Section Notes 2 (Solutions)

Memory and Data Representation

Useful GDB Commands

`run (r)` - Executes file passed as command line argument to GDB

`backtrace (bt)` - print the call stack

`frame <#>` - examine the stack frame at <#> given by backtrace

`up` - moves up the call stack n frames

`down` - moves down the call stack n frames

`break (b)` - Pauses execution when a particular point in the code is reached

- Syntax:
  - `break <FILENAME>:<LINE NUMBER>`
  - `break <FUNCTION NAME>`
  - `break <FILENAME>:<FUNCTION NAME>`

`step (s)` - Steps into a line of code (enters function calls)

`next (n)` - Steps to next line of code (does not enter function calls)

`advance <#> (adv)` - Steps to a specific line of code (does not enter function calls, useful to break out of loops)

`finish` - run until the current function returns then break and print the return value

`examine (x)` - Examine memory

- Syntax
  - `x/<# of Elements><Type of Elements><Additional Size Info>`
  - What would `x/20xg` print?
    - 20 hexadecimal words which are 8 bytes each (a quad)

`display (disp)` - Print something each time a command in GDB executes

`print (p)` - prints :)

`set` - Sets a value in the executing program

- Syntax
  - `set var <VARIABLE NAME>`

`thread` - Switches between threads in a given program

- Syntax
  - `thread <Thread Number>`

`thread info` - Shows existing threads

`thread apply all where` - Get backtrace for all threads

`q` - Quits

Check the man pages or run ‘help’ from inside gdb for more information

Quick Valgrind Overview

Valgrind provides a set of tools each of which performs some kind of debugging, profiling, or similar task that helps you improve your programs. You have probably only used the "memcheck" tool, which is a memory debugging tool used primary for Linux and Mac. It simulates an executable and tracks memory errors such as uninitialized variables, memory leaks, and buffer overflows. Valgrind does not detect static (stack) memory errors, and only detects errors that occur dynamically (during runtime).

Usage: `valgrind --tool=memcheck --leak-check=full <program>`

Output:
Uninitialized Variables:

```
==9388== Syscall param exit_group(status) contains uninitialised byte(s)
==9388== at 0x455F7364: _Exit (in /lib/libc-2.14.90.so)
==9388== Use --track-origins=yes to see where uninitialised values come from
```

Memory leaks:

```
==9892== LEAK SUMMARY:
==9892== definitely lost: 64 bytes in 4 blocks
```

Buffer Overflows/Invalid Writes:

```
==9736== Invalid write of size 4
==9736== at 0x8048435: main
==9736== total heap usage: 1 allocs, 1 frees, 16 bytes
==9736== All heap blocks were freed -- no leaks are possible
==9736== ERROR SUMMARY: 1 errors from 1 contexts
```

See http://valgrind.org/docs/manual/valgrind_manual.pdf for more information! Note that we have only used the "memcheck" tool thus far.

A buggy heap.

A heap is a tree-based data structure that satisfies something called the *heap property*. This property states that all pairs of parent/child nodes are ordered in the same way. For example, in a max heap a parent node is always greater than all its children. Therefore, you can see that the maximum element in a max heap is the root node of the tree. Heaps have O(log(n)) insertion and deletion time for their elements.

The code heap.c will read in integers as command line arguments and build a valid max heap from these numbers. Run "make" and then "./heap 3 4 5". It prints out the resulting heap; the max element should be first, and then the following rows print lesser items. We get this:

```
$ ./heap 3 4 5
5
4   3
```

This looks good—the maximum item is on the first line, and the other items are there too. But the next line doesn't look good:

```
*** Error in `./heap': munmap_chunk(): invalid pointer: 0xbffbde0c ***
Aborted (core dumped)
```

Core dump!! Time to boot up GDB to figure out what happened.

Bug 1 (from Section 1)

- Open up GDB with the command "gdb heap". You can also pass in the -tui flag ("gdb -tui heap") in order to have a graphical interface to see the code alongside the debugger.
- Run the program by typing "run 3 4 5". GDB conveniently stops where the Segmentation Fault occurred.
- We can print the contents of all the variables in scope and try to figure out what went wrong. Do you see what happened? What is the bug in the code that caused this to occur?

