Machine-Level Programming III: Procedures

15-213/18-213/14-513/15-513: Introduction to Computer Systems
7th Lecture, June 3rd 2020
Objectives

- Basic functionality of the pairs: push / pop and call / ret
- Students should be able to identify the different components of a stack (return address, arguments, saved registers, local variables)
- Explain the difference between callee and caller save registers
- Explain how a stack permits functions to be called recursively / re-entrant
Today

- Procedures
  - Mechanisms
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
Mechanisms in Procedures

- **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

- **Mechanisms all implemented with machine instructions**

- **x86-64 implementation of a procedure uses only those mechanisms required**

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

```c
P(...) {
    ...
    y = Q(x);
    print(y)
    ...
}
```
Mechanisms in Procedures

- **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

- **Mechanisms all implemented with machine instructions**

- **x86-64 implementation of a procedure uses only those mechanisms required**

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

```c
P(...) {
    ...
    y = Q(x);
    print(y)
    ...
}
```
Mechanisms in Procedures

- **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

- **Mechanisms all implemented with machine instructions**

- **x86-64 implementation of a procedure uses only those mechanisms required**

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

```c
P(...) {
    ...
    y = Q(x);
    print(y)
    ...
}
```
Mechanisms in Procedures

- **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

- **Mechanisms all implemented with machine instructions**

- x86-64 implementation of a procedure uses only those mechanisms required

```c
int Q(int i) {
    int t = 3*i;
    int v[10];
    return v[t];
}
```
Mechanisms in Procedures

Machine instructions implement the mechanisms, but the choices are determined by designers. These choices make up the **Application Binary Interface (ABI)**.

- Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required
Today

- Procedures
  - Mechanisms
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
x86-64 Stack

- Region of memory managed with stack discipline
  - Memory viewed as array of bytes.
  - Different regions have different purposes.
  - (Like ABI, a policy decision)
x86-64 Stack

- Region of memory managed with stack discipline

Stack Pointer: %rsp

Stack "Bottom"

Stack "Top"
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`
**x86-64 Stack: Push**

- `pushq Src`
  - Fetch operand at `Src`
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`

**Stack Pointer: `%rsp`**

**Stack “Bottom”**

**Stack “Top”**

**Increasing Addresses**

**Stack Grows Down**

`val`
x86-64 Stack: Push

- `pushq Src`
  - Fetch operand at `Src`
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`

Stack Pointer: `%rsp`

Stack "Bottom"

Stack "Top"

Stack Grows Down

Increasing Addresses
**x86-64 Stack: Pop**

- **popq Dest**
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)

[Diagram of stack with pointer and values]
x86-64 Stack: Pop

- `popq Dest`
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at `Dest` (usually a register)
x86-64 Stack: Pop

- `popq Dest`
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at `Dest` (usually a register)

(The memory doesn’t change, only the value of `%rsp`)
Today

- Procedures
  - Mechanisms
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}

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

0000000000400540 <multstore>:
    400540: push %rbx            # Save %rbx
    400541: mov %rdx,%rbx        # Save dest
    400544: callq 400550 <mult2> # mult2(x,y)
    400549: mov %rax,(%rbx)      # Save at dest
    40054c: pop %rbx            # Restore %rbx
    40054d: retq                # Return

0000000000400550 <mult2>:
    400550: mov %rdi,%rax        # a
    400553: imul %rsi,%rax       # a * b
    400557: retq                # Return
Procedure Control Flow

- Use stack to support procedure call and return
  - **Procedure call**: `call label`
    - Push return address on stack
    - Jump to `label`
  - **Return address**:
    - Address of the next instruction right after call
    - Example from disassembly
  - **Procedure return**: `ret`
    - Pop address from stack
    - Jump to address
Control Flow Example #1

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

0000000000400550 <mult2>:
  400550: mov %rdi, %rax
  
  400557: retq
Control Flow Example #2

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

0000000000400550 <mult2>:
  400550: mov %rdi,%rax
  
  400557: retq
Control Flow Example #3

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

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

0x400557
0x118
0x400549
0x400557
0x118
Control Flow Example #4

00000000000400540 <multstore>:
  ...
  ...
400544: callq 400550 <mult2>
400549: mov %rax,(%rbx)  
  ...
  ...

0x400549

0x120

0x120

%rsp

%rip 0x400549

0x130

0x128

0x120

00000000000400550 <mult2>:
  400550: mov %rdi,%rax
  ...
  ...
400557: retq
Today

- Procedures
  - Mechanisms
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustrations of Recursion & Pointers
Procedure Data Flow

Registers

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

- Return value
  - %rax

Stack

- Only allocate stack space when needed

Increasing Addresses
Data Flow Examples

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

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

00000000000400540 <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
  ...

