



Name: Anil Pavuluru

Login id:anpavulu

Password:2782551P

```
ciscuohio.edu - PuTTY
anpavulu@spirit's password:
Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 4.15.0-112-generic x86_64)

22 packages can be updated.
0 updates are security updates.

1 updates could not be installed automatically. For more details,
see /var/log/unattended-upgrades/unattended-upgrades.log
*** System restart required ***

Last login: Sun Sep  6 19:28:27 2020 from 137.148.204.40
spirit:~% cd hw1
spirit:~/hw1% ls
a.out  lbcount  lbcount.c  map.c
spirit:~/hw1% gcc lbcount.c -g -o lbcount
spirit:~/hw1% ./a.out
File name missing!
spirit:~/hw1% gdb lbcount
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0-20180409-git
Copyright (C) 2018 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:
<http://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"...
Reading symbols from lbcount...done.
(gdb) list
1      #include <stdio.h>
2      #include <stdlib.h>
3
4      int main(int argc, char *argv[]) {
5
6          FILE* input;
7          if (argc < 2) {
8              printf("File name missing!\n");
9              exit(1);
10         }
(gdb) list
11
12         input = fopen(argv[1], "r");
13         int ch;
14
15         int lineCount = 0;
16         int byteCount = 0;
17
18         while((ch = getc(input)) != EOF) {
19             byteCount += 1;
20             if (ch == '\n')
(gdb) break 20
Breakpoint 1 at 0x807: file lbcount.c, line 20.
(gdb) r lbcount.c
Starting program: /home/student/anpavulu/hw1/lbcount lbcount.c
Breakpoint 1, main (argc=2, argv=0x7ffffffe9f8) at lbcount.c:20
20         if (ch == '\n')
(gdb) continue
Continuing.
Breakpoint 1, main (argc=2, argv=0x7ffffffe9f8) at lbcount.c:20
20         if (ch == '\n')
(gdb) continue
Continuing.
Breakpoint 1, main (argc=2, argv=0x7ffffffe9f8) at lbcount.c:20
20         if (ch == '\n')
(gdb) continue
Continuing.
Breakpoint 1, main (argc=2, argv=0x7ffffffe9f8) at lbcount.c:20
20         if (ch == '\n')
(gdb) p argv
$1 = (char **) 0x7ffffffe9f8
(gdb) p argv[0]
$2 = 0x7ffffffec52 "/home/student/anpavulu/hw1/lbcount"
(gdb) p byteCount
$3 = 4
(gdb) p lineCount
$4 = 0
(gdb) p main
$5 = (int (int, char *)) 0x555555547aa <main>
(gdb) info stack
#0  main (argc=2, argv=0x7ffffffe9f8) at lbcount.c:20
(gdb) info frame
Stack level 0, frame at 0x7ffffffe920:
 rip = 0x55555554807 in main (lbcount.c:20); saved rip = 0x7ffff7a05b97
 source language c.
 Arglist at 0x7ffffffe910, args: argc=2, argv=0x7ffffffe9f8
 Locals at 0x7ffffffe910, Previous frame's sp is 0x7ffffffe920
 Saved registers:
 rbp at 0x7ffffffe910, rip at 0x7ffffffe918
(gdb)
```

- 1) What is the value of argv? (hint: print argv)
Ans: \$1 = (char **) 0x7ffffffe9f8
- 2) What is pointed to by argv? ? (hint: print argv[0])
Ans: \$2 = 0x7ffffffec52 "/home/student/anpavulu/hw1/lbcount"
- 3) What is the value of byteCount? lineCount?
Ans: \$3 = 4 byteCount
\$4 = 0 lineCount
- 4) What is the address of the function main?
Ans: \$5 = {int (int, char **) } 0x555555547aa <main>

5) Try info stack. Explain what you see.

Ans: #0 main (argc=2, argv=0x7fffffffe9f8) at lbcount.c:20

Info stack like wise back trace prints stack frame here the frame #zero it means current frame followed by the argv pointer address of array and string index address and break count number.

6) Try info frame. Explain what you see.

Ans: Stack level 0, frame at 0x7fffffffe920:

rip = 0x555555554807 in main (lbcount.c:20); saved rip = 0x7ffff7a05b97

source language c.

Arglist at 0x7fffffffe910, args: argc=2, argv=0x7fffffffe9f8

Locals at 0x7fffffffe910, Previous frame's sp is 0x7fffffffe920

Saved registers:

rbp at 0x7fffffffe910, rip at 0x7fffffffe918

based on my observation of info frame and explaining about what I seen spirit here in the below.

Info frame: nothing but display advanced info about stack frame parameters like frame number ,frame address

stack level 0

- Zero is current executing frame

frame at 0x7fffffffe920

- starting memory address of this stack frame

rip = 0x555555554807 in main (lbcount.c:20); saved rip = 0x7ffff7a05b97

- rip is the register like program counter for next instruction to execute.so at this moment, the next to execute is at 0x555555554807, which is line 20 of testing.cpp.

- **saved rip== 0x7ffff7a05b97**

we know instruction pointer normally located on the microprocessor which increments with 8 for 64bit system and 4 for 4 bytes system so that it points to next step of instruction.

When program calls main function, the address is saved in register here rip is that register pointer always it returns the address, function jump back after completed.

Source language c

- here “c” language is used.

Arglist at 0x7fffffff910, args: argc=2, argv=0x7fffffff9f8

- Starting address of the argument.args and argc are the command line arguments. Argv it is the pointer point to array.

Locals at 0x7fffffff910

- Local variables address.

Previous frame's sp is 0x7fffffff920

- Where previous frame pointer calls the frame while calling at that moment it become first memory address of stack frame.

Saved registers: save the address of the stack

rbp at 0x7fffffff910, rip at 0x7fffffff918

- **RIP** is the register automatically it pushed to the stack when you call.
- **RBP** is the register automatically popped from the stack when you return.
- rbp at 0x7fffffff910 that is the address where the "rbp" register of the caller's stack frame saved.
- rip at 0x7fffffff918 as mentioned before, but here is the address of the stack (which contains the value "0x55555554807")

```
ciscuohio.edu - PuTTY
spirit:~% cd hwl
spirit:~/hwl% objdump -x -d lbcount

lbcount:      file format elf64-x86-64
lbcount
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000000000000a0

Program Header:
  FHDR off 0x0000000000000040 vaddr 0x0000000000000040 paddr 0x0000000000000000
0040 align 2**3
  filesz 0x00000000000001f8 memsz 0x00000000000001f8 flags r--
  INTERP off 0x0000000000000238 vaddr 0x0000000000000238 paddr 0x0000000000000000
0238 align 2**0
  filesz 0x000000000000001c memsz 0x000000000000001c flags r--
  LOAD off 0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000
0000 align 2**21
  filesz 0x0000000000000a58 memsz 0x0000000000000a58 flags r-x
  LOAD off 0x0000000000000d90 vaddr 0x0000000000000d90 paddr 0x0000000000000020
0d90 align 2**21
  filesz 0x0000000000000280 memsz 0x0000000000000280 flags rw-
  DYNAMIC off 0x0000000000000da0 vaddr 0x0000000000000da0 paddr 0x0000000000000020
0da0 align 2**3
  filesz 0x00000000000001f0 memsz 0x00000000000001f0 flags rw-
  NOTE off 0x0000000000000254 vaddr 0x0000000000000254 paddr 0x0000000000000000
0254 align 2**2
  filesz 0x0000000000000044 memsz 0x0000000000000044 flags r--
  EH_FRAME off 0x0000000000000910 vaddr 0x0000000000000910 paddr 0x0000000000000000
0910 align 2**2
  filesz 0x000000000000003c memsz 0x000000000000003c flags r--
  STACK off 0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000
0000 align 2**4
  filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
  RELRO off 0x0000000000000d90 vaddr 0x0000000000000d90 paddr 0x0000000000000020
0d90 align 2**0
  filesz 0x0000000000000270 memsz 0x0000000000000270 flags r--

Dynamic Section:
NEEDED      libc.so.6
INIT        0x0000000000000608
FINI        0x00000000000008d4
INIT_ARRAY  0x00000000000020d90
INIT_ARRAYSZ 0x0000000000000008
FINI_ARRAY  0x00000000000020d98
FINI_ARRAYSZ 0x0000000000000008
GNU_HASH    0x0000000000000298
STRTAB      0x00000000000003d8
SYMTAB      0x00000000000002b8
STRTX      0x00000000000000a4
```

