

Procedures II

CSE 351 Autumn 2024

Instructor:

Ruth Anderson

Teaching Assistants:

Alexandra Michael

Chendur Jayavelu

Nahush Shrivatsa

Renee Ruan

Sean Siddens

Connie Chen

Joshua Tan

Naama Amiel

Rubee Zhao

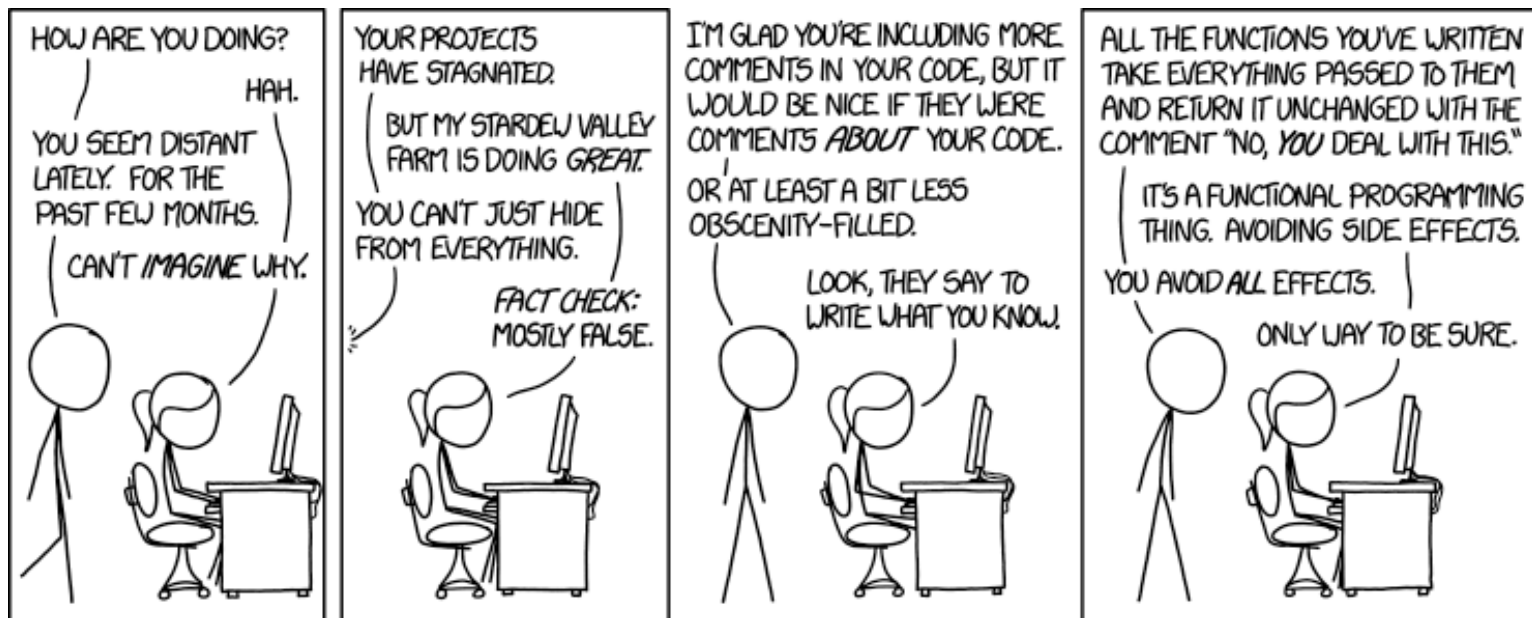
Waleed Yagoub

Chloe Fong

Nikolas McNamee

Neela Kausik

Samantha Dreussi



Relevant Course Information

- ❖ HW10 due TONIGHT, Monday (10/21) @ 11:59 pm
- ❖ Lab 2 (x86-64) due this Friday (10/25)
 - GDB [Demo Video on Ed!](#)
 - See GDB Tutorial and Phase 1 walkthrough in Section 4!
 - Since you are submitting a text file (`defuser.txt`), there won't be any Gradescope autograder output this time
- ❖ HW11 due Wednesday (10/23) @ 11:59 pm
- ❖ **Midterm Exam:** take home, on Gradescope, more details coming soon: <https://cs.uw.edu/cse351/exams/>
 - Released: Thursday 10/31 at 5pm
 - Due: Saturday 11/02 at 11:59pm
 - Study BEFORE the exam opens!

