CS356: Discussion #6

Assembly Procedures and Arrays

Marco Paolieri (paolieri@usc.edu)

USC University of Southern California

Procedures

Functions are a key abstraction in software

- They break down a problem into subproblems.
- Reusable functionality: they can be invoked from many points.
- Well-defined interface: expected inputs and produced outputs.
- They hide implementation details.

Problems of function calls

- Passing control to the function and returning.
- Passing parameters and receiving return values.
- Storing local variables during function execution.
- Using registers without interference with other functions.

Intel x86-64 solution

- **Instructions**, such as `callq` and `retq`
- **Conventions**, e.g., store the result in `%rax`

Application Binary Interface

Conventions are needed!

Caller and callee must agree on:

- How to pass control.
- How to pass parameters and receive return values.
- How to preserve register values during function calls.
- How to align values in memory.

System V ABI

- Used by most Unix operating systems (Linux, BSD, macOS)
- Different conventions for different architectures (e.g, i386, x86-64)

By disassembling binary code, we will see many of these conventions in action for the **x86-64 architecture**.

The **stack** plays a fundamental role in function calls...

Case study: a stack

Pushing a value

- Decrement stack pointer `%rsp`
- Store new value at address pointed by `%rsp`

Example: `pushq %rax` is equivalent to

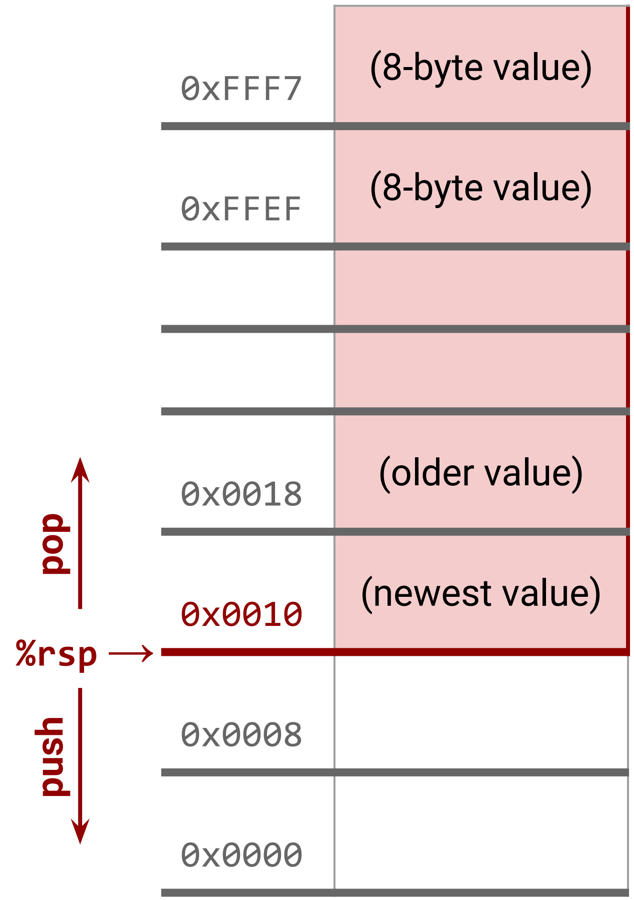
```
subq $8, %rsp
movq %rax, (%rsp)
```

Popping a value

- Read value at address pointed by `%rsp`
- Increment `%rsp`

Example: `popq %rax` is equivalent to

```
movq (%rsp), %rax
addq $8, %rsp
```



Note: Any stack element can be accessed with `%rsp`

Passing Control

Must save return address

- A function can be called from many points in the program.
- Recursive invocations are also possible.
- Where to return to?
  - A fixed return jump would not work: single return point.
  - Return address in a register: would be overwritten by nested calls.

Solution: use the stack!

- Last-In First-Out (LIFO) policy: pass control to the most recent caller.

- `callq label` is (more or less) equivalent to:

```
pushq %rip
jmp   label
```

- `retq` is (more or less) equivalent to:

```
popq %rip
```

Passing Control: Disassembling

```c
#include <stdio.h>

int sum(int x, int y, int *z) {
    return x + y + *z;
}

int main() {
    int z = 10;
    printf("%d\n", sum(1, 5, &z));
    return 0;
}
```

```
sum:
        addl   %esi, %edi
        movl   %edi, %eax
        addl   (%rdx), %eax
        ret
.LC0:
        .string     "%d\n"
```

```
main:
        subq   $24, %rsp
        movl   $10, 12(%rsp)
        leaq   12(%rsp), %rdx
        movl   $5, %esi
        movl   $1, %edi
        call   sum
        movl   %eax, %esi
        leaq   .LC0(%rip), %rdi
        movl   $0, %eax
        call   printf@PLT
        movl   $0, %eax
        addq   $24, %rsp
        ret
```

Passing Parameters

Conventions

- First six integer/pointer arguments on **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%rcx`**, **`%r8`**, **`%r9`**
- Additional ones are pushed on the stack in **reverse order** as **8-byte words**.
- The caller must also **remove** parameters from stack after the call.
- Parameters **may be modified** by the called function.

