Goals of this Lecture

- Help you learn:
  - Function call problems
  - x86-64 solutions
  - Pertinent instructions and conventions

Assembly Language: Function Calls

Princeton University
Computer Science 217: Introduction to Programming Systems

Function Call Problems

1. Calling and returning
   - How does caller function jump to callee function?
   - How does callee function jump back to the right place in caller function?
2. Passing arguments
   - How does caller function pass arguments to callee function?
   - Where does callee function store local variables?
3. Storing local variables
   - How does callee function store local variables?
4. Returning a value
   - How does callee function send return value back to caller function?
   - How does caller function access the return value?
5. Optimization
   - How do caller and callee function minimize memory access?

Function Call Problems

Princeton University
Computer Science 217: Introduction to Programming Systems

Running Example

- Calling and returning
- Passing arguments
- Storing local variables
- Returning a value
- Optimization

Problem 1: Calling and Returning

How does caller jump to callee?
- I.e., Jump to the address of the callee’s first instruction

How does the callee jump back to the right place in caller?
- I.e., Jump to the instruction immediately following the most-recently-executed call instruction

Agenda

- Calling and returning
- Passing arguments
- Storing local variables
- Returning a value
- Optimization

Running Example

Calls standard C `abs()` function

```
long absadd(long a, long b)
{
  long ab=abs(a);
  long ba=abs(b);
  long sum = ab + ba;
  return sum;
}
```

Problem 1: Calling and Returning

```
… absadd(3L, -4L);
…
```
Problem: callee may be called by multiple callers

```
f1:
    jmp g  # Call g
f1ReturnPoint:
    ...  

g:
    ...  
    jmp ???  # Return

f2:
    jmp g  # Call g
f2ReturnPoint:
    ...  

g:
    ...  
    jmp fReturnPoint  # Return
```

Problem: Cannot handle nested function calls

```
f:
    movq $fReturnPoint, %rax
    jmp g  # Call g
fReturnPoint:
    ...  

g:
    movq $gReturnPoint, %rax
    jmp h  # Call h
gReturnPoint:
    ...  
    jmp *%rax  # Return

h:
    ...  
    ret
```

Observations:
- May need to store many return addresses
  - The number of nested function calls is not known in advance
  - A return address must be saved for as long as the invocation of this function is live, and discarded thereafter
  - Stored return addresses are destroyed in reverse order of creation
    - f() calls g() ⇒ return addr for g is stored
    - g() calls h() ⇒ return addr for h is stored
    - h() returns to g() ⇒ return addr for g is destroyed
    - g() returns to f() ⇒ return addr for g is destroyed
  - LIFO data structure (stack) is appropriate

x86-64 solution:
- Use the STACK section of memory
- Via call and ret instructions

Special form of jmp instruction
Implementation of call

RSP (stack pointer) register points to top of stack

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq src</td>
<td>subq $8, %rsp</td>
</tr>
<tr>
<td></td>
<td>movq src, (%rsp)</td>
</tr>
<tr>
<td>popq dest</td>
<td>movq (%rsp), dest</td>
</tr>
<tr>
<td></td>
<td>addq $8, %rsp</td>
</tr>
<tr>
<td>call addr</td>
<td>pushq %rip</td>
</tr>
<tr>
<td></td>
<td>jmp addr</td>
</tr>
</tbody>
</table>

RIP (instruction pointer) register points to next instruction to be executed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq src</td>
<td>subq $8, %rsp</td>
</tr>
<tr>
<td></td>
<td>movq src, (%rsp)</td>
</tr>
<tr>
<td>popq dest</td>
<td>movq (%rsp), dest</td>
</tr>
<tr>
<td></td>
<td>addq $8, %rsp</td>
</tr>
<tr>
<td>call addr</td>
<td>pushq %rip</td>
</tr>
<tr>
<td></td>
<td>jmp addr</td>
</tr>
</tbody>
</table>

