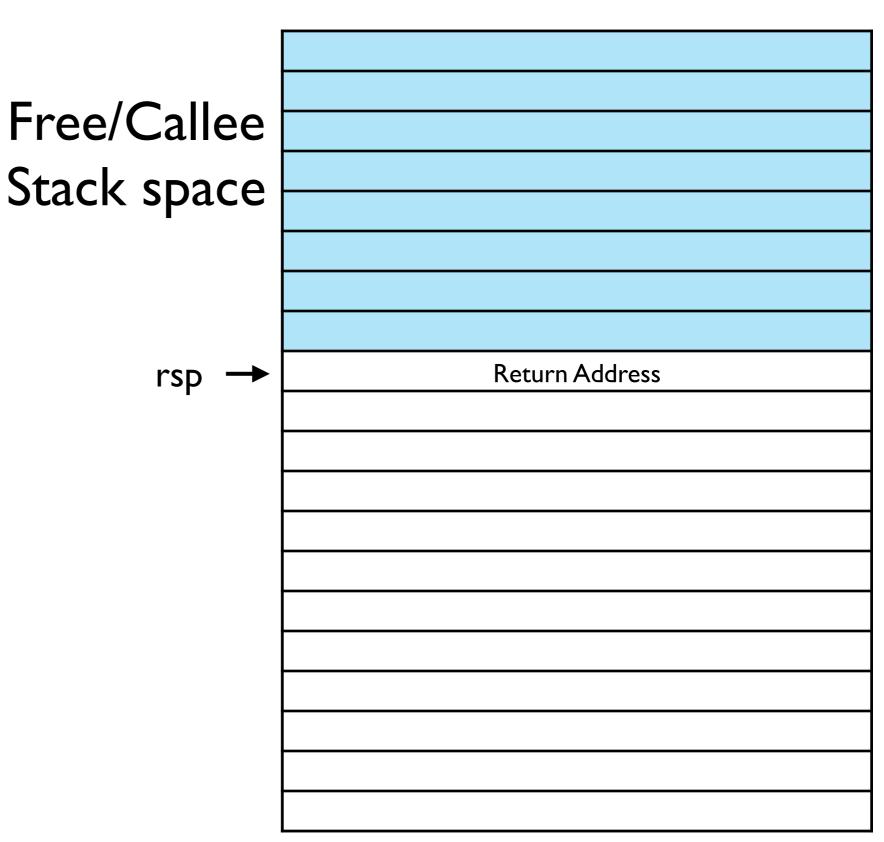
Calling Conventions

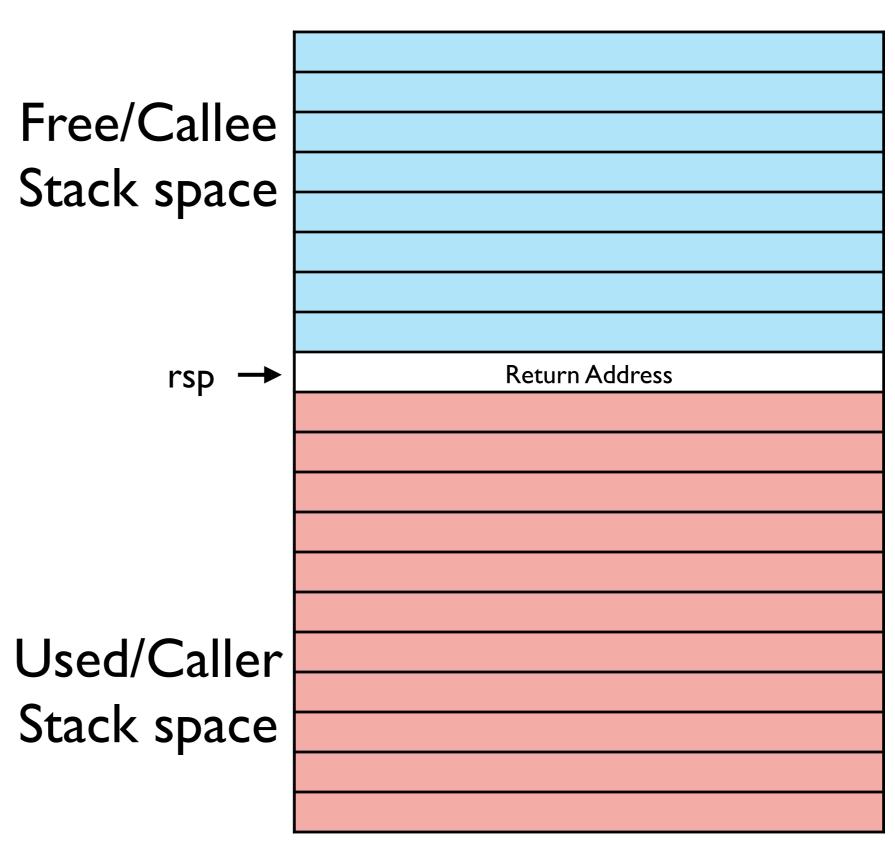
September 27, 2021

Calling Convention

- Where to put the return address?
- Where to put the returned value (if any)?
- Where to put the arguments?
- How to ensure local variables are preserved?
- How to ensure the stack is well-formed?

	D = 41, 11, 12 = 2
rsp →	Return Address







rsp

Local 2 Local I

. . . .

Local 0

Return Address

Used/Caller Stack space

Free/Callee Stack space

Local 2

LOCAI Z

Local I

Local 0

Return Address

What about function arguments?

rsp

Used/Caller Stack space

Free/Callee Stack space

Local 2

Local I

Local 0

rsp ·

Return Address

Arg 0

Arg I

Arg 2

. .

What about function arguments?

below rsp

Used/Caller Stack space

rsp points to the return address

• return value goes in rax

rsp

above rsp is free

below rsp is used by the caller

arguments go below rsp



rsp

- example
 - I local variable
 - call function of 2 arguments



- example
 - I local variable
 - call function of 2 arguments

```
mov [rsp - 16], ...
mov [rsp - 24], ...
sub rsp, 24
call fun
add rsp, 24
```

1 10
Local 0
Return Address
Arg 0
Arg I
Arg 2

rsp →

- example
 - I local variable
 - call function of 2 arguments

mov [rsp - 16], ... →mov [rsp - 24], ...

rsp →

sub rsp, 24

call fun

add rsp, 24

Arg I
Local 0
Return Address
Arg 0
Arg I
Arg 2

- example
 - I local variable
 - call function of 2 arguments

```
mov [rsp - 16], ...
mov [rsp - 24], ...
```

rsp →

sub rsp, 24 call fun add rsp, 24

Arg 0
Arg I
Local 0
Return Address
Arg 0
Arg I
Arg 2
•••

rsp

- example
 - I local variable
 - call function of 2 arguments

add rsp, 24

```
mov [rsp - 16], ...
mov [rsp - 24], ...
sub rsp, 24
call fun
```



- example
 - I local variable
 - call function of 2 arguments

```
mov [rsp - 16], ...
mov [rsp - 24], ...
sub rsp, 24
call fun
add rsp, 24
```

rsp →	Address of (add rsp, 24)
-	Arg 0
	Arg I
	Local 0
	Return Address
	Arg 0
	Arg I
	Arg 2

rsp

- example
 - I local variable
 - call function of 2 arguments

```
mov [rsp - 16], ...
mov [rsp - 24], ...
sub rsp, 24
call fun
add rsp, 24
```

Address of (add rsp, 24)
Arg 0
Arg I
Local 0
Return Address
Arg 0
Arg I
Arg 2
•••

