

cs5460/6460 Operating Systems

Lecture 4: Function invocations, and calling conventions

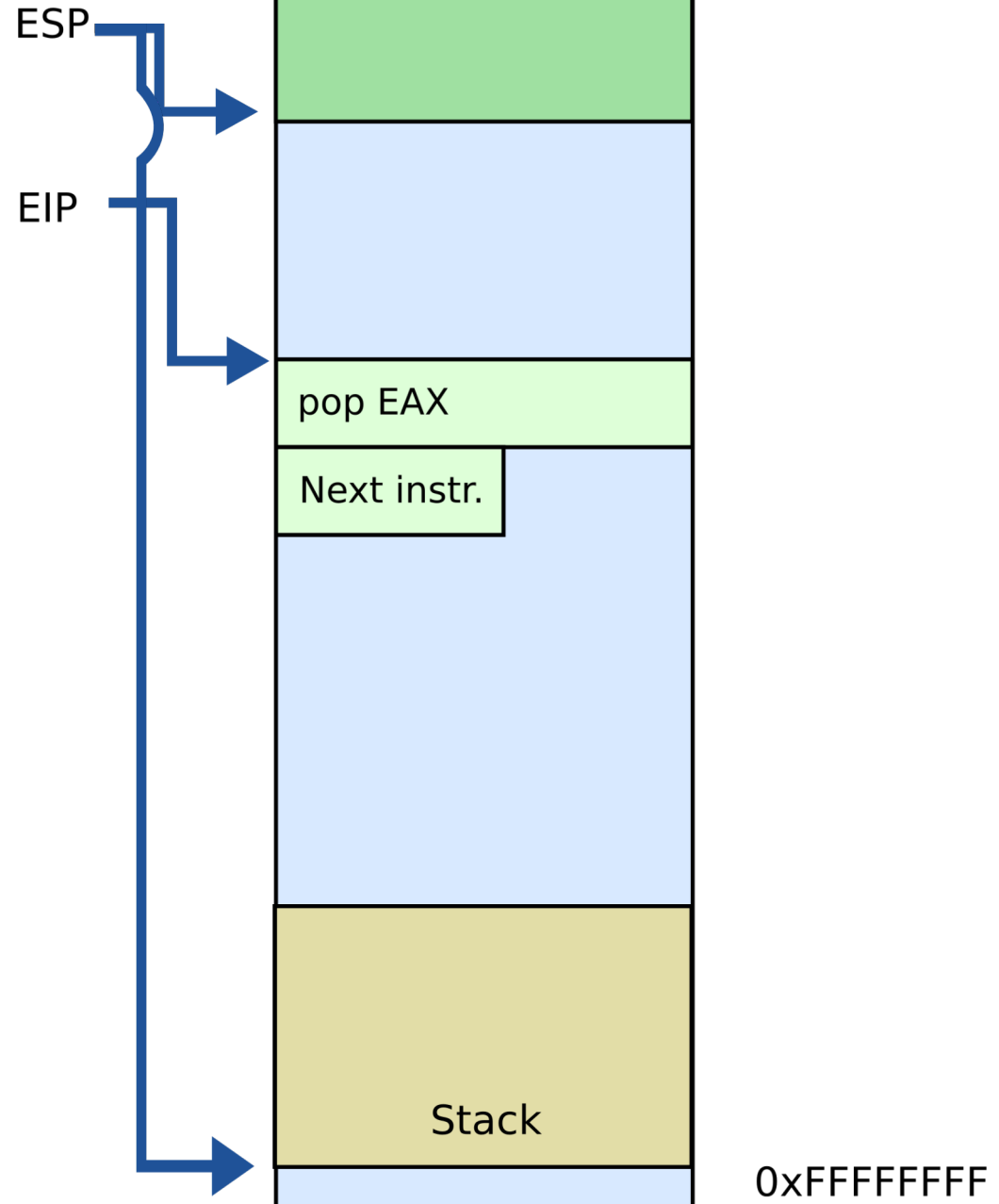
Anton Burtsev

January, 2024

Recap: stack

Stack

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



Why do we need stack?

