CS168: Debugging

Introduction to GDB, Wireshark and Valgrind

GDB: The GNU Debugger

- gdb is an executable file that serves as a portable debugger
  - Works for Ada, C, C++, Objective-C, Pascal, and others
- Useful for:
  - If a crash happens, then what statement or expression did the program crash on?
  - If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?
  - What are the values of program variables, or results of function calls at a particular point in the program?
- If you've used the Eclipse debugger for Java projects, GDB provides many of the same facilities in a command line interface.

Program Compilation

- To let GDB be able to read all debugging information from the symbol table, we need to compile our programs a bit differently. Normally we compile things as:
  - gcc –o hello hello.c
- Instead of doing this, we need to compile with the -g flag:
  - gcc –g –o hello hello.c
  - We recommend adding this flag to your Makefile.
- If you're debugging something particularly tricky, you may want to turn off optimization as well.
  - gcc –O0
  - Optimization changes your code! Single stepping in the debugger might not step to the next source line.
  - Many systems have Makefiles with debug + release modes.

Analyzing a Segfault

- The linux kernel is able to write a so called “core dump” if the application crashes. The core dump records the state of the process at the time of the crash.
- gdb can read core dumps to assist with debugging after the crash.
- First you must enable core dumps.
  - ulimit -c unlimited
    - Controls the resources available to a process started by the shell.
    - Try this on a department machine:
      - -bash: ulimit: core file size: cannot modify limit: Operation not permitted
      - You must run ulimit as root. “sudo ulimit –c unlimited”
- Upon segmentation fault:
  - "Segmentation fault (core dumped)"

Analyzing a Segfault

- Upon segfault you'll find a file named "core" or "core.*pid*" in the current directory.
- Run gdb, providing the executable that caused the crash, as well as the core dump file.
  - gdb hello core.65782
- gdb will analyze the debugging information in the executable along with the core dump, then greet you with a prompt:
  - (gdb)
- From here you can inspect the state of the program at crash time.
  - (gdb) bt
    - Produces a stack trace. We'll look at more useful commands soon.

Using GDB

- To run your program under the debugger:
  - gdb hello
- There are then three ways to run ‘hello’:
  - Run it directly.
    - (gdb) r
  - Pass arguments.
    - (gdb) r arg1 arg2
  - Feed in a file.
    - (gdb) r < file1
- Before running your program, you may want to set breakpoints!
  - b hello.c:50

Using GDB

- If your program has no serious problems (i.e. the normal program didn't have a segmentation fault, etc.) and you haven't set any breakpoints, the program should run as you'd expect under the debugger.
- If the program does encounter issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:
  - Program received signal SIGSEGV, Segmentation fault. 0x0000000000400524 in sum array region (arr=0x7fffc902a270, r1=2, c1=5, r2=4, c2=6) at sum-array-region2.c:12
- When the program crashes or hits a breakpoint, the gdb interactive prompt enables you to issue commands.

GDB Commands

- b main – Put a breakpoint at the beginning of the program
- b – Put a breakpoint at the current line
- b N – Put a breakpoint at line N
- b +N – Put a breakpoint N lines down from the current line
- b fn – Put a breakpoint at the beginning of function "fn"
- d N – delete breakpoint number N
- info break – list breakpoints
- r – Run the program until a breakpoint or error
- c – continue running the program until the next breakpoint or error
- f – Run until the current function is finished
- s – run the next line of the program
- s N – run the next N lines of the program
- n – like s, but don't step into functions
- p var – print the current value of the variable "var"
- bt – print backtrace
- frame num – change stack frame to the number provided
- q - Quit gdb

GDB Commands

- If you're ever confused about a command or just want more information, use the "help" command, with or without an argument:
  - (gdb) help [command]
  - You should get a nice description and maybe some more useful tidbits.

Attaching to a Running Process

- GDB has capabilities that allow you to debug an already-running process.
- Run gdb.
  - (gdb) attach <pid>
  - Find pid with ‘ps’
    - ps -u spoletto
- The first thing GDB does after arranging to debug the specified process is to stop it.
- You can examine and modify an attached process with all the GDB commands that are ordinarily available.
  - You can insert breakpoints; you can step and continue.
  - If you would rather the process continue running, you may use the ‘c’ command after attaching GDB to the process.

