Tampereen yliopisto

Elias Meyer

REAALIAIKAKÄYTTÖJÄRJESTELMÄT RUSTILLA

Tock OS, Drone OS ja RTIC

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Kesäkuu 2022

TIIVISTELMÄ

Elias Meyer: Reaaliaikakäyttöjärjestelmät Rustilla
Kandidaatintyö
Tampereen yliopisto
Ohjelmistotekniikka
Kesäkuu 2022

Monimutkaisten sulautettujen laitteiden ohjelmistokehitystä voidaan huomattavasti helpottaa ottamalla mukaan käyttöjärjestelmä, joka hoitaa usean prosessin vuorontamisen sekä niiden muistinhallinnan. Aikakriittisissä sovelluksissa käyttöjärjestelmän on kuitenkin pakko taata prosesseille suoritusaikaa tietyn aikajänteen sisällä, mikä on reaaliaikakäyttöjärjestelmän tehtävä. Perinteisesti reaaliaikakäyttöjärjestelmät on kirjoitettu C- tai C++-ohjelmointikielillä, jotka ovat kuitenkin virhealttiita niin muistinhallinnan kuin rinnakkaisuudenkin suhteen. Rust on käytännön ratkaisu C- ja C++-kielten ongelmiin muistinhallinnassa sekä rinnakkaisuudessa.

Tässä kandidaatintyössä tutkitaan kolmen Rust-ohjelmointikielellä kirjoitetun reaaliaikakäyttöjärjestelmän - Tock OS:n, Drone OS:n ja RTIC:n - rakenteita, toimintaperiaatteita sekä vertaillaan niiden eroja. Reaaliaikakäyttöjärjestelmien rakenteesta avataan niiden arkkitehtuuria sekä tarkastellaan millaisia mahdollisuuksia ne tarjoavat käyttäjälle.

Työn tarkoituksena on ohjata lukijaa löytämään projektilleen sopivin reaaliaikakäyttöjärjestelmä sekä tarjota teknistä tietoutta niin käyttöjärjestelmistä, vuorontamisesta kuin muistinhallinnastakin. Työn tarjoamien tietojen ja menetelmien avulla lukija voi itse vertailla tämän työn ulkopuolelle jääviä reaaliaikakäyttöjärjestelmiä.

Työssä pohditaan myös ylipäätään käyttöjärjestelmien käytännön hyötyjä ja tarvetta sulautettujen laitteiden ohjelmistokehityksessä. Reaaliaikakäyttöjärjestelmä helpottaa ohjelmiston kehitystä koska perusasioita, kuten vuorontamista tai muistinhallintaa, ei joka kerta tarvitse tehdä uudelleen. Lisäksi käyttöjärjestelmä toimii usein viitekehyksenä, joka tarjoaa helposti laajennettavan valmiin rakenteen ohjelmistolle. Kaikkia reaaliaikakäyttöjärjestelmiä ei kuitenkaan voida käyttää kaikkien laitteiden kanssa, minkä vuoksi työssä käydään läpi myös yleisimpiä laitteistovaatimuksia, joista tärkeimpiä ovat muistinhallintayksikkö (engl. Memory Management Unit, MMU) sekä muistinsuojausyksikkö (engl. Memory Protection Unit, MPU).

Työssä vertaillaan reaaliaikakäyttöjärjestelmien tehtävien turvallisuutta muistin suhteen, dynaamisen muistin jakamista sekä viestinvälitystä. Prosessien eristäminen osoittautuu parhaaksi Tock OS:ssä koska se on laitteistotasolla suojattu, mikä myös mahdollistaa prosessien kirjoittamisen millä tahansa ohjelmointikielellä. Tock OS on myös ainoa reaaliaikakäyttöjärjestelmä, joka tukee prosessien lataamista ajoaikana. Drone OS pyrkii helppokäyttöisyyteen: se tukee laitteistopohjaista muistin eristämistä mutta toimii ilmankin tukien näin monia erilaisia laitteita. RTIC on yksinkertainen ja kevyt vuoronnuskirjasto, minkä takia se on helposti lähestyttävä. Kaikki kolme reaaliaikakäyttöjärjestelmää tarjoavat myös prosesseille dynaamisen muistinhallinnan sekä viestinvälitysmekanismin, kukin omalla tavallaan.

Avainsanat: Rust, RTOS, Tock-OS, Drone-OS, RTIC, reaaliaikakäyttöjärjestelmä

ALKUSANAT

Työ sai alkunsa kesätyöstä *Wapice Oy*:llä 2021 jossa tehtävänäni oli rakentaa demoprojekti STM32-mikrokontrollerilla, Rustilla sekä jollain reaaliaikakäyttöjärjestelmällä. Kesä meni pitkälti Rustin sekä ARM-mikrokontrollerien ohjelmistokehityksen opetteluun ja syksyllä aloitin vertailemaan näitä kolmea reaaliaikakäyttöjärjestelmää. Haluankin kiittää työnantajaani työn aiheesta sekä tutkimustyöpaikasta.

Kiitos Henri Lunnikivelle joka oli työn aiheohjaaja ja tarkisti työni kolmesti sekä jaksoi vastailla kysymyksiini aidolla mielenkiinnolla. Kiitos myös Matti Haavistolle, joka oli työn varsinainen ohjaaja sekä tarkastaja. Kiva kun kommentoit työni, vaikka itse seminaari olikin jo päättynyt.

Kiitos myös kaikille kavereille, jotka ovat auttaneet jaksamaan opiskelujen aikana Tampereen teknillisessä yliopistossa sekä perheelle, joka on aina ollut läsnä.

Ehdin jo ansaita melkein kaikki maisterivaiheen opintopisteetkin kandin kirjoittamisen aikana, samalla kun etusivun hammasratas vaihtui violetiksi emojiksi. Tarkoituksena onkin aloittaa syksyllä diplomityöprojekti, toivottavasti samasta aiheesta käytännönläheisemmin.

Tampereella, 16. kesäkuuta 2022

Elias Meyer

SISÄLLYSLUETTELO

1 Johdanto 1
2 Käyttöjärjestelmät 3
2.1 Ydin 3
2.1.1 Vuorontaja 3
2.1.2 Rinnakkaisuus ja samanaikaisuus 5
2.2 Muistinhallinta 5
3 Käyttöjärjestelmien vaatima laitteistotuki 6
3.1 Keskeytykset 6
3.2 Laitteistoabstraktiokerros 7
3.3 Muistinsuojaus 7
4 Reaaliaikakäyttöjärjestelmät 8
4.1 Reaaliaikavuoronnus 8
4.2 Staattinen ja dynaaminen moniajo 9
4.3 Sulautetun käyttöjärjestelmän edut 9
4.3.1 Moniajo 9
4.3.2 Palvelut 10
4.3.3 Turvallisuus 10
5 Rust 11
5.1 Muistinhallinta 11
5.2 Rinnakkaisuus 12
5.3 Ohjelman hallittu keskeyttäminen 12
5.4 Rust reaaliaikakäyttöjärjestelmissä 13
6 Rust-reaaliaikakäyttöjärjestelmät 15
6.1 Tock OS 15
6.1.1 Arkkitehtuuri 15
6.1.2 Kapseli 16
6.1.3 Prosessi 17
6.1.4 Viestinvälitys 18
6.1.5 Laitteistotuki 18
6.2 Drone OS 20
6.2.1 Rinnakkaisuus 20
6.2.2 Dynaaminen muisti 22
6.2.3 Viestinvälitys 23
6.2.4 Laitteistotuki 24
6.2.5 Drone CLI 25
6.3 RTIC 26

6.3.1 Tehtävät 26
6.3.2 Resurssit 26
6.3.3 Viestinvälitys 27
6.3.4 Laitteistotuki 27
7 Reaaliaikakäyttöjärjestelmien vertailu 28
7.1 Prosessit 28
7.1.1 Ajoaikainen ohjelmien lataaminen 28
7.1.2 Muistinsuojaus 28
7.2 Dynaaminen muisti 29
7.3 Prosessien välinen viestinvälitys 29
7.4 Helppokäyttöisyys 30
7.5 Tock OS verrattuna muihin yleisiin reaaliaikakäyttöjärjestelmiin 30
8 Yhteenveto 32
Lähteet 33

KUVALUETTELO

2.1 Vuorontajan tilakaavio [2, s. 50] 4
2.2 Prosessorin ajankäyttö kahden prosessin sekä vuorontajan välillä ajan funktiona [3] 4

6.1 Tock OS:n arkkitehtuuri 16
6.2 Rinnakkaisuus käytännössä: useaa prosessia ajetaan samaan aikaan, jokaisella on oma pino [27] 20
6.3 Rinnakkaisuus teoriassa: jokaista prosessia ajetaan vuorotellen peräkkäin, jokaisella on oma pino [27] 21
6.4 Drone OS yhden pinon rinnakkaisuus [27] 21
6.5 Drone OS:n keko jaettuna muistilohkoihin [32, teksti suomennettu] 23

TAULUKKOLUETTELO

6.1 Tock OS:n suojamekanismit 17

7.1 Reaaliaikakäyttöjärjestelmien ominaisuudet 28
7.2 Tock OS verrattuna muihin yleisiin reaaliaikakäyttöjärjestelmiin [37] 30

OHJELMA- JA ALGORITMILUETTELO

5.1 Esimerkki unsafe-tilan käyttämiesestä Rustissa 12
5.2 Virheellinen koodiesimerkki C:ssä 13
5.3 Virheellinen koodiesimerkki Rustissa 13
6.1 Drone OS:n Muistilohkojen määrittely Drone.toml -tiedostossa [32] 23
6.2 Esimerkki oneshot-viestikanavan käytöstä 24

LYHENTEET JA MERKINNÄT

ansa ohjelmavirhekeskeytys; ohjelman virheellisen toiminnan seurauksena käyttöjärjestelmän ytimelle lähetetty keskeytys joka voi aiheutua esimerkiksi laittomaan muistialueeseen vaikuttamisesta (engl. *trap*)

grants Tock OS:n mekanismi dynaamiselle muistin varaamiselle kapseleille (suom. myönnytys)

HAL laitteistoabstraktiokerros (engl. *Hardware Abstraction Layer*)

IPC prosessien välinen kommunikaatio (engl. *Inter Process Communication*

LaTeX ladontajärjestelmä tieteelliseen kirjoittamiseen

MMU muistinhallintayksikkö (engl. Memory Management Unit)

MPU muistinsuojausyksikkö (engl. Memory Protection Unit)

MMIO muistikartoitettu rajapinta; oheislaitteiden kautta kommunikoidaan käsittelemällä tavallista ohjelmamuistia tarkasti määritellyistä osoitteista (engl. Memory Mapped I/O)

mutex poissulkeva säielukko resurssin samanaikaisen käytön estämiseen säikeiden välillä (engl. mutual exclusion, mutex)

NVIC sisäkkäinen keskeytysvektoriohjain (engl. Nested Vectored Interrupt Controller)

PAC oheislaitekirjasto; Rustissa käytetty tekniikka koota oheislaitteiden ohjauskoodi kirjastoihin (engl. *Peripheral Access Crate*)

RTIC Real-Time Interrupt-driven Concurrency, yksi tarkasteltavista reaaliaikakäyttöjärjestelmistä

RAM Random Access Memory, käyttömuisti

struct Tietorakenne C- ja Rust-ohjelmointikielissä jossa usea muuttuja on niputettu yhteen ja ne sijaitsevat muistissa peräkkäin

1 JOHDANTO

Nykyaikaiset sulautetut järjestelmät ovat hyvin monimutkaisia. Suuri osa laitteista kytketään joko internetiin tai muuhun verkkoon, mikä tarkoittaa että esimerkiksi TCP/IP-protokollapinon tai vastaavan toteuttaminen järjestelmään on tarpeellista. Tietoliikenne on yleisesti ottaen hidas verrattuna prosessorin suoritusnopeuteen, minkä takia sitä pitää suorittaa taustalla muun suorituksen ohella, jottei se estä muun ohjelman suoritusta datan lähettämisen ajaksi. Suoritusaikaa voi samalla käyttää esimerkiksi analytiikkaan.

Mikäli sulautetuissa järjestelmissä tehdään useaa asiaa samanaikaisesti, on hyödyllistä käyttää käyttöjärjestelmää. Käyttöjärjestelmän tärkein tehtävä on usean samanaikaisesti suoritettavan tehtävän vuorontaminen jolloin sovelluskehittäjä voi keskittyä itse sovelluskohteen ratkaisemiseen eikä hänen tarvitse panutua ongelmaan, johon on jo olemassa valmiita **viitekehyksiä** (engl. *framework*). Reaaliaikaisuus on vain pieni lisä käyttöjärjestelmään, se tarjoaa takuun, että vuoronnin antaa tehtäville suoritusaikaa viimeistään ennen tiettyä aikarajaa.

