

# Stack Arguments & Locals



- Until now,
  - We have been passing arguments in registers
  - ▶ We have only used global variables (.data)
- Now,
  - We will **pass arguments** on the stack (you did this in Lab 4)
  - We will learn how to **store** *local variables* on the stack

#### Stack Frames



- We have used the runtime stack to
  - ▶ save the address a procedure should return to (CALL)
  - save register values inside a procedure (PUSH)
- So, there may be several elements on the stack all relating to the same procedure call
  - ▶ E.g., return address, saved register values
- Collectively, the part of the stack containing the return address, saved registers, etc. for a procedure call is called a *stack frame* or *activation record* 
  - Stack frames will also contain arguments, local variables

# Stack Frames



```
main PROC
     call A
     exit
main ENDP
A PROC
     push eax
     push ebx
     call B
     pop ebx
                                                    Return address in main
     pop eax
                                                                                 Stack Frame
     ret
                                                     Saved value of EAX
                                                                                 for call to A
A ENDP
                                                     Saved value of EBX
                                                                                Stack Frame
                                                     Return address in A
                at this point, the stack contains:
B PROC
                                                                                for call to B
     ret
B ENDP
```

#### Stack Arguments



- ▶ Procedures written in high-level languages (e.g., C/C++) receive arguments *on the stack*, not in registers
- Arguments are conventionally pushed from *right to left*
- In C:

```
AverageOf3(10, 20, 30);
```

In assembly:

```
push 30
push 20
push 10
call AverageOf3
```

#### **Local Variables**



- ▶ *Local variables* (AKA "locals" or "temporary variables") are created "fresh" each time a procedure is invoked and disappear when the function returns
- Local variables are especially important when implementing recursive procedures (What would happen below if sum were a global variable?)

```
int BadSum(unsigned int n) {
  int sum;
  sum = n;
  if (n > 0) sum += BadSum(n-1);
  return sum;
}
```

- You do **not** need local variables for the recursive procedure in Homework 4. If your procedure stores values in registers and pushes/pops them, then registers are a lot like local variables.
- Local variables are especially useful when you run out of registers

# Creating a Stack Frame



- ▶ At the call site (i.e., inside the calling function)...
- 1. Push arguments onto the stack
- 2. Call the subroutine (CALL pushes the return address)
- ▶ Inside the function being called...
- 3. Push EBP (it will be used to retrieve the arguments)
- 4. Set EBP equal to ESP
- 5. Decrement ESP to allocate stack storage for locals
- 6. Save register values by pushing them on the stack

# Creating a Stack Frame



After the stack frame is created...

les	Argument 3	
Callee Pushes —— Caller Pushes	Argument 2	EBP contains the address of this stack element
	Argument 1	
	Return address	
	Saved value of EBP	
	Local variable 2	
	Local variable 1	
	Saved register 1	
	Saved register 2	
	Saved register 3	ESP contains the address of this stack element
		June Cronnelle

# Accessing Args & Locals



∟sə∟	Argument 3	EBP+16
-Caller Pushes	Argument 2	EBP+12
	Argument 1	EBP+8
ပ္မ	Return address	
	Saved value of EBP	<b>←</b> EBP
Callee Pushes	Local variable 2	EBP-4
	Local variable 1	EBP-8
llee	Saved register 1	
ပ္ပ	Saved register 2	
L	Saved register 3	<b>←</b> ESP

- ▶ Three 32-bit arguments are at
  - [ebp+8]
  - [ebp+12]
  - [ebp+16]
- ▶ Two 32-bit local variables are at
  - [ebp-4]
  - [ebp-8]

#### Terminating a Stack Frame



- ▶ Inside the function being called...
- 1. If the function returns a value, put it in EAX
- 2. Pop register values off the stack
- 3. Set ESP equal to EBP to remove local variables
- 4. Pop EBP
- 5. Return (RET pushes the return address); if using the *STDCALL calling convention*, remove arguments by supplying an immediate operand to the RET instruction
- **▶** Back in the calling function...
- 6. If using the C calling convention, remove arguments

#### Calling Convention



- A *calling convention* specifies how a procedure receives parameters and returns a result
  - ▶ How are arguments passed to the procedure: in registers or on the stack?
  - In what order are arguments pushed on the stack?
  - ▶ Who removes arguments from the stack?
  - What other steps are taken by the caller vs. callee before and after the function executes?
  - ▶ What registers may be overwritten by the callee?