Watchpoints

- Whereas breakpoints interrupt the program at a particular line or function, “watchpoints” act on variables. They pause the program whenever a watched variable’s value is modified.
  - (gdb) watch my_var
    - Now, whenever my_var’s value is modified, the program will interrupt and print out the old and new values.
  - Watchpoints are dependent upon scope.
    - If you have two variables named my_var, it will watch the variable in the current scope.

Conditional Breakpoints

- Just like regular breakpoints, except that you get to specify some criterion that must be met for the breakpoint to trigger. We use the same ‘b’ command as before:
  - (gdb) b file1.c:6 if i >= ARRAYSIZE
    - Triggers only if the variable i is greater than or equal to the size of the array (which probably is bad if line 6 does something like arr[i]).

Printing Data

- You can use pointer operations within the debugger, very similar to how you would in C:
  - struct product *p1 = <something>;
  - (gdb) p p1
    - Prints the value (memory address) of the pointer.
  - (gdb) p p1->name
    - Prints a particular field of the struct.
  - (gdb) print *p1
    - See the entire contents of the struct.
      - You can't do this easily in C code!
  - (gdb) print list->prt->next->next->next->data

Examining Stack Frames

- (gdb) list
  - Print ten lines of code after or around the current breakpoint.
- (gdb) info locals
  - Print local variables of current stack frame.
- (gdb) info args
  - Print argument variables of current stack frame.

- When combined with ‘p’, ‘n’ and ‘up’, these commands will save your life.


Networks — gdb-i386-apple-d — 235×69
bash
bash
snowcast_server
gdb-i386-apple-d
ssh
Stephens-MacBook-Pro:Networks spoletto$ gcc -g -o gdb_example gdb_example.c
Stephens-MacBook-Pro:Networks spoletto$ ./gdb_example
Khoor/#zruog1
Segmentation fault: 11
Stephens-MacBook-Pro:Networks spoletto$ gdb gdb_example
GNU gdb 6.3.50-20050815 (Apple version gdb-1708) (Mon Aug 8 20:32:45 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin"...Reading symbols for shared libraries .. done
(gdb) r
Starting program: /Users/spoletto/Dropbox/Networks/gdb_example
Reading symbols for shared libraries +..... done
Khoor/#zruog1
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x0000000000000000
0x0000000100000e67 in print_scrambled (message=0x0) at gdb_example.c:13
13 printf("%c", (*message)+i);
(gdb)

