

(Non-Tail) Function Calls

October 5, 2022

Previously on EECS 483...

- Calling into Rust: System V AMD64 calling convention
- Tail Calls: compile directly to jumps

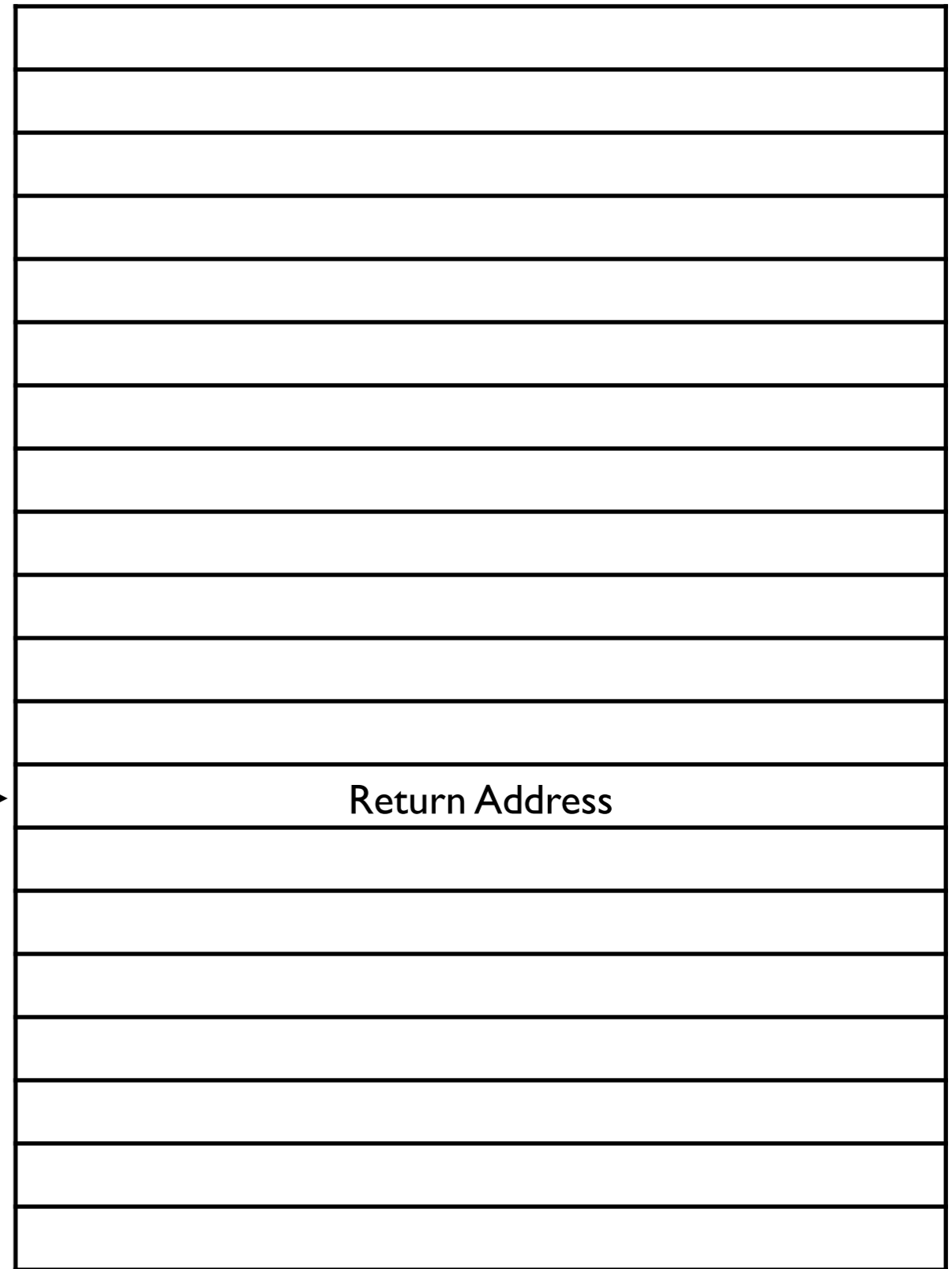
```
def f(y):  
    let z = 2 in  
    y * z  
end  
let a = 7 in  
let b = 13 in  
let x = f(a) in  
x + b
```

```
def f(y):  
    let z = 2 in  
        y * z  
end  
→ let a = 7 in  
    let b = 13 in  
        let x = f(a) in  
            x + b
```


Stack

```
def f(y):  
    let z = 2 in  
    y * z  
end  
→ let a = 7 in  
   let b = 13 in  
   let x = f(a) in  
   x + b
```

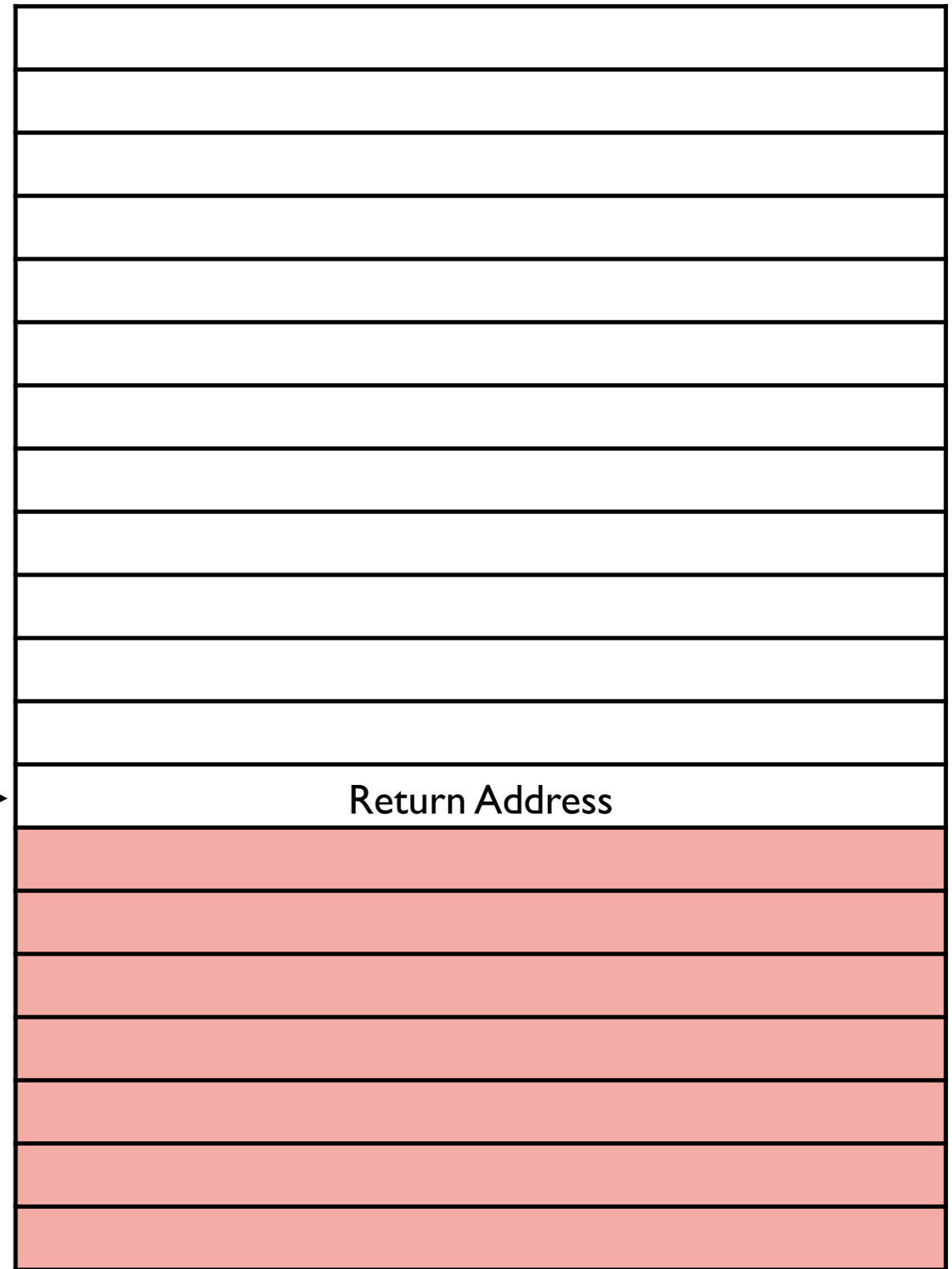
rsp →



Stack

```
def f(y):  
    let z = 2 in  
    y * z  
end  
→ let a = 7 in  
   let b = 13 in  
   let x = f(a) in  
   x + b
```

rsp →



Used/Caller

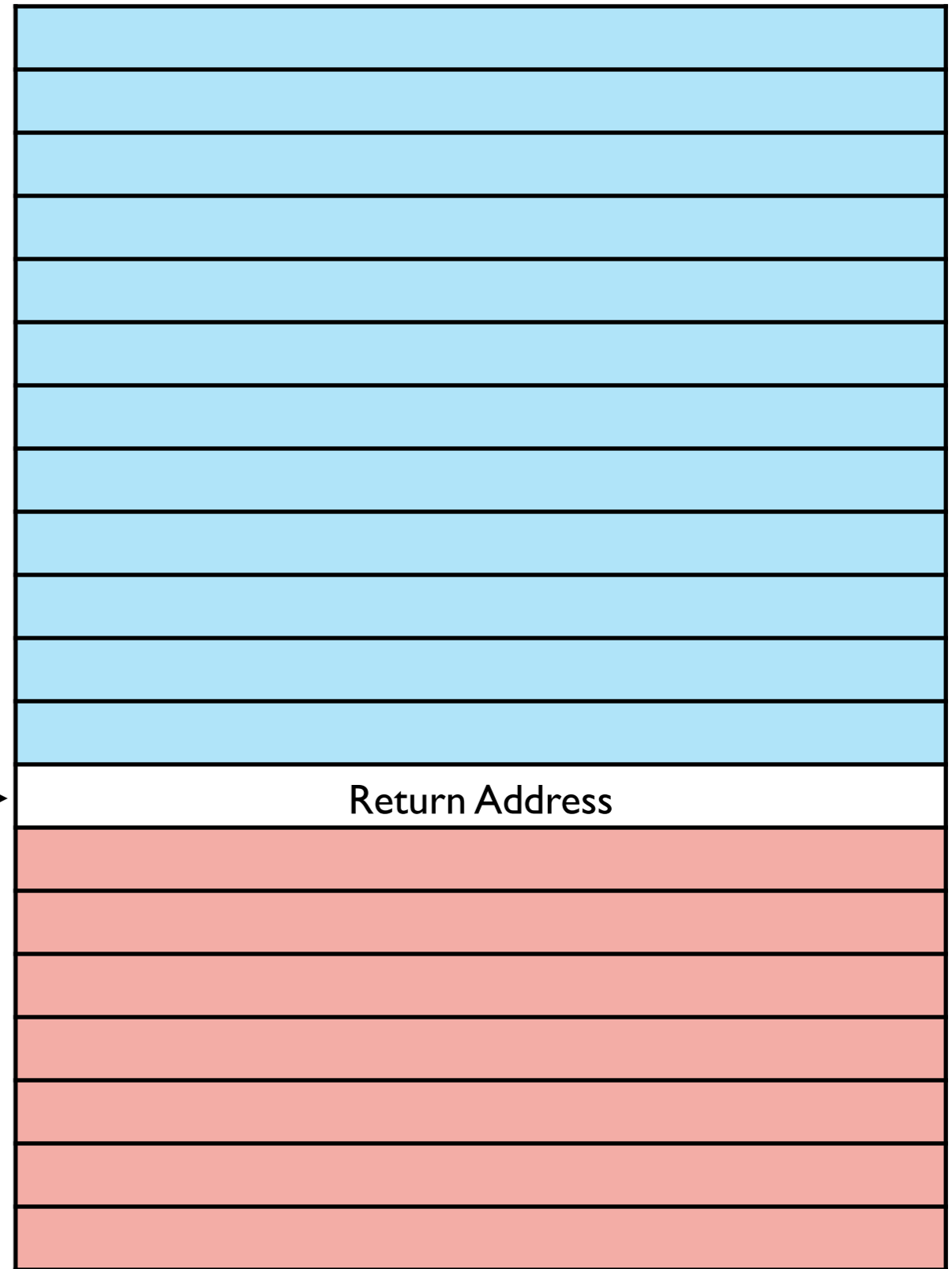
Stack

```
def f(y):  
    let z = 2 in  
    y * z  
end  
→ let a = 7 in  
   let b = 13 in  
   let x = f(a) in  
   x + b
```

Free/Callee

rsp →

Used/Caller



Stack

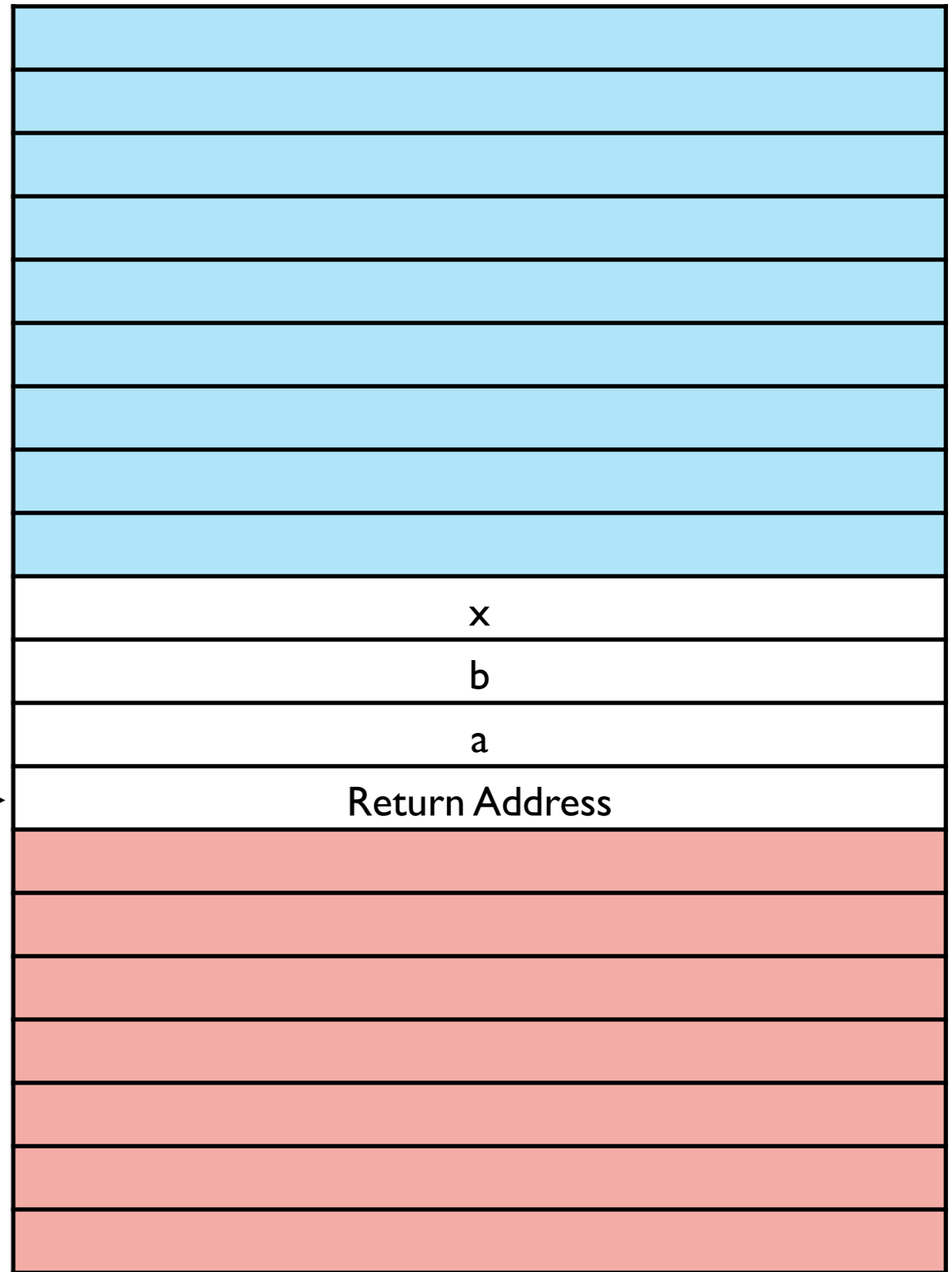
```
def f(y):  
    let z = 2 in  
    y * z  
end  
→ let a = 7 in  
   let b = 13 in  
   let x = f(a) in  
   x + b
```

