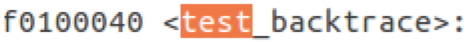
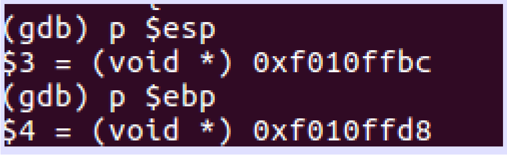
Exercise 10

find the address of the test_backtrace function in obj/kern/kernel.asm, set a breakpoint there, and examine what happens each time it gets called after the kernel starts.



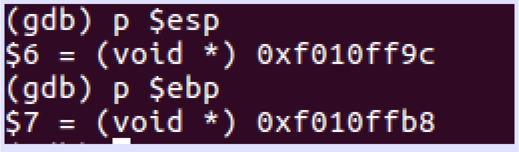
```
f0100040 <test_backtrace>:
```

The 2nd call to Test_backtrace ===> test_backtrace(4)



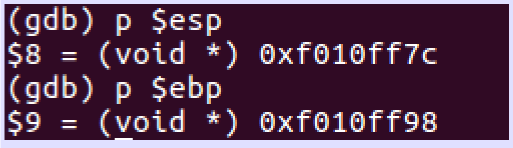
```
(gdb) p $esp
$3 = (void *) 0xf010ffbc
(gdb) p $ebp
$4 = (void *) 0xf010ffd8
```

The 3rd call to Test_backtrace ===> test_backtrace(3)



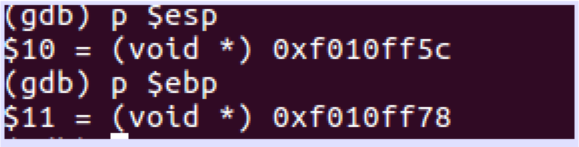
```
(gdb) p $esp
$6 = (void *) 0xf010ff9c
(gdb) p $ebp
$7 = (void *) 0xf010ffb8
```

The 4th call to Test_backtrace ===> test_backtrace(2)



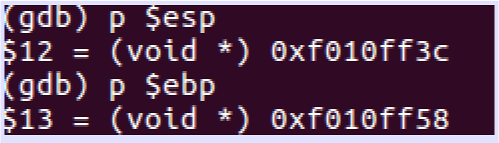
```
(gdb) p $esp
$8 = (void *) 0xf010ff7c
(gdb) p $ebp
$9 = (void *) 0xf010ff98
```

The 5th call to Test_backtrace ===> test_backtrace(1)



```
(gdb) p $esp
$10 = (void *) 0xf010ff5c
(gdb) p $ebp
$11 = (void *) 0xf010ff78
```

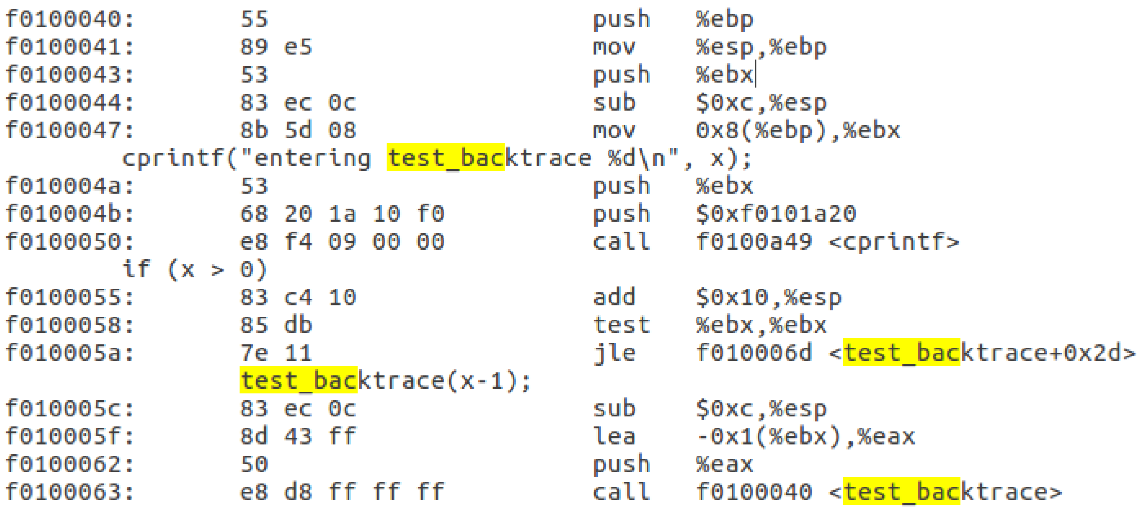
The last call to Test_backtrace ===> test_backtrace(0)



```
(gdb) p $esp
$12 = (void *) 0xf010ff3c
(gdb) p $ebp
$13 = (void *) 0xf010ff58
```

How many 32-bit words does each recursive nesting level of test_backtrace push on the stack, and what are those words?

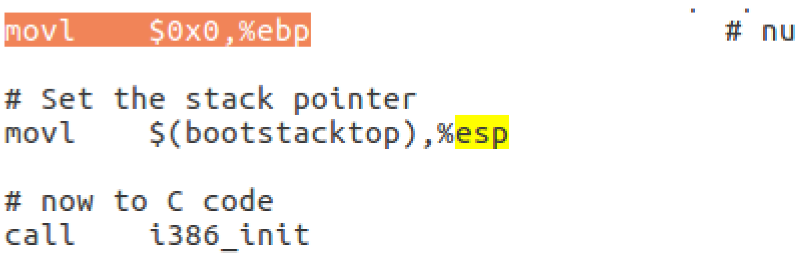
The difference in stack pointer between each successive calls of the function test_backtrace is 0x20 or 32 bytes.
It means each call of the test_backtrace makes the stack decreases by 32.



```
f0100040:       55                      push   %ebp
f0100041:       89 e5                   mov    %esp,%ebp
f0100043:       53                      push   %ebx
f0100044:       83 ec 0c                sub    $0xc,%esp
f0100047:       8b 5d 08                mov    0x8(%ebp),%ebx
        cprintf("entering test_backtrace %d\n", x);
f010004a:       53                      push   %ebx
f010004b:       68 20 1a 10 f0          push   $0xf0101a20
f0100050:       e8 f4 09 00 00          call   f0100a49 <cprintf>
        if (x > 0)
f0100055:       83 c4 10                add    $0x10,%esp
f0100058:       85 db                   test   %ebx,%ebx
f010005a:       7e 11                   jle    f010006d <test_backtrace+0x2d>
                test_backtrace(x-1);
f010005c:       83 ec 0c                sub    $0xc,%esp
f010005f:       8d 43 ff                lea    -0x1(%ebx),%eax
f0100062:       50                      push   %eax
f0100063:       e8 d8 ff ff ff          call   f0100040 <test_backtrace>
```

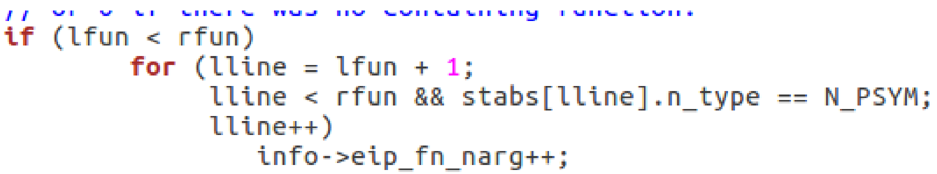
Exercise 11

By studying kern/entry.S you'll find that there is an easy way to tell when to stop.