Reading Review

❖ Terminology:

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

Polling Question

Vote only on 3rd question on Ed Lessons

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

```
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?

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

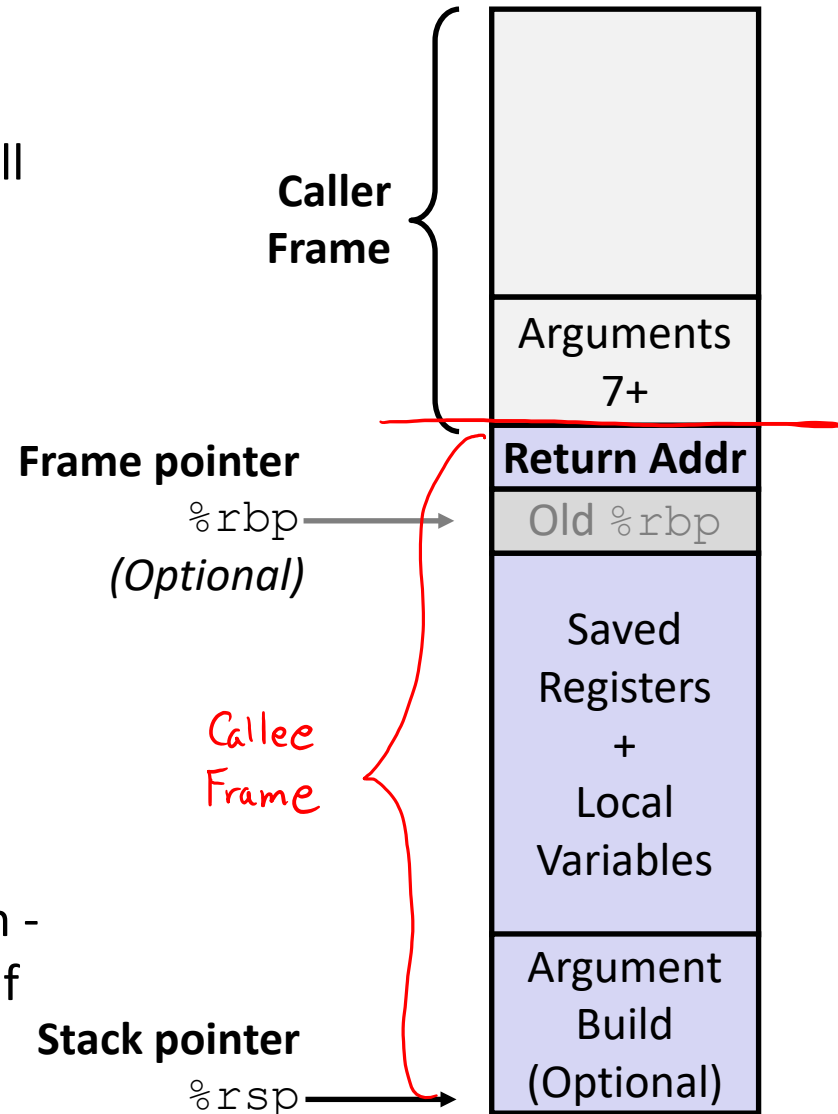
x86-64/Linux Stack Frame

❖ 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 Questions

- ❖ In the following function: how big (in bytes) is the stack frame? Which instruction(s) pertain to the **local variables** and **saved registers** portion of its stack frame?

