Sewanee: The University of the South

CS 370: Computer Organization

Procedure Call and Return

Instructor:
Professor Stephen P. Carl

Sewanee: The University of the South

Mechanisms in Procedures

Calling a procedure…

- Passes control to beginning of procedure code
- Returns to the point of call

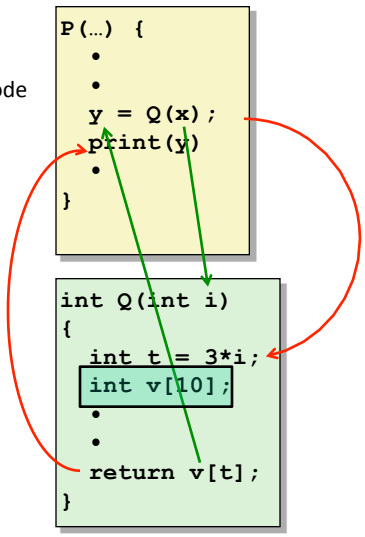
A procedure's arguments…

- Passes data to the procedure
- Procedure returns value back to caller

Memory management typically…

- Allocate new data for this execution
- Deallocate upon return

All such mechanisms are implemented using machine instructions



Sewanee CS

Procedure Control Flow

We use the *system stack* to support procedure call and return

Procedure call: `call label`

- Push *return address* on stack
- Jump to label

The return address is the machine address of the very next instruction after call

- can be seen easily from disassembly

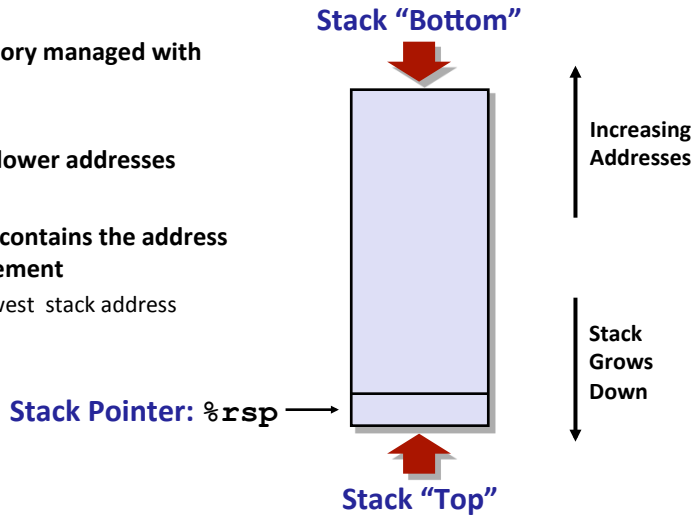
Procedure return: `ret`

- Pop return address from stack
- Jump to address

Sewanee CS

x86-64 Stack

- **Region of memory managed with stack discipline**
- **Grows toward lower addresses**
- **Register `%rsp` contains the address of the "top" element**
  - which is the lowest stack address

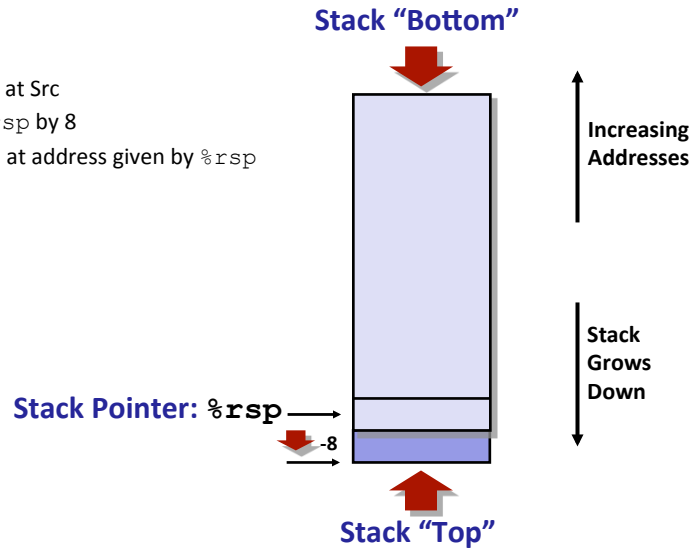


Sewanee CS

x86-64 Stack: Push an item

`pushq Src`

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

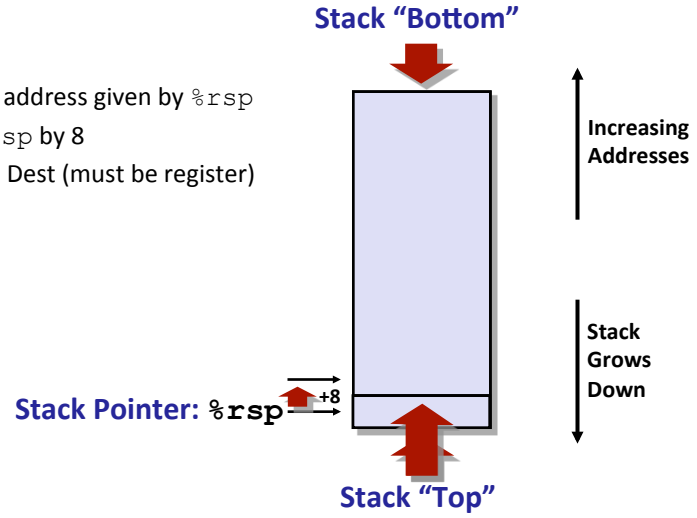


Sewanee CS

x86-64 Stack: Pop item off stack

`popq Dest`

- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at Dest (must be register)



Sewanee: The University of the South

Code Examples

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

```
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
```

