Chapter 8: Advanced Procedures

Chapter Overview

- Stack Frames
- Recursion

Stack Frames

- Stack Parameters
- Local Variables
- ENTER and LEAVE Instructions
- LOCAL Directive

Stack Frame

- Also known as an *activation record*
- Area of the stack set aside for a procedure's return address, passed parameters, saved registers, and local variables
- Created by the following steps:
  - Calling program pushes arguments on the stack and calls the procedure.
  - The called procedure pushes EBP on the stack, and sets EBP to ESP.
  - If local variables are needed, a constant is subtracted from ESP to make room on the stack.

Stack Parameters

- More convenient than register parameters
- Two possible ways of calling DumpMem. Which is easier?

```
pushad
mov esi,OFFSET array
mov ecx,LENGTHOF array
mov ebx,TYPE array
call DumpMem
popad
```

```
push TYPE array
push LENGTHOF array
push OFFSET array
call DumpMem
```

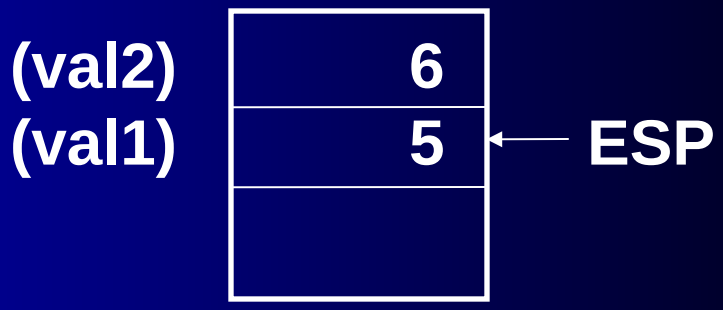
Passing Arguments by Value

- Push argument values on stack
  - (Use only 32-bit values in protected mode to keep the stack aligned)
- Call the called-procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called-procedure did not remove them

Example

```
.data
val1  DWORD 5
val2  DWORD 6

.code
push val2
push val1
```



Stack prior to CALL

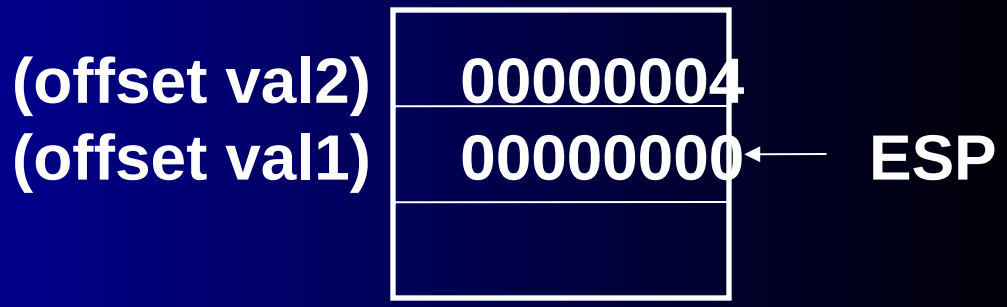
Passing by Reference

- Push the offsets of arguments on the stack
- Call the procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called procedure did not remove them

Example

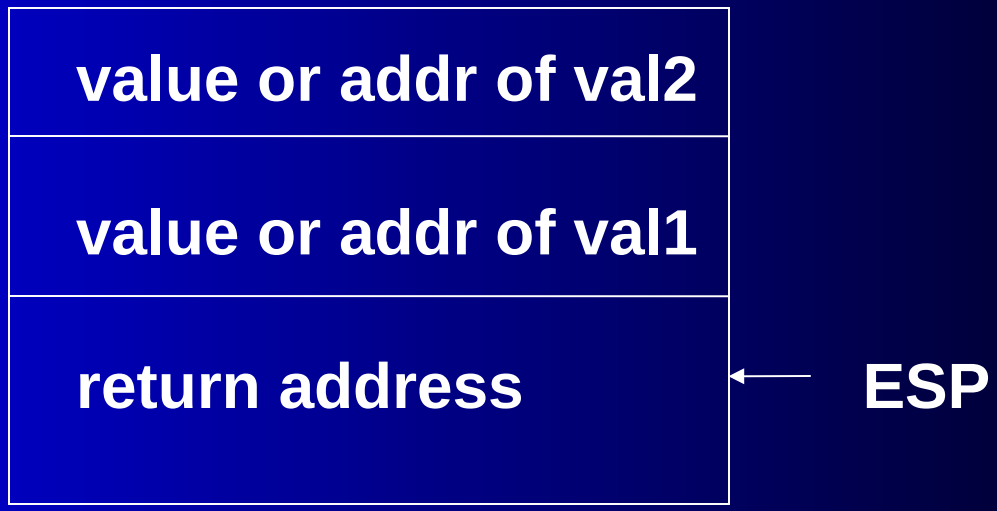
```
.data
val1  DWORD 5
val2  DWORD 6

.code
push OFFSET val2
push OFFSET val1
```



Stack prior to CALL

Stack after the CALL



Passing an Array by Reference (1 of 2)

