Introduction to Debugging

BASED ON LINUX

CHUMAO (FRANK) WU

- Teaching Fellow in Cybersecurity
- Pursing MCS at UoW during 2023~2024
- Software Developer in China (Huawei, Samsung, OPPO, etc)
- Focusing on security aspects of smart devices, including SoC security, mobile OS (Android & Tizen) security features, and DRM (**D**igital **R**ights **M**anagement)

https://www.linkedin.com/in/chumao-wu-8513b7231/

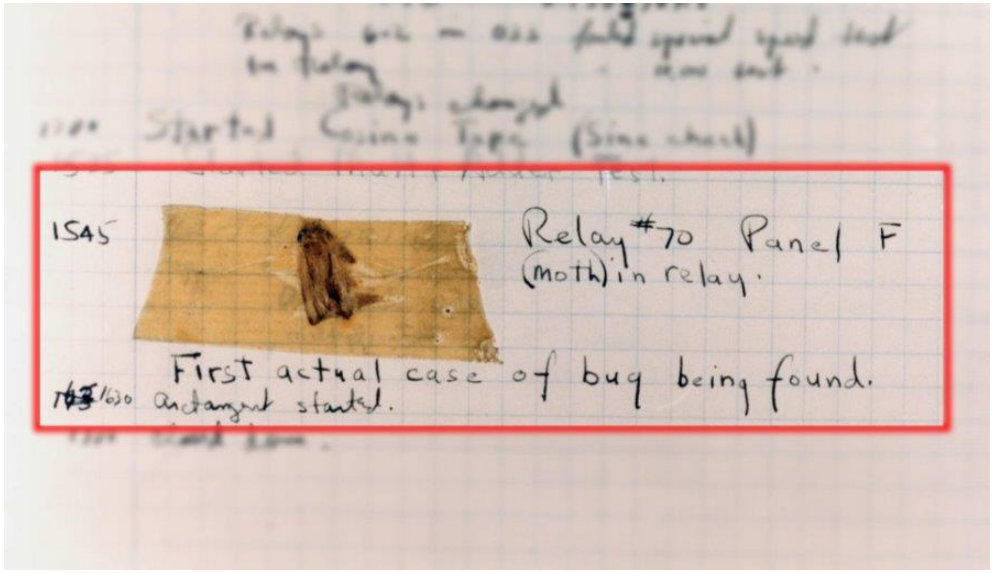
Contents

- Debugging Overview
- Dig Deeper Into A Debugger
- Demos

Debugging Overview

Software Debugging

- "Bug" is an engineering jargon to describe software or hardware problems.
- Finding and solving bugs that exist in your software.
- "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." -- *Brian Kernighan*



In computing, the term "bug" became famous in 1947 when engineers working on the Harvard Mark II relay computer found an actual moth stuck in a relay, causing a malfunction.

https://supercooldesign.co.uk/articles/the-moth-myth-aka-the-worlds-first-computer-bug

Types of Bugs

- Logic errors
  - Wrong specification or implementation
  - Incorrect API usage, integration issues, etc.
- Memory issues
  - Memory leakage
  - Invalid memory access, e.g., NULL pointer deference, use-after-free, out-of-boundaries, etc.
- Resource Leakage: file descriptors, sockets, etc.
- Multithreading/processing issues
  - Race condition
  - Deadlock
  - Livelock
- Performance issues
- Other types of bugs

Debugging Step-by-Step

- Observer the bug
- Create a reproducible input
- Narrow the search space
- Analyse
- Devise and run experiments
- Modify code and squash the bug

https://web.stanford.edu/class/archive/cs/cs107/cs107.1258/resources/debugging

Debugging Techniques

- Log analysis
- Interactive debugging
- Postmortem analysis
- Tracing
- Profiling
- Using checkers

Log analysis

- Often resort to logs for first-hand information
- Log contains valuable information to help understand the problem, particularly when errors occur
- A diff comparison between baseline log and error log could be very effective to locate the clues

Interactive Debugging

- Use a debugger to analyse the software **interactively** at **runtime**
- Most common used in a development environment
- Allow to stop and resume execution, inspect variables, registers and stack frames, etc.
- Example debuggers: GDB (GNU debugger), LLDB (LLVM Debugger), Visual Studio Debugger

Postmortem Analysis

- Generate a snapshot of a process' memory on its "demise", collecting information for "postmortem" analysis (e.g. backtrace, thread information, memory, registers, etc).
- Use debuggers or dedicated tools to analyse the coredump files for root cause.
- Used in automated test environment to reproduce and fix crash issues.
- Enabled in production environments to facilitate troubleshooting, with coredump files transferred to dedicated servers.
- Examples: Linux core files, Windows dump files, Android tombstones, etc.

