CSE 333 – SECTION 2

Memory Management, Valgrind, gdb

How are exercises going?

- In CSE333 exercise, watch out for these points:
- Comments
  - Program Comments – Author, copyright, problem description at the top
  - Function Comments – Near the prototype/declaration in header files
- `clint` or `cpplint` errors
  - Help you write standardized, clean codes.
  - However, we don't really care about some of its complaints. E.g. Sizeof() clint/cpplint errors can be ignored.
- Valgrind errors

Memory Management

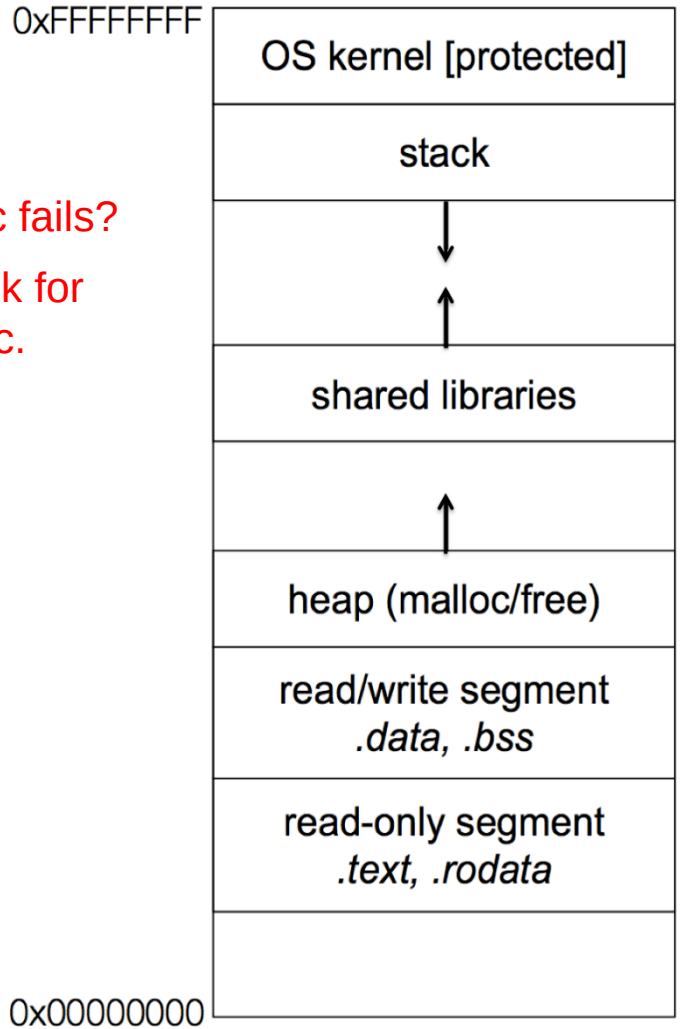
- Heap
  - Large pool of unused memory
  - malloc() allocates chunks of this memory

    Side note: What if malloc fails?

    In this class always check for the return value of malloc.
  - free() deallocates memory and reclaims space

- Stack and stack frame
  - Stores temporary/local variables
  - Each function has its own stack frame

- Lifetime on heap vs. Lifetime on stack



Symbol table

- .text(code)
- .data(contains initialized static variables, that is, global variables and static local variables)
- .rodata(*read-only data segment*)
- .bss(uninitialized data, both variables and constants)

Demo: buggy code

buggy.c demo + code fix

Some buggy code

https://courses.cs.washington.edu/courses/cse333/18wi/sections/sec2/buggy.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>

3.  // Returns an array containing [n, n+1, ... , m-1, m]. If n>m, then the
4.  // array returned is []. If an error occurs, NULL is returned.