Free/Callee

Stack
Frame

rsp →

Used/Caller



Stack

```
def f(y):
```

```
→ let z = 2 in  
  y * z
```

```
end
```

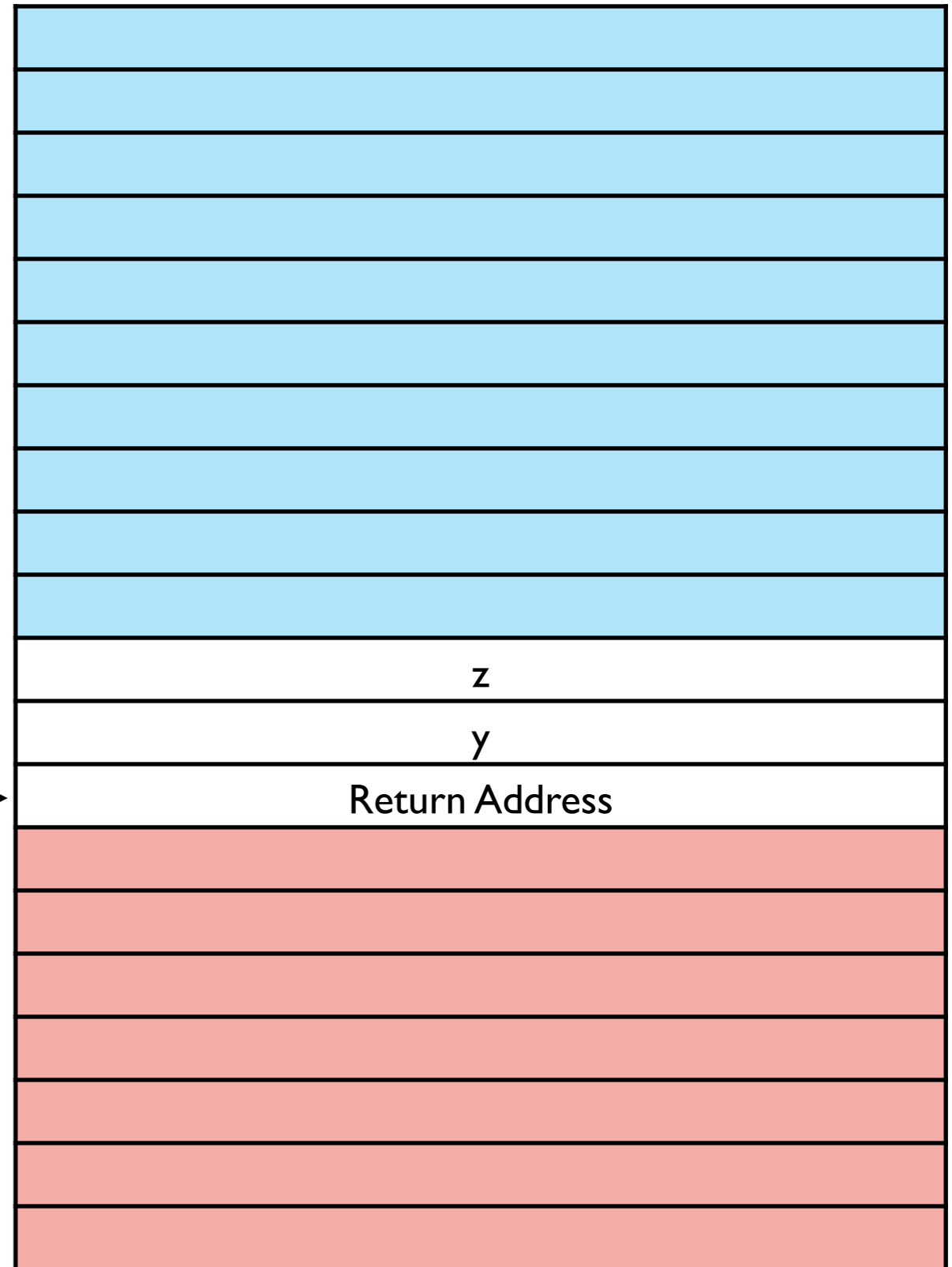
```
let a = 7 in
```

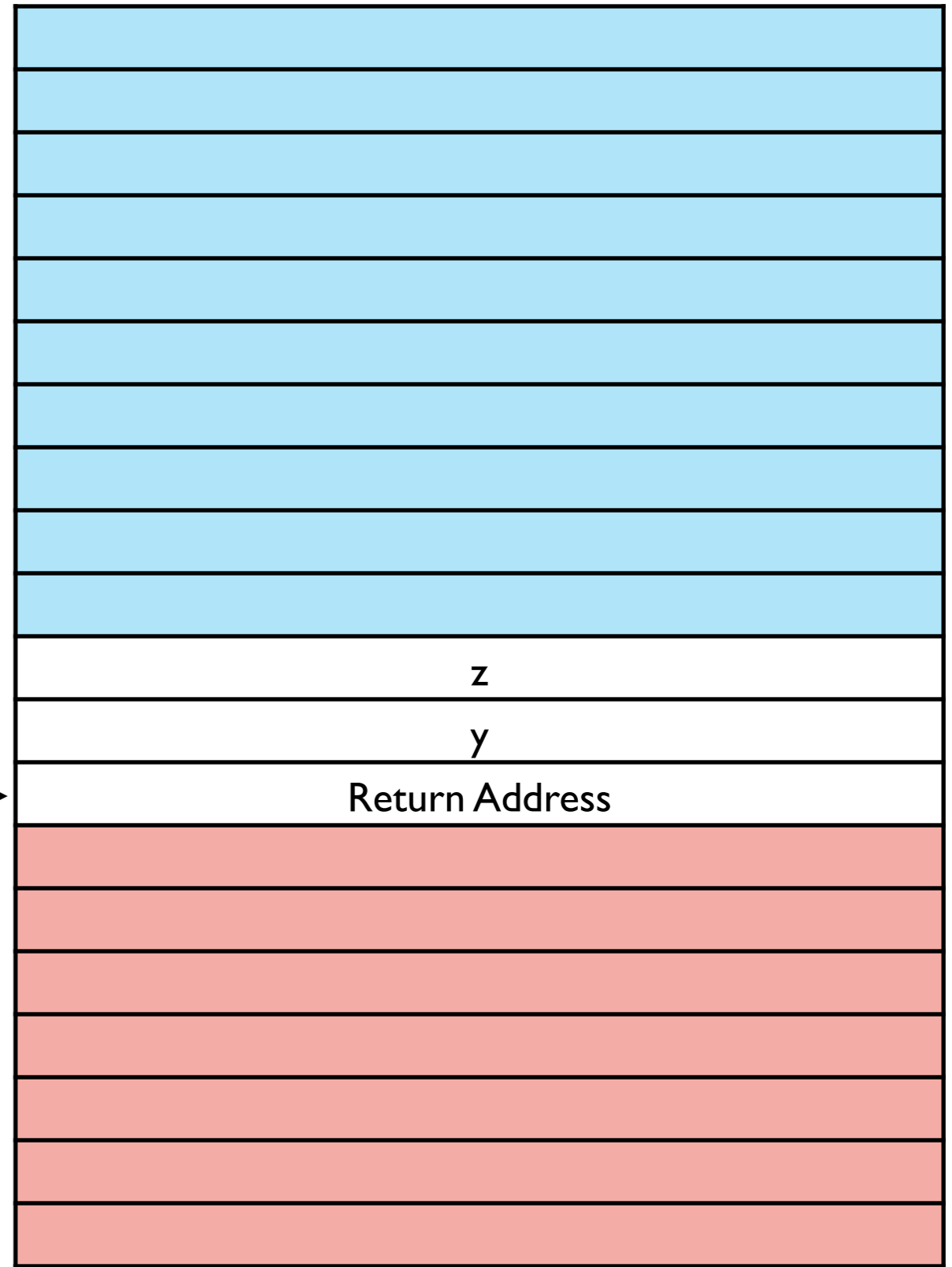
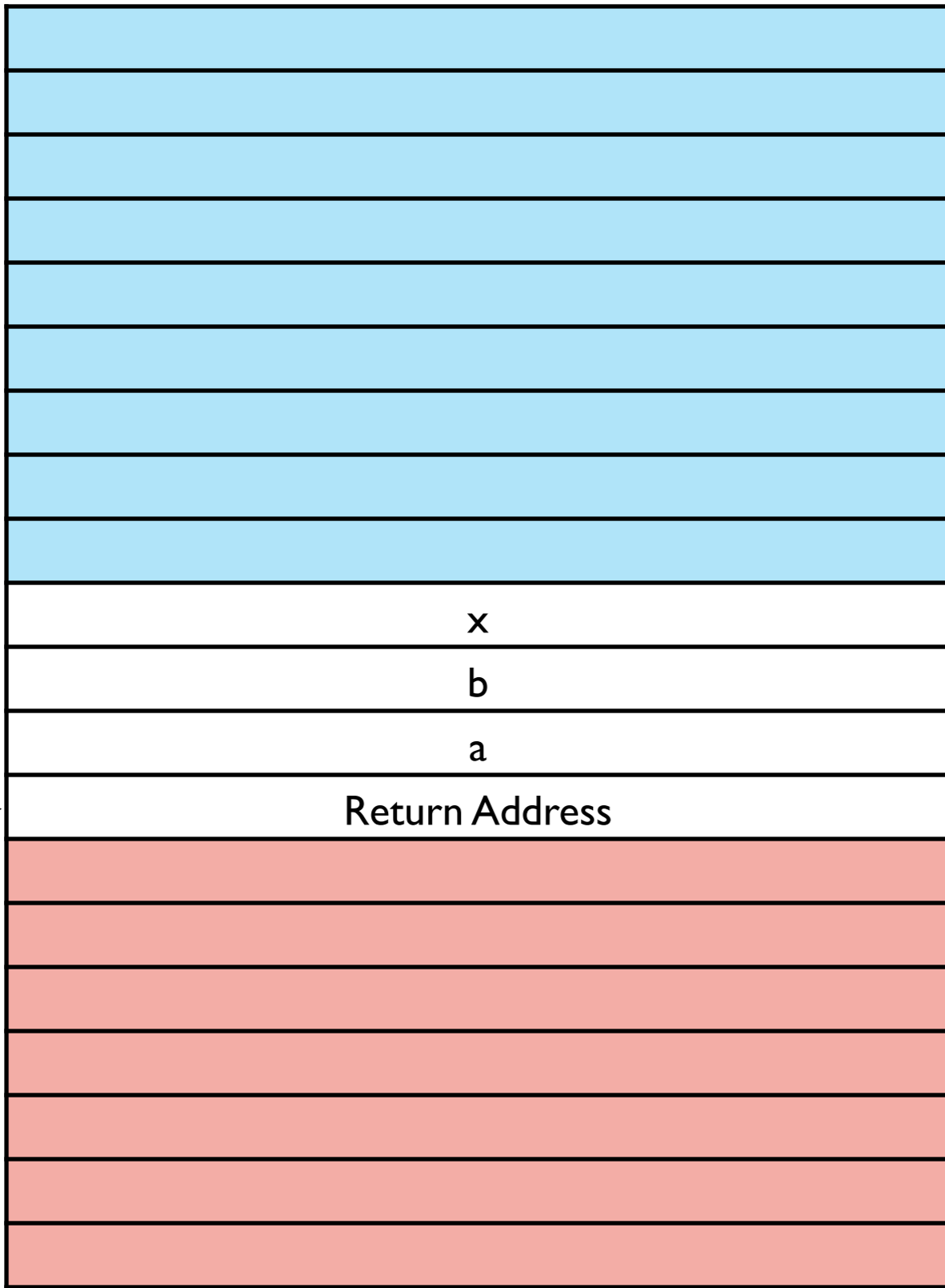
```
let b = 13 in
```

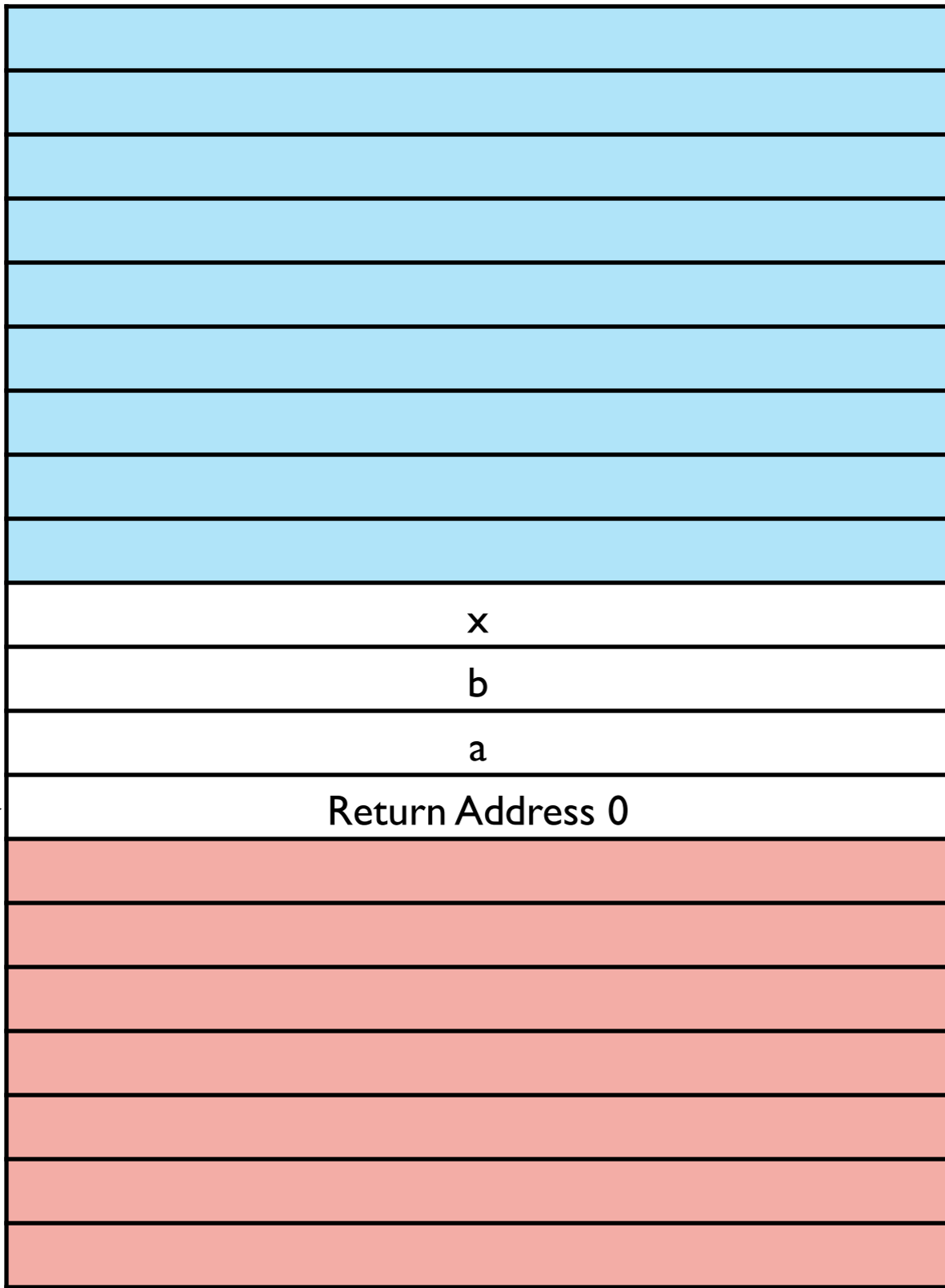
```
let x = f(a) in
```

```
x + b
```

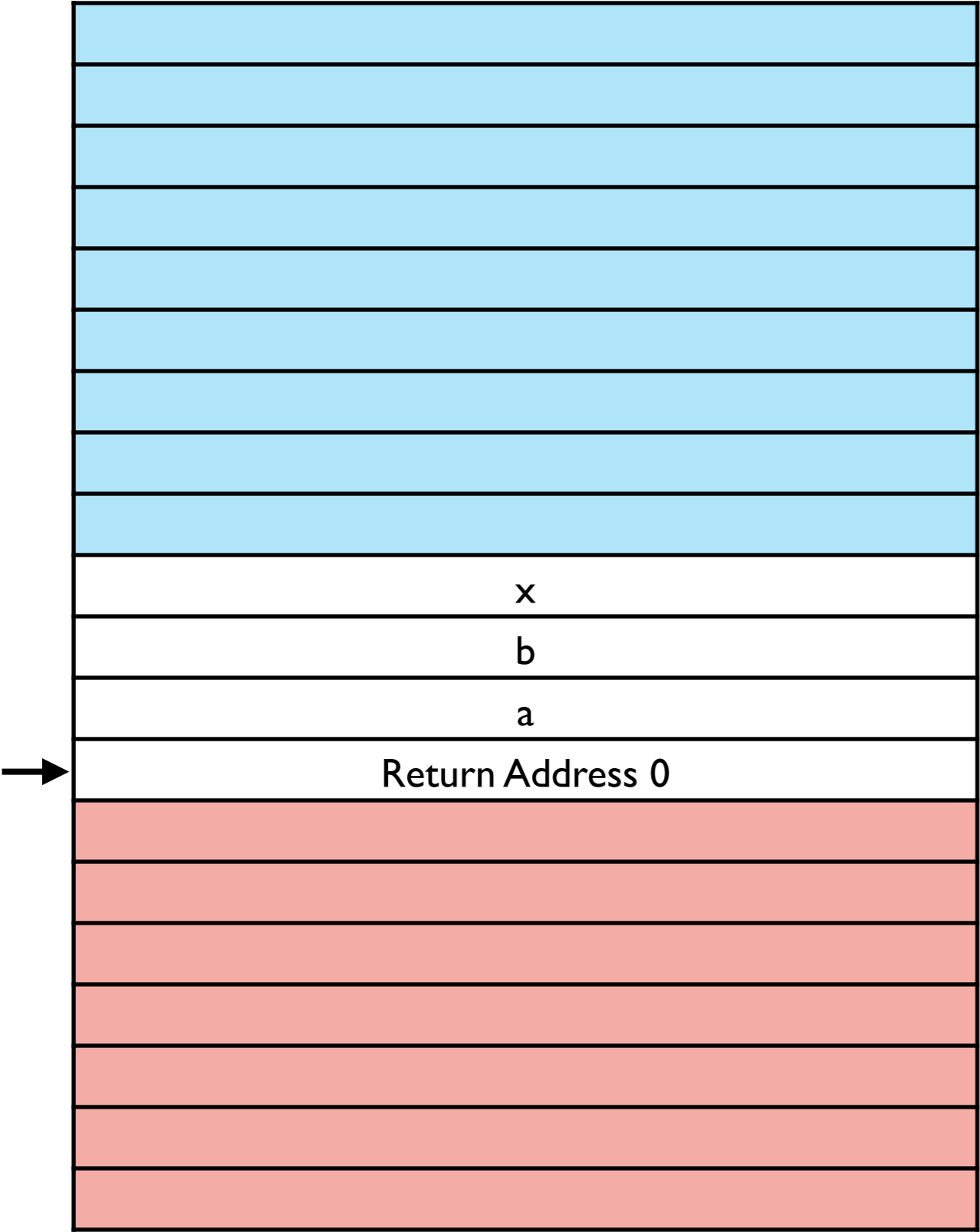
rsp →







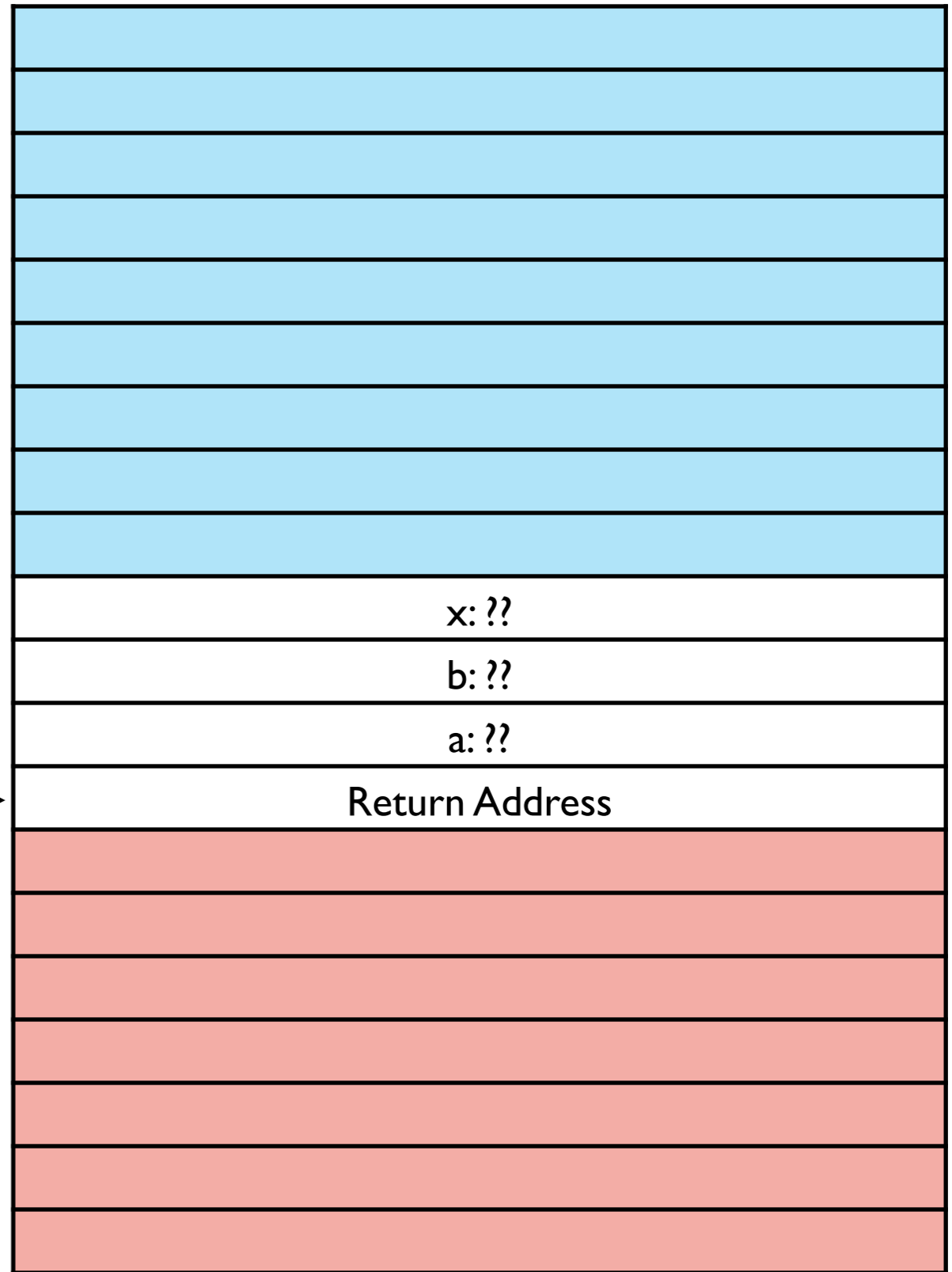
Address of a local is **relative** to the base of the stack frame



Stack

```
def f(y):  
    let z = 2 in  
        y * z  
end  
→ let a = 7 in  
    let b = 13 in  
        let x = f(a) in  
            x + b
```

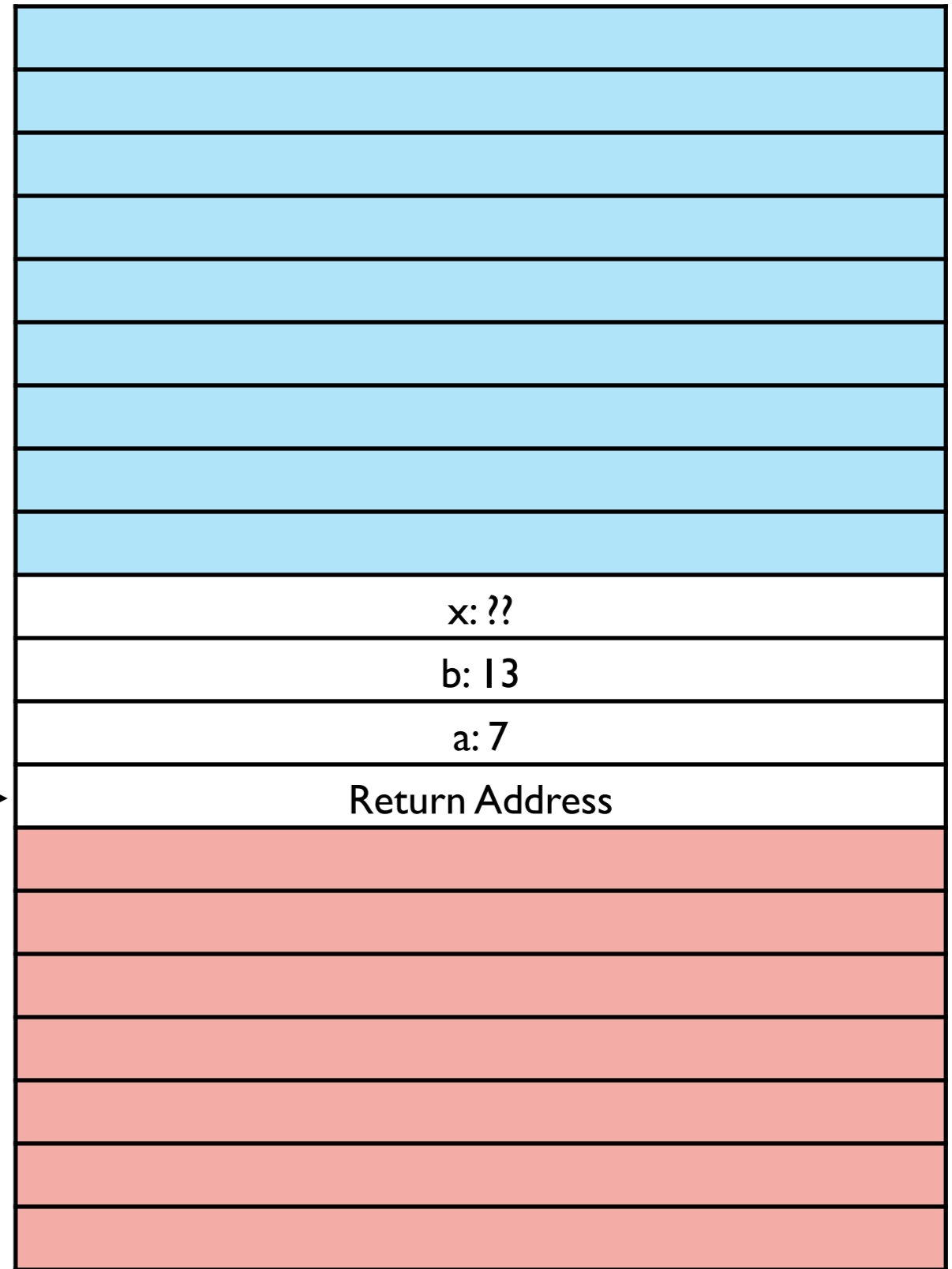
rsp →



Stack

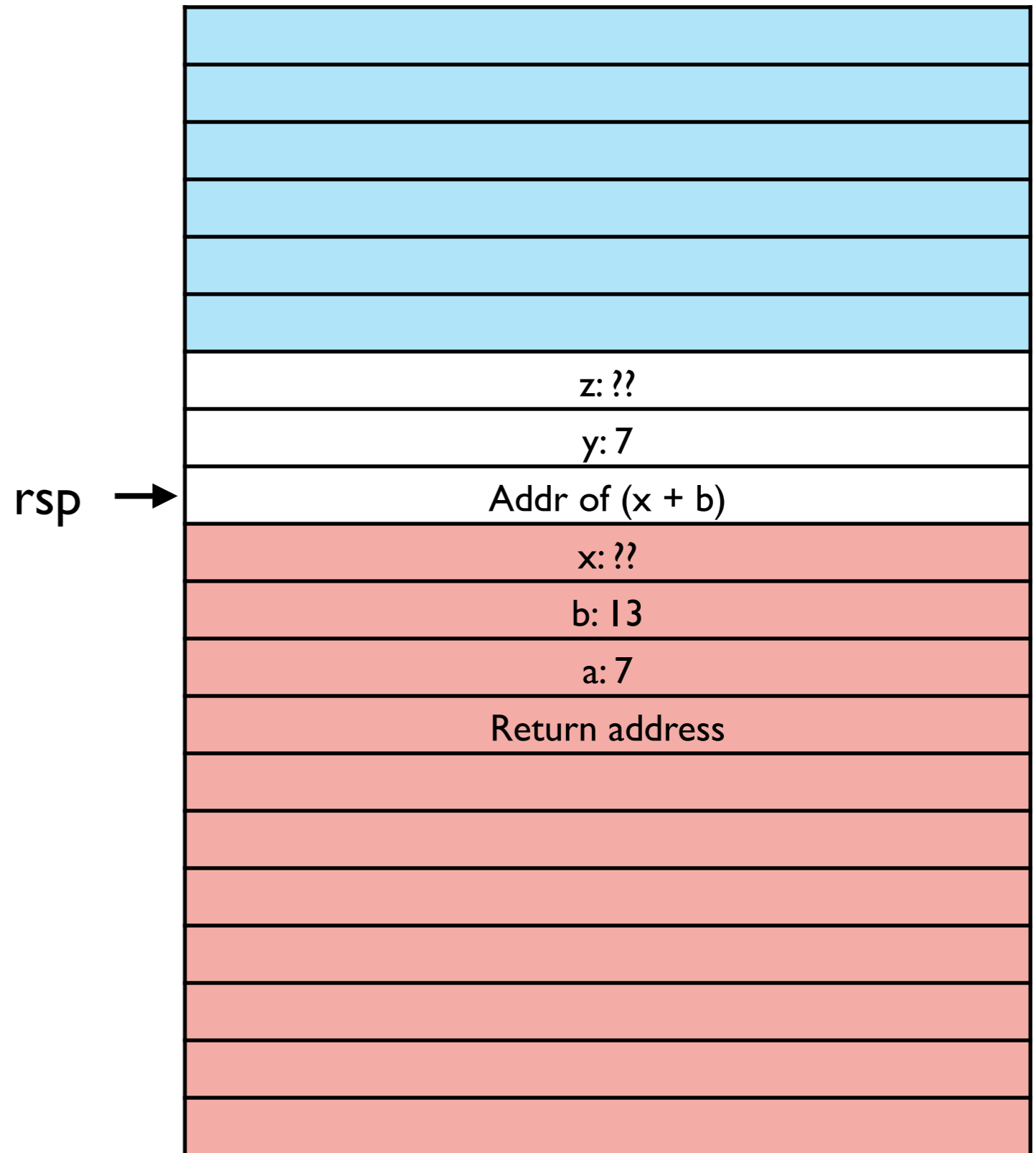
```
def f(y):  
  let z = 2 in  
  y * z  
end  
let a = 7 in  
let b = 13 in  
→ let x = f(a) in  
  x + b
```

rsp →




```
def f(y):  
→   let z = 2 in  
    y * z  
end  
let a = 7 in  
let b = 13 in  
let x = f(a) in  
x + b
```

