EECS 483: Compiler Construction

Lecture 8:
Non-tail Function Calls, Calling Conventions

February 10
Winter Semester 2025

Announcements

- Assignment 2 due Friday.
- Assignment 3 released next Monday, extends Assignment 2 with full support for functions (non-tail calls)

State of the Snake Language

Adder: Straightline Code

Boa: Conditionals + Tail-called functions = arbitrary "local" control flow.

Good for implementing the "bodies" of functions, but missing key features:

1. Interaction with the Operating System (stdin/stdout, file I/O, random number generation)

2. Reusable sub-procedures (functions with non-tail calls)

Add these in **Cobra**

Extending the Snake Language

When we implement a compiler (to assembly) we need to address the following questions:

1. What is the syntax of the language we are compiling?
2. What is the semantics of the language we are compiling?
3. How can we implement that semantics in assembly code?
4. How should we adapt our intermediate representation to new features?
5. How can we generate assembly code from the IR?

Extending the Snake Language

Want the ability to interact with the operating system

So far: add new primitives one at a time

More flexible: allow importing "extern" functions from our Rust stub.rs file

Extending the Snake Language

```
extern read()
extern print(x)

def main(x):
  def loop(sum):
    let _ = print(sum) in
    loop(sum + read())
  in
  loop(0)
```

Implement read, print in stub.rs

Concrete Syntax

‹prog›:

| ‹externs› `def` `main` `(` ***IDENTIFIER*** `)` `:` ‹expr›

| `def` `main` `(` ***IDENTIFIER*** `)` `:` ‹expr›

‹extern›:

| `extern` ***IDENTIFIER*** `(` `)`

| `extern` ***IDENTIFIER*** `(` ‹ids› `)`

‹externs›:

| `extern`

| `extern` ‹externs›

‹expr›: ...

‹ids›: ***IDENTIFIER*** | ***IDENTIFIER*** `,` ‹ids›

Abstract Syntax

```rust
pub struct Prog {
    pub externs: Vec<ExtDecl>,
    pub param: Var,
    pub main: Expr,
}
```

```rust
pub struct ExtDecl {
    pub name: FunName,
    pub params: Vec<Var>,
}
```

Well-formedness for Extern

1. Extern functions should be in the same scope as local function definitions. Local function definitions can shadow external function declarations

2. Can't have multiple external functions with the same name

2. In the name resolution phase, do **not** change the names of external functions, as the name is used for linking

SSA Changes

Add extern declarations to top-level SSA program

Add call as a new operation in SSA (not a terminator!)

```
x = call f(x1,...)
```

Change to lowering:

if a call to an **internal** function is a tail call, compile it as before as a branch with arguments

if a call to an **external** function is a tail call, compile it as a call and then return the result

SSA Changes

```
extern print(x)

def main(x):
  let y = x + 10
  print(y)
```

```
extern print(x)

main(x):
  y = x + 10
  res = call print(y)
  ret res
```

Code Generation

Each extern declaration becomes an x86 extern declaration

Function calls are compiled using the System V AMD64 Calling convention

Linking will fail unless the extern functions are implemented in stub.rs

SSA Changes

```
extern print(x)


main(x):

  y = x + 10

  res = call print(y)

  ret res
```

```
    section .text
    global entry
    extern print

entry:
  ...
```

Non-tail Procedure Calls

When a function is called it needs to know **where** to return to, i.e., what its **continuation** is.

- To implement this, procedure calls pass a **return address.** Think of this as an "extra argument" that is implicit in the original program.

When a function is called, it needs to know which registers and which regions of memory it is free to use

- Use **rsp** as a pointer to denote which part of the stack is free space
- Designate which registers are **volatile** or **non-volatile**

When implementing calls within a programming language we can decide these conventions for ourselves. When implementing calls that work with external code need to pick a standard **calling convention**

x86 Abstract Machine: The Stack

So far we have used rsp as a base pointer into our stack frame

But several instructions treat it as a "stack pointer"

x86 Instructions: push

```
push arg
```

Semantics:

```
sub rsp, 8
mov [rsp], arg
```

If rsp is the pointer to the "top" of the stack, this pushes a new value on top.