- Core dump on Linux

```
frank@ubuntu:~/github/debugging_intro$ coredumpctl list -r -q
TIME                            PID  UID  GID SIG     COREFILE EXE                                                  SIZE
Fri 2025-09-19 04:38:14 UTC 17583 1000 1000 SIGSEGV present  /home/frank/github/debugging_intro/demo                  17.8K
Sun 2025-09-07 04:29:02 UTC 20987 1000 1000 SIGABRT missing  /home/frank/github/debugging_intro/postmortem-demo/crash_demo     -
Wed 2025-08-27 03:26:32 UTC 14429 1000 1000 SIGABRT missing  /home/frank/github/debugging_intro/postmortem-demo/crash_demo     -
Wed 2025-08-27 03:15:11 UTC 14235 1000 1000 SIGABRT missing  /home/frank/github/debugging_intro/postmortem-demo/crash_demo     -
Wed 2025-08-27 03:11:05 UTC 14156 1000 1000 SIGABRT missing  /home/frank/github/debugging_intro/postmortem-demo/crash_demo     -
Wed 2025-08-27 02:55:53 UTC 13793 1000 1000 SIGSEGV missing  /home/frank/github/debugging_intro/postmortem-demo/crash_demo     -
frank@ubuntu:~/github/debugging_intro$
```

```
frank@ubuntu:~/github/debugging_intro$ coredumpctl debug -q
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /home/frank/github/debugging_intro/demo...
[New LWP 17583]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Core was generated by `./demo'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000645a74ce91c4 in crash () at ./demo.c:16
16          return *p;
(gdb) bt
#0  0x0000645a74ce91c4 in crash () at ./demo.c:16
#1  0x0000645a74ce9236 in main () at ./demo.c:26
```

Tracing

- Record the execution history of a program, including values of variables and context fields, with little execution overhead
- Understand the workflow and performance accounting
- Enabled by instrumenting code by tracers, no code changes or recompilation for the tracee.
- Instrumentation points could be:
  - **Dynamic**: dynamic library calls, system calls, **uprobe/uretprobe,** kprobe/kretprobe
  - **Static**: USDT, LTTng-UST, kernel tracepoints, ftrace
- Examples tracers on Linux:
  - strace/ltrace: tracing syscalls and library calls of an application
  - bpftrace/SystemTap/Dtrace/LTTng/: system-wide tracer following user space into kernel space

ltrace

ltrace.log

```
 1 17859 05:00:53.315779 SYS_brk(0)                                                    = 0x56c8dd809000
 2 17859 05:00:53.417486 SYS_mmap(0, 8192, 3, 34)                                      = 0x7f11001d4000
 3 17859 05:00:53.458419 SYS_access("/etc/ld.so.preload", 04)                          = -2
 4 17859 05:00:53.467542 SYS_openat(0xffffff9c, 0x7f110020a38f, 0x80000, 0)            = 3
 5 17859 05:00:53.472520 SYS_fstat(3, 0x7ffee0f2fa80)                                  = 0
 6 17859 05:00:53.475062 SYS_mmap(0, 0x11bdb, 1, 2)                                    = 0x7f11001c2000
 7 17859 05:00:53.482416 SYS_close(3)                                                  = 0
 8 17859 05:00:53.489566 SYS_openat(0xffffff9c, 0x7f11001d4140, 0x80000, 0)            = 3
 9 17859 05:00:53.497686 SYS_read(3, "\177ELF\002\001\001\003", 832)                   = 832
10 17859 05:00:53.501394 SYS_pread(3, 0x7ffee0f2f8d0, 784, 64)                         = 784
11 17859 05:00:53.504442 SYS_fstat(3, 0x7ffee0f2fb50)                                  = 0
12 17859 05:00:53.507703 SYS_pread(3, 0x7ffee0f2f7a0, 784, 64)                         = 784
13 17859 05:00:53.509957 SYS_mmap(0, 0x211d90, 1, 2050)                               = 0x7f10ffe00000
14 17859 05:00:53.513430 SYS_mmap(0x7f10ffe28000, 0x188000, 5, 2066)                   = 0x7f10ffe28000
15 17859 05:00:53.517491 SYS_mmap(0x7f10fffb0000, 0x4f000, 1, 2066)                    = 0x7f10fffb0000
16 17859 05:00:53.523075 SYS_mmap(0x7f10fffff000, 0x6000, 3, 2066)                     = 0x7f10fffff000
17 17859 05:00:53.528987 SYS_mmap(0x7f1100005000, 0xcd90, 3, 50)                       = 0x7f1100005000
18 17859 05:00:53.533324 SYS_close(3)                                                  = 0
19 17859 05:00:53.537660 SYS_mmap(0, 0x3000, 3, 34)                                    = 0x7f11001bf000
20 17859 05:00:53.542290 SYS_arch_prctl(4098, 0x7f11001bf740, 0xffff80eeffe3ff30, 34)  = 0
21 17859 05:00:53.562202 SYS_set_tid_address(0x7f11001bfa10, 0x7f11001bf740, 0x7f11002150c8, 34) = 0x45c3
22 17859 05:00:53.585213 SYS_set_robust_list(0x7f11001bfa20, 24, 0x7f11002150c8, 34)   = 0
23 17859 05:00:53.590434 SYS_334(0x7f11001c0060, 32, 0, 0x53053053)                    = 0
24 17859 05:00:53.593758 SYS_mprotect(0x7f10fffff000, 16384, 1)                        = 0
25 17859 05:00:53.604336 SYS_mprotect(0x56c8d9b3a000, 4096, 1)                         = 0
26 17859 05:00:53.608049 SYS_mprotect(0x7f1100212000, 8192, 1)                         = 0
27 17859 05:00:53.611660 SYS_prlimit64(0, 3, 0, 0x7ffee0f306a0)                        = 0
28 17859 05:00:53.616074 SYS_munmap(0x7f11001c2000, 72667)                             = 0
29 17859 05:00:53.632442 puts("Hello World!" <unfinished ...>
30 17859 05:00:53.637551 SYS_fstat(1, 0x7ffee0f307a0)                                  = 0
31 17859 05:00:53.639543 SYS_318(0x7f110000a178, 8, 1, 0x7ffee0f306a0)                 = 8
32 17859 05:00:53.642763 SYS_brk(0)                                                    = 0x56c8dd809000
33 17859 05:00:53.651397 SYS_brk(0x56c8dd82a000)                                       = 0x56c8dd82a000
34 17859 05:00:53.655140 SYS_write(1, "Hello World!\n", 13)                           = 13
35 17859 05:00:53.659625 <... puts resumed> )                                         = 13
36 17859 05:00:53.661264 printf("Factorial of %d is %d\n", 3, 6 <unfinished ...>
37 17859 05:00:53.666535 SYS_write(1, "Factorial of 3 is 6\n", 20)                     = 20
38 17859 05:00:53.681382 <... printf resumed> )                                       = 20
39 17859 05:00:53.683585 --- SIGSEGV (Segmentation fault) ---
40 17859 05:00:54.615562 +++ killed by SIGSEGV +++
```

