Systems Programming

Machine-Level Programming III: Procedures

Byoungyoung Lee
Seoul National University
byoungyoung@snu.ac.kr
https://lifeasageek.github.io

Lecture slides are prepared based on materials provided by CSAPP authors.

Objectives

- **Basic functionality of the pairs**
  - push / pop
  - call / ret
- **Students should be able to identify the different components of a stack**
  - return address, arguments, saved registers, local variables
- **Explain the difference between callee and caller save registers**
- **Explain how a stack permits functions to be called recursively / re-entrant**

Today

- **Procedures (Function)**
  - **Mechanisms**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data

Mechanisms in Procedures

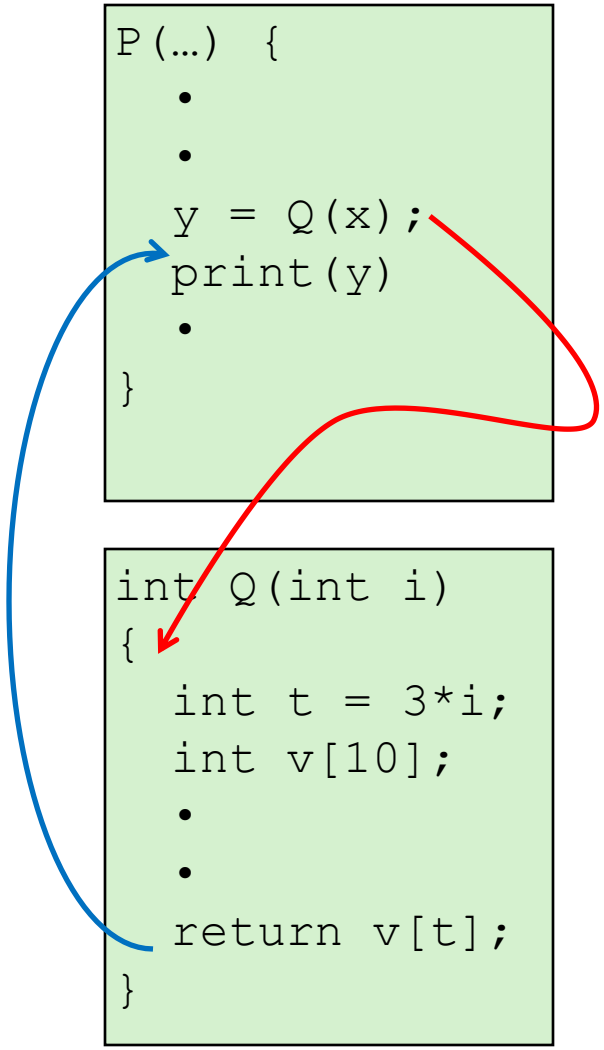
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**

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

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

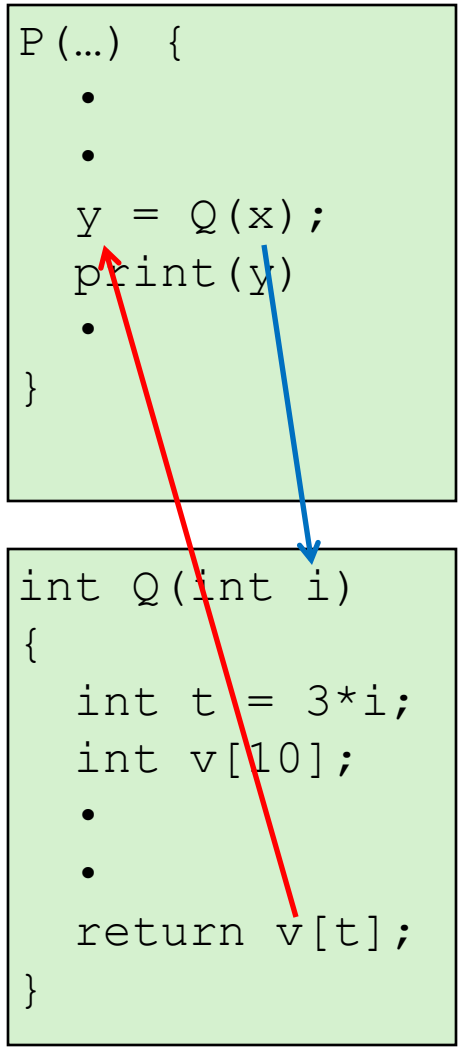
Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**



