cs5460/6460 Operating Systems
Lecture 4: Function invocations, and calling conventions

Anton Burtsev

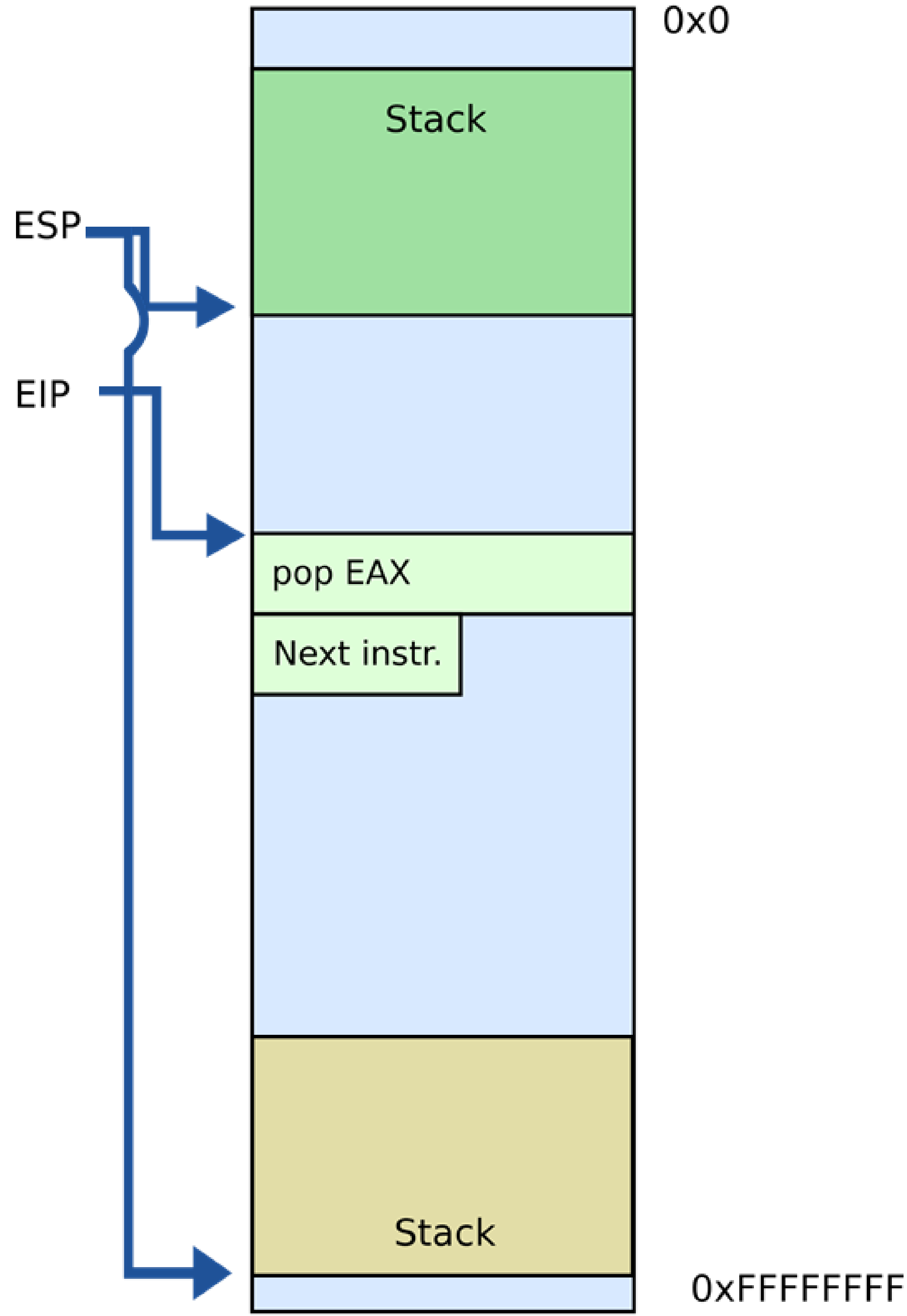
January, 2026

Stack and procedure calls

What is stack?

Stack

- It's just a region of memory
    - Pointed by a special register RSP
- You can change RSP
    - Get a new stack



Why do we need stack?

Stack allows us to invoke functions

Calling functions

```
// some code...
foo();
// more code..
```

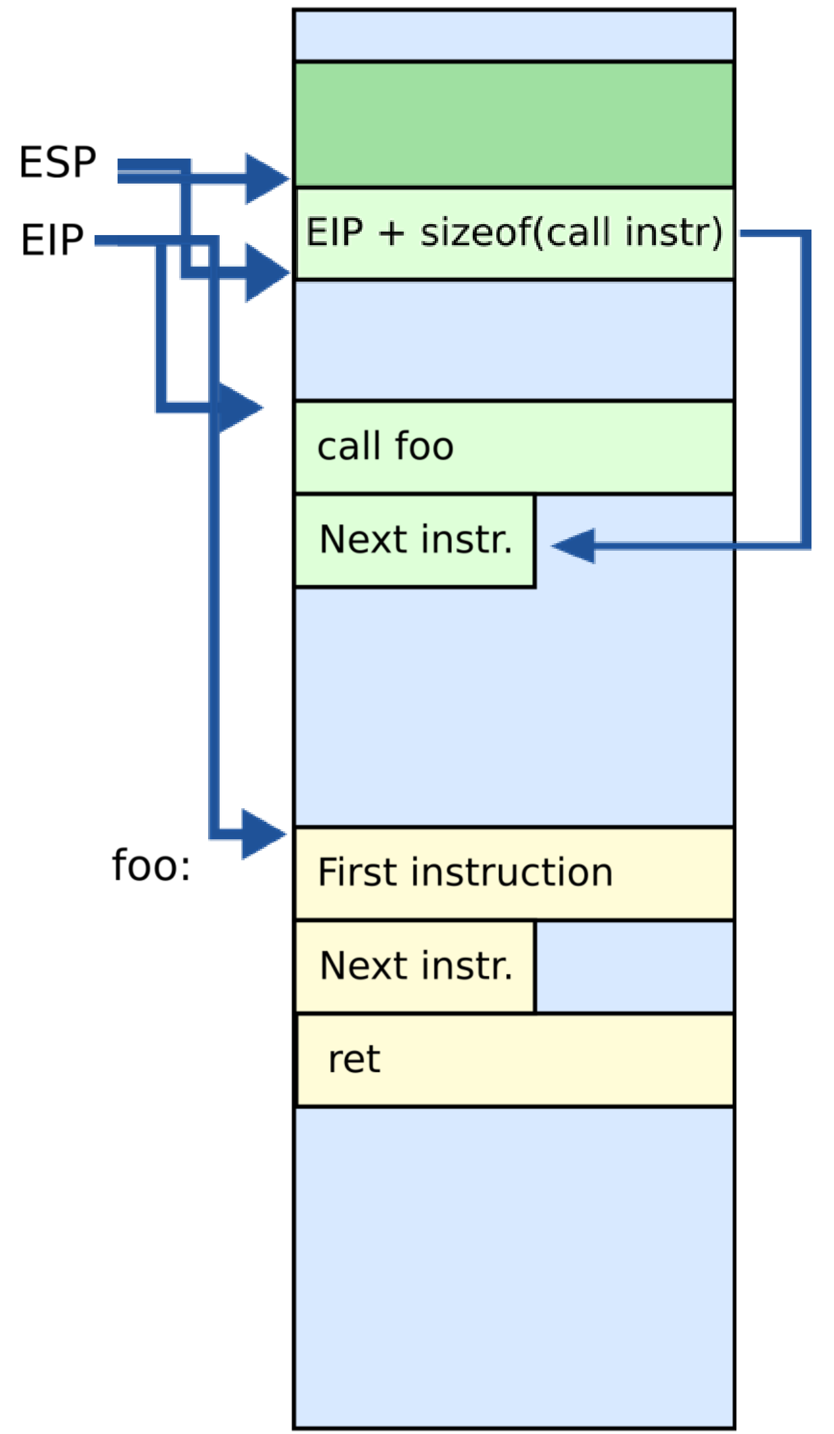
- Stack contains information for how to return from a subroutine
- i.e., from foo()

