрађене теме о тржишту паметних уређаја као и кратки описи водећих архитектура и оперативних система. Ове ставке су битне за разумевање образложења иза специфичних одлука које ће бити донесене у вези са предлогом архитектуре новог кернела, те се стога не штеди знатно на њиховом обиму. Ипак, за анализу ове проблематике у веће дубине, читалац се упућује на литературу цитирану унутар овог одељка.

[1] Израз "хакер" је контроверзан и без универзално прихваћене дефиниције. Ипак, овде је употребљен јер је врло прикладан у описивању конкретне појаве. За кратак преглед феномена ове речи, видети [5].

1.1 Мотивација

Паметни мобилни уређаји, у које се убрајају, између осталих, паметни телефони (енгл. *smartphones*), таблет рачунари (енгл. *tablet computers*), и паметни сатови (енгл. *smartwatches*) су преносиви уређаји који првенствено служе као средства за комуникацију, посао, разоноду, сурфовање Интернетом и слично, углавном се напајају из батерије и дизајнирани су тако да могу бити уз корисника било где—за разлику од, на пример, десктоп рачунара или сервера, чија је намена да буду стационирани на једном месту и одатле врше своју функцију. Паметне мобилне уређаје од сродних али дистинктних система (GPS навигационих уређаја, пејџера и традиционалних мобилних телефона, такође познатих под називом *feature phones*) разликује вишефункционалност, интуитиван и врло често скеуоморфан кориснички интерфејс, као и фокус на коришћењу од стране једног корисника, који их такође разликује од преносивих уређаја попут laptop или netbook рачунара [19]. У овом раду ћемо игнорисати неке "егзотичније" примере ових уређаја и фокусирати се на случајеве горепоменутих паметних телефона и таблет рачунара, који заједно чине убедљиву већину целог тржишта.

Тренутно доступни уређаји овог типа на глобалном тржишту долазе у широком спектру могућности и цене. Приликом одабира уређаја, најпре се уочава однос хардверских особина (рачунска моћ процесора, количина радне меморије, квалитет камере, физичка величина и резолуција екрана, могућности повезивања, капацитет батерије, итд.) и цене, али битну улогу игра и оперативни систем који је инсталиран на уређају. На тржишту десктоп и лаптоп рачунара постоји велики избор пропријетарних и слободних оперативних система и њихових варијанти (Microsoft Windows, Linux, Mac OS X, NetBSD, FreeBSD, и сл.)—унутар ових тржишта неки уређаји се продају и без оперативних система те се од корисника очекује да инсталирају свој. На тржишту паметних телефона и таблет рачунара слика је, ипак, мало другачија. Хардвер је врло затвореног типа, претежно се користе склопљене SoC (*system-on-chip*) шасије чија документација није отворена за јавност, и произвођачи имају пуну слободу да уређај испоруче са својим преинсталираним оперативним системом, уз мере безбедности које отежавају или онемогућују инсталацију другог системског софтвера, и уз услове гаранције који поништавају било какво право замене у случају да је системски софтвер замењен [15].

Кернели који се налазе у сржи ових оперативних система нису ра-

звијани са овом архитектуром на уму, већ су у питању портови постојећих кернела који су развијани за друге системе (најчешће неку x86 варијанту, али постоје и случајеви у којима је оригинални код писан и за неке архаичније процесорске архитектуре), као ни са конкретним уређајима и начином коришћења. Стога ови кернели потенцијално садрже много функционалности која је непотребна за функционисање у овом конкретном случајеву, иако је неопходна за нпр. серверске или десктоп примене. Познато је и да су *mainline* варијанте ових кернела знатно мењане путем прилагођених *patch*-ева, што указује на то да оригинал није у потпуности одговарајући за примену којој је подвргнут, при чему је дискутабилно да ли су ови *patch*-еви исправно интегрисани, у складу са постојећим дизајн одлукама (више о једном примеру у вези са Linux *mainline* кернелом и Android пројектом касније). Притом, стање у овој истраживачкој области је изузетно активно, али су пројекти који се у пракси користе стари и са великим бројем архаичних (енгл. *legacy*) решења која онемогућује да се побољшања врше на елегантан начин, због комплексности система.

Мотивација иза овог пројекта је вишеструка, али је њен најбитнији аспект жеља да се успостави скуп архитектуралних одлука које ће омогућити да се продукциони кернел који их представља развије тако да буде једноставан и елегантан, пратећи познате принципе доброг програмирања који ће омогућити да се постави темељ за мобилни оперативни систем који је у идеалном случају бржи, отворенији, слободнији, безбеднији, стабилнији и представља бољу алтернативу постојећим решењима. Аутор је мишљења да овакав подухват није немогуће извести уз праве, информисане одлуке, посебно уз помоћ великог опуса претходног истраживања у овој области, као и делова кода и имплементација лиценцираних под слободним лиценцама који се могу искористити (драјвери, фајл системи, TCP/IP *stack* имплементације и сл.) како би се избегло понављање одређених задатака који су раније задовољавајуће решени, или могу бити адаптирани за наш конкретан случај.

1.2 Неколико речи о опасностима теоретских расправа

Теоретске расправе у компјутерским наукама, посебно уколико нису везане за апстрактне појмове (на пример, анализу сложености алгоритама или теоретске аспекте ламбда калкулуса) су посебно про-

блематичне, због тога што се без адекватне имплементације већег дела разматраних одлука не може донети суд о њиховим перформансама, безбедности, лакоћи одржавања или елегантности. (У најекстремнијим случајевима, информисане одлуке донесене у најбољој намери могу имати далекосежне штетне последице у будућности [8].) Разлог овоме је што су рачунари и њихове компоненте довољно комплексни да се све потешкоће и компликације не могу узети у обзир приликом резоновања и планирања; да могу, онда би *"waterfall model"* софтверског развоја и данас био сасвим прикладан за развој било какве врсте софтвера, што није случај већ деценијама [16, 17]. Са свим овим на уму, дакле, закључује се да је наивно веровати да се оваква теоретска расправа може написати на начин који гарантује оптималну имплементацију, као и да се претпоставке направљене у њој не могу показати неисправним у тренутку имплементације, или када се ситуација у индустрији промени довољно да се јасно може приметити кратковидост приликом доношења истих.

Аутор покушава да минимизује ефекат оваквих проблема тиме што представља архитектуралне одлуке и намере у најопштијем могућем облику који дозвољава да се њихова срж искаже, са што мање претпоставки о њиховом понашању у пракси које би могле да се испоставе нетачним и тиме их инвалидирају. Такође је уложен труд да се оне рационализују путем постојећих сродних истраживања њихових импликација у пракси, али овај метод може само смањити проблем (никако га неутралисати) због тога што већина ових резултата изузетно зависи од контекста (што је променљива која се једино може контролисати имплементацијом целокупног система), као и од околности које се не могу никако предвидети, и самим тим ни контролисати.

1.3 ARM фамилија процесора и процесорских архитектура

ARM (историјски *Acorn RISC Machines* и *Advanced RISC Machines*) је фамилија RISC процесора и процесорских архитектура (енгл. *instruction set architecture, ISA*). RISC, акроним за *reduced instruction set computing* је подврста процесорских архитектура (и својеврсна "филозофија") у којој је број инструкција сведен на апсолутни минимум [3]. Често недостају комплексне инструкције (на пример, x86 фамилија инструкција `rep` које уводе процесор у петљу која може да ите-

рира арбитрарно много пута) које се у софтверу замењују композицијом једноставнијих. Оваква архитектура чини дизајн процесорског чипа једноставнијим, што омогућава веће процесорске тактове него када би на еквивалентном хардверу била имплементирана комплекснија архитектура, а такође се и приликом њиховог рада мање троши енергије, што даље имплицира да је загревање јединице знатно слабије. ARM процесори су данас вероватно најпознатији и најшире коришћени RISC процесори, и највише су у употреби унутар *embedded* система, преносивих уређаја и као делови микроконтролера који се користе у индустрији—ово су све случајеви у којима смањена потрошња енергије и остале карактеристике ових јединица доносе много користи.

ARM Holdings, компанија која стоји иза ових чипова, има апсолутну доминацију над тржиштем мобилних уређаја, са око 90% паметних телефона које покреће ARM-базиран процесор и 31% целокупног тржишта мобилних уређаја [24], при чему ове цифре имају тенденцију да и у будућности даље расту. Лако је одатле видети да су ове RISC архитектуре изузетно успешне и да је њихово широко прихватање доказ постојаног дизајна и интелигентних инжењерских одлука у вези са њиховим карактеристикама. Занемарујући раније ARM архитектуре које се користе углавном у *embedded* системима, разматраћемо две најшире коришћеније: ARMv7 фамилију, чији је инструкциони сет познат као A32, и ARMv8, новију фамилију 64-битних процесора чији је инструкциони сет познат као AArch64 или A64. За сада је ARMv8-A једина конкретна варијанта ARMv8 доступна, и на њу ће овај рад бити фокусиран. Без обзира на то, а имајући у виду да је ARMv7 парадигма на којој је изграђено много инфраструктуре тренутних паметних уређаја и оперативних система, а такође и чињеницу да AArch64 наслеђује већину семантике њеног ISA-а, у наставку биће дат кратак (и некомплетан) информалан опис ARMv7/A32 архитектуре и њеног начина функционисања, као и битне разлике које ARMv8-A односно AArch64 уводи. ARMv7-A је варијанта ARMv7 архитектуре која се користи за процесорске јединице опште намене, које се налазе у паметним уређајима. ARMv7-R је варијанта која се користи за *real-time* системе и имплементира одређене екстензије инструкционом сету које ово омогућују, док је ARMv7-M варијанта која се користи за микроконтролере и такође садржи посебан сет додатака које ово олакшавају (омогућујући, на пример, обрађивање прекида (енгл. *interrupts*) са ниским временом одзива и коришћењем архитектурално-агностичких виших програмских је-

зика). С обзиром на то да је овде највише од интереса варијанта која се користи за наше циљне уређаје, у наставку ћемо се фокусирати на ARMv7-A.

1.3.1 ARMv7-A

Сви подаци презентовани у овом одељку су преузети из архитектуралног референтног упутства за ARMv7-A и ARMv7-R инструкционе сетове [12]. Овај документ у наставку неће бити посебно цитиран.

ARMv7-A је, као што је споменуто, RISC архитектура, и карактеришу је типичне особине RISC система. У питању је *load-store* архитектура (што значи да се сав приступ меморији врши кроз посебне инструкције које оперирају над једним регистром и једном меморијском локацијом, а да се остале операције врше искључиво између регистара). Велики број инструкција се може условно извршити у зависности од индикатора процесорског стања, којим може да се управља извршавањем инструкција које врше компарацију, битовно тестирање и др. Архитектура је 32-битна, са машинском речи (енгл. *word*) од 4 бајта, што је и величина меморијских адреса (уз могућност коришћења специјалних екстензија за повећање адресног простора). ARMv7-A такође има знатно мање модова адресирања од својих CISC савременика.

У наставку биће дат преглед неких интересантних аспеката ове архитектуре, али ће велики број детаља бити изостављен јер је у питању комплексна област чије би потпуно обрађивање било непрактично.

Инструкциони сетови: ARMv7-A архитектура начелно подржава четири инструкциона сета:

- **ARM**. Ово је главни инструкциони сет за који се генерише извршни код већине компајлера који циљају на ову платформу. ARM инструкциони сет подразумева инструкције од 32 бита (4 бајта), које су увек поравнате (енгл. *aligned*) на 4-бајтне границе. Формално, ово значи да се свака инструкција може простирати само на меморијским адресама $[M, M+3]$, где је $M \equiv 0 \pmod 4$. У наставку ћемо се бавити искључиво овим инструкционим сетом, док ће остали бити споменути због комплетности.

