Harjoitustyöohje

ITKA203 Käyttöjärjestelmät -kurssin Harjoitustyö kesällä 2007. Paavo Nieminen / Jyväskylän yliopiston Tietotekniikan laitos.

1 Harjoitustyön tavoitteet

Päätavoite:

- opit esimerkin kautta, kuinka konekielinen ohjelma saa prosessorin suorittamaan aliohjelma-aktivaation. Opit suorituspinon toimintaperiaatteen ja roolin ohjelman suorituksessa.

Toissijainen tavoite:

- samalla näet hieman yleiskuvaa konekielisestä ohjelmoinnista.
- samalla syvennyt viitteen (tai muistiosoittimen) rooliin ohjelmoinnissa, mikä parantaa kykyäsi ohjelmoida.

Ideat:

- Tämän *pitäisi* käsittääkseni tuottaa vain vähän lisätyötä, jos on taustalla demo1, demo2 ja luennot käytynä / materiaalit ymmärrettynä tähän saakka. Ohjausta tarjotaan demotilaisuuksissa, sähköpostilistalla ja sopimuksen mukaan henkilökohtaisesti.
- Tarkoitus on, että tämä vastaisi aiempien kurssikertojen harjoitustyötä, paitsi että käytetään tutustumismielessä 2000-luvun työkaluja, ja C-ohjelmointiosuus on ohjelmointimielessä tasapäisempi eri aiheiden välillä ja yleisesti ottaen helpompi.

Sisältö

2 Työkalut

Eli meillä olis vanha tuttu `gcc` ja sitten olis lisäksi `gdb`, ja sitten olis pikkuohjelma `tee`. Ja joku tekstieditori, ihan sama mikä, ja sitten kirjallisuutta C-kielestä ja x86-86:sta. Lähestulkoon saattaa tulla toimeen päivitetyllä luentomateriaalilla ja demo 2:n ohjeistuksella. Lisämateriaalina voi käyttää mitä vaan löytää; jos löytää hyvän, voi ilmoittaa siitä muillekin. Prosessorimanuaalit ovat "the Lähde", jos niitä jaksaa lukea.

3 Työohje ja malliharjoitustyö

Tässä käydään läpi `malliharkka.c` -ohjelman kautta kaikki harjoitustyön vaiheet. Koodi ja sen pohjalta tehty vastausrunko on saatavilla kurssin nettisivulta. Vastausrunkoa varten tein ajon samalla tavoin uudelleen kuin tässä, mutta virtuaalimuistiosoitteiden lukuarvothan siinä ovat erilaiset kuin tähän kopioiduissa esimerkeissä, koska ajo on eri.

Tee verkkolevylle kurssin hakemistoosi demohakemistojen lisäksi oma hakemisto harjoitustyölle. Käy läpi työohjeen kohdat malliharjoitustyön kanssa. Osaat sitten toteuttaa vastaavat asiat oman aiheesi osalta.

3.1 Suunnittele ja toteuta algoritmi C-kielellä

Jokainen saa yhden yksinkertaisen algoritmin toteutettavakseen. Tuloksena pitäisi olla C-kielinen ohjelma, jossa on pääohjelma ja ainakin yksi aliohjelma, ja ainakin yksi aliohjelmakutsu, jossa on parametreja ja paluuarvo.

(Huom. tällä kertaa ei vaadita moduulijaon toteuttamista, vaan riittää tehdä yksi C-kooditiedosto, jossa on kaikki aliohjelmat.)

Varmistu, että C-ohjelma toimii, eli sen pitää tehdä harkka-aiheessasi vaadittu pikku toimenpide. Varmistu, että ymmärrät, miten C-kielinen ohjelmasi toimii, eli mitä oikein tulikaan koodattua ja miksi juuri tekemäsi koodi ratkaisee tehtävänannossa vaaditun ongelman.

Käytä kaikkiin kokonaislukumuuttujiisi 64-bittistä tyyppiä `long long int`; silloin kaikki kokonaisluvut ovat koko rekisterin levyisiä, ja uskoisin että se vähän yksinkertaistaisi asioiden ymmärtämistä tällä kertaa.

Lue demo2:n materiaalia tarvittaessa (tai muuta C-opastusta, jos tykkäät jostain toisesta matskusta enemmän...).

3.2 Käännä ohjelma x86-64 -assembleriksi GNU-kääntäjällä (gcc)

Kun C-ohjelma on mielestäsi lopullinen, käännä se seuraavalla tavoin. Kokeile malliharjoitustyön kanssa, niin opit, mistä on kyse...

Ensinnä, ihan kuriositeettina, kokeile seuraavaa:

```
gcc -E malliharkka.c | less
```

Em. komennolla ajoit vain C-käännöksen ensimmäisen vaiheen, jossa ns. esikääntäjä lisäilee omia rivejään lähdekoodiin ja käsittelee mm. `#include` -rivit. Ei siitä sen enempää; se oli vaan käytännön demonstraatio siitä, mikä on C-kielen esikääntäjä ja mitä se tekee.

Nyt itse asiaan. Anna seuraava komento:

```
gcc -O0 -S malliharkka.c
```

Optio `-O0` (eli iso oo ja nolla) tarkoitti, että kääntäjä ei optimoisi koodia. Koodin optimointi oletettavasti tekisi käännöksestä vaikeamman ymmärtää, kun tuotettu konekieli koettaisi oikoa mutkia suoriksi nopeamman suoritettavuuden ja lyhyemmän binääritiedoston toivossa. Optio `-S` tarkoitti, että ei käännetä valmiiksi konekieleksi asti, vaan tuotetaan vastaava assembler-koodi. Nyt pitäisi olla syntynyt tiedosto `malliharkka.s`. Sen pitäisi alkaa jotakuinkin näin:

```
.


        .file   "malliharkka.c"
        .text
.globl kaanna_taulukko
        .type   kaanna_taulukko, @function
kaanna_taulukko:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
```

```
        movq    %rdi, -40(%rbp)
        movq    %rsi, -48(%rbp)

...
```

Pisteellä alkavat tekstit ovat assemblerin ohjauskomentoja. Esim. `.file "malliharkka.c"` kertoo, että tämä on syntynyt tuon nimisestä C-koodista kääntämällä. `.text` kertoo, että seuraavat asiat pitää sijoittaa koodialueelle (jota sanotaan jostain syystä tekstialueeksi Unix-maailmassa...) `.globl kaanna_taulukko` tarkoittanee että meillä on globaali symboli nimeltä `kaanna_taulukko` ja `.type kaanna_taulukko, @function` kertonee että kyseinen symboli tarkoittaa funktiota eli aliohjelmaa.

