COM S 414 Operating Systems Laboratory
Summer 2004

Introduction

The goal of this laboratory is to give you an opportunity to examine and ultimately understand the C library functions setjmp and longjmp. In so doing, you will explore a number of important systems components by disassembling and then commenting the assembly instructions in a simple C program using Visual C++ in Windows NT. At the end of this laboratory you will have a basic understanding of the Intel 80x86 architecture and the C programming language.

Laboratory Description

You are going to document the assembly instructions generated by Visual C++ for setjmp.c. To do this you should copy the code disassembly window from Visual C++ corresponding to setjmp and longjmp. (That is, you should copy the _setjmp3 and _longjmp code into a text editor. Be sure that all disassembly view options are turned on except symbol names and code bytes. See the hint section for an explanation on why you are copying _setjmp3 and _longjmp rather than setjmp and longjmp.) You should comment the assembly statements in the two functions on line-by-line basis. Be sure to add comments for the individual assembly instructions coupled with a broader description of what blocks of code are doing. Finally, answer the questions in Part B of this handout.

How to Get Started

- Download the code at: www.cs.cornell.edu/cs415/f02/setjmp.zip
- Open Microsoft Visual Studio .NET C++.
- Unzip the "setjmp.zip" file in the directory for your project.
- Open the Visual Studio .NET command prompt and switch to your project directory.
- ONLY DO THIS STEP IF YOU ARE NOT WORKING IN THE UNDERGRADUATE LAB. If the computer you are using does not have Visual Studio .NET installed in the usual place (`c:\Program Files\Microsoft Visual Studio`), then change the line

  ```
  VISUALSTUDIO = c:\Program Files\Microsoft Visual Studio .NET
  ```

  in "`Makefile`" to point to the correct location.
- Run “nmake /f makefile” from the .NET prompt to build setjmp.exe. It should compile and link with a few warnings but no errors. If you execute setjmp.exe it will print out a number of variable names and their values.
- Open up Visual Studio .NET.
- Choose File->Open->Project and select your setjmp.exe file
- Read through the Visual C++ help file entries for setjmp and longjmp. They are included here, in the laboratory handout, for your convenience. Even so, you should

practice using the help system if you are not already familiar with its operation. Experiment by entering the Help->Search section of Visual Studio. Type in a C library function (like setjmp, printf, scanf, or whatever your favorite C library call happens to be) and read through the corresponding entries.

- Take a look at the source code for setjmp.c. You will see that it is a very simple program. Variables labeled v1, v2, and v3 are initialized with a series of values. Setjmp is called to save state information in a jmp_buf data stucture entitled mark. Setjmp always returns 0 when it called upon to store state information and returns a non-zero value when it is jumped to through a longjmp system call. Variable values are modified and printed next. Longjmp is called which returns us directly to the previous setjmp call where we print the final state of the v1, v2, and v3 values and then the program exits.
- Start the Visual C++ debugger by selecting Debug->Step Into.
- You will be prompted to create a setjmp text solution file. You can safely click the save button here and ignore the resulting setjmp.sln.
- Choose Debug->Windows->Disassembly. You will see the 80x86 assembly instructions that correspond to the setjmp.c program complete with all library files that are called from setjmp.c either directly or indirectly. Assembly instructions appear in gray prefixed by their addresses in memory. The corresponding lines of C code appear in black and are prefixed by their line numbers if any. The current state of the user-accessible registers can be viewed through Debug->Windows->Registers. All disassembly viewing options should be on except symbol names and code bytes. Symbol names must be off or you will not see the stack manipulation that you need to become familiar with. Code bytes are harmless, but are not necessary for us. You can change your viewing options by right clicking on the disassembly window.
- You can step through the individual assembly instructions using F10 to step over function calls and F11 to step into them. You can safely ignore library function calls other than setjmp, longjmp, and library functions called by setjmp and longjmp, but stepping through them may help you to get a feel for how the Visual C++ compiler has converted C instructions into assembly code and how the stack frame, local variables, and parameter passing works.
- Next, cut and paste the disassembly for the _setjmp3 and _longjmp functions into a text file and start to think about how to comment the assembly code for those two functions ONLY. No other functions need to be commented, but calls to other functions and the recording of return values from those other functions do need to be commented.
- The Registration, TryLevel, UnwindFunc, and UnwindData sections of the setjmp jmp_buf structure are there to handle C++ exception code properly (see the hint section for more detail). Similarly, global_unwind2, rt_probe_read4, local_unwind2, and NLG_notify exist for the same reason. You can label individual assembly instructions surrounding this code, but you are not responsible for understanding what it does in a global sense. That information is beyond the scope of the course.
- Once you have a good understanding of how things work you can begin commenting the assembly code in earnest. Comment every assembly line. You should also indicate what clumps of instructions are used. Commented code should look something like this example:

//////////////////
Call the printf function passing passing in v1 and a text string as the arguments
Put the first local variable (located one word below the stack frame) into eax
mov eax, dword ptr [ebp-4]
push that value onto the stack
push eax
push the offset of our printf string onto the stack (memory location 0041314c)
push offset string “v1=%d\n” (0041314c)
** add 8 bytes to esp (top of stack pointer) so that it is in its previous state**
add esp,8
//////////////////

A Quick Primer on Intel Architecture Assembly Instructions

Here is a description of mov(e), one of the instructions that you will see when you disassemble setjmp and longjmp. It is here to give you a feeling for the structure of the Intel instruction set. The second Intel Architecture manual (available on the course web page). gives a complete description of this (and all other) assembly instructions. It should be your primary reference throughout this laboratory. An understand of the Intel calling convention and stack operation is also important. It is covered in depth in the first Intel Architecture manual (also available on the course web page).

mov x,y

Copy the value of y into x. The x and y arguments can be registers (eax, ebx, . . .), memory locations, or constants. Samples of these three forms are mov eax,ebx (set eax equal to the value in ebx), mov ebx, [esi+4] (copy the memory location at four more than the value of esi and copy it into ebx), or mov ecx, 4 (place the number 4 into ecx). You will frequently see a qualifier like *dword ptr* in front of a memory address to signify that there is a 32-bit word at that location. Segment qualifiers may appear in memory arguments if needed. For example, mov ecx,dword ptr es:[eax] will load the 32-bit word located at the address value in eax into the ecx register.

Setjmp Description (Taken from the Visual C++ Help system)
Saves the current state of the program.
int setjmp(jmp_buf *env*);

Routine	Required Header	Compatibility
setjmp	<setjmp.h>	ANSI, Win 95, Win NT

Return Value
setjmp returns 0 after saving the stack environment. If setjmp returns as a result of a longjmp call, it returns the *value* argument of longjmp, or if the *value* argument of longjmp is 0, setjmp returns 1. There is no error return.
Parameter
env
Variable in which environment is stored
Remarks
The setjmp function saves a stack environment, which you can subsequently restore using longjmp. When used together, setjmp and longjmp provide a way to execute a "non-local goto." They are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal calling or return conventions.

A call to setjmp saves the current stack environment in *env*. A subsequent call to longjmp restores the saved environment and returns control to the point just after the corresponding setjmp call. All variables (except register variables) accessible to the routine receiving control contain the values they had when longjmp was called.

setjmp and longjmp do not support C++ object semantics. In C++ programs, use the C++ exception-handling mechanism.

Longjmp Description (Also Taken from the Visual C++ Help system)
Restores stack environment and execution locale.
void longjmp(jmp_buf *env*, int *value*);

Routine	Required Header	Compatibility
longjmp	<setjmp.h>	ANSI, Win 95, Win NT

Return Value
None
Parameters
env
Variable in which environment is stored
value
Value to be returned to setjmp call
Remarks
The longjmp function restores a stack environment and execution locale previously saved in *env* by setjmp. setjmp and longjmp provide a way to execute a nonlocal goto; they are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal call and return conventions.
A call to setjmp causes the current stack environment to be saved in *env*. A subsequent call to longjmp restores the saved environment and returns control to the point immediately following the corresponding setjmp call. Execution resumes as if *value* had just been returned by the setjmp call. The values of all variables (except register variables) that are accessible to the routine receiving control contain the values they had when longjmp was called. The values of register variables are unpredictable. The value returned by setjmp must be nonzero. If *value* is passed as 0, the value 1 is substituted in the actual return.
Call longjmp before the function that called setjmp returns; otherwise the results are unpredictable.
Observe the following restrictions when using longjmp:

- Do not assume that the values of the register variables will remain the same. The values of register variables in the routine calling setjmp may not be restored to the proper values after longjmp is executed.

- Do not use longjmp to transfer control out of an interrupt-handling routine unless the interrupt is caused by a floating-point exception. In this case, a program may return from an interrupt handler via longjmp if it first reinitializes the floating-point math package by calling _fpreset.