Stack allows us to invoke functions

Calling functions

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

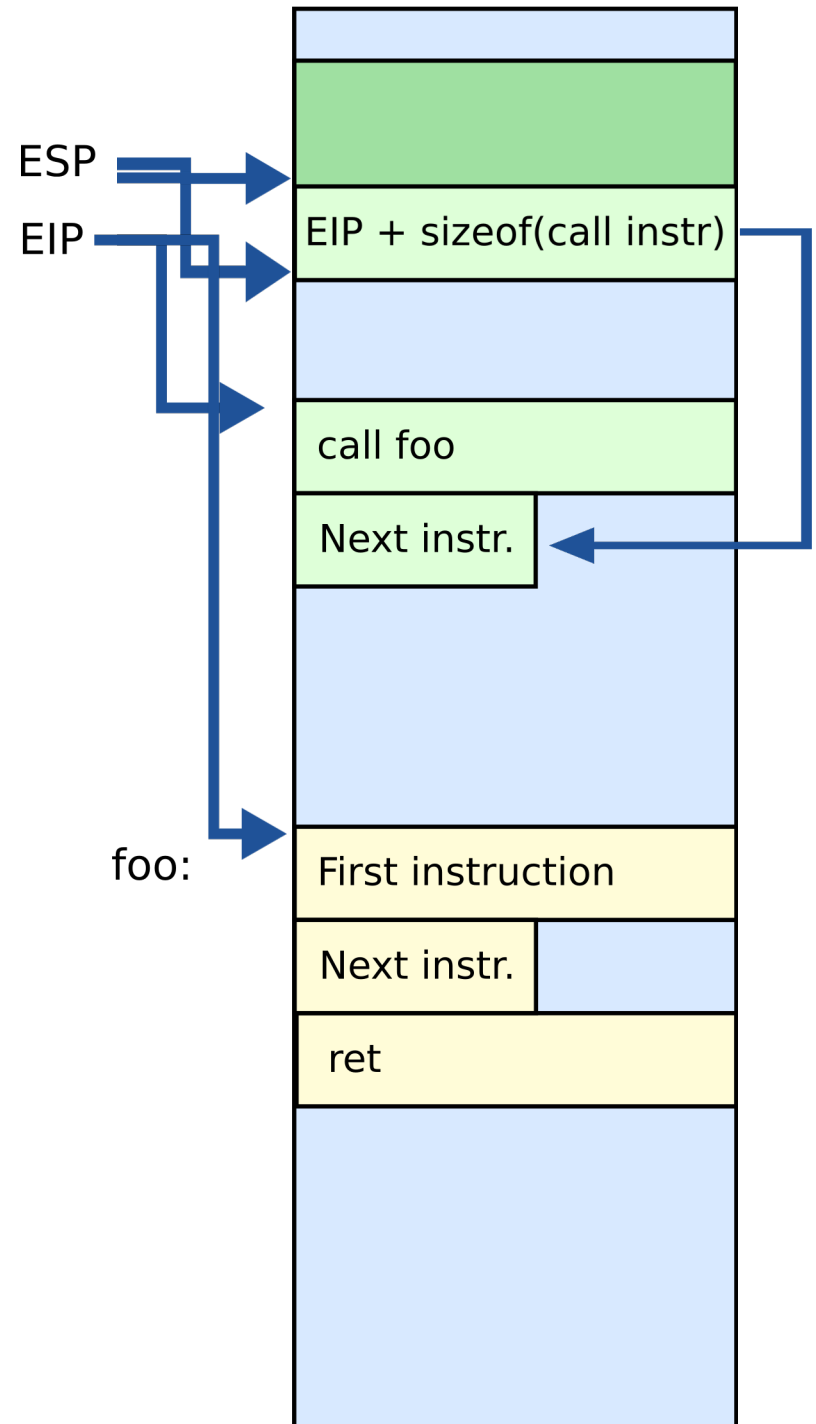
- Stack contains information for **how to return** from a subroutine
- i.e., from foo()

