Волохань Николай КН-403
Лабораторная работа №1

Упражнение 2. Используйте команду GDB `si` (Пошаговая инструкция), чтобы выполнить трассировку в BIOS ROM для получения еще нескольких инструкций, и попытайтесь угадать, что она может делать.

```
gdb -n -x .gdbinit
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
+ target remote localhost:26000
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
The target architecture is set to "i8086".
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
warning: A handler for the OS ABI "GNU/Linux" is not built into this configurati
on
of GDB.  Attempting to continue with the default i8086 settings.

(gdb)
```

После того, как мы установим среду отладки в Lab 1, Упражнение 2, первая команда для запуска BIOS после запуска ПК будет отображена в окне запуска gdb следующим образом ：

[f000:fff0] 0xffff0: ljmp $0x3630, $0xf000e05b

Это первая команда для запуска, это команда перехода, которая переходит к адресу 0xf000:e05.

```
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
```

В следующей инструкции сравнивается значение 0xffc8 со значением по адресу памяти, представленному %cs:(%esi). %cs:(%esi) - это формат формирования адреса в реальном режиме, который вычисляется путем объединения сегментного регистра “cs” и адреса, который находится в регистре “esi”.

```
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b0
0x0000e062 in ?? ()
(gdb)
```

Инструкция jne: если установлен флаг ZF равный 0, она переходит, то есть, когда значение по адресу %cs:(%esi) не равно 0xffc8, она переходит.

```
(gdb) si
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb)
```

Адрес следующей инструкции равен 0xfe066, видно, что приведенная выше инструкция перехода не выполняла переход. Функция этой инструкции состоит в том, чтобы очистить регистр edx до нуля.

```
(gdb) si
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
(gdb) si
[f000:e06a]    0xfe06a: mov    $0x7000,%sp
0x0000e06a in ?? ()
(gdb) si
[f000:e070]    0xfe070: mov    $0xfc1c,%dx
0x0000e070 in ?? ()
(gdb) si
[f000:e076]    0xfe076: jmp    0x5576cf2d
```

Следующие инструкции заключаются в установке значений некоторых реистров. В последней инструкции содержится безусловный переход.

```
(gdb) si
[f000:cf2b]    0xfcf2b: cli
0x0000cf2b in ?? ()
(gdb)
```

“cli” - отключает прерывания процессора. Эту инструкцию можно использовать, когда необходимо гарантировать, что никакие прерывания во время выполнения конкретного блока кода не будут вызваны.

```
(gdb) si
[f000:cf2c]    0xfcf2c: cld
0x0000cf2c in ?? ()
(gdb)
```

“cld” - устанавливает флаг направления (direction flag) в регистре флагов в значение "0". Флаг направления в регистре флагов используется для указания направления выполнения инструкций. Когда установлен флаг направления (DF = 1), указатель сдвигается назад при выполнении инструкций, а при сбросе флага направления (DF = 0), по умолчанию указатель сдвигается вперед.

При запуске BIOS настраивается таблица дескрипторов прерываний и инициализируются различные устройства, такие как дисплей VGA. После инициализации шины PCI и всех важных устройств, о которых известно BIOS, выполняется поиск загрузочного устройства, такого как дискета, жесткий диск или CD-ROM. В итоге, когда он находит загрузочный диск, BIOS считывает загрузчик с диска и передает ему управление.

Упражнение 3. Установите точку останова по адресу 0x7c00, по которому будет загружен загрузочный сектор. Продолжайте выполнение до этой точки останова. Выполните трассировку кода в `boot/boot.S`, используя исходный код и файл дизассемблирования `obj/boot/boot.asm`, чтобы отслеживать, где вы находитесь. Также используйте команду `x/i` в GDB для дизассемблирования последовательностей инструкций в загрузчике и сравните исходный код загрузчика как с дизассемблированием в `obj/boot/boot.asm`, так и с GDB.

Выполните трассировку в `bootmain()` в `boot/main.c`, а затем в `readsect()`. Определите точные инструкции по сборке, соответствующие каждому из операторов в `readsect()`. Выполните трассировку через остальную часть `readsect()` и вернитесь обратно в `bootmain()` и определите начало и конец цикла `for`, который считывает оставшиеся сектора ядра с диска. Выясните, какой код будет выполняться по завершении цикла, установите там точку останова и продолжайте до этой точки останова. Затем выполните остальные действия загрузчика.

