Practical Foundations of Debugging

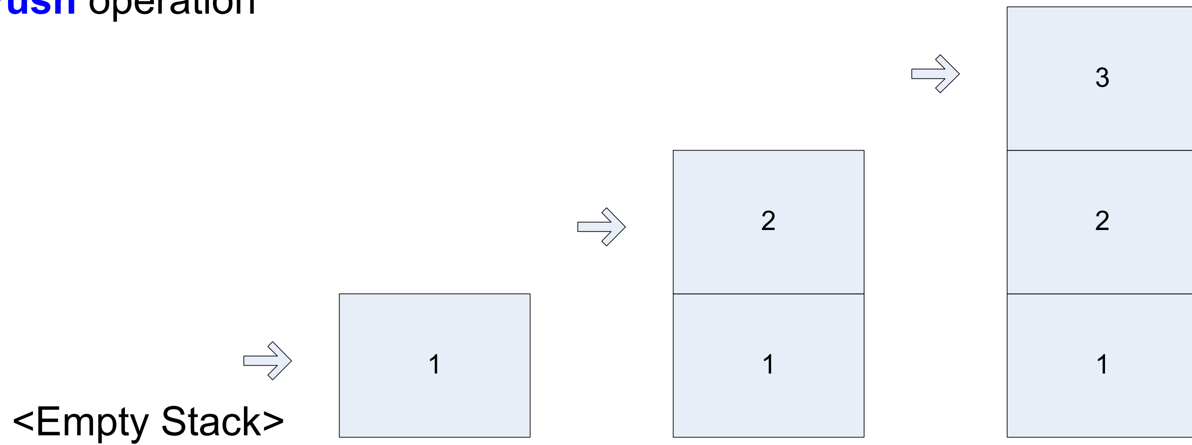
Chapter 5

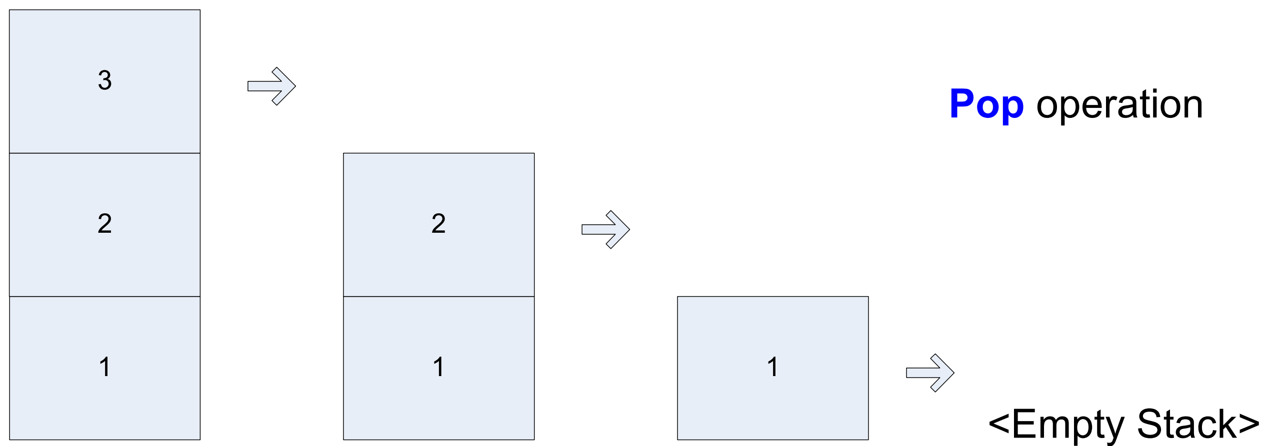
Memory and stacks

Designed, written and illustrated by Dmitry Vostokov

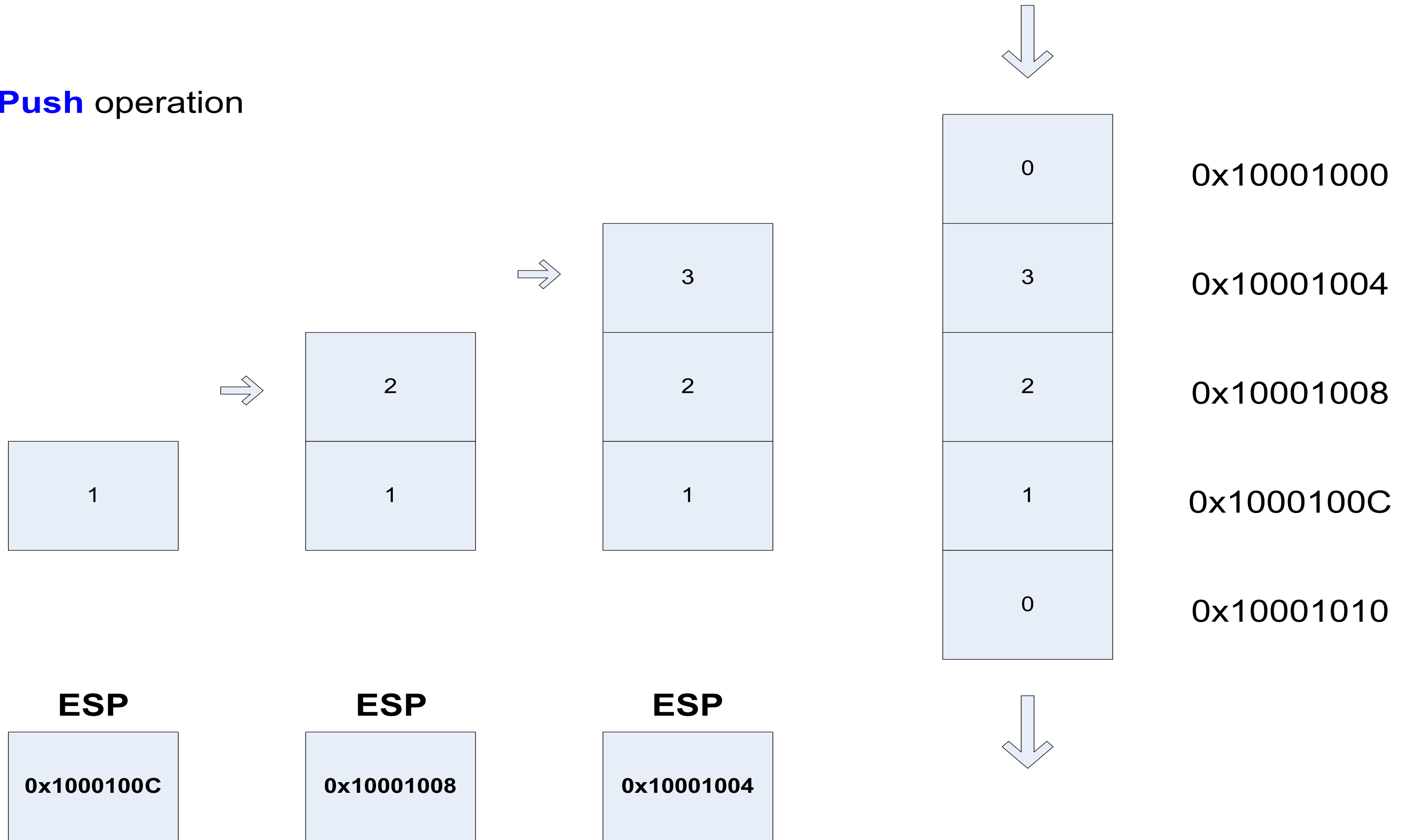
Stack

Push operation





Stack implementation in memory

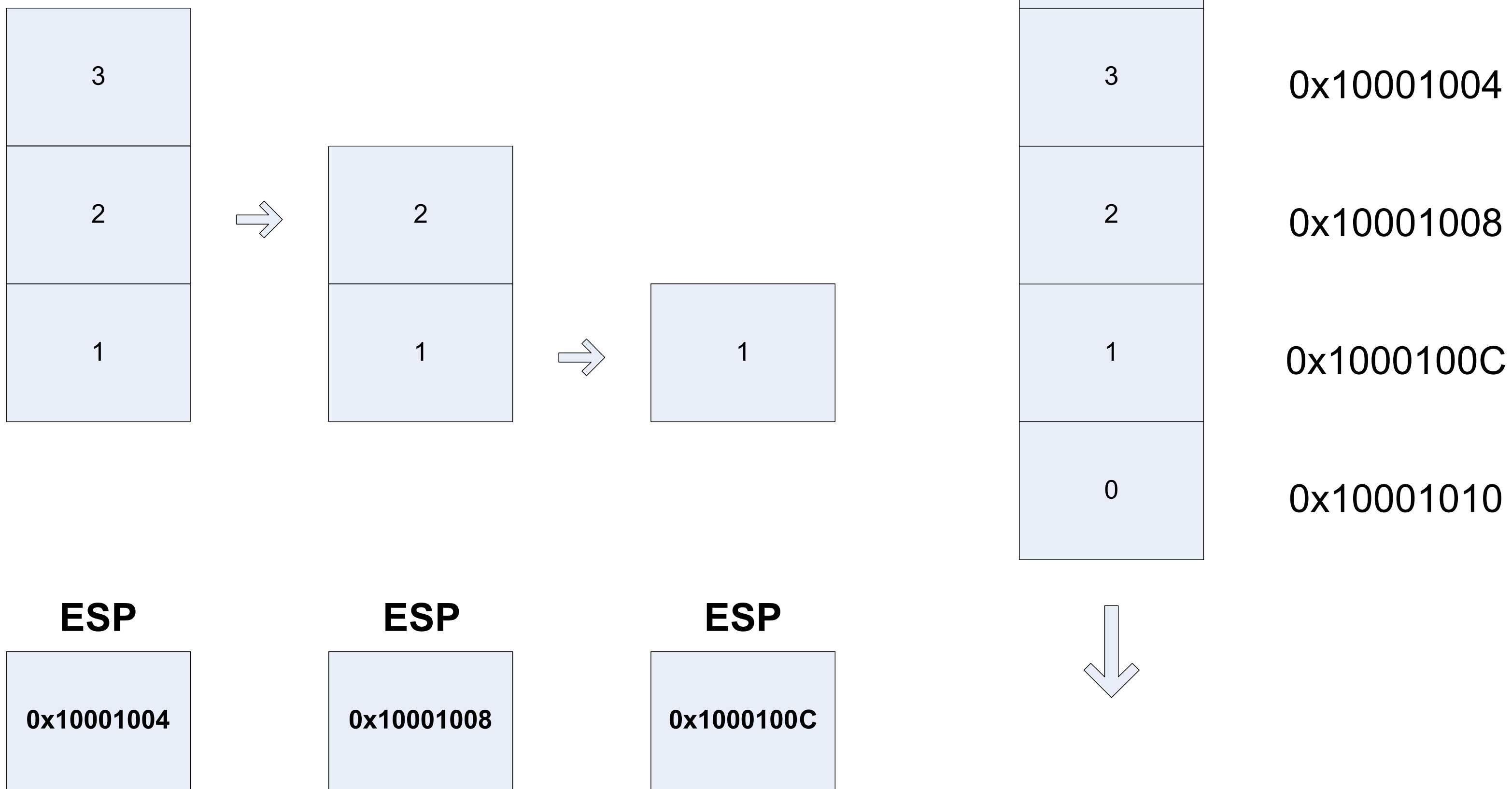


ESP – Stack Pointer

MOV EAX, [ESP] ; Reading top stack value

Pop operation

Pop operation



Things to remember

- Stack is LIFO – last input first output
- Stack grows down in memory
- ESP register points to top of stack

- Stacks are used to store return address for CALL instructions

- Stacks are used to pass parameters to functions and store temporary (local) variables

PUSH instruction

```
PUSH r/mem16/mem32/imm8/imm16/imm32

IF OperandSize = 32
   THEN
        ESP := ESP - 4
        [ESP] := OperandValue ; double word
   ELSE
        ESP := ESP - 2
        [ESP] := OperandValue ; word
FI

PUSH EAX
PUSH DWORD PTR [EAX]
PUSH WORD PTR [EAX]
PUSH 0
```

POP instruction

```
POP r/mem16/mem32

IF OperandSize = 32
   THEN
       OperandValue := [ESP] ; double word
       ESP := ESP + 4
   ELSE
       OperandValue := [ESP] ; word
       ESP := ESP + 2
FI

POP EAX
POP DWORD PTR [EAX]
POP WORD PTR [EAX]
```

