Machine-Level Programming III: Procedures

Topics
- x86-64 stack discipline
- Register-saving conventions
- Creating pointers to local variables

Mechanisms in Procedures

Passing control
- To beginning of procedure code
- Back to calling point

Passing data
- Procedure arguments
- Return value

Memory management
- Allocate variables during procedure execution
- Deallocate upon return

Mechanisms all implemented with machine instructions
x86-64 procedures use only what’s needed

x86-64 Stack
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %rsp indicates numerically lowest stack address
  - Always holds address of ’top’ element

x86-64 Stack Pushing

Pushing: pushq Src
- Fetch operand at Src
- Decrement %rsp by 8
- Then write operand at address given by %rsp
### x86-64 Stack Popping

**Popping:** `popq Dest`  
- Read memory data at address given by `%rsp`  
- Increment `%rsp` by 8  
- Write to `Dest`

### Stack Operation Examples

- **Push:** `pushq %rax`  
  - `%rax` pushed to `%rsp`  
  - `%rsp` decremented by 8
- **Pop:** `popq %rdx`  
  - `%rdx` read from `%rsp`  
  - `%rsp` incremented by 8

### Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call` or `callq`  
  - `call label`  
  - Push return address onto stack, jump to `label`
- **Return address value:**  
  - Address of instruction just beyond `call`
- **Procedure return:** `ret` or `retq (or rep; ret)`  
  - Pop address (of instruction after corresponding `call`) from stack  
  - Jump to that address

### Control-Flow Example #1

```
0000000000400540 <multstore>:  
  0x100:  mov  %rax,(%rbx)  
        0x108: callq  400550 <mult2>  
  0x108:  pushq %rax  
        0x108:  retq

0000000000400550 <mult2>:  
  0x120:  mov  %rdx,(%rbx)  
  0x120:  retq
```

- `%rbx` contains the address to store the result of `%rax` in
- `%rdx` contains the address to store the result of `%rax` in
- `%rbx` contains the address to store the result of `%rdx` in
- `%rdx` contains the address to store the result of `%rdx` in
Control-Flow Example #2

```
0x0000000000400550 <multstore>:
  •
  400544: callq 400550 <mult2>
  400549: mov %rax, (%rbx)
  •
	trap 0x4005549
	%rip 0x4005550

0x0000000000400540 <multstore>:
  •
  400544: callq 400550 <mult2>
  400549: mov %rax, (%rbx)
  •
	trap 0x4005549
	%rip 0x4005550
```

Control-Flow Example #3

```
0x0000000000400550 <mult2>:
  mov %rdi, %rax
  •
  retq

0x0000000000400540 <multstore>:
  callq 400550 <mult2>
  mov %rax, (%rbx)
  •
	trap 0x4005549
	%rip 0x4005557
```

Control-Flow Example #4

```
0x0000000000400550 <mult2>:
  mov %rdi, %rax
  •
  retq

0x0000000000400540 <multstore>:
  callq 400550 <mult2>
  mov %rax, (%rbx)
  •
	trap 0x4005549
	%rip 0x4005557
```

Procedure Data Flow

```
Registers
First 6 arguments: %rdi, %rax, %rbx, %rcx, %rdx, %r8

Stack:

Arg n

Arg 8

Arg 7

Return value: %rax

Only allocate stack space when needed
```
Stack-Based Languages

Languages That Support Recursion
- E.g., C, Pascal, Java, Python, Racket, Haskell, ...
- 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
- Callee returns before caller does

Stack Allocated in Frames
- State for single procedure instantiation

Call Chain Example

Code Structure

```
yoo(...)
{  *  
  *  who();  
  *  }
```

Call Chain

```
yoo  
  who  
   ami  ami  ami  
```

- Procedure ami is recursive
Stack Frames

Contents

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

Management

- Space allocated when procedure entered
  - "Set-up" code
  - Frame includes push done by call instruction
- Deallocation upon return
  - "Finish" code
  - Includes pop done by ret instruction

Example

```c
yoo(...)
{
  who();
  ami(...);
}
```
x86-64/Linux Stack Frame

Current Stack Frame (“Top” to Bottom)
- “Argument build:”
  Parameters for function about to be called
- Local variables (if can’t keep in registers)
- Saved register context
- Old frame pointer (optional)

Caller Stack Frame
- Return address
- Pushed by call instruction
- Arguments for this call

Example: incr

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

Example: Calling incr #1

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

Example: Calling incr #2

```c
long call_incr()
{
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1 + v2;
}
```
Example: Calling `incr` #3

```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 `x` be used for temporary storage?
- Contents of register `%rdx` overwritten by `who`
- This could be trouble -- something should be done!
  - Need some coordination
Register Saving Conventions

When procedure yoo calls who:
- yoo is the caller
- who is the callee

Can register $x$ be used for temporary storage?

Conventions
- "Caller Saved"
  - Caller saves temporary values in its frame before the call
- "Callee Saved"
  - Callee saves temporary values in its frame before using
  - Callee restores them before returning to caller

x86-64 Linux Register Usage #1

$rax$
- Return value
- Caller-saved
- Can be modified by procedure

$rdi, ..., $r9$
- Arguments (Diane's silk dress)
- Caller-saved
- Can be modified by procedure

$r10, r11$
- Caller-saved
- Can be modified by procedure

Callee-Saved Example #1

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

x86-64 Linux Register Usage #2

$r8, r10, r11$
- Caller-saved
- Can be used as frame pointer or as scratch
- Can mix & match

$rsp$
- Special form of callee save
- Restored to original value upon exit from procedure
Callee-Saved Example #2

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

call_incr2:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $15213, 8(%rsp)
    movl $3000, %esi
    leaq 8(%rsp), %rdi
    call incr
    addq %rbx, %rax
    addq $16, %rsi
    popq %rbx
    ret
```

Resulting Stack Structure

```
+------------------+
| Return address |
+------------------+
| Saved %rbx      |
+------------------+
| 15213           |
+------------------+
| Unused           |
+------------------+
```

Pre-return Stack Structure

```
+------------------+
| Return address   |
+------------------+
```

Recursive Function

```c
/* 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

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

Recursive Function Register Save

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

---

<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) | Type
%rdi     | x      | Argument
%rax     |        | Return value
```c
/* 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

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

### Recursive Function Call

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

### Recursive Function Result

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

### Recursive Function Completion

```
pcount_r:
    movq $0, %eax
testq %rdi, %rdi
    je .L6
pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
call pcount_r
addq %rbx, %rax
popq %rbx
.L6:
    rep; ret
```
Observations About Recursion

- 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 future lecture)
- 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

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