Accessing stack parameters

- At the beginning of a function, `%rsp` points to the return address.
- Stack parameters can be addressed as: `8(%rsp)`, `16(%rsp)`, ...

It is common practice to:

- Backup the initial value of `%rbp` (used by the caller): `pushq %rbp`
- Write `%rsp` (the current stack pointer) into `%rbp`: `movq %rsp, %rbp`
- Use `%rbp` to access parameters on the stack: `16(%rbp)` is the 7th param
- Restore the previous `%rbp` value at the end of the function: `popq %rbp`

(GCC optimizations avoid this use of `%rbp`, allowing its use as general register.)

Passing Parameters: Disassembling

```c
#include <stdio.h>

int sum(int x1, int x2, int x3,
    int x4, int x5, int x6, int x7) {
    return x1 + x2 + x3 + x4 +
           x5 + x6 + x7;
}

int main() {
    printf("%d\n",
           sum(1, 2, 3, 4, 5, 6, 7));
    return 0;
}
```

```
sum:
        addl   %esi, %edi
        addl   %edi, %edx
        addl   %edx, %ecx
        addl   %r8d, %ecx
        addl   %r9d, %ecx
        movl   %ecx, %eax
        addl   8(%rsp), %eax
        ret
.LC0:
        .string     "%d\n"
```

```
main:
        subq   $8, %rsp
        pushq  $7
        movl   $6, %r9d
        movl   $5, %r8d
        movl   $4, %ecx
        movl   $3, %edx
        movl   $2, %esi
        movl   $1, %edi
        call   sum
        addq   $8, %rsp
        movl   %eax, %esi
        leaq   .LC0(%rip), %rdi
        movl   $0, %eax
        call   printf@PLT
        movl   $0, %eax
        addq   $8, %rsp
        ret
```

Return Values and Registers

Return Values

- Integers or pointers: store return value in **`%eax`**
- Floating point: store return value in a floating-point register

Registers

- The caller must assume that **`%rax`**, **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%rcx`**, **`%r8`** to **`%r11`** may be changed by the callee (scratch registers / **caller-save**)
- Arithmetic flags are not preserved by function calls.
- The caller can assume that **`%rbx`**, **`%rbp`**, **`%rsp`**, and **`%r12`** to **`%r15`** will not change during function call.
    - The callee must save and restore them if necessary (**callee-save**).

Local Variables

When to use stack

Local variables must be allocated on the stack when:

- There are not enough registers.
- The address operator “&” is applied to a local variable.
- The variable is an array or a structure.