- example
 - I local variable
 - call function of 2 arguments

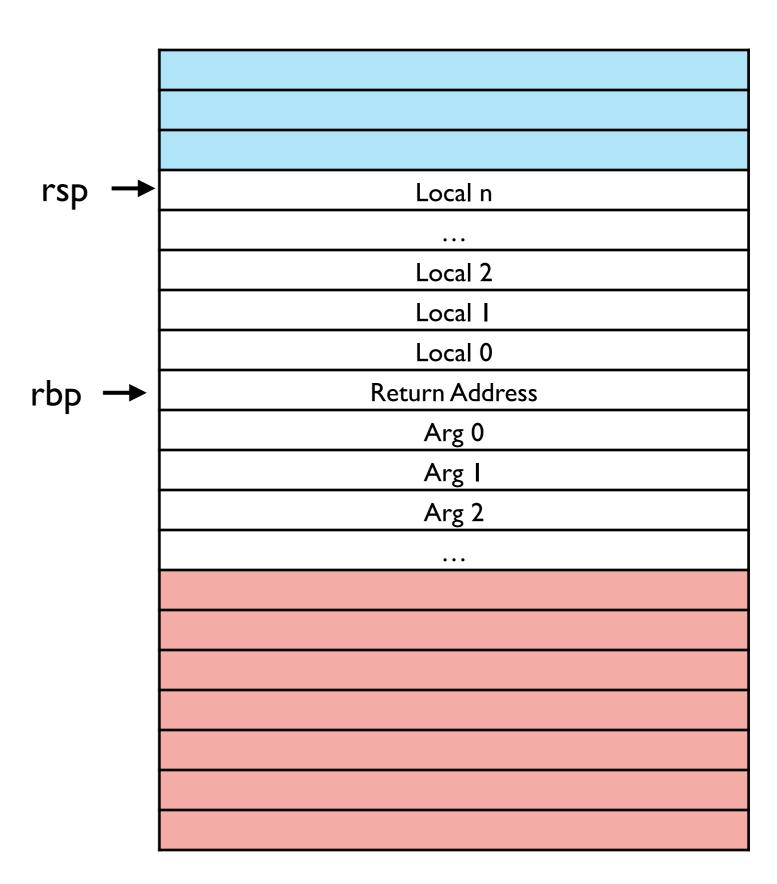
```
mov [rsp - 16], ...
mov [rsp - 24], ...
sub rsp, 24
call fun
add rsp, 24
```

	Address of (add rsp, 24)
	Arg 0
	Arg I
·	Local 0
	Return Address
	Arg 0
	Arg I
	Arg 2
	•••

rsp →

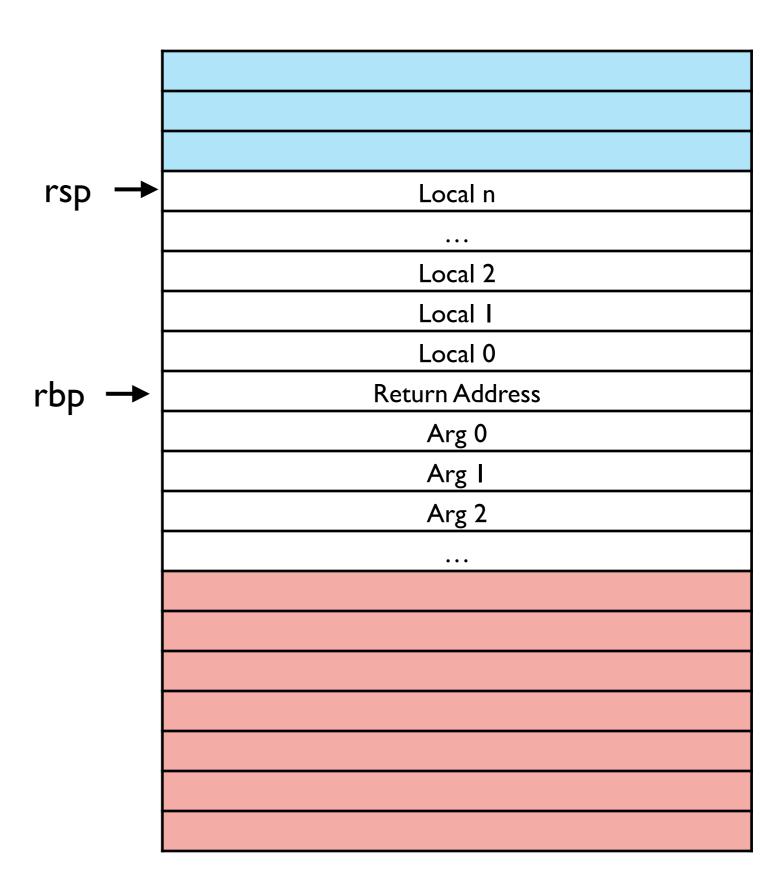
- Benefit of this approach:
 - just use one register for stack management
- Downside:
 - Can't tell where one stack "frame" begins and the next ends
 - stack trace, debugger, garbage collector need that info

- rbp: "base"/"frame"
 pointer: bottom of our
 portion of the stack
- rsp: stack pointer: top of our portion of the stack
- push instruction for arguments

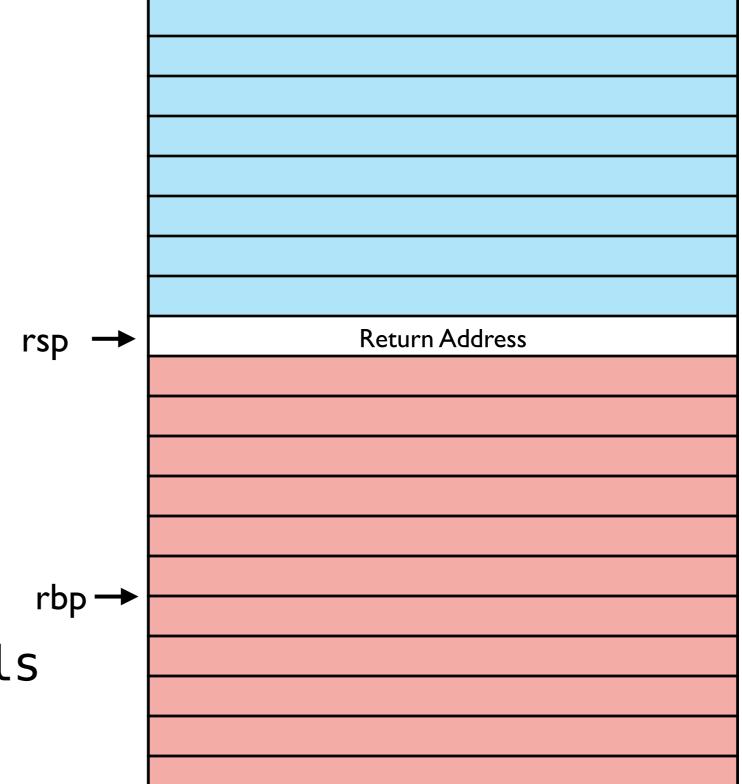


- rbp:"base"/"frame"
 pointer: bottom of our
 portion of the stack
- rsp: stack pointer: top of our portion of the stack
- push instruction for arguments

```
start_here:
mov rbp, rsp
mov [rbp - 8], 3
```



Stack



start_here:

>mov rbp, rsp

sub rsp, 8 * locals

Stack

rbp rsp →

Return Address

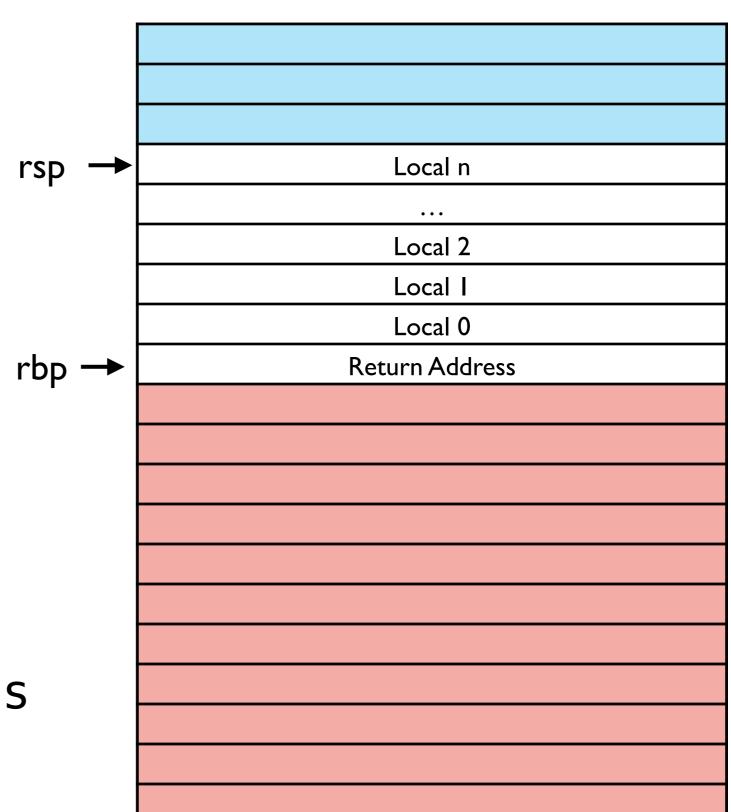
start_here:

mov rbp, rsp

 \rightarrow sub rsp, 8 * locals



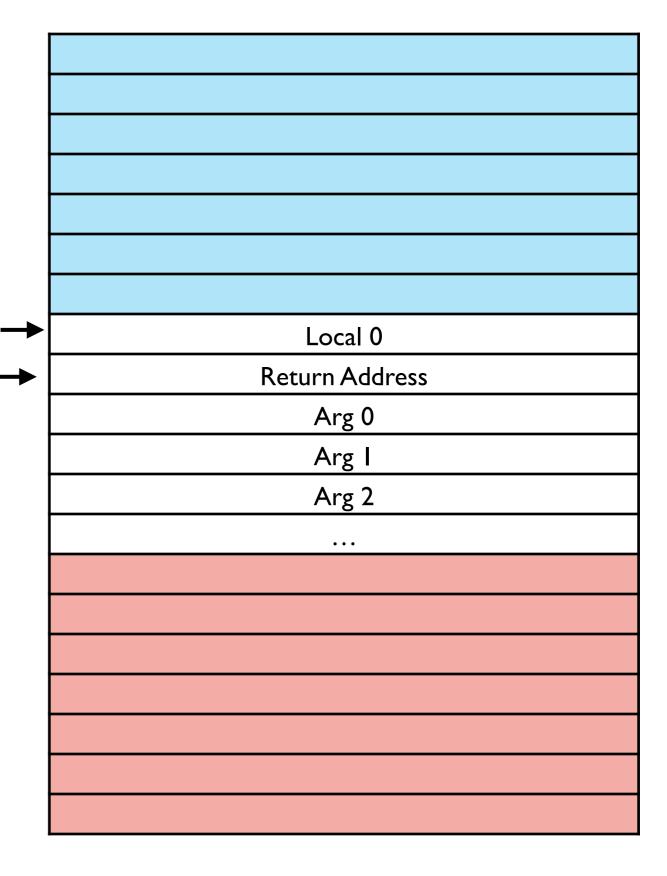
Stack



start_here:
mov rbp, rsp
sub rsp, 8 * locals

rsp

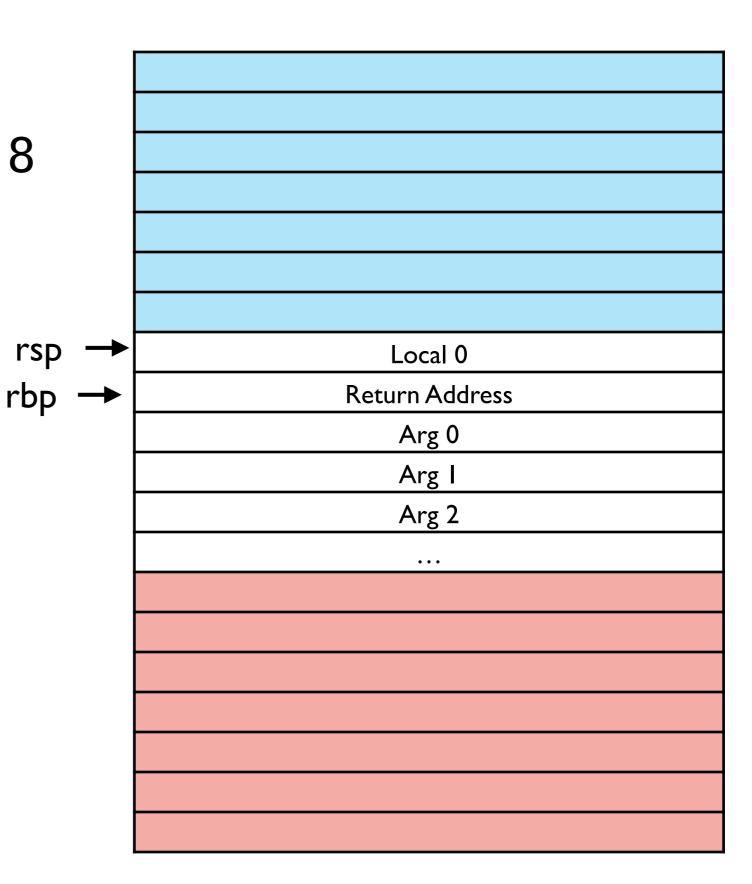
- example
 - I local variable
 - call function of 2 arguments



mov rbp, rsp sub rsp, locals * 8

•••

mov rsp, rbp ret



mov rbp, rsp sub rsp, locals * 8

rsp

rbp

•••

mov rsp, rbp ret

push arg1
push arg0

call fun add rsp, 16

→	Arg I
	Local 0
→	Return Address
	Arg 0
	Arg I
	Arg 2
	•••

mov rbp, rsp
sub rsp, locals * 8

•••

mov rsp, rbp

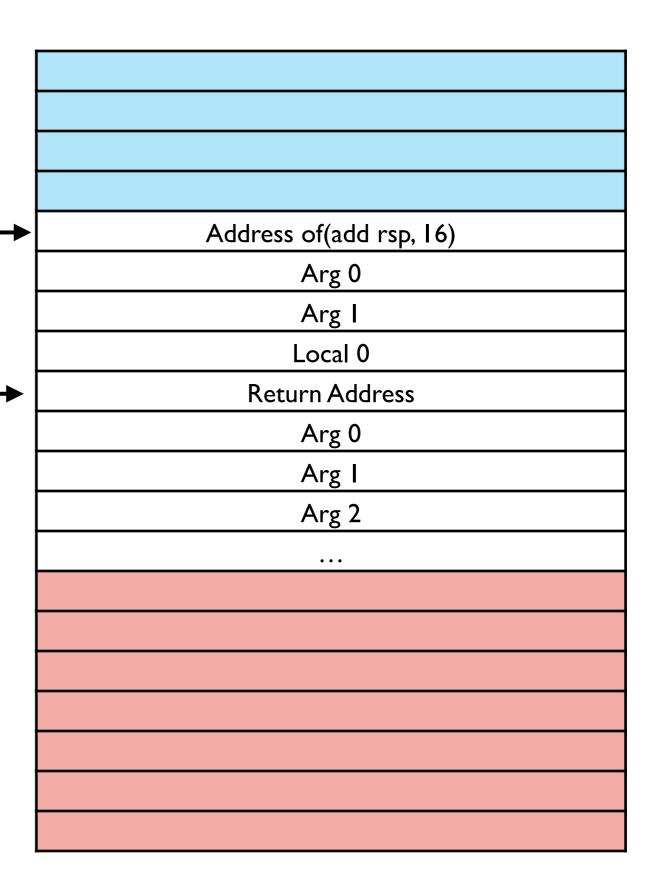
ret

,	
8	
rsp →	Arg 0
•	Arg I
	Local 0
rbp →	Return Address
•	Arg 0
	Arg I
	Arg 2
	•••

mov rbp, rsp sub rsp, locals * 8 rsp

rbp

mov rsp, rbp ret



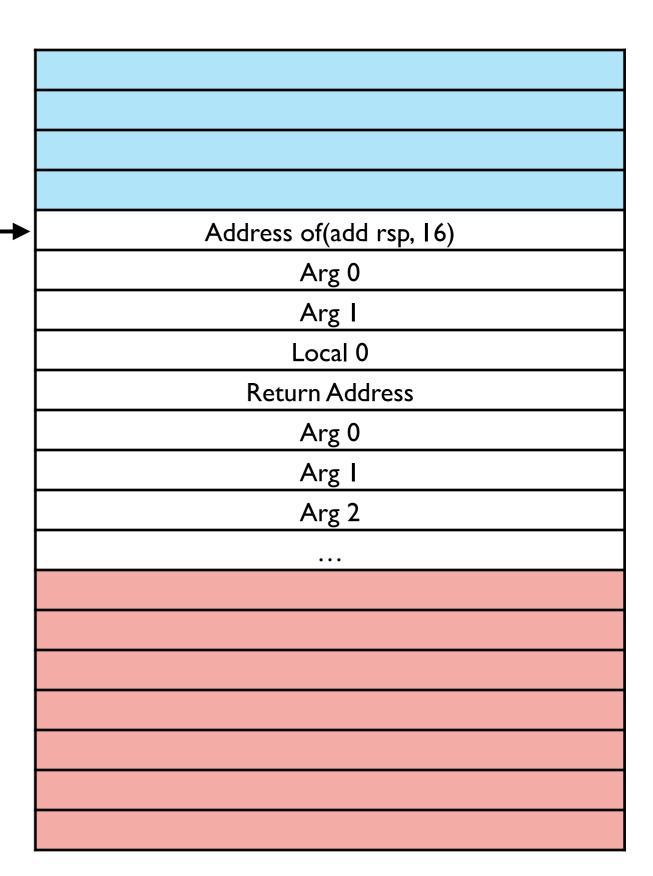
ret

mov rbp, rsp

sub rsp, locals * 8

rbp rsp

mov rsp, rbp



mov rbp, rsp rsp rsp → sub rsp, locals * 8

mov rsp, rbp ret

rsp -	
8	
rbp -	Address of(add rsp, 16)
•	Arg 0
	Arg I
	Local 0
	Return Address
	Arg 0
	Arg I
	Arg 2
	•••

mov rbp, rsp rsp → sub rsp, locals * 8

...

