Machine Level Programming: Procedures

CSCI 237: Computer Organization
16th Lecture, Wednesday, October 15

Kelly Shaw

1

1

Administrative Details

- Lab #3 due Thursday at 11pm
  - Any questions?
- Read CSAPP 3.8
- Midterm in lab on Wednesday
  - Closed book and closed notes
  - Material through the end of this week
  - Sample mid-term and solutions on Glow for practice
- Review session?

2

2

Last Time: Machine-Level Programming: Control

- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements

3

3

Today: Machine-Level Programming: Procedures

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

4

4

Practice On Your Own

```
void fcn(long *array, long i, long j)
{
   array[i-4] = array[j] + array[6] + 1;
}
```

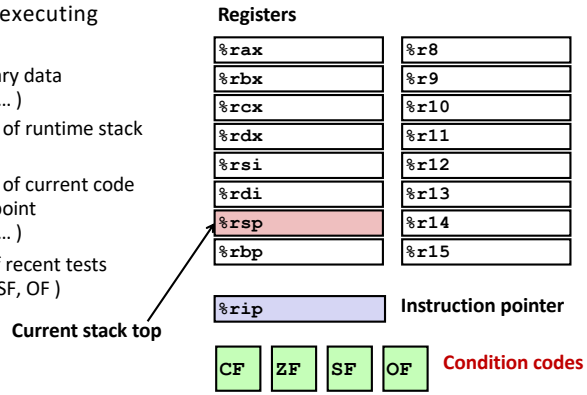
```
fcn:
movl $1, %ecx
addq _____, %rcx
movq _____, %r8
addq %r8, %rcx
leaq _____, %r9
movq %rcx, ____
ret
```

5

5

Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (`%rax`, …)
  - Location of runtime stack (`%rsp`)
  - Location of current code control point (`%rip`, …)
  - Status of recent tests (CF, ZF, SF, OF)



6

6

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
  - De-allocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

