# 143A: Principles of Operating Systems

# Lecture 5: Calling conventions

Anton Burtsev
January, 2017

# Stack and procedure calls

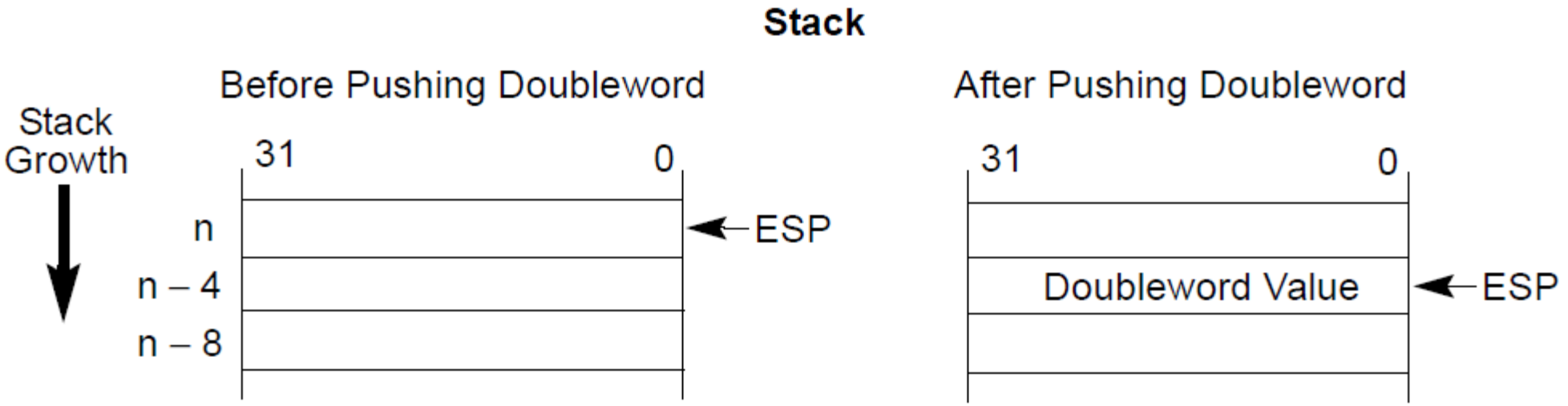
# Stack

- Main purpose:
  - **Store the return address** for the current procedure
  - Caller pushes return address on the stack
  - Callee pops it and jumps
- Other uses:
  - Local data storage
  - Parameter passing
  - Evaluation stack
    - Register spill

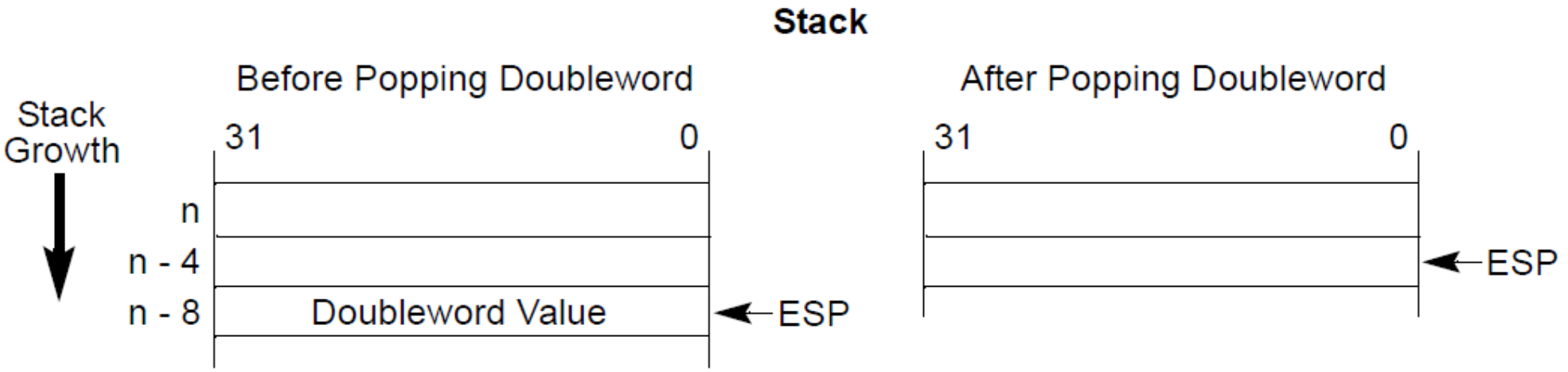
# Manipulating stack

- ESP register
  - Contains the memory address of the topmost element in the stack
- PUSH/POP instructions
  - Insert/remove data on the stack
  - Subtract/add 4 to ESP

