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, 14 s.

Joulukuu 2023

Asiasanat: sulautettu järjestelmä, muistin allokointi, pino, keko, osoitin

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 rajallisuus asettavat kehittäjälle haasteita, joiden ratkaisut voivat vaatia kehittäjältä hyvinkin kustomoituja ja vaativia ohjelmakoodin rakenteita, jos vertaillaan perinteisille henkilökohtaisille tietokoneille kehitettävien ohjelmien muistin rakennetta.

Kirjallisuuskatsauksessa tullaan tutkimaan sulautettujen järjestelmien muistinhallintaa näiden rajoitteiden vaikuttaessa. Päätutkimuskysymyksenä on "millaisia muistinhallinnan tekniikoita voidaan hyödyntää 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.

Kirjallisuuskatsaus keskittyy kehittäjän omiin henkilökohtaisiin ratkaisuihin ohjelmointikieli työkalunaan. Tietokoneiden resurssien virtualisointi on yleistynyt nykypäivänä hajautettujen järjestelmien ja pilvipalveluiden tullessa yhä yleisimmiksi, mutta tässä kirjallisuuskatsauksessa rajaamme aihepiirin käsittämään perinteiseen RAM-muistin hallintaan liittyviä konsepteja. Virtualisoidun muistin allokoiminen tullaan sivuuttamaan kokonaan. Lisäksi, katsauksessa ei tulla käymään läpi 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 muistinhallinan tekniikoita varten tietoa on haettu myös suoraan konseptien omilla nimillä.

Kirjallisuuskatsauksessa kaksi lukua ovat taustoittavia teoria lukuja, 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 havainot ja pyritään vastaamaan asetettuun tutkimuskysymykseen.

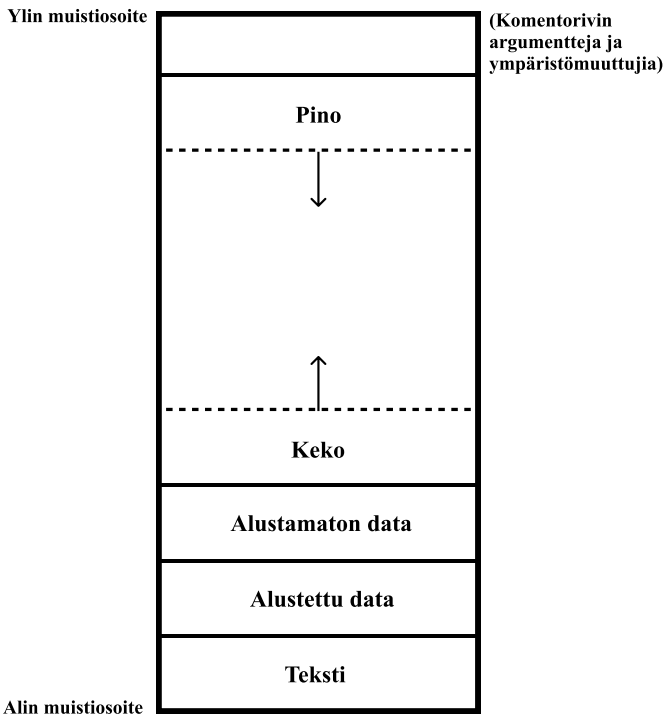
2 Muistinhallinta

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 muistialueisin. 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 (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] Pinon da-

tan 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.



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 pinoallokointi 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 ennen kääntöä.[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.

2.2 Muistin allokointi ohjelmointikielissä

Tietokone ohjelman käyttämä muisti voidaan jakaa kahden tyyppisen muistiin, staattiseen ja dynaamiseen muistiin. Staattisella muistilla tarkoitetaan yleensä globaalia data-aluetta ja pinoa, kun taas dynaamisella muistilla viitataan yleensä kekoon.[2]

Staattinen muistinhallinta

Dynaamista muistinhallintaa varten ohjelmointikielet tarjoavat erilaisia valmiista kirjastoista saatavia funktioita muistin varausta ja vapautamista 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 toinen 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. Realloc() on funktio, jolla pystyy muokkaamaan jo aikaisemmin varatun muistialueen kokoa. Realloc() ottaa argumentikseen muokattavan muistialueen osoittimen sekä muistilohkon uudelleen määritetyn koon.[4]

Memory fragmentation

Muistivuoto on muistinkäytön ongelmatilanne, joka tapahtuu kun ohjelma käyttää muistia, mutta ei kykene vapauttamaan sitä takaisin käyttöjärjestelmän käyttöön. 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 onglematilanteilta vältytään.

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;
}
```

3 Sulautetut järjestelmät

3.1 Mikä on sulautettu järjestelmä

Sulautetulle järjestelmälle on hyvin vaikeaa antaa yksikäsitteistä määritelmää, mutta yleisesti sulautetuilla järjestelmällä 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.[5] 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.[5]

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.[5] 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 kirjallisuuskatsauksen 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.[5] 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.[4]

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
}
```

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 ARM GNU ja LLVM.[6]

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. Segmentoitu 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 ylivuoto tilanteet on helppo selvittää, sillä pääpinon sisällä olevaa pinoa voidaan dynaamisella allokoinnilla kasvattaa tarpeen vaatiessa. Ajon aikainen kirjasto, joka allokoi ja vapauttaa segmentoidun pinon pienempiä pinoja, kohtelee tätä muistirakennetta linkitettynä listana.[6]

Segmentoitu pinon hyöty sulatetuissa järjestelmissä on aikaisemmin mainittujen haittojen häviäminen ja uusien muistin optimointimahdollisuuksien syntyminen.

Heikkoudet häviävät: 1. Muistin fragmentoituminen vähenee 2. Koodikannan kontrollin paraneminen 3. Suorituskyvyn heikentyminen on hyväksyttävämpää

Uusia mahdollisuuksia syntyy: 1. Muistinkäytön turvallisuus voidaan saavuttaa kääntäjällä 2. Laitteisto ja energia tehokkuus 3.

4.3 Buddy Memory Allocation

4.4 Memory Pooling

4.5 Fixed-size Block Allocation

4.6 Memory Banks

4.7 Object Pools

4.8 Slab allocation

5 Tekniikoiden soveltaminen sulautetuissa järjestelmissä

Mitä asioita pitää ottaa huomioon suunniteltaessa muistinhallinta rakennetta? Staattinen allokointi mielummin kuin dynaaminen, miksi?

Listaa sulautettujen järjestelmien haasteet ja analysoi niiden pohjalta, tee jokin yhteenveto?

6 Yhteenveto

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

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

[6] 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.