- **Thumb**. Thumb инструкциони сет је алтернативни сет инструкција дизајниран за већу густину машинског кода. У њему постоје 16-битне (2-бајтне) и 32-битне (4-бајтне) инструкције, са

поравнањем од 2 бајта. 16-битне инструкције у Thumb сету углавном имплементирају чешће коришћене операције (ALU операције, учитавања и уписивања и сл.) док су 32-битне инструкције типично резервисане за ређе операције. Већина 16-битних инструкција може приступити ограниченом подскупу процесорских регистара (`R0`–`R7`, у документацији названи *ниски регистри* (енгл. *low registers*)).

- **Jazelle**. Jazelle је инструкциони сет изведен како би омогућио хардверско убрзање приликом извршавања Јава *bytecode*-а. Процесори у овој архитектури углавном садрже *тривијалну Jazelle имплементацију*, што значи да се нативно извршавање Јава *bytecode*-а извршава у софтверу, те се не добија на перформансама уколико се за извршавање Јава кода користи овај метод. ARM не препоручује имплементацију Jazelle стања у новим процесорима, а њено постојање некомпатибилно је са новијим ARM екстензијама за виртуелизацију.

- **Thumb**. ThumbEE је варијанта Thumb инструкционог сета који је дизајниран за извршавање динамички генерисаног кода.

Процесор може да слободно мења оперативни инструкциони сет између ARM и Thumb варијанти—ARM код може да позива Thumb код и обрнуто. Ово се најчешће ради користећи инструкције *branch with exchange* (`bx`) и *branch with link-exchange* (`blx`). Ове инструкције се користе за безусловно гранање (скок на неку константну адресу или адресу која је смештена у неком процесорском регистру), а прелазак између инструкционих сетова се остварује постављањем најмање значајног бита у адреси на коју се грана. С обзиром на то да би адреса која има постављен најмање значајан бит била неисправна у оба инструкциона сета, ARM процесор увек претпоставља да је овај бит 0, а његову праву вредност користи као индикатор промене између два инструкциона сета. Са друге стране, извршавање ThumbEE и Jazelle кода су посебна стања процесора, до којих се мора доћи експлицитном манипулацијом контролних регистара. Детаљи овог поступка су неважни и изван обима овог одељка.

Регистри: Софтверу који се извршава на ARMv7-A процесорима је доступно 16 регистара свеукупно, од којих 13 регистара опште намене и 3 са специјалним значењем. Регистри су ширине 32 бита (4 бајта),

као што је и очекивано у 32-битној процесорској архитектури. Називи и значења регистара су:

- `R0`–`R12`. Ово су регистри опште намене које немају посебно резервисано значење. Користе се за интермедијарне вредности приликом израчунавања, за чување индекса, адреса индиректних грана и сл. Не постоје рестрикције о њиховој употреби и могу садржати било коју вредност.

- `SP`, или `R13`. SP (**stack pointer**) регистар држи тренутну локацију почетка активног стека над којим манипулишу стек инструкције (`push` и `pop`). У ARMv7-A архитектури, стек расте на доле (ка нижим меморијским локацијама), као и на x86 архитектури. Иако се `SP` може користити као регистар опште намене (наравно, уколико се у исто време не користе инструкције за операције над стеком), произвођач саветује против такве употребе, делом због тога што је то понашање програма довољно ретко да већина оперативних система, *debugger*-а и осталог системског софтвера претпоставља да `SP` увек показује на валидну адресу почетка стека па његова употреба у непредвиђене сврхе може довести до нежељених последица.

- `LR`, или `R14`. LR (**link register**) садржи тзв. *адресу линка*. На ARM архитектури, позиви процедура се реализују тако што се повратна адреса ставља у `LR`, након чега је позвана процедура дужна да сачува вредност овог регистра на стеку током пролога, а онда и да изврши гранање на сачувану адресу током епилога. Када се не користи у ове сврхе, овај регистар се употребљава као регистар опште намене, и тада се типично у изворном коду означава са `R14`. За више информација о детаљима позива процедура на овој архитектури, видети [14].

- `PC`, или `R15`. PC (**program counter**) је регистар који служи као показивач ка тренутној адреси која се извршава. Битно је напоменути да се приликом очитавања вредности овог регистра у инструкцији која почиње на локацији M добија вредност $M + 8$, односно две инструкције испред оне која се тренутно извршава. Ово омогућава да инструкција `mov lr, pc` праћена неком инструкцијом безусловног гранања (најчешће `b` или `bx`) изврши исправан позив нове процедуре и упише исправну повратну адресу у `LR`. ARMv7-A дозвољава директну манипула-

цију овог регистра—уписивање вредности у `PC` изазива гранање на новоуписану адресу.

Сем ових регистара, доступних апликационом (*user mode*) софтверу, постоје и специјални регистри који су доступни системском софтверу (кернелу оперативног система) који се извршава на процесору. У конкретној имплементацији процесора се ситуација додатно компликује; мора се узети у обзир *banking* регистара и различита стања (односно *модови*) у којима се процесор може наћи. Иако за наше потребе битни, ови посебни регистри ће бити уведени у случају да је њихова дефиниција потребна ради објашњења неке конкретне појаве, али ће се у генералном случају избегавати употреба свих концепата тесно везаних за архитектуру, јер постоји жеља да овај рад буде на најгенералнијем могућем нивоу на коме се може описати, без непотребног везивања за имплементационе детаље архитектуре.

Процесорски модови и нивои привилегија: ARMv7-A процесор се може наћи у неколико модова (енгл. *modes*) у току своје операције. Такође, процесорска архитектура разликује два нивоа привилегије (прстена), названих PL0 (*unprivileged execution*) и PL1 (*privileged execution*). У случају да процесор имплементира виртуализационе екстензије, постоји и ниво PL2 који користи хипервизор (о њему нећемо сада разговарати). У прстену PL0 се обично извршава изолиран софтвер (програми који раде под оперативним системом), док се у PL1 нивоу извршава привилегован кернелски код који има апсолутну контролу над процесором. Модови ARMv7-A процесора су следећи:

- **User**. Овај мод је једини који ради под PL0 и то је нормалан мод извршавања неприviлегованог корисничког софтвера.
- **FIQ**. У овај мод улази процесор приликом добијања FIQ захтева (*fast interrupt request*-а). Код који се извршава у FIQ моду има на располагању додатан сет банкованих регистара.
- **IRQ**. У овај мод улази процесор када добије обичан, неприоритизован прекид (енгл. *interrupt*).
- **Supervisor**. Ово је мод у који процесор улази приликом супервизорског позива (*supervisor call exception*). Приликом ресетовања процесора, ово је такође иницијално стање у коме се он налази.

- **Abort**. Аборт је посебно стање процесора у које улази када се догоди изузетак (енгл. *exception*) типа *Data Abort* или *Prefetch Abort*.
- **Undefined**. Извршавање недефинисане инструкције, или изузетак који се догодио на нивоу инструкције, уводи процесор у овај мод.
- **Hyp**. Мод у коме ради хипервизор у случају да су активне виртуализационе екстензије. Ово је једини мод који ради под PL2.
- **Monitor**. *Secure Monitor Call* изузетак уводи процесор у овај мод, и ово је мод који може да направи валидну транзицију између безбедног и небезбедног стања процесора. За детаље видети [12].
- **System**. Ово је мод у коме се извршава привилегован код оперативног система. У овај мод се типично улази из трамполина системских позива (енгл. *system calls, syscalls*), окидања тајмера, IRQ и FIQ *handler*-а, и слично.

У дубље детаље ове поделе и детаља сваког мода нећемо улазити; заинтересовани читалац може наћи све потребне информације у документима цитираним унутар овог одељка.

1.3.2 ARMv8-A

ARMv8-A је нова еволуција ARM процесора опште намене и представља велики скок од претходника због тога што је у питању 64-битна архитектура, при чему остаје и даље компатибилна са претходном верзијом и њеним инструкционим сетом. Нови инструкциони сет зове се AArch64, такође има 32-битне инструкције, и разликује се на неколико јако битних аспеката од ARMv7-A архитектуре, чија некомплетна листа следи:

- Као што је већ споменуто, AArch64 користи 64-битне основне регистре и 8 бајта као основну јединицу операција. Адресе су у њему такође широке 64 бита, што даје адресабилан простор од око 16 егзабајта.
- Битно је смањен број модова и банкованих регистара у процесорском дизајну.

- Додате су посебне инструкције које у хардверу извршавају неке криптографске операције, као на пример шифровање AES (Rijndael) 128-битних блокова и рачунање SHA-1 и SHA-256 хешева.
- Знатно је смањен број доступних условних инструкција.
- Постоји 31 регистар опште намене доступан у сваком тренутку, при чему се регистри опште намене никад не банкују.
- Додата су разна унапређења у SIMD инструкцијама (потпуно IEEE-754 конформантна имплементација за SIMD), промењен је подсистем за виртуелну меморију, и слично.

С обзиром на то да је велики број процесорске семантике и конкретних детаља ове архитектуре наслеђен из ARMv7-A, у детаље ове архитектуре нема потребе сада улазити. Релевантни детаљи биће представљени у тренутку када и ако буду били неопходни, а за комплетан и опширан опис ARMv8-A на 5886 страна, видети [13].

1.4 Кернели оперативних система

Кернел оперативног система (енгл. *operating system kernel*) је срж оперативног система, односно системски софтвер чији је задатак да координише рад осталог системског софтвера и корисничких програма и да им презентује интерфејс вишег нивоа, углавном са слојевима апстракције који омогућују да се хардверски детаљи енкапсулирају од корисничког кода. У зависности од конкретне намене и захтева, кернели могу да имају широк спектар дефинишућих карактеристика у начину на који обављају свој задатак. Због тога што се овај рад бави кернелима који покрећу данашње паметне уређаје, наредни текст ће бити применљивији на њих него на, на пример, *real-time* кернеле који се користе на системима за контролу индустријских машина или на *baseband* чиповима мобилних телефона, а немогуће је задовољавајуће покрити све случајеве без прибегавања исувише неспецифичним и генерализованим реченицама и изјавама које не би допринеле расправи. Задаци којима се често (али не увек; због великог броја ситуација за које се може развити кернел немогуће је пронаћи савршено тачну дефиницију) бави кернелска компонента оперативног система су следећи:

- **Изолација корисничких програма, односно процеса**. Кернел мора да осигура да малициозни програми и програми који

се недозвољено понашају не могу да компромитују ресурсе који им нису додељени, и да прекине процесе који прекрше ова правила. Дужност кернела је, такође, да осигура да малициозни процеси не могу да приступе подацима којима им није експлицитно дозвољено да приступе.

- **Inter-process communication (IPC)**, односно осигуравање безбедног механизма да процеси на систему комуницирају. У зависности од архитектуре кернела, у ову ставку може да спада и механизам којим процеси комуницирају са самим кернелом.

- **Scheduling**. Вишенаменски кернели које посматрамо треба да буду способни да координишу велики број процеса на систему, водећи рачуна о томе да сваки од њих добије једнаку (или од стране корисника посебно избалансирану) количину системских ресурса попут радне меморије и процесорског времена. На модерним системима с којима корисници интерагују, како би се постигло задовољавајуће време одзива, углавном се говори о *pre-emptive multitasking* дизајну, у коме се процеси паузирају када истекне њихов део процесорског времена, њихово стање памти од стране кернела, а затим се следећи процес пушта у рад. Стратегија и конкретни алгоритми на којима се заснивају имплементације је врло активна истраживачка област.

