CSCI-UA.0201

Computer Systems Organization

Machine Level – Function Calls

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
Suppose P calls Q

- **Passing control**
  - To beginning of procedure code
  - Back to return point

- **Passing data**
  - Procedure arguments
  - Return value

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

```c
int Q(int i) {
    int t = 3*i;
    int v[10];
    return v[t];
}
```
A quick glimpse at how stack works...
x86-64 Stack

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

Stack Pointer: `%rsp`

Stack “Bottom”

Stack Grows Down

Increasing Addresses

Stack “Top”
x86-64 Stack: Push

- **pushq** *Src*
  - Fetch operand at *Src*
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`
popq  Dest
- Read value at address given by %rsp
- Increment %rsp by 8
- Store value at Dest (must be register)
Examples:

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

```assembly
00000000000400540 <multstore>:
   400540: push %rbx            # Save %rbx
   400541: movq %rdx,%rbx       # Save dest
   400544: callq 400550 <mult2>  # mult2(x,y)
   400549: movq %rax,(%rbx)     # Save at dest
   40054c: popq %rbx            # Restore %rbx
   40054d: retq                 # Return
```

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

```assembly
00000000000400550 <mult2>:
   400550: movq %rdi,%rax       # a
   400553: imul %rsi,%rax       # a * b
   400557: retq                 # Return
```
P (caller) calls Q (callee)

Stack Frame of P

Arguments (build area)

Return Address

Saved Registers

Local Variables

Arguments (build area)

Stack Frame of Q

%rsp

Bottom Address
When P calls Q

• P is suspended and control moves to Q.
• A stack frame is setup on top of the stack for Q.
• That stack frame contains:
  – saved registers
  – local variables
  – arguments if Q is calling another function
• Some procedures may not need a stack frame (why?).
Procedure Control Flow

• Use stack to support procedure call and return
• Procedure call: **call label** [or **call *op**]
  – Push return address on stack
  – Jump to **label**
• Return address:
  – Address of the next instruction right after call
• Procedure return: **ret**
  – Pop address from stack
  – Jump to address
Example

0000000000400540 <multstore>:
•
•
400544: callq 400550 <mult2>
400549: movq %rax,(%rbx)
•
•

0000000000400550 <mult2>:
400550: movq %rdi,%rax
•
•
400557: retq

•
•

%rip 0x400544
%rsp 0x120
0x120
0x128
0x136
Example

00000000000400540 <multstore>:
  
  400544: callq 400550 <mult2>
400549: movq %rax,(%rbx)
  
00000000000400550 <mult2>:
  
  400550: movq %rdi,%rax
  
  400557: retq
Example

0000000000400540 <multstore>:
  
  400544: callq 400550 <mult2>
  400549: movq %rax,(%rbx)
  
  400550: movq %rdi,%rax
  
  400557: retq
Example

000000000000400540 <multstore>:
  •
  •
  400544: callq 400550 <mult2>
  400549: movq %rax,(%rbx)
  •
  •

000000000000400550 <mult2>:
  400550: movq %rdi,%rax
  •
  •
  400557: retq
Calling Conventions

Answers questions such as:
• Which arguments are passed in which registers?
• Which register holds the return value?
• Where can auxiliary arguments that don't fit into registers be found on the stack?
• Who is responsible for restoring which registers?
  • caller vs. callee-saved registers

Calling conventions are part of the Application binary interface (ABI), which are typically OS-specific.

We focus on the ABI for x86-64 adhered to by most Unix-like operating systems.
# Calling Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Preserved across function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>temporary register; with variable arguments passes information about the number of vector registers used; 1st return register</td>
<td>No</td>
</tr>
<tr>
<td>%rbx</td>
<td>callee-saved register</td>
<td>Yes</td>
</tr>
<tr>
<td>%rcx</td>
<td>used to pass 4th integer argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%rdx</td>
<td>used to pass 3rd argument to functions; 2nd return register</td>
<td>No</td>
</tr>
<tr>
<td>%rsp</td>
<td>stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>%rbp</td>
<td>callee-saved register; optionally used as frame pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>%rsi</td>
<td>used to pass 2nd argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%rdi</td>
<td>used to pass 1st argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%r8</td>
<td>used to pass 5th argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%r9</td>
<td>used to pass 6th argument to functions</td>
<td>No</td>
</tr>
<tr>
<td>%r10</td>
<td>temporary register, used for passing a function’s static chain pointer</td>
<td>No</td>
</tr>
<tr>
<td>%r11</td>
<td>temporary register</td>
<td>No</td>
</tr>
<tr>
<td>%r12–r14</td>
<td>callee-saved registers</td>
<td>Yes</td>
</tr>
<tr>
<td>%r15</td>
<td>callee-saved register; optionally used as GOT base pointer</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Procedure Data Flow

Registers

• First 6 arguments

%rdi
%rsi
%rdx
%rcx
%r8
%r9

• Return value

%rax

Stack

• Only allocate stack space when needed

• When passing parameters on the stack, all data sizes are rounded up to be multiple of eight.
Example:
multstore calls mult2

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

```assembly
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  400541: movq  %rdx,%rbx  # Save dest
  400544: callq  400550 <mult2>  # t = mult2(x,y)
  400549: mov    %rax,(%rbx)  # Save at dest

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

```assembly
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550: movq  %rdi,%rax  # s = a
  400553: imul   %rsi,%rax  # s = s * b
  400557: retq  # return s
```
What about local storage in stack?
When is local storage needed?

- Not enough registers
- A variable in high-level language is passed by reference ("&" in C) so it needs to have an address!
- Arrays, structures, ...
Example: incr

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

incr:

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument p</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument val, y</td>
</tr>
<tr>
<td>%rax</td>
<td>x, Return value</td>
</tr>
</tbody>
</table>
Example: Calling `incr`

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

Function call_incr:
```
subq $8, %rsp
movq $15213,(%rsp)
movq $3000, %rsi
leaq (%rsp), %rdi
call incr
addq (%rsp), %rax
addq $8, %rsp
ret
```
Example: Calling incr

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

call_incr:
  subq  $8, %rsp
  movq  $15213,(%rsp)
  movq  $3000, %rsi
  leaq  (%rsp), %rdi
  call  incr
  addq  (%rsp), %rax
  addq  $8, %rsp
  ret
Example: Calling `incr`

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

call_incr:
```
subq $8, %rsp
movq $15213,(%rsp)
movq $3000, %rsi
leaq (%rsp), %rdi
call incr
addq (%rsp), %rax
addq $8, %rsp
ret
```
Example: Calling `incr`

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

### Stack Structure

<table>
<thead>
<tr>
<th>Rtn address</th>
<th>%rsp</th>
</tr>
</thead>
<tbody>
<tr>
<td>18213</td>
<td></td>
</tr>
</tbody>
</table>

### Register Use(s)

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

### Updated Stack Structure

<table>
<thead>
<tr>
<th>Rtn address</th>
<th>%rsp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
long call_incr():
    subq $8, %rsp
    movq $15213,%(rsp)
    movq $3000, %rsi
    leaq (%rsp), %rdi
    call incr
    addq (%rspb), %rax
    addq $8, %rsp
    ret
```
Example: Calling incr

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

```
call_incr:  
    subq  $8, %rsp
    movq  $15213,(%rsp)
    movq  $3000, %rsi
    leaq  (%rsp), %rdi
    call  incr
    addq  (%rsp), %rax
    addq  $8, %rsp
    ret
```

Updated Stack Structure

```
...  
Rtn address  %rsp
```

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

Final Stack Structure

```
...  
%rsp
```