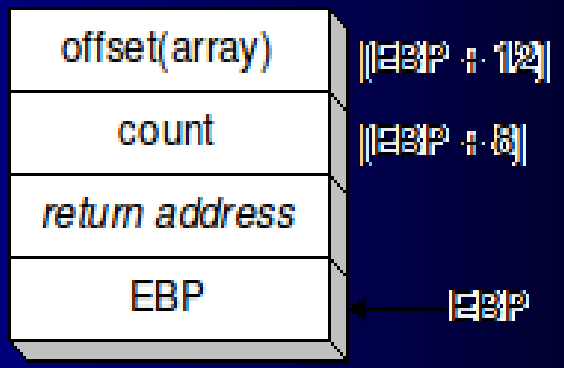
- The ArrayFill procedure fills an array with 16-bit random integers
- The calling program passes the address of the array, along with a count of the number of array elements:

```
.data
count = 100
array WORD count DUP(?)
.code
    push OFFSET array
    push COUNT
    call ArrayFill
```

Passing an Array by Reference (2 of 2)

ArrayFill can reference an array without knowing the array's name:

```
ArrayFill PROC
    push ebp
    mov  ebp,esp
    pushad
    mov  esi,[ebp+12]
    mov  ecx,[ebp+8]
    .
    .
```



ESI points to the beginning of the array, so it's easy to use a loop to access each array element.

View the complete program.

Accessing Stack Parameters (C/C++)

- C and C++ functions access stack parameters using constant offsets from EBP[1].
  - Example: [ebp + 8]
- EBP is called the base pointer or frame pointer because it holds the base address of the stack frame.
- EBP does not change value during the function.
- EBP must be restored to its original value when a function returns.

[1] BP in Real-address mode

RET Instruction

- *Return from subroutine*
- Pops stack into the instruction pointer (EIP or IP). Control transfers to the target address.
- Syntax:
  - RET
  - RET *n*
- Optional operand *n* causes *n* bytes to be added to the stack pointer after EIP (or IP) is assigned a value.

Your turn . . .

- Create a procedure named Difference that uses stack parameters to subtract the first argument from second

```
push 14 ; first argument
push 30  ; second argument
call Difference     ; EAX = 16
Difference PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp + 8]      ; second argument
    sub  eax,[ebp + 12]     ; first argument
    pop  ebp
    ret  8
Difference ENDP
```

Stack Affected by USES Operator

```
MySub1 PROC USES ecx edx
   ret
MySub1 ENDP
```

- **USES operator generates code to save and restore registers:**

```
MySub1 PROC
   push   ecx
   push   edx

   pop edx
   pop ecx
ret
```

What's Next

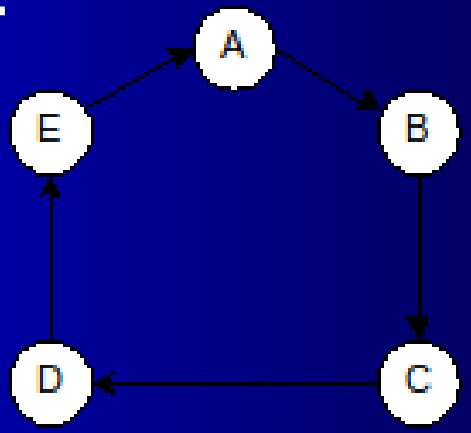
- Stack Frames
- Recursion

Recursion

- What is Recursion?
- Recursively Calculating a Sum
- Calculating a Factorial

What is Recursion?

- The process created when . . .
  - A procedure calls itself
  - Procedure A calls procedure B, which in turn calls procedure A
- Using a graph in which each node is a procedure and each edge is a procedure call, recursion forms a cycle:



Recursively Calculating a Sum

The CalcSum procedure recursively calculates the sum of an array of integers. Receives: ECX = count. Returns: EAX = sum

```
CalcSum PROC
    cmp ecx,0 ; check counter value
    jz L2  ; quit if zero
    add eax,ecx   ; otherwise, add to sum
    dec ecx    ; decrement counter
    call CalcSum  ; recursive call
L2: ret
CalcSum ENDP
```

Stack frame:

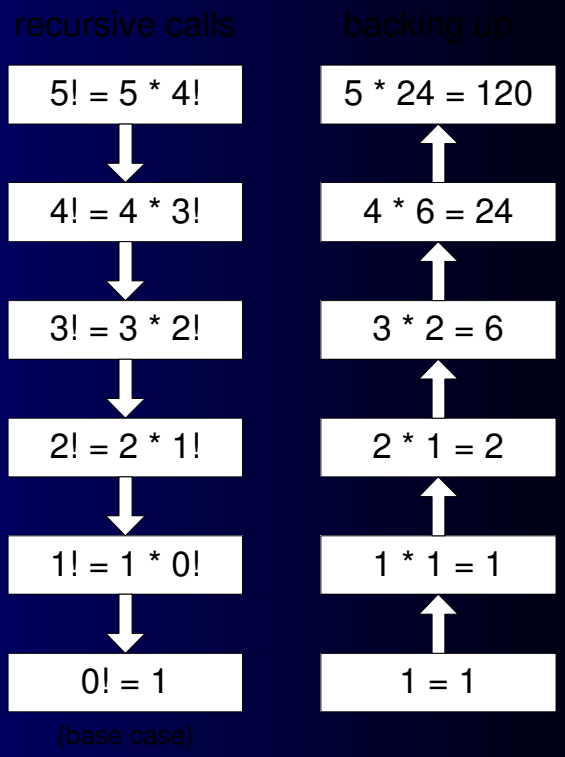
Pushed On Stack	ECX	EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

