UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Ratko Rudič

Mikrojedrni operacijski sistemi

MAGISTRSKO DELO

Mentor: prof. dr. Borut Robič

Ljubljana, 2016

Zahvala

Za pomoč pri izdelavi magistrske naloge,za vse nasvete in usmeritve, se zahvaljujem mentorju prof. Borutu Robiču. Zahvaljujem se tudi osebju v podiplomski pisarni, brez katerih bi mi ušel še zadnji rok.

Sodelavcem iz OnLine oddelka na POP TV-ju se zahvaljujem za moralno podporo in za pomoč pri dobavi strojne opreme, brez katere bi bil praktični preizkus v nalogi veliko zahtevnejši.

Ne nazadnje se zahvaljujem še širši in ožji družini, ki me je vas čas spodbujala in mi dala čas za ta veliki podvig.

Vsem mojim

Kazalo

1 Uvod 5

2 Operacijski sistem 7
- 2.1 Kaj je operacijski sistem 7
- 2.2 Nastanek operacijskih sistemov 9
  - 2.2.1 Prva generacija:1945 - 1955 9
  - 2.2.2 Druga generacija (1955 - 1965) 10
  - 2.2.3 Tretja generacija (1965 - 1980) 10
  - 2.2.4 Četrta generacija (1980 - danes) 10
  - 2.2.5 Peta generacija (1990 - danes) 10
- 2.3 Tipi operacijskih sistemov 11

3 Mikrojedrni operacijski sistemi 15
- 3.1 Osnove 15
- 3.2 Zakaj - primer uporabe 16
  - 3.2.1 Srčni spodbujevalnik 16
  - 3.2.2 Avtomobili in letala 17
- 3.3 Nastanek 17
- 3.4 Generacije mikrojedrnih sistemov 18
  - 3.4.1 Prva generacija 18
  - 3.4.2 Druga generacija 21
  - 3.4.3 Tretja generacija 24
- 3.5 Zmogljivosti 27
- 3.6 Prednosti 27

3.6.1 Varnost 27
3.6.2 Enostavnost 28
3.6.3 Razširljivost in prilagodljivost 28
3.6.4 Nadgradljivost 28
3.7 Tipi mikrojedrnih aplikacij 29

4 MINIX 31
4.1 Uvod 31
4.2 MINIX3 jedro 32
4.3 Reinkarnacija strežnikov 34
4.4 Komunikacija med procesi 35
4.5 Posodobitve med delovanjem 36
4.5.1 Koraki posodobitve 36
4.5.2 Sočasne posodobitve. 37
4.6 Zgradba gonilnikov in strežnikov 37
4.6.1 Primer preprostega gonilnika 39
4.6.2 Reinkarnacijski strežnik 40

5 L4 43
5.1 Kratka zgodovina 44
5.2 Abstrakcije 45
5.3 Fiasco.OC 51
5.4 seL4 52
5.4.1 Uvod 52
5.4.2 Izdelava in verifikacija jedra 53

6 Uporaba mikrojeder 60
6.1 Mach 60
6.2 L4 60
6.3 QNX 61
6.4 PikeOS 61
6.5 Symbian 62

6.6 Singularity 62
6.7 μ-velOSity 63

7 Kritike mikrojeder 64
7.1 Opis 64
7.2 Količina kode 64
7.3 Zahtevnost sistema 65
7.4 Stabilnost sistema 65
7.5 Boljše zmogljivosti 66

8 Kratka primerjava 67
8.1 Primerjava arhitektur 67
8.2 Primerjava hitrosti 68

9 Prihodnost 69

10 Zaključek 70

Seznam slik 71
Seznam tabel 72

Literatura 73

Povzetek

Magistrska naloga opisuje zgodovino, zasnovo in razvoj mikrojedrnih operacijskih sistemov. Ideje o mikrojedrni arhitekturi so se pojavile že zelo zgodaj v računalništvu, kar je presenetljivo glede na majhen delež, ki jih te sistemi danes obvladujejo.

Družina mikrojedrnih sistemov je sestavljena iz več različnih skupin, ki so velikokrat med seboj razvojno ali idejno povezane.Magistrska naloga opiše glavne predstavnike teh skupin,njihove medsebojne relacijeposebnostiter morebitne pomanjkljivosti.

Naloga predstavi nekaj primerov uporabe mikrojeder v specializiranih, vgradnih sistemih in poda razloge, zakaj je njihova izbira v teh primerih boljša od monolitne.

Ključne besede:

Mikrojedrni operacijski sistemi, hibridni sistemi, monolitni sistemi, Liedtke, L4, seL4, Mach, Minix

Abstract

This thesis outlines history, development and design of microkernel based operating systems. While the idea of microkernel based architecture emerged early in computer science, there are still very few systems adopting this technology.

Microkernel operating systems are divided into several different groups and families that frequently share rules, design decisions and development processes. This master thesis focuses on the main representatives of these groups as well as their mutual relationships, features and potential weaknesses.

In addition, it presents use cases of microkernel systems in various specialized embedded applications and offers insight and further explanations of the advantages that microkernel systems have over monolithic systems.

Keywords:

Microkernel operating systems, hybrid systems, monolythic systems, Liedtke, L4, seL4, Mach, Minix

Poglavje 1

Uvod

Operacijski sistem računalnika je uporabniku vse bolj skrit. Postaja le eden od procesov vedno bolj dinamičnih in vizualno lepih namizij na računalniku ali telefonu. Vsi ti računalniki so zapleteni sistemi, ki delujejo na na prav tako zapletenih operacijskih sistemih in njihovih jedrih.

Izoblikovalo se je obratno sorazmerje med uporabniki in jedri: operacijski sistemi imajo vse večjo zahtevnost in odgovornost ter hkrati vse manjšo prepoznavnost med uporabniki.

Pred leti je imel računalnik ime po operacijskem sistemu: podjetja so imela v lasti VAX, IBM ali Windows NT sisteme, danes pa uporabniki naslavljajo računalnike glede na grafično okolje, ki ga imajo pred seboj: Mac, Ubuntu, Windows, Android. Tako poimenovanje ni napačno, kaže pa, da se malo uporabnikov zaveda, kateri proces v računalniku ima največjo moč in najtežje delo.

Tema magistrske naloge so mikrojedrni operacijski sistemi - njihov razvoj in novosti, ki so jih bili deležni v zadnjih desetletjih. Ti sistemi uvajajo zelo zanimive spremembe v delovanju in zgradbi jedra od pomanjšanja jedra na minimalno osnovo, hitre med-procesne komunikacije do matematične verifikacije pravilnega delovanja. Nekatere od teh sprememb so v nalogi natančno opisane, za nekatere pa bi bilo potrebno veliko več časa in prostora. V tem primeru so v literaturi navedeni viri, kjer lahko bralec izve več podrobnosti.

Trenutno najbolj uporabljeni operacijski sistemi (Windows, MacOS, Linux) so bodisi monolitni bodisi hibridni sistemi. Pri njih deluje jedro kot homogena celota v prioritetnem načinu, zaradi česar lahko napaka zaustavi delovanje celega računalnika. Monolitni sistemi so zaradi naraščajoče zahtevnosti tudi težje razumljivi in precej zapleteni za obvladovanje ali spreminjanje.

Mikrojedra uvajajo v operacijske sisteme bolj sistematičen in pregleden pristop. Njihovo jedro je majhno in ima malo sistemskih klicev. Naloge datotečnega sistema, upravljanje s pomnilnikom, itd., opravljajo posebni operacijski strežniki, ki jih jedro povezuje v celotno okolje.

Magistrska naloga je razdeljena na devet delov. Prvi del razloži pojem operacijskega sistema, nastanek in trenutne različice. Drugi del opiše glavno temo magistrske naloge - mikrojedrne operacijske sisteme. Tretji in četrti del podrobno opišeta dva najbolj poznana sistema z mikrojedrom, peto poglavje našteje, kje se ti sistemi uporabljajo. Sledita poglavji, kjer so opisane kritike ter kratka primerjava med sistemi. Zadnji dve poglavji sta opis verjetnega prihodnjega razvoja in zaključek naloge.

Poglavje 2

Operacijski sistem

2.1 Kaj je operacijski sistem

Vsak računalnik je sestavljen iz več manjših delov, elektronskih komponent : procesor, tipkovnica, ekran, RAM, trdi diski in podobno. Da lahko računalnik deluje kot homogena celota, mora nekdo (ali nekaj) te komponente upravljati. Ta “nekdo” je jedro oziroma operacijski sistem računalnika.

Navadni uporabnik z operacijskim sistemom nima neposrednega stika. Rokuje z uporabniškimi aplikacijami (brskalnik, kalkulator, bralnik novic), jedro pa omogoča računalniku, da te aplikacije delujejo odzivno. Ko uporabniški program želi shraniti sliko, je ne shrani program sam, ampak to nalogo poda operacijskemu sistemu. Ta nalogo prevzame, shrani sliko in obvesti uporabniški program, da je nalogo uspešno zaključil. Uporabniški program nikoli ne upravlja sam s strojno opremo, vse poteka preko operacijskega sistema. S tem lahko jedro zagotavlja pravično razdelitev strojne opreme.

Danes najbolj uporabljeni operacijski sistemi so Microsoft Windows, Linux, Android in MacOS. Razvoj takšnega sistema je dolgotrajen in zelo zapleten proces [1]. Kljub temu se, predvsem na univerzah, nadaljuje razvoj tudi drugačnih, specializiranih operacijskih sistemov, ki bodo morda v nekem trenutku nadomestili te, ki so trenutno najbolj razširjeni.

Silberschatz [27] podaja primerjavo: “Operacijski sistem je v računalniku

podoben vladi v neki državi. Njegova naloga je vodenje, sam pa ne ustvarja nobene prave dobrine, ki bi jo lahko uporabnik neposredno uporabil".

V nadaljevanju primerjavo razširi na dve osnovni nalogi OS:

- Operacijski sistem (OS) mora zagotoviti programsko okolje, v katerem sta izdelava ter zagon uporabniških programov, čim lažja.
  OS nudi podlago in določa način izdelave uporabniških programov. Podaja vmesnik do strojne opreme, ki mora delovati in biti enostaven za uporabo. Je okolje, v katerem se izvajajo uporabniški programi. Dodeliti jim mora ustrezni del procesorskega časa, ustrezni del pomnilnika in dostop do vhodno / izhodnih (V/I) enot. Vse to mora opravljati brezhibno.

- Operacijski sistem mora zagotoviti učinkovito izrabo strojne opreme. Strojna oprema je velikokrat draga in fizično omejena. Jedro jo mora čimbolj učinkovito izkoristiti.

Podobno naloge jedra določa Irtegov [29]:

- operacijski sistem mora omogočiti nalaganje uporabniških programov v glavni pomnilnik ter njihovo izvajanje,
- znati mora usklajevati rabo pomnilnika,
- učinkovito mora uporabljati in izrabljati vhodno / izhodne enote,
- uporabniku mora omogočiti način preko katerega lahko komunicira (npr: ukazna vrstica, okensko okolje ali kako drugače).

Operacijski sistem komunicira s strojno opremo na eni strani ter uporabniškimi programi na drugi, pri čemer uporabnik zaupa v njegovo varno in pravilno delovanje.

Skozi leta so se operacijski sistemi razvijali od začetnih 4000 vrstic kode pri MS DOS 1.0 [28] do današnjih 70.000.000 vrstic v Windowsu 8 [1]. Z večanjem hitrosti računalnikov so uporabniki postali vedno bolj zahtevni in

operacijski sistemi se jim morajo prilagajati. Postali so bolj prijazni za uporabo, delujejo na več procesorjih hkrati, so 64 bitni in uporabljajo ogromne količine pomnilnika ter najrazličnejše vhodno/izhodne enote. Hkrati z večanjem števila vrstic kode so OS prešliv fazo, kjer jih lahko zaradi njihove velikosti in zahtevnosti obvladuje razmeroma majhno število razvijalcev.

Mikrojedrni operacijski sistemi so prinesli "svež veter" v ta prostor. Njihove funkcije so razdeljene na posamezne sklope (operacijske strežnike), jih povezuje majhno in zelo hitro jedro. Med seboj komunicirajo po točno določenih komunikacijskih kanalih, kar ima za posledico, da so operacijski strežniki med seboj neodvisni. Napaka v enem zato ne zaustavi celotnega sistema. Razvijalec posameznega operacijskega strežnika (npr. strežnika za mrežno komunikacijo) mora podrobno poznatile delovanje svojega dela. S tem lahko več ljudi sodeluje pri razvoju.

2.2 Nastanek operacijskih sistemov

A. S. Tanenbaum in H. Bos delita zgodovino operacijskih sistemov v 6 generacij [2]. Mejnik vsake od teh generacij je večji napredek v tehnologiji ali uporabi računalnikov kot npr. iznajdba tranzistorjev ali uporaba osebnih računalnikov.

2.2.1 Prva generacija: 1945 - 1955

Prvo generacijo računalnikov so okoli leta 1945 ustvarili John von Neuman v Princetonu, Howard Aiken na Howardu, J. Presper Eckert in John Mauchley v Pennsylvaniji ter Konrad Zuse v Nemčiji. Računalniki so bili ogromni stroji, s po 20.000 elektronkami. Programskih jezikov še ni bilo, programer je napisal program v strojnem jeziku.

Okoli leta 1950 so se pojavile kartice, preko katerih je bil vnos programa in vhodnih podatkov v računalnik lažji.

2.2.2 Druga generacija (1955 - 1965)

Druga generacija se je začela z iznajdbo tranzistorjev. Z njimi so računalniki pridobili stabilnost, pojavile so se prve komercialne izvedbe. Razvili so se programski jeziki - najprej zbirnik, nato višji jeziki.

Podjetja so začela zaposlovati profesionalne operaterje, predhodnike današnjih sistemskih inženirjev. Cena takratnih računalnikov je znašala nekaj milijonov dolarjev.

Programer je napisal program v programskem jeziku (npr. fortranu) in ga prinesel operaterju na luknjastih karticah. Ta je program vnesel, pognal in izpis iz tiskalnika prinesel nazaj programerju.

Programi so se zlagali v t. i. pakete, ki jih je računalnik vse naenkrat izvedel. Takšen paketni sistem je nadzoroval omejen in zelo preprost operacijski sistem.

2.2.3 Tretja generacija (1965 - 1980)

IBM je v zgodnjih 60. letih razvil sistem 360. To je bil prvi računalnik, ki je uporabljal integrirana vezja.

Računalniki so dobili prve prave operacijske sisteme (MULTICS).

2.2.4 Četrta generacija (1980 - danes)

Nastanek četrte generacije računalništva je spodbudil razvoj elektronskih čipov. Z njimi je bilo mogoče izdelati računalnike, ki so bili cenovno dostopni navadnim ljudem. S tem se je računalništvo razširilo in privedlo do nastanka družbe, kjer je njihova uporaba že nenadomestljiva.

2.2.5 Peta generacija (1990 - danes)

Peta generacija računalnikov predstavlja področje mobilnih računalnikov in se razvija vzporedno s četrto generacijo. Razlike med njima niso v tehnologiji, ampak v namenu in uporabi.

V peto generacijo spadajo tako prenosni telefoni, ki so bili v uporabi v 90. letih, kot tudi trenutni “prenosni žepni računalniki”, ki poleg večpredstavnosti omogočajo tudi uporabo telefona.

2.3 Tipi operacijskih sistemov

- Monolitni :
  Je najstarejša in najenostavnejša oblika: monolitno jedro je vsebovano v eni datoteki, ki se naloži v glavni pomnilnik. Vsa koda deluje v istem naslovnem področju, vsak del jedra lahko dostopa do katerega-koli dela ali funkcije. To pomeni, da lahko napaka v enem od delov sistema povzroči zaustavitev celotnega sistema. V primeru, da deluje sistem brez napake, je skupno naslovno področje prednost, saj lahko neka funkcija neposredno spreminja katerikoli del pomnilnika (sistem-ske spremenljivke, strukture ...).

  Predstavniki: UNIX, BSD, Linux

  Dobre lastnosti: hitrost
  Slabe lastnosti: težko vzdrževanje in odpravljanje napak

  Tanenbaum imenuje ta tip »the big mess« [30].

- Mikrojedrni :
  Mikrojedrni OS je razdeljen na več enot: operacijske strežnike in mi-krojedro. Vsak od njih opravlja točno določeno opravilo, vsi skupaj pa predstavljajo celotni operacijski sistem. Operacijski strežniki med seboj komunicirajo le s pošiljanjem sporočil točno določenih oblik. Je-dro je osrednja enota, ki skrbi za hiter in varen pretok sporočil. Je izredno majhno, saj je njegova naloga zelo omejena. Vse višje funk-cije (upravljanje s spominom, datotečni sistem, itd.) so zasnovane kot samostojni operacijski strežniki, ki delujejo kot uporabniški programi.

  Predstavniki: Mach, L4, Symbian, QNX, Minix

Dobre lastnosti: bolj zanesljivi in varni
Slabe lastnosti: počasnejša komunikacija med procesi in jedrom

- Hibridni :
  Hibridni sistemi so podobni mikrojedrom, razvijalci so nekaj funkcij, ki pri pravih mikrojedrnih sistemih niso del jedra, zaradi hitrosti (ali licence) premaknili nazaj vanj. Sistem je še vedno razdeljen na posamezne dele,vsi (oz. skoraj vsi) pa delujejo v prioritetnem načinu. S tem so izgubili zanesljivost, kot jo poznajo mikrojedra, pridobili so hitrost, ker ni nepotrebnega preklapljanja med prioritetnem in navadnim načinom delovanja.

