Lesson 3: The Debugger

Objectives:

(a) Demonstrate the ability to debug a running C program in memory, to include the inspection of processor registers and arbitrary memory locations

(b) Analyze existing programs for flaws with the `gdb` debugger.

**Program Autopsy: Case 1**

Now, to really see what is going on in main memory, we can run a program one line at a time, and, at each step in the process, examine the CPU registers and any other memory locations we care to. We can step through an executable file and examine registers and memory by using a debugger. A debugger is a program that allows you to test and examine other programs. Here’s how to get started:

**Step 1.** Startup VMware Workstation (if the virtual machine was not shutdown, restart the virtual machine), navigate to your work directory and then using `nano`, open a new file named `ch5demo.c` by entering:

```bash
cd work
nano ch5demo.c
```

Enter the following program:

```c
#include<stdio.h>
int main()
{
    int x = 7;
    x = 2001;
}
```

Compile the program and ensure that it contains no syntax errors. Then run the program.

```bash
gcc -g -o ch5demo.exe ch5demo.c
./ch5demo.exe
```

You should see the results shown in the screen capture below.

Wait – what happened? This program is very simple - it merely stores and changes the value of the variable `x` in memory. It doesn’t get input from the user (that is, doesn’t use `scanf`), and it doesn’t display output either (doesn’t use `printf`), so there’s not much to see “on the outside” when the program is run. But what’s happening “on the inside” (in memory)? The debugger will help us figure that out.

**Step 2.** Start the debugger by entering the following seven commands. **Don’t include the comments!** – those are provided just to explain what is accomplished by each command. Look at the screen capture that follows as you are entering commands. Your screen should look the same

```bash
gcc -g -o ch5demo.exe ch5demo.c // The `-g` part of this is new! Adding this provides
// some extra functionality for the debugger.

gdb -q ./ch5demo.exe // `gdb` is the name of the debugger. So, we are
// running the debugger on the executable file named
// `ch5demo.exe`

list // This repeats the source code for convenience
```
disassemble main

// This shows the assembly code starting with the line // that has main

break main

// This sets a “breakpoint” at main. Which means // that when we run the program, it will halt execution // at the first line of executable code that follows the // line in the C program that contains main

run

// This starts executing the program up to the first line // of executable code that has main.

midshipman@EC310:~$ gcc -g -o ch5demo.exe ch5demo.c
midshipman@EC310:~$ gdb -q ch5demo.exe
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1    #include <stdio.h>
2    int main( )
3    {
4        int x=7;
5        x=2001;
6    }
(gdb) disassemble main
Dump of assembler code for function main:
0x0048344 <main+0>:  push    ebp
0x0048345 <main+1>:  mov     ebp,esp
0x0048347 <main+3>:  sub     esp,0x8
0x004834a <main+6>:  and     esp,0xffffffff0
0x004834d <main+9>:  mov     eax,0x0
0x0048352 <main+14>: sub     esp,eax
0x0048354 <main+16>: mov     DWORD PTR [ebp-4],0x7
0x004835b <main+23>: mov     DWORD PTR [ebp-4],0x7d1
0x0048362 <main+30>: leave
0x0048363 <main+31>: ret
End of assembler dump.
(gdb) break main
Breakpoint 1 at 0x0048354: file ch5demo.c, line 4.
(gdb) run
Starting program: /home/midshipman/work/ch5demo.exe

Breakpoint 1, main () at ch5demo.c:4
4        int x=7;
(gdb) 

The program’s execution is “frozen” at the first real line of code (the first line of executable code that follows the line that has main) So… where exactly did the program freeze?

**Practice Problem 5.1**

In the screen capture above, at what assembly language instruction did the program stop?—that is, what is the next instruction that will execute, and what is the address in main memory where this instruction is stored?

**Solution:**

You might be wondering: What about all the instructions before this one? Do they matter? The answer is that those are instructions that the compiler has generated to set up memory for the program. We can safely ignore these for now.
Since the two last assembly language instruction – `leave` and `ret` – are basically mop-up operations (all programs end with these two instructions), we really only have to concentrate on these two instructions:

```
0x08048354 <main+16>:  mov   DWORD PTR [ebp-4], 0x7
0x0804835b <main+23>:  mov   DWORD PTR [ebp-4], 0x7d1
```

What do we make of these two cryptic lines? To find out, we introduce two powerful commands: the `info` command and the `examine` command.