DEMO

Wireshark

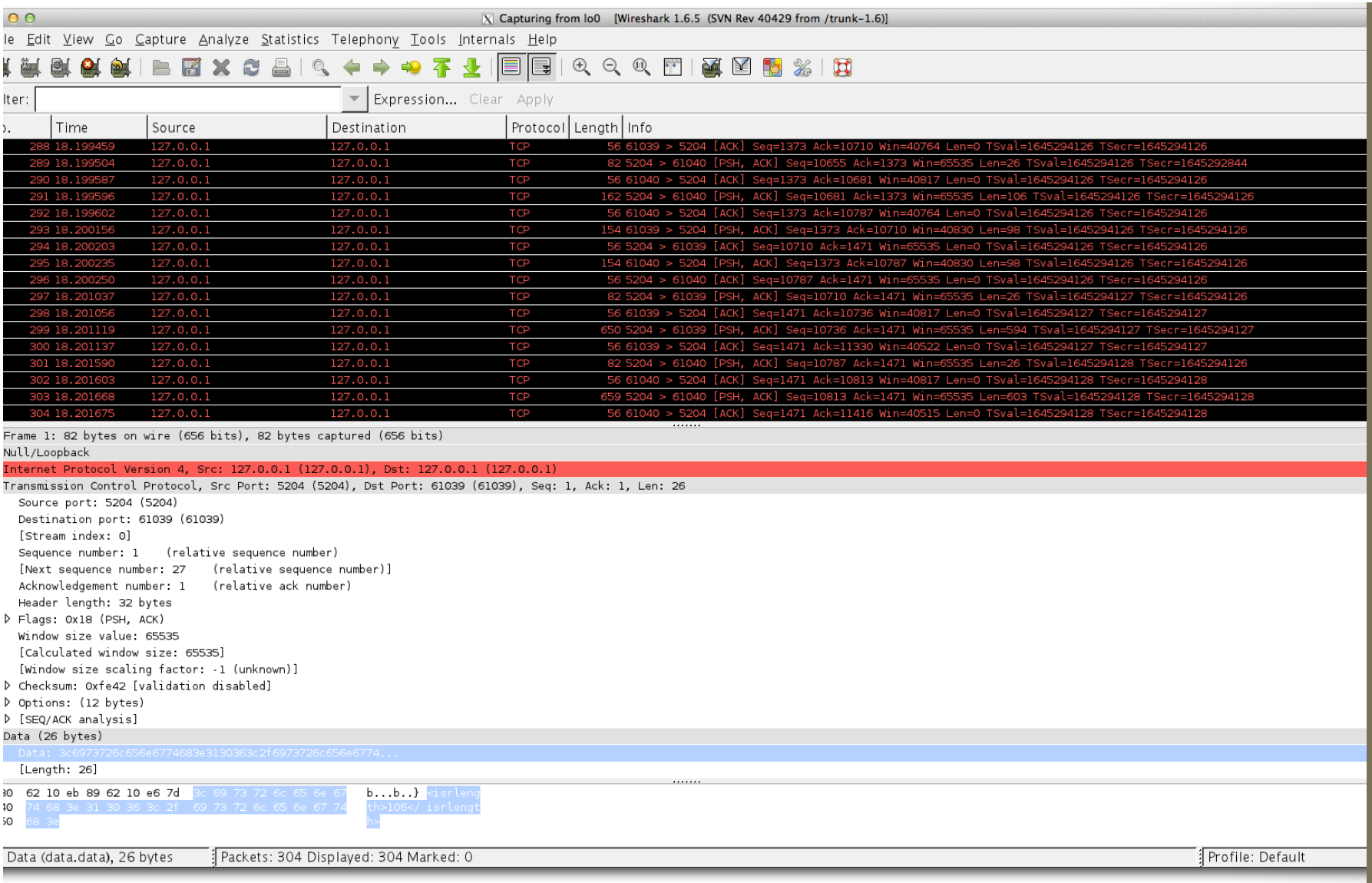
- Free and open-source packet analyzer.
  - Allows you to intercept packets from the network interface controller.
  - Useful for network troubleshooting and protocol development (i.e. when implementing the IP and TCP protocols).
- Setup
  - Download and install Wireshark from http://www.wireshark.org/
  - Running Wireshark (or any other network capture/analyzer, for that matter) on Linux needs root privileges.
    - Can't do this on department machines.
    - Have you been convinced to set up your own VM yet?

Wireshark Setup

- Capture -> Options
  - Pick the network interface (NIC) to listen on.
    - On a Mac, the BSD Device Name for the AirPort interface is "**en1**", the BSD Device Name for Ethernet is "**en0**"
    - But if we're sending packets locally (i.e. just using localhost), we don't want to snoop the real network devices. Instead, we want to snoop the loopback interface "**lo0**".
  - Click "Options" next to the device name.
    - Turn off "Promiscuous Mode."
      - In non-promiscuous mode, when a NIC receives a frame, it normally drops it unless the frame is addressed to that NIC's MAC address or is a broadcast or multicast frame.
      - In promiscuous mode, the card allows all frames through, thus allowing the computer to read frames intended for other machines or network devices.

Capture Filters

- We need to create a capture filter to prevent Wireshark from capturing all network traffic going through the interface we chose.
  - A lot of traffic goes through the network interfaces. We probably don't want to see all of it!
- In the text field next to the "Capture Filter" button, type: "port <port_number> and host <ip_address>"
  - This will create a filter that intercepts only traffic either originating from or going to the specified host, on the specified port.
  - If you're using the loopback interface, you don't need to specify the the IP address.
  - If you've set up your IP implementation properly, the port number specified should match what's in <filename>.lnx


