Princeton University
COS 217: Introduction to Programming Systems
GDB Tutorial for Assembly Language Programs (Part 2)

Motivation

Suppose you are developing the assembly language BigInt_add() function. Further suppose that the function assembles and links cleanly, but executes incorrectly. How can you use gdb to debug the function?

The BigInt_add() function is somewhat difficult to debug because it uses the stack, structures, and arrays. This is an appropriate sequence...

Building for gdb

To prepare to use gdb, build your program with the -g option.

```
% gcc -g -Wall -ansi -pedantic fib.c bigint.c bigintadd.s -o fib
```

Doing so places extra information into the fib file that gdb uses.

Running gdb

Run gdb from within xemacs.

```
% xemacs
<Esc key> x gdb <Enter key> fib <Enter key>
```

Setting Breakpoints

Set breakpoints at appropriate places. Breakpoints at the beginning of the main() and BigInt_add() functions would be appropriate.

```
(gdb) break main
(gdb) break BigInt_add
```

Running Your Program

Run the program, specifying some command-line argument.

```
(gdb) run 500000
```

Continue past the breakpoint at the beginning of the main() function.

```
(gdb) continue
```

Execution is paused after the two-instruction prolog of the first call of the BigInt_add() function. Issue the “continue” command nine more times. At this point the BigInt_add() function is being called to add the numbers 55 and 34.

Examining Memory

Use the print command to determine the contents of the EBP register:

```
(gdb) print/a $ebp
bffff7b8
```

Thus you know the address of the base of the current stack frame. (That address might be different each time you run the program.) Now use the x command repeatedly to examine the function’s parameters as they exist in the stack and the heap.

Examine the function’s stack frame, interpreting each value as an address:

```
(gdb) x/a 0xbffff7b8
bffff808
(gdb) x/a 0xbffff7bc
804868e
(gdb) x/a 0xbffff7c0
b7f1a008
```

```
(gdb) x/a 0xbffff7c4
b7f7c008
(gdb) x/a 0xbffff7c8
b7eb8008
```

Examine the heap, interpreting each value as a decimal integer:

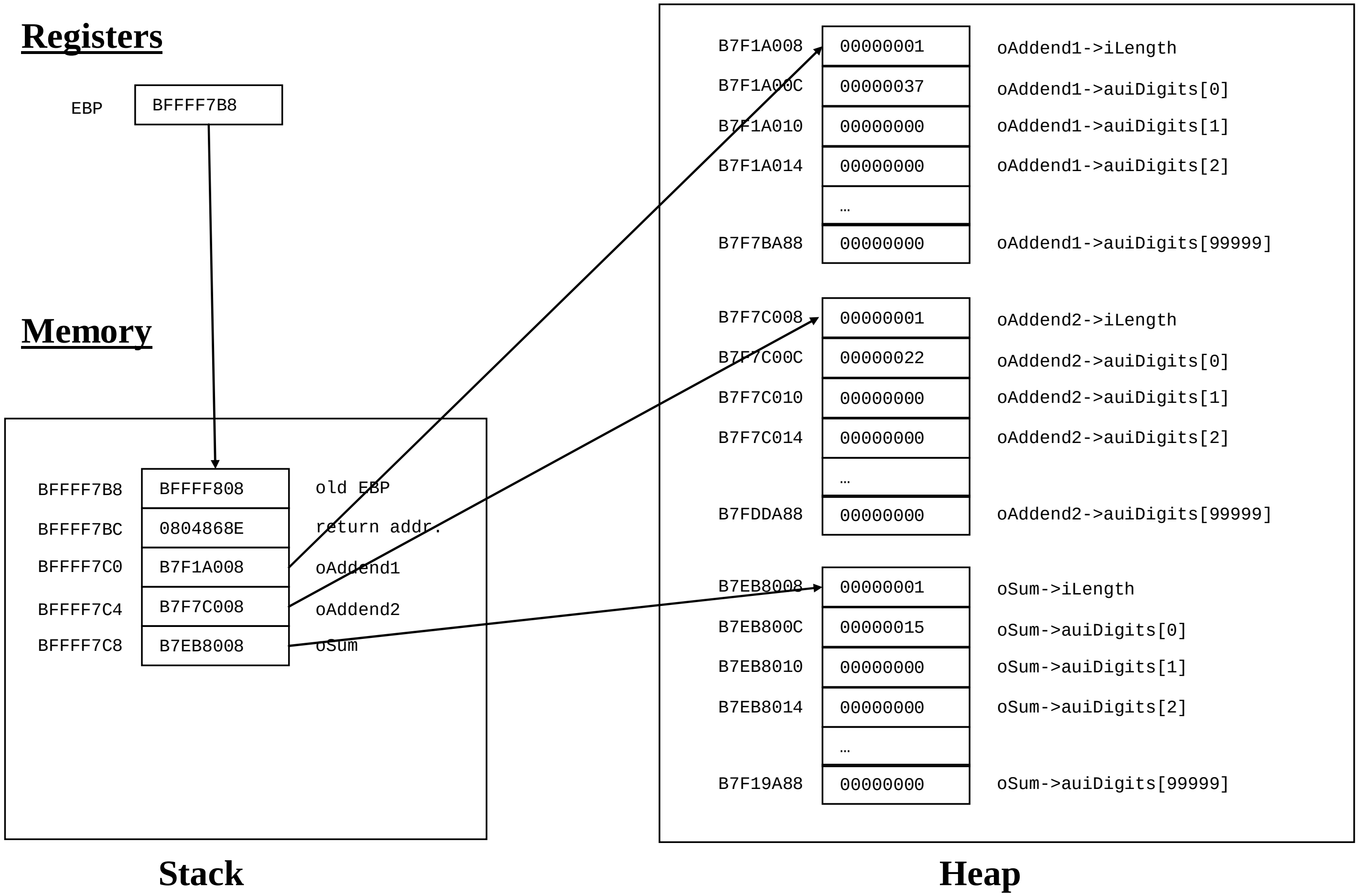
```
(gdb) x/d 0xb7f1a008
1
(gdb) x/d 0xb7f1a00c
55
(gdb) x/d 0xb7f1a010
0
(gdb) x/d 0xb7f1a014
0
(gdb) x/d 0xb7f7ba88
0

(gdb) x/d 0xb7f7c008
1
(gdb) x/d 0xb7f7c00c
34
(gdb) x/d 0xb7f7c010
0
(gdb) x/d 0xb7f7c014
0
(gdb) x/d 0xb7fdda88
0

(gdb) x/d 0xb7eb8008
1
(gdb) x/d 0xb7eb800c
21
(gdb) x/d 0xb7eb8010
0
(gdb) x/d 0xb7eb8014
0
(gdb) x/d 0xb7f19a88
0
```