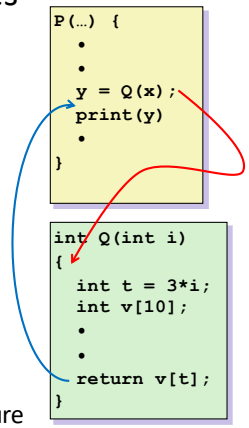
```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

7

7

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
  - De-allocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

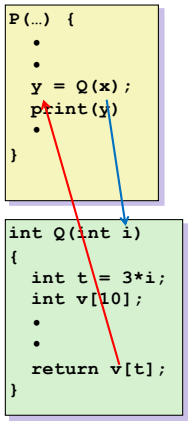


8

8

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
  - De-allocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

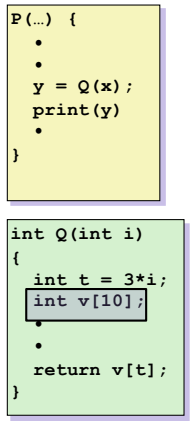


9

9

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
  - De-allocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

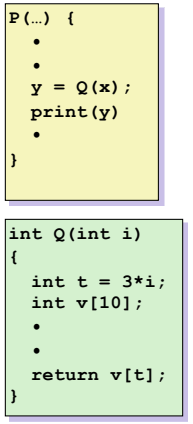


10

10

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
  - De-allocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

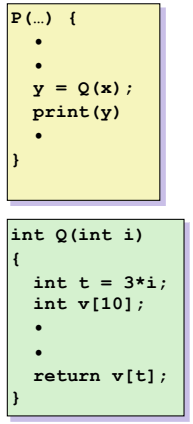


11

11

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
  - De-allocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

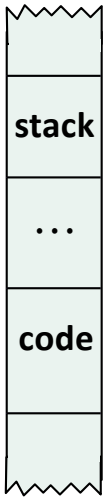


12

12

x86-64 Stack

- Region of memory managed with stack discipline
- Memory viewed as array of bytes
- Different regions have different purposes

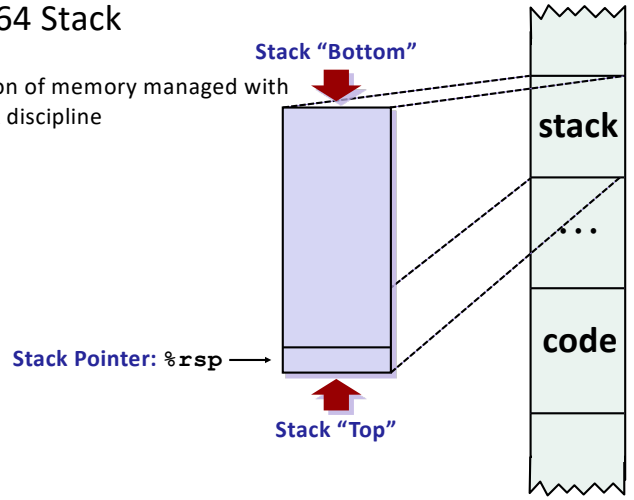