- Functions can be called from different places in the program

```
 if (a == 0) {
 foo();
 ...
} else {
 foo();
 ...
}
```

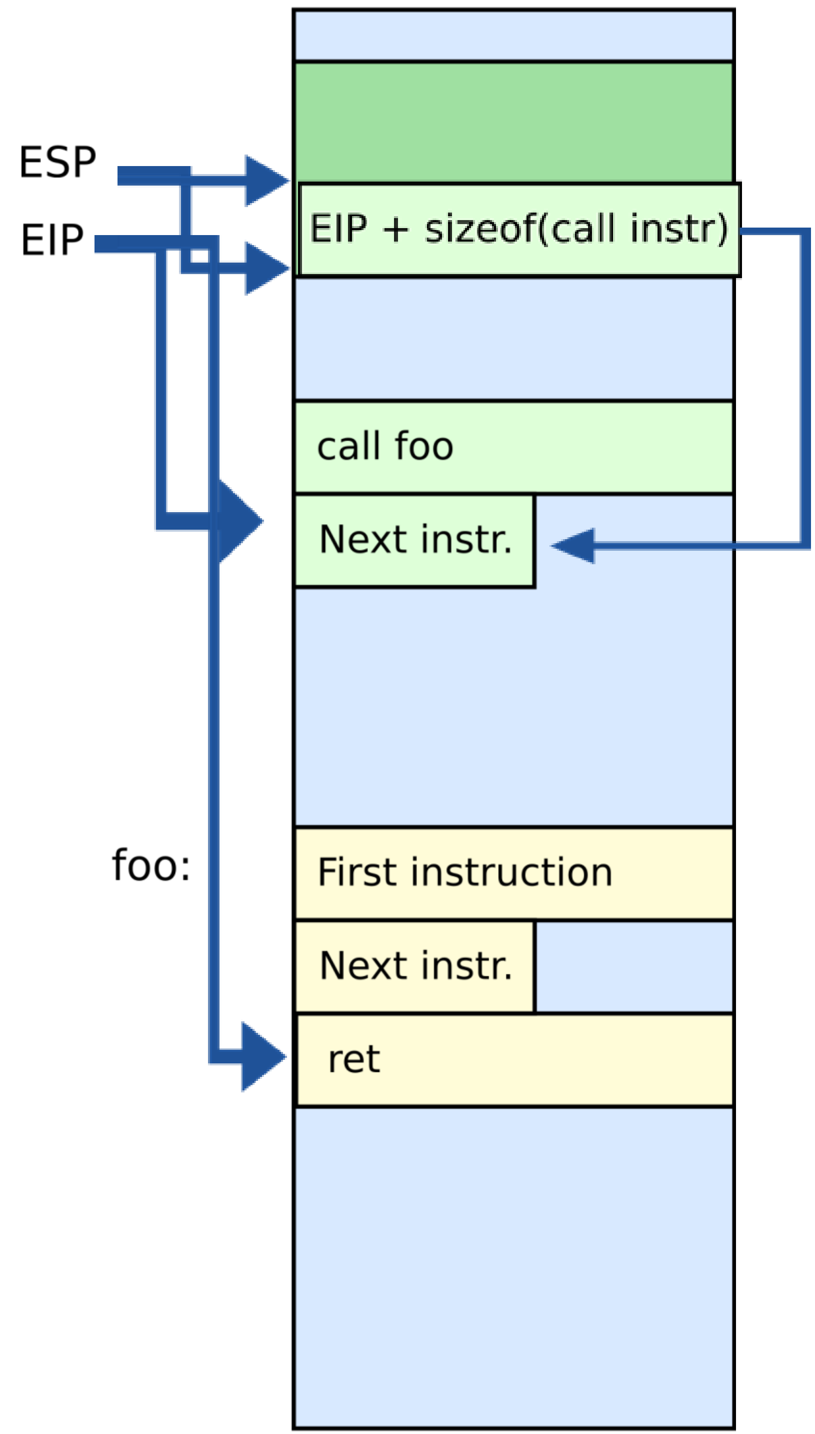
Stack

- Main purpose:
  - Store the return address for the current procedure
- Caller pushes return address on the stack
- Callee pops it and jumps



Stack

- Main purpose:
  - Store the return address for the current procedure
- Caller pushes return address on the stack
- Callee pops it and jumps



