CS 107
Lecture 12: Assembly Part III

Friday, February 16, 2018

Computer Systems
Winter 2018
Stanford University
Computer Science Department

Reading: Course Reader: x86-64 Assembly Language, Textbook: Chapter 3.1-3.4

Lecturer: Chris Gregg

Stack "bottom"
Earlier Frames

Increasing address

Frame for calling function P

Argument n

Frame for executing function Q

Argument 7

Return address

Saved Registers

Local Variables

Argument build area

Stack pointer %rsp

Stack "top"
Today's Topics

- Reading: Chapter 3.6.7, 3.7
- Programs from class: /afs/ir/class/cs107/samples/lect13
- Monday: there will be a screencast-only lecture (final preparation for binary bomb, out on Tuesday). No class next Friday.
- Comments on assembly writing
- More x86 Assembly Language
  - Branch walkthrough
  - Conditional moves
  - Loops: While, For
  - Procedures
    - The run-time stack
    - Control transfer (call/return)
    - Data transfer (arguments)
    - Local storage on the stack
    - Local storage in registers
    - Recursion
Comments on Assembly Deciphering

- You will practice using gdb to decipher assembly on assign5 (which is short) -- the bbsort reverse engineering is similar to the binary bomb assignment, which is next!
- The following commands are very useful:
  - `display/4i $rip`  This command displays four assembly instructions at once after each step. It is a great way to see what is coming up in the assembly.
  - `si`  This command steps one assembly instruction at a time
  - `disas function_name`  This will disassemble an entire function for you.
  - `p $rax`  This prints the value of $rax.
  - `p *(char **)($rsp)`  This dereferences a char ** pointer in $rsp and prints the string value.
  - `p *(char **)$rsp@10`  If $rsp points to the start of an array, this will print out the first ten elements in the array!

If you think of disassembling as a puzzle, it is actually kind of fun!
Practice: Reverse-engineer Assembly to C

• Let's look at the following (print_arr.c from samples/lect12)

```c
int main(int argc, char **argv)
{
    int i_array[] = ________________________________
    size_t i_nelems = ________________________________
    long l_array[] = ________________________________
    size_t l_nelems = ________________________________
    print_array(________, ________, __________, ________);
    print_array(________, ________, __________, ________);
    return 0;
}
```

```assembly
sub    $0x58,%rsp
movl   $0x0,0x30(%rsp)
movl   $0x1,0x34(%rsp)
movl   $0x2,0x38(%rsp)
movl   $0x3,0x3c(%rsp)
movl   $0x4,0x40(%rsp)
movl   $0x5,0x44(%rsp)
movq   $0x0,(%rsp)
movq   $0xa,0x8(%rsp)
movq   $0xa,0x8(%rsp)
movq   $0x14,0x10(%rsp)
movq   $0x1e,0x18(%rsp)
movq   $0x28,0x20(%rsp)
movq   $0x32,0x28(%rsp)
mov    $0x400596,%ecx
mov    $0x4,%edx
mov    $0x6,%esi
lea    0x30(%rsp),%rdi
callq  0x4005d5 <print_array>
mov    $0x4005b5,%ecx
mov    $0x8,%edx
mov    %rsp,%rdi
callq  0x4005d5 <print_array>
mov    $0x0,%eax
add    $0x58,%rsp
retq
```
Practice: Reverse-engineer Assembly to C

• Let's look at the following:

```c
int main(int argc, char **argv)
{
    int i_array[] = {0,1,2,3,4,5};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);
    long l_array[] = {0,10,20,30,40,50};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);
    print_array(i_array,i_nelems,sizeof(i_array[0]),print_int);
    print_array(l_array,l_nelems,sizeof(l_array[0]),print_long);
    return 0;
}
```

```assembly
sub    $0x58,%rsp
movl   $0x0,0x30(%rsp)
movl   $0x1,0x34(%rsp)
movl   $0x2,0x38(%rsp)
movl   $0x3,0x3c(%rsp)
movl   $0x4,0x40(%rsp)
movl   $0x5,0x44(%rsp)
movq   $0x0,(%rsp)
movq   $0xa,0x8(%rsp)
movq   $0x14,0x10(%rsp)
movq   $0x1e,0x18(%rsp)
movq   $0x28,0x20(%rsp)
movq   $0x32,0x28(%rsp)
mov    $0x400596,%ecx
mov    $0x4,%edx
mov    $0x6,%esi
lea    0x30(%rsp),%rdi
callq  0x4005d5 <print_array>
mov    $0x4005b5,%ecx
mov    $0x8,%edx
mov    $0x6,%esi
mov    %rsp,%rdi
callq  0x4005d5 <print_array>
mov    $0x0,%eax
add    $0x58,%eax
retq
```
void print_array(void *arr, size_t nelems, int width, void(*pr_func)(void *))
{
    for ( ________________________________ ) {
        void *element = ______________________
        pr_func(____________________ )
        i == nelems - 1 ? printf("\n") : printf(" , ");
    }
}
Let's look at the following:

```c
void print_array(void *arr, size_t nelems, int width, void(*pr_func)(void *))
{
    for (int i=0; i < nelems; i++) {
        void *element = (char *)arr + i * width;
        pr_func(element);
        i == nelems - 1 ? printf("\n") : printf("", ");
    }
}
```

