Procedures II
CSE 351 Winter 2020

Instructor: Ruth Anderson
Teaching Assistants: Jonathan Chen, Josie Lee, Eddy (Tianyi) Zhou, Justin Johnson, Jeffery Tian, Porter Jones, Callum Walker

http://xkcd.com/1790/
Administrivia

- HWs
  - Some require the textbook. For copyright reasons we cannot just post the book content or questions.
    - Those HWs have [CSPP] in their title
    - Copy of book on reserve at Odegaard the Engineering Library
  - Gradescope does not allow us to specify multiple correct answers 😞, check the “Tips”!

- Lab 2 due Friday (2/07)
  - Since you are submitting a text file (defuser.txt), there won’t be any Gradescope autograder output this time
  - Extra credit needs to be submitted to the extra credit assignment

- Midterm: Monday (2/10), during lecture
Example: \texttt{increment}

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

\textbf{Register Use(s)}

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1\textsuperscript{st} arg (\texttt{p})</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\textsuperscript{nd} arg (\texttt{val}), y</td>
</tr>
<tr>
<td>%rax</td>
<td>\texttt{x}, return value</td>
</tr>
</tbody>
</table>

\textbf{Assembly Code:}

```
movq (%rdi), %rax
addq %rax, %rsi
movq %rsi, (%rdi)
ret
```
Procedure Call Example (initial state)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

Call `increment`:

```assembly
subq $16, %rsp
movq $351, 8(%rsp)
movl $100, %esi
leaq 8(%rsp), %rdi
call increment
addq 8(%rsp), %rax
addq $16, %rsp
ret
```

- Return address on stack is the address of instruction immediately following the call to “call_incr”
  - Shown here as `main`, but could be anything
  - Pushed onto stack by `call call_incr`

Initial Stack Structure:

- Return addr `<main+8>`
Procedure Call Example (step 1)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

---

**Stack Structure**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;main+8&gt;</td>
<td>Return addr</td>
</tr>
<tr>
<td>351</td>
<td>Unused</td>
</tr>
</tbody>
</table>

Allocate space for local vars

- Setup space for local variables
  - Only v1 needs space on the stack

- Compiler allocated extra space
  - Often does this for a variety of reasons, including alignment

---

call_incr:

```
subq $16, %rsp
movq $351, 8(%rsp)
movl $100, %esi
leaq 8(%rsp), %rdi
call increment
addq 8(%rsp), %rax
addq $16, %rsp
ret
```
Procedure Call Example (step 2)

```
call_incr:                
  subq    $16, %rsp  
  movq    $351, 8(%rsp)  
  movl    8(%rsp), %rdi  
  call    increment  
  addq    8(%rsp), %rax  
  addq    $16, %rsp  
  ret
```

Aside: `movl` is used because 100 is a small positive value that fits in 32 bits. High order bits of `rsi` get set to zero automatically. It takes one less byte to encode a `movl` than a `movq`.

```
long call_incr() {
  long v1 = 351;
  long v2 = increment(&v1, 100);
  return v1 + v2;
}
```

Stack Structure

<table>
<thead>
<tr>
<th>Return addr &lt;main+8&gt;</th>
<th>351 Unused</th>
</tr>
</thead>
</table>

Register | Use(s) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>100</td>
</tr>
</tbody>
</table>

Set up parameters for call to `increment`
Procedure Call Example (step 3)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

call_incr:

- `subq $16, %rsp`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

increment:

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

State while inside `increment`

- **Return address** on top of stack is address of the `addq` instruction immediately following call to `increment`

Stack Structure:

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
</tbody>
</table>
Procedure Call Example (step 4)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
    subq  $16, %rsp
    movq  $351, 8(%rsp)
    movl  $100, %esi
    leaq  8(%rsp), %rdi
    call  increment
    addq  8(%rsp), %rax
    addq  $16, %rsp
    ret
```

increment:
```
    movq  (%rdi), %rax  # x = *p
    addq  %rax, %rsi  # y = x + 100
    movq  %rsi, (%rdi)  # *p = y
    ret
```

- State while inside `increment`
  - After code in body has been executed

### Stack Structure

<table>
<thead>
<tr>
<th>Return addr &lt;main+8&gt;</th>
<th>451</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>Return addr &lt;call_incr+?&gt;</td>
<td>%rsp</td>
</tr>
</tbody>
</table>

#### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 5)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**call_incr:**
- subq $16, %rsp
- movq $351, 8(%rsp)
- movl $100, %esi
- leaq 8(%rsp), %rdi
- call increment
- addq 8(%rsp), %rax
- addq $16, %rsp
- ret

- After returning from call to `increment`
  - Registers and memory have been modified and return address has been popped off stack

**Stack Structure**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 6)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**call_incr:**
- `subq $16, %rsp`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

**Stack Structure**