Example

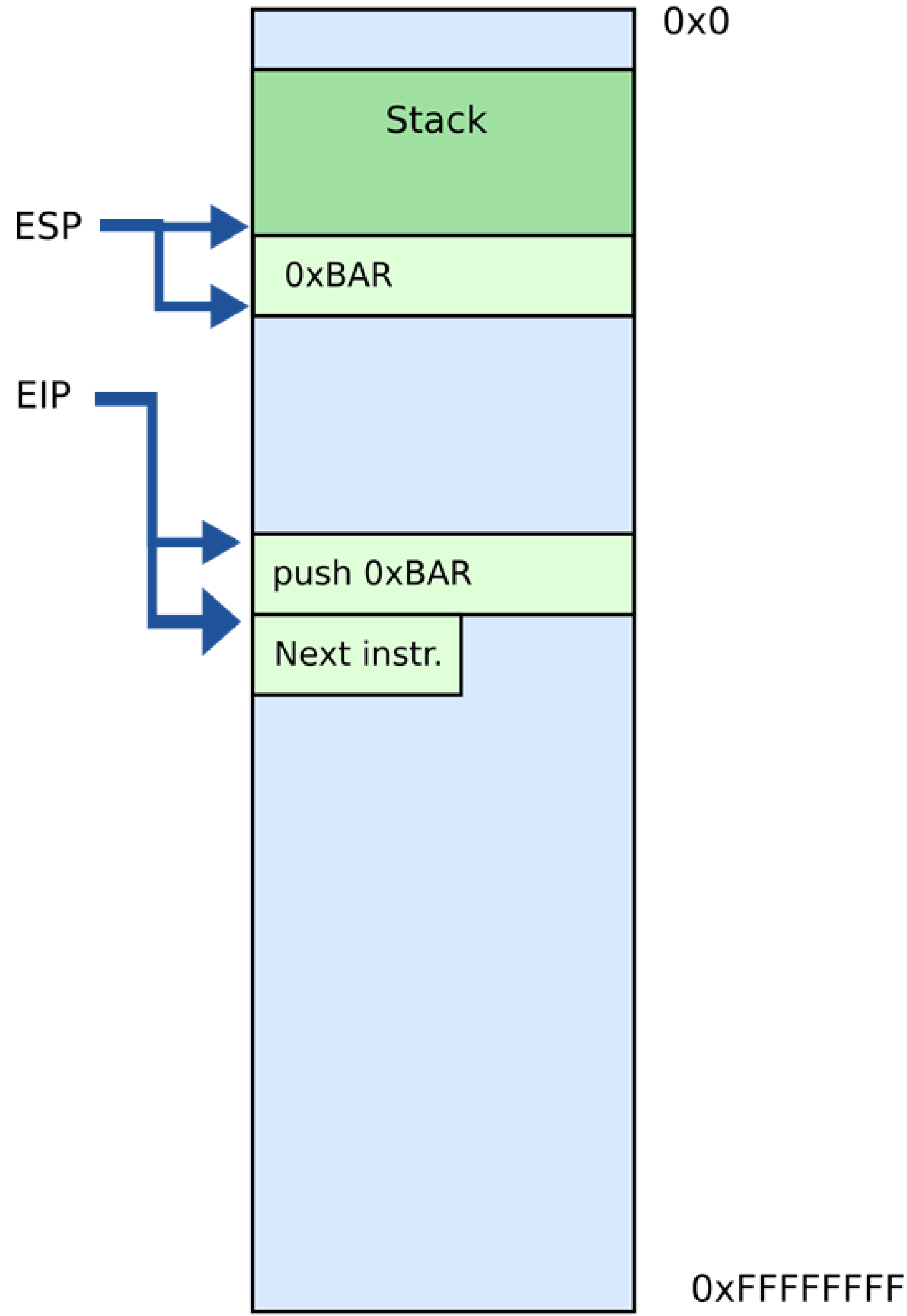
```
foo(int a) {
    if (a == 0)
        return;
    a--;
    foo(a);
    return;
}

foo(4);
```

Manipulating stack

- RSP register
- Contains the memory address of the topmost element in the stack
- PUSH instruction