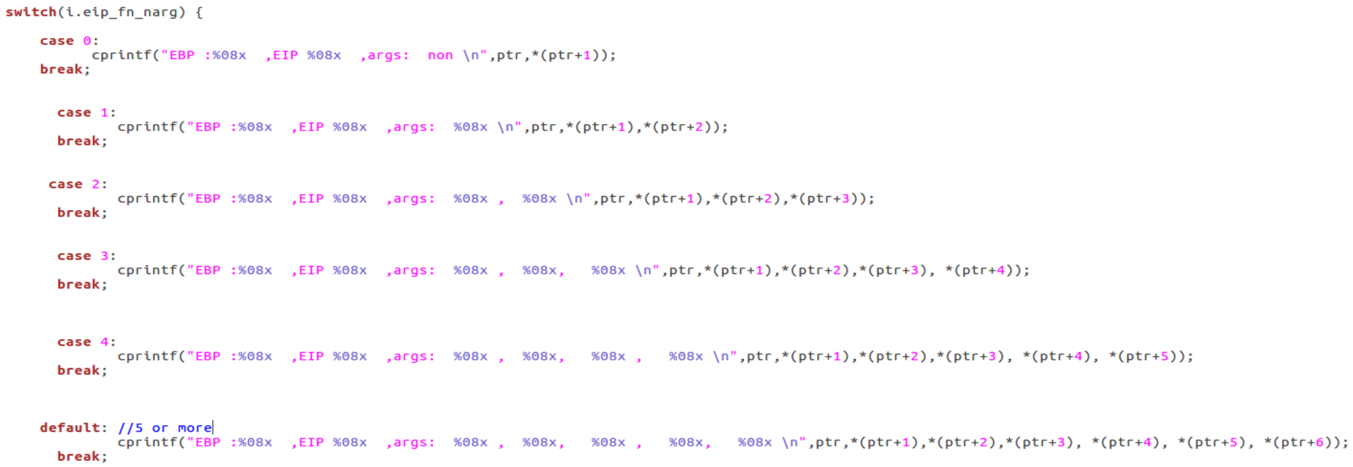
```
movl    $0x0,%ebp                       # nu

# Set the stack pointer
movl    $(bootstacktop),%esp

# now to C code
call    i386_init
```

(Why can't the backtrace code detect how many arguments there actually are? How could this limitation be fixed?)



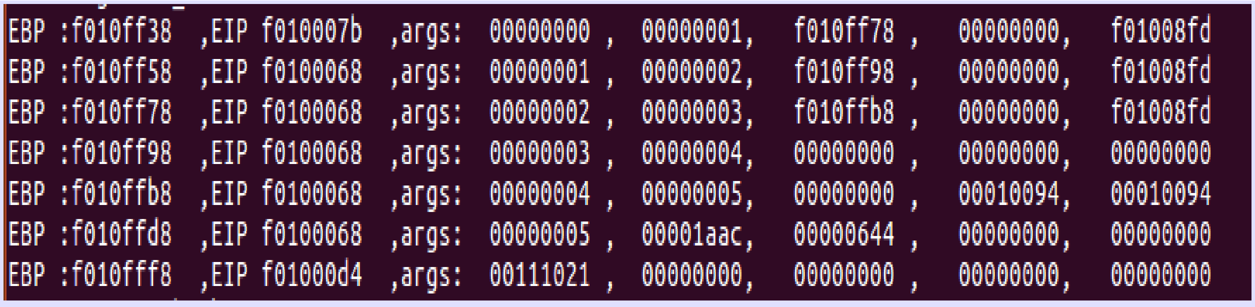
```
// or 0 if there was no containing function.
if (lfun < rfun)
        for (lline = lfun + 1;
             lline < rfun && stabs[lline].n_type == N_PSYM;
             lline++)
                info->eip_fn_narg++;
```

Then we used a switch statement to print the argument of a function.



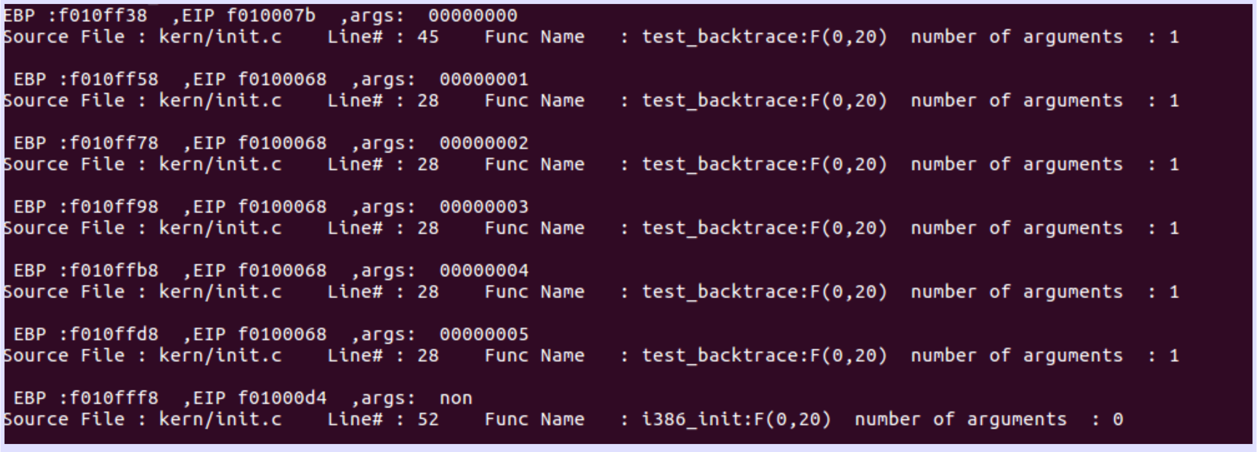
```
switch(i.eip_fn_narg) {

    case 0:
          cprintf("EBP :%08x  ,EIP %08x  ,args:  non \n",ptr,*(ptr+1));
    break;

      case 1:
            cprintf("EBP :%08x  ,EIP %08x  ,args:  %08x \n",ptr,*(ptr+1),*(ptr+2));
      break;

     case 2:
            cprintf("EBP :%08x  ,EIP %08x  ,args:  %08x ,  %08x \n",ptr,*(ptr+1),*(ptr+2),*(ptr+3));
      break;

      case 3:
            cprintf("EBP :%08x  ,EIP %08x  ,args:  %08x ,  %08x,   %08x \n",ptr,*(ptr+1),*(ptr+2),*(ptr+3), *(ptr+4));
      break;


      case 4:
            cprintf("EBP :%08x  ,EIP %08x  ,args:  %08x ,  %08x,   %08x ,   %08x \n",ptr,*(ptr+1),*(ptr+2),*(ptr+3), *(ptr+4), *(ptr+5));
      break;


    default: //5 or more
            cprintf("EBP :%08x  ,EIP %08x  ,args:  %08x ,  %08x,   %08x ,   %08x,   %08x \n",ptr,*(ptr+1),*(ptr+2),*(ptr+3), *(ptr+4), *(ptr+5), *(ptr+6));
      break;
```

