Muistinhallinnan tekniikat sulautetuissa järjestelmissä

Turun yliopisto
Tietotekniikan laitos
TkK-tutkielma
Joulukuu 2023
Matias Suksi

TURUN YLIOPISTO
Tietotekniikan laitos

MATIAS SUKSI: Muistinhallinnan tekniikat sulautetuissa järjestelmissä

TkK-tutkielma, 21 s.

Joulukuu 2023

Sovelluksen tehokas muistinkäyttö on tärkeää sujuvan käyttökokemuksen takaamiseksi, ja sulautetuissa järjestelmissä tämä sujuvuus korostuu entisestään. Tämä tutkielma pyrkii etsimään vastauksen tutkimuskysymykseen: "millaisia muistinhallinnan tekniikoita voidaan hyödyntää sulautetuissa järjestelmissä". Tutkielmassa käydään läpi sovelluksen muistin rakennetta, esitellään mitä rajoitteita ja haasteita sulautetut järjestelmät aiheuttavat tehokkaalle muistinhallinnalle.
Tutkielma ei kykene tuottamaan yleistystä, mitkä muistinhallintatekniikka ovat tehokkaimpia sulautetuissa järjestelmissä, vaan tutkielmassa havaitaan, että muistihallintatekniikan sovellus on täysin riippuvainen sovelluskohteesta. Täten myös tutkielmassa esitellyt muistinhallinan tekniikat ovat valittu tutkielmaan lähdekartoituksessa useimmiten vastaan tulleina muistirakenteita, joita sulautettuun järjestelmään on pyritty kussakin artikkelissa soveltamaan. Muitakin muistinhallintatekniikoita ja -rakenteita löytyi paljon ja näitä onkin syytä tutkia mahdollisissa jatkotutkimuksissa.

Asiasanat: sulautettu järjestelmä, muistin allokointi, staattinen allokointi, dynaaminen allokointi, keko, pino, reaaliaikaisuus

Sisällys

1 Johdanto

Sovelluksen tehokas muistinhallinta on keskeisessä osassa sulavan käyttökokemuksen takaamisessa. Sulautettujen järjestelmien tuomat haasteet korostavat muistinhallinnan merkitystä entisestään. Muistin, prosessorin ja laitteiston komponenttien ominaisuuksien rajoitteet asettavat kehittäjälle haasteita, joiden ratkaisut voivat vaatia kehittäjältä hyvinkin kustomoituja ja vaativia ohjelmakoodin rakenteita, jos vertaillaan perinteisille henkilökohtaisille tietokoneille kehitettävien ohjelmien muistin rakennetta.

Tutkielmassa tullaan tutkimaan sulautettujen järjestelmien muistinhallintaa näiden rajoitteiden vaikuttaessa. Päätutkimuskysymyksenä tutkielmassa on "millaisia muistinhallinnan tekniikoita voidaan hyödyntää sulautetuissa järjestelmissä". Tämä tutkimuskysymys tulee käsittämään erilaisten tekniikoiden esittelyn sekä niiden analysointia lähteiden pohjalta, mitä pitää ottaa huomioon tekniikoiden implementoimisessa sulautettuihin järjestelmiin. Toinen tutkimuskysymys tutkielmassa on "miten muistinhallinnan tekniikoiden suorituskykyä mitataan sulautetuissa järjestelmissä". Katsauksessa tullaan käsittelemään sovelluksen muistin ja muistinhallinnan teoriaa, ja perustietoa sulautetuista järjestelmistä. Näiden käsitteiden ymmärtäminen on keskeistä varsinaisten muistinhallinnan tekniikoiden ja rakenteiden ymmärtämisessä.

Tutkielma keskittyy kehittäjän omiin henkilökohtaisiin ratkaisuihin ohjelmointikieli työkalunaan. Tietokoneiden resurssien virtualisointi on yleistynyt nykypäivä-

nä hajautettujen järjestelmien ja pilvipalveluiden tullessa yhä yleisimmiksi, mutta tässä kirjallisuuskatsauksessa rajataan aihepiirin käsittämään perinteiseen RAM-muistin hallintaan liittyviä konsepteja. Virtualisoidun muistin allokoiminen tullaan sivuuttamaan kokonaan. Lisäksi katsauksessa ei tulla esittelemään, kuin ainoastaan pintapuoleisesti, ohjelmointikielien ja kääntäjien sisäisiä muistin allokointiominaisuuksia ja -algoritmeja. Tämä rajaus sivuuttaa muutamia merkittäviä aihepiirejä, kuten mm. roskien keruun ohjelmointikielissä. Katsauksessa on valittu esimerkkiohjelmointikieleksi C konseptien havainnollistamiseksi. Valinta on perusteltua, sillä C on yksi yleisimmistä ohjelmointikielistä sulautetuissa järjestelmissä. Suurimmassa osassa katsauksessa käsiteltävissä artikkeleissa myös implisiittisesti oletetaan, että ohjelmointikieli, jonka ympärille muistirakenteita toteutetaan on juuri C.

Taustoitusta varten tietoa on haettu Google Scholarista, IEEE Xplore:sta sekä ACM Digital Librarysta, ja sitä on haettu hakulauksekkeella: “embedded system” AND (”memory allocat*” OR “memory manag*”) AND (technique* OR method* OR solution*). Varsinaisia muistinhallinnan tekniikoita varten tietoa on haettu myös suoraan konseptien omilla nimillä.