To allocate (uninitialized) local variables on the stack: `subq $16, %rsp`

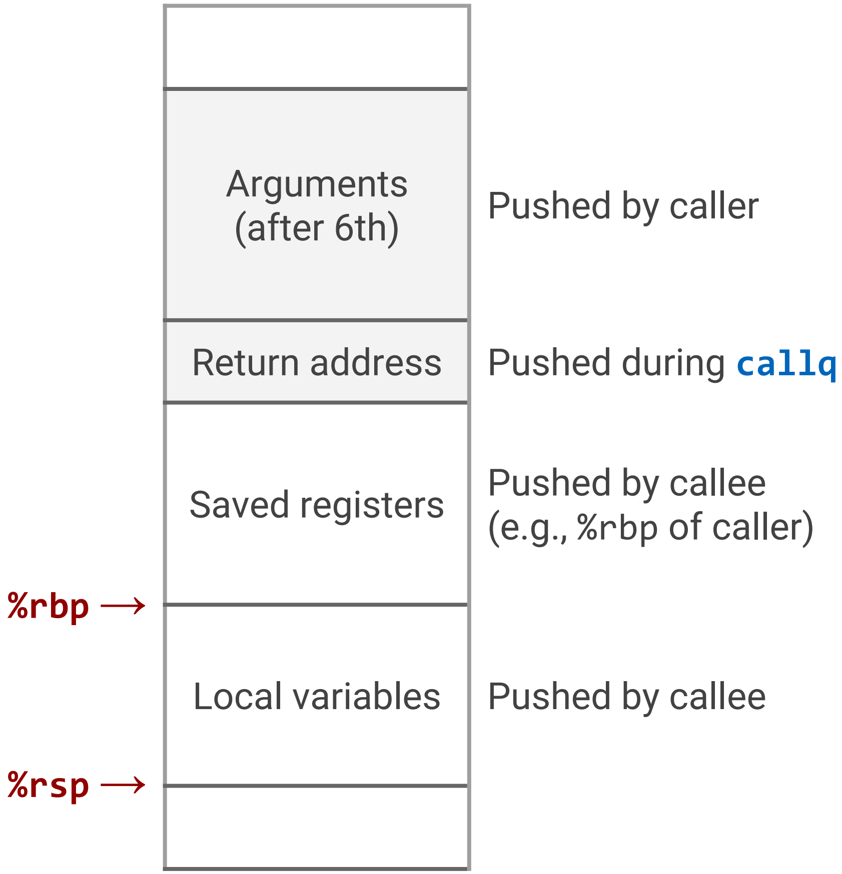
Conventions

- Local variables can be allocated using **any size** (e.g., 1 byte for a char)
- They must be aligned at an address that is a **multiple of their size**.
- The stack pointer **`%rsp` must be a multiple of 16 before calls to functions outside of the current module.**
- The frame pointer `%rbp` is never changed after prologue / before epilogue.
- Local variables must be allocated immediately after callee-save registers.

Putting it all together

1. **The caller prepares and starts the call**
    - Push `%rax`, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` to `%r11` if required after call
    - Save arguments on `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` or into the stack
    - Execute **`callq`** (which pushes `%rip` and jumps to subroutine)
2. **The callee prepares for execution**
    - Push `%rbx`, `%rbp`, and `%r12` to `%r15` if modified during execution.
    - Decrement `%rsp` and allocate local variables on the stack.
3. **The callee runs** (possibly, invoking other functions)
4. **The callee restores the state and returns**
    - Increment `%rsp` to deallocate local variables from the stack.
    - Pop `%rbx`, `%rbp`, `%rsp`, and `%r12` to `%r15` (if pushed)
    - Execute **`retq`** (stores the return address into `%rip`)
5. **The caller restores the state**
    - Increment `%rsp` to deallocate arguments from stack.
    - Pop saved registers from stack.

Putting it all together: stack frames



Arrays in C

When we define `int x[10];` we obtain:

- A block of size `(array size)*(element size) = 10*4` on the stack
- A variable `x` to access elements `0` through `9`
  - `x[9]` gives the 10th element (the last one!)
  - `*(x+9)` is equivalent (pointer arithmetic multiplies by data size)

Expression (x in %rdx, i in %rcx)	**Type**	**Assembly storing expression in** %rax
`x`	`int *`	`movq %rdx, %rax`
`x[0]`	`int`	`movl (%rdx), %eax`
`x[i]`	`int`	`movl (%rdx, %rcx, 4), %eax`
`&x[2]`	`int *`	`leaq 8(%rdx), %rax`
`x+i-1`	`int *`	`leaq -4(%rdx, %rcx, 4), %rax`
`*(x+i-3)`	`int`	`movl -12(%rdx, %rcx, 4), %eax`
`&x[i]-x`	`long`	`movq %rcx, %rax`

Nested Arrays

When we define `int x[10][2];` in a C program, we obtain:

- A block of size `(size1)*(size2)*(element size) = 10*2*4` on the stack
- A variable name `x` to access elements `0` through **19**
  - `x[0][0]` gives the 1st element (at memory address x)
  - `x[9][1]` gives the 20th element (the last one)
  - `x[i][j]` gives the element at address **x + (i*2 + j)*(element size)**
  - `*(x+i*2+j)` is equivalent

Data is stored on the stack in **row-major order**:

- First, the 2 elements of row 0, `x[0][0]` and `x[0][1]`
- Then, the 2 elements of row 1, `x[1][0]` and `x[1][1]`
- And so on…

  `x[i][j]` is the element at row `i` and column `j`.