Source: http://en.wikipedia.org/wiki/Calling convention

#### STDCALL Calling Convention



- Arguments are pushed from right to left
- ▶ The procedure issues a RET instruction with an immediate operand to remove arguments
- ▶ The instruction ret 8 means "pop the return address and set EIP, then add 8 to ESP to remove arguments"

```
push 6
push 5
push ebp
call AddTwo
mov ebp, esp
mov eax, [ebp+12]
add eax, [ebp+8]
pop ebp
ret 8
AddTwo ENDP
AddTwo ENDP
```

#### C Calling Convention



- Arguments are pushed from right to left
- After the CALL instruction, the caller adds a value to ESP to remove arguments from the stack

#### Which Calling Convention?



- ▶ The C calling convention allows functions with a variable number of arguments, like *printf* 
  - printf("OK"); printf("%d%s\n", 3, OK);
  - With STDCALL, every procedure must have a fixed number of arguments, since the function must supply an immediate value to the RET instruction
- ▶ But with the C calling convention, you *must* remember to clean up the stack after every CALL!
- Windows API functions use STDCALL
- **▶** We will use STDCALL in the future

#### Calling Conventions There are more Developer Network calling conventions Downloads that we won't use Collapse All Export (0) Print E.g., FASTCALL: like MSDN Library Argument Passing and Development Tools and Languages STDCALL but first two args in ECX and EDX, Naming Conventions ▶ Visual Studio 2013 not on the stack ▶ Visual C++ Visual Studio 2013 | Other Versions → | 1 out of 1 rated this helpful - Rate this topic ▶ C++ Language and Standard Libraries Microsoft Specific ▷ C/C++ Languages ▶ C++ Language Reference The Visual C++ compilers allow you to specify conventions for passing arguments Microsoft-Specific Modifiers and return values between functions and callers. Not all conventions are available on all supported platforms, and some conventions use platform-specific implementations. In most cases, keywords or compiler switches that specify an Calling Conventions Argument Passing and Naming Conventions unsupported convention on a particular platform are ignored, and the platform default convention is used. \_\_cdecl \_\_clrcall On x86 plaftorms, all arguments are widened to 32 bits when they are passed. \_\_stdcall Return values are also widened to 32 bits and returned in the EAX register, except for 8-byte structures, which are returned in the EDX:EAX register pair. Larger structures are returned in the EAX register as pointers to hidden return structures. \_\_fastcall \_\_thiscall Parameters are pushed onto the stack from right to left. Structures that are not PODs will not be returned in registers. vectorcall

# Prologue & Epilogue



Inside a procedure...

http://msdn.microsoft.com/en-us/library/984x0h58.aspx

- The procedure's *prologue* consists of the instructions to push EBP, set it equal to ESP, reserve space for locals, and save registers
- The procedure's *epilogue* consists of the instructions to pop registers, remove local variables, restore EBP, and return to the caller

# Symbols for Args & Locals



▶ For better readability, define symbolic constants for parameters and local variables inside the procedure using the EQU directive

```
AddTwo PROC

arg1 EQU DWORD PTR [ebp+8]
arg2 EQU DWORD PTR [ebp+12]
push ebp
mov ebp, esp

mov eax, arg1 ; Same as mov eax, DWORD PTR [ebp+8]
add eax, arg2 ; Same as add eax, DWORD PTR [ebp+12]

pop ebp
ret 8

AddTwo ENDP
```



Activity 11