Tutkielman kaksi ensimmäistä lukua ovat taustoittavia teorialukuja, jotka ovat tärkeitä varsinaisen tutkimuskysymyksen takana olevien käsitteiden ymmärtämiseen. Toinen luku on teoriaa sovelluksen muistin, ja sen hallinnasta ja kolmas luku käsittelee sulautettuja järjestelmiä. Neljännessä luvussa pureudutaan tarkemmin varsinaisiin muistinhallinnan tekniikoihin ja esitellään yksityiskohtaisemmin muistirakenteita, ja miten niitä voidaan hyödyntää sulautetuissa järjestelmissä. Viidennessä luvussa tehdään havaintoja näistä muistirakenteista ja mitä näiden toteutuksessa pitää ottaa huomioon sulautetuissa järjestelmissä. Yhteenvedossa kootaan yhteen työn havainnot ja pyritään vastaamaan asetettuun tutkimuskysymykseen.

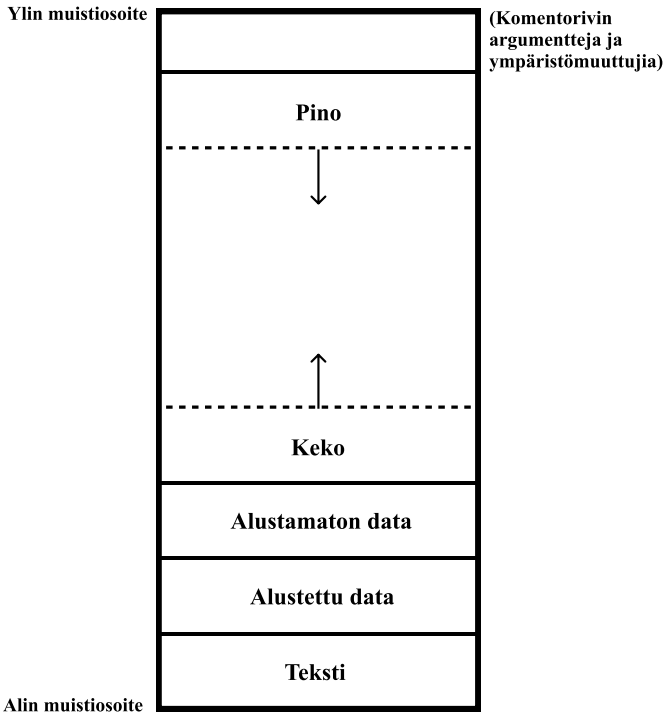
2 Muistinhallinta

Ohjelman muistin rakenteen ymmärtäminen on erityisen tärkeää tehokkaan muistin käytön saavuttamiseksi. Seuraavassa luvussa tullaan esittelemään miten muistin allokointi ohjelmissa toimii ja millaisista muistialueista ohjelman muisti koostuu. Huomioitavaa on, että seuraavaksi esiteltävä sovelluksen muistin rakenne, on tyypillisin malli kuvaamaan, miten tietokoneohjelman muisti koostuu. Ohjelman muistin rakenteeseen vaikuttaa mm. käytössä oleva suoritinarkkitehtuuri, ohjelmointikielen kääntäjä sekä kääntäjien tarjoamat muistin optimointityökalut.

2.1 Ohjelman muisti

Ohjelman suorituksen aikainen muisti jakautuu erilaisiin muistialueisiin. Koodiosa sisältää varsinaisesti ajettavan ohjelman binäärin eli ohjelmatiedoston, jonka prosessori suorittaa. Lisäksi ohjelmalla on olemassa dataosa, joka koostuu alustetusta datasta ja alustamattomasta datasta. Alustetun datan alueeseen kuuluvat globaalit ja staattiset muuttujat sekä vakioarvoiset muuttujat, joille on alustettu jokin arvo. Alustamattomassa data-alueessa on kaikki alustamaton data eli muuttujat, jotka ovat esitelty (engl. *declare*), mutta joille ei ole annettu mitään arvoa. Näiden muistialueiden data allokoidaan ajettavan ohjelman muistiin jo käännön aikana (engl. *compile time*) eli ohjelmointikielen kääntäjän kääntäessä lähdekoodin. Ohjelmalla on lisäksi myös kaksi muuta muistialuetta, pino ja keko. Tyypillisesti pino sijaitsee ylhäällä ja keko alhaalla ohjelman virtuaalisessa muistiavaruudessa.[1] Tällä tarkoi-

tetaan sitä, että pino alkaa numeerisesti suurimmasta muistiosoitteesta, kun taas keko numeerisesti pienimmästä muistiosoitteesta. Pinon datan allokointi tapahtuu jo käännön aikana, kun taas keon datan allokointi tapahtuu vasta ohjelman ajon aikana (engl. *runtime*)[2]. Pinon allokoinnin määrittely ei ole aivan yksikäsitteistä. Yleisesti lähteissä määritellään, että allokointi tapahtuu käännön aikana, mutta jotkin lähteet määrittelevät, että allokointi tapahtuu ajon aikana. Tämä johtuu pinon allokoinnin luonteesta, sillä vaikka pinon hallintaan liittyvät ohjeet syntyvät jo käännön aikana, varsinainen itse datan osoitteistus tapahtuu ajon aikana. Tarkemmin sanottuna heti järjestelmän alustuksen yhteydessä.



Kuva 2.1: Ohjelman muistin rakenne.[1] (suomennettu kuva lähteestä)