13

13

x86-64 Stack

- Region of memory managed with stack discipline

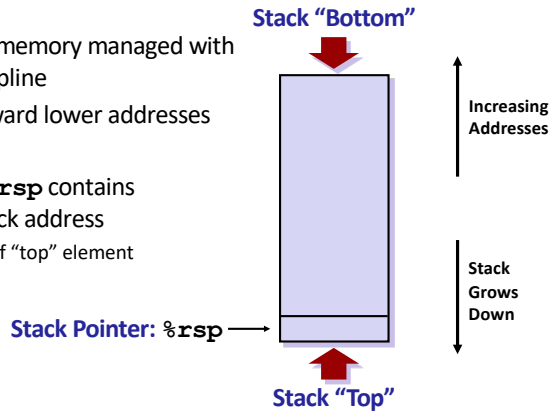


14

14

x86-64 Stack

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

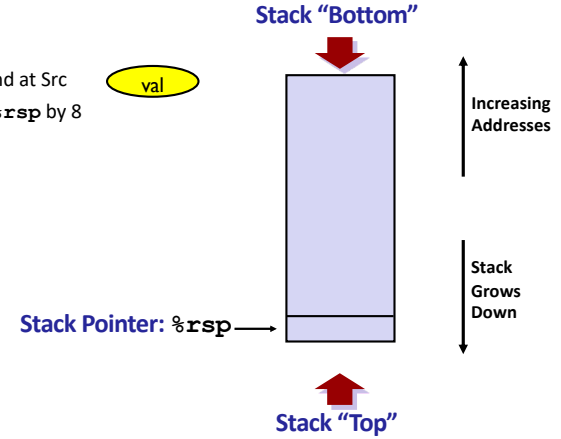


15

15

x86-64 Stack: Push

- `pushq` Src
  - Fetch operand at Src
  - Decrement `%rsp` by 8

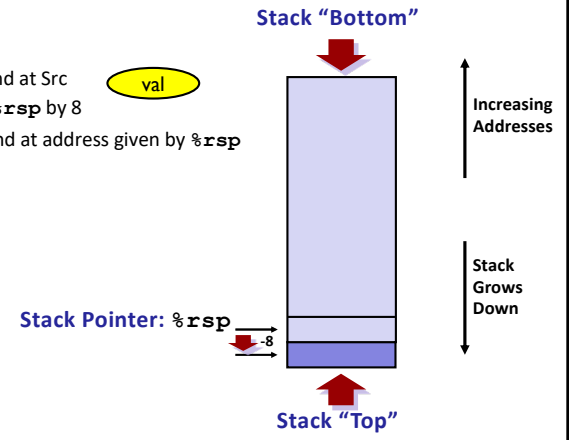


16

16

x86-64 Stack: Push

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

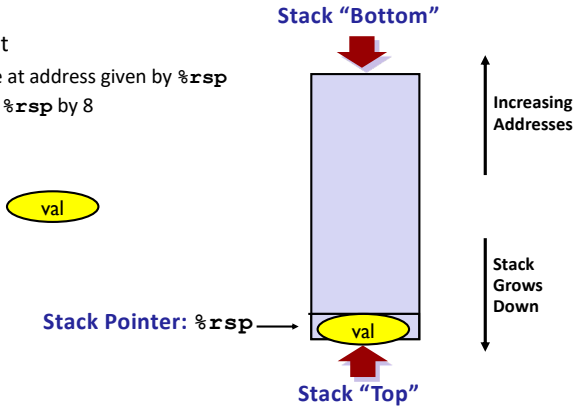


17

17

x86-64 Stack: Pop

- `popq` Dest
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8

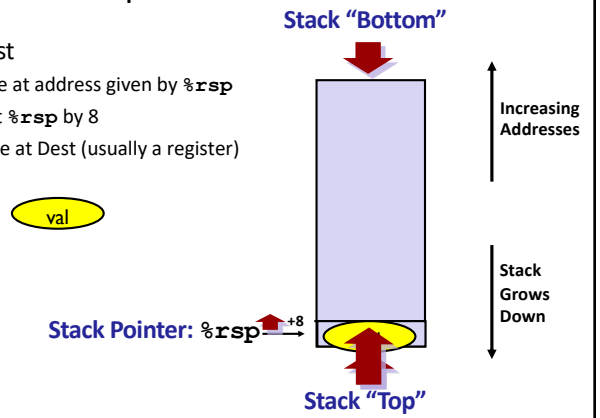


18

18

x86-64 Stack: Pop

- `popq` Dest
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)

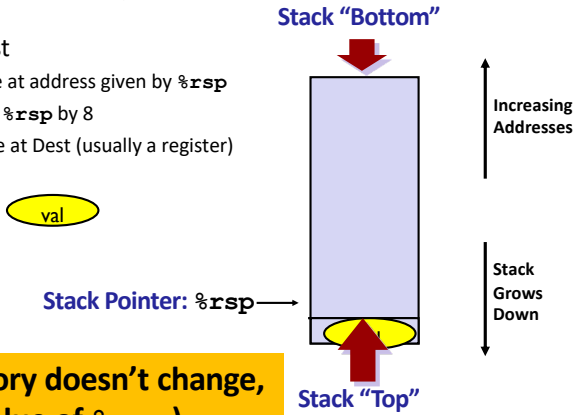


19

19

x86-64 Stack: Pop

- `popq` Dest
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)



(The memory doesn't change, only the value of `%rsp`)

20

20

Today: Machine-Level Programming: Procedures

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

21

21

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
```