```
...                       ←%rsp+8
Return addr <main+8>
451                       ←%rsp
Unused
```

**Update %rax to contain v1+v2**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>451+351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 7)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**Stack Structure**

- Return addr <main+8>
- 451
- Unused

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 8)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

`call_incr`:
- `subq $16, %rsp`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

- State *just before* returning from call to `call_incr`

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- **Stack Structure**

  Return addr <main+8> ← %rsp

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 9)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**call_incr:**
- subq $16, %rsp
- movq $351, 8(%rsp)
- movl $100, %esi
- leaq 8(%rsp), %rdi
- call increment
- addq 8(%rsp), %rax
- addq $16, %rsp
- ret

- State immediately after returning from call to call_incr
  - Return addr has been popped off stack
  - Control has returned to the instruction immediately following the call to call_incr (not shown here)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Register Saving Conventions

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

- Can registers be used for temporary storage?
  - No! Contents of register `%rdx` overwritten by `who`!
  - This could be trouble – something should be done. Either:
    - **Caller** should save `%rdx` before the call (and restore it after the call)
    - **Callee** should save `%rdx` before using it (and restore it before returning)
Register Saving Conventions

∙ **“Caller-saved” registers**
  
  ▪ It is the **caller**’s responsibility to save any important data in these registers before calling another procedure (i.e. the **callee** can freely change data in these registers)
  
  ▪ **Caller** saves values in its stack frame before calling **Callee**, then restores values after the call

∙ **“Callee-saved” registers**
  
  ▪ It is the callee’s responsibility to save any data in these registers before using the registers (i.e. the **caller** assumes the data will be the same across the **callee** procedure call)
  
  ▪ **Callee** saves values in its stack frame before using, then restores them before returning to **caller**
Silly Register Convention Analogy

1) Parents (*caller*) leave for the weekend and give the keys to the house to their child (*callee*)
   - Being suspicious, they put away/hid the valuables (*caller-saved*) before leaving
   - Warn child to leave the bedrooms untouched: “These rooms better look the same when we return!”

2) Child decides to throw a wild party (*computation*), spanning the entire house
   - To avoid being disowned, child moves all of the stuff from the bedrooms to the backyard shed (*callee-saved*) before the guests trash the house
   - Child cleans up house after the party and moves stuff back to bedrooms

3) Parents return home and are satisfied with the state of the house
   - Move valuables back and continue with their lives
x86-64 Linux Register Usage, part 1

- **%rax**
  - Return value
  - Also **caller**-saved & restored
  - Can be modified by procedure

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

- **%r10, %r11**
  - **Caller**-saved & restored
  - Can be modified by procedure

Diagram:
- **Return value**:
  - %rax
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9

- **Arguments**:
  - %r10
  - %r11

- **Caller-saved temporaries**:
x86-64 Linux Register Usage, part 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
### x86-64 64-bit Registers: Usage Conventions

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

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>Argument #5 - <strong>Caller saved</strong></td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6 - <strong>Caller saved</strong></td>
</tr>
<tr>
<td>%r10</td>
<td><strong>Caller saved</strong></td>
</tr>
<tr>
<td>%r11</td>
<td><strong>Callee saved</strong></td>
</tr>
<tr>
<td>%r12</td>
<td><strong>Callee saved</strong></td>
</tr>
<tr>
<td>%r13</td>
<td><strong>Callee saved</strong></td>
</tr>
<tr>
<td>%r14</td>
<td><strong>Callee saved</strong></td>
</tr>
<tr>
<td>%r15</td>
<td><strong>Callee saved</strong></td>
</tr>
</tbody>
</table>
Callee-Saved Example (step 1)

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

**call_incr2:**
- `pushq %rbx`
- `subq $16, %rsp`
- `movq %rdi, %rbx`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq %rbx, %rax`
- `addq $16, %rsp`
- `popq %rbx`
- `ret`

**Initial Stack Structure**