mov rsp, rbp ret

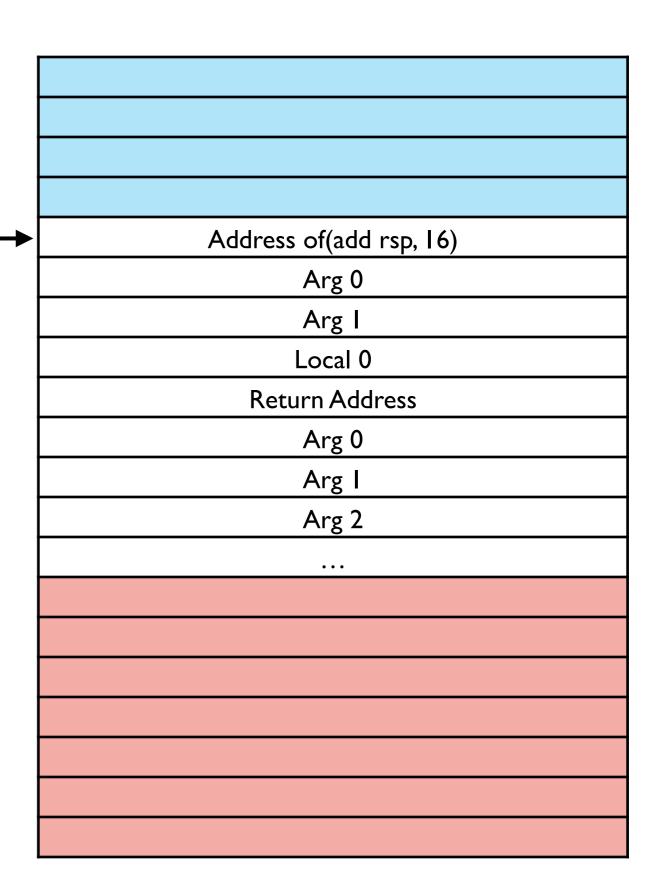
rsp →	
8	
rbp →	Address of(add rsp, 16)
	Arg 0
	Arg I
	Local 0
	Return Address
	Arg 0
	Arg I
	Arg 2
	•••

mov rbp, rsp sub rsp, locals * 8

rbp rsp

mov rsp, rbp

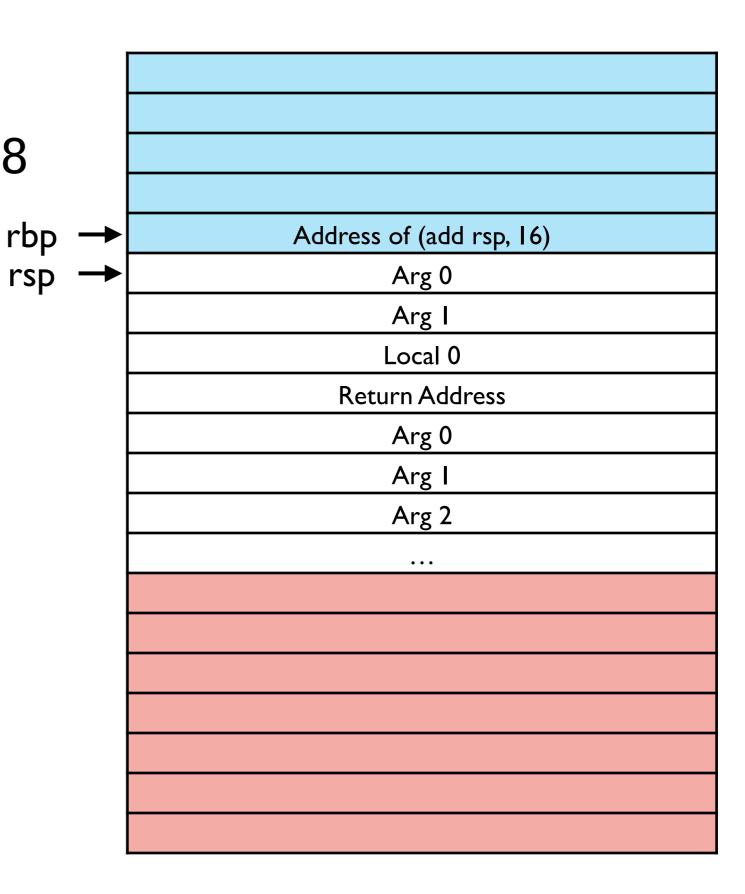
ret



mov rbp, rsp sub rsp, locals * 8

rsp

mov rsp, rbp ret



mov rbp, rsp sub rsp, locals * 8

mov rsp, rbp ret

push arg1 push arg0 call fun add rsp, 16

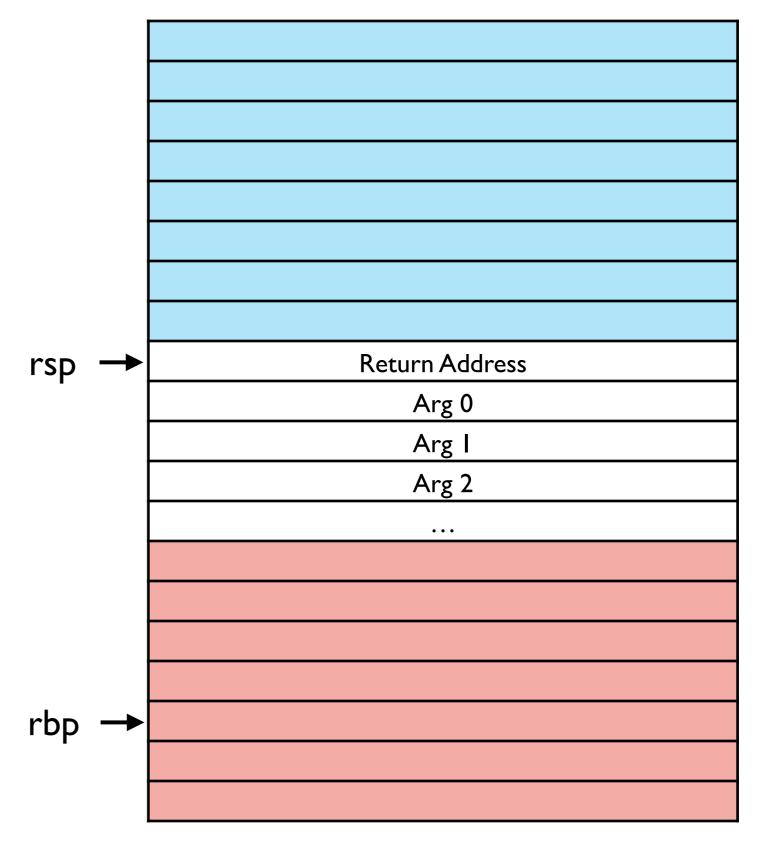
rbp is wrong!