Kaksoispisteeseen päättyvät rivit ovat symbolisia nimiä muistiosoitteille. Eli sekä `kaanna_taulukko` että `.LFB2` ovat nimiä sille muistipaikalle, jossa sijaitsee käskyn `pushq %rbp` konekielikoodin ensimmäinen tavu. Assembler-ohjelmoijan ei tarvitse tietää muistipaikan numeroarvoa; itse asiassa voi olla, että kukaan ei tiedä lopullisia osoitteita ennen kuin käyttöjärjestelmä lataa konekielisen ohjelman kovalevyltä muistiin suoritusta varten.

Tutustu lyhyesti assembler-käännökseen, pyri ymmärtämään, miten se toimii. Yritin kirjoittaa "Prosessorista" -nimiseen kurssimateriaaliin, eli luennolla 4 jaetun materiaaliin uuteen versioon, pienen opastuksen x86-64 -assemblerista esimerkkikäskyjen kautta. Sieltä pitäisi tutunnäköisiä komentoja löytyä. Kohta katsotaan asiaa vielä tarkemmin.

3.3 Tutustu GNU-debuggeriin (gdb)

Käännä ensin ohjelma komennolla:

```
gcc -g -O0 -o malliharkka malliharkka.c
```

Argumentti `-g` tuottaa debug-tiedot; niitä tarvitaan kohta. Jälleen `-O0` kieltää optimoimasta käännöstä ja `-o malliharkka` määrää nimen käännöksen tulostiedostolle. Pitäisi syntyä ajettava konekielinen tiedosto `malliharkka`.

Käynnistä komentorividebuggeri seuraavasti:

```
gdb malliharkka
```

Mitäs nyt? Tilanne näyttää seuraavalta ja vaatii selitystä:

```
[nieminen@jalava harkka]$ gdb malliharkka
GNU gdb Red Hat Linux (6.3.0.0-1.84rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, ...

...
(gdb)   [ja vilkkuva kursori ...]
```

Käynnistit debuggerin. Ohjelmointi 1:ltä on toivottavasti tuttu asia Eclipse IDE:n graafinen Java-debuggeri, jolla voi opiskella Java-ohjelman toimintaa, ja jota voi käyttää virheenetsinnässä (siitä niiden nimi "de-" "bugger"). Nyt käytettävä `gdb` on samanlainen, joskin tekstipohjainen. Tällekin on olemassa graafisia front-endejä, mutta interaktiivisten komentoriviohjelmien käyttö on yksi tämän kesäkurssin kieroja teemoja, joten noudatamme sitä loppuun asti. Tarkoitushan on käydä läpi virtuaalikonehierarkioita ja askeltaa vähän aikaa näillä matalammilla ja vähemmän graafisilla tasoilla, joille ensinnäkin saattaa joskus syystä tai toisesta pakosti joutua ja joiden olemassaolo on hyvä joka tapauksessa ymmärtää. Debuggeri odottaa nyt tekstimuotoisia komentoja, joiden avulla voit tutkia argumenttina annetun ohjelman toimintaa. Komenna debuggeria:

```
run
```

Debuggeri käynnisti malliharjoitustyön käännetyn ohjelman, antoi sen mennä niin pitkälle kuin se ilman virheitä etenee, tässä tapauksessa onnistuneeseen loppuun saakka. Tulostui toivon mukaan seuraavaa:

```
Starting program: /autohome/home3/363/nieminen/kj07kesa/harkka/malliharkka
-1 2 3 -16 2 8
8 2 -16 3 2 -1
Samoja lukuja oli 1 kpl.

Program exited normally.
```

Jos ohjelma olisi kaatunut, debuggerin avulla voisi alkaa selvittelemään, miksi niin kävi... arvokas ohjelmantekijän työkalu... Tekstipohjainen ohjelma on erilainen kuin graafinen ohjelma, mutta se voi olla käyttäjäystävällinen omalla tavallaan. Esim. gdb:n komennoista saa ohjeita "on-line" komentamalla `help`. Komennoissa on automaattitäydennys tabulaattorilla, ja niille on vähämerkkisiä "alias"-nimiä, jotka on nopea kirjoittaa. Komentohistoria on käytettävissä nuolinäppäimillä ja edellisen komennon voi toistaa painamalla pelkkää enteriä. Debuggerin käyttöliittymä on paljolti samanlainen kuin tehokkaan interaktiivisen shellin -- tai minkä tahansa hyvin tehdyn tekstipohjaisen ohjelman. Mieleen tulee itselle esimerkiksi laskentaohjelmisto Matlabin "Command Window", joka on tällä tavoin kätevä.

Käväistään vähän aikaa pois debuggerista; siitä pääsee pois komennolla:

```
quit
```

3.4 Aja ohjelmaa GNU-debuggerilla (gdb) käsky kerrallaan ja tallenna ajo

Nyt käynnistetään gdb uudelleen sillä tavoin, että koko tuleva tekstimuotoinen keskustelusi ohjelman kanssa tallentuu tiedostoon. Kun olet saanut oman harjoitustyösi C-ohjelman valmiiksi ja käännettyä, tämä on se, miten pohjustat harjoitustyön: toteutat seuraavat vaiheet omalle ohjelmallesi, ja saat näin tekstitiedoston, johon lisäät sitten myöhemmin vähän "oppimista ilmaisevaa tekstiä" muutamaan kriittiseen kohtaan. Kokeillaan prosessia malliharkan kanssa. Eli käynnistä debuggeri seuraavalla tavoin:

```
gdb malliharkka | tee malliharkka_ajo.txt
```

Ohjelma `tee` lukee syötteensä, kaiuttaa sen suoraan ulostulovirtaan, ja tallentaa sen myös argumenttina annettuun tiedostoon eli tässä tapauksessa syntyy `malliharkka_ajo.txt`, johon tallentuvat kaikki gdb-sessiossasi kirjoitettu ja tulostunut teksti. Gdb:ssä olisi mahdollisuus dumpata tulosteet tiedostoon ilman mitään apuohjelmaa, mutta käytetään `tee`:tä nyt, koska halutaan talteen koko dialogi komentoineen eikä pelkkiä tulosteita.

