Assembly III: Procedures

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
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

• All implemented with machine instructions

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

int P(...) {
    ...
    y = Q(x);
    print(y);
    ...
}
```
**x86-64 Stack**

- Region of memory managed with stack discipline
  - Last-In, First-Out (LIFO)
  - Push & Pop
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
  - Address of “top” element
x86-64 Stack: Push

• `pushq Src`
  – Fetch operand at `Src`
  – Decrement `%rsp` by 8
  – Write operand at address given by `%rsp`
x86-64 Stack: Pop

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

- Stack operation examples

```
pushq %rax

%rax  213
%rdx  555
%rsp  0x200

%rax  213
%rdx  555
%rsp  0x1f8

popq %rdx

%rax  213
%rdx  213
%rsp  0x200
```

Procedure Control Flow

• Use stack to support procedure call and return

• **Procedure call:** `call label`
  – Push return address on stack
    • Address of the next instruction right after call
  – Jump to `label`

• **Procedure return:** `ret`
  – Pop address from stack
  – Jump to address
Example

0000000000400546 <main>:
  400546:  48 8d 64 24 f8          lea -0x8(%rsp),%rsp
  40054b:  bf 0a 00 00 00 00  mov $0xa,%edi
  400550:  e8 13 00 00 00     callq 400568 <fact>
  400555:  48 89 c7          mov %rax,%rdi
  400558:  e8 21 00 00 00 00 callq 40057e <print>
  40055d:  b8 00 00 00 00 00 mov $0x0,%eax
  400562:  48 8d 64 24 08 lea 0x8(%rsp),%rsp
  400567:  c3                retq

0000000000400568 <fact>:
  400568:  b8 01 00 00 00 00 mov $0x1,%eax
  40056d:  eb 08          jmp 400577 <fact+0xf>
  40056f:  48 0f af c7 imul %rdi,%rax
  400573:  48 83 ef 01 sub $0x1,%rdi
  400577:  48 83 ff 01 cmp $0x1,%rdi
  40057b:  7f f2          jg 40056f <fact+0x7>
  40057d:  c3                retq
Procedure Call Example

400550:  e8 13 00 00 00  
callq 400568 <fact>
400555:  48 89 c7  
mov  %rax,%rdi

0x00400555 + 0x00000013 = 0x00400568

%rip is program counter
Procedure Return Example

40057d: c3 retq

%rsp 0x108
%rip 0x40057d

0x120
0x118
0x110
0x108 0x400555

0x120
0x118
0x110
0x108 0x400555

ret

%rsp 0x110
%rip 0x400555

%rip is program counter
Passing Arguments

• First 6 arguments:
  – “Diane’s silk dress costs $89”
    - %rdi
    - %rsi
    - %rdx
    - %rcx
    - %r8
    - %r9

• Return value
  - %rax

• Remaining arguments:
  - Push the rest on the stack in reverse order
  - Only allocate stack space when needed

Increasing Addresses

... Arg n ...
... Arg 8 ...
Arg 7
Stack-based Languages

• Languages that support recursion (e.g. C, 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 address

• 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
Stack Frame

- **Contents**
  - Return information
  - Arguments
  - Local variables & temp space

- **Management**
  - “Set-up” code: space allocated when enter procedure
  - “Finish” code: deallocate when return
  - Stack pointer `%rsp` indicates stack top
  - Optional frame pointer `%rbp` indicates start of current frame
Stack Frames: Example (1)

Code Structure

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

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

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

- Procedure `amI` recursive

Call Chain

```
yoo
  who
    amI
      amI
      amI
```

SSE2030: Introduction to Computer Systems, Spring 2018, Jinkyu Jeong (jinkyu@skku.edu)
Stack Frames: Example (2)

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

Call Chain

- Frame Pointer: `%rbp`
- Stack Pointer: `%rsp`

Diagram:
- yoo
- yoo

SSE2030: Introduction to Computer Systems, Spring 2018, Jinkyu Jeong (jinkyu@skku.edu)
Stack Frames: Example (3)

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

Call Chain

- Frame Pointer: `%rbp`
- Stack Pointer: `%rsp`

Diagram showing the call chain with `yoo` and `who` frames.
Stack Frames: Example (4)

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

Call Chain

Frame Pointer
%rbp

Stack Pointer
%rsp

yoo

who

amI
Stack Frames: Example (5)

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

Call Chain