22

22

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: **`call label`**
  - (you will see `callq label` in gdb output)
  - Does two things:
    - Pushes return address on stack
    - Jumps to label
- Return address:
  - Address of the next instruction right after call
  - (Example on next slide)
- Procedure return: **`ret`**
  - Does two things (inverse of call):
    - Pop address from stack
    - Jump to address

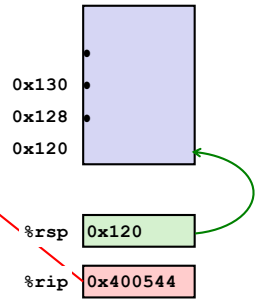
23

23

Control Flow Example #1

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

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

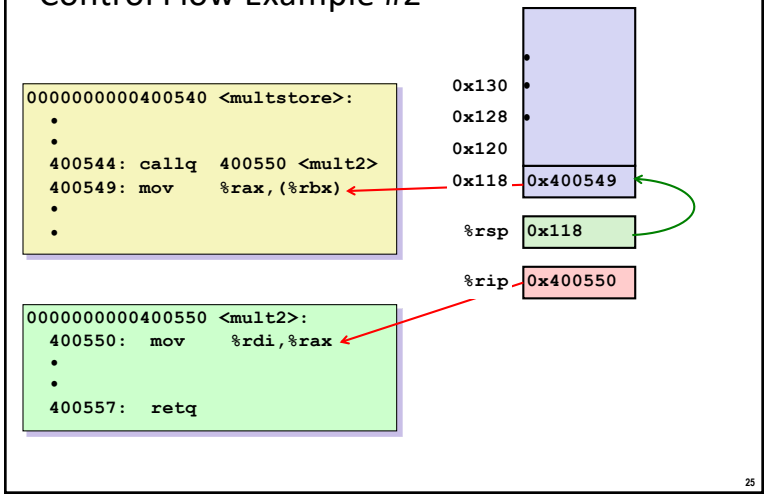


`callq 400550`: Push return addr onto stack, jump to label `<mult2>`.

24

24

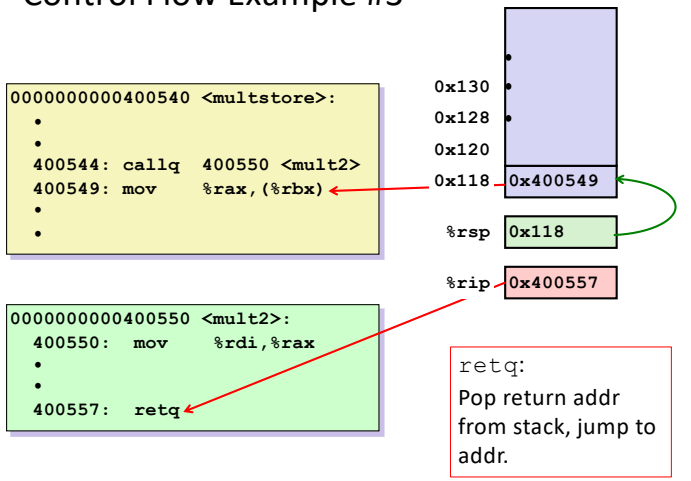
Control Flow Example #2



25

25

Control Flow Example #3

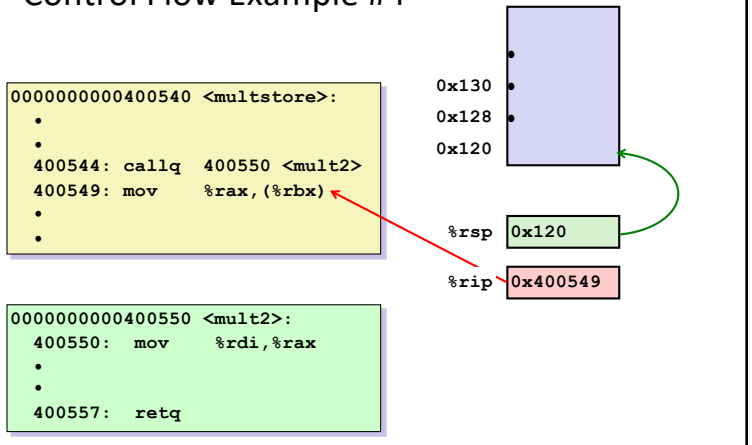


retq:
Pop return addr from stack, jump to addr.

26

26

Control Flow Example #4



27

27

Today: Machine-Level Programming: Procedures

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

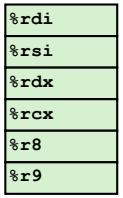
28

28

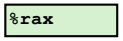
Procedure Data Flow

Registers

- First 6 arguments



- Return value



Stack

- Only allocate stack space when needed (e.g., more than 6 integral arguments)!

