COM S 414 Operating Systems Laboratory
Summer 2004

Setjmp/Longjmp Solutions

Part A

See handout.

Part B

1) (d) I don't like 1 very much, but 17 is my favorite number
Look at the jmp_buf structure that was used to restore the register values when longjmp was called. (We know that the structure can be found at address 00430240 based on the disassembly given below the longjmp macro call within current_function) The first word in the structure tells us that the restored stack frame (ebp) is set to address 0044fe38. Visual C++ .NET always leaves one 32-bit entry blank between ebp and the first variable. Two 32-bit entries are placed between each additional variable. Given this information, we know that the first variable value can be found at [ebp-8]. If we look at address 0044fe30 we see that the value there is 00000011 hexadecimal which is equal to 17 in decimal. The longjmp return value can never be 0. It is automatically set to a value of 1 if that would be the case.

2) (c) [ebp+12]. To see why, look back at your notes on the stack calling conventions used on the Intel architecture. When a new function is called, the first thing that happens is that all function parameters are pushed onto the stack beginning with the n'th paramter and ending with the first. The second thing that happens is that the instruction pointer (eip) in the current function is pushed onto the stack. Execution begins in the new function. Convention dictates that the function header will begin by pushing the old stack frame pointer (ebp) onto the stack. Finally, the new stack frame is set equal to the current top of the stack (the esp value). So, to summarize, the first function parameter was pushed onto the stack followed by the old function's return address followed by the old stack frame. Since esp always points to the address of the last thing pushed (rather than the next location to be pushed to) we only need to add 12 bytes to move three 32-bit positions up the stack to get to that second parameter.

3) (e) [ebp-8]. Local variables are placed from lowest to highest below the stack frame pointer (ebp). Since x is the one (and only) variable in the function it will be stored at ebp-8. The two assembly statements will not make any difference in this case.

4) (e) eax,ecx,edx,eflags are the only non-segment registers not saved by setjmp. It wouldn't do us any good to save the value of eax, though, since it is used to store the return value from longjmp. Hence, the answer is ecx, edx, and eflags.

5) (f) The first statement is not correct. Different processes may have different values in the segment registers and the operating system will not allow one process to jump into or otherwise execute/modify code in another process using the setjmp/longjmp library functions on their own. The second statement is also false. longjmp will restore local variables to their state at the time setjmp was called if they are register variables. The last statement is true. We can use setjmp/longjmp as a type of context switch between user-level threads. This is possible because user-level threads share the same code, data, and memory access permissions. You will be creating your own context-switching code based on this idea in Project 2.

6) setjmp only saves (and longjmp only restores) register values. Local variables and function parameters are stored on the stack. That is why the local variables v1, v2, and v3 are not changed by the call to longjmp. Under some circumstances, the Visual C++ compiler could choose to place one or more of the local variables into registers. If that happened then a call to longjmp *might* restore the value of a variable to its state when setjmp was called. (Might is used here because the variable will only be restored to its previous state if it is stored in a register that setjmp saves.)