Tampereen yliopisto

Niilo Rannikko

KÄYTTÖJÄRJESTELMÄYTIMIEN OHJELMOINTI RUST-OHJELMOINTIKIELELLÄ

Kandidaatintutkielma
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Joonas Multanen
Toukokuu 2023

TIIVISTELMÄ

Niilo Rannikko: Käyttöjärjestelmäytimien ohjelmointi Rust-ohjelmointikielellä
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma, tietotekniikka
Toukokuu 2023

Käyttöjärjestelmien ydinten ohjelmoinnissa käytettävällä ohjelmointikielellä on suuri merkitys. Ohjelmointikielen tulee tarjota käyttäjälleen välineet riittävään kontrolliin tehokkaan ohjelmakoodin kirjoittamiseksi samaan aikaan tuottaen järjestelmälle mahdollisimman vähän rasitusta. Tälläiset ohjelmointikielet ovat aikaisemmin olleet kompromissi korkeamman tehokkuuden saavuttamiseksi ohjelmointikielen turvallisuuden kustannuksella, jolloin myös vaikeita ongelmia aiheuttavien muistinhallinnan ja rinnakkaisuuden virheiltä suojautuminen on ollut ohjelmakoodin kehittäjän vastuulla.

Uudehko ohjelmointikieli Rust tarjoaa mahdollisuuden saavuttaa sekä tehokkaan ohjelmakoodin, että turvan muistin ja rinnakkaisuuden virheitä vastaan. Kirjallisuuskatsauksessa selvitetään käyttöjärjestelmäytimen kehityksen olennaisia piirteitä ja tutkitaan Rust-ohjelmointikielen sopivuutta siihen. Työssä perehdytään esimerkkinä myös tuoreeseen Linux-käyttöjärjestelmäytimeen lisättyyn Rust-tukeen.

Työssä todetaan Rustin sisäänrakennettujen muistiturvallisuuden ja toimivan rinnakkaisuuden mahdollistavien ohjelmakoodin käännösvaiheen (engl. *compile*) turvatarkastusten tuovan merkittävää helpotusta käyttöjärjestelmäydinten ja järjestelmätason ohjelmistokehitykseen. Rustin omistajuuden, lainaamisen ja eliniän käsitteet, staattinen tyypitysjärjestelmä ja rinnakkaisuuden työkalut muodostavat järjestelmän, jossa suuri osa muistinhallinnan virheistä karsiutuvat pois ohjelmakoodin käännösvaiheessa ja rinnakkaisten menetelmien käyttö on sujuvaa.

Rust-ohjelmointikielen tarjoamat helpotukset on myös huomattu ja hiljattain, ennen lähes yksinomaan C-ohjelmointikielellä kehitetty Linux-käyttöjärjestelmän ytimeen lisättiin tuki Rust-ohjelmointikielelle. Varhaisimmat Rustilla toteutetut Linux-ytimen ajurit ovat osoittaneet lupaavia tuloksia, niin ohjelmakoodin kehityskokemuksen, kuin ajurien tehokkuuden osalta. Rustin voidaankin ennustaa kasvattavan suosiotaan, niin Linux-ytimen kehityksessä, kuin muussakin järjestelmätason ohjelmoinnissa.

Avainsanat: Rust, käyttöjärjestelmä, ydin, muistiturvallisuus, rinnakkaisuus, Linux

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. KÄYTTÖJÄRJESTELMÄT 2
2.1 Vastuut 2
2.2 Käyttöjärjestelmän ydin 3
2.3 Ydinten kehitys 4
3. RUST-OHJELMOINTIKIELI 6
3.1 Erityispiirteet 7
3.1.1 Muistiturvallisuus ilman roskienkerääjää 7
3.1.2 Staattinen tyypitys 8
3.1.3 Rinnakkaisuuden menetelmät 9
3.2 Kritiikkiä 9
4. CASE: LINUX-YTIMEN OHJELMOINTI RUSTILLA 11
4.1 Toiminta 11
4.2 Toteutetut Linux-käyttöjärjestelmäytimen moduulit 12
4.2.1 NVMe Ajuri 12
4.2.2 M1 GPU ajuri 12
4.3 Linux-ytimen tulevaisuus 13
5. YHTEENVETO 15
LÄHTEET 17

1. JOHDANTO

Tietokoneiden käyttöjärjestelmät ovat tärkeä osa kaikkien tietokoneiden toiminnasta. Niiden tehtävänä on huolehtia tietokoneen sujuvasta ja tehokkaasta toiminnasta, tehtävien prosessien oikeellisuudesta ja monesta muusta asiasta, joista moni ei välttämättä edes näyttäydy tietokoneen käyttäjälle. Käyttöjärjestelmien vastuulla on, ettei virheellinen tietokoneohjelma riko koko järjestelmää ja että toisaalta käyttäjän hiiren ja näppäimistön signaalit tunnistetaan ja tietokoneen näytölle välittyy kuva. Monimutkaisen käyttöjärjestelmän ytimen kehittäminen on monimutkainen ja hidas prosessi, jossa virheiden vaikutukset ovat suuria. [1, ss. 1–2]

Käyttöjärjestelmien kehitys on jo useamman vuosikymmenen keskittynyt lähinnä C- ja C++ ohjelmointikielien ympärille. Nämä kielet ovat sopineet tehtävään tarjoamillaan tehokkailla muistinhallinnan ja rinnakkaisuuden työkaluilla sekä tarjoamalla monia ylemmän tason ohjelmointikieliä paremmat mahdollisuudet vaikuttaa tietokoneen osien täsmälliseen toimintaan.

C- ja C++-ohjelmointikielien tarjoaman hallinnan myötä seurauksena tulee korkea alttius virheille, joiden jäljittäminen on usein vaikeaa ja vaikutukset arvaamattomia [2]. Ohjelmointikielien kehityksen myötä käyttöjärjestelmän ytimen kehitykseen on vanhojen ohjelmointikielien rinnalle noussut uusia vaihtoehtoja, joista osa tarjoaa uudenlaista, ohjelmointikielen ominaisuuksiin pohjaavaa turvaa hankalilta virheiltä mahdollistaen samaan aikaan tehokkaan toiminnan. Yksi tällainen ohjelmointikieli on Rust, joka on hiljalleen ottanut jalansijaa käyttöjärjestelmien kehityksen piirissä ja johon tässä työssä perehdytään.

Tässä, kirjallisuuskatsauksena tehdyssä työssä käsitellään Rust-ohjelmointikieltä käyttöjärjestelmäytimen ohjelmoinnin näkökulmasta. Työn toisessa luvussa perehdytään ensin käyttöjärjestelmiin yleisesti, käyttöjärjestelmien ytimiin ja niiden kehitykseen. Kolmannessa luvussa käsitellään Rust-ohjelmointikieltä ja sen ominaisuuksia ytimien ohjelmoinnissa. Neljännessä luvussa otetaan vielä katsanto Rustin toimintaan Linux-ytimessä, jonne Rust tuki lisättiin vain vähän ennen tämän työn kirjoittamista. Viimeiseksi luvussa kuusi kootaan yhteen työn lopputulokset ja pohdinnat.