rsp



32-bit
C Calling Convention

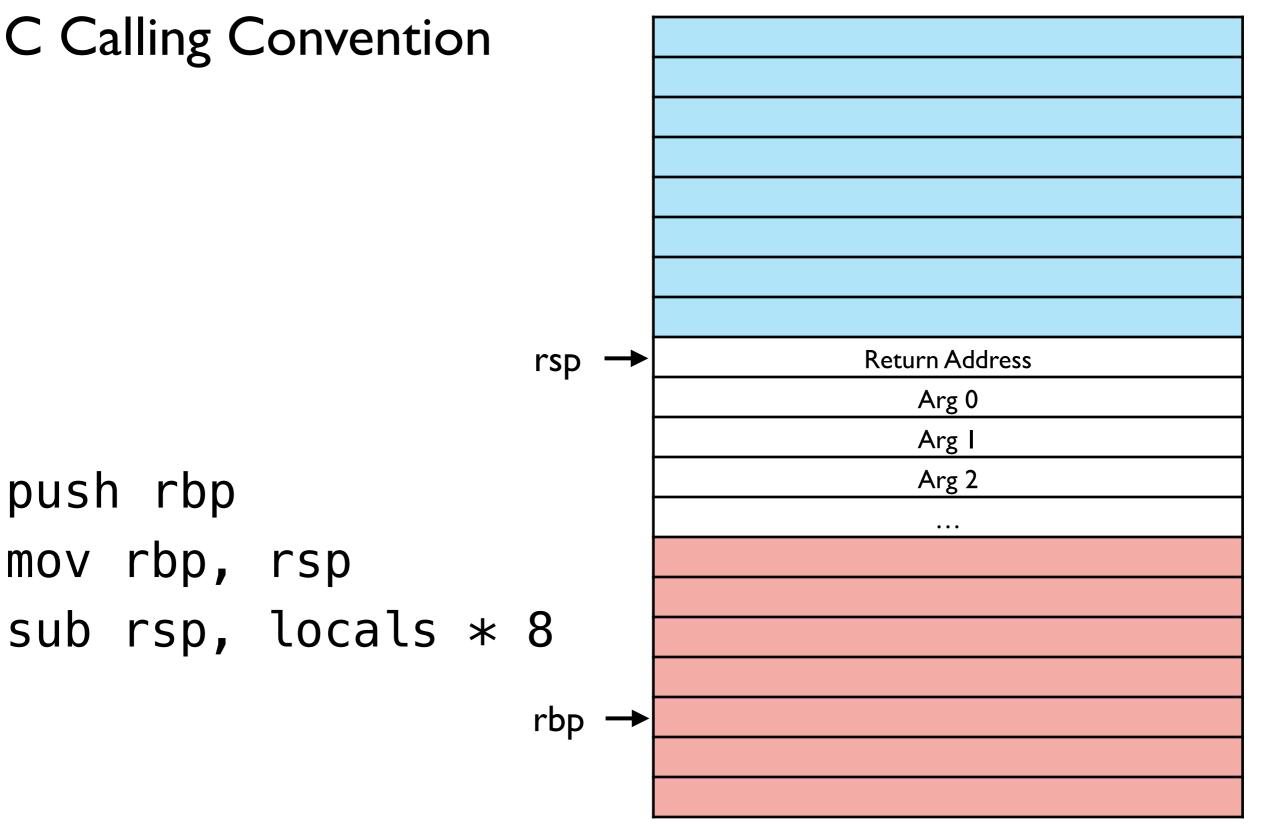
rbp is a *callee-save*
register, meaning the
callee is responsible for
ensuring the value is
restored before
returning