```
call_incr2:
1  pushq    %rbx
2  subq     $16, %rsp
3  movq     %rdi, %rbx
4  movq     $351, 8(%rsp)
5  movl     $100, %esi
6  leaq     8(%rsp), %rdi
7  call     increment
8  addq     %rbx, %rax
9  addq     $16, %rsp
10 popq     %rbx
11 ret
```

Example: increment

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

increment:

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

Register	Use(s)
%rdi	1 st arg (p)
%rsi	2 nd arg (val), y
%rax	x, return value

Procedure Call Example (initial state)

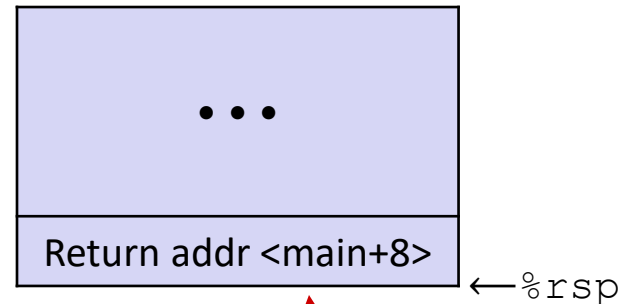


```
long call_incr() {  
    long v1 = 351;  
    long v2 = increment(&v1, 100);  
    return v1 + v2;  
}
```



```
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
```

Initial Stack Structure



- ❖ 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

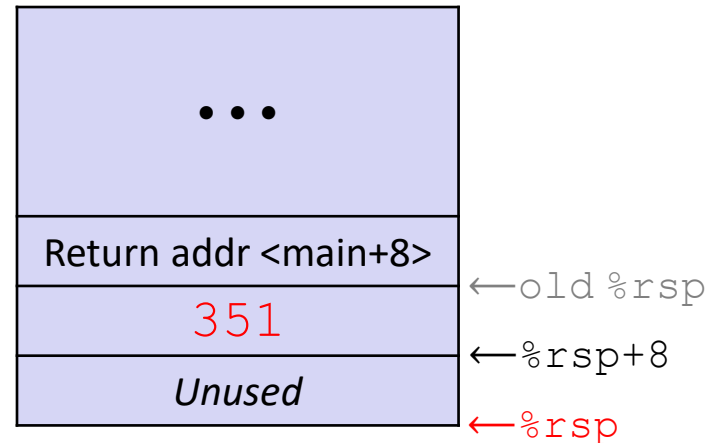
Procedure Call Example (step 1)

```
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
```

} Allocate space
for local vars

Stack Structure



- ❖ 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

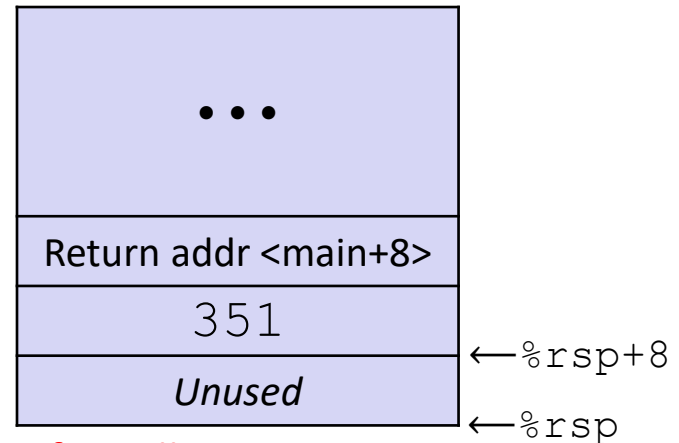
Procedure Call Example (step 2)

```
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
```

} Set up parameters for call
to increment

Stack Structure



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`.

Register	Use(s)
%rdi	&v1
%rsi	100

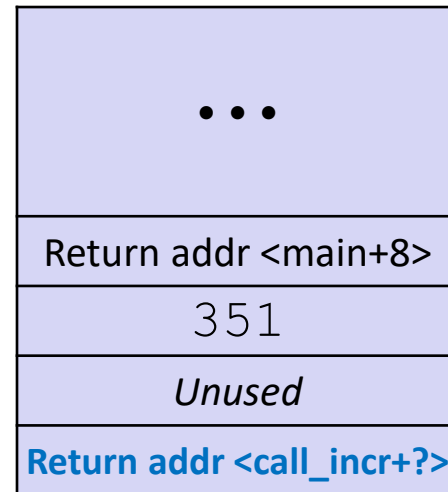
Procedure Call Example (step 3)