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
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**



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
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**

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

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

Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passin**
  - Proce
  - Retu
- **Memo**
  - Alloc
  - Deall
- **Mecha**
  machine instructions

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  return v[t];
}
```

Machine instructions implement the mechanisms, but the choices are determined by designers.

These choices make up
Application Binary Interface (ABI).

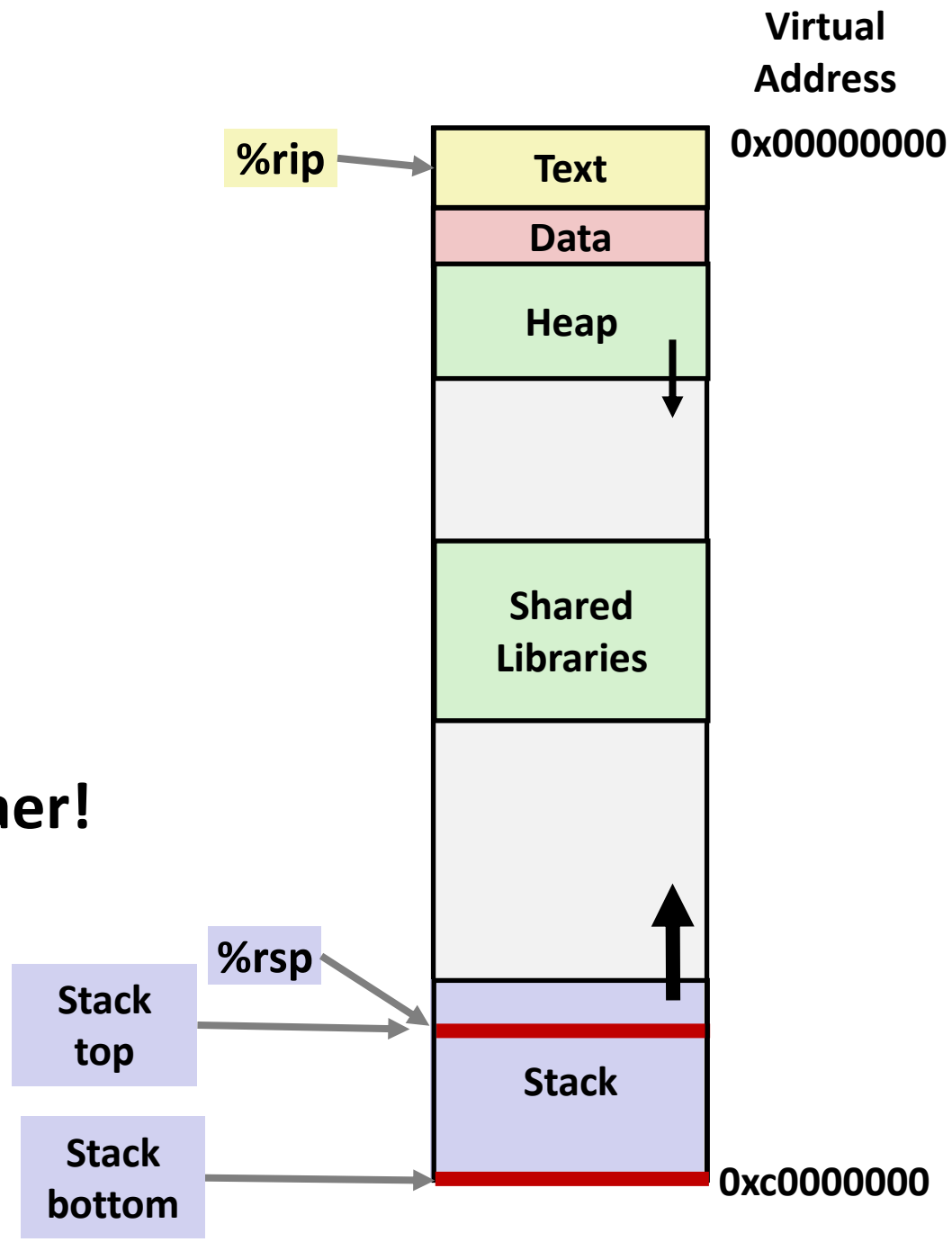
Today

- **Procedures**
  - Mechanisms
  - **Stack Structure**
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data

x86-64 Stack

- **Memory regions**
  - Memory viewed as array of bytes.
  - Different regions have different purposes.
- **Stack**
  - Grow from high to low addresses
  - Register %rsp points to the stack top

We will cover this memory layout laer!



x86-64 Stack: Push

- **Push [operand]**
  - Step #1: Decrease a stack top (i.e., %rsp)
  - Step #2: Write an operand at the address given by the stack top

  - Example: “pushq %rax” is the same as executing the following two instructions in order
    - subq $8, %rsp;
    - movq %rax, (%rsp)

  - Example: “pushq $5”
    - subq $8, %rsp
    - movq $5, (%rsp)

x86-64 Stack: Pop

- **Pop [operand]**
  - Step #1: Write a value at the stack top to where the operand specifies
  - Step #2: increase the stack top

  - Example: “popq %rax” is the same as …
    - movq (%rsp), %rax
    - addq $8, %rsp;

  - Example: “popq (%rax)” is the same as …
    - movq (%rsp), %_tmp
    - movq %_tmp, (%rax)
    - addq $8, %rsp

PUSH — Push Word, Doubleword, or Quadword Onto the Stack

Opcode[1]

	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FF /6	PUSH r/m16	M	Valid	Valid	Push r/m16.