2.1.1 Pino

Pino (engl. *stack*) on ohjelman muistialue, johon allokoidaan paikalliset muuttujat, funktioiden parametrit ja paluuosoitteet. Se noudattaa LIFO-periaateetta (engl. *Last In First Out*) eli pinoon viimeiseksi puskettu data poistetaan pinosta myös ensimmäisenä. LIFO-periaatteen ansiosta pinosta allokointi on tyypillisesti nopeampaa kuin kekoon allokoiminen, johtuen tavasta, miten pinon dataan päästään käsiksi. Lisäksi, pinon tavuja käytetään ohjelmassa säännöllisesti yhä uudelleen ja uudelleen, jolloin ne säilyvät hyvin prosessorin nopeassa välimuistissa. Data allokoidaan pinoon automaattisesti ja poistetaan sieltä, kun sen näkyvyysalue päätyy. Pino koostuu kehyksistä (engl. *frame*), joita pusketaan pinoon, kun ohjelma aloittaa uuden funktiokutsun suorittamisen. Tyypillisesti pinon koko on päätetty ennen ohjelman suorituksen aloittamista, ja pinoon allokoitavien muuttujien koko on tiedettävä etukäteen jo käännön aikana.[1] Pino-osoitin (engl. *stack pointer*) pitää yllä tietoa muistiosoitteesta, jossa pinon viimeinen elementti sijaitsee. Tätä osoitinta muuttamalla, ohjelma pitää yllä tietoa mihin uusi kehys lisätään tai mistä vanha kehys poistetaan. Pino-osoittimen arvo pienenee, kun dataa pusketaan pinoon ja vastaisuudessa kasvaa, kun dataa poistetaan pinosta (huomioi pinon sijainti ohjelman muistiavaruudessa, kts. kuva 2.1).[3]

2.1.2 Keko

Keko (engl. *heap*, suom. myös *kasa*) on muistialue, johon kehittäjä allokoi sekä josta myös vapauttaa muuttujat manuaalisesti. Keko sisältää käytettävissä olevien ja vapaiden muistilohkojen linkitetyn listan. Keolle annetaan aloituskoko ohjelman suorituksen alkaessa, mutta muistinvaraaja voi pyytää sitä lisää tarvittaessa käyttöjärjestelmältä. Vastapainona pinolle, keko on hyödyllinen, kun ei voida tietää etukäteen kuinka paljon muistia tarvitsee varata ajon aikana.[1] On syytä mainita, että tietokoneohjelman muistialue keko ei ole sama asia kuin tietorakenne keko.

Ohjelmalistaus 1 on käytännön esimerkki, joka havainnollistaa mihin muistialueeseen kukin koodirivi sijoittuu ohjelman muistissa.

Ohjelmalistaus 1 Demonstraatio muistin allokoinnista C-ohjelmointikielessä

```
#include <stdio.h>
#include <stdlib.h>

//Alustamaton muuttuja --> alustamaton data
int i;
//Alustettu muuttuja --> alustettu data
int n = 1;

//Funktiokutsu --> Pino
int main(void)
{
    //Paikallinen muuttuja --> Pino
    int numero = 10;
    //Dynaamisesti allokoitu muistilohko --> Keko
    int* osoitin = (int*)malloc(n * sizeof(int));
    //Dynaamisesti vapautettu muistilohko --> Vapautettu keosta
    free(osoitin);
    return 0;
}
```

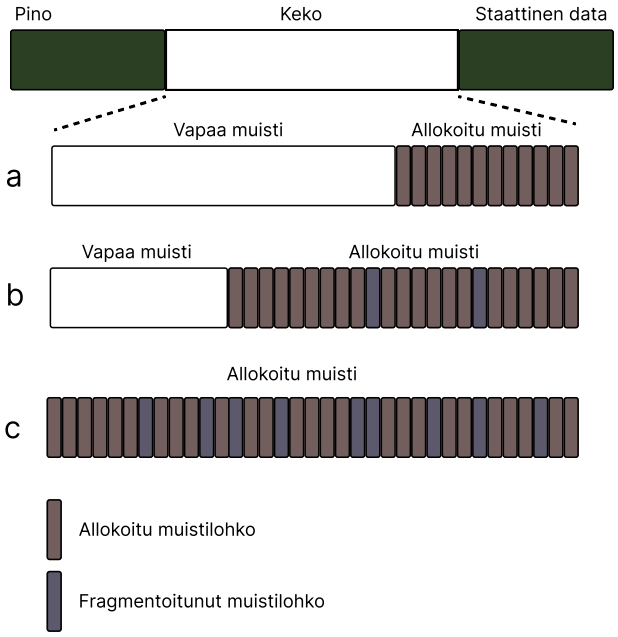
2.2 Muistin allokointi ohjelmointikielissä

Tietokoneohjelmissa hyödynnettävät muistinallokointimenetelmät jaotellaan tyypillisesti staattiseen ja dynaamiseen allokointiin. Staattinen allokointi tarkoittaa muistilohkojen allokointia niin, että lohkot allokoidaan sovellukselle välittömästi järjestelmän alustuksen jälkeen. Näillä lohkoilla on ennaltamääritetty koko, joka määritetään käännön aikana. Tämän jälkeen järjestelmä ei allokoi enää lisää muistia, jollei kehittäjä sitä itse allokoi lisää dynaamisesti, vaan kaikki tehtävät ja prosessit suoritetaan näissä lohkoissa. Staattisen allokoinnin heikkous on muistilohkojen ennaltamäärätty koko. Allokoiduilla lohkoilla on kiinteä koko, jota ei pysty jälkikäteen muuttamaan. Tämä aiheuttaa haasteen, että kehittäjän on tiedettävä sopiva lohkon muistikoko etukäteen. Staattisen allokoinnin vastakohta on dynaaminen allo-

kointi, missä muistilohkot allokoidaan globaalista muistiavaruudesta ohjelman ajonaikana tehtävän tai prosessin koon mukaan. Muistilohko vapautetaan, kun prosessi on valmis. Dynaamisen allokoinnin etuja ovat joustavuus ja muistinkäytön hyötysuhteen paraneminen, mutta dynaaminen allokointi heikentää järjestelmän vakautta.[4]Muistinkäytön hyötysuhteen paranemisella tarkoitetaan, että dynaamisella allokoinnilla muistia voidaan käyttää vain juuri sen verran kuin on pakko. Staattisessa allokoinnissa muistia joudutaan monesti varaamaan hieman ylimääräistä, koska kokoa ei pystytä jälkikäteen muuttamaan. Tämä aiheuttaa sen, että ohjelman ajonaikana staattisesti varattua muistia voi olla paljon käyttämättömänä, koska sitä on varattu varmuuden vuoksi hieman ylimääräistä.