```
  ^
  | yoo
  +--->
     | who
     |   +-->
     |       | amI
     |       +-->
     +---------+ amI
             %rbp
             %rsp
```

Frame Pointer
Stack Pointer
Stack Frames: Example (6)

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

Call Chain

```
Call Chain:
  frames: amI amI amI yoo
  frame pointer: %rbp
  stack pointer: %rsp
```

```
yoo
  who
    amI
      amI
```

SSE2030: Introduction to Computer Systems, Spring 2018, Jinkyu Jeong (jinkyu@skku.edu)
Stack Frames: Example (7)

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

Call Chain

```
Call Chain

Frame Pointer
%rbp

Stack Pointer
%rsp
```

```
Call Chain

amI

who

yoo

amI

amI

amI
```
Stack Frames: Example (8)

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

Call Chain

- Frame Pointer: `%rbp`
- Stack Pointer: `%rsp`
Stack Frames: Example (9)

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

**Call Chain**

- Frame Pointer: `%rbp` 
- Stack Pointer: `%rsp`

```plaintext
Call Chain:
```
```
Stack Pointer: who
```
```
Frame Pointer: yoo
```

```
who
```
```
amI
```
```
amI
```
```
amI
```
Stack Frames: Example (10)

Call Chain

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

Frame Pointer
\%rbp

Stack Pointer
\%rsp
Stack Frames: Example (11)

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

Call Chain

yoo

amI

amI

amI

Frame Pointer
%rbp

Stack Pointer
%rsp
Stack Frames: Example (12)

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

Call Chain:
- `yoo()`
- `who()`
- `amI` (repeatedly)

Stack Pointer `%rsp`
Frame Pointer `%rbp`

```
yoo
  ^
  | Frame Pointer
  | %rbp
  +-----
  | Stack Pointer
  | %rsp
  +-----
      `yoo`
```

Diagram showing the call chain and stack frames.
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
Revisiting Swap

```c
long v1 = 1111;
long v2 = 2222;

void swap (long *xp, long *yp) {
    long t = *xp;
    *xp = *yp;
    *yp = t;
}

int main (void) {
    swap (&v1, &v2);
    ...
}
```

```assembly
v2:             .quad 2222
                 ...
v1:             .quad 1111
                 ...
swap:
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret

main:
    ...
    movq $v2, %rsi
    movq $v1, %rdi
    call swap
    ...
    ret
```
Register Saving Problem

• 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
  
  --- who ---
  
  subq $91125, %rdx
  
  ret

--- Contents of register %rdx overwritten by who() ---
```
Register Saving Conventions

• “Caller saved” registers
  – Caller saves temporary values in its frame before the call
  – Contents of these registers can be modified as a result of procedure call
  – x86-64: %rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11

• “Callee saved” registers
  – Callee saves temporary values in its frame before using
  – Callee restores them before returning to caller
  – The contents of these registers are preserved across a procedure call
  – x86-64: %rbx, %r12, %r13, %r14, %r15, %rbp
x86-64/Linux Register Usage (I)

• `%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

• `%rsp`
  – Special form of callee-saved
  – Restored to original value upon exit from procedure
Recursive Factorial: \texttt{rfact}

- **Registers**
  - \%rax used without first saving
  - \%rbx used, but save at beginning & restore at end

```c
long rfact(long x) {
    long rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
```

\texttt{rfact:}
- \texttt{cmpq $1, %rdi}
- \texttt{jle .L3}
- \texttt{pushq %rbx}
- \texttt{movq %rdi, %rbx}
- \texttt{leaq -1(%rdi), %rdi}
- \texttt{call rfact}
- \texttt{imulq %rbx, %rax}
- \texttt{jmp .L2}

.L3:
- \texttt{movl $1, %eax}
- \texttt{ret}

.L2:
- \texttt{popq %rbx}
- \texttt{ret}
Example: $rfact(3)$

```
rfact:
  cmpq  $1, %rdi
  jle   .L3
  pushq %rbx
  movq  %rdi, %rbx
  leaq  -1(%rdi), %rdi
  call  rfact
A:
  imulq %rbx, %rax
  jmp   .L2
.L3:
  movl  $1, %eax
  ret
.L2:
  popq  %rbx
  ret
```