Perinteisesti sulautettuja järjestelmiä on kehitetty C-ohjelmointikielellä, sillä sen ohjelmointimalli vastaa tarkasti prosessorin todellista toimintaa. C:n matalatasoisuus tuo kuitenkin ongelmia etenkin huolimattomuuden kanssa: varattu muisti täytyy vapauttaa ja samaa muistialuetta voi käsitellä monesta paikasta yhtä aikaa. Muistin yli-indeksointi puolestaan on määrittelemätön tapahtuma. Mikäli kosketaan muistialueeseen, mikä sijaitsee käyttöjärjestelmän määrittelemän alueen ulkopuolella, voi käyttöjärjestelmä lopettaa ohjelman suorituksen välittömästi. Kevyemmissä sulautetuissa järjestelmissä, missä käyttöjärjestelmä ei huolehdi muistin sivuttamisesta, yli-indeksointi aiheuttaa todennäköisesti virheitä jotka havaitaan vasta myöhemmin ohjelman suorituksen aikana. Yli-indeksoinnilla voi myös paljastaa ohjelman sisäistä muistia ja täten vuotaa esimerkiksi salasanoja. Microsoftin mukaan jopa 70% tietoturvaongelmista johtuu muistinkäsittelyn turvallisuusongelmista [1].

Vuonna 2010 Mozillan tutkimusprojektista alkunsa saanut ohjelmointikieli Rust pyrkii korjaamaan C-kielen ongelmia etenkin muistinhallinnan suhteen ilman että ohjelman tehokkuus kärsii. Rust sisältää myös hyvän tuen rinnakkaisuudelle, mikä helpottaa suuresti muun muassa reaaliaikakäyttöjärjestelmien kirjoittamista.

Tämän kandidaatintyön tarkoitus on vertailla kolmea Rustilla kirjoitettua reaaliaikakäyttöjärjestelmää jotka ovat Tock OS, Drone OS sekä RTIC. Vertailussa selvitetään näiden toiminta- ja käyttöperiaatteet sekä niiden eroja.

Luvussa 2 tarkastellaan, mikä on käyttöjärjestelmä ja pohditaan ylipäätään tarvetta käyttöjärjestelmälle sulautetuissa järjestelmissä. Luvussa 3 käsitellään käyttöjärjestelmän vaatimaa laitteistotukea. Luvussa 4 käydään läpi, miten reaaliaikakäyttöjärjestelmä eroaa normaalista käyttöjärjestelmästä. Luvussa 5 käydään läpi Rust-ohjelmointikieltä sekä miksi se soveltuu niin hyvin reaaliaikakäyttöjärjestelmien kirjoittamiseen. Luvussa 6 kerrotaan kustakin edellämainitusta reaaliaikakäyttöjärjestelmästä sekä luvussa 7 vertaillaan edellä kuvattujen reaaliaikakäyttöjärjestelmien ominaisuuksia toisiinsa.

2 KÄYTTÖJÄRJESTELMÄT

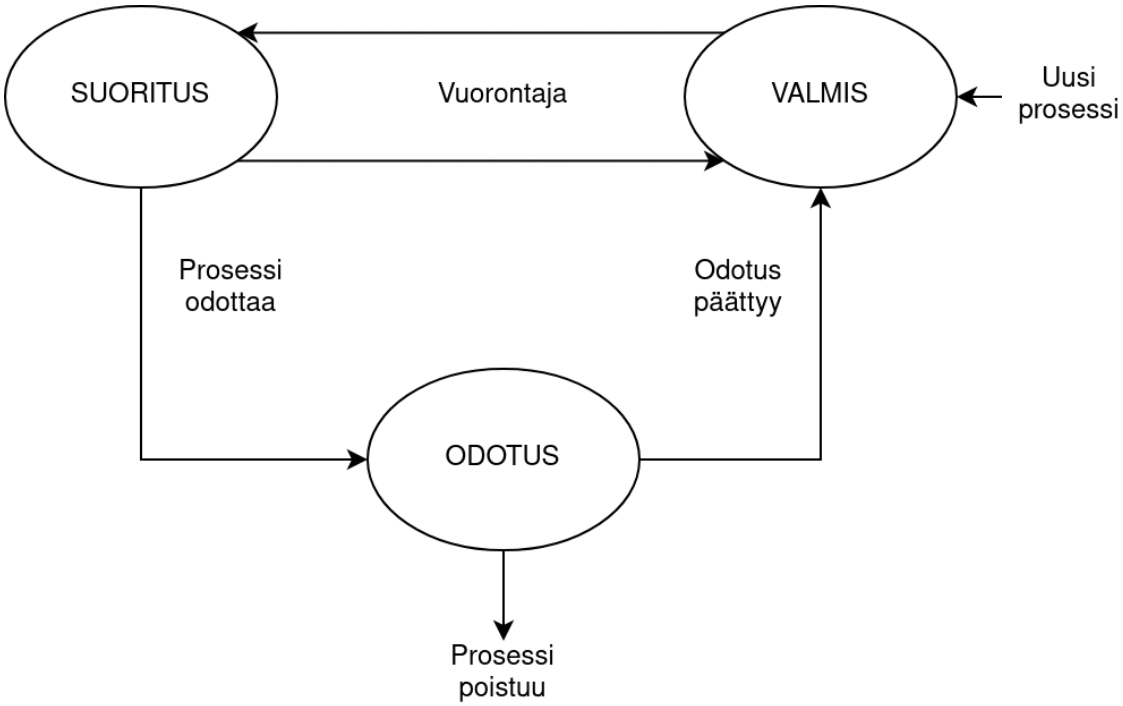
Käyttöjärjestelmällä tarkoitetaan arkikielessä yleensä eri asiaa kuin mikä sen määritelmä on. Arkikielessä tarkoitetaan usein käyttäjälle näkyvää, esimerkiksi Windows, Linux tai Android-käyttöjärjestelmää, mikä on paljon muutakin kuin itse käyttöjärjestelmän **ydin** (engl. *kernel*). Arkikielen käyttöjärjestelmässä on mukana paljon ulkopuolisia ohjelmia ytimen lisäksi, esimerkiksi **ikkunapalvelin** (engl. *window server*), **työpöytäympäristö** (engl. *desktop environment*) tai **äänipalvelin** (engl. *sound server*), jotka voivat olla tai olla olematta osa käyttöjärjestelmää. Käyttöjärjestelmällä voi olla monta erilaista käyttöliittymää. [2, s. 10].

2.1 Ydin

Moderni käyttöjärjestelmän ydin on vastuussa esimerkiksi prosessien vuorontamisesta sulavan moniajokokemuksen takaamiseksi ja on käyttäjälle täysin näkymätön. Muita sen tärkeitä tehtäviä ovat esimerkiksi resurssien, kuten muistin, jakaminen prosesseille sekä oheislaitteiden ohjaaminen.

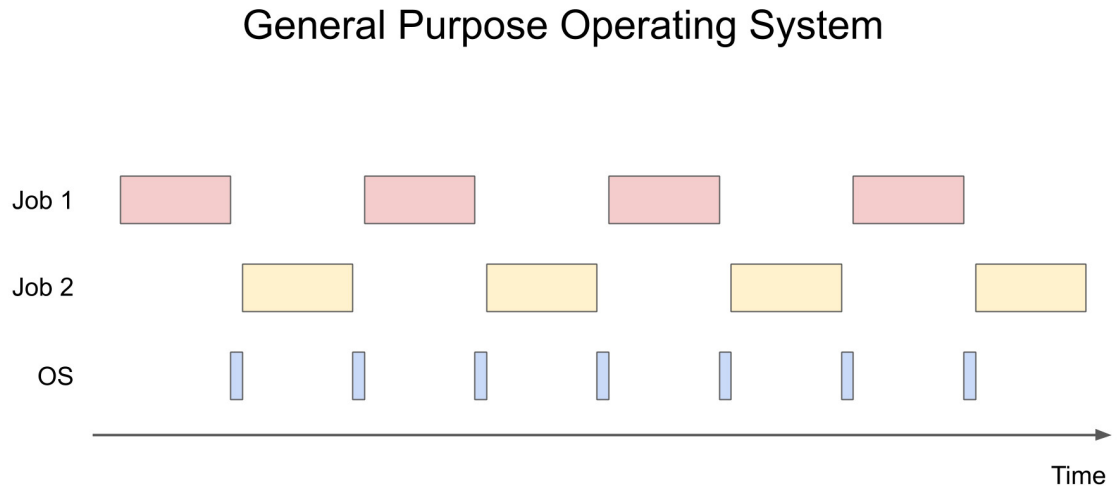
2.1.1 Vuorontaja

Vuorontaja on vastuussa suorittimen resurssien eli suoritusajan jakamisesta eri prosesseille. Sen tehtävä on käytännössä siirellä prosesseja kuvan 2.1 kolmen tilan välillä. Prosessi on aina joko suorituksessa, valmis suoritukseen tai odottaa ulkoista tapahtumaa (esimerkiksi tiedoston lukemista kiintolevyltä tai ulkoista keskeytystä). Vuorontaja valitsee suoritukseen aina yhden prosessin niiden prosessien joukosta, jotka ovat valmiita suoritukseen. Kun prosessin suoritusaika päättyy tai prosessi alkaa odottaa jotain ulkoista tapahtumaa, esimerkiksi levyltä lukemista tai aktiviteettia oheislaitteelta, siirtää vuorontaja prosessin odottavaan tilaan ja ottaa seuraavan suoritettavan prosessin.



Kuva 2.1. *Vuorontajan tilakaavio [2, s. 50]*

Kuvassa 2.2 on kuvattu mitä prosessia suoritin kullakin ajan hetkellä suorittaa: suurin osa suoritusajasta kuluu prosessien 1 (punainen) sekä 2 (keltainen) suorittamiseen mutta säännöllisin aikavälein myös itse ydin (sininen) saa suoritusaikaa. Tärkeää on huomata, että kun prosessit 1 tai 2 ovat suorituksessa, ei käyttöjärjestelmän koodia ajeta sillä välin lainkaan.



Kuva 2.2. *Prosessorin ajankäyttö kahden prosessin sekä vuorontajan välillä ajan funktiona [3]*

Lohkomalla aika suurempiin kokonaisuuksiin saadaan aika käytettyä kokonaisuudessa tehokkaammin hyödyksi, koska vähemmän aikaa kuluu prosessien vaihtamiseen.

Vapaa-ajalla, jolloin yksikään prosessi ei kaipaa suoritusaikaa, voi käyttöjärjestelmä sammuttaa osan prosessorista säästäen näin energiaa ja mahdollisen pariston elinikää. Sulautetut järjestelmät viettävät hyvinkin paljon käynnissäoloajastaan unitilassa, heräten säännöllisesti keskeytykseen.

2.1.2 Rinnakkaisuus ja samanaikaisuus

Rinnakkaisuus voi olla *todellista* tai *näennäistä*, riippuen siitä montako prosessoriydintä suorittavassa järjestelmässä on. Koska jokainen prosessorin ydin voi suorittaa kerrallaan vain yhtä säiettä, yksiydinprosessorissa todellisuudessa rinnakkaisuus tarkoittaa säikeiden peräjälkeistä - tai lomittaista - suoritusta. Todellisesta rinnakkaisuudesta puhutaan myös samanaikaisuutena (engl. *parallelism*) - moniydinprosessorissa jokaisessa ytimessä voi ajaa säikeitä samanaikaisesti ytimien määrän verran.

Sovellustason kehityksessä näillä kahdella ei ole merkittävää eroa, koska kummassakaan ei sovellustasolla voida olettaa koskaan turvalliseksi käyttää samaa resurssia kahdesta säikeestä ilman asiaan kuuluvaa lukitsemista. Käyttöjärjestelmä on yksin vastuussa säikeiden vaihtamisesta eikä mitään takuita säikeiden vaihtamisen ajoitukselle ole. Sen sijaan käyttöjärjestelmän kirjoittaminen yksinkertaistuu jonkin verran, jos suorittavia ytimiä on vain yksi. [2, s. 88]

Koska tässä työssä käsitellään reaaliaikakäyttöjärjestelmien rakenteita, ero on hyvä tiedostaa vaikka useimmissa sulautetuissa järjestelmissä onkin vain yksi fyysinen prosessoriydin.

2.2 Muistinhallinta

Käyttöjärjestelmän tehtävä ei pelkästään ole vuorontaa prosesseja, vaan myös huolehtia, että prosessit eivät pysty vaikuttamaan toistensa muistialueisiin estämällä niin luku kuin kirjoituskin silloin kun on tarpeellista. Nykyaikaiset prosessorit tukevat muistin virtualisointia, mikä tarkoittaa että kullekin prosessille on oma, virtuaalinen muistiavaruus joka koostuu muistisivuista. Virtuaalinen muisti tuo etuna muunmuassa dynaamisuuden: prosessi voi pyytää lisää muistisivuja käyttöönsä, eikä muistisivujen tarvitse olla fyysisesti peräkkäin muistissa.

