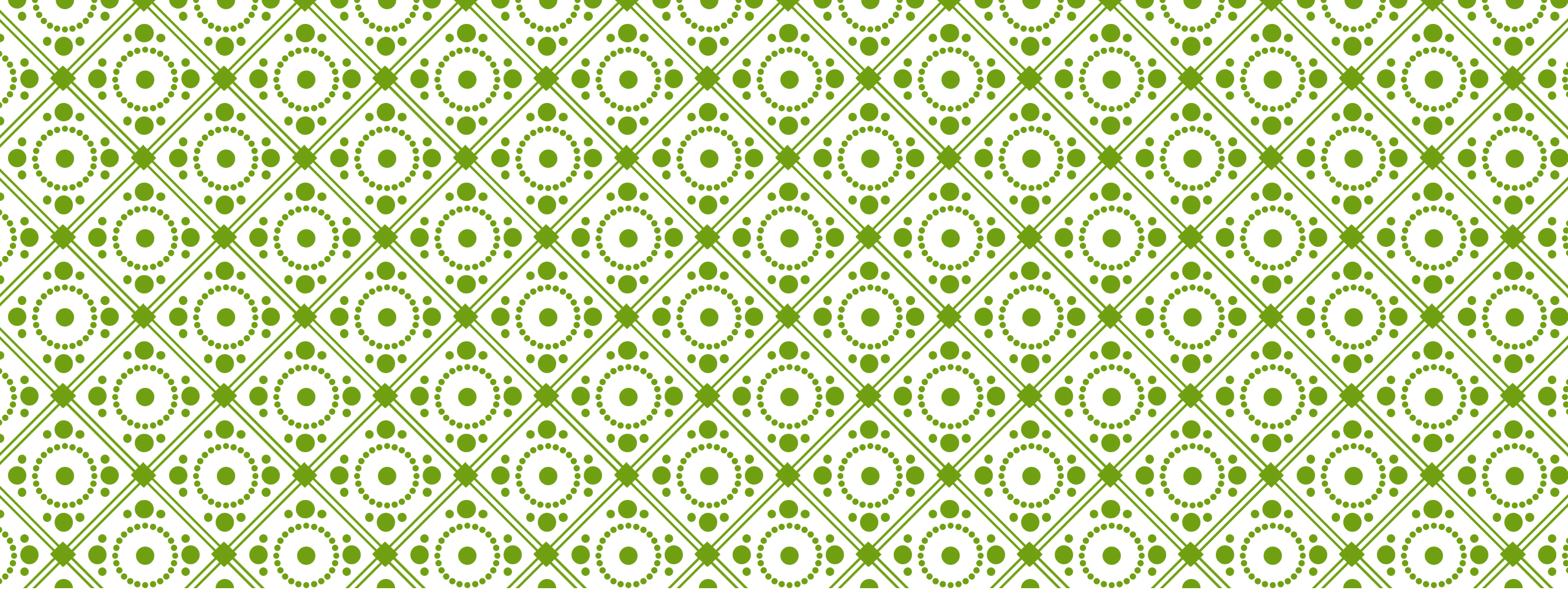


МЕМОРИЈА И ОСНОВЕ АЛАТА GDB

Програмирање у реалном времену 2020

Вјежба 1

УВОД

Потребна знања

- Це програмски језик, компајлирање
- Основе Линукса
- Основно познавање асемблерских наредби
- Одслушан дио вјежби са предмета Оперативни Системи у реалном времену које се односе на RaspberryPi уређај

Циљ вјежби

- Упознати се са одређеним аспектима меморије (стек сегмент, текст сегмент, сегмент података и BSS) и утицаја изворног кода на меморију
- Упознати се и оспособити се за коришћење gdb дебугера за проналажење грешака у изворном коду
- Препознати лоше концепте у програмирању са становишта меморије.

ШТА ЈЕ GDB?