Beware. Arrays are not pointers, but can be used similarly: www.c-faq.com/aryptr

Case study: sum over array

```c
#include <stdio.h>

int sum(int *a, int n) {
    int total = 0;

    for (int i = 0; i < n; i++) {
      total += a[i];
    }

    return total;
}

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    printf("%d\n", sum(numbers, 5));
    return 0;
}
```

```
sum:
      movl   $0, %edx
      movl   $0, %eax
      jmp    .L2
.L3:
      movslq %edx, %rcx
      addl   (%rdi,%rcx,4), %eax
      addl   $1, %edx
.L2:
      cmpl   %esi, %edx
      jl     .L3
      rep ret
.LC0:
      .string     "%d\n"
```

```
main:
      subq   $40, %rsp
      movl   $1, (%rsp)
      movl   $2, 4(%rsp)
      movl   $3, 8(%rsp)
      movl   $4, 12(%rsp)
      movl   $5, 16(%rsp)
      movq   %rsp, %rdi
      movl   $5, %esi
      call   sum
      movl   %eax, %esi
      leaq   .LC0(%rip), %rdi
      movl   $0, %eax
      call   printf@PLT
      movl   $0, %eax
      addq   $40, %rsp
      ret
```

Case study: compare arrays

```c
#include <stdio.h>

int array_cmp(int *x, int *y, int n) {
    for (int i = 0; i < n; i++) {
      int cmp = x[i]-y[i];
      if (cmp != 0) {
          return cmp;
      }
    }
    return 0;
}

int main() {
    int x[5] = {1, 2, 3, 4, 5};
    int y[5] = {1, 2, 3, 4, 7};
    printf("%d\n", array_cmp(x, y, 5));
    return 0;
}
```

```
array_cmp:
      movl   $0, %ecx
.L2:
      cmpl   %edx, %ecx
      jge    .L5
      movslq %ecx, %r8
      movl   (%rdi,%r8,4), %eax
      subl   (%rsi,%r8,4), %eax
      jne    .L1
      addl   $1, %ecx
      jmp    .L2
.L5:
      movl   $0, %eax
.L1:
      rep ret
.LC0:
      .string     "%d\n"
```

```
main:
      subq   $72, %rsp
      movl   $1, 32(%rsp)
      movl   $2, 36(%rsp)
      movl   $3, 40(%rsp)
      movl   $4, 44(%rsp)
      movl   $5, 48(%rsp)
      movl   $1, (%rsp)
      movl   $2, 4(%rsp)
      movl   $3, 8(%rsp)
      movl   $4, 12(%rsp)
      movl   $7, 16(%rsp)
      movq   %rsp, %rsi
      leaq   32(%rsp), %rdi
      movl   $5, %edx
      call   array_cmp
      movl   %eax, %esi
      leaq   .LC0(%rip), %rdi
      movl   $0, %eax
      call   printf@PLT
      movl   $0, %eax
      addq   $72, %rsp
      ret
```

Case study: row-column product

```c
#include <stdio.h>
#define N 3
typedef int matrix[N][N];
static int matmul(matrix x, matrix y,
                  int i, int k) {
    int result = 0;
    for (int j = 0; j < N; j++) {
       result += x[i][j]*y[j][k];
    }
    return result;
}

int main() {
    int x[N][N] = {{1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9}};

    int y[N][N] = {{3, 0, 1},
                   {4, 2, 8},
                   {0, 1, 7}};

    printf("%d\n", matmul(x, y, 0, 1));
    return 0;
}
```

```
matmul:
       movl   $0, %r10d
       movl   $0, %eax
       cmpl   $2, %r10d
       jg     .L7
       pushq %rbx
.L3:   movslq %edx, %r8
       leaq   (%r8,%r8,2), %r9
       leaq   0(,%r9,4), %r8
       addq   %rdi, %r8
       movslq %r10d, %r11
       leaq   (%r11,%r11,2), %rbx
       leaq   0(,%rbx,4), %r9
       addq   %rsi, %r9
       movslq %ecx, %rbx
       movl   (%r9,%rbx,4), %r9d
       imull (%r8,%r11,4), %r9d
       addl   %r9d, %eax
       addl   $1, %r10d
       cmpl   $2, %r10d
       jle    .L3
       popq   %rbx
       ret
.L7:   ret
```

```
main:
       subq   $104, %rsp
       movl   $1, 48(%rsp)
       [...]
       movl   $9, 80(%rsp)
       movl   $3, (%rsp)
       [...]
       movl   $7, 32(%rsp)
       movq   %rsp, %rsi
       leaq   48(%rsp), %rdi
       movl   $1, %ecx
       movl   $0, %edx
       call   matmul
       movl   %eax, %esi
       leaq   .LC0(%rip), %rdi
       movl   $0, %eax
       call   printf@PLT
       movl   $0, %eax
       addq   $104, %rsp
       ret
.LC0:
       .string     "%d\n"
```