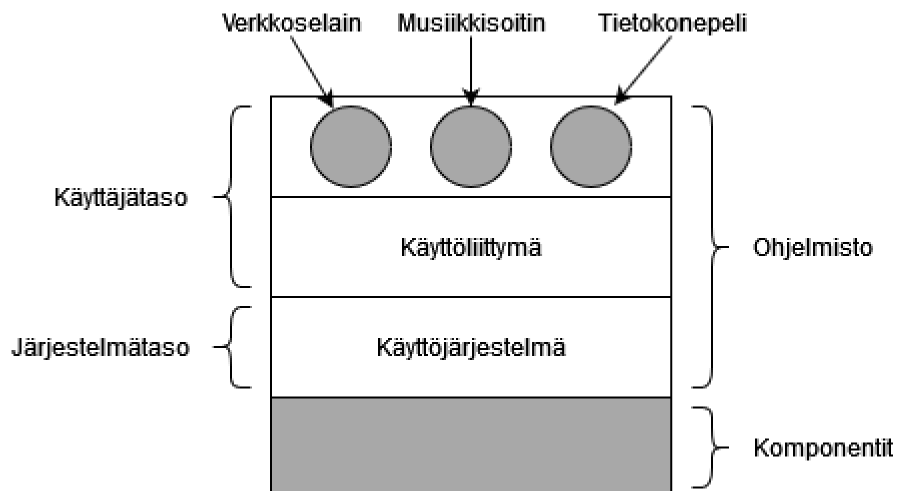
2. KÄYTTÖJÄRJESTELMÄT

Tietokoneet koostuvat sisuksiltaan erilaisista fyysisistä komponenteista, kuten prosessorista, muistiyksiköstä, grafiikkapiireistä, näytöistä sekä hiirestä ja näppäimistöstä, jotka saavat lopulta aikaan kaiken, mitä tietokoneella suoritetaan. Nämä komponentit ovat monimutkaisia ja vaativat tarkkaa tietoa ja osaamista oikeelliseen käyttöön, jotta halutut prosessit saataisiin suoritettua loppuun saakka ilman virheitä. Jotta oikeellinen käyttö olisi mahdollista, myös ilman syvää tuntemusta jokaisesta komponentista, on tietokoneisiin ohjelmoitu erillinen käyttöjärjestelmä. Käyttöjärjestelmän tehtävänä on ohjata tietokoneen komponentteja oikein ja muodostaa tietokoneella käytettäville ohjelmille abstraktio, jotta ohjelmien ei tarvitse huolehtia komponenttien hallinnasta. Kun käyttöjärjestelmä huolehtii komponenttien oikeellisesta käskytyksestä, ei varsinaisten tietokoneohjelmien ole mahdollista aiheuttaa komponenttien väärästä ohjauksesta johtuvia virheitä. [1, ss. 1–2]

Tässä luvussa esitellään seuraavaksi tietokoneiden käyttöjärjestelmän ja sen ytimen tehtäviä sekä katsotaan ydinten kehittämisen erityispiirteitä.

2.1 Vastuut

Kuvassa 1 on kuvattuna yksinkertaistettu tietokoneen rakenne, jossa nähdään, miten käyttöjärjestelmä asettuu käytettävien ohjelmien ja tietokoneen komponenttien välille.



Kuva 1. Käyttöjärjestelmän sijainti tietokoneen hierarkiassa, mukaillen lähteestä [1, s. 2].

Tanenbaum [1, s. 2] jakaa, kuvassa näkyvällä tavalla, tietokoneen ohjelmiston kahteen tasoon: *käyttäjätasoon* (engl. *user mode*) ja *järjestelmätasoon* (engl. *kernel mode*). Käyttäjätasolla ovat tietokoneella käytettävät ohjelmat sekä tietokoneen käyttämiseen tarkoitettu käyttöliittymä, kun käyttöjärjestelmä sijaitsee järjestelmätasolla. Jaon pääperiaatteena on, että käyttäjätasolla suoritettavilla toiminnoilla on pääsy ainoastaan pieneen osaan mahdollisista komennoista, kun järjestelmätasolla käyttöjärjestelmällä on pääsy kaikkiin tietokoneen komponentteihin ja sen on mahdollista suorittaa mitä tahansa toimintoja ja käskyjä. [1, ss. 1–2]

Käyttöjärjestelmien ydintehtävät ovat aina samat, mutta niiden koko ja rakenne vaihtelee käyttökohteen komponenttien ja vaatimusten mukaan. Vaatimukset käyttöjärjestelmältä vaihtelevat paljon esimerkiksi henkilökohtaisella käyttöön tarkoitetulla tietokoneen, mobiililaitteen, serverin ja sulautetun järjestelmän, kuten vaikka yksinkertaisella kodinkoneen, välillä. [1, ss. 35–38] Siinä missä sulautettujen järjestelmien ei koskaan tarvitse olettaa suorittavan kolmannen osapuolen epäluotettavia ohjelmia, on vaikkapa älypuhelimen varauduttava kaikenlaiseen käyttöön ja huolehdittava sen mukaisesta suojauksesta ja varmistuksesta. Myös esimerkiksi käytettävän muistin ja erilaisten komponenttien ja prosessoriytimien määrällä on vaikutus käyttöjärjestelmän kokoon ja rakenteeseen. [1, ss. 35–38]

2.2 Käyttöjärjestelmän ydin

Käyttöjärjestelmän ydin on ohjelma, jonka tehtävänä on huolehtia tietokoneen komponenttien oikeellisesta käskyttämisestä ja tarjota muille tietokoneella käytettäville ohjelmille turvalliset työkalut tarvittavien prosessien suorittamiseksi. Käyttöjärjestelmäytimet ovat valtavan suuria ja kompleksisia ohjelmia. Käyttöjärjestelmäytimien tulee ymmärtää tietokoneen erilaisiin sisääntuloihin, kuten *Peripheral Component Interconnect* (PCI), *Peripheral Component Interconnect Express* - (PCIe), PCI, U*niversal Serial Bus* - (USB) ja S*erial AT Attachment* (SATA) -portteihin kiinnittyviä komponentteja [1, s. 32]. Niiden tulee osata vuorottaa prosessorilla suoritettavat komennot tehokkaasti ja varmasti, huolehtia muistin varaamisesta, lukemisesta ja kirjoittamisesta, sekä muista tietokoneen resurssien hallintaan liittyvistä asioista, sekä yhdistää kaikki nämä toimivaksi, virheettömäksi, järjestelmäksi.