FF /6	PUSH r/m32	M	N.E.	Valid	Push r/m32.
FF /6	PUSH r/m64	M	Valid	N.E.	Push r/m64.
50+rw	PUSH r16	O	Valid	Valid	Push r16.
50+rd	PUSH r32	O	N.E.	Valid	Push r32.
50+rd	PUSH r64	O	Valid	N.E.	Push r64.
6A ib	PUSH imm8	I	Valid	Valid	Push imm8.
68 iw	PUSH imm16	I	Valid	Valid	Push imm16.
68 id	PUSH imm32	I	Valid	Valid	Push imm32.
0E	PUSH CS	ZO	Invalid	Valid	Push CS.
16	PUSH SS	ZO	Invalid	Valid	Push SS.
1E	PUSH DS	ZO	Invalid	Valid	Push DS.
06	PUSH ES	ZO	Invalid	Valid	Push ES.
0F A0	PUSH FS	ZO	Valid	Valid	Push FS.
0F A8	PUSH GS	ZO	Valid	Valid	Push GS.

https://www.felixcloutier.com/x86/push

POP — Pop a Value From the Stack

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
8F /0	POP r/m16	M	Valid	Valid	Pop top of stack into m16; increment stack pointer.
8F /0	POP r/m32	M	N.E.	Valid	Pop top of stack into m32; increment stack pointer.
8F /0	POP r/m64	M	Valid	N.E.	Pop top of stack into m64; increment stack pointer. Cannot encode 32-bit operand size.
58+ rw	POP r16	O	Valid	Valid	Pop top of stack into r16; increment stack pointer.
58+ rd	POP r32	O	N.E.	Valid	Pop top of stack into r32; increment stack pointer.
58+ rd	POP r64	O	Valid	N.E.	Pop top of stack into r64; increment stack pointer. Cannot encode 32-bit operand size.
1F	POP DS	ZO	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	ZO	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	ZO	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	ZO	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
0F A1	POP FS	ZO	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
0F A1	POP FS	ZO	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
0F A9	POP GS	ZO	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
0F A9	POP GS	ZO	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
0F A9	POP GS	ZO	Valid	N.E.	Pop top of stack into GS; increment stack pointer by 64 bits.

https://www.felixcloutier.com/x86/pop

Today

- **Procedures**
  - Mechanisms
  - Stack Structure
  - **Calling Conventions**

Calling Conventions