- **Комуникација са хардвером**. Кернел мора да има способност да адекватно комуницира са хардвером када кориснички софтвер то захтева. Да би ово урадио, кернел мора да буде свестан постојања *драјверског програма* који је специјализован да разговара са конкретним типом хардвера.

- **Генерално одржавање система у употребљивом стању**. Кернел је задужен за, на пример, прекидање неких процеса када систем нема довољно ресурса, одржавање сервера који су покренути на систему, снижавање процесорског такта како би се сачувала енергија у ситуацијама ниске заузетости, елегантно и безбедно замрзавање система у случају фаталних хардверских грешака, и слично.

Постоји велики број различитих подела OS кернела; неке од њих су само од историјске важности, док су неке од њих применљиве на данашње пројекте којима се посвећује пажња, било у пракси или у

академским радовима који се овом проблематиком баве. Једна од грубљих, али врло битних подела, је подела кернела по начину на који им је функционисање организовано у смислу расподеле одређених задатака по процесорским нивоима и, имплицитно, гаранција безбедности које пружају. Старије поделе (попут поделе на *single-tasking* и *multi-tasking* кернеле) су за данашње стање индустрије и конкретан случај о коме се овде дискутује махом ирелевантне; за више информација, може се консултовати било какав уџбеник на тему оперативних система који ће се готово сигурно дотаћи ових тема.

Унутар ове поделе, која је можда и најшире позната и примењена категоризација различитих типова OS кернела, разликујемо неколико категорија, чије ће главне карактеристике и примери бити описани у наставку.

1.4.1 Монолитни кернели

Монолитни кернели (енгл. *monolithic kernels*) су кернели које карактерише једнокомпонентност и тенденција да се основни кернелски програм манифестује као велики објектни фајл који садржи највећи део функционалности кернела. Ово такође значи да се цео код који сачињава кернел извршава у истом, дељеном адресном простору, као и да не постоји експлицитан механизам да се кернелске компоненте изолују тако да не могу да приступају подацима других компоненти. Поред извршавања у истом адресном простору, наравно, подразумева се да се све компоненте кернела извршавају у привилегованом процесорском прстену, а да су корисничке апликације једини део система који се покреће у непривилегованом, ограниченом процесорском прстену. Ово не значи да кернел није екстензибилан; на пример, Linux, најпознатији монолитни кернел данашњице, дозвољава учитавање посебних `.ko` фајлова који представљају кернелске објектне фајлове (*kernel objects*) који се у току извршавања могу учитати у адресни простор кернела. На овај начин је имплементирана већина драјвера у Linux-у, као и разни други подсистеми, али због монолитне структуре овог система, то значи да сваки од овако динамички учитаних модула има несметан приступ целом кернелу. Монолитни дизајн је историјски један од првих кернелских дизајнова који су били у употреби; рани *time-sharing* оперативни системи (на пример Multics, Unix варијанте и OS/360) су користили монолитни кернелски дизајн. Каснији примери укључују DOS и ране Windows кернеле, док су од монолитних кернела у данашњој употреби свакако најпознатији Linux и Windows

NT[2].

У монолитним кернелима комуникација између компонената остварује се углавном дељеном меморијом унутар кернелског простора, а узевши у обзир чињеницу да све компоненте деле меморију у том смислу, ово је логичан избор и природна последица дизајна самог кернела. У овом случају IPC механизми попут Unix сокета (енгл. *sockets*) су ту само да омогуће комуникацију између корисничких процеса, док су IPC механизми које кернел користи сведени на синхронизационе примитиве (енгл. *synchronization primitives*), као што су *spinlock*-ови, *readers-writer lock*-ови, или егзотичнији системи попут RCU (*read-copy update*) [18].

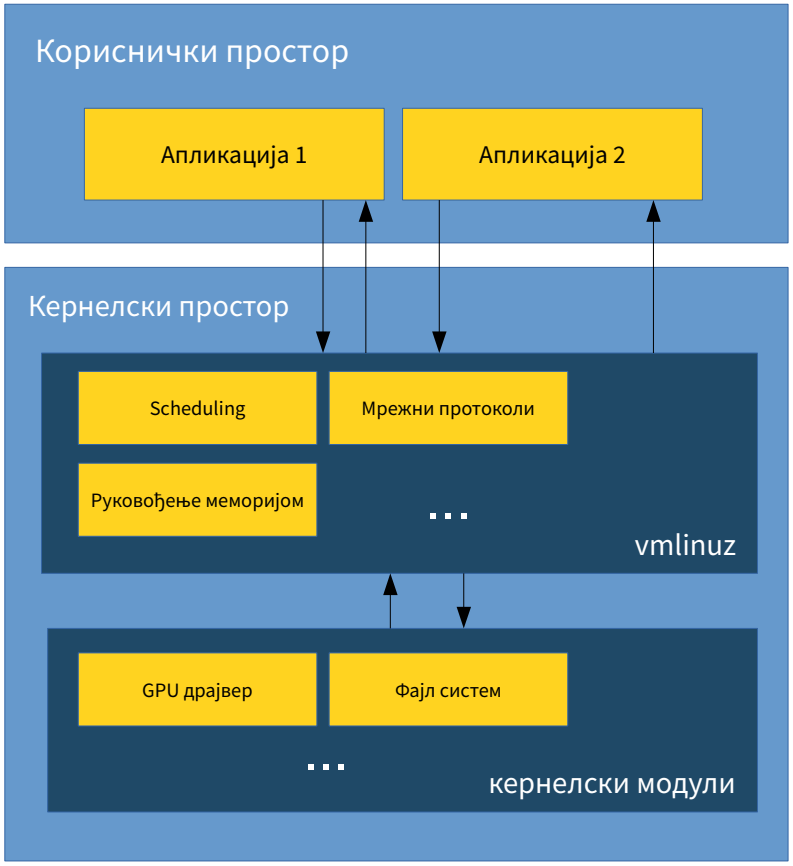
Иако апсолутно доминантна парадигма у практичним кернелима који су данас у широкој употреби, монолитни кернели су и од релативно раних дана у области развоја и истраживања оперативних система били мета оштрих критика, углавном од стране поборника микрокернелског дизајна (о коме ће бити говора убрзо). Један од најистакнутијих критичара је легендарни професор Ендру С. Тененбом (*Andrew S. Tanenbaum*), познати академик са широким опусом научних радова, уџбеника и чланака у угледним часописима, као и архитекта MINIX микрокернела, чија га је широко публицизирана и прилично остраћена јавна расправа са Линусом Торвалдсом управо о овој теми приближила и људима који се не баве развојем и дизајном оперативних система [22]. Критичари монолитних кернела цитирају разне недостатке у овом дизајну: на пример, било који неисправан или малициозан кернелски модул може довести до пада целог система, а због недостатка јасно дефинисаних граница између ових модула, који директно произилази из саме природе овог интерфејса, повећава се комплексност система, као и потешкоће у његовом одржавању, а смањује његова естетска лепота и елегантност.

И поред ових критика, монолитни кернели остају доминантни у практичној употреби. Linux кернел, као што је већ споменуто, је мо-

[2]Windows NT кернел се понекад сврстава и у хибридне кернеле, а како је сама дефиниција хибридних кернела врло крхка и отворена за различите интерпретације, овај резон се не може у потпуности ни оповргнути ни потврдити. Ипак, ауторово лично мишљење је да Windows NT, иако је у њему одређена функционалност која није витална за функционисање оперативног система померена у *user space* што би га учинило хибридним кернелом, тај феномен је имплементациони детаљ пре него свесна и прагматична архитектурална одлука, те сам кернел остаје фундаментално монолитички дизајн. За више информација о овој занимљивој теми, читалац се упућује на пост и дискусију на [7].

нолитан, а тај аспекат његовог дизајна је изузетно активно брањен од стране његовог творца, Линуса Торвалдса, који цитира важност да кернел буде употребљив и у стварном свету, односно да је перципирана елегантност микрокернелског дизајна искључиво од академског интереса, те да је у практичном домену смисленије дизајнирати кернел на овај начин. Windows NT, кернел који покреће све инкаранције Windows оперативног система од верзије NT па надаље, такође је у својој бити монолитан. На основу ова два високопрофилна примера можемо закључити да се у пракси овај дизајн показује врло фаворабилно, а на будућности је да одлучи хоће ли он бити замењен неким новим или остати највећим делом исти.

Слика 1: Поједностављени графички приказ монолитне архитектуре, на начин на који се она појављује у Linux кернелу. Један објектни фајл, `vmlinuz`, садржи основне кернелске компоненте, док се хардверски-специфични драјвери и додаци кернелу учитавају у његов адресни простор по потреби.



1.4.2 Микрокернели

Микрокернелска архитектура је настала убрзо после монолитне, и карактеришу је неке врло битне разлике од њеног "претходника". У монолитној архитектури, у кернелском прстену и адресном простору извршава се само једна уска компонента, микрокернелски сервер, која пружа само основне функције синхронизације, IPC механизама који користе слање порука као главни метод комуникације, као и координисања приступа хардверу. Драјвери, фајл систем, и остале компоненте кернела се извршавају као обични кориснички процеси који комуницирају са микрокернелским сервером и на тај начин захтевају разне системске операције које су им неопходне како би функционисале, а кориснички програми углавном комуницирају са овим процесима како би захтевали разне услуге оперативног система, попут уписивања фајлова на диск, захтевања још меморије, и слично. При томе се све ове операције извршавају углавном слањем имутабилних порука између ентитета који комуницирају, а посредством микрокернелског сервера, чиме се избегава коришћење дељене меморије и, судећи по поборницима овог дизајна, добија систем који је елегантнији, робустнији и лакши за одржавање.

Друге предности микрокернелског система укључују безбедност коју монолитна архитектура не може да гарантује, због тога што је нерегуларан драјвер само још један кориснички процес и не може да приступи кернелским структурама или изазове пад система. Такође је повећана и стабилност, јер се тада драјвери и остали OS-витални процеси извршавају са најмањим могућим привилегијама које су потребне за њихово функционисање, па кернелски модул са багом који се сруши може бити рестартован, а систем може наставити функционисање за то време без бојазни да му је стабилност нарушена. Као што је већ споменуто, слање порука као главни IPC механизам често обезбеђује да се број *data race* ситуација смањи и њихово решавање поједностави, а такође и охрабрује дефинисање стабилних интерфејса за комуникацију између различитих компоненти. Такође, микрокернели су по правилу мањи и компактнији по количини изворног кода потребног да се постигне еквивалентна функционалност, па се и ово наводи као предност и доказ елегантности и минимализма овог дизајна.

Ипак, микрокернелске архитектуре нису без својих недостатака. Због потребе да се за било коју комуникацију између два процеса користи кернел који мора да осигура да се порука преноси безбедно,

потребно је у овом случају направити две хардверске промене контекста (енгл. *hardware context switches*), једна да се процесор пребаци у кернелски мод, где ће микрокернелски сервер обрадити поруку и баферовати је или копирати директно у адресни простор примаоца, а друга да се процесор врати у кориснички мод где ће прималац поруке урадити тражену акцију. С обзиром на то да су хардверске промене контекста релативно скупа операција [9], при којој се ствари додатно компликују јер долази до губљења података из процесорске кеш меморије, као и до избацивања свих или одређених ставки из TLB-а (*translation lookaside buffer*), што такође смањује перформансе контекстне промене, ово чини првобитне микрокернелске са наивном имплементацијом преласка из кернелског у кориснички простор много споријим од монолитних решења. Ово је једна од главних критика микрокернелског дизајна и разлог због кога он није видео ширу адопцију у практичним решењима.

