

# Aula 8 - Estudo Avançado de Procedimentos

## Arquitetura de Computadores I

Prof. MSC. Wagner Guimarães Al-Alam

Universidade Federal do Ceará  
Campus de Quixadá

2020-1



# Agenda

- Stack Frames
- Recursão
- Criando Programas em Múltiplos Módulos



# Stack Frames

- Parâmetros de Pilha
- Variáveis Locais



# Stack Frame

- Também conhecido com registro de ativação
- Área da pilha definida a parte antes do endereço de retorno do procedimento, contendo parâmetros passados, registradores salvos e variáveis locais.
- Criada pelos seguintes passos:
  - Chamada para o programa empilhar argumentos na pilha e chamar o procedimento.
  - O procedimento chamado empilha EBP na pilha, e define EBP apontando para ESP.
  - Se variáveis locais são necessárias, uma constante é subtraída de ESP para reservar espaço na pilha.



# Stack Parameters

- Mais conveniente que parâmetros em registradores
- Duas maneiras possíveis de chamar DumpMem. Qual é a mais fácil?

```
1      pushad
2      mov esi, array
3      mov ecx, sizeofArray
4      mov ebx, 4
5      call DumpMem
6      popad
```

```
1      push 4
2      push sizeofArray
3      push array
4      call DumpMem
```

# Passando Argumentos por Valor

- Empilhe os valores dos argumentos na Pilha
  - (Use somente valores 32-bit no modo protegido para manter a pilha alinhada)
- Chame o respectivo procedimento
- Aceite o valor de retorno em EAX
- Remova os argumentos da pilha se o procedimento não os removeu





# Passagem por Referência

- Empilhe os offsets dos argumentos na Pilha
- Chame o procedimento
- Aceite o valor de retorno em EAX
- Remova os argumentos da pilha se o procedimento não os removeu





# Exemplo

```
.data
```

```
val1    DWORD 5
```

```
val2    DWORD 6
```

(offset val2)

(offset val1)

00000004

00000000

← ESP

```
.code
```

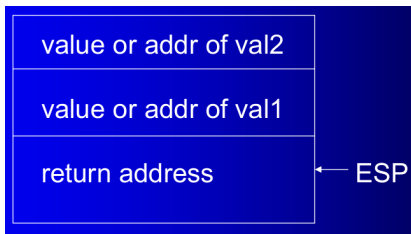
```
push OFFSET val2
```

```
push OFFSET val1
```

Stack prior to CALL



# Pilha Após a Instrução CALL



## Passando um Array por Referência (1 / 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:

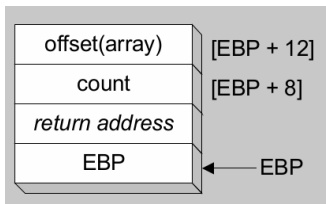
```
1  section .data
2      count equ 100
3  section .bss
4      array RESW count
5  section .text
6  global CMAIN
7  CMAIN:
8      push array
9      push count
10     call ArrayFill
11     xor eax, eax
12     ret
```



## Passing an Array by Reference (2 / 2)

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

```
1  ArrayFill :  
2      push ebp  
3      mov  ebp, esp  
4      pushad  
5      mov  esi, [ebp+12]  
6      mov  ecx, [ebp+8]  
7      .  
8      .
```



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

# Accessing Stack Parameters (C/C++)

- C and C++ functions access stack parameters using constant offsets from EBP<sup>1</sup>.
  - 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.

---

<sup>1</sup>BP em modo Real

# 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.



# Who removes parameters from the stack?

## Caller (C)

```
1  push val2
2  push val1
3  call AddTwo
4  add esp,8
```

...Or...