sudo ltrace -S -f -tt -s 200 -o ltrace.log -- ./demo

bpftrace -e 'uprobe:./demo:* { printf("FUNC: %s\n", func); }

tracepoint:syscalls:sys_enter_* /pid == cpid/ { printf("SYSCALL: %s\n", probe); }' -c./demo

bpftrace

```
Attaching 363 probes...
SYSCALL: tracepoint:syscalls:sys_enter_close
SYSCALL: tracepoint:syscalls:sys_enter_execve
SYSCALL: tracepoint:syscalls:sys_enter_brk
SYSCALL: tracepoint:syscalls:sys_enter_mmap
SYSCALL: tracepoint:syscalls:sys_enter_access
SYSCALL: tracepoint:syscalls:sys_enter_openat
SYSCALL: tracepoint:syscalls:sys_enter_newfstat
SYSCALL: tracepoint:syscalls:sys_enter_mmap
SYSCALL: tracepoint:syscalls:sys_enter_close
SYSCALL: tracepoint:syscalls:sys_enter_openat
SYSCALL: tracepoint:syscalls:sys_enter_read
SYSCALL: tracepoint:syscalls:sys_enter_pread64
SYSCALL: tracepoint:syscalls:sys_enter_newfstat
SYSCALL: tracepoint:syscalls:sys_enter_pread64
SYSCALL: tracepoint:syscalls:sys_enter_mmap
SYSCALL: tracepoint:syscalls:sys_enter_mmap
SYSCALL: tracepoint:syscalls:sys_enter_mmap
SYSCALL: tracepoint:syscalls:sys_enter_mmap
SYSCALL: tracepoint:syscalls:sys_enter_mmap
SYSCALL: tracepoint:syscalls:sys_enter_close
SYSCALL: tracepoint:syscalls:sys_enter_mmap
SYSCALL: tracepoint:syscalls:sys_enter_arch_prctl
SYSCALL: tracepoint:syscalls:sys_enter_set_tid_address
SYSCALL: tracepoint:syscalls:sys_enter_set_robust_list
SYSCALL: tracepoint:syscalls:sys_enter_rseq
SYSCALL: tracepoint:syscalls:sys_enter_mprotect
SYSCALL: tracepoint:syscalls:sys_enter_mprotect
SYSCALL: tracepoint:syscalls:sys_enter_mprotect
SYSCALL: tracepoint:syscalls:sys_enter_prlimit64
SYSCALL: tracepoint:syscalls:sys_enter_munmap
FUNC: _start
FUNC: 0x59b7ebafc000
FUNC: 0x59b7ebafc160
FUNC: 0x59b7ebafc0e0
FUNC: main
SYSCALL: tracepoint:syscalls:sys_enter_newfstat
SYSCALL: tracepoint:syscalls:sys_enter_getrandom
SYSCALL: tracepoint:syscalls:sys_enter_brk
SYSCALL: tracepoint:syscalls:sys_enter_brk
SYSCALL: tracepoint:syscalls:sys_enter_write
FUNC: add
FUNC: factorial
FUNC: factorial
FUNC: factorial
SYSCALL: tracepoint:syscalls:sys_enter_write
FUNC: crash
```

```
root@ubuntu:/home/frank/github/debugging_intro# bpftrace -lv 'uprobe:./demo:*'
uprobe:./demo:__do_global_dtors_aux
uprobe:./demo:_fini
uprobe:./demo:_init
uprobe:./demo:_start
uprobe:./demo:add
    int x
    int y
uprobe:./demo:crash
uprobe:./demo:deregister_tm_clones
uprobe:./demo:factorial
    int n
uprobe:./demo:frame_dummy
uprobe:./demo:main
uprobe:./demo:register_tm_clones
```

```
root@ubuntu:/home/frank/github/debugging_intro# sudo bpftrace -e '
uprobe:./demo:add    { printf("add(x=%d, y=%d)\n", args.x, args.y); }
uretprobe:./demo:add { printf("add retval %lld\n", retval); }
uprobe:./demo:factorial    { printf("factorial(n=%d)\n", args.n); }
uretprobe:./demo:factorial { printf("factorial retval %lld\n", retval); }
' -c ./demo -o btsites.log
Hello World!
Factorial of 3 is 6



root@ubuntu:/home/frank/github/debugging_intro# cat btsites.log
Attaching 4 probes...
add(x=1, y=2)
add retval 3
factorial(n=3)
factorial(n=2)
factorial(n=1)
factorial retval 1
factorial retval 2
factorial retval 6
```