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb)
```

```
(gdb) x/30i 0x7c00
=> 0x7c00:      cli
   0x7c01:      cld
   0x7c02:      xor    %eax,%eax
   0x7c04:      mov    %eax,%ds
   0x7c06:      mov    %eax,%es
   0x7c08:      mov    %eax,%ss
   0x7c0a:      in     $0x64,%al
   0x7c0c:      test   $0x2,%al
   0x7c0e:      jne    0x7c0a
   0x7c10:      mov    $0xd1,%al
   0x7c12:      out    %al,$0x64
   0x7c14:      in     $0x64,%al
   0x7c16:      test   $0x2,%al
   0x7c18:      jne    0x7c14
   0x7c1a:      mov    $0xdf,%al
   0x7c1c:      out    %al,$0x60
   0x7c1e:      lgdtl  (%esi)
   0x7c21:      fs jl  0x7c33
   0x7c24:      and    %al,%al
   0x7c26:      or     $0x1,%ax
   0x7c2a:      mov    %eax,%cr0
   0x7c2d:      ljmp   $0xb866,$0x87c32
   0x7c34:      adc    %al,(%eax)
   0x7c36:      mov    %eax,%ds
   0x7c38:      mov    %eax,%es
   0x7c3a:      mov    %eax,%fs
   0x7c3c:      mov    %eax,%gs
   0x7c3e:      mov    %eax,%ss
   0x7c40:      mov    $0x7c00,%esp
```

Эта инструкция gdb дизассемблирует инструкции, хранящиеся в 0x7c00 и последующих 30 байтах памяти.

```
12 .globl start
13 start:
14   .code16                     # Assemble for 16-bit mode
15   cli                         # Disable interrupts
16   cld                         # String operations increment
17
18   # Set up the important data segment registers (DS, ES, SS).
19   xorw    %ax,%ax             # Segment number zero
20   movw    %ax,%ds             # -> Data Segment
21   movw    %ax,%es             # -> Extra Segment
22   movw    %ax,%ss             # -> Stack Segment
23
24   # Enable A20:
25   #   For backwards compatibility with the earliest PCs, physical
26   #   address line 20 is tied low, so that addresses higher than
27   #   1MB wrap around to zero by default.  This code undoes this.
28 seta20.1:
29   inb     $0x64,%al               # Wait for not busy
30   testb   $0x2,%al
31   jnz     seta20.1
32
33   movb    $0xd1,%al               # 0xd1 -> port 0x64
34   outb    %al,$0x64
```

boot / boot.S

```
 10 .globl start
 11 start:
 12   .code16                     # Assemble for 16-bit mode
 13   cli                         # Disable interrupts
 14     7c00:       fa                      cli
 15   cld                         # String operations increment
 16     7c01:       fc                      cld
 17
 18   # Set up the important data segment registers (DS, ES, SS).
 19   xorw    %ax,%ax             # Segment number zero
 20     7c02:       31 c0                   xor    %eax,%eax
 21   movw    %ax,%ds             # -> Data Segment
 22     7c04:       8e d8                   mov    %eax,%ds
 23   movw    %ax,%es             # -> Extra Segment
 24     7c06:       8e c0                   mov    %eax,%es
 25   movw    %ax,%ss             # -> Stack Segment
 26     7c08:       8e d0                   mov    %eax,%ss
 27
 28 00007c0a <seta20.1>:
 29   # Enable A20:
```

obj/boot/boot.asm

Видно, что между тремя инструкциями нет никакой разницы, но в исходном коде мы указываем множество идентификаторов, таких как “set20.1”, “.start”. После компиляции эти идентификаторы преобразуются в физические адреса, которые будут использоваться в машинном коде. Например, set20.1 преобразуется в 0x7c0a и будет представлен в файле obj/boot/boot.asm. Однако при

выполнении программы на компьютере никаких меток не будет видно, и будут использоваться только физические адреса.

- В какой момент процессор начинает выполнять 32-разрядный код? Что именно вызывает переключение с 16-разрядного режима на 32-разрядный?

Переход из 16-битного (реального) режима в 32-битный (защищенный) режим имеет два предварительных условия: необходимо загрузить глобальную таблицу дескрипторов, чтобы мы могли переключиться на сегмент кода, поддерживающий 32 бита, и нам нужно включить защищенный режим в CR0.

Код выглядит следующим образом :

