Machine-Level Programming III: Procedures

CSci 2021: Machine Architecture and Organization
Lectures #11-12, February 12th-15th, 2016
Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O’Hallaron

Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
    - Illustration of Recursion

x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %rsp contains lowest in-use stack address
  - address of “top” element

x86-64 Stack: Push

- pushq Src
  - Fetch operand at Src
  - Decrement %rsp by 8
  - Write operand at address given by %rsp

x86-64 Stack: Pop

- popq Dest
  - Read value at address given by %rsp
  - Increment %rsp by 8
  - Store value at Dest (must be register)
Today

- Procedures
  - Stack Structure
  - Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
  - Illustration of Recursion

Code Examples

```c
long mult2(long a, long b)
{
    long s = a * b;
    return s;
}

void multstore(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: call label
  - Push return address on stack
  - Jump to label
- Return address:
  - Address of the next instruction right after call
  - Example from disassembly
- Procedure return: ret
  - Pop address from stack
  - Jump to address

Control Flow Examples

1. Call and return operations

2. Memory usage during function calls

3. Stack usage for function calls
Control Flow Example #4

```
0000000000400550 <multstore>:

• 400544: callq 400550 <mult2>
  400549: mov %rdi,(%rbx)

0000000000400550 <mult2>:

• 400544: mov %rdi,%rax
  400547: imul %rsi,%rax
  400550: retq
```

Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
  - Managing local data
  - Illustrations of Recursion & Pointers

Procedure Data Flow

<table>
<thead>
<tr>
<th>Registers</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>***</td>
</tr>
<tr>
<td>%rax</td>
<td>Arg n</td>
</tr>
<tr>
<td>%rdx</td>
<td>***</td>
</tr>
<tr>
<td>%rcx</td>
<td>Arg B</td>
</tr>
<tr>
<td>%r8</td>
<td>Arg ?</td>
</tr>
</tbody>
</table>

- First 6 arguments
- Only allocate stack space when needed
- Return value

Data Flow Examples

```
long mult2 (long a, long b)
{ long s = a * b;
  return s;
}
```

```
void multstore (long x, long y, long *dest)
{ long t = mult2(x, y);
  *dest = t;
}
```

Stack-Based Languages

- Languages that support recursion
  - e.g., C, Pascal, Java
  - Code must be "reentrant"
    - Multiple simultaneous instantiations of single procedure
    - Need some place to store state of each instantiation
      - Arguments
      - Local variables
      - Return pointer

- Stack discipline
  - State for given procedure needed for limited time
  - From when called to when return
    - Caller returns before caller does

- Stack allocated in Frames
  - State for single procedure instantiation
**Call Chain Example**

Example Call Chain

```
who (...) {
  ...
  who ();
  ...
  amI ();
  ...
  ...
}
```

Procedure `amI()` is recursive

**Stack Frames**

- **Contents**
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)

- **Management**
  - Space allocated when enter procedure
    - “Set-up” code, also called “prolog”
    - Includes push by `call` instruction
  - Deallocated when return
    - “Finish” code, also called “epilog”
    - Includes pop by `ret` instruction

**Stack “Top”**

Previous Frame

```
Frame Pointer: %rbp
```

Frame for `proo`

```
Stack Pointer: %rsp
```

**Contents**

- Return information
- Local storage (if needed)
- Temporary space (if needed)

**Management**

- Space allocated when enter procedure
  - “Set-up” code, also called “prolog”
  - Includes push by `call` instruction
- Deallocated when return
  - “Finish” code, also called “epilog”
  - Includes pop by `ret` instruction
Example

```
void f() {
    int x = 4;
    int y = x + 1;
    return y;
}
```

Stack

```
%rbp %rax %rsp
```

Example: incr

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

Register | Use(s) |
--- | --- |
%rdi | Argument p |
%rsi | Argument val, y |
%rax | x, Return value |

Example: Calling incr #1

```
long call_incr() {  
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Stack Structure

```
| Rtn address | %rax |
--- | --- |
15213 |  
```

Register | Use(s) |
--- | --- |
%rdi | &v1 |
%rsi | 3000 |

Example: Calling incr #2

```
long call_incr() {  
    long v1 = 15213;
    long v2 = incr(v1, 3000);
    return v1+v2;
}
```

Stack Structure

```
| Rtn address | %rax |
--- | --- |
15213 |  
```

Register | Use(s) |
--- | --- |
%rdi | v1 |
%rsi | 3000 |

Example: Calling incr #3

```
long call_incr() {  
    long v1 = 15213;
    long v2 = incr(v1, 3000);
    return v1+v2;
}
```