- Functions can be called from different places in the program

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

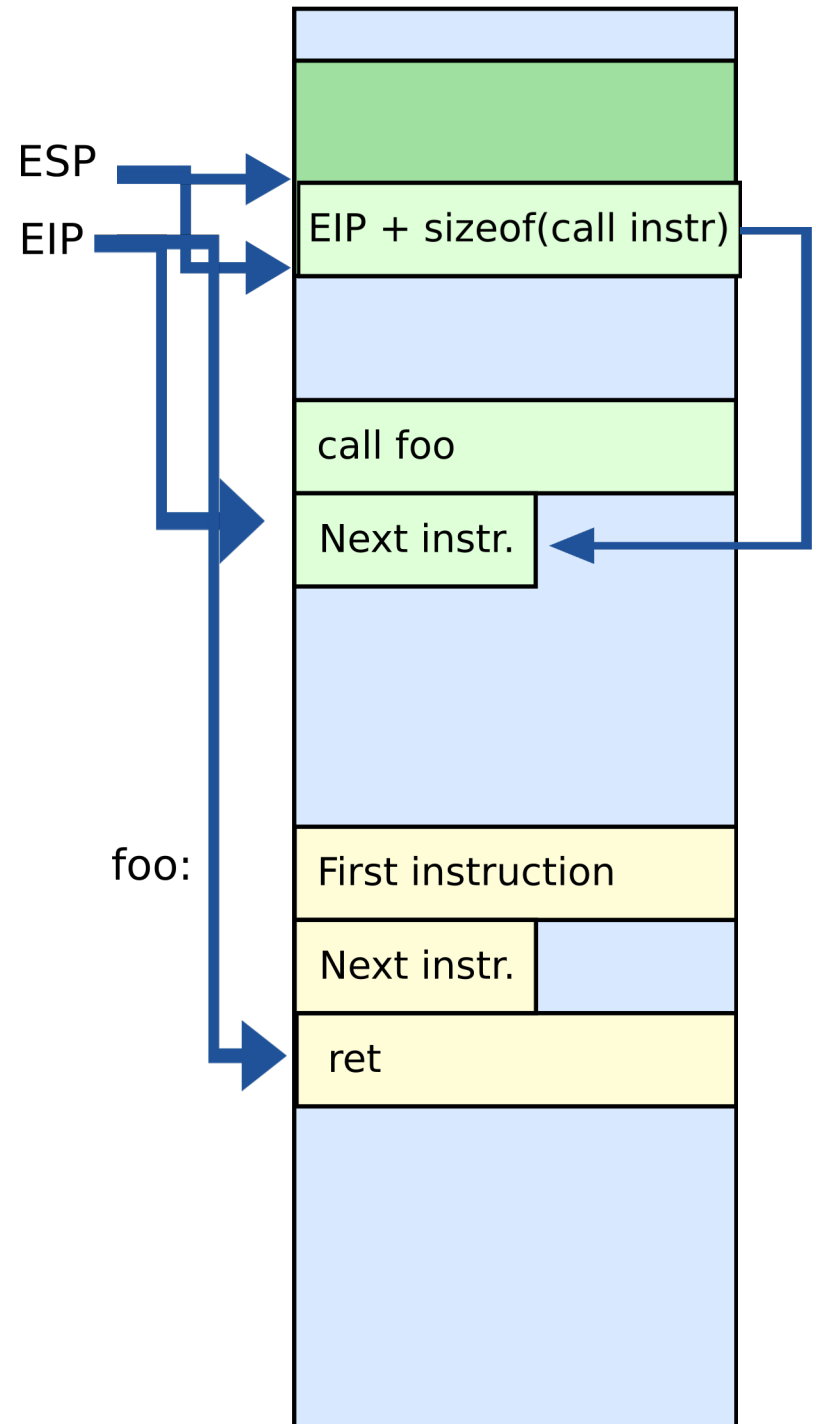
Stack

- 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



Calling functions

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

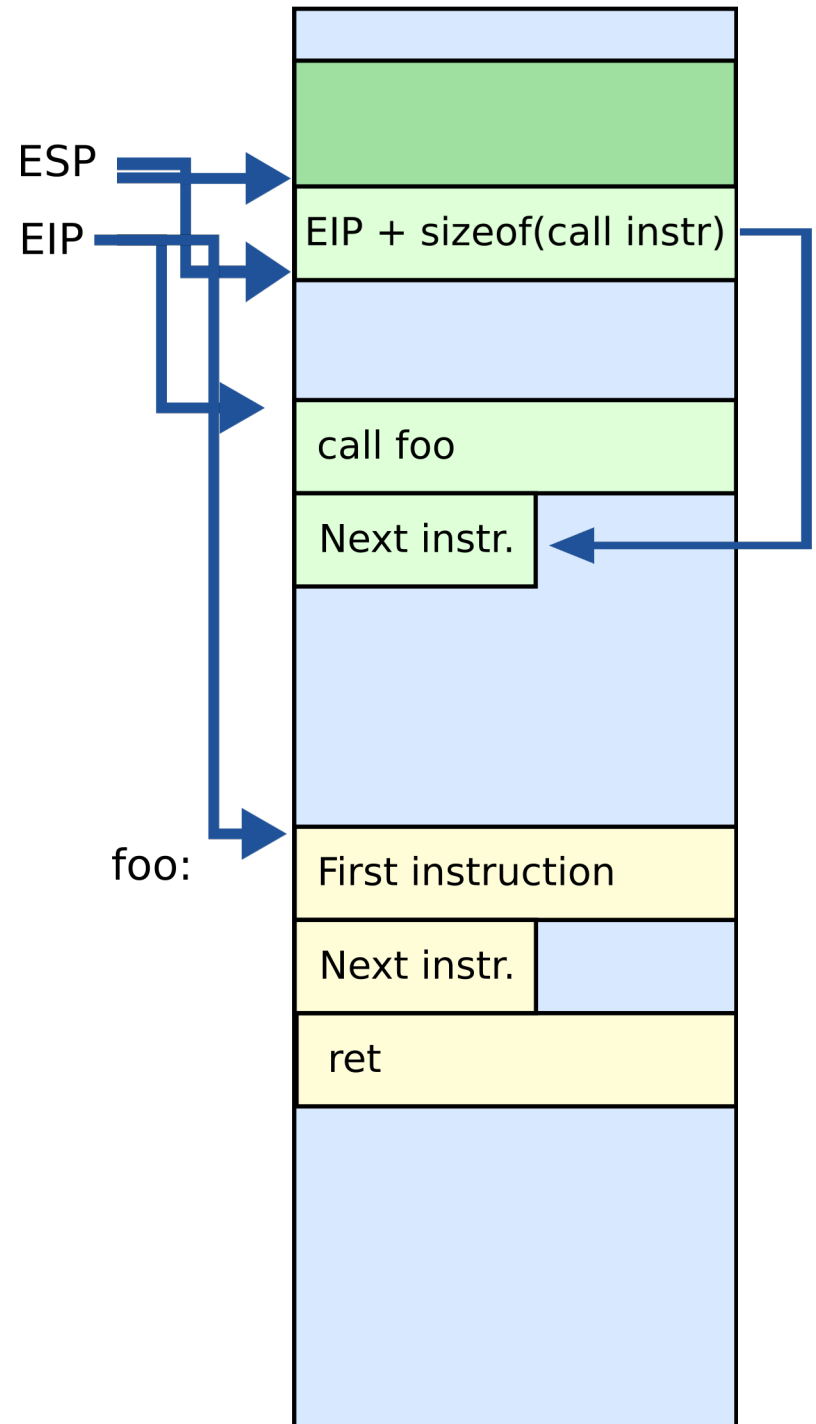
.Stack contains
information for **how to
return** from a subroutine
.i.e., from foo()