29

29

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
```

30

30

Today: Machine-Level Programming: Procedures

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

31

31

Stack-Based Languages

- Languages that support recursion
  - e.g., C, Java
  - Code must be "Reentrant"
    - There can be multiple simultaneous instantiations of a single procedure
  - We need some place to store the **state** of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- Stack discipline
  - The state for a given procedure is only needed for a limited time
    - From when the procedure is called to when the return occurs
  - **Callee** returns before **caller** does
- Stack allocated in **Frames**
  - A frame stores the state for single procedure instantiation

32

32

Call Chain Example

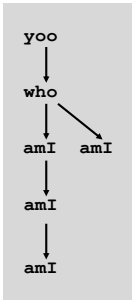
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

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

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

Procedure `amI()` is recursive

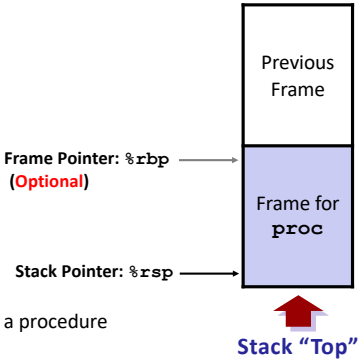
Example Call Chain



33

Stack Frames

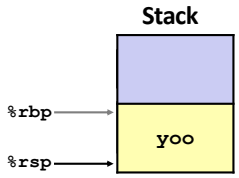
- Contents:
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- Management
  - Space is allocated when entering a procedure
    - "Set-up" code
    - Includes push by `call` instruction
  - Space is deallocated when a return occurs
    - "Finish" code
    - Includes pop by `ret` instruction



34

Example

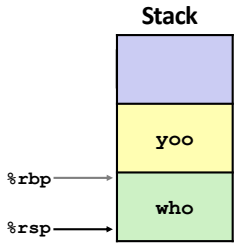
```
yoo(…)
{
  •
  who();
  •
  •
}
```



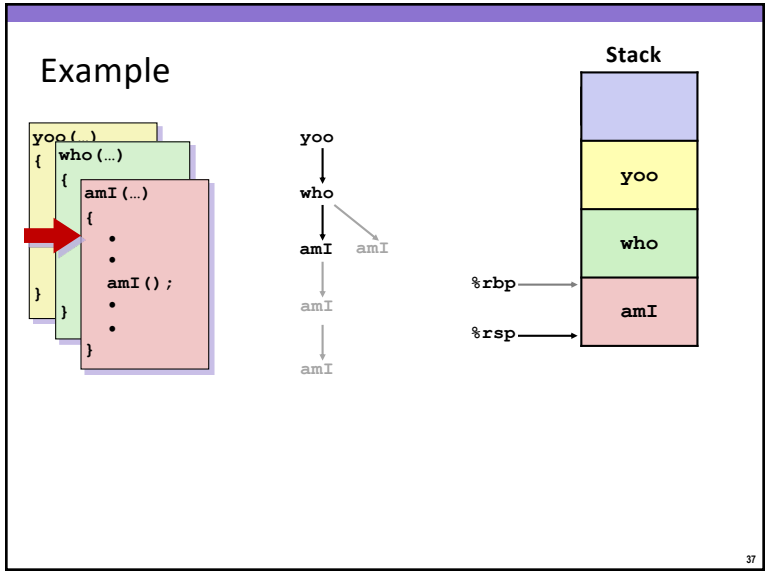