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

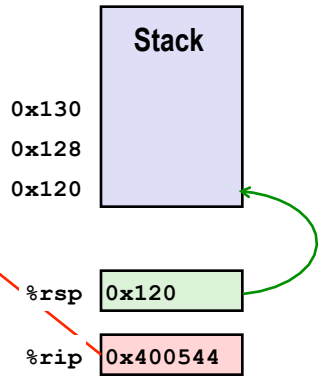
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax        # a
  400553:  imul   %rsi,%rax        # a * b
  400557:  retq                    # Return
```

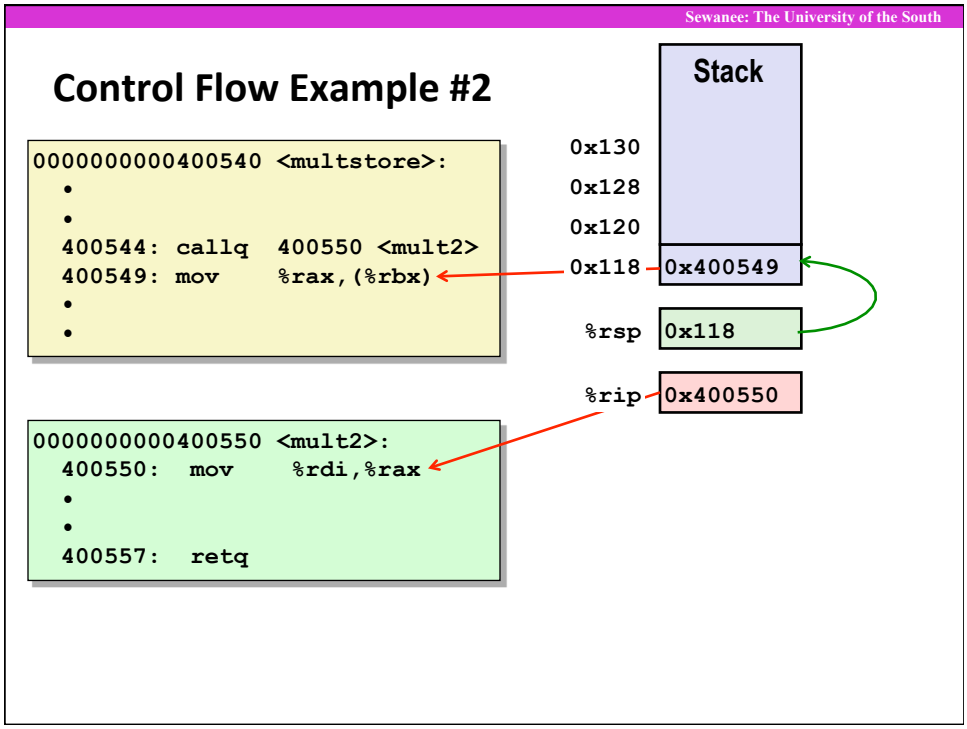
Sewanee: The University of the South

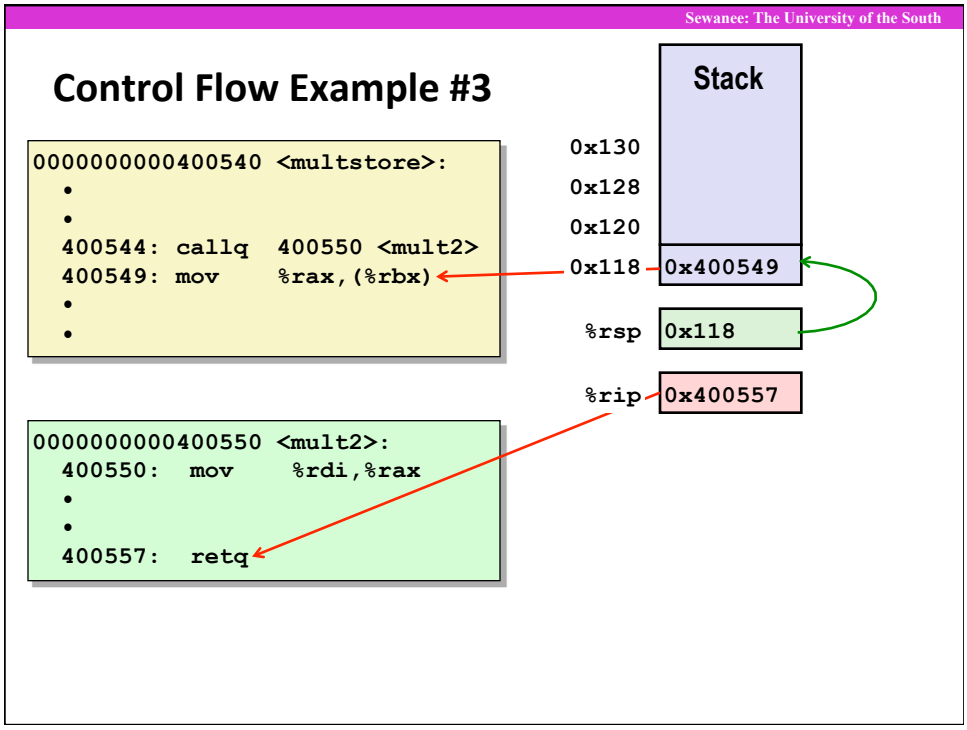
Control Flow Example #1

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

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



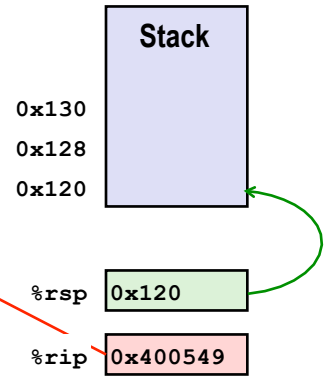

Sewanee: The University of the South
Control Flow Example #2
0000000000400540 <multstore>:
•
•
400544: callq 400550 <mult2>
400549: mov %rax,(%rbx)
•
•
0000000000400550 <mult2>:
400550: mov %rdi,%rax
•
•
400557: retq
Stack
0x130
0x128
0x120
0x118
0x400549
%rsp
0x118
%rip
0x400550


Sewanee: The University of the South
Control Flow Example #3
0000000000400540 <multstore>:
•
•
400544: callq 400550 <mult2>
400549: mov %rax,(%rbx)
•
•
0000000000400550 <mult2>:
400550: mov %rdi,%rax
•
•
400557: retq
Stack
0x130
0x128
0x120
0x118
0x400549
%rsp
0x118
%rip
0x400557

Control Flow Example #4

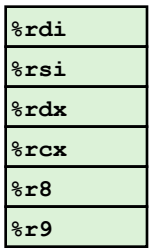
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

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

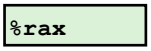


Procedure Data Flow

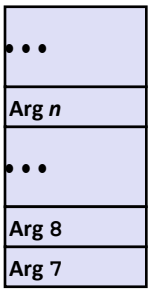
Pass first 6 arguments via registers:



Register with return value



More than 6 arguments? Use Stack:



Only allocate stack space when absolutely necessary

Sewanee: The University of the South

Data Flow Examples

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