## Called-procedure (STDCALL):

```
1  AddTwo:
2  push ebp
3  mov ebp, esp
4  mov eax, [ebp+12]
5  add eax, [ebp+8]
6  pop ebp
7  ret 8
```



# Passing 8-bit and 16-bit Arguments

- Cannot push 8-bit values on stack
- Pushing 16-bit operand may cause page fault or ESP alignment problem
  - incompatible with Windows API functions
- Expand smaller arguments into 32-bit values, using MOVZX or MOVSBX:

```
1  section .data
2  charVal DB 'x'
3
4  section .text
5  global CMAIN
6  CMAIN:
7  movzx eax, BYTE [charVal]
8  push eax
9  call Uppercase
10 .
11 .
```





# Saving and Restoring Registers

- Push registers on stack just after assigning ESP to EBP
  - local registers are modified inside the procedure

```
1  MySub:  
2  push ebp  
3  mov ebp, esp  
4  push ecx      ;save local registers  
5  push edx
```



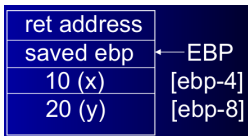
# Local Variables

- Only statements within subroutine can view or modify local variables
- Storage used by local variables is released when subroutine ends
- local variable name can have the same name as a local variable in another function without creating a name clash
- Essential when writing recursive procedures, as well as procedures executed by multiple execution threads



# Creating LOCAL Variables

- Example - create two DWORD local variables:  
Say: int x=10, y=20;



```

1  MySub:
2  push ebp
3  mov  ebp, esp
4  sub  esp, 8           ; create 2 DWORD variables
5
6  mov  DWORD [ebp-4], 10 ; initialize x=10
7  mov  DWORD [ebp-8], 20 ; initialize y=20

```



# LEA Instruction

- LEA returns offsets of direct and indirect operands
- OFFSET operator only returns constant offsets
- LEA required when obtaining offsets of stack parameters & local variables
- Example

```
1 section .data
2     count equ 100
3 section .bss
4     array RESW count
5 section .text
6 global CMAIN
7 CMAIN:
8     mov ebp, esp; for correct debugging
9     mov esi, 0
10    mov eax, array+(esi*4)      ;invalid operand
11    lea eax, [array+(esi*4)]    ;ok
12    xor eax, eax
13    ret
```



# LEA Example

- Suppose you have a Local variable at [ebp-8]
- And you need the address of that local variable in ESI
- You cannot use this:  
mov esi, OFFSET [ebp-8] ; error
- Use this instead:  
lea esi,[ebp-8]



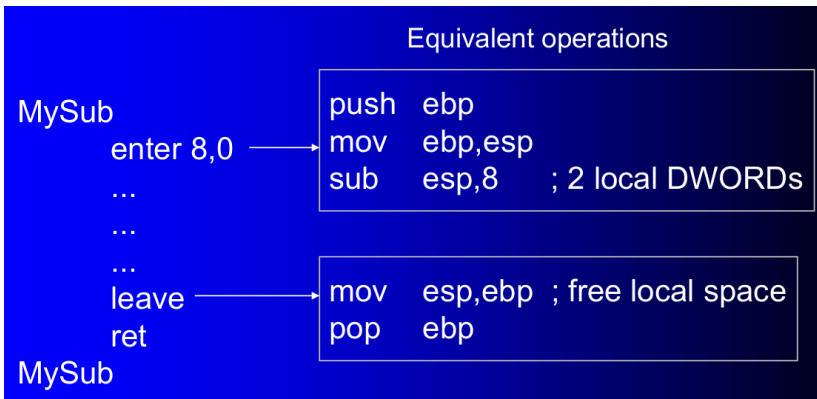
# ENTER Instruction

- ENTER instruction creates stack frame for a called procedure
  - pushes EBP on the stack
  - sets EBP to the base of the stack frame
  - reserves space for local variables
  - Example:  
MySub:  
enter 8,0
  - Equivalent to:  
MySub:  
push ebp  
mov ebp,esp  
sub esp,8



# LEAVE Instruction

- Terminates the stack frame for a procedure.



# Recursão

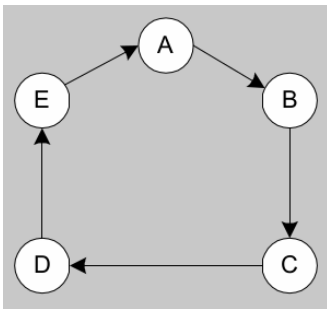
- What is Recursion?
- Recursively Calculating a Sum
- Calculating a Factorial





# 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

```

1  CalcSum :
2      cmp ecx,0          ;check counter value
3      jz  L2             ;quit if zero
4      add eax,ecx        ;otherwise, add to sum
5      dec ecx            ;decrement counter
6      call CalcSum       ;recursive call
7  L2: ret