32-bit C Calling Convention

push rbp

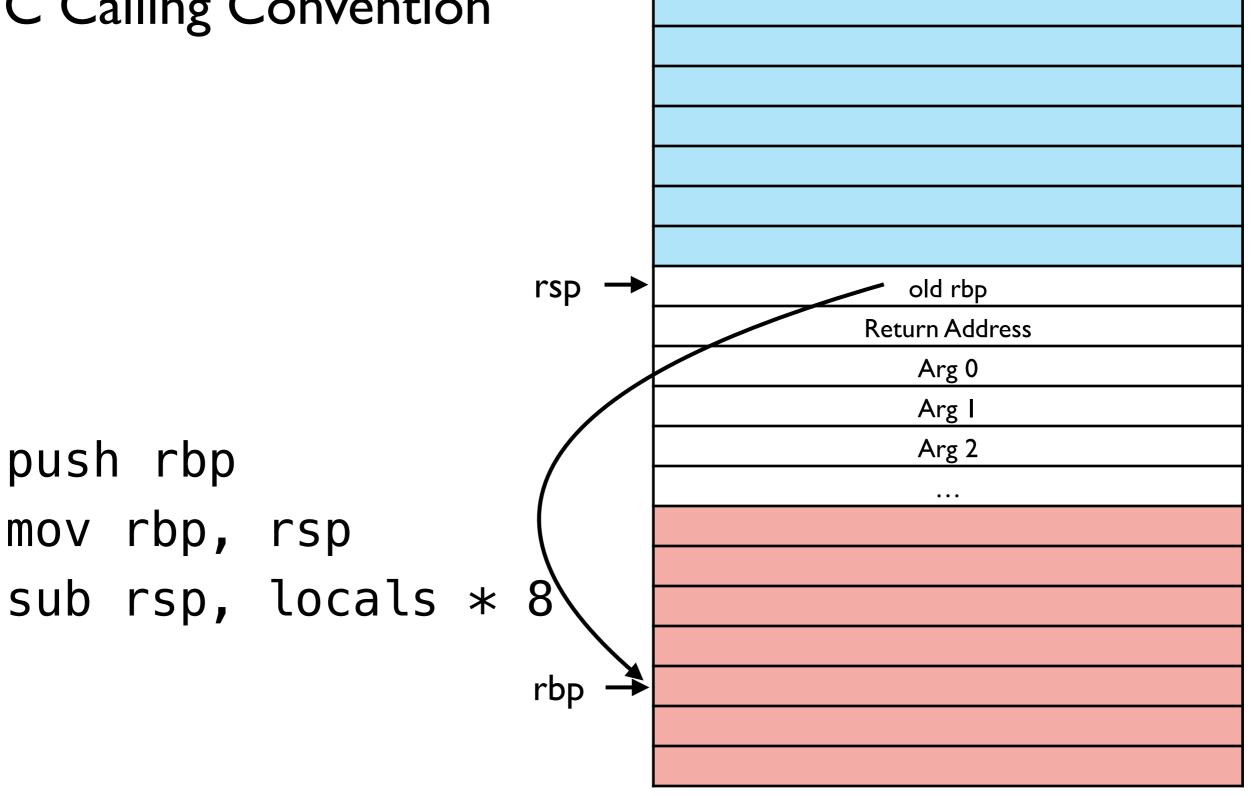
mov rbp, rsp



32-bit C Calling Convention

push rbp

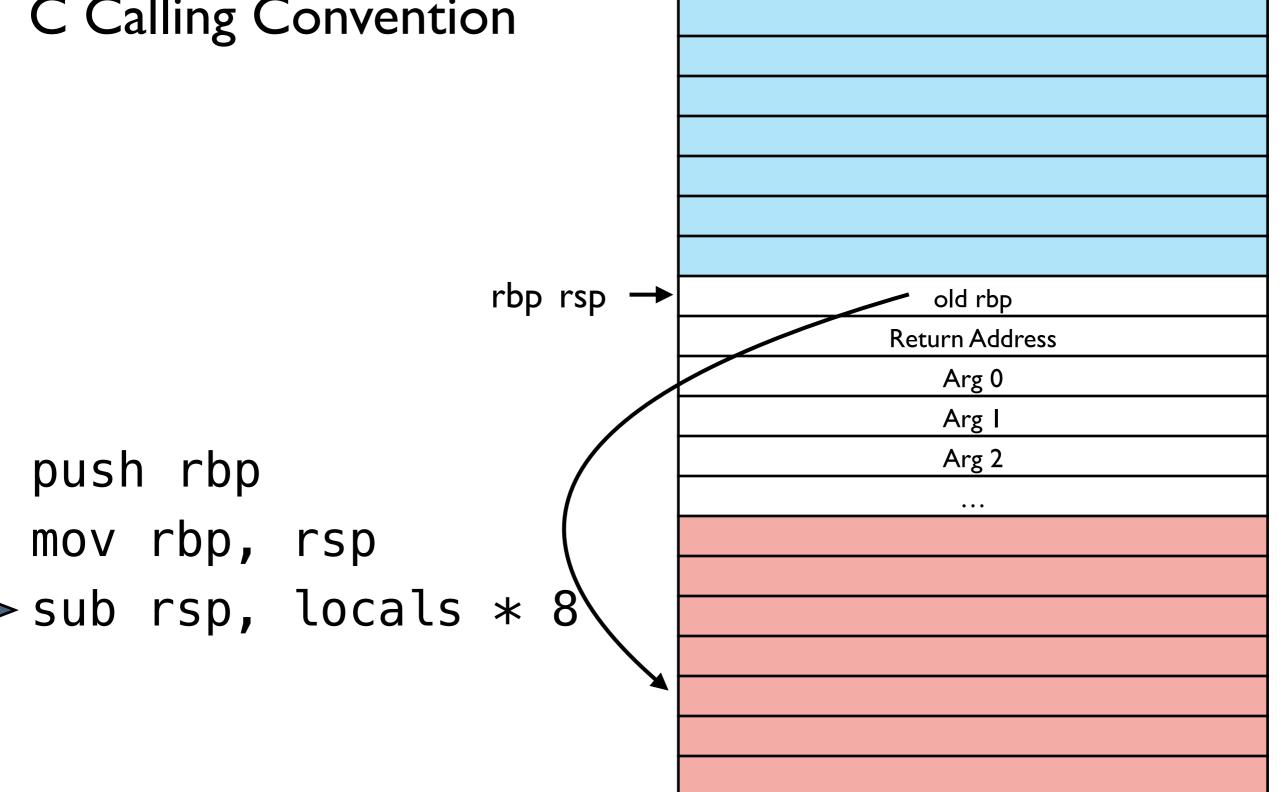
mov rbp, rsp



32-bit C Calling Convention

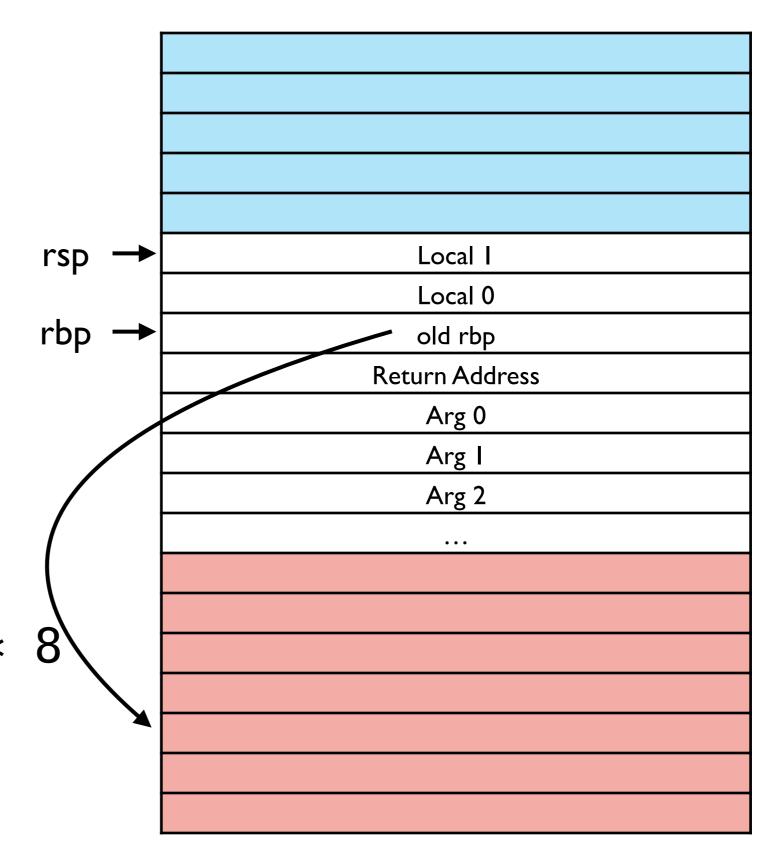
push rbp

mov rbp, rsp



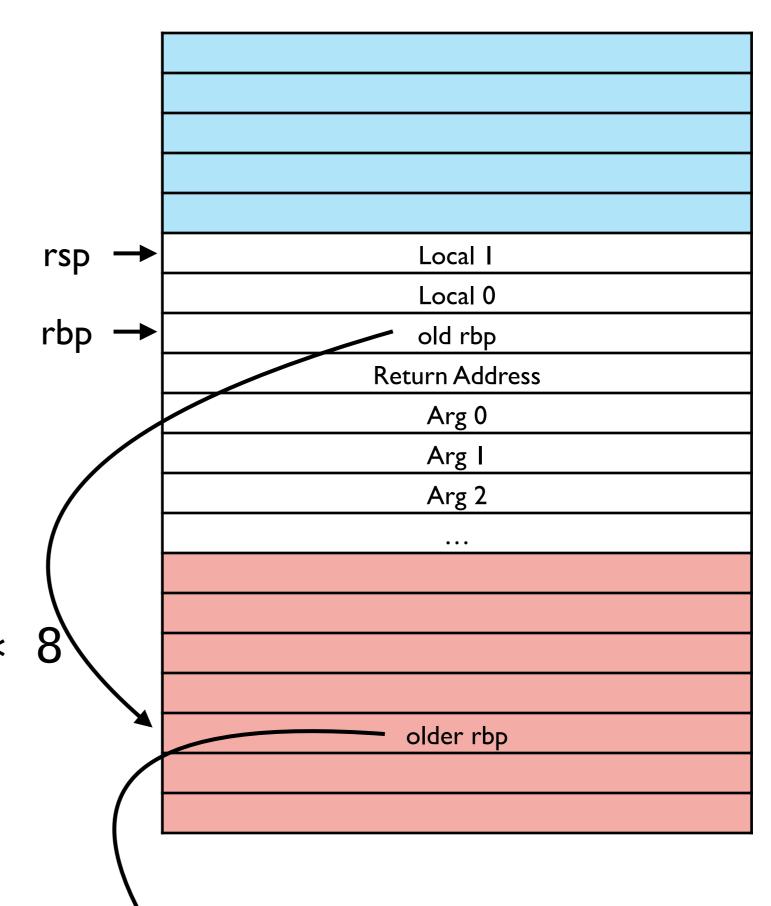
- example
 - I local variable
 - call function of 2 arguments

push rbp
mov rbp, rsp
sub rsp, locals * 8



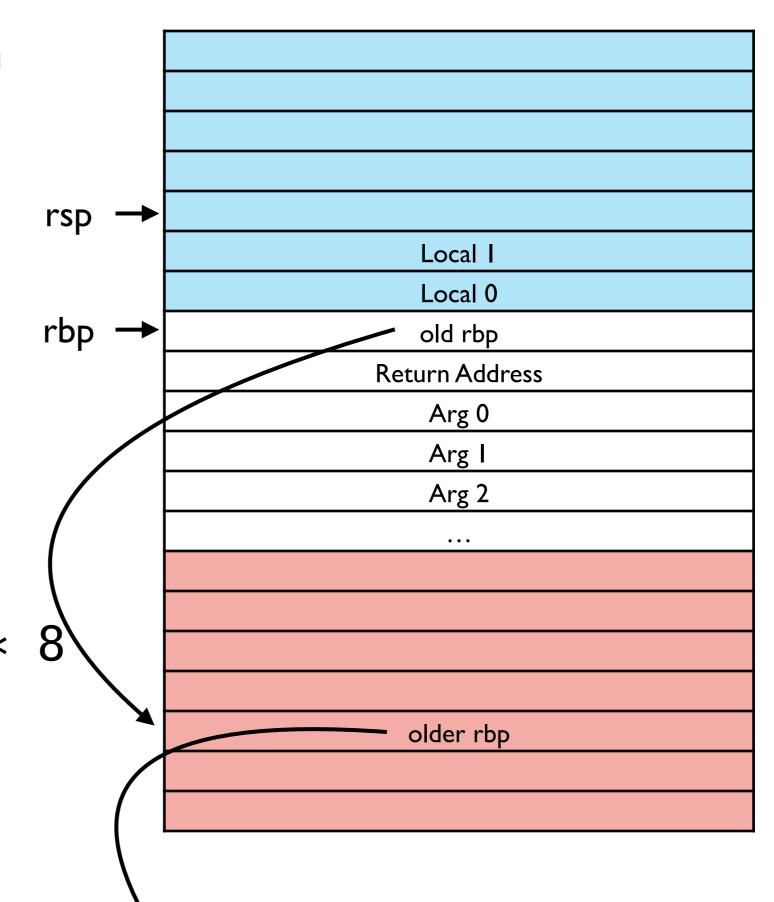
- example
 - I local variable
 - call function of 2 arguments

