Advanced Debugging with gdb

David Khosid

Sept 21, 2009

david.kh@gmail.com

Agenda

- Techniques for debugging big, modern software:
  - STL containers and algorithms, Boost
    Ex: how to see containers
  - Signals
  - Multi-threaded (ex.: how to follow a thread?)
  - Repetitive tasks on the almost unchanging code base
- Examples

Sources of information

GDB was first written by Richard Stallman in 1986 as part of his GNU system

- Richard Stallman, “Debugging with gdb”

  www.gnu.org/software/gdb/documentation

- Help: $gdb –h
  (gdb) h
  (gdb) apropos

  Command names may be truncated if the abbreviation is unambiguous. TAB completion.

- Command Cheat Sheet
  www.yolinux.com/TUTORIALS/GDB-Commands.html

- Last GDB version is 6.8, new 7.0 soon: 2009-09-23

Item #1: C++ and STL - Containers

How to see container's content?

1. **Commands file, ex. .gdbinit**
   http://www.yolinux.com/TUTORIALS/src/dbinit_stl_views-1.03.txt
   Limitations: a little
2. **libstdc++ compiled in debug mode**
   Limitations:
   - different product , not for QA, not for client, not in performance tuning stage
   - performance

Item #1: C++ and STL - Containers

How to see container's content?

3. Auxiliary functions

```
typedef map<string, float> MapStringFloat;
void mapPrint(const MapStringFloat& m){
    for(MapStringFloat::const_iterator pos = m.begin(); pos != m.end(); ++pos){
            cout << pos->first << " : " << pos->second << "\n";
}
```

Limitations:

- you can't do that without a process to debug (investigating core files)
- optimization of unused functions. Solution: 'volatile'

4. Pretty-printing of STL containers in future versions of GDB

Item #2: Extending GDB - User-defined commands

- **(gdb) show user** *commandname*

- **Example:**

```
(gdb)define adder
     print $arg0 + $arg1 + $arg2
end
(gdb) adder 1 2 3
```

Item #3: Automating repetitive tasks

- **What GDB Does During Startup**

  1. Executes all commands from system init file

  2. Executes all the commands from ~/.gdbinit

  3. Process command line options and operands

  4. Executes all the commands from ./.gdbinit

  5. reads command files specified by the `-x' option

  6. …

Automating tasks - history, recording

- continue **What GDB Does During Startup**

  … 6. Reads the command history recorded in the *history file*.

- (gdb) **set history filename** *fname*
  (gdb) **set history save** on/off
- (gdb) **show history**
- (gdb) **show commands**

Item #4: Signals

- '**i handle**' or '**i signals**'

  Print a table of all the signals and how gdb has been told to handle each one.

- **handle signal** [keywords...]

  keywords: nostop|stop, print|noprint and pass|nopass

  Ex: handle SIG35 nostop print pass

  handle SIG36 stop (implies the 'print' as well)

  handle SIG37 nostop print nopass

  handle SIG38 nostop noprint nopass

Item #5: Multi-threads

- Use case: debugging specific thread, while controlling behavior of others.
- facilities for debugging multi-thread programs:
  - automatic notification of new threads
  - ‘thread threadno’, to switch among threads
  - ‘info threads’, to inquire about existing threads
  - thread-specific breakpoints
  - set mode for locking scheduler during execution

    (gdb) set scheduler-locking step/on/off

  others: Interrupted System Calls
- Example:

  (gdb) **i threads**

  (gbd) **b foo.cpp:13 thread 28 if x > lim**

Item #5: Remote debugging

- **Use case**:
  - GDB runs on one machine (host) and the program being debugged (exe.verXYZ.stripped) runs on another (target).
  - GDB communicates via Serial or TCP/IP.
  - Host and target: exactly match between the executables and libraries, with one exception: stripped on the target.
  - Complication: compiling on one machine (CC view), keeping code in different place (ex. /your/path/verXYZ)
- Solution:
  - Connect gdb to source in the given place:

  (gdb) set substitute-path /usr/src /mnt/cross

  (gdb) dir /your/path/verXYZ

Remote debugging - example

- Using gdbserver through TCP connection:
  remote (10.10.0.225)> gdbserver :9999 *program_stripped*
  or remote> ./gdbserver :9999 –attach <pid>

- host> gdb *program*
  host>(gdb) handle SIGTRAP nostop noprint pass
  to avoid pausing when launching the threads
  host> (gdb) target remote 10.10.0.225:9999

Item #6: Back to the past

- **Convenience variables** are used to store values that you may want to refer later. Any string preceded by $ is regarded as a convenience variable.

  Ex.: **set `$table` = `*table_ptr`**

  (gdb) **show conv**
- **Checkpoint -** a snapshot of a program's state
  (gdb) **checkpoint**
  (gdb) **i checkpoint**
  (gdb) **restart** *`checkpoint-id`*
- **Value history-** values printed by the `print` command.

Small Items: #7, #8

#7. **How to see macros?**

$ g++ **-gdwarf-2 -g3** a.cpp -o prog

#8. **64 bit .vs. 32bit**

- -m32 flag
- On 64-bit machine, install another 32-bit version of GDB

  $ **ls -l `which gdb32`**

  /usr/bin/gdb32 -> '/your/install/path'

Lightweight how-to's

1. **How to remove a symbol table from a file?**
   A: **strip**
2. **How to supply arguments to your program in GDB?**
   A1: With --args option
   #sudo gdb -silent **--args** */bin/ping google.com*
   A2: As arguments to **run**: (gdb) **run** *arg1 arg2*
   `run` without arguments uses the same arguments used by the previous `run`.
   A3: With set args command:
   (gdb) set args *arg1 arg2*
   (gdb) show args
   `set args` without arguments – removes all arguments.
3. **How to know where you are (file, next execution line)?**
   A: (gdb) f

Lightweight how-to's - continue

4. **How to find out the crash file executable?**
   A1: #file *core.1234*
   A2: #gdb core.1234
   A3: use /proc/sys/kernel/core_pattern
   #echo "core_%e.%p" > /proc/sys/kernel/core_pattern
   if the program **foo** dumps its core,
   the core_foo.1234 will be created.
5. **How to find out why your program stopped?**
   A: (gdb) i prog
6. **Which command(s) can be used to exit from loops?**
   A: (gdb)until *lineNo*
7. **'print', 'info', 'show'- what is a difference?**
   'print' – print value of expression
   'info' – showing things about the **program** being debugged
   'show' – showing things about the **debugger**

Problem Determination Tools for Linux

- -Wall ☺
- Code review
- Program's traces, syslog, profilers
- Static Source Code Analysis:
  - scan.coverity.com – free for FOSS
  - Flexelint
- Dynamic analysis: Valgrind,
- strace, /proc filesystem, lsof, ldd, nm, objdump, wireshark

Summary

1. Start from thinking of **Use Case**, then look in the manual, use ‘apropos’ and ‘help’
2. **Productivity**:
   Stepping through a program is less productive than thinking harder and adding output statements and self-checking code at critical places.
3. **When to use GDB?**
   - core file,
   - when a problem can be reproduced**,** repeating errors
   - self-educating
4. **When not?**
   Other tools, traces

5. Questions?