Dynaamista muistinhallintaa varten ohjelmointikielet tarjoavat erilaisia valmiista kirjastoista saatavia funktioita muistin varausta ja vapauttamista varten. C-ohjelmointikielessä funktio, jolla varataan muistia on malloc(), joka ottaa vastaan argumenttina muistin koon tavuina. Muistia vapautetaan funktiolla free(), joka ottaa parametrina osoittimen osoittaman muistilohkon. Lisäksi, on olemassa calloc() ja realloc(), joilla voi myös varata muistia. Calloc() ottaa kaksi argumenttia, joista yksi on varatun muistilohkon koko ja toinen määrittää kuinka monta näitä lohkoja varataan. Malloc()- ja calloc()-funktioiden keskeinen ero on, että calloc()-funktio myös alustaa varatun muistialueen nolliksi. Malloc() ei tätä alustusta suorita, vaan malloc()-funktion varaama muistialue sisältää mielivaltaisia alustamattomia arvoja.[5]Nämä mielivaltaiset alustamattomat arvot ovat siis joitain satunnaisia arvoja, jotka ovat jääneet muistiosoitteeseen edellisestä muistin varauksesta. Realloc() on funktio, jolla pystyy muokkaamaan jo aikaisemmin varatun muistialueen kokoa. Realloc() ottaa argumentikseen muokattavan muistialueen osoittimen sekä muistilohkon uudelleen määritetyn koon[5].

2.3 Muistinhallinnan ongelmatilanteet



Kuva 2.2: Muistin fragmentoituminen.[6] (suomennettu kuva lähteestä)

Muistin fragmentoituminen on muistin tila, jossa usein tapahtuva muistilohkojen allokointi ja vapautus on aiheuttanut sen, että ohjelma ei kykene enää varaamaan käyttöönsä riittävän suurta jatkuvaa muistilohkoa, vaan vapaa muisti on pirstoutunut yksittäisiksi lohkoiksi jo allokoitujen lohkojen välin. Muistin fragmentoitumista pidetään yhtenä sulautettujen järjestelmien muistinkäytön haasteena.[6]Kuva 2.2 havainnollistaa muistin fragmentoitumisen erilaisia tiloja. A-kuva on ideaalitilanne, jossa muisti ei ole olleenkaan fragmentoitunut. B-kuvassa kaksi lohkoa on jäänyt allokoitujen lohkojen väliin. C-kuvassa muisti on jo pahasti fragmentoitunut. Fragmentoituneet lohkot ovat vapaita lohkoja, mutta koska ne eivät muodosta ohjelman muistiavaruudessa jatkuvaa tyhjää tilaa, yksittäisiä vapaita lohkoja suuremmat tietorakenteet eivät pysty tätä tyhjää muistitilaa hyödyntämään.

Muistivuoto on muistinkäytön ongelmatilanne, joka tapahtuu kun ohjelma käyttää muistia, mutta ei kykene vapauttamaan sitä takaisin käyttöjärjestelmän käyt-

töön. Muistivuodot ovat hyvin ikäviä ongelmatilanteita, sillä niiden jäljittäminen vaatii pääsyn ohjelman lähdekoodiin, ja monesti muistivuodot ilmenevät ohjelmassa lukuisina muina ongelmina. Usein ajatellaan virheellisesti, että yleisesti ohjelman lisääntynyt muistinkäyttö on muistivuoto, vaikka tämä ei pidä paikkaansa. Monesti muistivuodot eivät ilmene ohjelmaa ajettaessa välittömästi, vaan hitaasti ohjelman ajon aikana, kun ohjelma varaa yhä enemmän ja enemmän muistia. Lopulta tämä ilmenee ohjelman kaatumisena tai käyttöjärjestelmän hidastumisena.[1]

Dynaaminen muistinhallinta tuo kehittäjälle vapautta, mutta myös suuren vastuun. Aikaisemmin mainitut muistinkäytön ongelmatilanteet ja virheet voivat aiheuttaa päänvaivaa kokemattomalle kehittäjälle, mutta kokeneelle kehittäjälle osoittimet ja manuaalinen allokointi tarjoavat tehokkaat työkalut sovelluksen muistinkäytön tehostamiselle. Ohjelmointikielissä, jotka tarjovat kehittäjälle dynaamisen muistinhallinnan mahdollisuuden, kehittäjän on hyvin tärkeää ymmärtää, miten ohjelman muisti toimii, jotta näiltä ongelmatilanteilta vältytään.

3 Sulautetut järjestelmät

3.1 Mikä on sulautettu järjestelmä

Sulautetulle järjestelmälle on hyvin vaikeaa antaa yksikäsitteistä määritelmää, mutta yleisesti sulautetuilla järjestelmillä viitataan näkymättömiin ja ubiikkeihin tietokoneisiin, jotka ovat suunniteltu jonkin spesifisen toiminnallisuuden suorittamiseen. Esimerkkejä sulautetuista järjestelmistä ovat mm. autot. Autoissa polttoaineen syöttöä, automaattista jarrutusjärjestelmää ja navigointijärjestelmää ohjaa loppujen lopuksi tietokone.[7] Nämä esimerkit symboloivat hyvin sulautetun järjestelmän yleistä kuvausta. Perinteistä henkilökohtaisia tietokonetta käyttäessään, käyttäjä on käytännössä jatkuvasti tietoinen siitä, että hän käyttää tietokonetta, mutta autoa ajaessaan kuljettaja ei tätä jatkuvasti tiedosta eikä hänen tarvitse sitä tiedostaa.