  Predstavniki: Windows NT, BeOS, MacOS X

  Dobre lastnosti: prednosti, ki so pri monolitnih in mikrojedrnih sistemov
  Slabe lastnosti: težko vzdrevanje in odpravljanje napak

- Exojedrni :
  Ti sistemi so po zgradbi manjši kot mikrojedrni sistemi. Mikrojedrni in monolitni sistemi podajajo uporabniškim programov standardne načine, s katerimi lahko shranjujejo datoteke, komunicirajo preko mreže, ali opravljajo kakšno drugo delo. Exojedrni sistem ne podaja nobenih standardov uporabniškim programom,temveč skrbi, da je strojna oprema pošteno razdeljena medn je. Vsak program na svoj način uporablja strojno opremo: lahko zapisuje datoteke na disk v BrtFS formatu, v NTFS ali na kakšen čisto svoj način.Lahko si shranjuje stare verzije datotek ali prepisuje uporabljene bloke z ničlami, da zakrije prejšnje vsebine - vsak program si ustvari lastni način dela.

  Predstavniki: Aegis, ExOS, Nemesis.

- Nano / pico jedro
  Je najmanjše jedro v računalništvu. Vsebuje manj funkcij in kod kot mikro ali exojedro: ne vsebuje niti prekinitvenih ali časovnih funkcij.

Večinoma opravlja zelo omejeno delo, največkrat prenos višjega operacijskega sistema na drugo strojno platformo.

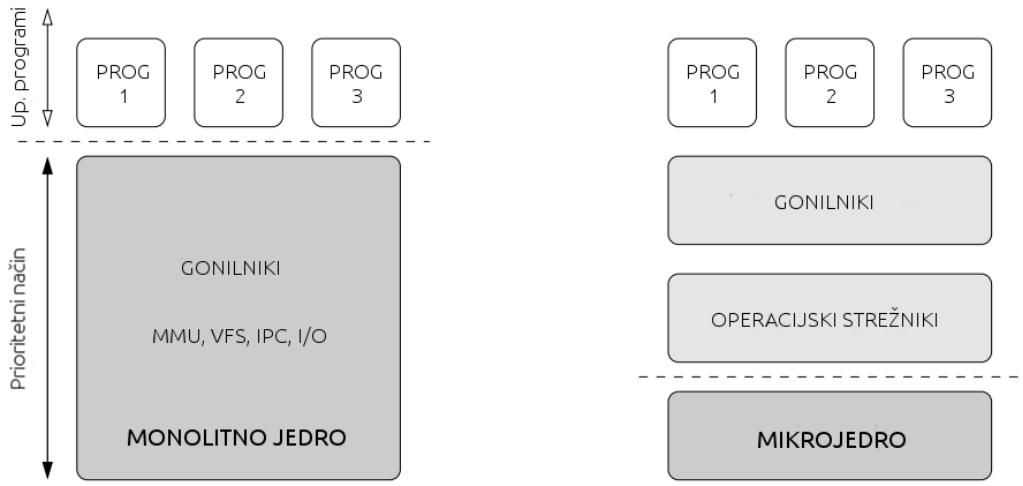
Apple je uporabil nanojedro pri starem MacOS sistemu, da je preuredil prekinitve, generirane na PowerPC računalnikih, v prekinitve, ki jih je razumel njihov 68K emulator [31].

Predstavnik: Adeos [32]

- Virtualni operacijski sistemi
  Virtualni operacijski sistemi ne upravljajo z uporabniškimi programi. Uporabljajo se za multipleksiranje strojnih komponent, tako da lahko na njem sočasno teče več neodvisnih sistemov. Vsak od teh sistemov misli, da ima na razpolago vso strojno opremo, čeprav v resnici ni tako.

  Predstavniki: VMWare [33], Xen [34].

- Megalitni sistemi
  So nasprotje vsem obstoječim sistemom. Ne omogočajo poganjanje uporabniških programov v klasičnem uporabniškem načinu, temveč v prioritetnem načinu. Cel sistem (jedro in programi) delujejo kot skupna celota in so največkrat uporabljeni za reševanje zelo specifičnih problemov. Njihova zgradba je poenostavljena, saj funkcij za upravljanje s programi (kontekstualni preskoki, itd.) ne potrebujejo.



Slika 2.1: Prikaz razlik med monolitnimi in mikrojedrnimi sistemi

Poglavje 3

Mikrojedrni operacijski sistemi

3.1 Osnove

Mikrojedrni sistemi imajo dva temeljna cilja: modularnost in enostavnost.

Oba sta nastala zaradi naraščajoče zahtevnosti pri monolitnih sistemih. Postajali so vse težje obvladljivi, zato se je pojavila ideja o razdelitvi sistema na posamezne module. Le en modul - jedro, naj bi delal v prioritetnem načinu, vsi ostali (vse, kar ni nujno pomembno za delovanje), se pretvori v aplikacijski strežnik oz. modul: modul za datotečni sistem, moduli za gonilnike, moduli za prikaz na ekran, itd.

S tem so odgovornosti bolj jasno določene, saj vsak modul skrbi le za svoje delo. Programska koda je lažje obvladljiva, ker je velikost vsakega posameznega modula majhna. Iskanje napak je zaradi tega lažje, saj je napaka (navadno) omejena le na posamezen modul.

Vsak računalnik ima dva načina delovanja: Prioritetni in uporabniški način. Programom v prioritetnem načinu je dovoljeno spreminjanje registrov, onemogočanje prekinitev, itd., medtem ko programi v uporabniškem načinu nimajo takih pravic. Zato vsa jedra operacijskih sistemov delujejo v prioritetnem načinu, uporabniški programi pa v uporabniškem. Pri monolitnem jedru pomeni, da celotno jedro z vsemi vsebovanimi funkcijami deluje v prioritetnem načinu, četudi ga samo majhen del zares potrebuje. Nasprotno pri

mikrojedru vse funkcije, ki ne potrebujejo prioritetnega načina, delujejo kot operacijski strežniki v uporabniškem načinu. S tem ne morejo vplivati na npr. delovanje prekinitev in napake niso tako uničujoče.

Z razbitjem sistema na majhno jedro in sodelujoče strežnike se poveča modularnost, razširljivost in prenosljivost. Module, ki delujejo po enakem protokolu (npr. datotečni sistemi), se lahko zamenja brez spreminjanja drugih modulov.

3.2 Zakaj - primer uporabe

Spodnji dve poglavji opisujeta sistema, kjer je uporaba mikrojedrnega sistema smiselna in bolje izkorišča strojno opremo. Oba primera se lahko izdela tudi v monolitni verziji, toda takšen korak vključuje več testiranj, večjo pozornost pri odzivih sistema na napake in strogo ločevanje med komponentami.

3.2.1 Srčni spodbujevalnik

Današnji srčni spodbujevalniki vsebujejo poleg osnovnega dela, ki spodbuja delovanje srca, še brezžični (wireless ali bluetooth) modul. Preko njega se lahko v računalnik naložijo nadzorni podatki o delovanju enote ali se preko njega oddaljeno nadzoruje delovanje.

Brezžični modul je v tej napravi drugotnega pomena. Primarna naloga je ohranjanje pravilnega delovanja srca, ki v nobenem primeru ne sme prenehati delovati, da ne ogrozi pacienta. Če se pojavi napaka v delu za brezžično komunikacijo (povezava v napačno omrežje, TCP/SYN napad, neskončna zanka v brezžičnem gonilniku, “buffer overflow“ ali karkoli podobnega), ta ne sme vplivati na spodbujevalnik.

Pri uporabi mikrojedra je brezžična enota ločena od kritičnega dela naprave. Napaka v njej se ne prenese na ostali (kritični del) sistema in ostaja omejena znotraj modula. Modul se zato lahko zaustavi ali ponovno požene, spodbujevalnik pa ves čas nemoteno deluje. Brez mikrojedra se takšen pro-

blem rešuje z več procesorji, ki med seboj ne sodelujejo. Kar pomeni večjo ceno, težo in porabo elektrike.

3.2.2 Avtomobili in letala

Tako avtomobili kot letala vsebujejo sisteme, kjer morajo med seboj sodelovati varnostno bolj in manj pomembne komponente. Med varnostno manj pomembne spadajo uporabniški vmesnik (t. i. "infotainment" sistem) in naprave, ki skrbijo za boljše vzdušje potnikov. Med kritične in varnostno pomembne komponente spadajo kontrola pogona proti spodrsavanju v ovinku, ABS kontrola, urejanje temperature motorja in podobno.

Enako kot pri srčnem spodbujevalniku, tudi tu napaka v varnostno manj pomembni komponenti ne sme ogroziti ostalega dela sistema. Napaka v varnostno bolj pomembni komponenti ne sme spremeniti delovanja druge takšne komponente. Če je le mogoče, mora sistem nedelujočo komponento čim prej obnoviti, pri čemer mora ostali del sistema delovati ves čas nemoteno naprej - napaka pri branju CD plošče ne sme ogroziti delovanja ABS sistema.

DO-178B [89] pri letalih in ISO 26262 [90] sta varnostna standarda, ki določata, kako mora sistem v avtomobilu ali letalu delovati. Z uporabo mikrojedrnega sistema se lahko izdela sistem, ki deluje po teh navodilih. V avtomobilski industriji se tako uporabljata predvsem PikeOS in QNX mikrojedra, v letalski pa Integrity, VxWorks 7 in OSE RTOS.

3.3 Nastanek

Začetki mikrojedrnih operacijskih sistemov segajo v leto 1975, ko so na univerzi v Rochesterju izdelali "Rochester Intelligent Gateway" [3, 47]. Z njim so dokazali, da se lahko izdela modularen računalnik, kjer procesi komunicirajo med seboj in z jedrom s pošiljanjem sporočil. Pošiljanje sporočil je lahko lokalno (dostop do diskov) ali oddaljeno, preko takratnega ARPANET-a. Operacijski sistem v tem računalniku je bil Aleph.

Razvoj Aleph-a je bil po nekaj letih končan, razvojna skupina na čelu

z Richard Rashidom, se je preselila na univerzo Carnegie Mellon, kjer so še naprej razvijali operacijski sistem s podobnimi posebnostmi. Leta 1981 so objavili nastanek novega sistema, imenovanega Accent [5]. Accent so razvijali do leta 1984, ko so zaradi velike priljubljenosti sistema UNIX prišli do zaključka, da potrebujejo popolnoma nov sistem. Nov sistem bi moral biti kompatibilen s sistemom UNIX, da bi lahko uporabljali vso programsko opremo, napisano zanj.

Leta 1984 so na univerzi Carnegie Melon začeli z razvojem novega operacijskega sistema Mach [3]. Leta 1986 je bil razvoj končan in Mach je postal najpopularnejši mikrojedrni operacijski sistem prve generacije.

3.4 Generacije mikrojedrnih sistemov

Mikrojedrni sistemi se delijo v tri generacije.

Prvo generacijo zaznamuje več tehnoloških problemov, ki jih niso uspešno rešili. Največji med njimi je bila počasna komunikacija med procesi. Kljub uspešnemu reševanju ostalih pomanjkljivosti je problem počasne komunikacije ostal in postal temelj, zaradi katerega se je razvoj in uporaba mikrojedrnih sistemov praktično ustavila.

To je trajalo do leta 1993, ko je Jochen Liedtke v svojem novem L3 in L4 sistemu dokazal [15], da je hitra komunikacija med procesi mogoča. Liedtke je začetnik druge generacije mikrojedrnih sistemov.

Tretjo generacijo zaznamujejo napredne rešitve pri problemih varnosti, virtualizaciji in formalni verifikaciji pravilnega delovanja.

3.4.1 Prva generacija

Mach je najbolj znan mikrojedrni operacijski sistem prve generacije. Razvijali so ga na univerzi Carnegie Mellon med leti 1984 in 1994 ter je prvi pravi mikrojedrni operacijski sistem. Razvil se je iz idej, ki sta ju pred časom izpostavila sistema Aleph in Accent. Njegova zadnja verzija je imela oznako Mach 3.0.

V času Machovega razvoja je bil sistem UNIX že zelo razširjen - prisoten je bil že več kot 15 let in posledično zelo kompleksen in velik. Richard Rashid ga je imenoval “odlagališče za vsako novo funkcijo ali značilnost” [8]. Arhitektura Mach-a je posledično odgovor na naraščajočo kompleksnost tedanjega UNIX sistema.

Mach naj bi bil najmanjša osnova, na kateri bi lahko delovali drugi strežniki oziroma operacijski sistemi.Samo jedro je nudilo podporo za upravljanje s pomnilnikom in procesorjem ni pa imelo datotečnega sistema, podpore pri povezovanju v mrežo, upravljanju z vhodno / izhodnimi enotami in z drugimi (višjimi) funkcijami. Bil je napisan v jeziku C, kar naj bi mu pomagalo pri postavljanju na druge arhitekture.

Njegova začetna osnova je bil sistem 4.3BSD. Med razvojem so posamezne dele 4.3BSD-ja odstranili ali zamenjali z novimi implementacijami v skladu z mikrojedrno arhitekturo. Tako so počasi dobivali želene funkcionalnosti, dokler niso leta 1986 objavili njegove prve verzije. Največkrat uporabljena verzija je bila 2.6, ki pa ni bila pravi mikrojedrni sistem. V njem sta Mach in 4.3BSD delovala v istem naslovnem prostoru, kar je definicija monolitnih sistemov.

Verzija 3 je to napako popravila - v njej je bil 4.3BSD pognan kot navaden uporabniški program. Toda (pravilna) mikrojedrna arhitektura je pokazala njene šibkosti. Cena za komunikacijo med procesi je bila prevelika in Mach 3 je imel slabe zmogljivosti glede na takratne monolitne sisteme. Nekateri [9] so razmišljali, da bi morali glavne procese (datotečni sistem, I/O) vgraditi nazaj v samo jedro, drugi [10] so izpostavljali tezo, da se temu problemu ne da izogniti, ker jo tako določa sama mikrojedrna arhitektura.

Kmalu po predstavitvi verzije 3 se je Machov razvoj ustavil. Kljub končanem razvoju je Mach pustil izredno velik pečat v računalniški industriji. Mach sistem je namreč univerza razvijala v odprtokodnem načinu, zato so ga lahko posvojila tudi druga podjetja. Eno takih je bilo podjetje NeXT, ki jo je ustanovil Steve Jobs, ko je leta 1985 moral zapustiti podjetje Apple. Na osnovi sistemov Mach 2.6 in 4.3BSD so izdelali svoj operacijski sistem XNU.

Leta 1997 je Apple kupil podjetje NeXT in sistem XNU. XNU so posodobili na Mach 3 in 4.3BSD zamenjali s FreeBSD sistemom. XNU, FreeBSD in gonilniki tvorijo Applov sistem Darwin, ki še danes poganja vse MacOS in iOS sisteme [11].

Mach sestavljajo štiri glavne abstraktne ideje [6]:

- Proces (“Task”) predstavlja okolje, v katerem se lahko poganjajo “niti”. Je osnovna enota, kamor se dodeljuje vire - ko se dodeli določen del spomina, se ga dodeli procesu, ne posamezni niti. Vsebuje svoj naslovni prostor v spominu ter ima dostop do sistemskih virov. Za primerjavo velja, da je proces, kot ga pozna sistem UNIX enak enemu procesu z eno nitjo v Mach-u.
- Nit (“Thread”) je enota, ki se lahko naenkrat izvaja na enem CPU procesorju. Vse niti, ki so znotraj enega procesa imajo dostop do vseh sistemskih virov, ki jih lahko uporablja ta proces.
- Komunikacijski kanal (“Port”) je način komuniciranja med procesi in tudi med procesi in jedrom. Osnovna ukaza nad komunikacijskim kanalom sta “Send” in “Receive”.
- Sporočilo (“Message”) je enota podatkov, ki se uporabi pri komunikaciji med nitmi. Sporočila so lahko poljubno velika.

Vpliv, ki ga je imel sistem Mach v računalništvu, je velik. R. Rashid, eden od glavnih razvijalcev sistema Mach, je nadaljeval delo v Microsoftu, kjer je postal vodja razvoja. Avie Tevanian, ki je prav tako sodeloval pri razvoju sistema Mach, je postal vodja programske opreme pri Applu.

V prvo generacijo mikrojedrnih sistemov spadajo še:

- ChorusOS [12], razvit v Franciji leta 1979. Podjetje Chorus Systemes je prevzelo komercialen razvoj, leta 1997 jih je prevzel Sun Microsystems. Julija 2007 je bila izdana zadnja verzija 5.1, od takrat se ne razvija več.

- Amoeba [13], ki ga je razvil Andrew Tanenbaum s sodelavci iz Nizozemske univerze Vrije. Je distribuiran sistem, ki je omogočil uporabnikom transparentni dostop do vseh računalnikov v skupini. Program se je lahko ločeno izvajal na kateremkoli procesorju kateregakoli računalnika v tej skupini. Od leta 1996 se ne razvija več. Tanenbaum razvija še operacijski sistem Minix (mikrojedro druge generacije).