Implement the backtrace function as specified above.



```
EBP :f010ff38  ,EIP f010007b  ,args:  00000000 ,  00000001,   f010ff78 ,   00000000,   f01008fd
EBP :f010ff58  ,EIP f0100068  ,args:  00000001 ,  00000002,   f010ff98 ,   00000000,   f01008fd
EBP :f010ff78  ,EIP f0100068  ,args:  00000002 ,  00000003,   f010ffb8 ,   00000000,   f01008fd
EBP :f010ff98  ,EIP f0100068  ,args:  00000003 ,  00000004,   00000000 ,   00000000,   00000000
EBP :f010ffb8  ,EIP f0100068  ,args:  00000004 ,  00000005,   00000000 ,   00010094,   00010094
EBP :f010ffd8  ,EIP f0100068  ,args:  00000005 ,  00001aac,   00000644 ,   00000000,   00000000
EBP :f010fff8  ,EIP f01000d4  ,args:  00111021 ,  00000000,   00000000 ,   00000000,   00000000
```

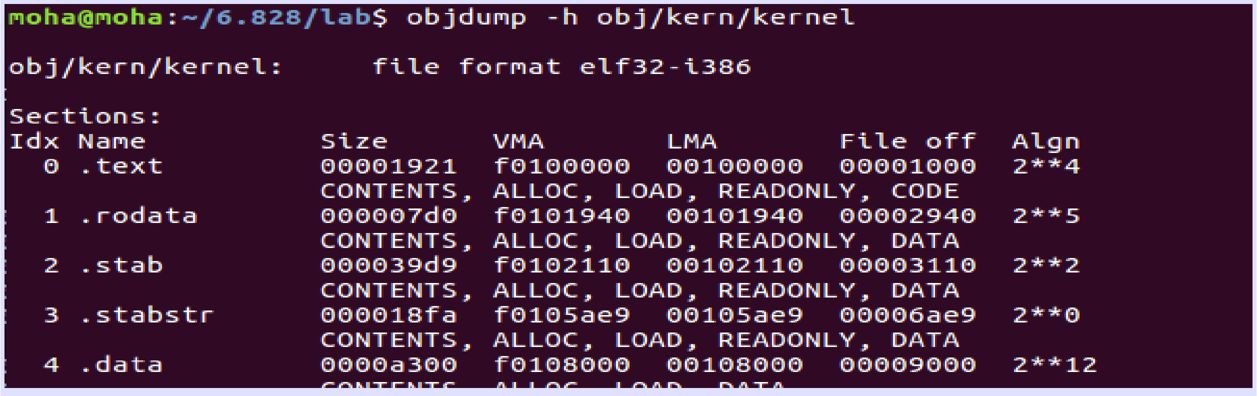
Exercise 12. Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.



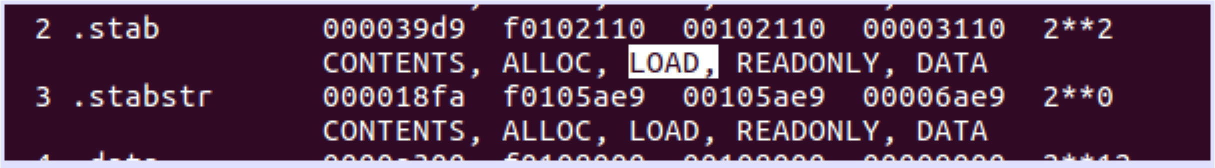
```
EBP :f010ff38  ,EIP f010007b  ,args:  00000000
Source File : kern/init.c    Line# : 45    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010ff58  ,EIP f0100068  ,args:  00000001
Source File : kern/init.c    Line# : 28    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010ff78  ,EIP f0100068  ,args:  00000002
Source File : kern/init.c    Line# : 28    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010ff98  ,EIP f0100068  ,args:  00000003
Source File : kern/init.c    Line# : 28    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010ffb8  ,EIP f0100068  ,args:  00000004
Source File : kern/init.c    Line# : 28    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010ffd8  ,EIP f0100068  ,args:  00000005
Source File : kern/init.c    Line# : 28    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010fff8  ,EIP f01000d4  ,args:  non
Source File : kern/init.c    Line# : 52    Func Name   : i386_init:F(0,20)  number of arguments  : 0
```

In debuginfo_eip, where do __STAB_* come from?

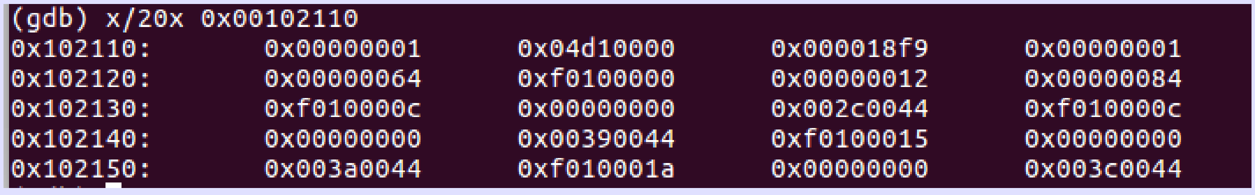
- ***see if the bootloader loads the symbol table in memory as part of loading the kernel binary***



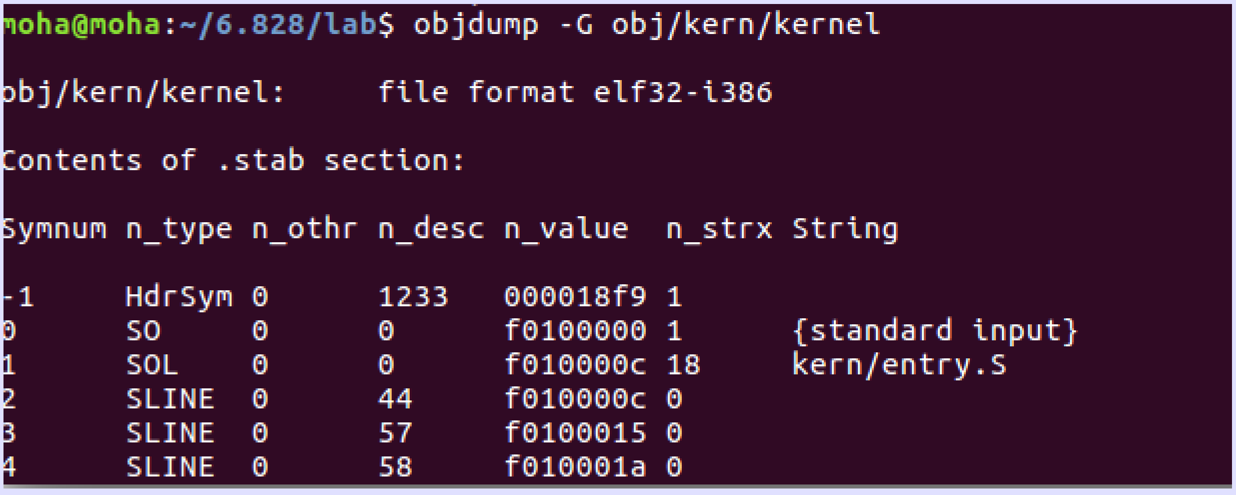
```
moha@moha:~/6.828/lab$ objdump -h obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001921  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000007d0  f0101940  00101940  00002940  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         000039d9  f0102110  00102110  00003110  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      000018fa  f0105ae9  00105ae9  00006ae9  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f0108000  00108000  00009000  2**12
```



```
  2 .stab         000039d9  f0102110  00102110  00003110  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      000018fa  f0105ae9  00105ae9  00006ae9  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```