```
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
```

Stack Structure



← %rsp

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

Register	Use(s)
%rdi	&v1
%rsi	100
%rax	

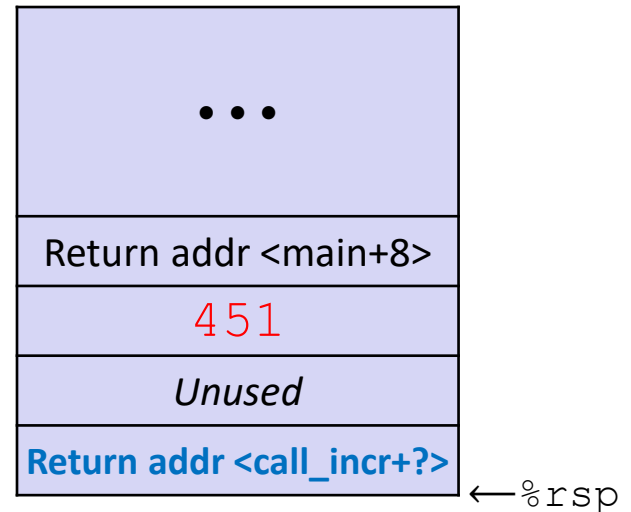
Procedure Call Example (step 4)

```
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
```

Stack Structure



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

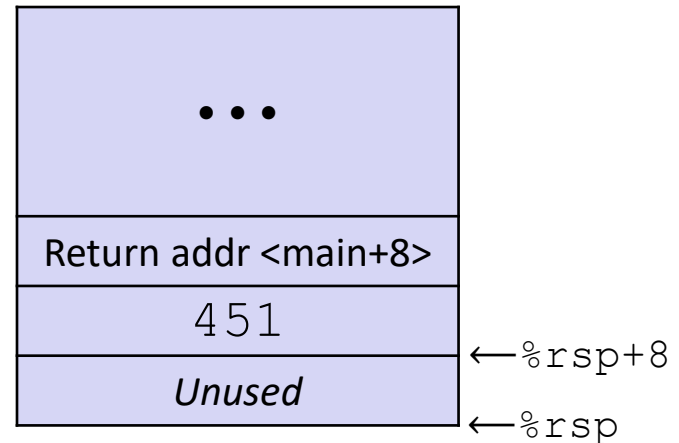
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

Procedure Call Example (step 5)

```
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



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

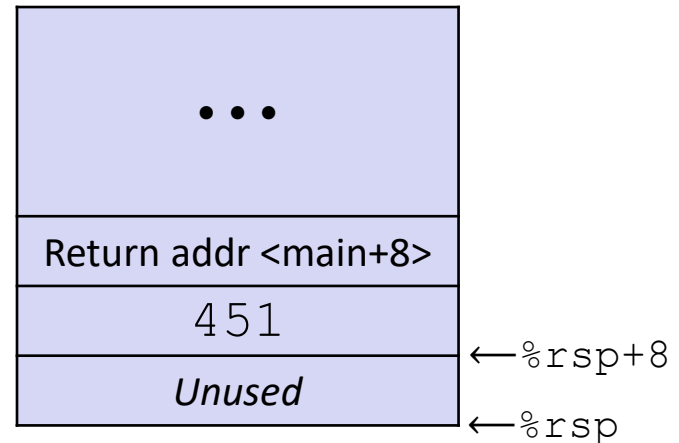
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

Procedure Call Example (step 6)

```
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



← Update $\%rax$ to contain $v1 + v2$

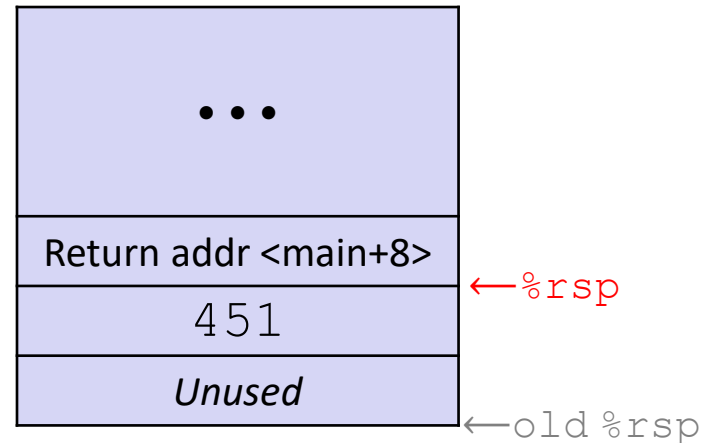
Register	Use(s)
$\%rdi$	$\&v1$
$\%rsi$	451
$\%rax$	451+351

Procedure Call Example (step 7)