- **Calling conventions**
  - Describe how two different functions interact
    - i.e., Describe the call interface
  - How to pass parameters? (register or stack)
  - Who takes care of old register values? (callee or caller)
  - The part of Application Binary Interface

- **There are many variations: each compiler offers various options**

x86-64	Microsoft x64 calling convention[21]	Windows (Microsoft Visual C++, GCC, Intel C++ Compiler, Delphi), UEFI	RCX/XMM0, RDX/XMM1, R8/XMM2, R9/XMM3	RTL (C)	Caller	Stack aligned on 16 bytes. 32 bytes shadow space on stack. The specified 8 registers can only be used for parameters 1 through 4. For C++ classes, the hidden `this` parameter is the first parameter, and is passed in RCX.[31]
	vectorcall	Windows (Microsoft Visual C++, Clang, ICC)	RCX/[XY]MM0, RDX/[XY]MM1, R8/[XY]MM2, R9/[XY]MM3 + [XY]MM4–5	RTL (C)	Caller	Extended from MS x64.[11]
	System V AMD64 ABI[28]	Solaris, Linux,[32] BSD, macOS, OpenVMS (GCC, Intel C++ Compiler, Clang, Delphi)	RDI, RSI, RDX, RCX, R8, R9, [XYZ]MM0–7	RTL (C)	Caller	Stack aligned on 16 bytes boundary. 128 bytes red zone below stack. The kernel interface uses RDI, RSI, RDX, R10, R8 and R9. In C++, `this` is the first parameter.

https://en.wikipedia.org/w/index.php?title=X86_calling_conventions

Passing control across functions

- **Use stack to support call and return**

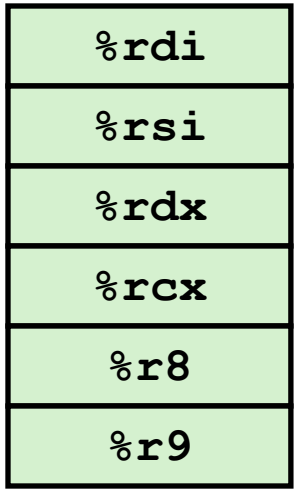
- **Procedure call: `call label`**
  - Step#1. Push return address on stack (➔ **pushq %rip**)
  - Step#2. Jump to ***label*** (➔ **jmp label**)

- **Procedure return: `ret`**
  - Step#1. Pop address from stack (➔ **popq %tmp**)
  - Step#2. Jump to address (➔ **jmp *%tmp**)
  - Return address:
    - Address of the next instruction when the previous call was invoked

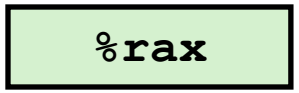
Passing data across functions

Registers

- Passing first 6 arguments

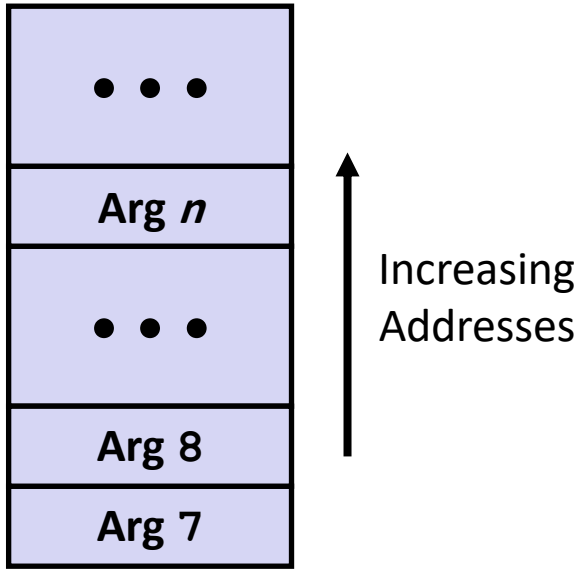


- Return value



Stack

- Passing beyond first 6 arguments



Example