.Functions can be called
from different places in
the program

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

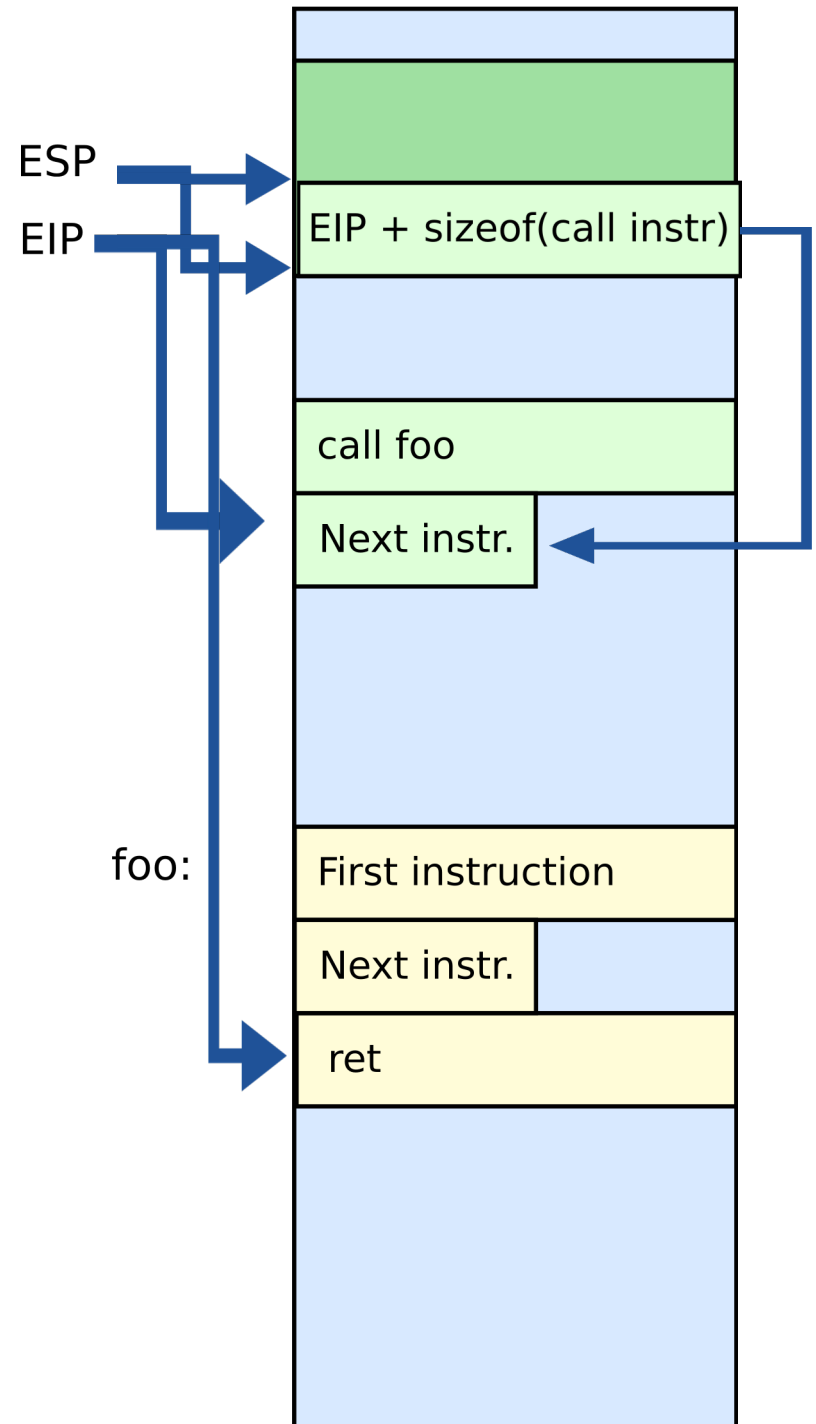
Stack

- 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);
```

Calling conventions

Calling conventions

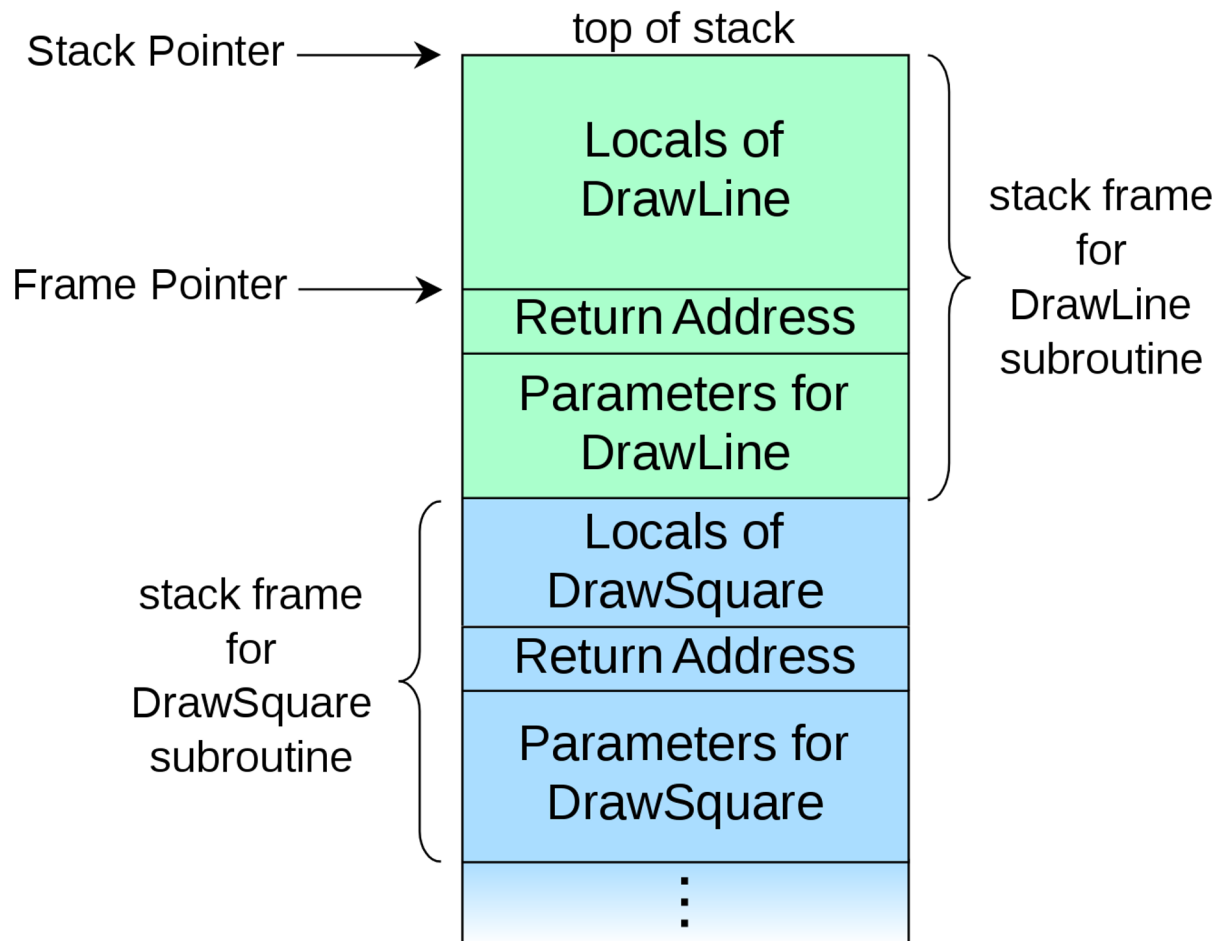
- Goal: re-entrant programs
- How to pass arguments
 - On the stack?
 - In registers?
- How to return values
 - On the stack?
 - In registers?
- Conventions vary between compilers, optimizations, etc.

Maintain stack as frames

- Each function has a new frame