Capturing from lo0 [Wireshark 1.6.5 (SVN Rev 40429 from /trunk-1.6)]
le Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
lter: Expression... Clear Apply
). Time Source Destination Protocol Length Info
288 18.199459 127.0.0.1 127.0.0.1 TCP 56 61039 > 5204 [ACK] Seq=1373 Ack=10710 Win=40764 Len=0 TSval=1645294126 TSecr=1645294126
289 18.199504 127.0.0.1 127.0.0.1 TCP 82 5204 > 61040 [PSH, ACK] Seq=10655 Ack=1373 Win=65535 Len=26 TSval=1645294126 TSecr=1645292844
290 18.199587 127.0.0.1 127.0.0.1 TCP 56 61040 > 5204 [ACK] Seq=1373 Ack=10681 Win=40817 Len=0 TSval=1645294126 TSecr=1645294126
291 18.199596 127.0.0.1 127.0.0.1 TCP 162 5204 > 61040 [PSH, ACK] Seq=10681 Ack=1373 Win=65535 Len=106 TSval=1645294126 TSecr=1645294126
292 18.199602 127.0.0.1 127.0.0.1 TCP 56 61040 > 5204 [ACK] Seq=1373 Ack=10787 Win=40764 Len=0 TSval=1645294126 TSecr=1645294126
293 18.200156 127.0.0.1 127.0.0.1 TCP 154 61039 > 5204 [PSH, ACK] Seq=1373 Ack=10710 Win=40830 Len=98 TSval=1645294126 TSecr=1645294126
294 18.200203 127.0.0.1 127.0.0.1 TCP 56 5204 > 61039 [ACK] Seq=10710 Ack=1471 Win=65535 Len=0 TSval=1645294126 TSecr=1645294126
295 18.200235 127.0.0.1 127.0.0.1 TCP 154 61040 > 5204 [PSH, ACK] Seq=1373 Ack=10787 Win=40830 Len=98 TSval=1645294126 TSecr=1645294126
296 18.200250 127.0.0.1 127.0.0.1 TCP 56 5204 > 61040 [ACK] Seq=10787 Ack=1471 Win=65535 Len=0 TSval=1645294126 TSecr=1645294126
297 18.201037 127.0.0.1 127.0.0.1 TCP 82 5204 > 61039 [PSH, ACK] Seq=10710 Ack=1471 Win=65535 Len=26 TSval=1645294127 TSecr=1645294126
298 18.201056 127.0.0.1 127.0.0.1 TCP 56 61039 > 5204 [ACK] Seq=1471 Ack=10736 Win=40817 Len=0 TSval=1645294127 TSecr=1645294127
299 18.201119 127.0.0.1 127.0.0.1 TCP 650 5204 > 61039 [PSH, ACK] Seq=10736 Ack=1471 Win=65535 Len=594 TSval=1645294127 TSecr=1645294127
300 18.201137 127.0.0.1 127.0.0.1 TCP 56 61039 > 5204 [ACK] Seq=1471 Ack=11330 Win=40522 Len=0 TSval=1645294127 TSecr=1645294127
301 18.201590 127.0.0.1 127.0.0.1 TCP 82 5204 > 61040 [PSH, ACK] Seq=10787 Ack=1471 Win=65535 Len=26 TSval=1645294128 TSecr=1645294126
302 18.201603 127.0.0.1 127.0.0.1 TCP 56 61040 > 5204 [ACK] Seq=1471 Ack=10813 Win=40817 Len=0 TSval=1645294128 TSecr=1645294128
303 18.201668 127.0.0.1 127.0.0.1 TCP 659 5204 > 61040 [PSH, ACK] Seq=10813 Ack=1471 Win=65535 Len=603 TSval=1645294128 TSecr=1645294128
304 18.201675 127.0.0.1 127.0.0.1 TCP 56 61040 > 5204 [ACK] Seq=1471 Ack=11416 Win=40515 Len=0 TSval=1645294128 TSecr=1645294128
Frame 1: 82 bytes on wire (656 bits), 82 bytes captured (656 bits)
Null/Loopback
Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
Transmission Control Protocol, Src Port: 5204 (5204), Dst Port: 61039 (61039), Seq: 1, Ack: 1, Len: 26
Source port: 5204 (5204)
Destination port: 61039 (61039)
[Stream index: 0]
Sequence number: 1 (relative sequence number)
[Next sequence number: 27 (relative sequence number)]
Acknowledgement number: 1 (relative ack number)
Header length: 32 bytes
Flags: 0x18 (PSH, ACK)
Window size value: 65535
[Calculated window size: 65535]
[Window size scaling factor: -1 (unknown)]
Checksum: 0xfe42 [validation disabled]
Options: (12 bytes)
[SEQ/ACK analysis]
Data (26 bytes)
Data: 3c6973726c656e6774683e3130363c2f6973726c656e6774...
[Length: 26]
62 10 eb 89 62 10 e6 7d 3c 69 73 72 6c 65 6e 67 b...b..} <isrleng
74 68 3e 31 30 36 3c 2f 69 73 72 6c 65 6e 67 74 th>106</ isrlengt
68 3e h>
Data (data.data), 26 bytes
Packets: 304 Displayed: 304 Marked: 0
Profile: Default