```

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

Stack Frame



# Calculating a Factorial (1 / 3)

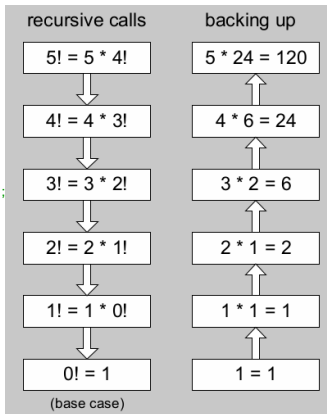
- This function calculates the factorial of integer  $n$ . A new value of  $n$  is saved in each stack frame:

```

1  int function factorial(int n)
2  {
3      if (n == 0)
4          return 1;
5      else
6          return n * factorial(n-1);
7  }

```

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



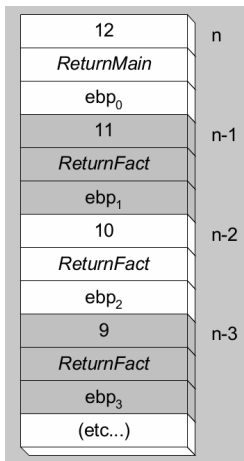
# Calculating a Factorial (2 / 3)

```
1  Factorial:
2  push ebp
3  mov ebp,esp
4  mov eax,[ebp+8]          ;get n
5  cmp eax,0               ;n < 0?
6  ja L1                   ;yes: continue
7  mov eax,1               ;no: return 1
8  jmp L2
9  L1: dec eax
10 push eax                 ;Factorial(n-1)
11 call Factorial
12
13 ; Instructions from this point on execute when each
14 ; recursive call returns.
15 ReturnFact:
16 mov ebx,[ebp+8]          ;get n
17 mul ebx                  ;eax = eax * ebx
18
19 L2: pop ebp               ;return EAX
20 ret 4                    ;clean up stack
```



# Calculating a Factorial (3 / 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.



# Introdução

- Até agora usamos uma IDE (SASM) para criar nossos programas
- No SASM, alguns detalhes foram simplificados
- Iremos agora trabalhar direto na linguagem NASM...



# Compilando o Primeiro Programa em NASM

- Instale o NASM em seu computador e tenha certeza que o **nasm** e o **ld**.
- Escreva no editor de texto o código do slide seguinte e salve como **hello.asm**.
- Tenha certeza que você está no mesmo diretório que salvou **hello.asm**.
- Para montar o programa, escreva `nasm -f elf hello.asm`.
- Se ocorrer algum erro, você será informado. Caso contrário, um arquivo objeto de seu programa, chamado **hello.o**, será criado.
- Para ligar o arquivo objeto e criar um arquivo executável chamado **hello**, escreva  
`ld -m elf_i386 -s -o hello hello.o`.
- Execute o programa escrevendo **./hello**.
- Se ocorreu tudo corretamente, será mostrado **'Hello, world!'** na tela.



# Código Exemplo

```
1  section .text
2  global _start ;deve ser declarado para o ligador (ld)
3
4  _start:        ;indica ao ligador o ponto de entrada
5      mov edx,len      ;comprimento da mensagem
6      mov ecx,msg      ;message a ser escrita
7      mov ebx,1        ;descriptor de arquivo (stdout)
8      mov eax,4        ;numero de chamada de sistema(sys_write)
9      int 0x80         ;call kernel
10
11     mov eax,1        ;numero de chamada de sistema (sys_exit)
12     int 0x80         ;call kernel
13
14  section .data
15  msg db 'Hello, world!', 0xa ;string a ser escrito
16  len equ $ - msg      ;comprimento da string
```



# Criando um Programa em Múltiplos Módulos

- Um programa multi-módulo é um programa cujo código fonte é dividido em arquivos ASM separados.
- Cada arquivo ASM (módulo) é montado em um arquivo objeto diferente.
- Todos arquivos OBJ que pertencem ao mesmo programa devem ser ligados usando o utilitário ligador para gerar o arquivo executável.
- Este processo é chamado ligação estática.



# Vantagens

- Programas grandes são mais fáceis de escrever, manter e depurar quando são escritos em módulos separados.
- Quando uma alteração em uma linha de código é feita, somente um módulo precisa ser montado novamente. Ligar módulos já montados requer pouco tempo.
- Um módulo pode ser um container para código e dados(pensando em orientação a objetos...)
  - encapsulamento: procedimentos e variáveis são automaticamente não visíveis no módulo, ao menos que você declare como público.