push 0xBAR

- Subtract 8 from RSP
- Insert data on the stack

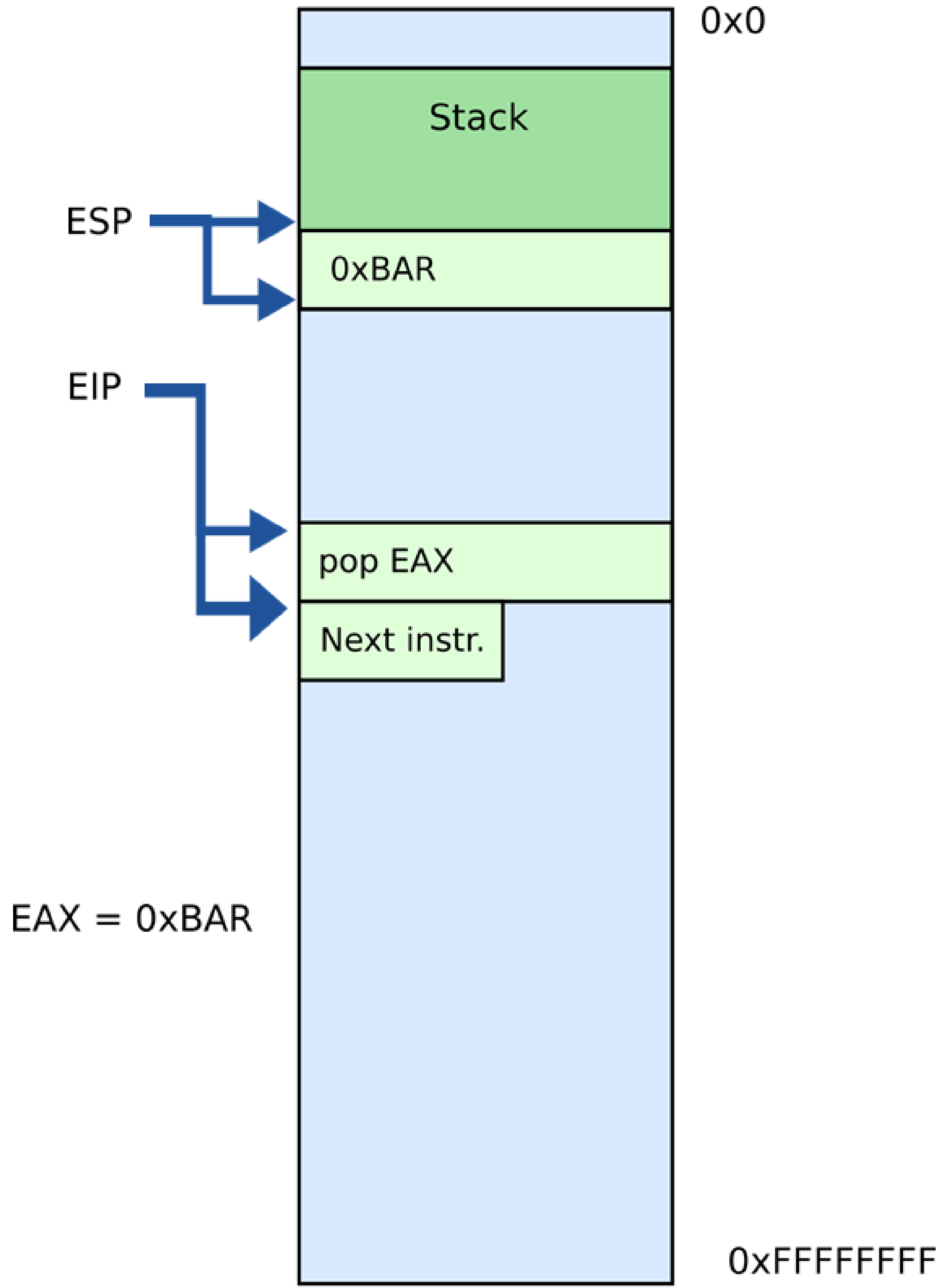


Manipulating stack

- POP instruction

  pop RAX

- Removes data from the stack
- Saves in register or memory
- Adds 8 to RSP



Calling conventions

Calling conventions

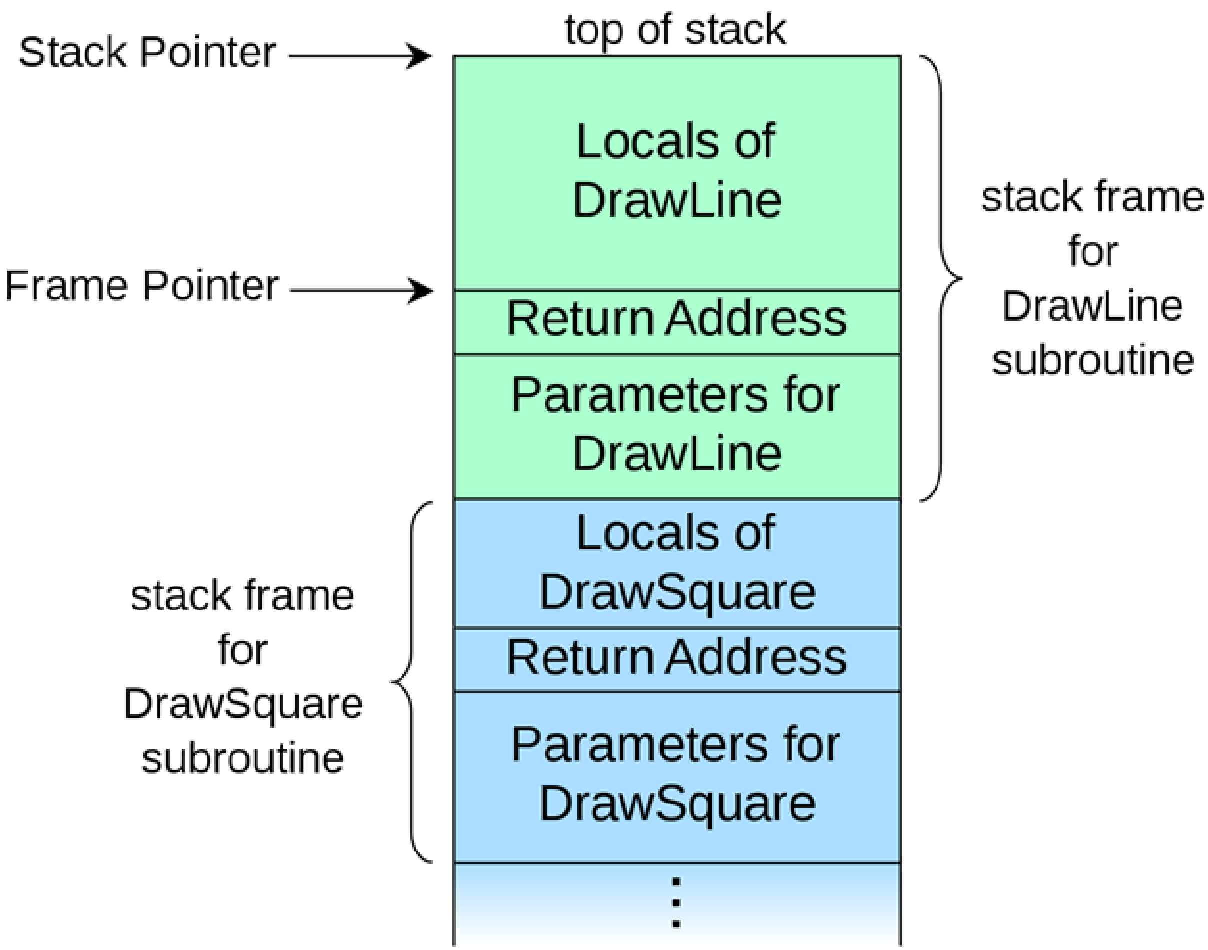
- Goal: re-entrant programs
- How to pass arguments
  - On the stack?
  - In registers?
- How to return values
  - On the stack?
  - In registers?
- How to save and restore registers across invocations
- Conventions vary between compilers, optimizations, etc.

Idea 1: Maintain stack as frames

- Each function has a new frame

```
void DrawSquare(...)
{
   ...
   DrawLine(x, y, z);
}
```