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

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

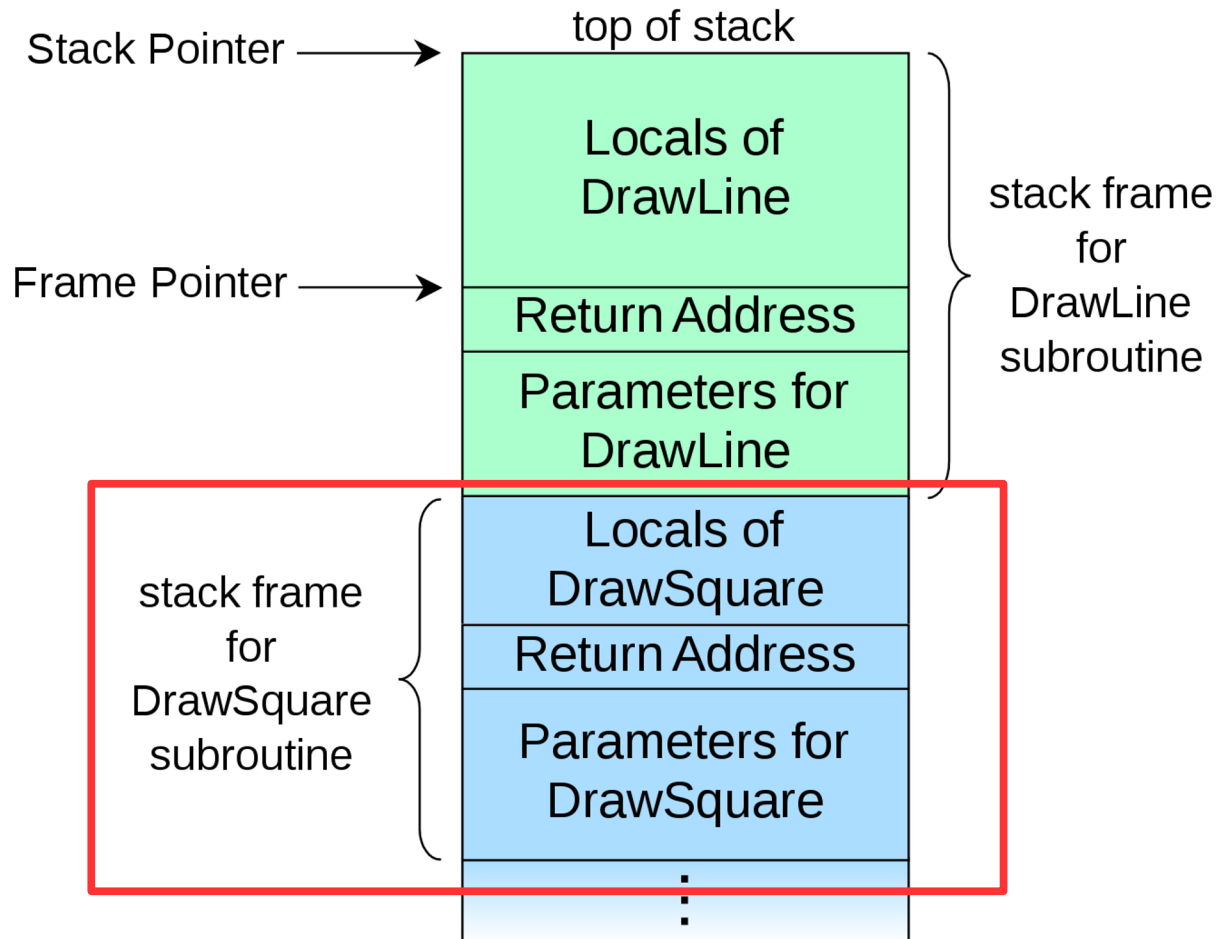


Maintain stack as frames

- Each function has a new frame

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

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

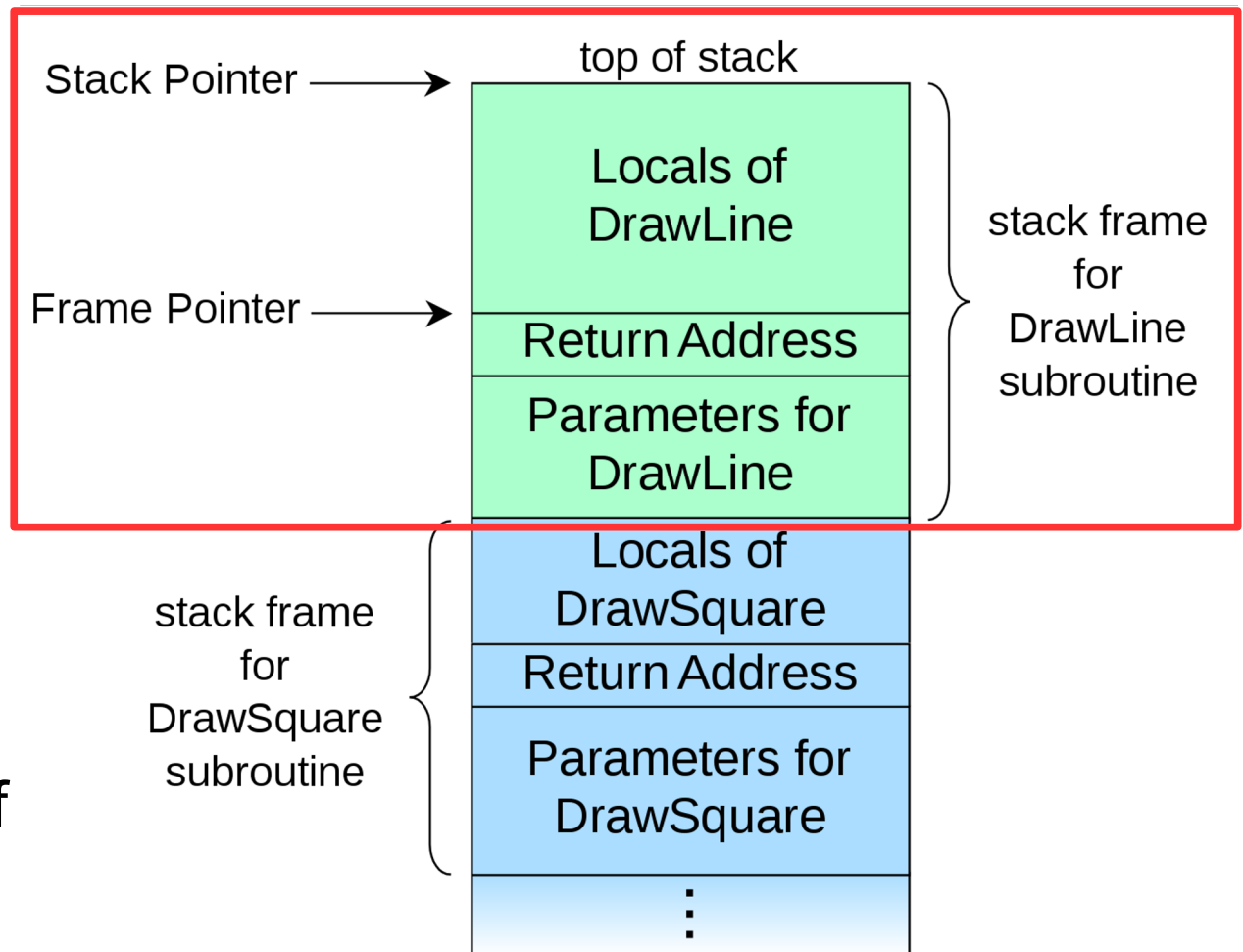


Maintain stack as frames

- Each function has a new frame

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

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



Prologue/epilogue

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

