

Lecture 7: Advanced Procedures and Arrays

Table of contents

1	Objectives	2
2	Advanced Procedures	2
2.1	Introduction	2
2.2	Stack Frame	3
2.2.1	Creating and Destroying Stack Frame	6
2.3	32-bit Calling Conventions	10
2.3.1	C Calling Conventions (cdecl)	10
2.3.2	STDCALL Calling Convention	13
2.4	LEA Instruction	14
2.4.1	Flags Affected	16
2.5	ENTER and LEAVE Instructions	16
3	Manipulating Arrays	17
3.1	Repeated String Operations	19
3.2	Examples	19

1 Objectives

In this lecture, you will learn:

1. How to pass arguments to the calling procedure using the run-time stack
2. How to define and release a stack frame.
3. the 32-bit calling conventions.
4. How to manipulate arrays using string primitive instructions.

2 Advanced Procedures

2.1 Introduction

- Consider the following C/C++ program:

```
int getMin (int, int);

int main()
{
    int num1 = 10;
    int num2 = 8;

    int min;

    min = getMin(num1, num2);

    printf("The minimum of %d and %d is %d", num1, num2, min);

    return 0;
}

int getMin (int a, int b)
{
    if (a<b)
        return a;
```

```
else
    return b;
}
```

- The program consists of two functions `main()` and `getMin()`.
- The main function invokes `getMin()` function and **passes** two **arguments**: `num1` and `num2`. Precisely, the main function passes the **values** of `num1` and `num2`.
- In this section, you will learn how to pass arguments to procedures using assembly language. Eventually, you will learn how C/C++ and other languages passes arguments to functions?
- Before we proceed, we need to agree on the following terminology:
 - The `main()` function is the **caller** function/procedure and `getMin()` is the **callee** function.
 - The values of `num1` and `num2` are called **arguments**. The receiving variables `a` and `b` are known as **formal parameters** or simply **parameters**.

2.2 Stack Frame

- Most modern languages push function arguments on the run-time stack before calling functions.
- Functions also use the run-time stack to store their local variables.
- Moreover, the CPU uses the run-time stack to store the return address (to return to the caller function).

Definition

An area in the stack segment that is setting aside to store passed arguments, procedure return address, and local variables is known as a **stack frame** or **activation record**.

- In 32-bit mode, C/C++ functions and Windows API use stack to store passed arguments and local variables. However, In 64-bit mode, the situation is different due to the expanded set of registers (it has 16 GPRs as opposed to 8 GPRs in 32-bit mode) . Most 64-bit compilers try to store arguments and local variables in CPU registers using a technique known as “**spilling the registers**”.
- In this course, we only consider the 32-bit mode.
- In 32-bit mode, a stack frame consists of:
 1. *passed arguments*, if any, are pushed onto the stack
 2. *subroutine return address*, which will be pushed automatically when executing CALL instruction.
 3. *the old EBP register*, which is pushed onto the stack and a new EBP is set equal to ESP.
 4. *local variables*, if any, are pushed onto the stack. You can simply decrement the value of ESP to reserve space for the local variables.
 5. *registers*, that need to be saved, are pushed onto the stack.
- Figure 1 illustrates the main components of a stack frame that should be established when calling a procedure.
- Figure 2 shows a typical stack frame for a procedure. The procedure received three arguments and has two local variables. For simplicity, the size of all variables is doubleword (4-byte).
- The standard way to access passed arguments and local variables inside assembly program is by using base+displacement addressing mode. The base address is EBP register and the displacement is the offset from the base register. For example, the address of the first argument is determined by the expression $EBP+8$. The second argument is located at address $EBP+12$. In similar way, the first local variable is located at address $EBP-4$. Please refer to Figure 2 for the details of memory addresses of the other variables.