Ипак, микрокернели новијег датума се активно труде да минимизују ову цену. На пример, L4 микрокернел фамилија, дизајнирана од стране Јохена Литгеа (*Jochen Liedtke*) успева да смањи цену ових IPC операција на минималне нивое [21], а рад у овој области и даље је врло активна истраживачка област. Данас, модерни микрокернели су нашли своју примену махом као виртуализациони хипервизори и у неким специјализованим нишама и *embedded* системима, али се и микрокернели старијих датума користе у практичне сврхе. Mac OS X, оперативни систем који покреће Apple Mac рачунаре, као и iOS, његова варијанта која ради на мобилним уређајима попут iPhone-а или iPad-а, базирана је на варијанти Mach микрокернела, једног од успешнијих микрокернелских дизајнова у историји који је био мета критика о његовом дизајну у којима су критичари наводили многе његове аспекте који много више подсећају на традиционални монолитни дизајн (Mac OS X и iOS раде под XNU кернелом, хибридним дизајном који комбинује Mach 2.5 као микрокернелски сервер и велики део FreeBSD компоненти који се извршавају у кернелском простору, уместо у традиционално корисничком како је то очекивано од микрокернелске архитектуре). Такође, на варијанти Mach-а базиран је и GNU Mach, интегрални део потпуно слободног оперативног система GNU/Hurd, који је релативно скоро био доведен у употребљиву фазу у којој може да се користи на продукционим системима, након две деценије развоја.

Такође, међу овим примерима се налази и велики број кернела из L4 породице, укључујући и врло познати seL4, први кернел чији су

интерфејси и начин функционисања формално доказани безбеднима, MINIX (укључујући и његову најсавременију инкарнацију MINIX 3), као и хипервизори попут Xen-а.

Слика 2: Графички приказ теоретски идеалног микрокернелског дизајна који поштује већину различитих дефиниција микрокернела. Све операције сем основних *message-passing* примитива се извршавају у корисничком простору, заједно са корисничким процесима. Одређени поверљиви даемони имају експлицитне дозволе за приступање хардверу или одређеним ограниченим подацима микрокернелског сервера.

1.4.3 Хибридни кернели

Хибридни кернели су неки спој оба екстремума кернелског дизајна—монолитних и микрокернела. С обзиром на то да се ова два при-

ступа могу спојити на најразличитије начине, не постоји општеприхваћена дефиниција овог појма, па се појам хибридних кернела овде помиње више због комплетности него из праве потребе да се објасни дистинктна категорија у кернелском дизајну. Често, хибридни кернели почињу са микрокернелским дизајном у коме је одређена функционалност враћена у кернелски простор, и са IPC механизмима који су комбинација система оријентисаног ка порукама и система оријентисаног ка дељеној меморији. Овакве промене и компромиси се праве из више разлога; најчешће, разлог је побољшање брзине за неки конкретни случај.

Познатији хибридни кернели су, на пример, BeOS, XNU и OS/2. Windows NT се конвенционално сматра хибридним кернелом, иако његова архитектура ипак указује на класичан монолитни дизајн.

1.5 Мобилни оперативни системи

Оперативних система за паметне мобилне уређаје има компаративно врло мало у односу на оперативне системе доступне за кућне рачунаре. Разлог овоме је релативно већа затвореност платформе и чињеница да је ова конкретна ниша оперативних система нова област у поређењу са оперативним системима за PC-јеве који се већ деценијама истражују и имплементирају. У овој секцији навешћемо неколико главних мобилних оперативних система за мобилне телефоне и таблет рачунаре, заједно са њиховим главним карактеристикама. Нећемо се знатно задржавати на сваком од њих, због тога што тада морамо говорити о великом броју ситних детаља који су махом последица пословних одлука компанија које су их израдиле, па тиме недовољно интересантни са техничког аспекта.

1.5.1 Google Android

Google Android је мобилни оперативни систем развијен од компаније Google за паметне мобилне уређаје. Android-ов кернел је модификована Linux варијанта, док сам систем покреће Dalvik, Јава виртуелна машина развијена за његове потребе. Кроз низ апстракција и Java API-ја, програмерима који развијају апликације за овај оперативни систем се представља унификован интерфејс за комуникацију са разним компонентама целокупног система. Овакве апликације су примарно писане у Java-и и дистрибуирају се као упаковани `.apk` фајлови, који садрже Java бајт код (енгл. *bytecode*) у dex формату. Мо-

гуће је у овакве архиве упаковати и нативни код у ELF формату, који се помоћу JNI (*Java Native Interface*) може позвати из главних компоненти апликације. Екосистем апликација је под контролом Google-а и "званично" место за прибављање апликација је *Google Play Store*, репозиторијум апликација који омогућује аутоматско ажурирање, као и друштвене додатке попут коментарисања и оцењивања апликација. Како би приступиле разним подацима или хардверским могућностима телефона, апликације могу декларисати захтев за одређеним *дозволама* (енгл. *permissions*) које корисник мора да прихвати како би покренуо апликацију.

Кернел који се налази у Android систему је, као што је већ споменуто, Linux разних верзија у зависности од верзије Android-а, при чему се користе портови овог кернела на ARM и x86 архитектуре. *Vanilla* Linux кернел је модификован разним додацима које су омогућиле његову адаптацију на ове мобилне платформе, при чему су неке од њих одбијене или и даље у статусу "чекања" да буду интегрисане у оригиналан кернел [25]. Такође је битно напоменути да је цео Андроид систем отвореног кода, што се делом може приписати чињеници да GPL, лиценца под којом се налази Linux кернел, захтева да сви изведени радови буду лиценцирани на начин који одржава корисникова фундаментална права у употреби софтвера.

1.5.2 Apple iOS

iOS је оперативни систем за паметне мобилне уређаје који се налази на уређајима компаније Apple—међу њима се налазе iPhone, iPad, Apple Watch и др. iOS користи исти кернел (XNU) и слој апстракције изнад њега (Darwin) као и Mac OS X, Apple-ов оригинални оперативни систем развијен примарно за десктоп и лаптоп Mac рачунаре. За развој апликација за овај оперативни систем примарно се користи Apple-ово проприетарно окружење, Xcode, а језици који се употребљавају су Objective-C и, скорије, Swift, језик развијен управо од стране Apple-а за овај оперативни систем.

Екосистем апликација је такође затворен и апликације се прибављају помоћу *Apple Store*-а, система сличне функције као и горепоменути Google Play Store.

1.5.3 Microsoft Windows Phone

Microsoft-ов мобилни оперативни систем базиран је на Windows NT

кернелу и носи назив Windows Phone. Апликације се развијају примарно помоћу Visual Studio алата, у програмским језицима везаним за Microsoft NET платформу—међу њима су C#, C++/CLI i Visual Basic. Маркет за апликације је такође затворен на исти начин као код претходно поменутих система, тако да га нећемо посебно објашњавати, и зове се *Windows Store*.

2 Нови кернел: НК

Као што је раније најављено, у овом одељку ће се посматрати и дискутовати начин развијања новог кернелског дизајна који треба да опслужи мобилне паметне уређаје. С обзиром на то да је непрактично описати све имплементационе детаље у вези са конкретном архитектуром, па чак и описати све детаље у вези са дизајном на апстрактнијем нивоу, фокус ће бити на оним архитектуралним одлукама које овај систем разликују од других постојећих решења, као и на онима које су посебно битне за задовољење неких од захтева који ће бити у наставку описани. Исказује се убеђење да ће на основу описа ових фундаменталних идеја моћи да се саставе и детаљнији планови рада, као и да се на крају дође до перформантне, функционалне имплементације која их задовољава што је могуће ближе.

Из чисто естетских разлога, хипотетчки кернел о чијој архитектури се дискутује унутар овог одељка звaћемо *НК (нови кернел)*, како бисмо избегли редундантне конструкције, а узевши у обзир чињеницу да ћемо га врло често помињати.

2.1 Захтеви

Гореописани кернел који би радио на овим уређајима мора да задовољи неколико фундаменталних захтева како би био употребљив и од било какве практичне релевантности. Ваља приметити да је већина ових захтева инхерентна за било какав кернел оперативног система, с тим што је образложење иза сваког од њих нешто другачије. С тим у вези, дају се три фундаментална захтева које се НК мора трудити да равноправно испуни, док ће сви остали захтеви који се од њега очекују да на неки начин проистекну од основних. Наравно, као и у сваком инжењерском подухвату, компромиси на спектру између ових захтева се могу (и морају) појавити, али се ниједан од ових примарних захтева неће значајно компромитовати. Дакле, уочавамо три циља које НК треба да оствари:

1. **Безбедност.** С обзиром на важност личних података који ће потенцијално бити смештени на оваквим паметним уређајима, као и битност интеракција које је он ауторизован да изврши у име конкретног корисника, потребно је направити систем који је робусно заштићен од малициозних намера, које се могу манифестовати било у виду спољних нападача, било у виду локалних

апликација које имају за циљ да преузму контролу над уређајем или дођу у посед недозвољених података. Такође се треба обезбедити од ненамерних грешака и програма који се понашају на недозвољен начин, што су фактори који могу да доведу до потпуно непредвиђених акција које могу да учине штету крајњем кориснику.

2. **Ефикасност.** Посебно узевши у обзир хардвер на коме се извршава овакав оперативни систем и велики раскорак између могућности хардвера и циља да он буде што енергетски ефикаснији и дуготрајнији, и очекивања које корисник има од свог мобилног уређаја, ефикасност овог оперативног система мора се посебно узети у обзир. Широк дијапазон радњи којима он треба да послужи (од претраживања Интернета до играња 3D игара) диктира да се свака компонента мора израдити са перформансама на уму, у смислу потрошње енергије, брзине извршавања и конзумпције меморије, као и ефикасне комуникације са хардвером.

3. **Елегантност.** Елегантан дизајн оперативног система не доноси велику почетну корист, с обзиром на то да се и са неелегантним и слабије осмишљеним дизајном може постићи еквивалентна функционалност до одређеног нивоа. Ипак, на дуге стазе овај захтев је и те како битан, због тога што се инжењерска "цена" (појам познат као *technical debt*) повећава експоненцијално са комплексношћу софтера и количином неоптималних одлука донесених у ранијем периоду. Такође, ауторово лично мишљење и филозофија је да архитектура софтвера, слично традиционалној архитектури у смислу грађевине, садржи и естетско-уметничку компоненту поред оне функционалне, те се с тога и на овај аспект треба обратитити посебна пажња. На крају, познато је да добре одлуке и добри системи преживљавају дуги низ година са мало адаптација, због своје инхерентне елегантности, док се лоше архитектовани системи често покажу застарелим и после једне или две године. С тим у вези, било би кратковидо и неувиђајно игнорисати потребу за елегантним дизајном науштрб потреби да се имплементација и комплетан скуп архитектуралних одлука направе што је брже могуће.

2.2 Тип кернела

У складу са малопређашњом дискусијом разних типова кернела који се могу срести у литератури и у пракси, потребно је одабрати тип кернела који ће НК претежно пратити, и њега се држати. Раније је поменуто да је дебата између поборника монолитног и микрокернелског дизајна изузетно остршћена, при чему обе стране имају одређене исправне, и одређене неисправне или неажурне аргументе. У овој подсекцији ћемо мало проширити ту дискусију, и на крају донети одлуку о фундаменталној архитектури НК-а.