```
44   # Switch from real to protected mode, using a bootstrap GDT
45   # and segment translation that makes virtual addresses
46   # identical to their physical addresses, so that the
47   # effective memory map does not change during the switch.
48   lgdt    gdtdesc
49   movl    %cr0, %eax
50   orl     $CR0_PE_ON, %eax
51   movl    %eax, %cr0
52
53   # Jump to next instruction, but in 32-bit code segment.
54   # Switches processor into 32-bit mode.
55   ljmp    $PROT_MODE_CSEG, $protcseg
```

- Какая *последняя* выполняемая инструкция загрузчика?

В main.c это

```
60          ((void (*)(void)) (ELFHDR->e_entry))();
```

в obj/boot/boot.asm это

```
306         ((void (*)(void)) (ELFHDR->e_entry))();
307     7d71:       ff 15 18 00 01 00       call   *0x10018
```

После выполнения этой инструкции загрузчик передает управление ядру.

- И какая *первая* инструкция ядра, которое он только что загрузил?

В entry.S это

```
44          movw    $0x1234,0x472                   # warm boot
```

В obj/kern/kernel.asm это

```
  13          movw    $0x1234,0x472                   # warm boot
  14 f0100000:        02 b0 ad 1b 00 00       add    0x1bad(%eax),%dh
```

- *Где* находится первая инструкция ядра?

Поскольку последней командой, выполненной загрузчиком, является вызов *0x10018, первая команда ядра должна быть *0x10018. Проверим *0x10018 с помощью gdb:

```
(gdb) x/1x 0x10018
0x10018:        0x0010000c
(gdb)
```

- Как загрузчик решает, сколько секторов он должен прочитать, чтобы извлечь все ядро с диска? Где он находит эту информацию?

Это известно из заголовка ELF, местоположение первого сегмента известно с помощью e_phoff, а e_phnum может знать, сколько сегментов необходимо загрузить.

Упражнение 5. Еще раз проследите за первыми несколькими инструкциями загрузчика и определите первую инструкцию, которая "сломалась" или иным образом повела бы себя неправильно, если бы вы неправильно указали адрес ссылки загрузчика. Затем измените адрес ссылки в `boot/Makefrag` на что-то неправильное, запустите **`make clean`**, перекомпилируйте лабораторную работу с помощью **`make`** и снова выполните трассировку в загрузчике, чтобы посмотреть, что произойдет. Не забудьте изменить адрес ссылки обратно и **`make clean`** еще раз после этого!

мы изменили адрес ссылки загрузчика в boot/Makefrag с 0x7c00 на 0x7e00 и перекомпилировали лабораторную работу. Вот файл boot/boot.asm

```
26 $(OBJDIR)/boot/boot: $(BOOT_OBJS)
27         @echo + ld boot/boot
28         $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out $^
29         $(V)$(OBJDUMP) -S $@.out >$@.asm
30         $(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
31         $(V)perl boot/sign.pl $(OBJDIR)/boot/boot
```

Вот файл boot/boot.asm до изменений

```
10 .globl start
11 start:
12   .code16                     # Assemble for 16-bit mode
13   cli                         # Disable interrupts
14     7c00:       fa                        cli
15   cld                         # String operations increment
16     7c01:       fc                        cld
17
18   # Set up the important data segment registers (DS, ES, SS).
19   xorw    %ax,%ax             # Segment number zero
20     7c02:       31 c0                     xor    %eax,%eax
```

И после:

```
10 .globl start
11 start:
12   .code16                     # Assemble for 16-bit mode
13   cli                         # Disable interrupts
14     7e00:       fa                        cli
15   cld                         # String operations increment
16     7e01:       fc                        cld
17
18   # Set up the important data segment registers (DS, ES, SS).
19   xorw    %ax,%ax             # Segment number zero
20     7e02:       31 c0                     xor    %eax,%eax
```

ljmp $PROT_MODE_CSEG, $protcseg - это первая инструкция, которая ломается:

```
(gdb) b *0x7c2d
Breakpoint 1 at 0x7c2d
(gdb) c
Continuing.
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x87e32