```
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



← De-allocate space for local vars

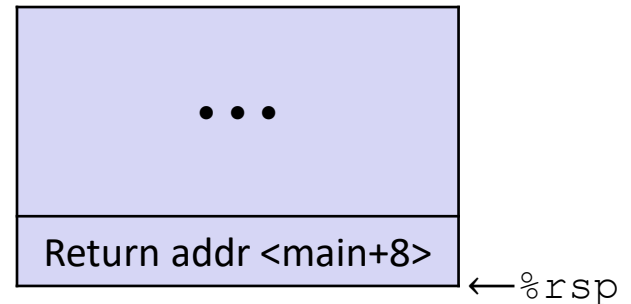
Register	Use(s)
<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	451
<code>%rax</code>	802

Procedure Call Example (step 8)

```
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



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

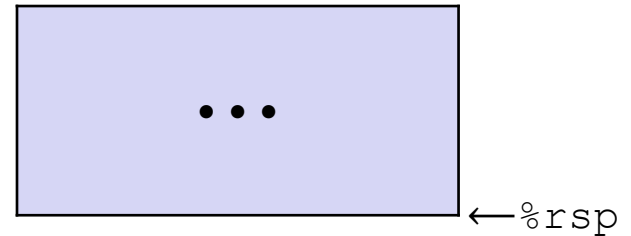
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedure Call Example (step 9)

