Debugging with GDB

Sakeeb Sabakka
What is GDB

- The **Gnu DeBugger**
- As gdb(1) man page says: gdb “allow you to see what is going on inside another program while it executes—or what another program was doing at the moment it crashed.”
Getting Started…

- Compile the source code for debugging with -g option.

override CFLAGS += -g
Getting Started...

- `gdb` to start GDB.
- `quit` or `Ctrl-d` to exit.

- The most usual way to start GDB is with one argument, specifying an executable program:
  ```
  # gdb a.out
  ```

- You can also start with both an executable program and a core file specified:
  ```
  # gdb a.out core
  ```

- You can, specify a process ID if you want to debug a running process:
  ```
  # gdb -p 4321
  ```
Getting Started...

- Command Completion
  (gdb) info bre < TAB >
  (gdb) info breakpoints

(gdb) b make_ TAB
  make_a_section_from_file make_envirn make_abs_section
  make_function_type make_blockvector make_pointer_type
  make_cleanup make_reference_type make_command
  make_symbol_completion_list

(gdb) b make_
Getting Started…

- (gdb) b 'bubble( M-?
  bubble(double,double)
  bubble(int,int)

(gdb) b 'bubble(

Type in quotes and press TAB or type M-?
Useful in function overloading scenarios

(gdb) shell [shell command]
  execute the specified command in shell
Getting Started...

- (gdb) help
- (gdb) help command
- (gdb) aprorpos < regex >
  - Search for commands matching the given regex.

- (gdb) info
- (gdb) help info
- (gdb) info set
- (gdb) help show
Getting Started...

- (gdb) run [ args ]
  - (gdb) file /home/sakeeb/bin/gdb_ex1
  - (gdb) run 87 178

- (gbd) set args [ args ]
  - (gdb) file /home/sakeeb/bin/gdb_ex2
  - (gdb) set args 87 178
  - (gdb) run

- (gdb) kill
  stop the execution of the binary being debugged.
Breakpoints...

- A **breakpoint** makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops.
- Breakpoints are set using **break** command
  
  (gdb) break location

  *The breakpoint location can be any of the following*

  *linenum*

  *filename:linenum*

  *function*

  *filename:function*
Breakpoints...

- Conditional Breakpoints
  - \texttt{(gdb) break if (expression)}
  - \texttt{(gdb) condition bN (expression)}
  - \texttt{(gdb) until [location]} — goto location or skip the loop
- Setting break points for a single time
  - \texttt{(gdb) tbreak args}
- Using regular expression to set breakpoints
  - \texttt{(gdb) rbreak regex}
    - e.g \texttt{(gdb) rbreak link*}
      - \texttt{(gdb) rbreak}
- Specify the commands to be executed at a breakpoint
  - \texttt{(gdb) commands [ bN ]}
Breakpoints...

(gbd) break traverse_tree
(gdb) commands
> if(node == 0x0)
    > print head
    > else
    > print *head
    > end
> continue
> end

(gdb) condition 1 (node == 0x0)

(gdb) break random_tree if (branch_count <= 2)
Breakpoints...

✓ continue [COUNT]
   *Continue execution, ignore the current break point for COUNT-1 times.*

✓ ignore <BP# [COUNT]>
   *Set ignore-count of breakpoint*

✓ save-breakpoints
   *Save current breakpoint definitions as a script*

✓ info breakpoints [N]
   *Display information about current breakpoint[s].*
Stepping...

- Single stepping through the function is done through step or stepi command. Step steps into the functions calls.
- Stepping over the function call is done using next or nexti command.
  - stepei and nexti do stepping at assembly instruction where step and next do stepping at source code level.
- The continue statement continues the execution of the program till it reaches next break point or program termination.
Function Manipulation...

- Returning from a function without executing the remaining code
  `(gdb) return`

- Get the control after executing the function
  `(gdb) finish`

- The value of the variable can be changed in the frame where it is defined.
  Set the value of i to 20 in the current frame/function
  `(gdb) print i=20`

- Invoking a function from gdb.
  `(gdb) call <function name>`
Watchpoints

- Watchpoint stop the program execution whenever the value of expression changes
  
  (gdb) `watchpoint temp->next`
  
  (gdb) `watchpoint *(int*)0x56ad52`
  
  (gdb) `watchpoint 'a+b / c+d'`

- Information on currently set watchpoints
  
  (gdb) `info watchpoints`
Core files...