```
(gdb) x/20x 0x00102110
0x102110:       0x00000001      0x04d10000      0x000018f9      0x00000001
0x102120:       0x00000064      0xf0100000      0x00000012      0x00000084
0x102130:       0xf010000c      0x00000000      0x002c0044      0xf010000c
0x102140:       0x00000000      0x00390044      0xf0100015      0x00000000
0x102150:       0x003a0044      0xf010001a      0x00000000      0x003c0044
```

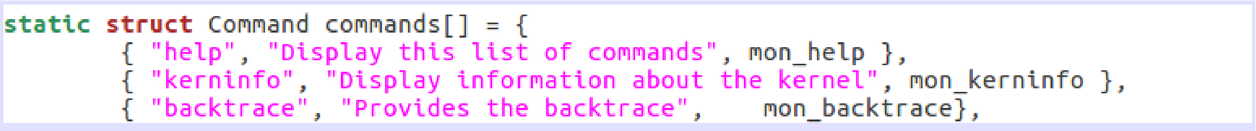


```
moha@moha:~/6.828/lab$ objdump -G obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Contents of .stab section:

Symnum n_type n_othr n_desc n_value  n_strx String

-1     HdrSym 0      1233   000018f9 1
0      SO     0      0      f0100000 1      {standard input}
1      SOL    0      0      f010000c 18     kern/entry.S
2      SLINE  0      44     f010000c 0
3      SLINE  0      57     f0100015 0
4      SLINE  0      58     f010001a 0
```

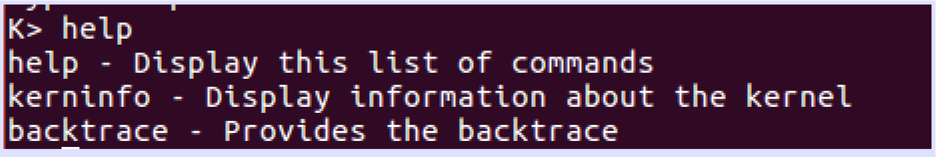
Complete the implementation of debuginfo_eip by inserting the call to stab_binsearch to find the line number for an address.

```
stab_binsearch(stabs, &lline, &rline, N_SLINE    , addr);

info->eip_line = stabs[lline].n_value;
```

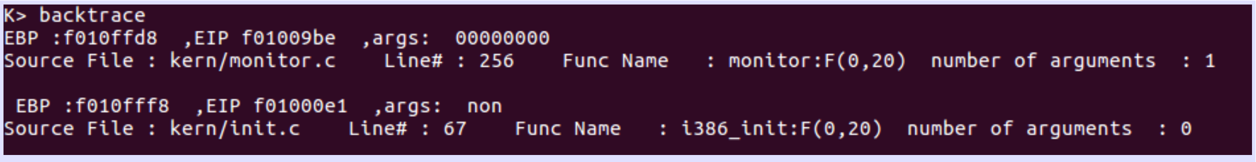
Add a backtrace command to the kernel monitor.



```
static struct Command commands[] = {
        { "help", "Display this list of commands", mon_help },
        { "kerninfo", "Display information about the kernel", mon_kerninfo },
        { "backtrace", "Provides the backtrace",    mon_backtrace},
```



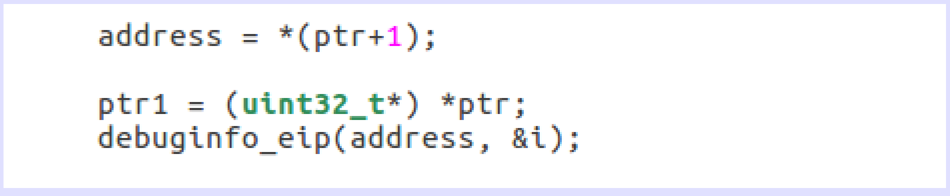
```
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
backtrace - Provides the backtrace
```



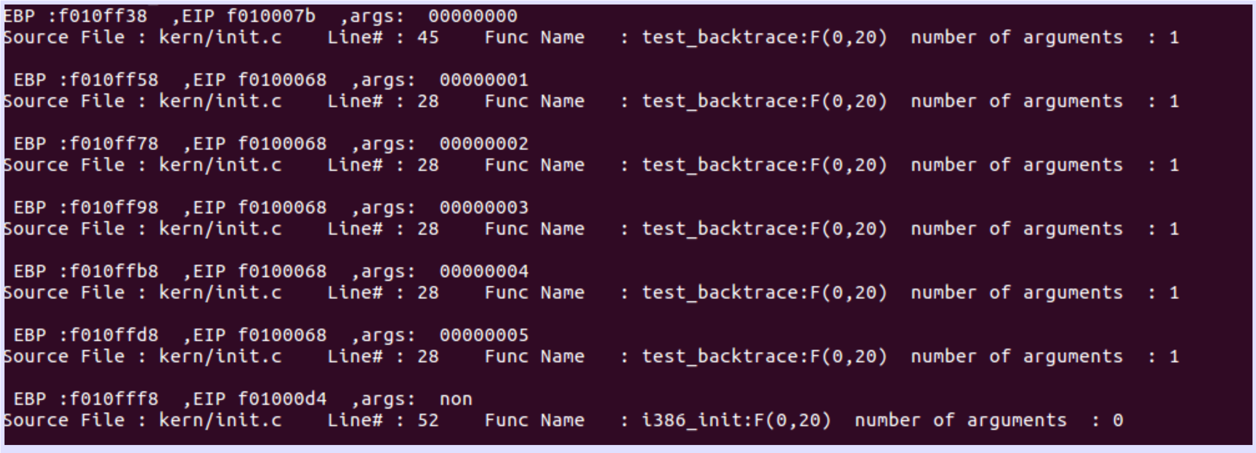
```
K> backtrace
EBP :f010ffd8  ,EIP f01009be  ,args:  00000000
Source File : kern/monitor.c    Line# : 256    Func Name   : monitor:F(0,20)  number of arguments  : 1

 EBP :f010fff8  ,EIP f01000e1  ,args:  non
Source File : kern/init.c    Line# : 67    Func Name   : i386_init:F(0,20)  number of arguments  : 0
```

extend your implementation of mon_backtrace to call debuginfo_eip and print a line for each stack frame of the form:

```
K> backtrace
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
         kern/monitor.c:143: monitor+106
```



```
    address = *(ptr+1);

    ptr1 = (uint32_t*) *ptr;
    debuginfo_eip(address, &i);
```



```
EBP :f010ff38  ,EIP f010007b  ,args:  00000000
Source File : kern/init.c    Line# : 45    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010ff58  ,EIP f0100068  ,args:  00000001
Source File : kern/init.c    Line# : 28    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010ff78  ,EIP f0100068  ,args:  00000002
Source File : kern/init.c    Line# : 28    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010ff98  ,EIP f0100068  ,args:  00000003
Source File : kern/init.c    Line# : 28    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010ffb8  ,EIP f0100068  ,args:  00000004
Source File : kern/init.c    Line# : 28    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010ffd8  ,EIP f0100068  ,args:  00000005
Source File : kern/init.c    Line# : 28    Func Name   : test_backtrace:F(0,20)  number of arguments  : 1

 EBP :f010fff8  ,EIP f01000d4  ,args:  non
Source File : kern/init.c    Line# : 52    Func Name   : i386_init:F(0,20)  number of arguments  : 0
```

The rest of exercise 10

Before test called

esp 0xf010ffe0

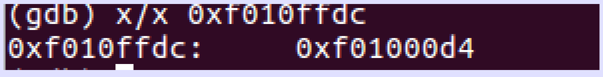
ebp 0xf010fff8

After the call of backtrace , bcz the eip has been pushed onto the stack

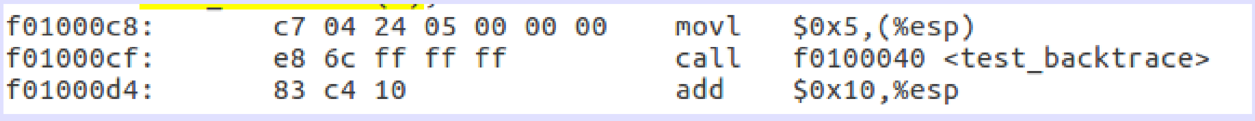
esp 0xf010ffdc

ebp 0xf010fff8

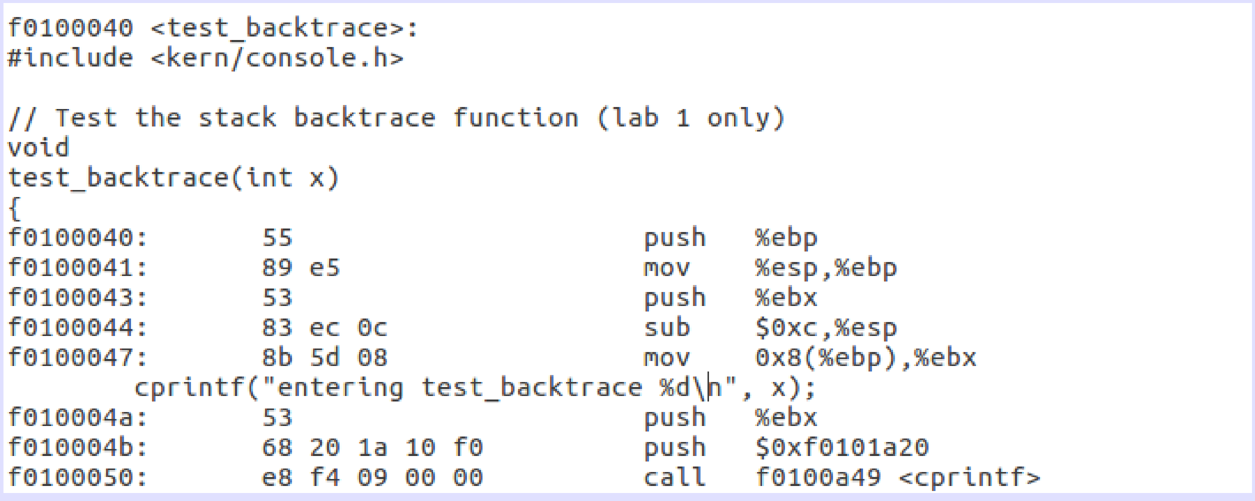
To confirm that, we can read the memory of 0xf010ffdc, which is the address of the instruction after the call



```
(gdb) x/x 0xf010ffdc
0xf010ffdc:     0xf01000d4
```



```
f01000c8:       c7 04 24 05 00 00 00    movl   $0x5,(%esp)
f01000cf:       e8 6c ff ff ff          call   f0100040 <test_backtrace>
f01000d4:       83 c4 10                add    $0x10,%esp
```

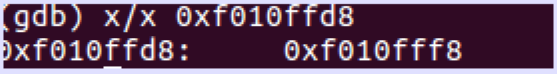


```
f0100040 <test_backtrace>:
#include <kern/console.h>

// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
f0100040:       55                      push   %ebp
f0100041:       89 e5                   mov    %esp,%ebp
f0100043:       53                      push   %ebx
f0100044:       83 ec 0c                sub    $0xc,%esp
f0100047:       8b 5d 08                mov    0x8(%ebp),%ebx
        cprintf("entering test_backtrace %d\n", x);
f010004a:       53                      push   %ebx
f010004b:       68 20 1a 10 f0          push   $0xf0101a20
f0100050:       e8 f4 09 00 00          call   f0100a49 <cprintf>
```

After the push instruction

%esp will be decreased by 4, so it becomes ⇒ 0xf010ffd8

And %ebp will remain the same ⇒ 0xf010fff8, but this value is now pushed on the stack as we can see by checking the memory of 0x...d8



```
(gdb) x/x 0xf010ffd8
0xf010ffd8:     0xf010fff8
```

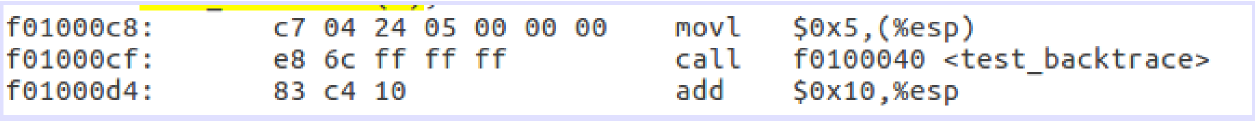
After the mov instruction

After the mov instruction, %esp will remain the same 0xf010ffd8, but now %ebp will have the same value too.
%ebx 65684

After push %ebx
%esp decreases by 4 and becomes 0xf010ffd4,
Because Ebx is a Callee saved register (EBX, ESI & EDI), and since we are going to use it, then we have to save it.

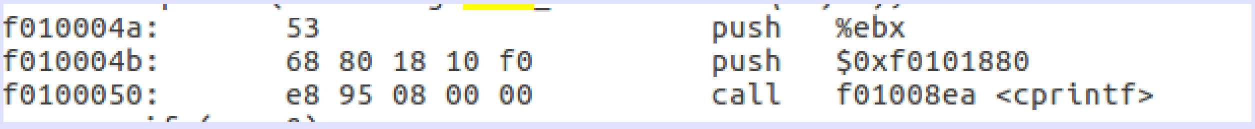
sub $0xc,%esp
Esp becomes 0xf010ffc8, this instructions it will give some storage to the local variables and complicated operations.
Keep in mind that the Test_Backtrace function is written in C, so these instructions are done by the compiler.

mov 0x8(%ebp),%ebx
By convention, the first argument is always stored here, so this will put x which is 5 into ebx
Note that x, was pushed by the function that called Test_backtrace



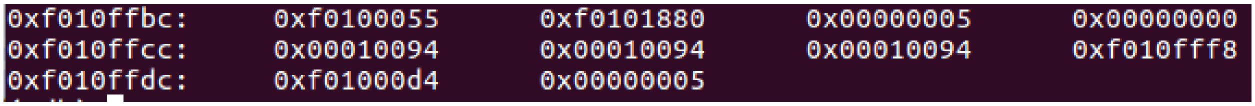
```
f01000c8:       c7 04 24 05 00 00 00    movl   $0x5,(%esp)
f01000cf:       e8 6c ff ff ff          call   f0100040 <test_backtrace>
f01000d4:       83 c4 10                add    $0x10,%esp
```

Now we will start preparing to call the function cprintf(), ,so we need first to save its 2 arguments (x and the string on the stack)



