Last Lecture

- Function call mechanisms: passing control and data, managing local data on memory (stack)
- Memory layout
  - Stack (local variables …)
  - Heap (dynamically allocated data)
  - Data (statically allocated data)
  - Text / Shared Libraries (program code)
- Stack is the right data structure for function call / return
  - If `multstore` calls `mult2`, then `mult2` returns before `multstore` returns
- x86-64 stack register and instructions: stack pointer `rsp`, `push` and `pop`
- x86-64 function call instructions: `call` and `ret`
Today’s Menu

- Introduction
  - C program -> assembly code -> machine level code
  - Assembly language basics: data, move operation
    - Memory addressing modes
- Operation leaq and Arithmetic & logical operations
- Conditional Statement – Condition Code + cmovX
- Loops
- Function call – Stack – Recursion
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Floating-point operations
Passing data mechanism – using stack x86-64 function call convention (ABI)

1. **Caller** and **callee** must obey *function call convention* when passing data during function call

   - **Caller:**
     - Before calling a function, must copy the first 6 arguments (integers/memory addresses) into registers `%rdi, %rsi, %rdx, %rcx, %r8, %r9`, respectively

   - **Callee:**
     - Before returning to caller, must copy returned value into `%rax`
Passing data mechanism – Example of passing arguments in registers

```c
#define N 4
// Global arrays A and C
char A[N][N] = { ... };
char C[N][N];

void main() {
    ...
    copy(A, C, N);
    printMatrixByRow(A, N);
    printMatrixByRow(C, N);
    transpose(C, N);
    reverseColumns(C, N);
    ...
    return;
}
```

```assembly
main:
    ...
    movl $4, %edx
    movl $C, %esi
    movl $A, %edi
    call copy
    movl $4, %esi
    movl $A, %edi // optional
    call printMatrixByRow
    movl $4, %esi // optional
    movl $C, %edi
    call printMatrixByRow
    movl $4, %esi // optional
    movl $C, %edi // optional
    call transpose
    movl $4, %esi // optional
    movl $C, %edi // optional
    call reverseColumns
    ...
```
Passing data mechanism – using stack x86-64 function call convention (ABI)

1. **Caller** and **callee** must obey *function call convention* when passing data during function call

   - **Caller:**
     - Before calling a function, must copy the first 6 arguments (integers/memory addresses) into registers \%rdi, \%rsi, \%rdx, \%rcx, \%r8, \%r9, respectively
     - Then must push the rest of the arguments on the stack in reverse order

   - **Callee:**
     - Before returning to caller, must copy returned value into \%rax
Passing data mechanism – using stack x86-64 function call convention (ABI)

2. When passing data that is a memory address (i.e., a pointer) during function call

- **Caller:**
  - Must make use of the stack in order to create such memory address
Passing data mechanism
x86-64 function call convention

Registers

- First 6 arguments

%rdi
%rsi
%rdx
%rcx
%r8
%r9

- Return value

%rax

Stack

Why in reverse order?

Each argument is stored in a quad word (8 bytes) on the stack – even if the argument is a char (1 byte), a short (2 bytes) or an int (4 bytes).

Why is that?
Stack frame

- Stack-based languages -> languages that support recursion such as C, Pascal, Java
- Because the code must be “reentrant”, i.e., it must allow multiple simultaneous invocations (executions) of a single function
- The state (i.e., arguments (if any), local variables (if any), return address) of each of these invocations (executions) must be stored somewhere
- That “somewhere” is in a stack frame on the stack
- One stack frame per function call
Content of stack frame and its ordering

Stack

- callee saved regs
- local vars
- caller saved regs
- args 7 ... n
- return address

M[]

%rsp

Increasing memory addresses

stack frame
Stack Frame Management

- If **callee** requires space on the stack frame, this space is allocated as follows:
  - Set-up code:
    - Example: `subq $72, %rsp`
  - During **callee’s** execution, this space on the stack frame is used as follows:
    - Example: `leaq, mov*`
  - Just before **callee** terminates (before the `ret` instruction is executed), this space is deallocated as follows:
    - Clean-up code:
      - Example: `addq $72, %rsp`
Passing data mechanism – Examples of local variables, arguments and pointer on the stack

```
long call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:
```
    subq   $40, %rsp
    movq   $1, 32(%rsp)
    movl   $2, 28(%rsp)
    movw   $3, 26(%rsp)
    movb   $4, 25(%rsp)
    movq   32(%rsp), %rdi
    movl   28(%rsp), %edx
    leaq   25(%rsp), %rax
    movq   %rax, 8(%rsp)
    movl   $4, (%rsp)
    leaq   32(%rsp), %rsi
    leaq   28(%rsp), %rcx
    leaq   26(%rsp), %r9
    movl   $3, %r8d
    callq proc ...
```
<table>
<thead>
<tr>
<th>base + displacement</th>
<th>Stack Variables</th>
<th>Purpose</th>
</tr>
</thead>
</table>

Register Table:
Summary

- Passing data mechanism – using stack
- x86-64 function call convention (ABI):
  - Before calling a function, **caller** must copy the first 6 (integral) arguments into registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, respectively
  - Then it must push the rest of the arguments on the stack in reverse order
  - When passing data that is a memory address (i.e., a pointer) during function call, **caller** must make use of the stack in order to create such memory address
  - Before returning to **caller**, **callee** must copy returned value into `%rax`
- Stack frame content
Next Lecture

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, move operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack – Recursion
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Floating-point operations