- Use dedicated register RBP (frame pointer)
- Points to the base of the frame

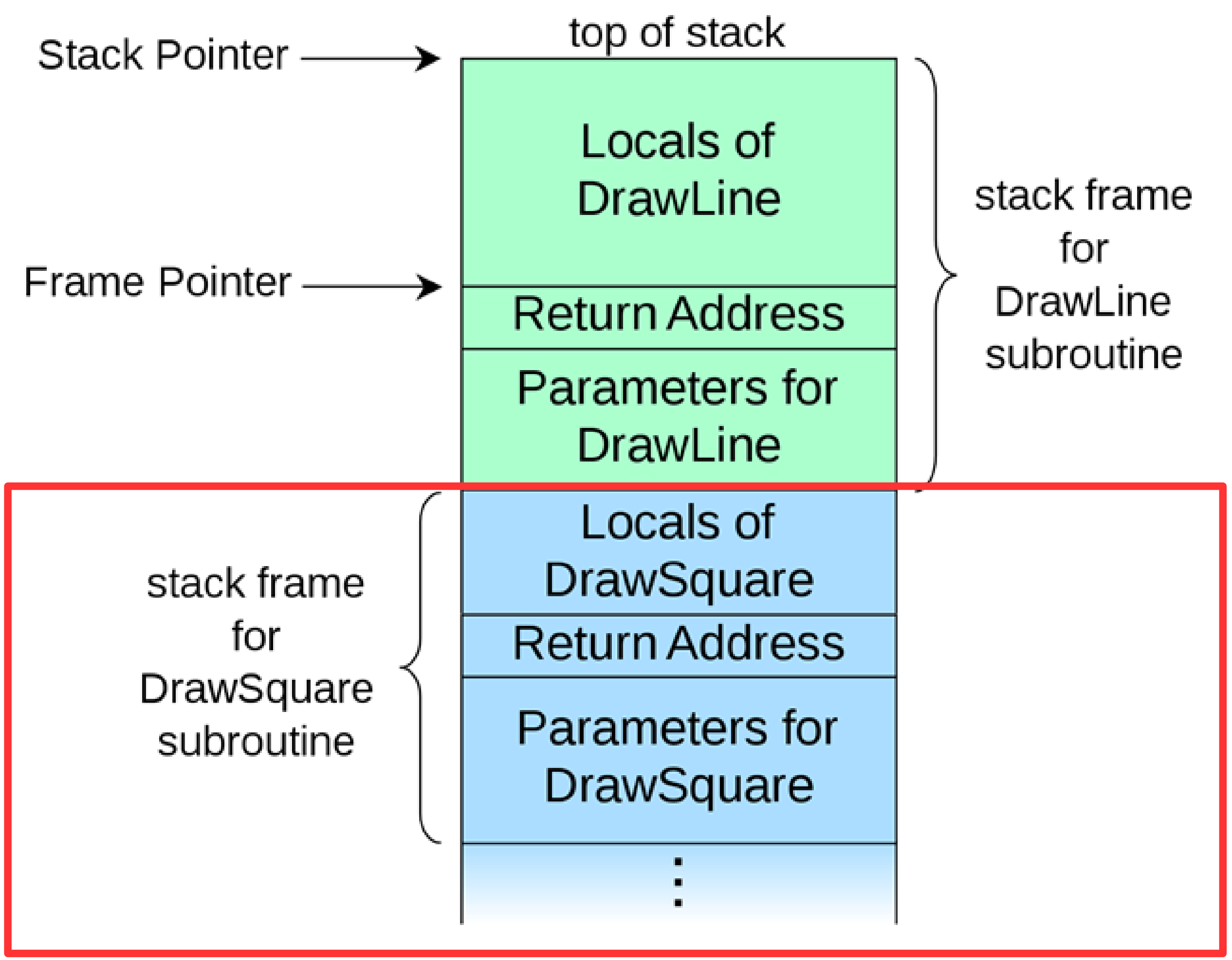


Idea 1: Maintain stack as frames

- Each function has a new frame

```
void DrawSquare(...)
{
   ...
   DrawLine(x, y, z);
}
```

- Use dedicated register RBP (frame pointer)
- Points to the base of the frame

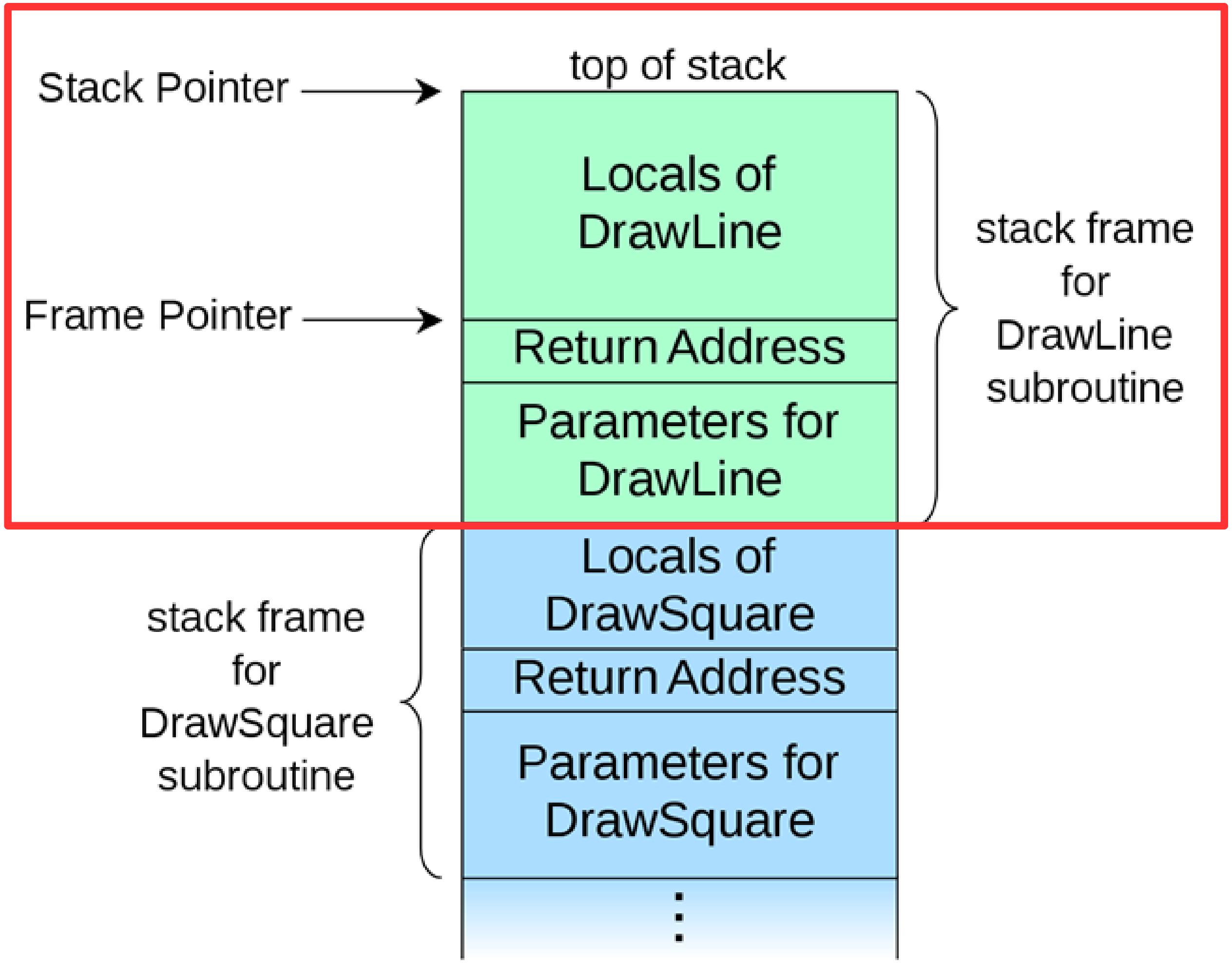


Idea 1: Maintain stack as frames

- Each function has a new frame

```
void DrawSquare(...)
{
   ...
   DrawLine(x, y, z);
}
```