# Criando um Programa com Módulos

- Segue alguns passos básicos para criar um programa com módulos:
  - Crie o módulo main.
  - Crie um módulo de código fonte para cada procedimento o conjunto de procedimentos relacionados.
  - Crie um arquivo de include que contém os protótipos dos procedimentos externos (aqueles que são chamados entre módulos).
  - Use a visibilidade GLOBAL para fazer seu protótipo disponível para os demais módulos.
  - Use a diretiva EXTERN para indicar ao montador que o procedimento será passado pelo ligador posteriormente.



# Exemplo - Módulo add

```
1 ;nasm -f elf add.asm
2 global add
3
4 section .data
5
6 section .text
7
8 add:
9     mov     eax, [esp+4]    ; argument 1
10    add     eax, [esp+8]    ; argument 2
11    ret
```

# Exemplo - Usando o Módulo add

```
1      ;nasm -f elf teste.asm
2      extern add
3
4      section .text
5
6      global _start
7
8      _start:
9
10     mov eax, 0
11     mov al, byte [msg]
12     push eax
13     push 1
14     call add
15     mov byte[msg], al
16     mov edx, len      ;message length
17     mov ecx, msg      ;message to write
18     mov ebx, 1        ;file descriptor (stdout)
19     mov eax, 4        ;system call number (sys_write)
20     int 0x80          ;call kernel
21
22     mov eax, 1        ;system call number (sys_exit)
23     int 0x80          ;call kernel
24     ret
25
26     section .data
27     msg db 'Hello , world!', 0xa ;string to be printed
28     len equ $ - msg             ;length of the string
```



# Ligando e Executando

- O ligador do Linux é o **LD**
  - **ld** combina uma quantidade de arquivos objeto, reloca seus dados e os liga as referencias a símbolos.
  - Usualmente, o último passo da compilação de um programa é executar o **ld**.

```
1 ld -m elf_i386 -s -o hello teste.o add.o
```



# Juntando C e Assembly

- Podemos usar um módulo escrito em Assembly, dentro de um programa em C.
- Também podemos usar um módulo escrito em C, dentro de um programa Assembly.
- Podemos incorporar código Assembly dentro de uma função no C (formato AT&T).



# C no Assembly - 1/2

## Ligando com o LD

```
1  int add(int a, int b)
2  {
3  return a + b;
4  }
```





# C no Assembly (LD)- 2/2

## Ligando com o LD

```
1 ;gcc -c -m32 add.c
2 ;nasm -f elf32 start.asm
3 ;ld -m elf_i386 -o start start.o add.o -lc -l /lib/ld-linux.so.2
4
5 extern add
6 extern printf
7 extern exit
8
9 global _start
10
11 section .data
12 format db "%d", 10, 0
13
14 section .text
15
16 _start:
17
18 push 16
19 push 2
20 call add ; add(2, 6)
21 add esp,8
22
23 push eax
24 push format
25 call printf ; printf(format, eax)
26 add esp,8
27
28 push 0
29 call exit ; exit(0)
```



# C no Assembly (GCC)- 2/2 (Alternativa)

## Ligando com o GCC

Repare que aqui usamos o ponto de entrada como **main**

```
1      ;gcc -c -m32 add.c
2      ;nasm -f elf32 main.asm
3      ;gcc -m32 -o main main.o add.o
4
5      extern add
6      extern printf
7
8      global main
9
10     section .data
11     format db "%d", 10, 0
12
13     section .text
14
15     main:
16
17     push 16
18     push 2
19     call add      ; add(2, 6)
20     add esp,8
21
22     push eax
23     push format
24     call printf   ; printf(format, eax)
25     add esp,8
26
27     xor eax, eax
```



# Assembly no C (GCC)- 1/2

```
1  global add
2
3  section .data
4
5  section .text
6
7  add:
8  mov     eax, [esp+4]    ; argument 1
9  add     eax, [esp+8]    ; argument 2
10 ret
```



# Assembly no C (GCC)- 1/2

```
1 //gcc -c -m32 main.c
2 //nasm -f elf32 add.asm
3 //gcc -m32 -o main main.o add.o
4
5 #include <stdio.h>
6
7 int add(int a, int b);
8
9
10 int main(int argc, char **argv){
11     printf("%d\n", add(2, 6));
12     return 0;
13 }
```



