Cleaning up the Stack

To remove parameters from the stack when a subroutine returns, the subroutine must perform stack cleanup. Stack cleanup is the process of removing the subroutine's parameters from the stack so that the stack can be used by other subroutines.

There are two ways to perform stack cleanup:

Explicit stack cleanup: The subroutine explicitly removes its parameters from the stack using the POP instruction. This is done by popping the parameters off the stack in reverse order from which they were pushed.

Implicit stack cleanup: The subroutine leaves the stack cleanup to

the caller. This is done by using a CALL instruction that specifies the number of bytes to be removed from the stack when the subroutine returns.

The following example shows how to perform **explicit stack cleanup** in the AddTwo subroutine:

```
273 AddTwo PROC
274   push ebp
275   mov ebp, esp
276   ; ...
277   ; Calculate the sum of the two parameters.
278   ; ...
279   pop ebp
280   ret
281 AddTwo ENDP
```

The pop ebp instruction at the end of the subroutine removes the base pointer register (EBP) from the stack.

This is done to restore the original value of EBP, which was pushed onto the stack at the beginning of the subroutine.

The following example shows how to use **implicit stack cleanup** in the AddTwo subroutine:

```
289 AddTwo PROC
290   push ebp
291   mov ebp, esp
292   ; ...
293   ; Calculate the sum of the two parameters.
294   ; ...
295   ret 8
296 AddTwo ENDP
```

The ret 8 instruction at the end of the subroutine tells the caller to remove 8 bytes from the stack when the subroutine returns.

This is the same as the size of the two parameters that were pushed onto the stack at the beginning of the subroutine.

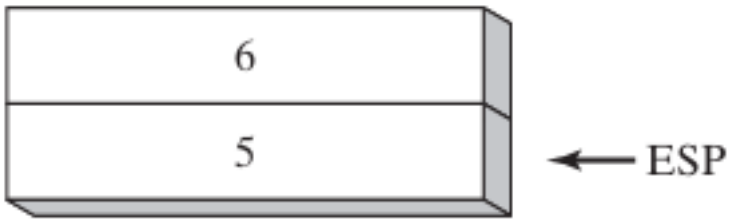
==========

Stack OverFlow

==========

Assuming that AddTwo leaves the two parameters on the stack, the following illustration shows the stack after returning from the call:

This image shows the stack after the call AddTwo instruction in main has been executed:



Inside main, we might try to ignore the problem and hope that the program terminates normally. But if we were to call AddTwo from a loop, the stack could overflow.

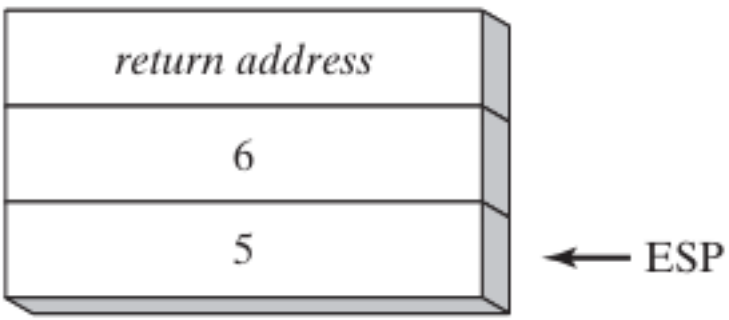
Each call uses 12 bytes of stack space - 4 bytes for each parameter,

plus 4 bytes for the CALL instruction's return address. A more serious problem could result if we called Example1 from main, which in turn calls AddTwo:

```
319 main PROC
320   call Example1
321   exit
322 main ENDP
323
324 Example1 PROC
325   push 6
326   push 5
327   call AddTwo
328   ret                    ;stack is corrupted
329 Example1 ENDP
```

In the image below, the return address for the **call AddTwo** instruction is still on the stack. This is because the AddTwo subroutine did not perform any stack cleanup.

When the RET instruction in Example1 is about to execute, ESP points to the integer 5 rather than the return address that would take it back to main:



The RET instruction loads the value 5 into the instruction pointer and attempts to transfer control to memory address of 5.

Assuming that this address is outside the program's code boundary, the processor issues a **runtime exception**, which tells the OS to terminate the program.

Are these notes well organized??

Yes, your notes are in good order and well explained. You have clearly explained the problem of stack corruption and how to prevent it. You have also provided a good example of how stack corruption can occur.

Stack overflow: Stack overflow occurs when the stack uses more memory than is available. This can happen if a subroutine leaves its parameters on the stack when it returns.

Stack corruption: Stack corruption occurs when the stack is not used correctly. This can happen if a subroutine does not perform stack cleanup or if a subroutine accesses the stack incorrectly.

Preventing stack corruption: To prevent stack corruption, subroutines must perform stack cleanup when they return. This can be done

explicitly or implicitly. Your example of Example1 is a good example of how stack corruption can occur.

If the Example1 subroutine does not perform stack cleanup, the stack will be corrupted when the ret instruction is executed. This will cause the program to terminate with a runtime exception.