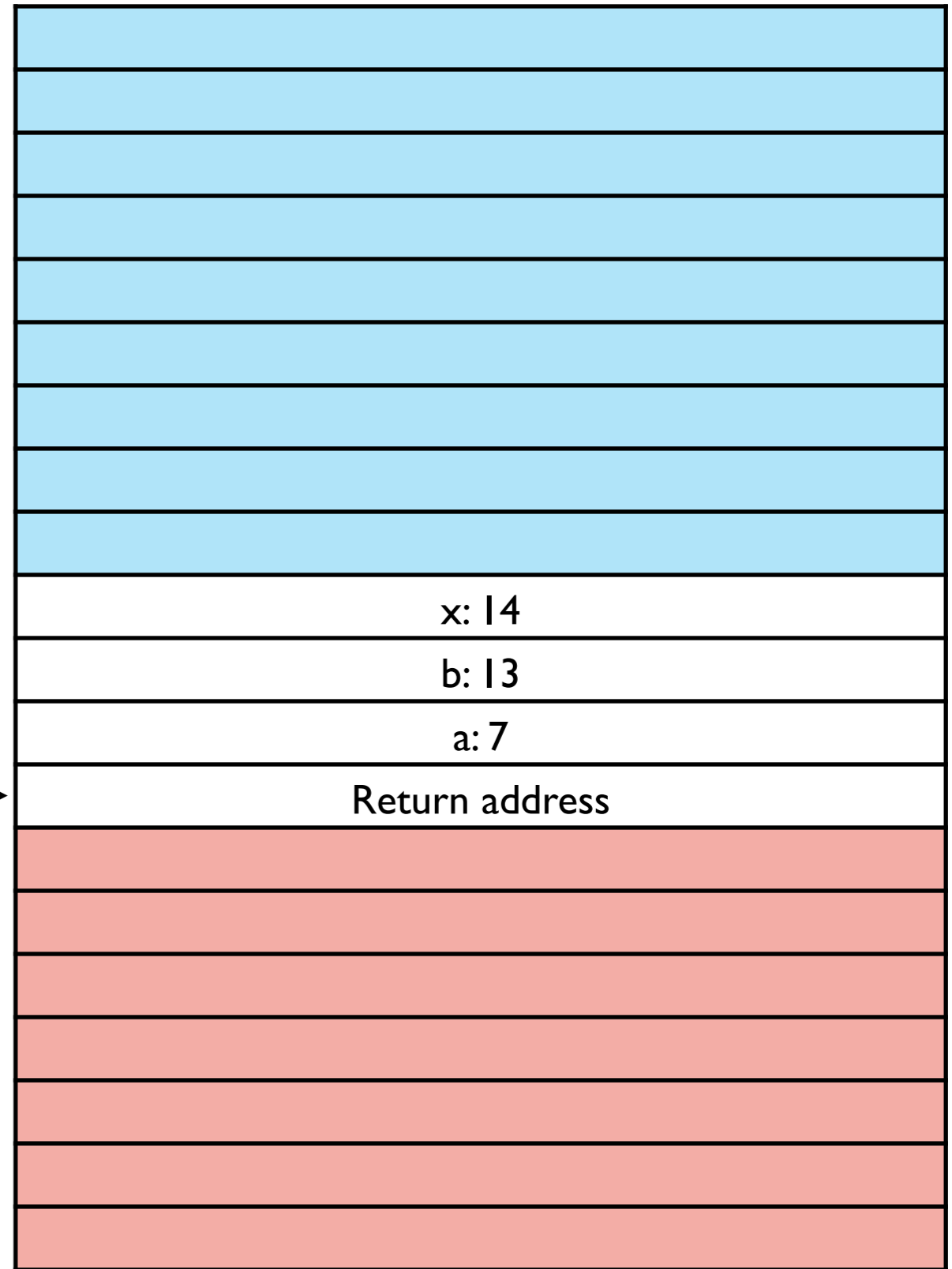
Stack



Stack

```
def f(y):  
    let z = 2 in  
    y * z  
end  
let a = 7 in  
let b = 13 in  
let x = f(a) in  
→ x + b
```

rsp →



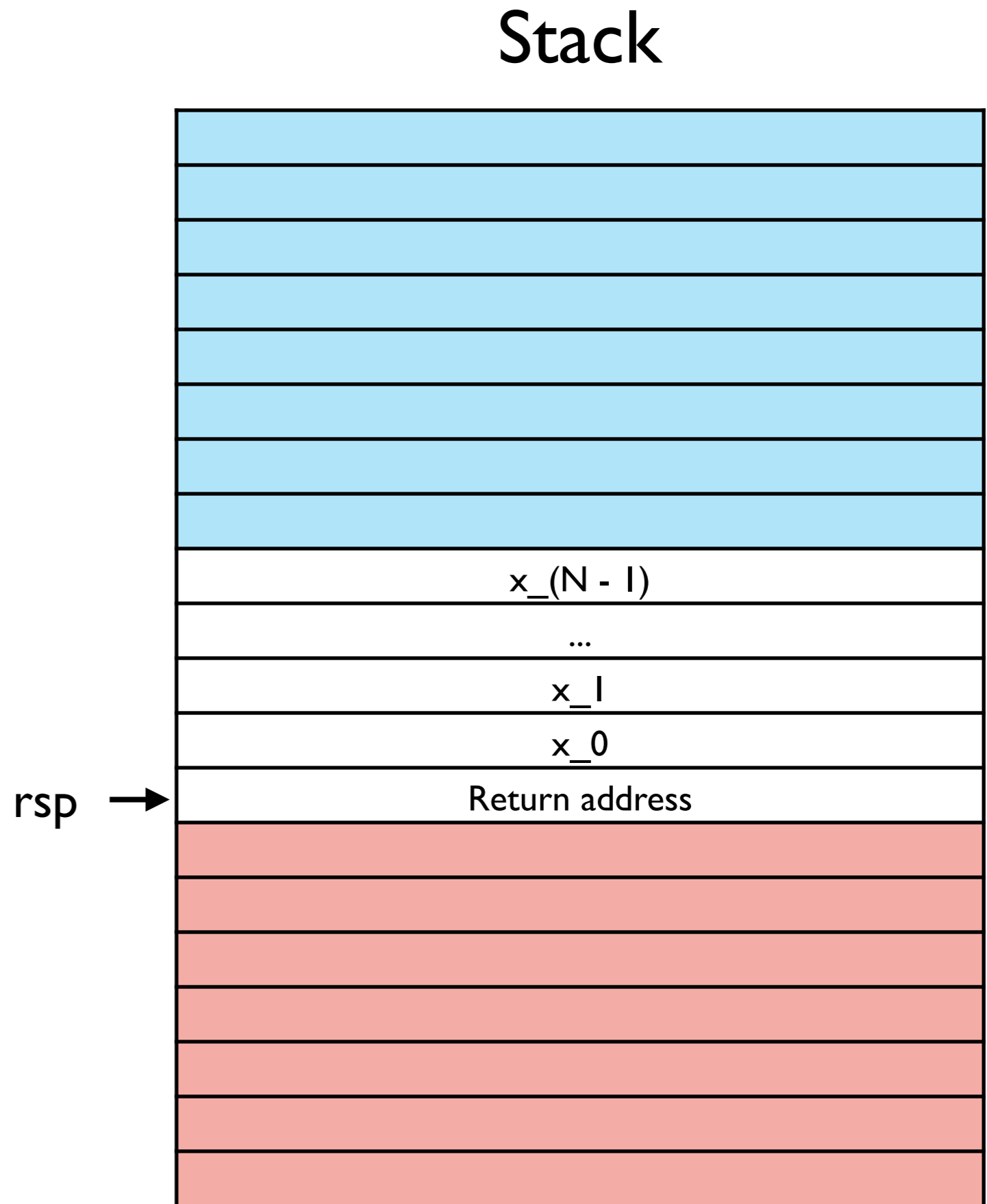
Snake Calling Convention (v0)

Snake Calling Convention (v0)

- Where are the arguments stored (relative to `rsp`)?
- Where is the return address stored (rel to `rsp`)?
- Where should we put the return value?
- Where should `rsp` point when we return?
- We don't use registers yet so don't worry about them

To call `f` with
`N` arguments
`x_0` to `x_(N-1)`

- `Rsp` points at the return address
- Args go at lower addresses, with first arg closest to `rsp`
- Caller's stack frames are at higher addresses
- Return value in `RAX`
- Upon return, `rsp` points to top of caller's stack frame



Benefits of Snake CC

- Easy to implement tail calls
- Arguments are treated uniformly with locals
- Return overlaps with SysV CC when there are no stack args

WARNING

WARNING

- RBP

WARNING

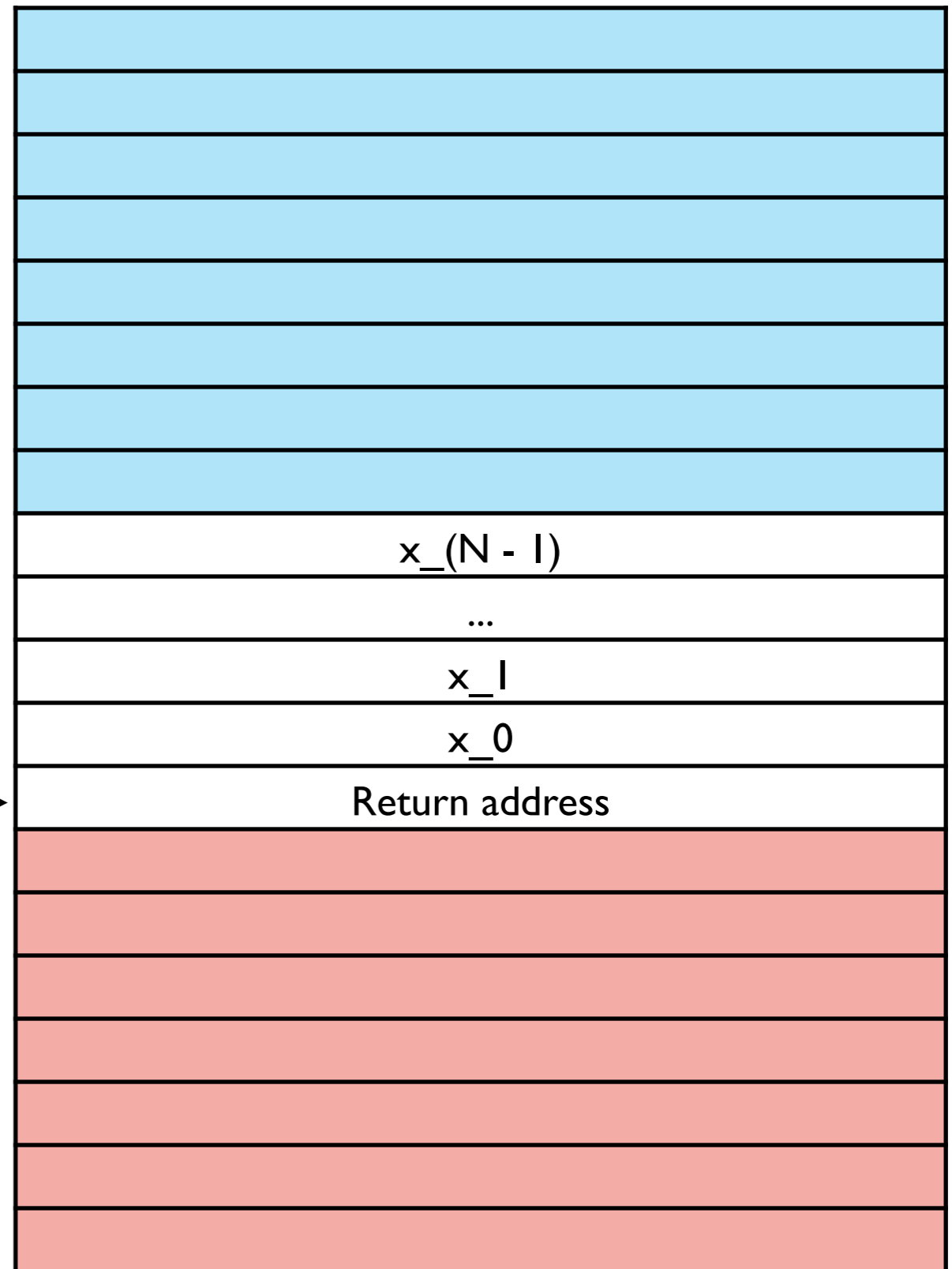
- RBP
- Different from SysV ABI:
 - No register args (for now)
 - Placement of Args
 - "Caller-cleanup" vs "Callee-cleanup"

To call `f` with
`N` arguments
`x_0` to `x_(N-1)`

SYSCALL ABI

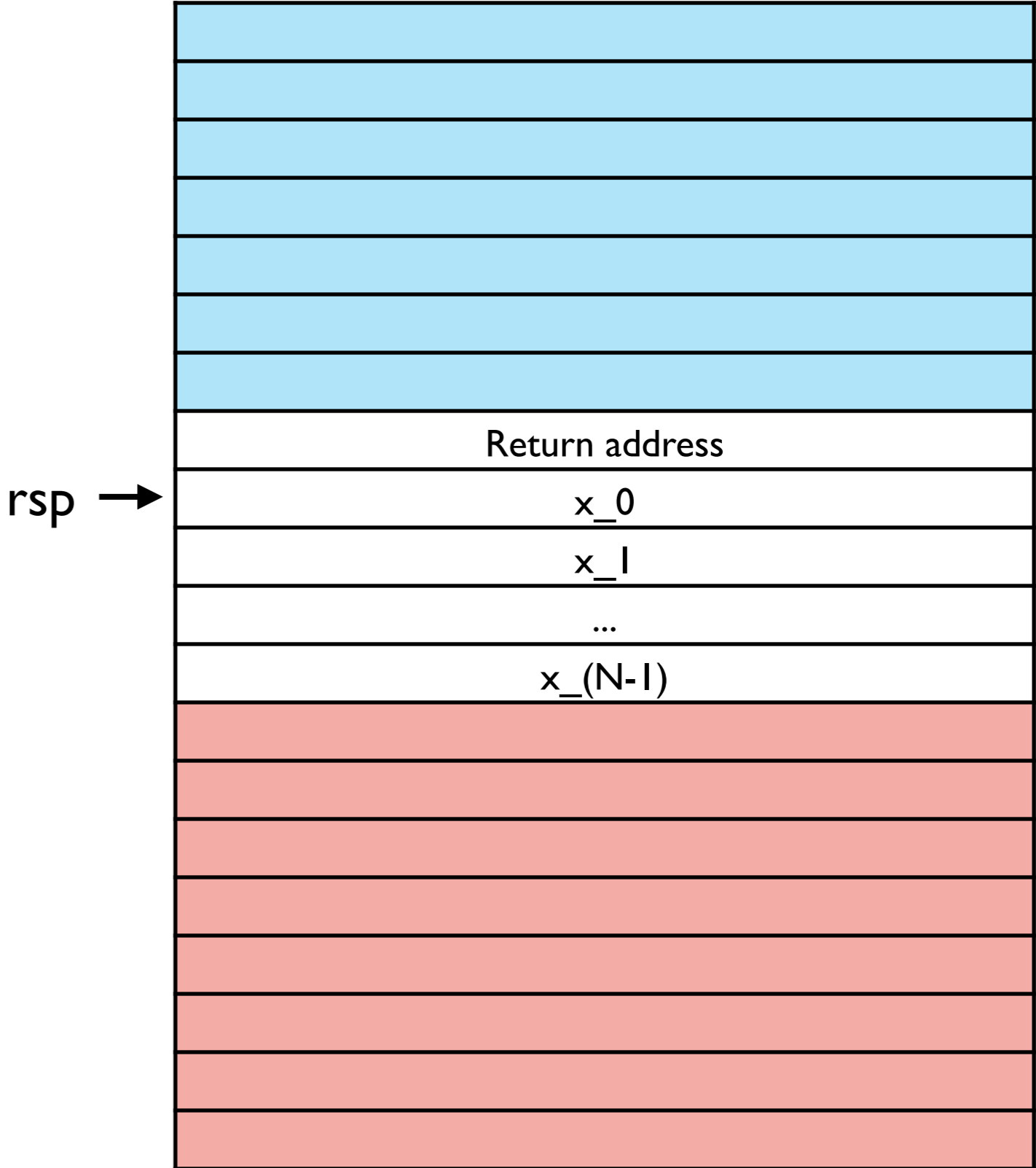
`rsp` →

Stack



To call f with
N arguments
 x_0 to $x_{(N-1)}$

Stack



SysV
after return

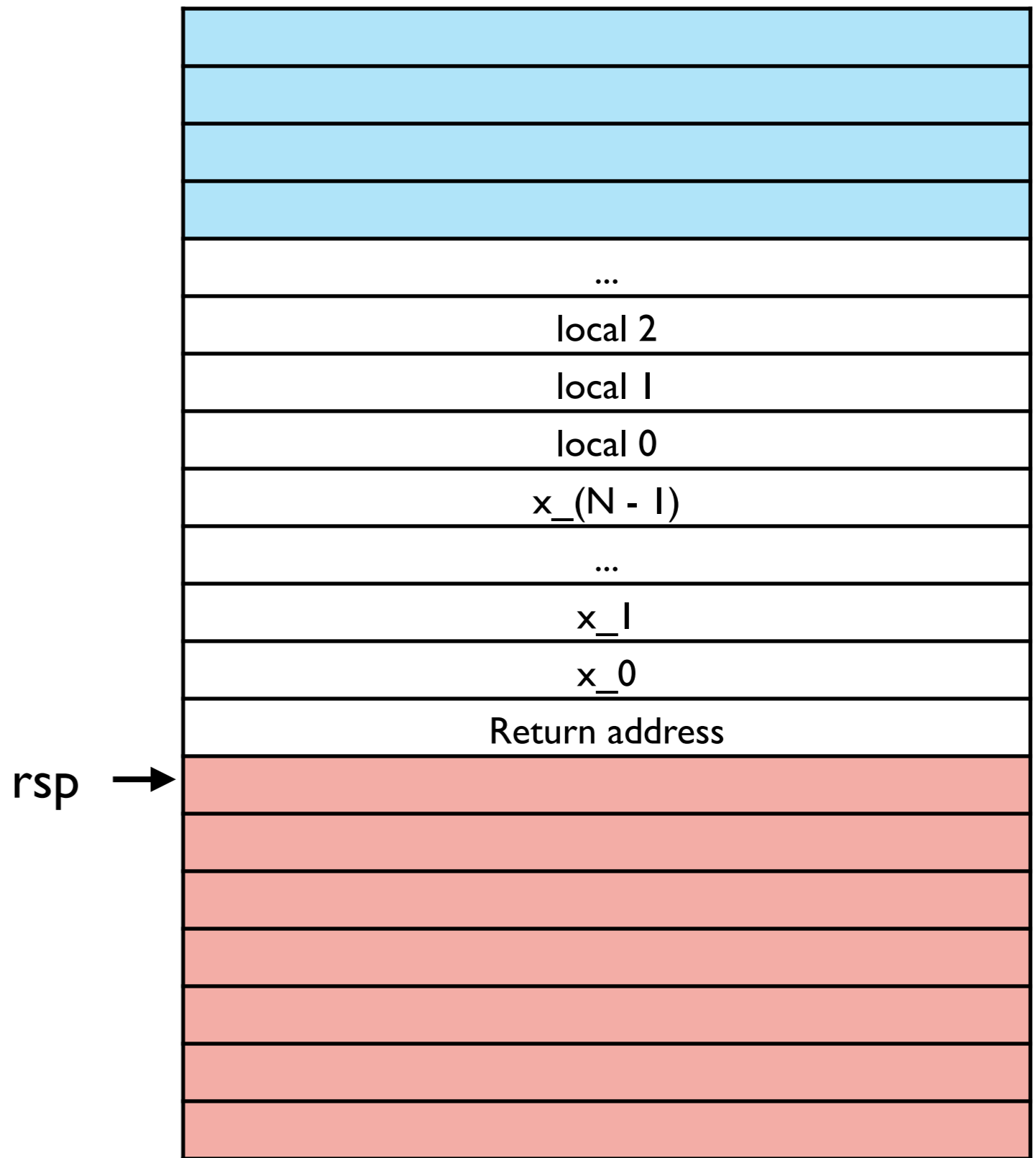
Implementing the Snake CC

- How to return?
- How to call?
- How to tail call?

To return x

```
mov rax, [loc of x]
pop rbx
jmp rbx
```

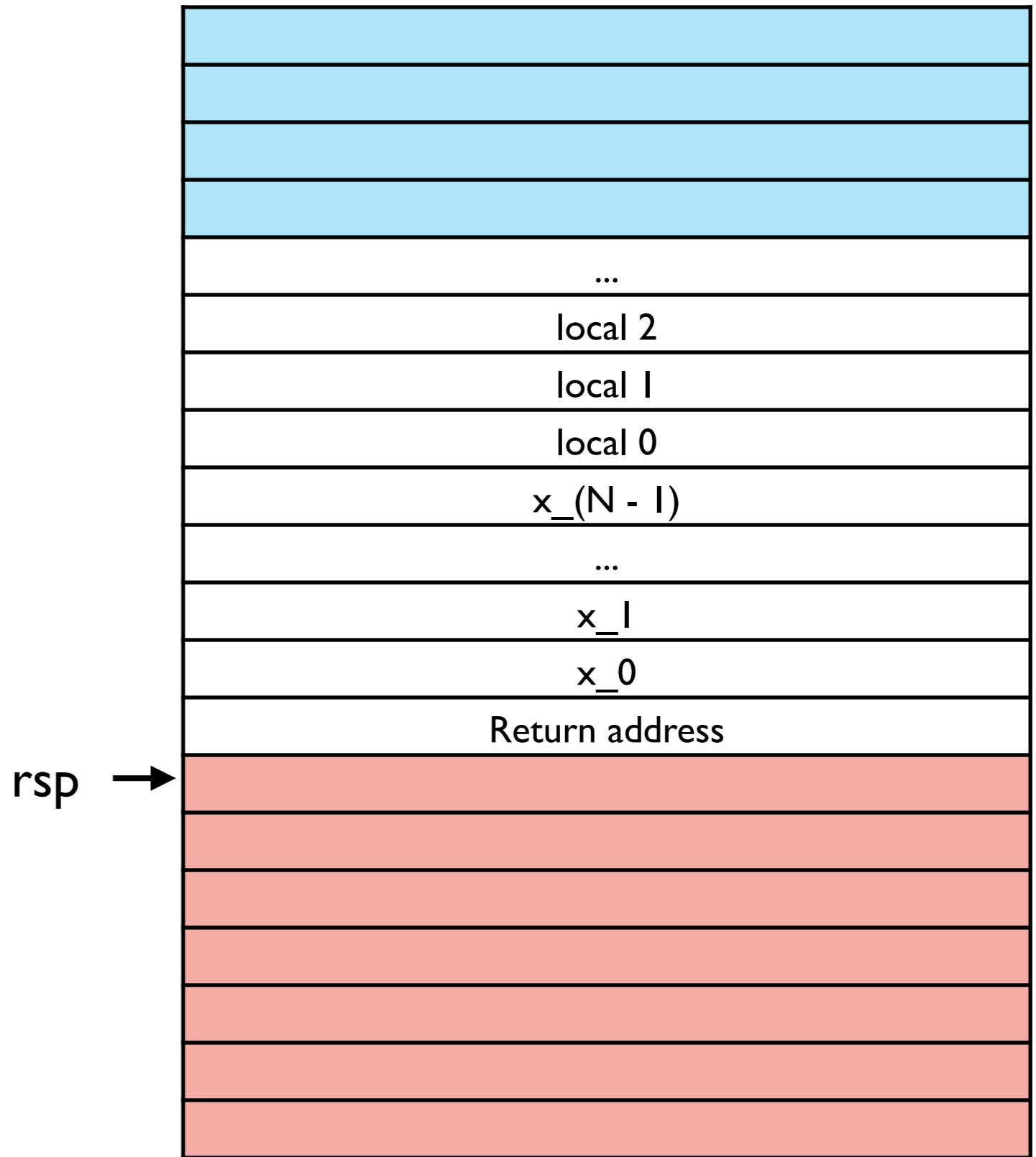
Stack



To return x

```
mov rax, [loc of x]  
ret
```

Stack



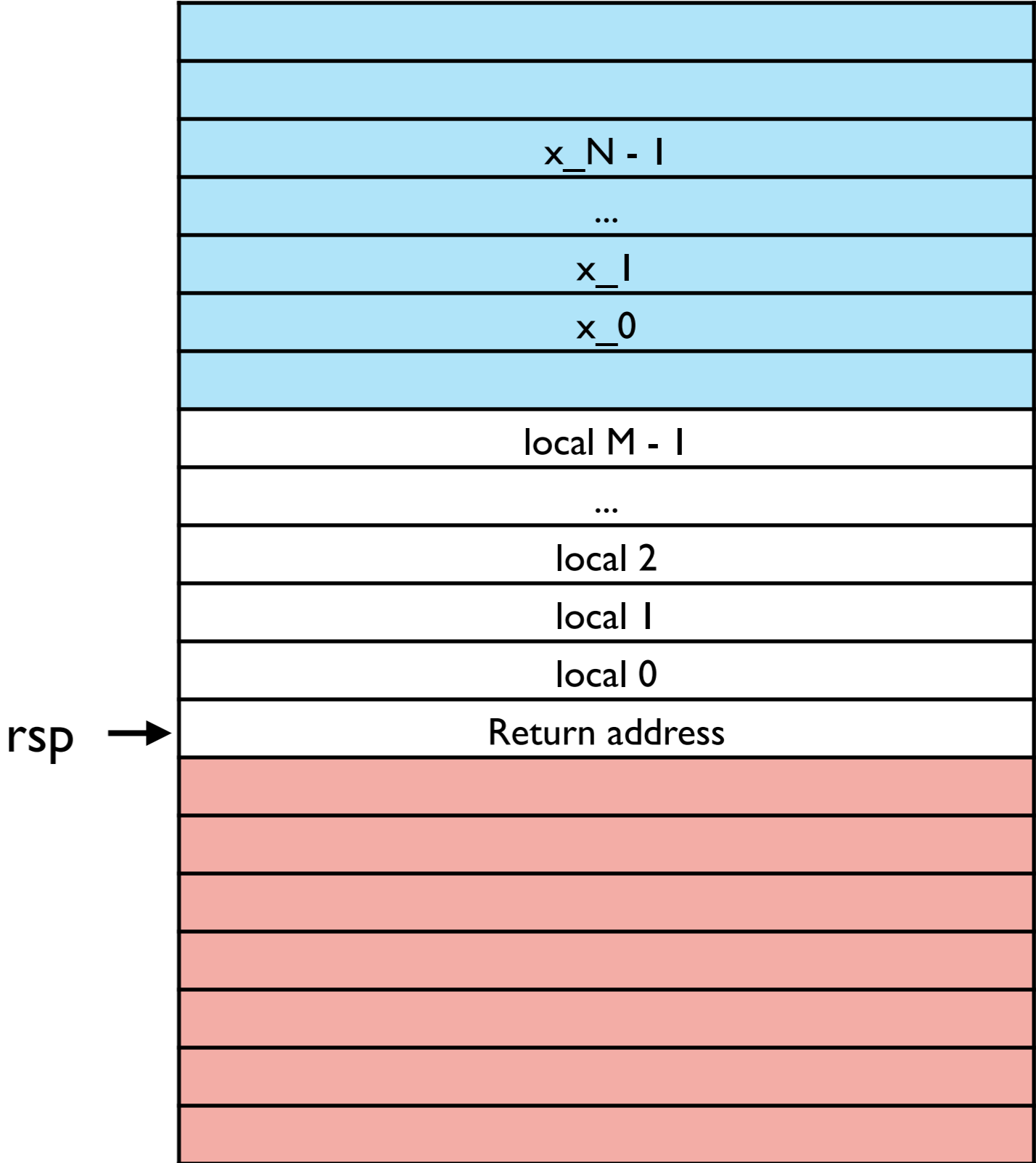
To call `f` with
`N` arguments
`x_0` to `x_(N-1)`

Stack



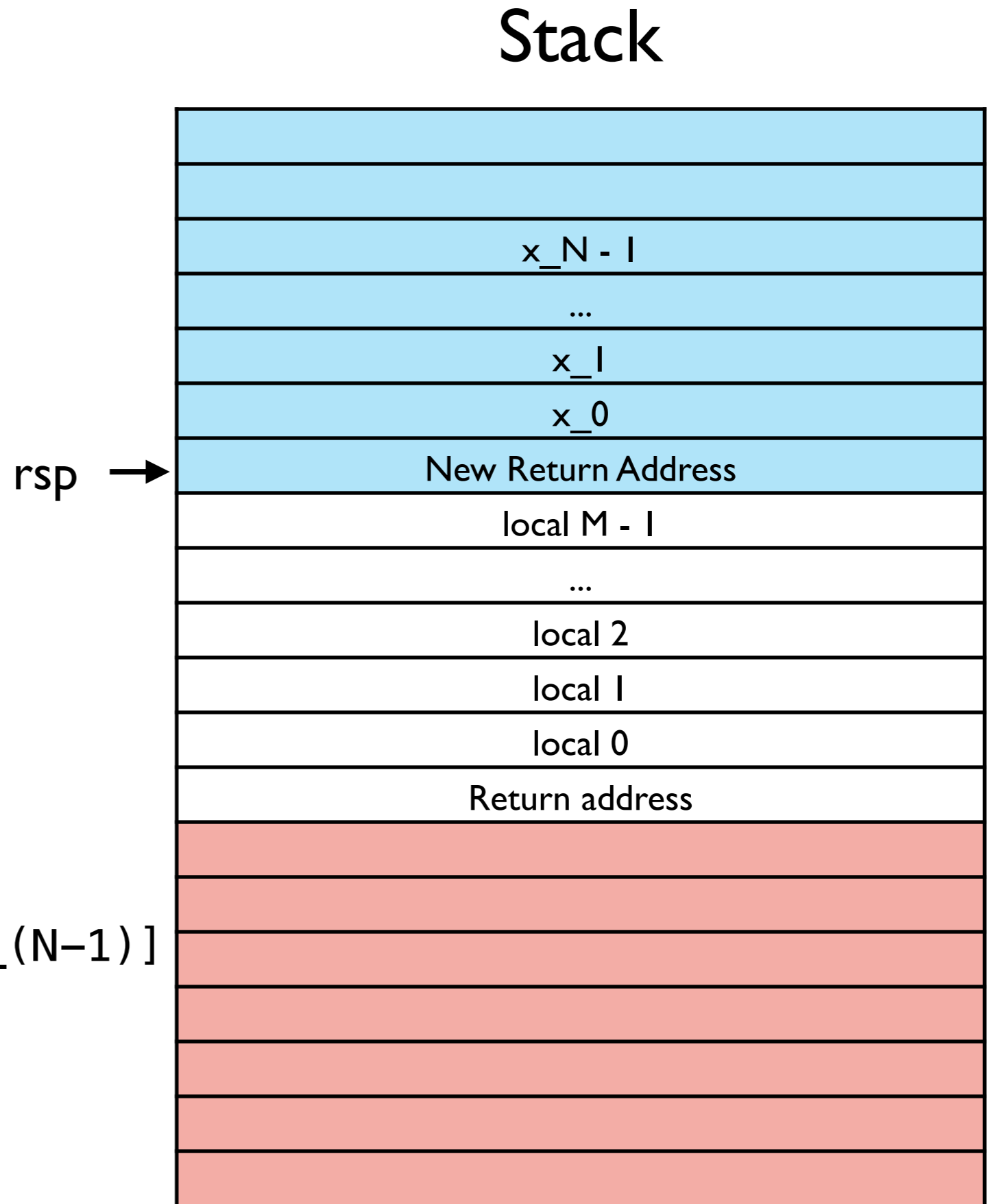
To call f with
N arguments
 x_0 to $x_{(N-1)}$

Stack



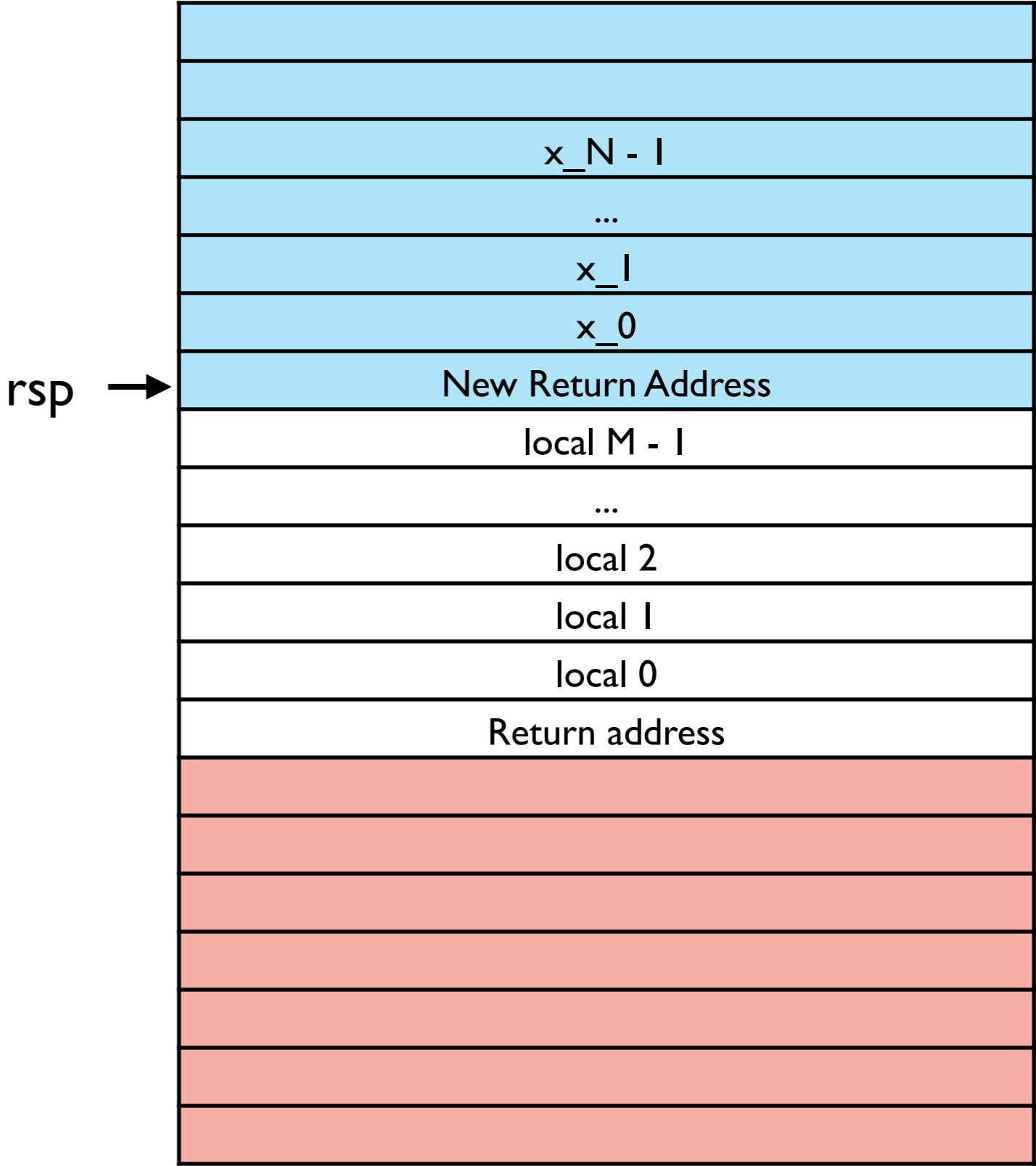
To call `f` with
`N` arguments
`x_0` to `x_(N-1)`

```
mov [rsp - 8 * (M + 2)], [x_0]
mov [rsp - 8 * (M + 3)], [x_1]
...
mov [rsp - 8 * (M + 2 + N)], [x_(N-1)]
sub rsp, 8 * M
call
```