Profiling

- Output a profile of the program to identify the performance bottleneck, i.e. statistically summary of observed events, via counting or sampling.
- Events sources:
  - Instrumentation points (same as tracers): uprobe, kprobe, tracepoints, USDT.
  - Software counters: CPU migrations, page faults, context switches, etc.
  - Hardware counters: CPU performance monitoring counters, cpu-cycles, instructions, branches, cache-misses, etc.
- Example profilers on Linux: perf

- perf

```
root@rp5:/home/frank/debugging_intro# perf stat -d
Hello World!
Factorial of 3 is 6
Run hotspot 100000 times.
./demo: Segmentation fault

 Performance counter stats for './demo':

              3.19 msec task-clock
                 3      context-switches
                 0      cpu-migrations
                36      page-faults
         7,602,244      cycles
        15,027,342      instructions
         3,302,768      branches
             7,263      branch-misses
         6,392,757      L1-dcache-loads
             6,167      L1-dcache-load-misses
            19,176      LLC-loads
     <not counted>      LLC-load-misses

       0.004670055 seconds time elapsed

       0.003692000 seconds user
       0.000000000 seconds sys
```

```
Samples: 32  of event 'cycles:Pu', Event count (approx.): 6720247
  Children      Self  Command  Shared Object          Symbol
+   98.74%     0.00%  demo     demo                   [.] _start
+   98.74%     0.00%  demo     libc.so.6              [.] __libc_start_main_impl (inlined)
+   98.74%     0.00%  demo     libc.so.6              [.] __libc_start_call_main
+   98.74%     0.00%  demo     demo                   [.] main
-   98.74%     4.75%  demo     demo                   [.] hotspot
   - 93.99% hotspot
      - factorial
         - factorial
            + factorial
   + 4.75% _start
+   93.99%    93.99%  demo     demo                   [.] factorial
+    1.26%     0.00%  demo     ld-linux-aarch64.so.1  [.] _start
+    1.26%     0.02%  demo     ld-linux-aarch64.so.1  [.] _dl_start
+    1.24%     0.00%  demo     ld-linux-aarch64.so.1  [.] _dl_start_final (inlined)
+    1.24%     0.00%  demo     ld-linux-aarch64.so.1  [.] _dl_sysdep_start
+    1.12%     1.12%  demo     ld-linux-aarch64.so.1  [.] _dl_relocate_object
+    1.12%     0.00%  demo     ld-linux-aarch64.so.1  [.] dl_main
+    1.12%     0.00%  demo     ld-linux-aarch64.so.1  [.] elf_dynamic_do_Rela (inlined)
+    1.12%     0.00%  demo     ld-linux-aarch64.so.1  [.] elf_machine_rela_relative (inlined)
     0.13%     0.13%  demo     ld-linux-aarch64.so.1  [.] __GI___tunables_init
     0.13%     0.00%  demo     ld-linux-aarch64.so.1  [.] get_next_env (inlined)
     0.02%     0.00%  demo     ld-linux-aarch64.so.1  [.] elf_get_dynamic_info (inlined)
     0.00%     0.00%  demo     [unknown]              [.] 0xffffd06fce411550
     0.00%     0.00%  demo     [unknown]              [k] 0xffffd06fcf1bde34
     0.00%     0.00%  demo     [unknown]              [.] 0xffffd06fcf1be9d8
     0.00%     0.00%  demo     [unknown]              [k] 0xffffd06fcf1be3d0
     0.00%     0.00%  demo     [unknown]              [.] 0xffffd06fcf1be9f0
```

perf record -F 1000 --call-graph dwarf –c ./demo & perf report

Using Checkers

- Using dedicated checkers is especially helpful to debug memory issues, resource leakage, and multithreading issues
- Memory error detectors: Valgrind Memcheck, MTrace, DMalloc.
- Multithreading issues: Helgrind

- valgrind --leak-check=yes myprog arg1 arg2

```c
#include <stdlib.h>

void f(void)
{
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;        // problem 1: heap block overrun
}                    // problem 2: memory leak -- x not freed

int main(void)
{
   f();
   return 0;
}
```

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (a.c:5)
==19182==    by 0x80483AB: main (a.c:11)
```

- valgrind --leak-check=full --track-fds=yes myprog arg1 arg2

https://valgrind.org/docs/manual/quick-start.html

- valgrind --tool=helgrind myprog arg1 arg2

a. Misuse of pthread APIs

```
Thread #1 unlocked a not-locked lock a
   at 0x4C2408D: pthread_mutex_unlock
   by 0x40073A: nearly_main (tc09_bad_
   by 0x40079B: main (tc09_bad_unlock.
  Lock at 0x7FEFFFA90 was first observe
   at 0x4C25D01: pthread_mutex_init (h
   by 0x40071F: nearly_main (tc09_bad_
   by 0x40079B: main (tc09_bad_unlock.
```

b. Inconsistent lock ordering

```
Thread #1: lock order "0x7FF0006D0 before 0x7FF6

Observed (incorrect) order is: acquisition of lo
   at 0x4C2BC62: pthread_mutex_lock (hg_intercep
   by 0x400825: main (tc13_laog1.c:23)

followed by a later acquisition of lock at 0x7F
   at 0x4C2BC62: pthread_mutex_lock (hg_intercep
   by 0x400853: main (tc13_laog1.c:24)