```
my_function:
    push ebp          ; save original EBP value on stack
    mov ebp, esp      ; new EBP = ESP
    ...               ; function body
    pop ebp           ; restore original EBP 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
 - Uninitialized → BSS
- Dynamic variables
 - Heap
- Local variables
 - Stack

Global variables

```
1. #include <stdio.h>

2. char hello[] = "Hello";
3. int main(int ac, char **av)
4. {
5.     static char world[] = "world!";
6.     printf("%s %s\n", hello, world);
7.     return 0;
8. }
```

Global variables

```
1. #include <stdio.h>
```

```
2. char hello[] = "Hello";
```

```
3. int main(int ac, char **av)
```

```
4. {
```

```
5.     static char world[] = "world!";
```

```
6.     printf("%s %s\n", hello, world);
```

```
7.     return 0;
```

```
8. }
```

- Allocated in the data section
- It is split in initialized (non-zero), and non-initialized (zero)
- As well as read/write, and read only data section

Global variables

Dynamic variables (heap)

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>

4. char hello[] = "Hello";
5. int main(int ac, char **av)
6. {
7.     char world[] = "world!";
8.     char *str = malloc(64);
9.     memcpy(str, "beautiful", 64);
10.    printf("%s %s %s\n", hello, str, world);
11.    return 0;
12. }
```

Dynamic variables (heap)

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>

4. char hello[] = "Hello";
5. int main(int ac, char **av)
6. {
7.     char world[] = "world!";
8.     char *str = malloc(64);
9.     memcpy(str, "beautiful", 64);
10.    printf("%s %s %s\n", hello, str, world);
11.    return 0;
12. }
```

- 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

```
1. #include <stdio.h>

2. char hello[] = "Hello";
3. int main(int ac, char **av)
4. {
5.     //static char world[] = "world!";
6.     char world[] = "world!";
7.     printf("%s %s\n", hello, world);
8.     return 0;
9. }
```

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?