- Use dedicated register RBP (frame pointer)
- Points to the base of the frame



Prologue/epilogue

- Each function maintains the frame
- A dedicated register RBP is used to keep the frame pointer
- Each function uses prologue code (blue), and epilogue (green) to maintain the frame

```
my_function:
    push rbp        ; save original RBP value on stack
    mov rbp, rsp    ; new RBP = RSP
    …               ; function body
    pop rbp         ; restore original RBP value
    ret
```

Local variables

What types of variables do you know?

- Or where these variables are allocated in memory?

What types of variables do you know?

- Global variables
  - Initialized → data section
  - Uninitalized → BSS
- Dynamic variables
  - Heap
- Local variables
  - Stack

Global variables

```c
#include <stdio.h>

char hello[] = "Hello";
int main(int ac, char **av)
{
    static char world[] = "world!";
    printf("%s %s\n", hello, world);
    return 0;
}
```

Global variables

```c
#include <stdio.h>

char hello[] = "Hello";
int main(int ac, char **av)
{
    static char world[] = "world!";
    printf("%s %s\n", hello, world);
    return 0;
}
```

- Allocated in the data section
- It is split into
  - Initialized (non-zero), and uninitialized (zero)
  - Read/write, and read-only data sections

Global variables

Dynamic variables (heap)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char hello[] = "Hello";
int main(int ac, char **av)
{
    char world[] = "world!";
    char *str = malloc(64);
    memcpy(str, "beautiful", 64);
    printf("%s %s %s\n", hello, str, world);
    return 0;
}
```

Dynamic variables (heap)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char hello[] = "Hello";
int main(int ac, char **av)
{
    char world[] = "world!";
    char *str = malloc(64);
    memcpy(str, "beautiful", 64);
    printf("%s %s %s\n", hello, str, world);
    return 0;
}
```

- Allocated on the heap
- Special area of memory provided by the OS from where malloc() can allocate memory

Dynamic variables (heap)

Local variables

- Local variables

```
#include <stdio.h>

char hello[] = "Hello";
int main(int ac, char **av)
{
    //static char world[] = "world!";
    char world[] = "world!";
    printf("%s %s\n", hello, world);
    return 0;
}
```

Local variables...

- Each function has private instances of local variables

```
foo(int x) {
  int a, b, c;
  ...
  return;
}
```

- Function can be called recursively

```
foo(int x) {
  int a, b, c;
  a = x + 1;
  if ( a < 100 )
    foo(a);
    return;
}
```

How to allocate local variables?

```
foo(int x) {
  int a, b, c;
  ...
}
```

How to allocate local variables?

```
foo(int x) {
  int a, b, c;
  ...
}
```

- On the stack!

Poll Q1: Where do we allocate global variables

https://pollev.com/cs5460

Poll Q2: Where do we allocate dynamic variables

https://pollev.com/cs5460

Allocating local variables

- Stored right after the saved RBP value on the stack
- Allocated by subtracting the number of bytes required from RSP

```
_my_function:
    push rbp            ; save original RBP value on stack
    mov rbp, rsp        ; new RBP = RSP
    sub rsp, LOCAL_BYTES ; locals (round up to keep 16B
                        ; alignment)
    …                   ; function body
    mov rsp, rbp        ; deallocate locals
    pop rbp             ; restore original RBP value
    ret
```

Example

```
void my_function() {
   int a, b, c;
   …
```

```
_my_function:
  push rbp      ; save the value of rbp
  mov rbp, rsp ; rbp = rsp, set rbp to be top of the stack (rsp)
  sub rsp, 16  ; move rsp down to allocate space for the
               ; local variables on the stack
```

- With frames local variables can be accessed by de-referencing RBP

```
mov [rbp -  4], 10  ; location of variable a
mov [rbp -  8], 5   ; location of b
mov [rbp - 12], 2   ; location of c
```

Example

```
void my_function() {
    int a, b, c;
    …

_my_function:
   push rbp     ; save the value of rbp
   mov rbp, rsp ; rbp = rsp, set rbp to be top of the stack (rsp)
   sub rsp, 16  ; move rsp down to allocate space for the
                ; local variables on the stack
```

- With frames local variables can be accessed by de-referencing RBP

```
mov [rbp -  4], 10  ; location of variable a
mov [rbp -  8], 5   ; location of b
mov [rbp - 12], 2   ; location of c
```

How to pass arguments?

- Possible options:
- In registers
- On the stack

How to pass arguments?

- Linux x86-64 user-space uses the System V AMD64 ABI
    - Integer/pointer args #1–#6: RDI, RSI, RDX, RCX, R8, R9
    - Floating-point args #1–#8: XMM0–XMM7
- Remaining args: on the stack (8-byte slots)
    - Return values: RAX (and RDX if needed); FP return: XMM0
    - Stack alignment: RSP must be 16-byte aligned before call
    - Red zone: 128 bytes below RSP may be used by leaf functions (user-space)
    - Note: Linux syscalls use a different convention (RAX= syscall number, args RDI/RSI/RDX/R10/R8/R9)

x86-64 (System V): passing arguments in registers

- Example function

```
void my_function(long x, long y, long z)
{ … }
```

- Example invocation

```
my_function(2, 5, 10);
```

- Generated code

```
mov rdi, 2
mov rsi, 5
mov rdx, 10
call my_function
```

Example stack

```
:    :
| arg8 | [RBP + 24] (8th argument, stack-passed)
| arg7 | [RBP + 16] (7th argument, stack-passed)
| RA   | [RBP + 8]  (return address)
| FP   | [RBP]      (old RBP value) ← RBP points here
| ...  | [RBP - 8]  (1st local variable)
:    :
:    :
| ...  | [RBP - X]  (RSP - the current stack pointer)
```

args 1–6 are in registers:
RDI, RSI, RDX, RCX, R8, R9

Example: caller side code

```c
long callee(long, long, long);

long caller(void)
{
    long ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