Prosessit tarvitsevat myös erilaisia muistialueita käyttöönsä: osassa muistisivuista on niiden kirjoitussuojattu koodi jota ne suorittavat, toisilla sivuilla luku- ja kirjoituskelpoinen muistialue **pinolle** (engl. *stack*) sekä **keolle** (engl. *heap*). Prosessit voivat jopa jakaa kirjoitussuojattuja muistisivuja joissa on dynaamisten kirjastojen ohjelmakoodia, joita monet prosessit tarvitsevat yhtä aikaa [2] (Linuxissa .so ja Windowsissa .dll -tiedostot).

3 KÄYTTÖJÄRJESTELMIEN VAATIMA LAITTEISTOTUKI

Käyttöjärjestelmillä on tiettyjä laitteistovaatimuksia ilman mitä ne eivät toimi, esimerkiksi muistinhallintayksikkö on välttämätön modernille käyttöjärjestelmälle muistin virtualisointia varten. STM32 on hyvä esimerkki yleisestä sulautetusta ARM-piiriperheestä, joka ei sisällä MMUta.

3.1 Keskeytykset

Mikrokontrollereilla täytyy olla jokin mekanismi jolla voidaan reagoida ulkoa tuleviin ärsykkeisiin, esimerkiksi napinpainalluksesta aiheutuvaan jännitemuutokseen digitaalisessa sisääntulossa. Näitä varten prosessoreissa on **keskeytykset** (engl. *interrupts*) jotka keskeyttävät pääohjelman suorituksen siksi aikaa, kunnes ulkoinen ärsyke saadaan käsiteltyä. Keskeytystä käsittelevä koodi laitetaan **keskeytyskäsittelijäfunktioon** (engl. *Interrupt Service Routine*, ISR).

Aikakriittiset osat, kuten esimerkiksi oheislaitteiden tilan lukeminen, on perinteisesti suoritettu näiden keskeytysten avulla. Keskeytyskäsittelijän funktiokutsu tulisi olla mahdollisimman nopea koska se keskeyttää pääohjelman suorituksen jota jatketaan vasta, kun keskeytyskäsittelijästä ollaan palattu. Kaikki aika mikä käytetään keskeytyskäsitteljässä, on suoraan pois pääohjelman suorittamisesta.

Käyttöjärjestelmä voi olla keskeytyksen aikana tuntemattomassa tilassa, missä ei ole kaikkia käyttöjärjestelmän ominaisuuksia saatavilla. Tämän takia keskeytyskäsittelijä jaetaan Linuxissa kahteen osaan: **yläpuoliskoon** (engl. *top half*) ja **alapuoliskoon** (engl. *bottom half*) [4].

Ylempi puolisko on tarkoitettu mahdollisimman nopeasti suoritettavaksi, siinä voidaan esimerkiksi lukea oheislaitteelta tullut viesti ja tallettaa se pinoon. Lisäksi se voi aktivoida alapuoliskon keskeytyksen jolloin se suoritetaan heti, kun käyttöjärjestelmä niin sallii. Alapuoliskon keskeytyksessä on enemmän käyttöjärjestelmän ominaisuuksia saatavilla.

On myös huomioitava, estääkö keskeytyskäsittelijän suoritus muut keskeytykset (kuten esimerkiksi AVR tekee [5]), ovatko sisäkkäiset keskeytykset sallittuja (rekursiivinen keskeytys) vai suoritetaanko keskeytykset peräkkäin prioriteettien mukaan [6].

3.2 Laitteistoabstraktiokerros

Laitteistoabstraktiokerroksen (engl. *Hardware Abstraction Layer*, **HAL**) tarkoitus on tarjota laitteistoläheiselle ohjelmoinnille vakaa rajapinta, joka ei ole riippuvainen alla olevasta laitteistosta [7, luku 'Introduction']. Ohjelmoijan on mielekkäämpää työskennellä esimerkiksi protokollatasolla kuin rekisteritasolla. Näin samaa ohjelmaa voi myös käyttää saman valmistajan eri piirien välillä vaikka alla oleva laitteistototeutus muuttuukin [8, luku 'Design Goals']. HAL myös mahdollistaa virheellisten konfiguraatioiden tarkistamisen staattisesti.

Rustissa laitteistoabstraktiokerros kasataan oheislaitekirjastosta (engl. *Peripheral Access Crate*, **PAC**). Oheislaitekirjasto on käytännössä yksi Rust-paketti (engl. *Crate*) joka sisältää oheislaitteiden rajapinnan kokonaiselle tuoteperheelle [9]. Yksi laite on aina yksi moduuli kyseisen paketin sisällä. PAC on kätevä tapa järjestää **MMIO** (engl. *Memory Mapped IO*) Rustissa. MMIO on nimitys mekanismille jonka kautta jokainen oheislaite löytyy mikrokontrollerin yleisestä muistiavaruudesta. Oheislaitekirjastot rakennetaan puoliautomatisoidusti ohjelmalla *svd2rust* [10].

3.3 Muistinsuojaus

Kehittyneemmissä prosessoreissa, kuten x86-arkkitehtuurissa tai joissain ARM-prosessoreissa muistin virtualisoinnin ja suojauksen prosessorissa hoitaa muistinhallintayksikkö (engl. *Memory Management Unit*, **MMU**). [11, s. 169].

Sulautetuissa järjestelmissä muistin virtualisointia tukevaa MMUta ei kuitenkaan yleensä ole. Reaaliaikakäyttöjärjestelmissä, jotka ajavat useaa prosessia, olisi kuitenkin hyödyllistä olla mekanismi, jolla voi rajoittaa prosesseja koskemasta toistensa muistialueisiin.

Esimerkiksi ARM Cortex-M3 sisältää muistinsuojausyksikön (engl. *Memory Protection Unit*, **MPU**). [12] joka käytössä ollessaan suojaa esimerkiksi käyttöjärjestelmän muistialuetta käyttäjän tilassa suoritettavilta prosesseilta tai suojaa prosessien muistialuetta toisilta prosesseilta. Muistinsuojausyksikön avulla muistialueita voi myös asettaa kirjoitussuojattuun tilaan jolloin tietoa ei voi vahingossakaan ylikirjoittaa. MPU osaa myös havaita muun muassa pinon korruptoitumisen mikäli keko ja pino törmäävät.

Pienemmät sulautetut järjestelmät eivät sisällä edes MPUta. Esimerkki tällaisesta järjestelmästä on BluePillinä tunnetun kehitysalustan mikrokontrolleri STM32F103.

4 REAALIAIKAKÄYTTÖJÄRJESTELMÄT

Normaali, ei reaaliaikainen ydin pyrkii vuoronnusalgoritmista riippuen antamaan mahdollisimman paljon suoritusaikaa eri prosesseille takaamatta kuitenkaan milloin prosessi viimeistään saa suoritusaikaa. Vaikka arkikielessä ’reaaliaikaisuus’ tarkoittaakin usein että asia tapahtuu mahdollisimman nopeasti, käyttöjärjestelmissä tämä tarkoittaa ennakoitavuutta. Reaaliaikakäyttöjärjestelmä takaa, että vuoronnus tapahtuu tietyn aikajänteen sisällä, toisin sanoen se määrittelee alku- ja loppuhetken jonka välissä prosessin on saatava suoritusaikaa. [2, s. 81-83]

Reaaliaikaisuus voi olla **pehmeää** (engl. *soft*) tai **kovaa** (engl. *hard*) [2, s. 81] riippuen käyttötapauksesta. Pehmeä tarkoittaa, että mikäli prosessi myöhästyy - tai aikaistuu - liikaa, epäonnistuminen johtaa vain huonompaan lopputulokseen. Kova taas tarkoittaa, että epäonnistuminen johtaa vakavaan virheeseen - käyttötapauksesta riippuen mahdollisesti oikeaan vaaratilanteeseen tai tapaturmaan. Lähteen [11, s. 401] mukaan jako pehmeän ja kovan välillä tehdään taas sen mukaan, saavuttaako reaaliaikakäyttöjärjestelmä aikamääreen aina, vai vain yleensä eikä yksinomaan tapahtuman viivästymisen tai aikaistumisen seurauksen vakavuudesta. Jälkimmäinen olisi parempi määritelmä, koska se on tarkka.

4.1 Reaaliaikavuoronnus

Reaaliaikakäyttöjärjestelmään sopii parhaiten irrottava kiinteän prioriteetin vuorontaja koska se on ennalta arvattavin ja sille voidaan antaa joitakin takeita vasteajoille. Irroittava tarkoittaa, että vuorontaja voi ajan loputtua vaihtaa vuoronnettavaa prosessia, vaikkei prosessi olisikaan saatu vielä päätökseen tai odottavaan tilaan [2, s. 29]. Prioriteetti on prosessiin liitetty kokonaisluku [2, s. 32], joka kuvaa prosessin kiireellisyyttä. Suoritusaikaa saa aina se prosessi, joka on valmis suoritukseen (prosessi on valmis-tilassa, ks. kuva 2.1) ja jolla on korkein prioriteetti.

Reaaliaikakäyttöjärjestelmä on yleisesti ottaen tehottomampi (suoritettavien prosessien suhteen) kuin vastaava käyttöjärjestelmä jossa vuoronnusalgoritmi on optimoitu prosessien suorittamiseen eikä tarkkoihin ajoituksiin.

4.2 Staattinen ja dynaaminen moniajo

Arkipäiväisessä käyttöjärjestelmässä ohjelmien moniajo on dynaamista. Esimerkiksi verkkoselain ei ole käännetty staattisesti käyttöjärjestelmään mukaan vaan se on erillinen binääri, jonka käyttöjärjestelmä lataa muistiin ja suorittaa. Käyttäjä voi myös käynnistää ja sammuttaa ohjelmia käyttöjärjestelmän ajon aikana.

Reaaliaikakäyttöjärjestelmän voi tehdä sulautettuun järjestelmään joko erillisenä komponenttina joka pystyy ajoaikaisesti lataamaan ohjelmia muistiinsa ja suorittamaan niitä, tai hyvinkin yksinkertaisena kirjastona johon itse suoritettavat ohjelmat käännetään staattisesti. Tässä suoritettavat prosessit ovatkin vain ohjelman moduuleita joiden suoritusfunktio annetaan vuorontajalle aloitettavaksi. Kirjastotyyppisen reaaliaikakäyttöjärjestelmän haittapuoli ajoaikaisen ohjelmien lataamisen mahdottomuuden lisäksi on kernel-tilan muistinsuojauksen hankaluus.

Tietokoneella suoritetuista ohjelmista puhutaan prosesseina. Niillä voi olla suorituksessa useita säikeitä ja niillä on oma virtuaalinen muistiavaruutensa. Reaaliaikakäyttöjärjestelmissä, joissa jokainen prosessi on periaatteessa vain yksi säie ja muistinsuojaus riippuu alustasta, puhutaan sekalaisesti prosesseista, säikeistä sekä tehtävistä (engl. *task*) [11]. Tässä työssä pyritäänkin käyttämään kulloinkin sitä termiä, jota virallinen dokumentaatiokin käyttää. Usein nämä tarkoittavatkin eri asiaa yhden reaaliaikakäyttöjärjestelmän kontekstissa.

4.3 Sulautetun käyttöjärjestelmän edut

Käyttöjärjestelmä helpottaa ohjelmistokehitystä sulautetuissa järjestelmissä. Ne luovat ohjelmalle hyvän rakenteen jota on helppo laajentaa myöhemmin, joka on helpommin siirrettävissä ja turvallinen. Ohjelmiston ylläpito helpottuu mikä säästää aikaa ja rahaa [13].

4.3.1 Moniajo

Sulautetussa ohjelmoinnissa on perinteisesti ollut tapana kirjoittaa omavarainen ohjelma suoraan laitteiston muistiin suoritettavaksi, ilman käyttöjärjestelmää. Usein kuitenkin sulautettu järjestelmä tekee monia asioita, jolloin tehtäviä joudutaan vuorontamaan. Helpoin tapa on suorittaa tehtävät peräjälkeen silmukassa [13], mikä on kuitenkin vaikeasti ylläpidettävä ohjelmiston kasvaessa - joka kerta kun funktion suoritusaika pitenee, täytyy laskea ajoitukset ja viiveet uudelleen toisia tehtäviä varten. Lisäksi, jos toisia asioita pitää suorittaa useammin kuin toisia, tarvitaan monimutkaisempi hallintarakenne.

