Tölvutækni og forritun Heimadæmi 10

brj46

október 2024

HÁSKÓLI ÍSLANDS

1

a)

Gefin er forritsbútur í C. Hvert er lokagildið a breytinnu **x**

```
short int x;
x = 0x2051;
x = (x | 12) >> 3;
```

Skoðum gildin á x

1. x fær hex gildið 0x2051 sem er í binary:

   ```
   0010 0000 0101 0001
   ```

2. síðan eð-um við x með 12

   ```
   0010 0000 0101 0001
   OR
   0000 0000 0000 1100
   -------------------
   0010 0000 0101 1101
   ```

3. hliðrum næst með 3 til hægri:

   ```
   0000 0100 0001 0111
   ```

4. Að lokum er gildið á **x**: **0000 0100 0001 0111** sem er **1047** í tugakerfinu og **0x0417** i hex

b)

við fáum gefinn forritsbútinn í c

```
int v = 13;
for (c = 0; v; c++)
v = v & (v-1);
```

i)

Hvert er lokagildið á breytunum **v** og **c**?

Förum í gegnum forritið

1. v fær 13 sem er 1101_2
2. for lykkja þar sem c upphafsstillist sem 0 og keyrist þegar **V** er satt eða ekki 0 og c hækkar um 1 eftir hverja umferð

   í lykjunni er v = v og-uð við v -1
3. fyrsta keyrsla á lykkjunni
   - v = 13
   - c = 0
   - v = 1101 og-uð við 1100

     ```
     1101
     AND
     1100
     ----
     1100
     ```

   - **v** = 1100_2 (12)
   - c++;
   - c = 1;
4. Önnur umferð lykkjunar
   - v = 1100_2
   - c = 1
   - v = 1100 og-uð við 1011

     ```
     1100
     AND
     1011
     ----
     1000
     ```

   - **v** = 1000_2 (8)
   - C++
   - C = 2

5. Þriðja umferð
    - **v** = 1000_2 (8)
    - **c** = 2
    - v = 1000 og-uð við 0111 = 0000
    - v = 0
    - c++
    - c = 3
6. Lykkjan stöðvast og því $v = 0$
7. Lokagildi breytanna v og c

$$\mathbf{v} = 0$$

$$\mathbf{c} = 3$$

ii)

Hvert verður almennt lokagildi á **c** út frá upphafsgildi **v**?

Ef við skoðum þá forritið í almennum skilningi

```
int v ...;
for(c = 0; v; c++)
    v = v & (v-1);
```

Lykkjan keyrir meðan v er ekki núll og í hverri ítrun er v uppfært með og-un við $v - 1$

þessi og-unar aðgerð er notuð til að telja fjölda ǫn"bita í heiltölu svo ef V er t.d. $v = 1111_2$ ætti lykkjan að keyra 4 sinnum þar sem það eru fjórir "1"bitar og $v = 1001$ ætti að keyra tvisvar þar sem það eru tveir "1"bitar eða ǫn"bitar.

c heldur utan um hversu oft lykkjan keyrist þar sem c hækkar um einn í við hverja umferð lykkjunar.

Almment lokagildi á c er háð því hversu margir "1"bitar eru í upphafsgildi v

2

Við viðhald á hugbúnaði hefur komið í ljós að upphaflegur frumkóði (source code) eins fallsins hefur glatast, en það er til viðfangsskrá þess, **bla.o.** Einnig hefur varðveist haus fallsins og hann er "**int bla(int i, int j, int k)**". Hér fyrir neðan er úttakið úr skipuninni **objdump -d bla.o**"

Endurskrifum frumkóða fallsins í C út frá smalarmálskóðanum hér fyrir neðan:

```
0000000000000000 <bla>:
   0:    48 89 d0         mov %rdx,%rax
   3:    48 39 f7         cmp %rsi,%rdi
   6:    7c 03            jl b <bla+0xb>
   8:    48 89 fe         mov %rdi,%rsi
   b:    48 39 c6         cmp %rax,%rsi
   e:    7c 03            jl 13 <bla+0x13>
   10:   48 89 f0         mov %rsi,%rax
   13:   c3               ret
```

Byrjum á að skoða hvernig fallið virkar.