Као што је већ споменуто, микрокернелски дизајн резултује у знатно краћем коду самог кернела. Сем тога, овај дизајн је новијег датума, а такође и пружа врло јаке гаранције у вези са безбедношћу и изолацијом, као и стабилношћу. Багови у монолитном оперативном систему могу довести до већег безбедносног ризика и ризика за стабилност целокупног система—а кернелски код, као уосталом и сваки други, садржи солидну концентрацију багова по написаној линији. С друге стране, у микрокернелском дизајну могуће је контролисати штету и повратити се од оваквих грешака, много више налик конвенционалним електронским уређајима од којих се ни не очекују падови система, попут телевизијских уређаја [23]. Такође, резултантан кернел је модуларнији и елегантнији од својих монолитних и хибридних сродника, те можемо са сигурношћу рећи да микрокернелски дизајн, или хибридни дизајн са фундаментално микрокернелским концепцијама, задовољава наше захтеве безбедности (1) и елегантности (3). Међутим, када гледамо кроз објектив перформанси, микрокернели се показују доста мање фаворабилно од монолитних архитектура. Конвенционална мудрост (коју Јохен Литге, архитекта L3 и L4 микрокернелских фамилија назива "фолклором"), чије је објашњење такође и интуитивно, говори нам да микрокернели имају знатно додатне трошкове у смислу времена егзекуције одређених основних кернелских операција. Ово произилази из основне идеје микрокернела да се за IPC користи слање имутабилних порука и идеје да се и комуникација са драјверима и осталим даемонима унутар микрокернелског система врши помоћу овог механизма. У поређењу са традиционалним монолитним кернелима, овај приступ је спорији због дупле контекстне промене (једном да се систем пребаци у кернелски мод, како би се посредством кернела послала порука, и други пут, када прималац бива нотификован од стране кернела да је порука пристигла), као и разних других операција које се тада морају извршити, а које зависе и

од имплементационих детаља (на пример, прослеђивање порука као низ бајтова подразумева одређено утрошено време на маршаловање и демаршаловање ове поруке (енгл. *marshalling/demarshalling*), где пошиљалац маршалује сложене објекте у низ бајтова који се кернелским примитивама може послати, а прималац демаршалује назад у оригиналан облик). С друге стране, када кориснички процес комуницира са монолитним кернелом, догађа се само једна контекстна промена, а када кернелске компоненте међусобно комуницирају, ових промена нема, а потреба за маршаловањем и демаршаловањем порука не постоји јер се ова комуникација врши директним прослеђивањем адреса ка структурама података, будући да су ове компоненте у истом адресном простору.

Међутим, истраживање на овом простору је врло активно, и у прошлости је дало сјајне резултате. Конкретно, L3 кернел успева да смањи IPC цену неколико редова величине од претходних дизајна (на пример, QNX-а и Mach-а), док његов наследник L4 даље побољшава ове резултате [10]. Такође, међу дизајнерима процесорских јединица овај проблем је врло познат, и чине се велики напори да се ово превазиђе—на пример, ARM процесори новијег датума садрже виртуелно-индексирано физички-таговану (енгл. *virtually-indexed physically-tagged, VIVT*) кеш меморију, која омогућава да оперативни систем знатно смањи трошкове који произилазе из чишћења кеш меморије (енгл. *cache flushing*) приликом контекстних промена и резултујућим промашајима кеш меморије (енгл. *cache misses*) који могу бити јако погубни за перформансе процеса између којих се ова промена дешава [12, 13]. Јасно је, дакле, да се овај конкретан проблем, који је у прошлости био изузетно велика препрека адопцији и задовољавајућим перформансама микрокернела, смањује великом брзином и са софтверске и са хардверске стране. Данас, микрокернелске виртуелизационе технологије попут Xen или seL4 базираних хипервизора могу да покрену виртуелне машине са целокупним паравиртуелним оперативним системом уз занемарљиво смањење брзине, а чак негде и извршавајући неке системске позиве брже него немодификован нативни систем [1].

Узимајући у обзир горенаведено, за НК се предлаже микрокернелска архитектура, са нагласком на идеју да је слање порука главни начин комуникације између процеса, као и да се драјверски и сви остали процеси невезани уско за функционисање кернела извршавају у посебним процесима који користе искључиво ове механизме за спољну комуникацију, без дељења меморије или претпостављања

било каквог дељења адресног простора са другим процесима или микрокернелским сервером. У наставку ћемо истражити на који начин ће предложени НК дизајн одступати од неких микрокернелских идеја, заједно са дискусијом о импликацијама ових одлука. IPC механизми ће такође бити дискутовани до разумног нивоа детаља, као и остали фундаментални концепти и примитиве које омогућују грађење комплексних нивоа апстракције над самим микрокернелским сервером.

2.3 Софтверски-одржана безбедност

Ипак, колико год се хардверски дизајнери трудили да минимизују *overhead* који се појављује приликом разних операција у вези са контекстним променама и колико год се кернелски дизајнери трудили да оптимизују своје имплементације тако да буду што ефикасније на датом хардверу, контекстне промене ће увек са собом носити одређену неповољну количину "баченог" времена. Промашаји кеш меморије, процесорски циклуси који се користе за чување и читање банкованих регистара, инвалидације TLB-а су проблеми који се могу смањити, али не и минимизовати. Свака промена адресног простора и улазак процесора у други прстен ће увек (бар у разумној даљој будућности) са собом носити не-нулти *overhead* приликом ових операција, који ће често бити незанемарљив са перформансне тачке гледишта. Иако се овакво чињенично стање ствари може интерпретирати као проста замена перформанси за безбедност (мотив који се у генералном облику појављује широм не само компјутерских наука, већ и инжењерства уопште), могу се тражити алтернативна решења која су способна да донесу одређене гаранције безбедности и одређене добитке на перформансама.

Тако је скорашњи тренд у истраживању кернелског дизајна и виртуелизационим технологијама, конкретно областима изолације неповерљивог кода и гаранције да се одређене класе багова критичних за безбедност и стабилност неће догађати, или да ће вероватноћа да се они догоде бити знатно смањена, посебна област која се може грубо назвати **софтверски-одржана безбедност (енгл. *software maintained safety*)**. У најопштијим цртама, ово је приступ у коме се не ослања на неке или све традиционалне хардверске механизме изолације процеса (*memory mapping*, процесорски прстени и сл.), већ се ова заштита постиже у софтверу.

Логично питање које се поставља у вези са овим приступом је—зашто у софтверу имплементирати неки механизам који је сам проце-

сор способан да нативно изврши? Можемо ли заиста да постигнемо већу ефикасност или безбедност "емулираном" изолацијом и протекцијом? Одговор, изгледа, лежи много дубље, у темељима нашег резоновања о оперативним системима и њиховим кернелима, те ћемо се сада осврнути на неке пројекте који су показали да можемо добити изузетно безбедне системе који су такође перформантнији: овај пут не помоћу паметних оптимизационих трикова већ одбацивањем четрдесетак година старе конвенционалне мудрости о начину на који се приступа развијању нових система.

Први пројекат који користи софтверски-одржану безбедност а који ћемо разматрати је *Google Native Client*, скраћено NaCl [26]. Битно је нагласити да овај пројекат нема за циљ да се користи за изолацију корисничких процеса у кернелу оперативног система, већ да безбедно покрене машински код који долази са Интернета. На x86 архитектури, за коју је оригинално развијен, NaCl користи врло креативан и необично маштовит приступ у коме се x86 сегментација, стара и махом некоришћена способност процесорске архитектуре, користи да изолује (*sandbox*-ује) машински код који је претходно генерисан модификованим LLVM генератором који поштује нека правила о емитованом коду. У детаље начина рада овог система нећемо сада улазити; довољно је рећи да се LLVM бајт код претвара у машински код који поштује неколико инваријанти које га чине статички анализабилним: да сви директни скокови буду на исправним границама инструкција, да се не позива софтверски прекид који може да оствари несупервизовану комуникацију са оперативним системом, а да индиректни скокови буду замењени псеудо-инструкцијом коју ће NaCl валидатор касније да претвори у специјалну секвенцу која проверава адресу како би осигурао да и она поштује инваријанте које морају да поштују и директни скокови. Заштита уписивања и читања из меморије потиче од споменутог креативног коришћења x86 сегментације; за више детаља видети оригиналан рад на ову тему. На основу NaCl-а изграђен је и *Zero VM*, виртуелизациона технологија која користи исти начин да покрене код за који може да гарантује да се неће понашати недозвољено и приступати недозвољеним ресурсима. ARM имплементација за 32-битну ARMv7 архитектуру такође постоји, међутим, она захтева да се све load и store инструкције замењују псеудо-инструкцијама, због недостатка хардверске сегментације на овој процесорској архитектури [4].

Следећи пројекат који ћемо размотрити је *RuntimeJS* кернел [20]. У питању је кернелска архитектура у којој се сви процеси (у тер-

минологији овог кернела названи *изолатима* (енгл. *isolates*)) извршавају у истом адресном простору, при чему кернел експлицитно не контролише никаква права приступа меморији нити разним другим ресурсима. Уместо тога, заштита се постиже имплицитно, самом чињеницом да су сви изолати писани у JavaScript програмском језику, који по својој семантици гарантује меморијску безбедност и изолираност кода који се извршава. Срж кернела је само V8 engine који покреће овај код, као и танак слој који поставља фондације за покретање и *sandboxing* изолата који се извршавају, као и рад са хардвером, учитавање драјвера и сл. Овај дизајн је интересантан по томе што се добија на перформансама јер се све извршава у једном адресном простору и процесорском прстену, тако да се сви проблеми са контекстним променама потпуно превазилазе, а такође се добија и на безбедности, јер постоји само једна *point of failure* тачка, а то је V8 који компајлује и покреће JavaScript код. Уколико је V8 безбедан и гарантује безбедност у складу са спецификацијом језика, елиминише се велика класа могућих багова који се једноставно не могу десити, са изузетком логичких багова у имплементацији одређених интерфејса, као и сигурносних пропуста у њиховом самом дизајну.

Највисокопрофилнији пројекат којим ћемо се бавити је *Microsoft Singularity* [6]. Ово је микрокернелска архитектура код које се такође сви кориснички процеси извршавају под најпривилегованијим процесорским прстеном, у истом адресном простору. Микрокернелска архитектура је одржана јер се одржава изолација *објектних простора* (енгл. *object spaces*) уместо адресних простора—то јест, изолација између процеса постиже се на нивоу објекта у програмском језику који се користи, уместо на нивоу меморије. Да би се ово постигло, за овај пројекат је развијен посебан програмски језик, Sing#, који гарантује меморијску безбедност на сличан начин на који то ради JavaScript у RuntimeJS кернелу. Разлика између ова два пројекта је знатно већи обим Singularity пројекта, као и експлицитан фокус на *релијабилност*, не на перформансе резултујућег система. IPC се постиже слањем порука као у стандардном микрокернелу, са интересантном разликом што је подршка за *уговоре* (*contracts*) уграђена у сам језик. *Уговори* су начин да се интерфејс и комуникација између нека два процеса у систему формално специфицира, заједно са типовима порука које могу да се шаљу путем одређеног канала, као и стањима у којима се канал може наћи, и транзицијама између њих. Ово омогућује нивое безбедности и стабилности који досад нису виђени у овој области, јер кернел експлицитно осигурава да се сва комуникација између процеса

догађа по спецификацији, да се канал не може наћи у недозвољеном стању или проћи кроз нелегалну транзицију, као и да се поруке морају послати и интерпретирати тачно на унапред дефинисан начин. Овде је такође једина "слаба" компонента *runtime* који покреће Sing# процесе, а већина гаранција безбедности директно проистиче из безбедности овог темеља. Тим који ради на овом пројекту такође изражава неке врло валидне поенте о застарелости оперативних система који су тренутно у употреби и наслеђу које сеже до седамдесетих година прошлог века, и постављају инспиративно питање—шта би било када бисмо сада, у ово време, од нуле написали нов оперативни систем, али овај пут са релијабилношћу, не перформансама на уму?