Ključni problem vseh mikrojedrnih sistemov prve generacije je počasna komunikacija med procesi in jedrom. Ta problem je bil tako velik, da se njihova prava komercialna uporaba ni začela. Sistemi so bili nekonkurenčni proti stabilnim in hitrim monolitnim sistemom.

Vse, dokler ni leta 1995 Jochen Liedtke predstavil svoj L4 sistem.

3.4.2 Druga generacija

L4

Druga generacija se je začela leta 1995, ko je J. Liedtke s svojo skupino izdelal mikrojedro L4 [14]. L4 je vseboval radikalne spremembe, vključno s programiranjem v zbirniku. Počasno komunikacijo med procesi so rešili s sinhronimi sporočili, vse komponente, ki jih je Mach zaradi večje hitrosti prenesel v jedro, so odstranili. Zmanjšali so število sistemskih klicev na 7 (Mach jih je imel 140) in oklestili velikost kode jedra na samo 12KB (Mach 330KB), zaradi tega je lahko bilo v celoti shranjeno v procesorjevem prvem predpomnilniku. Rezultat je bil več kot 20x pohitritev izvajanja med-procesorske komunikacije [15].

Tako velika pohitritev delovanja je zelo redka pri operacijskih sistemih in je omogočila nadaljnji razvoj mikrojeder na univerzah in raziskovalnih inštitutih. Pojavilo se je več ABI združljivih L4 variacij. Zato kmalu L4 ni več bil le en sistem, temveč cela družina. Razlikovali so se v načinu implementacije in vrsti arhitekture, kjer so delovali. Vsem so ostale skupne originalne usmeritve: minimalizem in hitrost.

Leta 1999 je Liedtke na univerzi Karlsruhe predstavil L4Ka::Hazelnut, ki je bil prvi L4 operacijski sistem, napisan v višjem programskem jeziku C++. Razlike v zmogljivostih glede na verzijo, napisano v zbirniku so bile manjše, kot je bila pridobitev lažjega programiranja in lažjega prenosa na druge arhitekture. Verzijo v zbirniku so zato prenehali razvijati.

Na univerzi v Dresdnu so leta 1998 začeli z razvojem svoje verzije L4 mikrojedra, imenovane L4/Fiasco (ime je dobil zaradi problemov z intelektualno lastnino glede jedra L4 [16]). Napisan je v jeziku C++, njegov razvoj je še vedno aktiven. Je osnova za DROPS - sistem v realnem času, ki je bil prav tako razvit na tej univerzi.

Naslednja verzija L4 sistema je bila L4::Pistachio. Njen cilj je bil boljša prenosljivost in podpora večprocesorskim sistemom. Leta 2001 je Liedtke umrl, njegovi študenti so delo zaključili in objavili tudi ta sistem.

Na univerzi New South Walles so preučevali L4::Pistachio na različnih 64 bitnih platformah in objavili L4/MIPS ter L4/Alpha verzije. NICTA (avstralski nacionalni raziskovalni center) je razvil L4 verzijo za vgrajene sisteme (embedded systems) imenovano NICTA::Pistachio-embedded. Ta verzija L4 sistema je vgrajena v Qualcommove procesorje za brezžične modeme. Poleg brezžičnih modemov imajo L4 procesorje tudi v avtomobilski, letalski, medicinski in vojaški industriji, velikokrat v delu, ki zahteva sistem v realnem času. Applov procesor A7 poganja v svojem ko-procesorju sistem L4 in je tako prisoten v vseh Apple telefonih [17].

Leta 2006 se je začel komercialni razvoj L4 sistema. Podjetje Open Kernel Labs je začelo nuditi podporo in razvoj sistema L4 v komercialne namene, prav tako podjetje Sysgo, ki je ponujalo njihov lastni L4 sistem imenovan PikeOS.

QNX

Drugi zelo popularen sistem, ki ni povezan z družino L4, je QNX s svojim lastnim QNX Neutrino mikrojedrom. To je komercialni in zaprt sistem, uporabljen predvsem v vgrajenih sistemih v avtomobilski in mobilni industriji.

Od leta 2010 je podjetje QNX last podjetja Blackberry, ki je to mikrojedro uporabilo v svojih verzijah telefonov Blackberry 10 [19].

RedoxOS

Je najnovejše mikrojedro,katerega razvoj poteka na internetu od leta 2015 [75]. Glavni programski jezik RedoxOS-ja je jezik Rust, arhitekturno je sistem najbolj podoben MINIX-u [76]. Vsebuje 16.000 vrstic kode,vse višje aplikacije delujejo v uporabniškem načinu kot svoji strežniki.

RedoxOS je delujoč sistem, nima pa še večjega uporabnika.

Podobno kot v sistemu UNIX, kjer lahko uporabnik vse vire naslavlja kot datoteke (“Everything is a file“), RedoxOS to generalizira v naslavljanje po URL-ju: “Everything is a URL“. Spodnja koda prikazuje, kako se naslovi TCP naslov kot URL.

```
1  |
2  |   use std::fs::OpenOptions;
3  |   use std::io::prelude::*;
4  |
5  |
6  |   fn main() {
7  |   //  Let's read from a TCP stream
8  |   let tcp = OpenOptions::new()
9  |   .read(true) // readable
10 |   .write(true) // writable
11 |   .open("tcp:0.0.0.0");
12 |   }
13 |
14 |
```

Koda 3.1: Primer naslavljanja TCP naslova po URL-ju [77]

MINIX3

MINIX3 je zelo popularen sistem druge generacije mikrojeder. Njegov razvoj se odvija na univerzi Vrije v Amsterdamu, glavni razvijalec je bil do nedavno profesor A. Tanenbaum. MINIX3 je podrobno opisan v poglavju 4.

HelenOS

Je sistem, ki je enako kot MINIX, sestavljen iz več aplikacijskih strežnikov [91]. Jedro se imenuje SPARTAN, vsebuje podporo za večopravilnost, navidezni pomnilnik, IPC komunikacijo in večprocesorske sisteme.

Sistem v času pisanja naloge še ni bil pripravljen za praktično uporabo, temveč le kot pripomoček pri učenju in razvoju delovanja mikrojeder (na primer na Fakulteti za Matematiko in Fiziko v Pragi).

F9

Je experimantalno mikrojedro, uporabljeno predvsem za vgradne sisteme. Razvoj F9 se je leta 2015 zaustavil. V času razvoja se je zgledoval po jedru L4.

3.4.3 Tretja generacija

Tretja generacija ima velik poudarek na varnosti: varnost znotraj jedra in varnost oz. pravilnost izvajanja. Varnost znotraj jedra je določena z delovanjem na podlagi zmogljivosti (capabilities), pravilnost delovnja pa z matematičnim dokazovanjem.

Univerza New South Wales je razvila nov L4 sistem, imenovan seL4, ki ima vgrajeno podporo zmogljivostim. Z matematičnimi funkcijami so dokazali, da se jedro obnaša v vseh primerih tako, kot je določeno [16]. Dokazali so, ne samo pravilno delovanje (delovanje brez napak), temveč tudi pravilno delovanje prevedenega jedra v binarni obliki.

Vsak sistem je lahko stabilen in varen toliko, kot je stabilen in varen njegov operacijski sistem. Operacijski sistem deluje v prioritetnem načinu in vsaka napaka v njem lahko povzroči, da cel sistem ne bo deloval pravilno. Statistično naj bi bilo na vsakih 1000 vrstic kode od 15 do 50 napak [20]. Kar pomeni, da je lahko v Linux jedru s 17 milijoni vrstic kode, po najmanjši oceni, 250.000 napak. Če je jedro zelo majhno, je s tem seveda majhno tudi število napak. SeL4 jedro ima 8700 vrstic C kode in 600 vrstic kode v

zbirniku. Pri tako majhnem jedru je možno varnost in pravilnost delovanja dokazati tudi s formalno matematično verifikacijo. S tem se jamči, da je program brez napak, da se ne bo nikoli sesul in ne bo izvedel nevarne operacije - preveri se namreč delovanje jedra v vseh različnih stanjih.

Po [21] je sistem seL4:

- pripravljen za vsakodnevno uporabo in po zmogljivosti, primerljiv z najboljšimi mikrojedri, ki so trenutno v uporabi,
- njegovo delovanje je formalno točno določeno,
- z njegovo formalno določeno zasnovo so dokazane tudi vse lastnosti, na primer odpovedi in pravilno ter varno izvajanje,
- zasnova je uradno dokazana, da izpolnjuje specifikacije,
- nadzor dostopa je formalno dokazan in podaja trdno varnostno garancijo.

SeL4 je prvi operacijski sistem za splošno rabo, ki je formalno dokazan, da pravilno deluje. Verifikacija je bila opravljena za arhitekturo ARMv6, za x86 je še v teku. Tehnika formalnega dokazovanja je bila narejena s programom Isabele/HOL [22].

Kako dolgotrajen in zapleten je potek preverjanja pravilnega delovanja, pove podatek, da je bilo potrebno za jedro, ki ima manj kot 10.000 vrstic kode, za verifikacijo napisati 200.000 vrstic [21]. V [21] še dodajo podatek, da je čas za izdelavo seL4 jedra ocenjen na 4 človek-leta, čas za verifikacijo pa na 20 človek-let.

Med postopkom verifikacije so našli in odpravili 160 napak v C kodi, 150 napak v zasnovijedra in 150 napak v specifikaciji. Za primerjavo so podali podatek, da so med standardnim testiranjem našli in popravili le 16 napak - tako pomembna je lahko verifikacija [25].

Drugi operacijski sistemi tretje generacije so še:

- EROS
  Sistem EROS se je razvil na podlagi specifikacij sistema KeyKOS. Je sistem s podporo zmogljivostim, periodično si shranjuje posnetke sistema (system snapshot) na disk, da lahko ob restartu obnovi stanje. EROS je priporočal, da so programerji izdelali aplikacije kot majhne aplikacijske strežnike, ki jih je bilo lažje programirati, testirati in preveriti, ali ima pravilno nastavljene zmogljivosti [62]. Leta 2005 se je razvoj sistema EROS ustavil in nadaljeval na sistemu Coyotos.

- Coyotos
  Tako kot EROS in nekateri sistemi družine L4, ima dostope do sistemskih virov rešene z zmogljivostmi. Bil je v poteku izdelave formalne verifikacije, a ga je prehitel se L4. Jonathan Shapiro, glavni programer sistema Coyotos (in pred tem tudi sistema EROS), se je leta 2010 zaposlil pri podjetju Microsoft, zato se je razvoj na mikrojedru zaustavil.

- Nova
  je razvojni projekt univerze v Dresdnu [24], ki poskuša narediti varen in učinkovit sistem za virtualizacijo drugih sistemov (hypervisor). Vsebuje mikrojedro, ki so ga poimenovali microhypervisor. Za uporabo sistemskih virov uporablja zmogljivosti (enako kot Coyotos in L4), vsebuje še virtual machine monitor, ki deluje kot navaden uporabniški program nad mikrojedrom. Je konkurenčni produkt trenutnim virtualizacijskim rešitvam, kot so XEN, KVM in drugi.

  Razlika med hypervisorjem in mikrojedrom je ta, da je hypervisor narejen za poganjanje drugih celotnih operacijskih sistemov (Linux, Windows), medtem ko je mikrojedro narejeno za ustvarjanje enega poljubnega sistema, sestavljenega iz več manjših delov. Mikrojedro je lahko hypervisor, obratno pa navadno ne.

3.5 Zmogljivosti

Mikrojedrni sistemi so počasnejši od monolitnih, ker mora za izvršitev določene naloge sodelovati več procesov hkrati. Pri tem pride do dodatne komunikacije med sodelujočimi procesi, saj lahko komunicirajo le preko vnaprej določenih komunikacijskih kanalov. Pri monolitnih sistemih lahko katerakoli funkcija dostopa neposredno do katerekolisistemske strukture in dodatna komunikacija ni potrebna. Pri vsakem pošiljanju sporočil med sodelujočimi procesi v mikrojedrnem OS pride do kontekstnega preskoka, ki je že sam po sebi počasen.

3.6 Prednosti

3.6.1 Varnost

V računalništvu je varnost relativna. Dobro zgrajeni monolitni sistem je lahko varnejši kot slab mikrojedrni. Toda arhitekturni principi, po katerih so mikrojedrni sistemi zgrajeni, jim vsaj po definiciji omogočajo večjo varnost.

Liedtke je določil princip minimalnosti, zaradi katerega je jedro zmanjšano na zelo majhno količino kode. V prioritetnem načinu, kjer so varnostni problemi najvažnejši, deluje samo jedro. V primerjavi z monolitnimi sistemi, kjer v prioritetnem načinu deluje celotno jedro, skupaj z gonilniki, datotečnimi sistemi in ostalim, je problematične kode zelo malo.

Linux jedro ima 7-krat več napak v gonilnikih kot v ostali kodi jedra [36]. Podobno je tudi pri Windows XP sistemu: 85 % neželenih prekinitev delovanja je zaradi napak v gonilnikih [35]. Mikrojedrni sistemi so veliko manj občutljivi na napake v gonilnikih, saj v primeru napake, koda nima dostopa do spremenljivk in struktur jedra. Napaka prekine delovanje gonilnika, vse ostale komponente sistema, (če so pravilno narejene) lahko delujejo naprej.

3.6.2 Enostavnost

Količina kode, ki sestavlja mikrojedrni operacijski sistem, je enako velika (ali celo večja) kot koda pri podobnem monolitnem sistemu. Razlika je v razporeditvi kode. Pri monolitnih sistemih je vsa koda združena v eno skupno binarno datoteko, medtem ko je pri mikrojedrih razdeljena na operacijske strežnike. Tako obstajajo npr.: mrežni operacijski strežnik, datotečni operacijski strežnik, itd.

Vsak operacijski strežnik je svoja logična enota, ima točno določene vhodne in izhodne parametre. Spominske strukture in interno delovanje so lastni le njej. Zaradi tega je strežnike lažje spreminjati, njihovo delovanje se lahko hitreje razume, saj je interakcija z drugimi točno določena. Spremembe, narejene znotraj enega operacijskega strežnika ne vplivajo na druge, če se zagotovi, da ostane princip komuniciranja enak. Vsak del ima sebi lastne teste za preverjanje delovanja. Ob spremembah se izvedejo le njegovi testi, zamudno testiranje vseh ostalih delov, v tem primeru ni potrebno.

3.6.3 Razširljivost in prilagodljivost

Razdelitev na manjše logične enote doda mikrojedrnemu sistemu tudi dobro razširljivost. Sistemu se lahko dodajajo novi operacijski strežniki ali pa obstoječe zamenja z novimi in boljšimi. Za arhitekture s slabšimi viri in zmogljivostmi (ARM) se mikrojedrni sistem sestaviz manjšim številom strežnikov. Uporabi se le tiste, ki so nujno potrebni za delovanje.

Optimizacija sistema je lažja, ker se lahko prenovi vsako logično enoto posebej, brez spreminjanja ostalih delov.

3.6.4 Nadgradljivost

Komunikacija med operacijskimi strežniki v mikrojedrnem sistemu poteka s pošiljanjem sporočil. Sporočilo se vedno pošlje jedru, jedro ga preusmeri želenemu naročniku. Pošiljatelju je vseeno, ali je prejemnik na istem ali na oddaljenem računalniku. Posledica tega je odlično izhodišče za nadgradnjo

v porazdeljen sistem, kjer se izboljšanje zmogljivosti doseže že samo z dodajanjem novih računalnikov.

3.7 Tipi mikrojedrnih aplikacij

Mikrojedro podaja malo sistemskih klicev, ki po večini niso primerni za uporabo v uporabniških aplikacijah. Sistemske klice, ki jih omogočajo monolitni sistemi in so uporabljeni v uporabniških programih, so v mikrojedrnih sistemih zagotovljeni z aplikacijskimi strežniki.

Obstajajo različni tipi mikrojedrnih modulov oz. aplikacijskih strežnikov:

- Eno-strežniški operacijski sistem
  V eno-strežniškem operacijskem sistemu poganja mikrojedro celotni monolitni operacijski sistem (na primer Linux) kot navaden uporabniški program. Monolitni sistem se mora pred tem prilagoditi, da lahko deluje kot proces nad mikrojedrom.Vsi uporabniški programi delujejo nespremenjeni,saj jih še vedno upravlja monolitni sistem. Prednost te postavitve je ta, da se lahko na istem sistemu poganjajo tako stari programi, napisani za monolitne sisteme, kot tudi tiste, ki so narejene za sistem z mikrojedrom.
  Sistem “L4Linux“ [26] je primer takega eno-strežniškega sistema, ki deluje nad jedrom L4. Tudi sistem Mach omogočalizvajanje celotnega monolitnega sistema (v tem primeru BSD UNIX ali OSF/1) [44].

- Več-strežniški operacijski sistem
  Več-strežniškisistem vsebuje več aplikacijskih strežnikovpri katerem vsak opravlja točno določeno funkcijo. Celotni sistem je organiziran v module, s čimer se kompleksnost sistema zmanjša. Komunikacija med moduli se odvija preko sporočil, ki jih jedro prenaša med samimi strežniki.
  MINIX je primer več-strežniškega sistema.

- Namenski sistem Pri namenskih sistemih se modularnega pristopa ne

potrebuje, saj vse delo opravlja le en specializiran program. Ta program potrebuje jedro le za dostop do določenih sistemskih virov. V tem je mikrojedro je dobra izbira, ker je majhno in je zanj lažje dokazati varno delovanje in okolje.