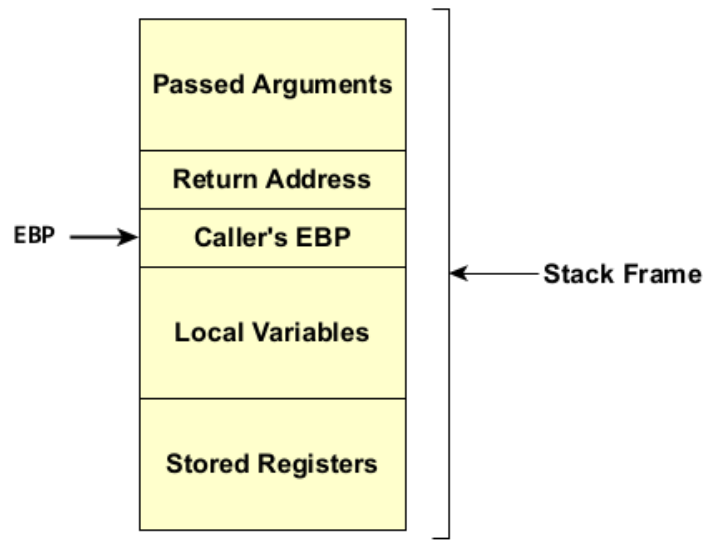


Figure 1

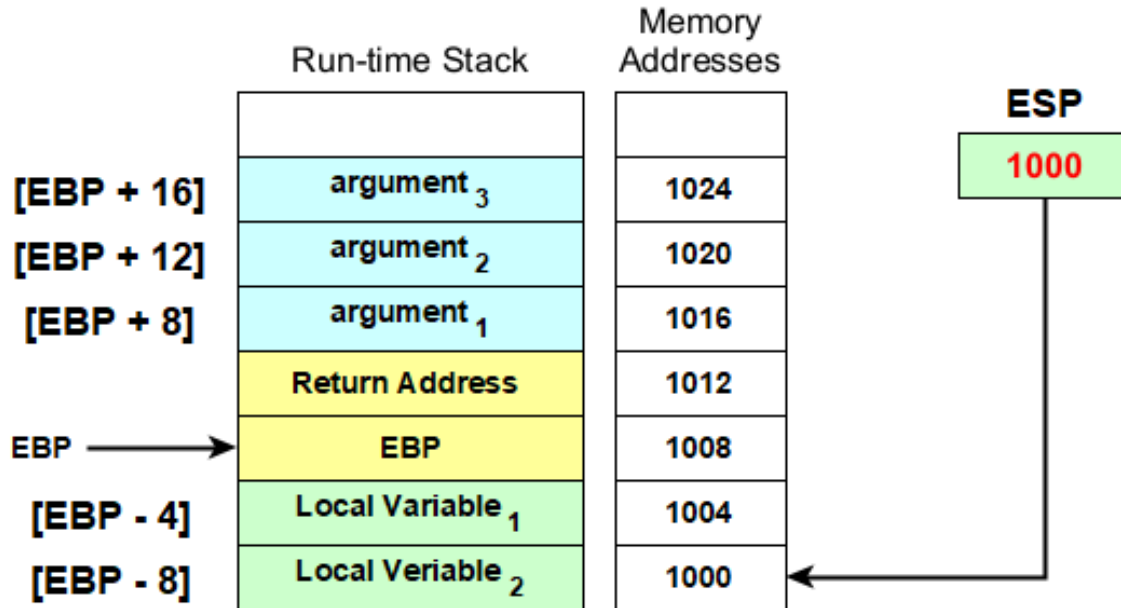


Figure 2

2.2.1 Creating and Destroying Stack Frame

- Here is a typical procedure skeleton that uses a stack frame:

```
push    ebp        ; save EBP in stack
mov     ebp, esp    ; set EBP points to its old value
sub     esp, <n>    ; allocate n bytes for local
                    ; variables
.
.
.    <procedure body>
.
.

mov     esp, ebp    ; delete local variables
pop     ebp        ; restore old EBP value
ret
```

- Procedures begin with a **prologue** consisting of instructions that save the EBP register and point EBP to the top of the stack.
- The end of the procedure consists of an **epilogue** in which local variables are removed, if any, and the EBP register is restored.

Example 2.1 (AddTwo Procedure). The following AddTwo function, written in C, receives two integers as parameters and returns their sum.

```
int AddTwo (int x, int y)
{
    return (x+y);
}
```

Let's create an equivalent procedure in assembly language. Please note that the function has no local variable.

```

AddTwo:
    push    ebp
    mov     ebp, esp

    mov     eax, [ebp + 12] ; second parameter
    add     eax, [ebp + 8]  ; first parameter

    pop     ebp
    ret

```

To make our code more readable, we can define symbolic constants to represent the formal parameters.

```

AddTwo:

#define x_param DWORD [ebp + 8]
#define y_param DWORD [ebp + 12]

    push    ebp
    mov     ebp, esp
    mov     eax, y_param
    add     eax, x_param
    pop     ebp
    ret