SOLUTION: Bug: passing &h instead of h to heap_free; h is on the stack, not the heap. Use `bt` to get to user code, since we assume almost always that there is no bug in library code.

Good! Having fixed the bug, we can try another order:

```
$ ./heap 5 3 4
5
3   4
```

A little different, but that's still fine—we still are providing the heap property. Do you know the `seq` command? Try this:

```
$ seq 1 4
```

What does seq do?

Bug 2

OK, great. Now try this, which is a short-hand way to type `./heap 1 2 3 4 5 6`:

$ ./heap `seq 1 6`
5 6
4 2 1
***** Error in `./heap': free(): invalid next size (fast): 0x0000000000779030 *****

Ouch! We seem to have a core dump! (This is on a 32-bit Linux VM. On Mac, `seq 1 4` works, but we get an error with `seq 1 17`.)

And this, which is a short-hand way to type `./heap 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100`:

$ ./heap `seq 1 100`

gives a different error! Time for some more gdb—and maybe some even smarter tools.

SOLUTION: We have three additional problems. First we are allocating bytes without multiplying by sizeof(int). As a result, writes to h->array[i] overwrite allocation metadata. This is easiest to track, not in gdb, but in valgrind, but gdb will show you that the program goes wrong on the free() — and students should be able to use logic to figure out thereafter that the problem is heap corruption, and that valgrind makes sense. We recommend using gdb first, then valgrind.

The second problem is that we are ignoring the return value of realloc(). This too will show up in valgrind.

The third bug is when we pass an even number of integers to the program. The macro RIGHT will end up with an out-of-bound array index if heap->size is even. This too can be detected by valgrind.

(fourth bug) Also, consider the warnings about uninitialized variables in comparisons. Note that realloc does not initialize the new space!

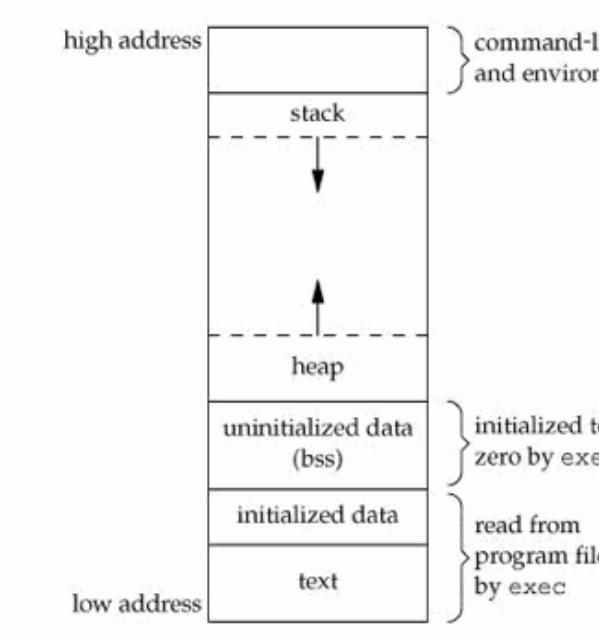
The Three Levels of Memory

1. Operating System Level:

- Memory is a giant array of bytes to the operating system
- The operating system presents data to each process (program) running on the computer through virtual memory. (We will learn about this concept later)

2. Program Level:

- The address space for a running program is divided into sections called *segments*:
  - Stack: Contains local variables and functions
  - Heap: Dynamic variables
  - Data: Global variables
  - Text: Code



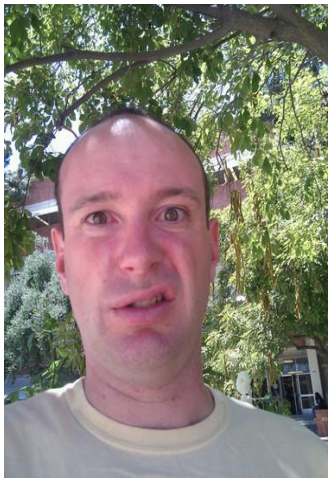
3. Programmer Level:

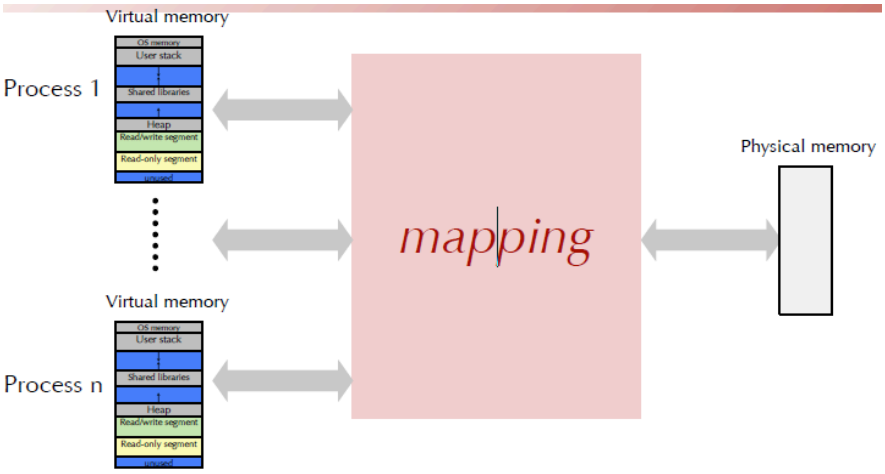
- The programmer is able to access the memory of his or her process through the C programming language
- Memory management of static and local variables is done automatically.
  - This is why you can just declare global and local variables without any function calls. The compiler arranges for these variables to get space in the data and stack sections.
- Dynamic memory has to be explicitly asked for by the programmer through use of library functions. The programmer should explicitly free each piece of dynamic memory when they're done with it, or the program will run out of memory.
  - `malloc`
  - `calloc`
  - `realloc`
  - `free`
- C also offers other library functions that treat memory generically, as an array of bytes. You may find these functions useful as you implement memory management functionality.
  - `memcpy, memmove, memchr`...

Programmer Asks for Memory

Program asks the OS

OS get's space from RAM





Exercises:

Problem 1:

Place all of the variables in the correct section of memory and identify when in execution each variable goes in and out of scope:

```
1       #include <stdlib.h>
2       #include <stdio.h>
3
4       #define MY_VALUE 10 // Q#1
5       int g = 45; // Q#2
6
7       const char* a_string = "A fixed string"; // Q#3
8
9       void add_helper(int* c, int* d, int* e) {// Q#4
10          *c = *d + *e; // Q#5
11       }
12
13      int add(int a, int b) {
14        int* answer = malloc(sizeof(int)); // Q#6
15        add_helper(answer, &a, &b);
16        int ans = *answer; // Q#7
17        free(answer);
18        return ans;
19      }
20
21      int main(void) {
22        printf("%d\n", add(MY_VALUE,6));
24        return 0;
25      }
```

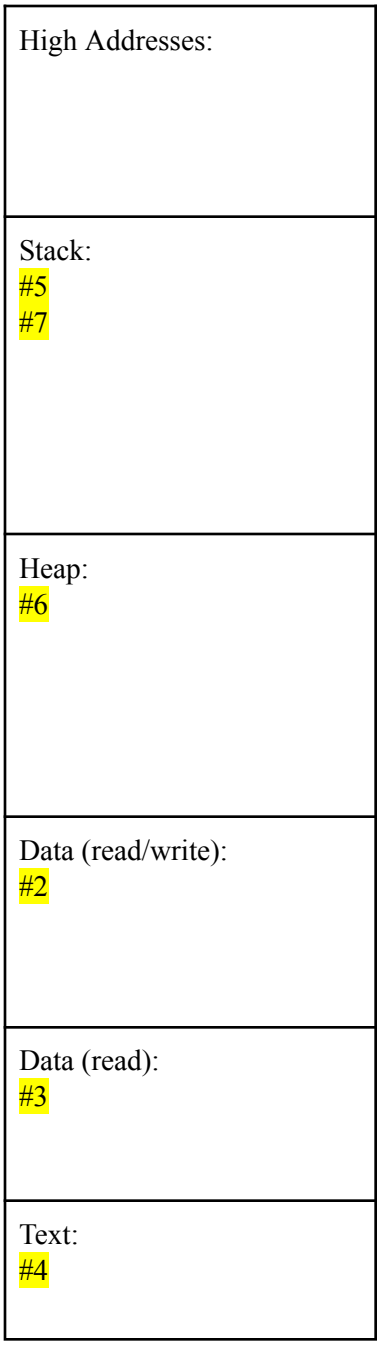
What is the construct called and where does it live in memory:

#1 MY_VALUE (preprocessed directive)
#2 g (global int)
#3 a_string (constant global)
#4 add_helper (function)
#5 c, d, and e (local variables)
#6 answer (dynamically allocated variable)
#7 ans (local variable)

What is the scope of each of #1 through #7?

#1 Not a variable, unscoped
#2 Global, 1-28
#3 Global, 1-28
#4 Global, 1-28
#5 local, 9-11
#6 local, 14-18 (note that free doesn't change scope, just marks the memory for the OS)
#7 local, 16-18

(Not to scale...duh)
Q#1 is a trick question since it's not a variable but rather a preprocessor text substitution.



Problem 2:

```
void while_one() { //#1
   while_one(); // #2
}

int main(void) {
   while (1) {
       while_one(); //#3
   }
}
```

1. **Of the three tokens `while_one` which has the greatest impact on memory at compilation time? at run time? why?**

#1 uses the most memory at compile time, since it is a function definition, while #2 and #3 are just calls to that function. When the program is run, #3 is only called once; it would be called infinitely often, but before main regains control, #2 is called repeatedly (infinitely) until we run out of memory. Each time through the function, a new stack frame is allocated (return values, args, locals) and since none of them ever return we run out of memory.

2. **What kind of error would you expect to receive if you ran this program? Why?**

Segmentation fault -- the program (not optimized for tail recursion) will continuously push new stack frames on the stack, and since none of them return, we will run out of memory space on the stack.

Problem 3:

Program:

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char* argv[]){
  char username[30];
  printf("Please enter your name:
");
  gets(username);
  printf("Your name is: %s\n",
username);
  return 0;
}
```

Shell:

```
$ Please enter your name:
```

1. **What's wrong with the code here?**

We are using the unsafe library call gets, which does not specify a length of the input and assigning that input to a buffer of a fixed size.

2. **As a programmer what kind of attack have you exposed yourself to? (see what's going on in the shell below!)**

Stack smashing. More specifically a buffer overrun. (The shell input is overwriting the return address of your function's stack frame!)

3. **How can you fix the code?**

Use safe functions such as fgets which allows the programmer to specify a maximum length for the