Breakpoint 1, 0x00007c2d in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b0
0x0000e062 in ?? ()
(gdb) si
[f000:d0ae]    0xfd0ae: cli
0x0000d0ae in ?? ()
(gdb) si
[f000:d0af]    0xfd0af: cld
0x0000d0af in ?? ()
(gdb)
```

Правильные инструкции:

```
(gdb) b *0x7c2d
Breakpoint 1 at 0x7c2d
(gdb) c
Continuing.
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x87c32

Breakpoint 1, 0x00007c2d in ?? ()
(gdb) si
The target architecture is set to "i386".
=> 0x7c32:      mov    $0x10,%ax
0x00007c32 in ?? ()
(gdb) si
=> 0x7c36:      mov    %eax,%ds
0x00007c36 in ?? ()
(gdb) si
=> 0x7c38:      mov    %eax,%es
0x00007c38 in ?? ()
(gdb) si
=> 0x7c3a:      mov    %eax,%fs
0x00007c3a in ?? ()
(gdb)
```

Упражнение 6. Перезагрузите компьютер (выйдите из QEMU / GDB и запустите их снова). Проверьте 8 слов памяти с номером 0x00100000 в момент входа BIOS в загрузчик, а затем еще раз в момент входа загрузчика в ядро. Почему они отличаются? Что находится во второй точке останова? (На самом деле вам не нужно использовать QEMU для ответа на этот вопрос. Просто подумайте.)

Ответ должен быть очевиден. Когда BIOS входит в загрузчик, все 8 слов после 0x100000 в памяти равны нулю, потому что программа ядра в это время не была загружена в память. Загрузка ядра выполняется с помощью функции bootmain.

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x /8wx 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 2, 0x0010000c in ?? ()
(gdb) x /8wx 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x2000b812      0x220f0011      0xc0200fd8
(gdb)
```

Упражнение 7. Используйте QEMU и GDB для трассировки в ядро JOS и остановитесь на `movl %eax, %cr0`. Проверьте память на 0x00100000 и на 0xf0100000. Теперь выполните один шаг по этой инструкции, используя `stepi` команду GDB. Снова проверьте память на 0x00100000 и на 0xf0100000. Убедитесь, что вы понимаете, что только что произошло.

Какая первая инструкция *после* установления нового сопоставления, которая не смогла бы работать должным образом, если бы сопоставление не было установлено? Прокомментируйте `movl %eax, %cr0` в `kern/entry.S`, выполните трассировку в нем и посмотрите, были ли вы правы.

```
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) x/4xb 0x00100000
0x100000:       0x02    0xb0    0xad    0x1b
(gdb) x/4xb 0xf0100000
0xf0100000 <_start-268435468>:  Cannot access memory at address 0xf0100000
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/4xb 0x00100000
0x100000:       0x02    0xb0    0xad    0x1b
(gdb) x/4xb 0xf0100000
0xf0100000 <_start-268435468>:  0x02    0xb0    0xad    0x1b
(gdb)
```

Ошибка "Cannot access memory at address" означает, что gdb не может получить доступ к запрошенному адресу памяти.

Затем вводим команду si, а затем проверяем два местоположения: мы обнаруживаем, что адрес уже доступен и значения, сохраненные в этих двух местах, совпадают. Можно увидеть, что содержимое, в 0xf0100000, было сопоставлено с 0x00100000.

Перед включением подкачки x/8x *0x00100000 показывает байты с некоторыми значениями в них. После того, как мы включим подкачку x/8x *0xf0100000 указывает на ту же память, что и 0x00100000, потому что наш адрес теперь преобразуется через модуль управления памятью.

Теперь закомментируем movl %eax, %cr0 в kern/entry.S

```
(gdb) si
=> 0x100025:    mov    $0xf010002c,%eax
0x00100025 in ?? ()
(gdb) si
=> 0x10002a:    jmp    *%eax
0x0010002a in ?? ()
(gdb) si
=> 0xf010002c <relocated>:      Error while running hook_stop:
Cannot access memory at address 0xf010002c
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame pointer
```

Среди них, для инструкции jmp в 0x10002a местоположение, к которому нужно перейти, равно 0xf010002C. Поскольку управление подкачкой отсутствует, преобразование виртуального адреса в физический в это время выполняться не будет. jmp *%eax завершится ошибкой, поскольку 0xf010002c находится за пределами оперативной памяти.