Required order was established by acquisition of
   at 0x4C2BC62: pthread_mutex_lock (hg_intercep
   by 0x40076D: main (tc13_laog1.c:17)

followed by a later acquisition of lock at 0x7F
   at 0x4C2BC62: pthread_mutex_lock (hg_intercep
   by 0x40079B: main (tc13_laog1.c:18)
```

c. Data race

```
Thread #1 is the program's root thread

Thread #2 was created
   at 0x511C08E: clone (in /lib64/libc-2.8.so)
   by 0x4E333A4: do_clone (in /lib64/libpthread-2.8.so)
   by 0x4E33A30: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.8.so)
   by 0x4C299D4: pthread_create@* (hg_intercepts.c:214)
   by 0x400605: main (simple_race.c:12)

Possible data race during read of size 4 at 0x601038 by thread #1
Locks held: none
   at 0x400606: main (simple_race.c:13)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
   at 0x4005DC: child_fn (simple_race.c:6)
   by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
   by 0x511C0CC: clone (in /lib64/libc-2.8.so)

Location 0x601038 is 0 bytes inside global var "var"
declared at simple_race.c:3
```

https://valgrind.org/docs/manual/hg-manual.html

Dig Deeper Into A Debugger (GDB)

A Brief History of GDB (GNU debugger)

- Came into existence around 1985, written by **Richard Stallman** along other early components of GNU Project.
- Maintained by **John Gilmore** from 1990-1993, later shifted to GDB Steering Committee under the Free Software Foundation.
- Initially a source-level debugger for Unix systems, supporting C and later C++ and Fortran
- Expanded to support many architectures (x86, ARM, MIPS, PowerPC, RISC-V, etc.) and multiple languages (Ada, Rust, Go, etc.).
- Modern GDB is still command-line based but integrates with many GUIs and IDEs (Eclipse, Qt Creator, Visual Code, etc).

https://en.wikipedia.org/wiki/GNU_Debugger

Key Capabilities

- Execution control
  - Breakpoints
  - Watchpoints
  - Catchpoints
- Step navigation
  - Step in/out/over
  - Step by line or by instruction
- State inspection
  - Variables, registers, frames
  - Expressions
  - Modify states on the fly
- Remote debugging (with Remote Stub)
  - Embedded system devices
  - OS kernel, e.g. debugging Linux kernel running in QEMU
- Extending GDB via Python scripting
- Tracepoints
  - Non-intrusive tracing
  - Remote targets only, relies on remote stub support
- Reverse Execution (Time Travel Debugging)
  - Debugging intermittent bugs (Heisenbug)
  - May need record and replay

https://sourceware.org/gdb/current/onlinedocs/gdb.html/

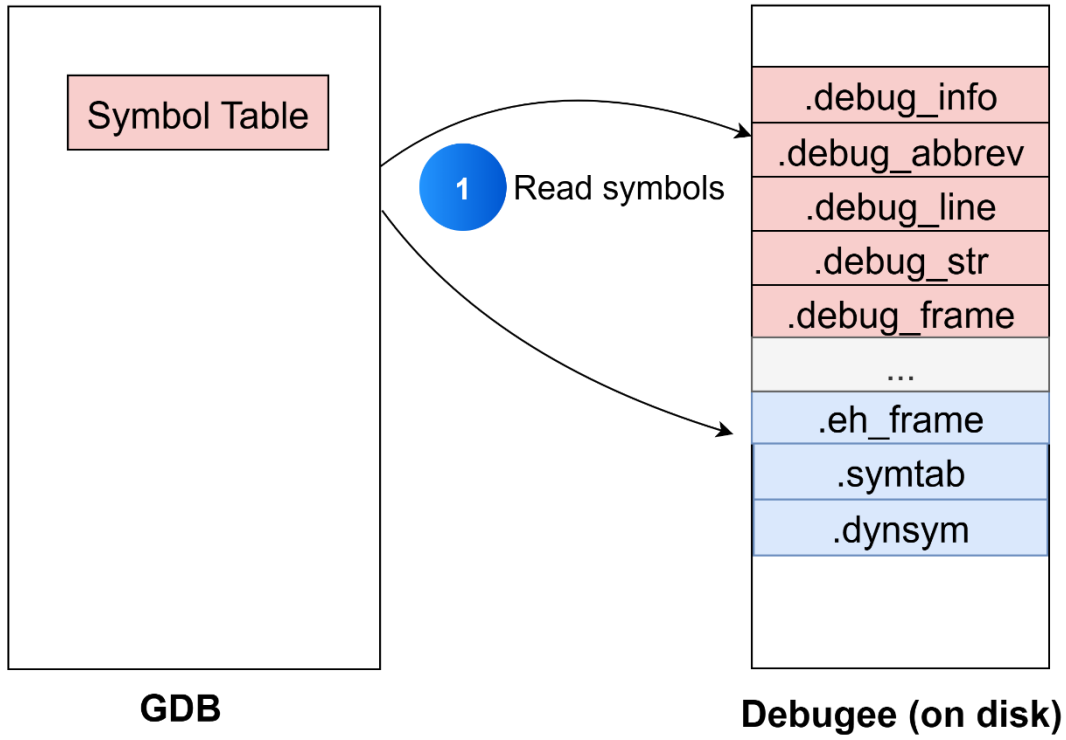
Typical GDB Workflow

1. Load debug symbols
2. Set a breakpoint
3. Run the debuggee until a breakpoint is hit
4. Inspect debuggee states
5. Step through code, e.g. single step