```c
int add(int x, int y) {
    int res = x + y;
    return res;
}

int main(void) {
    return add( 1, 2);
}
```

call_convention.c

```asm
add:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movl    -20(%rbp), %edx
        movl    -24(%rbp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret

main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $2, %esi
        movl    $1, %edi
        call    add
        popq    %rbp
        ret
```

call_convention.s

```
gcc -O0 -S call_convention.c -o call_convention.s
```

Example

- **Do it yourself: Check the followings using GDB**
  - Check how the return address is pushed right after executing ‘call’
  - Check where %rsp points right before executing ‘ret’
  - When “main()” calls “add(1, 2)”, check how the parameter is handed over.

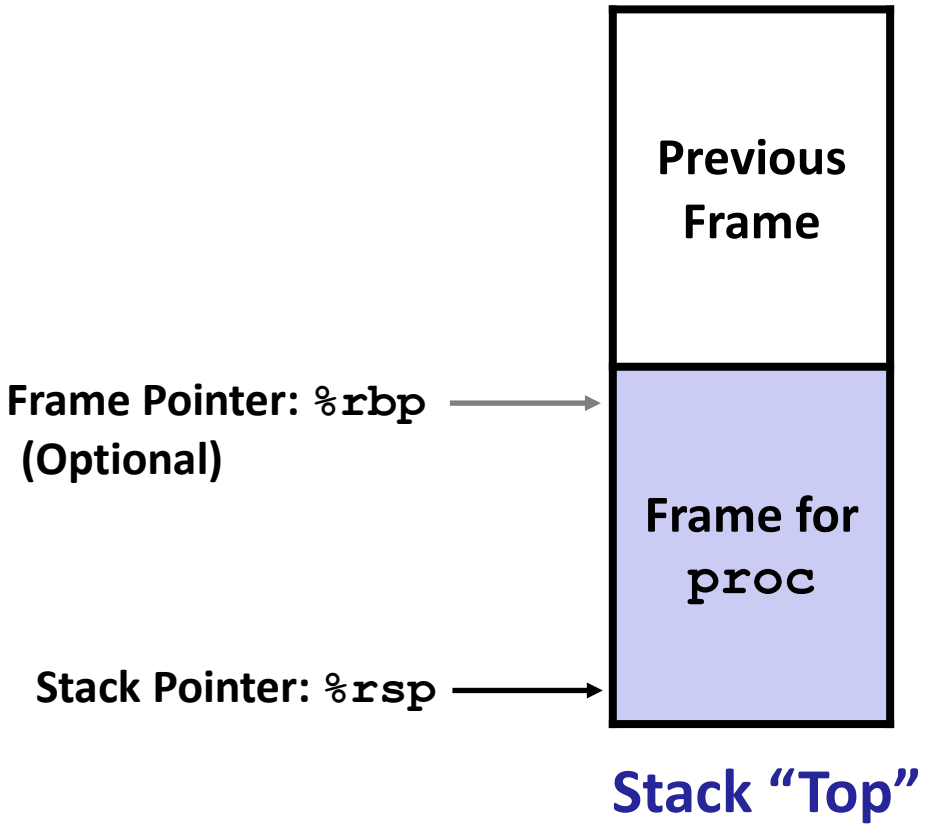
https://asciinema.org/a/Q67lcky1gUcXrsKY7VTDeDEgB

Data storage for function

- **Stack as function's data storage**
  - Code must be "*Reentrant*"
    - Multiple simultaneous instantiations of single procedure
    - Will be covered more later
  - Function needs some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return address
- **Stack discipline**