```
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
```

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)
%rdi	&v1
%rsi	451
%rax	802

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?

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

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

- 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 from the *first floor* (*caller-saved*) before leaving
 - Warn child to leave the *second floor* 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 *second floor* to the backyard shed (*callee-saved*) before the guests trash the house
 - Child cleans up house after the party and moves stuff back to *second floor*
- 3) Parents return home and are satisfied with the state of the house
 - Move valuables back into *first floor* and continue with their lives

x86-64 Linux Registers – **caller-saved** (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

Return value

%rax

Arguments

%rdi

%rsi

%rdx

%rcx

%r8

%r9

Caller-saved
temporaries

%r10

%r11

x86-64 Linux Registers – **callee-saved** (Review)

❖ **%rbx, %r12, %r13, %r14, %r15**

- **Callee**-saved
- **Callee** must save & restore

❖ **%rbp**

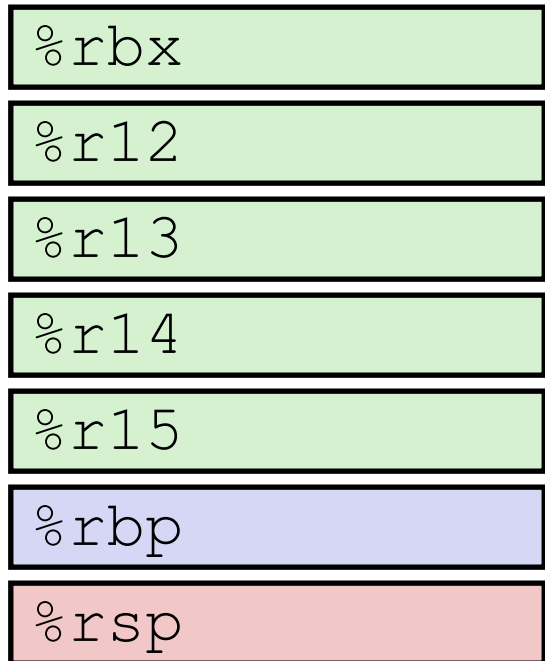
- **Callee**-saved
- **Callee** must save & restore
- May be used as frame pointer
- Can mix & match

❖ **%rsp**

- Special form of **callee** save
- Do not need to explicitly save
- But should be restored to original value upon exit from procedure (i.e. should point to return address when **ret** is executed)

Callee-saved
Temporaries

Special



x86-64 Linux Register Usage (Review)

<code>%rax</code>	Return value - Caller saved
-------------------	------------------------------------

<code>%rbx</code>	Callee saved
-------------------	---------------------

<code>%rcx</code>	Argument #4 - Caller saved
-------------------	-----------------------------------

<code>%rdx</code>	Argument #3 - Caller saved
-------------------	-----------------------------------

<code>%rsi</code>	Argument #2 - Caller saved
-------------------	-----------------------------------

<code>%rdi</code>	Argument #1 - Caller saved
-------------------	-----------------------------------

<code>%rsp</code>	Stack pointer
-------------------	---------------

<code>%rbp</code>	Callee saved
-------------------	---------------------

<code>%r8</code>	Argument #5 - Caller saved
------------------	-----------------------------------

<code>%r9</code>	Argument #6 - Caller saved
------------------	-----------------------------------

<code>%r10</code>	Caller saved
-------------------	---------------------

<code>%r11</code>	Caller Saved
-------------------	---------------------

<code>%r12</code>	Callee saved
-------------------	---------------------

<code>%r13</code>	Callee saved
-------------------	---------------------

<code>%r14</code>	Callee saved
-------------------	---------------------

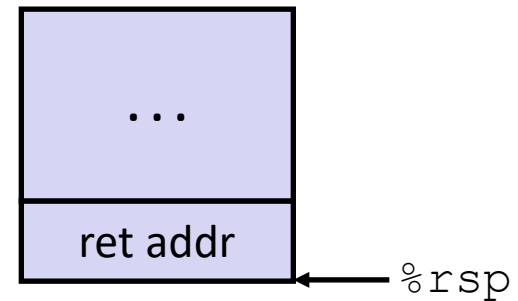
<code>%r15</code>	Callee saved
-------------------	---------------------

Callee-Saved Example (step 1)

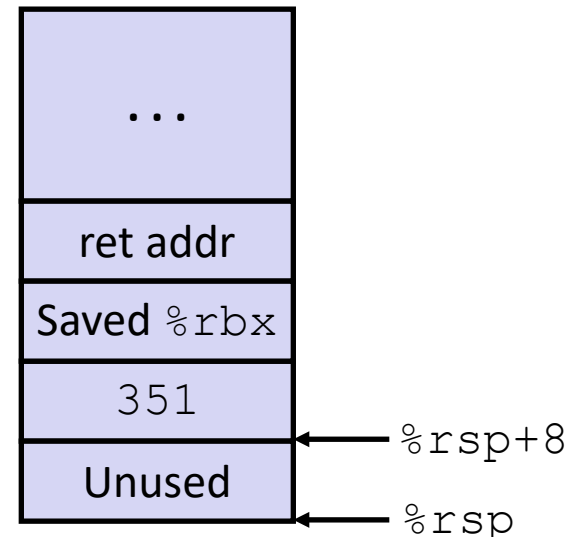
```
long call_incr2(long x) {  
    long v1 = 351;  
    long v2 = increment(&v1, 100);  
    return x + v2;  
}
```

```
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



Resulting Stack Structure

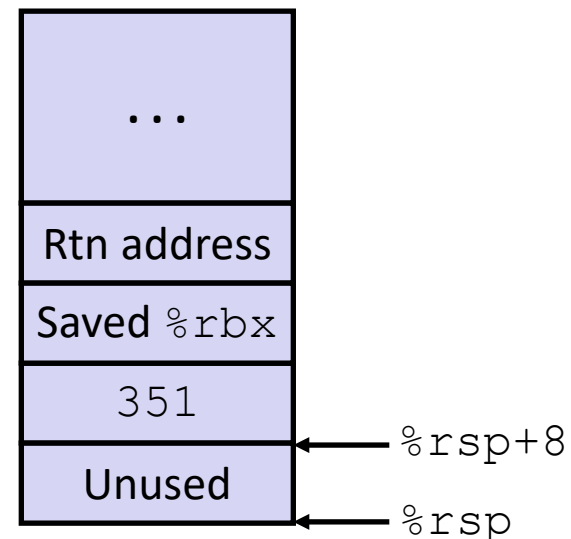


Callee-Saved Example (step 2)