Komenna gdb:tä seuraavasti:

```
display /3i $rip
display /t $eflags
display /x $rbp
display /x $rsp
display /x $rip
```

Voit copy-pastata suoraan tästä harkkaohjeesta. Näiden komentojen merkitys on, että tästedes `gdb` näyttää aina automaattisesti seuraavat tiedot (viimeksi komennettu ensimmäisenä):

`/x $rip`	käskyosoiterekisterin arvo heksalukuna, seuraavaksi suoritettava konekielikomento tullaan noutamaan (fetch) tästä muistiosoitteesta
`/t $eflags`	lippurekisterin lippubitit
`/x $rsp`	pino-osoittimen arvo heksalukuna
`/x $rbp`	pinokehyksen kantaosoittimen arvo heksana
`/3i $rip`	kolme konekielikäskyä alkaen muistipaikasta RIP

Ohjelma pitää ensin käynnistää. Nyt ei ajeta sitä läpi `run`:illa vaan aloitetaan käsky kerrallaan suorittaminen. Komenna:

```
start
```

Tutki debuggerin tulostetta, joka tuli välittömästi. Pitäisi olla jotakuinkin seuraavaa:

```
Breakpoint 1 at 0x400620: file malliharkka.c, line 102.
Starting program: /autohome/home3/363/nieminen/kj07kesa/harkka/malliharkka
main (argc=1, argv=0x7fffb9bbd3a8) at malliharkka.c:102
102       long long int testitaulu[] = {-1, 2, 3, -16, 2, 8};
5: /x $rip = 0x400620
4: /x $rsp = 0x7fffb9bbd270
3: /x $rbp = 0x7fffb9bbd2c0
2: /t $eflags = 1000000010
1: x/3i $rip
0x400620 <main+15>:     movq   $0xffffffffffffffff,0xffffffffffffffc0(%rbp)
0x400628 <main+23>:     movq   $0x2,0xffffffffffffffc8(%rbp)
0x400630 <main+31>:     movq   $0x3,0xffffffffffffffd0(%rbp)
```

Suoritus pysähtyi nyt `main()` -aliohjelman alussa ennen sitä konekielikäskyä, joka on syntynyt sovellusohjelman ensimmäisen suoritettavan C-kielisen rivin kääntämisestä (siis rivin numero 102, jossa on taulukon esittely ja alustus). Koska käännöksessä oli mukana debug-tiedon tallennus ja lähdekoodi on saatavilla, debuggeri kykenee näyttämään C-kielisen koodirivin, jota ollaan suorittamassa. Sitten näkyvät edellisissä `display`-komennoissa pyydetyt asiat.

Pysähdy fiilistelemään: Kun pääset tähän vaiheeseen omaa harjoitustyötäsi, olet sillä äärilaidalla, johon sovellusohjelmoija syvimmillään pääsee tietokoneen käytössä. Olet tehnyt ohjelman ja kääntänyt sen tietylle prosessorille (ja tietylle käyttöjärjestelmälle). Näet ohjelman pysäytettynä debuggerissa, ja ymmärrät, mistä on kyse: Seuraavassa hetkessä prosessori noutaisi käskyn. Se olisi virtuaalimuistiosoitteesta `0x400620`, jonka prosessori muuntaisi fyysiseksi osoitteeksi väylälle käyttäen 48-bittisen virtuaaliosoitteen osia osoitteenmuodostuksessa:

```
+---------------------------------+----------+----------------+
|Hierarkkinen indeksointi         | Sivun    | Muistipaikan   |
|sivutaulun hakemiseen            | indeksi  | indeksi sivulla |
|                                 | taulussa |                |
|                                 |          |                |
|000000000   000000000   000000010 | 000000000 | 011000100000   |
|                                 |          |                |
+---------------------------------+----------+----------------+
|       Korvautuu osoitteenmuodostuksessa     | pysyy samana   |
+--------------------------------------------+----------------+
```

Saattaisi tulla sivuviittausvirhe, voi olla että kovalevy olisi herätettävä virransäästötilasta swappaamista varten... sitä ei tiedä, missä vaiheessa käsky suoritettaisiin. Lopulta se kuitenkin tulisi noudetuksi prosessorin suoritettavaksi. Käsky, joka tuosta kohtaa virtuaalimuistia löytyy, on malliharjoitustyön osalta seuraavanlainen:

```
0x400620:     movq   $0xffffffffffffffff,0xffffffffffffffc0(%rbp)
```

Tämän käskyn suoritus tulee siirtämään lukuarvon -1 (kaikki bitit ykkösiä, heksaesitys ffffffffffffffff) virtuaalimuistiosoitteeseen, joka saadaan vähentämällä RBP:n arvosta 64 (lisätään heksaluku ffffffffffffffc0 eli etumerkillisenä -0x40). Kyseinen muistiosoite, ja siis siirron kohde, tulee olemaan:

```
RBP-rekisterin arvo
|
|             plus
|             |
|             | Käskyn binäärikoodiin sisällytetty vakiosiirros
|             | |
|             | |                    64-bittiseen rekisteriin
|             | |                    jää tässä ynnäyksessä
|             | |                    tulokseksi tämä luku
|             | |                    |
0x7fffb9bbd2c0 + 0xffffffffffffffc0 == 0x7fffb9bbd280

Eli bitteinä:

+---------------------------------+----------+----------------+
|Hierarkkinen indeksointi         | Sivun    | Muistipaikan   |
|sivutaulun hakemiseen            | indeksi  | indeksi sivulla |
|                                 | taulussa |                |
|                                 |          |                |
|011111111   111111110   111001101 | 110111101 | 001010000000   |
|                                 |          |                |
+---------------------------------+----------+----------------+
|       Korvautuu osoitteenmuodostuksessa     | pysyy samana   |
+--------------------------------------------+----------------+
```