Eri tyyppisten prosessien vuorontaminen voi olla myös haastavaa. Tarkastellaan esimerkiksi modernia valvontakameraa joka ottaa kuvan ja lähettää sen TCP/IP-verkon yli palvelimelle. Kamera voi joutua tekemään erilaisia korjauksia kuvaan, kuten kohinanpoistoa, mikä algoritmeista riippuen voi olla raskaskin operaatio. Lisäksi kamera joutuu lähettä-

mään kuvan palvelimelle mikä on sinällään helppoa, mutta tietoliikenteen nopeudesta riippuen operaatio saattaa kestää pitkäänkin. Sen sijaan, että kuvan lähettämisen ajaksi jäätäisiin odottamaan lähetyksen valmistumista, on viisaampaa käyttää tämä odotteluaika hyväksi jo seuraavan kuvan ottamiseen ja muokkaamiseen. Reaaliaikakäyttöjärjestelmä on erikoistunut tämänkaltaisten eri prosessien tehokkaaseen vuorontamiseen.

4.3.2 Palvelut

Reaaliaikakäyttöjärjestelmät usein tarjoavat valmiina käteviä palveluita moniajoa ajatellen, esimerkiksi muistinhallintaa joka vaatii monimutkaista laitteiston ja ohjelmiston yhteistyötä, prosessien välisen kommunikaatiopalvelun sekä keskeytyskäsittelyn [13]. Nämä kaikki ovat tärkeitä perusominaisuuksia, joita ei sovelluskehittäjän tarvitse joka kerta ratkaista uudelleen kun ne on jo rakennettu pohjalla olevaan käyttöjärjestelmään.

4.3.3 Turvallisuus

Ilman reaaliaikakäyttöjärjestelmän tuomaa prosessin välistä kommunikaatiota muun muassa keskeytyskäsittelijän ja pääohjelman välinen kommunikaatio on perinteisesti toteutettu julkisilla muuttujilla ja jaetuilla muistialueilla [14].

On hyvä huomata, että mikäli haluaa käyttää jaettuja muistialueita edellämainittuun tapaan useasta säikeestä reaaliaikakäyttöjärjestelmästä, on otettava huomioon **poissulkemisongelma** (engl. *mutual exclusion*) eli muistialueiden samanaikainen kirjoittaminen useasta säikeestä on estettävä lukitsemalla resurssi poissulkevalla säielukolla eli **mutex**illa. [2, s. 89]

5 RUST

Rust on käytännön ratkaisu laitteistoläheiseen turvalliseen ja tehokkaaseen muistinhallintaan ja sen kautta rinnakkaisuuteen. Rust pyrkii samaan aikaan produktiivisuuteen ja tehokkuuteen, käyttäen C++:n pioneeraamaa “zero-cost abstraction” -ajattelua. Samalla se tarjoaa C/C++:n muistin optimoitavuuden. Rustin monipuolisuuden hinta on kuitenkin jyrkempi oppismikäyrä [15].

5.1 Muistinhallinta

Yleisesti ohjelmointikielissä on kahdenlaista muistinhallintaa. Esimerkiksi C/C++ -kielissä muistinhallinta on manuaalista, mikä tarkoittaa, että ohjelmoija on itse vastuussa ohjelman muistin varaamisestä sekä vapauttamisesta. Manuaalinen muistinhallinta mahdollistaa muistin hallinnan tarkasti, mutta on samalla virhealtista muistivuodoille ohjelmoijan unohtaessa vapauttaa varatun muistin.

Toinen vaihtoehto on käyttää automaattista **roskienkeruujärjestelmää** (engl. *garbage collector*) mikä on käytössä esimerkiksi Javassa tai Pythonissa, joka säännöllisin väliajoin käy vapauttamassa tarpeettoman muistin. Roskienkeruu säästää ohjelmoijalta muistin vapauttamisen vaivan. Automaattisen roskienkeruun huonona puolena on sen arvaamattomuus, koska muistin vapauttaminen on raskas operaatio ja voi merkittävästi haitata muun ohjelman suoritusta. Viitteiden laskemisella ei pystytä poistamaan tietorakenteita joissa on kehäriippuvuuksia, toisin kuin roskienkeruulla [16].

Rust ottaa kokonaan uuden lähtökohdan muistinhallintaan, omistajuuden. Jokaisella varatulla muistialkiolla on vain yksi omistaja [17]. Kun muistialkioon liittyvien viitteiden **näkyvyysalueet** (engl. *scope*) ovat päättyyneet, muisti vapautetaan automaattisesti.

Rustissa ei käytetä termiä **muuttuja** vaan **sidoste** (engl. *binding*). Mutta koska termi muuttuja on niin yleisessä käytössä suomen kielessä, käytetään tässä työssä sanoja *muuttuja* ja *sidoste* keskenään vaihtokelpoisina selkeyden vuoksi. Koska muistialkiolla voi olla vain yksi omistaja, tarkoittaa se myös, että muistialkiota voi muokata vain yhdestä paikasta ohjelmaa kerrallaan. Nykyaikaisessa kääntäjässä poissulkeva muuttuvuus mahdollistaa muistinkäytön huolellisen, käännösaikaisen optimoinnin jopa paremmin kuin perinteisemmät osoitinpohjaiset muistinhallintatyökalut.

Rustissa muistinhallinta on käännösaikaista. Kääntäjä ei käännä ohjelmaa, mikäli se ei noudata muistinhallinnan sääntöjä. Ohjelmien kirjoittaminen voi tuntua aluksi vaikealta

koska kääntäjä on hyvin tarkka ohjelman oikeellisuudesta, mutta etuna on roskienkeruujärjestelmän tarpeettomuus sekä muistinhallinan tarkkuus ja turvallisuus [15].

Vaarallinen tila

On parempi, että Rust-kääntäjä hylkää oletuksena koodit, joista se ei voi olla aivan varma ovatko ne sääntöjen mukaisia kuin että se hyväksyisi ne, koska silloin voisi joskus tapahtuakin ajoaikainen virhe. Esimerkiksi raakojen C-tyylisten osoitinten viitteiden ratkaiseminen/seuraaminen on Rust-kielessä normaalisti kielletty, sillä osoitinten semantiikka ei seuraa Rustin muistinhallinnan sääntöjä. Rustissa on myös **vaarallinen tila** (engl. *Unsafe Rust*), jota käyttäjä voi käyttää mikäli tietää kontekstin paremmin kuin kääntäjä.

```
1     let ptr = 0xdead_beef as *const usize;
2     unsafe {
3         // TURVATON! Viitteen takana ei ehkä ole arvoa
4         println!("Mita␣viitattiinkaan?␣{}", *ptr);
5     }
6
```

Listaus 5.1. *Esimerkki unsafe-tilan käyttämiesestä Rustissa*

Ilman tätä tilaa laitteistoläheisyyskään ei olisi mahdollista koska laitteistorekisterien käsittely ei luonnostaan ole turvallista. [18]

5.2 Rinnakkaisuus

Valtaosa rinnakkaisuuteen liittyvistä ongelmista liittyy muistinhallintaan, minkä vuoksi Rustissa hyvän muistinhallinnan takia rinnakkaisuuskin onnistuu hyvin. Monet korkean tason kielet joutuvat karsimaan rinnakkaisuuden ominaisuuksista korkean abstraktiotason vuoksi, mutta matalan tason kielissä kuten Rustissa on tärkeää, että rinnakkaisuus voidaan tehdä juuri siten mikä on tehokkain tapa [19] kyseisessä laitteistossa.

Rustissa muuttujat ovat oletuksena muuttumattomia mikä edesauttaa turvallisuutta, sillä käyttäjän on eksplisiittisesti kerrottava `mut`-avainsanalla jos haluaa varata muistialkion jota voi muokata jälkikäteen. Muuttumattomat muuttujat ovat myös tärkeitä rinnakkaisuuden turvallisessa toteuttamisessa, koska rinnakkaisuudessa on tärkeä kiinnittää huomiota siihen, mistä säikeistä muuttujia muokataan.

5.3 Ohjelman hallittu keskeyttäminen

Rustin tärkeä ominaisuus on ohjelman hallittu keskeytys. Otetaan esimerkiksi koodiesimerkki 5.2 jossa luodaan 3 alkiota pitkä taulukko `v` ja sitä indeksoidaan 100. alkion kohdalta:

```
1
2       #include <stdio.h>
3
4       int main() {
5           int v[3] = {1,3,4};
6
7           printf("%d\n", v[100]);
8       }
```

Listaus 5.2. *Virheellinen koodiesimerkki C:ssä*

C:ssä koodi kääntyy ja sen käyttäytyminen suorituksessa on määrittelemätöntä. C ei yksinkertaisesti ota kantaa, mitä tapahtuu, kun kosketaan muistialueeseen joka ei ole vektorille `v` annettu. Käyttöjärjestelmä voi lopettaa ohjelman suorituksen mikäli muistialue ei kuulu ohjelmalle, mutta mikäli ohjelman muistinkäyttöä vartioivaa käyttöjärjestelmää ei ole esimerkiksi laitteiston puutteellisuuden takia, voidaan ohjelma suorittaa ongelmitta. Laittomalle muistialueelle kirjoittaminen voidaan huomata vasta tulevaisuudessa muistin korruptoiduttua ohjelman epämääräisenä toimintana.

```
1       fn main() {
2           let v = vec![1, 2, 3];
3
4           v[99];
5       }
6
```

Listaus 5.3. *Virheellinen koodiesimerkki Rustissa*

Tässä esimerkissä on kysymys rajatestistä (engl. *bounds check*). Rajatestissä Rust-kääntäjä lisää vastaavassa ohjelmakoodissa 5.3 taulukon indeksointioperaation eteen tarpeen mukaan ‘cmp‘-vertauskäskyn ja ‘j‘-hyppykäskyn. Hyppykäskyn kohteena on ’panic handler’ -nimellä kutsuttu funktio, joka lopettaa ohjelman suorituksen hallitusti [20].

5.4 Rust reaaliaikakäyttöjärjestelmissä

Rustin muistiturvallisuus auttaa myös sulautetuissa järjestelmissä kun kääntäjä huomaa monia virheitä ennen kuin itse ohjelmaa edes kokeillaan mikrokontrollerissa ajoaikana. Mikrokontrollerin ohjelmointi on kuitenkin hitaampi prosessi kuin pelkän koodin kääntäminen. Lisäksi edellä mainitussa rajatestissä kutsuttavasta ’panic handler’ -funktiosta saadaan tieto virheenkorjaimen (engl. *debugger*) kautta toisin kuin C-ohjelmakoodiversiosta, jonka virhettä on hyvin hankala paikallistaa.

Myös hyvä rinnakkaisuuden tuki on tärkeää käyttöjärjestelmän kehityksessä mikäli käyttöjärjestelmää on tarkoitus suorittaa useampiytimisellä suorittimella. Kaikkien käyttöjärjestelmän komponenttien, esimerkiksi kommunikaatiokanavien, tietotyyppien tai ajurien,

tulee olla säieturvallisia.

6 RUST-REAALIAIKAKÄYTTÖJÄRJESTELMÄT

Seuraavaksi esitellään Rust-reaaliaikakäyttöjärjestelmät ja niiden toimintaperiaatteet. Suurin osa viittauksista on kyseisten reaaliaikakäyttöjärjestelmien omiin dokumentaatioihin, koska ne ovat kattavimpia ja varmasti parhaiten ajan tasalla.

6.1 Tock OS

Tock OS on isompi ja monimutkaisempi reaaliaikakäyttöjärjestelmä kuin Drone OS tai RTIC. Tock tukee ohjelmien dynaamista lataamista sekä huolehtii prosessien turvallisuudesta myös silloin kun niitä ajetaan rikkinäisten tai vahingollisten prosessien kanssa samaan aikaan [21].