  - Lifetime: a given procedure is active for a limited time
    - From "the time it's called" to "the time it returns"
  - Return order: Callee should return before caller returns
- **Stack is allocated by the unit of *Frame***
  - Stack memory area for single procedure instantiation

Stack Frames

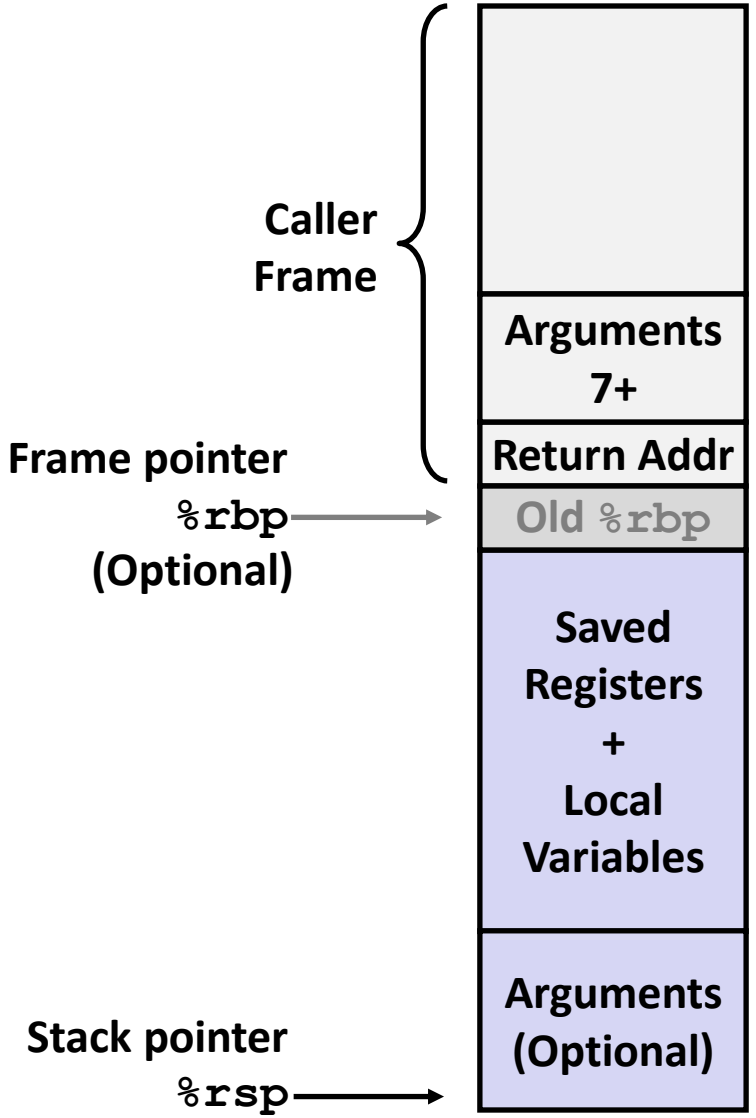
- **Contents**
  - Return address
  - Local storage (if needed)
- **Management**
  - Space allocated when enter procedure
    - "function prologue" code
  - Deallocated when return
    - "function epilogue" code



x86-64/Linux Stack Frame

- **Stack Frame**
  - Local variables
  - Saved register context
  - Old frame pointer (optional)

- **Caller's stack frame includes**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



Register Saving Conventions

- **When procedure `A` calls `B`:**
  - `A` is the ***caller***
  - `B` is the ***callee***
- **Can register be used for temporary storage?**
  - True: as long as registers are exclusively used by each function
- **Conventions**
  - ***"Caller Saved"***
    - Caller saves temporary values in its frame before the call
    - Caller restores them after the call
  - ***"Callee Saved"***
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller