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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

Revisiting swap, IA32 vs. x86-64 versions

- Operands passed in registers
  - First (xp) in %rdi, second (yp) 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”

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:

```assembly
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

%rsp

\[ \text{NB: Details may vary depending on compiler.} \]

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:

```assembly
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

x1 x3 x2

%rsp

x3 x4
Example

```
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

```
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

```
x4  x3  x2
   x1
  Arg 8
  Arg 7
   %rsp
```

Example

```
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

```
x4  x3  x2
   x1
  Arg 8
  Arg 7
   %rsp
```
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