DEMO

Valgrind

- Valgrind is a memory mismanagement detector.
- Can detect:
  - Use of uninitialised memory
  - Reading/writing memory after it has been free'd
  - Reading/writing off the end of malloc'd blocks
  - Reading/writing inappropriate areas on the stack
  - Memory leaks -- where pointers to malloc'd blocks are lost forever
  - Mismatched use of malloc vs free
  - Overlapping src and dst pointers in memcpy() and related functions
  - Some misuses of the POSIX pthreads API

Running Valgrind

- valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-callers=20 --track-fds=yes ./myProg

Illegal read/write errors

- Invalid read of size 4

  at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)

  by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)

  by 0x40B07FF4: read_png_image__FP8QImageIO (kernel/qpngio.cpp:326)

  by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)

  Address 0xBFFFF0E0 is not stack'd, malloc'd or free'd

- This happens when your program reads or writes memory at a place which Memcheck reckons it shouldn't.
  - If it points into a block of memory which has already been freed, you'll be informed of this, and also where the block was free'd at.
  - Likewise, if it is just off the end of a malloc'd block, a common result of off-by-one-errors in array subscripting, you'll be informed of this fact, and also where the block was malloc'd.

Use of uninitialized values

- Conditional jump or move depends on uninitialised value(s)

  at 0x402DFA94: _IO_vfprintf (_itoa.h:49)

  by 0x402E8476: _IO_printf (printf.c:36)

  by 0x8048472: main (tests/manuel1.c:8)

  by 0x402A6E5E: __libc_start_main (libc-start.c:129)

- int main()

```
{
    int x;
    printf ("x = %d\n", x);
}
```

Use of values in syscalls

- Memcheck checks all parameters to system calls:
  - It checks all the direct parameters themselves, whether they are initialized.
  - If a system call needs to read from a buffer provided by your program, Memcheck checks that the entire buffer is addressable and its contents are initialized.
  - If the system call needs to write to a user-supplied buffer, Memcheck checks that the buffer is addressable.

Illegal frees

- Invalid free()

  at 0x4004FFDF: free (vg_clientmalloc.c:577)

  by 0x80484C7: main (tests/doublefree.c:10)

  Address 0x3807F7B4 is 0 bytes inside a block of size 177 free'd

  at 0x4004FFDF: free (vg_clientmalloc.c:577)

  by 0x80484C7: main (tests/doublefree.c:10)
- Memcheck keeps track of the blocks allocated by your program with malloc, so it can know exactly whether or not the argument to free is legitimate or not.
- Here, this test program has freed the same block twice. As with the illegal read/write errors, Memcheck attempts to make sense of the address freed.

Memory leak detection

- Memcheck keeps track of all heap blocks issued in response to calls to malloc(). So when the program exits, it knows which blocks have not been freed.
- For each remaining block, Memcheck determines if the block is reachable from pointers within the root-set.
- The root-set consists of:
  - General purpose registers of all threads
  - Initialized, aligned, pointer-sized data words in accessible client memory, including stacks.

Leak summary messages

- "Still reachable”
  - A start-pointer or chain of start-pointers to the block is found.
  - Since the block is still pointed at, the programmer could, at least in principle, have freed it before program exit.
  - Very common and arguably not a problem.
- "Definitely lost”
  - This means that no pointer to the block can be found.
  - The block is classified as "lost", because the programmer could not possibly have freed it at program exit, since no pointer to it exists.
  - This is likely a symptom of having lost the pointer at some earlier point in the program.
  - Such cases should be fixed.

Leak summary messages

- "Indirectly lost".
  - This means that the block is lost, not because there are no pointers to it, but rather because all the blocks that point to it are themselves lost. For example, if you have a binary tree and the root node is lost, all its children nodes will be indirectly lost.
  - Because the problem will disappear if the definitely lost block that caused the indirect leak is fixed, Memcheck won't report such blocks individually unless --show-reachable=yes is specified.
- "Possibly lost"
  - Complicated. Read about it in Valgrind's documentation.
  - In general, you shouldn't have "Possibly lost" blocks in your program.

Debugging with Valgrind

- If you see garbled stack contents in your debug messages, it's time to fire up Valgrind.
  - "TCP header length is 800192543223"
- Useful for profiling your programs before submission.
  - Part of your grade depends on code quality, including proper memory management.
  - Inspect your program for memory leaks before turning in.