Poglavje 4

MINIX

4.1 Uvod

Leta 1979 je podjetje AT&T kot lastnik intelektualnih pravic nad sistemom Unix, izdalo verzijo V7 skupaj s spremenjeno licenco. Nova licenca je prepovedovala komurkoli pisati knjigo ali poučevati o njihovem V7 produktu. To je bilo takrat presenetljivo, saj je veliko univerz poučevalo delovanje operacijskih sistemov z njihovim Unix sistemom. Od takrat so morali vsi uporabljati simulatorje ali spremeniti področje poučevanja.

Zato se je leta 1984 Andrew Tanenbaum, profesor na Univerzi Vrije v Amsterdamu, odločil izdelati svojo verzijo V7, ki bi bila v pomoč pri njegovih predavanjih. Ime je sestavljenka besed “MIni“ in “uNIX“ = “MINIX“.

Leta 1987 je bil MINIX prvič objavljen in predstavljen, od leta 2000 je prosto dostopen z BSD licenco. Leta 2004 je MINIX dobil sponzorstvo Nizozemske in Amsterdamske akademije, leta 2004 pa še sponzorstvo EU [37], kar mu je omogočilo nadaljnji razvoj do trenutne verzije 3.3.

Verzija 3.3 je definirana kots POSIX-om skladen strežnik, ki je sestavljen iz več delov in tolerira napake ter deluje nad mikrojedrom (“fault-tolerant multiserver POSIX-compliant operating system on top od microkernel“) [37]. MINIX je trenutno osredotočen na vgrajene (embedded) sisteme, saj je tam visoka razpoložljivost običajno nujna.

Od verzije 3.2 naprej je MINIX združljiv z NetBSD sistemom. Večino programov, ki so pripravljeni za NetBSD se lahko brez sprememb poganja v MINIX-u.

4.2 MINIX3 jedro

MINIX-ovo mikrojedro je sestavljeno iz 12.000 vrstic kode in 1400 vrstic zbirnika [38]. Uporabniškim programom podaja 40 sistemskih klicev, sam pa ureja in dodeljuje časovne rezine procesom, se odziva na prekinitve in omogoča med-procesno komunikacijo s pošiljanjem sporočil med njimi.

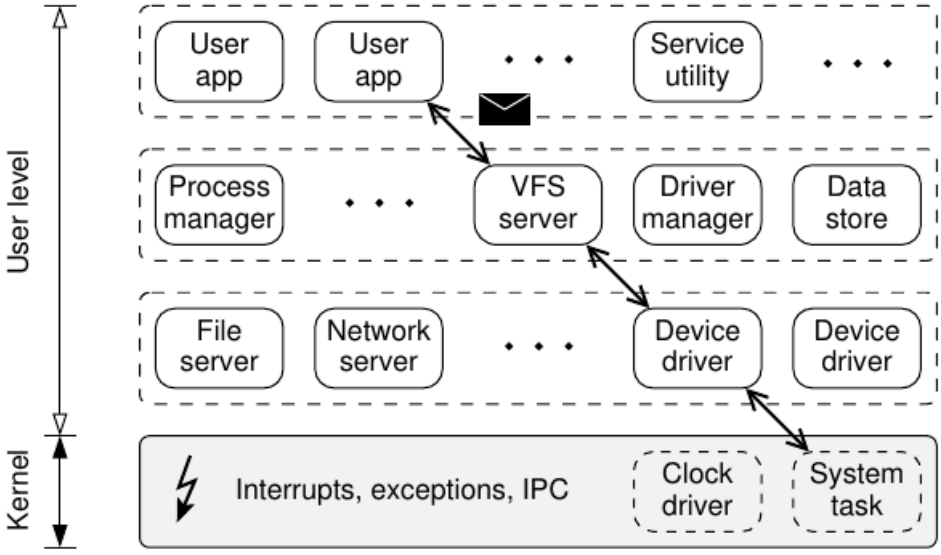
V jedro sta vključena dva procesa, pomembna za podporo uporabniškim programom; gonilnik za čas ter sistemski proces. Gonilnik za čas ("clock driver") je vključen v jedro zaradi boljše zmogljivosti in hitrosti, saj se razporejanja procesov ne da narediti brez tesnega sodelovanja s tem gonilnikom. Sistemski proces omogoča 35 sistemskih klicev, ki jih avtorizirani programi uporabljajo pri svojem delu (delo z vhodno/izhodnimi podatki, kopiranje podatkov v spominu, itd.). [39]

Nad mikrojedrom so programi in aplikacijski strežniki razdeljeni v tri plasti:

- gonilniki
  Gonilniki so v plasti takoj nad mikrojedrom. Vsak gonilnik je svoj proces, ki je (enako kot uporabniški programi) zagnan na procesorju, ko mu to določi jedro ("scheduler"). Ker delujejo gonilniki v navadnem, uporabniškem načinu, ne morejo sami dostopati ali nadzirati vhodno / izhodnih enot. To lahko naredijo le z ustreznim sistemskim klicem jedru, jedro pa preveri, ali ima ustrezne pravice. V monolitnem sistemu takega preverjanja ni, zato lahko nepravilno napisan gonilnik (npr. gonilnik za zvok), ponesreči prepiše datoteke na disku, saj deluje v prioritetnem načinu in ima dostop do vseh funkcij v jedru.
  Primeri gonilnikov: gonilnik za tipkovnico, miško, mrežno kartico, zvok, itd.

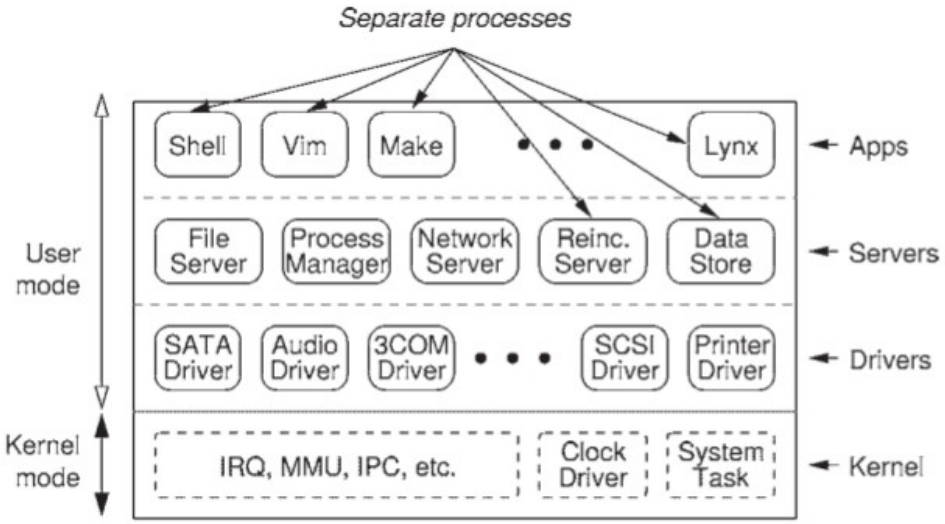
- aplikacijski strežniki
  Nad gonilniki so aplikacijski strežniki. So logične enote, pri katerih vsaka opravlja specializirano opraviloObstajajo strežniki za upravljanje z uporabniškimi procesi, strežniki za prikaz grafičnega vmesnika, strežnik za datotečni sistem, itd.

  Datotečni strežnik je sestavljen iz 4500 vrstic C kode in čaka na POSIX zahteve iz uporabniških programov, da odpre, prebere, zapiše ali zapre datoteke. Ko uporabniški program pošlje zahtevo za branje določene datoteke, sam najprej preveri, ali ima vsebino že v predpomnilniku. Če podatkov še nima v njem, pošlje diskovnemu gonilniku sporočilo z zahtevo za branje te datoteke. Ko gonilnik zapiše vsebino v svoj naslovni prostor datotečni strežnik naroči jedru, naj to vsebino prenese uporabniškemu programu.Prenaša se enaka količina podatkov kot na vseh drugih monolitnih sistemih.Dodatna so le štiri pošiljanja sporočil ter dva kontekstna prehoda (slika 4.1).



Slika 4.1: Delovanje datotečnega strežnika.(Povzeto po [88])

- uporabniški programi
  Uporabniški programi predstavljajo zadnjo plast v sistemu. Ko ti želijo storitev katerega od aplikacijskih strežnikov mu preko jedra pošljejo sporočilo. Ta storitev obdela in uporabniškemu programu pošlje odgovor oz. povratno sporočilo. Ko želi (na primer) uporabniški program zapisati datoteko na disk, to s posebnim sporočilom pove strežniku za datotečni sistem.



Slika 4.2: Plasti v MINIX3 sistemu. (Povzeto po [39])

4.3 Reinkarnacija strežnikov

MINIX sistem vsebuje dva strežnika s posebnimilastnosti: “reinkarnacijski strežnik“ (”reincarnation server”) in “podatkovno skladišče“ (“data store“).

Reinkarnacijski strežnik je nadrejeni (“parent“) proces vsem gonilnikom in strežnikom, zato lahko zazna, ko se kateri izmed njih ne odziva. V tem primeru reinkarnacijski strežnik prebere in izvede proceduro za obnovitev

tega neodzivnega strežnika. Vsak strežnik ima lahko svojo proceduro ponovne vzpostavitve. Navadno to pomeni vnovični zagon, lahko pa je tudi bolj zapleteno: lahko prenese podatke neodzivnega sistema na drugo mesto, lahko zapiše podatek o neodzivnosti v sistemski dnevnik, ali obvesti skrbnika o napaki.

Ker se ponovni zagon zgodi izredno hitro, uporabniški programi tega večinoma niti ne zaznajo. Napaka v mrežnem gonilniku povzroči, da ga reinkarnacijski strežnik ponovno zažene in v podatkovno skladišče zapiše novo konfiguracijo. Podatkovno skladišče obvesti mrežni strežnik o novem gonilniku, ki ga mora le-ta najprej nastaviti. Ker je TCP/IP protokol zanesljiv, mrežni strežnik kasneje ponovno pošlje izgubljene pakete, s čimer omogoči popolno in uporabniku transparentno obnovitev. S tem se sistem samostojno zaceli in omogoča visoko stopnjo razpoložljivosti.

Podatkovno skladišče vsebuje varnostne kopije stanja sistema, vanj reinkarnacijski strežnik shranjuje informacije o sistemski konfiguraciji [39].

Reinkarnacijski strežnik s pomočjo planerja (scheduler) ureja in ponovno zaganja neodzivne gonilnike. Če se kateri odziva počasi in uporablja veliko procesorskih ciklov, je zelo verjetno, da se je ujel v neskončno zanko. Planer mu zato postopoma manjša prioriteto, dokler ne postane proces nedejaven (idle process). Ker reinkarnacijski strežnik preverja odzivnost, bo ta gonilnik (ko postane nedejaven), ponovno zagnal.

4.4 Komunikacija med procesi

Procesi komunicirajo med seboj in z jedrom s pošiljanjem sporočil. Sporočila imajo točno definiran tip in dolžino (64 bajtov [40]), da se lahko izognemo problemom s prekoračitvijo medpomnilnika. So sinhroni, kar pomeni, da prejemnik čaka, dokler mu pošiljatelj sporočila ne dostavi. Ko pošiljatelj pošlje sporočilo, čaka v pripravljenosti, dokler ne dobi odgovora. Ko sta oba (pošiljatelj in prejemnik) pripravljena na prejemanje in oddajanje sporočila, jedro skopira sporočilo iz pošiljateljevega naslovnega področja v prejemnikovo.

Če se komunikacije ne da opraviti s sinhronimi sporočili, se lahko uporabi princip z obvestili. V tem primeru pošiljatelj ne čaka na odgovor, temveč lahko nadaljuje s svojim delom. Ko prejemnik sporočila zaključi dodeljeno nalogo, pošlje jedro pošiljatelju obvestilo, da se je naloga zaključila [39].

Pošiljanje večje količine podatkov je preko kratkih sporočil zamudno, zato ima MINIX pripravljen t. i. "pomnilniške donacije" ("memory grants"). S tem proces dovoli drugemu procesu branje in pisanje v njegovo naslovno področje. Naslovno področje, kamor ima drugi proces dovoljen dostop, je točno definirano in omejeno, dejanski dostop v naslovno področje prvega procesa pa opravi jedro.

4.5 Posodobitve med delovanjem

MINIX3 omogoča posodobitve programskih komponent, medtem ko so te komponente aktivne in v delujočem stanju. Med posodobitvijo se delovanje sistema ne prekine, po končani posodobitvi pa deluje normalno. Sistema ni potrebno zaradi tega ponovno zagnati.

Posodobitve so trenutno namenjene le gonilnikom in aplikacijskim strežnikom (1. in 2. plasti nad jedrom). Posodabljanje jedra in uporabniških programov med delovanjem še ni podprto.

Posodobitve upravlja reinkarnacijski strežnik, ki lahko posodobi tudi samega sebe. Med težje izvedljivimi je posodobitev strežnika za navidezni pomnilnik, saj strežnik upravlja s celotnim pomnilnikom in obdeluje napake strani v njem. Obe funkcionalnosti morata biti v času posodobitve v mirovanju, kar je pri delujočem sistemu težko doseči.

4.5.1 Koraki posodobitve

Vsaka posodobitev poteka v več korakih:

- zaustavitev trenutnega dela
  Strežnik, ki bo posodobljen, je potrebno najprej začasno zaustaviti

ne pomeni, da se delovanje strežnika povsem ustavi, temveč se preneha le prejemanje novih sporočil. Strežnik prestopi v fazo mirovanja, da se sprememba, ki bo nastala, čim bolj omeji.

- zagon nove verzije strežnika
- prenos stanja
  Strežnik, ki je bil zagnan, preveri stanje v starem strežniku. Vse podatke (strukture v spominu, nastavitve) prenese k sebi in jih spremeni v verzijo, kot jo potrebuje sam (naredi transformacijo podatkov v obliko, kot jo potrebuje).
- zaključek
  Če se prenos podatkov zaključi uspešno, se stari strežnik ugasne, novi pa ostane v delujočem stanju. V primeru, da prenos konča z napako, se obdrži stari strežnik.

4.5.2 Sočasne posodobitve

Sistem omogoča posodobitev več strežnikov hkrati, v enem koraku. Posodobitev je uspešna, če jo vsi strežniki zaključijo z uspehom. Razširjena verzija take posodobitve je posodobitev, v katero so vključeni vsi gonilniki in vsi aplikacijski strežniki na sistemu. S tem se celotni MINIX sistem, brez vmesnega zaustavljanja nadgradi na novejšo verzijo.

4.6 Zgradba gonilnikov in strežnikov

Gonilniki in strežniki so zgrajeni na podoben način. Pred začetkom delovanja se morajo registrirati, oboji imajo enako vstopno točko in oboji delujejo v neskončni zanki, v kateri čakajo na vhodna sporočila. Glede na tip in vsebino prejetega sporočila obdelajo svojo nalogo ter pošiljatelju vrnejo odgovor.

Gonilnik ima poleg programske kode še posebno nastavitveno datoteko, v kateri so navedeni sistemski klici in vhodno/izhodna naslovna področja, do

katerih ima dostop. Če bi gonilnik zaradi napake hotel dostopati do drugega dela spomina, bi mu to jedro to onemogočilo.

Gonilnik za ustvarjanje naključnih števil (drivers/system/random) ne uporablja nobenega sistemskega klica in ne naslavlja nobene vhodno / izhodne enote, zato je njegova konfiguracija prazna:

```
service random
{
};
```

Koda 4.1: Nastavitvena datoteka za gonilnik “random“

Gonilnik za tiskanje (drivers/printer/printer) mora imeti možnost komuniciranja s povezanim tiskalnikom. To opravlja s pošiljanjem podatkov na LPT1 in LPT2 vrata. Gonilnik pošlje tiskalniku podatke za tiskanje, tiskalnik pa odgovori z ACK signalom in generira IRQ 7, da lahko gonilnik pošlje nov set. Vhodno izhodni prostor obeh vrat se nahaja v naslovnem področju 378h-37Fh za LPT1 in 278h-27Fh za LPT2. Zato je njegova nastavitvena datoteka temu primerno sestavljena:

```
service printer
{
    io      378:4           # LPT1
            278:4           # LPT2
    ;
    irq
            7               # PRINTER_IRQ
    ;
    system
            KILL            #  6
            UMAP            # 14
            IRQCTL          # 19
            DEVIO           # 21
            VDEVIO          # 23
            READBIOS        # 35
    ;
};
```

Koda 4.2: Nastavitvena datoteka za gonilnik “printer“

Vsak gonilnik ali strežnik se mora pred začetkom delovanja v MINIX sistem najprej registrirati. Pri tem uporabi posebno SEF (System Event Framework) funkcijo sef_startup().