```
$ gcc -g -o test test.c
$ gdb ./test
```

1 Loade debug symbols

- Program should be compiled with debug info
- Read both debug info and standard symbols
- Only build partial symbol table for quick start up, other symbols are loaded on-demand



❖ Debug info:

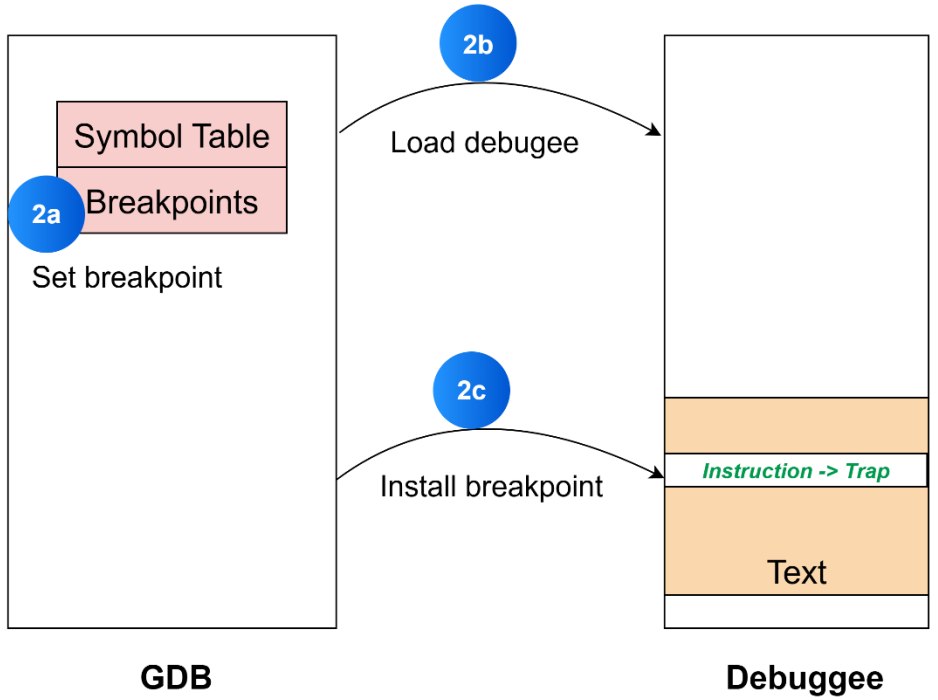
- Build with *–g*
- Format: DWARF (ELF), PDB (PE), COFF, STABS
- Embedded into binary or in a separate file
- Contains: variable and function names, type info, line number, call frame info, etc

https://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf

2 Set a breakpoint (software)

a. Set breakpoint internally, containing **location**, instruction, and address information
b. Load the debugee into a controlled environment (e.g. child process via ptrace syscall on Linux)
c. Replacing the instruction with "trap" instruction