...pushes, move $rsp

```
0x4005e3 <+14>:  mov    %rdi,%r15
0x4005e6 <+17>:  mov    %rsi,%r12
0x4005e9 <+20>:  mov    %edx,%r14d
0x4005ec <+23>:  mov    %rcx,%r13
0x4005ef <+26>:  mov    $0x0,%ebx
0x4005f4 <+31>:  jmp    0x400632 <print_array+93>
0x4005f6 <+33>:  mov    %ebx,%edi
0x4005f8 <+35>:  imul   %r14d,%edi
0x4005fc <+39>:  movslq %edi,%rdi
0x4005ff <+42>:  add    %r15,%rdi
0x400602 <+45>:  callq  *%r13
0x400609 <+48>:  lea    -0x1(%r12),%rax
0x40060d <+52>:  cmp    %rax,%rbp
0x400612 <+57>:  jne    0x40061b <print_array+70>
0x400614 <+63>:  callq  0x400460 <putchar@plt>
0x400619 <+68>:  jmp    0x40062f <print_array+90>
0x40061b <+70>:  mov    $0x40077b,%esi
0x400620 <+75>:  mov    $0x1,%edi
0x400625 <+80>:  mov    $0x0,%eax
0x40062a <+85>:  callq  0x400480 <__printf_chk@plt>
0x40062f <+90>:  add    $0x1,%ebx
0x400632 <+93>:  movslq %ebx,%rbp
0x400635 <+96>:  cmp    %r12,%rbp
0x400638 <+99>:  jb     0x4005f6 <print_array+33>
0x40063a <+101>: add    $0x8,%rsp
```
Conditional Moves

- The x86 processor provides a set of "conditional move" instructions that move memory based on the result of the condition codes, and that are completely analogous to the jump instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Move Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmove S,R</td>
<td>cmovz</td>
<td>Equal / zero (ZF=1)</td>
</tr>
<tr>
<td>cmovne S,R</td>
<td>cmovnz</td>
<td>Not equal / not zero (ZF=0)</td>
</tr>
<tr>
<td>cmovs S,R</td>
<td></td>
<td>Negative (SF=1)</td>
</tr>
<tr>
<td>cmovns S,R</td>
<td></td>
<td>Nonnegative (SF=0)</td>
</tr>
<tr>
<td>cmovg S,R</td>
<td>cmovnle</td>
<td>Greater (signed &gt;) (SF=0 and SF=OF)</td>
</tr>
<tr>
<td>cmovge S,R</td>
<td>cmovnl</td>
<td>Greater or equal (signed &gt;=) (SF=OF)</td>
</tr>
<tr>
<td>cmovl S,R</td>
<td>cmovnge</td>
<td>Less (signed &lt;) (SF != OF)</td>
</tr>
<tr>
<td>cmovle S,R</td>
<td>cmovng</td>
<td>Less or equal (signed &lt;=) (ZF=1 or SF!=OF)</td>
</tr>
<tr>
<td>cmova S,R</td>
<td>cmovnbe</td>
<td>Above (unsigned &gt;) (CF = 0 and ZF = 0)</td>
</tr>
<tr>
<td>cmovae S,R</td>
<td>cmovnb</td>
<td>Above or equal (unsigned &gt;=) (CF = 0)</td>
</tr>
<tr>
<td>cmovb S,R</td>
<td>cmovnae</td>
<td>Below (unsigned &lt;) (CF = 1)</td>
</tr>
<tr>
<td>cmovbe S,R</td>
<td>cmovna</td>
<td>Below or equal (unsigned &lt;=) (CF = 1 or ZF = 1)</td>
</tr>
</tbody>
</table>

- With these instructions, we can sometimes eliminate branches, which are particularly inefficient on modern computer hardware.
Jumps -vs- Conditional Move

**long absdiff(long x, long y)**
{
    long result;
    if (x < y)
        result = y - x;
    else
        result = x - y;
    return result;
}

**long cmovdiff(long x, long y)**
{
    long rval = y - x;
    long eval = x - y;
    long ntest = x >= y;
    if (ntest) rval = eval;
    return rval;
}

# x in %rdi, y in %rsi
absdiff:
    cmpq %rsi, %rdi
    jge .L2
    movq %rsi, %rax
    subq %rdi, %rax
    ret
.L2:
    movq %rdi, %rax
    subq %rsi, %rax
    ret

cmovdiff:
    movq %rsi, %rax
    subq %rdi, %rax
    movq %rdi, %rdx
    subq %rsi, %rdx
    cmpq %rsi, %rdi
    cmovge %rdx, %rax
    ret

Which is faster?
Let's test!

$ ./abstest 100000 a < /dev/urandom
$ ./abstest 100000 c < /dev/urandom
$ ./abstest 100000 c < /dev/zero
Loops: While loop

In C, the general form of the while loop is:

```c
while (test_expr)
    body_statement
```

gcc often translates this into the following general assembly form:

```assembly
jmp test // unconditional jump
loop:
    body_statement instructions
test:
    cmp ... // comparison instruction
    j.. loop // conditional jump based on comparison
    // can think of as "if test, goto loop"
```
Loops: While loop

Let's look at the following C factorial function, and the generated assembly:

```c
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n - 1;
    }
    return result;
}
```

```assembly
// long fact_while(long n)
// n in %rdi
fact_while:
    movl $1, %eax           // set result to 1
    jmp .L5                // jmp to test
.L6                    // loop:
    imulq %rdi, %rax       // compute result *=n
    subq $1, %rdi          // decrement n
.L5                    // test:
    cmpq $1, %rdi          // compare n:1
    jg .L6                // if >, jmp to loop
rep; ret                // return
```

Often, the key to reverse-engineering these loops is to recognize the form — from there, the rest is relatively straightforward, with practice (and it is fun!)
Fill in the missing C code:

```c
long loop_while(long a, long b) {
    long result = __________;
    while (__________) {
        result = _________________;
        a = __________;
    }
    return result;
}
```

```assembly
// long loop_while(long a, long b) // a in %rdi, b in %rsi
loop_while:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```
fill in the missing C code:

```c
long loop_while(long a, long b)
{
    long result = 1;
    while (a < b) {
        result = result * (a + b);
        a = a + 1;
    }
    return result;
}
```

```c
// long loop_while(long a, long b)
// a in %rdi, b in %rsi
loop_while:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```
Loops: While loop, alternate form

In C, the general form of the while loop is:

```c
while (test_expr)
    body_statement
```

On higher levels of optimization, gcc composes this assembly:

```assembly
cmp ... // comparison instruction
j.. done // conditional jump based on comparison
    // can think of as "if not test, goto done"
loop:
    body_statement instructions
    cmp ... // comparison instruction
    j.. loop // "if test, goto loop"
done:
```
Let's look at the following C factorial function, and the generated assembly:

```c
long fact_while(long n) {
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n - 1;
    }
    return result;
}
```

```assembly
// long fact_while(long n)
// n in %rdi
fact_while:
    cmpq $1, %rdi          // compare n:1
    jle .L7               // if <=, jmp done
    movl $1, %eax          // set result to 1
.L6                     // loop:
    imulq %rdi, %rax       // compute result *=n
    subq $1, %rdi          // decrement n
    cmpq $1, %rdi          // compare n:1
    jne .L6               // if !=, goto loop
    rep; ret               // return
.L7                     // done:
    movl $1, %eax          // compute result = 1
    rep; ret               // return
```
Loops: While loop, reverse engineering

Fill in the missing C code:

```c
long loop_while2(long a, long b) {
    long result = __________;
    while (__________) {
        result = _________________;
        b = __________;
    }
    return result;
}
```

// long loop_while2(long a, long b) // a in %rdi, b in %rsi
loop_while2:
    testq %rsi, %rsi
    jle .L8
    movq %rsi, %rax
.L7
    imulq %rdi, %rax
    subq %rdi, %rsi
    testq %rsi, %rsi
    jg .L7
    rep; ret
.L8
    movq %rsi, %rax
    ret
Fill in the missing C code:

```c
long loop_while2(long a, long b) {
  long result = b;
  while (b > 0) {
    result = result * a;
    b = b - a;
  }
  return result;
}
```

// long loop_while2(long a, long b)  // a in %rdi, b in %rsi
loop_while2:
  testq %rsi, %rsi
  jle .L8
  movq %rsi, %rax
.L7
  imulq %rdi, %rax
  subq %rdi, %rsi
  testq %rsi, %rsi
  jg .L7
  rep; ret
.L8
  movq %rsi, %rax
  ret
Loops: For loop

In C, the general form of the for loop is:

```c
for (init_expr; test_expr; update_expr)
    body_statement
```

This is the same as the following (unless you have a continue statement; see Problem 3.29 in the text):

```c
init_expr;
while (test_expr) {
    body_statement
    update_expr;
}
```

The program first evaluates `init_expr`. Then it enters the loop where it first evaluates the test condition, `test_expr`, exiting if the test fails. Then, it continues the `body_statement`, and finally evaluates the update expression, `update_expr`. 
Loops: For loop

There are two forms that gcc might create; this is the first form:

```plaintext
init_expression
jmp test // unconditional jump
loop:
  body_statement instructions
  update_expression
test:
  cmp ... // comparison instruction
  j.. loop // conditional jump based on comparison
    // can think of as "if test, goto loop"
```
Loops: For loop

There are two forms that gcc might create; this is the second form:

```
init_expression
cmp ... // comparison instruction
done // conditional jump based on comparison
    // can think of as "if not test, goto done"

loop:
    body_statement instructions
    update_expression
cmp ... // comparison instruction
    j.. loop // "if test, goto loop"

done:
```
Loops: For loop

Let's look at another C factorial function, and the generated assembly:

```c
long fact_for(long n)
{
    long i;
    long result = 1;
    for (i = 2; i <= n; i++)
        result *= i;
    return result;
}
```

```assembly
// long fact_for(long n)
// n in %rdi
fact_for:
    movl $1, %eax    // set result to 1
    movl $2, %edx    // set i = 2
    jmp .L8          // jmp to test
.L9
    // loop:
    imulq %rdx, %rax  // compute result *=i
    addq $1, %rdx    // increment i
.L8
    // test:
    cmpq %rdi, %rdx  // compare n:i
    jle .L9          // if <=, jmp to loop
    rep; ret         // return
```

Again: recognize the form (while loop, for loop), and reverse engineering it is not too difficult.
Take a break!
Procedures (Functions)

- Procedures are key software abstractions: they package up code that implements some functionality with some arguments and a potential return value.
- In C, we have to handle a number of different attributes when calling a function.
- For example, suppose function P calls function Q, which executes and returns to P. We have to:
  1. Pass control: update the program counter (%rip) to the start of the function, and then at the end of the function, set the program counter to the instruction after the call.
  2. Pass data: P must be able to provide parameters to Q, and Q must be able to return a value back to P.
  3. Allocate and deallocate memory: Q may need to allocate space for local variables when it begins, and free the storage when it returns.
- The x86-64 implementation has special instructions and conventions that allow this to be smooth and efficient, and we only need to implement the parts necessary for our particular function.
The Run-Time Stack

- Procedures make use of the run-time stack that we have discussed before.
- For the previous example, while function Q is executing, P (along with any of the calls up to P) is temporarily suspended.
- While Q is running, it may need space for local variables, and it may call other functions.
- When Q returns, it can free its local storage.
- Data can be stored on the stack using `push` or `pop`, or stack allocation can happen by decrementing the stack pointer.
- Space can be deallocated by incrementing the stack pointer.
When an x86-64 procedure requires storage beyond what it can store in registers, it allocates space on the stack.

This is the **stack frame** for that procedure.