x86 Instructions: pop

```
pop reg
```

Semantics:

```
mov reg, [rsp]
add rsp, 8
```

If rsp is the pointer to the "top" of the stack, this pops the current value off of it

x86 Instructions: call

call loc

Semantics is a combination of **jmp** and **push**

sets rip to loc (like jmp loc)

and pushes the address of the next instruction onto the stack (like jmp next)

Example:

x86 Instructions: ret

ret

Semantics is a combination of **pop** and **jmp**

pops the return address off of the stack and jumps to it

like

```
pop r
jmp r
```

except without using up a register r

Calling Convention

Calling Convention

When implementing a call into Rust, we need to use a common **calling convention**

We use the System V AMD 64 ABI

- Standard "C" calling convention for 64-bit x86 code on Linux/Intel Macs
- Has some idiosyncracies from supporting C code and SSE instructions

Calling Convention

A calling convention is a protocol that a caller and callee follow in order to implement a procedure call and return

Caller and callee need to agree on:

1. State of memory/registers when the callee begins executing

2. State of memory/registers once the callee has returned

So a calling convention is really a combination of a "calling" convention and a "returning" convention

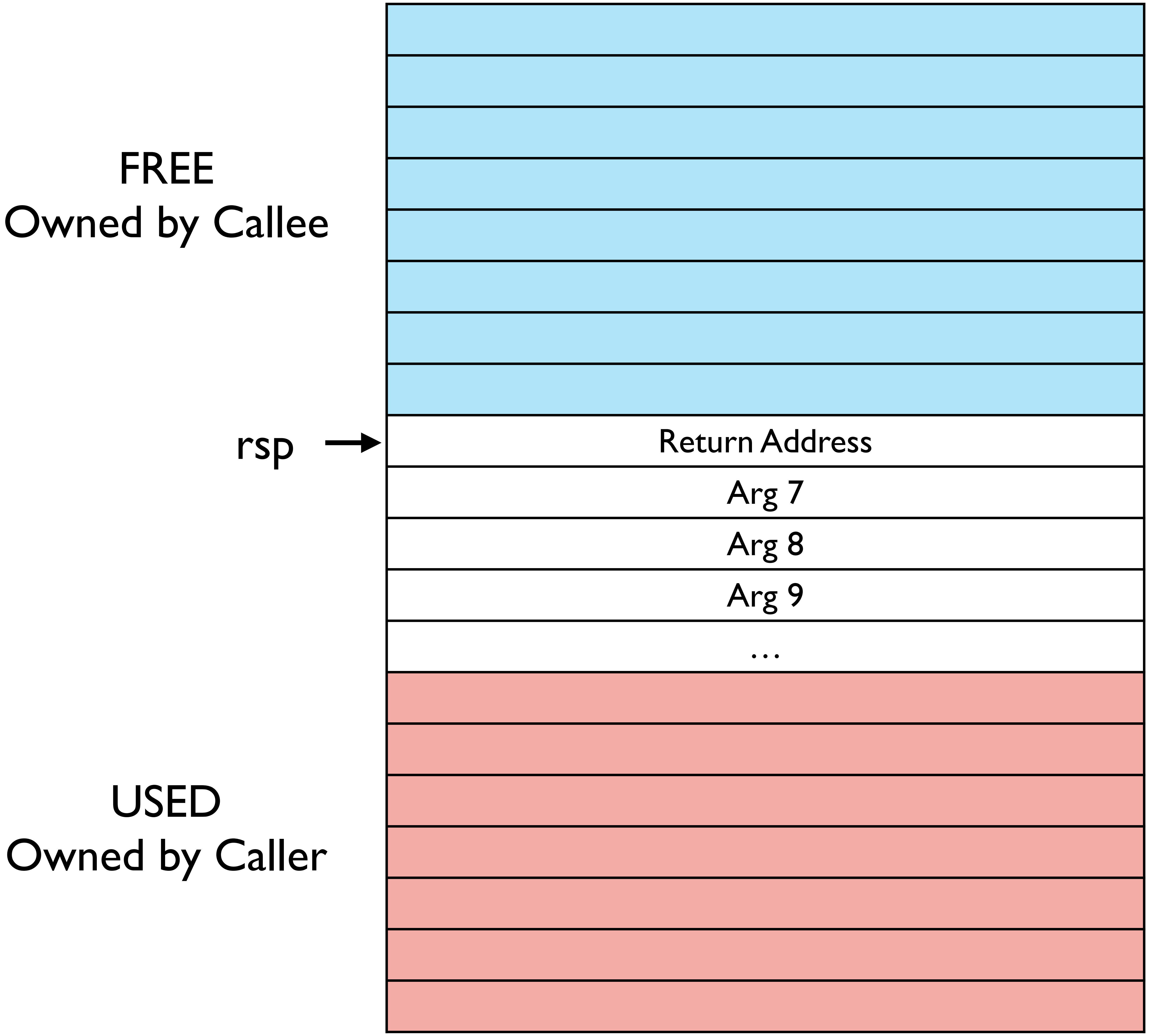
System V AMD 64

Calling protocol: When a called function starts executing the machine state is as follows:

1. Arguments 1-6 are stored in rdi, rsi, rdx, rcx, r8, r9
2. Arguments 7-N are stored in [rsp + 1 * 8], [rsp + 2 * 8],...[rsp + (N - 6) * 8]
3. rsp points to the return address.
4. Stack Alignment: rsp % 16 == 8

System V AMD 64

rdi	Arg 1
rsi	Arg 2
rdx	Arg 3
rcx	Arg 4
r8	Arg 5
r9	Arg 6
rsp	0xXX…X8



System V AMD 64

Returning protocol: When a called function returns to its caller

1. Return value is stored in rax
2. Registers rbx, rbp, r12-r15 are in their original state when the function was called **(non-volatile aka callee-save)**
3. Stack memory at higher addresses than rsp is in the original state when the function was called
4. Original value of rsp holds the return address, pop this address and jump to it

Volatile/Non-volatile Registers

A register is **volatile** if its value may be changed by a function call

- This means the **callee** can set the register as they see fit, no promises on what its value will be when the callee returns