00000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov   %rdi,%rax    # a
  400553:  imul   %rsi,%rax   # a * b
  # s in %rax
  400557:  retq    # Return
Today

- Procedures
  - Mechanisms
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
Stack-Based Languages

Languages that support recursion
- e.g., C, Pascal, Java
- 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

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

```
who(...) 
{ 
  
  amI(); 
  
}
```

```
amI(...) 
{ 
  
  amI(); 
  
}
```

Procedure `amI` () is recursive

Example Call Chain

```
yoo ----- who ----- amI ---- amI ---- amI
```
Stack Frames

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

- **Management**
  - Space allocated when enter procedure
    - “Set-up” code
    - Includes push by `call` instruction
  - Deallocated when return
    - “Finish” code
    - Includes pop by `ret` instruction

Frame Pointer: `%rbp` (Optional)

Stack Pointer: `%rsp`
Example

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

Stack

```
%rbp
%rsp
```

```c
yoo

who

amI

amI

amI
```
Example

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

Stack

```
yoo
%rbp
%rsp
who
```

Diagram:

- `yoo` function
- `who` function
- Stack with `yoo`, `%rbp`, and `%rsp`
Example

```
who();
%
```

Stack

```
yoo
who
amI
%rbp
%rsp
```
Example

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

Stack

```
<table>
<thead>
<tr>
<th>%rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>amI</td>
</tr>
<tr>
<td>amI</td>
</tr>
<tr>
<td>who</td>
</tr>
<tr>
<td>yoo</td>
</tr>
</tbody>
</table>
```
Example

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

who(…)
{
• • •
  amI ();
  • • •
}

amI (…)
{
• • •
  amI ();
  • • •
}

%rbp

%rsp
```

Stack

- yoo
- who
- amI
- amI
- amI
- %rbp
- %rsp
Example

```
yoo(…)
{  who (…)
  {  amI (…)
    • amI (…)
    • {  
    •  
    • amI ();
    •  
  }
}
```
Example

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

```
Stack
```

```
yoo
who
amI
%rbp
%rsp
```
Example

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

Stack

```
%rbp
%
%rsp
```
Example

```
yoo(...) {
    who(...) {
        amI(...) {
            •
            •
            amI();
            •
            •
        }
    }
}
```

```
yoo
  who
    amI
      amI
```

Stack

```
%rbp
%rsp
```

Example

```c
yoo(...) {
    who(...) {
        • • •
        amI();
        • • •
    }
    • • •
    amI();
    • • •
}
```

Stack

```c
yoo
who
%rbp
%rsp
```
Example

```c
yoo (...) {
    •
    •
    who();
    •
}
```

Stack

```
%rbp
%rsp
```

```
\begin{itemize}
  \item who()
  \item amI
  \item amI
  \item amI
\end{itemize}
```

x86-64/Linux Stack Frame

■ Current Stack Frame ("Top" to Bottom)
  ▪ "Argument build:"
    Parameters for function about to call
  ▪ 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
Register Saving Conventions

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

- Can register be used for temporary storage?

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

who:
  ...  
  subq $18213, %rdx
  ...  
  ret
```

- 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 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
  - Also caller-saved
  - Can be modified by procedure

- **%rdi, ..., %r9**
  - Arguments
  - Also caller-saved
  - Can be modified by procedure

- **%r10, %r11**
  - Caller-saved
  - Can be modified by procedure
x86-64 Linux Register Usage #2

- **%rbx, %r12, %r13, %r14, %r15**
  - Callee-saved
  - Callee must save & restore

- **%rbp**
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match

- **%rsp**
  - Special form of callee save
  - Restored to original value upon exit from procedure
Activity
Today

- Procedures
  - Mechanisms
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}

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

### 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
```

<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>Return value</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);
}
```

### Recursive Function Register Save

**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`

<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>

...  

```
Rtn address  
Saved %rbx  
%rsp
```
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);
}

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x &gt;&gt; 1</td>
<td>Recursive argument</td>
</tr>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
</tbody>
</table>

pcount_r:
movl     $0, %eax
movl     $1, %ebx
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

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

Register Use(s) Type
%rbx x & 1 Callee-saved
%rax Recursive call
            return value
Recursive Function Result

```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>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
<td></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 );
}
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 Lecture 9)
- 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