GDB Cheatsheet (Basic)

Author: Guanzhou Jose Hu 胡冠洲 @ ShanghaiTech

GNU's official doc: https://sourceware.org/gdb/current/onlinedocs/gdb/.

Remember to compile with debugging info by setting flag `-g`.

Invoke / Exit GDB

`$ gdb your_executable` ; `$ gdb -p pid` to invoke GDB

- often used trick: `$ gdb -p $(pgrep exec_name)` to debug a running executable
- flag `-f` forces GDB to catch every child process that the executable forks

`target remote host:port`

- link a remote target who is listening for GDB connection on `host:port`
- commonly used pattern is `localhost:1234` for debugging a kernel with QEMU

`r [args]`

- start running the program
- command-line args to the program should be provided here, *not* at GDB invocation

`q`

- quit GDB, need confirmation if program is still running

Breakpoints

`b function` ; `b file:function`

- break at function entrance, right after stack setup
- file name can be a path

`b file:line`

- break at line in file, right before this line is executed

`b ... if cond`

- conditionally break if `cond != 0` when reaching the breakpoint
- `cond` can be any valid expression

`b *addr`

- break at code instruction at address `addr`

`rbreak regex` ; `rbreak file:regex`

- match function name with regex expression
- regex specification follows `$ man grep`

`info b`

- view the info of all breakpoints

`clear ...`

- clear a breakpoint by `...` (just like how you set them)

`delete`

- delete all breakpoints (and watchpoints)

Watchpoints

`watch var` ; `watch file::var` ; `watch function::var`

- break whenever `var` is written *and* the value changes
- watching requires special hardware support, otherwise GDB inserts software watchpoints that make the execution very slow

`watch *addr`

- break whenever address `addr` is written into *and* is modified

`rwatch ...`

- break whenever the thing is read

`awatch ...`

- break whenever the thing is read or written

`info watch`

- view the info of all watchpoints

Continue Execution

`c`

- continue running, until finish or reaching the next breakpoint

`s` ; `s count`

- step forward 1 / `count` source lines
- will go *inside* function calls

`n` ; `n count`

- step forward 1 / `count` source lines
- will *proceed* until function calls in the line return

`u location`

- continue running until either the specified location or the current stack frame returns
- `location` can be in any acceptable form that `b` accepts

`si`

- step forward one machine code instruction

`ni`

- step forward one machine code instruction
- if that is a function call, will *proceed* until the function returns

Code Listing

`l` ; `l line` ; `l file:line` ; `l function`

- show 10 source lines around certain line
- default is the current position of execution

`l -` ; `l +`

- show 10 source lines before/after the previous listing

`info line` ; `info function`

- show the starting & ending address of the compiled code of source line/function

`disass function` ; `disass start, end`

- show disassembled machine instructions of function `function` or instructions from memory address `start` → `end`
- if current execution is pausing at somewhere within the instructions shown, that instruction will be prefixed with a `=>`

`disass /m ...`

- show disassembled stuff along with the source code

Examine / Modify Data

`p expr`

- execute the expression `expr` and show its value
- `expr` can be any valid expression:
  - variable name
  - `arr[idx]`
  - `&arr[0]`: start address of a buffer
  - `*arr@len`: first `len` elements in `arr`
  - `arr[idx]@len`: the `idx`th to `idx + len - 1`th elements in `arr`
- `expr` can modify the program state (e.g., an assignment), so it can be used for modifying variables

`x/<n><f><u> addr`

- examine memory content at `addr`
- configurations:
  - `<n>` is the repeat count (how much after `addr` you want to examine)
  - `<f>` is the display format just like those in `printf` (e.g., `x`, `d`, `c`, ...)
  - `<u>` is the unit size, which is any of:
    - `b` - byte
    - `h` - halfword (2 bytes)
    - `w` - word (4 bytes)
    - `g` - giantword (8 bytes)

`display expr`

- automatically show the value of `expr` every time we break

Examine Frame

`f`

- show a brief info of the current stack frame

`info f`

- show a more verbose info of the current stack frame

`info locals`

- print *current* value of all local variables in current stack frame

`info args`

- print *current* value of all arguments of current function

`p local_var@entry`

- print `local_var`'s value at the time *when we entered* the current function

Backtracing

`bt`

- show a calling stack back-tracing
- order is later $\rightarrow$ earlier from top $\rightarrow$ bottom

`bt full`

- show a calling stack back-tracing with all their local variables' values

`f frame_id`

- force jumping to the specified stack frame
- `frame_id` can be acquired from `bt`

`up num` ; `do num`

- go up/down `num` stack frames

Multiple Threads

`info threads`

- list ids, info, and current status of all current threads

`thread thread_id`

- switch to the specified thread
- `thread_id` is the first field shown by `info threads`

`thread apply <id_list|all> command`

- apply command `command` to specified threads
- `<id_list>` is a space-separated sequence, where each element can be:
  - a single thread id number
  - a range, e.g., `3-7`
- Or use `all` to apply to all threads

Jumping

`j line` ; `j function`

- jump to and immediately start execution from the specified location
- `line` and `function` can be given just like how you list source code using `l`

`j *addr`

- jump to and immediately start execution from instruction at address `addr`

Examine Registers

`info registers [reg]`

- show value of register `reg`
- default is to show values of all registers

Miscellaneous

Run a shell command `cmd` by: `!cmd` or `shell cmd`.

Auto completion of symbol names/commands by *double-tapping* TAB.

Abbreviations:

- `r` = `run`
- `q` = `quit`
- `c` = `continue`
- `s` = `step`
- `n` = `next`
- `u` = `until`
- `si` = `stepi`
- `ni` = `nexti`
- `l` = `list`
- `p` = `print`
- `f` = `frame`
- `bt` = `backtrace`
- `do` = `down`
- `j` = `jump`