gdb – The GNU Debugger

Kurt Schmidt

Dept. of Computer Science, Drexel University

November 28, 2018
gdb – The GNU Debugger

Kurt Schmidt

Intro

Invocation

Commands
Getting Help
Essentials
Running
Listing
Breakpoints
Execution
Data
Call Stack
Trickier

Corefiles
Summary
A debugger is closely tied to the compiler

- *gdb* is the command-line debugger for all GNU compilers
  - Language is irrelevant
  - Back end of the compiler is the same (for a given platform)
  - An executable is just a program; it’s not a “C program”, nor a “FORTRAN program”, etc.
Invocation
First, use the –g option, compile your program with extra (debugging) information

```bash
$ gcc -g source files... -o prog
```

Then, load the executable into the debugger:

```bash
$ gdb prog
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
...
(gdb) _
```
Commands
Using GDB

- GDB is very powerful
  - Attach to a running process
  - Examine a corefile
  - Debug multi-threaded programs
- Lots of commands
  - Don’t be intimidated
  - I don’t know many of them
  - Just knowing some of the basics will get you far
GDB commands are divided into categories

Type `help` to see these categories:

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without ...
user-defined -- User-defined commands
```
To see commands in a category (class):

(gdb) help running
Running the program.

List of commands:

continue -- Continue program being debugged
finish -- Execute until selected stack frame returns
jump -- Continue program being debugged at specified ...
kill -- Kill execution of program being debugged
next -- Step program
run -- Start debugged program
start -- Run the debugged program until the beginning ...
step -- Step program until it reaches a different source line

I’ve only listed some of the handier commands
Use `help cmd` for help on that command:

```
(gdb) help break
Set breakpoint at specified location.
break [PROBE_MODIFIER] [LOCATION] [thread THREADNUM] [if CONDITION]
PROBE_MODIFIER shall be present if the command is to be placed in a
probe point. Accepted values are ‘-probe’ (for a generic,
automatically guessed probe type), ‘-probe-stap’ (for a SystemTap
probe) or ‘-probe-dtrace’ (for a DTrace probe).
LOCATION may be a linespec, address, or explicit location as
described below.

With no LOCATION, uses current execution address of the selected
stack frame. This is useful for breaking on return to a stack
frame.

THREADNUM is the number from "info threads".
CONDITION is a boolean expression.
...```
Note, many of the commands can be abbreviated.

```
break b [location]  Set breakpoint
kill                Kill running process
run [arglist]       Run your program
print p [expr]      Print $expr$
step s              Next line, stepping into functions
next n              Next line, stepping over functions
continue c         Continue to next break
quit q              Exit GDB
```
Running Your Program

- `set args args` Set command-line arguments
- `set env var val` Set environment `var` to `val` (for next run)
- `show args` Show command-line args
- `show env [var]` Show environment variables [or `var`]
- `run [args]` Run your program [with `args`]
- `start [args]` Run your program until beginning of main procedure
- `kill` Kill running process
Looking at Your Code

```
list or l
  * list
  * list line_no
  * list beg,end
  * list file:line_no
  * list func_name
```
Setting Breakpoints

- A place (and/or condition) where execution pauses, waits for a user command
- Can break conditionally at a function or a line number
  - `break func_name`
  - `break line_no`
  - `break file:line_no`
  - `break ... if cond`

```
info break    show breakpoints
delete [n]   delete breakpoints [breakpoint n]
disable [n]  disable breakpoints [breakpoint n]
enable [n]   enable breakpoints [breakpoint n]
```
Execution Control

**step s**
Next line, stepping into functions

**next n**
Next line, stepping over functions

**continue c**
Continue to next break

**until loc**
Run until \( loc \); same args as **break**

**finish**
Run until frame returns

**return \([expr]\)**
Pop frame w/out executing \([using expr]\) as return value
Examining Data

- `print p [/f] expr` prints `expr`. `f` is a format character.
- `display [/f] expr` prints `expr` each time execution pauses.
- `info display` lists displayed expressions.
- `undisplay n` removes `n` from display list.
The Call Stack

- `backtrace` or `bt` Print trace of all frames in stack
- `frame [n]` Select current frame [frame # n]
- `info frame` Information on selected frame
- `info args` Arguments of selected frame
- `info locals` Local variables of selected frame
Some Trickier (but Useful) Commands

set var VAR=expr  Actually modify variables in the program being debugged
  ■ Assignment operator from the language (e.g., :=)
  ■ Keyword var is optional
  ■ Useful when symbol name clashes with a GDB command

jump line  Resume execution at line
jump *address  Resume execution at address
Corefiles
Examining Corefiles

- A corefile is a snapshot of a process (image) in memory, when it died

- To allow corefiles on Linux (Bash)
  
  ```
  $ ulimit -c unlimited
  ```

- Upon a crash, find the corefile, `core`

- Load the executable, along with the corefile, into the debugger
  
  ```
  $ gdb prog -c core
  ```

- Examine the program:
  
  ```
  (gdb) bt
  ```

- Note, `prog` needn’t have been compiled with debug information
Summary
More Power

- Only common commands (and uses) are shown here
- There is more functionality available
  - You can catch events and signals
  - Debuggers handle multi-threaded programs
  - Look at machine instructions
- Get comfortable with basic commands
  - This much will prove quite useful
- As you need more, explore