Procedures II
CSE 351 Autumn 2021

Instructor:
Justin Hsia

Teaching Assistants:
Allie Pfleger
Anirudh Kumar
Assaf Vayner
Atharva Deodhar
Celeste Zeng
Dominick Ta
Francesca Wang
Hamsa Shankar
Isabella Nguyen
Joy Dang
Julia Wang
Maggie Jiang
Monty Nitschke
Morel Fotsing
Sanjana Chintalapati

http://xkcd.com/1270/
Relevant Course Information

- Lab 1b grades released
  - Regrade requests open Tuesday – Thursday
- Lab 2 due Friday (10/29)
  - Since you are submitting a text file (defuser.txt), there won’t be any Gradescope autograder output this time
  - Extra credit (bonus) needs to be submitted to the extra credit assignment

- Midterm (take home, 11/3–11/5)
  - Make notes and use the midterm reference sheet
  - Form study groups and look at past exams!
Polling Question

- Answer the following questions about when `main()` is run (assume `x` and `y` stored on the Stack):

  ```c
  int main() {
    int i, x = 0;
    for(i=0;i<3;i++)
      x = randSum(x);
    printf("x = %d\n",x);
    return 0;
  }

  int randSum(int n) {
    int y = rand()%20;
    return n+y;
  }
  ```

- Higher/larger address: `x` or `y`?
- How many total stack frames are created?
- What is the maximum depth (# of frames) of the Stack?

  ![Diagram of stack frames]

  A. 1  B. 2  C. 3  D. 4
Reading Review

❖ Terminology:
  ▪ Stack frame: return address, saved registers, local variables, argument build
  ▪ Register saving conventions: callee-saved and caller-saved

❖ Questions from the Reading?
x86-64/Linux Stack Frame (Review)

- **Caller’s Stack Frame**
  - Extra arguments (if > 6 args) for this call

- **Current/Callee Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Saved register context
    (when reusing registers)
  - Local variables
    (If can’t be kept in registers)
  - “Argument build” area
    (If callee needs to call another function -
     parameters for function about to call, if needed)
Review Question

In the following function, which instruction(s) pertain to the **local variables** and **saved registers** portions of its stack frame?

```
call_incr2:
  1  pushq  %rbx  # save a register value
  2  subq  $16, %rsp  # allocates space for local variables
  3  movq  %rdi, %rbx
  4  movq  $351, 8(%rsp)  # initializes local variable value on stack
  5  movl  $100, %esi
  6  leaq  8(%rsp), %rdi  # gets address of local variable (but doesn't actually use local var)
  7  call  increment
  8  addq  %rbx, %rax
  9  addq  $16, %rsp  # deallocates space for local variables
 10  popq  %rbx  # restore the register value
 11  ret
```
**Example: increment**

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

**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 (p)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\textsuperscript{nd} arg (val), y</td>
</tr>
<tr>
<td>%rax</td>
<td>x, return value</td>
</tr>
</tbody>
</table>

*written this way to correspond to assembly*

*adding val to value store at p*
Procedure Call Example (initial state)

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

- 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`

Return address on stack is the address of instruction immediately *following* the call to "call_incr"
# Procedure Call Example (step 1)

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

- **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

### Stack Structure
```
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
```

- **Allocate space for local vars**
  - "manual push"
- **Setup space for local variables**
  - Only `v1` needs space on the stack

- **Return addr** `<main+8>`
Procedure Call Example (step 2)

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

**Stack Structure**

- **Return addr** `<main+8>`
- **351**
- **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>100</td>
</tr>
</tbody>
</table>

**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`. 
Procedure Call Example (step 3)

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

---

**Stack Structure**

- **Return addr** `<main+8>`
- **351**
- **Unused**
- **Return addr** `<call_incr+?>`

**Stack Frame**

- `frames` can be different sizes
- `%rsp`

**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>100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
</tbody>
</table>

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

**Instruction Set**

- **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`
Procedure Call Example (step 4)

```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>351</td>
</tr>
</tbody>
</table>

**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:**

1. `movq (%rdi), %rax` # `x = *p`
2. `addq %rax, %rsi` # `y = x + 100`
3. `movq %rsi, (%rdi)` # `*p = y`
4. `ret`

State while inside `increment`

- **After** code in body has been executed

- Returned address pushed onto stack
- `%rsp` popped off stack by `ret` instruction
- `351` read
- `add 100`
- `store 451`
Procedure Call Example (step 5)

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

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

- 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 (v2)</td>
</tr>
</tbody>
</table>
**Procedure Call Example (step 6)**

```c
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>451</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused</td>
<td>← %rsp+8</td>
</tr>
<tr>
<td></td>
<td>← %rsp</td>
</tr>
</tbody>
</table>