С обзиром на врло очевидне предности овог приступа, предлаже се да НК кернел буде написан у неком слично безбедном програмском језику, при чему ће се гаранција безбедности делегирати на једну компоненту, а то је компајлер коме се "верује" да неће емитовати небезбедан код, у ком случају се даемони оперативног система и кориснички процеси за које се може потврдити да нису, по дизајну, способни да изврше нелегалне операције, могу покренути у истом адресном простору и процесорском прстену као и кернел, што би довело до елегантног дизајна, умањило непотребну комплексност и знатно подигло безбедност овог микрокернела. Нажалост, до скоро није постојао квалитетан програмски језик са задовољавајућом брзином и минималним *overhead*-ом који се добија у замену за гарантовану безбедност генерисаног кода. Међутим, и та ситуација се скоро променила, изласком дуго очекиваног и развијаног програмског језика из бета верзије у верзију 1.0.0. У питању је *Rust* програмски језик, продукт пажљивог и вредног рада Mozilla заједнице.

2.4 Rust програмски језик

Нажалост, обим овог документа не дозвољава улажење у детаље функционисања и семантику Rust програмског језика. Напоменућемо да је у питању језик специфично дизајниран за системско програмирање и високе перформансе, као и да гарантује меморијску безбедност и његов *threading* модел предупређује *data race* ситуације. Rust компајлер није по дизајну онемогућен да емитује небезбедан код, али се све операције за које компајлер може закључити да нису безбедне морају извршити унутар посебних `unsafe` блокова и функција, што значи да се само у овом програмском језику може написати целокупан кернел, а да се делови ван `unsafe` блокова (велика већина самог

кода) неће понашати на непредвиђен начин. Rust не поседује *garbage collector*, већ користи концепт "поседовања" одређених података да гарантује правовремену деалокацију нерефернцираних објеката. Такође не поседује никакву виртуелну машину или *runtime* који надгледа рад програма, што значи да се све ове гаранције осигуравају у чистом машинском коду који генерише компајлер. Овај програмски језик је врло погодан за дизајн оперативног система, а због јаких гаранција безбедности које пружа његов компајлер идеалан је као језик избора за пројекат попут НК. С тим у вези, предлаже се да се НК кернелски код, као и драјвери и даемони који раде поред њега у кернелском простору, развије у овом програмском језику, како би се осигурао одређен ниво безбедности против безбедносних пропуста и багова који би могли да угрозе стабилност и безбедност целокупног система. Приметити да овакав приступ одступа од традиционалне микрокернелске архитектуре, јер ништа не спречава серверски даемон да експлицитно приступи или мења меморију која му не припада. Ипак:

- Сав део кода који има способност да ово уради мора бити експлицитно обележен, као гаранција компајлеру да неће урадити ништа што угрожава инваријанте које компајлер сматра тачнима у сваком тренутку. Веровање је да је у добро дизајнираном коду, постотак оваквог понашања изузетно мали, и да ће се ове компоненте кернела врло пажљиво тестирати како би се осигурало да не садрже безбедносне пропусте или багове горепоменутог типа.

- У било ком оперативном систему, било микрокернелском или монолитном, корисник мора веровати самом кернелском коду да је безбедан и да се неће понашати на недозвољен начин. Ово је фундаментална претпоставка која је у сваком случају тачна. НК мора имати и теоријски механизам хардверске изолације процеса из неповерљивих извора, али овакви процеси свакако не би требало да имају дозволу да оперирају као микрокернелски даемони и приступају хардверу или им бити поверени било какви подаци. (Ни у најбезбеднијем кернелском дизајну, ништа не спречава малициозни драјвер да угрози стабилност система злонамерним инструкцијама хардверу који контролише, нити мрежном драјверу да не прослеђује поверљиве податке трећој партији на мрежи.)

- Основни систем комуникације и даље остаје фундаментално микрокернелски; даемони не би требало да користе своје могућности да неометано приступе хардверу или арбитрарној меморији да угрозе иједан од основних принципа и претпоставки који су на снази у НК кернелу. У свим кернелским дизајнима, на пример, хардверски драјвери морају у већини случајева да буду способни да уписују и читају физичку меморију директно, како би остварили DMA (*direct memory access*) комуникацију са уређајима са којима су дужни да комуницирају. Ово значи да и традиционални микрокернели (у одсуству IOMMU контролера који би отклонили и овај безбедносни ризик) морају да теоријски дозволе да један драјвер заправо сруши цео систем, приступи поверљивим подацима и слично. Наравно, због уских захтева које DMA драјвери имају, верује се да ово није велики проблем, због тога што јако мало кернелских архитектура (и то врло специјализованих) пружа заштиту против *малициозних* драјвера—оно ка чему се стреми је пружање заштита против драјверских багова и сигурносних пропуста у њима.

Узевши у обзир ово, дакле, мишљење је аутора да ова одлука не нарушава микрокернелску архитектуру којој НК стреми, нити да је систем инхерентно мање безбедан и са мање безбедносних гаранција од постојећих микрокернела који се срећу у пракси.

2.5 Микрокернелске примитиве

Сваки микрокернелски сервер мора да подржава одређен број примитивних операција које би се тада имплементирале унутар њега, и оне би биле једине операције и функционалност самог микрокернелског сервера. Сви остали механизми, нивои апстракције, и функционалност се тада делегира осталим даемонима на конкретном систему, који су дужни да користе ове фундаменталне функције микрокернела као начин да изврше арбитрарне операције. Фокусираћемо се више на теоријске концепте него на начине да микрокернел омогући конкретне хардверске детаље, на пример, приступ разним BIOS функцијама, меморијски-мапована комуникација са хардверском, преемпција микронити, и слично. Ваља напоменути да је већина ових идеја инспирисана начином рада L3 и L4 микрокернелских фамилија [11, 10]. Ова кернелска породица (посебно L4, новијег датума) позната је по својој ефикасности и елегантности решења, у шта се и читалац

може уверити пажљивим проучавањем њеног дизајна или читањем неких од многобројних научних радова њеног примарног архитекте о дизајну микрокернела; свакако, топло се препоручују бар два горепоменута рада који ће свакако бити темељ за идеје и концепте који ће уследити. Такође треба имати на уму да сви ови концепти нису објашњени на најдетаљнији начин, са обзиром на просторна ограничења овог рада; количина детаља је баланс између употребљеног простора и нивоа који је потребно како би се концепт задовољавајуће разумео. Поново, за потпуно разумевање ових концепата, читалац се упућује на горепоменуте радове Ј. Литгеа, који је ове идеје и имплементирао у својим микрокернелима, чиме је доказао њихову практичну вредност за конкретну примену којом их је подвргнуо.

Овако дефинисана микрокернелска архитектура своди микрокернелски сервер у најужем смислу само на *interrupt vector*-е који одговарају на разне догађаје у систему и брину се о правилном и фер извршавању свих микронити на систему, као и на софтверски IRQ канал који служи да имплементира мали број системских позива који треба да имплементирају примитивне операције описане у овом одељку. Остатак функционалности имплементиран је ван овог уског система, по микрокернелској теорији.

2.5.1 Микронити

Микронит (енгл. *microthread*) је природна апстракција једног процеса на систему. Операције над микронитима су једне од основних функција кернела. На минималном нивоу, микрокернел мора да, на основу неког окидања тајмера или некој сличној *scheduling* стратегији да преемптује тренутно активне микронити, памтећи њихов егзекуциони контекст, и покреће оне које су тренутно на врху *scheduling* реда. Одређени микрокернели померају и ову функционалност ван кернела, али је дискутабилно да ли ово поједностављује или додатно компликује систем, с обзиром на то да је преемпција витална функционалност кернела и изузетно специфична процесорској архитектури на којој се он извршава, па се не може са сигурношћу тврдити да се њеним померањем ван микрокернелског сервера постиже бржи и робуснији систем.

Микронит може радити у корисничком или кернелском моду, а и сам микрокернелски сервер се теоријски може састојати из више микронити које одговарају на одређене IPC поруке процеса у систему, могућно и на различитим физичким процесорима. У овом тренутку

није јасно хоће ли постојати потреба за покретањем и микрокернелског сервера као микронити са неким специјалним својствима, или ће се интуитивније приказати приступ у коме се и та функционалност делегира даемонима ван самог микрокернелског сервера.

Свака микронит је повезана са одређеним адресним простором који постаје активан када микронит дође на ред за извршавање на процесору. У случају *kernel-mode* микронити, физички адресни простор остаје исти и, теоријски, не постоје никакве рестрикције које је микронит приморана да испоштује. Ипак, НК би требало да има и Rust апстракциони ниво који неће дозволити да микронит приступи меморијским странама које јој не припадају, нити да проба да изврши неку илегалну операцију—сепарација и изолација у том смислу ослања се на безбедност Rust кода, евентуално модификованог компајлера и стандардне библиотеке тако да подржава само безбедне операције унутар адресног простора микронити. Приметити да, с обзиром на то да нам је циљна архитектура ARMv8-A, имамо на располагању огроман 64-битни адресни простор, тако да можемо да приуштимо јединствене адресне псеудо-просторе унутар главног кернелског виртуелног адресног простора. (Више о адресним просторима касније.) Микронит је такође асоцирана са јединственим идентификатором који је јединствено дефинише у времену од кад је систем подигнут. Јединствени идентификатор је потребан како би се успоставио начин да микронити међусобно комуницирају помоћу IPC-а. Једна кернелска микронит (*нулта микронит*) чији је јединствени идентификатор увек дефинисан као константа 0 постоји како би одговорила на *bootstrapping* захтеве нових микронити, односно да одговори на упите о тренутно покренутим даемонима и слично. Микронит такође може да буде мапирана тако да јој буде дозвољено да прима одређене, одобрене хардверске прекиде преведене од стране микрокернелског сервера у IPC поруке, а регулација комуникације са хардвером помоћу меморијски-мапираног улаза/излаза се постиже поново кроз Rust подсистем који је дужан да осигура да се приступа само портовима и хардверски-видљивој меморији на предвиђен, безбедан начин.

2.5.2 IPC

Слање и примање порука је фундаментална карактеристика микрокернела, тако да се мора дефинисати унутар микрокернелског сервера јер представља градивни блок за све остале функционалности

које ће систем моћи да подржи. IPC је такође градивни блок за следећу ставку којом ћемо се бавити (руковођење меморијом), тако да без дефинисања IPC-а не можемо ни да дефинишемо остале микрокернелске примитиве. Сродно већини модерних микрокернела [10], предлаже се да НК пружа примитиве које омогућују синхрону комуникацију слањем арбитрарних порука, при чему перформанса овог јако битног градивног елемента изузетно зависи од конструкције ових порука и начину на који се оне преносе, што је имплементациони детаљ. Синхрона комуникација такође има природну последицу да се комуникација увек врши са пристанком обе партије (једна, која позива блокирајући `receive` системски позив који ће блокиати док год не пристигне порука за ту микронит, и друга, која позива блокирајући системски позив `send`, који блокира све док друга партија не прими поруку). При овоме се треба контролисати *timeout* вредност и разне шеме којима малициозна партија може да изазове *denial-of-service* напад над кернелом. Улажење у овакве детаље је ван обима овог рада, читалац се упућује на имплементације у L4 фамилији микрокернела, као и MINIX 3, као примере добро архитектованих IPC механизама. Асинхрона комуникација се може изградити изнад ове постојеће методе синхроне комуникације.