When procedure P calls procedure Q, it pushes the *return address* onto the stack. This indicates where in P the function should resume when Q returns (and the return address is part of P's stack frame).

If a procedure needs more than 6 arguments, they go on the stack.

Some functions don't even need a stack frame, if all the local variables can be held in registers, and the function does not call any other functions (this is a *leaf* function).
• When function P calls function Q, the program counter is set to the first instruction in Q.
• But, before this happens, the next instruction in P after the call is pushed onto the stack. This is the return address.
• When Q executes a `ret` instruction, the return address is popped off the stack and the PC is set to that value.
• The `call` instruction can either be to a label, or based on the memory address in a register or memory (e.g., a function pointer).
The Run-Time Stack: Example

Disassembly of leaf(long y), y in %rdi:

L1  400540: 48 8d 47 02  lea 0x2(%rdi),%rax  // z+2
L2  400544: c3              retq  // return

Disassembly of top(long x), x in %rdi:

T1  400545: 48 83 ef 05  sub $0x5,%rdi  // x-5
T2  400549: e8 f2 ff ff ff  callq 400540 <leaf>  // Call leaf(x-5)
T3  40054e: 48 01 c0  add %rax,%rax  // Double result
T4  400551: c3              retq  // Return

... Call to top from function main

M1  40055b: e8 e5 ff ff ff  callq 400545, <top>  // Call top(100)
M2  400560: 48 89 c2  mov %rax,%rdx  // Resume

Instruction | State values (at beginning) | Description
--- | --- | ---
| Label | PC | Instruction | %rdi | %rax | %rsp | *%rsp | Description |
| M1 | 0x40055b | callq | 100 | – | 0x7fffffffff820 | – | Call top(100) |
| T1 | 0x400545 | sub | 100 | – | 0x7fffffffff818 | 0x400560 | Entry of top |
| T2 | 0x400549 | callq | 95 | – | 0x7fffffffff818 | 0x400560 | Call leaf(95) |
| L1 | 0x400540 | lea | 95 | – | 0x7fffffffff810 | 0x40054e | Entry of leaf |
| L2 | 0x400544 | retq | – | 97 | 0x7fffffffff810 | 0x40054e | Return 97 from leaf |
| T3 | 0x40054e | add | – | 97 | 0x7fffffffff818 | 0x400560 | Resume top |
| T4 | 0x400551 | retq | – | 194 | 0x7fffffffff818 | 0x400560 | Return 194 from top |
| M2 | 0x400560 | mov | – | 194 | 0x7fffffffff820 | – | Resume main |
Disassembly of leaf(long y), y in %rdi:

L1  400540: 48 8d 47 02  lea 0x2(%rdi),%rax  // z+2
L2  400544: c3              retq  // return

Disassembly of top(long x), x in %rdi:

T1  400545: 48 83 ef 05  sub $0x5,%rdi  // x-5
T2  400549: e8 f2 ff ff ff  callq 400540 <leaf>  // Call leaf(x-5)
T3  40054e: 48 01 c0  add %rax,%rax  // Double result
T4  400551: c3              retq  // Return

The Run-Time Stack: Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>State values (at beginning)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>40055b</td>
<td>Call top(100)</td>
</tr>
<tr>
<td>T1</td>
<td>400545</td>
<td>Entry of top</td>
</tr>
<tr>
<td>T2</td>
<td>400549</td>
<td>Cal leaf(95)</td>
</tr>
<tr>
<td>L1</td>
<td>400540</td>
<td>Entry of leaf</td>
</tr>
<tr>
<td>L2</td>
<td>400544</td>
<td>Return 97 from leaf</td>
</tr>
<tr>
<td>T3</td>
<td>40054e</td>
<td>Resume top</td>
</tr>
<tr>
<td>T4</td>
<td>400551</td>
<td>Return 194 from top</td>
</tr>
<tr>
<td>M2</td>
<td>400560</td>
<td>Resume main</td>
</tr>
</tbody>
</table>
The Run-Time Stack: Example

Disassembly of leaf(long y), y in %rdi:
00000000000400540 <leaf>:
L1  400540: 48 8d 47 02  lea 0x2(%rdi),%rax // z+2
L2  400544: c3              retq // return

Disassembly of top(long x), x in %rdi:
00000000000400540 <top>:
T1  400545: 48 83 ef 05  sub $0x5,%rdi // x-5
T2  400549: e8 f2 ff ff ff  callq 400540 <leaf> // Call leaf(x-5)
T3  40054e: 48 01 c0        add %rax,%rax // Double result
T4  400551: c3              retq // Return

... Call to top from function main

M1  40055b: e8 e5 ff ff ff  callq 400545, <top>  // Call top(100)
M2  400560: 48 89 c2        mov %rax,%rdx // Resume

Stack "bottom"

Stack "top"

<table>
<thead>
<tr>
<th>Instruction</th>
<th>PC</th>
<th>Instruction</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>*%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>0x40055b</td>
<td>callq</td>
<td>100</td>
<td>—</td>
<td>0x7fffffff820</td>
<td>—</td>
<td>Call top(100)</td>
</tr>
<tr>
<td>T1</td>
<td>0x400545</td>
<td>sub</td>
<td>100</td>
<td>—</td>
<td>0x7fffffff818 0x400560</td>
<td>Entry of top</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>0x400549</td>
<td>callq</td>
<td>95</td>
<td>—</td>
<td>0x7fffffff818 0x400560</td>
<td>Cal leaf(95)</td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>0x400540</td>
<td>lea</td>
<td>95</td>
<td>—</td>
<td>0x7fffffff810 0x40054e</td>
<td>Entry of leaf</td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>0x400544</td>
<td>retq</td>
<td>—</td>
<td>97</td>
<td>0x7fffffff810 0x40054e</td>
<td>Return 97 from leaf</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>0x40054e</td>
<td>add</td>
<td>—</td>
<td>97</td>
<td>0x7fffffff818 0x400560</td>
<td>Resume top</td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>0x400551</td>
<td>retq</td>
<td>—</td>
<td>194</td>
<td>0x7fffffff818 0x400560</td>
<td>Return 194 from top</td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td>0x400560</td>
<td>mov</td>
<td>—</td>
<td>194</td>
<td>0x7fffffff820</td>
<td>—</td>
<td>Resume main</td>
</tr>
</tbody>
</table>
Disassembly of leaf(long y), y in %rdi:

L1  400540: 48 8d 47 02  lea 0x2(%rdi),%rax  // z+2
L2  400544:  c3              retq  // return

Disassembly of top(long x), x in %rdi:

T1  400545: 48 83 ef 05     sub $0x5,%rdi        // x-5
T2  400549: e8 f2 ff ff ff  callq 400540 <leaf>  // Call leaf(x-5)
T3  40054e: 48 01 c0        add %rax,%rax        // Double result
T4  400551: c3              retq                 // Return

... Call to top from function main

M1  40055b: e8 e5 ff ff ff  callq 400545, <top>  // Call top(100)
M2  400560: 48 89 c2        mov %rax,%rdx        // Resume

The Run-Time Stack: Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>State values (at beginning)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>%rax</td>
</tr>
<tr>
<td>M1 0x40055b</td>
<td>callq</td>
</tr>
<tr>
<td>T1 0x400545</td>
<td>sub</td>
</tr>
<tr>
<td>T2 0x400549</td>
<td>callq</td>
</tr>
<tr>
<td>L1 0x400540</td>
<td>lea</td>
</tr>
<tr>
<td>L2 0x400544</td>
<td>retq</td>
</tr>
<tr>
<td>T3 0x40054e</td>
<td>add</td>
</tr>
<tr>
<td>T4 0x400551</td>
<td>retq</td>
</tr>
<tr>
<td>M2 0x400560</td>
<td>mov</td>
</tr>
</tbody>
</table>
## The Run-Time Stack: Example

### Disassembly of leaf(long y), y in %rdi:
```
00000000000400540 <leaf>:
L1  400540: 48 8d 47 02       lea 0x2(%rdi),%rax  // z+2
L2  400544: c3              retq  // return
```

### Disassembly of top(long x), x in %rdi:
```
00000000000400540 <top>:
T1  400545: 48 83 ef 05     sub $0x5,%rdi  // x-5
T2  400549: e8 f2 ff ff ff  callq 400540 <leaf>  // Call leaf(x-5)
T3  40054e: 48 01 c0        add %rax,%rax  // Double result
T4  400551: c3              retq  // Return
```

---

### Call to top from function main
```
M1  40055b: e8 e5 ff ff ff  callq 400545, <top>  // Call top(100)
M2  400560: 48 89 c2        mov %rax,%rdx  // Resume
```

---

### Stack "bottom"
```
- 0x7fffffffde820
```

### Stack "top"
```
- 0x7fffffffde818 0x400560
- 0x7fffffffde810 0x40054e
```

---

### Instruction
<table>
<thead>
<tr>
<th>Label</th>
<th>PC</th>
<th>Instruction</th>
<th>State values (at beginning)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>0x40055b</td>
<td>callq</td>
<td>%rdi: 100 %rax: – %rsp: 0x7fffffffde820 *%rsp: –</td>
</tr>
<tr>
<td>T1</td>
<td>0x400545</td>
<td>sub</td>
<td>%rdi: 100 %rax: – %rsp: 0x7fffffffde818 %rdx: 0x400560</td>
</tr>
<tr>
<td>T2</td>
<td>0x400549</td>
<td>callq</td>
<td>%rdi: 95  %rax: – %rsp: 0x7fffffffde818 0x400560</td>
</tr>
<tr>
<td>L1</td>
<td>0x400540</td>
<td>lea</td>
<td>%rdi: 95  %rax: – %rsp: 0x7fffffffde810 %rdx: 0x40054e</td>
</tr>
<tr>
<td>L2</td>
<td>0x400544</td>
<td>retq</td>
<td>%rdi: – %rax: 97 %rsp: 0x7fffffffde810 0x40054e</td>
</tr>
<tr>
<td>T3</td>
<td>0x40054e</td>
<td>add</td>
<td>%rdi: – %rax: 97 %rsp: 0x7fffffffde818 0x400560</td>
</tr>
<tr>
<td>T4</td>
<td>0x400551</td>
<td>retq</td>
<td>%rdi: – %rax: 194 %rsp: 0x7fffffffde818 0x400560</td>
</tr>
<tr>
<td>M2</td>
<td>0x400560</td>
<td>mov</td>
<td>%rdi: – %rax: 194 %rsp: 0x7fffffffde820 –</td>
</tr>
</tbody>
</table>
### The Run-Time Stack: Example

**Disassembly of leaf(long y), y in %rdi:**

```
0000000000400540 <leaf>:
L1 400540: 48 8d 47 02     lea 0x2(%rdi),%rax  // z+2
L2 400544: c3              retq               // return
```

**Disassembly of top(long x), x in %rdi:**

```
0000000000400540 <top>:
T1 400545: 48 83 ef 05     sub $0x5,%rdi     // x-5
T2 400549: e8 f2 ff ff ff  callq 400540 <leaf> // Call leaf(x-5)
T3 40054e: 48 01 c0        add %rax,%rax       // Double result
T4 400551: c3              retq               // Return
```