Stack Structure

```
| Rtn address | %rax |
--- | --- |
15213 |  
```

Register | Use(s) |
--- | --- |
%rdi | v1 |
%rsi | 3000 |
**Example: Calling incr #4**

```c
long call_incr() {
    long v1 = 15213;
    return v1+v2;
}
```

**Example: Calling incr #5**

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(v1, 3000);
    return v1+v2;
}
```

**Register Saving Conventions**

- **When procedure yoo calls who:**
  - yoo is the **caller**
  - who is the **callee**
- **Can register be used for temporary storage?**
  - **yoo:**
    - `movq $15213, %rdx`
     - overwrite `%rdx` before call
    - `addq 0(%rax), %rax`
     - overwrite `%rax` before return
    - `ret`
  - **who:**
    - `subq $18213, %rdx`
     - overwrite `%rdx` before call
    - `addq 0(%rax), %rax`
     - overwrite `%rax` before return
    - `ret`

**x86-64 Linux Register Usage #1 (scratch)**

- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- `%rdi`, `%rsi`, `%rdx`
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- `%r10`, `%r11`
  - Caller-saved
  - Can be modified by procedure

**x86-64 Linux Register Usage #2 (preserved)**

- `%rbx`, `%r12`, `%r13`, `%r14`
  - Caller-saved
  - Caller must save & restore
- `%rbp`
  - Caller-saved
  - Caller must save & restore
  - May be used as frame pointer
  - Can mix & match
- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure
**Callee-Saved Example #1**

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(v1, 3000);
    return x+v2;
}
```

**Initial Stack Structure**

- ...  
- Rtn address  
- trap

**Resulting Stack Structure**

- ...  
- Rtn address  
- trap
- Saved rax
- 15213  
- Unused  
- trap+8

**Callee-Saved Example #2**

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(v1, 3000);
    return x+v2;
}
```

**Initial Stack Structure**

- ...  
- Rtn address  
- trap

**Resulting Stack Structure**

- ...  
- Rtn address  
- trap
- Saved rax
- 15213  
- Unused  
- trap

**Today**

- Procedures
  - Stack Structure
  - Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
  - Illustration of Recursion

**Recursive Function**

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

**Recursive Function Terminal Case**

```
Pcount_r:  
  movl  $0, %eax
  testq %rdi, %rdi
  je   .L6
  pushq %rbx
  movq %rdi, %rbx
  addq $1, %rbx
  shrq %rdi
  call pcount_r
  addq %rbx, %rax
  popq %rbx
  .L6:
  rep; ret
```

**Recursive Function Register Save**

```
Pcount_r:  
  movl  $0, %eax
  testq %rdi, %rdi
  je   .L6
  pushq %rbx
  movq %rdi, %rbx
  addq $1, %rbx
  shrq %rdi
  call pcount_r
  addq %rbx, %rax
  popq %rbx
  .L6:
  rep; ret
```

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td>Return value</td>
</tr>
</tbody>
</table>

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rtn address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Saved rax</td>
</tr>
</tbody>
</table>

**Recursion Function**

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```
Recursive Function Call Setup

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

Register | Use(s) | Type   
---------|--------|--------
%rdi     | x >> 1 | Rec. argument
%rbx     | x & 1  | Callee-saved
%rax     |        | Return value

Recursive Function Call

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

Register | Use(s) | Type   
---------|--------|--------
%rbx     | x & 1  | Callee-saved
%rax     |        | Return value

Recursive Function Result

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

Register | Use(s) | Type   
---------|--------|--------
%rbx     | x & 1  | Callee-saved
%rax     |        | Return value

Recursive Function Completion

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

Register | Use(s) | Type   
---------|--------|--------
%rax     |        | Return value

Observations About 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 (e.g., buffer overflow in Lecture 9)
  - 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

Discussion interlude

- Does a recursive function always have to save one or more registers on the stack?
  - If yes, why?
  - If no, what’s an example of a function that doesn’t need to?
- Talk with your neighbors, then put your answer on ChimeIn

https://chimein.cla.umn.edu/course/view/2021
Recursive function examples

- `void loop(void) { loop(); }`

```c
int fact(unsigned n, int prod) {
    if (n == 0)
        return prod;
    else
        return fact(n - 1, n * prod);
}
```

- But if storing a value across a call, the stack is needed
  - If caller-save, need to save because callee will use it
  - If callee-save, need to save caller’s value
  - Changing the calling convention would not help

---

x86-64 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
  - If P calls Q, then Q returns before P

- **Recursion (& mutual recursion) handled by normal calling conventions**
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%rax`

- **Pointers are addresses of values**
  - On stack or global