```
... ret addr
%rsp
```

**Resulting Stack Structure**

```
... ret addr
Saved %rbx
351 Unused
%rsp+8 %rsp
```
Callee-Saved Example (step 2)

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

call_incr2:  
```asm
pushq %rbx
subq $16, %rsp
movq %rdi, %rbx
movq $351, 8(,%rsp)
movl $100, %esi
leaq 8(%rsp), %rdi
call increment
addq %rbx, %rax
addq $16, %rsp
popq %rbx
ret
```

Stack Structure
- ...  
- Rtn address  
- Saved %rbx  
- 351  
- Unused %rsp+8 %rsp

Pre-return Stack Structure
- ...  
- Rtn address %rsp
Why Caller and Callee Saved?

- We want one calling convention to simply separate implementation details between caller and callee

- In general, neither caller-save nor callee-save is “best”:
  - If caller isn’t using a register, caller-save is better
  - If callee doesn’t need a register, callee-save is better
  - If “do need to save”, callee-save generally makes smaller programs
    - Functions are called from multiple places

- So... “some of each” and compiler tries to “pick registers” that minimize amount of saving/restoring
Register Conventions Summary

- **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - **Callee** may change those register values

- **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - **Caller** expects unchanged values in those registers

- Don’t forget to restore/pop the values later!
Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Recursive Function

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

Compiler Explorer: https://godbolt.org/z/xFCrsw

- Compiled with `-O1` for brevity instead of `-Og`
- Try `-O2` instead!

### pcount_r:
```
movl $0, %eax
testq %rdi, %rdi
jne .L8
rep ret
.L8:
pushq %rbx
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
ret
```
Recursive Function: Base Case

```c
/* 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
---
%rdi x Argument
%rax Return value Return value

Trick because some AMD hardware doesn’t like jumping to `ret`
Recursive Function: **Callee Register Save**

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

**The Stack**

```
...  
rtm <main+?>
 saved %rbx

%rsp →
```

Need original value of \( x \) **after** recursive call to \( pcount_r \).

“Save” by putting in \%rbx (callee saved), but need to save old value of \%rbx before you change it.

<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>

**pcount_r:**

```
movl $0, %eax  
testq %rdi, %rdi  
jne .L8  
rep ret  
.L8:  
pushq %rax  
movq %rdi, %rbx  
shrq %rdi  
call pcount_r  
andl $1, %ebx  
addq %rbx, %rax  
popq %rbx  
ret```

"Save" by putting in \%rbx (callee saved), but need to save old value of \%rbx before you change it.
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);
}

The Stack

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x (new)</td>
<td>Argument</td>
</tr>
<tr>
<td>%rbx</td>
<td>x (old)</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

The Stack

%rsp →

...  

rtn <main+?>  

saved %rbx

```
; pcount_r:
movl $0, %eax  
testq %rdi, %rdi  
jne .L8  
rep ret  
.L8:  
pushq %rbx  
movq %rdi, %rbx  
shrq %rdi  
call pcount_r  
andl $1, %ebx  
addq %rbx, %rax  
popq %rbx  
ret
```
Recursive Function: Call

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

The Stack

```
%rsp →
... 

rtn <main+?>

saved %rbx

rtn <pcount_r+22>

... 
```

### Register Use(s) Table

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Recursive call return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>x (old)</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

### Code Snippet

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    rep ret .L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```
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);
}
```

The Stack

```
...  
rtm <main+?>  
saved %rbx
```

The Stack Diagram:

```
%rsp →
```

Register Use(s) Type

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>x&amp;1</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    rep ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```
Recursive Function: Completion

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

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Previous %rbx value</td>
<td>Callee restored</td>
</tr>
</tbody>
</table>

#### pcount_r:

- `movl $0, %eax`
- `testq %rdi, %rdi`
- `jne .L8`
- `rep ret`
- `.L8:
  - `pushq %rbx`
  - `movq %rdi, %rbx`
  - `shrq %rdi`
  - `call pcount_r`
  - `andl $1, %ebx`
  - `addq %rbx, %rax`
  - `popq %rbx`
  - `ret`
Observations About Recursion

- Works without any special consideration
  - Stack frames mean that 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 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 (LIFO)

- Also works for mutual recursion (P calls Q; Q calls P)
x86-64 Stack Frames

- Many x86-64 procedures have a minimal stack frame
  - Only return address is pushed onto the stack when procedure is called

- A procedure *needs* to grow its stack frame when it:
  - Has too many local variables to hold in *caller*-saved registers
  - Has local variables that are arrays or structs
  - Uses & to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Is using *caller*-saved registers and then calls a procedure
  - Modifies/uses *callee*-saved registers
x86-64 Procedure Summary

- Important Points
  - Procedures are a combination of instructions and conventions
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
  - Recursion handled by normal calling conventions

- Heavy use of registers
  - Faster than using memory
  - Use limited by data size and conventions

- Minimize use of the Stack