Приликом слања ових порука између две корисничке микронити, наивна имплементација би поруку копирала прво једном у кернелски простор из пошиљаоца, а онда из кернелског простора у адресни простор примаоца. Разлог овоме је што оба адресна простора типично имају *read-only* мапиране кернелске странице које служе као бафери управо у ове сврхе. Међутим, исто решење се може постићи и коришћењем *привременог мапинга* (енгл. *temporary mapping*), који захтева само једну копију поруке. За детаље око конкретне имплементације и врло битних платформских оптимизација који чине овај цео IPC механизам практично применљив, видети [10]. Ваља напоменути да је у комуникацији две кернелске микронити, један IPC позив само прост позив у микрокернелски сервер који ће онда извршити неопходне операције (копирање поруке пошиљаоцу и склањање истог из кернелске листе чекања (енгл. *wait queue*), на пример), што га чини изузетно јефтином операцијом.

2.5.3 Руковођење меморијом

Предлаже се да НК користи сличне меморијске примитиве као и L3/L4 породица микрокернела. Детаљи се могу видети у [11], следи

скраћено и незнатно поједностављено објашњење овог решења.

Фундаментални концепт је да свака микронит τ има свој адресни простор σ_τ, односно, скуп уређених парова облика (V_i, P_i) где је V_i виртуелна адреса а P_i физичка адреса одређене странице (енгл. *page*) у меморији. Микронитима су дозвољене примитивне операције над овим адресним просторима, које испуњава микрокернелски сервер, при чему се ни у једном тренутку не може извршити ниједна операција која мења адресни простор неке микронити у нови адресни простор σ'_τ тако да σ'_τ даје микронити веће право над операцијама над меморијом него σ_τ. У почетку, основна, коренска микронит има адресни простор σ_0 у коме се мапиране налазе све странице у физичкој меморији, а адресни простори свих осталих микронити су празне. Коренска микронит онда може да подели овај адресни простор на произвољан начин, што је и довело до тога да се овај приступ назива *рекурзивним адресним просторима*. Ова елегантна архитектурална одлука има као резултат могућност да се меморијски руководиоци (енгл. *memory managers*) и пејџери имплементирају као посебни кориснички процеси, ефективно одвајајући руководство меморијом од микрокернелског сервера. Примитивне операције које овакав микрокернел подржава над адресним просторима следе. Све операције захтевају да се обе партије сложе и прихвате операцију помоћу IPC-а, пре него што се она стварно и деси.

1. **Grant.** Микронит τ_1 предаје микронити τ_2 меморијску страну P. P бива мапирана на некој адреси V у τ_2, док се мапирање P брише из адресног простора τ_1 (оригинална микронит је се одриче). Ово је корисно када треба проследити неке податке без потребе да пошиљалац поново приступа тим подацима; на пример, ово омогућава ефикасно читање фајла у меморију од стране *filesystem* микронити и прослеђивање прочитаног некој корисничкој која је затражила ту операцију.

2. **Map.** Микронит τ_1 мапира у адресни простор микронити τ_2 меморијску страну P, при чему сада две микронити имају ту исту страну мапирану негде у виртуелној меморији, на адресама V_1 и V_2.

3. **Flush.** Микронит τ позива flush операцију над својом меморијском страном P. Страна P остаје мапирана у адресном простору τ али бива одмапирана у адресним просторима свих микронити у којима је досад била мапирана. Ова операција је безбедна

јер по дизајну, приликом мапирања микронит у коју се мапира страница се сложила и мора бити спремна на потенцијалну flush операцију над новомапираном страницом од стране правог власника.

Ове примитивне операције могу инкорпорирати и специфичне операције над дозволама (read, write, execute) за сваку страницу, као укључивати и архитектурално-специфичне додатке традиционалном руководству меморије. Ипак, дискусија на ту тему је посебна и превише обимна за укључивање овде, поново, читалац се може обратити [11] за више информација о овој техници. Предлаже се да ово буде начин на који микрокернелски сервер у НК руководи меморијом.

2.6 Егзекуциони сервери

Када се микрокернел покрене, он мапира свој иницијални код на одређене адресе у виртуелној меморији, и покреће нулту микронит (аналогно концепту `init` процеса на GNU/Linux системима, на пример) која је такође кернелска микронит. Микрокернелски сервер пружа системски позив који може да направи нову нит, као и ниво апстракције који може да мапира одређени код у егзекутабилни адресни простор микронити и изврши скок на њега, тиме замењујући тренутни *executable image* другим (слично као `fork()/exec()` позив на Unix системима). Корисничке микронити, које су хардверски изоловане од кернелског адресног простора, ово могу даураде без икаквих рестрикција, наравно, под условом да се одрже инваријанте о адресном простору, и да, уколико постоји неки конкретан систем контроле приступа хардверу, фајловима или сл. (на пример, *multi-user system*), наследнички процеси морају да наследе атрибуте еквивалентне онима које има родитељ, или строжије. Ово је врло слично процесном систему у традиционалним Unix—наравно, изнад примитивних функција микрокернелског сервера морао би да се изгради солидан ниво апстракције који би могао да апстракује микронити као процесе и да прати њихове рестрикције и регулације у вези са дозвољеним и недозвољеним функцијама.

Што се тиче кернелских микронити, уводимо концепт *егзекуционих сервера*. Егзекуциони сервер је одређена кернелска микронит која је способна да ради као сурогат и изврши неки код у произвољном формату. На пример, Java виртуелна машина би могла бити имплементирана као егзекуциони сервер који прихвата `.jar` фајлове

и извршава их у исправном контексту и са исправним привилегијама, у кернелском адресном простору. Такође, могао би да се добије и ефекат сличан RuntimeJS кернелу тако што би се V8 виртуелна машина за JavaScript покренула као егзекуциони сервер и била способна да извршава JavaScript код. По дефиницији и као свесна архитектурална одлука, од егзекуционог сервера се може затражити извршавање одређеног кода, и на њему је да осигура да не буде било каквог пробијања безбедности, као и да извршава тај код са истим привилегијама које има микронит која је његово извршавање затражила. Било која кернелска микронит може да региструје себе као егзекуциони сервер помоћу поруке нултој микронити, а по конвенцији, нулта микронит тада има функционалност којом може да буде упитана за покретање одређеног фајла од стране арбитрарне корисничке микронити. Када прими такав захтев, нулта микронит (такође названа и *коренски егзекуциони сервер*) испитује егзекуционе сервере који су код ње регистровани, и одлучује да ли може да покрене одређени тип кода. Исто тако, и егзекуциони сервери могу да прихвате регистрацију неког подсервера којима су они сурогат, тако да ово ствара хијерархију егзекуционих сервера где се, приликом покретања неког фајла, силази низ стабло у потрази за тачним сервером који може адекватно да изврши одређени фајл. Наравно, овај апстрактни концепт мора бити аугментован апстрактним API-јем који ће омогућити интуитивно и транспарентно покретање разних извршних фајлова, када неки процес то затражи.

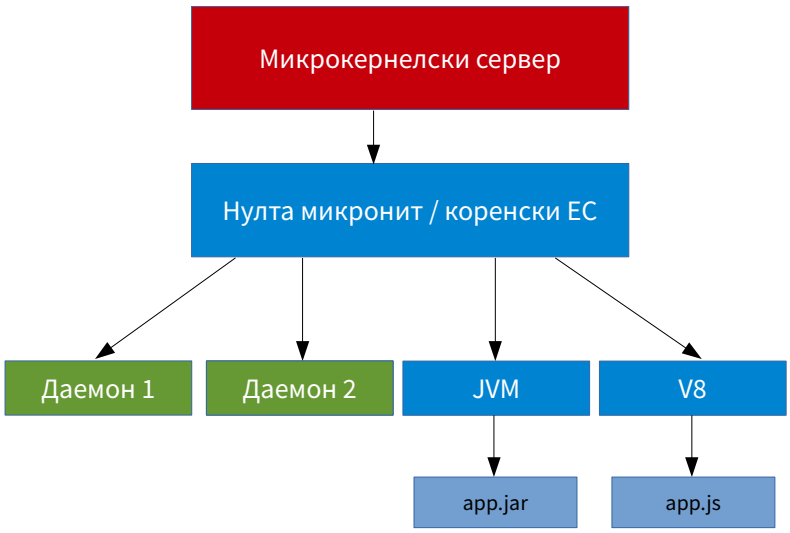
При томе, нулта микронит односно коренски егзекуциони сервер би дозвољавао егзекуцију само провереном и машинском коду коме се верује—ово би могло бити сачувано, на пример, као криптографски хешеви извршних датотека за које се зна да потичу од оперативног система и не долазе из неповерљивих извора. Тако би, иако би било који кориснички процес могао да захтева егзекуцију произвољне обичне извршне датотеке од коренског егзекуционог сервера, овај захтев би био одбијен јер коренски егзекуциони сервер верује само провереним извршним датотекама. На тај начин би поверљиви даемони који пружају разне услуге могли да се покрену као корисничке микронити и да захтевају извршење самог себе од нулте микронити, што би ефективно довело до њихове промоције у кернелски простор, као кернелска микронит.

Разлог овакве организације је једноставан и посебно користан за паметне уређаје који су наше циљно тржиште. Апликације за мобилни оперативни систем изграђен на оваквој архитектури се тада

могу пласирати као нативни код који би се извршавао микронит у корисничком моду, или као бајт или изворни код у неком другом језику који мора да буде извршен од стране неког *run-time*-а, односно, није самостојећи. У том случају би се, ако се виртуелна машина покрене као обична корисничка микронит, нажалост, плаћала цена и хардверске изолације и софтверске, која свакако мора да се плати како би се одржале гаранције сигурности које захтевају ови програмски језици од својих виртуелних машина. У том случају би, ако жели да се покрене у кернелском простору као и остале микронити, такав програм морао бити у форми машинског кода компајлованог неким безбедним језиком попут Rust-а, али у том случају кернел никако не може проверити да ли је машински код малициозан, јер се не може доказати да је конкретан машински код настао од неког конкретног компајлера (а и поред тога, Rust компајлер дозвољава експлицитне небезбедне операције по дизајну), па би такав ризик био превелик и апсолутно недопустив. Са друге стране, помоћу модела извршних датотека којима се имплицитно верује, покрећу се, између осталих, и даемони који представљају виртуелне машине за више програмских језика и окружења и који су спремни да изврше њихов код на захтев и "у име" неког корисничког процеса, постижу се супериорне перформансе јер такав код ужива брзу комуникацију са кернелом јер се извршава у истом адресном простору, а одржава се и безбедност, али овај пут не хардверским већ софтверским путем, као у Microsoft Singularity или RuntimeJS кернелу.

Такође би се, за дистрибуцију нативног кода који би ипак могао да ради у истом адресном простору као и кернел, могла истражити и могућност да се имплементира егзекуциони сервер који прати Google NaCl (или PNaCl) систем, при чему би такав сервер прво извршио верификацију над извршним фајлом, а уколико верификација буде успешна, и покренуо га.

Слика 3: Графички приказ предложеног модела егзекуционих сервера за НК у хипотетичкој ситуацији из праксе где су подигнути Java виртуелна машина и JavaScript V8 који раде као егзекуциони сервери и *sandbox*-ујју непривилегован код у кернелском простору.



3 Поговор

3.1 Необрађене теме и даље истраживање

Нажалост, због ограниченог обима овог рада, није обрађен ни мали део аспеката који чине један комплетан микрокернел. Уложен је труд да се представе кључне и најбитније одлуке, углавном оне које са најмање среће у другим системима, и да се представе иновативна савремена решења из којих је корисно црпети инспирацију. Ипак, поставља се питање употребне вредности описаних одлука, с обзиром на то да нису тестиране у пракси, или бар нису тестиране у исправном контексту за наш конкретан случај. НК је тренутно само хипотетички микрокернел са неколико кључних идеја које се могу показати исправним, али и беспотребним или много мање корисним него што се чинило. Ипак, надам се да сам представио интересантне, мало виђене приступе развоју кернела оперативних система, које не морају бити употребљиве у изворном облику како би се видела њихова вредност. Начин размишљања који је довео до њих је сасвим другачији од начина размишљања који је довео до идеја које тренутно доминирају екосистемом, а можда је нов начин размишљања баш оно што треба том екосистему како би откључао врата до још већих, нових, и ултимативно јако корисних иновација.