7)What file format is used for this binary? And what architecture is it compiled for?

Ans: lbcount: file format elf64-x86-64

Here elf is used in the file format ELF stands executable and linkable file format

ELF is used as standard file format for object files on Linux.

Now we are using the a.out file format was being used as a standard but lately ELF is used to standard the file format.

64 means number of bits.

x-86 means The regular Program Files folder holds 64-bit.

architecture: i386:x86-64,

```
ciscuocho.edu - PuTTY
00000000000007aa <main>:
7aa: 55                push    %rbp
7ab: 48 89 e5          mov     %rsp,%rbp
7ac: 48 83 ec 30       sub     $0x30,%rsp
7ad: 89 7d dc          mov     %edi,-0x24(%rbp)
7ae: 48 89 75 d0       mov     %rsi,-0x30(%rbp)
7af: 83 7d dc 01       cmpl    $0x1,-0x24(%rbp)
7b0: 7f 16            jg      7d5 <main+0x2b>
7b1: 48 8d 3d 1e 01 00 00 lea     0x1e(%rip),%rdi    # 8e4 <_IO_stdin_
used+0x4>
7b2: e8 65 fe ff ff    callq   630 <puts@plt>
7b3: bf 01 00 00 00    mov     $0x1,%edi
7b4: e8 ab fe ff ff    callq   680 <exit@plt>
7b5: 48 8b 45 d0       mov     -0x30(%rbp),%rax
7b6: 48 83 c0 08       add     $0x8,%rax
7b7: 48 8b 00         mov     (%rax),%rax
7b8: 48 8d 35 10 01 00 00 lea     0x10(%rip),%rsi    # 8f7 <_IO_stdin_
used+0x17>
7b9: 48 89 c7         mov     %rax,%rdi
7ba: e8 81 fe ff ff    callq   670 <fopen@plt>
7bb: 48 89 45 f8       mov     %rax,-0xc(%rbp)
7bc: c7 45 ec 00 00 00 00 movl    $0x0,-0x14(%rbp)
7bd: c7 45 e0 00 00 00 00 movl    $0x0,-0x10(%rbp)
7be: eb 0e            jmp     811 <main+0x67>
7bf: 83 45 f0 01       addl    $0x1,-0x10(%rbp)
7c0: 83 7d f4 0a       cmpl    $0xa,-0xc(%rbp)
7c1: 75 04            jne     811 <main+0x67>
7c2: 83 45 ec 01       addl    $0x1,-0x14(%rbp)
7c3: 48 8b 45 f8       mov     -0xc(%rbp),%rax
7c4: 48 89 c7         mov     %rax,%rdi
7c5: e8 43 fe ff ff    callq   660 <_IO_getc@plt>
7c6: 89 45 f4         mov     %eax,-0xc(%rbp)
7c7: 83 7d f4 ff       cmpl    $0xffffffff,-0xc(%rbp)
7c8: 75 dd            jne     803 <main+0x59>
7c9: 48 8b 45 f8       mov     -0xc(%rbp),%rax
7ca: 48 89 c7         mov     %rax,%rdi
7cb: e8 0e fe ff ff    callq   640 <fclose@plt>
7cc: 8b 55 f0         mov     -0x10(%rbp),%edx
7cd: 8b 45 ec         mov     -0x14(%rbp),%eax
7ce: 89 c6            mov     %eax,%esi
7cf: 48 8d 3d b8 00 00 00 lea     0xb8(%rip),%rdi    # 8f9 <_IO_stdin_u
sed+0x19>
7d0: b8 00 00 00 00    mov     $0x0,%eax
7d1: e8 05 fe ff ff    callq   650 <printf@plt>
7d2: b8 00 00 00 00    mov     $0x0,%eax
7d3: c9              leaveq   %eax,%edi
7d4: c3              retq
7d5: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
7d6: 00 00 00        
```