Kaksi yleisintä käyttöjärjestelmäytimen tyyppiä ovat monoliittinen ydin ja mikroydin. Monoliittinen ydin on ylivoimaisesti yleisin käyttöjärjestelmäydinten tyyppi. Se on rakenteeltaan yksi iso ohjelma, joka kattaa kaikki käyttöjärjestelmän palvelut yhtenä isona kokonaisuutena. Monoliittinen ydin on parhaimmillaan tehokas ja sujuva. Monoliittisessa ytimessä jokainen prosessi voi keskustella toinen toisensa kanssa ja kaikki toiminnot ovat

vapaasti saatavilla, eivätkä piilotettuina erillisten moduulien ja abstraktioiden taakse. [1, ss. 62–68]

Laaja ja monimutkainen monoliittinen ydin voi olla kuitenkin vaikeasti ymmärrettävä ja hallittava sekä altis virheille. Monoliittisessa ytimessä prosessin kaatumisen aiheuttava virhe yhdessä kohdassa ydintä aiheuttaa koko käyttöjärjestelmäytimen kaatumisen. [1, s. 63] Osittain tätä haavoittuvuutta lievittämään on kehitetty käyttöjärjestelmäydintyyppi mikroydin.

Monoliittisen ytimen yhtenäisen, yhden ison ohjelman rakenteen sijaan, mikroytimessä rakenne on hajautettu useisiin eri moduuleihin. Hajautetussa rakenteessa vain yksi, mikroytimeksi kutsuttu, moduuli toimii riskialttiissa järjestelmätasossa ja loput ytimen toiminnoista on eristetty omiin moduuleihinsa, tavanomaisten käyttäjätason prosessien tapaan. Tällaisessa rakenteessa virhe yhdessä moduulissa voi kaataa sen kyseisen ytimen komponentin, mutta ei koko käyttöjärjestelmää. Mikroytimet ovat erityisen hyödyllisiä käyttökohteissa, joissa järjestelmän luotettavuus on tärkeää. [1, s. 66]

2.3 Ydinten kehitys

Käyttöjärjestelmien ydinten kehittäminen eroaa tavanomaisesta ohjelmistokehityksestä, erityisesti ohjelmoitavan ympäristön herkkyyden ja sen laajuuden ja monimutkaisuuden osalta [3]. Ohjelmointivirheiden vaikutus käyttöjärjestelmäytimessä voi olla suuri koko järjestelmälle, minkä vuoksi sen kehittämisessä on oltava erityisen huolellinen. Käyttöjärjestelmäydinten kehittäminen vaatii käytettävältä ohjelmointikieleltä myös enemmän. Järjestelmätasolla tehokkuus ja tarkkuus on ovat tärkeitä seikkoja. Tarvittava hallinnan määrä on suuri ja käytettävällä ohjelmointikielellä on pystyttävä tarkkaan muistinhallintaan, rinnakkaisten prosessien kontrollointiin ja komponenttien kanssa keskusteluun, samalla vaatien järjestelmältä mahdollisimman vähän resursseja omaan toimintaansa. Käyttöjärjestelmäydinten kehitystä onkin jo pitkään hallinnut erityisesti ohjelmointikieli C, jolla suurin osa yleisimmistä käyttöjärjestelmistä on ohjelmoitu [4].

C:n tehokkuus ja sen tarjoamat mahdollisuudet erityisesti muistinhallinnan osalta sopivat hyvin käyttöjärjestelmäydinten tarpeisiin, ja se alun perin kehitettiinkin juuri helpottamaan käyttöjärjestelmien kehittämistä 1970-luvulla, jolloin käyttöjärjestelmien kehitykseen oli käytössä vain nykystandardeilla vaativa ja monimutkainen Assembly-kieli [5, Luku 1]. Sittemmin lukuisia ohjelmointikieliä on kehitetty yhä kehittyneimmillä syntakseilla, mutta vain harva niistä on toiminut järjestelmätason kehittämiseen.

Vaatimukset matalaan ajonaikaiseen rasitukseen tekevät esimerkiksi monissa ohjelmointikielissä käytettävän *roskienkerääjän* (engl. *garbage collector*) sopimattomaksi järjestelmätason ohjelmistokehitykseen. Roskienkerääjä on muistiturvallisuuteen tähtäävä järjestelmä, joka määräajoin tehtävien tarkistusten avulla vapauttaa varatun muistin automaattisesti, jos ohjelmassa ei ole enää siihen viittaavia muuttujia [6, s. 156]. Roskienkerääjän kaltainen järjestelmä ei kuitenkaan ole sopiva järjestelmätason ohjelmistokehitykseen, koska sen tuoman ajonaikaiset tarkistukset eivät ole mahdollisia järjestelmätasolla. Tarkistukset vaativat tietyissä tilanteissa koko muun ohjelman suorituksen keskeyttämistä, mikä ei ole mahdollista esimerkiksi tietokoneen käyttöjärjestelmän tapauksessa.

Muistinhallinta on olennainen osa tehokkaan järjestelmätason ohjelmakoodin kirjoittamisessa, mutta suurten ohjelmien muistiturvallisuudesta ei ole aina helppoa huolehtia. Yleisesti käytettyjen C- ja C++-kielien tapauksessa muistiturvallisuudesta huolehtiminen lankeaa viime kädessä ohjelmakoodin kirjoittajan tehtäväksi, eikä siitä huolehtimiseksi ole erityistä automatiikkaa. Varsinkin tällaisella ohjelmointikielellä kirjoitetulla ohjelmointikielellä muistinhallinnan virheet ovat yleisiä. 2019 laaditussa Microsoft Security Centerin raportissa kerrotaan noin 70 %:n Microsoftin korjaamista turvallisuusuhista johtuvan muistiturvallisuuden ongelmista [7]. Paremmalla ratkaisulla muistiturvallisuuden hallitsemiseksi voitaisiinkin saavuttaa huomattavaa parannusta käyttöjärjestelmäytimien turvallisuudessa ja seuraavassa luvussa tutustutaankin yhteen mahdolliseen ratkaisuun.

3. RUST-OHJELMOINTIKIELI

Rust on Graydon Hoaren vuonna 2006 kehittämä tehokkaaseen ja turvalliseen ohjelmistokehitykseen suunniteltu ohjelmointikieli, joka sittemmin on siirtynyt Mozilla yhtiön kehitykseen avoimen lähdekoodin kehittäjien yhteistyön kanssa. Ajatus Rustin takana on ratkaista perinteiseen C/C++-voittoiseen, tehokkuuteen tavoittelevan ohjelmistokehitykseen liittyviä ongelmia, tarjoamalla sisäänrakennettua muistiturvallisuutta, oikeellisuuteen ja turvallisuuteen tähtäävät rinnakkaisuuden menetelmät, sekä matalan ajonaikaisen rasitteen. Vaikka Rustin kehittämisen pääajatus on matalamman tason tehokkuuskeskeisessä ohjelmoinnissa, on se helppokäyttöisen syntaksinsa puolesta sopiva käytettäväksi esimerkiksi verkkoapplikaatioiden ja -palveluiden kehitykseen. [6, ss. 8–10]

Rustin kehitys ohjautuu vahvasti kehittäjäyhteisönsä toiveiden mukaisesti. Rustin lähdekoodi on avoimesti saatavilla ja kenen tahansa on mahdollista ehdottaa ohjelmointikieleen lisättäviä ominaisuuksia erityisen *Request For Comments* (RFC) -ehdotusjärjestelmän kautta, jonka välityksellä muotoutuneet hyväksytyt ominaisuudet lopulta lisätään kielen kuuden viikon välein päivittyvään versioon.