Note: Can't really access RIP directly, but this is implicitly what call is doing

call instruction pushes return addr (old RIP) onto stack, then jumps

Implementation of ret

RSP before ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effective Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq src</td>
<td>subq $8, %rsp</td>
</tr>
<tr>
<td></td>
<td>movq src, (%rsp)</td>
</tr>
<tr>
<td>popq dest</td>
<td>movq (%rsp), dest</td>
</tr>
<tr>
<td></td>
<td>addq $8, %rsp</td>
</tr>
<tr>
<td>call addr</td>
<td>pushq %rip</td>
</tr>
<tr>
<td></td>
<td>jmp addr</td>
</tr>
</tbody>
</table>

RSP after ret

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effective Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq src</td>
<td>subq $8, %rsp</td>
</tr>
<tr>
<td></td>
<td>movq src, (%rsp)</td>
</tr>
<tr>
<td>popq dest</td>
<td>movq (%rsp), dest</td>
</tr>
<tr>
<td></td>
<td>addq $8, %rsp</td>
</tr>
<tr>
<td>call addr</td>
<td>pushq %rip</td>
</tr>
<tr>
<td></td>
<td>jmp addr</td>
</tr>
</tbody>
</table>

Note: can't really access RIP directly, but this is implicitly what ret is doing

ret instruction pops stack, thus placing return addr (old RIP) into RIP

Running Example

```
# long absadd(long a, long b)
absadd:
    # long absA, absB, sum
    # absA = labs(a)
    # call labs
    # absB = labs(b)
    # call labs
    # sum = absA + absB
    # return sum
    ret
```
### Agenda

- Calling and returning
- Passing arguments
- Storing local variables
- Returning a value
- Optimization

### Problem 2: Passing Arguments

**Problem:**
- How does caller pass arguments to callee?
- How does callee accept parameters from caller?

```c
long absadd(long a, long b)
{
    long absA, absB, sum;
    absA = labs(a);
    absB = labs(b);
    sum = absA + absB;
    return sum;
}
```

### X86-64 Solution 1: Use the Stack

**Observations (déjà vu):**
- May need to store many arg sets
- The number of arg sets is not known in advance
- Arg set must be saved for as long as the invocation of this function is live, and discarded thereafter
- Stored arg sets are destroyed in reverse order of creation
- LIFO data structure (stack) is appropriate

### Running Example

```c
long absadd(long a, long b)
{
    pushq %rdi
    pushq %rsi
    # long absA, absB, sum
    ...# absA = labs(a)
    movq 8(%rsp), %rdi
    call labs...# absB = labs(b)
    movq 0(%rsp), %rdi
    call labs...
    # sum = absA + absB...
    addq $16, %rsp
    ret
}
```

### x86-64 Solution: Use the Stack

**x86-64 solution:**
- Pass first 6 (integer or address) arguments in registers
  - RDI, RSI, RDX, RCX, R8, R9
- More than 6 arguments ⇒
  - Pass arguments 7, 8, ... on the stack
  - (Beyond scope of COS 217)
- Arguments are structures ⇒
  - Pass arguments on the stack
  - (Beyond scope of COS 217)

**Callee function then saves arguments to stack**
- Or maybe not!
  - See “optimization” later this lecture
  - Callee accesses arguments as positive offsets vs. RSP
Problem 3: Storing Local Variables

Where does callee function store its local variables?

long absadd(long a, long b)
{
    long absA, absB, sum;
    absA = labs(a);
    absB = labs(b);
    sum = absA + absB;
    return sum;
}

Problem 4: Return Values

Problem:
- How does callee function send return value back to caller function?
- How does caller function access return value?

x86-64 Solution: Use RAX

In principle
- Store return value in stack frame of caller

Or, for efficiency
- Known small size ⇒ store return value in register
- Other ⇒ store return value in stack

x86-64 convention
- Integer or address:
  - Store return value in RAX
- Floating-point number:
  - Store return value in floating-point register
    - (Beyond scope of COS 217)
- Structure:
  - Store return value on stack
    - (Beyond scope of COS 217)
Running Example

```
# long absadd(long a, long b)
absadd:
pushq %r13 # Save R13, use for absA
pushq %r14 # Save R14, use for absB
pushq %r15 # Save R15, use for sum
...
```

Agenda

- Calling and returning
- Passing arguments
- Storing local variables
- Returning a value
- Optimization

Problem 5: Optimization

Observation: Accessing memory is expensive
- More expensive than accessing registers
- For efficiency, want to store parameters and local variables in registers (and not in memory) when possible