3.1.1 Reaaliaikanen sulautettu järjestelmä

Reealiaikainen sulautettu järjestelmä (engl. *real-time embedded system*) on yleinen käsite johon törmää usein sulautetuista järjestelmistä puhuttaessa. Reaaliaikainen sulautettu järjestelmä on sulautettu järjestelmä, joka vastaa järjestelmän ulkopuolisiin tapahtumiin reaaliajassa. Tämä tarkoittaa, että sulautettu järjestelmä kykenee havaitsemaan ulkoisen tapahtuman, pystyy reagoimaan ja prosessoimaan tapahtuman sekä tuottamaan tarvittavan tuloksen tietyssä aikarajassa.[7] Reaaliaikaisuuden käsite on hyvin tärkeä tässä tutkimuksessa, sillä lähdekartoitusta tehdessä ja

artikkeleita tutkiessa on huomattu, että suurin osa tutkielman aihepiiriin liittyvissä artikkeleissa muistinhallinnan tekniikoita pyritään toteuttamaan nimenomaan juuri reaaliaikaiseen sulautettuun järjestelmään.

3.2 Sulautettujen järjestelmien haasteet

Sulautettujen järjestelmien periaate tietokoneesta, joka on luotu nimenomaisesti jonkin spesifin toiminnallisuuden suorittamiseen tehokkaasti, aiheuttaa rajoitteita järjestelmän toteutuksen suunnitteluun. Keskeisiä rajoitteita ovat mm. tuotantokustannukset, virrankulutus ja tietokoneen fyysinen koko. Lisäksi itse sulautettujen järjestelmien kehittäjän on omattava monipuolisesti osaamista monilta eri teknologioiden osa-alueilta.[7] Aikaisemmin mainitut rajoitteet aiheuttavat sen, että sulautetuissa järjestelmissä monesti muistin määrä on hyvin rajallinen, ja muistinhallinnan mahdollistavien komponenttien ominaisuudet ovat hyvin rajallisia. Lisäksi, sulautetulle järjestemälle asetetut vaatimukset, kuten aikaisemmin mainittu reaaliaikaisuus, jo valmiiksi rajoitetuilla resursseilla, tekevät kehitystyöstä entistä haastavampaa. Nämä haasteet ovat keskeinen osa tutkielman analyysiä, ja muistinhallinan tekniikoita tullaan peilaamaan juuri näiden rajoitteiden aiheuttamiin haasteisiin.

Perinteisten henkilökohtaisien tietokoneiden prosessorien rakenne on monimutkainen, ja ne tarjoavat monipuolisesti ominaisuuksia monipuolisten tehtävien suorittamisen. Sulautetuissa järjestelmissä prosessorit ovat usein huomattavasti yksinkertaisempia, sillä ne ovat optimoitu juuri tietyn tehtävän suorittamista varten. Modernit prosessorit sisältävät hyvin usein sisäänrakennetun muistinhallintayksikön (engl. *memory management unit, MMU*), joka suojaa muistia ja tarjoaa virtuaalimuistin moniajoa varten. Sulautetun järjestelmän prosessorissa muistinhallintayksikköä ei välttämättä ole ollenkaan.[7] Tämä on esimerkki rajoitteesta, jolloin ei ole riittävää pohtia ainoastaan menetelmiä miten muistinkäyttöä voitaisiin tehostaa järjestelmän nopeuden ja muistin rajallisuuden puolesta, vaan nyt kehittäjä joutuu pohtimaan

ratkaisua komponenttien rajallisten ominaisuuksien näkökulmasta.

4 Muistinhallinan tekniikoita ja rakenteita

Tässä luvussa tullaan esittelemään yleisiä muistinhallinan tekniikoita, joita voidaan hyödyntää sulautetuissa järjestelmissä.

4.1 Rengaspuskuri

Rengaspuskuri (engl. *circular buffer* on järjestetty tietorakenne, jossa viimeisen alkion jälkeen palataan takaisin ensimmäiseen alkioon. Yleensä rengaspuskuri toteutetaan, joko järjestettynä taulukkona tai linkitettynä listana, jonka viimeinen alkio osoittaa takaisin ensimmäiseen alkioon. Rengaspuskurin etuindeksi (engl. front index) osoittaa tyhjän paikan, johon seuraavaksi lisättävä alkio laitetaan. Takaindeksi (engl. back index) osoittaa seuraavaksi poistettavan alkion paikan. Rengaspuskurit ovat erittäin yleisiä tietorakenteita juuri reealiaikaisissa sulautetuissa järjestelmissä, joissa useat prosessit kommunikoivat keskenään. Rengaspuskuri toimii väliaikaisena muistina prosesseille, jolloin prosessit voivat toimia asynkronisesti toistensa suhteen.[5]

Seuraavaksi esitellään yksinkertaisen kokonaislukuja sisältävän rengaspuskurin totetus.

Ohjelmalistaus 2 Rengaspuskurin implementaatio

```
typedef struct RengasPuskuri_t {
    int* taulukko;  //Osoitin taulukkoon
    int koko;       //Maksimikoko
    int alkioiden_lukumaara;    //
    int ensimmainen_alkio;  //Indeksi puskurin alkuun
    int viimeinen_alkio;    //Indeksi puskurin loppuun
} RengasPuskuri;
```