4.6.1 Primer preprostega gonilnika

Primer kode (Koda 4.3) je iz lib/libnetdriver/libnetdriver.c in prikazuje vhodno rutino, ki je skupna vsem mrežnim gonilnikom. V zgornjem delu sef_setcb_init_fresh() nastavi funkcijo do_init(), ki se bo zagnala ob prvem zagonu gonilnika. Nato sef_setcb_signal_handler() nastavi funkcijo got_signal() kot vhodno točko za obdelavo vsakega novega signala.
Takoj za njima se izvede še sef_startup():

```
1  void
2  netdriver_task(const struct netdriver *ndp)
3  {
4    /* ... */
5
6    /* Perform SEF initialization */
7    sef_setcb_init_fresh(do_init);
8    sef_setcb_signal_handler(got_signal);
9
10   netdriver_ndp = ndp;
11
12   sef_startup();
13
14   netdriver_ndp = NULL;
15
16   /* The main message loop. */
17
18   /* ... */
19 }
```

Koda 4.3: Inicializacija gonilnika

Preostali del kode (koda po inicializacijskih funkcijah) opravlja delo lastno gonilniku. Gonilnik za generiranje naključnih števil pripravi naključno število, mrežni gonilnik pa čaka na sporočilo o novih paketih.
Večina jih vsebuje neskončno zanko, saj morajo delovati ves čas. Vsi aplikacijski strežniki vsebujejo neskončno zanko, znotraj katere izvajajo svoje delo. Spodaj je prikazan del kode sistemskega planerja (“scheduler”), ki v taki zanki izbira proces, ki se bo izvajal naslednji.

```
1  /* Main routine of the process manager. */
2  int main()
3  {
4    sef_local_startup();
5
6    /* This is PM's main loop - get work and do it, forever and
       forever. */
7    while (TRUE) {
8
9      /* ... */
10
11     /* Send reply. */
12     if (result != SUSPEND) reply(who_p, result);
13   }
14
15   return(OK);
16 }
```

Koda 4.4: Neskončna zanka v strežniku

4.6.2 Reinkarnacijski strežnik

Reinkarnacijski strežnik (opisan v poglavju 4.3) preverja delovanje strežnikov in gonilnikov, ter jih ob nedelovanju ponovno zažene. Periodično jim pošilja t. i. "keep alive" sporočila, gonilnik ali strežnik pa mora na njih odgovoriti. Spodaj je bolj nadrobno predstavljena koda, povzeta iz njihovega git repozitorija.

Koda 4.5 je del datoteke servers/rs/main.c (skopiran je samo del, ki je zanimiv za preverjanje procesov). Prikazuje neskončno zanko reinkarnacijskega strežnika, v kateri le-ta preverja odzive ostalih strežnikov. Izvajanje preverjanja je periodično, za njeno izvajanje skrbi sistemska časovna ura (timer). Strežnik pošlje drugim delom sistema sporočilo, s katerim zahteva kratko poročilo o delovanju (vrstica 11). Odgovor strežnika se shrani in označi, da je bilo prejeto (vrstica 19). To je pomembno, saj se na podlagi tega podatka strežnike, ki niso podali odgovora, ponovno zažene.

```
1  /* Glavna zanka v Reinkarnacijskem strežniku */
2  while(TRUE) {
3    if (is_ipc_notify(ipc_status)) {
4      switch(who_p) {
5      case CLOCK:
6        /**
7         * Timer povzroči, da se reinkarnacijski strežnik
8         * zažene periodično in preveri delovanje
9         * strežnikov v do_period() funkciji.
10        */
11       do_period(&m);
12       continue;
13     default:
14       /**
15        * Če sporočilo ne pošlje timerja sporočilo
16        * pošlje strežnik, ki sistemu javi, da deluje v
17        * redu.
18        */
19       rproc_ptr[who_p]->r_alive_tm = m.m_notify.timestamp;
20     }
21   }
22 }
```

Koda 4.5: Preverjanje odzivnosti strežnikov - main.c

Koda 4.6 je iz servers/rs/request.c. Prikazuje del, ki nedelujoči strežnik ugasne in ponovno zažene.

Koda v zanki pregleda vse registrirane strežnike. V vrstici 34 se strežnik ugasne, če prej ni odgovoril na sporočilo o delovanju. Če je strežnik podal informacijo o delovanju, potem reinkarnacijski strežnik sklepa, da strežnik deluje pravilno in mu v naslednjem časovnem intervalu pošlje enako sporočilo (vrstica 44).

```
/* do_period() funkcija */
void do_period(m_ptr)
{
  /* Zanka pregleda vse strežnike v sistemu */
  for(rp=BEG_RPROC_ADDR;rp<END_RPROC_ADDR;rp++) {

    /* Preveri, če je strežnik označen da je v delovanju */
    if((rp->r_flags&RS_ACTIVE) && (!SRV_IS_UPDATING(rp)) {

    /* Tu se strežnik, ki je tako označen, ponovno zažene */
    if(rp->r_backoff==0) {
      restart_service(rp);
    }

    /*
     * Če se strežnik ki odzvanja na signal SIGTERM, se ga
     * ubije na črtno so crash_service()
     */
    else if(rp->r_stop_tm > 0 && now - rp->r_stop_tm > 2*
    RS_DELTA_T) {
      crash_service(rp);
    }

    /*
     * Preverjanje odziva strežnika
     */
    else if(period > 0) {

      if(rp->r_alive_tm < rp->r_check_tm) {
        if(now - rp->r_alive_tm > 2*period && rp->r_pid > 0)
    {
          /*
           * Če se strežnik ki javil v predvidenem času,
           * sega načrtnu uboije.
           */
          crash_service(rp);
        }
      }
    }

    /*
     * Če je strežnik ke odgovoru mu Reinkarnacijski
    streznik
     * ponovno pošlje povpraševanje o njegovem statusu.
     */
    else if(now - rp->r_check_tm > rp->r_period) {
      ipc_notify(rpub->endpoint);
      rp->r_check_tm = now;
    }
  }
}
```

Koda 4.6: Preverjanje odzivnosti strežnikov - request.c

Poglavje 5

L4

Z razvojem prve generacije mikrojedernih OS (predvsem Mach-a), s primerjavami med njimi in monolitnimi sistemi, je bilo dokazano, da mikrojedrni OS delujejo znatno počasneje. Začetno navdušenje je hitro upadlo, obveljalo je mnenje, da dovolj hitrega mikrojedra ni mogoče narediti [41].

Profesor Jochen Liedtke je, s svojim, v zbirniku napisanem mikrojedrom L3, dokazal nasprotno. Na podlagi napak pri prvi generaciji mikrojedrnih OS je odkril, da je bilo jedro Mach-a narejeno presplošno. Napisal je popolnoma novo jedro, v katerem je pustil le najnujnejše štiri funkcije (address space, threads, UID, IPC). Programiral ga je v zbirniku in izkoristil vsa pospeševanja ter prednosti izbrane strojne arhitekture. Jedro je postalo neprenosljivo, toda izredno hitro.

L4 je nadaljevanje tega razvoja: ima še manj vrstic kode in je lažje prenosljivo kot L3. Za opravila ali funkcije, ki jih opravlja L4, velja: “Lastnost oz. funkcija je lahko v jedru samo, če je za njeno delovanje potreben prednostni način“ (“A feature should be in the microkernel if and only if security requires that the feature be implemented in privileged mode“) [42].

Študenti na univerzi Karlsruhe so prepisali L4 v C++, s čimer so dokazali, da sistem deluje malo počasneje, če je napisan v višjem programskem jeziku. S prehodom na programski jezik C++, se pospeši razvoj in izboljša razumevanje delovanja sistema. Ves razvoj jedra L4 se danes izvaja le na

C++ verziji.

5.1 Kratka zgodovina

L3 [43] je bilo mikrojedro, iz katerega se je razvilo trenutno najbolj razširjeno jedro L4. L3 je nemški profesor Jochen Liedtke začel razvijati leta 1984.

L4 je bil, kot naslednik jedra L3, izdelan leta 1995 na i486 in Pentium računalnikih. Vsa koda je bila napisana v zbirniku, da so bile uporabljene vse prednosti strojne arhitekture. Skupaj z jedrom je Liedtke postavil še specifikacijo delovanja, t.i. V2 specifikacijo.

Od leta 1996 do leta 1998 sta univerzi v New South Wales-u in Dresden-u prenesli L4 mikrojedro na arhitekturi MIPS in Alpha. Obe verziji sta delovali po V2 specifikaciji.

Od leta 1997 do 1999 je Liedtke delal v IBM-jevem laboratoriju, kjer je izdelal novo eksperimentalno verzijo API sistemskih klicev za L4 imenovala se je X.0. Še vedno je programiral v zbirniku. Medtem je Michael Hohmuth prvi predelal L4 v jezik C++. Nastal je L4::Fiasco.

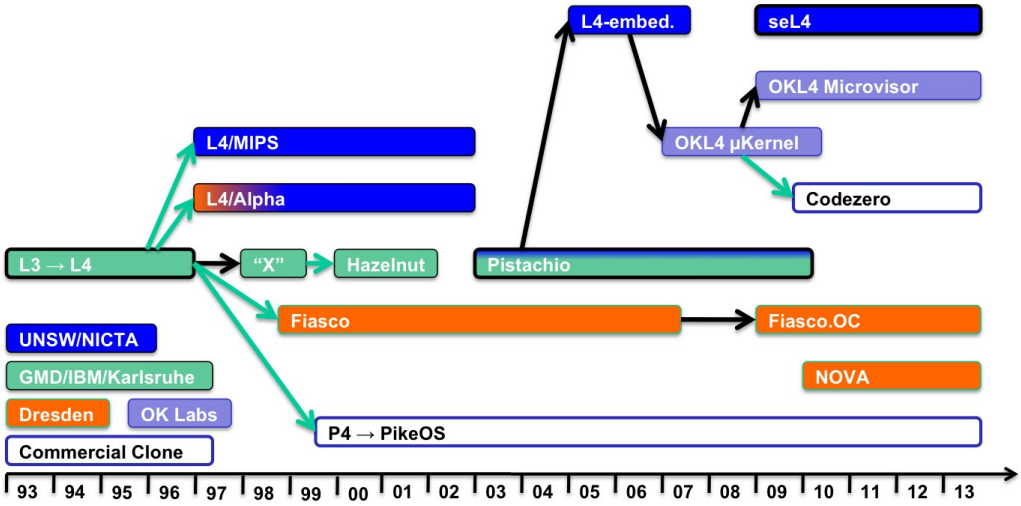
Leta 1999 je Liedke odšel iz IBM-ja na univerzo Karlsruhe, kjer je s svojo skupino prepisal L4 v jezik C. Nastal je "L4::Hazelnut" Delal je tako na i386 kot tudi na ARM procesorjih.

Po Liedkejevi smrti leta 2001, so njegovo delo nadaljevali na univerzi Karlsruhe in leta 2002 objavili novo specifikacijo sistemskih klicev, imenovali so jo X.2. Specifikacija je zagotavljala večjo prenosljivost jedra na različne vrste strojnih arhitektur. Novo jedro se imenuje "L4::Pistachio" in deluje na i386, Itanium, MIPS, Alpha in ARM procesorjih. Že leta 2003 je Pistachio deloval enako hitro kot V2 jedro narejeno v zbirniku. Dokazali so, da je jedro lahko hkrati hitro in prenosljivo.

Avstralski center NICTA je na podlagi L4::Pistachio jedra ustvaril jedro L4::Pistachio-embedded, pripravljeno za vgrajene sisteme. Povezano podjetje OKL4 je na podlagi tega izdelala verzijo L4 jedra, imenovano seL4.

seL4 in L4::Fiasco (sedaj preimenovan v "Fiasco.OC") sta dva najaktiv-

nejša projekta tretje generacije L4 jeder.



Slika 5.1: Družina L4 mikrojeder

Liedtkejev “princip minimalizma“ in vedno boljša optimizacija pošiljanja sporočil med aplikacijami sta prisotna ves čas razvoja vseh L4 sistemov. Pošiljanje sporočil (in pripadajoči kontekstni preskok) je programiran tako, da ga ni mogoče izvesti v manj procesorskih ciklih.

Tabela 5.1 prikazuje število ciklov za en prenos sporočila. Razlike v številu ciklov nakazujejo, kako je določena strojna arhitektura “prijazna“ pri kontaktualnih preskokih.

5.2 Abstrakcije

Jedro tretje generacije L4 sistemov določa naslednje abstrakcije:

- Niti (threads):
  Nit predstavlja programsko kodo, ki se izvaja na procesorju. TCB (Thread Control Block) je skupek podatkov, ki so lastni posamezni niti. Poleg TCB bloka, hrani jedro za vsako nit še EIP (Instruction pointer), ESP (Stack pointer) ter vrednost CPU registrov. Niti tečejo

Ime	Leto	Procesor	MHz	Št. ciklov	μs
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	21064	433	45	0.10
Hazelnut	2002	Pentium 4	1,400	2.000	1.38
Pistachio	2005	Itanium 2	1,500	36	0.02
OKL4	2007	XScale 255	400	151	0.64
NOVA	2010	Core i7 (Bloomfield) 32-bit	2,660	288	0.11
seL4	2013	Core i7 4770 (Haswell) 32-bit	3,400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex A9	1.000	316	0.32

Tabela 5.1: Število ciklov za prenos enega sporočila [16]

v navadnem uporabniškem načinu in lahko preidejo v jedro s sistemskim klicem “sysenter“ (INT 0x30).

Vsaka nit spada v eno izmed 256 prednostnih stopenj, na podlagi katere se določa,katera se bo po Round-Robin protokolu izvajala naslednja. Nit teče toliko časa, dokler se ne izteče njena časovna rezina ali dokler ne postane blokirana. Prav tako nit z višjo prioriteto zaustavi nit z nižjo prioriteto. Ko nit ustvari novo, ji lahko dodeli prioriteto, ki je manjša ali enaka prioriteti, ki jo ima sama.

Poleg kontrolnih blokov ima vsaka nit svojo vstopno točko za primer izjem (“Exception handler endpoint“). Če nit izvede akcijo, ki ni veljavna, jedro pošlje IPC sporočilo z opisom izjeme na to vstopno točko.

- Naslovni prostor (Address space):
  Sistem L4 sestavljajo niti, ki se združujejo v procese.Vsak proces ima svoj prostor naslavljanja pomnilnika.

  Fiasco.OC sistem uporablja starejši način dodeljevanja naslovnega pro-

stora. Pri njem je proces “sigma“, po zagonu računalnika, lastnik vsega naslovnega področja pomnilnika. Sigma proces se zažene ob napaki strani, ko želi kateri drug proces dostopati do prostega pomnilnika. Prosti del naslovnega področja dodeli procesu, le-ta ga lahko rekurzivno dodeljuje naprej ostalim procesom, ki jih ustvari sam. Operacije, pomembne za urejanje naslovnega področja so:

– preslikovanje (mapping):
nit lahko preslika nekaj svojih pomnilniških okvirjev v naslovno območje druge niti. Od takrat lahko do okvirjev dostopata obe niti hkrati. Lastnik (preslikovalec) ohrani vse pravice nad preslikanim okvirjem in jih lahko kadarkoli prekliče.

– dovoljevanje (granting):
nit lahko svoje pomnilniške okvirje odda v uporabo drugim procesom. Za razliko od preslikovanja se pri dovoljevanju okvirji odstranijo iz naslovnega prostora osnovne niti in se prištejejo k procesu, ki sprejema okvirje. Proces, ki sprejme del naslovnega območja, postane njihov lastnik z vsemi pravicami. Prvotni lastnik lahko dovoljevanje kadarkoli prekliče.

– od-preslikovanje (unmapping):
je operacija nasprotna preslikovanju. Nit, ki je dala svoj del pomnilnika v skupno uporabo, lahko to kadarkoli prekliče. S tem se povezanim nitim zmanjša naslovno področje za število okvirjev, ki so jih imele preslikane.

se L4 uporablja drugačen sistem dodeljevanja naslovnega področja procesom. Naslovno področje je sestavljeno iz okvirjev, ki so identični fizičnim okvirjem spomina. Preslikava okvirja v naslovno področje procesa je mogoča le, če ima proces ustrezne pravice v tabeli zmogljivosti.

- Medprocesna komunikacija (Inter-process communication):
Lidtkejev L4 sistem je omogočal le sinhrono pošiljanje sporočil. Ta tip

pošiljanja je najbolj preprost in najbolj sledi minimalističnemu principu, ki ga je Liedtke zagovarjal. Zaradi sinhrone narave, sporočila ni potrebno shranjevati v medpomnilniku jedra, niti ga ni potrebno kopirati (lahko se samo prenese k naslovniku).

Zaradi kasnejših problemov z zapletenimi procedurami zagotavljanja sinhronih operacij, so v seL4 in Fiasco.OC jedrih dodali še asinhrona. Razvijalci predvidevajo, da bo v prihodnosti edini način komunikacije asinhron [16].

Sporočila se prenašajo preko registrov. Del registrov v računalniku je rezerviranih za sporočila. To so t. i. "zero-copy" sporočila. Pošiljatelj zapiše sporočilo v registre, jedro naredi kontekstni preskok in ob tem pusti vrednost v registrih nespremenjeno. Naslovnik prebere vsebino teh registrov, pri čemer ni bilo potrebno noben del sporočila nikamor poslati (ostal je vpisan v registru).

Problem pri pošiljanju preko registrov nastane zaradi različnih zasnov registrov na različnih strojnih arhitekturah. Zato so bili dodani "virtualni" registri, ki so delno pravi registri, delno pa so le del naslovnega področja v niti, ki sporočilo pošilja. Virtualni registri omogočajo boljšo prenosljivost na različne strojne arhitekture.