Calculating a Factorial (1 of 3)

This function calculates the factorial of integer *n*. A new value of *n* is saved in each stack frame:

```
int function factorial(int n)
{
    if(n == 0)
      return 1;
    else
      return n * factorial(n-1);
}
```

As each call instance returns, the product it returns is multiplied by the previous value of n.



Calculating a Factorial (2 of 3)

```
Factorial PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp+8]        ; get n
    cmp  eax,0      ; n < 0?
    ja   L1     ; yes: continue
    mov  eax,1      ; no: return 1
    jmp  L2

L1: dec  eax
    push eax        ; Factorial(n-1)
    call Factorial

; Instructions from this point on execute when each
; recursive call returns.

ReturnFact:
    mov  ebx,[ebp+8]          ; get n
    mul  ebx                ; eax = eax * ebx

L2: pop  ebp        ; return EAX
    ret  4      ; clean up stack
Factorial ENDP
```

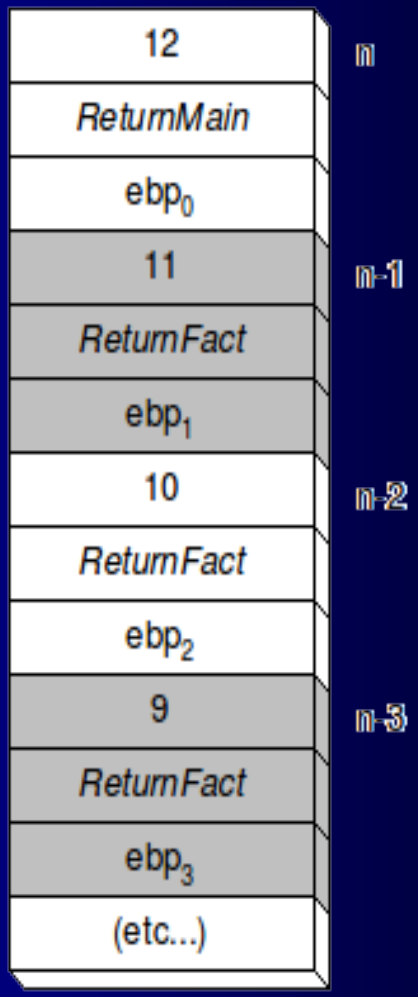
See the program listing

Calculating a Factorial (3 of 3)

Suppose we want to calculate 12!

This diagram shows the first few stack frames created by recursive calls to Factorial

Each recursive call uses 12 bytes of stack space.



What's Next

- Stack Frames
- Recursion
- INVOKE and PROTO

PROTO DIRECTIVE

Creates a procedure prototype

Syntax:

```
procName  PROTO  paramList
```

Uses the same parameter list that appears in procedure

Prototypes are required for …

- Procedures called by INVOKE
- Calling external procedures

Standard configuration:

- PROTO appears at top of the program listing
- Procedure implementation occurs later in the program

PROTO DIRECTIVE

Prototype for the **ArraySum** procedure:

```
ArraySum PROTO,
arrayPtr: PTR DWORD, ; array pointer
arrayLen: DWORD ; array length
```

Prototype for the **swap** procedure:

```
swap PROTO,
ptr1:PTR DWORD, ; 1st int pointer
ptr2:PTR DWORD  ; 2nd int pointer
```

INVOKE DIRECTIVE

INVOKE is a powerful replacement for CALL instruction

Lets you pass multiple arguments

Syntax: **`INVOKE procName paramList`**

`ParamList` is an optional list of procedure arguments

MASM requires that every procedure called by the INVOKE directive to have a prototype

Arguments can be:

Immediate values and integer expressions

Variable names

Addresses

Register names

INVOKE DIRECTIVE

Consider the following procedure prototypes:

```
ArraySum PROTO, arrayPtr:PTR DWORD, arrayLen:DWORD
swap PROTO, ptr1:PTR DWORD, ptr2:PTR DWORD
```

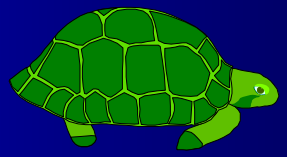
We can use INVOKE to call these procedures:

```
INVOKE ArraySum,ADDR array,ecx
```

```
INVOKE swap, ADDR var1, esi
```

YOUR TASK

Write a procedure in assembly for calculating factorial of a number using INOVOKE and PROTO directives.



53 68 75 72 79 6F