8)What segment/section contains main (the function) and what is the address of main? (The last few hex digits should be the same as what you saw in gdb)

Ans:0000000000000007aa <main>:it is located under .txt

9)Do you see the stack segment anywhere? What about the heap? Explain

Ans: stack segment and heap are created while running the program we can say dynamically allocated the memory. But, here we don't execute any output we just examine the object file code. Here there is no chance to create heap and stack segment in object file part.

```

ciscuohio.edu - PuTTY
spitit:~/hw1% ./map
_main @ 0x55b3d724764
recur @ 0x55b3d7246fa
main stack: 0x7ffcc4f3a474
Static data: 0x55b3d7449010
Heap: malloc 1: 0x55b3d7afa260
Heap: malloc 2: 0x55b3d7afa2d0
recur call 3: stack8 0x7ffcc4f3a444
recur call 2: stack8 0x7ffcc4f3a414
recur call 1: stack8 0x7ffcc4f3a3e4
recur call 0: stack8 0x7ffcc4f3a3b4
spitit:~/hw1% objdump -d map

map: file format elf64-x86-64

Disassembly of section .init:
0000000000000580 <.init>:
580: 48 83 ec 08      sub    $0x8,%rsp
584: 48 8b 05 5d 0a 20 00 mov    0x200a5d(%rip),%rax    # 200fe8 <__gmon_start__>
58b: 48 85 c0          test   %rax,%rax
58e: 74 02           je     592 <.init+0x12>
590: ff d0          callq  *%rax
592: 48 83 c4 08      add    $0x8,%rsp
596: c3             retq

Disassembly of section .plt:
00000000000005a0 <.plt>:
5a0: ff 25 0a 0a 20 00 pushq  0x200a0a(%rip)    # 200fb0 <_GLOBAL_OFFSET_TABLE+0x0>
5a6: ff 25 0c 0a 20 00 jmpq   0x200a0c(%rip)    # 200fb8 <_GLOBAL_OFFSET_TABLE+0x10>
5ac: 0f 1f 40 00      nopl   0x0(%rax)

00000000000005b0 <_stack_chk_fail@plt>:
5b0: ff 25 0a 0a 20 00 jmpq   0x200a0a(%rip)    # 200fc0 <_stack_chk_fail@GLIBC_2.4>
5b6: 68 00 00 00 00 00 pushq  $0x0
5bb: e9 e0 ff ff ff jmpq   5a0 <.plt>

00000000000005c0 <printf@plt>:
5c0: ff 25 02 0a 20 00 jmpq   0x200a02(%rip)    # 200fc8 <printf@GLIBC_2.2.5>
5c6: 68 01 00 00 00 00 pushq  $0x1
5cb: e9 d0 ff ff ff jmpq   5a0 <.plt>

00000000000005d0 <malloc@plt>:
5d0: ff 25 fa 09 20 00 jmpq   0x2009fa(%rip)    # 200fd0 <malloc@GLIBC_2.2.5>
5d6: 68 02 00 00 00 00 pushq  $0x2
5db: e9 c0 ff ff ff jmpq   5a0 <.plt>

Disassembly of section .plt.got:
614: 5d             pop     %rbp
615: e9 66 ff ff ff jmpq    660 <register_tm_clones>

00000000000006fa <recur>:
6fa: 55             push    %rbp
6fb: 48 89 e5       mov     %rsp,%rbp
6fc: 48 83 ec 20     sub     $0x20,%rsp
702: 89 7d ec       mov     %edi,-0x14(%rbp)
705: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
70c: 00 00
70e: 48 89 45 f8     mov     %rax,-0x8(%rbp)
712: 31 c0          xor     %eax,%eax
714: 8b 45 ec       mov     -0x14(%rbp),%eax
717: 89 45 f4       mov     %eax,-0xc(%rbp)
71a: 48 8d 55 f4     lea     -0xc(%rbp),%rdx
71e: 8b 45 ec       mov     -0x14(%rbp),%eax
721: 89 c6          mov     %eax,%esi
723: 48 8d 3d ba 01 00 00 lea     0x1ba(%rip),%rdi    # 8e4 <_IO_stdin_used+0x4>
72a: b8 00 00 00 00 00 mov     $0x0,%eax
72f: e8 8c fe ff ff callq   5c0 <printf@plt>
734: 83 7d ec 00     cmpl    $0x0,-0x14(%rbp)
738: 7e 0f          jle     749 <recur+0x4f>
73a: 8b 45 ec       mov     -0x14(%rbp),%eax
73d: 83 e8 01       sub     $0x1,%eax
740: 89 c7          mov     %eax,%edi
742: e8 b3 ff ff ff callq   6fa <recur>
747: eb 05          jmp     74e <recur+0x54>
749: b8 00 00 00 00 00 mov     $0x0,%eax
74e: 48 8b 4d f8     mov     -0x8(%rbp),%rcx
752: 64 48 33 0c 25 28 00 xor     %fs:0x28,%rcx
759: 00 00
75b: 74 05          je      762 <recur+0x60>
75d: e8 4e fe ff ff callq   5b0 <_stack_chk_fail@plt>
762: c9            leaveq  %rsp
763: c3            retq

0000000000000764 <main>:
764: 55             push    %rbp
765: 48 89 e5       mov     %rsp,%rbp
768: 48 83 ec 30     sub     $0x30,%rsp
76c: 89 7d dc       mov     %edi,-0x24(%rbp)
76f: 48 89 75 d0     mov     %rsi,-0x30(%rbp)
773: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
77a: 00 00
77c: 48 89 45 f8     mov     %rax,-0x8(%rbp)
77e: 31 c0          xor     %eax,%eax
780: 31 c0          xor     %eax,%eax
782: bf 64 00 00 00 00 mov     $0x64,%edi
787: e8 44 fe ff ff callq   5d0 <malloc@plt>
78c: 48 89 45 e8     mov     %rax,-0x18(%rbp)
790: bf 64 00 00 00 00 mov     $0x64,%edi