- Start gdb with executable and core file
  
  `$ gdb a.out core ( or gdb a.out -c core)
  
or specify corefile from gdb prompt

  `(gdb) core core`
Core files...

- To set the core file size
  # ulimit -c
  # ulimit -c <size>
Stack Frames...

- When program stops GDB helps you in finding all the information stored in the *stack frame* (frame pointer register)
- Frame contains the **arguments given to the function**, **functions local variables** and the **address at which the function is executing**
- GDB by default is in the **currently executing frame (frame 0)** and displays information pertaining to that frame.
- Back trace – summary of how your program got to place where it is.
Function Arguments & Environment Variables

**Stack**
Collection of stack frames. Grows Downward.

**Unused Memory**

**Heap**
Dynamic memory for programs and libraries.
Grows upward.

**Un-initialized Data Segment (BSS)**
Executable stores only size required for this area, memory will
allocated and initialized to zero on process instantiation.

**Initialized Data Segment**
Initialized global variables from executable

**Text Segment**
Machine Code.
Stack Frames...

- (gdb) backtrace / bt / where - Prints the entire stack
  (gdb) bt n - prints the innermost n frames
  (gdb) bt –n - Prints the outermost n frames
  (gdb) bt full - Prints the information of local variables

- Selecting a frame
  (gdb) frame <num>
  (gdb) frame addr
  (gdb) up n – towards outermost frame
  (gdb) down n – towards innermost frame
Information about frames

(gdb) info frame / f
Print -

  address of the current frame
  address of previous and next frames
  address of local variables and parameters
  program counter
  registers saved in the frame

(gdb) info args – Prints the argument of the selected frame
(gdb) info locals – Prints the local variables
(gdb) info scope <function> - info about locals in the function
(gdb) info catch - Prints all exception handlers active
Examining the Data...

- Display the value of an expression
  
  \texttt{(gdb) print[/FMT] (expression)}
  
  (gdb) print i
  
  (gdb) print/s string

- \texttt{(gdb) output (expression)}

  same as print, but don't put in value history and don't print newline.

- Examine the value of memory
  
  \texttt{(gdb) x /FMT <address>}
  
  (gdb) x/s string

- Display the value of expression each time program stops
  
  \texttt{(gdb) display/FMT (expression)}
  
  (gdb) display i

- \texttt{(gdb) help set radix}
Examining the Data...

Print format modifiers

- x Print in hexadecimal
- d Print as a signed decimal
- u Print as an unsigned decimal
- o Print as octal
- t Print in binary (t is for two)
- a Print as an address
- c Print as character

*(gdb) x/FMT (expression)*

- (gdb) x/d &i
- (gdb) x/s s
- (gdb) x/c s
- (gdb) x/4c s
- (gdb) x/t s
- (gdb) x/3x s
Examining the Source...

- Display source code
  
  \texttt{(gdb) list \textsc{linenum} | \textsc{file:linenum} | \textsc{function} | \textsc{file:function} | *\textsc{address}}

  \begin{align*}
  \text{(gdb)} & \text{ list 5,} \\
  \text{(gbd)} & \text{ list ,28} \\
  \text{(gbd)} & \text{ list 21, 38} \\
  \text{(gdb)} & \text{ list parseArgs} \\
  \text{(gbd)} & \text{ list main.c:doit} \\
  \end{align*}

  \text{(gdb) set listsize 24}

- Searching the source code
  
  \texttt{(gbd) search regex}
  
  \texttt{(gdb) reverse-search regex}
Examining the Source...

- **(gdb) whatis insert::node**
  Display the type of the expression.

- **(gdb)ptype insert::node**
  Display the description of the type of the expression.

  Note: - insert::node refers to node variable in function insert

- `@` is a binary operator for treating parts of memory as arrays
  eg: `- int *array = (int *) malloc (len * sizeof (int))
  (gdb) print *array@len`

- `{type}addr` is used for display the value at addr as type

- `(type)addr` is used for display the value as type
  (gdb) p/x (short[2])0x12345678
Logging...