GNU дебугер

Користи се за неколико програмских језика укључујући Це и Це++

Омогућује увид у то шта се дешава у одређеној програмској тачки током извршења програма

Најзанимљивије грешке у Це језику, *Segmentation Fault* се лакше пронађу са дебугером

Документација се може наћи на https://sourceware.org/gdb/documentation/

КОМПАЈЛИРАЊЕ

Компајлирање је мало другачије

Уобичајено се компајлира:

```
gcc [flags] <source files> -o <output file>
```

Примјер

```
gcc –Wall –Werror –ansi –pedantic-errors program.c –o program.exe
```

Додавање опције –g како би се омогућила подршка дебуговању:

```
gcc [flags] –g <source files> -o <output file>
```

```
gcc –Wall –Werror –ansi –pedantic-errors –g program.c –o program.exe
```

ПОКРЕТАЊЕ ДЕБУГЕРА

Два начина покретања дебугера:

- `gdb program.exe`
- `gdb`
  - У овом случају, потребна је додатна команда када се уђе у дебугеров едитор режим *(shell)*:
  - **`(gdb)`** `file program.exe`
  - Гдје је **`file`** команда којом се учитава програм

Када смо већ код режима, сличан је Линуксовом:

- Са стрелицама се може „ходати“ по историји
- Постоји и `echo`
- ТАБ омогућује завршавање наредбе, и тако даље...
- А ако не знаш шта нека команда ради, покушај:
  - **`(gdb)`** `help [komanda]`

ПОКРЕТАЊЕ ПРОГРАМА КОЈИ СЕ ДЕБАГУЈЕ

```
(gdb) run
```

Очигледно!

Ова команда ће покренути програм који је учитан (на један од два претходно наведена начина).

- Ако нема проблема, исти програм (у нашем случају program.exe) ће се исправно извршити, у којем случају и нема неке потребе користити дебугер. Осим, можда у едукативне сврхе.
- Ако има проблема, онда ће моћи да се виде много корисније информације од оних које се иначе добију приликом покретања програма. Нпр број линије и параметри функције која их је узроковала

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400524 in funkcija(podatak=0x7fffc90270,
drugipodatak=2) at program.c:12
```

ЗАУСТАВНЕ ТАЧКЕ *(BREAKPOINTS)*

Нема превише смисла без зауставних тачки. Имају исти ефекат као и у графичким развојним окружењима. Заустављају даље извршавање програма на датом мјесту у програму.

Подешавање зауставне тачке на линију у фајлу:

```
(gdb) break program.x:6
```

или без назива програма

```
(gdb) break 6
```

Подешавање зауставне тачке на main функцију:

```
(gdb) break main
```

Уобичајен је случај да се не стаје сваки пут на неку линију или функцију, него када се испуни неки услов:

```
(gdb) break program.x:6 if i < 5
```

Број зауставних тачки је произвољан, а програм се, наравно, заустављa сваки пут када наиђе на неку од њих приликом извршавања програма.

Наравно, зауставне тачке не морају бити дефинисане прије покретања програма (`run`)

ШТА КАДА СЕ ПРОГРАМ ЗАУСТАВИ?

Двије ствари имају смисла:

Исписати податке из меморије:

- **(gdb)** `display/[format] izraz` – израчунава и исписује `izraz` сваки пут када се извршавање програма заустави. Овај скуп информација се дефинише прије покретања програма (`run`).
- **(gdb)** `x/[format] adresa` – исписује садржај адресе у датом формату.
- **(gdb)** `print /[format] varijabla` – исписује вриједност варијабле.
- Формат се састоји од типа и дужине.
  - Тип је слово: Октални **o**, хексадецимални **x**, децимални **d**, неозначени децимални **u**, бинарни **b**, покретни зарез **f**, адреса **a**, инструкција **i**, карактер **c**, стринг **s** или хексадецимални за нулама **z.**
  - Дужина је број праћен словом: бајт **b**, пола ријечи **h**, ријеч **w** или осам бајтова **g**
  - На примјер: **8bx** (осам бајтова у хексадецималном формату) или **2wd** (двије ријечи у децималном формату)
- **(gdb)** `watch varijabla` – Исписује вриједност варијабле када год се она промијени. Односи се на варијаблу која се тако зове у том скопу, у којем се програм налази када се наредба откуца!!!
- **(gdb)** `info [komanda]` – практично све остало исписује.
- **(gdb)** `disas /s` – исписује асемблер код. Опција /s има највише смисла, пошто исписује и изворни код
- **(gdb)** `list – исписује изворни код`

Наставити даље (у наставку)...

ШТА КАДА СЕ ПРОГРАМ ЗАУСТАВИ? [НАСТАВАК ☺]

Како наставити када смо се зауставили? Било то очигледно или не, али ако укуцамо **`run`** поново ћемо покренути програм из почетка!

- `(gdb) continue` – наставља до сљедеће заустављене тачке