Käskyn suorituksessa prosessori tekee kohdeoperandin osoitteenmuodostuksen fyysiseksi osoitteeksi; väylä siirtää luvun keskusmuistiin, ja RIP:n arvo päivittyy seuraavan käskyn osoitteksi. Tässä tapauksessa lukuarvoksi 0x400628. Totea, että näin käy, eli komenna yhden käskyn suorittaminen ("*step instruction*"):

```
stepi
```

Tulostus vahvistaa, että näin juuri kävi; seuraava tilanne on debuggerin mukaan tällainen:

```
5: /x $rip = 0x400628
4: /x $rsp = 0x7fffb9bbd270
3: /x $rbp = 0x7fffb9bbd2c0
2: /t $eflags = 1000000010
1: x/3i $rip
0x400628 <main+23>:     movq   $0x2,0xffffffffffffffc8(%rbp)
0x400630 <main+31>:     movq   $0x3,0xffffffffffffffd0(%rbp)
0x400638 <main+39>:     movq   $0xfffffffffffffff0,0xffffffffffffffd8(%rbp)
```

Suoritetaan kohta ohjelmaa käsky kerrallaan eteenpäin. Ensin kuitenkin tehdään pääohjelmalle disassembly:

```
disassemble
```

Jos koko listaus ei mahdu yhteen ruutuun, gdb sivuttaa sen; katso loppuun asti painamalla tarvittaessa enter. Pääteikkunan selaus taaksepäin on mahdollista myöhemmin, ja kaikki tuloksethan tallentuvat jatkuvasti myös tekstitiedostoon, jota voit tarkastella myöhemmin.

Harmi kyllä en löytänyt mitään optiota, jolla olisi saanut gdb:n näyttämään suhteelliset muistiviittaukset negatiivisiin offsetteihin negatiivisilla luvuilla (AT&T -syntaksia käytettäessä). Jotenkin se luultavasti olisi mahdollista, mutta ei ihan triviaalisti... Tämä on nyt pienenpieni hankaluus, jonka kanssa pitää elää. Tässä on taulukko joidenkin negatiivisten lukujen heksakoodauksista:

64-bittinen heksal.	**desimaaliluku**
`0x0000000000000008`	8
`0x0000000000000001`	1
`0x0000000000000000`	0
`0xffffffffffffffff`	-1
`0xfffffffffffffffc`	-4
`0xfffffffffffffff8`	-8
`0xfffffffffffffff0`	-16
`0xffffffffffffffe8`	-24
`0xffffffffffffffe0`	-32
`0xffffffffffffffd8`	-40
`0xffffffffffffffd0`	-48
`0xffffffffffffffc8`	-56
`0xffffffffffffffc0`	-64

64-bittinen heksal.	desimaaliluku
0xffffffffffffffb8	-72
0xffffffffffffffb0	-80
0xffffffffffffff00	-256

Eli gdb:n näyttämät negatiiviset luvut ovat ikään kuin positiivisia, isoja lukuja, ns. kahden komplementtimuunnoksen kautta. Näinhän negatiiviset luvut esitetään tietokoneessa. Etumerkkiä ei gdb:n tulosteessa siis käytetä, mikä voi olla aluksi vähän raflaavaa.

Tutkitaanpa muistia kantaosoitteen RBP ympäriltä, esim:

```
x /32xg $rbp-128
```

Muistutetaan itseämme vielä rekisterien arvoista tässä vaiheessa ohjelman suoritusta:

```
display
```

Tuloste oli seuraavanlainen (huomioita lisätty jälkeenpäin):

```
(gdb)     x /32xg $rbp-128
0x7fffb9bbd240: 0x00007fffb9bbd328      0x00002aaaaaaab000
0x7fffb9bbd250: 0x0000000000000000      0x0000000000000000
0x7fffb9bbd260: 0x0000000000000000      0x0000003aaf319be0
0x7fffb9bbd270: 0x00007fffb9bbd3a8 *1*  0x00000001004003fb
0x7fffb9bbd280: 0xffffffffffffffff *2*  0x00000000004006d0
0x7fffb9bbd290: 0x0000000000000000      0x0000003aaf319be0
0x7fffb9bbd2a0: 0x00000000004006a0      0x0000000000400440
0x7fffb9bbd2b0: 0x00007fffb9bbd3a0      0x0000000000000000
0x7fffb9bbd2c0: 0x00000000004006a0 *3*  0x0000003aaf61c40f *4*
0x7fffb9bbd2d0: 0x0000000000000000      0x00007fffb9bbd3a8
0x7fffb9bbd2e0: 0x0000000100400440      0x0000000000400611
0x7fffb9bbd2f0: 0x0000003aaf319be0      0x00000000004006a0
0x7fffb9bbd300: 0x0000000000400440      0x00007fffb9bbd3a0
0x7fffb9bbd310: 0x0000000000000000      0x0000000000000000
0x7fffb9bbd320: 0x00007fffb9bbd2d0      0x0000003aaf61c3ca
0x7fffb9bbd330: 0x0000000000000000      0x0000000000000000

*1*: Pinon huippu on osoitteessa $rsp = 0x7fffb9bbd270
*2*: Äskeinen sijoitus muutti osoitteen -64($rbp) = 0x7fffb9bbd280
     sisältöä.
*3*: Main-aliohjelman pinokehyksen kanta on $rbp = 0x7fffb9bbd2c0.
     Tässä osoitteessa on aiemman pinokehyksen kantaosoite; se on
     0x4006a0, eli mainia kutsuneen koodin pinokehys onkin samalla sivulla
     kuin konekielikoodi, heti viimeisen konekäskyn jälkeen. En osaa
     kertoa miksi, mutta näköjään näin se on tehty... luultavasti ihan
     hyvästä syystä...
*4*: Tässä on paluuosoite, eli main-aliohjelmaa kutsuneen call-käskyn
```

```
jälkeisen käskyn osoite. Tämä on dynaamisesti ladatun C-kirjaston
(tiedostosta /lib64/libc.so.6) koodia, joka jatkuu siis osoitteessa
0x0000003aaf61c40f siten että siellä kutsutaan käyttöjärjestelmän
exit()-palvelua antaen parametriksi sovellusohjelman mainista
palauttama kokonaisluku
```

```
(gdb) display
5: /x $rip = 0x400628
4: /x $rsp = 0x7fffb9bbd270
3: /x $rbp = 0x7fffb9bbd2c0
2: /t $eflags = 1000000010
1: x/3i $rip
0x400628 <main+23>:     movq   $0x2,0xffffffffffffffc8(%rbp)
0x400630 <main+31>:     movq   $0x3,0xffffffffffffffd0(%rbp)
0x400638 <main+39>:     movq   $0xfffffffffffffff0,0xffffffffffffffd8(%rbp)
```