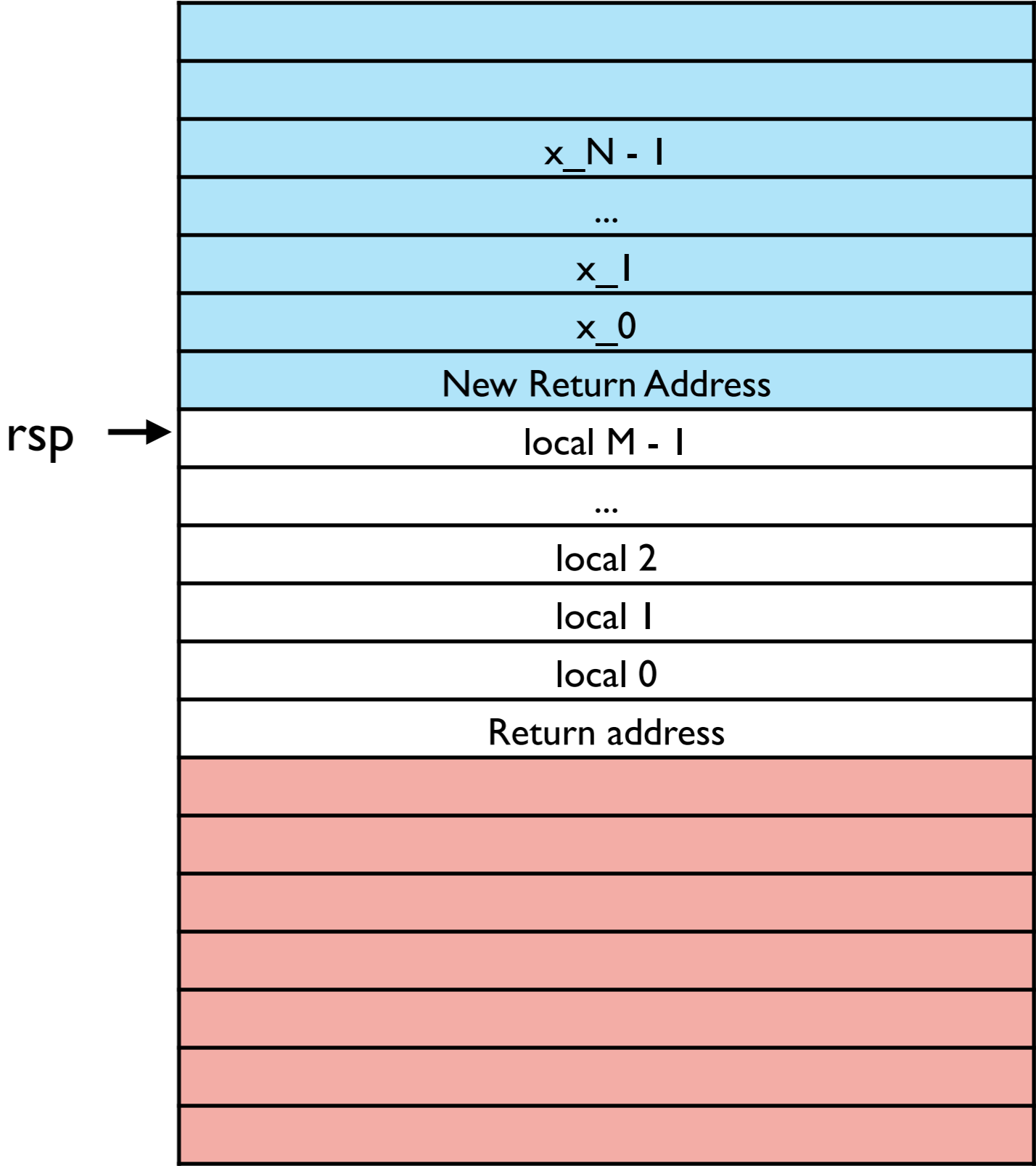
After the call...

Stack



After the call...

Stack



To TAIL call f
with
N arguments

Stack



To TAIL call f
with
N arguments

Stack



To TAIL call f
with
N arguments

if $N < M$

rsp →

Stack



To TAIL call f
with
N arguments

if $N > M$

rsp →

Stack



To TAIL call f
with
N arguments

Careful not to
overwrite locals we are using!