```
0000000000400540 <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
  •••
```

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

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax       # a
  400553:  imul   %rsi,%rax       # a * b
  # s in %rax
  400557:  retq                   # Return
```

Sewanee: The University of the South

Stack-Based Languages

…are languages that support recursion

- e.g., C, Java, Lisp, Python
- Code must be "*Reentrant*"
  - There can be multiple *simultaneous* instantiations of a single procedure
- Need some place to store the *state* of each instantiation of a procedure. State is:
  - Arguments
  - Local variables
  - Return address

This calls for *stack discipline*

- Procedure's state only needed from the **call** until it **returns**
- Callee **always** returns before caller does (well…99% of the time)

Stack is allocated in *Frames*

- A frame is the state for a single procedure instantiation

Sewanee CS

Call Chain Example

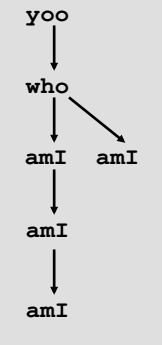
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

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

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

Procedure `amI()` is recursive

Example Call Chain



Sewanee CS

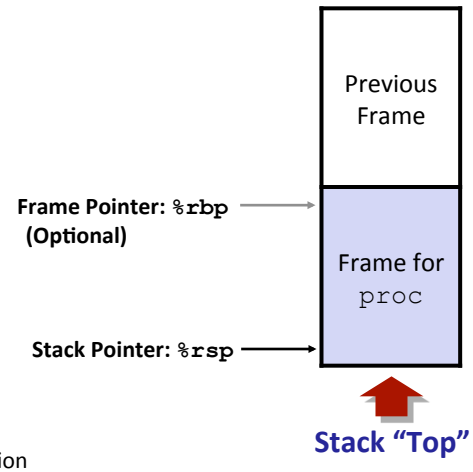
Stack Frames

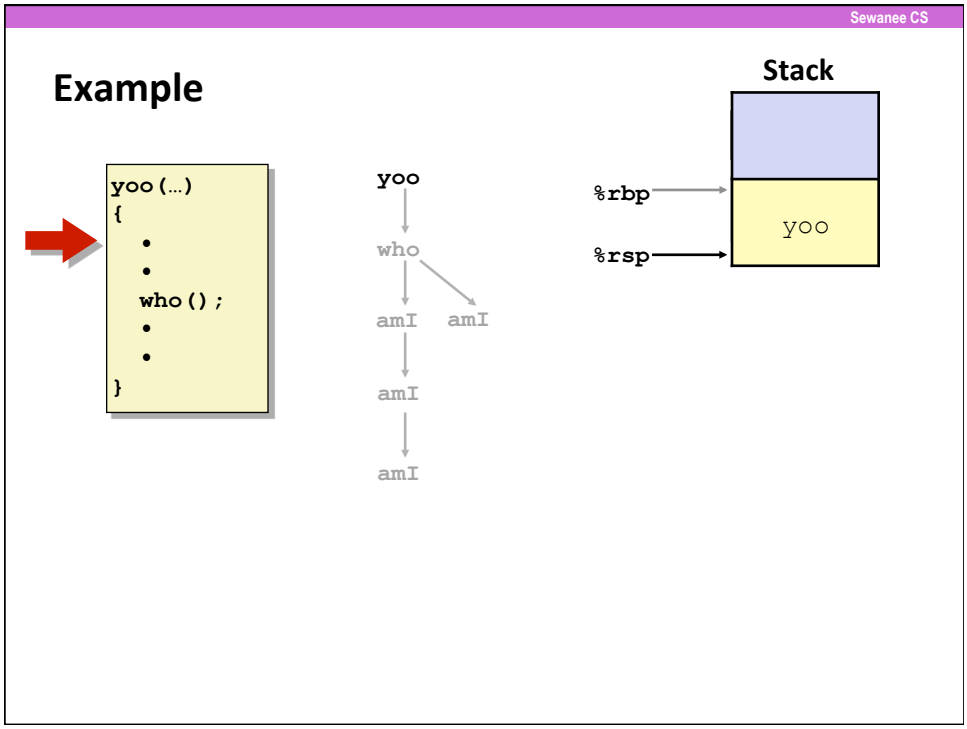
Contents

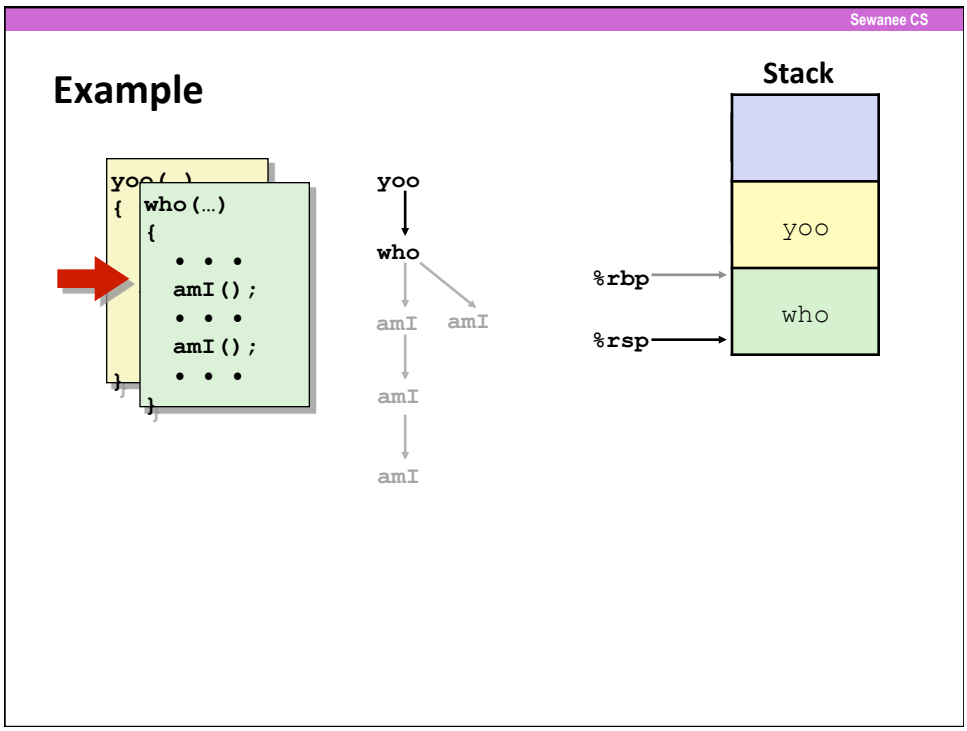
- Return information
- Local storage (if needed)
- Temporary space (if needed)

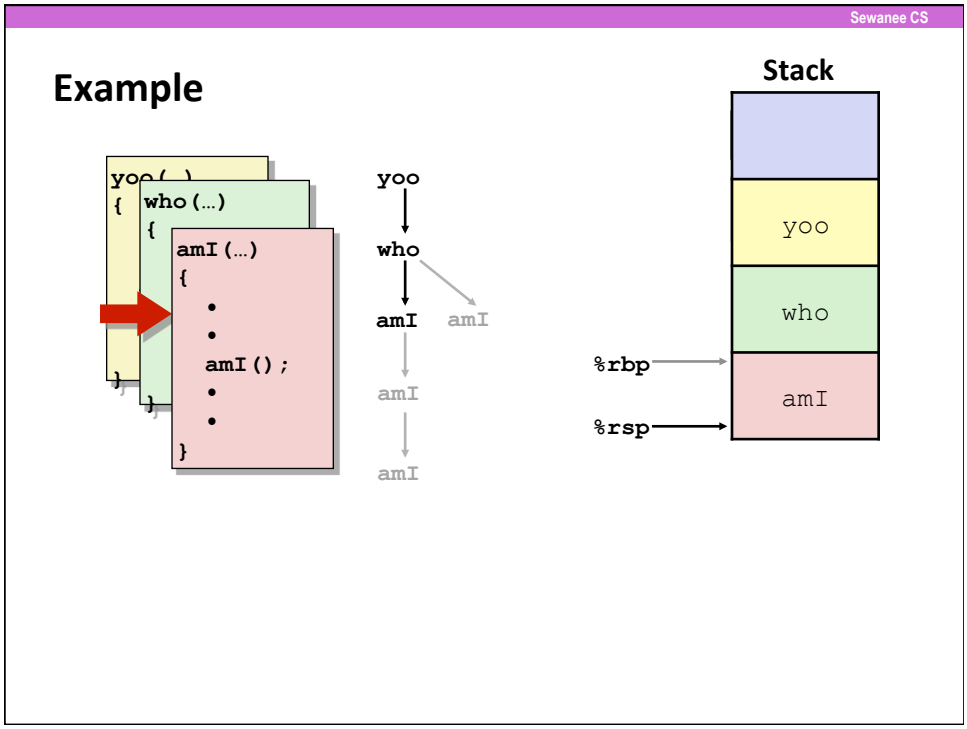
Management

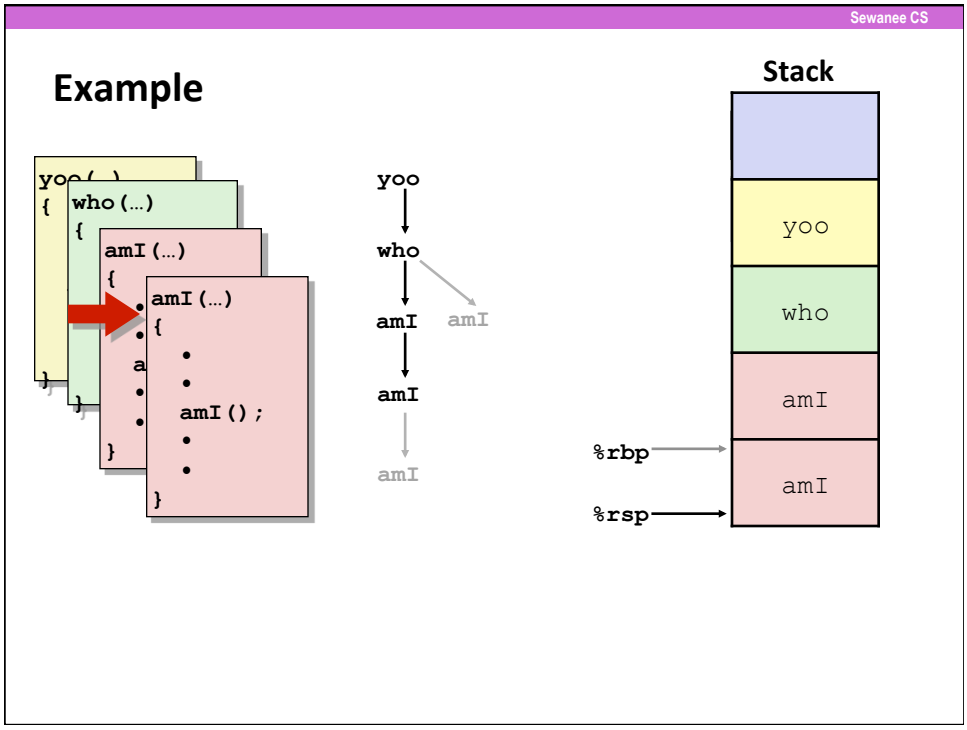
- Allocated at start of procedure
  - "Set-up" code
  - Includes push by **`call`** instruction
- Deallocated when return
  - "Finish" code
  - Includes pop by **`ret`** instruction