**Step 3. The info command.** In the GDB debugger, the `info` command (or just ‘i’ for short) returns information about registers, files, functions, memory, breakpoints, etc. We will use it to query the registers. For example, to see the address stored in the `eip` register, you would enter the command:

```
i r eip
```

and to see the address stored in the `esp` register, you would enter the command:

```
i r esp
```

You can also see the addresses of more than one register at a time with the `info` command. For example, to display the `eip`, `esp`, and `ebp` simultaneously, you would enter the command:

```
i r eip esp ebp
```

**Practice Problem 5.2**

What is the value (address) stored in the `eip` register?

**Solution:**

**Step 4. The examine command.** To examine the value stored at a memory location, we use the examine `(x)` command. The format for the `x` command is:

```
x/  location we want to display (address)
```

The first position specifies the repeat count, this is optional.

The second position specifies the format for the display. Use this table:

| x | hexadecimal |
| d | decimal     |
| u | unsigned decimal |
| i | assembly language |
| s | string      |
| c | character   |

The third position specifies the number of items we want to display. Use this table:

| b | byte |
| w | word (four bytes) |

To see the contents of a memory location, simply place the memory location here. To see the contents of an address whose location is stored in a register, place the register here, preceded by a dollar sign. (e.g. `$esp`)

So, the examine command starts with an `x` followed by a slash. Then we tell the debugger how we would like the memory location contents to be displayed. If we want the value to be displayed in hexadecimal, the display option is `x`. If we want the value to be displayed in decimal, the display option is `d`.

If we want to display the contents of a memory location, we simply supply the memory location as the last argument. If we instead want to see the contents of a memory location whose address is in a register, we supply the register name preceded by a dollar sign (`eip`).
We can also control "how much" data is displayed. By default, the debugger displays 4 bytes for its answer.\(^1\) If we only want to display a single byte, we place the letter \texttt{b} right after the display option. To display a word, we place the letter \texttt{w} after the display option. As we go along, we will see that there are other ways to display other amounts of data.

To summarize the examine command:

If the foregoing paragraphs have you bewildered, do not fear! We will do some more examples! To answer the following practice problems, let’s take another look at the program instructions as machine language, assembly language, and their address in memory.

Here is another way to look at the text segment of our program.

\(^1\) This default value; however, can be changed on the debugger, but we will use the default.
Practice Problem 5.3

Refer to the pictures shown above. What should be printed out by each of the following commands? In each case, enter the command to confirm your answer.

(a) \texttt{x/xb 0x08048354}
(b) \texttt{x/xb 0x08048355}
(c) \texttt{x/xb 0x08048356}
(d) \texttt{x/xb 0x08048357}

Solution: (a) (b) (c) (d)

Now, recall that when we use \texttt{b} in the examine command, as in \texttt{x/xb}, the \texttt{b} stands for byte. When we issue the command \texttt{x/xb 0x08048354}
we are saying: "Show me the contents of main memory, starting at address 0x08048354, but only a single byte."

We take a snippet of the text segment above and look at it again with the examine command.

\begin{verbatim}
  8048354 | c7
  8048355 | 45
  8048356 | fc
  8048357 | 07
  8048358 | 00
\end{verbatim}

If we want to see the contents of memory starting at address 0x08048354, but see four bytes we would enter: \texttt{x/4xb 0x08048354}.

Practice Problem 5.4

What will be displayed by the command: \texttt{x/4xb 0x08048354}? What do you notice about the order of the bytes in the value displayed?

Solution:

The x86 processor stores values in so-called little-endian order, which we were introduced to in Chapter 2. If we are looking at more than one byte in memory, the debugger will reverse the bytes when we use the examine command with the \texttt{w} option. The debugger reverses the bytes for us automatically (that is, it undoes the little-endian format) because the debugger thinks that those bytes represents a single value (like an integer). Practice Problem 5.5

What do you think will be displayed by the command: \texttt{x/xw 0x08048354}? Confirm your result.

Solution:

If we wish to look at multiple bytes of memory and see exactly how they are stored, we would use the \texttt{b} option and include the number of bytes we wish to see. For example, to see 2 bytes of memory as they are stored in memory at address 0x08048354, we would use

\begin{verbatim}
(gdb) x/2b 0x08048354
0x08048354 <main+16>: 0xc7 0x45
\end{verbatim}

This is confusing, but that’s ok! The point we want to reinforce here is that everything in stored in memory as it should be but it is important to understand the commands you are entering and how to comprehend the results!
Practice Problem 5.6
What do you think will be displayed by the command: \texttt{x/4xb 0x08048354}? Confirm your result.