---

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>3</td>
</tr>
<tr>
<td>%rax</td>
<td>?</td>
</tr>
<tr>
<td>%rbx</td>
<td>?</td>
</tr>
</tbody>
</table>
Example: rfact(3)

rfact:
    cmpq  $1, %rdi
    jle   .L3
    pushq %rbx
    movq  %rdi, %rbx
    leaq  -1(%rdi), %rdi
    call  rfact
A:    imulq %rbx, %rax
    jmp   .L2
.L3:
    movl  $1, %eax
    ret
.L2:
    popq  %rbx
    ret

Registers

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>3</td>
</tr>
<tr>
<td>%rax</td>
<td>?</td>
</tr>
<tr>
<td>%rbx</td>
<td>?</td>
</tr>
</tbody>
</table>
Example: rfact(3)

rfact:
  cmpq $1, %rdi
  jle .L3
  pushq %rbx
  movq %rdi, %rbx
  leaq -1(%rdi), %rdi
  call rfact
A:
  imulq %rbx, %rax
  jmp .L2
.L3:
  movl $1, %eax
  ret
.L2:
  popq %rbx
  ret

<table>
<thead>
<tr>
<th>Registers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>3</td>
</tr>
<tr>
<td>%rax</td>
<td>?</td>
</tr>
<tr>
<td>%rbx</td>
<td>3</td>
</tr>
</tbody>
</table>

SSE2030: Introduction to Computer Systems, Spring 2018, Jinkyu Jeong (jinkyu@skku.edu)
Example: \texttt{rfact(3)}

\begin{verbatim}
\texttt{rfact:}  
cmpq  $1, %rdi 
jle   .L3  
pushq  %rbx  
movq   %rdi, %rbx  
lea   -1(%rdi), %rdi  
call  rfact  
A:  
\texttt{imulq}  %rbx, %rax  
jmp   .L2  
.L3:  
\texttt{movl}  $1, %eax  
ret  
.L2:  
\texttt{popq}  %rbx  
ret
\end{verbatim}

\begin{tabular}{|c|c|}
\hline
\textbf{Registers} & \textbf{} \\
\hline
%rdi & 2 \\
%rax & ? \\
%rbx & 3 \\
\hline
\end{tabular}
Example: \texttt{rfact(3)}

\begin{verbatim}
rfact:
    cmpq $1, %rdi
    jle .L3
    pushq %rbx
    movq %rdi, %rbx
    leaq -1(%rdi), %rdi
    call rfact
    imulq %rbx, %rax
    jmp .L2
.L3:
    movl $1, %eax
    ret
.L2:
    popq %rbx
    ret
\end{verbatim}

\begin{tabular}{|l|l|}
\hline
<table>
<thead>
<tr>
<th>Registers</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>2</td>
</tr>
<tr>
<td>%rax</td>
<td>?</td>
</tr>
<tr>
<td>%rbx</td>
<td>3</td>
</tr>
</tbody>
</table>
\hline
\end{tabular}
Example: rfact(3)

```
rfact:
cmpq  $1, %rdi
jle   .L3
pushq %rbx
movq  %rdi, %rbx
leaq  -1(%rdi), %rdi
call  rfact
imulq %rbx, %rax
jmp   .L2
.L3:
movl  $1, %eax
ret
.L2:
popq  %rbx
ret
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>2</td>
</tr>
<tr>
<td>%rax</td>
<td>?</td>
</tr>
<tr>
<td>%rbx</td>
<td>3</td>
</tr>
</tbody>
</table>
Example: rfact(3)

```assembly
rfact:  
    cmpq $1, %rdi
    jle .L3
    pushq %rbx
    movq %rdi, %rbx
    leaq -1(%rdi), %rdi
    call rfact
    imulq %rbx, %rax
    jmp %rbx, %rax
.L3:
    movl $1, %eax
    ret
.L2:
    popq %rbx
    ret

A:  
    registers

%rdi  2
%rax  ?
%rbx  3
```

SSE2030: Introduction to Computer Systems, Spring 2018, Jinkyu Jeong (jinkyu@skku.edu)
Example: `rfact(3)`