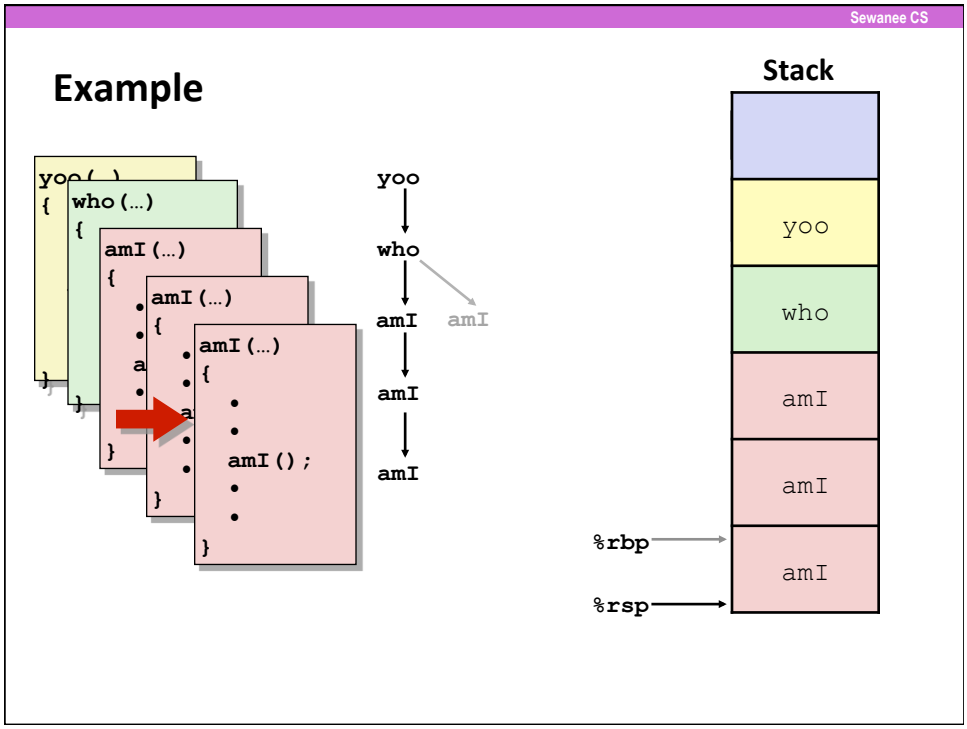


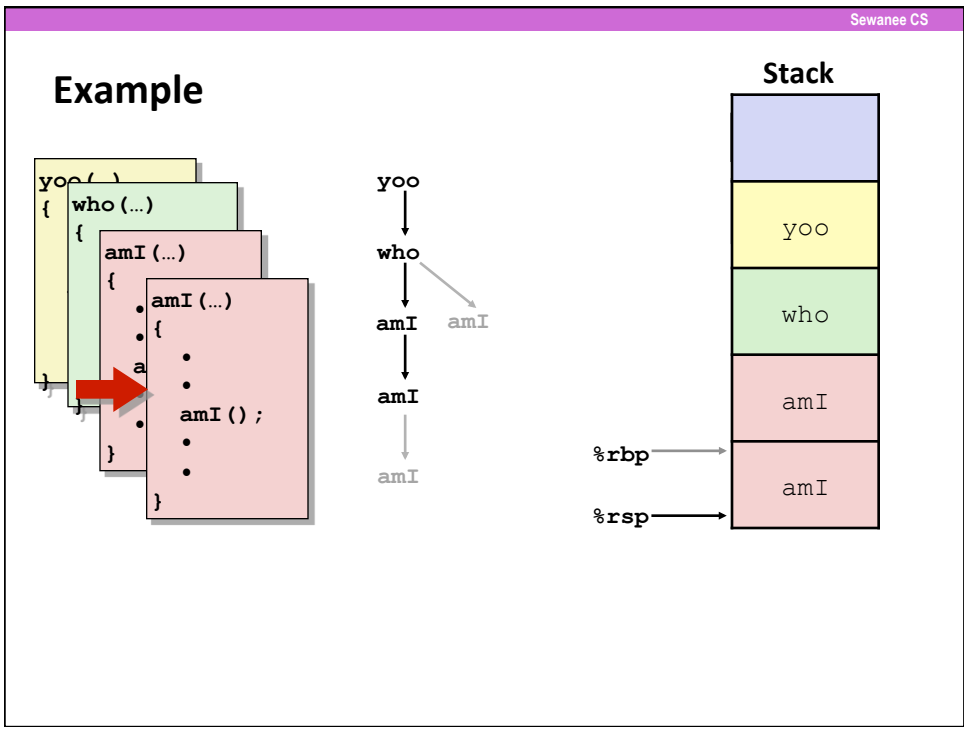

Sewanee CS
Example
yoo(…)
{
•
•
who();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
%rbp
%rsp
yoo