```
...
$ gdb ./test
(gdb) break test.c:12
(gdb) run
```



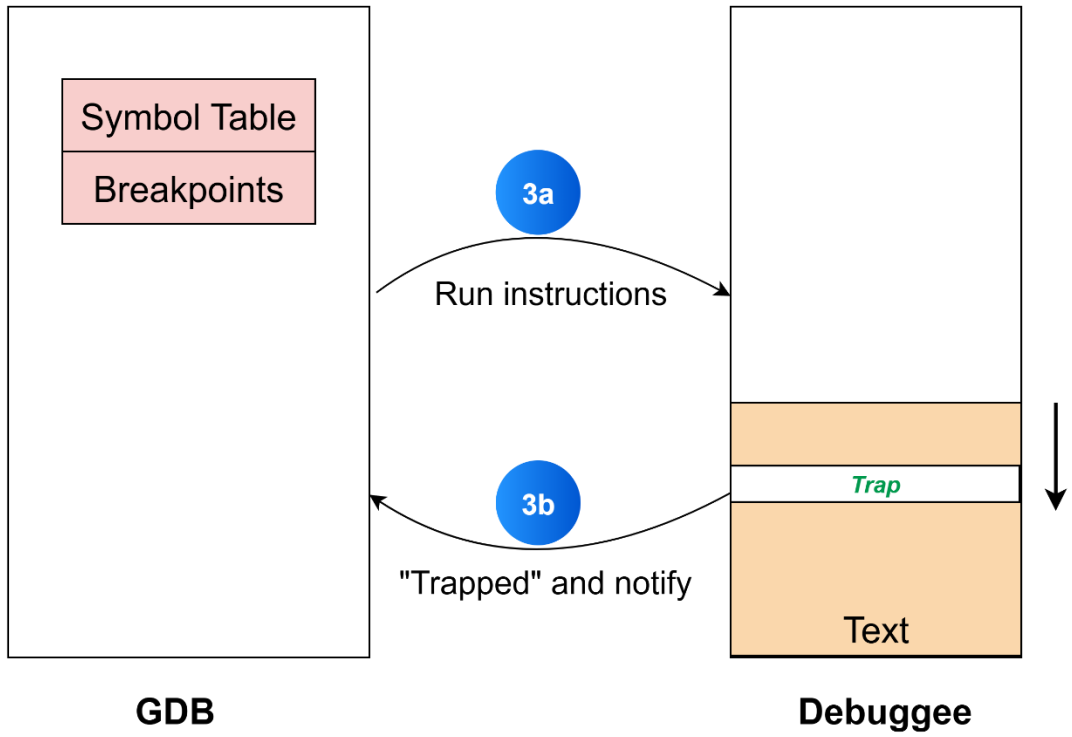
❖ Hardware breakpoints / watchpoints:

- Supported in many architectures including x86, ARM, etc.
- Only a few breakpoints / watchpoints are available,
- Installed by configuring debug registers instead of mutating debuggee memory
- GDB opt for software breakpoints by default
- As for watchpoints, GDB opt for hardware watchpoints by default since software watchpoints can slow down debugging significantly.

3 Run the debugee until breakpoint is hit

a. Run the debugee through
b. The breakpoint (i.e. trap instruction) is hit and debugee is stopped, and then GDB process is notified and take control

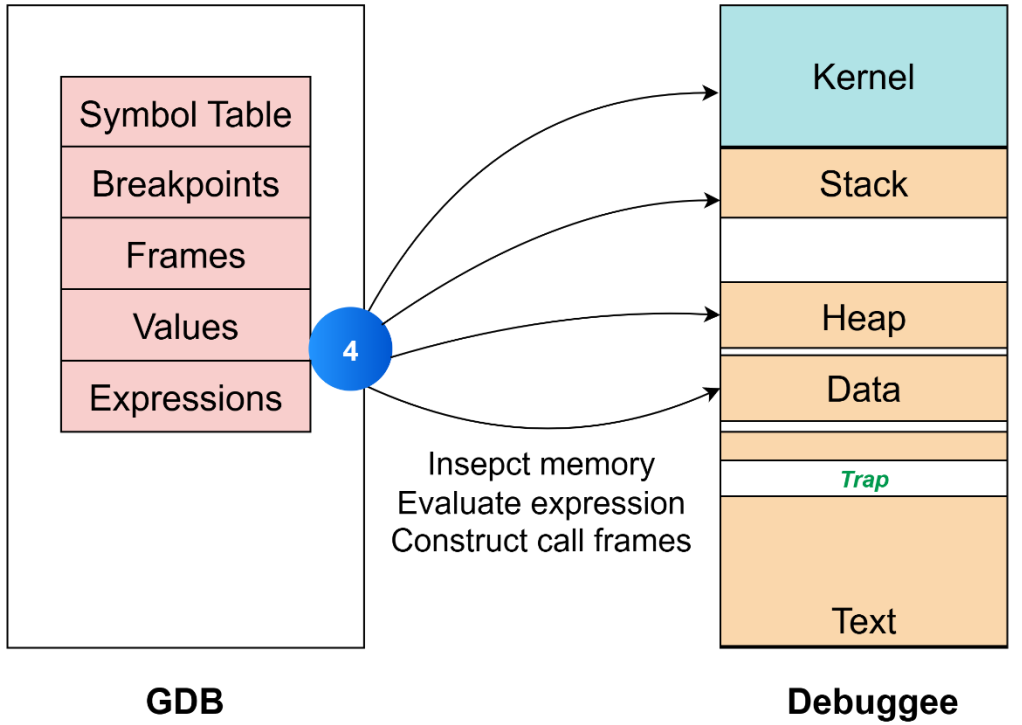
```
...
$ gdb ./test
(gdb) break test.c:12
(gdb) run
Breakpoint 1 at 0x12345678, file test.c, line12
```



4 Inspect the debuggee states

- Inspect variables, registers, frames, etc
- Fetched from both user and kernel space memory of the debugee process
- Support expressions of values