Iso osa Rustin kirjastoista ja moduuleista toimii Cargon, eli Rustin paketinhallinnan kautta. Cargon kautta Rustilla laadittaviin ohjelmiin on mahdollista helposti lisätä muiden ohjelmistokehittäjien laatimia ulkopuolisia kirjastoja. Työn kirjoitushetkellä Cargon kautta on ladattavissa hieman alle 110 000 erilaista moduulia, eli *cratea* [8].

Rustin omistautunut käyttäjäyhteisö on myös yksi sen erityispiirteistä. Vuosittain koostettu suositun Stack Overflow -verkkosivuston vuoden 2022 kyselyssä Rust listautui seitsemättä kertaa parhaaksi kyselyssä käyttäjiensä tyytyväisyydestä käyttämäänsä ohjelmointikieleen, sekä jaetulle ensimmäiselle sijalle kielenä, jota kehittäjät eivät vielä käytä, mutta ovat kiinnostuneita tutustumaan [9]. Rust on silti käyttäjäkunnaltaan pieni kieli. Saman Stack Overflow -kyselyn mukaan 9,32 % kaikista ohjelmistokehittäjistä ja 8,80 % ammattikehittäjistä käyttää Rustia, kun esimerkiksi C-ohjelmointikielellä vastaavat luvut ovat 19,27 % ja 16,70 % ja C++:n 22,55 % ja 20,17 % [9]. Sharma [6, s. 14] tiivistää, että Rustista tulisi kiinnostua, jos tavoitteena on kirjoittaa tehokaita ohjelmia, vähemmällä määrällä ohjelmointivirheitä, samalla nauttien useista modernin ohjelmointikielen ominaisuuksista ja loistavasta kehittäjäyhteisöstä.

Tässä luvussa esitellään erityisesti järjestelmätason kehityksen kannalta oleellisia Rustin ominaisuuksia ja arvioidaan sen sopivuutta käyttöjärjestelmien kehittämiseen, pitäen vertailukohtana järjestelmätason ohjelmistokehitystä hallitsevaa C-ohjelmointikieltä.

3.1 Erityispiirteet

Matalan tason tehokkuutta tavoittelevaa ohjelmistokehitystä silmällä pitäen suunniteltu Rust on erityinen sen tarjoaman muistiturvallisuuden, tehokkuuden ja rinnakkaisuuden yhdistelmän vuoksi. Rust on alkujaan ottanut vaikutteita monista ohjelmointikielistä, kuten muistiturvallisuuden osalta Cyclonesta, Haskellista sen tyyppijärjestelmästä ja C++:sta resurssien varaamisen ja hallinnoinnin osalta. Rust on toisaalta lähellä C++-ohjelmointikieltä sen hyvien, tehokkuuteen tähtäävien työkalujen myötä, mutta kuitenkin nauttii useiden korkeamman tason kielien eduista, kuten automaattisesta muistiturvallisuudesta ja selkeästä ja pitkälle kehitetystä kääntäjästä.

Tutustutaan tässä luvussa seuraavaksi tarkemmin Rustin muistinhallintaan, rinnakkaisuuden välineisiin, sekä hieman sen tyypitysjärjestelmään.

3.1.1 Muistiturvallisuus ilman roskienkerääjää

Kuten kappaleessa 2.3 kerrotaan, yleinen tapa ratkaista muistiturvallisuus ohjelmointikielissä on käyttää roskienkerääjää, joka ei kuitenkaan käyttöjärjestelmäytimien tapauksessa ole toimiva ratkaisu. Rustissa muistiturvallisuus saavutetaan sen sijaan ohjelmakoodin käännösvaiheessa tehtävillä tarkistuksilla. Rustin muistiturvallisuuteen liittyvät oleellisesti Rustin käsitteet *omistajuus* (engl. *ownership*), *lainaaminen* engl. *borrowing*) ja *elinaika* (engl. *lifetime*), mitkä mahdollistavat Rustin turvallisen ja tehokkaan muistinhallinnan. [6, s. 164]

Rustissa käsite omistajuus tarkoittaa, että muuttujasta, jolle luodaan arvo tai resurssi, tulee sen resurssin omistaja. Kun resurssi tai arvo siirretään toiselle muuttujalle, siirtyy myös omistajuus uudelle muuttujalle, jolloin vanhasta muuttujasta tulee epäkelpo eikä sitä voi jatkossa enää käyttää. Rustissa muuttujat ja arvot vapautetaan muistipaikoiltaan, niiden *toiminta-alan* (engl. *scope*) sulkeutuessa. Omistajuuden myötä yhdelle arvolle voi olla vain yksi omistaja, jonka kautta arvoa voidaan käsitellä. Sen myötä esimerkiksi manuaalisessa muistinhallinnassa tapahtuvat, jo vapautettuun muuttujaan viittaamisesta aiheutuvat virheet eliminoidaan. [6, s. 165]

Lainaaminen ja elinajan käsite liittyvät omistajuuden ympärille. Lainaamisella tarkoitetaan johonkin muuttujaan viittaamista siten, että se noudattaa omistajuuden sääntöjä [6, s. 177]. Lainaamisessa resurssin omistajuus ei muutu, vaan siihen vain viitataan. Viittaamisen kohdalla myös elinaika astuu kuvioihin. Elinajalla tarkoitetaan juuri viittauksen tarpeellista olemassaolon aikaa, joka on aina kuhunkin viittaukseen sidottu tieto. Usein

Rust-koodin kääntäjä osaa asettaa viittausten elinajat automaattisesti tulkitsemalla koodia sen käännösvaiheessa, mutta toisinaan ohjelmoijan on itse määriteltävä viittauksen elinaika, silloin kun kääntäjälle ei ole selvää, miten kauan viitettä tarvitaan tai mihin sitä on tarkoitus käyttää [6, s. 183]. Tällaisessakin tapauksessa viite poistetaan varatusta muistista sen elinajan päättyessä, huolehtien ettei muistia varata tarpeettomille viittauksille, ja etteivät lainaukset aiheuta ongelmia muistiturvallisuuden suhteen.

Sharma [6] kutsuu omistajuutta, lainaamista ja elinaikaa muistiturvallisuuden kolminaisuudeksi, mitkä tekevät muistivuotojen tahattoman ohjelmoinnin vaikeaksi [6, s. 164]. Sharman näkemyksen mukaan nämä Rustin sisäänrakennetut muistiturvallisuuden turvatoimet pohjimmiltaan ohjaavat ohjelmoijia pois huonoista ohjelmointitavoista ja kohti muistiturvallisia toimintatapoja [6, s. 152].