push rbp
mov rbp, rsp
sub rsp, locals * 8



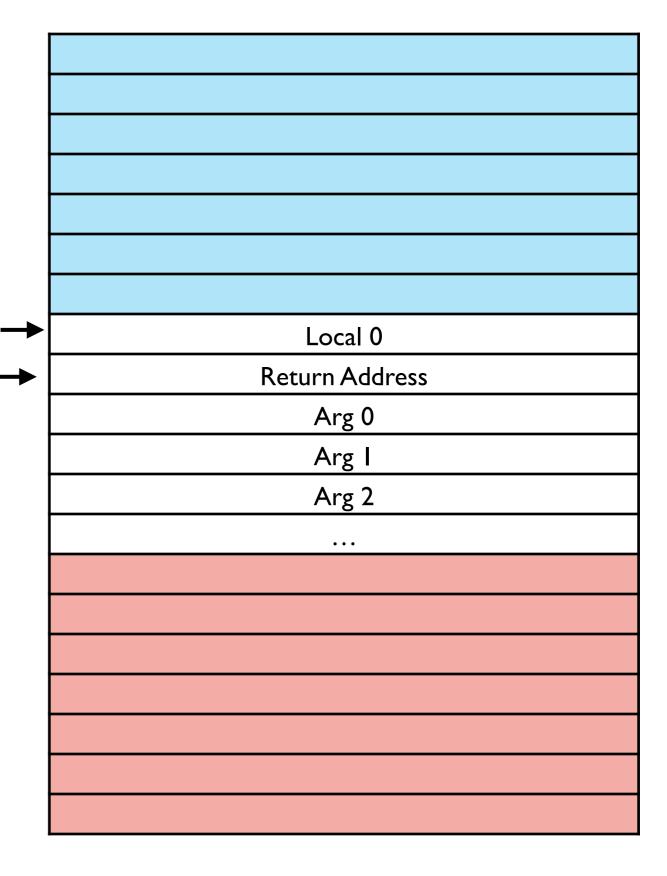
 rbp values form a linked list of stack frames

push rbp
mov rbp, rsp
sub rsp, locals * 8

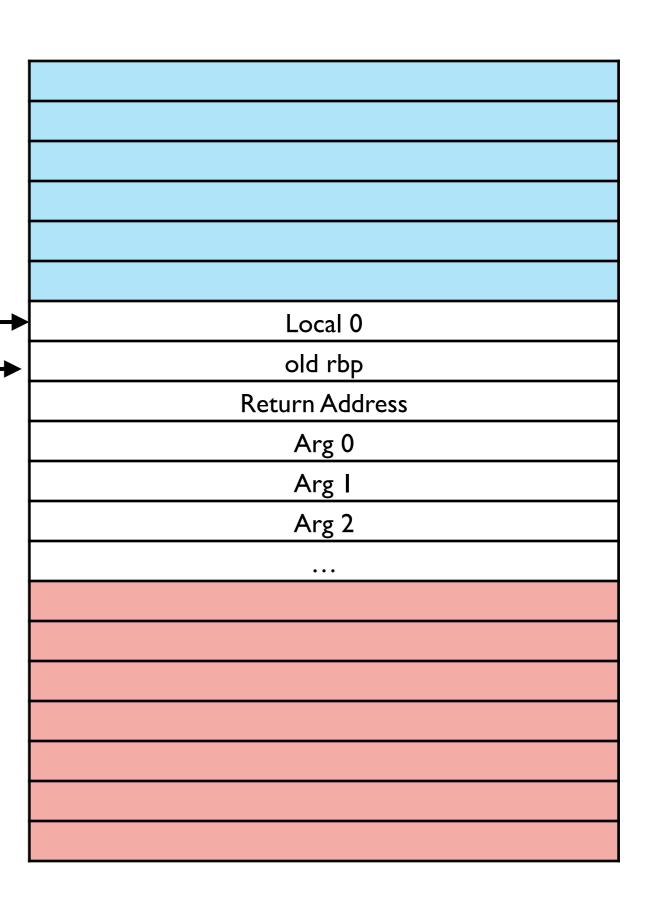


rsp

- example
 - I local variable
 - call function of 2 arguments



```
fun:
  push rbp
  mov rbp, rsp
  sub rsp, locals * 8
  rsp '
  mov rsp, rbp
                       rbp
  pop rbp
  ret
push arg1
   push arg0
   call fun
```



fun: push rbp mov rbp, rsp sub rsp, locals * 8 rsp mov rsp, rbp rbp pop rbp ret push arg1 push arg0 call fun add rsp, 16



fun: push rbp mov rbp, rsp sub rsp, locals * 8 _{rsp} → Arg I Arg 0 Local 0 mov rsp, rbp rbp old rbp Return Address pop rbp Arg 0 Arg I ret Arg 2 push arg1 push arg0 - call fun add rsp, 16

fun:

→push rbp mov rbp, rsp Address of(add rsp, 16) sub rsp, locals $* 8^{rsp}$ Arg I Arg 0 Local 0 mov rsp, rbp rbp old rbp Return Address pop rbp Arg 0 Arg I ret Arg 2 push arg1 push arg0 call fun add rsp, 16

fun: push rbp mov rbp, rsp rsp Address of(add rsp, 16) sub rsp, locals * 8 Arg I Arg 0 Local 0 mov rsp, rbp rbp old rbp Return Address pop rbp Arg 0 Arg I ret Arg 2 push arg1 push arg0 call fun add rsp, 16

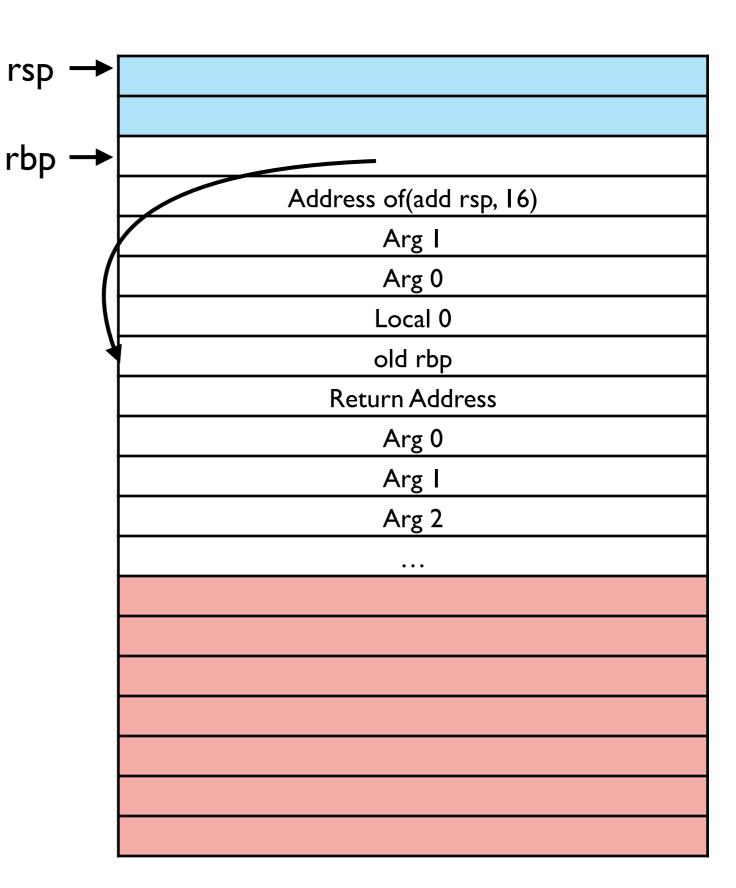
```
fun:
  push rbp
 mov rbp, rsp rbp rsp
\Rightarrowsub rsp, locals * 8
  mov rsp, rbp
  pop rbp
  ret
   push arg1
   push arg0
   call fun
```



