gdb-2 Tutorial

Ανάργυρος Αργυρός
Αλέξανδρος Τεβρεντζίδης

Recap on gdb-1

GNU project's Debugger:

- Allows us to inspect the runtime state of a process
- Useful for testing our program and finding bugs
- Compatible with many languages (C, C++, Rust, Go, **Assembly** and more!)

Running an application with gdb

You can find gdb installed on all of the departments machines.

1. Compile your application with debug symbols (-g option for gcc)
2. Run gdb with: gdb <executable_name>
3. Begin execution with gdb's "run" command
4. ???
5. profit
6. Exit gdb using ctrl + D

Basic gdb utilities recap

- breakpoint(b):
  - Execution will pause at breakpoint locations allowing you to inspect the state
  - Place breakpoints with the break command eg:
    - break main -> place breakpoint at the beginning of “main” function
    - break main.c:15 -> place breakpoint at line 15 of main.c
    - break -> place breakpoint at current location
    - break *<addr> -> place breakpoint at code in address
- list(l):
  - Display code around given line number or function
    - list main -> show “main” function’s code
    - list -> show code currently being executed
- backtrace(bt):
  - Display call stack -> Which functions have been called up to this point? (and have not returned yet)
- info(i):
  - Display info about various resources (breakpoints, local variables, arguments ,functions, threads + many more)

Program execution with gdb

After setting a breakpoint we can:

- next(n) -> execute current line and move to the next
- nextinstruction(ni) -> execute next asm instruction and move to the next
- continue(c) -> let program continue execution
- delete(d) -> delete a breakpoint

Inspecting memory with gdb

Print command:

- Prints the value of a variable or expression
  - C-like syntax and understanding of C entities
  - print i -> prints variable i
  - e.g print fib(10 + fib(3)) -> evaluates expression and prints result

Inspecting memory with gdb

x command (examine):

- Displays the memory contents at a given address using the specified format.
- syntax: x/[Length][Format] [Address]
  - Length -> number of elements that will be displayed
  - Format -> “type” of element, if not specified gdb will try to guess using debug info
- e.g
  - x &my_int -> prints value of int
  - x/f &my_int -> prints value of int interpreted as a float value
  - x/10 my_int_array -> prints 10 integer elements starting at address of my_int_array
  - x/16x $rsp -> prints 16 bytes of memory from the address pointer to by the stack pointer register -> essentially prints contents of current stack frame
  - x/16i main -> prints the 16 first (asm) instructions of “main” function

Modifying memory with gdb

set command:

- Assign values to variables
  - e.g. set var i = 10 -> sets value of variable i to 10
  - set *<mem_addr> = 10 -> sets contents of memory address <mem_addr> to value 10
  - set ***(int** *)**my_int_ptr = 10 -> set value of memory pointed to by my_int_ptr to 10

Navigating the stack with gdb

Each function has its local variables stored in its stack frame

We can navigate up / down the current stack frames with the **up, down** and **frame <#>** commands.

- up command:
  - Navigate to the caller stack frame.
- down command:
  - Navigate to calling stack frame
- frame command:
  - Directly jump to a specific frame, use backtrace to see all frames along with the numbers that correspond to them

Helpful links

- gdb cheat sheet: https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf
- gdb command reference: https://visualgdb.com/gdbreference/commands/
- gdb documentation: https://sourceware.org/gdb/documentation/