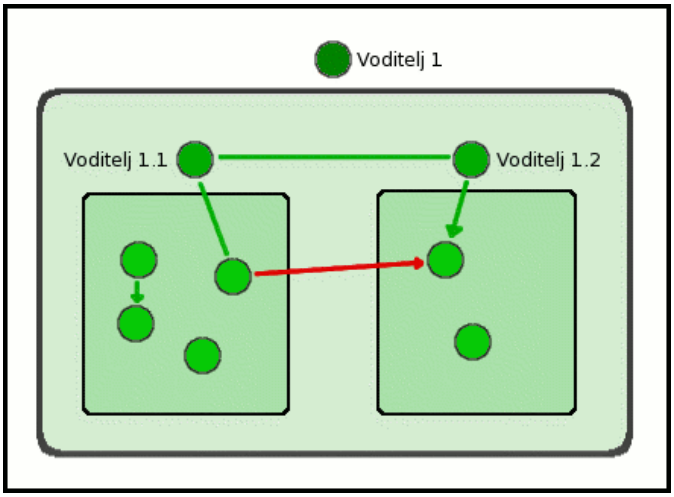
Liedtkejev L4 je omogočal pošiljanje zelo dolgih sporočil, ki so jih razvijalci seL4 in Fiasco.OC zaradi problemov onemogočili.

- Primitivi naprav (Device primitives):
  omogočajo izdelavo gonilnikov naprav, ki delujejo kot uporabniške aplikacije v nepriviligiranem načinu. Jedro spremeni strojne prekinitve v IPC asinhrona sporočila, ki jih dostavi gonilniku.

- Zmogljivosti (Capability spaces):
  zmogljivosti se uporabljajo za nadzor komunikacij med posameznimi aplikacijskimi strežniki.

  Pred nadzorom z zmogljivostmi je L4 deloval tako, da je jedro poslalo

naslovniku sporočilo in identifikacijsko številko pošiljatelja. Ta je nato (glede na pošiljatelja) dobljeno sporočilo sprejel ali izbrisal. Ta način preverjanja ni dober, ker mora aplikacijski strežnik sam nadzirati dostop, čeprav mu pošiljatelj niti ne bi smel poslati sporočila. Prejemniku bi se lahko z veliko nepravilnimi sporočili onemogočilo delovanje. Ves čas, ki bi mu ga jedro dodelilo, bi preverjal pravice in ne bi opravljal svojega osnovnega dela. Liedtke je ta problem rešil z metodo "Klani in voditelji" ("clans and chiefs") (slika 5.3). Vsi procesi s skupnim voditeljem (procesom, ki jih je ustvaril) so del enega klana. Vsaka nit lahko brez preusmerjanj pošlje sporočilo v nit, ki je del istega procesa ali v nit, ki pripada procesu iz istega klana. Vsa druga sporočila se preusmerijo. Sporočilo najprej dobi voditelj klana. Ta ga preusmeri k voditelju klana, ki vsebuje nit, kamor je sporočilo namenjeno. Voditelj drugega klana sporočilo prenese v ustrezno nit. Nit, ki pošilja sporočilo, se te preusmeritve ne zaveda in je zanjo popolnoma transparentna. Vodja klana lahko vsako sporočilo zavrne ali spremeni, preden ga pošlje naprej.



Slika 5.2: Klani in voditelji

Preusmeritve sporočil so predstavljale veliko obremenitev sistema in so zato v obeh novejših L4 sistemih umaknjena. Namesto klanov in

voditeljev se uporablja nadzor dostopa z zmogljivostmi.

Vsak proces ima svoj "CSpace"prostor ("capability space"), kjer ima shranjene vse zmogljivosti.Zmogljivost je definirana kot žeton (oziroma dovoljenje), da lahko ta proces dostopa do drugih virov. Sestavljena je iz kazalca na želeni objekt in pravic, ki jih ima proces nad tem objektom. Dostop do dela spomina, kjer so shranjene zmogljivosti, ima samo jedro.

Ko proces kreira nov proces, mu podeli začetni set zmogljivosti.Zmogljivosti so nastavljene,še preden se prva nit novega procesa prestavi v stanje delovanja. Proces lahko svoje zmogljivosti kopira, prenaša preko IPC sporočil k drugim procesom, ali jih prekliče. Z njimi operacijski sistem izolira komponente (aplikacijske strežnike) med seboj, posamezna komponenta ne more pošiljatisporočil drugim, za katere nima ustreznih zmogljivosti. Sporočilo do naslovnika ne biprišlo, ker jo jedro ne bi preusmerilo.

```
1
2 local loader = L4.default_loader;
3 local svc = loader:new_channel();
4 loader:start({caps = {service = svc:full()}}, "rom/server");
5 loader:start({caps = {service = svc:m("rw")}}, "rom/client")
    ;
```

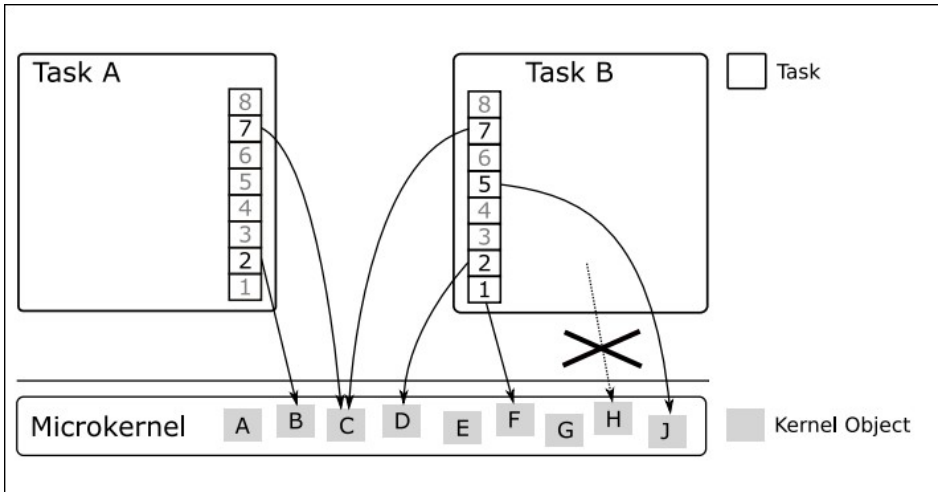
Koda 5.1: Kreiranje novega procesa v L4Re [70]

Koda 5.1 prikazuje primer, ki ustvari dva procesa (strežnik in klient). Procesa med seboj komunicirata preko zmogljivosti.V prvem koraku "loader" (prevzet "loader" je "Moe") ustvari IPC tunel z imenom "svc", preko katerega bo potekala njuna komunikacija.Moe nato ustvari strežniški proces,ki ima vse pravice nad tem tunelom. Proces klienta je ustvarjen naslednji in ima samo pravice branja in pisanja (read/write).

seL4 določa zmogljivosti za vse objekte v jedru: procese,niti, IPC usmernik, semaforji, strani v spominu, IO porti.Če ima neka aplikacija zaupne podatke,se ji ne podeli nobena zmogljivost za komunikacijo z drugimi. Noben drug proces ne bo mogel komunicirati z njo in dostopati

do njenih podatkov.

Posebna vrsta zmogljivostije t. i. zmogljivost za odgovor (“one-time (reply) capability“). Uporablja se pri komuniciranju med klientom in aplikacijskim strežnikom. Strežnik večinoma nima zmogljivosti, da bi lahko poslal sporočilo z odgovorom nazaj klientu. Zato klient ob pošiljanju sporočila, strežniku pošlje še zmogljivost za odgovor. Ko strežnik uspešno odpošlje procesu odgovor, jedro to zmogljivost uniči. Klientu ni potrebno skrbeti, da bi lahko strežnik še kdaj drugič naredil kontakt, brez njegove vednosti.



Slika 5.3: Prikaz uporabe zmogljivosti za dostop do objektov v jedru [46]

Princip zmogljivosti sta predstavila že leta 1966 Dennis in Van Horn [45].

5.3 Fiasco.OC

Fiasco.OC je mikrojedro tretje generacije. Je objektno-orientiran, napisan v jeziku C++ in omogoča omejitev varnosti z zmogljivostmi. Namesti se lahko na x86, MIPS in ARM arhitekture.

Fiasco.OC jedro ima 7 sistemskih klicev. Poleg njih lahko programerji

uporabljajo tudi L4Re (“L4 Runtime Environment“) knjižnice in abstrakcije, s katerimi si olajšajo delo: knjižnica jezika C, pthreads, libstdc++.

Osnovne strežniške komponente v L4Re so [47]:

- Sigma0 pager
  Sigma0 je poseben strežnikpognan takoj nad mikrojedrom. Njegova naloga je razreševanje napak strani, ki jih generira prvi proces (Moe).
- root task Moe
  Moe je prvi proces,ki se požene v L4Re sistemih.Ostalim procesom omogoča,da dobijo sistemske vire: dodeljuje regije spomina,z osnovnim planerjem (scheduler) določa,kateri proces bo pognan naslednji, upravlja z virtualnim pomnilnikom, itd. Da se lahko začnejo izvajati uporabniški aplikacijski strežniki, Moe zažene inicializacijskiprogram Ned.
- init process Ned
  Ned je proces, ki zažene vse aplikacijske strežnike.

5.4 seL4

5.4.1 Uvod

seL4 mikrojedro je zasnovano kot varno in zanesljivo jedro, nad katero lahko uporabniki gradijo svoje sisteme.Aplikacijam omogoča varen dostop do spomina, ustvarjanje procesov ter komunikacijo med njimi.Vsebuje 8700 vrstic kode napisane v jeziku C in je prvi operacijski sistem, ki je matematično dokazan, da je varen.

seL4 ima 8 sistemskih klicev [48]:

- seL4_Send()
  pošlje sporočilo naslovnikuČe noben naslovnik ni pripravljen sporočila sprejeti, se aplikacija zaustavi (blokira), dokler ni sporočilo uspešno dostavljeno.

- seL4_NBSend()
  pošlje sporočilo naslovniku in pri tem ne zaustavi aplikacije, tudi če noben naslovik ne more sprejeti sporočila.

- seL4_Call()
  je podoben kot seL4_Send(), le da aplikacijo blokira, dokler ne dobi odgovora od naslovnika.

- seL4_Wait()
  Ta sistemski klic uporabi nit, ko čaka na sporočilo. Dokler ne pride novo sporočilo, je nit v blokiranem stanju.

- seL4_Reply()
  uporablja se pri pošiljanju odgovora na sporočilo prejeto s seL4_Call() klicem. seL4_Call() je s sporočilom poslal tudi ustrezno zmogljivost (capability), preko katere lahko naslovnik enkratno odgovori pošiljatelju.

- seL4_ReplyWait()
  sta v bistvu dva sistemska klica (seL4_Reply() in seL4_Wait()), povezana skupaj zaradi lažjega programiranja.

- seL4_Poll()
  S tem klicem lahko nit preveri koliko sporočil čaka na pošiljanje.

- seL4_Yield()
  Ta klic uporabi nit, ko ne potrebuje več svojega časovnega intervala. Preostanek njenega intervala se bo dodelil naslednji niti enake prioritete.

5.4.2 Izdelava in verifikacija jedra

Običajno se varnostne pomanjkljivosti odkrijejo s testi in pregledom kode. Matematična utemeljitev in formalna verifikacija delovanja sta dve napredni tehniki, s katerima se lahko odkrijejo pomanjkljivosti ter hkrati dokaže,

da sistem deluje v skladu s pričakovanji[49]. To pomeni, da obstajajo matematični dokazi, da jedro nima preplavitve medpomnilnika (“buffer overflow“), nima dereferenciranja kazalcev z NULL vrednostjo (“null pointer dereference“), uhajanja pomnilnika (“memory leak“) ali nedefiniranega izvajanja.

seL4 je prvo verificirano mikrojedro, ki se ga lahko uporablja v splošne namene. Ob verifikaciji se privzame, da strojna oprema, zaganjalnik (“boot loader“) in prevajalnik (“compiler“) delujejo pravilno. Vse ostalo (vseh 8830 vrstic kode) je matematično preverjeno.

Verifikacija potrjuje, da v sistemu, ki je fizično varen (nihče ga ne more odpreti ali odnesti), ki deluje v eno-procesorskem načinu s strojno opremo, ki ne vsebuje varnostnih pomanjkljivosti, delujejo informacijske poti le tako, kot so bile načrtovane. To pomeni, da nobena aplikacija ne more prebrati podatkov, ki so shranjeni v drugi aplikaciji, ne da bi ji ta to dovolila. Prva aplikacija lahko pri tem uporabi katerikoli način - uporabi lahko katerikoli sistemski klic ali ukaz, vendar vseeno ne bo mogla dostopati do podatkov, ki niso njeni.

Formalna verifikacija predvideva, da je vodilo za direktni dostop do pomnilnika (DMA) ugasnjeno in da ima enota za upravljanje s pomnilnikom (MMU) edina dostop do njega. Enote, ki uporabljajo DMA vodilo, imajo dostop do katerega koli prostora v pomnilniku, vključno z objekti v jedru in prostorom, kjer je zapisana programska koda. Tak dostop negira teorem, da lahko enota dostopa samo do svojega naslovnega področja v spominu. seL4 sistemi lahko še vedno uporabljajo DMA vodila, ob zaupanju, da bo njegov uporabnik deloval varno.

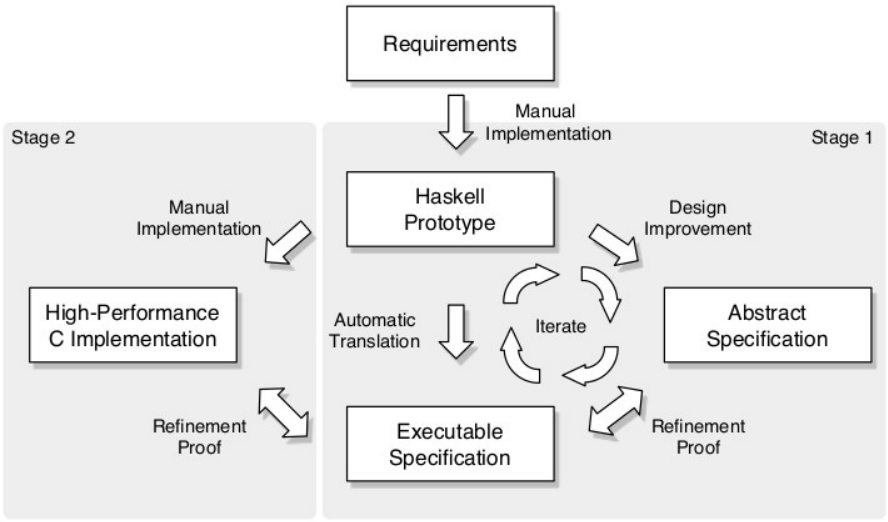
Klein [56] postavlja dve obljubi verifikacije. Verifikacija:

- potrjuje, da program deluje,
- prepriča ostale, da res deluje.

Obljuba o pravilnem delovanju ne določa le, da program deluje brez napak, temveč določa tudi, da program deluje, kot naročnik zahteva, ali želi. Poleg formalne verifikacije (ki je najtežja in hkrati najbolj zanesljivo potrdi

domnevo o pravilnem delovanju) se v tem koraku navadno uporablja samo pregled kode in testiranje.

Druga točka se nanaša na projekte, pri katerih sodelujejo še tretje osebe. Na primer: pri izdelavi letala je potrebno naročniku (npr. Boeingu) dokazati, da letalo deluje v skladu z njihovimi željami.Poleg tega, je to potrebno dokazati tudi ustreznim državnim službam (FAA). Tako dokazovanje je najpreprostejše s predložitvijo potrjene dokumentacije o formalni verifikaciji.



Slika 5.4: Princip izdelave seL4 jedra [51]

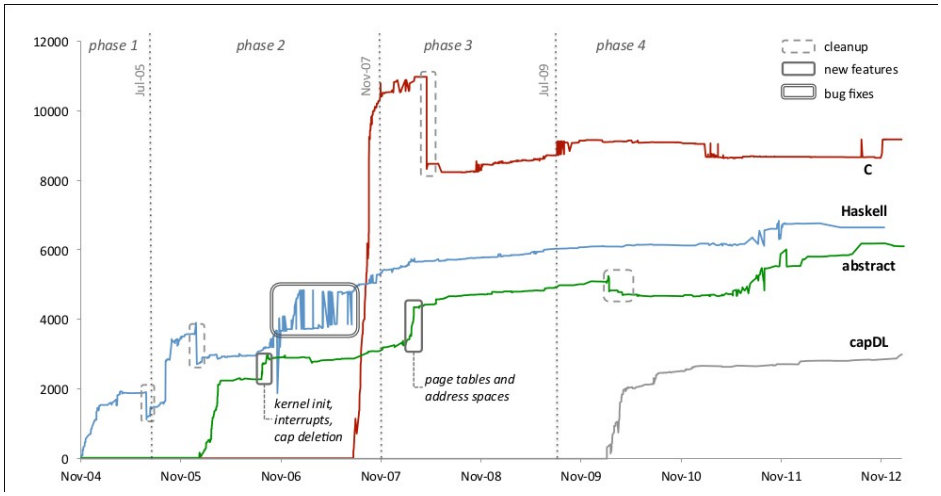
seL4 jedro in njegova formalna verifikacija sta bila izdelana paralelno. Programerji so na podlagi neformalnih zahtev za delovanje in na podlagi specifikacij strojne opreme, najprej izdelali prototip jedra v jeziku Haskell. Medtem je skupina za verifikacijo izdelala okvir (“framework“) in prvo specifikacijo delovanja. Program napisan v jeziku Haskellse lahko avtomatsko pretvori v specifikacije v jeziku Isabelle/HOL [50], na podlagi katerih se izdelajo dokazi za potrjevanje teoremov delovanja.Prototip se lahko izvaja na QEMU simulatorju, kar omogoča uporabnikom testiranje njihovih aplikacij na novih ali spremenjenih sistemskih klicih.