Register review

General purpose

EAX
EBX
ECX
EDX

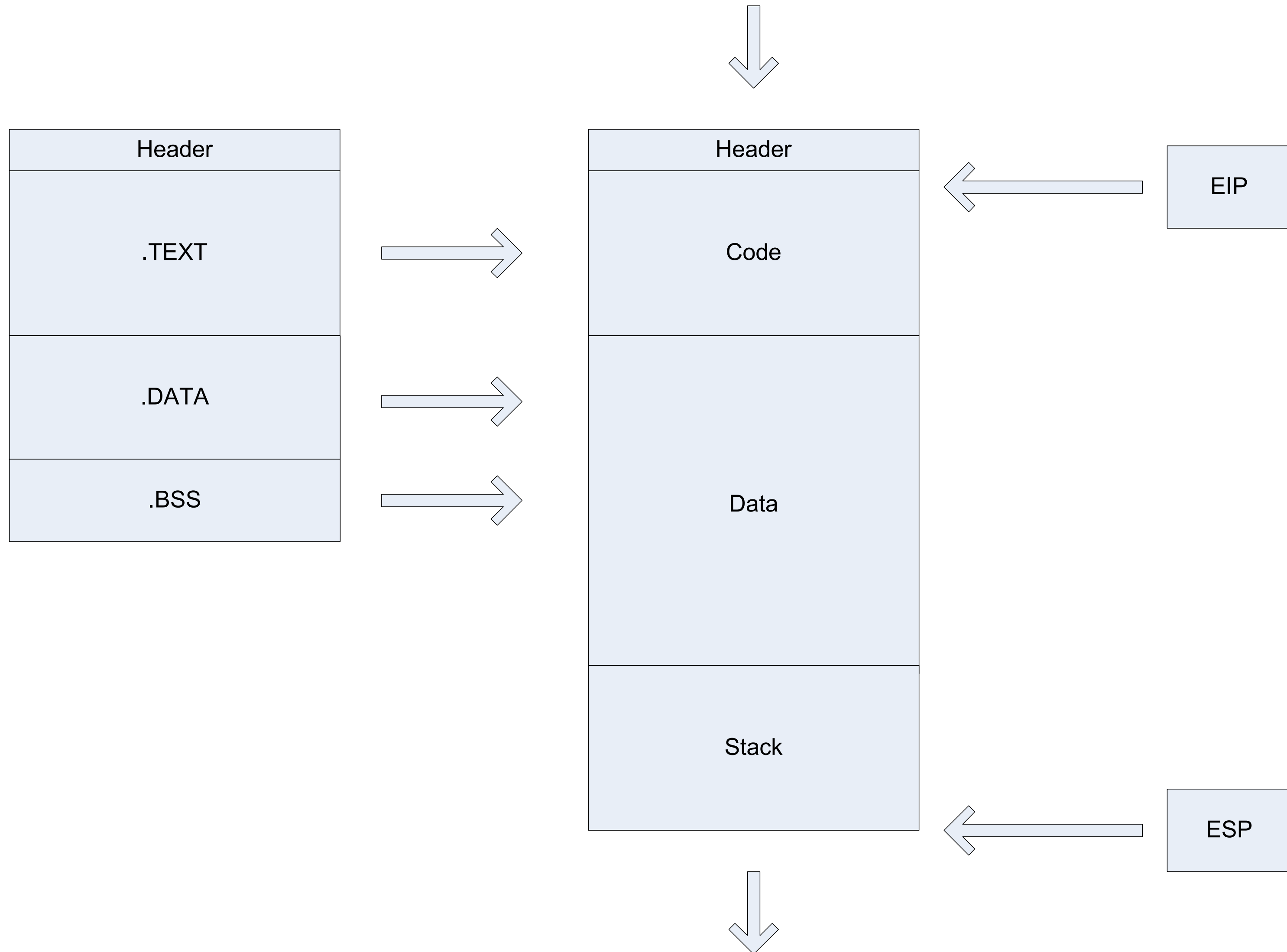
Instruction Pointer

EIP

Stack Pointer

ESP

Application memory (simplified)



Stack overflow

- By default stack size is 1Mb (compiler dependent), however this limit can be changed by linker /STACK option
- If stack grows beyond the limit than stack overflow exception occurs (exception code C00000FD)
- Caused by:

unlimited recursion

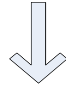
```
int func()
{
        func();
        return 0;
}
```

very big local variables

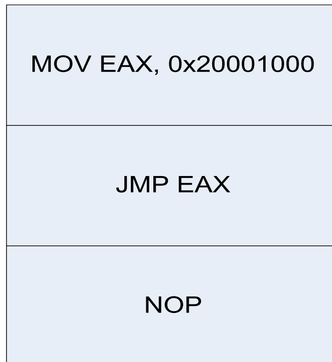
```
int func()
{
        int array[1000000] = { 1  };
        printf("%d", array[1000000-1]);  // use array to prevent the compiler to optimize it away
}
```

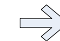
Jumps

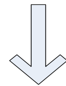
JMP instruction

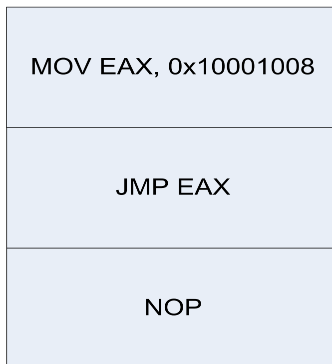


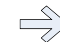
EIP

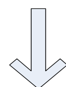


Address	EIP		
0x10001000	0x10001004		
0x10001004	0x10001008		0x20001000
0x10001008	0x1000100C		