```asm
caller:
  ; manage own stack frame
  push    rbp
  mov     rbp, rsp
  ; set call arguments (registers)
  mov     rdi, 1
  mov     rsi, 2
  mov     rdx, 3
  ; call subroutine 'callee'
  call    callee
  ; no stack cleanup needed
  ; (args in regs)
  ; stack is 16B-aligned before call
  ; use subroutine result
  add     rax, 5
  ; restore old call frame
  pop     rbp
  ; return
  ret
```

Example: caller side code

```c
long callee(long, long, long);

long caller(void)
{
   long ret;

   ret = callee(1, 2, 3);
   ret += 5;
   return ret;
}
```

```
caller:
  ; manage own stack frame
  push    rbp
  mov     rbp, rsp
  ; set call arguments (registers)
  mov     rdi, 1
  mov     rsi, 2
  mov     rdx, 3
  ; call subroutine 'callee'
  call    callee
  ; no stack cleanup needed
  ; (args in regs)
  ; stack is 16B-aligned before call
  ; use subroutine result
  add     rax, 5
  ; restore old call frame
  pop     rbp
  ; return
  ret
```

Example: caller side code

```c
long callee(long, long, long);

long caller(void)
{
   long ret;

   ret = callee(1, 2, 3);
   ret += 5;
   return ret;
}
```

```
caller:
  ; manage own stack frame
  push    rbp
  mov     rbp, rsp
  ; set call arguments (registers)
  mov     rdi, 1
  mov     rsi, 2
  mov     rdx, 3
  ; call subroutine 'callee'
  call    callee
  ; no stack cleanup needed
  ; (args in regs)
  ; stack is 16B-aligned before call
  ; use subroutine result
  add     rax, 5
  ; restore old call frame
  pop     rbp
  ; return
  ret
```

Example: caller side code

```c
long callee(long, long, long);

long caller(void)
{
    long ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

```asm
caller:
  ; manage own stack frame
  push    rbp
  mov     rbp, rsp
  ; set call arguments (registers)
  mov     rdi, 1
  mov     rsi, 2
  mov     rdx, 3
  ; call subroutine 'callee'
  call    callee
  ; no stack cleanup needed
  ; (args in regs)
  ; stack is 16B-aligned before call
  ; use subroutine result
  add     rax, 5
  ; restore old call frame
  pop     rbp
  ; return
  ret
```

Example: caller side code

```c
long callee(long, long, long);

long caller(void)
{
   long ret;

   ret = callee(1, 2, 3);
   ret += 5;
   return ret;
}
```

```asm
caller:
  ; manage own stack frame
  push    rbp
  mov     rbp, rsp
  ; set call arguments (registers)
  mov     rdi, 1
  mov     rsi, 2
  mov     rdx, 3
  ; call subroutine 'callee'
  call    callee
  ; no stack cleanup needed
  ; (args in regs)
  ; stack is 16B-aligned before call
  ; use subroutine result
  add     rax, 5
  ; restore old call frame
  pop     rbp
  ; return
  ret
```

Example: caller side code

```c
long callee(long, long, long);

long caller(void)
{
    long ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

```asm
caller:
  ; manage own stack frame
  push    rbp
  mov     rbp, rsp
  ; set call arguments (registers)
  mov     rdi, 1
  mov     rsi, 2
  mov     rdx, 3
  ; call subroutine 'callee'
  call    callee
  ; no stack cleanup needed
  ; (args in regs)
  ; stack is 16B-aligned before call
  ; use subroutine result
  add     rax, 5
  ; restore old call frame
  pop     rbp
  ; return
  ret
```

Example: callee side code

```
void my_function(long x, long y, long z)
{
   long a, b, c;
   …
   return;
}
```

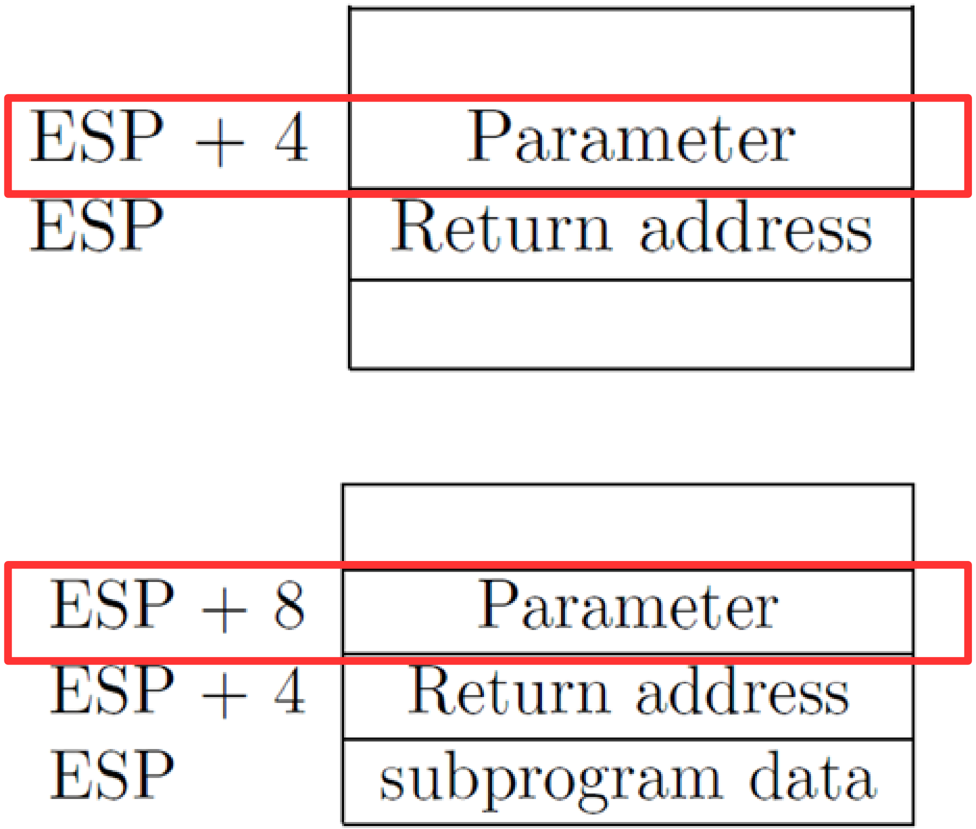
```
my_function:
  push rbp
  mov rbp, rsp
  sub rsp, 32 ; allocate locals + padding (keep 16B alignment)
              ; x in RDI, y in RSI, z in RDX   (System V AMD64)
              ; a=[rbp-8], b=[rbp-16], c=[rbp-24]


