CS 240 Programming in C

GNU Debugger (gdb)

October 16, 2019
A debugger is a program that simulates/runs another program which can help you find where goes wrong of your program. Specifically it allows:

- Pause and continue its execution
- Set "break points" or conditions where the execution pauses so you can look at its state
- View and "watch" variable values
- Trace back segmentation faults
- etc.

GNU Debugger (gdb) https://www.gnu.org/software/gdb/

GNU https://en.wikipedia.org/wiki/GNU

For small and trivial verifying and debugging, we can always just use 'printf' statements even though it is often referenced as the "poor man’s" debugging.
Getting Started

- Compile for debugging:
  
  ```
  gcc -g program.c
  ```

- The flag ‘-g’ preserves all the identifiers symbols of variables and functions in program which are lost when the program gets compiled, such that we can utilize when we are debugging.
A segmentation fault (aka segfault) is a common condition that causes programs to crash.

They are often associated with a file named core or core dump. https://wiki.archlinux.org/index.php/Core_dump

Core dumps may be produced on-demand (such as by a debugger), or automatically upon fault termination.

System usually disable automatic core dumps:
- Performance: generating core dumps for memory-heavy processes can waste system resources and delay the cleanup of memory.
- Disk space: core dumps of memory-heavy processes may consume disk space equal to, if not greater, than the process’s memory footprint if not compressed.
- Security: core dumps, although typically readable only by root, may contain sensitive data (such as passwords or cryptographic keys), which are written to disk following a crash.

The maximum core dump size is enforced by ulimit, and it is often set to be zero as default, and this is why you don’t see it when segfault happens.
We will just debug from the source code.

Segfaults are caused by a program trying to read or write an illegal memory location, for example, an array in a stack/data segment reaches out the boundary of a stack/data segment.

Program memory is divided into different segments:
- a text segment for program instructions,
- a data segment for variables and arrays defined at compile time,
- a stack segment for temporary (or automatic) variables defined in subroutines and functions
- a heap segment for variables allocated during runtime by functions, such as malloc (in C) and allocate (in Fortran).

You don’t have to know all these segments, just understand what segfaults means.
when segfaults happens, using backtrace/where command in gdb will show you in which line your program went seg fault.

A backtrace is a summary of how your program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.

This is a little early to get directly to backtrace, but I want to put it together with segfaults so you can easily find the solution to problem later.

Anyway we will get there gradually, now let’s starting from beginning of the hands-on gdb.
- There is a cheat sheet post on class page.
- Start gdb with program.
  
  \[
  \text{gdb [program]} \\
  \text{file program}
  \]
breakpoints:
http://visualgdb.com/gdbreference/commands/
  - break
  - clear
  - disable
  - disable
  - enable
  - info breakpoints
debugged program settings:
http://visualgdb.com/gdbreference/commands/

- set args
- info args
Execution control commands
http://visualgdb.com/gdbreference/commands/

- continue
- until
- kill
- start
- run
Expression evaluating commands
http://visualgdb.com/gdbreference/commands/