- Also known as **caller-save** because if a local variable is stored in a volatile register it must be "saved" somewhere non-volatile if its value is needed after the call

A register is **non-volatile** it it must be preserved by a function call

- This means the **callee** must ensure that when the function returns, the register has the same value as it did when the function began execution

- Also known as **callee-save** because if the callee wants to use a non-volatile register, the original value must be saved somewhere and restored before returning

Volatile/Non-volatile Registers

Current Strategy:

We use registers as scratch registers, but always store local variable values on the stack

Should a scratch register be **volatile** or **non-volatile**?

volatile: we are free to change it without worrying about the caller

don't need to worry about saving it when we make a call because we don't store long-lived values in it

examples: rax, r10, r11, any argument register if we move its value to the stack

Revisit this once we implement **register allocation**

Stack Alignment

When a function is called, rsp % 16 == 8

Needed for certain SSE instructions which require data alignment

To ensure correct alignment: rsp % 16 == 0 **before** executing the **call** instruction (since call pushes an 8-byte address)

Caller cleanup

In the SysV AMD 64 calling convention, the **caller** is responsible for "cleanup" of the arguments.

That is, when a function returns, the arguments that are passed on the stack are still there, even though they are not necessarily needed

Why?

Used to implement C-style variadic function. In C, a variadic function doesn't know how many arguments have been passed, so impossible for it to perform caller cleanup.

Downside:

Impossible to perform tail call to a function that takes more stack-allocated arguments than the caller using SysV AMD 64 calling convention.

Stack Frame Management

When making a function call we need to ensure that the newly allocated stack frame is "above" all of our local variables

Stack Frame Management

The calling convention dictates an **interface** between the caller and the callee.

It does not dictate **internal** details of a function implementation, e.g., where in registers, memory, local variables are stored