# Example: PUSH



# Example: POP


Stack
Before Popping Doubleword
Stack Growth
31
0
n
n - 4
n - 8
Doubleword Value
ESP
After Popping Doubleword
31
0
ESP

# Call/return

- CALL instruction
  - Makes an unconditional jump to a subprogram and pushes the address of the next instruction on the stack

```
push eip + 2 ; save return address

jmp _my_function
```

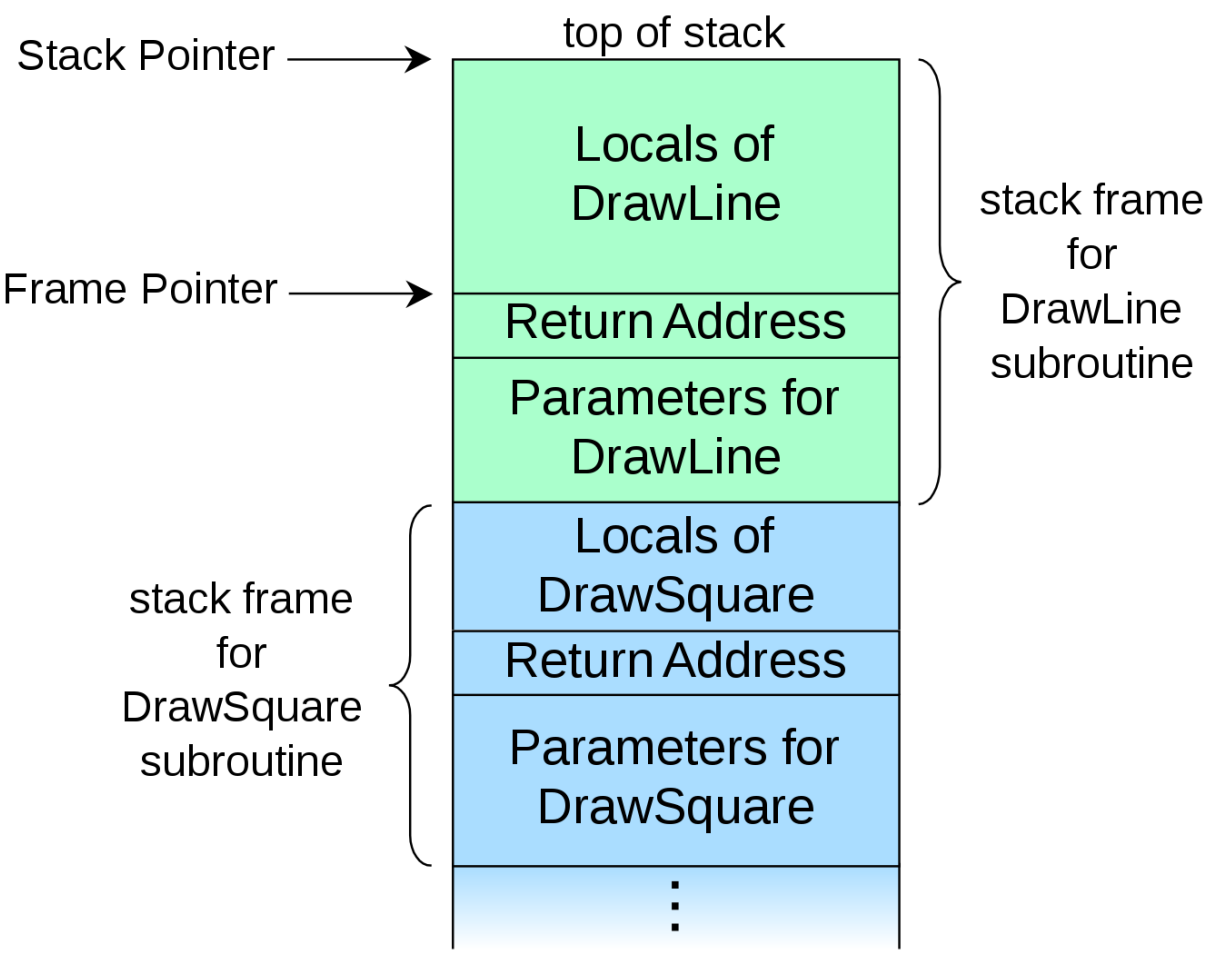
- RET instruction
  - Pops off an address and jumps to that address

# Calling conventions

- Goal: reentrant programs
  - How to pass arguments
    - On the stack?
    - In registers?
  - How to return values
    - On the stack?
    - In registers?
- Conventions differ from compiler, optimizations, etc.