Упражнение 8. Мы опустили небольшой фрагмент кода - код, необходимый для печати восьмеричных чисел с использованием шаблонов вида "%o". Найдите и заполните этот фрагмент кода.

Заменяем код в lib/printfmt.c

```
206                 // (unsigned) octal
207                 case 'o':
208                         // Replace this with your code.
209                         putch('X', putdat);
210                         putch('X', putdat);
211                         putch('X', putdat);
212                         break;
```

На

```
207                 case 'o':
208                         num = getuint(&ap, lflag);
209                         base = 8;
210                         goto number;
```

```
running JOS: (1.7s)
  printf: OK
```

Упражнение 9. Определите, где ядро инициализирует свой стек и где именно в памяти расположен его стек. Как ядро резервирует пространство для своего стека? И на какой "конец" этой зарезервированной области инициализируется указатель стека, указывающий на?

В entry.S ядро инициализирует свой стек

```
71          # Clear the frame pointer register (EBP)
72          # so that once we get into debugging C code,
73          # stack backtraces will be terminated properly.
74          movl    $0x0,%ebp                       # nuke frame pointer
75
76          # Set the stack pointer
77          movl    $(bootstacktop),%esp
```

В obj/kern/kernel.asm видно, что вершина стека находится в 0xf010f000, а размер стека равен KSTKSIZE

```
50          # Clear the frame pointer register (EBP)
51          # so that once we get into debugging C code,
52          # stack backtraces will be terminated properly.
53          movl    $0x0,%ebp                        # nuke frame
   pointer
54 f010002f:       bd 00 00 00 00          mov    $0x0,%ebp
55
56          # Set the stack pointer
57          movl    $(bootstacktop),%esp
58 f0100034:       bc 00 f0 10 f0          mov    $0xf010f000,%esp
```

```
95 // Kernel stack.
96 #define KSTACKTOP       KERNBASE
97 #define KSTKSIZE        (8*PGSIZE)              // size of a kernel
   stack
98 #define KSTKGAP         (8*PGSIZE)              // size of a kernel
   stack guard
```

мы также видим, что из memlayout.h KSTKSIZE = 8*PGSIZE = 8 * 4096 = 32768 = 0x8000 таким образом, в стеке выделяется диапазон адресов от 0xf0107000 до 0xf010f000, и, таким образом, $esp изначально будет указывать на 0xf010f000 (что является фактическим значением KSTKSIZE).

Поскольку стек растет вниз, указатель стека естественным образом указывает на самый высокий адрес - 0xf010f000

Упражнение 10. Чтобы ознакомиться с соглашениями о вызовах C на x86, найдите адрес `test_backtrace` функции в `obj/kernel/kernel.asm`, установите там точку останова и изучите, что происходит при каждом ее вызове после запуска

ядра. Сколько 32-разрядных слов каждый уровень рекурсивной вложенности `test_backtrace` помещает в стек, и что это за слова?

В obj/kern/kernel.asm находим test_backtrace, и видим, что его начальный адрес равен 0xf0100040:

```
70 f0100040 <test_backtrace>:
71 #include <kern/console.h>
72
73 // Test the stack backtrace function (lab 1 only)
74 void
75 test_backtrace(int x)
76 {
77 f0100040:       55                          push   %ebp
```

```
(gdb) b *0xf0100040
Breakpoint 1 at 0xf0100040: file kern/init.c, line 13.
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=5) at kern/init.c:13
13      {
(gdb) p $sp
$1 = (void *) 0xf010efdc
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=4) at kern/init.c:13
13      {
(gdb) p $sp
$2 = (void *) 0xf010efbc
```

Разница между адресами (например, 0xf010efdc - 0xf010efbc) составляет 32 байта. Это означает, что каждый вызов функции занимает 32 байта в стеке. Каждое слово на 32-разрядной машине занимает 4 байта, поэтому у нас всего 8 слов на функцию.

```
(gdb) x/8xw 0xf010efbc
0xf010efbc:     0xf0100076      0x00000004      0x00000005      0x00000000
0xf010efcc:     0xf010004a      0xf0110308      0x00010094      0xf010eff8
(gdb)
```

содержимое стека во время test_backtrace(5)