Није обрађен процес подизања НК система, симетрични мултипроцесинг (енгл. *symmetric multiprocessing, SMP*), виртуелни фајл систем, нивои апстракције који су апсолутно неопходни како би НК био комплетан и употребљив систем, као и сијасет других ставки које су подједнако битне и подједнако одсутне из овог рада. Ипак, верујем да је НК размишљање у добром, или бар маштовитијем правцу од оног на кога су истраживачи који су специјализовани у традиционалним и општепознатим системима навикли.

Постоји много идеја које сам желео да споменем и објасним у овом раду, али сам на крају морао да се одлучим и одаберем неколико кључних, најбитнијих, без којих остатак не би имао много смисла. НК је кернел који је могуће архитектовати, полазећи од идеја донесених овде (или неког њиховог подскупа) и имплементирати, како би се ултимативно добио нови мобилни оперативни систем који је, надајмо се, ефикаснији, безбеднији, бржи и (што је можда још битније од свега) отворенији, за разлику од опција које се тренутно нуде.

3.2 Наслеђе и парализа

Као што је Гери Бернхарт (*Gary Bernhardt*) то врло лепо рекао у свом хумористичном али болно истинитом говору [2], софтверска индсутрија се налази у "наслеђу и парализи" (*legacy and paralysis*). Брзина којом је ово поље просто експлодирало од неколико универзитета у Америци и мултимилионских *mainframe* рачунара па све до феномена који је данас омогућио да носимо целокупно знање човечанства у џепу, није оставила довољно времена да заправо размишљамо о дуготрајности онога што правимо. У журби да се што пре стигне са иновацијом, иновација је заостала, а оно што је заправо стигло биле су закрпе преко закрпа старих решења чије се комплексности већина нас искрено боји. Овај исти (или јако сличан) сентимент исказао је и изврстан тим из Microsoft Research-а у прегледу њиховог новог Singularity система о коме је било речи раније [6], а изгледа да то нису једина два таква мишљења. Генерално пољем рачунарства струји нека лагана али упорна језа—јесмо ли нешто заборавили? Јесмо ли дали све од себе?

Данас, више од 40 година након што су Брајан Керниган и Денис Ричи (*Brian Kernighan, Dennis Ritchie*) осмислили програмски језик C, у њему и даље пишемо нове оперативне системе и нове алате за које је он сасвим погрешан избор. Људи су готово религиозно везани за технологије које су им познате, због тога што се плаше да ће све то њихово знање постати беспотребно када се те технологије замене. Ово је инстинктивна реакција—то није нешто што људи (бар већина нас) може да контролише. То, и притисак на индустрију да доноси још иновације—сада, одмах—довели су до тога да сервери који процесуирају милијарде долара дневно стоје на систему који опонаша други систем који је настао пре више од 40 година. У своје време, Unix је био дело генија, али зашто се и даље сматра правом архитектуром, када је просечан корисник рачунара од MIT студента који ради на својој докторској тези постао тинејџер у средњој школи, и када је број повезаних рачунара скочио са можда пар стотина на више од 4 милијарде? Није било времена за праву иновацију—иновацију која се појави и покаже нам колико смо дуго тапкали у мраку и унесе револуцију у целу индустрију.

Људи приказују очигледну љутњу када се уопште спомене тема о томе како нас старе технологије и огромно наслеђе из зоре компјутерских наука више не служи како треба. Оно што ће вам тада рећи је да не разумете, да су ту технологију осмислили много паметнији људи

од вас, и да су уложили много више времена да о њој размишљају него што ви икада хоћете, и да је нисте схватили довољно како бисте о њој судили. А ја питам, да ли је и најпаметнији човек у то време могао да предвиди колико ће далеко догурати архитектура коју је тад осмислио? И да јесте, да ли би тада могао да донесе исправну одлуку о правцу у коме треба да је одведе? Моје мишљење је да није могао, и да је то разлог због кога понекад морамо да се поздравимо са тим масивним наслеђем и да почнемо изнова.

Али почети изнова није лако. У питању је индустрија од готово пола трилиона долара која лежи на свом том наслеђу које је најбоље само ишчупати одатле. Али колико ће нас дуго темељи које смо имали служити док не почну да попуштају? Одлука да се C стрингови завршавају нултим бајтом уместо да се користе два или четири додатна бајта да се упише њихова дужина, како би се уштедео један или три бајта (у то време сасвим исправна имплементациона одлука), данас је одлука која је донела незамисливо много штете целој индустрији због сигурносних пропуста који су због тога настали [8]. И да ли је то била само једна таква одлука, или је свака од њих у неком капацитету, због тога што је нисмо исправили, донела много штете, изгубљеног времена и тапкања у мраку како би се задовољили стари стандарди уместо да се осмисле нови, савремени?

Овим својим радом и предлагањем нове кернелске архитектуре, ја стојим на раменима џинова—академика, програмера и хакера који су сви дали допринос истраживању ове области у нади да ће тиме допринети рачунарској индустрији. Њихов рад није заборављен, али не може се рећи да га се много људи сећа. Сви су презаузети прављењем нове апликације која ће им донети брзу зараду, од које ће постати милионери преко ноћи. Нико од њих се не брине колико смо тешку зграду саградили на старом темељу, и колико ће бити времена док се тај темељ коначно не уруши.

Без обзира на узалудност истраживања једне нове идеје, и старих идеја које је неко гурнуо са стране и ту их заборавио, програмирању је потребан свеж почетак (или бар мало свежији од почетака које је досад имало). У време када је настао Unix, програмери су програмирали само зато што су искрено уживали у томе. Та култура и даље постоји, али је невидљива у мору оних које није брига какве жртве се праве, док год је њихов циљ—а циљ је, разуме се, краткотрајни пословни добитак—испуњен. На коју год страну да крене та индустрија, чак и ако у наредних месец дана поново доживимо фамозни *dot-com crash*, ово је мој мали допринос померању са статуса кво, и иновацији,

елегантности и естетској лепоти програмерског заната. То је почетнички допринос, и допринос који вреди ништавно мало у поређењу са доприносом генија који су се овом облашћу бавили деценијама, али је ма какав допринос дужност сваког коме је стало да једног дана индустријом доминирају системи који су наше слуге, а не обрнуто.

Уосталом, како је могуће чисте савести мирно седети по страни и гледати кулу од карата како се руши, ако бар верујете да можете допринети њеном спашавању?

Референце

[1] Paul Barham et al. “Xen and the art of virtualization”. English. Унутар: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 164–177.

[2] Gary Bernhardt. *A Whole New World*. 2012. URL: `https://www.destroyallsoftware.com/talks/a-whole-new-world` (visited on 05/28/2015).

[3] Michael Flynn. *Computer architecture : pipelined and parallel processor design*. English. Boston, MA: Jones and Bartlett, 1995. ISBN: 9780867202045.

[4] Google. *ARM 32-bit Sandbox*. English. URL: `https://developer.chrome.com/native-client/reference/sandbox_internals/arm-32-bit-sandbox` (visited on 05/28/2015).

[5] Paul Graham. *The Word "Hacker"*. English. 2004. URL: `http://www.paulgraham.com/gba.html` (visited on 05/10/2015).

[6] Galen Hunt et al. “An Overview of the Singularity Project”. English. Унутар: (2005).

[7] *Hybrid (micro)kernels*. English. 2006. URL: `http://www.realworldtech.com/forum/?threadid=65936&curpostid=65915` (visited on 05/25/2015).

[8] Poul-Henning Kamp. “The most expensive one-byte mistake”. English. Унутар: *Communications of the ACM* 54.9 (2011), pp. 42–44.

[9] Chuanpeng Li, Chen Ding, and Kai Shen. “Quantifying the cost of context switch”. English. Унутар: *Proceedings of the 2007 workshop on Experimental computer science*. ACM. 2007, p. 2.

[10] Jochen Liedtke. “Improving IPC by kernel design”. English. Унутар: *ACM SIGOPS Operating Systems Review*. Vol. 27. 5. ACM. 1994, pp. 175–188.

[11] Jochen Liedtke. *On micro-kernel construction*. English. Vol. 29. 5. ACM, 1995.

[12] ARM Holdings Ltd. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. English. Верзија 0406C.c. May 20, 2014. URL: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html`.

[13] ARM Holdings Ltd. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. English. Верзија 0487A.f. 2015. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.architecture/index.html.

[14] ARM Holdings Ltd. *Procedure Call Standard for the ARM Architecture*. English. Верзија ABI r2.09. 2012. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0041c/BGBGFIDA.html.

[15] Samsung Asia Pte Ltd. *Warranty Information*. English. 2015. URL: http://www.samsung.com/sg/support/warranty/ (visited on 05/10/2015).

[16] Alan MacCormack. "Product-Development Practices That Work: How Internet Companies Build Software". English. Унутар: *MIT Sloan Management Rev.* 42.2 (2001), pp. 75–84.

[17] Daniel D. McCracken and Michael A. Jackson. "Life Cycle Concept Considered Harmful". English. Унутар: *SIGSOFT Softw. Eng. Notes* 7.2 (Apr. 1982), pp. 29–32. ISSN: 0163-5948. DOI: 10.1145/1005937.1005943. URL: http://doi.acm.org/10.1145/1005937.1005943.

[18] Paul E McKenney et al. "Read-copy update". English. Унутар: *AUUG Conference Proceedings*. AUUG, Inc. 2001, p. 175.

[19] Stefan Poslad. *Ubiquitous computing : smart devices, environments and interactions*. English. Chichester, U.K: Wiley, 2009. ISBN: 978-0-470-03560-3.

[20] runtime.js. *Runtime.JS - kernel built on V8 JavaScript engine*. English. URL: http://runtimejs.org (visited on 05/28/2015).

[21] Carl van Schaik and Gernot Heiser. "High-performance microkernels and virtualisation on ARM and segmented architectures". English. Унутар: *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems, Sydney, Australia*. 2007.

[22] Andrew S. Tanenbaum. *LINUX is obsolete*. English. 1992. URL: https://groups.google.com/forum/#!topic/comp.os.minix/wlhw16QWltI[1-25] (visited on 05/26/2015).

[23] Andrew S Tanenbaum, Jorrit N Herder, and Herbert Bos. "Can we make operating systems reliable and secure?" English. Унутар: *Computer* 39.5 (2006), pp. 44–51.

[24] Darcy Travlos. “ARM Holdings and Qualcomm, The Winners in Mobile”. English. Унутар: *Forbes Magazine* (2013). URL: http://www.forbes.com/sites/darcytravlos/2013/02/28/arm-holdings-and-qualcomm-the-winners-in-mobile (visited on 05/13/2015).

[25] Karim Yaghmour. *Embedded Android*. English. Index. Sebastopol, CA: O’Reilly Media, 2013. ISBN: 978-1-449-30829-2. URL: http://opac.inria.fr/record=b1134213.

[26] Bennet Yee et al. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”. English. Унутар: *IEEE Symposium on Security and Privacy (Oakland’09)*. 2009. URL: http://nativeclient.googlecode.com/svn/data/docs_tarball/nacl/googleclient/native_client/documentation/nacl_paper.pdf.