Lecture 19: Debugging with GDB

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/19-debugging.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

Here we discuss debugging with GDB.

Buggy Program

Consider the following program:

```c
/* buggy-program.c: */

#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE  100

int fill_array(int *array, int size) {
    for(int i=0; i<size; i++)
        array[i] = rand()%10;
}

/* array[slot] = value */
int update_slot(int *array, int slot, int value) {
    array[slot] = value;
    printf("Updated index %d to %d\n", slot, value);
}

int process_array(int *array, int size) {
    int ii = 1000000;

    for(int i=0; i<size; i++) {
        /* If the value is even, change it to 1000000 */
        if(!(array[i] % 2)) {
            update_slot(array, ii, i);
        }
    }
}

int main(int argc, char **argv) {
```

```
31      int *array = malloc(ARRAY_SIZE * sizeof(int));
32
33      fill_array(array, ARRAY_SIZE);
34      process_array(array, ARRAY_SIZE);
35
36      free(array);
37      return 0;
38  }
```

It contains a bug and crashes at runtime:

```
$ gcc buggy-program.c -o buggy-program
$ ./buggy-program
[2]    9983 segmentation fault  ./buggy-program
```

This behaviour gives very little information about what actually went wrong. Blindly debugging, e.g. by adding `printf` statements, to try to understand what happens is useful but very cumbersome, especially on large codebases. We'll use a debugger to easily find the bug.

Fixing the Issue with GDB

GDB is the GNU debugger, a command line tool that helps inspect the state of the program at runtime to understand and fix bugs. To use it, the program should be compiled with debug symbols using the `-g` flag passed to the compiler:

```
$ gcc -g buggy-program -o buggy-program
$ gdb buggy-program
```

To launch the execution of the program within the debugger, use the `run` command:

```
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
0x00005640b87c51fe in update_slot (array=0x56..., slot=1000000, value=1) at buggy-program.c:13
13      array[slot] = value;
```

The debugger executed the program until the crash happened. It indicates us the faulty line of code, line 13 in `buggy-program.c`. Something wrong happens when trying to write in a slot of the array. We can see from the value of `update_slot`'s parameters that `slot` is way too large (`1000000`) with respect to the array size (`100`). In effect the program tries to access the memory at address `array + 1000000 * sizeof(int)`, which is probably not mapped, so a page fault happens, and the operating system kills the program. That's our bug, the programmer inverted the second and third arguments when calling `update_slot`.

Other Basic GDB Features

Breakpoints

Breakpoints can be placed at various locations (referenced by their line number in the source files) in the program. When a breakpoint is hit during execution under the debugger, the program will pause and the debugger will let the user inspect the state of the program. Here is an example, still using our buggy program, we want to pause the execution when entering the function `fill_array` which is at line 6:

```
(gdb) br buggy-program.c:6
Breakpoint 1 at 0x5640b87c5178: file buggy-program.c, line 7.
(gdb) run
Breakpoint 1, fill_array (array=0x5568886282a0, size=100) at buggy-program.c:7
7       for(int i=0; i<size; i++)
```

The execution starts and pauses when `fill_array` is called. Notice that the debugger corrected the line to 7 of the source file, i.e. the first instruction of the function. The user can then choose different ways to continue the execution of the program:

- The `continue` command will resume execution until the next breakpoint/crash is hit.

- The `step` command will execute the next line of code and will pause right after that. If that line of code is a function call, the debugger will dive into the function and the pause will happen on the first instruction of that function.
- The `next` command will execute the next line of code and will pause right after that. If that line of code is a function call, the debugger will execute the entire function call before pausing on the next line of code of the calling context.

Breakpoints can be deleted with the `del` command, followed by the breakpoint identifier (e.g. `1` for the one seen above).

Printing Values and Addresses

During execution, when a breakpoint/crash is hit, the user can print values and addresses to inspect the state of the program with the `p` command:

```
(gdb) p array
$1 = (int *) 0x5568886282a0
(gdb) p array[0]
$2 = 0
(gdb) p size
$3 = 100
(gdb) p &size
$4 = (int *) 0x7ffdde0ba444
```

Navigating the Call Stack

When inspecting the state of the program, the user can go up and down the call stack with the `up` and `down` commands:

```
(gdb) up
#1  0x0000556886ba22ac in main (argc=1, argv=0x7ffdde0ba5a8) at buggy-program.c:31
31      fill_array(array, ARRAY_SIZE);
(gdb) down
#0  fill_array (array=0x5568886282a0, size=100) at buggy-program.c:7
7       for(int i=0; i<size; i++)
```

Conditional Breakpoints

The user can instruct GDB to pause execution when a breakpoint is hit only under certain condition, e.g. a variable having a certain value. Here is an example in which we put a breakpoint in `fill_array` at the 42nd iteration of the loop:

```
br buggy-program.c:8 if i==42
Breakpoint 1 at 0x1181: file buggy-program.c, line 8.
(gdb) run
Breakpoint 1, fill_array (array=0x5586448b22a0, size=100) at buggy-program.c:8
8           array[i] = rand()%10;
(gdb) p i
$1 = 42
```

Further Resources

The official documentation is available here: https://www.sourceware.org/gdb/documentation/. A reference card listing the most important commands is available here: https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf. GDB's interface can be heavily customised and made more user-friendly, see for example https://github.com/cyrus-and/gdb-dashboard.