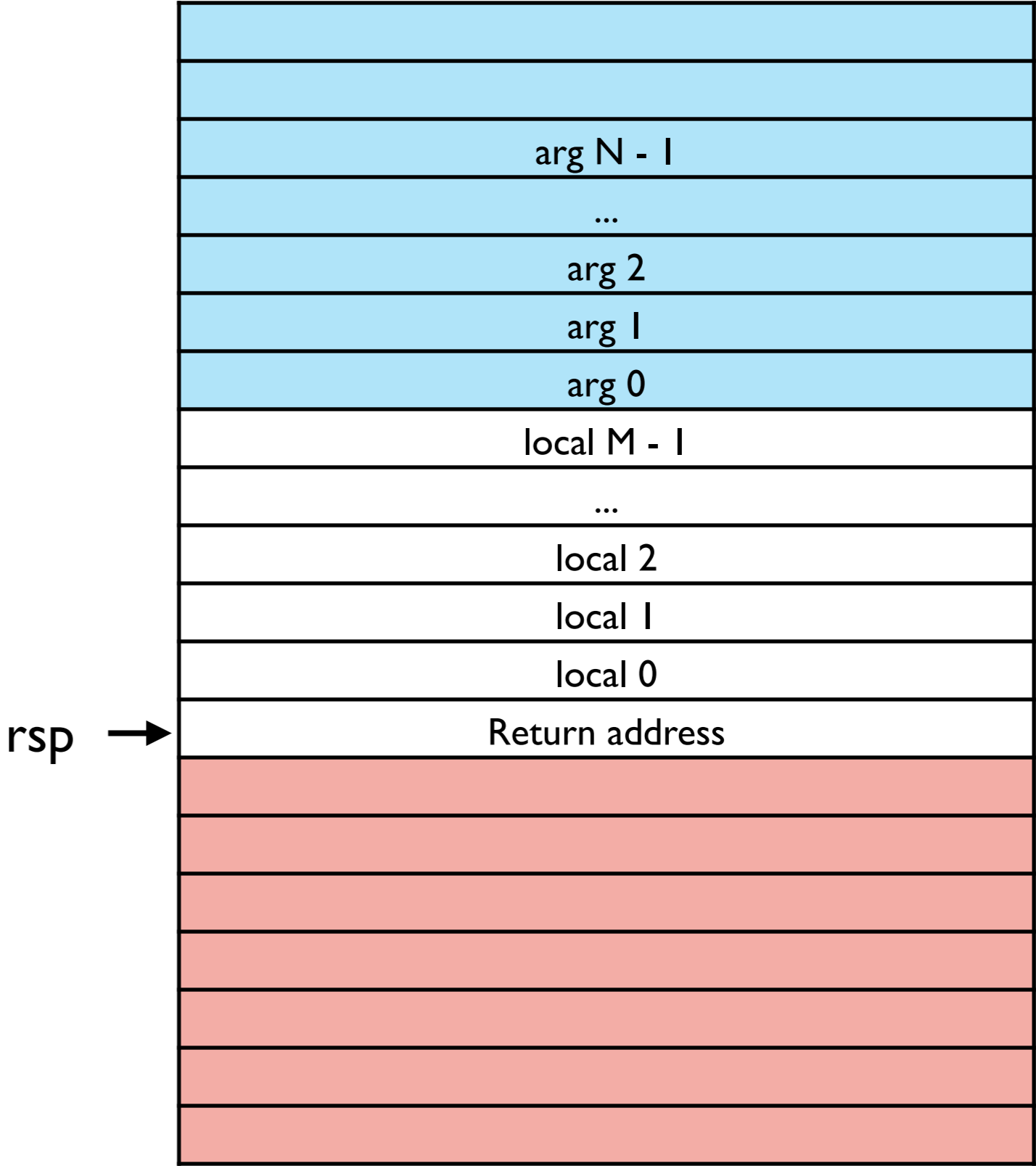
rsp →

Stack



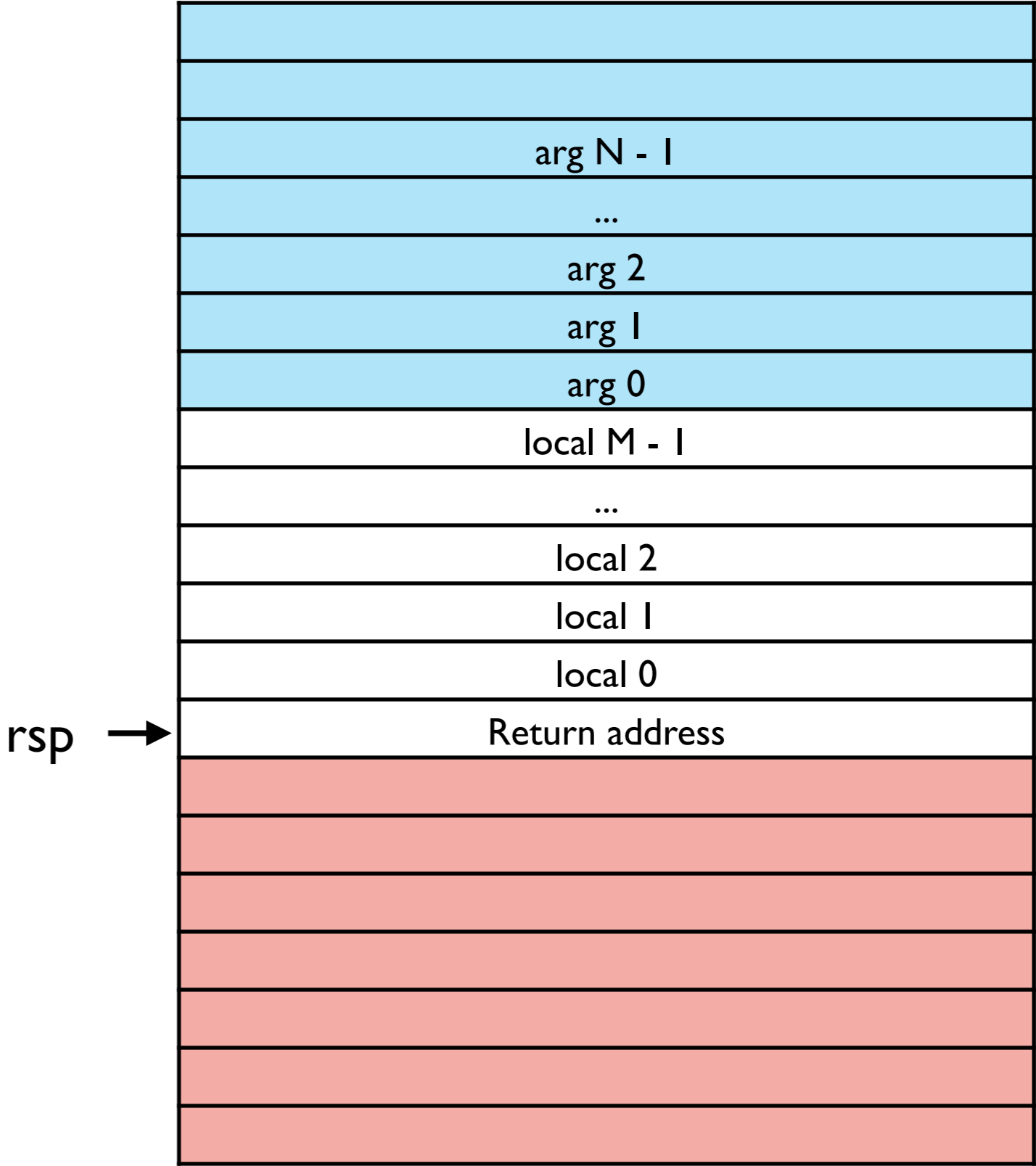
To TAIL call f
with
N arguments

Stack



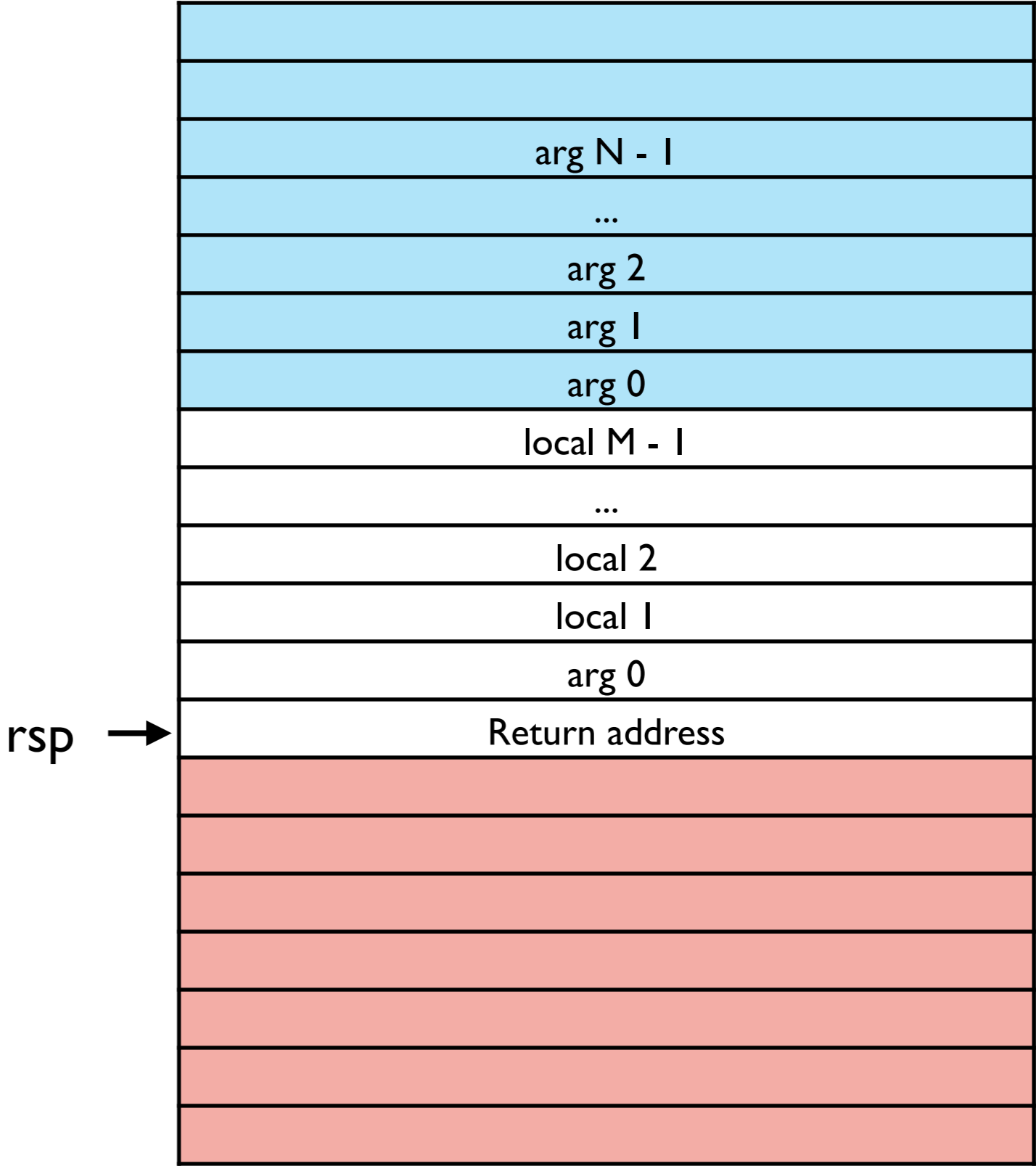
To TAIL call f
with
N arguments

Stack



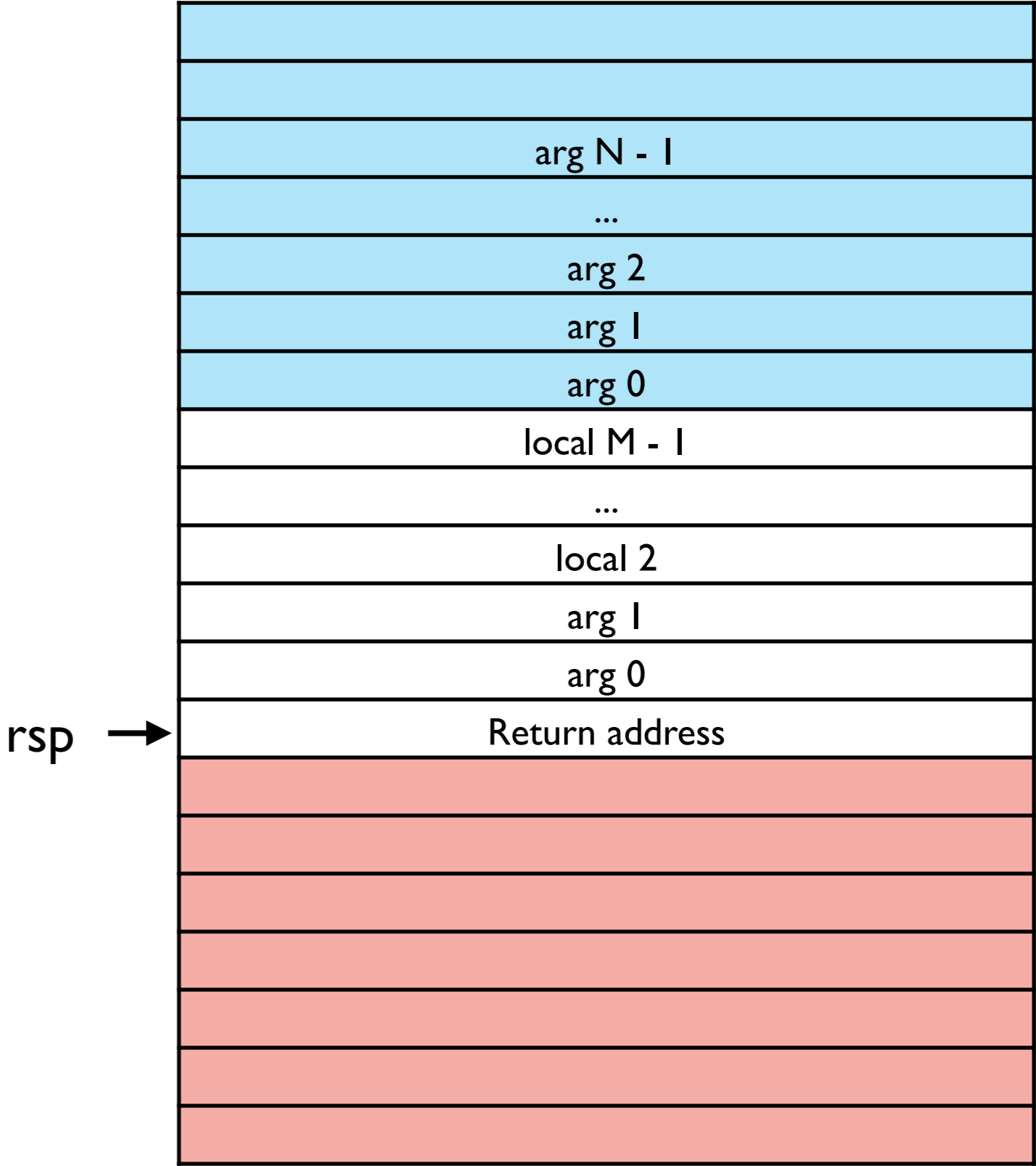
To TAIL call f
with
N arguments

Stack



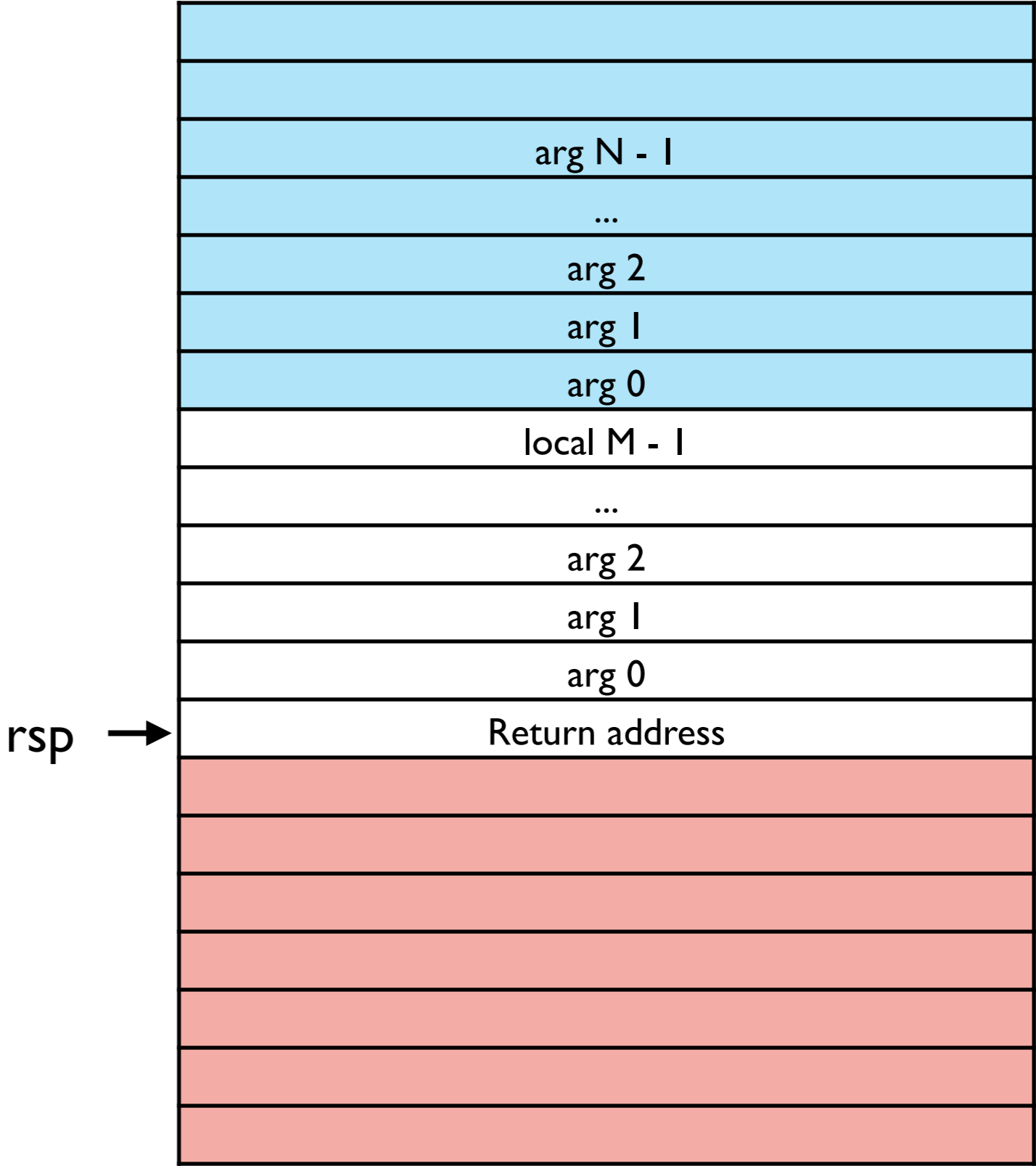
To TAIL call f
with
N arguments

Stack



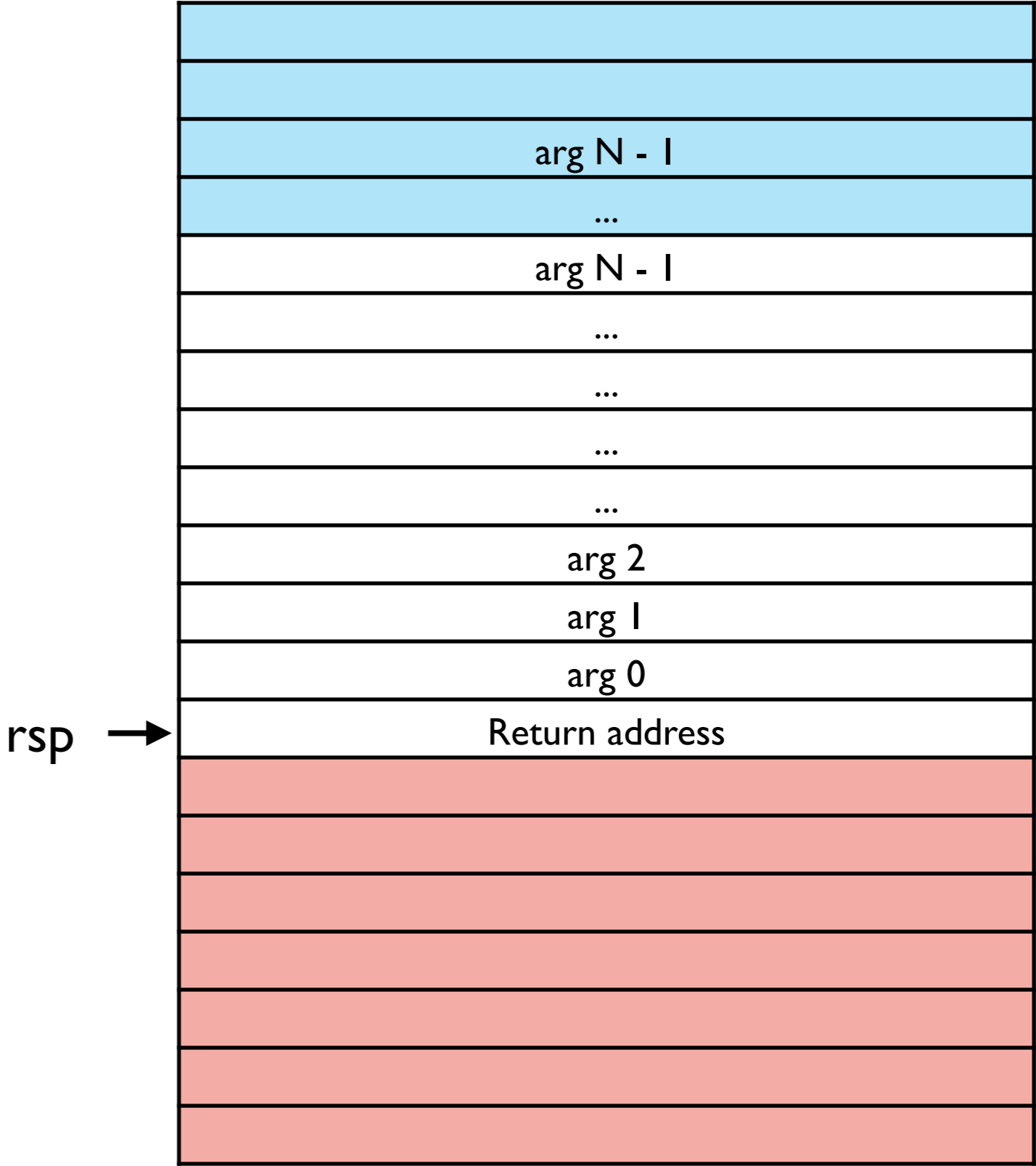
To TAIL call f
with
N arguments

Stack



To TAIL call f
with
N arguments

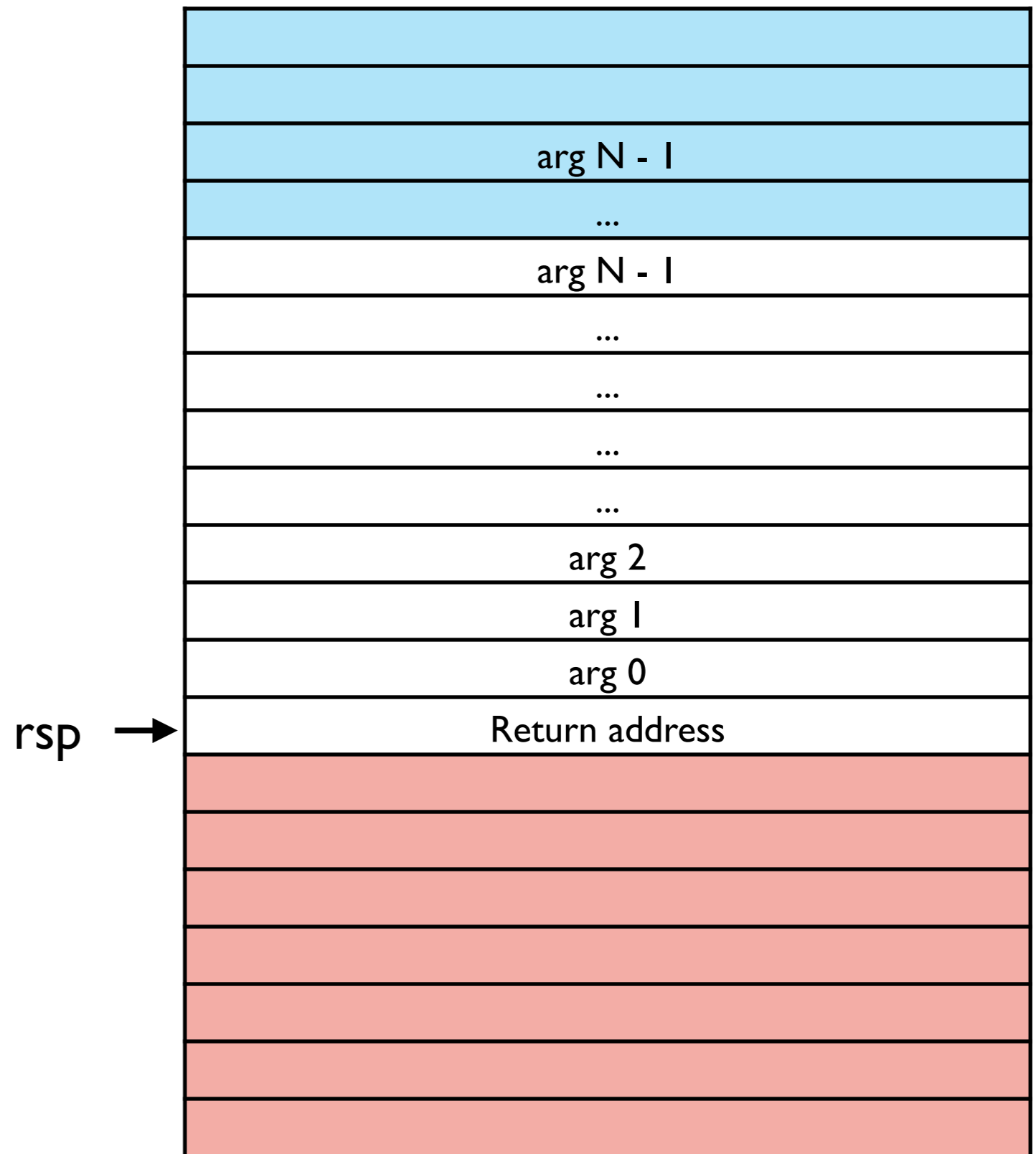
Stack



To TAIL call f
with
N arguments

$f(x_0, \dots, x_{(N-1)})$

Stack



To TAIL call f
with
N arguments

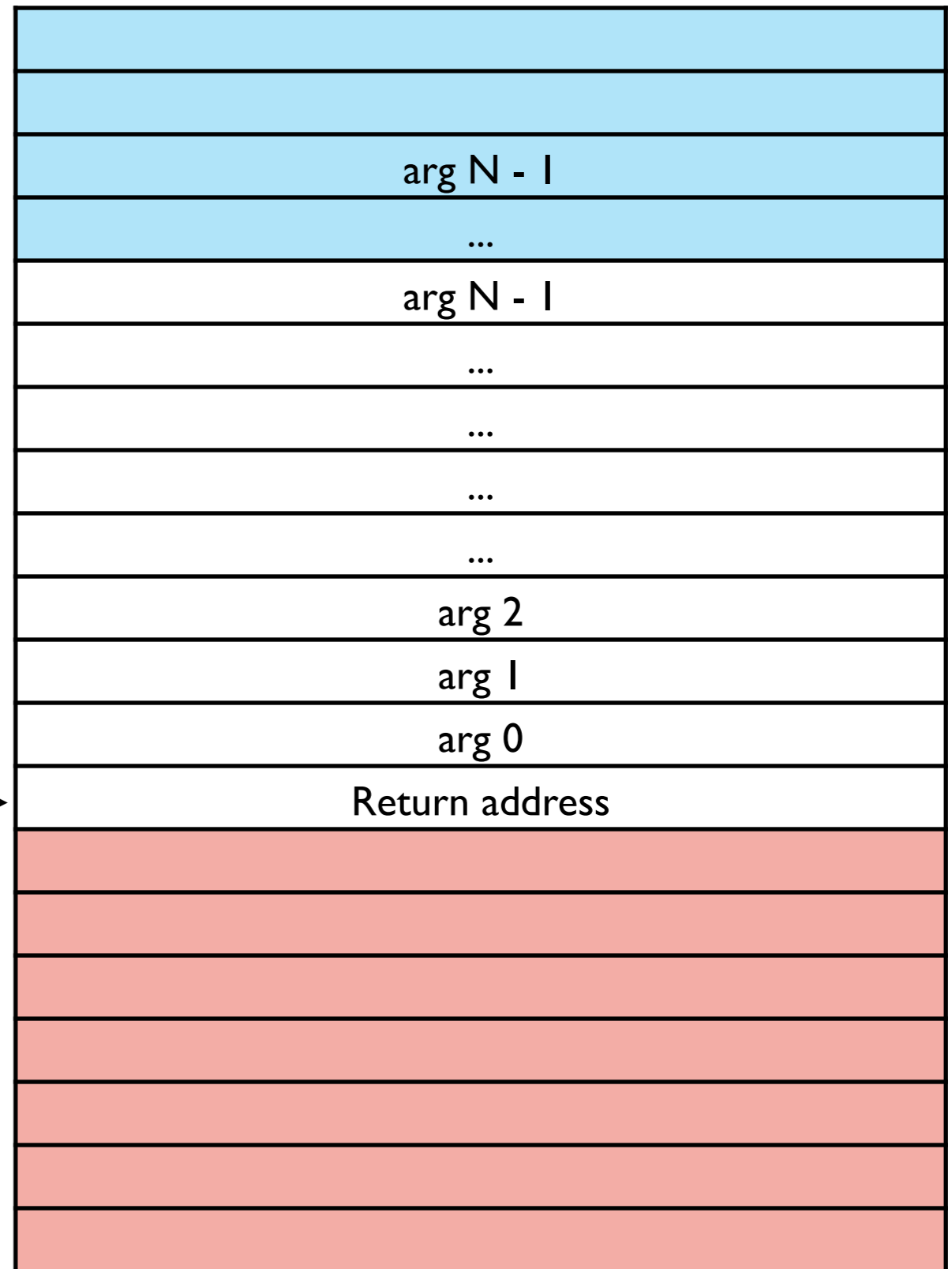
$f(x_0, \dots, x_{(N-1)})$

let $y_0 = x_0, \dots$
in

$f(y_0, \dots, y_{(N-1)})$

rsp →

Stack



generate these "unnecessary"
temporaries in sequentialize

Alignment(!)

- We want to be able to call into Rust with the Sys V CC at any time
- But *that* calling convention has an alignment restriction.
- So to make it easy to implement that alignment, we should require a similar alignment so that we don't have to check alignment *dynamically*.
- tradeoff: we use potentially? more space to avoid branches at runtime (v slow)

Alignment(!)

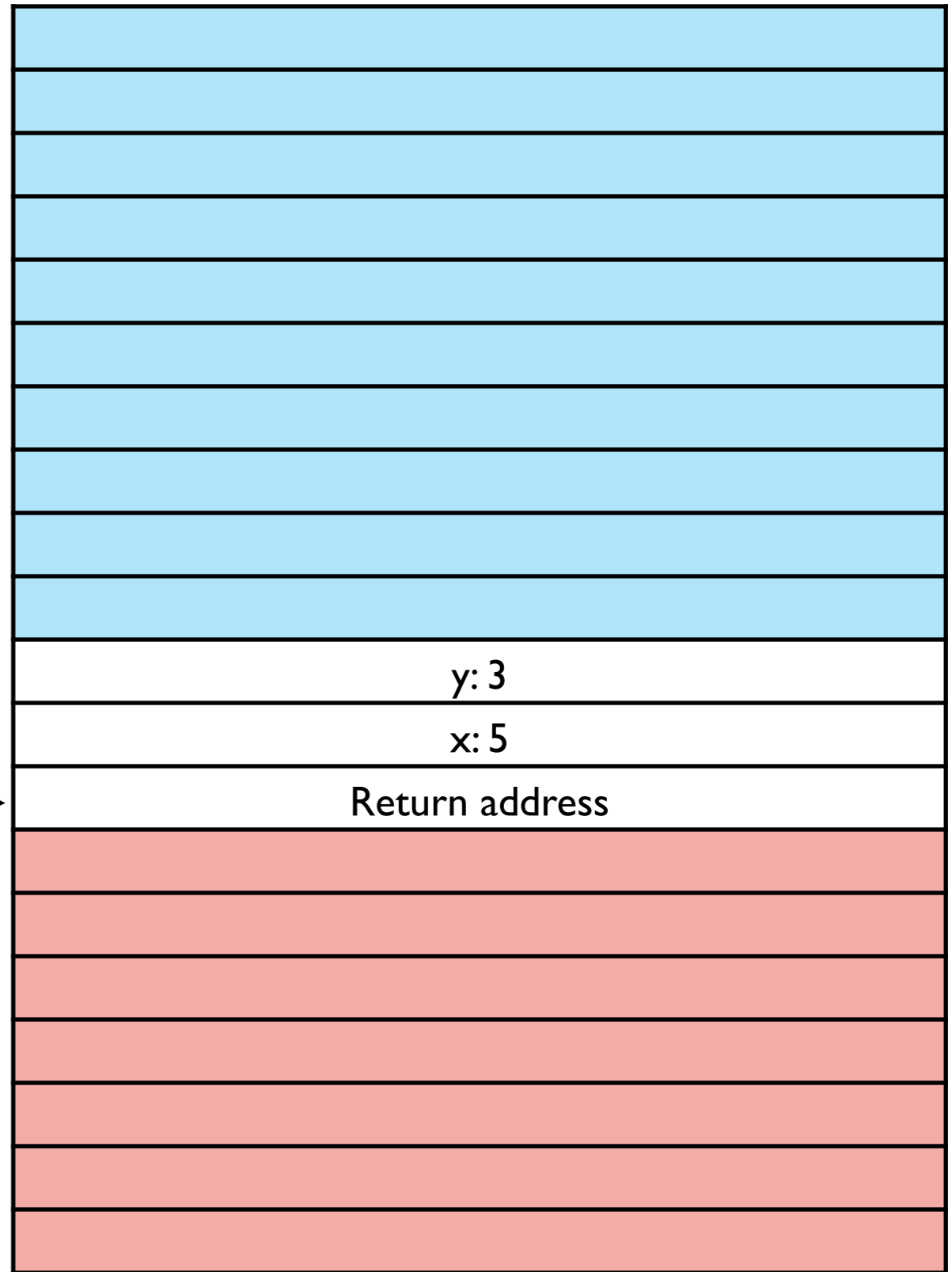
- But Sys V has the following alignment requirement:
 - Upon entry into a function, $\text{rsp} + 8 \% 16 == 0$
- To make this statically determined, we add a similar alignment requirement to Snake:
 - Upon entry into a function, $\text{rsp} \% 16 == 0$
 - This way if we have no saved locals, we can just call
 - Sometimes need to include a dummy local

```
def multiply(x, y):  
    def loop(y):  
        if y == 0:  