```
f010004a:       53                      push   %ebx
f010004b:       68 80 18 10 f0          push   $0xf0101880
f0100050:       e8 95 08 00 00          call   f01008ea <cprintf>
```

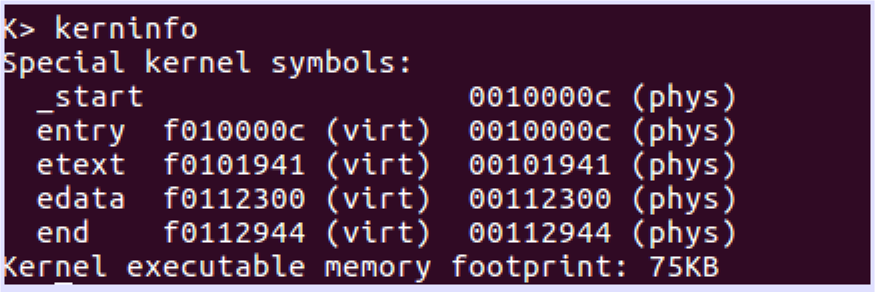
So the esp will decrease by 12, and becomes 0xf010ffc8, %ebp still the same,
But let’s check the contents of the stack:
It shd be the %eip address f0100055, then the second argument 0xf...880 , then the first argument x which is 5, then 4 garbage words bcz of the subtraction.



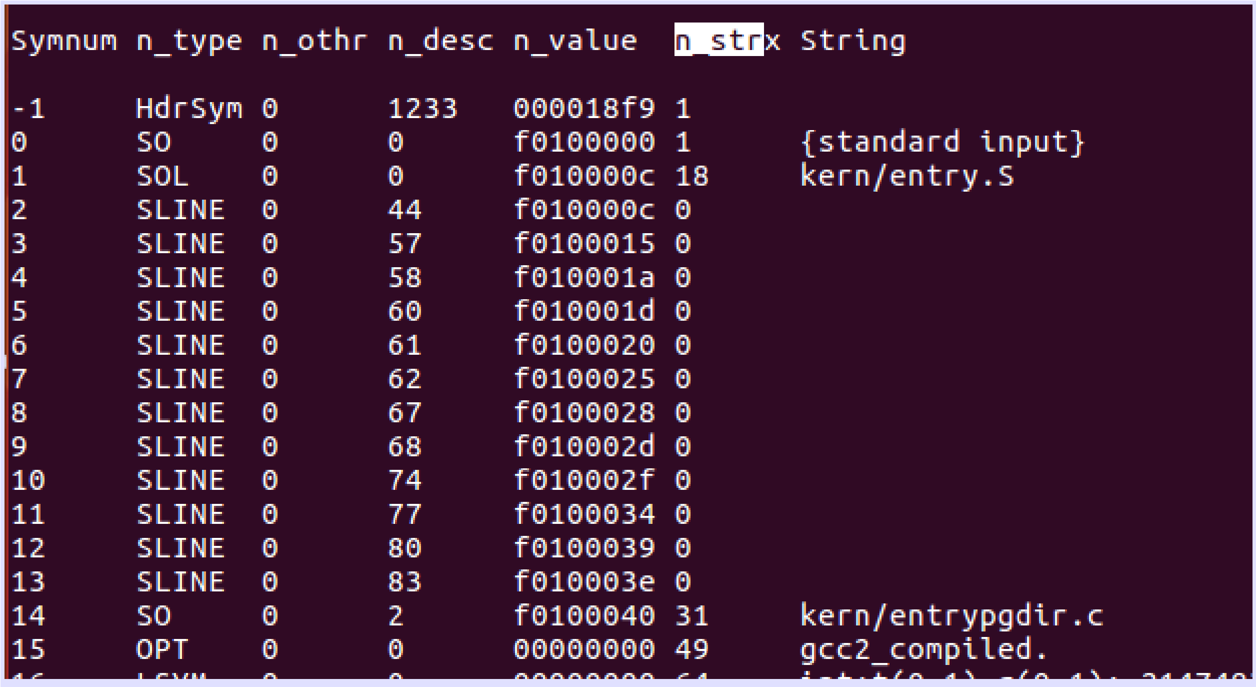
```
0xf010ffbc:     0xf0100055      0xf0101880      0x00000005      0x00000000
0xf010ffcc:     0x00010094      0x00010094      0x00010094      0xf010fff8
0xf010ffdc:     0xf01000d4      0x00000005
```

Now the Cprintf Prologue, which is saving the base pointer of the Test_Backtrace, then assigning a new base pointer to the current function.

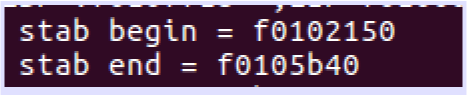
Both %esp and %ebp will be 0xf010ffb4,   and the value saved at this stack address is 0xf010ffd8 which is the base pointer of the Test_Backtrace.
Then subtract 16 from the esp, so it becomes 0xf010ffa4



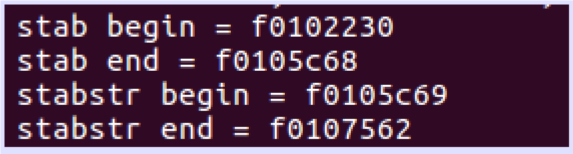
```
K> kerninfo
Special kernel symbols:
  _start                  0010000c (phys)
  entry  f010000c (virt)  0010000c (phys)
  etext  f0101941 (virt)  00101941 (phys)
  edata  f0112300 (virt)  00112300 (phys)
  end    f0112944 (virt)  00112944 (phys)
Kernel executable memory footprint: 75KB
```



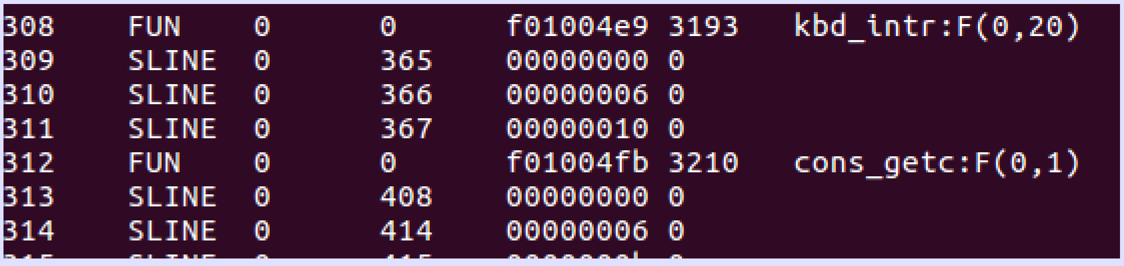
```
Symnum n_type n_othr n_desc n_value   n_strx String

