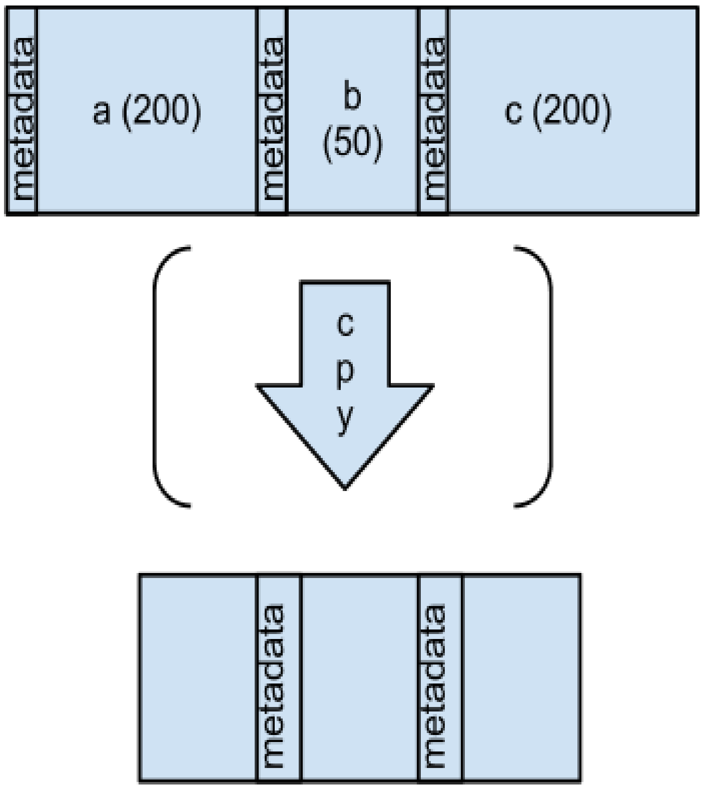
Scribe Notes - Lecture 08 (September 28th)

Kevin Mazige, Lucas Freitas, and Zach Hamed

Related to Assignment 1: Debugging Memory Allocator

- "Everybody gets an A+" -Professor Kohler
- All students get 3 extra late days for use on any assignment
- No special grading breakdown
- Using `compare.pl`
    - `perl compare.pl o test001.c`
    - Will print `o OK` if test passes
- Questions about `test026.c`
    - How to do the test in O(1) time
    - Allocates 4 different regions of memory
    - We expect that those regions are allocated sequentially
        - reasonable assumption for this problem set

    - `memcpy(p, b - 200, 450)` copies part of region a, b, and c (including metadata of b, and maybe even c depending on the size of metadata) to `char *p`



    - `memcpy(b - 200, p, 450)` is a copy in the reverse direction, and the metadata of b (freed) is copied back to its original place.
    - Should print out that some sort of invalid free occurred (double free)
    - How can you tell in O(1) time to see if something has happened?
        - Check the pointers that point to b (a's pointer to b and c's pointer to b)

        - This is O(1) work
    - Efficiency time (making things O(1)) goes into the 15% of the problem set's design grade.

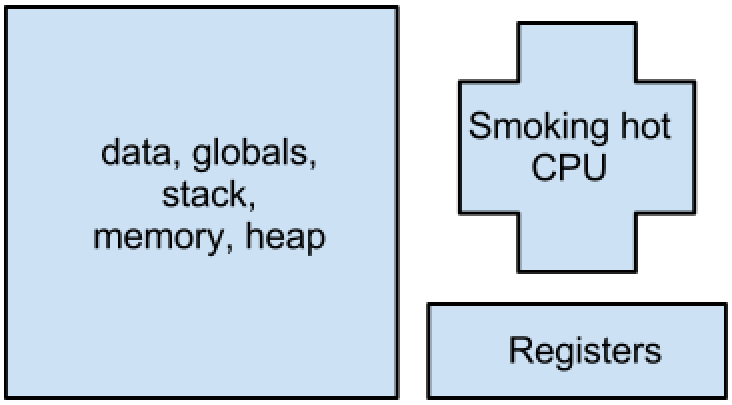
How code is represented in memory

- We will start studying assembly x86
- We study this not because we will write lots of assembly code, but because we might have to write little snippets of assembly code or read assembly code to analyze the performance of code
    - Reading assembly x86 code
- The x86 architecture is weird
    - Created in the 1970s
    - 16-bit address spaces
    - Every instruction was 64 bits long
    - Nowadays instructions are from 1 to 15 bytes long
- Let's take a look at some code
    - Each program was compiled by typing gcc -01 -s
        - - 01 → a little optimization
        - - s → don't generate binary, generate Assembly
- Assembly files contain
    - Data: instructions and constants
    - Metadata: debugging info, more metadata
        - Indicated by lines that start with a period
- `f:`
    - causes the binary to have a link from the name f to the instructions to follow it: f is a label (not code)
- `rep` and `ret` are actual instructions
    - `rep`: repeat
        - Not necessary
        - Makes sure that function has more than 1 byte in it
    - `ret`: return
        - Necessary
        - Return from function
- Function in the example (f0.s) doesn't take any arguments, and doesn't return a value (void return type)
- `New example (f01.s), new instruction: movl $0, %eax`
    - not a void function (or optimized compiler wouldn't generate extra code if not returning anything)
    - returns 0 (int return type)
    - puts 0 in the eax register (that's where x86 functions return their values)

What is a register?