```
rfact:
    cmpq   $1, %rdi
    jle    .L3
    pushq %rbx
    movq   %rdi, %rbx
    leaq   -1(%rdi), %rdi
    call   rfact
A: imulq  %rbx, %rax
    jmp    .L2
.L3:
    movl   $1, %eax
    ret
.L2:
    popq   %rbx
    ret
```
Example: rfact(3)

```
rfact:
  cmpq  $1, %rdi
  jle .L3
  pushq %rbx
  movq  %rdi, %rbx
  leaq  -1(%rdi), %rdi
  call rfact
A:  imulq %rbx, %rax
  jmp .L2
.L3:
  movl  $1, %eax
  ret
.L2:
  popq  %rbx
  ret
```

Registers:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1</td>
</tr>
<tr>
<td>%rax</td>
<td>?</td>
</tr>
<tr>
<td>%rbx</td>
<td>2</td>
</tr>
</tbody>
</table>
Example: rfact(3)

```
rfact:
    cmpq $1, %rdi
    jle .L3
    pushq %rbx
    movq %rdi, %rbx
    leaq -1(%rdi), %rdi
    call rfact
    imulq %rbx, %rax
    jmp .L2
.L3:
    movl $1, %eax
    ret
.L2:
    popq %rbx
    ret
```

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1</td>
</tr>
<tr>
<td>%rax</td>
<td>?</td>
</tr>
<tr>
<td>%rbx</td>
<td>2</td>
</tr>
</tbody>
</table>

RetAddr

%rsp

%rip

Lower Address

A

3

A

SSE2030: Introduction to Computer Systems, Spring 2018, Jinkyu Jeong (jinkyu@skku.edu)
Example: rfact(3)

rfact:

```
    cmpq $1, %rdi
    jle .L3
    pushq %rbx
    movq %rdi, %rbx
    leaq -1(%rdi), %rdi
    call rfact
    imulq %rbx, %rax
    jmp .L2
.L3:
    movl $1, %eax
    ret
.L2:
    popq %rbx
    ret
```

A:

```
imulq %rbx, %rax
jmp .L2
```

Registers

<table>
<thead>
<tr>
<th>Registers</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1</td>
</tr>
<tr>
<td>%rax</td>
<td>?</td>
</tr>
<tr>
<td>%rbx</td>
<td>2</td>
</tr>
</tbody>
</table>
Example: rfact(3)

```
rfact:
  cmpq $1, %rdi
  jle .L3
  pushq %rbx
  movq %rdi, %rbx
  leaq -1(%rdi), %rdi
  call rfact
A:  imulq %rbx, %rax
  jmp .L2
.L3:
  movl $1, %eax
  ret
.L2:
  popq %rbx
  ret
```

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1</td>
</tr>
<tr>
<td>%rax</td>
<td>1</td>
</tr>
<tr>
<td>%rbx</td>
<td>2</td>
</tr>
</tbody>
</table>

**Example Assembly Code**

- `cmpq $1, %rdi` compares the value in `%rdi` with 1.
- `jle .L3` jumps to the label `.L3` if `%rdi` is less than or equal to 1.
- `pushq %rbx` pushes the value in `%rbx` onto the stack.
- `movq %rdi, %rbx` moves the value from `%rdi` to `%rbx`.
- `leaq -1(%rdi), %rdi` loads the address of the next value in the stack.
- `call rfact` calls the `rfact` function.
- `imulq %rbx, %rax` multiplies `%rbx` by `%rax`.
- `jmp .L2` jumps to the label `.L2`.
- `movl $1, %eax` moves the value 1 into the `%eax` register.
- `ret` returns from the current function.
Example: \texttt{rfact(3)}

\begin{itemize}
\item \texttt{rfact}: \texttt{cmpq $1, \%rdi}
\item \texttt{jle .L3}
\item \texttt{pushq \%rbx}
\item \texttt{movq \%rdi, \%rbx}
\item \texttt{leaq -1(\%rdi), \%rdi}
\item \texttt{call \texttt{rfact}}
\item \texttt{imulq \%rbx, \%rax}
\item \texttt{jmp .L2}
\item \texttt{.L3: movl $1, \%eax}
\item \texttt{ret}
\item \texttt{.L2: popq \%rbx}
\item \texttt{ret}
\end{itemize}

\begin{tabular}{|l|c|}
\hline
\textbf{Registers} & \\
\hline
\%rdi & 1 \\
\%rax & 1 \\
\%rbx & 2 \\
\hline
\end{tabular}
Example: rfact(3)

rfact:
  cmpq $1, %rdi
  jle .L3
  pushq %rbx
  movq %rdi, %rbx
  leaq -1(%rdi), %rdi
  call rfact
  imulq %rbx, %rax
  jmp .L2

.L3:
  movl $1, %eax
  ret

.L2:
  popq %rbx
  ret

<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
</tr>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rbx</td>
</tr>
</tbody>
</table>
Example: rfact(3)