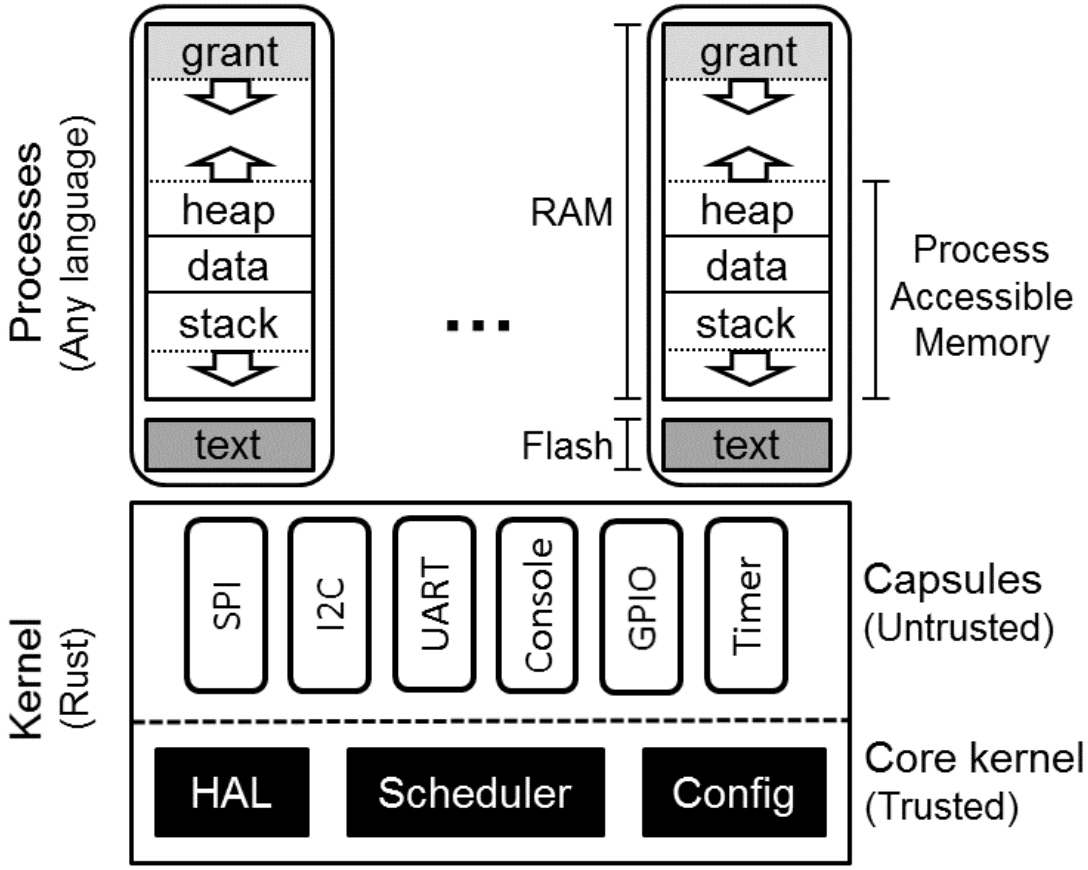
Prosessit ja niille annettavat muistiavaruudet on hyvä tapa erotella tehtävät toisistaan ja tuoda rinnakkaisuutta järjestelmään, mutta vähämuistisissa mikrokontrollereissa tästä tulee kompromissi muistinkulutuksen sekä eristämisen tarkkuuden välille. Tock OS ratkaisee tämän kompromissin käyttäen ohjelmointikielen - eli Rustin - ominaisuuksia eristämään komponentit toisistaan.

Tock OS on kätevä isommissa sulautetuissa järjestelmissä jossa prosessit halutaan erottaa toisistaan kokonaan. Hyvänä esimerkkinä toimii sensoriverkon sulautettu keskuskäsittelija, joka vastaanottaa yhteyksiä ja joutuu ajamaan monia prosesseja, jotka pitää eristää toisistaan. Toinen mahdollinen käyttötapaus olisi järjestelmä, josta halutaan tarjota käyttöön resursseja hallitusti ja turvallisesti, kuten kiihdytin.

Kappale on pitkälti referoitu lähteestä [22].

6.1.1 Arkkitehtuuri

Tock OS koostuu kolmesta tärkeästä palasta: käyttöjärjestelmän ytimestä, kapseleista sekä prosesseista. Ydin ja kapselit on käännetty Rustin tyyppitarkastuksen kanssa, mutta prosessit voivat olla kirjoitettu millä tahansa kielellä. Niiden suojana toimii laitteiston muistinsuojausyksikkö, MPU, taatakseen ajoaikaisen muistinsuojauksen toisilta prosesseilta.



Kuva 6.1. *Tock OS:n arkkitehtuuri [22, Luku 'Architecture']*

6.1.2 Kapseli

Kapseli (engl. *capsule*) on rakenteeltaan Rustin *struct* sekä siihen liitetyt metodit [22, kappale 'Capsules']. Koska kapselit on käännetty käyttäen Rustin tyyppitarkastusta, ovat kapselit tyyppiturvallisia: ne voivat luottaa siihen että viite tiettyyn muistialueeseen sisältää viitteen osoittaman tyypin mukaisen tietueen. Kapselit vuorovaikuttavat suoraan keskenään kutsumalla toistensa julkista rajapintaa. Mikäli halutaan suojata kapselin sisäinen osa toisilta kapseleilta, noudatetaan Rustin tyyppijärjestelmää ja kyseinen tietue merkitään yksityiseksi.

Kapselit suoritetaan ytimen sisällä etuoikeutetussa tilassa mikä käytännössä tarkoittaa sitä, että kapseleilla on täysi pääsy kaikkialle laitteen muistiin - myös ytimen tietueihin. Rustin tyyppijärjestelmä takaa, että rikkinäiset tai haitalliset kapselit eivät pääse haittaamaan käyttöjärjestelmän ydintä. Koska Rustin tyyppitarkastelu on käännösaikaista, ei suoritus hidastu juurikaan tyyppivarmistusten tai virheentarkastelun takia.

Koska kapselit suoritetaan ytimen kanssa samassa yksisäikeisessä tapahtumasilmukassa, kapselin panikointi aiheuttaa virhetilanteen josta ei ole mahdollista selvitä ilman että koko käyttöjärjestelmä - tai laite - uudelleenkäynnistetään.

Taulukko 6.1. *Tock OS:n suojamekanismit [22, Luku 'Architecture']*

Kategoria	**Kapseli**	**Prosessi**
Suojamekanismit	Ohjelmointikieli	Laitteisto
Muistin jakaminen	Sama muistiavaruus	Erillinen pino
Suojauksen karkeus	Hieno	Karkea
Tyhjentävä vuoronnus	Ei	Kyllä
Ajoaikainen päivitys	Ei	Kyllä

6.1.3 Prosessi

Prosessit ovat Tock OSn pääasiallinen tapa käyttää järjestelmää. Ne ovat verrattavissa PC:n käyttäjätason prosesseihin, eli esimerkiksi käyttäjän suorittamiin ohjelmiin Linuxissa. Yleensä uudesta tehtävästä halutaan tehdä Tock OS:ssä prosessi ellei erityinen toive taulukosta 6.1 johdattele tekemään kapselia.

Prosessit ovat paremmin suojattuja kuin kapselit. Jokaisella prosessilla on oma muistialueensa [22, kappale 'Processes'] ja niitä ajetaan rajoitetuilla oikeuksilla omissa säikeissään. Ydin vuorontaa prosesseja tyhjentävästi (engl. *pre-emptive*), mikä tarkoittaa että esimerkiksi ikuiseen silmukkaan jumiutunut prosessi ei jumita koko järjestelmää. Tock käyttää myös laitteistosuojausta valvoakseen prosessien eristämistä ajoaikana. Tämä mahdollistaa ohjelmien kirjoittamisen millä tahansa kielellä ja niiden lataamisen sekä suorittamisen ajoaikana.

Muistin asettelu

MPU suojaa prosessin muistialueen siten että mikään toinen prosessi ei pääse lukemaan tai kirjoittamaan sinne [22, kappale 'Memory Layout']. Mikäli prosessi yrittää vaikuttaa muistialueensa ulkopuolelle, aiheutuu siitä käyttöjärjestelmän ytimeen **ohjelmavirhekeskeytys** eli **ansa** (engl. *trap*).

Prosessin ohjelmakoodi on tallennettu flash-muistiin, mikä on asetettu kirjoitussuojatuksi alueeksi kun taas ohjelman muistialue on varattu yhtenäisenä muistialueena käyttömuistista - kyseisellä alueella on sekä luku- että kirjoitusoikeudet.

Grants

Kapselit eivät saa varata dynaamista muistia suoraan koska mikäli se epäonnistuu, koko käyttöjärjestelmän ydin kaatuu yhden kapselin takia [22, kappale 'Grants']. Kapselit kuitenkin voivat tarvita dynaamista muistia prosessien pyyntöjen seurauksena, minkä vuoksi Tock OS tarjoaa rajoitetun dynaamisen muistinvarauksen niille tarjoamalla muistia prosessin omasta muistialueesta, joka näkyy kuvassa 6.1 prosessin muistialueen ylimpänä lohkona.

Kapseli ei kuitenkaan voi säilyttää viitettä muistialueeseen. Prosessit saattavat kaatua joten viiteen oikeellisuus on tarkistettava ytimen ohjelmakoodissa. Käyttöjärjestelmän ytimen täytyy valvoa kolmea ominaisuutta:

1. Kapselit eivät voi rikkoa tyyppijärjestelmää varatun muistin avulla
2. Kapselit voivat viitata prosessin muistiin vain prosessin ollessa elossa
3. Käyttöjärjestelmän ytimen on pystyttävä vapauttamaan lopetetun prosessin muisti takaisin käyttöönsä.

Muistialueet pakataan tyyppiturvalliseen *struct*-tietokenttään minkä avulla Tock tarkistaa, että prosessi on edelleen elossa ennen viitteen käyttämistä. Muistialue voidaan määritellä minkä tyyppiseksi tahansa, minkä takia itse prosessi, jolta se varattiin, ei voi koskea muistialueeseen jotta tyyppiturvallisuus voidaan taata.

6.1.4 Viestinvälitys

Tock OSssä prosessien välinen viestintä toimii asiakas-palvelin -periaatteella. Jokainen palvelu - eli prosessi - voi toimia yhtenä palvelimena kerrallaan. Toiset prosessit voivat sen jälkeen löytää kyseisen palvelun sen omalla nimellään ja jakaa puskurin (varatun muistialueen) niiden kanssaan. [23]

6.1.5 Laitteistotuki

Tock OS vaatii laitteilta MPUn. Tällä hetkellä Tock OS tukee seuraavia laiteperheitä [24]:

- Vakaa tuki
  - nRF52-tuoteperhe
  - Hail & imix
- Kehitteillä oleva tuki
  - STM-tuoteperhe
  - Aconno
  - TI Launch XL
- Varhaisessa vaiheessa oleva tuki
  - SiFive HiFive1
  - OpenTitan
  - Arty e21
- Vanhentunut tuki
  - nRF51-DK

Uuden laitteen tuen tuominen Tock OS:lle vaatii laitteistoabstraktiokerroksen (HAL) toteuttamisen laitteelle. Minimivaatimuksina ovat GPIO (engl. *General Purpose Input/Out-*

put), ajastin (*engl. Timer*) sekä UART (engl. *Universal Asynchronous Receiver/Transmitter*) [25].

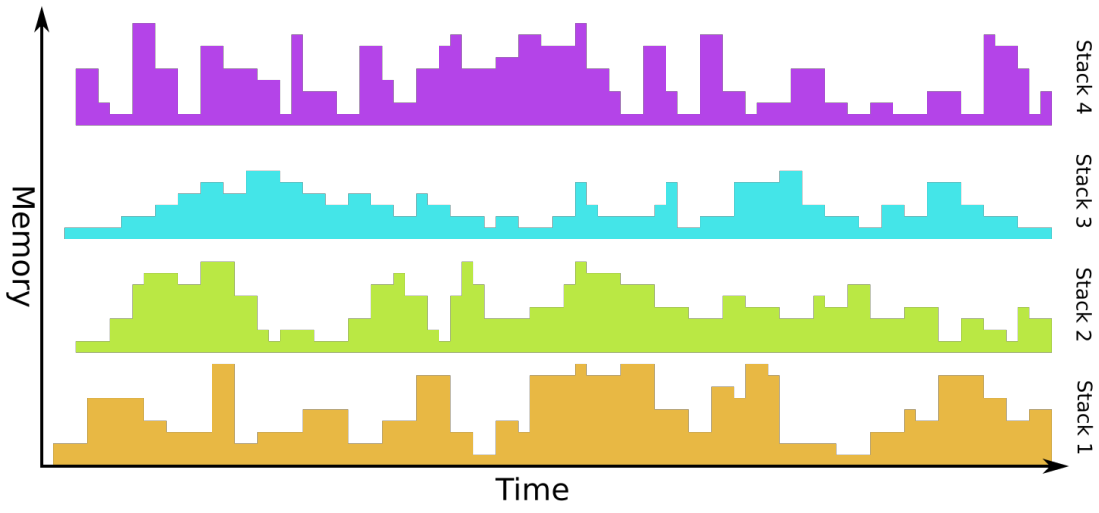
6.2 Drone OS

Drone OS ei tue sovellusten dynaamista lataamista toisin kuin Tock OS mikä tekee siitä huomattavasti yksinkertaisemman kuin Tock OS. Se ei myöskään suorita prosesseja aikalohkoina vaan keskeytysmekanismilla eli korkeamman prioriteetin prosessi voi keskeyttää matalemman prioriteetin omaavan prosessin suorituksen. Drone OS on minimalistinen mikrokontrollerikäyttöjärjestelmä, jonka vahvuus on helppo ohjelmistokehitys. Sille löytyy *cargon* kaltainen komentorivityökalu jolla voidaan helposti luoda uusi projekti ja hallita sitä. [26]

6.2.1 Rinnakkaisuus

Kappale on referoitu pääosin lähteestä [27].