4.2 Segmentoitu pino

Suurin osa ohjelmista käyttävät aikaisemmassa luvussa esiteltyä pinoa yhtenä muistirakenteena. Tällaista pinoa voidaan tarkemmin nimittää jatkuvaksi pinoksi (engl. *contiguous stack*), sillä tämäntapainen pino varaa yhtenäisen muistialueen ohjelman muistiavaruudesta. Tällaisella perinteisellä jatkuvalla pinolla on kuitenkin heikkouksia sulautetuissa järjestelmissä. Muisti allokoidaan pinolle staattisesti eli jo ennen ohjelmakoodin kääntöä, jolloin pinon maksimikoko ajonaikana on ennaltamääritety. Lisäksi jatkuvan pinon ylivuoto (engl. *stackoverflow*) on vaikea havaita, sillä monesti sulautetuissa järjestelmissä käytettävissä matalan tason mikrokontrollereista puuttuu kokonaan muistinsuojausyksikkö (engl. *MPU, memory protection unit*), jolla ylivuoto voitaisiin havaita. Lisäksi muut vaihtoehtoiset menetelmät ylivuodon tunnistamiseen ovat monesti muistinkäytön tehokkuuden kannalta hyvin tehottomia, ja joita monet yleiset sulautetut ohjelmointikielien kääntäjät eivät tue tai tukevat hyvin heikosti. Näistä kääntäjistä esimerkkejä ovat mm. ARM GNU ja LLVM.[8] Tarkennettuna ARM GNU:lla viitataan GNU GCC -kääntäjän optimointiin ARM-prosessoreilla eikä tätä pidä sekoittaa ARM GNU Toolchain:iin, joka on huomattavasti laajempi kokonaisuus kuin pelkkä ohjelmointikielenkääntäjä.(kts. [9])

Jatkuvan pinon vastapainona on segmentoitu pino (engl. *segmented stack*), joka eroaa jatkuvasta pinosta niin, että pino on jaettu pienempiin erillisiin pienempiin pinoihin (engl. *stacklet*), jolloin säikeet ja funktiot sekä niiden data allokoidaan omissa pinoissaan. Nämä pienemmät pinot allokoidaan dynaamisesti keosta. Seg-

mentoitu pino on harvoin käytetty muistirakenne, koska sillä on tunnetusti huono suorituskyky ja muistinkäyttö on tehotonta. Kuitenkin, monissa mikrokontrolleripohjaisissa järjestelmissä, joita sulautetuissa järjestelmissä paljon käytetään, nämä suorituskykyongelmat häviävät. Lisäksi pinon ylivuototilanteet on helppo selvittää, sillä pääpinon sisällä olevaa pinoa voidaan dynaamisella allokoinnilla kasvattaa tarpeen vaatiessa. Ajonaikainen kirjasto, joka allokoi ja vapauttaa segmentoidun pinon pienempiä pinoja, kohtelee tätä muistirakennetta linkitettynä listana.[8]

Seuraavaksi tullaan perustelemaan miksi segmentoidun pinon aikaisemmin mainitut heikkoudet häviävät.

- Muistin fragmentoituminen vähenee: Mikrokontrolleripohjaisissa systeemeissä on usein tarpeetonta varata suurta vapaata tilaa segmentoituun pinoon, sillä usein suoritettavien ohjelmatiedostojen vedokset (engl. *image*) linkitetään kokonaan käännön aikana, jolloin dynaamista linkitystä ei tapahdu. Lisäksi, takaisin kutsufunktiot voivat toimia erillisissä segmentoidun pinon pienissä pinoissa, koska käyttöjärjestelmäydin (engl. *kernel*) on tietoinen segmentoidusta pinosta.

- Koodikannan kontrolli: Hyvin usein kehittäjillä on täysi pääsy koko ohjelman lähdekoodiin, jolloin segmentoitu pino on helppo implementoida koko järjestelmään eikä kehittäjän tarvitse tehdä toteutusta, joka toimisi yhdessä jatkuvan pinon kanssa. Tilanteissa, jossa ulkoinen oheislaitteiden toimittaja tarjoaa binäärikirjastoja laitteilleen, on helppoa tarjoata vain segmentoidulla pinolla käännetty versio.

- Suorituskyvyn heikentyminen on hyväksyttävämpää: Monesti mikrokontrolleripohjaisissa järjestelmissä hyvä suorituskyky ei synny pelkästään puhtaasta nopeudesta, vaan järjestelmän tilan ennustettavuudesta. Lisäksi nopeuden

menetyksestä syntävät energiakustannukset voidaan kompensoida käyttämällä pienempää muistia.

[8]

Segmentoitu pino tuo esiin uusia mahdollisuuksia ja hyötyjä sulautetuissa järjestelmissä.

- Muistinkäytön turvallisuus voidaan saavuttaa ohjelmointikielen kääntäjällä: Monesti sulautettujen järjestelmien mikrokontrollerit eivät sisällä muistinhallintayksikköä tai edes muistinsuojausyksikköä. Tällöin järjestelmän muistiosoitteavaruutta ei ole virtualisoitu, vaan järjestelmän tehtävät käyttävät fyysisiä muistiosoitteita eikä pinon ylivuodolle ole automaattista suojautumista. On suuri riski, että yhden pinon ylivuoto helposti korruptoi huomaamatta toisen pinon dataa toisessa tehtävässä. Muistiturvallisuus voidaan tässä tilanteessa pyrkiä saavuttamaan kääntäjän tarjoamilla pinon testauskäskyillä. Ennen funktioiden ajoa, nämä testauskäskyt testaavat pinokehyksien välisiä muistiosoitteita kirjoittamalla niihin tasaisten intervallien välein. Tämä menetelmä on tehokas segmentoidussa pinossa, kun pinokehykset ovat tarpeeksi isoja.

- Laitteisto- ja energiatehokkuus: Segmentoitu pino mahdollistaa staattisen RAM-muistin (lyh. *SRAM*) vähentämisen järjestelmässä, koska staattista RAM-muistia voidaan jakaa pinojen välillä väliaikaisesti. Staattisen RAM-muistin vähentämisellä on kaksi etua. Järjestelmän laitteiston valmistaminen on halvempaa, koska muistia on vähemmän, ja koska muistia on vähemmän niin järjestelmän virrankulutus pienenee. Myös, aikaisemmin mainittu usein jatkuvan pinon varaama tyhjä käyttämätön tila kuluttaa turhaa energiaa.