... Call to top from function main

```
M1 40055b: e8 e5 ff ff ff  callq 400545, <top> // Call top(100)
M2 400560: 48 89 c2        mov %rax,%rdx      // Resume
```

**Stack "bottom"**

```
0x7fffffff820
0x7fffffff818 0x400560
0x7fffffff810 0x40054e
```

**Stack "top"**

```
```

### Instruction Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>PC</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1 callq</td>
<td>0x40055b</td>
<td>100</td>
<td>-</td>
<td>0x7fffffff820 -</td>
<td>Call top(100)</td>
</tr>
<tr>
<td>T1 sub</td>
<td>0x400545</td>
<td>100</td>
<td>-</td>
<td>0x7fffffff818 0x400560</td>
<td>Entry of top</td>
</tr>
<tr>
<td>T2 callq</td>
<td>0x400549</td>
<td>95</td>
<td>-</td>
<td>0x7fffffff818 0x400560</td>
<td>Cal leaf(95)</td>
</tr>
<tr>
<td>L1 lea</td>
<td>0x400540</td>
<td>95</td>
<td>-</td>
<td>0x7fffffff810 0x40054e</td>
<td>Entry of leaf</td>
</tr>
<tr>
<td>L2 retq</td>
<td>0x400544</td>
<td>-</td>
<td>97</td>
<td>0x7fffffff810 0x40054e</td>
<td>Return 97 from leaf</td>
</tr>
<tr>
<td>T3 add</td>
<td>0x40054e</td>
<td>-</td>
<td>97</td>
<td>0x7fffffff818 0x400560</td>
<td>Resume top</td>
</tr>
<tr>
<td>T4 retq</td>
<td>0x400551</td>
<td>-</td>
<td>194</td>
<td>0x7fffffff818 0x400560</td>
<td>Return 194 from top</td>
</tr>
<tr>
<td>M2 mov</td>
<td>0x400560</td>
<td>-</td>
<td>194</td>
<td>0x7fffffff820 -</td>
<td>Resume main</td>
</tr>
</tbody>
</table>

...
### Disassembly of leaf(long y), y in %rdi:

<table>
<thead>
<tr>
<th>Label</th>
<th>PC</th>
<th>Instruction</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>*%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>0x400540</td>
<td>lea 0x2(%rdi),%rax // z+2</td>
<td>98</td>
<td>96</td>
<td>0x7fffffffe820</td>
<td>0x40054e</td>
<td>Entry of leaf</td>
</tr>
<tr>
<td>L2</td>
<td>0x400544</td>
<td>retq</td>
<td>98</td>
<td>96</td>
<td>0x7fffffffe820</td>
<td>0x40054e</td>
<td>Return 96 from leaf</td>
</tr>
</tbody>
</table>

### Disassembly of top(long x), x in %rdi:

<table>
<thead>
<tr>
<th>Label</th>
<th>PC</th>
<th>Instruction</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>*%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>0x40055b</td>
<td>callq 400545, &lt;top&gt; // Call top(100)</td>
<td>100</td>
<td>100</td>
<td>0x7fffffffe820</td>
<td>0x400560</td>
<td>Call top(100)</td>
</tr>
<tr>
<td>M2</td>
<td>0x400560</td>
<td>mov %rax,%rdx // Resume</td>
<td>100</td>
<td>100</td>
<td>0x7fffffffe818</td>
<td>0x400560</td>
<td>Resume main</td>
</tr>
</tbody>
</table>

---

### Stack "bottom"

- 0x7fffffffe820
- 0x7fffffffe818
- 0x400560

### Stack "top"

- 0x7fffffffe820
- 0x7fffffffe818
- 0x400560

---

### The Run-Time Stack: Example

The Run-Time Stack is used to store variables and function call arguments during program execution.

- **Stack "bottom"**
  - 0x7fffffffe820
  - 0x7fffffffe818
  - 0x400560

- **Stack "top"**
  - 0x7fffffffe820
  - 0x7fffffffe818
  - 0x400560

---

### Stack Values

- **%rdi**: 100
- **%rax**: 98
- **%rsp**: 0x7fffffffe820
- ***%rsp**: 0x400560

---

### Instructions

1. **M1**: `callq 400545, <top>` // Call top(100)
2. **T1**: `sub $0x5,%rdi` // x-5
3. **T2**: `callq 400540 <leaf>` // Call leaf(x-5)
4. **L1**: `lea 0x2(%rdi),%rax` // z+2
5. **L2**: `retq` // Return
6. **T3**: `add %rax,%rax` // Double result
7. **T4**: `retq` // Return
8. **M2**: `mov %rax,%rdx` // Resume

---

### Call to top from function main

- **M1**: `callq 400545, <top>` // Call top(100)
- **M2**: `mov %rax,%rdx` // Resume
### The Run-Time Stack: Example

#### Disassembly of leaf(long y), y in %rdi:

```
L1  400540: 48 8d 47 02     lea 0x2(%rdi),%rax   // z+2
L2  400544: c3              retq                 // return
```

#### Disassembly of top(long x), x in %rdi:

```
T1  400545: 48 83 ef 05     sub $0x5,%rdi        // x-5
T2  400549: e8 f2 ff ff ff  callq 400540 <leaf>  // Call leaf(x-5)
T3  40054e: 48 01 c0        add %rax,%rax        // Double result
T4  400551: c3              retq                 // Return
```

...  