```
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy0x00
00f679
```

users input.
Followup: would canaries help you here?

Memory Alignment

While humans may think of memory as a continuous array of bytes, computers view memory as chunks of **word-sized** memory blocks. When you hear *x* bit system, like 32 bit or 64 bit, the number of bits refers to the word size. Thus, on a 64 bit system, words are 8 bytes each. This means that the CPU accesses memory in 8 byte chunks.

This can have important consequences for performance. Say, for example, that we have the following situation: (each bracket is a chunk of 8-byte memory, starting at address 0x8000000000000000).

[00 00 01 a3 24 f2 e3 45] [a1 00 01 a3 24 f2 00 00]

^ 0x8000000000000000 ^ 0x8000000000000008

If we store a pointer (which is 8 bytes on a 64 bit machine) using the last 4 bytes of one 8-byte aligned slot and the first 4 bytes of the next, then the processor would need to access both chunks memory to get the pointer, as opposed to only accessing one chunk of 8-byte memory if the pointer was **aligned** to the beginning of the word chunk.

Definition: a memory address *a* is said to be ***n-byte aligned*** when *a* is divisible by *n*.

As you may have seen working across different machines and environments, there is no single standard for C data type sizes, only certain guarantees. The following table lists different sizings (in bytes) for common standards LP, ILP, and LLP across both 32 and 64 bit systems. **The VM is an LP64 system**.

Data Type	LP32	ILP32	ILP64	LLP64	LP64
char	1	1	1	1	1
short	2	2	2	2	2
int32			4		
int	2	4	8	4	4
long	4	4	8	4	8
long long (int64)				8	
pointer	4	4	8	8	8

Source: http://www.unix.org/whitepapers/64bit.html

On most machines, the alignment will be equal to the size, but not always! For example, 64 bit pointers returned from malloc() are sometimes 16 bit aligned to account for vectors. Moral of the story, never assume, and always check via the built-in ***sizeof(TYPE)***.

Derived Data Type Alignment:

- Arrays: Same alignment of the data type of the array elements
- Structs:
    - Each member of a struct will be aligned based on their type
    - Variables inside a struct are laid out in the order that they are declared
    - ***align(struct)***, the final alignment of a struct, is the lcm of the alignment of its component variables
    - ***sizeof(struct)***, the memory size of the struct, is the smallest number that is a multiple of *align(struct)* that holds the data structure when each individual component variable is correctly aligned. Can you think of why?

```
struct data {
    char data1;
    short data2;
    int data3;
}

align(data) = 4, while sizeof(data) = 8.
```

Malloc also has its own alignment, usually either 8 or 16 bytes.

Exercise 1

Write down the size and the alignment of the following structs. Assume an LP64 architecture. Hint: draw out how an instance of the struct would look in memory.

`` struct struct1 { int i; } ``` Alignment: 4 Size: 4	``` struct struct2 { short a; short b; short c; } ``` Alignment: 2 Size: 6
`` struct struct3 { char a; int b; } ``` Alignment: 4 Size: 8	``` struct struct4 { int b; char a; } ``` Alignment: 4 Size: 8
`` struct struct5 { char a; char b; int c; } ``` Alignment: 4 Size: 8	``` struct struct6 { char a; int b; char c; short d; int e; } ``` Alignment: 4 Size: 16

Exercise 2

Write a program which contains two structs made up of the same variable types but which have two different sizes in memory.

```
struct s1 {
  char char1;
  short short1;
  int int1;
  char char2;
};
```

```
struct s2 {
  char char1;
  char char2;
  short short1;
  int int1;
};
```

```
int main(int argc, char const *argv[]) {
  printf("Why these no the same: %d %d\n",
      sizeof(s1), sizeof(s2));
  return 0;
}
```

Why does this occur?

At compile time padding is added to all data structures to ensure that they are appropriately byte aligned for fast access from memory. That is, we like to have data end on word boundaries. It is bad for performance to have a small piece of data require two separate word reads.

More Assignment 1 Hints

Heavy hitters: don't freak out! The algorithms in the paper are not as complicated as you think. Read the intros of the papers, look at the algorithms, and pick out an algorithm to implement.

(From last time): Identify the Undefined Behavior!

(there are also some behaviors that perhaps aren't undefined, but will cause compiler warnings/failures)

```
int lotsOfUndefinedBehavior {
    int i = 0;
    int j;
    char* message = "Hello World";
    message[0] = 'J';
      // Cannot change a string literal (could be in read-only memory along
      // with the rest of the code, for instance)
    printf("%s", message);

    i++;
    printf("New value is: %d", i);

    j++; // Use of uninitialized variable
    printf("New value is: %d", j);

    char* p = (char*) malloc(sizeof(char)/sizeof(int));
    p[0] = 'a'; // malloc 0 bytes means this is a null ptr dereference!
    printf("Our string is: %s", p);

    int* nats = (int*) malloc(sizeof(int)*42);
    nats[0] = 1;
    for( int k; k < 43; k++ ) { // k was never initialized!
        nats[k] = 2*nats[k-1];
            // when k > 32, we get integer overflow.
            // when k > 42 or k < 0, accessing an out-of-bounds array element
    }
      // we don't return anything in a non-void function!
}
```