- **Logging Output**

- **set logging on**
  - Enable logging.

- **set logging off**
  - Disable logging.

- **set logging file file**
  - Change the name of the current logfile. The default logfile is `gdb.txt`.

- **set logging overwrite [on|off]**
  - By default, GDB will append to the logfile. Set overwrite if you want set logging on to overwrite the logfile instead.

- **set logging redirect [on|off]**
  - By default, GDB output will go to both the terminal and the logfile. Set redirect if you want output to go only to the log file.

- **show logging**
  - Show the current values of the logging settings.
The `.gdbinit` File...

- GDB will look for this file in two places
  - First, In your home directory
  - Second, In the current directory

- comments are started with "#" and blank lines are ignored.

- Put redundant commands into `.gdbinit`
  - Eg:- `cat .gdbinit`
    - `set listszie 24`
    - `set break main`
    - `set sourcedir /home/sakeeb/imported_sources`
    - `set objectdir /home/sakeeb/impored_objects`
    - `set logging file /tmp/sakeeb.gdb.log`
    - `set loggin on`

- Automate most of debugging tasks
  - Use `.gdbinit` to automate debugging
  - `gdb` commands stored in alternate files can be executed using `-x` option.
  - Make use of `-batch` option to run `gdb` in batch mode
    - eg: `- gdb -batch -x /home/sakeeb/batch_debug_commands`
Threads...

- List of threads
  
  \texttt{(gdb) info thread}  
  \texttt{(gdb) thread <thread-id>}  
  \texttt{(gdb) thread apply <thread-id | all > command}  
  
  Apply the command on the specified thread.

To view the backtrace of all the threads use the following
\texttt{(gdb) thread apply all backtrace}
The child process...

(gdb) set follow-fork-mode [parent | child | ask | serial]
The unfollowed process will continue to run.
By default, the debugger will follow the parent process.

(gdb) set follow-fork-mode child
(gdb) break execve
(gdb) continue

When child spawns:
(gdb) step
!!! we are inside the spawned process..
Catchpoints....

- Catchpoints causes gdb to stop for certain kinds of events.

  \textit{(gdb) catch event}

  “event” can be

  \texttt{exec} call to exec

  \texttt{fork} | \texttt{vfork} call to fork or vfork
Signals...

- GDB has ability to detect any occurrence of signal in execution
  
  (gdb) info signals
  (gdb) info handle
  (gdb) info signal [Signal num]
  Print table of how gdb handles signals

- Change the GDB behavior for signal handling
  
  (gdb) handle signal  [keywords]
  signal can be signal name or Num.
  Keywords can be
  nostop – Not to stop Program execution
  stop- Stop Program execution
  print / noprint – should / should not print the signal info
  pass / noignore – allow program to see the signal
  nopass / ignore – should not allow program to see the signal
(gdb) info shared
  details about shared library mapping into memory.

(gdb) info registers
  details about registry contents.

(gdb) disassemble [address | function]
  disassemble the program to assembly instructions.
Low level debugging...

(gdb) disass main
Dump of assembler code for function main:
0x080484a4 <main+0>:  lea 0x4(%esp),%ecx
0x080484a8 <main+4>:  and $0xfffffffff0,%esp
0x080484ab <main+7>:  pushl -0x4(%ecx)
0x080484ae <main+10>: push %ebp
0x080484af <main+11>: mov %esp,%ebp
0x080484b1 <main+13>: push %ecx
0x080484b2 <main+14>: sub $0x24,%esp
0x080484b5 <main+17>: movl $0x5,(%esp)
Other tools

- valgrind(1) command
  
  ```bash
  # valgrind ./buggy_pointer
  # valgrind --log-file=filename ./buggy_pointer
  # valgrind -v -leak-check=full --./buggy_pointer
  ```

- /usr/lib/libmudflap library
  
  - Compile with `-fmudflap`
  - Link with `-lmudflap`

- /lib/libSegFault.so library
  
  ```bash
  # export LD_PRELOAD=/lib/libSegFault.so
  # ./buggy_pointer
  ```

- backtrace* functions for self-debugging programs
  
  - Compile with “-g -rdynamic” option
  - backtrace(3)
  - backtrace_symbols(3)
  - backtrace_symbols_fd(3)