Call to top from function main

```
M1  40055b: e8 e5 ff ff ff  callq 400545, <top>  // Call top(100)
M2  400560: 48 89 c2        mov %rax,%rdx        // Resume
```

---

#### Stack "bottom"

- `0x7fffffffef820`
- `0x7fffffffef818`
- `0x400560`

#### Stack "top"

- `0x7fffffffef820`
- `0x7fffffffef818`
- `0x400560`

---

### Instruction Table

<table>
<thead>
<tr>
<th>Label</th>
<th>PC</th>
<th>Instruction</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>*%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>0x40055b</td>
<td>callq</td>
<td>100</td>
<td>–</td>
<td>0x7fffffffef820</td>
<td>–</td>
<td>Call top(100)</td>
</tr>
<tr>
<td>T1</td>
<td>0x400545</td>
<td>sub</td>
<td>100</td>
<td>–</td>
<td>0x7fffffffef818</td>
<td>0x400560</td>
<td>Entry of top</td>
</tr>
<tr>
<td>T2</td>
<td>0x400549</td>
<td>callq</td>
<td>95</td>
<td>–</td>
<td>0x7fffffffef818</td>
<td>0x400560</td>
<td>Cal leaf(95)</td>
</tr>
<tr>
<td>L1</td>
<td>0x400540</td>
<td>lea</td>
<td>95</td>
<td>–</td>
<td>0x7fffffffef810</td>
<td>0x40054e</td>
<td>Entry of leaf</td>
</tr>
<tr>
<td>L2</td>
<td>0x400544</td>
<td>retq</td>
<td>–</td>
<td>97</td>
<td>0x7fffffffef810</td>
<td>0x40054e</td>
<td>Return 97 from leaf</td>
</tr>
<tr>
<td>T3</td>
<td>0x40054e</td>
<td>add</td>
<td>–</td>
<td>97</td>
<td>0x7fffffffef810</td>
<td>0x400560</td>
<td>Resume top</td>
</tr>
<tr>
<td>T4</td>
<td>0x400551</td>
<td>retq</td>
<td>–</td>
<td>194</td>
<td>0x7fffffffef810</td>
<td>0x400560</td>
<td>Return 194 from top</td>
</tr>
<tr>
<td>M2</td>
<td>0x400560</td>
<td>mov</td>
<td>–</td>
<td>194</td>
<td>0x7fffffffef820</td>
<td>–</td>
<td>Resume main</td>
</tr>
</tbody>
</table>

---

### Stack Diagram

- Stack "bottom":
  - `0x7fffffffef820`  
  - `0x7fffffffef818`  
  - `0x400560`

- Stack "top":
  - `0x7fffffffef820`  
  - `0x7fffffffef818`  
  - `0x400560`
The Run-Time Stack: Example

Disassembly of leaf(long y), y in %rdi:

<table>
<thead>
<tr>
<th>Label</th>
<th>PC</th>
<th>Instruction</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>*%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>0x400540</td>
<td>lea 0x2(%rdi),%rax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>// z+2</td>
</tr>
<tr>
<td>L2</td>
<td>0x400544</td>
<td>retq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>// return</td>
</tr>
</tbody>
</table>

Disassembly of top(long x), x in %rdi:

<table>
<thead>
<tr>
<th>Label</th>
<th>PC</th>
<th>Instruction</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>*%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0x400545</td>
<td>sub $0x5,%rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>// x-5</td>
</tr>
<tr>
<td>T2</td>
<td>0x400549</td>
<td>callq 4000540 &lt;leaf&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Call leaf(x-5)</td>
</tr>
<tr>
<td>T3</td>
<td>0x40054e</td>
<td>add %rax,%rax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>// Double result</td>
</tr>
<tr>
<td>T4</td>
<td>0x400551</td>
<td>retq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>// Return</td>
</tr>
</tbody>
</table>

Call to top from function main

<table>
<thead>
<tr>
<th>Label</th>
<th>PC</th>
<th>Instruction</th>
<th>%rdi</th>
<th>%rax</th>
<th>%rsp</th>
<th>*%rsp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>0x40055b</td>
<td>callq 4000545, &lt;top&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Call top(100)</td>
</tr>
<tr>
<td>M2</td>
<td>0x400560</td>
<td>mov %rax,%rdx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Resume</td>
</tr>
</tbody>
</table>
Data Transfer

- Procedure calls can involve passing data as arguments, and can return a value to the calling function, as well.
- Most often, the arguments can fit into registers, so we don't need to involve the stack. If there are more than six integer arguments, we put them onto the stack.
- When one function calls another, the calling function needs to copy the arguments into the proper registers (\%rdi, \%rsi, \%rdx, \%rcx, \%r8, \%r9, in that order)
- If there are more than 6 arguments, the calling function allocates space on the stack in 8-byte chunks (even if smaller values are passed).
- The return value (if an integer) is returned in the \%rax register.
Data Transfer

• Example:

```c
void proc(long a1, long *a1p,
   int a2, int *a2p,
   short a3, short *a3p,
   char a4, char *a4p)
{
   *a1p += a1;
   *a2p += a2;
   *a3p += a3;
   *a4p += a4;
}
```

Arguments passed as follows:

- `a1` in `%rdi` (64 bits)
- `a1p` in `%rsi` (64 bits)
- `a2` in `%edx` (32 bits)
- `a2p` in `%rcx` (64 bits)
- `a3` in `%r8w` (16 bits)
- `a3p` in `%r9` (64 bits)
- `a4` at `%rsp+8` (8 bits)
- `a4p` at `%rsp+16` (64 bits)

```c
int main()
{
   long a = 1;
   int b = 2;
   short c = 3;
   char d = 4;

   proc(a,&a,b,&b,c,&c,d,&d);
   printf("a:%ld, b:%d, c:%d, d:%d\n",a,b,c,d);
   return 0;
}
```

```c
proc:
movq 16(%rsp), %rax // fetch a4p
addq %rdi, (%rsi) // *a1p += a1
addl %edx, (%rcx) // *a2p += a2
addw %r8w, (%r9) // *a3p += a3
movl 8(%rsp), %edx // Fetch a4
addb %dl, (%rax) // *a4p += a4
ret
```
Data Transfer