5.  int *RangeArray(int n, int m) {
6.    int length = m-n+1;
7.
8.    // Heap allocate the array needed to return
9.    int *array = (int*) malloc(sizeof(int)*length);
10.
11.    // Initialize the elements
12.    for(int i=0;i<=length; i++)
13.      array[i] = i+n;

14.    return array;
15.  }

16.  // Accepts two integers as arguments
17.  int main(int argc, char *argv[]) {
18.    if(argc != 3) return EXIT_FAILURE;
19.    int n = atoi(argv[1]), m = atoi(argv[2]);  // Parse cmd-line args
20.    int *nums = RangeArray(n,m);

21.    // Print the resulting array
22.    for(int i=0; i<= (m-n+1); i++)
23.      printf("%d", nums[i]);
24.    puts("");

25.    return EXIT_SUCCESS;
26.  }
```

Valgrind output

```
==22891== Command: ./warmup 1 10
==22891==
==22891== Invalid write of size 4
==22891== at 0x400616: RangeArray (warmup.c:14)
==22891== by 0x400683: main (warmup.c:22)
==22891== Address 0x51d2068 is 0 bytes after a block of size 40 alloc'd
==22891== at 0x4C2A93D: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22891== by 0x4005EC: RangeArray (warmup.c:10)
==22891== by 0x400683: main (warmup.c:22)
==22891==
==22891== Invalid read of size 4
==22891== at 0x4006A5: main (warmup.c:26)
==22891== Address 0x51d2068 is 0 bytes after a block of size 40 alloc'd
==22891== at 0x4C2A93D: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22891== by 0x4005EC: RangeArray (warmup.c:10)
==22891== by 0x400683: main (warmup.c:22)
==22891==
1 2 3 4 5 6 7 8 9 10 11
==22891==
==22891== HEAP SUMMARY:
==22891== in use at exit: 40 bytes in 1 blocks
==22891== total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==22891==
==22891== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==22891== at 0x4C2A93D: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22891== by 0x4005EC: RangeArray (warmup.c:10)
==22891== by 0x400683: main (warmup.c:22)
==22891==
==22891== LEAK SUMMARY:
==22891== definitely lost: 40 bytes in 1 blocks
==22891== indirectly lost: 0 bytes in 0 blocks
==22891== possibly lost: 0 bytes in 0 blocks
==22891== still reachable: 0 bytes in 0 blocks
==22891== suppressed: 0 bytes in 0 blocks
==22891==
==22891== For counts of detected and suppressed errors, rerun with: -v
==22891== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 3 from 3)
```

Valgrind errors

- An `Invalid read` means that the memory location that the process was trying to read is outside of the memory addresses that are available to the process. size 8 means that the process was trying to read 8 bytes. On 64-bit platforms this could be a pointer, but also for example a long int.

Valgrind memory leak report

- "definitely lost" means your program is leaking memory -- fix those leaks!
- "indirectly lost" means your program is leaking memory in a pointer-based structure. (E.g. if the root node of a binary tree is "definitely lost", all the children will be "indirectly lost".) If you fix the "definitely lost" leaks, the "indirectly lost" leaks should go away.
- "possibly lost" means your program is leaking memory, unless you're doing unusual things with pointers that could cause them to point into the middle of an allocated block; see the user manual for some possible causes.
- "still reachable" means your program is probably ok -- it didn't free some memory it could have. This is quite common and often reasonable. Don't use --show-reachable=yes if you don't want to see these reports.
- "suppressed" means that a leak error has been suppressed. There are some suppressions in the default suppression files. You can ignore suppressed errors.

- What are some common complaints?

- Invalid writes
- Invalid reads
- Use of uninitialized memory

- For Explanation of error messages, refer to the Valgrind user manual
- http://valgrind.org/docs/manual/mc-manual.html

Note from section:
C99 standard: **free**(**NULL**) is guaranteed to be safe. Checking just adds unnecessary clutter to your code.

Memory Errors

- Use of uninitialized memory
- Reading/writing memory after it has been freed – Dangling pointers
- Reading/writing to the end of malloc'd blocks
- Reading/writing to inappropriate areas on the stack
- Memory leaks where pointers to malloc'd blocks are lost
- Mismatched use of malloc/new/new[] vs free/delete/delete[]

Valgrind is your friend!!

gdb

- How to debug using gdb
- Set a breakpoint there with the GDB **break (b)** command.
- Using the **run (r)** command, to start running under GDB control
- Use the command **next (n)** to advance execution to the next line of the current function.
- We can go into a subroutine by using the command **step (s)** instead of next.
- Use the command **print (p)** to see their values.
- Use the **backtrace (bt)** command to see where we are in the stack as a whole: the **backtrace** command displays a stack frame for each active subroutine.