- Be careful when using setjmp and longjmp in C++ programs. Because these functions do not support C++ object semantics, it is safer to use the C++ exception-handling mechanism.

Laboratory Hints

- Look over the laboratory slides. They explain important project/architecture details that you will need to complete the lab.
- setjmp/longjmp are macros. That means that they do not follow regular function calling conventions. The _setjmp3 and _longjmp functions are called as part of setjmp and longjmp respectively and that is why they are the functions that you need to document.
- setjmp/longjmp perform extra checks to make sure that the machine is kept in a sane state after they operate. You do not need to understand everything that is happening in the code, but here is a brief overview:
    - One of the jmp_buf entries contains a "Cookie" value. This is a fixed value that is always located at the same offset within the structure. It is set within setjmp and then checked in longjmp
    - setjmp/longjmp attempt to deal with C++ exceptions conflicts. To do so, the address of an exception unwind function is saved as well as six words of state information. This is stored in setjmp and restored in longjmp if exception handlers are in place. Otherwise the code is skipped over and zeros are placed in those seven words of the jmp_buf structure.
    - The jmp_buf structure saves the 0'th byte of the fs register for a future check (in what is known as the jmp_buf's Registration field). The fs register is significant because it is the beginning of the Thread Environment Block in Windows NT where state information is kept for the current thread of execution. The 0'th byte of that structure points to the current chain of exception handling chain. If the values do not match then a function called global_unwind2 will be called to roll the exception handler to its previous setting.
- The Visual C++ compiler does not save critical values in every register. Some are used as scratch registers for temporary calculations. That is why setjmp does not save all of the general-purpose registers. (In fact, only five of the eight general-purpose registers are used to store important information under most circumstances) An implementation that was not Visual C++ specific would need to save every general-purpose register to be sure that the thread of execution could be properly restored.
- The disassembly code that you are given here uses setjmp and longjmp to jump within the same stack frame as a very simple example that you can document yourself. It is important to understand that the real power of setjmp and longjmp is the ability to jump between functions/stack frames. You may want to create your own example to see how this works for your own benefit (but you are not required to do so and should not hand anything in for this)

Helpful Resources

Assembly Language and Intel/Windows architectural issues

- The Intel Architecture Software Developer's Manual Volume 1, Basic Architecture, explains a number of important elements in the Intel 80x86 processor family. In particular, you will most likely want to read through Chapter 3 (Basic Execution Environment), Chapter 4 (Procedure Calls, Interrupts, and Exceptions), and possibly Chapter 5 (Data Types and Addressing Modes). Chapter 6 (Instruction Set Summary) gives a summary of the Intel assembly instruction set, of course. Full coverage of the instruction set can be found in the Intel Architecture Software Developer's Manual Volume 2, Instruction Set Reference. Both Manuals can be found on the CS 414 web page.
- Descriptions of setjmp, longjmp, and all other members of the C library can be found under the Visual C++ help menu.
- The Microsoft Developer's Journal has two articles explaining the most commonly found assembly output from C/C++ code compiled using Visual C++ in the February and June 1998 issues. They deal with some Windows-specific code that you may encounter in your disassembly efforts.

Understanding C

- *CS 113: Introduction to C* is being offered for the first four weeks of class
- Kernigan and Richie have written the definitive ANSI C reference in their book *The C Programming Language*
- *A Book on C* by Poe is more of a tutorial than the K&R book. It also serves as an excellent reference to the language.
- Indranil Gupta has created a series of slides entitled *C Programming from Java* that are available from the course web page.
- Dietel and Dietel's *C: How to Program* serves as a gradual introduction to the language.

Part B Questions

You are running a program with your calling stack set several segments deep. For the purpose of this example you may assume that all segment registers are set to a value of 200h other than FS, which is set to a value of 17ff86a0h. Please answer the following multiple-choice questions based on what you know. Do not make any assumptions about the value of registers, memory locations, or the value of the instruction pointer unless explicitly instructed to do so:

The scenario that we would like to examine is this:

The program execution ran through many function calls that eventually resulted in a call to original_function. A function call was made from original_function to current_function, and execution has just returned back to original_function following that function call. yet_another_function may have altered the stack and/or mark data structure, but will not have changed the instruction pointer portion of that structure or in the return address for any stack frame. The longjmp statement in the code fragment below is the only such statement throughout the program.

