Lecture 7: Procedure Calls in Assembly

CS 105

February 12, 2020
Assembly/Machine Code View

Programmer-Visible State
- PC: Program counter
- 16 Registers
- Condition codes

Memory
- Byte addressable array
- Code and user data
- Stack to support procedures
Assembly Characteristics: Operations

• Transfer data between memory and register
  • Load data from memory into register
  • Store register data into memory

• Perform arithmetic function on register or memory data

• Transfer control
  • Conditional branches
  • Jumps to/from procedures
Procedures

• Procedures provide an abstraction that implements some functionality with designated arguments and (optional) return value
  • e.g., functions, methods, subroutines, handlers

• To support procedures at the machine level, we need mechanisms for:
  1) **Passing Control**: When procedure P calls procedure Q, program counter must be set to address of Q, when Q returns, program counter must be reset to instruction in P following procedure call
  2) **Passing Data**: Must handle parameters and return values
  3) **Allocating memory**: Q must be able to allocate (and deallocate) space for local variables
The Stack

• the stack is a region of memory (traditionally the "top" of memory)

• grows "down"

• provides storage for functions (i.e., space for allocating local variables)

• `%rsp` holds address of top element of stack
Modifying the Stack

- \textbf{pushq }S:\n  \text{R}[\%rsp] \leftarrow \text{R}[\%rsp] - 8
  \text{M}[\text{R}[\%rsp]] \leftarrow S

- \textbf{popq }D:\n  D \leftarrow \text{M}[\text{R}[\%rsp]]
  \text{R}[\%rsp] \leftarrow \text{R}[\%rsp] + 8

- \textbf{explicitly modify }\%\text{rsp}:\n  \text{subq }$4, \%\text{rsp}
  \text{addq }$4, \%\text{rsp}

- \textbf{modify memory above }\%\text{rsp}:\n  \text{movl }$47, 4(\%\text{rsp})

- \textbf{call/return from procedure}:\n  \text{call function}
  \text{ret}
### X86-64 Register Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>(function result)</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td></td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>(fourth argument)</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>(third argument)</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>(second argument)</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>(first argument)</td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td>(stack pointer)</td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td></td>
</tr>
<tr>
<td><code>%r8</code></td>
<td>(fifth argument)</td>
</tr>
<tr>
<td><code>%r9</code></td>
<td>(sixth argument)</td>
</tr>
<tr>
<td><code>%r10</code></td>
<td></td>
</tr>
<tr>
<td><code>%r11</code></td>
<td></td>
</tr>
<tr>
<td><code>%r12</code></td>
<td></td>
</tr>
<tr>
<td><code>%r13</code></td>
<td></td>
</tr>
<tr>
<td><code>%r14</code></td>
<td></td>
</tr>
<tr>
<td><code>%r15</code></td>
<td></td>
</tr>
</tbody>
</table>

Callee-saved registers are shaded
Procedure Calls, Division of Labor

**Caller**

- Before
  - Save registers, if necessary
  - Put arguments in place
  - Make call

- After
  - Restore registers, if necessary
  - Use result

**Callee**

- Preamble
  - Save registers, if necessary
  - Allocate space on stack

- Exit code
  - Put return value in place
  - Restore registers, if necessary
  - Deallocate space on stack
  - Return
Stack Frames

• Each function called gets a stack frame

• Passing data:
  • calling procedure P uses registers (and stack) to provide parameters to Q.
  • Q uses register %rax for return value

• Passing control:
  • call <proc>
    • Pushes return address (current %rip) onto stack
    • Sets %rip to first instruction of proc
  • ret
    • Pops return address from stack and places it in %rip

• Local storage:
  • allocate space on the stack by decrementing stack pointer, deallocate by incrementing

Procedure Call Example: Stack Frame

```c
int proc(int *p);

int example1(int x) {
    int a[4];
    a[3] = 10;
    return proc(a);
}

example1:
    subq $16, %rsp
    movl $10, 12(%rsp)
    movq %rsp, %rdi
    call 0x400546 <proc>
    addq $16, %rsp
    ret
```
Procedure Call Example: Arguments

```c
int func1(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8) {
    int l1 = x1 + x2;
    int l2 = x3 + x4;
    int l3 = x5 + x6;
    int l4 = x7 + x8;
    int l5 = 4;
    int l6 = 13;
    int l7 = 47;
    int l8 = l1 + l2 + l3 + l4 + l5 + l6 + l7;
    return l8;
}

int main(int argc, char *argv[]){
    int x = func1(1, 2, 3, 4, 5, 6, 7, 8);
    return x;
}
```
Exercise

0x400540 <last>:

400540: 48 89 f8                   mov %rdi, %rax   L1
400543: 48 0f af c6                 imul %rsi, %rax   L2
400547: c3                           ret     L3

0x400548 <first>:

400548: 48 8d 77 01                 lea 0x1(%rdi),%rsi F1
40054c: 48 83 ef 01                 sub $0x1, %rdi   F2
400550: e8 eb ff ff ff ff           callq 400540 <last> F3
400555: f3 c2                       rep; ret   F4

0x400556 <main>:

...                                     
400560: e8 e3 ff ff                   callq 400548 <first> M1
400565: 48 89 c2                   mov %rax, %rdx   M2
...                                      

Recursion

• Handled Without Special Consideration
  • Stack frames mean that each function call has private storage
    • Saved registers & local variables
    • Saved return pointer
  • Register saving conventions prevent one function call from corrupting another’s data
    • Unless the C code explicitly does so (more later!)
  • Stack discipline follows call / return pattern
    • If P calls Q, then Q returns before P
    • Last-In, First-Out

• Also works for mutual recursion
  • P calls Q; Q calls P
Recursive Function

/* Recursive bitcount */
long bitcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + bitcount_r(x >> 1);
}

What is in the stack frame?

bitcount_r:
    testq %rdi, %rdi
    je .L3
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call bitcount_r
    addq %rbx, %rax
    jmp .L2
.L3: # Base Case
    movl $0, %eax
    ret
.L2:
    popq %rbx
    ret