# My code doesn't crash – why should I still use Valgrind?

Tyson Whitehead

April 16, 2014

## Notes

Modern digital computers are binary systems.

- frequently numbers are powers of two
- 1000 multiple is usually 1024 instead
- will not bother distinguishing

## Memory

Physical computer memory is a large 1D byte array.

- 1B = number between 0-255
- 8GB = 8 billion byte entries

Linux divides 1D array into chunks called pages.

- 4KB = 4 thousand byte entries
- pages can be read, write, and/or execute

## Application

Program memory is a sparse 1D byte array.

- physical memory mapped in in page sized chunks
- not all indices may be accessed (segfault)
- not all incorrect access may an invalid index (segfault)

# Diagram

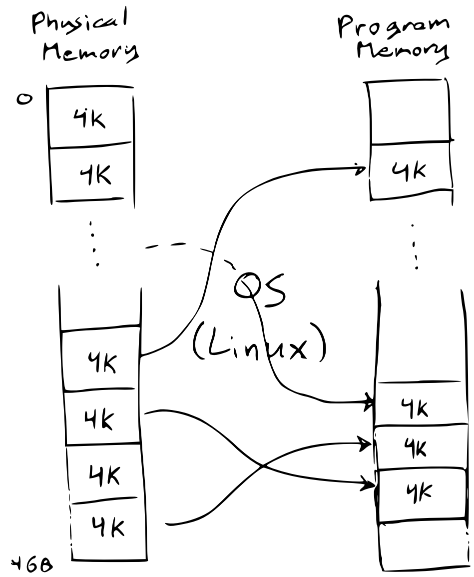


Figure 1: Physical memory

# Layout

Program memory (1D byte array) broken up into

- null catch area (unmapped)
- code (read/execute)
- constant data (read)
- mutable data (read/write)
- heap (read/write)
- code, constant and mutable data for libraries
- stack (read/write)
- kernel interface (read/execute)

Linux does not care as long as index is valid for operation

**executable layout** `readelf -t` *EXE*
**process layout** `cat /proc/`*PID*`/maps`

# Diagram

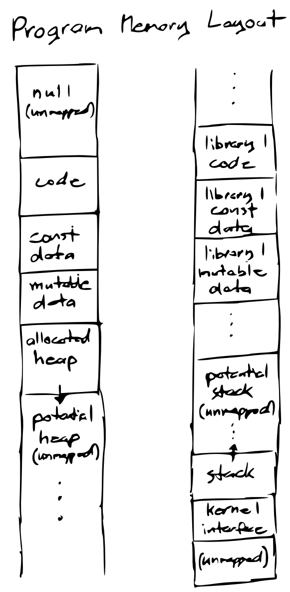


Figure 2: Program memory

# Heap

Area of read/write memory for dynamic memory allocation

- expanded by mapping new pages to bottom
- managed by GNU C library (glibc) via malloc and free
- includes records for tracking allocations
- allocating and releasing leaves holes

What can go wrong

- allocating without releasing will eventually exhaust memory
- releasing non-allocated memory will mess up glibc
- invalid reads will return other data unless outside entire region
- invalid writes will overwrite other data unless outside entire region
- other data includes glibc memory management structures

*source of problem may not be where program dies*

## Diagram

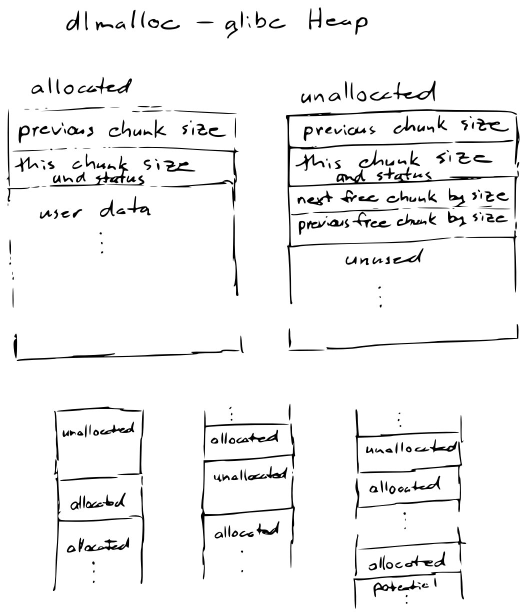


Figure 3: Heap memory

## Stack

Area of read/write memory for handling function calls

- expanded by mapping new pages to top
- addresses of calling function for return
- arguments passed to functions
- local variables used by functions

What can go wrong

- invalid reads will return other data unless outside entire region
- invalid writes will overwrite other data unless outside entire region
- other data includes return addresses

*source of problem may not be where program dies*

# Diagram

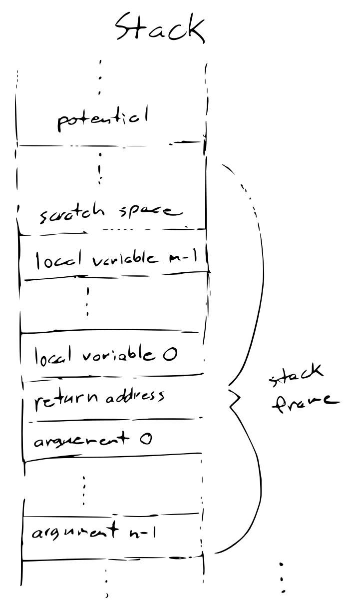


Figure 4: Stack memory

# Valgrind

Dynamic binary instrumentation framework

- dynamically translates executables to add instrumentation
- tracks all memory and register usages by a program

**memcheck** memory error detector
**cachegrind** cache and branch-prediction profiler
**callgrind** call-graph generating cache and branch prediction profiler
**helgrind** thread error detector
**DRD** thread error detector
**Massif** heap profiler
**DHAT** dynamic heap analysis tool
**SGCheck** experimental stack and global array overrun detector
**BBV** experimental basic block vector generation tool

## Usage

Advantages

- can be directly run on any executable
- dynamic translation allows ultimate instrumentation

Disadvantages

- 5-100 x slow down depending on tool
- 12-18 x increase in size of translated code
- corner cases may exist between translated code and original

*run on small test cases – can save hours and hours of debugging*

## MemCheck

Default valgrind tool that detect several common memory errors

- overrunning and underrunning heap blocks
- overrunning top of stack
- continuing to access released memory
- using uninitialized values
- incorrectly using memory copying routines
- incorrectly paired allocation/release calls
- relasing unallocated memory
- not releasing memory