Fallhausinn: **int bla(int i, int j, int k);** við vitum að i fer í `%rdi`, j fer í `%rsi` og k fer í `%rdx` og niðurstöðum er skilað í `%rax`

1. `0:  48 89 d0  mov %rdx,%rax`

   Þetta setur rdx í rax (setur k í rax)

2. `3:  48 39 f7  cmp %rsi,%rdi`

   þetta ber saman rsi við rdi, eða öllu heldur **ef j er stærra en i**

3. `6:  7c 03  jl b <bla+0xb>`

   ef skilyrðið $i < j$ fer forritið í linu 0xb

4. `8:  48 89 fe  mov %rdi,%rsi`

   ef að j var minna eða jafnt og i er þessi lína framkvæmd og setur hún rdi í rsi (setur j = i)

5. `b:  48 39 c6  cmp %rax,%rsi`

   Þessi lina ber saman rax sem er núna k við j ($j < k$)

6. `e:  7c 03  jl 13 <bla+0x13>`

   Ef að ($j < k$) þá er hoppað í línu 13

7. `10:  48 89 f0  mov %rsi,%rax`

   Þá ef ($j >= k$) er sett j í rax

8. `13:  c3  ret`

   return skilar rax gildinu.

Skrifum nú forritið í C kóða

```c
int bla(int i, int j, int k) {
    if (i >= j)
        j = i;
    return (j >= k) ? j : k;
}
```

3

Hér fyrir neðan eru tvær skrár **main.c** og **fun.c**, með einu falli hver.

```c
void fun();
extern int b;
short int c = 5;
int main() {
    b = 3;
    fun();
    c = 6;
    return 0;
}
```

og

```c
#include <stdio.h>
short int a = 2
short int b = 8;
extern short int c;
void fun() {
    printf("a: %d\n", a);
    printf("b: %d\n", b);
    printf("c: %d\n", c);
}
```

a)

Fyrir hvert víðvært tákn (global symbol) í skránum tveimur ætlum við að segja hvort skilgreining þess sé sterk eða veik

Byrjum að skoða main.c

- **fun**: þetta er ytri tilvísun og er því **ekki skilgreining**
- **b**: þetta er einnig ytri tilvísun og því **ekki skilgreining**
- **c**: þetta er **Sterk** skilgreining þar sem hún er skilgreind og frumstillt:(**short int c = 5;**)
- **main**: Hér er main fallið **Sterk** skilgreining þar sem fallið er skilgreint

Skoðum núna fun.c:

- **a**: þetta er **Sterk** skilgreining þar sem hún er skilgreind og frumstillt(**short int a = 2;**)
- **b**: Þetta er **Sterk** skilgreininga þar sem hún er skilgreind og frumstillt(**short int b = 8;**)
- **c**: þetta er ytri tilvísun og er því **ekki skilgreining**
- **fun**: þetta er skilgreint og er því **Sterk** skilgreining

b)

Þegar við þýðum forritin og leiðréttum að það vanti semíkommu í short int a = 2 í fun.c

```
gcc -Og -o main main.c fun.c
```

og keyrum síðan forritið þá fáum við:

```
./main
a: 0
b: 3
c: 5
```

setjum upp töflu:

breyta	Gildi	Skýring
a	0	Við fáum út 0 en ekki 2 eins og var búist við sennilega vegna þess að það er verið að skrifa eitthversstaðar yfir a. ef við myndum setja static fyrir framan a þá myndum við fá 2 út eins og búist var við
b	3	Við sjáum að extern int í main er með forgang og er þess vegna b = 3 frekar en b = 8.
c	5	þetta er skilgreint í main með short int c = 5 og fun er kallað áður er c er breytt í 6.

4

Kyrrleg (static) tenging og kvik (dynamic) tenging eru tvær ólíkar aðferðir við að tengja forritasöfn við forrit sem nota þau.

a)

Nefnum tvo kosti kyrrlegrar tengingar miðað við kvika tengingu og rökstyðum þá

1. **Sjálfstæð keyrsluskrá:** Með kyrrstæðri tengingu eru öll nauðsynleg forritasöfn innbyggð beint í keyrsluskrána. Þetta þýðir að forritið þarf ekki að treysta á ytri forritasöfn á keyrslutíma.