```
rfact:
    cmpq $1, %rdi
    jle .L3
    pushq %rbx
    movq %rdi, %rbx
    leaq -1(%rdi), %rdi
    call rfact
A: imulq %rbx, %rax
    jmp .L2
.L3:
    movl $1, %eax
    ret
.L2:
    popq %rbx
    ret
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1</td>
</tr>
<tr>
<td>%rax</td>
<td>2</td>
</tr>
<tr>
<td>%rbx</td>
<td>3</td>
</tr>
</tbody>
</table>

RetAddr

? A 3 A

Lower Address

%rsp

%rip

Lower Address

%rdi
%rax
%rbx
Example: rfact(3)

rfact:
  cmpq  $1, %rdi
  jle   .L3
  pushq %rbx
  movq  %rdi, %rbx
  leaq  -1(%rdi), %rdi
  call  rfact
  imulq %rbx, %rax
  jmp   .L2

.L3:
  movl  $1, %eax
  ret

.L2:
  popq  %rbx
  ret

RetAddr

%rsp

Lower Address

 Registers
| %rdi | 1 |
| %rax | 2 |
| %rbx | 3 |
Example: \texttt{rfact(3)}

\begin{verbatim}
rfact:
cmpq $1, %rdi
jle .L3
pushq %rbx
movq %rdi, %rbx
leaq -1(%,%rdi), %rdi
call rfact
imulq %rbx, %rax
jmp .L2
.L3:
movl $1, %eax
ret
.L2:
popq %rbx
ret
\end{verbatim}

\begin{center}
\begin{tabular}{|c|c|}
\hline
| Registers | \hline
| %rdi      | 1  |
| %rax      | 6  |
| %rbx      | 3  |
\hline
\end{tabular}
\end{center}
Example: rfact(3)

rfact:
  cmpq  $1, %rdi
  jle   .L3
  pushq %rbx
  movq  %rdi, %rbx
  leaq  -1(%rdi), %rdi
  call  rfact
.A:
  imulq %rbx, %rax
  jmp   .L2
.L3:
  movl  $1, %eax
  ret
.L2:
  popq  %rbx
  ret

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1</td>
</tr>
<tr>
<td>%rax</td>
<td>6</td>
</tr>
<tr>
<td>%rbx</td>
<td>?</td>
</tr>
</tbody>
</table>

RetAddr

%rsp

Lower Address

A

3

A
Example: rfact(3)

rfact:
  cmpq $1, %rdi
  jle .L3
  pushq %rbx
  movq %rdi, %rbx
  leaq -(%rdi), %rdi
  call rfact

A: imulq %rbx, %rax
  jmp .L2

.L3:
  movl $1, %eax
  ret

.L2:
  popq %rbx
  ret

<table>
<thead>
<tr>
<th>Registers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1</td>
</tr>
<tr>
<td>%rax</td>
<td>6</td>
</tr>
<tr>
<td>%rbx</td>
<td>?</td>
</tr>
</tbody>
</table>
Observations about Recursion

• **Handled without special consideration**
  – Stack frames mean each function call has private storage
    • Saved registers + local variables
    • Saved return address
  – Register saving conventions prevent one function call from corrupting another’s data
    • Unless the C code explicitly does so (e.g. buffer overflow)
  – 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
Summary

• Stack is the right data structure for procedure call / return
  – Private storage for each instance of procedure call
  – Recursion handled by normal calling conventions

• Mechanisms
  – call, ret, push, pop, etc. instructions
  – Registers for passing arguments and return value
  – Stack memory

• Policies
  – Register usage (caller / callee save, %rbp & %rsp)
  – Stack frame organization