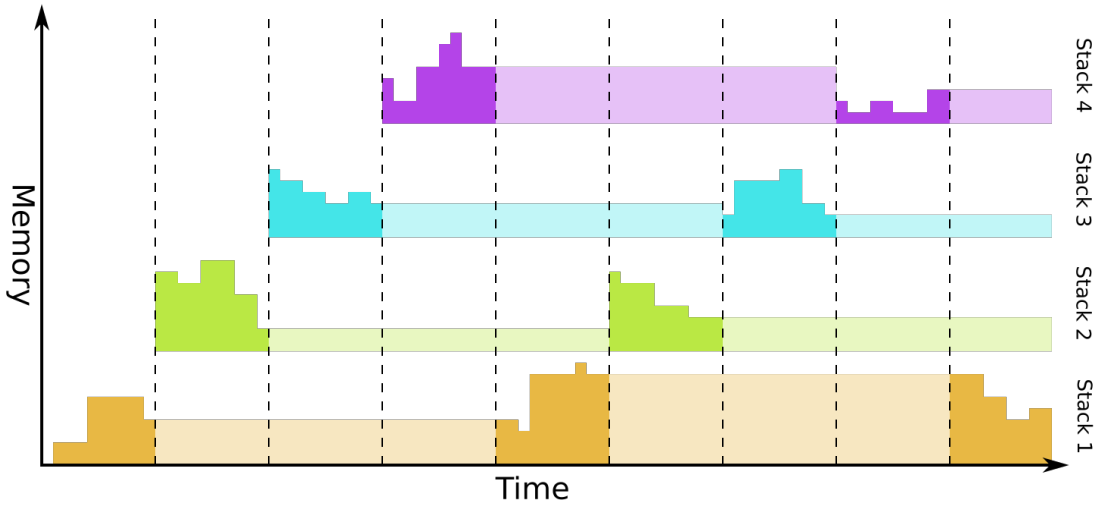
Yleensä vuoronnuksesta puhuttaessa puhutaan kappaleen 2.1.1 mukaisesta vuorontamisesta. Siinä nopea vuorontaminen voi luoda mielikuvan prosessien samanaikaisuudesta kuvan 6.2 mukaisesti jossa jokaisella prosessilla on oma pino ja joita suoritetaan yhtä aikaa eteenpäin. Todellisuudessa laitteisto ei näin kuitenkaan toimi, vaan vuoronnin jakaa prosessorin suoritusajan prosessien kesken kuvan 6.3 mukaisesti. Nopean vuorontamisen ansiosta käyttäjä saa käsityksen, että prosessit olisivat kaikki samanaikaisesti ajossa.



Kuva 6.2. *Rinnakkaisuus käytännössä: useaa prosessia ajetaan samaan aikaan, jokaisella on oma pino [27]*

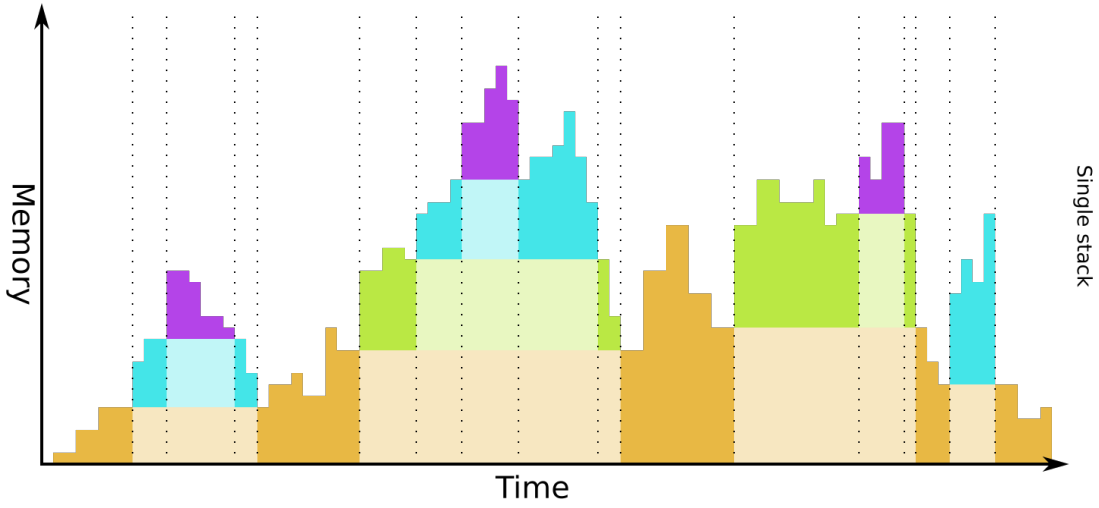
Mekanismi on yksinkertainen ja tehokas työasema- sekä palvelinympäristöissä, mutta vähämuistisissa sulautetuissa järjestelmissä mekanismi on kallis esimerkiksi muistin suhteen. Lisäksi, jotta pinot voidaan suojata toisilta prosesseilta, tarvitaan muistinsuojaus- tai muistinhallintayksikkö (ks. kappale 3). Näiden syiden takia Drone OS käyttää vaihtoehtoista lähestymistapaa vuorontamiseen.

Esimerkiksi Cortex-M mikroprosessoreissa on NVIC-keskeytysohjain, jolla voi toteuttaa tehokkaasti keskeytysohjattua toimintaa. Drone OS käyttää tätä toteuttaakseen priorisoidun tyhjentävän vuorontamisen.



Kuva 6.3. *Rinnakkaisuus teoriassa: jokaista prosessia ajetaan vuorotellen peräkkäin, jokaisella on oma pino [27]*

Tehtävän voi syrjäyttää vain toinen, korkeammalla prioriteetilla suorituksessa oleva tehtävä. Tehtävä joutuu myös kokonaan tyhjentämään pinon omalta osaltaan odottaakseen ulkoista keskeytystä tai resurssia. Näin on mahdollista, että Drone OS käyttää vain yhtä pinoa, minkä takia MPU (tai MMU) ei ole välttämätön) vaikka se onkin hyödyllinen. Pinon asettaminen muistiavaruuden reunalle suojaa muistia lisäksi pinon ylivuodolta.



Kuva 6.4. *Drone OS yhden pinon rinnakkaisuus [27]*

Drone OS:ssä on neljä erilaista rinnakkaisuusmekanismia: **kuidut** (engl. *fibers*), **proses-**

sit (engl. *processes*), **säikeet** (engl. *threads*) sekä **tehtävät** (engl. *tasks*). [27]. Kuidut ovat yksinkertaisimpia, ne ovat tilakoneita jotka toimivat jokseenkin generaattorien tapaan: niillä voi kutsua funktiota `yield` joka palauttaa uuden arvon. [28]

Prosessi on kuidun erikoistuma, joka voidaan pysäyttää. Toisin kuin kuitujen sisällä ajettava koodi, prosessien sisällä oleva koodi ajetaan täysin synkronisesti. Prosesseille varataan erikseen dynaamisesti muistia pinoa varten. Funktiolla `fib::new_proc` luodulla prosessilla muisti on suojattu ylivuodoilta muistinsuojausyksiköllä. Jos laitteessa ei ole MPUta - kuten esimerkiksi piirissä STM32F103 - voidaan käyttää prosessin luomiseen turvattomaksi merkittyä funktiota `new_proc_unchecked`. Toisin kuin generaattorille, prosesseille voidaan antaa parametrejä ja ne voivat myös palauttaa arvon tai olla palauttamatta mitään [29].

Säie on kokoelma kuituja joita keskeytysohjain ohjaa yksitellen. Säikeitä ei voi luoda dynaamisesti vaan ne pitää luoda käännösaikana, mutta niihin voi dynaamisesti liittää kuituja ajoaikana. [30]

Säiettä voi kutsua implisiittisesti lauseella `core::task::Waker`, eksplisiittisesti lauseella `thr.my_thread.trigger()` tai suoraan laitteistokeskeytyksestä. Mikäli ajossa olevalla säikeellä on pienempi prioriteetti kuin uudella säikeellä, syrjäyttää uusi säie vanhan. Muuten se jää odottamaan korkeampiprioriteettisten säikeiden suorittamista loppuun.

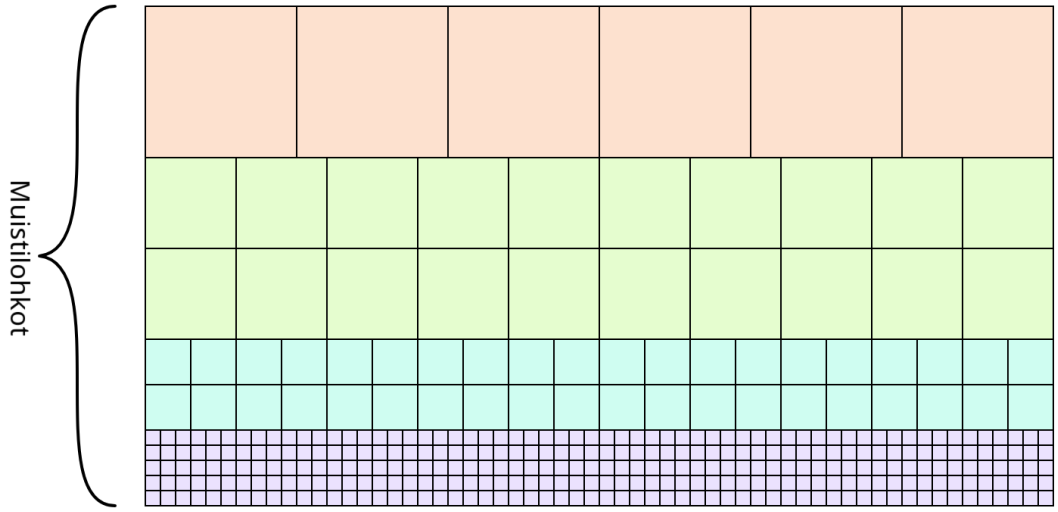
Drone OSn käyttäjätason prosessia kutsutaan tehtäväksi. Jokainen tehtävä kirjoitetaan omaan moduuliinsa ja ne sisältävät ainakin yhden funktion nimeltä `handler` jota lähdetään suorittamaan omassa säikeessään, mikäli se on merkitty avainsanalla `async`. Yleensä tehtävälle varataan yksi käyttämätön keskeytys säikeeksi. [31]

6.2.2 Dynaaminen muisti

Drone OS tarjoaa myös dynaamista muistia toteuttamalla **muistinjakajan** (engl. *allocator*). Sulautetun järjestelmän muistinjakajan on oltava [32]

1. Deterministinen. Reaaliaikakäyttöjärjestelmässä on tärkeää että muistin varaamiseen tai vapauttamiseen kuluu ennalta tiedettävä aika.
2. Pieni muistin suhteen. Sulautettu järjestelmä ei sisällä paljoa muistia ja esimerkiksi *jemalloc* voi olla satoja kilotavuja kun joissain mikrokontrollereissa voi olla vain 64 kilotavua muistia.

Drone OS:n mukana tulee yksinkertainen muistinjakaja joka toteuttaa yllä käsitellyt kohdat. Se jakaa koko keon muistialueen ennalta määrätyn kokoisiin lohkoihin:



Kuva 6.5. *Drone OS:n keko jaettuna muistilohkoihin [32, teksti suomennettu]*

Muistilohkojen koot määritellään käännösaikana *Drone.toml* -tiedostossa. Esimerkiksi:

```
1   [heap.main]
2   size = "10K"
3   pools = [
4       { block = "4", capacity = 118 },
5       { block = "8", capacity = 148 },
6       { block = "20", capacity = 82 },
7       { block = "56", capacity = 34 },
8       { block = "116", capacity = 16 },
9       { block = "208", capacity = 8 },
10      { block = "336", capacity = 3 },
11      { block = "512", capacity = 1 },
12  ]
```

Listaus 6.1. *Drone OS:n Muistilohkojen määrittely Drone.toml -tiedostossa [32]*

Määrittämällä muistilohkojen koot käännösaikana on muistin käyttäminen ajoaikana täysin vakioaikaista. Haittapuolena käännösaikaisessa muistin lohkomisessa on että lohkojen koot täytyy optimoida jokaiselle käyttötapaukselle erikseen. [32]

6.2.3 Viestinvälitys

Drone OS tarjoaa kolmea erilaista viestimekanismia säikeiden väliseen kommunikaatioon: kertajulkaisukanavan (engl. *oneshot*), kehäpuskurikanavan sekä pulssikanavan [33].

