X86 Debug

Computer Systems Section 3.11

GDB is a “Source Level” debugger

- We have learned how to debug at the C level
- But the machine is executing X86 object code!
- How does GDB play the shell game?
  - Makes it seem like we execute C code
  - Actually we are executing X86 Assembler Code
- How do we debug at the X86 level?

Simple C Function

```
int add(int x, int y) {
        int op1=x;
        int op2=y;
        int res=op1+op2;
        return res;
}
```

gcc -O0 -S x86Math.c

x86Math.c

```
int add(int x, int y) {
        int op1=x;
        int op2=y;
        int res=op1+op2;
        return res;
}
```

x86Math.s

```
                .globl  add
                .type   add, @function
add:
.LFB3:
                .cfi_startproc
                pushq   %rbp
                .cfi_def_cfa_offset 16
                .cfi_offset 6, -16
                movq    %rsp, %rbp
                .cfi_def_cfa_register 6
…
                movl    -12(%rbp), %eax
                popq    %rbp
                .cfi_def_cfa 7, 8
                ret
                .cfi_endproc
.LFE3:
                .size   add, .-add
```

gcc –O0 -g -Wall -fverbose-asm -Wa,-adhln=x86Math.s x86Math.c

x86Math.c

```
int add(int x, int y) {
        int op1=x;
        int op2=y;
        int res=op1+op2;
        return res;
}
```

x86Math.s

```
132                                             .globl          add
 134                            add:
                    32:x86Math.c    ****        int add(int x, int y) {
 136                                            .loc 1 32 0
 137                                            .cfi_startproc
 138 00db 55                                    pushq           %rbp            #
 139                                            .cfi_def_cfa_offset 16
 140                                            .cfi_offset 6, -16
 141 00dc 4889E5                                movq            %rsp, %rbp #,
…
                    36:x86Math.c    ****                        return res;
 156                                            .loc 1 36 0
 157 00fc 8B45F4                                movl            -12(%rbp), %eax
                    # res, D.2781
  37:x86Math.c      ****    }
 158                                            .loc 1 37 0
 159 00ff 5D                                    popq            %rbp            #
 160                                            .cfi_def_cfa 7, 8
 161 0100 C3                                    ret
 162                                            .cfi_endproc
```

add x86 view

```
int add(int x, int y) {
        int op1=x;
        int op2=y;
        int res=op1+op2;
        return res;
}
```

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -20(%rbp)
movl    %esi, -24(%rbp)

movl    -20(%rbp), %eax
movl    %eax, -4(%rbp)

movl    -24(%rbp), %eax
movl    %eax, -8(%rbp)

movl    -4(%rbp), %edx
movl    -8(%rbp), %eax
addl    %edx, %eax
movl    %eax, -12(%rbp)

movl    -12(%rbp), %eax

popq    %rbp
ret
```

Memory		
	xFFFF E860	
op1	xFFFF E85C	x0000 0000
op2	xFFFF E858	x0000 0004
res	xFFFF E854	x0000 0004
	xFFFF E850	
x	xFFFF E84C	x0000 0000
y	xFFFF E848	x0000 0004
	...	

Reg	Value
rbp	x7FFF FFFF FFFF E860
rdi	x0000 0000 0000 0000
rsi	x0000 0000 0000 0004
rax	temp-> x0000 0004
rdx	temp

Instruction in Memory