            0  
        else:  
            x + loop(y - 1)  
        end  
        loop(y)  
    end  
multiply(5, 3)
```


Stack

```
def multiply(x, y):  
    def loop(y):  
        if y == 0:  
            0  
        else:  
            x + loop(y - 1)  
        end  
    → loop(y)  
end  
multiply(5, 3)
```

rsp →

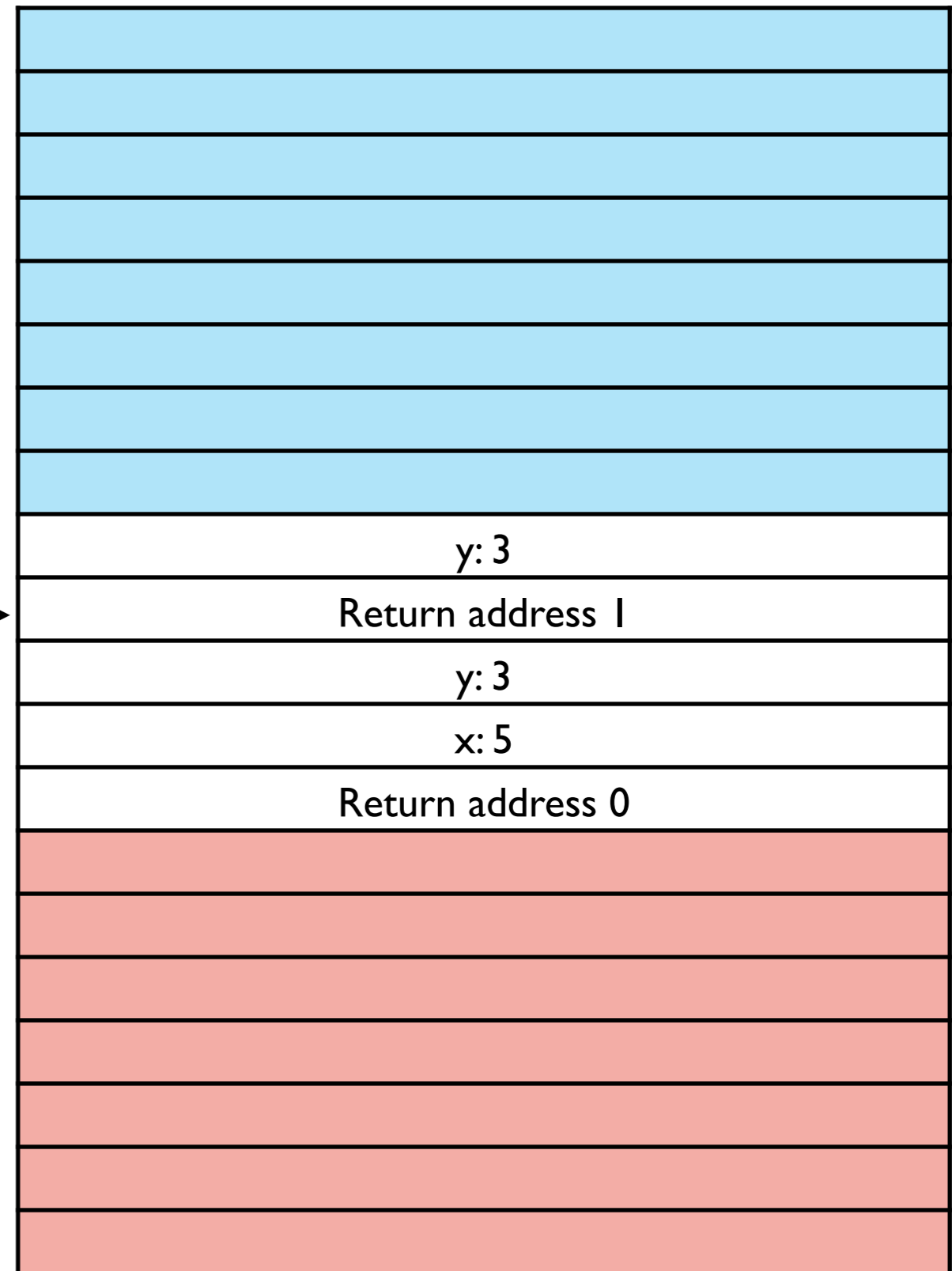


y is in the wrong spot!!

Stack

```
def multiply(x, y):  
    def loop(y):  
        → if y == 0:  
            0  
        else:  
            x + loop(y - 1)  
    end  
    loop(y)  
end  
multiply(5, 3)
```

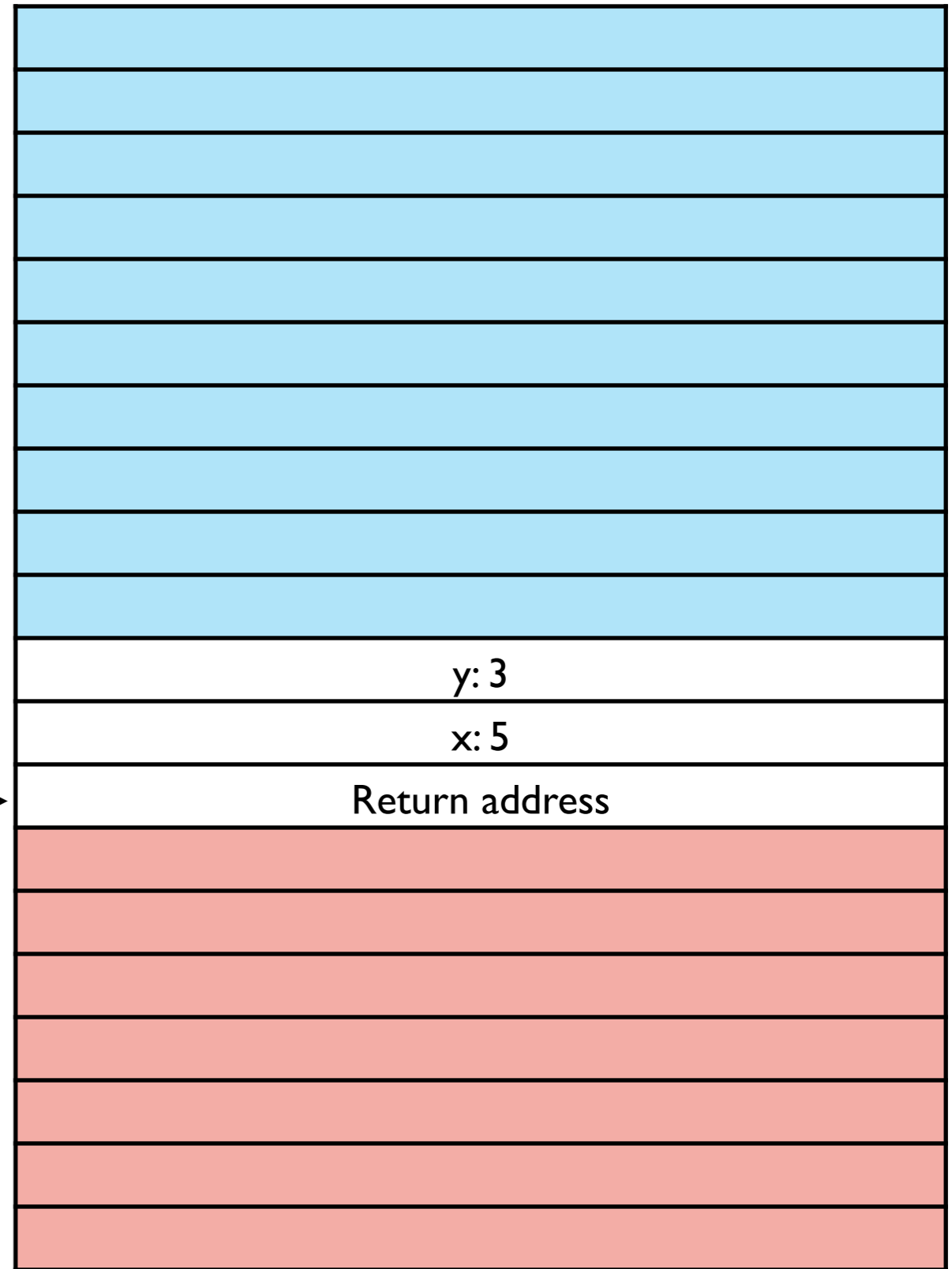
rsp →



Stack

```
def multiply(x, y):  
    def loop(y):  
        → if y == 0:  
            0  
        else:  
            x + loop(y - 1)  
        end  
        loop(y)  
    end  
    multiply(5, 3)
```

rsp →



"Lambda Lifting"

```
def multiply(x, y):  
  def loop(y):  
    if y == 0:  
      0  
    else:  
      x + loop(y - 1)  
  end  
  loop(y)  
end  
multiply(5, 3)
```

```
def multiply(x, y):  
  loop(x, y)  
and  
def loop(x, y):  
  if y == 0:  
    0  
  else:  
    x + loop(x, y - 1)  
end  
multiply(5, 3)
```

A Problem?

```
def multiply(x, y):  
    def loop(y):  
        if y == 0:  
            0  
        else:  
            x + loop(y - 1)  
        end  
    end  
end  
multiply(5, 3)
```

Summary

- When calling Rust code, use the Sys V
- When calling Snake functions, use the Snake calling convention
 - Tail Call: overwrite our stack frame with arguments
 - Non Tail Call: push return address/args above our stack frame
- Returning:
 - works the same way for both: same "returning convention"
- Alignment
- Lambda Lifting