fun:
push rbp
mov rbp, rsp
sub rsp, locals * 8

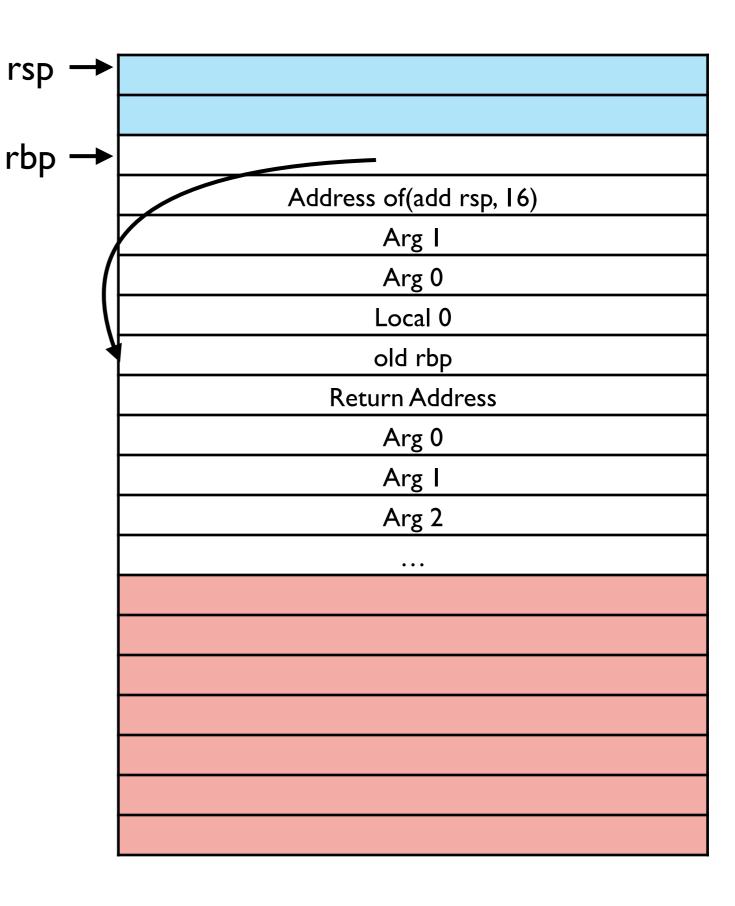
mov rsp, rbp
pop rbp
ret

push arg1
push arg0
call fun
add rsp, 16

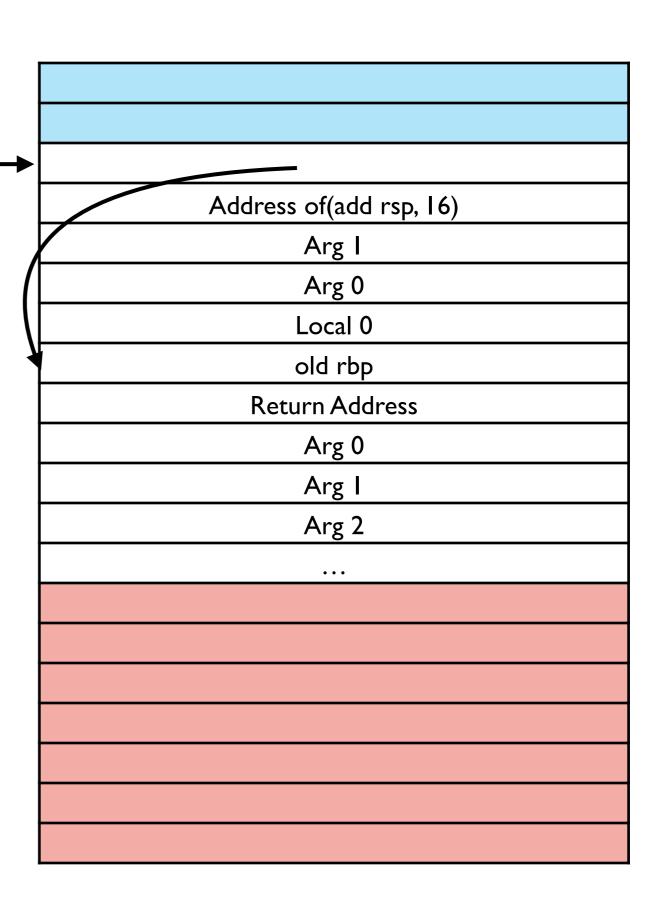


```
fun:
push rbp
mov rbp, rsp
sub rsp, locals * 8
mov rsp, rbp
pop rbp
ret
 push arg1
 push arg0
```

call fun



```
fun:
push rbp
mov rbp, rsp rbp rsp
sub rsp, locals * 8
...
mov rsp, rbp
>pop rbp
ret
  push arg1
  push arg0
  call fun
```



```
fun:
push rbp
mov rbp, rsp
                                          Address of(add rsp, 16)
sub rsp, locals *8^{rsp}
                                               Arg I
                                               Arg 0
Local 0
mov rsp, rbp
                            rbp
                                               old rbp
                                            Return Address
pop rbp
                                               Arg 0
                                               Arg I
ret
                                               Arg 2
  push arg1
  push arg0
  call fun
  add rsp, 16
```

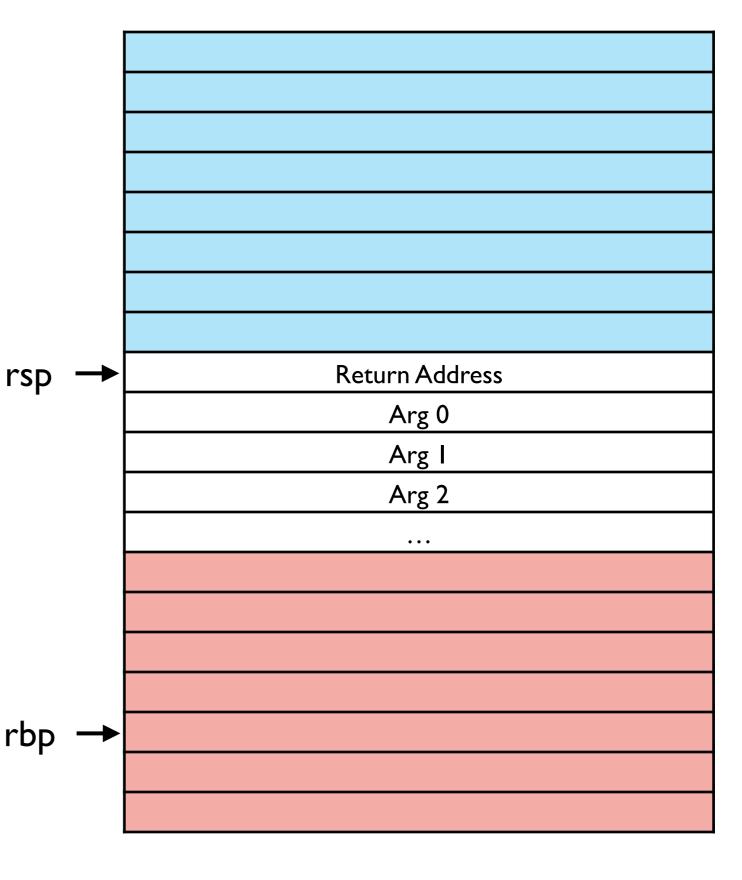
```
fun:
push rbp
mov rbp, rsp
sub rsp, locals * 8 <sub>rsp</sub>
                                               Arg I
                                               Arg 0
Local 0
mov rsp, rbp
                            rbp -
                                              old rbp
                                            Return Address
pop rbp
                                               Arg 0
                                               Arg I
ret
                                               Arg 2
  push arg1
  push arg0
  call fun
  add rsp, 16
```

```
fun:
push rbp
mov rbp, rsp
sub rsp, locals * 8
rsp
mov rsp, rbp
                     rbp →
pop rbp
ret
 push arg1
 push arg0
 call fun
 add rsp, 16
```



rsp

- rsp points to the return address
- arguments are below (higher addresses) return address
- return value goes in rax
- rbp, rbx, r12-r15 are callee-save



64-bit

C Calling Convention

rsp

- rsp points to the return address
- arguments are below (higher addresses) return address
- return value goes in rax
- rbp, rbx, r12-r15 are callee-save



64-bit

C Calling Convention

- rsp points to the return address
- arguments

- callee-save
- rdi, rsi, rdx, rcx, Return Address rsp r8, r9 Arg 0 Arg I more on stack (if Arg 2 needed) return value goes in rax rbp, rbx, r12-r15 are rbp → before a call,

rsp % 16 == 0

Calling Conventions

- When calling rust code, we need to use the 64 bit calling convention
- When calling "snake" functions, we will use the 32-bit calling convention
 - Makes it easier to test passing arguments on the stack without making huge functions