```
400621:   55          push   %rbp
400622:   48 89 e5    mov    %rsp,%rbp
400625:   89 7d ec    mov    %edi,-0x14(%rbp)
400628:   89 75 e8    mov    %esi,-0x18(%rbp)
40062b:   8b 45 ec    mov    -0x14(%rbp),%eax
40062e:   89 45 fc    mov    %eax,-0x4(%rbp)
400631:   8b 45 e8    mov    -0x18(%rbp),%eax
400634:   89 45 f8    mov    %eax,-0x8(%rbp)
400637:   8b 55 fc    mov    -0x4(%rbp),%edx
40063a:   8b 45 f8    mov    -0x8(%rbp),%eax
40063d:   01 d0       add    %edx,%eax
```

Memory		
	xFFFF E860	
op1	xFFFF E85C	x0000 0004
op2	xFFFF E858	x0000 0003
	xFFFF E854	
	xFFFF E850	
x	xFFFF E84C	x0000 0000
y	xFFFF E848	x0000 0004
	...	
35	0040 0638	x55fc 8b45
	0040 0634	x8945 f88b
34	0040 0630	xfc8b 45e8
	0040 062C	x45ec 8945
33	0040 0628	x8975 e88b
	0040 0624	xe589 7dec
32	0040 0620	x??55 4889

Displaying x86 instructions

(gdb) disas[semble] [/m]

- prints object and x86 assembler version of current function
- /m – include C symbols/instructions when available

Example of disassemble with debug

```
(gdb) disassemble /m
Dump of assembler code for function add:
32          int add(int x, int y) {
  0x0000000000400621 <+0>:    push   %rbp
  0x0000000000400622 <+1>:    mov    %rsp,%rbp
  0x0000000000400625 <+4>:    mov    %edi,-0x14(%rbp)
  0x0000000000400628 <+7>:    mov    %esi,-0x18(%rbp)
33              int op1=x;
=> 0x000000000040062b <+10>:   mov    -0x14(%rbp),%eax
  0x000000000040062e <+13>:   mov    %eax,-0x4(%rbp)
34              int op2=y;
  0x0000000000400631 <+16>:   mov    -0x18(%rbp),%eax
  0x0000000000400634 <+19>:   mov    %eax,-0x8(%rbp)
35              int res=op1+op2;
  0x0000000000400637 <+22>:   mov    -0x4(%rbp),%edx
  0x000000000040063a <+25>:   mov    -0x8(%rbp),%eax
  0x000000000040063d <+28>:   add    %edx,%eax
  0x000000000040063f <+30>:   mov    %eax,-0xc(%rbp)
36              return res;
  0x0000000000400642 <+33>:   mov    -0xc(%rbp),%eax
37          }
  0x0000000000400645 <+36>:   pop    %rbp
  0x0000000000400646 <+37>:   retq
End of assembler dump.
```

C line number

C Instruction

Next instruction waiting to execute

Address of x86 Instruction

x86 instruction

Offset from start of function

gdb w/o debug

Notice… break in add AFTER function entry!

