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, 11 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 merkistystä entisestään. Muistin, prosessorin ja laitteiston komponenttien ominaisuuksien rajallisuus asettavat kehittäjälle haasteita, joiden ratkaisut voivat vaatia kehittäjältä hyvinkin kustomoituja ja vaativia rakenteita, jos vertaillaan perinteisille PC-tietokoneille kehitettävien ohjelmien muistin rakennetta.

Kirjallisuuskatsauksessa tullaan tutkimaan sulautettujen järjestelmien muistinahallintaa 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. Virtualisodun 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 ai-

hepiirejä, kuten mm. roskien keruun ohjelmointikielissä. Katsauksessa on valittu esimerkkiohjelmointikieleksi C konseptien havainnollistamiseksi. Valinta on perusteltua C:n alkuperäisen luonteen vuoksi sekä se on yksi yleisimmistä ohjelmointikielistä sulautetuissa järjestelmissä. Suurimmassa osassa katsauksessa käsiteltävissä artikkeleissa myös implisiittisesti oletetaan, että ohjelmointikieli, jonka ympärille muistirakenteita toteutetaan on juuri C. Lisäksi monet esimerkit C:llä voidaan yleistää C:n laajennukselle C++:lle, joka on myös yksi yleisimmistä kielistä sulautetuissa järjestelmissä.

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.

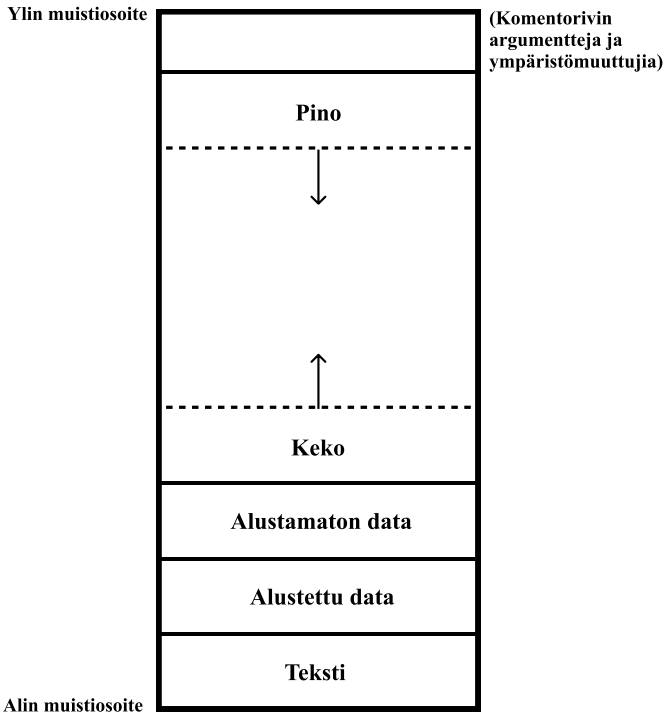
2 Muistinhallinta

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

2.1 Ohjelman muisti

Ohjelman 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 ja keon datan allokointi tapahtuu

vasta ohjelman ajon aikana (engl. *runtime*). Täsmennettävää on, että vaikka kääntäjän luomat ohjeet pinon hallintaan syntyvät jo käännön aikana, varsinainen itse datan allokointi ja vapautus tapahtuu ajon aikana. [2]



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 niiden 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-osoitin 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 *heap*) 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 analogiassa tietorakenne keon kanssa.

2.2 Muistin allokointi ohjelmointikielissä

Tietokone ohjelman käyttämä muisti voidaan jakaa kahden tyyppisen muistiin, staattiseen ja dynaamiseen muistiin.[2]

Dynaamista muistinhallintaa varten ohjelmointikielet tarjoavat erilaisia valmiista kirjaistoista 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]

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. Edm. 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 vältyttäisiin.

Ohjelmalistaus 1 Demonstraatio muistin allokoinnista C-ohjelmointikielessä

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

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

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

3 Sulautetut järjestelmät

3.1 Mikä on sulautettu järjestelmä

Real-Time System Single core vs Many Core?

3.2 Sulautettujen järjestelmien haasteet kehittäjälle

Puuttuu yleenssä MMU ja MPU

3.3 Sulautettujen järjestelmien tulevaisuus

4 Muistinhallinan tekniikoita ja rakenteita

4.1 Fixed-size Block Allocation

4.2 Buddy Memory Allocation

4.3 Memory Pooling

4.4 Circular Buffer

An introduction to the C Programming Language, 15.4 Circular Buffers (Tim Bailey)

4.5 Memory Banks

4.6 Object Pools

4.7 Slab allocation

5 Tekniikoiden soveltaminen sulautetuissa järjestelmissä

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

6 Yhteenveto

Lähdeluettelo

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

[2] D. A. Alonso, S. Mamagkakis, C. Poucet et al., "Dynamic Memory Management for Embedded Systems", s. 1–100, 2015.

[3] H. Erives, ""The stack and the stack pointer"", http://www.ee.nmt.edu/ erives/308L$_0$5, 2006.

[4] T. Bailey, *An Introduction to the C Programming Language and Software Desing*. Sydney: The University of Sydney, 2015.