  mov rsp, rbp ; deallocate local variables
  pop rbp
  ret
```

Back to stack frames, so why do we need them?

- … They are not strictly required
- GCC compiler option -fomit-frame-pointer can disable them

Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. **It also makes debugging impossible on some machines.**

Referencing args without frames



- In x86-64 SysV, most arguments arrive in registers
- Stack-passed args (arg7+) start at [RSP + 8] on entry (above return address)
- As you push/pop or allocate locals, RSP changes → offsets change too

- Debugging becomes hard
  - As RSP changes one has to manually keep track where local variables are relative to RSP (RSP + 4 or +8)
- **Compiler can easily do this and generate correct code!**
- **But it's hard for a human**
  - It's hard to unwind the stack in case of a crash
  - To print out a backtrace

And you only save...

- A stack frame typically costs a couple instructions + one register (RBP as frame pointer)
  - x86-64 has 16 general-purpose registers, so keeping RBP as a frame pointer is usually affordable
  - Optimizing compilers often omit the frame pointer by default (-fomit-frame-pointer)
  - GCC/Clang at -O1/-O2/... typically enable that option
- For easier backtraces/profiling: compile with -fno-omit-frame-pointer

Relevant part of the GCC manual

3.10 Options That Control Optimization
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Poll

- In the calling convention we just discussed how do we pass the first argument to the funciton?

https://pollev.com/cs5460

Saving and restoring registers

Saving register state across invocations

- Processor doesn't save registers
- General purpose, segment, flags
- Again, a calling convention is needed
- Agreement on what gets saved by the callee and the caller

Saving register state across invocations

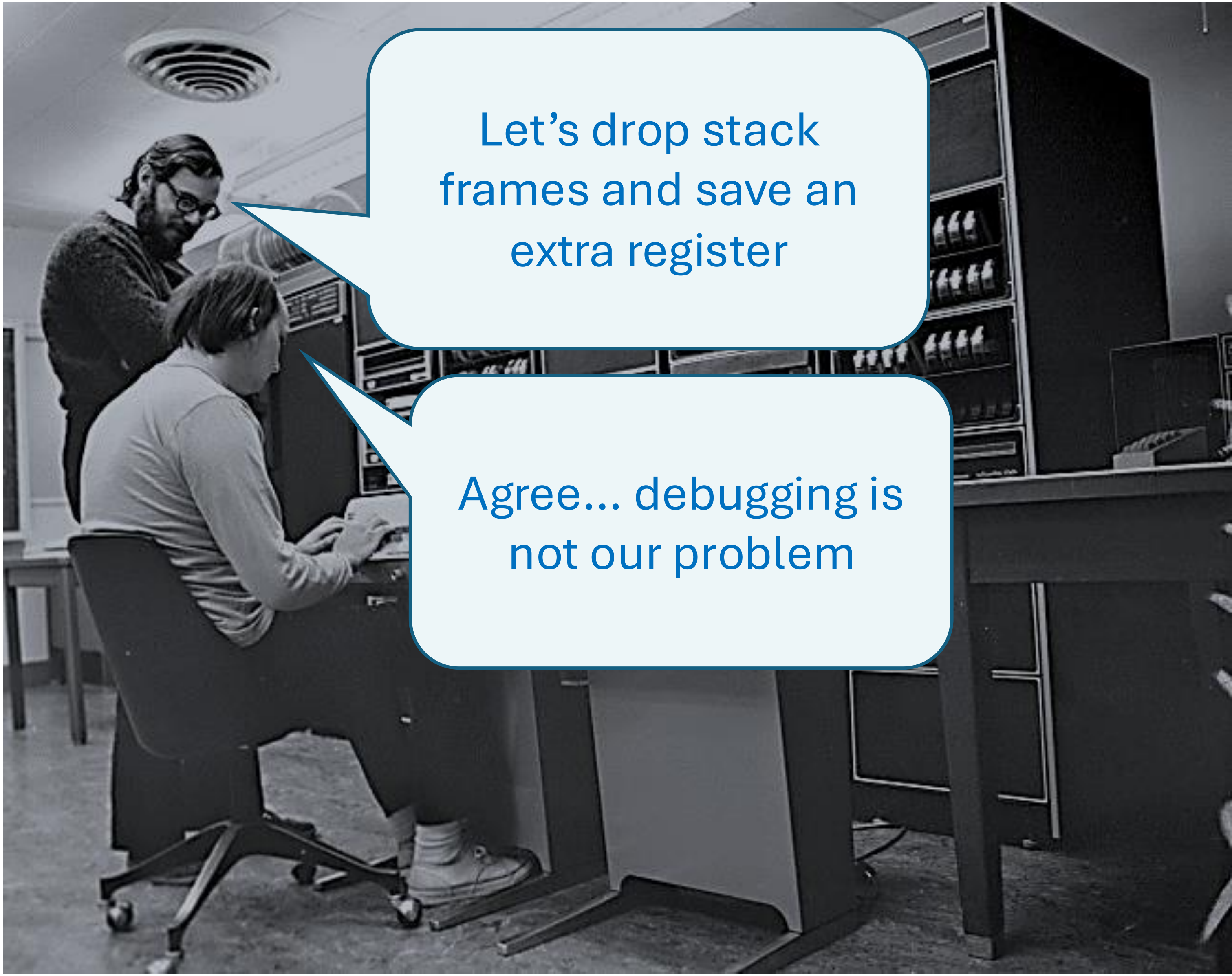
- System V AMD64 (Linux): caller-saved (volatile): RAX, RCX, RDX, RSI, RDI, R8–R11 (+ XMM0–XMM15)
  - Callee-saved (non-volatile): RBX, RBP, R12–R15 (and RSP must be restored)
- Rule: if the callee uses a callee-saved register, it must restore it before returning
  - Caller must assume caller-saved regs are clobbered across a call

- In general there are multiple calling conventions / ABIs
- On Linux x86-64 user-space the standard is the System V AMD64 ABI
- 32-bit code often used cdecl/stdcall/fastcall (different rules)
- Docs: https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI
- Be careful when mixing C/assembly, calling into libraries, or crossing ABIs

References

- https://en.wikibooks.org/wiki/X86_Disassembly/Functions_and_Stack_Frames
- https://en.wikipedia.org/wiki/Calling_convention
- https://en.wikipedia.org/wiki/X86_calling_conventions
- http://stackoverflow.com/questions/14666665/trying-to-understand-gcc-option-fomit-frame-pointer
- https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI

Thank you!


Let’s drop stack frames and save an extra register
Agree… debugging is not our problem