3.1.2 Staattinen tyypitys

Rustin oleellisimpia erityispiirteitä on sen vahva staattinen tyypitysjärjestelmä [6, s. 115]. Staattisella tyypityksellä tarkoitetaan järjestelmää, jossa luontitilanteessa muuttujalle annettua tietotyyppiä ei voi muuttaa jälkikäteen. Tällainen järjestelmä parantaa osaltaan koko järjestelmän muistiturvallisuutta, esimerkiksi huolehtimalla, että määriteltyjä arvoja ei päästä käsittelemään muistialueidensa ulkopuolelta virhetilanteessa, johon esimerkiksi tietoturvauhan luovat puskurin ylivuotohyökkäykset pohjautuvat [10]. Staattinen tyypitys vaatii toisaalta ohjelmoijalta tarkempaa suunnittelua, esimerkiksi tilanteissa, joissa funktion haluttaisin hyväksyvän useampaa eri tietotyyppiä. Toisaalta se lisää ohjelmakoodin turvallisuutta ilman suorituksen aikaista rasitusta, kun tietotyyppitarkistukset tehdään koodin käännösvaiheessa, eikä ajon aikana.

Rustin tyyppijärjestelmä on ottanut vaikutteita erityisesti funktionaalisista Ocaml- ja Haskell-ohjelmointikielistä. Staattisen tyypitysjärjestelmän suunnittelu vaatii rajan vetoa käytettävissä olevien tyyppien määrän suhteen. Vähäinen määrä tyyppejä yksinkertaistaa ohjelmien toteutusta, mutta rajoittaa ohjelmoijan mahdollisuuksia, kun taas runsas määrä eri tyyppejä lisää ohjelmointikielen ilmaisukykyä, mutta monimutkaistaa ohjelmointia ja ohjelmointikielen toteutusta. Rustin tapauksessa sisäänrakennettuja tyyppejä on enemmän kuin vaikkapa C-ohjelmointikielessä, mutta vähemmän kuin ilmaisukyvystään tunnetulla Haskellilla. [6, s. 115].

Rust tarjoaa staattisen tyypityksen tueksi työkaluja erilaisten tyypitykseen liittyvien ongelmien ratkaisemiseksi. Rustissa on mahdollista määritellä geneerisiä muuttujia (engl. *generics*) funktioille ja rakenteille, helpottaen useamman tietotyypin käsittelyä ilman toisteisen koodin kirjoittamista. Rustissa on myös mahdollista määritellä tyypeille erityisiä *piirteitä* (enlg. *traits*), joita tyypeiltä voidaan odottaa. Piirteiden avulla eri tyypeille voidaan

määritellä samoja toiminnallisuuksia. Esimerkiksi erilaisilla ajoneuvoluokilla voisi olla kaikilla yhteinen piirre ”ajettava”, johon voisi sisältyä vaikkapa kaikille ajoneuvoille yhteiset metodit ”kiihdytä” ja ”jarruta”. Piirteet toimivat monien ohjelmointikielen ominaisuuksien, kuten älykkäiden osoittimien, silmukoiden ja käännösvaiheen tarkistusten perustana. [6, ss. 123–126]

Vahva tyypitys, geneeriset muuttujat ja piirteet rakentavat kaikki yhdessä Rustin toiminnan turvallisuutta ja antavat ohjelmoijalle hyvän mahdollisuuden tarkkaan ohjelmakoodin toiminnan määrittelyyn.

3.1.3 Rinnakkaisuuden menetelmät

Sharman [6] sanoin, Rust tarjoaa ”pelotonta rinnakkaisuutta”. Rust tarjoaa muiden matalan tason kielien tapaan käyttäjilleen tavanomaiset työkalut tarkkaan rinnakkaisuuden toteuttamiseen, mutta vähentää rinnakkaisuuteen tavallisesti liitettyjä haasteita, yhdistämällä aiemmin esiteltyjä omistajuuden ja piirteiden periaatteita myös useamman säikeen kanssa tehtäviin suorituksiin. Rustin omistajuus estää esimerkiksi *irrallisten osoittimien* (engl. *dangling pointers*) muodostumisen, estämällä periytettyjä säikeitä poistamasta dataa. Rustissa on myös mahdollista varmistaa tyyppien säieturvallinen käyttö, hyödyntämällä Send- ja Sync-piirteitä, joilla eri tyypit voidaan varmistaa turvallisiksi, merkitsemällä ne Send-piirteellä turvallisiksi lähettää useille säikeille yhtä aikaa ja Sync-piirteellä, että niitä on turvallista referoida useammasta säikeestä. [6, ss. 312–317]

Rustin standardikirjasto tarjoaa käyttäjälle kuitenkin vain perustyökalut rinnakkaiseen ohjelmointiin. Edistyneemmät rinnakkaisuuden menetelmät löytyvät Rustin ulkoisista kirjastoista, kuten Rayon ja Tokio, jotka tarjoavat käyttäjille yksinkertaisia ja tehokkaita menetelmiä asynkronisuuteen ja muihin rinnakkaisuuden eri menetelmiin [11].

3.2 Kritiikkiä

Rustin intohimoinen yhteisö ja menestys käyttäjäkokemuskyselyissä antaa kielestä helposti yksinomaan positiivisen kuvan, mikä ei kaikilta osin ole täysin totta. Kuten kaikilla ohjelmointikielillä, on Rustillakin heikot kohtansa. Rustin kääntäjä on kilpailijoitaan hitaampi [12, Luku 1.8]. Rustin kääntäjä käy ohjelmakoodin käännösvaiheessa läpi monimutkaisen käännösprosessin, lukuisine tarkastuksineen ja moduuleineen, mikä vaikuttaa ohjelmakoodin käännösaikaan. Vaikka sanotaan, ettei Rustin turvallisuus näy sen tehokkuudessa, on sillä vaikutuksensa koodin käännösaikaan.