Muistin tulostus tehdään myöhemminkin, jotta huomataan, miten pinomuisti muuttuu.

Sitten ajetaan ohjelmaa konekäsky kerrallaan:

```
stepi
```

Huomioi, miten rekisterit muuttuvat aina käskyn jälkeen. Edellisen komennon saat toistettua painamalla pelkästään enteriä. Jossain vaiheessa tulee vastaan malliharjoitustyön ensimmäinen aliohjelmakutsu:

```
0x400660 <main+79>:     callq  0x4005bc <tulosta_taulukko>
```

Mennään tähän aliohjelmaan, eli suoritetaan vielä kerran:

```
stepi
```

Mutta tämä ei ole nyt se, mikä meitä kiinnostaa, vaan kiinnostaa se varsinainen algoritmitoteutus, johon suoritus päätyy myöhemmin. Katso, että suoritus on tosiaan tullut aliohjelmaan `tulosta_taulukko` eli gdb näyttää jotakin seuraavanlaista:

```
92      void tulosta_taulukko(long long int taulu[], long long int koko){
...
```

Eli suoritus päätyi C-koodiriville 92. Annetaan nyt koko aliohjelman mennä loppuun, eli komenna:

```
finish
```

Tällä tavoin voi gdb:llä suorittaa loppuun aliohjelman. (Tämä vastaa "step out" -komentoa joissakin debuggereissa).

Malliharjoitustyössä on seuraavaksi vuorossa kiinnostuksemme kohde, eli varsinaisen taulukonkääntö-algoritmin yksi suorituskerta. Debuggeri kertoo, millä rivillä suoritus on menossa:

```
107       samoja = kaanna_taulukko(testitaulu, koko);
...
```

Aja nyt käsky kerrallaan ja tarkkaile, mitä ohjelma tekee... miten ja mitä kautta parametrit siirretään, ja miten suoritus siirretään aliohjelman koodiin:

```
stepi
```

Askella ohjelmaa tarkkaavaisesti, kunnes seuraavaksi tulisi vuoroon `call`-käsky. Tässä kohtaa komenna gdb:tä taas tulostamaan muistin sisältöä RBP:n ympäriltä, ja muistuta sen jälkeen itseäsi rekisterien arvoista (eli tehdään sama kuin aiemmassa kohdassa):

```
x /32xg $rbp-128
display
```

Askella `call`-käsky. Pysyithän kärryillä siinä, miten rekisterit muuttuivat. Vuorossa on ensimmäinen käsky aliohjelmasta `kaanna_taulukko`.

Tee tästäkin aliohjelmasta disassembly:

```
dissassemble
```

Tutki, mitä kertoo komento:

```
info registers
```

Se näyttää prosessin koko kontekstin, eli käyttäjän näkemät rekisterit. Huomataan esimerkiksi, että rekisterissä RSI on lukuarvo kuusi ja rekisterissä RDI on muistiosoite johonkin kohtaan muistia. Katsotaan, mitä siellä kohtaa muistia on:

```
x /12xg $rdi
```

Siellähän on testiohjelman taulukon lukuarvot. Taulukko on siis välitetty aliohjelmalle muistiosoitteena rekisterissä RDI. Ihan niinkuin ABI-dokumentaation mukaan pitikin. Varsinaiset taulukon luvut sijaitsevat pinomuistissa kutsuvan ohjelman pinokehyksen alueella. Mainin alussahan oli sijoituskäskyjä, jotka laittoivat nuo luvut pinomuistiin. C-ohjelmassa taulukko alustettiin esittelyn yhteydessä, mutta konekielikäännöksessä se näköjään tehdään erillisillä peräkkäin suoritettavilla operaatioilla.

Nyt ilmeisesti ohjelmalla on tarkoitus luoda juuri kutsutulle aliohjelmalle oma kehys. Katso käsky käskyltä, miten se tapahtuu:

```
stepi
```

Katso tulosteesta, miten RSP ja RBP toimivat. Aliohjelman alussa myös siirretään parametrit rekistereistä uuden pinokehyksen sisään, ikään kuin ne olisivat lokaaleja muuttujia. Jatkossa niitä käytetään pinomuistin kautta noista osoitteista eli 0xffffffffffffffd8(%rbp) ja 0xffffffffffffffd0(%rbp).