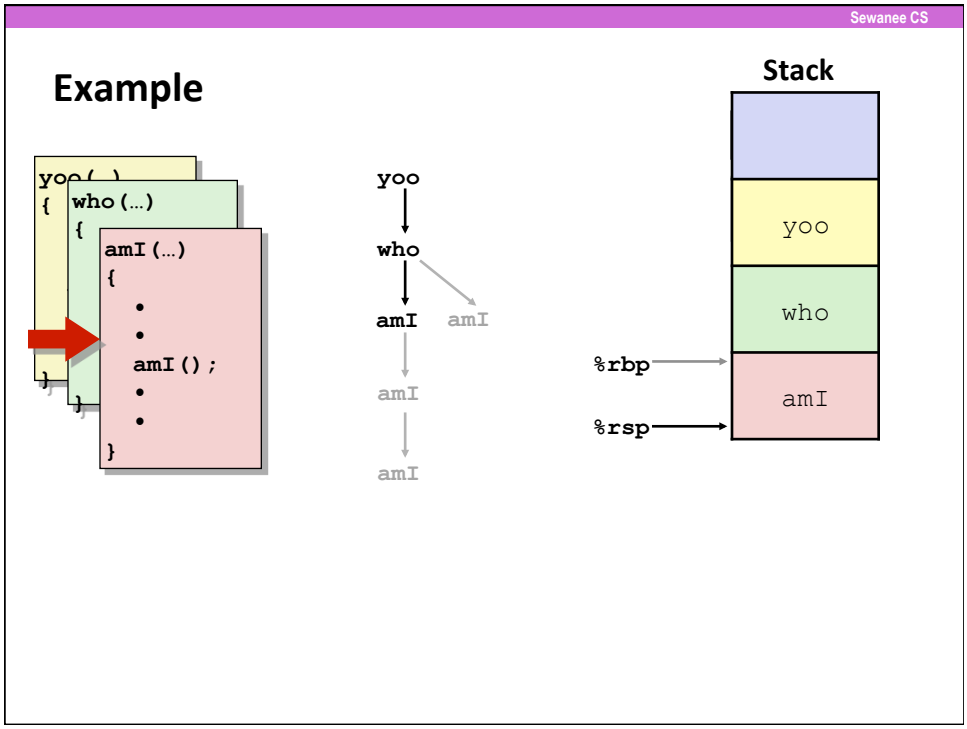

Sewanee CS
Example
who(…)
{
• • •
amI();
• • •
amI();
• • •
}
yoo
who
amI
amI
amI
amI
Stack
%rbp
%rsp
yoo
who

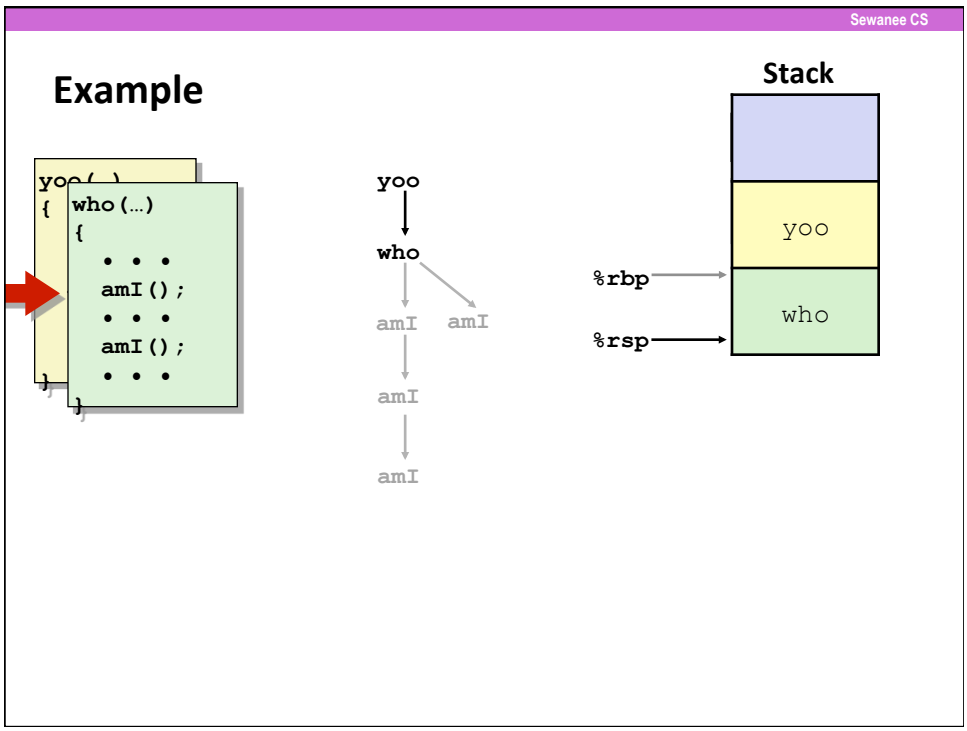

Sewanee CS
Example
Stack
who(…)
{
amI(…)
{
amI();
}
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
%rbp
%rsp

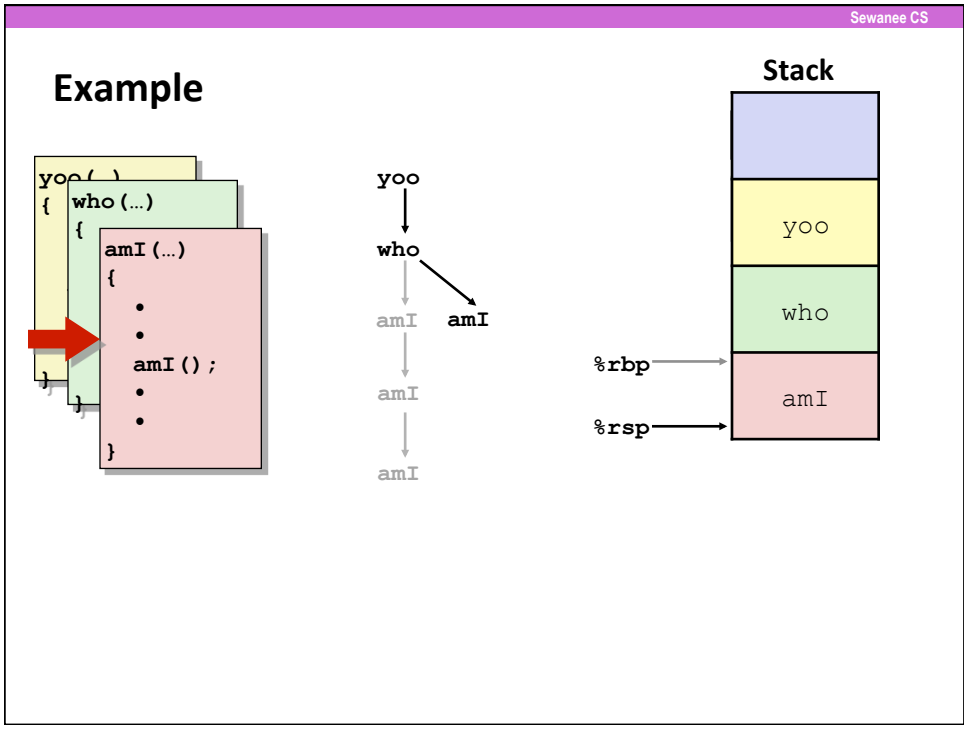

Sewanee CS
Example
Stack
who(…)
{
amI(…)
{
amI(…)
{
amI();
}
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
amI
%rbp
%rsp