You have the following memory dumps for the running program. The leftmost column in each dump represents memory addresses. The subsequent columns representing bytes located in consecutive memory locations. Each byte is represented by two hexadecimal characters. The tens place represents the high-order nibble (4-bits) of an 8-byte character and the ones place represents the low-order nibble. For example, F8 would be 11111000 where memory addresses are increasing from right to left.

00430240 38 fe 44 00 00 00 54
00430247 00 f8 fd 64 00 4c bb
0043024e dd 81 14 fe 44 00 6a
00430255 10 40 00 28 fe 64 00
0043025c 00 00 00 00 30 32 43
00430263 56 00 00 00 00 00 00

0044fe2a ?? ?? ?? ?? ?? ?? 11
0044fe31 00 00 00 af 00 00 00
0044fe38 10 fe 43 00 08 17 1f
0044fe3f 06 8c 17 26 cd 18 26
0044fe46 57 9b fa 23 83 21 2b

```
int original_function(int a, int b, int c, int d)
{
  int x;
  int return_value;
  jmp_buf mark;

  x = 175;
  while(1) {
    return_value= setjmp(mark);
    if(return_value == 0) {
      return_value = current_function(mark);
    }
    else {
      printf("I don't like %d very much, but %d is my favorite number\n", return_value, x);
      break;
    }
  }

  exit(0);
}

int current_function(jmp_buf mark)
{
  int x;

  x = 11;

  yet_another_function(mark);
  longjmp(mark,0)
/* disassembly code for this longjmp instruction
004011AB   push        0
004011AD   push        offset _mark (00430240)
004011B2   call        _longjmp (004012e8)
end corresponding assembly code */

  return 0;
}
```

Question 1

What output will the **remainder** of the original_function's execution result in?
A) I don't like 0 very much, but 175 is my favorite number
B) I don't like 0 very much, but 11 is my favorite number
C) I don't like 0 very much, but 17 is my favorite number
D) I don't like 1 very much, but 17 is my favorite number
E) I don't like 1 very much, but 175 is my favorite number
F) I don't like 1 very much, but 11 is my favorite number
G) There is no way to know the variable values, but a print statement will occur
H) This is an infinite loop that will never print anything

Question 2

Where can we find original_function's second parameter, b, if we are about to execute the "x = 175" statement there?
A) [ebp + 4]
B) [esp + 4]
C) [ebp + 12]
D) [ebp – 4]
E) [esp – 4]
F) [esp – 12]
G) There is no way to know

Question 3
Assume that the following assembly instructions were embedded at the beginning of current_function (directly after we set the ebp and esp registers so as to initialization the function's stack frame).

```
push eax
push ebx
```

Where should we look for function variable x if we are about to execute the "x = 11" statement within the function?
A) [ebp + 4]
B) [ebp + 16]
C) [esp – 8]
D) [esp – 20]
E) [ebp - 8]
F) [ebp – 16]
G) [esp + 20]
H) There is no way to know

Question 4

If we wanted to modify setjmp/longjmp to save as much register state as possible (excluding segment registers) which registers would we need to save in addition to the registers already saved by setjmp/longjmp? Exclude any register that will be automatically overwritten before execution is resumed after a longjmp.
A) eax, ecx, edx, eflags
B) eax, ecx, edx, eflags, ss
C) eax, ecx, edx, eflags, cs, ds, es, fs, gs, ss
D) ecx, edx
E) ecx, edx, eflags
F) eax, ebx, ecx, edx, esi, edi, ebp, esp, eip, eflags
G) eax, ebx, ecx, edx, esi, edi, ebp, esp, eip, eflags, cs, ds, es, fs, gs, ss

Question 5

Which of the following applies to the setjmp/longjmp C library functions?
1) setjmp/longjmp could be used to make jumps between processes if we had a way to pass the mark structure from setjmp in one process to longjmp in another
2) Longjmp never restores local variables to their settings at the time setjmp was called.
3) If we had multiple call stacks then we could use setjmp and longjmp to save the state of the current user-level thread's stack, swap stacks with that of another user-level thread, and then use longjmp to both restore the cpu's state and to resume execution in the new thread where we left off (that is at the next executable assembly statement after the call to setjmp). Hence, setjmp and longjmp can act as a context switch for user-level threads.

A) 1, 2, and 3
B) 1 and 3 only
C) 2 and 3 only
D) 1
E) 2
F) 3
G) None of the above

Question 6

Go back to the code that you examined in Part A of this laboratory. Explain why the variables printed in the second print statement are not the same as those in the first print statement. What would happen if the Visual C++ compiler had stored some of the variables in registers? Would it make a difference?