Observation: Registers are a finite resource
- In principle: Each function should have its own registers
- In reality: All functions share same small set of registers

Problem: How do caller and callee use same set of registers without interference?
- Call may use register that the caller also is using
- When callee returns control to caller, old register contents may have been lost
- Caller function cannot continue where it left off

x86-64 Solution: Register Conventions

Callee-save registers
- RBX, RBP, R12, R13, R14, R15
- Callee function must preserve contents
- If necessary...
  - Callee saves to stack near beginning
  - Callee restores from stack near end

Caller-save registers
- RDI, RSI, RDX, RCX, R8, R9, RAX, R10, R11
- Callee function can change contents
- If necessary...
  - Caller saves to stack before call
  - Caller restores from stack after call

Running Example

Local variable handling in unoptimized version:
- At beginning, absadd() allocates space for local variables (absA, absB, sum) in stack
- Body of absadd() uses stack
- At end, absadd() pops local variables from stack

Local variable handling in optimized version:
- absadd() keeps local variables in R13, R14, R15
- Body of absadd() uses R13, R14, R15
- Must be careful:
  - absadd() cannot change contents of R13, R14, or R15
  - So absadd() must save R13, R14, and R15 near beginning, and restore near end

Running Example

absadd() stores local vars in R13, R14, R15, not in memory
absadd() cannot destroy contents of R13, R14, R15
So absadd() must save R13, R14, R15 near beginning and restore near end
Running Example

Parameter handling in unoptimized version:
- `absadd()` accepts parameters (a and b) in RDI and RSI
- At beginning, `absadd()` copies contents of RDI and RSI to stack
- Body of `absadd()` uses stack
- At end, `absadd()` pops parameters from stack

Parameter handling in optimized version:
- `absadd()` accepts parameters (a and b) in RDI and RSI
- Body of `absadd()` uses RDI and RSI
- Must be careful:
  - Call of `labs()` could change contents of RDI and/or RSI
  - `absadd()` must save contents of RSI (value of b) across 1st call of `labs()`, and restore contents after call

Non-Optimized vs. Optimized Patterns

Unoptimized pattern
- Parameters and local variables strictly in memory (stack) during function execution
- Pro: Always possible
- Con: Inefficient
- gcc compiler uses when invoked without --O option

Optimized pattern
- Parameters and local variables strictly in registers during function execution
- Pro: Efficient
- Con: Sometimes impossible
- More than 6 local variables
- Local variable is a structure or array
- Function computes address of parameter or local variable
- gcc compiler uses when invoked with --O option, when it can!

Summary

Function calls in x86-64 assembly language
Calling and returning
- `call` instruction pushes RIP onto stack and jumps
- `ret` instruction pops from stack to RIP

Passing arguments
- Caller copies args to caller-saved registers (in prescribed order)
- Unoptimized pattern:
  - Callee pushes args to stack
  - Callee uses args as positive offsets from RSP
  - Callee pops args from stack
- Optimized pattern:
  - Callee keeps args in callee-saved registers
  - Be careful!

Storing local variables
- Unoptimized pattern:
  - Callee pushes local vars onto stack
  - Callee uses local vars as positive offsets from RSI
  - Callee pops local vars from stack
- Optimized pattern:
  - Callee keeps local vars in callee-saved registers
  - Be careful!

Returning values
- Callee places return value in RAX
- Callee accesses return value in RAX

Summary (cont.)

Hybrid Patterns
Hybrids are possible
- Example
  - Parameters in registers
  - Local variables in memory (stack)

Hybrids are error prone for humans
- Example (continued from previous)
  - Step 1: Access local variable < local var is at stack offset X
  - Step 2: Push caller-save register
  - Step 3: Access local variable < local var is at stack offset X+8!!!
  - Step 4: Call `labs()`
  - Step 6: Access local variable < local var is at stack offset X+8!!!
  - Step 7: Pop caller-save register
  - Step 8: Access local variable < local var is at stack offset X

Avoid hybrids for Assignment 4
struct tree {
    int key;
    struct tree *left;
    struct tree *right;
};

int sum (struct tree *t) {
    if (t==NULL)
        return 0;
    else return t->key +
          sum(t->left) +
          sum(t->right);
}

This would make an excellent exam question...