0xf010ffe0 <arg to test_backtrace (5)>
0xf010ffdc <ret address into i386_init (f0100132)>
0xf010ffd8 <previous ebp> <- test_backtrace(5)'s %ebp points here
0xf010ffd4 <previous ebx>
0xf010ffd0
0xf010ffcc
0xf010ffc8
0xf010ffc4 <value of x (5)>
0xf010ffc0 <value of x-1 (4)> <- test_backtrace(5)'s %esp points here
0xf010ffbc <ret address into test_backtrace(5) (f0100069)>

Упражнение 11. Реализуйте функцию обратного отслеживания, как указано выше. Используйте тот же формат, что и в примере, поскольку в противном случае скрипт оценки будет сбит с толку. Когда вы решите, что у вас все работает правильно, запустите `make grade`, чтобы проверить, соответствует ли его результат ожиданиям нашего скрипта оценки, и исправьте это, если это не так.

```
57 int
58 mon_backtrace(int argc, char **argv, struct Trapframe *tf)
59 {
60         unsigned int *ebp = ((unsigned int*)read_ebp());
61         cprintf("Stack backtrace:\n");
62         while(ebp) {
63                 cprintf("ebp %08x ", ebp);
64                 cprintf("eip %08x args", ebp[1]);
65                 for(int i = 2; i <= 6; i++)
66                         cprintf(" %08x", ebp[i]);
67                 cprintf("\n");
68                 ebp = (unsigned int*)(*ebp);
69         }
70         return 0;
71 }
```

```
running JOS: (1.2s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: FAIL
    AssertionError: got:
```

Упражнение 12. Измените функцию обратного отслеживания стека, чтобы для каждого `eip` отображались имя функции, имя исходного файла и номер строки, соответствующие этому `eip`.

В `debuginfo_eip` где делать `__STAB_*` откуда взялись?

Завершите реализацию `debuginfo_eip`, вставив вызов в `stab_binsearch`, чтобы найти номер строки для адреса.

Добавьте команду `обратного отслеживания` в монитор ядра и расширьте свою реализацию, `mon_backtrace` чтобы вызывать `debuginfo_eip` и печатать строку для каждого стекового фрейма формы:

```
K> backtrace
Stack backtrace:
 ebp f010ff78 eip f01008ae args 00000001 f010ff8c 00000000 f0110580 00000000
 kern/monitor.c:143: monitor+106
 ebp f010ffd8 eip f0100193 args 00000000 00001aac 00000660 00000000 00000000
 kern/init.c:49: i386_init+59
 ebp f010fff8 eip f010003d args 00000000 00000000 0000ffff 10cf9a00 0000ffff
 kern/entry.S:70: <unknown>+0
K>
```

___STAB_BEGIN__ и __STAB_END__ — это символы, определенные в скрипте компоновщика kernel.ld, если мы посмотрим на файл компоновщика:

```
25          /* Include debugging information in kernel memory */
26          .stab : {
27                  PROVIDE(__STAB_BEGIN__ = .);
28                  *(.stab);
29                  PROVIDE(__STAB_END__ = .);
30                  BYTE(0)         /* Force the linker to allocate
   space
31                                     for this section */
32          }
```

Указатели begin и end указывают начало и конец раздела stab программы и раздела stabstr соответственно.

Добавим в kdebag.c

```
182          stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
183                  if (lline <= rline)     {
184                          info->eip_line = stabs[lline].n_desc;
185                  }
186                  else return -1;
```

И в monitor.c

```
57 int
58 mon_backtrace(int argc, char **argv, struct Trapframe *tf)
59 {
60          unsigned int *ebp = ((unsigned int*)read_ebp());
61          cprintf("Stack backtrace:\n");
62
63          while(ebp) {
64                  unsigned int eip = ebp[1];
65                  struct Eipdebuginfo info;
66                  debuginfo_eip(eip, &info);
67                  cprintf("ebp %08x ", ebp);
68                  cprintf("eip %08x args", ebp[1]);
69                  for(int i = 2; i <= 6; i++)
70                          cprintf(" %08x", ebp[i]);
71                  cprintf("\n");
72
73                  cprintf("\t%s:%d: %.*s+%d\n",
74                  info.eip_file, info.eip_line,
75                  info.eip_fn_namelen, info.eip_fn_name,
76                  eip - info.eip_fn_addr);
77
78                  ebp = (unsigned int*)(*ebp);
79          }
80          return 0;
81 }
```

```
running JOS: (1.0s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
nikolai@nikolai-VirtualBox:~/6.828/lab$
```