Po potrditvi prvega pravilnega delovanja prototipa, se je v jeziku C začelo

izdelovati pravo mikrojedro. Ves proces izdelave je deloval v enakem ciklu, kjer se je najprej nova funkcionalnost naredila v Haskellprototipu, se nato pretvorila v abstraktno specifikacijo in implementirala v pravo jedro.

Prepisovanje jedra v jezik C je bilo potrebno zaradi več razlogov [51]:

- Haskell runtime (okolje, ki poganja Haskell programe) je velik skupek kode, za katerega bilo težko potrditi pravilno delovanje. Tako velike kode ni mogoče potrditi v enakem času, kot se to lahko naredi za majhno jedro v jeziku C.
- Haskell runtime deluje na principu “garbage collection“, ki ni optimalen za sisteme z odzivom v realnem času.
- v jeziku C se lahko izvede veliko optimizacij, ki so nujne za izboljšanje hitrosti pri mikrojedrih.



Slika 5.5: Časovnica izdelave seL4 jedra [51]

NICTA ocenjuje [51], da je potrebovala 2 človek-leti (čl) za izdelavo Haskell prototipa (design, dokumentacija, programiranje in testiranje), za implementacijo jedra v jezik C pa dodatna 2 človek-meseca (čm)Za izdelavo in verifikacijo pravilnega delovanja je bilo potrebnih kar 20 čl, saj obsega

več kot 200.000 vrstic. Cena za izdelavo jedra je ocenjena na 362 USD na vrstico kode, kar je malo, saj jedra z manj popolno verifikacijo ocenjujejo na vrednost 1000 USD na vrstico kode.

Med razvojem je bilo najdenih 160 napak v kodi. Le 16 jih je bilo odkritih v fazi testiranja, predno se je začela formalna verifikacija. Med verifikacijo se je odkrilo dodatnih 144 napak, kar kaže, kako zelo pomemben je ta korak [52].

Primer verifikacije

Spodnji trije primeri prikazujejo povezavo med Isabelle/HOL kodo za verifikacijo, prototipom v Haskellu in končno kodo v jeziku C. Prikazani del kode, pripada planerju (“scheduler“), ki ureja, katera nit bo naslednja dobila čas na procesorju.

Isabelle/HOL koda za definicijo delovanja planerja je povzeta iz seL4 repozitorija [58].

```
1  |
2  | schedule_ end:
3  |     cur ← cur_thread ;
4  |     threads ← allActiveTCBs ;
5  |     thread ← select threads ;
6  |       if thread = cur then
7  |       return () OR switch_to_thread     thread
8  |     else
9  |       switch_to_thread     thread
10 |  od OR
11 |  switch_to_idle_thread
```

Koda 5.2: Isabelle/HOL za planer

Haskell koda za planer, povzeta po [59]:

```
1
2  > schedule :: Kernel()
3  > schedule = do
4  >   curThread ← getCurThread
5  >   action ← getSchedulerAction
6  >   case action of
7  >
8  >     ResumeCurrentThread -> return()
9  >
10 >     SwitchToThread t -> do
11 >       curRunnable ← isRunnable curThread
12 >       when curRunnable $ tcbSchedEnqueue curThread
13 >       switchToThread t
14 >       setSchedulerAction ResumeCurrentThread
15 >
16 >     ChooseNewThread -> do
17 >       curRunnable ← isRunnable curThread
18 >       when curRunnable $ tcbSchedEnqueue curThread
19 >       domainTime ← getDomainTime
20 >       when (domainTime ≡ 0) $ nextDomain
21 >       chooseThread
22 >       setSchedulerAction ResumeCurrentThread
23 >
24 Threads are scheduled using a simple multiple-priority round
   robin algorithm.
25 >
26 > chooseThread :: Kernel()
27 > chooseThread = do
28 >   curdom ← if numDomains > 1 then curDomain else return 0
29 >   l1 ← getReadyQueuesL1Bitmap curdom
30 >   if l1 /= 0
31 >     then do
32 >       let l1index = wordLog2 l1
33 >       l2 ← getReadyQueuesL2Bitmap curdom l1index
34 >       let l2index = wordLog2 l2
35 >       let prio = l1IndexToPrio l1index .|. fromIntegral
   l2index
36 >       queue ← getQueue curdom prio
37 >       let thread = head queue
38 >       runnable ← isRunnable thread
39 >       assert runnable "Scheduler: non-runnable thread"
40 >       switchToThread thread
41 >     else
42 >       switchToIdleThread
```

Koda 5.3: Haskell koda za planer

Končna koda za planer v jeziku C. Povzeto po [60]:

```
1  |
2  |void
3  |schedule(void)
4  |{
5  |  word_t action;
6  |
7  |  action = (word_t)ksSchedulerAction;
8  |  if (action == (word_t)SchedulerAction_ChooseNewThread){
9  |    if (isRunnable(ksCurThread))
10 |        tcbSchedEnqueue(ksCurThread);
11 |    }
12 |    if (ksDomainTime == 0) {
13 |        nextDomain();
14 |    }
15 |    chooseThread();
16 |    ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
17 |  } elseif (action != (word_t)
   |    SchedulerAction_ResumeCurrentThread)    {
18 |    if (isRunnable(ksCurThread))
19 |      tcbSchedEnqueue(ksCurThread);
20 |    }
21 |    /* SwitchToThread */
22 |    switchToThread(ksSchedulerAction);
23 |    ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
24 |  }
25 |}
```

Koda 5.4: C koda za planer

Binarna verifikacija

Ob verifikaciji binarne datoteke, se primerja njena vsebina z vsebino C datoteke iz katere je bila prevedena. Najprej se osnovna C koda spremeni v bolj abstraktno verzijo, kjer se globalne spremenljivke spremeni toliko, da so v spominu, ki se ga lahko naslovi. Nato se kodo preslika v jezik grafov (“intermediate graph language“)Binarno datoteko se povratno prevede (“decompile“) preko HOL4 in Isabelle/HOL v enak jezik grafov (povzeto po [56]).

Poglavje 6

Uporaba mikrojeder

6.1 Mach

Operacijski sistem Applovih računalnikov (“Darwin“) vsebuje jedro, imenovano XNU [57]. Predstavljajo ga kot hibridno jedro, zasnovano na MACH 3.0 mikrojedru, ki so mu dodali kodo iz operacijskega sistema FreeBSD. Pri tem MACH upravlja med-procesno komunikacijo, klic oddaljenih funkcij (RPC), planer (scheduler), storitve, ki delujejo v realnem času in navidezni pomnilnik. BSD del pa datotečne sisteme, povezovanje (networking), način varnosti, kot jo pozna UNIX, ustvarjanje procesov, kernel API in POSIX niti.

MACH sistem je mikrojedrni sistem. XNU, ki združuje MACH in BSD jedro, pa po definiciji minimalizma to ni več.

6.2 L4

L4 jedro se največkrat uporablja v vgradnih sistemih, takrat, ko je pomembna stroga ločitev med aplikacijami, ali ko je zaradi majhnega pomnilnika nemogoče vgraditi celotno monolitno jedro.

Do danes največja vgradnja L4 sistema je sistem OKL4, ki je vgrajen v vse iOS naprave [53]. Predvideva se, da je vseh vgradenj več kot milijarda [54]. OKL4 skrbi za delovanje t. i. varnega dela v napravi (“Secure Enclave“). To

je popolnoma ločen del iOS naprave, ima svoj pomnilnik, svoj procesor in Applovo verzijo OKL4 sistema.

Ko se iOS naprava izdela, se v ta del zapiše unikatna identifikacijska številka, ki ni dostopna iz drugih delov sistema. Tudi Apple ne pozna njene vrednosti. Z njo se šifrira varni del, da ni berljiv, tudi če si ga kdo preslika. Z iOS sistemom je povezan preko posebnega vodila. Skrbi za izdelavo varnih naključnih števil, za shranjevanje podatkov o prstnih odtisih, varuje uporabniško geslo - upravlja delovanje vseh varnostnih mehanizmov.

DARPA je leta 2014 naročila verzijo seL4 sistema za poganjanje dronov [55]. Sistem v dronu je ločen na dva dela prvi del vsebuje kontrolo nad letenjem, drugi del pa dodatne aplikacije (fotografije, snemalnik, itd.). Nobena aplikacija ne more ponesreči ali pod prisilo (če je bila prevzeta s strani nasprotnikov), spreminjati načina in kontrole letenja.

6.3 QNX

QNX je komercialen sistem s QNX Neutrino mikrojedrom. Od leta 2010 je njegov lastnik podjetje BlackBerry [63].

Uporabljen je v avtomobilski, medicinski in telefonski industriji. Blackberry PlayBook tablica in Blackberry 10 imata QNX operacijski sistem. QNX sistem za podporo informacijskim zaslonom in telemetriji v avtomobilih je vgrajen v preko 20 milijonov vozil [64].

QNX Nukleus je prisoten v defibrilatorjih [65], ultrazvočnih napravah [68], Garmin in Honeywell ga uporabljata v letalski industriji [66, 67], poganja FlexStor podatkovno skladišče [69], ...

6.4 PikeOS

PikeOS je zaprtokodna verzija sistema L4, ki je certificiran za delovanje v kritičnih aplikacijah v letalih in vlakih.

Primeri uporabe so npr.: brezžični nadzor vlaka (gibanje, lokacija, za-

vorna dolžina - Samsung SDS aplikacija),aplikacije v pilotski kabini letala Airbus A350, itd. [61].

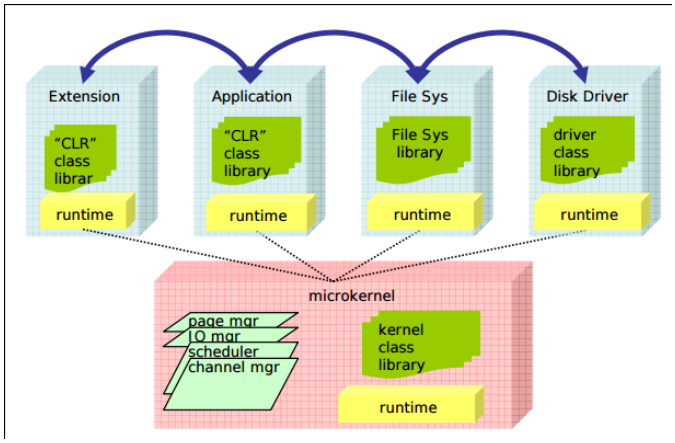
6.5 Symbian

Symbian je operacijski sistem, ki je leta 2006 poganjal kar 73 % vseh mobilnih telefonov [71]. Njegovo mikrojedro vsebuje planer, enoto za upravljanje s spominom in gonilnike.Podpora za dostop do mreže, telefonija ter datotečni sistemi so postavljeni v plast s strežniki [72].

6.6 Singularity

Je bilo eksperimentalno mikrojedro podjetja Microsoft [73]. Razvoj je potekal od leta 2003 do leta 2010.

Del jedra, ki je bil zadolžen za hitre odzive na prekinitve, je bil napisan v zbirniku in jeziku C. Ostala koda (ter tudi gonilniki) so bili izdelani v jeziku Sing# (razširjen C#). Komercialna verzija Singularity sistema se je imenovala “Midori“ in je bila nekaj časa osnova za iskanje z naravnim jezikom (Natural language search) na ameriški zahodni obali in v Aziji [73].



Slika 6.1: Struktura Singularity sistema [74]

6.7 μ-velOSity

Je operacijski sistem z odzivom v realnem času [30]. Uporabljajo ga predvsem v letalski in vojaški industriji, povsod, kjer je še posebno velik poudarek na zanesljivih sistemih.

Poglavje 7

Kritike mikrojeder

7.1 Opis

Debata o dobrih in slabih lastnostih mikrojeder ter o boljših in slabših zmogljivostih glede na monolitne sisteme se odvija že od leta 1992. Največ sta o tem “debatirala“ Linus Torvalds (zagovornik monolitnih sistemov, vodja razvoja jedra Linux) in Andrew Tanenbaum (zagovornik mikrojedrnih sistemov, (zdaj že upokojeni) profesor na Nizozemski fakulteti Vrije). Prva debata je bila leta 1992 [81], ko je Linus izdal prvo verzijo Linux jedra, druga pa leta 2006 [82, 83],ko je izšel članek “Can We Make Operating Systems Reliable and Secure? “ [85] v IEEE reviji.

V to debato so se vključili tudi drugi razvijalci [84], nekateri [86, 87]so enake probleme izpostavili v svojih člankih.

V tem delu naloge so strnjene najbolj znane kritike in odgovori na njih.

7.2 Količina kode

Glede na podatke o velikosti mikrojeder, bi lahko bralec mislil, da je količina kode pri njihovih sistemih veliko manjša kot pri monolitnih. To je napačno, saj je že samo mikrojedro skupek kode, ki je monolitna jedra ne potrebujejo (npr. IPC komunikacija).

Kodo, ki je pri monolitnih sistemih del jedra (gonilniki, datotečni sistemi), je potrebno v mikrojedrnem sistemu prav tako programirati. V določenih primerih lahko postane programiranje težje, saj je potrebno za zagon funkcije iz drugega dela operacijskega sistema, izvesti IPC klic.

Število napak v kodi pri mikrojedrih ni manjše. Statistično ni pomembno, kakšno jedro se izdeluje, odstotek napak je vedno enak. Lahko da pri mikrojedrih (lahko) napake v aplikacijskih strežnikih naredijo manj škode.

7.3 Zahtevnost sistema

Mikrojedra naj bi bila bolj preprosta za razumevanje, ker so višje funkcije (datotečni sistemi, gonilniki, itd.) postavljeni v ločene module. Torvalds temu nasprotuje in dodaja, da se prava kompleksnost sistema pozna šele pri interakciji med posameznimi komponentami. Ločeni (modularni) strežniki ne morejo uporabljati skupnih struktur, zaklepanj in predpomnilnika, ker ima sistem vzpostavljene meje med njimi.

Iskanje napak in mest, kjer so se te napake zgodile, je lahko pri bolj razdeljenih sistemih težje.

Tanenbaum in Shapiro zagovarjata nasprotno načelo, kjer je nič ali zelo malo podatkovnih struktur in zaklepanj. Vsak sistem naj bi skrival svoje lastne strukture, z zunanjimi pa naj ne bi sodeloval.

Oba se strinjata s Torvaldsovim komentarjem, da mikrojedrni sistemi niso preprosti. Trdita, da se posamezne mikrojedrne strežnike lažje naredi, testira in razume, toda celotni sistem je enako zahteven kot monolitni.

7.4 Stabilnost sistema

Minix3 omogoča transparentni ponovni zagon aplikacijskega strežnika ali gonilnika, če ta preneha z delovanjem. Čeprav to funkcionalnost dokazujejo tudi na konferencah [79], Torvalds meni, da so velikokrat strežniki tako zelo povezani med seboj, da napačen odziv enega povzroči napako ali neuspešen

zaključek operacije tudi v drugih.

Shapiro temu nasprotuje in odgovarja, da ne obstaja niti en monolitni sistem, ki bi bil zrel za uporabo v sistemu z visoko stopnjo varnosti ali robustnosti. Prav vsi sistemi, ki so v teh primerih uporabljeni, so mikrojedrni.

7.5 Boljše zmogljivosti

V mikrojedrnih sistemih so vse višje funkcije razdeljene na posamezne strežnike, zato bi bilo možno vsakega od njih postaviti na svoj namenski računalnik. Toda zaradi optimizacije hitrosti kopiranja podatkov, se velikokrat uporablja preslikovanje pomnilnika med procesi, ki med seboj komunicirajo. Zaradi tega sistema ni možno preurediti v distribuiranega.

Poglavje 8

Kratka primerjava

8.1 Primerjava arhitektur

V spodnji primerjalni tabeli so prikazani vsi glavni mikrojedrni operacijski sistemi, z izjemo QNX. Le-ta je zaprtokodni in preverjenih tehničnih podatkov o njem ni lahko dobiti. Poleg mikrojeder sta, za boljšo referenco, na zadnjih dveh mestih predstavljena še Linux in Windows 8.

Navajanje razlik med različnimi operacijskimi sistemi seveda ne določa njihove kvalitete. Podatki, ki so prikazani ne pomenijo, da je sistem slabši ali boljši od drugega (ker ima npr. manjše število sistemskih klicev).

Ime	Leto	Št. sist. klicev	Št. vrstic kode	Hitrost IPC	Status
Mach	1987 - 1994	140	100.000	100μs	Se ne razvija
Minix	1987 -	40	5.000	10μs	Aktiven
Fiasco.OC	2009 -	7	10.000	1μs	Aktiven
seL4	2008 -	3	9.000	0.2 - 1μs	Aktiven
Linux 4.3	1991 -	380	18.000.000	-	Aktiven
Windows 8	1993 -	410+	50.000.000	-	Aktiven

Tabela 8.1: Primerjava operacijskih sistemov

Pri Windows 8, predstavlja večina kode grafični vmesnik, ki ga Linux

jedro nima v istem paketu in zato tam ni všteto.