Muu Rustin saama kritiikki liittyy pitkälti kielen tarkkoihin sääntöihin ja niistä johtuviin hankaluuksiin. Rust on laaja ja monipuolinen kieli, mikä yhdistettynä tarkkoihin sääntöihin ja huolellisuutta vaativaan ohjelmakoodin kirjoitukseen, tekee siitä hitaan ja vaikean

kielen opetella. Rustin turvallisuutta luovat tarkistukset ja säännöt hankaloittavat myös erityisesti syklisten datarakenteiden luomista. Tällaisten rakenteiden toteuttaminen toisilla ohjelmointikielillä on melko triviaali tehtävä, mutta Rustin omistajuus ja viittaussäännöt tekevät työstä vaikean kokemattomalle Rustin käyttäjälle. [12, Luku 1.8]

4. CASE: LINUX-YTIMEN OHJELMOINTI RUSTILLA

Linux on Unix-pohjainen avoimen lähdekoodin käyttöjärjestelmä, joka on tunnettu erityisesti muokattavuudestaan ja matalista järjestelmävaatimuksistaan, mitkä ovat tehneet siitä erityisen suositun esimerkiksi palvelinkäytössä. Linux-käyttöjärjestelmästä onkin lukuisia eri versioita erilaisilla konfiguraatioilla, jotka kaikki rakentuvat saman Linux-käyttöjärjestelmäytimen ympärille. Kuten valtaosa merkittävimmistä käyttöjärjestelmien ytimistä, on myös Linux-ydin kirjoitettu valtaosin C-ohjelmointikielellä ja sen luomishetkestä lähtien, C on ollut käytännössä ainoa ohjelmointikieli, jolla avoimesti kehitettävää Linux-ydintä on ollut mahdollista ohjelmoida ja kehittää, kunnes 11.12.2022 julkaistussa Linux Kernel 6.1 päivityksessä Linux-käyttöjärjestelmäytimeen lisättiin virallinen tuki Rust-ohjelmointikielelle [13]. Vaikka Rust-tuki Linux-ytimessä on vielä kokeiluasteella, odotetaan sen muisti- ja säieturvallisuutta varmistavien turvatoimien tuovan helpotusta käyttöjärjestelmäytimen kehitystyöhön ja vähentävän kalliita ja hankalasti todennettavia ohjelmointivirheitä, tinkimättä käyttöjärjestelmäytimelle oleellisesta tehokkuudesta [14].

Tarkastellaan tässä työn luvussa hieman, miten Rust on tuotu Linux-käyttöjärjestelmäytimeen mukaan, sekä otetaan lyhyt katsanto, mitä Rustilla on tähän mennessä Linux-ytimessä toteutettu, sekä mietitään mahdollisia tulevaisuuden suuntia Rustin ja Linux-ytimen osalta.

4.1 Toiminta

Rustin käytön Linux-ytimen kehittämisessä on mahdollistanut huhtikuussa 2021 aloitettu Rust for Linux -projekti, jonka tavoitteena on saada Rust-tuki Linux-ytimeen [15]. Projektin myötä Rust saikin virallisen tuen Linux-käyttöjärjestelmäytimeen ytimen 6.1 päivityksessä, jossa julkaistiin tuen toteutuksen ensimmäinen, hyvin alustava virallinen versio [13].

Rust toimii Linux-ytimen olemassa olevan C-ohjelmakoodin kanssa erilaisten käärimien (wrapper) avulla, jotka toimivat abstraktiona ytimen ja Rust-ohjelmakoodin välillä ja siten mahdollistavat ytimen funktioiden käytön Rustilla [16]. Vaatimukset käärimille riippuvat siitä, mitä ytimen ominaisuuksia kehitettävällä ajurilla, moduulilla tai ohjelmalla tarvitaan ja varsinkin Rustin ollessa vielä uusi kieli Linux-ytimessä, ei valmiita käärimiä aina löydy.

Rustin suurin rajoite Linux-ytimessä liittyy sen kääntäjään. Rust-tuen ensimmäisessä julkaistussa versiossa vain tietty versio Rust-kääntäjä rustc:stä toimii Linux-ytimen kanssa ja on käännettävä käyttäen *Low level Virtual machine* (LLVM) -käännöstyökaluja ja C- ja C++-ohjelmointikielille suunnattua Clang:ia, mikä ei ole tavanomainen tapa Linux-ytimen kääntämisessä [17]. Tavallisesti Linux-ytimen kanssa käytetään *GNU's Not Linux* (GNU) -työkalusarjoja. LLVM:n käyttö rajoittaa myös erilaisten tuettujen suoritinarkkitehtuurien määrää [17]. Tilanne rajoitusten suhteen tulee todennäköisesti parantumaan Rust-tuen kehityksen edetessä.

4.2 Toteutetut Linux-käyttöjärjestelmäytimen moduulit

Vaikka Linux-ytimen Rust tuki on työn kirjoitushetkellä vielä verrattain uusi asia, on varhaisimpia Rustilla toteutettuja moduuleita jo raportoitu. Esitellään seuraavaksi lyhyesti kaksi Linux-käyttöjärjestelmäytimelle toteutettua tai toteutuksessa olevaa ajuria.

4.2.1 NVMe Ajuri

Ensimmäisiä Rustilla toteutettuja Linux-ytimen moduuleja on syyskuussa 2022 Linux Plumbers Conferencessäkin esitelty *Non-Volatile Memory Host Controller Interface Specification* (NVMe) -ajuri [18]. Ajuri oli ensimmäisiä monimutkaisempia Rustilla toteutettuja Linux-käyttöjärjestelmäytimen ajureita ja on toiminut esimerkkinä Rustin toiminnasta Linux-ytimessä. Ajuria ei toteutettu niinkään puhtaasta tarpeesta, vaan esittelemään Rustin toimintaa Linux-ytimessä. Moduuli toteutettiin jo ennen virallisen Rust-tuen julkistamista ja se luotiin todistamaan, että Rustilla on mahdollista saavuttaa C:llä toteutettujen ajurien vertaista tehokkuutta. Rust-ajuria ja C:llä koodattua NVMe-ajuria verratessa ajurien tehokkuus on hyvin lähellä toisiaan, C:llä toteutetun ajurin päihittäessä Rust-ajurin datan lukutehossa parhaimmillaan vain noin 6 % verran. Tehoeron aiheutti Rust-ajurin C-ajuria korkeampi ajonaikainen resurssien tarve. [18] Vertailu osoittaakin hyvin, että jo verrattain lyhyellä kehityksellä tuotetut Rust-ajurit voivat saavuttaa kilpailukykyistä tehokkuutta muiden Rustin mukanaan tuomien etujen ohella. Rust-ajurin kehittäjä oli esittänyt myös selkeitä parannuskohtia kehittämänsä ajurin toteutuksesta, mutta ei toistaiseksi suosittele ajuria yleiseen käyttöön sen keskeneräisessä tilassa.

4.2.2 M1 GPU ajuri

Toinen Rustilla toteutettu Linux moduuli on Applen Silicon-suorittimellisiin Mac-tietokoneisiin keskittyvälle Asahi Linux versiolle Linux-käyttöjärjestelmästä toteutettu M1 prosessorin graafisen suorittimen (GPU) ajuri [19]. Asahi Linuxin blogissa Linux-kehittäjä Asahi Lina kertoo järjestelmä- ja käyttäjäavaruuden ajureista koostuva M1 GPU-ajurin kehitysprosessista.