- `(gdb) step [n]` – извршава сљедећих **`n`** линија програмског кода. Уколико је параметар изостављен, извршава се једна линија програмског кода. Уколико је линија програмског кода позив функције, улази у функцију.
- `(gdb) next [n]` – у већој мјери ради исто што и `step`, са разликом да уколико је линија програмског кода позив функције, дебагер не улази у функцију.
- `(gdb) stepi [n]` – извршава се **`n`** (или једна уколико је изостављено) инструкција.

Пошто је много заморно стално куцати ове наредбе, добро је да се зна да ЕНТЕР понавља задњу откуцану наредбу

ОСТАЛЕ КОРИСНЕ КОМАНДЕ

(gdb) `backtrace` – исписује траг стека када се програм убије

(gdb) `where` – исто као и `backtrace`, само што је програм још жив!

(gdb) `finish` – иде до краја функције

(gdb) `delete` – брише зауставну тачку

(gdb) `set var=val` – поставља нову вриједност за дату варијаблу у датом скопу

(gdb) `set logging [komanda]` – поставља логовање. Команда може бити: `file`, `on/off`, `overwrite`, `redirect`

(gdb) `quit` – излаз из дебугера

Добро је знати да све ове команде имају своје сктраћене називе:

run – r, break – b, delete - d, continue – c, finish – f, step – s, print – p, quit – q, backtrace – bt, next – n

ЧИТАЊЕ СТЕКА

Битни регистри

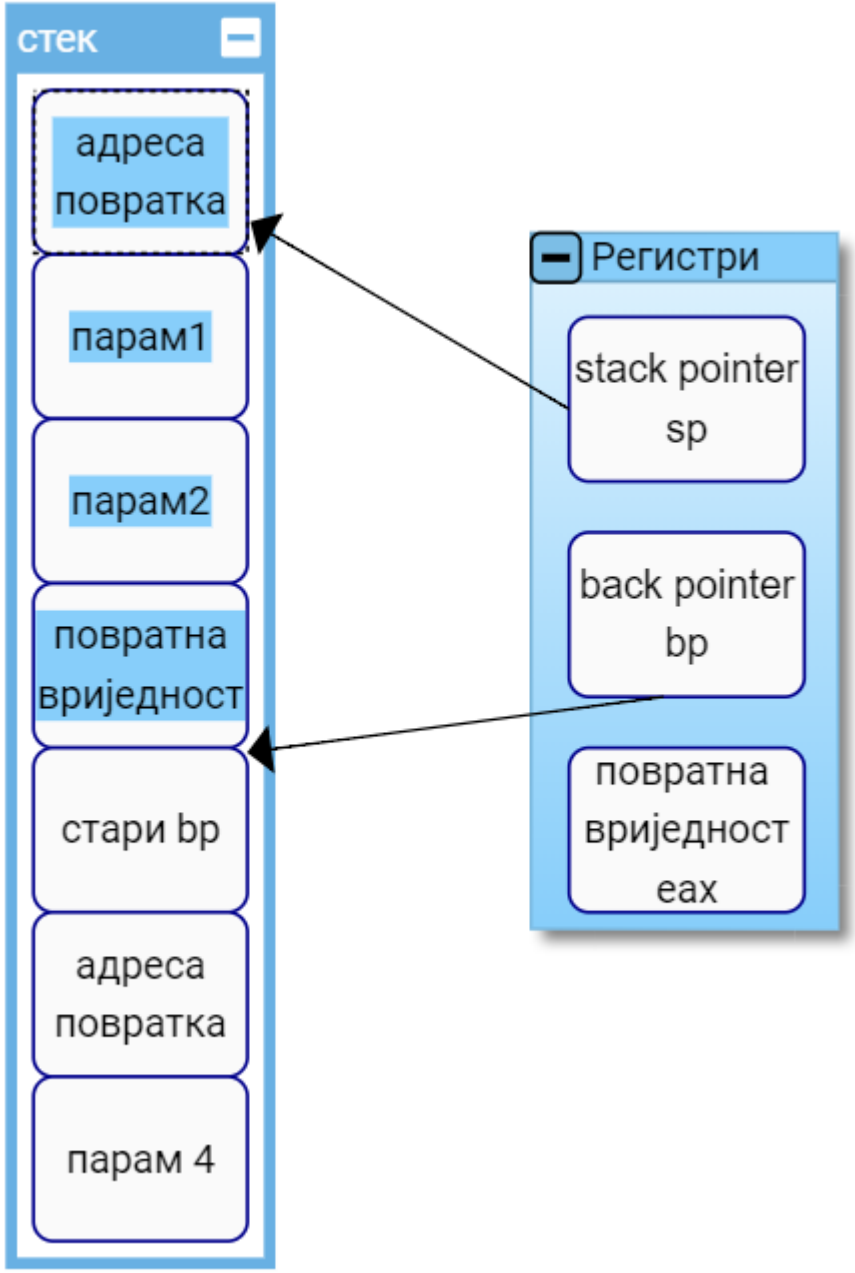
`(e)sp` – показивач на врх стека

`(e)bp` – користи се као референца када се приступа локалним промјењивим и аргументима функције

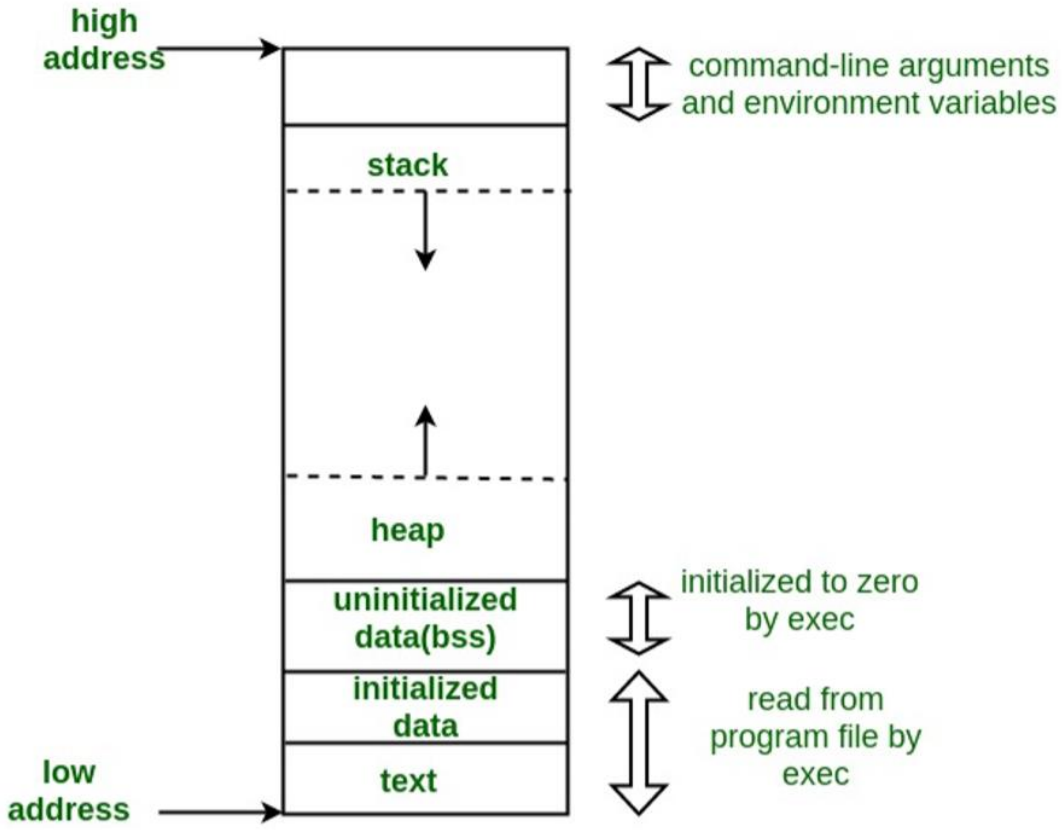
(gdb) `info frame`

(gdb) `info registers sp bp`

(gdb) `x/20xw $sp`



МЕМОРИЈА



Стек сегмент садржи податке унутар функција. И main је функција.

ХИП садржи динамички алоциране податке

Bss садржи податке који су иницијализовани са нулама или нису иницијализовани

Сегмент података садржи иницијализоване податке.

Текст сегмент садржи програмски код. Овај дио меморије је *read only*

ЛИНУКС КОМАНДА SIZE

Команда `size` исписује величине меморијских сегмената за неки програм:

```
srdjan@ubuntu:~/ppurv/vjezba1$ size addnumber
   text    data     bss     dec     hex filename
   1677     608       8    2293     8f5 addnumber
```

На овај начин се може статички провјерити програм. Док није у извршавању.

Пробати сљедеће ствари:

1. Избрисати `static` промјењивој *`poruka`* (компајлирати под другим именом и провјерити *`size`*)
2. Додати још једну глобалну промјењиву по узору на *`suma`* – без иницијализовања. Компајлирати и погледати величине.
3. Иницијализовати нову глобалну промјењиву. Погледати величине сегмената

ЛИНУКС КОМАНДА NM

Команда nm исписује све симболе који су присутни у компајлиранм програму.

На слици је приказан дио листинга за компајлирани програм.

Прва колона је адреса у фајлу, друга је тип (D,d – сегмент података, T,t – Текстуални сегмент, B,b – bss... Остали типови тренутно нису битни), а трећа садржај меморије.

Како је ова листа превелика углавном се грепује:

```
nm addnumber | grep poruka
```

ПОТРАЖИТИ poruka У СВАКОМ КОМПАЈЛИРАНОМ ФАЈЛУ.

```
srdjan@ubuntu:~/ppurv/vjezba1$ nm addnumber
000000000000065a T add_numbers
0000000000201018 B __bss_start
0000000000201018 b completed.7698
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000201000 D __data_start
0000000000201000 W data_start
0000000000000580 t deregister_tm_clones
0000000000000610 t __do_global_dtors_aux
0000000000200dc0 t __do_global_dtors_aux_fini_a
0000000000201008 D __dso_handle
0000000000200dc8 d _DYNAMIC
0000000000201018 D _edata
0000000000201020 B _end
```

ЗАДАТАК

Користећи програм add_number.c и gdb команде из gdb_komande.txt:

- Компајлирати Це програм за дебаговање
- Дебаговати програм упознавајући се са адресама појединих промјењивих, као и стања стека
- Користећи size наредбу препознати промјене у величини меморијских сегмената промјеном начина дефинисања промјењивих.

Користећи изворни код програма kanarinac.c као и стање стека при дебаговању add_number.c, препознати разлоге заштите стека.

ДОМАЋИ ЗАДАТАК

Користећи gdb алат, дебаговати добијени задатак и пронаћи грешку.

Како би били сигурни да Чак Норис није помагао у дебаговању, резултат дебаговања мора бити лог фајл креиран на сличан начин као што је урађено у фајлу gdb_komande.txt из којег ће бити евидентан процес дебаговања.

Услов за присуствовање сљедећим лабораторијским вежбама је успјешно урађен домаћи задатак.