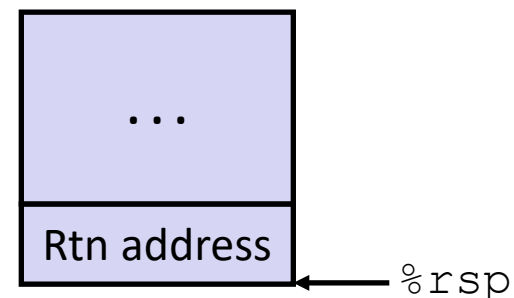
```
long call_incr2(long x) {  
    long v1 = 351;  
    long v2 = increment(&v1, 100);  
    return x + v2;  
}
```

```
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
```

Stack Structure



Pre-return Stack Structure



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

```
/* Recursive popcount */  
long pcount_r(unsigned long x) {  
    if (x == 0)  
        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/47oo11EEv>

- Compiled with `-O1` instead of `-Og` for more natural instruction ordering

```
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: 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
```

Recursive Function: **Callee** Register Save

```
/* 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

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.

```
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
```

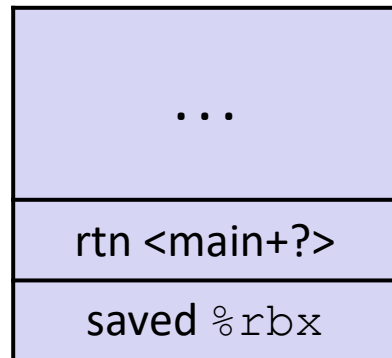
Recursive Function: Call Setup

```

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

```

The Stack



Register	Use(s)	Type
%rdi	x (new)	Argument
%rbx	x (old)	Callee saved

```

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

```

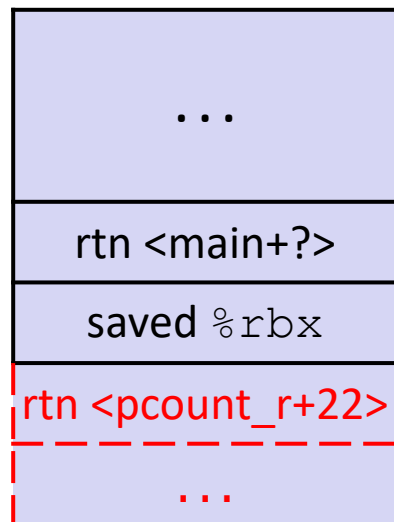
Recursive Function: Call

```

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

```

The Stack



Register	Use(s)	Type
<code>%rax</code>	Recursive call return value	Return value
<code>%rbx</code>	<code>x</code> (old)	Callee saved

```

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

```

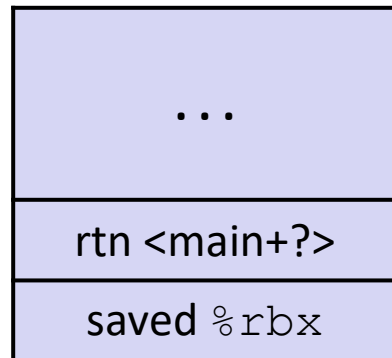
Recursive Function: Result

```

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

```

The Stack



Register	Use(s)	Type
%rax	Return value	Return value
%rbx	x&1	Callee saved

```

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: Completion

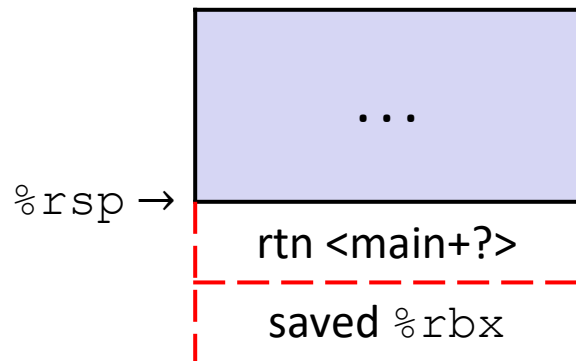
```

/* 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	Return value	Return value
%rbx	Previous %rbx value	Callee restored

The Stack



```

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

```

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