35

Example

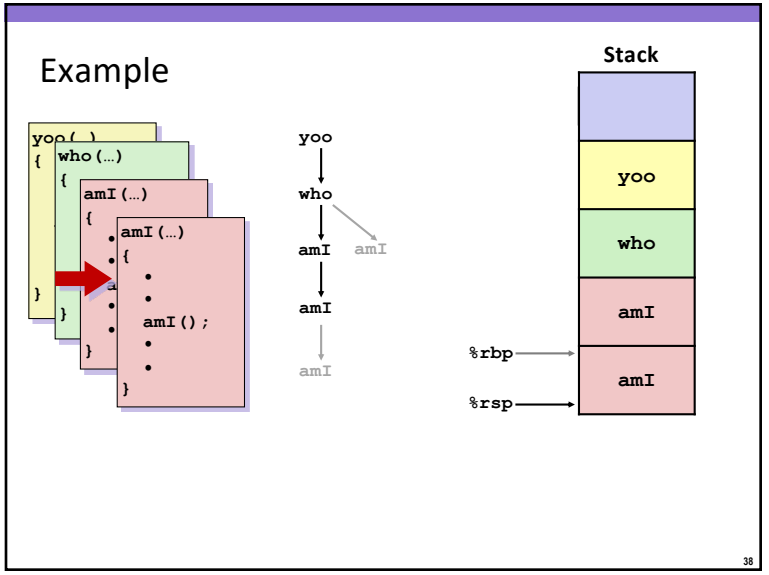
```
yoo(…)
{
  who(…)
  {
    • • •
    amI();
    • • •
    amI();
    • • •
  }
}
```



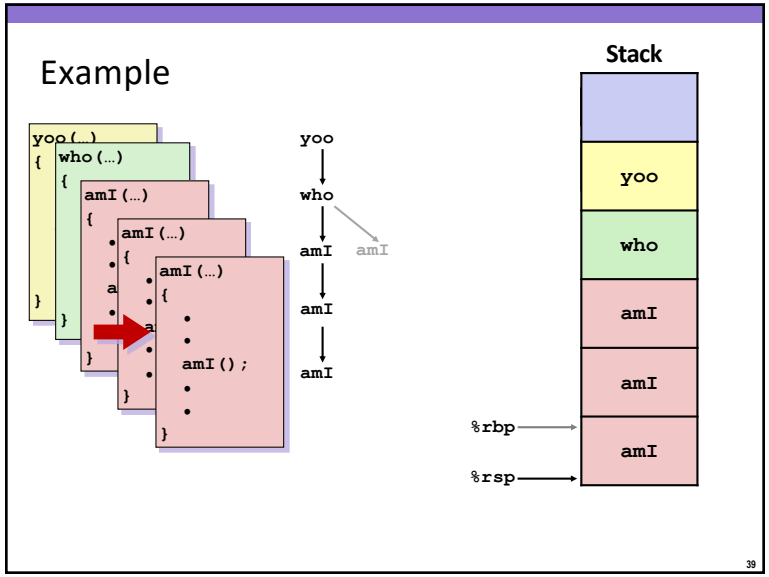
36


Example
Stack
yoo ( )
{
who (…)
{
amI (…)
{
amI ();
}
}
}
yoo
who
amI
amI
amI
yoo
who
amI
%rbp
%rsp
37

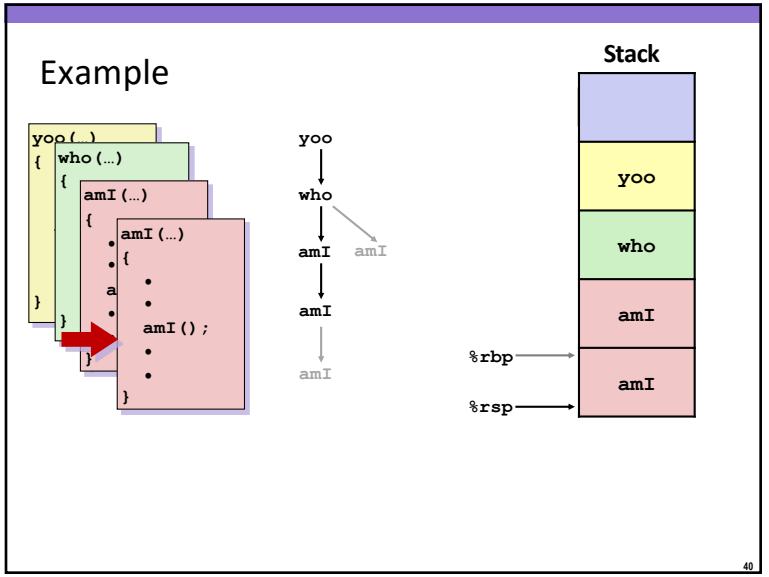
37


Example
Stack
yoo ( )
{
who (…)
{
amI (…)
{
amI ();
}
}
}
yoo
who
amI
amI
amI
yoo
who
amI
%rbp
%rsp
38

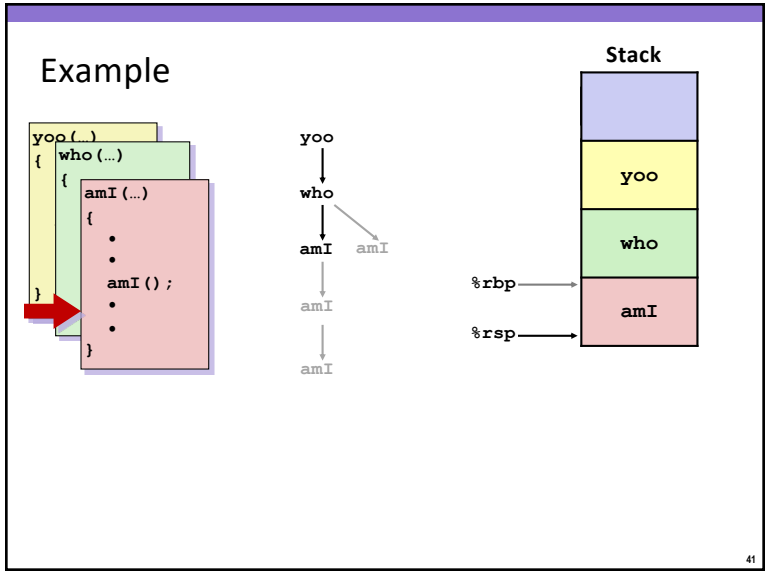
38


Example
Stack
yoo ( )
{
who (…)
{
amI (…)
{
amI (…)
{
amI (…)
{
amI ();
}
}
}
}
yoo
who
amI
amI
amI
yoo
who
amI
amI
amI
%rbp
%rsp
39

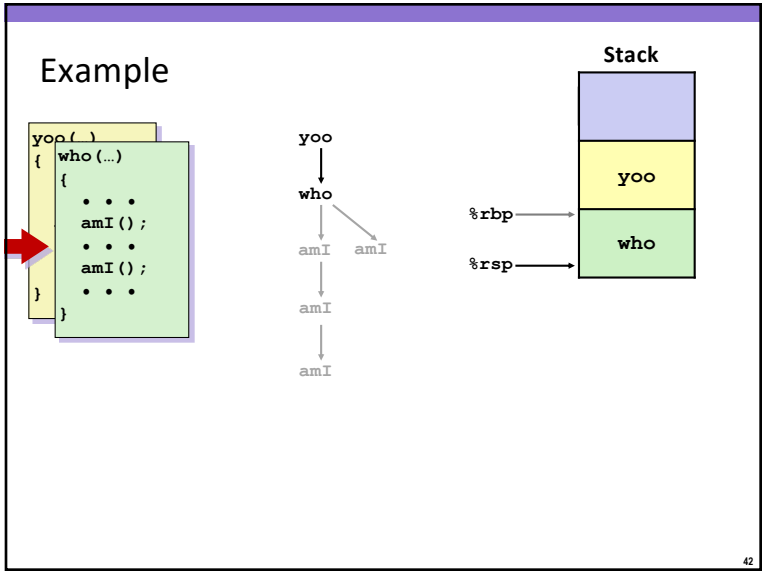
39


Example
Stack
yoo ( )
{
who (…)
{
amI (…)
{
amI (…)
{
amI ();
}
}
}
yoo
who
amI
amI
amI
yoo
who