- Example:

```c
void proc(long a1, long *a1p,
   int a2, int *a2p,
   short a3, short *a3p,
   char a4, char *a4p)
{
   *a1p += a1;
   *a2p += a2;
   *a3p += a3;
   *a4p += a4;
}
```

Arguments passed as follows:
- `a1` in `%rdi` (64 bits)
- `a1p` in `%rsi` (64 bits)
- `a2` in `%edx` (32 bits)
- `a2p` in `%rcx` (64 bits)
- `a3` in `%r8w` (16 bits)
- `a3p` in `%r9` (64 bits)
- `a4` at `%rsp+8` (8 bits)
- `a4p` at `%rsp+16` (64 bits)

```asm
proc:
    movq 16(%rsp), %rax  // fetch a4p
    addq %rdi, (%rsi)    // *a1p += a1
    addl %edx, (%rcx)    // *a2p += a2
    addw %r8w, (%r9)     // *a3p += a3
    movl 8(%rsp), %edx   // Fetch a4
    addb %dl, (%rax)     // *a4p += a4
    ret
```

Why `movl` and not `movb`?!
Data Transfer

- Example:

```c
void proc(long a1, long *a1p, int a2, int *a2p, short a3, short *a3p, char a4, char *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

Arguments passed as follows:

- `a1` in `%rdi` (64 bits)
- `a1p` in `%rsi` (64 bits)
- `a2` in `%edx` (32 bits)
- `a2p` in `%rcx` (64 bits)
- `a3` in `%r8w` (16 bits)
- `a3p` in `%r9` (64 bits)
- `a4` at `%rsp+8` ( 8 bits)
- `a4p` at `%rsp+16` (64 bits)

**Why `movl` and not `movb`?!**

Under the hood, a mov instruction will still fetch an entire 64-bits, so this is actually faster than retrieving 64-bits and masking out the upper bits (not required information for you to know!)
Local Stack Storage

- We haven't seen much use of local storage on the stack, but there are times when it is necessary:
  - When there are not enough registers to hold the local data
  - When the address operator `&` is applied to a local variable, and we have to be able to generate an address for it.
  - Some of the local variables are arrays or structs, and must be accessed by array or structure references.
  - The typical way to allocate space on the stack frame is to decrement the stack pointer.
  - Remember, a function must return the stack pointer to the proper value (such that the top of the stack is the return address) before it returns.
Local Stack Storage

• Example:

```c
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

caller:

```assembly
subq $16, %rsp      // allocate 16 bytes for stack frame
movq $534, (%rsp)   // store 534 in arg1
movq $1057, 8(%rsp) // store 1057 in arg2
lea 8(%rsp), %rsi   // compute &arg2 as second argument
movq %rsp, %rdi     // compute &arg1 as first argument
call swap_add       // call swap_add(&arg1, &arg2)
movq (%rsp),%rdx    // get arg1
subq 8(%rsp), %rdx  // compute diff = arg1 - arg2
imulq %rdx, %rax    // compute sum * diff
addq $16, %rsp      // deallocate stack frame
ret
```

• The caller must allocate a stack frame due to the presence of address operators.
Local Storage in Registers

- You may not have noticed, but none of the examples in the book so far have used `%rbx` for anything, and gcc often uses `%rax`, `%rcx`, `%rdx`, etc., but skips right over `%rbx`.
- One reason is that `%rbx` is designated, *by convention*, to be a "caller owned" register:

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%ax</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bl</td>
</tr>
</tbody>
</table>

- What that means is that if a function uses `%rbx`, it guarantees that it will restore `%rbx` to its original value when the function returns.
- The full list of callee saved registers are `%rbx`, `%rbp`, and `%r12-%r15`.
- If a function uses any of those registers, it must save them on the stack to restore.
Local Storage in Registers

• The other registers are "callee owned", meaning that if function P calls function Q, function P must save the values of those registers on the stack (or in caller owned registers!) if it wants to retain the data after function Q is called.

• Example:

```c
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

• The first time Q is called, x must be saved for later, and the second time Q is called, u must be saved for later.

```c
long P(long x, long y),
x in %rdi, y in %rsi:
push %rbp
push %rbx
mov %rdi,%rbp
mov %rsi,%rdi
callq 40056d <Q(long)>
mov %rax,%rbx
mov %rbp,%rdi
callq 40056d <Q(long)>
add %rbx,%rax
pop %rbx
pop %rbp
retq
```
• The conventions we have been discussing allow for functions to call themselves. Each procedure call has its own private space on the stack, and the local variables from all of the function calls do not interfere with each other.
• The only thing a program needs to worry about is a stack overflow, because the stack is a limited resource for a program.
• Example:

```c
long rfact(long n)
{
    long result;
    if (n <= 1)
        result = 1;
    else
        result = n * rfact(n-1);
    return result;
}
```

```asm
rfact:
    pushq %rbx
    save %rbx
    movq %rdi,%rbx
    store n in callee-saved reg
    movl $1,%eax
    set return value = 1
    cmpq $1,%rdi
    compare n:1
    jle .L35
    if <=, jump to done
    leaq -1(%rdi),%rdi
    compute n-1
    call rfact
    recursively call rfact(n-1)
    imulq %rbx,%rax
    multiply result by n
    .L35
    done:
    pop %rbx
    restore %rbx
    retq
    return
```
References and Advanced Reading

• References:
  • Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  • CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  • gdbtui: https://beej.us/guide/bggdb/
  • More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  • Compiler explorer: https://gcc.godbolt.org

• Advanced Reading:
  • Stack frame layout on x86-64: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
  • history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  • x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64
Extra Slides