8. Advanced Procedures

Tuesday, October 22, 2024 11:39 AM

* How do you provide a procedure what it needs?
  * Passing arguments through registers
  * Arguments to a procedure can be pushed onto the stack before calling a procedure.

old way	new way
mov eax, num1	push num1 = 8
mov ebx, num2	push num2 = 10
call AddTwo	call AddTwo

* pass arguments by value or pass arguments by reference
  - ↳ Array or String
* pushing arguments onto the stack (cannot push a byte)
  * push word or dword only
* ESP - register that points to the last element added to the stack.

```
push num1 (dword)   ← 8
push num2 (dword)   ← 10
call AddTwo
```

↓ push, jump

	Offset	Stack	
	1000	8	[EBP+12]
stack pointer	996	10	[EBP+8]
	992	Return Address	[EBP+4]
ESP →	988	EBP	

↓ EBP ↗

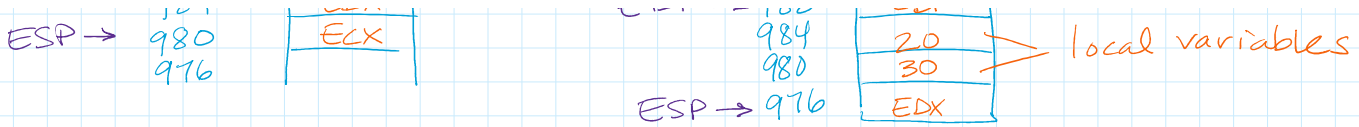
* dec. or inc. with push or pop.
* will not directly modify ESP.

```
AddTwo proc
    push EBP          ← preserve its value
    mov EBP, ESP      ← make a copy of ESP (base stack pointer)

    mov EAX, [EBP + 12]   ; move 8 into EAX
    add EAX, [EBP + 8]    ; add 10 to EAX
    pop EBP           - restore EBP
    ret 8             → add 8 to ESP (clears the arguments of the stack)
                      → pop & jump.
AddTwo endp
```

* Justification for Base Stack Pointer (EBP)

	offset	Stack
+12	1000	8
+8	996	10
+4	992	ret. addr.
EBP →	988	EBP
	984	EDX
ESP →	980	ECX
	976	

offset	Stack	
1000	8	arguments
996	10	
992	ret. addr.	
EBP → 988	EBP	
984	20	local variables
980	30	




```
SomeProc  Proc
          push  EBP
          mov   EBP, ESP
          sub   ESP, 8
          push  EDX

          mov [EBP-4], 20    ; local var
          mov [EBP-8], 30    ; local var
          pop EDX            ; restore EDX
          mov ESP, EBP       ; removes local variables
          pop EBP

          ret 8
SomeProc  endp
```

8.3 Recursion

* When a procedure calls itself

Example: 5! → 5 factorial → 5 · 4 · 3 · 2 · 1 = 120
↑ factorial

* Recursive implementations can be implemented with looping.

```
int x = factorial(5);        // x = 120
```

method: public int factorial (int n)

Java

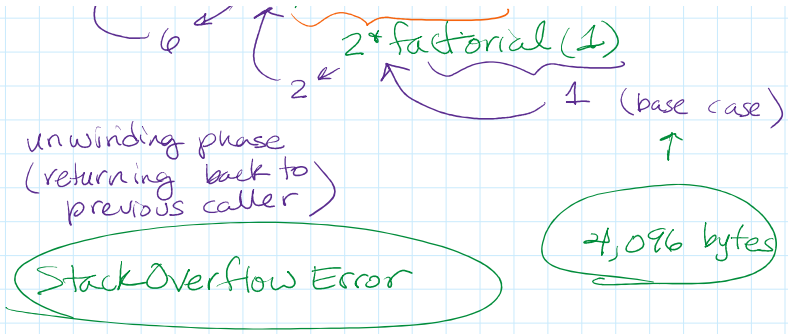
```
{
    if (n == 1)
        return 1;            // base case
    else
        return n * factorial (n-1);
}
```

method body

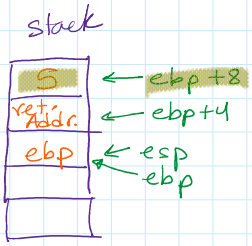
↑ recursive call

5 · factorial (4)
120 ← 24 ← 4 * factorial (3)
6 ← 3 * factorial (2)
2 ← 2 * factorial (1)
1

winding phase (engaging in recursive calls)

