# Debugging

Mathieu Lacage - DREAM

INRIA
SOPHIA ANTIPOLIS

# Overview

- The Debugging Rules
  - reproduce the bug
  - understand the system
  - narrow the search
  - get a fresh view
- Understanding debugging tools and their limitations:
  - Print statements
  - Log files
  - Debuggers
  - Memory checkers
- Demonstration of gdb
- Demonstration of valgrind

# Debugging

- A bug happens when there is a difference between:
  - how you expect the system to work
  - how the system works for real
- To find the source of a bug, you need to understand how the system works for real.
- This can be a daunting task because you don't know where the bug comes from (otherwise, there would be no bug ;)
- Which is why you need to follow the Debugging Rules

# The Debugging Rules

- The rules are easy to remember
- Using the rules is hard but it is easy to try to use them:
  1. reproduce the bug: understand what the user expects
  2. understand the system
  3. narrow the search
  4. get a fresh view

# *Rule 1*: Reproduce the bug

- Why ?
  - verify that the bug exists before fixing the bug (maybe the user does not understand what the system is expected to do)
  - verify that the bug does not exist anymore after fixing the bug
  - understand the bug parameters, i.e., the conditions which trigger the bug.
- Write down:
  - the steps to reproduce the bug
  - what happens ?
  - what should happen ?
  - how often does it happen ?

# *Rule 2*: Understand the system

- What is the system ?
  - the source code
  - the build tools and the build environment
  - the runtime environment
- Read the manual !!
  - build tools have great documentation
  - standard libraries have great documentation
  - developer documentation

# *Rule 2*: Understand the system -- example

- The code:

```
{
  char *str = g_strconcat (“one”, “two”, “three”);
}
```

- The manual:

```
The variable argument list must end with NULL. If
 you forget the NULL, g_strconcat() will start
 appending random memory junk to your string.
```

# *Rule 3*: Narrow the search

- Apply dichotomy: “Divide and Conquer”
  - The range of possible values/solutions/problems is halved in two after each test
- Start from the failure
  - ask yourself: “what is failing?” and not “what is working?”
- Change one thing at a time

# *Rule 3*: Narrow the search

- The light bulb does not work:
  - make sure this is not a power outage in your house (try to test another electric device)
  - If it does not work, look out the window to see if other houses have the lights on or off.
  - If it works:
    - replace the failing bulb with a bulb which you know works (ideally, make sure it works by lighting it once somewhere else)
    - verify the wiring to the bulb
    - verify the wiring to the light switch
    - verify the wiring between the light switch and the light bulb
    - verify the wiring between the power input to your house and the light switch

# *Rule 4*: Get a fresh view

- Show the bug to someone else
- Do not explain where you think the bug comes from: do not “pollute” the newcomer.

# Software tools

Make sure you understand their limitations

- Print statements
- Log files
- Debuggers
- Memory checkers

# Print statements

- How to use them ?
  - module name (filename, source line)
  - timestamp (date/time)
  - Example in C:

```
#define my_debug(str,...) \
if (g_debug) { \
    printf (“(%s:%d:%s) ” str, __FILE__, __LINE__, __func__ \
      __VA_ARGS__); \
}
```

- Limitations:
  - They can change program behavior in the presence of threads or IO/networking applications.
  - How big a change ? It depends on the amount of IO/networking and the amount of printing and the speed and the RAM of your machine.

# Log Files

- How to use them ?
  - Follow the print statements' rules
  - Archive your log files
  - Leave the log capability in your production builds/systems and disable it by default.
- Limitations:
  - They can change program behavior in the presence of threads or IO/networking applications.
  - How big a change ? It depends on the amount of IO/networking and the amount of printing and the speed and the RAM of your machine.

# Debuggers

- They work by placing breakpoints in the running assembly code.
- They have a deep influence on the system timing characteristics with or without threads.
- They require “debugging information” to be embedded in the executable to debug.
  - the “-g” (g for debug, of course) compiler flag is used to generate this “debugging information”.
  - consider the use of -g3 with gcc or -gdwarf-2
- They can:
  - step over code, functions
  - print backtraces
  - print variable content

# Memory Checkers: valgrind

- It ensures that each bit accessed in memory is:
  - correctly allocated
  - correctly initialized
  - correctly freed
- Limitations:
  - requires large amounts of RAM (roughly, 8 times more memory than your program)
  - is much slower especially if you have less than 1GB of RAM

# References

- “Debugging: the 9 indispensable rules for finding the most elusive software and hardware problems” by David J. Agans
- Tool manuals
- Library manuals
- Developer documentation
- Urls:
  - Valgrind: http://valgrind.org/
  - Gdb: http://www.gnu.org/software/gdb/