amI
amI
amI
%rbp
%rsp
40

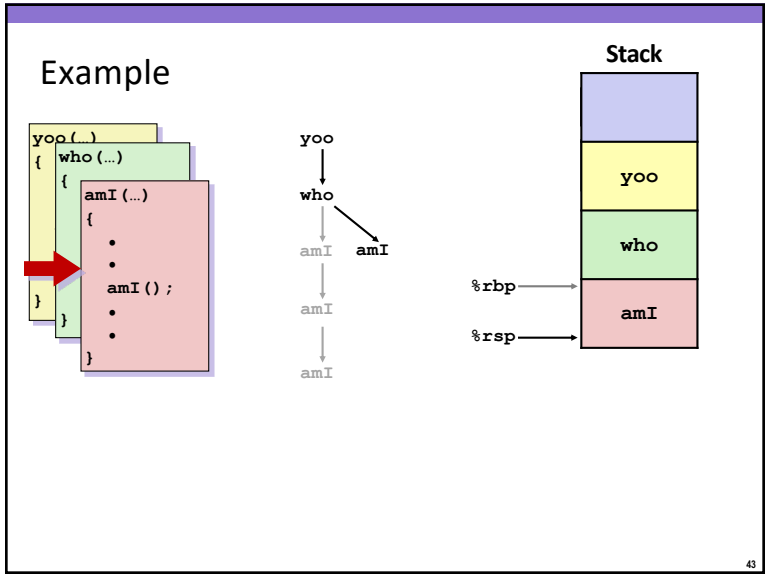
40


Example
Stack
yoo (…)
{
who (…)
{
amI (…)
{
amI ();
}
}
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
%rbp
%rsp
41

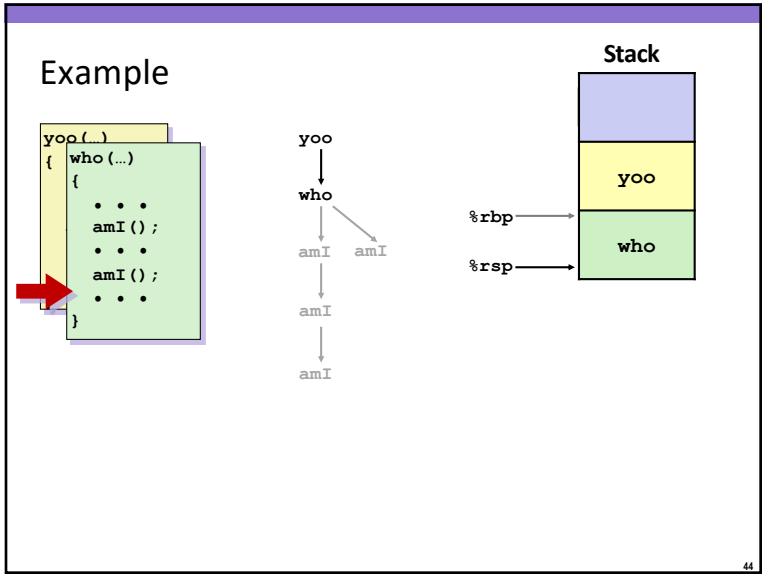
41


Example
Stack
yoo (…)
{
who (…)
{
amI ();
amI ();
}
}
yoo
who
amI
amI
amI
amI
yoo
who
%rbp
%rsp
42

42

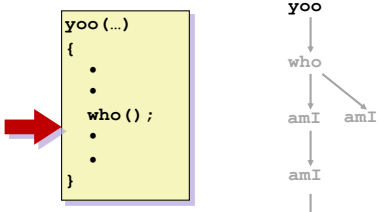

Example
Stack
yoo (…)
{
who (…)
{
amI (…)
{
amI ();
}
}
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
%rbp
%rsp
43

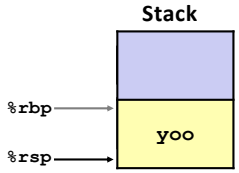
43


Example
Stack
yoo (…)
{
who (…)
{
amI ();
amI ();
}
}
yoo
who
amI
amI
amI
amI
yoo
who
%rbp
%rsp
44

44

Example



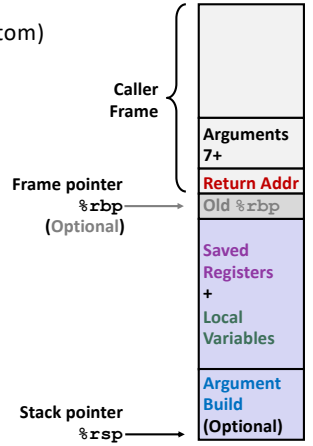


45

45

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



46

46