2 common strategies for managing stack-allocated variables

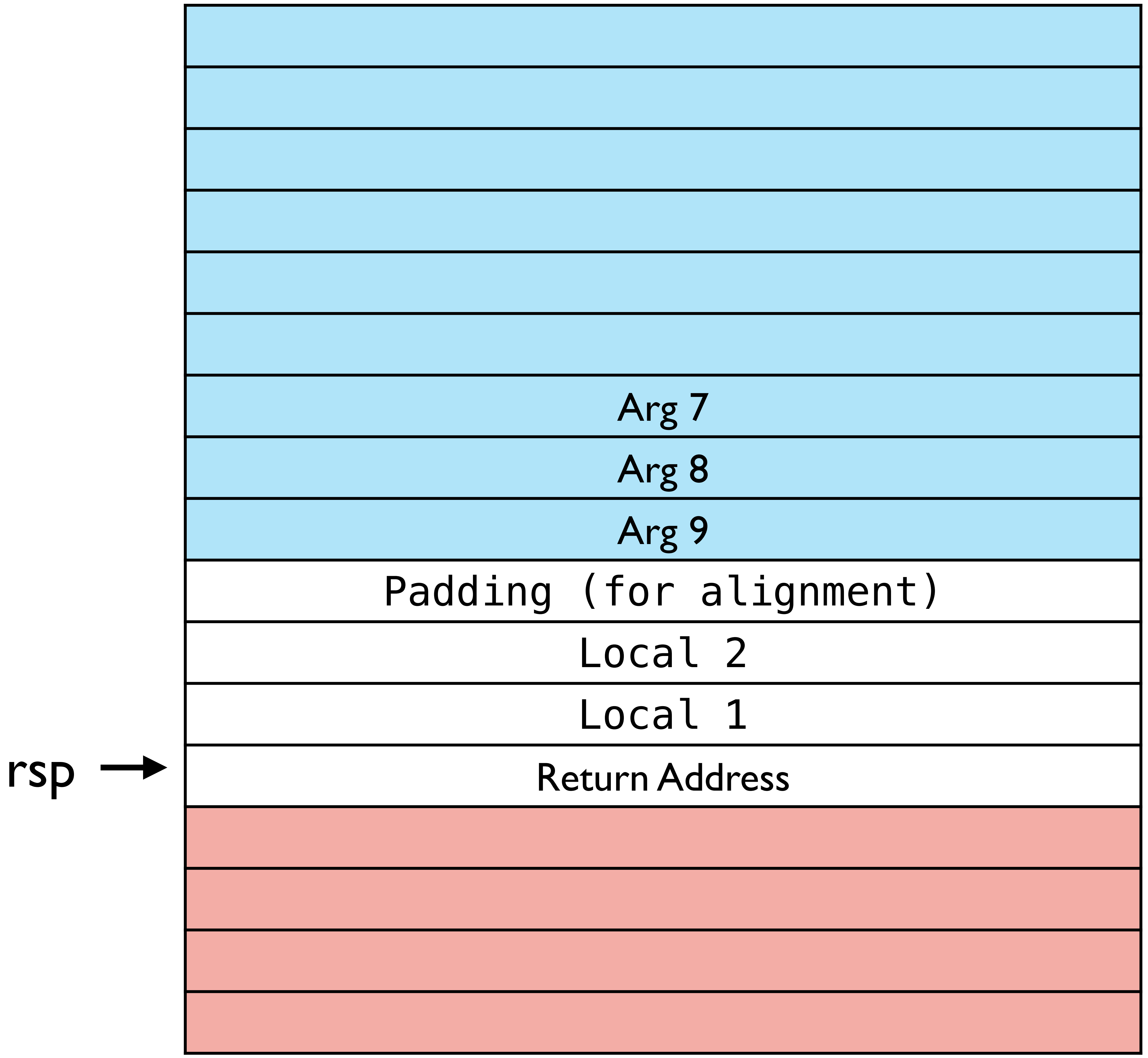
1. (Modern) Use rsp as the base pointer of the stack frame
2. (C style) Use rbp as the base pointer of the stack frame and rsp as the pointer to the **top** of the stack frame

Sys V AMD64 Calls

Live code examples: summer, big_fun

Sys V AMD64 Call

```
mov rdi, arg1
mov rsi, arg2
mov rdx, arg3
mov rcx, arg4
mov r8, arg5
mov r9, arg6
mov QWORD [rsp - 32], arg7
mov QWORD [rsp - 40], arg8
mov QWORD [rsp - 48], arg9
sub rsp, 48
call big_fun
add rsp, 48
```



Sys V AMD64 Call

```
mov rdi, arg1
mov rsi, arg2
mov rdx, arg3
mov rcx, arg4
mov r8, arg5
mov r9, arg6
mov QWORD [rsp - 32], arg7
mov QWORD [rsp - 40], arg8
mov QWORD [rsp - 48], arg9
sub rsp, 48
call big_fun
add rsp, 48
```