```
...
Breakpoint 1 at 0x12345678, file test.c, line12
(gdb) print val
(gdb) print a + b
(gdb) info registers
(gdb) backtrace
```



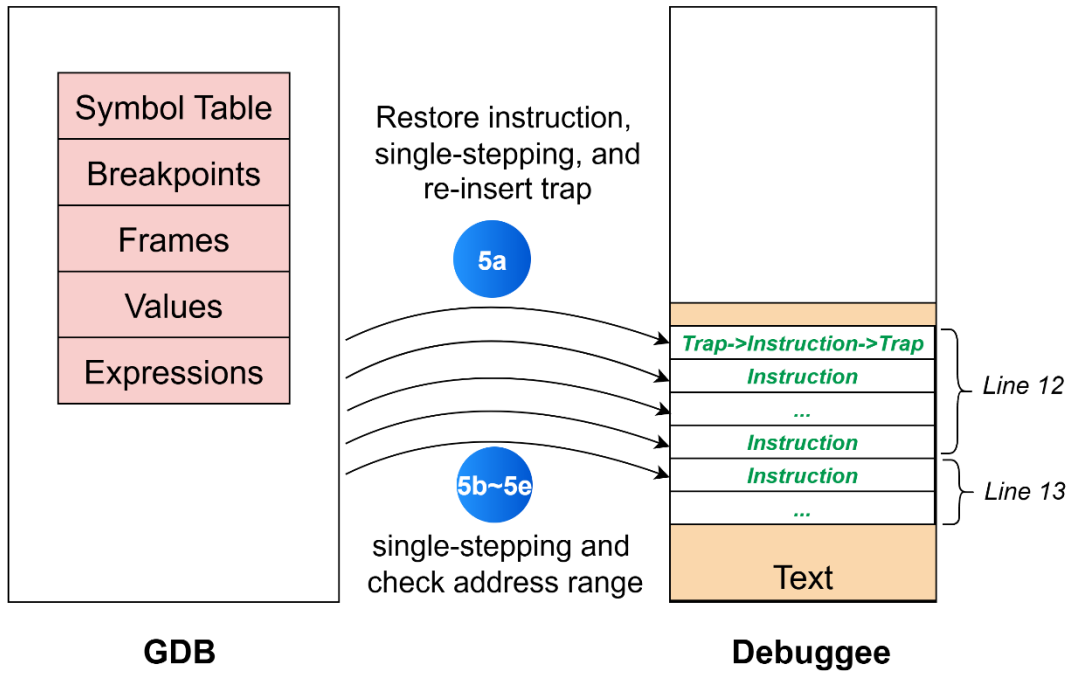
5 Single step to the next line

a. Restore the instruction, single-step the instruction, and re-install the breakpoint

b~e. Loop until instruction of next line is hit:

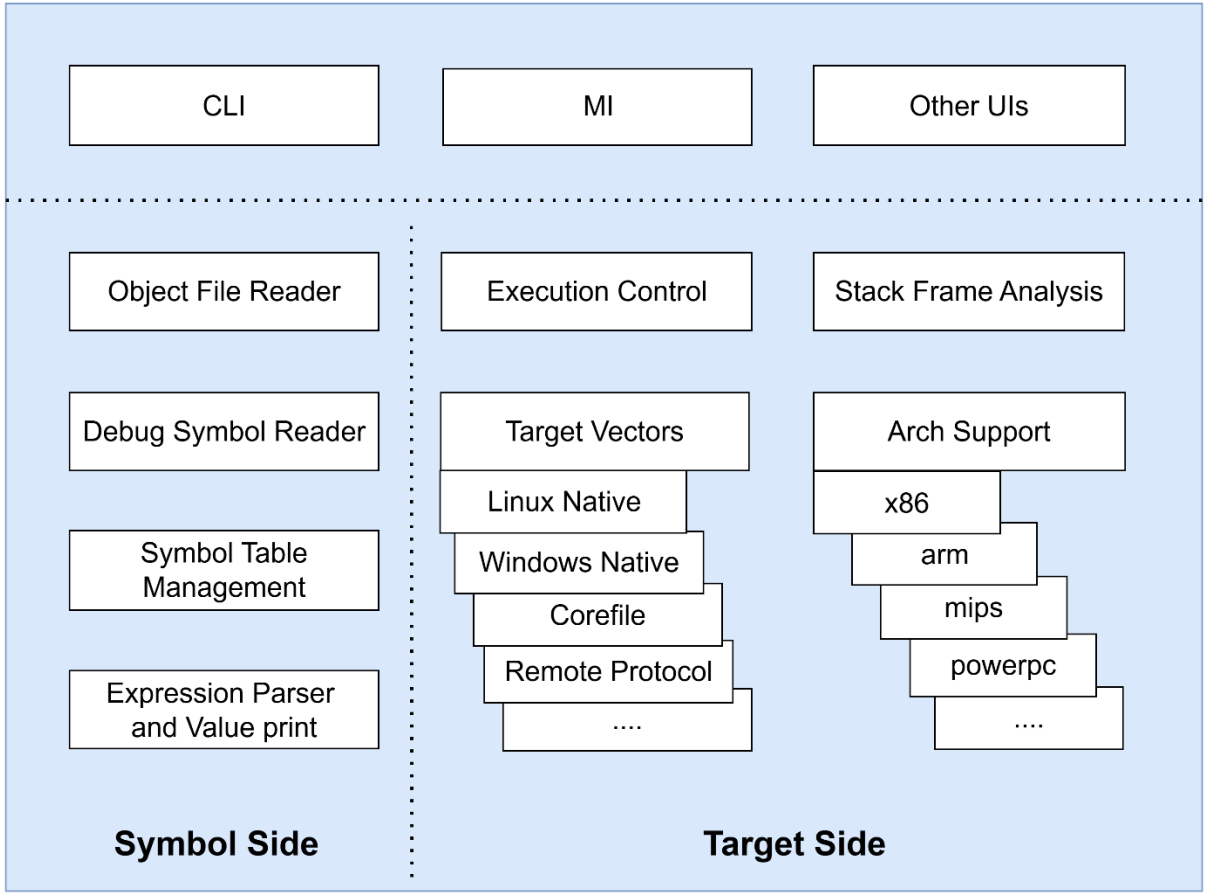
1) check if current instruction address passed the address range of current line;
2) if not, single-step current instruction;
3) if yes, we've arrived at the next source line, stop and notify GDB

```
...
(gdb) info registers
(gdb) backtrace
(gdb) step
13          int c = add (a + b);
```



GDB Internals

- CLI / (Machine Interface MI) / Other UIs
- Symbol Side
  - Extracted symbolic info from executable files
  - Parse expressions
  - Find memory address of a given line number
  - List source code
- Target Side
  - Manipulating with the target system (start/stop execution, read/write memory and registers, analyse call frames, etc)
  - Target vectors: interface for operations
  - Arch support (gdbarch): Architecture constraints



Based on: https://aosabook.org/en/v2/gdb.html?

Demo

Setup

- OS: Ubuntu
- IDE: Visual Studio Code + C/C++ Extension + GCC + GDB
- Compiler: GCC, options: -g -O0

Basics

- Workspace
- Breakpoint
- Step Out/In/Over
- Memory Inspection: variable, expression (incl. function),call stack

Advanced

- Conditional breakpoint (buggy1.c)
- Watchpoint (buggy2.c)
- Reverse Execution (from 8:40)

Thank You