Rust liittyi projektiin varsinaisen Linux-käyttöjärjestelmäydinajurin kehitysvaiheessa. Huhut Rust-tuen lisäämisestä Linux-ytimeen saivat kehittäjä Linan kiinnostumaan sen tarjoamista mahdollisuuksista GPU-ajurin kehityksessä ja lyhyiden ohjelmointikielen testien, sekä Rust-asiantuntijoiden konsultaation jälkeen Lina päätti toteuttaa GPU-ajurin järjestelmätason osan Rustilla. [19]

Rustia ei projektin aikana ollut vielä liitetty virallisesti Linuxiin ja vaikka Rust for Linux -projektin tekemä kehitys kielen tuen eteen oli pitkällä, tarkoitti oletusarvoisesta ohjelmointikieli C:stä poikkeaminen paljon ylimääräistä työtä. Lina kertoo joutuneensa kirjoittamaan itse ajurin lisäksi myös paljon muuta ohjelmakoodia, saadakseen Rust-ajurinsa toimimaan muun Linux-ytimen kanssa. [19]

Lina kuitenkin sanoo pitäneensä paljon Rustin kanssa työskentelystä. Hän kehui erityisesti sen hyvään ohjelmointityyliin ohjaavaa tarkkaa kääntäjää, joka oli hänenkin kohdallaan paljastanut virheitä Linan ohjelmiston suunnittelusta, mitkä olisivat muuten voineet jäädä huomaamatta. Tavallisesti uuden järjestelmätason GPU-ajurin kanssa siirryttäessä yksinkertaisista demoista monimutkaisempaan, useampia ohjelmia yhtäaikaisesti käyttävään työpöytäympäristöön, tulee vastaan lukuisia virheitä rinnakkaisuuden, muistivuotojen ja muiden ongelmien muodossa, mutta Lina kertoo Rust-ajurin olleen lähes täysin vapaa kaikista sellaisista ongelmista. [19]

Ajurin alpha-versio julkaistiin Asahi Linuxin käyttäjille joulukuussa 2022 ja sen kehitys jatkuu yhä työn kirjoitushetkellä. Viimeisimpien päivitysten myötä avoimen kehityksen Rustilla toteutetulla M1 GPU-ajurilla on saavutettu jopa parempaa suorituskykyä, kuin Applen MacOS-käyttöjärjestelmän omalla ajurilla. Parempi suorituskyky ei ole pelkästään käytetyn ohjelmointikielen ansiota, mutta on hyvä merkki myös Rustilla toteutettujen Linux-ytimen ajureiden toimivuudesta.

4.3 Linux-ytimen tulevaisuus

Molemmat esimerkit Rustilla toteutetuista Linux-ytimen laajennuksista antavat lupaavia tuloksia ohjelmointikielen sopivuudesta Linux-käyttöjärjestelmäytimen kehittämiseen. Esimerkeissä toteutettujen ajureiden tehokkuudet olivat jo lyhyen kehitystyön jälkeen kilpailukykyisiä vastaavien, toisilla kielillä toteutettujen ajureiden kanssa. Rustilla toteutetun M1 GPU -ajurin kehittäjä Asahi Lina [19] kertoo blogikirjoituksessaan, kappaleessa ”Rust is magical!” useista positiivisista kokemuksistaan ytimen kehittämisestä Rustilla ja sanoo muun muassa kuinka kaikista oudoista ja virhealttiista C-ytimen ohjelmointikaavoista

muuntuvat kauniiksi Rustissa. Rust vaikuttaa jättäneen hyvän vaikutelman käytettävyydestään Linux-ytimen kehitykseen jo sen lyhyen integraation aikana ja tulee varmasti saamaan lisää käyttöä kasvavassa määrin.

Linuxin ydin on yli kolmen kymmenen vuoden kehityksensä aikana muodostunut valtaisaksi ja monimutkaiseksi kokonaisuudeksi, eikä Rust-tuen lisääminen sen ytimeen tarkoita aikeista uudelleenkirjoittaa koko ydintä uudella kielellä. Linux-ydin on kuitenkin vuosikymmenten kehityksen aikana muotoutunut eheäksi ja tehokkaaksi kokonaisuudeksi, mistä virhealttiimman C-ohjelmointikielen tuomat ongelmat on korjattu huolellisella ohjelmointityylillä, testauksella ja iteroinnilla. Toisin sanoen ongelmat, joita Rustin on tarkoitus ratkaista, on suurilta osin ratkottu olemassa olevan ytimen ohjelmakoodin kohdalla. On varmasti osa-alueita, joita voitaisiin parantaa Rustin avulla, mutta suurinta osaa ohjelmakoodista tuskin uudelleen kirjoitetaan. Tulevan kehityksen kohdalla Rustista on varmasti hyötyä ja uskon sen helpottavan Linux-ytimen kehitystyötä huomattavasti.

Rust for Linux -projektin aloituksesta kertovassa viestissä, Linux-ytimen kehittäjien viestikanavalla, Rust-tuen tavoitteiksi listataan pienempi riski muisti- ja logiikkavirheille, suurempi varmuus uusien päivitysten luotettavuudesta, uusien kehittäjien houkutteleminen modernilla ohjelmointikielellä, uusien moduulien ja ajurien kehityksen helpottuminen ja dokumentaation sääntöjen noudattamisen helpottuminen ohjelmointikielen ominaisuuksien avulla [15]. Aikaisemmin esiteltyjen projektien antaman vaikutelman perusteella tavoitteet vaikuttavat hyvin saavutettavilta ja monilta osin toteutuneilta jo Rust-tuen varhaisessa vaiheessa. Uskon, että kehityksen jatkuessa Rustin asemaa Linux-ytimen kehityksessä vakiintuu ja tulee vain kasvamaan. En ajattele, että C-ohjelmointikielen käyttö tulee poistumaan uusienkaan Linux-ytimen moduulien ja ajurien kehityksessä, mutta ajan myötä uskon Rustin nousevan ainakin varteenotettavaksi vaihtoehdoksi, jos ei hallitsevaksi ohjelmointikieleksi Linux-ytimen kehityksessä.

5. YHTEENVETO

Käyttöjärjestelmien ydinten kehittäminen vaatii käytettävältä ohjelmointikieleltä paljon. Yhtä aikaa kehittäjän tulisi olla mahdollista kontrolloida tarkasti ohjelman muistinhallintaa, rinnakkaisuutta ja logiikkaa, aiheuttaen mahdollisimman vähän rasitusta järjestelmälle ja samalla huolehtia, ettei herkästi isoja ongelmia aiheuttavia ohjelmointivirheitä tapahdu näillä osa-alueilla. Hyvin pitkään ydinten kehitys on ohjelmointikielen suhteen ollut kompromissi, jossa painopiste on ollut mahdollisimman tehokkaassa toiminnassa, ohjelmakoodin turvallisuuden kustannuksella. Uudet ohjelmointikielet ovat kuitenkin vähentäneet tarvetta tinkiä ohjelmakoodin turvallisuudesta, esittelemällä ohjelmointikieleen sisäänrakennettuja käännösvaiheen automaattitarkastuksia, joiden avulla ohjelmakoodin on mahdollista saavuttaa sekä korkea tehokkuus, että hyvä turvallisuuden taso.