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

**Register Use(s)**

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

De-allocate space for local vars

(make sure %rsp points to return addr before ret)
Procedure Call Example (step 8)

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

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

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

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

**Final Stack Structure**

- 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)

---

**Register Use(s) Table**

<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 (Review)

- 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 (Review)

❖ “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 (Review)

- **%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
x86-64 Linux Register Usage (Review)

- `%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 Linux Register Usage (Review)

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>Argument #5 - <strong>Caller</strong> saved</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6 - <strong>Caller</strong> saved</td>
</tr>
<tr>
<td>%r10</td>
<td><strong>Caller</strong> saved</td>
</tr>
<tr>
<td>%r11</td>
<td><strong>Caller</strong> Saved</td>
</tr>
<tr>
<td>%r12</td>
<td><strong>Callee</strong> saved</td>
</tr>
<tr>
<td>%r13</td>
<td><strong>Callee</strong> saved</td>
</tr>
<tr>
<td>%r14</td>
<td><strong>Callee</strong> saved</td>
</tr>
<tr>
<td>%r15</td>
<td><strong>Callee</strong> saved</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;
}
```

Initial Stack Structure

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

Resulting Stack Structure

```
...  
ret addr
Saved %rbx
351
Unused
%rsp+8
%rsp
```

```c
long call_incr2:
    pushq %rbx ← save old %rbx
    subq $16, %rsp
    movq %rdi, %rbx ← change %rbx
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment across procedure call
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

focused on this interaction

\[ \text{main} \downarrow \]
\[ \text{call\_incr2} \downarrow \]
\[ \text{increment} \]

↑ need \( x \) (in \%rdi) after procedure call
Callee-Saved Example (step 2)

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

call_incr2:

1. pushq %rbx
2. subq $16, %rsp
3. movq %rdi, %rbx
   movq $351, 8(%rsp)
   movl $100, %esi
   leaq 8(%rsp), %rdi
   call increment
   addq %rbx, %rax
   addq $16, %rsp
3. popq %rbx
   ret

Memory Stack Structure

- Rtn address
- Saved %rbx
- 351
- Unused

Pre-return Stack Structure

- Rtn address

Registers:

1. %rbx
2. %rdi
3. %esi

Pre-Return Stack Structure

- Rtn address

Stack discipline:
- add/sub push/pull must be symmetric within procedure

Unused

%rsp+8

%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) /* stop once all 1's shifted off */
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

Counts the number of 1’s in the binary representation of x.

Compiler Explorer: [https://godbolt.org/z/naP4ax](https://godbolt.org/z/naP4ax)

- Compiled with `-O1` instead of `-Og` for more natural instruction ordering
Recursive Function: Base Case

/* 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

(pcount_r):
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    ret

.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret

(just worry about it)
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**

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.

### Stack

- `rtn <main+?>`
- `saved %rbx`

### Code Snippet

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

```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) and Type

<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 →
saved %rbx
rtn <main+?>
...  
```

### pcount_r:

```
movl $0, %eax
	testq %rdi, %rdi
	jne .L8
	ret

.L8:
	pushq %rbx
	movq %rdi, %rbx
	shrq %rdi
	call implicit 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);
}
```

Register | Use(s)                  | Type
---------|-------------------------|------
%rax     | Recursive call          | Return value
%rbx     | x (old)                 | Callee saved

The Stack

- If original: `x = 0b101`
- `main`
- `pcount_r(5)`
- `pcount_r(2)`
- `saved %rbx= ?`
- `rtn <main+?>`
- `rtn <pcount_r+22>`
- `%rsp → %rdi = 2`
- `%rbx` saved

Assembly Code:

```assembly
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    ret

.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
    ret
```
Recursive Function: Result

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

The Stack

<table>
<thead>
<tr>
<th>The Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>rtn &lt;main+?&gt;</td>
</tr>
<tr>
<td>saved %rbx</td>
</tr>
</tbody>
</table>

%rsp →

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq
    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) and 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>Previous %rbx value</td>
<td>Callee restored</td>
</tr>
</tbody>
</table>

**The Stack**

- **%rsp →**
  - ...  
  - rtn <main+?>
  - saved %rbx

**pcount_r:**

- `movl $0, %eax`
- `testq %rdi, %rdi`
- `jne .L8`
- `ret`

**.L8:**

- `pushq %rbx`
- `movq %rdi, %rbx`
- `shrq %rdi`
- `call pcount_r`
- `andl $1, %ebx`
- `addq %rbx, %rax`
- `popq %rbx`
- `ret` <restore before returning>
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