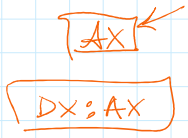



```
* main proc
    push 5          ;( 4 bytes worth of data)
    call factorial  ; factorial of num will be returned in EAX.
```

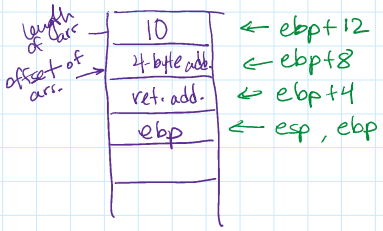



```
factorial proc
    push ebp
    mov ebp, esp        ; copy base-stack pointer

    mov ebx, [ebp+8]    ; move arg. into ebx
    cmp ebx, 1          ; check if base case
    je foundBC
    dec ebx             ; reduce arg. by 1
    push ebx
    call factorial      ; recursive call to factorial

foundBC:
    mov eax, 1
    pop ebp
    ret
```

multiply the argument



Arguments on Stack (arr of words)




```
mov esi, [ebp+8]
mov bx, [esi]      -> move first element of array into bx
mov bx, [esi+2]
```
doesn't change esi

* Recursion (winding phase & push/pop)

```
factorial proc
    push ebp
    ...
    push nextArg
    call factorial
    ...
```

base case

```
push ebp
    push nextArg
        call factorial
            push ebp
                push nextArg
                    call factorial
                        push ebp ✓
```

push nextArg
call factorial
:
pop ebp
ret

found BC:

pop ebp
ret

push ebp
push nextArg
call factorial
push ebp ✓
pop ebp ✓

pop arg
pop ebp

pop nextArg
pop ebp

INVOKE Directive — only available in 32-bit mode.
(form of abstraction)

* pushes args. onto stack & calls procedures

Before:

```
push type array
push lengthof array
push offset array
call DumpArray
```

```
INVOKE DumpArray,
    *offset array,
    * lengthof array,
    * type array
```

* If argument is less than 32-bits, then EAX, or EDX are used to expand argument to 32 bits, before pushing onto the stack.

* INVOKE fillArray, ADDR myArray

Like offset, can only be used with INVOKE.

PROC Directive — declare a procedure with comma-separated list of named parameters.

* use named parameters instead of [ebp+8]

```
AddTwo PROC,
    val1: DWORD,
    val2: DWORD

    mov eax, val1
    add eax, val2
    ret
AddTwo ENDP
```

→ List of parameters follows the PROC directive

* val1 & val2 are double words (4 bytes)

* INVOKE AddTwo, 8, 10

↓ Equivalent

```
AddTwo PROC
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    add eax, [ebp+12]
    ret 8
AddTwo ENDP
```

works with call instruction args. must be pushed before call.

```
push 10
push 8
call AddTwo
```

```
        mov eax, [ebp+8]
        add eax, [ebp+12]
        ret 8
AddTwo  ENDP
```

```
remote call:
push 10
push 8
call AddTwo
```

```
CountChars PROC,
        ptrString : PTR BYTE,      ← still 4-bytes
        someChar : DWORD           ← zero-extended byte character is contained in low byte.

        ;code not shown
        ret
CountChars ENDP
```

PROTO Directive – a way to identify a procedure and its parameters.

* creates a procedure prototype for an existing procedure.
* Allows us to call a procedure before defining it.
* MASM requires a prototype for each procedure called by INVOKE.
* PROTO must appear first before INVOKE

Example:

```
ArraySum PROC,
        ptrArray : PTR WORD
        szArray : DWORD
        :
ArraySum ENDP
```

copy & paste to make prototype

Prototype

```
ArraySum PROTO, ptrArray : PTR WORD, szArray : DWORD
```

* Calling convention is used by MS Windows 32-bit services.

```
ExitProcess proto, dwExitCode : dword
```
↑ prototype

in main

```
invoke ExitProcess, 0
```
↑ invoke built-in MS Windows service