Kertajulkaisukanavassa on mahdollista lähettää vain yksi ainoa viesti. Tietorakenne on rakennettu siten, että `send`-funktiota voi kutsua vain kerran ja useamman kerran kutsuminen aiheuttaa käännösvirheen.

Kehäpuskurissa voi lähettää monia viestejä jotka toinen säie vastaanottaa lähettämisjärjestyksessä. Kehäpuskurin alustamisessa voidaan valita sen koko eli maksimimäärä viestejä, jotka voivat olla jonossa kerralla.

Pulssikanava on tarkoitettu toistuvaan ilmoitusten lähettämiseen ilman hyötykuormaa. Toiminta on kehäpuskurin kaltaista, mutta kehäpuskuri on turhan iso rakenne pelkän ilmoituksen lähettämiseen.

```
1  use drone_core::sync::spsc::oneshot;
2
3  // Viestikanavan rakentaminen. rx annetaan toiselle säikeelle.
4  let (tx, rx) = oneshot::channel();
5
6  tx.send(my_message);
7
8  // Viestin vastaanottaminen toisesta säikeestä
9  let my_message = rx.await;
10
```

Listaus 6.2. *Esimerkki oneshot-viestikanavan käytöstä*

Ohjelmakoodista 6.2 näkyy miten kertajulkaisukanavaa käytetään. Kaikki viestikanavat noudattavat hyvin samankaltaista syntaksia: ne alustetaan `channel`-funktiolla, viestin lähettäminen tapahtuu kutsumalla `send`-metodia ja vastaanottaminen tapahtuu asynkronisesti. Viestiä jäädään odottamaan vastaanottavan puolen kutsumassa `await`-lauseessa.

6.2.4 Laitteistotuki

Tällä hetkellä Drone OS tukee seuraavia laiteperheitä [26, kappale 'Supported hardware']:

Tuetut arkkitehtuurit

- ARMv7-M
- ARMv8-M
- RISC-V

Prosessorit

- Cortex©-M3
- Cortex©-M4
- Cortex©-M33
- Nuclei Bumblebee -ytimet

- STM32
- nRF91
- GD32VF103 MCU

Tuki uudelle laittelle

Drone OS vaatii CAS -laitteisto-operaation (compare-and-swap) [26, kappale 'Supported hardware'], koska Drone on riippuvainen hyvästä laitteistotason atomisuuden tuesta.

6.2.5 Drone CLI

Drone OS:lle löytyy myös komentoriviohjelma, jolla voi luoda uuden ohjelman, jonka saa suoraan käännettyä ja ohjelmoitua laitteelle. `drone` myös luo vianetsintäkonfiguraatiotiedoston, jolla voi ajaa esimerkiksi OpenOCDta. [34]

Dronen komentorivityökalussa on myös ominaisuus jolla ohjelman muistinkulutusta voi seurata ja luoda dynaamisen muistin varaamiseen sopivan määrittelytiedoston 6.1.

6.3 RTIC

Real-Time Interrupt-driven Concurrency (RTIC, suom. *Reaaliaikainen keskeytysajettu rinnakkaisuus*) on kirjasto joka tarjoaa rinnakkaisuutta ARM Cortex-M -mikrokontrollereille. RTIC on minimalistinen ja kevyt, erityisesti reaaliaikaisuuden hallintaan suunniteltu käyttöjärjestelmä. Nimensä mukaisesti sen vuoronnus pohjautuu mikrokontrollerin keskeytyskäsittelijöihin minkä vuoksi sillä kirjoitetut ohjelmat voivat suoraan reagoida ulkoa tuleviin keskeytyksiin.

Kappale on pitkälti referoitu lähteestä [35].

6.3.1 Tehtävät

RTICssä rinnakkaisuuden yksikköä nimitetään tehtäväksi (engl. *task*) [36]. Ohjelman suoritus aloitetaan `#[init]`-määreellä merkitystä funktiosta [35, luku '1.3 The init task'], minkä jälkeen siirrytään suorittamaan `#[idle]`-määreellä merkittyä funktiota [35, luku '1.4 The idle task']. Nimensä mukaisesti tämä on tehtävä johon RTIC palaa aina, kun sillä ei ole tärkeämpää tehtävää suoritettavana. Sinne voi esimerkiksi laittaa koodin joka vie laitteiston lepotilaan virran säästämiseksi.

`#[init]`-tehtävän jälkeen vuoronnin alkaa vuorontaa muita tehtäviä eli funktioita, jotka on merkitty `#[task]`-määreellä [35, luku '1.5.2 Software tasks & spawn'].

Kaikki tehtävät toimivatkin keskeytyskäsittelijäfunktioiden tapaan ja ne voidaan myös liittää suoraan laitteistokeskeytyksiin [35, luku '1.5.1 Hardware tasks']. Tämän avulla voidaan esimerkiksi ajaa tehtävää joka kerta kun sarjaväylä vastaanottaa uuden merkin.

Priorisointi

Tehtäville voidaan antaa prioriteetti kokonaislukuna, esimerkiksi `#[idle]`-tehtävällä on alin prioriteetti eli 0. Tehtävien maksimiprioriteetti määräytyy ARM-piirin NVIC-keskeytysohjaimesta. [35, luku '1.5.4 Task priorities']

RTIC vuorontaa tehtäviä tyhjentävästi eli mikäli prioriteetin 1 tehtävä on juuri suorituksessa kun prioriteetin 2 tehtävä alkaa, keskeyttää ydin prioriteetin 1 tehtävän ja alkaa suorittaa korkeamman prioriteetin tehtävää. Matalamman prioriteetin tehtävään siirrytään vasta, kun kaikki korkeamman prioriteetin tehtävät on suoritettu loppuun.

6.3.2 Resurssit

RTIC mahdollistaa järjestelmänlaajuisen muistinvaraamisen `#[local]` ja `#[shared]` -määreillä merkittyjen *struct*ien avulla [35, luku '1.2 Resources'].

`#[local]`-määreellä merkittyjä resursseja voi käyttää vain yhdestä tehtävästä. Määre mahdollistaakin resurssin alustamisen `#[init]`-funktiossa ja sen antamisen eteenpäin

yhdelle tehtävälle, mikä on kätevää esimerkiksi ajureille tai suurille muistialkioille.

Rustin tyyppiturvallisuuden periaatetta noudattaen `#[local]` -määreellisen muuttujan luovuttaminen useammalle tehtävälle tuottaa käännösaikaisen virheen eikä näin mahdollista rinnakkaisuusongelmaa vahingossakaan.

Usean tehtävän kesken jaettuja muistialkioita voi myös varata. Rinnakkaisuusongelman takia `#[shared]`-määreellä merkityillä *struct*eilla on sisäisesti käytössä *mutex*, jolla resurssi voidaan lukita vain yhden tehtävän käyttöön kerrallaan mahdollistaen näin saman resurssin käyttämisen rinnakkaisuusturvallisesti useasta tehtävästä.

6.3.3 Viestinvälitys

RTICssä on myös yksinkertainen viestinvälitysjärjestelmä, tehtäville voidaan määritellä parametrit. Esimerkiksi tehtävälle *foo* voidaan antaa argumentti 3 kutsumalla sitä lauseella `foo::spawn(3)`. Tehtävää luodessa sille voi myös antaa määreen *capacity* joka määrää montako viestiä voi olla jonossa. [35, luku '1.5.3 Message passing & capacity']

6.3.4 Laitteistotuki

Kaikki Cortex-M laitteet ovat täysin tuettuja [36].

7 REAALIAIKAKÄYTTÖJÄRJESTELMIEN VERTAILU

Taulukko 7.1. *Reaaliaikakäyttöjärjestelmien ominaisuudet*

Ominaisuus	**Tock OS**	**Drone OS**	**RTIC**
Ajoaikainen ohjelmien lataaminen	Kyllä	Ei	Ei
Mielivaltainen ohjelmointikieli prosesseissa	Kyllä	Ei	Ei
Tyhjentävä vuoronnus	Kyllä	Kyllä	Kyllä
Viestinvälitys	Kyllä	Kyllä	Kyllä

7.1 Prosessit

7.1.1 Ajoaikainen ohjelmien lataaminen

Vertailtavista käyttöjärjestelmistä ainoastaan Tock OS tukee ajoaikasta ohjelmien lataamista kunten taulukossa 7.1 listataan. Drone OS sekä RTIC ovat staattisempia, niissä ohjelmat on määriteltävä käännösaikana eikä niitä voi ajoaikana ladata tai poistaa.

Tock OS:ssä prosessit voidaan kirjoittaa millä tahansa kielellä kuten kappaleessa 6.1.3 todetaan, koska prosessien muistialueet on laitteistotasolla erotettu toisistaan [37, s. 6].

Drone OS:ssä ja RTIC:ssä prosessit ovat staattisesti linkitettyjä ja täten kirjoitetaan Rustilla. Kappaleessa 6.2.1 kerrotaan, että Drone OS:ssä ei prosesseja voida ladata ajoaikana, mutta kuituja voi liittää säikeisiin dynaamisesti ajoaikana.

7.1.2 Muistinsuojaus

Tock OS tarvitsee MPUn eli muistinsuojausyksikön prosessien muistiavaruuksien suojaamiseen kuten kappaleessa 6.1.3 mainitaan, mikä suoraan rajoittaa laitteita joilla kyseistä käyttöjärjestelmää voidaan käyttää. Siitä hyvästä Tock OS tarjoaa laitteilla, joilla MPU on saatavilla, laitteistotasolla suojatun muistiavaruuden kaikille prosesseille mikä mahdollistaa prosessien kirjoittamisen millä tahansa ohjelmointikielellä. Laitteistopohjainen suojaus on yleisesti hyvä tehokkuuden ja turvallisuuden kannalta mutta se pienentää merkittävästi laitteistovalikoimaa jolla Tock OS toimii.

Drone OS tarjoaa eri vaihtoehtoja riippuen siitä, onko sillä käytössään MMU tai MPU vai ei. Drone OS:ssä tehtävien pino suojataan erilaisella mekanismilla jossa pino on kerrallaan varattu vain yhden tehtävän käyttöön (kappale 6.2.1), mutta kaikki tehtävät käyttävät samaa pinoa. Pino myös asetetaan täysin muistin reunaan suojaan ylivuodolta. Prosesseille taas varataan kaikille oma pino keosta, mutta niiden suojaaminen vaatii MMUn tai MPUn. Mikäli näitä ei ole saatavilla, pitää koodi merkata vaaralliseksi käyttämällä `unsafe`-lohkoa.

RTIC ei tarvitse muistinsuojausyksikköä vaan suojaa muistialueet Rust-kielen mekanismeilla, joista on kerrottu kappaleessa 5.1. Muuttujan luovuttamisesta usealle tehtävälle samanaikaisesti seuraa käännösaikainen virhe.

7.2 Dynaaminen muisti

Tock OS:ssä prosesseilla on laitteistotasolla eristetyt muistialueet (kappale 6.1.3) joten muistin sijoittelu muistuttaa enemmän PC-ohjelman muistialuetta: ohjelmakoodi, pino ja keko ovat erillään. Prosessi voi siis varata omasta keostaan muistia dynaamisesti. Kapselit eivät voi varata muistia dynaamisesti suoraan, mutta ne voivat aina kulloisenkin prosessin kautta saada varattua sitä.

Drone OS toteuttaa oman muistinjakajan jonka kautta on mahdollista varata muistia dynaamisesti (kappale 6.2.2). Muisti on kuitenkin jaettu lohkoihin jo käännösvaiheessa minkä takia sen käsittely ajoaikana on vakioaikaista, mikä on tärkää reaaliaikakäyttöjärjestelmälle.

RTIC:ssä voi varata muistia `#[local]` ja `#[shared]` -määreillä merkittyjen `struct`ien avulla. `#[local]`-määreellä merkittyä `struct`ia voi käyttää vain yhdestä säikeestä, kun taas `#[shared]`-määreellä merkitty `struct` käyttää sisäisesti poissulkevaa säielukkoa, jolloin sitä voi käyttää monesta säikeestä samanaikaisesti kuten kappaleessa 6.3.2 kerrotaan.

7.3 Prosessien välinen viestinvälitys

Kappaleessa 6.1.4 kerrotaan kuinka Tock OS:ssä jokainen prosessi voi rekisteröityä palveluksi jonka jokainen toinen prosessi voi löytää ja jota ne voivat käyttää. Kutsun yhteydessä asiakas voi jakaa palvelun kanssa puskurin jonka kautta tietoa voi jakaa. Tock OS:n virallinen rajapinta on kuitenkin määritelty C-kielellä, mistä johtuen tyyppiturvallisuus ei säily puskuria jakaessa. Rajapinnasta on kuitenkin epävirallinen Rust-versio olemassa. Tyyppiturvallisuuden puute ei kuitenkaan ole Tock OS:n ongelma vaan siinä ajettavien prosessien ongelma. Tock OS:n ydin ei voi kaatua prosessien viestinvälityksen protokollan puutteeseen.