```
(gdb) b add
Breakpoint 1 at 0x400625
(gdb) run 4
Starting program: /import/linux/home/tbartens/CS220/lab07_sol/x86Math 4
Breakpoint 1, 0x0000000000400625 in add ()
(gdb) disassemble /m
Dump of assembler code for function add:
   0x0000000000400621 <+0>:     push   %rbp
   0x0000000000400622 <+1>:     mov    %rsp,%rbp
=> 0x0000000000400625 <+4>:     mov    %edi,-0x14(%rbp)
   0x0000000000400628 <+7>:     mov    %esi,-0x18(%rbp)
   0x000000000040062b <+10>:    mov    -0x14(%rbp),%eax
   0x000000000040062e <+13>:    mov    %eax,-0x4(%rbp)
   0x0000000000400631 <+16>:    mov    -0x18(%rbp),%eax
   0x0000000000400634 <+19>:    mov    %eax,-0x8(%rbp)
   0x0000000000400637 <+22>:    mov    -0x4(%rbp),%edx
   0x000000000040063a <+25>:    mov    -0x8(%rbp),%eax
   0x000000000040063d <+28>:    add    %edx,%eax
   0x000000000040063f <+30>:    mov    %eax,-0xc(%rbp)
   0x0000000000400642 <+33>:    mov    -0xc(%rbp),%eax
   0x0000000000400645 <+36>:    pop    %rbp
   0x0000000000400646 <+37>:    retq
End of assembler dump.
```

GDB at the Assembly Level

- To step through C code, use “step” or “next”
- To step through code at the X86 Assembly level use “stepi” or “nexti”
  - Executes a single x86 instruction
- If that instruction is a function call (mnemonic “call”)
  - nexti stops when that function returns
  - stepi stops at the first instruction in the function

Continuous x86 Assembler Print

- When I debug at the C level, gdb prints out the C instruction it is about to execute
- When I do “stepi” or “nexti”, all I get is an address…
- Until I execute:

(gdb) set disassemble-next-line on

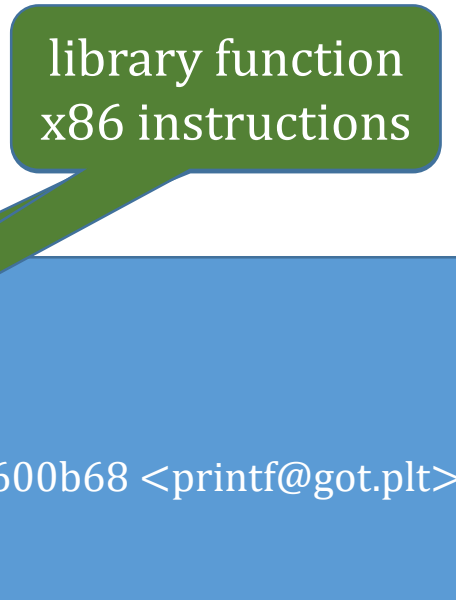
Stepping through C/x86 code

- With debug, step executes to next (debugged) C instruction
  - If you invoke a function that was compiled without debug, skips that function!
- Without debug, step moves to next (debugged) C instruction
  - If code compiled without –g, executes until main ends
  - Practically Useless!
- Alternatives : nexti and stepi
  - Executes to the next x86 instruction
  - nexti skips function calls
  - stepi steps into functions [but might be protected (invisible) code!]

Avoid Stepping Into Protected/Lib Code

- If you do, stepi continues to work…
- But you can't see where you are
- You can't see the instructions you are executing
- You may use the "finish" command to continue this function until it returns to its caller

stepi with disassemble-next-line




```
(gdb) set disassemble-next-line on
=> 0x0000000000400615 <main+207>:      e8 f6 fd ff ff  callq  0x400410 <printf@plt>
(gdb) stepi
0x0000000000400410 in printf@plt ()
=> 0x0000000000400410 <printf@plt+0>:   ff 25 52 07 20 00      jmpq   *0x200752(%rip)        # 0x600b68 <printf@got.plt>
(gdb)
0x0000000000400416 in printf@plt ()
=> 0x0000000000400416 <printf@plt+6>:   68 00 00 00 00  pushq  $0x0
(gdb) finish
Run till exit from #0  printf@plt ()
    at ../sysdeps/x86_64/dl-trampoline.S:41
x=4, x squared - 4x + 4 =4 divided by x-2=2
0x000000000040061a in main ()
=> 0x000000000040061a <main+212>:      b8 00 00 00 00  mov    $0x0,%eax
(gdb)
```

Breakpoints in X86

- Its no fun to step through an entire program.
- I want to set a breakpoint… but there is no line number
  - Especially if there is no debug turned on!

- (gdb) break *<address>

- Sets a breakpoint at a specific instruction address
  - To specify a hexadecimal address, use “0x” prefix!

break at addr

```
(gdb) disassemble
Dump of assembler code for function add:
   0x0000000000400621 <+0>:    push   %rbp
   0x0000000000400622 <+1>:    mov    %rsp,%rbp
=> 0x0000000000400625 <+4>:    mov    %edi,-0x14(%rbp)
   0x0000000000400628 <+7>:    mov    %esi,-0x18(%rbp)
...
End of assembler dump.
(gdb) break *0x400621
Breakpoint 3 at 0x400621
(gdb) run 4
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /import/linux/home/tbartens/CS220/lab07_sol/x86Math 4