Solution:

Step 5. Using the examine command with registers. As mentioned earlier, if we instead want to see the contents of a memory location whose address is in a register, we supply the register name preceded by a dollar sign. So, the command \texttt{x/xb \$eip}

means the following: “Display one byte of memory at the address stored in the instruction pointer.”

Practice Problem 5.7
What do you think will be displayed by the command: \texttt{x/xb \$eip}? Confirm your result.

Solution:

The preceding example is explained by the picture below. The command \texttt{x/xb \$eip} means to proceed to the memory location that is contained in the instruction pointer, and read off one byte.

Practice Problem 5.8
What do you think will be displayed by the command: \texttt{x/2xb \$eip}? Confirm your result.

Solution:

Practice Problem 5.9
What do you think will be displayed by the command: \texttt{x/i \$eip}? Confirm your result.

Solution:

Step 6. Wonderful… so what does the program actually do? We mentioned that our program has two lines of code we care about:

\begin{verbatim}
0x08048354 <main+16>: mov    DWORD PTR [ebp-4],0x7
0x0804835b <main+23>: mov    DWORD PTR [ebp-4],0x7d1
\end{verbatim}

We know that the \texttt{eip} contains the address of the next instruction to be executed: \texttt{0x8048354}. If we were to execute one instruction and then freeze again, the instruction executed would be:

\begin{verbatim}
mov    DWORD PTR [ebp-4],0x7
\end{verbatim}

What does this cryptic instruction do?

For starters, the register \texttt{ebp} is the base pointer (which, you may recall chapter 2, points to the memory address immediately below the bottom of the stack). The stack is a section of memory that our program has available to store any values it needs.
The esp register contains the address of the "top" of the stack, and the ebp contains the address below the bottom. As mentioned in Chapter 4, DWORD refers to "double word", which is a 4-byte value.

This assembly language instruction means (in plain English):

Move the value 0x7 into the address pointed to by ebp-4 (the base pointer, minus 4). The value will occupy 4 bytes.

The base pointer contains an address; this instruction will write the value 0x00000007 into the address 4 above the address contained in the base pointer.

Let's look at a picture of the bottom of the stack. Suppose the base pointer contained the address 0xbffff818. Then that would mean that my program is storing all the information it needs (for example, variables) just above address 0xbffff818. See the picture below:

So… If I know the value 0x00000007 is going to be placed in the address 4 bytes above ebp in memory, how does that change the memory layout above?

First, let’s figure out the address where the value 7 is placed (ebp-4):

\[
\begin{align*}
0xbffff818 - 4 &= 0xbffff814 \\
\end{align*}
\]

That’s not so bad. So the 4-byte value 0x00000007 is going to begin at address 0xbffff814.

Next, we have to remember the order in which those bytes are stored. (If you’re thinking, little-endian – GREAT!)

Recall from Chapter 4 that little-endian order means that the least significant byte goes in the first address, the second-least-significant byte goes in the next address, and so on, so let’s take a look at how that applies to a 4-byte integer.

The integer “7” is represented by the following 4 bytes:

\[
\begin{align*}
0x 00 00 00 07 \\
\end{align*}
\]

In memory, the least significant byte goes in the first address, like this: (Most significant Byte) (Least significant Byte)

To tie it all together - the “big picture,” if you will – the 4 bytes are placed in memory, with the least significant byte beginning at address 0xbffff814, like this:
That’s probably enough pontificating about what will happen when the next instruction is executed... Let’s actually execute a single instruction, and then freeze again! Enter the command:

`nexti`

After you enter this command, you should see (note: you cannot go backward in the debugger once you enter `nexti`):

```
5
x = 2001;
```

**Practice Problem 5.10**

When you execute a command (as you just did when you typed `nexti`), what happens to the instruction pointer (`eip`)?

**Solution:**

We have advanced to the next instruction. The instruction at address `0x0804835b` will be the next instruction to execute.

**Practice Problem 5.11**

What is the value (address) stored in the `eip` register? Does this answer make sense?

**Solution:**

**Practice Problem 5.12**

The instruction corresponding to `int x = 7;` has just been executed. What should I type to examine memory to see the integer value 7 on the stack? Confirm your result!

**Solution:**

**Practice Problem 5.13**

What assembly language instruction is located at `0x0804835b`?