```

The main procedure can call AddTwo procedure as follows:

```

    push    6
    push    5
    call    AddTwo
    add     esp, 8

```

Example 2.2 (MinThree Procedure). Consider the following program in C language:

```

int MinThree (int a, int b, int c)
{
    int min = a;

    if (b < min) min = b;
    if (c < min) min = c;

    return min;
}

int main()
{
    printf("Minimum number is %d\n", MinThree(15, 10, 13));

    return 0;
}

```

The equivalent program in assembly language is

```

                global _main
                extern _printf

                section .data
prnt_fmt       db      "Minimum number is %d.", 13, 10, 0

                section .code
MinThree:
%define        a_param DWORD [ebp + 8]
%define        b_param DWORD [ebp + 12]
%define        c_param DWORD [ebp + 16]
%define        min_loc DWORD [ebp - 4]

                ; prologue
                push    ebp
                mov     ebp, esp
                sub     esp, 4                ; allocate memory for local variable

```



```

        ; set a as the min value
        mov     eax, a_param
        mov     min_loc, eax
        ; if b < min => min = b
        mov     eax, b_param
        cmp     eax, min_loc
        jnl     next1
        mov     min_loc, eax
next1:   mov     eax, c_param
        cmp     eax, min_loc
        jnl     next2
        mov     min_loc, eax
next2:   mov     eax, min_loc

        ; epilogue
        mov     esp, ebp          ; destroy local variable
        pop     ebp
        ret

_main:   push    15
        push    10
        push    13
        call    MinThree
        add     esp, 12

        push    eax
        push    prnt_fmt
        call    _printf
        add     esp, 8

        xor     eax, eax
        ret

```

2.3 32-bit Calling Conventions

- Calling conventions can be described as **standardized interface** between the caller and callee procedures. The interface determines:
 - The order in which arguments are passed.
 - How parameters are passed (pushed on the stack, placed in registers, or a mix of both)
 - Which registers the callee function must preserve for the caller (known as **callee-saved registers**)
 - How the task of preparing the stack frame for, and restoring after, a function call is divided between the caller and the callee.

Application Binary Interface (ABI)

An application binary interface (ABI) is an interface between two binary program modules. Basically, the interface consists of calling conventions, type representations and name mangling.

- In this course, we present the two most commonly used calling conventions for 32-bit programming in a Windows environment: **cdecl** and **stdcall**.

2.3.1 C Calling Conventions (cdecl)

- **cdecl** (C declaration) was established by the C programming language, and used by C and C++ programming languages.
- The caller pushes the arguments on the stack in reverse order.
- If the return values are integer values or memory addresses, they are put into the EAX register by the callee.
- The caller cleans the stack after the function call returns.
- The procedures in Example [2.1](#) and Example [2.2](#) are following cdecl convention.

Example 2.3 (cdecl convention). The following program consists of two files: `main.c` and `minthree.asm` files. The `main.c` file has the `main()` function written in C language. The `minthree.asm` file has the `MinThree` function written in assembly language. Since the later function uses cdecl convention, the `main()` function can invoke `MinThree` function without any problem.

FILE: main.c

```
#include <stdio.h>

int MinThree (int, int, int);

int main()
{
    int m = MinThree(3, 2, 1);
    printf("Minimum number is %d\n", m);
}
```

FILE: minthree.asm

```
global _MinThree

section .code

_MinThree:
%define    a_param  DWORD [ebp + 8]
%define    b_param  DWORD [ebp + 12]
%define    c_param  DWORD [ebp + 16]
%define    min_loc   DWORD [ebp - 4]

; prologue
push    ebp
mov     ebp, esp
sub     esp, 4           ; allocate memory for local variable

; set a as the min value
mov     eax, a_param
mov     min_loc, eax
; if b < min => min = b
mov     eax, b_param
cmp     eax, min_loc
jnl     next1
mov     min_loc, eax
next1:   mov     eax, c_param
        cmp     eax, min_loc
        jnl     next2
        mov     min_loc, eax
next2:   mov     eax, min_loc

; epilogue
mov     esp, ebp        ; destroy local variable
pop     ebp
ret
```

- To compile the above files using 32-bit GCC, type:

```
nasm -fwin32 minthree.asm
gcc -c main.c
gcc main.o minthree.obj -o main
```

- The first command, if success, generates `minthree.obj` object file
- The second command, if success, generates `main.o` object file
- The third command, if success, links two object files and generates an executable file named `main.exe`.