Drone OS:ssä on kolme eri kanavaa: kertajulkaisukanava, kehäpuskuri sekä pulssikanava joita käsitellään kappaleessa 6.2.3. Ne on tarkoitettu yksittäiselle viestille, monelle peräkkäiselle viestille sekä signalointitarkoituksiin ilman välitettävää parametria vastaavasti.

RTIC:ssä on yksinkertainen viestinvälitysjärjestelmä jossa tehtäville voidaan määritellä parametrit ja tehtävät voivat kutsua toisiaan antamalla argumentteja kuten kappeleessa 6.3.3 mainitaan.

7.4 Helppokäyttöisyys

Drone OS vaikutti helpommin lähestyttävältä kuin Tock OS sen yksinkertaisuuden takia. Drone OS:llä on myös komentorivityökalu `drone` jolla voi luoda pohjan uudelle projektille. RTIC vaikuttaa myös helposti lähestyttävältä koska se on pelkkä viitekehys joka toimii vuorontajana.

7.5 Tock OS verrattuna muihin yleisiin reaaliaikakäyttöjärjestelmiin

Kappale on referoitu lähteestä [37, luku 2.2].

Taulukossa 7.2 on Tock OS verrattuna muihin yleisiin sulautettuihin käyttöjärjestelmiin joista Arduino ei ole varsinaisesti käyttöjärjestelmä, koska se ei esimerkiksi tue rinnakkaisuutta. Arduino on enemmän verrattavissa laitteistoabstraktiokerrokseen koska se abstrahioi laitteiston rekisterirajapinnan pois tarjoamalla funktioita, joilla voi suoraan ohjata oheislaitteita.

Luotettavuus on tärkeää sulautetuissa järjestelmissä koska ne ovat usein täysin itsenäisesti toimivia laitteita ilman valvontaa. Muistin loppuminen on yksi vakavimmista virheistä minkä vuoksi esimerkiksi Tiny OS varaa muistia staattisesti pitkään kestäville arvoille, tai FreeRTOS rajoittaa muistinvaraamisen välittömästi käynnistämisen jälkeiselle ajalle.

Taulukko 7.2. *Tock OS verrattuna muihin yleisiin reaaliaikakäyttöjärjestelmiin [37]*

Järjestelmä	Rinnak-kaisuus	Muisti-tehokkuus	Luotet-tavuus	Vian eristä-minen	Ladattavat ohjelmat
Arduino		✓			
RIOT OS		✓			
Contiki	✓	✓			✓
FreeRTOS	✓		✓		
TinyOS	✓	✓	✓		
TOSThreads	✓		✓		✓
SOS	✓	✓			✓
Tock	✓	✓	✓	✓	✓

Rinnakkaisuus auttaa säästämään sähköä, koska päällekäiset I/O-kutsut auttavat laitetta

pysymään pidempään lepotilassa. Tämän takia myös useimmat käyttöjärjestelmät mahdollistavat useiden operaatioiden suorittamisen samanaikaisesti.

Myös käyttömuisti on kallis resurssi. Esimerkiksi Tiny OS laskee staattisesti takaisinkutsufuntioiden (engl. *callback function*) määrän jotta se voi varata juuri sopivan määrän muistia niille. Käyttöjärjestelmät SOS ja TOSThreads tukevat dynaamista ohjelmien lataamista. SOS esimerkiksi tarjoaa muistia jaetusta keosta mikä on tehokasta, mutta se heikentää luotettavuutta koska ohjelma voi kaatua toisen ohjelman takia.

Suurin osa mikrokontrollerikäyttöjärjestelmistä ei tue vian eristämistä koska mikrokontrollerien laitteistotuki tälle on verraittain uusi asia. Ongelmaa on pyritty ratkaisemaan muunmuassa huolellisella rajapintasuunnittelulla ja muistisuoja-alueilla. Jotkut järjestelmät ajavat tavukooditulkkia mikä tarjoaa ohjelmistotason ratkaisun vianeristämiselle.

Tock OS tukee kaikkia näitä viittä edellä mainittua ominaisuutta hyödyntäen niin mikrokontrollerien kuin ohjelmointikielten kehitystä. [37, luku 2.2]

8 YHTEENVETO

Työssä tutustuttiin kolmeen Rust-ohjelmointikielellä kirjoitettuun reaaliaikakäyttöjärjestelmään: Tock OS, Drone OS sekä RTIC. Työssä avattiin niiden toimintaperiaatteet sekä käyttötarkoitukset. Konseptin ymmärtämistä varten selitettiin myös ylipäätään käyttöjärjestelmien toiminnasta, niiden laitteistovaatimukset, miten reaaliaikakäyttöjärjestelmä eroaa tavallisesta käyttöjärjestelmästä sekä tutustuttiin hieman uudehkoon ohjelmointikieleen Rustiin.

Lähteitä työhön löytyi hyvin, Rustiin sekä reaaliaikakäyttöjärjestelmiin käytettiinkin pääasiassa niiden omaa dokumentaatiota sen ollen kaikista tarkinta. Itse reaaliaikakäyttöjärjestelmät ovat kuitenkin vielä niin nuoria että niihin ei löytynyt oikein muita lähteitä kuin niiden omat dokumentaatiot. Käyttöjärjestelmien toiminta yleisesti ei ole muuttunut merkittävästi minkä vuoksi lähde [2] on loistava.

Tock OS on laajin reaaliaikakäyttöjärjestelmä, se myös eristää prosessit laitteistotasolla kuten esimerkiksi Linux. Tämä kuitenkin vaatii laitteistolta MMUn tai MPUn joita ei kaikissa sulautetuissa järjestelmissä ole. Tock OS tukee ainoana vertailtavista prosessien ajoaikaista lataamista. Viestinvälitys on hyvä, mutta koska prosessit voivat olla kirjoitettu millä tahansa kielellä, jaetaan viestinvälityksessä pelkkä muistialue mikä ei ole Rust-tyyppiturvallinen.

Drone OS on helppokäyttöisyyteen pyrkivä reaaliaikakäyttöjärjestelmä joka osaa hyödyntää MPUta mutta toimii myös ilman sitä. Drone OS:n prosessien välinen viestintä pitää sisällään Rustin tyyppiturvallisuuden.

RTIC on yksinkertaisin ja siksi myös helpoiten ymmärrettävissä. Se on reaaliaikavuoronnin joka sisältää tarvittavat komponentit kuten muistin jakamisen sekä yksinkertaisen viestinvälityksen. Sitä voidaan käyttää monella mikrokontrollerilla koska se ei ole riippuvainen MPUsta vaan tarjoaa turvallisuuden kokonaan Rustin tyyppiturvallisuuden kautta.

Vertailtavista reaaliaikakäyttöjärjestelmistä Tock OS osoittautui turvallisimmaksi ja hienostuneimmaksi, Drone OS käyttäjäystävällisimmäksi ja RTIC pienimmäksi ja yksinkertaisimmaksi - myös helpoimmaksi ymmärtää.

LÄHTEET

[1] Microsoft: 70 percent of all security bugs are memory safety issues, 11. helmikuuta 2019, 11. helmikuuta 2019, Saatavilla (viitattu 21.4.2022): https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[2] I. Haikala, *Käyttöjärjestelmät*, Helsinki, 2004.

[3] What for a FreeRTOS is used?, Saatavilla (viitattu 15.9.2021): https://iq.direct/blog/314-what-for-a-freertos-is-used.html.

[4] Interrupt Handlers, Saatavilla (viitattu 22.2.2022): https://tldp.org/LDP/lkmpg/2.4/html/x1210.html.

[5] Atmega 328P datasheet, Rev. 7810D–AVR–01/15, 2015, Saatavilla (viitattu 15.9.2021): https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf.

[6] What is nested vector interrupt control (NVIC)?, 20. maaliskuuta 2020, 20. maaliskuuta 2020, Saatavilla (viitattu 15.9.2021): https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/.

[7] S. Deshmukh, *Hardware Abstraction Layer*, 2022, Saatavilla (viitattu 29. 03. 2022): URL: https://ww1.microchip.com/downloads/en/DeviceDoc/hardware-abstraction-layer.pdf.

[8] Crate embedded$_{h}al$, 2022, 2022, Saatavilla (viitattu 29.3.2022): https://docs.rs/embedded-hal/0.2.2/embedded_hal/.

[9] *STM32 Peripheral Access Crates*, 2022, Saatavilla (viitattu 29. 03. 2022): URL: https://github.com/stm32-rs/stm32-rs.

[10] *svd2rust*, 2022, Saatavilla (viitattu 30. 03. 2022): URL: https://crates.io/crates/svd2rust.

[11] K. Wang, *Embedded and Real-Time Operating Systems*, Cham, 2017.

[12] J. Yiu, *The definitive guide to the ARM Cortex-M3*, Amsterdam ; 2010.

[13] Why Use an RTOS?, 18. lokakuuta 2016, 18. lokakuuta 2016, Saatavilla (viitattu 13.9.2021): https://www.smxrtos.com/articles/Why%20Use%20an%20RTOS.pdf.

[14] RTOS Advantages, Saatavilla (viitattu 14.9.2021): https://www.keil.com/rl-arm/rtx_rtosadv.asp.

[15] J. M. Perkel, Why scientists are turning to Rust, 2022, 2022, Saatavilla (viitattu 18.3.2022): https://www.nature.com/articles/d41586-020-03382-2.

[16] E. Mortoray, Why garbage collection is not necessary and actually harmful, 30. maaliskuuta 2011, 30. maaliskuuta 2011, Saatavilla (viitattu 22.3.2022): https://mortoray.com/2011/03/30/why-garbage-collection-is-not-necessary-and-actually-harmful/.

[17] Rust ownership model, 2022, 2022, Saatavilla (viitattu 18.3.2022): https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[18] Unsafe Rust, 2022, 2022, Saatavilla (viitattu 18.3.2022): https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[19] Rust Fearless Concurrency, 2022, 2022, Saatavilla (viitattu 18.3.2022): https://doc.rust-lang.org/book/ch16-00-concurrency.html.

[20] Unrecoverable Errors with panic!, 2022, 2022, Saatavilla (viitattu 18.3.2022): https://doc.rust-lang.org/book/ch09-01-unrecoverable-errors-with-panic.html.

[21] Tock OS, 2022, 2022, Saatavilla (viitattu 27.3.2022): https://www.tockos.org/.

[22] Tock OS Design, 2021, 2021, Saatavilla (viitattu 25.10.2021): https://www.tockos.org/documentation/design.

[23] Tock OS inter-process Communication, Saatavilla (viitattu 21.4.2022): https://book.tockos.org/tutorials/05_ipc.html.

[24] Tock OS hardware support, 2022, 2022, Saatavilla (viitattu 14.3.2022): https://www.tockos.org/hardware.

[25] Porting Tock OS, 2022, 2022, Saatavilla (viitattu 14.3.2022): https://github.com/tock/tock/blob/master/doc/Porting.md.

[26] Drone OS Introduction, 2022, 2022, Saatavilla (viitattu 15.3.2022): https://book.drone-os.com/.

[27] Drone OS concurrency, Saatavilla (viitattu 23.11.2021): https://book.drone-os.com/concurrency.html.

[28] Drone OS Fibers, Saatavilla (viitattu 2.12.2021): https://book.drone-os.com/fibers.html.

[29] Drone OS Processes, Saatavilla (viitattu 2.12.2021): https://book.drone-os.com/processes.html.

[30] Drone OS Threads, Saatavilla (viitattu 2.12.2021): https://book.drone-os.com/threads.html.

[31] Drone OS Tasks, Saatavilla (viitattu 2.12.2021): https://book.drone-os.com/tasks.html.

[32] Drone CLI, Saatavilla (viitattu 26.5.2022): https://book.drone-os.com/heap.html.

[33] Drone OS Message-Passing, Saatavilla (viitattu 2.12.2021): https://book.drone-os.com/message-passing.html.

[34] Drone CLI, Saatavilla (viitattu 24.5.2022): https://book.drone-os.com/extensibility/cli.html.

[35] RTIC Resource usage, 2022, 2022, Saatavilla (viitattu 30.3.2022): https://rtic.rs/1/book/en/by-example.html.

[36] Real-Time Interrupt-driven Concurrency, 2022, 2022, Saatavilla (viitattu 30.3.2022): https://rtic.rs/1/book/en/.

[37] A. Levy et al., Multiprogramming a 64kB Computer Safely and Efficiently, teoksessa: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP'17, ACM, Shanghai, China, lokakuu 2017, s. 234–251, Saatavilla: https://www.tockos.org/assets/papers/tock-sosp2017.pdf.