[8]

4.3 Buddy-algoritmi

Buddy-algoritmi (engl. *buddy algorithm*, myös *buddy memory allocation*) on muistinhallintatekniikka, joka yhdistää ja jakaa muistilohkoja tarpeen vaatiessa. Jos jokin tehtävä vaatii suurta muistilohkoa, buddy-algoritmi yhdistää lohkot, joilla on peräkkäiset muistiosoitteet. Vastapuolisesti se jakaa suuren lohkon, ja antaa siitä osan tehtävällä joka pyytää käyttöönsä lisää muistia. Alkuperäisessä buddy-algoritmissa on ongelmana se, että kun yksi tehtävä vapauttaa lohkon, algoritmi pyrkii yhdistämään lohkon välittömästi muiden lohkojen kanssa. Kun yhdistäminen on tapahtunut ja jokin tehtävä pyytää uudelleen samaa lohkokokoa eikä sen kokoista lohkoa ole vapaana saatavilla, buddy-algoritmi joutuu jakamaan saman lohkon uudelleen. Kun tätä edestakaisin tapahtuvaa jakamista ja yhdistämistä tapahtuu usein, järjestelmän suorituskyky alkaa heikentymään, joka on hyvin huono asia sulautetussa järjestelmässä. Kuitenkin, buddy-algorimistä on useita versioita ja yksi näistä onkin hyvin käyttökelpoinen sulautetussa järjestelmässä. Tämä on niin sanottu laiska buddy-algoritmi. Laiska buddy-algoritmi (engl. *lazy buddy algorithm*) on buddy-algroitmi, joka viivästyttää yhdistämisen ajankohtaa, kun tehtävä on vapauttanut lohkon. Yhdistämisprosessi tapahtuu ainoastaan silloin, kun se on aivan välttämätöntä. Tämä parantaa suorituskykyä, sillä nyt tätä edestakaisin tapahtuvaa yhdistämistä ja jakamista tapahtuu huomattavasti vähemmän.[10]

5 Muistinhallintatekniikoiden analysointi ja suorituskyvyn mittaaminen sulautetuissa järjestelmissä

Tässä luvussa pyritään pyritään summaamaan edellisen luvun menetelmiä sekä esitellään miten sulautetun järjestelmän muistinkäyttöä voidaan mitata ja arvioida.

5.1 Tekniikoiden analysointi

Monet viitatut ja lähdekartoituksessa tutkitut artikkelit lähtevät lähtökohdasta, että halutaan toteuttaa staattinen menetelmä, sillä ne ovat tunnetusti nopeampia muistinkäytön kannalta kuin dynaamiset menetelmät. Staattisen allokoinnin suurin etu on se, että sillä kyetään vastaamaan reaaliaikaisuuden asettamiin haasteisiin parhaiten ja staattiset menetelmät ovat stabiilimpia kuin dynaamiset[4]. Kuitenkin monesti joudutaan tyytymään dynaamiseen ratkaisuun, sillä staattisen allokoinnin luonne aiheuttaa liian paljon rajoituksia tehokkaan ratkaisun tuottamiseen. Täten käsitellyt artikkelit ja niin myös tämä tutkielma, keskittyy pääasiassa dynaamisiin menetelmiin sekä niiden kehittämiseen ja optimointiin. Tutkielmaa varten lähteistöä staattisista muistinhallintatekniikoista sekä hybridimalleista, jotka koostuivat

dynaamisista ja staattisista piirteistä, löytyi, mutta ne sivuutettiin tutkielmasta niiden vaativuuden sekä kompleksisuuden takia.

Staattiset allokointimenetelmät ennaltaehkäisevät muistin fragmentoitumista[4]. Tämä fragmentoitumisen vähentäminen oli artikkeleissa yhtenä vallitsevana teemana, ja tämä tuntuu olevan keskeisenä ongelmana tehokkaan muistinhallinan kehittämisessä sulautetuissa järjestelmissä. Monissa artikkeleissa, jotka käsittelivät dynaamisia menetelmiä, fragmentoituminen listattiin merkittävänä ongelmana ja käsiteltävää dynaamista menetelmää esiteltiin siinä valossa, että miten menetelmä kykenee vastaamaan fragmentoitumisen ongelmiin. Lisäksi tutkielmassa löydettiin uusia keinoja mm. miten ohjelmointikielenkääntäjän avulla voidaan tehostaa sovelluksen muistinkäyttöä.

5.2 Suorituskyvyn mittaaminen

Seuraavaksi esitellyt muistinkäytön tehokkuuden mittarit ovat työkaluja erityisesti reaaliaikaisten sulautettujen järjestelminen dynaamisen muistinkäytön mittaamiseen, mutta nämä ovat muutenkin varsin käyttökelpoisia yleisellä tasolla sulautetuissa järjestelmissä (kts. luku 3.1.1).

- A. Suurimman-Pienimmän lohkon mittari (engl. *Smallest-Biggest Block Metric (SBBM)*): Kun sovellus saa pyynnön allokoida muistia, suurin huoli on löytää pyynnöllä tarpeeksi suuri yhtäjaksoinen muistilohko. Fragmentoitumisen takia on mahdollista, että tarpeeksi suurta vapaata muistilohkoa ei ole saatavilla, vaikka järjestelmän yhteenlaskettu vapaa muisti olisikin huomattavasti suurempi kuin pyydetyn muistilohkon koko. SBBM-mittari mittaa ohjelman ajon aikana suurimpien vapaiden lohkojen kokoja, ja lopuksi mittauksen jälkeen, se valitsee näistä lohkoista pienimmän lohkon.

- B. Vapaan lohkon mittari - Keskimääräisen koon mittari (engl. *Free Block Met-*

ric - Average Size (FBM-AS)): FBM-AS -mittari ilmaisee ajonaikana vapaiden lohkojen koon keskiarvon.