Breakpoint 3, 0x0000000000400621 in add ()
=> 0x0000000000400621 <add+0>:  55     push   %rbp
(gdb)
```

Register Information

- (gdb) info reg
- Displays x86 regs and values
- Individual registers can be used like variables
  - Use “$” prefix
  - e.g. print $eax
  - or print *((int *)$rbp-4)

```
(gdb) info reg
rax            0x0      0
rbx            0x0      0
rcx            0x0      0
rdx            0x10     16
rsi            0x4      4
rdi            0x0      0
rbp            0x7fffffffe860   0x7fffffffe860
rsp            0x7fffffffe838   0x7fffffffe838
r8             0x7ffff7dd6f20   140737351872288
r9             0x7fffffffebf0   140737488350192
r10            0x0      0
r11            0x4      4
r12            0x400450 4195408
r13            0x7fffffffe940   140737488349504
r14            0x0      0
r15            0x0      0
rip            0x400621 0x400621 <add>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

Memory

- (gdb) x /*nfu address_expression*
  - eXamines memory starting at *address_expression* using format *nfu*
  - *n* - Number of values to print
  - *f* - Format:
  - *u* - Unit size (width)

- *address_expression* can be
  - Constant, e.g. (gdb) x /4i 0x1004011b5
  - Register, e.g. (gdb) x /4i $eip
  - Pointer variable, e.g. (gdb) x /8cb argv[0]
  - Expression, e.g. (gdb) x /d $rbp-0x20

f	
x	hexadecimal
d	decimal
u	unsigned dec
f	floating point
a	address
c	character
s	string
i	instruction

u		
b	1	8
h	2	16
w	4	32
q	8	64

Examine Examples

10 decimal 4 byte numbers starting at %rbp-32

same, but in hex

null terminated string starting at 0x400760

next 4 x86 instructions

```
(gdb) p $rbp
$2 = (void *) 0x7fffffffe860
(gdb) x /10dw $rbp-32
0x7fffffffe840: -5816   32767   4195408 2
0x7fffffffe850: -5824   16      0       4
0x7fffffffe860: 0       0
(gdb) x /10xw $rbp-32
0x7fffffffe840: 0xffffe948      0x00007fff      0x00400450      0x00000002
0x7fffffffe850: 0xffffe940      0x00000010      0x00000000      0x00000004
0x7fffffffe860: 0x00000000      0x00000000
(gdb) x /s 0x400760
0x400760:       "x=%d, x squared - 4x + 4 =%d divided by x-2=%d\n"
(gdb) x /4i $rip
=> 0x400621 <add>:      push   %rbp
   0x400622 <add+1>:    mov    %rsp,%rbp
   0x400625 <add+4>:    mov    %edi,-0x14(%rbp)
   0x400628 <add+7>:    mov    %esi,-0x18(%rbp)
```

Don’t Forget Other Cool GDB stuff

```
(gdb)break *0x080483c3 if $eax > 13
(gdb) commands
x /4dw $rsp
stepi
end
(gdb)
```

Parameter Passing Conventions

- Parameters are put into the following registers by the caller:
  - (Parameters which don't fit are pushed on the stack)

Parm 1	Parm 2	Parm 3	Parm 4	Parm 5	Parm 6
%rdi	%rsi	%rdx	%rcx	%r8	%r9
Arg 1	Arg 2	Arg 3	Arg 4	Arg 5	Arg 6

- Arguments are read from these registers by the callee
  - (Parameters which don't fit are read from caller's stack frame)

Return Value Convention

- Callee will put return value in %rax before return
- Caller can read return value from %rax after return

Working on the Bomb Project

- Use gdb (at X86 level)
- Abstract as much as possible
  - If you invoke a function called “compare_two_strings”, don’t step into that function…. you can guess what it does
- Look up X86 assembler you don’t understand
  - e.g. “test %eax,%eax” – bitwise and’s %eax with itself to set condition codes
  - e.g. “bne <label>” – branch not equal… branches if ZF is off
  - The two together are an idiom for “branch if %eax is not zero” (and %eax is the 32 bit return value from a called function)