2.3.2 STDCALL Calling Convention

- `stdcall` is the standard calling convention for the Microsoft Win32 API.
- The caller pushes the arguments on the stack in reverse order.
- If the return values are integer values or memory addresses, they are put into the `EAX` register by the callee.
- The callee cleans the stack before the function call returns.

Example 2.4 (AddTwo Procedure).

AddTwo:

```
%define x_param DWORD [ebp + 8]
%define y_param DWORD [ebp + 12]

    push    ebp
    mov     ebp, esp
    mov     eax, y_param
    add     eax, x_param
    pop     ebp
    ret     8           ; clean up the stack
```

2.4 LEA Instruction

Syntax

```
LEA    <destination>, <source>
```

- This instruction computes the effective address of the source operand and stores it in the destination operand.

i Rules

- The source operand must be a memory address specified with one of the processors addressing modes
- The destination operand is general-purpose register.

- The valid forms are:

```
LEA    reg16, mem
```

```
LEA    reg32, mem
```

- The source operand can be in:
 - base addressing mode
 - base+displacement addressing mode
 - index addressing mode
 - base+index addressing mode
 - base+index+displacement addressing mode
- This instruction is quite useful to get the address of a stack parameter.

Example 2.5 (Initialize an array). The following code segment, which is inside a function, declares a local array of characters named `myString` and set each element in the array to character `'*'`, as listed below:

```

void formatArray ()
{
    char myString[30];
    int i;

    for (i=0; i<30; i++)
        myString[i] = '*';
    .
    .
    .
}

```

The equivalent code in assembly language is listed below:

```

formatArray:
    push    ebp
    mov     ebp, esp
    sub     esp, 32      ; allocate 32 bytes

    lea     esi, [ebp - 32]
    mov     ecx, 30
top:    mov     BYTE [esi], '*'
    inc     esi
    loop    top

    .
    .
    .

```

Remarks

1. Although the array size is only 30 bytes, the ESP register is decremented by 32 to keep it **aligned** on a doubleword boundary.
2. Notice how LEA instruction is used to assign array's address to ESI, which is equivalent to assign ESI to the value of $EBP - 32$.

3. Without LEA, we can let ESI points to the array using the following code:

```
mov     esi, ebp
sub     esi, 32
```

4. LEA is more efficient than the above code, since these two instructions can be executed by LEA in one CPU cycle or less.

2.4.1 Flags Affected

- None.

2.5 ENTER and LEAVE Instructions

Syntax

```
ENTER    <numbytes>, <nestinglevel>
```

```
LEAVE
```

- The ENTER instruction automatically creates a stack frame for a called procedure.
- It performs three actions:
 1. Pushes EBP onto the stack (`push ebp`)
 2. Set EBP to the base of the stack frame (`mov ebp, esp`)
 3. Reserve space for local variables (`sub esp, numbytes`)
- Both operands must be immediate values. **Numbytes** specifies the number of bytes of stack space to reserve for local variables. **Nestinglevel** specifies the lexical nesting level of the procedure (read Intel document).

- **Numbytes** is always roundup to a multiple of 4 to keep ESP on adouble-word bounday.
- The LEAVE instruction terminates the stack frame for a procedure.
 - It reverse the action of a previous ENTER instruction by:
 1. restoring ESP (mov esp, ebp), and
 2. restoring EBP to the value they were assigned when the procedure was called (pop ebp).

Example 2.6 (Revisit AddTwo Procedure).

```
AddTwo:
%define x_param DWORD [ebp + 8]
%define y_param DWORD [ebp + 12]

    enter    0, 0
    mov     eax, y_param
    add     eax, x_param
    leave
    ret
```

3 Manipulating Arrays

- The IA-32 has FIVE groups of instructions for processing arrays of bytes, words, and doublewords. Theses groups are known as **string primitives**.

Instruction Group	Description
MOVSx	<i>Move string</i> : copy data from memory addressed by ESI to memory addressed by EDI.