# Stack consists of frames

- Each function has a new frame
- Use dedicated register **EBP** (frame pointer)
  - Points to the base of the frame



# Prologue/epilogue

- Each function maintains the frame
  - Uses prologue (blue), and epilogue (yellow)

```
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
```

# How to allocate local variables

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

# Allocating local variables

On the stack!

- Each function has a private instance
- Can call recursively

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

# Allocating local variables

- Stored right after the saved EBP value in 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  ; 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 dereferencing 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?

- Options
  - 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 - 4]  (1st local variable)
:     :
:     :
|    | [ebp - X]  (esp - the current stack pointer)
```

# Example: calle 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 varaibles
              ; 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
```

# Example: caller side code

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

int caller(void)
{
    int ret;

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

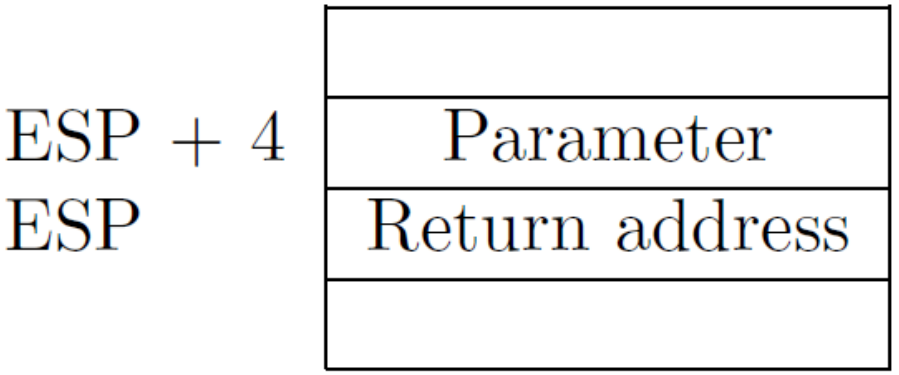
```
caller:
  ; make new call frame
  push    ebp
  mov     ebp, esp
  ; push call arguments
  push    3
  push    2
  push    1
  ; call subroutine 'callee'
  call    callee
  ; remove arguments from frame
  add     esp, 12
  ; use subroutine result
  add     eax, 5
  ; restore old call frame
  pop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

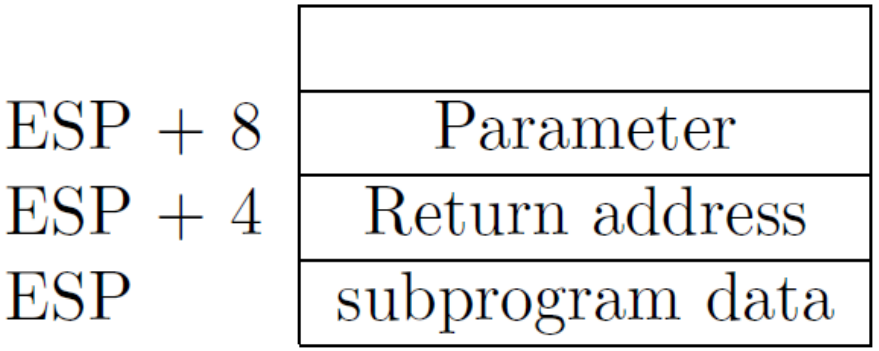
# Back to stack frames, so why do we need them?

- … 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.
```

# Stack 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 do this!
    - 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
- And 1 register (EBP)
  - x32 has 8 registers (and one is ESP)
    - So taking another one is 12.5% of register space
    - Sometimes its worse it!
  - 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

# 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 a callee and 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#List_of_x86_calling_conventions
  - It's easy as long as you know how to read the table

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