A GDB Tutorial

Stojche Nakov

CS Departement
Princeton University

DECADES subgroup meeting, November 11, 2021

GDB: The GNU Project Debugger

GDB overview

- First release in 1986.
- Supports various languages: **C, C++, FORTRAN, Objective-C** etc.
- Multiple GUI extensions (DDD, kDgb, Nemiver ...).

GDB: The GNU Project Debugger

GDB overview

- First release in 1986.
- Supports various languages: **C, C++, FORTRAN, Objective-C** etc.
- Multiple GUI extensions (DDD, kDgb, Nemiver ...).

GDB's Purpose

- Allows you to see what is going on 'inside' another program while it executes.
- Makes the program stop on specific conditions.
- Examine what has happened.
- Change things in your program.

GDB Basics

Compiling

Must be compiled using the flag "**-g**". Also it is recommended that the optimization flags are removed and the "**-Og**" flag is added.
To run the program use "**gdb –args ./exe arg1 arg2**"

Control + x a for better view.

GDB Basics

Compiling

Must be compiled using the flag "**-g**". Also it is recommended that the optimization flags are removed and the "**-Og**" flag is added.
To run the program use "**gdb –args ./exe arg1 arg2**"

Control + x a for better view.

Basic commands

- **run / r** – Begins running the program. If the program is already active, it restarts it.
- **continue / c** – Continues the execution of the program.
- **break / b** – Sets a breakpoint.
- **print / p item** – Prints an item (value of variable, function, structure, class etc).
- **backtrace / bt** – Shows the calling sequence.
- **list / l** – Prints the code.
- **frame / f #** – Changes to the given frame.

Parallel Debugging

Multi-Threading

- **info threads / t** – Prints information for each thread.
- **thread / t #** – Switches to the thread.
- **thread apply all "cmd" / t a a "cmd"** – Applies "cmd" to all threads (usually **backtrace** is used).

Parallel Debugging

Multi-Threading

- **info threads / t** – Prints information for each thread.
- **thread / t #** – Switches to the thread.
- **thread apply all "cmd" / t a a "cmd"** – Applies "cmd" to all threads (usually **backtrace** is used).

Distributed memory

MPI is **SIMD** parallel model.

Parallel Debugging

Multi-Threading

- **info threads / t** – Prints information for each thread.
- **thread / t #** – Switches to the thread.
- **thread apply all "cmd" / t a a "cmd"** – Applies "cmd" to all threads (usually **backtrace** is used).

Distributed memory

MPI is **MIMD** parallel model.

Parallel Debugging

Multi-Threading

- **info threads / t** – Prints information for each thread.
- **thread / t #** – Switches to the thread.
- **thread apply all "cmd" / t a a "cmd"** – Applies "cmd" to all threads (usually **backtrace** is used).

Distributed memory

MPI is **MIMD** parallel model.

```
mpirun -np 2 ./exe1 : -np 2 ./exe2 : -np 2 ./exe3
```

Parallel Debugging

Multi-Threading

- **info threads / t** – Prints information for each thread.
- **thread / t #** – Switches to the thread.
- **thread apply all "cmd" / t a a "cmd"** – Applies "cmd" to all threads (usually **backtrace** is used).

Distributed memory

MPI is **MIMD** parallel model.

mpirun -np 1 **gdb ./exe1** : -np 1 ./exe1 : -np 2 ./exe2 : -np 2 ./exe3

Parallel Debugging

Multi-Threading

- **info threads / t** – Prints information for each thread.
- **thread / t #** – Switches to the thread.
- **thread apply all "cmd" / t a a "cmd"** – Applies "cmd" to all threads (usually **backtrace** is used).

Distributed memory

MPI is **MIMD** parallel model.

Alternatively, open one xterm per process:
mpirun -np 4 **xterm -e gdb ./exe**

Launching gdb

‘

- Execute **gdb**, and then specify the exec file using the command **file**.
- Arguments can be supplied to the **run** command.
- By setting **ulimit -c unlimited**, and then use **gdb exec_file core_file**.
- Attaching to a running process: **gdb attach $pid**.
- **vgdb**: Valgrind + GDB. In the first terminal, launch valgrind and add these arguments: **–vgdb=yes –vgdb-error=0**. Then open an other terminal, start gdb and copy paste the proposed commands in the first terminal.

Improve gdb's output

- Hit **Control + x 2**, hit it again! And AGAIN!!
- **printf** is available.
- User define helpers in the **.gdbinit** file. First add one of the next two lines to your global gdbinit (~/.initgdb):

```
# disable safe checks
1) set auto-load safe-path /

# a bit safer
2) add-auto-load-safe-path
/path/to/your/working/gdbinit
```

A local .gdbinit file example

```
# break on main automatically
break main

#define a function
define my_print
   set language c # tell gdb what language to use
   printf "Here comes the first arg: <%d> \n" , $arg0
   set $n = 0
   while $n != $arg1
      printf "Second argument is not %d \n", $n
      set $n = $n + 1
   end
   printf "Second argument is  %d !!!!\n", $arg1
end
```

GDB advanced breaks

- Conditional breakpoints: **break if condition — break if not condition**
- **watch, rwatch, awatch** watch the memory!
- **info breakpoint** prints the breakpoints (including wathe).
- **command # instruction 1 instruction2 end**, will execute all instruction each time that breakpoint is accessed.

```
command 2
# suppresses the normal output of the breakpo
silent
print var
next
print var2
end
```

Other GDB's advance toys

- **call** allows to call functions/methods defined in the code.
- **set var variable=value** sets the value of **variable**.
- **Convenience Functions**, (help function), like **$_caller_is**, for example.
- **Reverse debugging** Yes it exists. Here is how to use it:
  - Break on main, and then type **record**. All following instructions are recorded.
  - When you hit a breakpoint or the program crashes, use **reverse-continue, reverse-next** etc.
  - Hint: **set can-use-hw-watchpoints 0**, if planning to use watchpoints in reverse debugging.

Thank you for your attention