Instruction Group	Description
CMPSx	<i>Compare strings</i> : compare the contents of two memory locations addressed by ESI and EDI. It implicitly performs $[ESI] - [EDI]$
SCASx	<i>Scan string</i> : compare the accumulator register (AL, AX or EAX) to the contents of memory addressed by EDI. (i.e., $Accumulator - [EDI]$)
STOSx	<i>Store string</i> : store the accumulator register content into memory addressed by EDI.
LODSx	<i>Load string</i> : Load memory addressed by ESI into the accumulator register.

- The symbol x can be B for byte, W for word, D for doubleword, or Q for quadword (64-bit).
- Each string primitive instruction implicitly uses ESI, EDI, or both registers to address memory.
- References to the accumulator register imply the use of AL, AX, or EAX, depending on the instruction data size.
- After performing string primitive operation, the source index (ESI) and/or the destination index (EDI) are/is moved to the next memory address based on the CPU Direction flag (DF).
- If $DF = 0$, ESI or/and EDI are incremented. Otherwise, ESI or/and EDI are decremented.
- The instructions CLD and STD clear or set DF flag, respectively.
- String primitives can be repeated using a **Repeat Prefix** (explained next).

3.1 Repeated String Operations

- Each of string primitives performs one iteration of string operation.
- To operate on arrays, the string primitives can be combined with a repeat prefix (REP) to create a repeating instruction or be placed in a loop.
- The number of repetitions is determined by ECX register.
- The following repeated prefixes can be used in conjunction with a count in the ECX register to cause a string instruction to repeat:
 - REP: Repeat while ECX \neq 0.
 - REPE/REPZ: Repeat while ECX \neq 0 and ZF == 1.
 - REPNE/REPNZ: Repeat while ECX \neq 0 and ZF \neq 0.

3.2 Examples

Example 3.1 (Initialize an array). Given an array of short integers named arrX. We need to initialize all its elements to zero using string primitive:

```
cld                ; clear DF flag
mov     edi, arrX   ; destination
mov     ax, 0       ; initializer value
mov     ecx, 10     ; # of elements in arrX
rep     stosw       ; store string
```

Example 3.2 (Copy Arrays). In this example, let us translate the following for loop into assembly language using string primitives.

```
int x[100];
int y[100];
.
.
```

```
for (i=0; i<100; i++)
    y[i] = x[i];
```

The equivalent code in assembly language without using string primitive:

```

    mov     esi, x        ; the source array
    mov     edi, y        ; the destination array
    mov     ecx, 100      ; counter
top:  mov     eax, [esi]    ;
    mov     [edi], eax     ; y[i] = x[i]
    add     esi, 4         ; move to next elm in x
    add     edi, 4         ; move to next elm in y
    loop    top           ; repeat
```

Now, let us write the same code using string primitive:

```

    cld                     ; clear direction flag
    mov     esi, x          ; ESI points to array x (source)
    mov     edi, y          ; EDI points to array y (target)
    mov     ecx, 100        ; repeat 100 times
    rep     movsd           ; copy string doublewords
```

Example 3.3 (Finding a target element). In this example, let us translate the following for loop into assembly language using string primitives.

```

int x[100];
int target = 8;
int found = 0;
.
.
for (i=0; i<100; i++)
    if (target == x[i]) {
        found = 1;
        break;
```

```

    }
if (found)
    printf("Target is found\n");
else
    printf("Target is not found");

```

The equivalent for-loop in assembly language without using string primitive:

```

        mov     esi, x        ; the source array
        mov     eax, 15       ; the target value
        mov     ecx, 100      ; counter
top:    cmp     eax, [esi]     ; compare
        je      BREAK        ; exit loop
        add     esi, 4        ; move to next elm in x
        loop    top          ; repeat
        .
        .

```

Now, let us write the whole code using string primitive:

```

section .data
arrX      dd      10, 13, 18, 14, 20, 11, 17, 18, 14, 12
ARRSIZE   EQU     ($-arrX)/4
fnd_msg    db      "Target is found", 13, 10, 0
not_fnd_msg db      "Target is not found", 13, 10, 0

section .code
        .
        .

        cld                      ; clear DF flag
        mov     EDI, arrX        ; source array
        mov     EAX, 15          ; target
        mov     ecx, ARRSIZE     ; counter
        repne   scasd

```

```
je FOUND
; not found
push not_fnd_msg
jmp PRINT
FOUND: push fnd_msg
PRINT: call _printf
add esp, 4
.
.
```