-1     HdrSym 0      1233   000018f9 1
0      SO     0      0      f0100000 1      {standard input}
1      SOL    0      0      f010000c 18     kern/entry.S
2      SLINE  0      44     f010000c 0
3      SLINE  0      57     f0100015 0
4      SLINE  0      58     f010001a 0
5      SLINE  0      60     f010001d 0
6      SLINE  0      61     f0100020 0
7      SLINE  0      62     f0100025 0
8      SLINE  0      67     f0100028 0
9      SLINE  0      68     f010002d 0
10     SLINE  0      74     f010002f 0
11     SLINE  0      77     f0100034 0
12     SLINE  0      80     f0100039 0
13     SLINE  0      83     f010003e 0
14     SO     0      2      f0100040 31     kern/entrypgdir.c
15     OPT    0      0      00000000 49     gcc2_compiled.
```



```
stab begin = f0102150
stab end = f0105b40
```



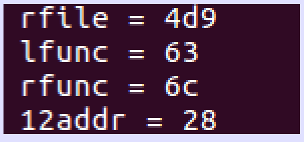
```
stab begin = f0102230
stab end = f0105c68
stabstr begin = f0105c69
stabstr end = f0107562
```



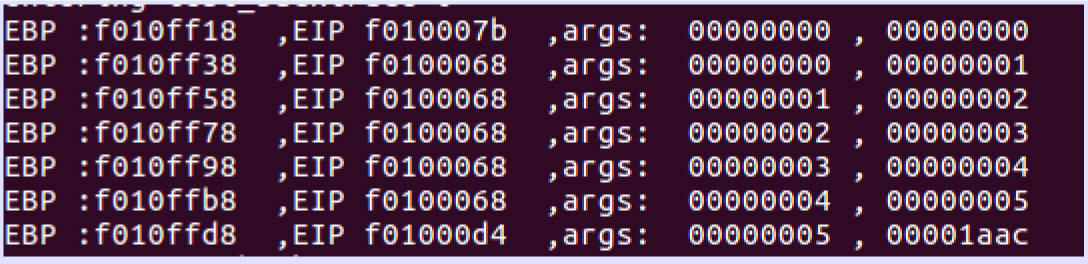
```
308     FUN     0       0       f01004e9 3193   kbd_intr:F(0,20)
309     SLINE   0       365     00000000 0
310     SLINE   0       366     00000006 0
311     SLINE   0       367     00000010 0
312     FUN     0       0       f01004fb 3210   cons_getc:F(0,1)
313     SLINE   0       408     00000000 0
314     SLINE   0       414     00000006 0
```

n_value for a function contains its address

But for an SLINE, it contains its offset i believe



```
 rfile = 4d9
 lfunc = 63
 rfunc = 6c
 12addr = 28
```

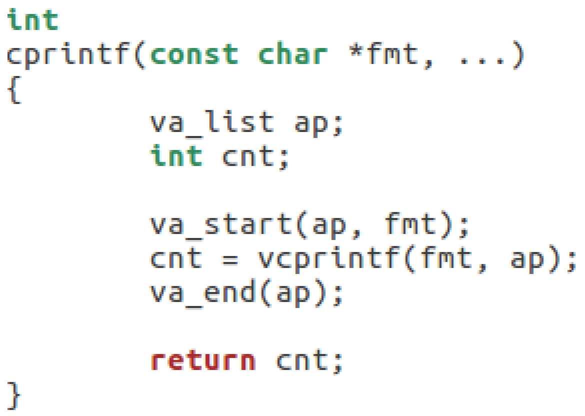


```
EBP :f010ff18  ,EIP f010007b  ,args:  00000000 , 00000000
EBP :f010ff38  ,EIP f0100068  ,args:  00000000 , 00000001
EBP :f010ff58  ,EIP f0100068  ,args:  00000001 , 00000002
EBP :f010ff78  ,EIP f0100068  ,args:  00000002 , 00000003
EBP :f010ff98  ,EIP f0100068  ,args:  00000003 , 00000004
EBP :f010ffb8  ,EIP f0100068  ,args:  00000004 , 00000005
EBP :f010ffd8  ,EIP f01000d4  ,args:  00000005 , 00001aac
```

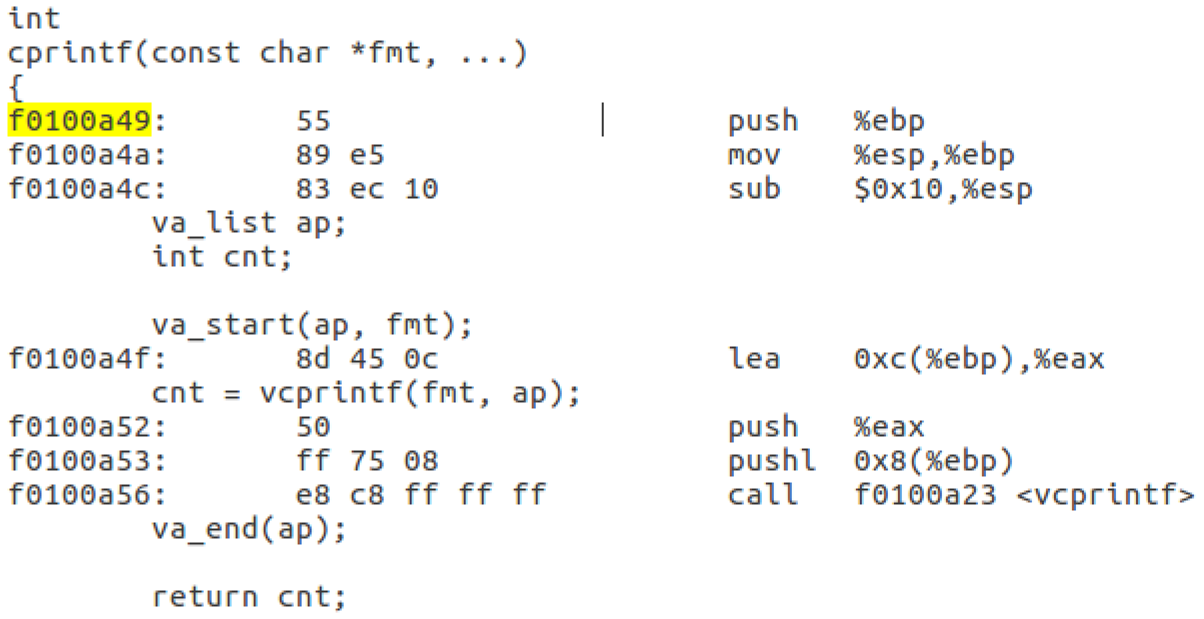
What are these 32?

Test_backtrace ()
Push both ebx and ebp besides subtracting the stack by 12   ⇒ these are 16 bytes

Cprintf()
Pushing the ebx and the string ⇒ 8 bytes

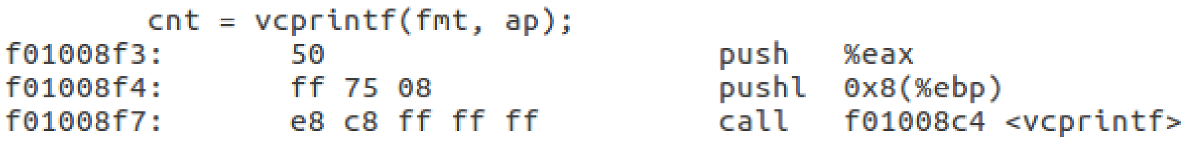