Tutki vielä tässäkin kohtaa muistin sisältö seuraavasti:

```
x /32xg $rbp-128
display
```

Löydätkö pinomuistista edellisen pinokehyksen kantaosoitteen? Entä osoitteen käskyyn, johon aliohjelmasta tulee palata? Pitäisi olla asiat niissä muistipaikoissa, joissa pinokehysmallin mukaan olettaisitkin. Jos aliohjelma ei kutsu mitään muita aliohjelmia, pinon huippua ei luultavasti ole päivitetty lokaalien muuttujien mukaan, koska uutta aktivaatiota ei tarvitse luoda. Tämä on siis "lehtialiohjelma", jonka lokaalit muuttujat mahtuvat kutsuvan ohjelman kehyksen "red zoneen" (ks. kurssimateriaali). Siksi suorituksen ajan RBP==RSP. Sen sijaan main() -aliohjelman suorituksesta nähdään normaali tilanne, jossa SP:n täytyy olla valmiina uuden kehyksen luontiin. 128 tavun "Red zone" on siis ABIssa sovittu asia, joka on x86-64:n GNU-C-kääntäjässä näin; muissa toteutuksissa voi olla sovittu ihan millä tavoin vaan.

Käy sitten aktivaatiota läpi eteenpäin käsky käskyltä. Silmukan ja ehtolauseiden toteutustavasta ei olla kauhean kiinnostuneita, mutta suoritetaan sitä kuitenkin askel askeleelta läpi ja katsotaan päällisin puolin, millaisista käskyistä se muodostuu:

```
stepi
```

Ja toista tätä. Huomaa, miten debugger näyttää aina C-kielisen koodirivin, jota ollaan suorittamassa, ja voit käydä läpi komento komennolta sen, millaiseksi konekielikäskyjen ryppääksi kukin lähdekoodirivi on käännetty. Sijoituksista näet mm. missä muistiosoitteissa (suhteessa kantaosoiterekisteriin) säilytetään mitäkin muuttujia. Siitä näkee myös, miten `for`-silmukka toteutuu siten että yhdestä lähdekoodirivistä muodostaan käännöksessä käskyjä ennen ja jälkeen silmukkalohkon sisällön: Silmukka alkaa itseasiassa hyppykäskyllä sen loppuun, jossa testataan lopetusehto. Jos ehto ei vielä päde, hypätään silmukkalohkon koodin alkuun. Näin voidaan assembler-koodista havaita, että for-silmukkaa ei suoriteta kertaakaan, jos sen ehto ei ole alussa tosi. Nyt sille on myös konkreettinen näkökulma prosessorikomentoina. Mielestäni tämän pitäisi olla valaisevaa... Ehdolliset hypyt toimivat EFLAGS-rekisterin bittien mukaan. Observoi, että EFLAGSin bitit muuttuvat aika usein käskyjen seurauksena.

Huomaa, että osoitteiden muodostaminen kokonaisluvulla indeksoituun taulukkoon koostuu monesta vaiheesta: Indeksimuuttuja otetaan ensin pinomuistista rekisteriin, kerrotaan rekisterissä sitten kahdeksalla (bittien siirto kolmella pykälällä vasemmalle) koska indeksoidaan kahdeksan tavun mittaisia alkioita, ja osoitteethan puolestaan osoittavat yhden tavun mittaisia peräkkäisiä muistilokeroita. Sitten lisätään tulokseen ensimmäisen alkion muistiosoite, joka sijaitsee pinomuistissa. Lopulta laskemisen jälkeen rekisterissä on sen muistipaikan numero, jota voidaan käyttää halutun alkion osoittamisessa.

Okei, koko algoritmia ei tarvitse askeltaa läpi... Kun mielestäsi kykenet hyvin uskomaan sen asian, että C-koodi on kääntynyt käsky kerrallaan useaksi konekielikäskyksi, joita prosessori suorittaa yksi kerrallaan, ja että muuttujat ilmenevät tiettyinä muistipaikkoina suorituspinossa, voit siirtyä tämän aliohjelman loppuun. Tee se esim. seuraavasti. Aseta breakpoint return-lauseeseen, eli komenna:

```
break 88
```

(tai mikä nyt omassa lähdekoodissasi onkaan aliohjelman return-lauseen rivinumero). Sitten komenna:

```
continue
```

Nyt ohjelman suoritus jatkui breakpointtiin asti, joka on siis se kohta, johon on sijoitettu ensimmäinen rivillä 88 sijaitsevan return-lauseen suoritukseen kuuluva konekielikäsky. (Tästedes suoritus pysähtyisi aina tuohon kohtaan, kunnes breakpoint otettaisiin pois päältä. Breakpointit helpottavat debuggausta; sellainen laitetaan paikkaan, joka on juuri ennen sitä kohtaa, jossa ohjelma esimerkiksi tuntuu kaatuvan. Sitten tarkkaillaan käsky käskyltä, miten koodi esim. sen kaatumisen saa aikaan: mitä on missäkin rekisterissä, mistä tulee se väärä muistiosoite, "null pointer" tai muu.)

Nyt taas ole hyvin tarkkana: Mitä tekee `leaveq` ja `retq`. Mitä on pinon päällä milläkin hetkellä, miten kontrolli palaa kutsuneeseen ohjelmaan. Mitä kutsuva ohjelma vielä tekee ennen kuin koko aliohjelmaa kutsuva C-koodirivi on suoritettu loppuun.

Muita aliohjelmia ei tarvitse enää askeltaa. Katsotaan kuriositeettina, mitä tapahtuu C-ohjelman lopuksi. Suorita ohjelmaa riville, jossa on malliharjoitustyön loppu eli main-ohjelman return:

```
until 111
```

Aja tästä eteenpäin käsky käskyltä kunnes kontrolli päätyy jonnekin aivan muualle kuin itse ohjelmoituun koodiin. Viimeinen retq -komento vie johonkin tällaiseen paikkaan:

```
0x0000003aaf61c40f in __libc_start_main () from /lib64/libc.so.6
```

Eli prosessin virtuaalimuistiosoitteen 0x0000003aaf61c40f paikkeilla on ohjelman käynnistyksen yhteydessä dynaamisesti linkitetty kaikkien C-ohjelmien yhteinen apukirjasto `libc`, jonka tehtävä on mm. tehdä alustuksia ja lopetteluja käyttöjärjestelmäkutsujen avulla, ja siinä välissä kutsua sovellusohjelmoijan kirjoittamaa tietynlaista `main` -nimistä aliohjelmaa. Näin siis C-ohjelma päätyy sen sovittuun aloituspisteeseen, ja näin se jatkaa vielä sovitun päätepisteen jälkeenkin. Nyt tiedät senkin; se on jo aika paljon enemmän kuin Ohjelmointi 1:llä kerrottiin ohjelmoinnista... ja vasta alkua siitä, mistä todellisessa maailmassa olisi kyse, jos asia vietäisiin "oikeasti tekniseksi" (mitä edelleenkään mielestäni tällä kurssilla ei oikeasti tehty missään vaiheessa...).