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

How to allocate local variables?

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

- On the stack!

Allocating local variables

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

```
_my_function:
    push ebp                ; save original EBP value on stack
    mov ebp, esp           ; new EBP = ESP
    sub esp, LOCAL_BYTES   ; = # bytes needed by locals
    ...                    ; function body
    mov esp, ebp           ; deallocate locals
    pop ebp                ; restore original EBP value
    ret
```

Example

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

```
_my_function:  
    push ebp          ; save the value of ebp  
    mov ebp, esp      ; ebp = esp, set ebp to be top of the stack (esp)  
    sub esp, 12       ; move esp down to allocate space for the  
                      ; local variables on the stack
```

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

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

Example

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

```
_my_function:  
    push ebp          ; save the value of ebp  
    mov ebp, esp      ; ebp = esp, set ebp to be top of the stack (esp)  
    sub esp, 12       ; move esp down to allocate space for the  
                      ; local variables on the stack
```

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

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

How to pass arguments?

- Possible options:
- In registers
- On the stack

How to pass arguments?

- x86 32 bit
 - Pass arguments on the stack
 - Return value is in EAX and EDX
- x86 64 bit – more registers!
 - Pass first 6 arguments in registers
 - RDI, RSI, RDX, RCX, R8, and R9
 - The rest on the stack
 - Return value is in RAX and RDX

x86_32: passing arguments on the stack

- Example function

```
void my_function(int x, int y, int z)
{ ... }
```

- Example invocation

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

- Generated code

```
push 10
push 5
push 2
call _my_function
```

Example stack

:	:	
10	[ebp + 16]	(3rd function argument)
5	[ebp + 12]	(2nd argument)
2	[ebp + 8]	(1st argument)
RA	[ebp + 4]	(return address)
FP	[ebp]	(old ebp value) ← EBP points here
	[ebp - 4]	(1st local variable)
:	:	
:	:	
	[ebp - X]	(esp - the current stack pointer)

Example: caller side code

```
int callee(int, int, int);
```

```
int caller(void)
```

```
{
```

```
    int ret;
```

```
    ret = callee(1, 2, 3);
```

```
    ret += 5;
```

```
    return ret;
```

```
}
```

caller:

```
; manage own stack frame
```

```
push    ebp
```

```
mov     ebp, esp
```

```
; push call arguments
```

```
push    3
```

```
push    2
```

```
push    1
```

```
; call subroutine 'callee'
```

```
call    callee
```

```
; remove arguments from frame
```

```
add     esp, 12
```

```
; use subroutine result
```

```
add     eax, 5
```

```
; restore old call frame
```

```
pop     ebp
```

```
; return
```

```
ret
```

Example: caller side code

```
int callee(int, int, int);

int caller(void)
{
    int ret;

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

caller:

```
; manage own stack frame
push    ebp
mov     ebp, esp
; push call arguments
push    3
push    2
push    1
; call subroutine 'callee'
call    callee
; remove arguments from frame
add     esp, 12
; use subroutine result
add     eax, 5
; restore old call frame
pop     ebp
; return
ret
```

Example: caller side code

```
int callee(int, int, int);

int caller(void)
{
    int ret;

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

caller:

```
; manage own stack frame
push    ebp
mov     ebp, esp
; push call arguments
push    3
push    2
push    1
; call subroutine 'callee'
call    callee
; remove arguments from frame
add     esp, 12
; use subroutine result
add     eax, 5
; restore old call frame
pop     ebp
; return
ret
```

Example: caller side code

```
int callee(int, int, int);

int caller(void)
{
    int ret;

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

caller:

```
; manage own stack frame
push    ebp
mov     ebp, esp
; push call arguments
push    3
push    2
push    1
; call subroutine 'callee'
call    callee
; remove arguments from frame
add     esp, 12
; use subroutine result
add     eax, 5
; restore old call frame
pop     ebp
; return
ret
```

Example: caller side code

```
int callee(int, int, int);

int caller(void)
{
    int ret;

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

caller:

```
; manage own stack frame
push    ebp
mov     ebp, esp
; push call arguments
push    3
push    2
push    1
; call subroutine 'callee'
call    callee
; remove arguments from frame
add     esp, 12
; use subroutine result
add     eax, 5
; restore old call frame
pop     ebp
; return
ret
```

Example: caller side code

```
int callee(int, int, int);

int caller(void)
{
    int ret;

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

caller:

```
; manage own stack frame
push    ebp
mov     ebp, esp
; push call arguments
push    3
push    2
push    1
; call subroutine 'callee'
call    callee
; remove arguments from frame
add     esp, 12
; use subroutine result
add     eax, 5
; restore old call frame
pop     ebp
; return
ret
```

Wait! Where is return ret;?