```
int
cprintf(const char *fmt, ...)
{
        va_list ap;
        int cnt;

        va_start(ap, fmt);
        cnt = vcprintf(fmt, ap);
        va_end(ap);

        return cnt;
}
```



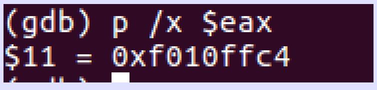
```
int
cprintf(const char *fmt, ...)
{
f0100a49:       55                      push   %ebp
f0100a4a:       89 e5                   mov    %esp,%ebp
f0100a4c:       83 ec 10                sub    $0x10,%esp
        va_list ap;
        int cnt;

        va_start(ap, fmt);
f0100a4f:       8d 45 0c                lea    0xc(%ebp),%eax
        cnt = vcprintf(fmt, ap);
f0100a52:       50                      push   %eax
f0100a53:       ff 75 08                pushl  0x8(%ebp)
f0100a56:       e8 c8 ff ff ff          call   f0100a23 <vcprintf>
        va_end(ap);

        return cnt;
```

The following is the preparing to call vcprintf(), so its 2 arguments are pushed on the stack (

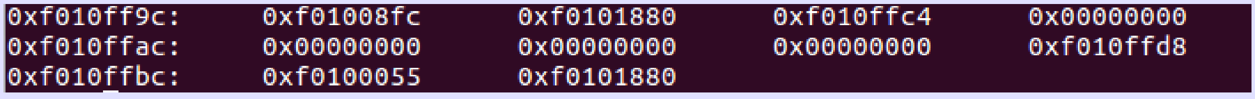


```
        cnt = vcprintf(fmt, ap);
f01008f3:       50                      push   %eax
f01008f4:       ff 75 08                pushl  0x8(%ebp)
f01008f7:       e8 c8 ff ff ff          call   f01008c4 <vcprintf>
```

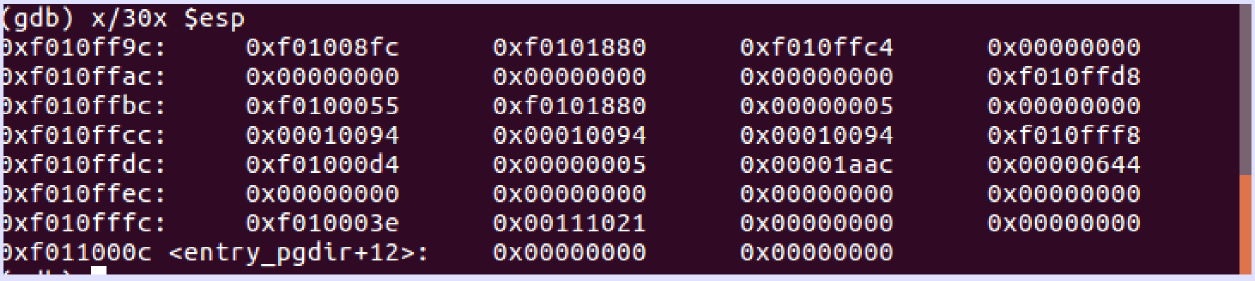
va_list ap; is the first argument so it is at 8+ebp, the 2nd argument fmt is saved by the previos instruction in eax. Then call the function vcprintf
%esp decreased by 12, so will be 0xf010fef8 and %ebp will still be 0xf010ffb4



```
(gdb) p /x $eax
$11 = 0xf010ffc4
```

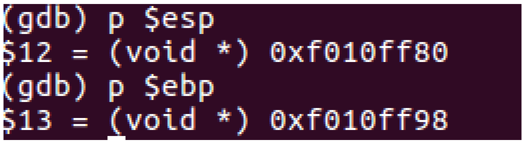


```
0xf010ff9c:     0xf01008fc      0xf0101880      0xf010ffc4      0x00000000
0xf010ffac:     0x00000000      0x00000000      0x00000000      0xf010ffd8
0xf010ffbc:     0xf0100055      0xf0101880
```



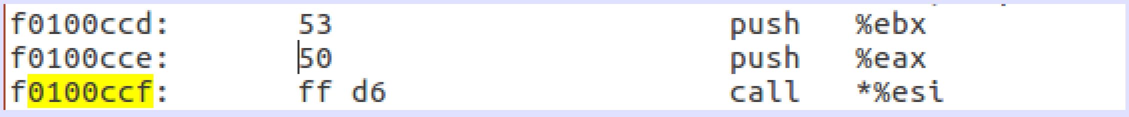
```
(gdb) x/30x $esp
0xf010ff9c:     0xf01008fc      0xf0101880      0xf010ffc4      0x00000000
0xf010ffac:     0x00000000      0x00000000      0x00000000      0xf010ffd8
0xf010ffbc:     0xf0100055      0xf0101880      0x00000005      0x00000000
0xf010ffcc:     0x00010094      0x00010094      0x00010094      0xf010fff8
0xf010ffdc:     0xf01000d4      0x00000005      0x00001aac      0x00000644
0xf010ffec:     0x00000000      0x00000000      0x00000000      0x00000000
0xf010fffc:     0xf010003e      0x00111021      0x00000000      0x00000000
0xf011000c <entry_pgdir+12>:    0x00000000      0x00000000
```

after making the new frame of vcprintf()



```
(gdb) p $esp
$12 = (void *) 0xf010ff80
(gdb) p $ebp
$13 = (void *) 0xf010ff98
```

Then it calls vprintfmt with 4 arguments, so 4 words are pushed on the stack
Then vmprintf stores the 3 callee registers
Then it pushes ebx and eax in there…



```
f0100ccd:       53                      push   %ebx
f0100cce:       50                      push   %eax
f0100ccf:       ff d6                   call   *%esi
```

How many 32-bit words does each recursive nesting level of test_backtrace push on the stack, and **what are those words?**

```
mov %ebp, esp
```

```
pop %ebp
```

`CALL`, `(push eip)`
`Ret instruction => pop %Eip`

```
f01008ea <cprintf>
f0100040 <test_backtrace>:
```