As you traverse memory, draw a map of it as shown on the next page.

Suppose oAddend1 = 55, oAddend2 = 34, and oSum = 21



Registers

EBP	BFFFF7B8

Memory

Stack

BFFFF7B8	BFFFF808	old EBP
BFFFF7BC	0804868E	return addr.
BFFFF7C0	B7F1A008	oAddend1
BFFFF7C4	B7F7C008	oAddend2
BFFFF7C8	B7EB8008	oSum

Heap

B7F1A008	00000001	oAddend1->iLength
B7F1A00C	00000037	oAddend1->auiDigits[0]
B7F1A010	00000000	oAddend1->auiDigits[1]
B7F1A014	00000000	oAddend1->auiDigits[2]
	…	
B7F7BA88	00000000	oAddend1->auiDigits[99999]
B7F7C008	00000001	oAddend2->iLength
B7F7C00C	00000022	oAddend2->auiDigits[0]
B7F7C010	00000000	oAddend2->auiDigits[1]
B7F7C014	00000000	oAddend2->auiDigits[2]
	…	
B7FDDA88	00000000	oAddend2->auiDigits[99999]
B7EB8008	00000001	oSum->iLength
B7EB800C	00000015	oSum->auiDigits[0]
B7EB8010	00000000	oSum->auiDigits[1]
B7EB8014	00000000	oSum->auiDigits[2]
	…	
B7F19A88	00000000	oSum->auiDigits[99999]

Using the Memory Map

Such a memory map can help with debugging. Moreover, such a memory map can help with writing assembly language code in the first place. (Indeed if you did not have such a memory map, you probably would find it helpful/necessary to create one using pretend memory addresses before writing your assembly language code.)

For example, suppose you must write assembly language code to access oAddend2->auiDigits[2]. Using the memory map, it is easy to see that either of these instruction sequences would work:

Using indirect addressing:

```
movl %ebp, %eax   # EAX contains BFFFF7B8
addl $12,  %eax   # EAX contains BFFFF7C4, alias &oAddend2
movl (%eax), %eax # EAX contains B7F7C008, alias oAddend2
addl $4, %eax     # EAX contains B7F7C00C, alias oAddend2->auiDigits
movl $2, %ecx     # ECX contains 2, alias the index
sall $2, %ecx     # ECX contains 8, alias a byte offset
addl %ecx, %eax   # EAX contains B7F7C014, alias oAddend2->auiDigits + 2
movl (%eax), %eax # EAX contains 00000000, alias oAddend2->auiDigits[2]
```

Using base-pointer and indexed addressing:

```
movl 12(%ebp), %eax         # EAX contains B7F7C008, alias oAddend2
movl $2, %ecx               # ECX contains 2, alias the index
movl 4(%eax, %ecx, 4), %eax # EAX contains 00000000, alias oAddend2->auiDigits[2]
```