Machine-Level Programming III: Procedures

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

CS 105
“Tour of the Black Holes of Computing”

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

int Q(int i) {
    int t = 3*i;
    int v[10];
    return v[t];
}

P(…) {
    y = Q(x);
    print(y);
}

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
  - Always changes by multiples of 8

Pushing: pushq Src

- Fetch operand at Src
- Decrement %rsp by 8 (regardless of operand size)
- Then write operand at address given by %rsp

Stack Grows Down
Increasing Addresses
Stack “Bottom”

Stack Pointer %rsp

New Stack “Top”
Increasing Addresses
Stack Grows Down

Stack “Bottom”
Increasing Addresses
Stack Grows Down

Stack “Top”
Stack “Bottom”
Stack Grows Down
Increasing Addresses
Stack Pointer %rsp

x86-64 Stack Popping

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

---

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

---

Stack Operation Examples

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x100</td>
<td>213</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>555</td>
</tr>
<tr>
<td>0x100</td>
<td>213</td>
</tr>
</tbody>
</table>

Control-Flow Example #1

```assembly
0000000000400544 <multstore>:
  400544: callq 400550 <mult2>
  400549: mov %rax,(%rbx)

0000000000400550 <mult2>:
  400550: mov %rdi,%rax
  400555: retq
```
Control-Flow Example #2

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

0000000000400550 <mult2>:
  mov %rdi, %rax

0x400557: retq

Control-Flow Example #3

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

0000000000400550 <mult2>:
  mov %rdi, %rax

0x400557: retq

Control-Flow Example #4

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

0000000000400550 <mult2>:
  mov %rdi, %rax

0x400557: retq

Procedure Data Flow

Registers
First 6 arguments
%rdi
%rsi
%rdx
%rcx
%r8
%r9

Stack

%rax

Return value

Only allocate stack space when needed
Diane’s Silk Dress Cost $89

Registers

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

Data-Flow Example

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

```assembly
0000000000400550 <mult2>:  
    # a in %rdi, b in %rsi
    400550:  mov    %rdi,%rax # a
    400553:  imul   %rsi,%rax # a * b
    # s in %rax
    400557:  retq # Return
```

```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:  mov    %rdx,%rbx # Save dest
    400544:  callq  400550 <mult2> # mult2(x,y)
    # t in %rax
    400549:  mov    %rax,(%rbx) # Save at dest
```

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
```c
yoo(…){
    who(…);
}
who(…){
    amI();
}
amI(…){
    amI();
}
```

Call Chain
- 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
- Deallocated upon return
  - "Finish" code
  - Includes pop done by ret instruction

Example

Example

Example
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 7+ for this call

Example:Calling incr

**long call_incr()**

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

**call_incr:**

```
subq $16, %trap
movq $15213, %trap
movl $3000, %esi
leaq %rdi, %esi
call incr
addq %rax, %esi
addq %r16, %trap
ret
```

Example: Calling incr #2

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

```
call_incr:
subq $16, %trap
movq $15213, %trap
movl $3000, %esi
leaq %rdi, %esi
call incr
addq %rax, %esi
addq %r16, %trap
ret
```
Example: Calling incr #3

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

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

Example: Calling incr #4

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

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

Example: Calling incr #5

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

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

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

yoo:
```assembly
  * * *
  movq $15213, %rdx
  call who
  addq %rdx, %rax
  * * *
  ret
```

who:
```assembly
  * * *
  subq $18213, %rdx
  * * *
  ret```
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|

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rdi, ..., %r9</td>
<td>Caller-saved</td>
</tr>
</tbody>
</table>
  - Can be modified by procedure
| %r10, %r11 | Caller-saved |
  - Can be modified by procedure

Arguments (Diane’s silk dress)
- %rdi
- %rsi
- %rdx
- %rcx
- %r8
- %r9

Callee-saved Example #1

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

Callee-Saved Example #2

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

|x86-64 Linux Register Usage #2|

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>Callee-saved</td>
</tr>
<tr>
<td>%r12, %r13, %r14</td>
<td>Callee must save &amp; restore</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee-saved</td>
</tr>
</tbody>
</table>
  - May 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 v1 + v2;
}
```

Recursive Function

```c
/* 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>

---

Recursive Function Terminal Case

```c
/* 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>
```

---

Recursive Function Register Save

```c
/* 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>
</tbody>
</table>
```

---
/* 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

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

Recursive Function Call

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

Recursive Function Result

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

Recursive Function Completion

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