Tässä oli kaikki. Kun tämä on tehty omalle harjoitustyöllesi, ja ajo on tallessa tekstitiedostossa, voit siirtyä tekemään varsinaisen työn.

Java-ohjelman suorittaminen muuten puolestaan alkaisi sillä, että Java-virtuaalikone käynnistyy, lukee sitten argumenttina annetun luokan tavukoodin, ja tekee ties mitä ennen kuin JVM:n virtuaalisessa käskyosoiterekisterissä on ajettavan luokan julkisen `main`-luokkametodin ensimmäisen käskyn osoite ... Yritin debugata gdb:llä Java-virtuaalikoneen suoritusta, mutta en onnistunut siinä äkkiseltään... pitäisi opiskella jonkin verran lisää, että tietäisin miksi.

Vielä kerran: perustelu sille, että kaikki tämä käydään läpi ja opitaan, on se, että opiskellaan korkeakoulutason informaatioteknologiaa, jossa ohjelmointiin kuuluu ymmärtää *mitä tarkoittaa ohjelman suorittaminen*, koska suorittamista varten kai kaikki ohjelmat tehdään, eikö niin. Ja jos jossain on epätriviaali tietojärjestelmä, siellä suoritetaan helkutinmoista määrää ohjelmia yhtäaikaa eri puolilla järjestelmää. Asia kuuluu *ohjelmoinnin opetuksen* alkupäähän, ja siellähän se itse asiassa onkin: toisen vuoden kaikille pakollisella kurssilla nimeltä Käyttöjärjestelmät. Vaikka asiaa käytiin läpi C-kielellä, ja esimerkkinä oli fyysisen x86-64 -laitteiston konekieli, samat universaalit tosiasiat pätevät kaikkien ohjelmien suorittamisessa. Alla on ote Javan virtuaalikoneen dokumentaatiosta. Tunnistanet termistön olevan tuttua Käyttöjärjestelmät-kurssilta (Olen luennoinut kurssin aikana ainakin seuraavien merkityksestä: *Program counter*, *register*, *virtual machine*, *thread*, *execution*, *method*, *address*, *instruction*, *pointer*, *specific platform*, *stack*, *frame*. Ja tuo sana *native* tarkoittaa "natiivia" eli tietylle prosessorille konekieleksi käännettyä ohjelman osaa, *native pointer* on tietysti jonkun tietyn prosessorin muistiosoite). Ja toisin päin: olet opiskellut kurssin, *jotta* tunnistaisit termistön esim. tällaisessa pätkässä:

" 3.5.1 The pc Register

> The Java *virtual machine* can support many *threads* of *execution* at once (§2.19). Each Java *virtual machine thread* has its own *pc (program counter) register.* At any point, each Java *virtual machine thread* is *executing* the *code* of a single *method*, the current *method* (§3.6) for that *thread.* If that *method* is not *native*, the *pc register* contains the *address* of the Java *virtual machine instruction* currently being *executed.* If the *method* currently being *executed* by the *thread* is *native*, the *value* of the Java *virtual machine's pc register* is undefined. The Java *virtual machine's pc register* is wide enough to hold a *returnAddress* or a *native pointer* on the *specific platform.* "

" 3.5.2 Java Virtual Machine Stacks

> Each Java *virtual machine thread* has a private Java *virtual machine stack*, created at the same time as the *thread.* A Java *virtual machine stack* stores *frames* (§3.6). "

Eli mikä hyöty? Ainakin olet abstraktimmalla tasolla kuin jos vain osaisit ohjelmoida yhdellä työkalulla: ymmärrät, miten tietokoneet suorittavat ohjelmia riippumatta työkalusta jolla koodi on kirjoitettu. Toisaalta on ehkä syntynyt pienet edellytykset lukea dokumentaatiota erilaisten koneiden tai virtuaalikoneiden toteutuksesta ja vertailla eri alustojen teoreettista soveltuvuutta suunnitteilla olevaan järjestelmääsi. "Toimiiko se nyt sitten noin vai näin?" on kysymys, johon usein täytyy saada täsmällinen vastaus eikä arvaus. Edelleen kysymys tarkoittaa usein "Toimiiko *softan suoritus* nyt sitten *missäkin erityistilanteessa* noin vai näin?". Varminta on aina lukea speksi (kääntäjän ja tarvittaessa laitteiston) ja todeta mitä se sanoo suhteessa lähdekoodiin.

3.5 Tuota lopullinen opinnäyttö

Oletamme, että olet ajanut oman konekielelle käännetyn C-ohjelmasi gdb:llä samoin kuin edellä tehtiin malliharjoitustyölle. Ajo on tallessa tekstitiedostossa. Saat toki soveltaa gdb:n käskyjä ja suoritusjärjestystä vapaasti, kunhan pystyt tuottamaan alla mainitun ohjeen mukaisen vastauksen.