- CPU can't speak of memory directly - it is only able to act on a very small handful of variables that live very close to the CPU (registers) - there is just one set of registers per CPU
- C language doesn't talk about registers
- x86 can't do arithmetic on memory

- it does arithmetic on registers, then moves results into memory
- `x += 5` in C turns into
    - `%eax <- x` (eax gets x)
    - `%eax <- %eax + 5` (eax gets eax plus five)
    - `x <- %eax` (x gets eax)
- Anything in a function is processed in register
    - x86 processor is like a turing machine following some program (like a tape) and eip is the current state
- Most important x86 general registers
    - `%eax, %ebx, %ecx, %edx, %esi, %edi, %ebp, %esp` are seen commonly in code
    - `%eip` is the address of the currently executing instruction
- %eax stores return values for functions



Form of an arithmetic instruction

- x86 arithmetic or move instruction format:
    - operation, source, destination
- `movl $0, %eax`
    - Read as “move the constant value 0 into eax”
- `xorl %eax, %eax`
    - Read as “eax becomes eax xor eax”
    - `%eax <- %eax ^ %eax`
    - Xor of anything with itself is 0
    - Equivalent to returning 0
    - Shorter in terms of code length than saying “return 0” (1 byte rather than 5 bytes)

Interpreting assembly

```
short/unsigned char/int/unsigned/unsigned char ******f(void) {
      return 0/0/0/0/NULL
}
```

- All have same interpretation
- Return type is lost when translated into assembly

- Processor doesn't understand types
    - Processor doesn't know what's an address and what's not an address
    - It only understands numbers
- When translating from assembly to x86 code, we lose the return information

More examples

- Example 1
    - The assembly
      ```
      movl (move) a, %eax
      subl (subtract) b, %eax
      ret
      ```
    - returns a - b
    - a and b are global variables. How do we know?
        - If they were locals, there would be a definition for them beforehand
        - Not heap addresses because this is code that goes into static storage duration of the program. Heap addresses come in at runtime - there is no way of linking in a
        - Thus, a and b must be global variables.
    - Just names (like a and b) are references to global variables
    - Sources
        - %eax → source is address
        - name (no % sign) → global variable
    - Declaring variables as extern says variables are declared elsewhere, so don't allocate space
- Example 2
    - The assembly
      ```
      movl b, %eax
      addl a, %eax
      ret
      ```
    - Adds s a and b and returns a + b
- Example 3
    - The assembly
      ```
      movl x, %eax
      movl (%eax), %eax
      ret
      ```
    - Returns *x
    - Parentheses () in assembly are like the asterisk in C (treats number as address)
    - %eax is just 32 undifferentiated bits. Your job is to determine how those bits are being used
    - If you see parentheses, chances are it's an address
- Example 4
    - The assembly
      ```
      movl x, %eax
      ret
      ```
    - Global variable x was an array - returns x[0]

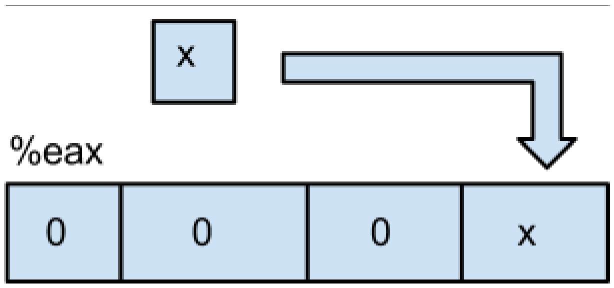
    - In memory, either if there is a single variable with some address x, or an array at some address x, the address of the single variable and of the first element of the array is the same.

- Example 5
    - The assembly

```
movzbl x, %eax
ret
```

- ‘z’: refers to “zero”
- ‘b’: regers to “byte
- ‘l’ in x86 assembly means 32 bits
- If you tried to do `movl %ax`, which is only 16 bits big, the assembler would complain about a mismatch in **size**
- movezbl → moves **byte** with **zero** extension into **32-bit** (‘l’) register
- The program that this corresponds to is

```
unsigned char x;
int f() {
      return (int)x;
}
```



    - `movsbl` performs a **sign** extension, which preserves the sign of signed numbers
        - If number is positive, `movsbl` and `movzbl` do the same thing
- Example 6
    - Assembly code

```
movl x, %eax
movzbl (%eax), %eax
ret
```

- x is a pointer (%eax)
- x is unsigned char (movzbl)
- Thus, in C, this corresponds to

```
unsigned char *x;
unsigned f(void) {
      return *x;
}
```

- If it was unsigned short *x, we would have:

```
movl x, %eax
movzwl (%eax), %eax
ret
```

- w stands for word (16 bits)

- Example 7
    - Assembly code
      ```
      movl x, %eax
      movzbl 1(%eax), %eax
      ret
      ```
    - 1(%eax) means *(%eax + 1), or add 1 to %eax and then dereference it
    - In C, the program just has `return x[1];`
- Example 8
    - Assembly code
      ```
      movl 4(%esp), %eax
      ret
      ```
    - s stands for stack → parameters

- In C, this is
  ```
  unsigned f(unsigned i) {
        return i;
  }
  ```

- We will understand why next week!