   Þar sem forritið inniheldur allt sem það þarf, getur það keyrt á hvaða tölvu sem er án þess að þurfa sérstök forritasöfn uppsett. Þetta eykur flytjanleika og gerir dreifingu á forritinu einfaldari.

2. **Hraðari keyrsla:** Kyrrstæð tenging getur leitt til hraðari keyrslu þar sem allar tengingar eru gerðar við þýðingu, ekki á keyrslutíma.

   Þar sem allar nauðsynlegar upplýsingar eru þegar til staðar í keyrsluskránni, þarf forritið ekki að leita að og hlaða inn ytri söfnum þegar það keyrir. Þetta getur minnkað upphafstíma og bætt heildarframmistöðu.

b)

Nefnum tvo kosti kvikrar tengingar miðað við kyrrlega tengingu og rökstyðjum þá

1. **Minni notkun á plássi og minni:** Með hreyfanlegri tengingu geta mörg forrit deilt sömu forritasöfnum, bæði á diski og í vinnsluminni.

   Þetta sparar pláss þar sem aðeins eitt afrit af hverju forritasafni er geymt, sem forritin deila. Einnig sparar þetta vinnsluminni þar sem kóðinn þarf ekki að vera hlaðinn inn fyrir hvert forrit sérstaklega.

2. **Auðveldari uppfærslur:** Hreyfanleg tenging gerir það einfaldara að uppfæra forritasöfn án þess að þurfa að endurþýða öll forrit sem nota þau.

   Ef villur finnast eða ný virkni er bætt við í forritasafni, er nóg að uppfæra safnið sjálft. Öll forrit sem nota það munu þá sjálfkrafa nýta sér uppfærsluna við næstu keyrslu, sem einfalda viðhald og eykur öryggi.

5

Hér fyrir neðan eru tvö pör af forritunarskrám sem við tengjum saman. Svörum í hvoru tilfelli eftirfarandi spurningum: kemur villa í tengingu? Ef svo er, hver er vandamálið? Ef ekki kemur villa sýnið úttakið og rökstyðjið það í nokkrum orðum:

a1.c

```
#include <stdio.h>
void f();
extern int x;
int main() {
    int x = 2;
    f();
    printf("x: %d\n", x);
    return 0;
}
```

a2.c

```
int x = 3;
void f() {
    x = 4;
}
```

b1.c

```
void g();
int y = 10;
int main() {
    g();
    y = 12;
    g();
    return 0;
}
```

b2.c

```
#include <stdio.h>
extern int y;
void g() {
    printf("y: %d\n", y);
}
```

a

Við tengingu með:

```
gcc -Og -o a a1.c a2.c
```

kemur engin villa.

Við keyrslu fáum við:

```
./a x: 2
```

Skýring:

Í **a1.c** er breytan `x` skilgreind sem staðvær/local breyta inni í `main()` og hefur því aðeins scope innan þess falls. Í **a2.c** er breytan `x` skilgreind sem global breyta. Þrátt fyrir að nota `extern int x;` í **a1.c**, þá er staðværa breytan `x` í `main()` að skyggja á global breytuna `x` úr **a2.c**.

Þegar fallið `f()` er kallað í **a1.c**, þá breytir það global breytunni `x` í 4. Hins vegar hefur það engin áhrif á local breytuna `x` í `main()`, sem heldur áfram að hafa gildið 2. Þess vegna prentast `x: 2`.

b

Við tengingu með:

```
gcc -Og -o b b1.c b2.c
```

kom engin villa.

Við keyrslu fáum við:

```
./b y: 10 y: 12
```

Skýring:

Í **b1.c** er breytan `y` skilgreind sem global breyta með upphafsgildið 10. Í **b2.c** er `y` lýst sem ytri breytu með `extern int y;`, sem vísar í sömu breytu og í **b1.c**.

Fallið `g()` prentar gildið á `y`. Í `main()` er fyrst kallað á `g()`, sem prentar `y: 10`. Síðan er `y` breytt í 12, og aftur er kallað á `g()`, sem prentar `y: 12`.

Í hvorugu tilfellinu kemur villa við tengingu. Í a hefur local breytan `x` í `main()` forgang yfir global breytunni `x` í **a2.c**, þannig að breytingar í `f()` hafa ekki áhrif á hana. Í b er global breytan `y` sameiginleg milli skráa og breytingar á henni endurspeglast í báðum skrám.