```

10) Use objdump with the -D flag on the map executable. Which of the addresses from the output of running ./map are defined in the executable, and which segment/section is each defined in?

Ans: _main @ 0x55f30ee6f764

recur @ 0x55f30ee6f6fa remember these two address and check this address matches by using obj dump also .

this two address are located under disassembly section.text;

0000000000000764 <main>:

00000000000006fa <recur>:

11) Where is the heap? What direction is it growing in?

Ans: based on the examine the code the heap is allocated with uninitialized variables and after creating the memory and it delete and basically it is on the ram side. I think here in the code it is located under the .txtfile like .bss and .relocate the heap is created in that place.

As we see in the direction of heap in malloc is increasing (Upward) whereas recur call it is decreasing (going downward).

12) Are the two malloc()ed memory areas contiguous? (e.g. is there any extra space between their addresses?)

Ans: We allocate memory of malloc100 in the program but when we type the command ./map we can see that malloc1 start address at 60 it occupies spaces from 60 to 6f(16), 70 to 7f(32), 80 to 8f(48), 90 to 9f(64), a0 to af(80), b0 to bf(96), c0, c1, c2, c3(4) total 100bits.

Coming to malloc2 start address d0 here the continued is not there we can 12 bit space .

13) What direction is the stack growing in?

Ans: Downward (e4 to b4 to 84 to 54).

14) How large is the stack frame for each recursive call?

Ans: difference between two frames is 48bits(e4 to b4 or else b4 to 84 or else 84 to 54).