```
int callee(int, int, int);

int caller(void)
{
    int ret;

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

```
caller:
    ; manage own stack frame
    push    ebp
    mov     ebp, esp
    ; push call arguments
    push    3
    push    2
    push    1
    ; call subroutine 'callee'
    call    callee
    ; remove arguments from frame
    add     esp, 12
    ; use subroutine result
    add     eax, 5
    ; restore old call frame
    pop     ebp
    ; return
    ret
```

Example: callee side code

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

```
_my_function:
    push ebp
    mov ebp, esp
    sub esp, 12 ; allocate local variables
                  ; sizeof(a) + sizeof(b) + sizeof(c)
                  ; x=[ebp + 8], y=[ebp + 12], z=[ebp + 16]
                  ; a=[ebp-4]=[esp+8],
                  ; b=[ebp-8]=[esp+4], c=[ebp-12] = [esp]
    mov esp, ebp ; deallocate local variables
    pop ebp
    ret
```

leave instruction

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

```
_my_function:
    push ebp
    mov ebp, esp ; ebp = esp
    sub esp, 12 ; allocate local variables
                  ; sizeof(a) + sizeof(b) + sizeof(c)
    ; x = [ebp + 8], y = [ebp + 12], z = [ebp + 16]
    ; a=[ebp-4]=[esp+8],
    ; b=[ebp-8]=[esp+4], c=[ebp-12] = [esp]
    mov esp, ebp
    pop ebp
    ret
```

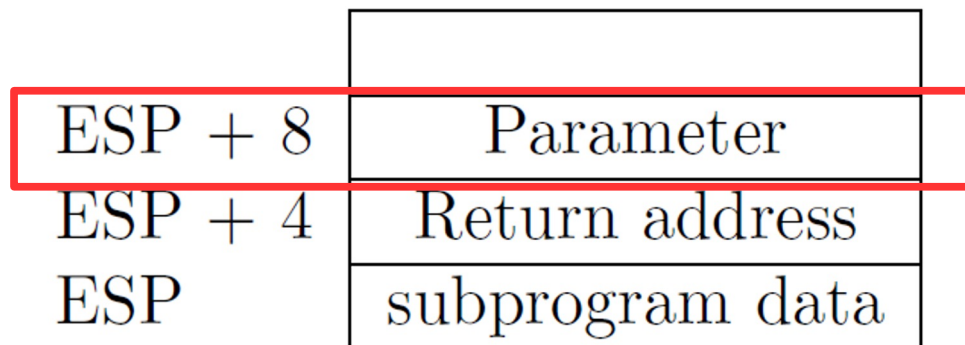
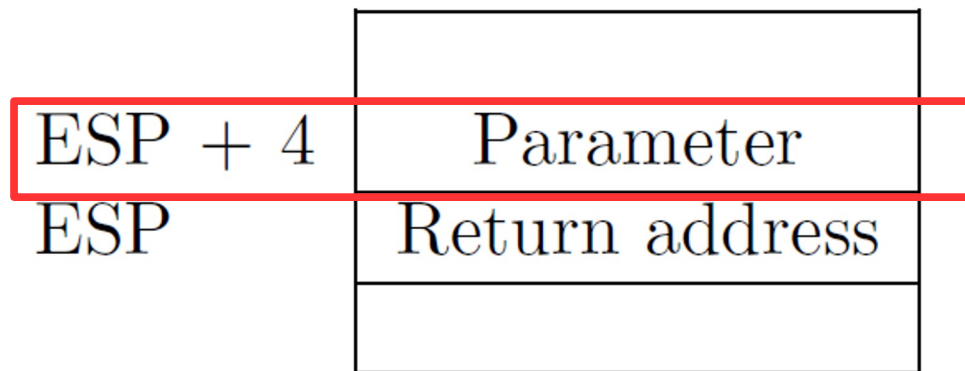
- x86 has a special instruction for this
- leave

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



- Initially parameter is $[ESP + 4]$
- Later as the function pushes things on the stack it changes, e.g., $[ESP + 8]$

- Debugging becomes hard
 - As ESP changes one has to manually keep track where local variables are relative to ESP (ESP + 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 couple instructions required to maintain the stack frame: **one register (EBP)**
 - x32 has 8 registers (and one is ESP, so 7 are left)
 - So taking another one is 1/7 or 14.28% of register space
 - Sometimes it makes sense!
- x64 has **16 registers**, so it doesn't really matter
- That said, GCC sets `-fomit-frame-pointer` to “on”
 - At `-O`, `-O1`, `-O2` ...
 - Don't get surprised

Relevant part of the GCC manual

3.10 Options That Control Optimization

<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

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

- Registers EAX, ECX, and EDX are caller-saved
- The function is free to use them
- ... the rest are callee-saved
- If the function uses them it has to restore them to the original values

- In general there multiple calling conventions
- We described **cdecl**
- **Make sure you know what you're doing**

https://en.wikipedia.org/wiki/X86_calling_conventions#cdecl

- It's easy as long as you know how to read the table

Intel vs GNU ASM

Intel

- Copy ebx into eax

```
mov eax, ebx
```

- Move the 4 bytes in memory at the address contained in EBX into EAX

```
mov eax, [ebx]
```

- Move 4 bytes at memory address ESI + (-4) into EAX

```
mov eax, [esi-4]
```

GNU

```
mov %ebx, %eax
```

```
mov (%ebx), %eax
```

```
mov -4(%esi), %eax
```

Questions?

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>