Tässä työssä tutkittiin tällaisen modernin ohjelmointikieli Rustin sopivuutta käyttöjärjestelmien ydinten kehitykseen ja todettiin sen sopivan siihen hyvin. Työssä tutustuttiin aluksi yleisesti käyttöjärjestelmien ytimiin ja niiden kehityksen erityispiirteisiin, jonka jälkeen paneuduttiin Rust-ohjelmointikielen ominaisuuksiin erityisesti käyttöjärjestelmäytimien kehityksen kannalta. Viimeisimpänä katsottiin hiljattain julkaistun Linux-käyttöjärjestelmän ytimeen lisättyä Rust-tukea, sen toimintaa ja vaikutusta, sekä ensimmäisiä Rustilla toteutettuja Linux-ytimen ajureita.

Rustin sisäänrakennetut muistiturvallisuuden ja turvallisen rinnakkaisuuden järjestelmät tuovat suurta helpotusta käyttöjärjestelmäydinten kehitykseen, missä mainitut ongelmakohdat ovat perinteisesti olleet yleinen ohjelmointivirheiden aiheuttaja. Rustin omistajuuden, elinajan ja lainaamisen periaatteet yhdistettynä sen vahvaan staattiseen tyypitysjärjestelmään tekevät muistinhallinnan virheiden pääsemisen ohjelmakoodin käännösvaiheen ohitse harvinaisiksi ja ohjelmoija voikin olla luottavainen käännetyn ohjelmakoodinsa toimivuuteen.

Rust on päässyt näyttämään hyviä puoliaan myös Linux-ytimen kanssa, Linux-käyttöjärjestelmäytimen myöntämän Rust-tuen myötä. Vaikka tuki on yhä alkutekijöissään, ovat ensimmäiset Rustilla toteutetut projektit antaneet hyviä merkkejä ohjelmointikielen toimivuudesta, niin tehokkuutensa, turvallisuutensa, kun ohjelmointikokemuksen osalta. Rustin tulevaisuus Linux-ytimessä vaikuttaakin lupaavalta, eikä ole syytä odottaa sen jatkokehityksen päättymistä.

Tämän työn lopputulemana on, että Rust sopii käyttöjärjestelmäytimien kehittämiseen hyvin ja sen sisäänrakennetut turvatoimet ja hyvään ohjelmointityyliin ohjaava logiikka

tuovat kehitykseen suurta etua. Kiinnostavaa on seurata erityisesti Rustin kehitystä Linux-ympäristössä. Uusi kieli todennäköisesti houkuttelee uusia kehittäjiä Linux-ytimen pariin ja Rustin voisi ajatella helpottavan varsinkin uusien ajurien kehittämistä. Kaikkia Rustin avaamia Linux-ytimen kehitysmahdollisuuksia tullaan varmasti käsittelemään vielä paljon tulevissa julkaisuissa ja se onkin hyvä aihe tulevalle tutkimukselle, kunhan ytimen Rust-tuen kehitystä jatketaan ja lisää uusia Rustilla toteutettuja projekteja ilmaantuu.

LÄHTEET

[1] A. S. Tanenbaum, *Modern operating systems*, Fourth edition. teoksessa Always learning. Boston: Pearson, 2015.

[2] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamari, ja L. Ryzhyk, ”System Programming in Rust”, teoksessa *Operating systems review*, 2017, ss. 94–99. doi: 10.1145/3139645.3139660.

[3] O. H. Halvorsen ja D. Clarke, ”Xcode and the Kernel Development Environment”, teoksessa *OS X and iOS Kernel Programming*, Berkeley, CA: Apress, 2011. doi: 10.1007/978-1-4302-3537-8_3.

[4] S. Lankes, J. Breitbart, ja S. Pickartz, ”Exploring Rust for Unikernel Development”, teoksessa *Proceedings of the 10th Workshop on programming languages and operating systems*, teoksessa PLOS'19. ACM, 2019, ss. 8–15. doi: 10.1145/3365137.3365395.

[5] Peter. van der Linden, *Expert C programming : deep C secrets*, 1st edition. Englewood Cliffs, N.J: SunSoft Press, 1994.

[6] R. Sharma, *The complete rust programming reference guide : design, develop, and deploy effective software systems using the advanced constructs of rust*, 1st edition. Birmingham, UK: Packt Publishing, 2019.

[7] R. Levick ja S. Fernandez, ”We need a safer systems programming language”, *Microsoft Security Response Center Blog*, 18. heinäkuuta 2019. https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/ (viitattu 16. huhtikuuta 2023).

[8] ”The Rust community's crate registry”. crates.io (viitattu 20. maaliskuuta 2023).

[9] ”Stack Overflow 2022 Developer Survey”, 2022. https://survey.stackoverflow.co/2022/ (viitattu 13. maaliskuuta 2023).

[10] N. Popescu, Z. Xu, S. Apostolakis, D. I. August, ja A. Levy, ”Safer at any speed: automatic context-aware safety enhancement for Rust”, *Proceedings of ACM on programming languages*, vsk. 5, nro OOPSLA, ss. 1–23, 2021, doi: 10.1145/3485480.

[11] R. Pieper, J. Löff, R. B. Hoffmann, D. Griebler, ja L. G. Fernandes, ”High-level and efficient structured stream parallelism for rust on multi-cores”, *J Comput Lang*, vsk. 65, s. 101054, 2021, doi: 10.1016/j.cola.2021.101054.

[12] T. McNamara, *Rust in Action*. New York: Manning Publications Co. LLC, 2020.

[13] S. De Simone, ”Linux 6.1 Officially Adds Support for Rust in the Kernel”, 20. joulukuuta 2022.

[14] S.-F. Chen ja Y.-S. Wu, ”Linux Kernel Module Development with Rust”, teoksessa *The Institute of Electrical and Electronics Engineers, Inc. (IEEE) Conference Proceedings*, Piscataway: The Institute of Electrical and Electronics Engineers, Inc. (IEEE), 2022. doi: 10.1109/DSC54232.2022.9888822.

[15] M. Ojeda, ”Linux Kernel mailing List: [PATCH 00/13] [RFC] Rust support”, 14. huhtikuuta 2021. https://lkml.org/lkml/2021/4/14/1023 (viitattu 8. huhtikuuta 2023).

[16] ”Rust-for-linux: Crate kernel”, 2023. https://rust-for-linux.github.io/docs/kernel/index.html (viitattu 8. huhtikuuta 2023).

[17] ”The Linux kernel documentation for Rust”, 2023. https://docs.kernel.org/rust/index.html (viitattu 8. huhtikuuta 2023).

[18] Rust for Linux, ”NVMe Driver”, tammikuuta 2023. https://rust-for-linux.com/nvme-driver (viitattu 22. maaliskuuta 2023).

[19] L. Asahi, ”Tales of the M1 GPU”, 29. marraskuuta 2022. https://asahi-linux.org/2022/11/tales-of-the-m1-gpu/ (viitattu 27. helmikuuta 2023).