Address	EIP		
0x20001000	0x20001004		
0x20001004	0x20001008		0x10001008
0x20001008			



JMP instruction (absolute)

- JMP r/mem32

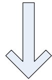
EIP := DEST

JMP EAX

JMP [EAX]

Calls

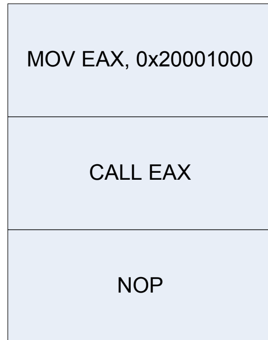
CALL instruction



Stack

EIP

ESP



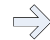
0x10001000

0x10001004

0x10001008

0x10001004

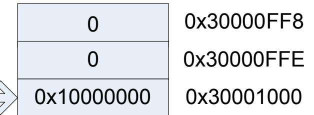
0x10001008



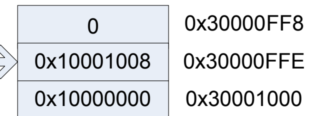
0x20001000

0x1000100C

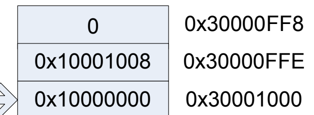
0x30001000

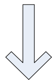


0x30000FFE



0x30001000





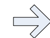
0x20001000

0x20001004

0x20001008

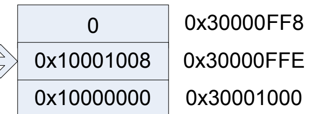
0x20001004

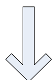
0x20001008



0x10001008

0x30000FFE





CALL and RET instructions (absolute)

- CALL r/mem32

PUSH (EIP)
EIP := DEST

CALL EAX
CALL [EAX]

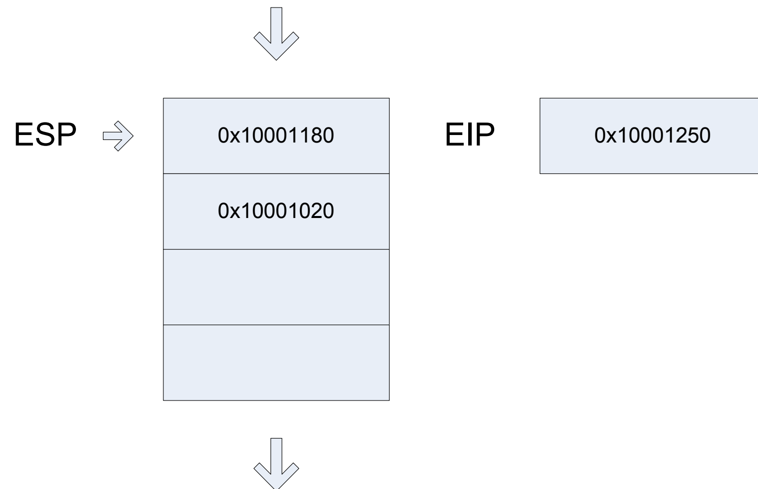
- RET

EIP := POP()

Call stack

func	0x10001000 – 0x10001100
func2	0x10001101 – 0x10001200
func3	0x10001201 – 0x10001300

```
func3
func2
func
```



```
void func()
{
    func2();
}

void func2()
{
    func3();
}
```

Exploring stack in WinDbg

```
0:000> r
eax=00321428 ebx=7ffdf000 ecx=00000001 edx=7ffe0304 esi=00000a28 edi=00000000
eip=00401033 esp=0012fecc ebp=0012fecc iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
SimpleStack!func3+0x3:
00401033 cc               int     3
```

```
0:000> dds esp

0012fecc  0012fed4
0012fed0  00401028 SimpleStack!func2+0x8
0012fed4  0012fedc
0012fed8  00401018 SimpleStack!func+0x8
0012fedc  0012fee4
0012fee0  00401008 SimpleStack!main+0x8
0012fee4  0012ffc0
0012fee8  004011ce SimpleStack!mainCRTStartup+0x173
…
…
…
```

What’s next?

- More on addressing modes
- Frame pointer
- Local variables
- “Arithmetical” project with local variables