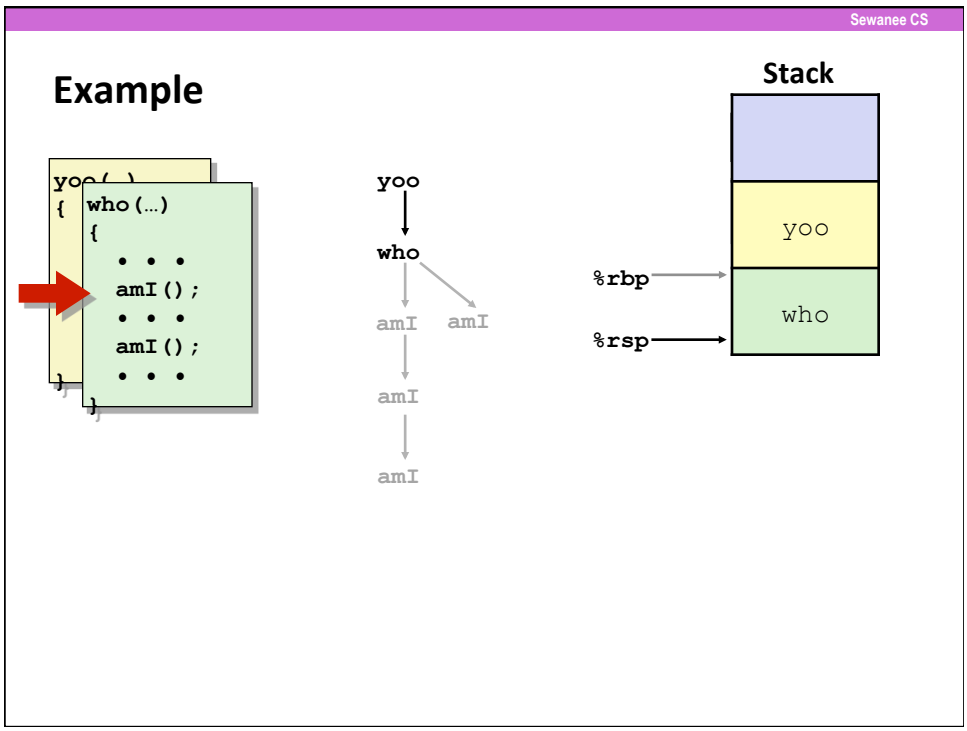

Sewanee CS
Example
Stack
who (…)
amI (…)
amI (…)
amI (…)
amI ();
yoo
who
amI
amI
amI
amI
yoo
who
amI
amI
amI
%rbp
%rsp


Sewanee CS
Example
Stack
who (…)
amI (…)
amI (…)
amI ();
yoo
who
amI
amI
amI
yoo
who
amI
amI
%rbp
%rsp


Sewanee CS
Example
yoo ( )
{
who (…)
{
amI (…)
{
•
•
amI ();
•
•
}
}
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
%rbp
%rsp

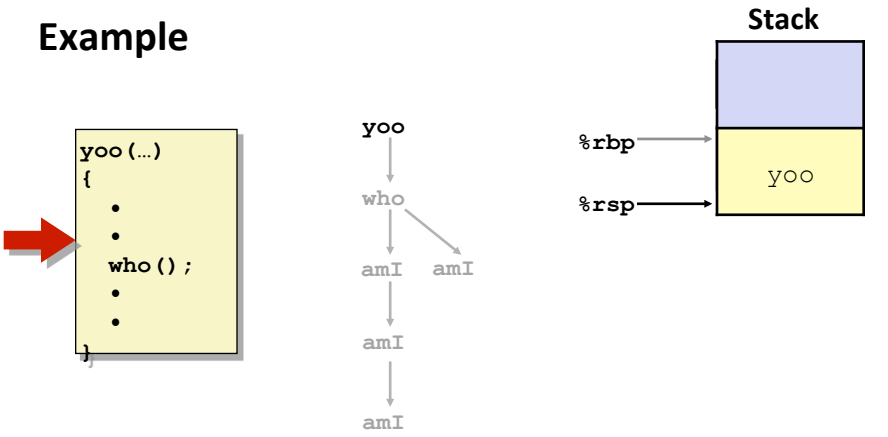

Sewanee CS
Example
yoo ( )
{
who (…)
{
• • •
amI ();
• • •
amI ();
• • •
}
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
%rbp
%rsp


Sewanee CS
Example
yoo ( )
{
who (…)
{
amI (…)
{
•
•
amI ();
•
•
}
}
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
%rbp
%rsp


Sewanee CS
Example
yoo ( )
{
who (…)
{
• • •
amI ();
• • •
amI ();
• • •
}
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
%rbp
%rsp

Sewanee CS

Example



Sewanee CS

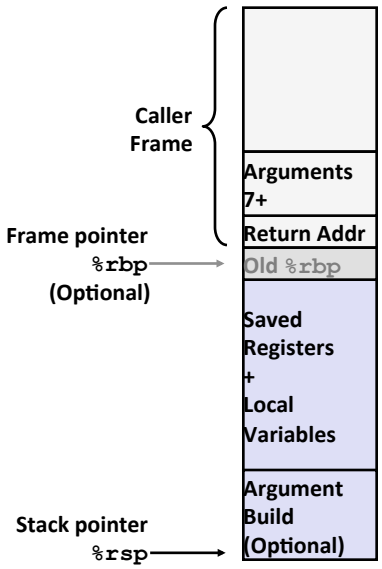
x86-64/Linux Stack Frame

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



Sewanee CS

Example: `incr`

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

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

Register	Use(s)
`%rdi`	Argument `p`
`%rsi`	Argument `val`, `y`
`%rax`	`x`, Return value