- C. Sisäinen fragmentoituminen (engl. *Internal Fragmentation (IF)*): IF-mittari mittaa muistin tuhlausta kun muistipyyntöön vastataan suuremmalla lohkolla kuin on välttämätöntä. Muistin tuhlaus on sisäistä suhteessa allokoituun lohkoon. Nythän muistia ei ole pirstoutunut koko ohjelman muistiavaruudessa olevien lohkojen väliin, vaan se on tuhlaantunut itse muistilohkon sisään. Siksi tätä sanotaan sisäiseksi fragmentoitumiseksi.

- D. Kustannuksen mittari (engl. *Cost Metric (CM)*): CM-mittari mittaa järjestelmän tehokkuutta suhteessa saatavilla olevien resurssien määrään. CM-mittari mittaa kuinka paljon muistia järjestelmä vaatii sovelluksen tärkeimpien toiminnallisuuksien suorittamiseen.

- E. Suorituskyvyn mittari (engl. *Performances Metric (PM)*): PM-mittari mittaa muistinhallintajärjestelmän suorituskykyä. Tämä mitataan laskemalla kuinka monta skannausta järjestelmä tarvitsee muistilohkoihin käsiksi pääsyyn. Eli tarkemmin tämä mittari mittaa kuinka nopeasti sovellus löytää pyyntöä vastaavan tarpeeksi ison muistilohkon järjestelmästä, kuinka nopeasti lohko voidaan vapauttaa ja asettaa takaisin oikealle paikalleen vapaiden lohkojen listaan.[11]

6 Yhteenveto

Tutkielma esittelee sovelluksen muistin teoriaa, joka on johdattelua itse sovelluksen muistinhallintaan. Seuraavaksi tutkielmassa esitellään ja määritellän sulautetun järjestelmän käsite, ja tuodaan esille mitä haastetia sulautetut järjestelmät aiheuttavat kehittäjälle. Tutkielma päättyy erilaisten muistinhallintatekniikoiden esittelyyn, niiden analysointiin ja suorituskyvyn mittaamiseen esittelyyn. Tutkielman edetessä huomataan, että staattinen muistinhallinta aiheuttaa tietynlaisia rajoitteita, jonka takia monesti jäädään keskittymään dynaamisien menetelmien optimointiin.

Tutkielmassa ei kyetty antamaan yhtä oikeaa vastausta päätutkimuskysysmykseen: "millaisia muistinhallinnan tekniikoita voidaan hyödyntää sulautetuissa järjestelmissä". Sulautettuja järjestelmiä on paljon erilaisia ja oikeanlainen tehokas muistinkäytön ratkaisu on täysin tapauskohtainen. Ratkaisua suunnitellessa on otettava huomioon järjestelmän laitteisto, vaatimukset ja käyttökohde. Vaikka tutkielma ei kyennyt antamaan yksikäsitteistä vastausta ongelmaan, tutkielma esittelee faktoja mitä pitää ottaa huomioon muistinhallintatekniikoiden soveltamisessa sulautetuissa järjestelmissä, ja tuo esiin mahdollisia eteentulevia haasteita.

Tukielman aiheeseen liittyvää käsitteistöä on hyvin paljon, minkä takia tämä tutkielma on hyvin teoriapainoitteinen pintapuolinen katsaus aiheeseen. Tutkielmaa voisi laajentaa tulevaisuudessa mm. kokeellisella tutkielmalla, jossa testattaisiin eri muistinhallintatekniikoiden suorituskykyä erityppisissä sulautetuissa järjestelmissä, ja toteuttamalla yksityiskohtaisempaa analyysiä muistinhallinantekniikoista.

Lähdeluettelo

[1] L. Ferres, "Memory management in C: The heap and the stack", Luentomuistiinpanot, 2010.

[2] D. A. Alonso, S. Mamagkakis, C. Poucet et al., *Dynamic Memory Management for Embedded Systems*. Springer Cham, 2015, s. 1–49.

[3] H. Erives, "The stack and the stack pointer", http://www.ee.nmt.edu/ erives/308L_05, Luentomuistiinpanot, 2006.

[4] H. Zhe, Z. Jun ja L. Xiling, "Design and Realization of Efficient Memory Management for Embedded Real-Time Application", teoksessa *2006 6th International Conference on ITS Telecommunications*, 2006, s. 174–177. DOI: 10.1109/ITST.2006.288827.

[5] T. Bailey, *An Introduction to the C Programming Language and Software Desing*. Sydney: The University of Sydney, 2015, s. 73–84.

[6] I. Deligiannis ja G. Kornaros, "Adaptive memory management scheme for MMU-less embedded systems", teoksessa *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, 2016, s. 1–8. DOI: 10.1109/SIES.2016.7509439.

[7] Q. Li ja C. Yao, *Real-Time Concepts for Embedded Systems*. San Francisco, CA 94107 USA: CMP Books, 2003, s. 9–27.

[8] Z. Ma ja L. Zhong, "Bringing Segmented Stacks to Embedded Systems", teoksessa *HotMobile '23: Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications*, 2023, s. 117–123.

[9] *ARM GNU GCC, GCC-kääntäjän optimointi ARM-prosessoreille*, GNU GCC:n virallinen dokumentaatio. url: `https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html`.

[10] X.-H. Cheng, Y.-m. Gong ja X.-z. Wang, "Study of Embedded Operating System Memory Management", teoksessa *2009 First International Workshop on Education Technology and Computer Science*, vol. 3, 2009, s. 962–965. DOI: `10.1109/ETCS.2009.753`.

[11] C. D. Rosso, "The Method, the Tools and Rationales for Assessing Dynamic Memory Efficiency in Embedded Real-Time Systems in Practice", teoksessa *2006 International Conference on Software Engineering Advances (ICSEA'06)*, 2006, s. 56–56. DOI: `10.1109/ICSEA.2006.261312`.