Razlika v hitrosti IPC komunikacije med sistemoma Minix in seL4 je velika. seL4 je namreč optimiziran do te mere, da je komunikacija najhitrejša. Npr., prenašanje kratkih sporočil se kopira preko registrov, tako da se vsebina teh ohranja, ko se preklaplja med procesi. Minix je narejen tako, da ga je lahko razumeti, saj je del predavanj na univerzi Vrije.

Mach sistem je edini, ki se ne razvija več aktivno.

8.2 Primerjava hitrosti

Leta 1997 [26] je bila narejena primerjava med sistemoma MkLinux in L4Linux.

MkLinux je celotno Linux jedro, ki deluje na Mach 3 platformi. Jedro je spremenjeno do te mere, da sistemski klici ne dostopajo direktno do strojne opreme, temveč jo naslavljajo posredno preko Mach-ovih. Linux sistem je virtualiziran na Mach jedru.

L4Linux je podoben sistem kot MkLinux, le da je Linux virtualiziran nad jedrom L4. Rezultati analize so pokazali, da je uporaba mikrojedra povzročila 5 % do 10 % počasnejše delovanje uporabniških aplikacij.

L4Linux je dosegal 95 % hitrosti Linuxa, medtem ko je MkLinux dosegel zgolj 62 %. Kar nakazuje da je uporaba hitrega mikrojedra res smiselna in ne povzroča velikega odstopanja hitrosti. Hkrati analiza pokaže, da je bil Mach veliko počasnejši in bi le s težavo opravičili njegovo uporabo pri virtualizaciji.

Poglavje 9

Prihodnost

Vsi trije “glavni“ predstavniki mikrojedrnih sistemov (MINIX, Fiasco.OC in seL4) se bodo razvijali še naprej. Razlike med seL4 in Fiasco.OC se bodo kljub skupnemu začetku poglobile. V prihodnosti bosta razvila vsak svoje lastne rešitve za hitrejše delovanje ali dodatne funkcionalnosti. Vsi imajo močno bazo uporabnikov, predvsem v svetu vgradnih računalnikov. Le-ta je tako velika, da lahko omogoča obstoj vseh treh.

Prednost, ki jo ima seL4 zaradi formalne verifikacije, ga bo utrdila povsod tam, kjer je ta z zakonom predpisana. Formalno verifikacijo bodo razširili še na ostale arhitekture, izboljšali knjižnice ter poenostavili programiranje aplikacijskih strežnikov.

Fiasco.OC in L4Re sta oba dokončana produkta. Razvoj, ki pri njiju trenutno poteka, je vezan na zahteve končnih uporabnikov. Predvsem gre za podporo novim strojnim arhitekturam (na primer “MIPS“) in dodajanje novih lastnosti, ki jih končni produkt potrebuje. Ker je projekt odprtokodne narave, se veliko novih zamisli lahko prenese nazaj v originalni Fiasco.OC, čeprav jim (tako pravijo razvijalci) velikokrat za take podvige zmanjkuje virov.

Poglavje 10

Zaključek

Mikrojedra odpirajo zelo zanimivo smer razvoja operacijskih sistemov. Današnji monolitni sistemi so ogromni, z več milijoni vrstic kode. Sledenje spremembam v njihovi kodi ter preverjanje delovanja postaja vedno težje. Kljub trudu, da bi njihova jedra ostala manjša, je iz arhitekturnega stališča težko narediti večje spremembe.

Eden od načinov, da se lažje obvlada količina kode, ki jo morajo današnji operacijski sistemi upravljati, je razdelitev na mikrojedro in module. S tem se porazdeli zahtevnost in količina kode. Posledica tega je manj napak in bolj jasno določena razmerja med posameznimi moduli. Take module je lažje zamenjati ali ponovno zagnati, če ne delujejo pravilno. Pri modulih manjše velikosti postane formalna verifikacija izvedljiva opcija. Število dobrih lastnosti, ki jih verifikacija prinese sistemu, se lahko vidi pri jedru seL4.

Verjetnost, da bi se kateri od obstoječih monolitnih sistemov spremenil v mikrojedrnega je zelo majhna. Količina dela, ki bi bila za to potrebna, je prevelika. Večja verjetnost je ta, da bodo, na kateri izmed univerz začeli razvijati splošen mikrojedrni sistem, ki bo nekoč dovolj priročen za vsakodnevno uporabo.

Slike

2.1 Prikaz razlik med monolitnimi in mikrojedrnimi sistemi 14

4.1 Delovanje datotečnega strežnika. (Povzeto po [88]) 33
4.2 Plasti v MINIX3 sistemu. (Povzeto po [39]) 34

5.1 Družina L4 mikrojeder 45
5.2 Klani in voditelji 49
5.3 Prikaz uporabe zmogljivosti za dostop do objektov v jedru [46] 51
5.4 Princip izdelave seL4 jedra [51] 55
5.5 Časovnica izdelave seL4 jedra [51] 56

6.1 Struktura Singularity sistema [74] 62

Tabele

5.1 Število ciklov za prenos enega sporočila [16]. 46

8.1 Primerjava operacijskih sistemov 67

Literatura

[1] A. S. Tanenbaum, H. Bos *Modern Operating Systems, 4th edition*, Pearson Education, str. 862 7, 8

[2] A. S. Tanenbaum, H. Bos *Modern Operating Systems, 4th edition*, Pearson Education, str. 6 9

[3] A. Singh, *Mac OS X Internals: A systems approach*, June 2006, strani 17, 18

[4] *History of mach*, Free Software Foundation, 2001 17

[5] R. Rashid, *From RIG to Accent to MAch: The evolution of a Network Operating System* 18

[6] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, *Mach: A New Kernel Foundation For UNIX Development* 20

[7] R. F. Rashid, G. Robertson, *Accent: A communication oriented network operating system kernel*, strani 64–75. ACM, December 1981.

[8] R. Rashid, *Unix Review*, Avgust 1986 19

[9] M. Condict, D. Bolinger, D. Mitchell, E. McManus, *Microkernel modularity with integrated kernel performance*, Technical report, OSF Research Institute, June 1994. 19

[10] J. B. Chen, B. N. Bershad, *The impact of operating system structure on memory system performance*, in Proceedings of the 14th ACM Symposium on Operating Systems Principles, December 1993. 19

[11] Apple, developerlibrary, dostopno na naslovu https://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/OSX_Technology_Overview/SystemTechnology/SystemTechnology.html#//apple_ref/doc/uid/TP40001067-CH207-BCICAIFJ 20

[12] M. Guillemont, *The Chorus distributed operation system: Design and implementation*, In Proceedings of the ACM International Symposium on Local Computer Networks (Firenze, Italy, Apr. 1982) ACM Press, 1982 20

[13] A. S. Tanenbaum, M. F. Kaashoek, *The Amoeba Microkernel*, Vrije Universiteit 21

[14] J. Liedtke, *Toward Real Microkernels*, Communications of the ACM, 39(9):70-77, September 1996 21

[15] J. Liedtke *Improving IPC by kernel design*, 14th ACM Symposium on Operating System Principles. Asheville, NC, USA, December 1993 18, 21

[16] K. Elphinstone, G. Heiser, *From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?*, SOSP, November 2013 22, 24, 46, 48, 72

[17] *NICTA L4 Microkernel to be Utilised in Select QUALCOMM Chipset Solutions*, (Press release), NICTA, November 24, 2005 22

[18] QNX, Blackberry Subsidiary, Dostopno na naslovu http://www.qnx.com/

[19] BlackBerry 10, dostopno na naslovu https://en.wikipedia.org/wiki/BlackBerry_10 23

[20] Steve McConnel, Code Complete, Microsoft press, 2nd edition, 2004 24

[21] G.Klein, K. Elphinstone, G Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood, *seL4: Formal Verification of an OS Kernel*, Open Kernel Labs 25

[22] T. Nipkow, L. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of LNCS. Springer, 2002 25

[23] J. S. Shapiro, J. M. Smith, and D. J. Farber *EROS, A Fast Capability System* Proc. 17th ACM Symposium on Operating Systems Principles, Kiawah Island Resort, SC, USA, December 1999, strani. 170–185.

[24] NOVA Microhypervisor, dostopno na: http://hypervisor.org/ 26

[25] 8.700 LOC, 1 Microkernel, 0 Bugs, dostopno na: http://research.microsoft.com/en-us/events/vsworkshop2009/gernot_heiser.pdf 25

[26] H. Hartig, M. Hormuth, J. Liedtke, S. Schonberg, J. Wolter, *The performance of micro-Kernel-Based Systems*, 16th ACM Symposium in OS Principles, Oktober 1997 29, 68

[27] Silberschatz, Galvin, Gagne. *Applied Operating System Concepts*. John Wiley & Sons, Inc. 2003 7.

[28] Andrew S. Tanenbaum, *Modern operating systems*, Prentice Hall. 1992 8

[29] D. Irtegov, *Operating Systems Fundamentals*, Charles River Media, 2003 8

[30] A. S. Tanenbaum, A. S. Woodhull, *Operating systems, 3nd editon*, Prentice Hall. 2006, stran 42. 11

[31] Apple uporabi nanojedro za MacOS, dostopno na: http://developer.apple.com/technotes/tn/tn2028.html 13

[32] Adeos OS, dosegljivo na: http://home.gna.org/adeos/ 13

[33] WMWare, dostopno na: http://www.vmware.com 13

[34] Xen, dostopno na: http://www.xenproject.org/ 13

[35] M. Swift, M. Annamalai, B. Bershad, and H. Levy, *Recovering Device Drivers. Proc. Sixth Symp. on Oper. Syst. Design and Impl.*, 2004 27

[36] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, *An Empirical Study of Operating System Errors.*, 2001 27

[37] A. S. Tanenbaum, dosegljivo na: http://cacm.acm.org/magazines/2016/3/198874-lessons-learned-from-30-years-of-minix/fulltext 31

[38] A. S. Tanenbaum, H. Bos *Modern Operating Systems, 4th edition*, Pearson Education, str. 66 32

[39] Herder, Bos, Gras, Homburg, Tanenbaum, *MINIX 3: A Highly Reliable, Self-Repairing operating System*, Dept. of Computer Science, Vrije University, Amsterdam. 32, 34, 35, 36, 71

[40] MINIX3 developer guide, dostopno na: http://wiki.minix3.org/doku.php?id=developersguide:overviewofminixarchitecture 35

[41] Tanenbaum Torvalds debate, dosegljivo na: https://en.wikipedia.org/wiki/Tanenbaum%E2%80%93Torvalds_debate 43

[42] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter. The performance of micro-kernel-based systems. In 16th ACM Symposium on Operating System Principles (SOSP), pages 66-77, Saint-Malo, France, October 1997. http://os.inf.tu-dresden.de/pubs/sosp97/ 43

[43] L3, dostopno na http://os.inf.tu-dresden.de/L4/l3doc.html 44

[44] A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concept*, John Wiley & Sons, Inc., sixth edition, 2001. Appendix B, The Mach system. 39

[45] J.B. Dennis, E.C. Van Horn, *Programming Semantics for Multiprogrammed Computations.* Communications of the ACM, 9(3):143–155, March 1966. 51

[46] A. Lackorzynski, A. Warg, *Taming Subsystems: Capabilities as Universal Resource Access Control in L4* 51, 71

[47] L4Re - L4 Runtime Environment, dosegljivo na https://l4re.org/doc/l4re_servers.html 52

[48] seL4 Reference Manual, For v3 development branch 52

[49] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, G. Klein, *seL4: from General Purpose to a Proof of Information Flow Enforcement*, 2013 IEEE Symposium on Security and Privacy 54

[50] T. Nipkow, L. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283. 52

[51] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, G. Heiser, *Comprehensive Formal Verification of an OS Microkernel*, NICTA, UNSW, Sydney, Australia 55, 56, 71

[52] Proof Statistics, dosegljivo na: http://ssrg.nicta.com.au/projects/TS/l4.verified/numbers.pml 57

[53] iOS Security, dostopno na: https://www.apple.com/business/docs/iOS_Security_Guide.pdf 60

[54] Currently maintained kernel implementations, OKL4, doesgljivo na: http://l4hq.org/projects/kernel/ 60

[55] Secure drones a step closer following NICTA tests, dostopno na: http://www.computerworld.com.au/article/585167/secure-uavs-step-closer-following-nicta-tests/ 61

[56] G. Klein, *Operating System Verification — An Overview*, ACM Transactions on Computer Systems (TOCS), Volume 32 Issue 1, February 2014 54, 59

[57] Apple KernelProgramming guide, dostopno na https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/build/build.html 60

[58] Primer Isabelle kode, dostopno na: https://github.com/seL4/l4v/blob/094fb48623d473dad551938c5329b67f998d4133/spec/abstract/Schedule_A.thy 57

[59] Primer Haskell kode, dostopno na: https://github.com/seL4/l4v/blob/218f6ccbf3441e14aacd5f85828a337fc3acff30/spec/haskell/src/SEL4/Kernel/Thread.lhs 57

[60] Primer C kode, dostopno na: https://github.com/seL4/seL4/blob/b327dac1f9dc45b62e02d891d268424a35083d66/src/kernel/thread.c 58

[61] PikeOS Hypervisor, dostopno na: *https://www.sysgo.com/products/pikeos-hypervisor/why-pikeos/* 62

[62] J. S. Shapiro, N. Handy, *EROS: A Principle-Driven Operating System from the Ground Up*, IEEE Software, January / February 2002, 26 - 37 26

[63] Qnx, dostopna na http://www.qnx.com/ 61

[64] Ford Ditches Microsoft For QNX In Latest In-Vehicle Tech Platform, dostopno na: http://techcrunch.com/2014/12/11/ford-ditches-microsoft-for-qnx-in-latest-in-vehicle-tech-platform/ 61

[65] QNX v defibilatorju, dostopno na https://www.mentor.com/embedded-software/success/zoll 61

[66] QNX v letalski industriji, dostopno na https://www.mentor.com/embedded-software/success/garmin_at 61

[67] QNX v napravi za preverjanje višine terena, dostopno na https://www.mentor.com/embedded-software/success/honeywell 61

[68] QNX v napravi za utrazvok, dostopno na https://www.mentor.com/embedded-software/success/convertible_ultrasound 61

[69] QNX v napravi za podatkovno skladišče, dostopno na https://www.mentor.com/embedded-software/success/bdt-ag-success 61

[70] Connecting clients and servers, dostopno na: https://l4re.org/doc/l4re_concepts_env_and_start.html 50

[71] Symbian market share analysis, dostopno na: http://www.allaboutsymbian.com/news/item/5059_Symbian_market_share_analysis_.php 62

[72] Symbian, dostopno na: https://en.wikipedia.org/wiki/Symbian#Operating_system 62

[73] Singularity, dostopno na: http://research.microsoft.com/en-us/projects/singularity/ 62

[74] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, B. Zill, *An Overview of the Singularity Project*, Microsoft Research Technical Report, MSR-TR-2005-135 62, 71

[75] RedoxOS, dosegljivo na: http://www.redox-os.org/ 23

[76] The kernel, dosegljivo na: https://doc.redox-os.org/book/introduction/how_redox_compares_to_other_operating_systems.html 23

[77] URLs, dosegljivo na: https://doc.redox-os.org/book/design/url/urls.html 23

[78] Multics Concepts For the Contemporary World, dosegljivo na: https://groups.google.com/forum/#!topic/comp.arch/w80NfXLjaF0%5B76-100%5D

[79] Minix auto-healing, dostopno na: https://www.youtube.com/watch?v=vlOsy0PZZyc 65

[80] μ-velOSity real-time operating system, dostopno na: http://www.ghs.com/products/rtos/integrity.html 63

[81] LINUX is obsolete, dostopno na: https://groups.google.com/forum/#\mskip-\thinmuskiptopic/comp.os.minix/wlhw16QWltI%5B1-25%5D 64

[82] Hybrid (micro)kernels, dostopno na: http://www.realworldtech.com/forum/?threadid=65936&curpostid=65915 64

[83] Tanenbaum-Torvalds debate: Part II, dostopno na http://www.cs.vu.nl/~ast/reliable-os/ 64

[84] Debunking Linus's latest, dostopno na: http://www.coyotos.org/docs/misc/linus-rebuttal.html 64

[85] A. S. Tanenbaum, J. N. Herder, H. Bos, *Can We Make Operating Systems Reliable and Secure?*, IEEE Computer, May 2006, strani 44 - 51 64

[86] Microkernel Debate, dostopno na: http://tunes.org/wiki/microkernel_20debate.html 64

[87] Microkernels are slow, dostopno na: http://blog.darknedgy.net/technology/2016/01/01/0/ 64

[88] Experiments with Minix Operating System, dostopno na: http://sudevambadi.me/MinixMajor/ 33, 71

[89] DO-178B standard, dostopno na https://en.wikipedia.org/wiki/DO-178B 17

[90] ISO 26262 standard, dostopno na https://en.wikipedia.org/wiki/ISO_26262 17

[91] HelenOS, dostopno na: http://www.helenos.org/ 24

[92] F9, dostopno na: https://github.com/f9micro/f9-kernel