The Hardware/Software Interface
CSE351 Winter 2013

Procedures and Stacks II
x86-64 Procedure Calling Convention

- Doubling of registers makes us less dependent on stack
  - Store argument in registers
  - Store temporary variables in registers

- What do we do if we have too many arguments or too many temporary variables?
# x86-64 64-bit Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
Revisiting swap, IA32 vs. x86-64 versions

**swap:**

```
pushl %ebp
movl %esp,%ebp
pushl %ebx

movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %ebx,(%ecx)

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

**Body**

```
Body
```

**Set Up**

```
Set Up
```

**Finish**

```
Finish
```

**swap (64-bit long ints):**

```
movq (%rdi), %rdx
movq (%rsi), %rax
movq %rax, (%rdi)
movq %rdx, (%rsi)
ret
```

- **Operands passed in registers**
  - First \( (x_p) \) in %rdi, second \( (y_p) \) in %rsi
  - 64-bit pointers

- **No stack operations required (except ret)**

- **Avoiding stack**
  - Can hold all local information in registers
X86-64 procedure call highlights

- Arguments (up to first 6) in registers
  - Faster to get these values from registers than from stack in memory

- Local variables also in registers (if there is room)

- `callq` instruction stores 64-bit return address on stack
  - Address pushed onto stack, decrementing `%rsp` by 8

- No frame pointer
  - All references to stack frame made relative to `%rsp`; eliminates need to update `%ebp`/ `%rbp`, which is now available for general-purpose use

- Functions can access memory up to 128 bytes beyond `%rsp`: the “red zone”
  - Can store some temps on stack without altering `%rsp`

- Registers still designated “caller-saved” or “callee-saved”
x86-64 Stack Frames

- Often (ideally), x86-64 functions need no stack frame at all
  - Just a return address is pushed onto the stack when a function call is made

- A function *does* need a stack frame when it:
  - Has too many local variables to hold in registers
  - Has local variables that are arrays or structs
  - Uses the address-of operator (&) to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Needs to save the state of callee-save registers before modifying them
Example

```c
long int 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);
}
```

```assembly
call_proc:
    subq  $32,%rsp
    movq  $1,16(%rsp)
    movl  $2,24(%rsp)
    movw  $3,28(%rsp)
    movb  $4,31(%rsp)
    ...
```

Return address to caller of call_proc

NB: Details may vary depending on compiler.
Example

```c
long int 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 $32,%rsp
    movq $1,16(%rsp)
    movl $2,24(%rsp)
    movw $3,28(%rsp)
    movb $4,31(%rsp)
    
    Return address to caller of call_proc
    
    | x4 | x3 | x2 |
    |----|----|----|
    |    |  x1 |
    
    %rsp
```
Example

```c
long int 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:
```
    leaq 24(%rsp),%rcx
    leaq 16(%rsp),%rsi
    leaq 31(%rsp),%rax
    movq %rax,8(%rsp)
    movl $4,(%rsp)
    leaq 28(%rsp),%r9
    movl $3,%r8d
    movl $2,%edx
    movq $1,%rdi
    call proc
```

Arguments passed in (in order): rdi, rsi, rdx, rcx, r8, r9
Example

```c
long int 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:
• • •
leaq 24(%rsp),%rcx
leaq 16(%rsp),%rsi
leaq 31(%rsp),%rax
movq %rax,8(%rsp)
movl $4,(%rsp)
leaq 28(%rsp),%r9
movl $3,%r8d
movl $2,%edx
movq $1,%rdi
call proc
• • •

Return address to caller of call_proc

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>x1</td>
</tr>
<tr>
<td>Arg 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arg 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Return address to line after call to proc

%rsp
long int 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);
}

proc(x1, &x1, x2, &x2,
     x3, &x3, x4, &x4);

call_proc:
    ...
    movswl 28(%rsp),%eax
    movsbl 31(%rsp),%edx
    subl %edx,%eax
    cltq
    movslq 24(%rsp),%rdx
    addq 16(%rsp),%rdx
    imulq %rdx,%rax
    addq $32,%rsp
    ret

Return address to caller of call_proc

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Arg 8

Arg 7

%rsp
Example

```c
long int 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:

```
    movswl 28(%rsp),%eax
    movsbl 31(%rsp),%edx
    subl %edx,%eax
    cltq
    movslq 24(%rsp),%rdx
    addq 16(%rsp),%rdx
    imulq %rdx,%rax
    addq $32,%rsp
    ret
```

Return address to caller of call_proc
x86-64 Procedure Summary

■ Heavy use of registers (faster than using stack in memory)
   - Parameter passing
   - More temporaries since more registers

■ Minimal use of stack
   - Sometimes none
   - When needed, allocate/deallocate entire frame at once
   - No more frame pointer: address relative to stack pointer

■ More room for compiler optimizations
   - Prefer to store data in registers rather than memory
   - Minimize modifications to stack pointer