# Incorporando Assembly no Código C

```
1  #include <stdio.h>
2
3  int
4  add(int x, int y)
5  {
6      int a = 0;
7      // poderia ser asm() ou __asm__() ...
8      __asm__(
9          // primeiro argumento para edx
10         "movl 8(%ebp), %edx\n"
11         // segundo argumento para eax
12         "movl 12(%ebp), %eax\n"
13         // soma os dois e fica em eax
14         "addl %edx, %eax\n"
15         // movemos para var "a",
16         // se tive-se segunda var como "b" faríamos -8(%ebp)
17         "mov %eax, -4(%ebp)\n"
18     );
19     return a;
20 }
21
22 int
23 main(void)
24 {
25     printf("%d\n", add(3,2));
26     return 0;
27 }
```



# Entendendo as Chamadas de Sistemas

- APIs para interfaceamento entre o espaço de usuário e o espaço do sistema(kernel).
- Já usamos chamadas de sistema anteriormente de forma indireta, no SASM quando usávamos algum GET (*sys\_read*) ou PRINT(*sys\_write*) e *sys\_exit* para sair do programa.

## Leia Mais

Esta seção é uma tradução da página de [https://www.tutorialspoint.com/assembly\\_programming/assembly\\_system\\_calls.htm](https://www.tutorialspoint.com/assembly_programming/assembly_system_calls.htm), consulte o restante do tutorial que é muito interessante e apresentado em uma linguagem simples.



# Chamada de Sistema no Linux

- Para fazer uma chamada de sistema no Linux, siga os seguintes passos:
  - Coloque o número da chamada de sistema no registrador EAX<sup>2</sup>.
  - Armazene os argumentos da chamada de sistema nos registradores EBX, ECX, etc.
  - Chame a interrupção de chamada de sistema no Linux (80h)
  - O resultado geralmente é retornado no registrador EAX.

## Mais sobre argumentos

Existem 6 registradores que podem armazenar os argumentos de uma chamada de sistema, eles são: EBX, ECX, EDX, ESI, EDI, e EBP. Esses registradores armazenam argumentos consecutivos, começando no registrador EBX. Se existirem mais de 6 argumentos, a localização do primeiro argumento deve ser armazenada no registrador EBX.

<sup>2</sup>Para a relação completa dos números digite:

# Exemplo 1 - Saída de um Programa

- Chamada de *sys\_exit*

```
1  mov eax,1      ; system call number (sys_exit)
2  int 0x80        ; call kernel
```





## Exemplo 2 - Escrever na Tela

- Chamada de *sys\_exit*

```
1  mov edx,4      ; message length
2  mov ecx,msg    ; message to write
3  mov ebx,1      ; file descriptor (stdout)
4  mov eax,4      ; system call number (sys_write)
5  int 0x80       ; call kernel
```



# Exemplo Completo

```

1      section .data                                ;Data segment
2      userMsg db 'Please enter a number: ' ;Ask the user to enter a number
3      lenUserMsg equ $-userMsg                ;The length of the message
4      dispMsg db 'You have entered: '
5      lenDispMsg equ $-dispMsg
6
7      section .bss                                ;Uninitialized data
8      num resb 5
9
10     section .text                               ;Code Segment
11     global _start
12
13     _start:                                     ;User prompt
14     mov eax, 4
15     mov ebx, 1
16     mov ecx, userMsg
17     mov edx, lenUserMsg
18     int 80h
19
20     ;Read and store the user input
21     mov eax, 3
22     mov ebx, 2
23     mov ecx, num
24     mov edx, 5                                ;5 bytes (numeric, 1 for sign) of that information
25     int 80h

```



# Exemplo Completo

```
1
2
3 ;Output the message 'The entered number is: '
4 mov eax, 4
5 mov ebx, 1
6 mov ecx, dispMsg
7 mov edx, lenDispMsg
8 int 80h
9
10 ;Output the number entered
11 mov eax, 4
12 mov ebx, 1
13 mov ecx, num
14 mov edx, 5
15 int 80h
16
17 ; Exit code
18 mov eax, 1
19 mov ebx, 0
20 int 80h
```



# Exemplos de Chamadas de Sistema

%eax	Name	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-