**Solution:**
Practice Problem 5.14

Sketch what you expect the stack to look like after the instruction at address 0x0804835b is executed. Fill in the blocks in the memory diagram.

Solution:

Let’s execute a single instruction, and then freeze again! Enter the command:

`nexti`

Practice Problem 5.15

What two things happened when you entered `nexti`?

Solution: 1.
### Practice Problem 5.16
What should you type to examine memory for the hex values you sketched in Practice Problem 5.14? Confirm your result!

Solution:

### Practice Problem 5.17
What should you type to examine memory in decimal where the integer value 2001 is stored? Confirm your result!

Solution:

Congratulations! You've completed your first program autopsy!
CH 3. Problems

1. Consider the picture below, where all memory contents are in hexadecimal:

   ![Memory Contents Diagram]

   a) In words: what is held in the eip register, i.e., what is the purpose of this register? (Your answer should not be: "804838d".)

   b) What would be displayed on the monitor by the command: i r eip?

   c) What would be displayed on the monitor by the command: x/xb $eip?

   d) What would be displayed on the monitor by the command: i r esp?

   e) What would be displayed on the monitor by the command: x/xw $esp?

   f) What would be displayed on the monitor by the command: x/xb 0x08048475?

   g) What would be displayed on the monitor by the command: x/xs 0x08048475?

2. In this problem we are going to use the program named hwk5.c, which is located in the booksrc directory. We need to copy this file to the work directory. To copy the file named hwk5.c from the booksrc directory to the work directory, first ensure you are in your home directory be entering

   `cd`

   and then carefully enter the following at the home directory prompt:

   `cp ec310code/hwk5.c work`

   Verify that you have hwk5.c in your work directory by changing to the work directory:

   `cd work`

   and then listing the files in the work directory:

   `ls`

   You should see the hwk5.c file (along with perhaps some additional files from recent labs).
The program `hwk5.c` is shown below:

```c
#include <stdio.h>
int main()
{
    int i;
    for (i = 0; i < 10; i++)
    {
        printf("Hello World!\n");
    }
}
```

Compile your program using `gcc`:

```
gcc -g -o hwk5.exe hwk5.c
```

and then run your program

```
./hwk5.exe
```

to confirm it executes as expected.

Then start the debugger by entering the following commands (hitting ENTER after each command)

```
gdb -q ./hwk5.exe
list
disassemble main
break main
run
```

(a) The program has now stopped at the first line of code after the line `int main()`.
Recall that the `eip` register holds the address of the next instruction that will be executed. What is the address stored in the `eip` register?

(b) What is the next assembly language instruction that will be executed?

(c) Consider the assembly language instruction

```
mov DWORD PTR [ebp-4], 0x0
```

This instruction places the value 0 into the memory location whose address is stored at `ebp-4` and stores it in 4 bytes.

Enter `nexti` to execute this instruction

What is the value of `ebp`?

(d) What is the value of `ebp-4`?

(e) What is stored in the address specified by the value `ebp-4`? Hint: Use the `x/xw` with your answer to question (d).

(f) Look at the value of the instruction pointer (`eip`). Has it changed from your answer to part (a)? Why? /Why not?

(g) The next assembly language instruction that will be executed is:
cmp    DWORD PTR [ebp-4],0x9

This instruction will compare the value of 9 to the value you examined in question (e). Referring back to the C source code, what do you think this assembly language instruction is doing?

Enter **nexti** to execute this instruction

**h)** The assembly language instruction that will be executed next is

jle   0x8048393 <main+31>

This instruction means:

*If the result of the preceding comparison showed that the value stored at the memory location whose address is stored at ebp-4 is less than or equal to 9, jump to address 0x8048393.*

Enter **nexti** once. What is the value of the **eip** register?

If you answer to (h) is not **0x8048393** then you have gone off the rails! **STOP!** See your instructor (or MGSP).

**i)** Explain, in words, why the instruction pointer has the value that it has.

**j)** The assembly language instructions

```asm
mov    DWORD PTR [esp],0x8048484
```

moves the value 8048484 into the location pointed to by the stack pointer.

Enter **nexti** once.

What is the address stored in the stack pointer (**esp**)?

**k)** What is stored at the memory location whose address is in the stack pointer? (Hint: use `x/xw` to examine the value stored at the address specified by the stack pointer.

**l)** We would like to know the significance of the address 0x8048484. What is stored at this location? (Hint: Examine the first four bytes stored starting at this memory location…think ASCII…could this be a string?)