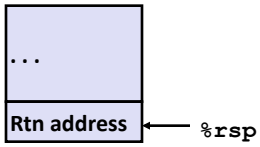
Sewanee CS

Example: Calling `incr` #1

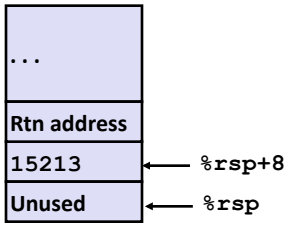
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



Resulting Stack Structure



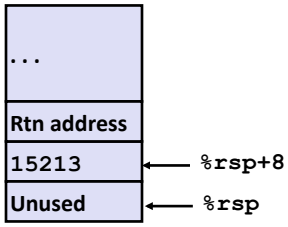
Sewanee CS

Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rdi	&v1
%rsi	3000

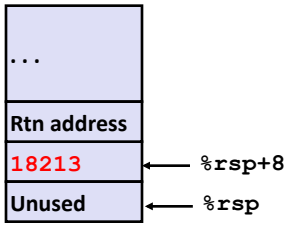
Sewanee CS

Example: Calling `incr` #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rdi	&v1
%rsi	3000

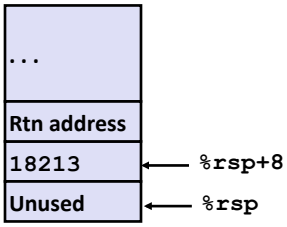
Sewanee CS

Example: Calling `incr` #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

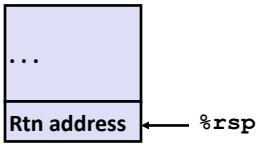
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rax	Return value

Updated Stack Structure



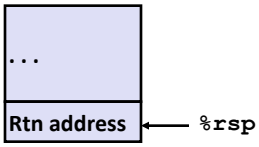
Sewanee CS

Example: Calling `incr` #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

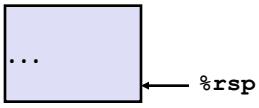
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



Register	Use(s)
%rax	Return value

Final Stack Structure



Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the *caller*
- `who` is the *callee*

Can register be used for temporary storage?

- Note that contents of register `%edx` overwritten by `who`

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

```
who:
    • • •
    addq $91125, %rdx
    • • •
    ret
```

Sewanee: The University of the South

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 Save" - Caller saves temporary values in its frame before calling

"Callee Save" -

- Callee saves temporary values in its frame before using
- Callee restores them before returning to caller

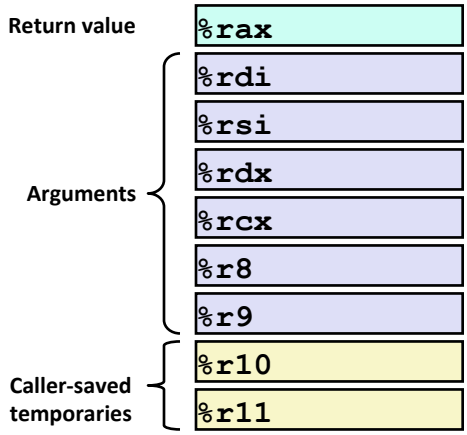
Sewanee CS

x86-64 Linux Register Usage #1

`Register %rax`
- Holds return value
- Must be caller-saved
- Can be modified by procedure

`Registers %rdi`, ..., `%r9`
- Holds procedure arguments
- Also caller-saved
- Can be modified by procedure

`Registers %r10`, `%r11`
- Caller-saved
- Can be modified by procedure



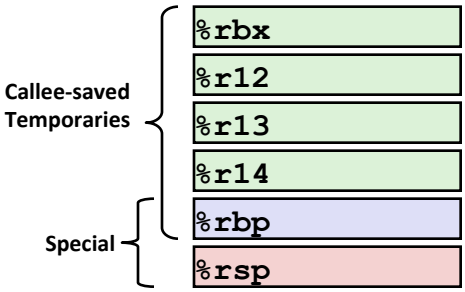
Sewanee CS

x86-64 Linux Register Usage #2

`For %rbx`, `%r12`, `%r13`, `%r14`
- Callee-saved
- Callee must save & restore

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

`Register %rsp`
- Special form of callee save
- Restored to original value upon exit from procedure



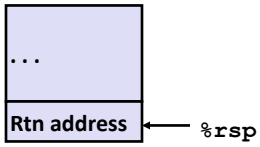
Sewanee CS

Callee-Saved Example #1

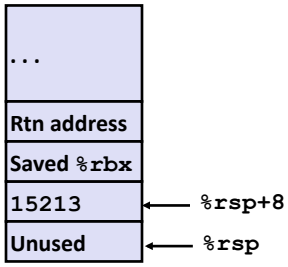
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure



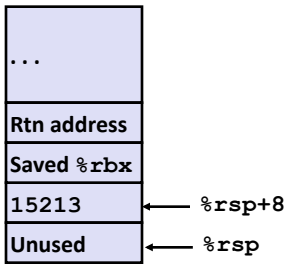
Sewanee CS

Callee-Saved Example #2

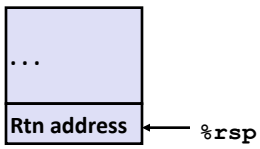
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure



Pre-return Stack Structure



Sewanee: The University of the South

X86-64 Procedure Summary

The Stack Makes Procedure Calls Work

- Private storage for each *instance* of procedure call
  - Instantiations don't clobber each other
  - Addressing of locals + arguments can be relative to stack positions
- Managed by stack discipline
  - Procedures return in inverse order of calls

Recursion handled by normal calling conventions

- Safely store values in local stack frame and callee-saved registers
- Put function arguments at top of stack
  - Caller / Callee save
  - `%ebp` and `%esp`
- Return result in `%rax`