Ota uudelleen käsittelyyn se, mitä gdb-sessiossa tapahtui. Eli ota syntynyt tekstitiedosto auki lemppaitekstieditoriisi ja kirjoita sen sekaan seuraavat huomiot:

- Otit heti alussa pääohjelmastasi disassemblyn. Myöhemmin otit aliohjelmasta vastaavan. Osoita tallentuneiden disassembly-rivien perään kirjoitetulla tekstillä, että olet ymmärtänyt miten aliohjelmaa kutsutaan ja miten sieltä palataan, nimenomaan oman ohjelmasi suorituksessa. Eli kohdenna vastaus aliohjelman disassemblyn alkuun ja loppuun sekä main-aliohjelman disassemblyssä niiden käskyjen kohdalle, jotka kävit kutsun osalta läpi konekäsky kerrallaan. Jokaisen aktivaatioon liittyvän koodirivin perään kirjoita seuraavan muotoinen kommentti:

  ```
  0x000000000040066d <main+92>:   callq  0x4004f8 <kaan-
  na_taulukko>
  HARKKA: Yllä olevan käskyn rooli aliohjelmakutsuni toteutuksessa
          on ... [tai vastaava aloitus, jolla aloitat selityksen,
          mikä on käskyn merkitys kokonaisuuden kannalta]
  ```

  Aliohjelman disassemblyssä tee samoin jokaisen aktivaatioon liittyvän koodirivin perään, siis aliohjelman alussa ja lopussa tiettyyn määrään rivejä. Muita kuin aliohjelmamekanismiin liittyviä rivejä ei tarvitse kommentoida mitenkään. Esimerkki:

  ```
  0x00000000004004f9 <kaanna_taulukko+1>: mov    %rsp,%rbp
  ```

```
HARKKA: Yllä olevan käskyn rooli aliohjelmakutsuni toteutuksessa
        on ... [tai vastaava aloitus, jolla aloitat selityksen
        mikä on käskyn merkitys kokonaisuuden kannalta]
0x00000000004004fc <kaan-
na_taulukko+4>: mov    %rdi,0xffffffffffffffd8(%rbp)
HARKKA: ...[Tämän rivin merkitys ei ole niin kriittinen,
        mutta saa nämäkin toki huvikseen kommentoida;
        liittyväthän nämä pinokehyksen käyttöön... ]

... sitten kun tulee algoritmin toteutuksellisten rivien
käännös, ei tarvitse kommentoida ollenkaan...
```

Tarkastan harjoitustyöt seulomalla automaattisesti rivit, joilla lukee `HARKKA:` ja pari riviä siitä ympäriltä; silmäilen ne läpi, ja tekstien pitää osoittaa ymmärrystä. Pitkiä niiden ei tarvitse olla, mutta ymmärrystä ei vielä osoita pelkästään sanoa "käsky siirtää RSP:n RBP:hen" tai muu suomennos käskystä "mov %rsp,%rbp"; jokaiseen käskyyn on joku syy/merkitys joka pitää ymmärtää ja siis osata selittää nimetyillä käsitteillä!

- Otit myös aliohjelmastasi disassemblyn, ja yhden kutsun osalta tulostit pinomuistin sisällön ennen ja jälkeen pinokehyksen luonnin. Tulosteita käyttäen hahmota, missä kohtaa muistia (tai prosessoria) mikäkin asia sijaitsee. Kirjoita seuraavan muotoiset rivit, joissa kerrot jokaisen C-lähdekoodissasi olevan muuttujan sekä asian muistiosoitteet juuri kyseisen aliohjelma-aktivaation aikana. Eli sijoita rivit vaikkapa siihen kohtaan tekstitiedostoasi, missä gdb suorittaa jo ensimmäistä aliohjelman C-koodiriviä, ts. aktivaatio on kunnolla aloitettu. Käytä osoitteille heksalukuja kuten gdb. Käytä nyt absoluuttisia virtuaalimuistiosoitteita, älä suhteellisia rekistereihin nähden:

```
HARKKA: Muuttujani i on muistiosoitteessa 0x287562858
HARKKA: Muuttujani j on muistiosoitteessa 0x287562850
...

[Käsittele kaikki C-lähdekoodissasi olevat muuttujat, niin
lokaalit muuttujat kuin parametritkin.]

Taulukkoparametrin osalta selvennä seuraavin tavoin:

HARKKA: Aliohjelmani taulukkoparametrin NIMI tiedot
        sijaitsevat muistiosoitteissa ALKUOSOITE - LOPPUOSOITE

HARKKA: Aliohjelmani saa taulukkoparametrin NIMI sisältämän
        datan käyttöönsä seuraavalla tavoin: ...
```

- Samaan kohtaan kerro, mitä heksalukuja seuraavat muistiosoitteet olivat siinä vaiheessa kun gdb-ajosi tapahtui:

```
HARKKA: Tämän aktivaation pinokehyksen kantaosoite on ...
HARKKA: Edellisen aktivaation kehyksen kantaosoite on ...
HARKKA: Edellisen aktivaation kehyksen kantaosoite on tallennet-
tu
        muistiosoitteeseen ...
HARKKA: Pinon huipun osoite tällä hetkellä on ...
```

- Pääohjelmassa aktivaation jälkeen kerro seuraavaa:

```
HARKKA: Aliohjelmastani on juuri palattu; sen paluuarvo
        sijaitsee nyt ... [missä]
```

Siinäpä tärkeimmät. Nämä asiat tulee ymmärtää konekieliohjelmoinnista ja aliohjelman suoritusperiaatteesta Käyttöjärjestelmät -kurssin puitteissa. Yllämainitut vastaukset lisättynä gdb-tulosteeseen riittävät mielestäni asian osoittamiseksi.

4 Palautus

Harjoitustyössä tuotetaan edellä kerrottujen ohjeiden mukainen tekstitiedosto, jonka sisältö tulee lähettää heinäkuun loppuun mennessä Jalavasta mail-ohjelmalla samalla tavalla kuin demot; sähköpostin otsikon tulee olla täsmälleen "`mun harkka`"

Ai niin, **lähetä samassa pötkössä myös C-kielinen koodisi**; muistuu sitten paremmin mieleen, mikä algoritmi siellä olikaan toteutettu.

Jos työ on valmis ja palautettu ennen kesän ensimmäistä tenttipäivää (26.7. klo 23:59 mennessä), annan siitä +1 pistettä bonusta ensimmäiseen tenttiyritykseen; tämä ajatellaan olevan demo4:n toinen osio, josta lupasin bonuspisteen.