Operating Systems [2021-2022]

Tutorial – GNU Debugger

<u>Note</u>: this tutorial will <u>not</u> be done during the practical classes, although questions will be clarified by the teacher.

Introduction

The GNU Debugger (GDB) is a tool to analyze a running program by stopping its execution on specific points of interest. It supports several languages, like C and C++, and can be used to detect and understand errors in the code. This tutorial presents the basic concepts and commands required to use GDB.

Concepts and Commands

In order to use GDB with a program it must be compiled with the extra information that is required for debugging. This can be achieved by adding the **-g** flag into the **gcc** compile instruction, like: `gcc -Wall -g <source files> -o <output file>`.

To start debugging a program one can execute **gdb** on the shell followed by the path to it, like: `gdb <program file>`. Alternatively, the program can be loaded afterwards with the command **file**, like: `file <program file>`. GDB works as an interactive shell, receiving commands as input. To run the loaded program the **run** keyword is used and, when the debugging process ends, GDB can be closed with **quit**.

Defining breakpoints is often the proper way to analyze a running program. These are the specific points where the execution is going to stop, allowing the user to continue running the program line by line and see the variables content. Breakpoints can be added using the **break** command, followed by the number of the line of code where the execution will stop, like `break <number>`. Alternatively, the command can be followed by the name of a function, which in this case inserts the breakpoint into its first line. Moreover, breakpoints can be triggered only if a certain condition is true. This can be helpful because it avoids stopping the execution in unnecessary scenarios for the debugging purpose. The command syntax is the same explained before, but an if condition is added at the end, like: `break <number> if <condition>`. This condition can include variables that are within the scope of the target line of code. Breakpoints are associated with an integer identifier that starts with 1, and these IDs can be used to delete

them through the command **delete** followed by the integer, like: `delete <id>`. The full list of breakpoints can be presented using the command **info breakpoints**.

A program that is currently stopped can proceed to the next breakpoint by using the command **continue**. However, the user often needs to execute the following lines of code one by one, in order to understand the changes they cause. The commands **step** and **next** serve this purpose by resuming the execution line by line. The difference between them is that **next** does not enter inside functions when they are called, treating them as a single line.

Something that is often useful during the debugging process is to check the contents of specific variables. This can be done using the command **print** followed by the name of the variable, like: `print <variable>`. Changing the value of a variable is also possible using the command **set variable**, like: `set variable <variable> = <value>`.

Similarly to breakpoints, one can also create watchpoints. These stop the execution of the program when the value of the variable being watched changes. They can be inserted using the command **watch** followed by the variable, like: `watch <variable>`.

The **help** keyword can be used to understand how other commands work by receiving them as argument, like: `help <command>`.

The following list presents a brief overview of the commands described above and adds some others that are also commonly used:

- **run** [parameter1] [parameter2] […]   # run the loaded program
- **continue**   # continue execution
- **step**   # advance one step in the program (including any called function)
- **next**   # execute until next line in source code (function calls are executed without stopping)
- **finish**   # execute a function till its end

- **list** {line or function}   # show source code
- **print** {variable or expression}   # inspect values of variables or evaluate expressions
    - By appending / and a formatting specifier, you can change the way gdb outputs a value (e.g. /x results in hexadecimal output, /t in binary output). Use **help x** to view other formats.
- **display** {variable or expression}   # display the variable or expression each time the program stops
- **x** {address}   # examine memory (uses the same formats as **print**)
- **set** {variable} = {value}   # update the value of an existing variable
- **backtrace**   # show the list of current function calls
- **where**   # show where the crash occurred
- **frame** {frame number}   # show the currently selected stack frame, or select another stack frame by number

- **Ctrl+C** interrupts the program execution at any time and returns to gdb prompt.

- **info break**   # display all active breakpoints
- **break** {line or function}   # set a breakpoint in a line or function

- **enable** {breakpoint}   # enable breakpoint
- **disable** {breakpoint}   # disable breakpoint
- **delete** {breakpoint}   # delete breakpoint
- **break** {line or function} **if** {condition}   # set a conditional breakpoint
- **condition** {breakpoint} {condition}   # add a condition to an existing breakpoint
- **watch** {variable}   # set a watchpoint to a variable

- **quit**   # close debugger

Exercise 1

Analyze the following code that prints an integer with *n* digits, each one generated randomly. Save the code to a file called "debug1.c".

```c
1.   #include <stdio.h>
2.   #include <stdlib.h>
3.   #include <time.h>
4.
5.   int generate_number(int max)
6.   {
7.       return rand() % max + 1;
8.   }
9.
10.  int main(void)
11.  {
12.     int i, n;
13.
14.     srand(time(NULL));
15.
16.     printf("n = ");
17.     scanf("%d", &n);
18.
19.     for (i = 0; i < n; i++)
20.         printf("%d", generate_number(9));
21.
22.     printf("\n");
23.     return 0;
24.  }
```

a) Compile the code.
   Command: *gcc -Wall -g debug1.c -o debug1*

b) Start GDB.
   Command: *gdb debug1*

c) Add a breakpoint to the function *generate_number* (see Figure 1).
   Command: *break generate_number*

d) Add a conditional breakpoint to the line inside the *for* loop that only stops the execution in the last iteration (see Figure 1).
   Command: *break 20 if i == n-1*

Figure 1 - Breakpoints

e) Run the program. The execution should stop on line 7, the first and only one from the function *generate_number* (see Figure 2).
Command: **run**

```
n = 5

Thread 2 hit Breakpoint 1, generate_number (max=9) at debug.c:7
7           return rand() % max + 1;
(gdb) ▯
```

Figure 2 - Execution stopped on line 7

f) Check the breakpoints previously created and delete the one from the function *generate_number* (see Figure 3).
Command: **info breakpoints** and **delete 1**

```
n = 5

Thread 2 hit Breakpoint 1, generate_number (max=9) at debug.c:7
7           return rand() % max + 1;
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000100000e8b in generate_number at debug.c:7
        breakpoint already hit 1 time
2       breakpoint     keep y   0x0000000100000eff in main at debug.c:20
        stop only if i == n-1
(gdb) delete 1
(gdb) ▯
```

Figure 3 - List of the previously created breakpoints

g) Proceed to the next breakpoint. Notice that the execution now stops on line 20 (the conditional breakpoint). Print the contents of the variable *i* to confirm that it is the last iteration from the *for* loop (see Figure 4).
Command: **continue** and **print i**

```
(gdb) continue
Continuing.

Thread 2 hit Breakpoint 2, main () at debug.c:20
20              printf("%d", generate_number(9));
(gdb) print i
$1 = 4
(gdb) ▯
```

Figure 4 - Execution stopped on line 20

h) Change the value of the variable *i* to 3. Replace the current breakpoint to a watchpoint for the same variable.
Command: **set variable i = 3** and **delete 2** and **watch i**

i) Proceed to the watchpoint. Notice that the execution now stops on line 19, where the variable *i* changes its value (see Figure 5).
Command: ***continue***

```
(gdb) continue
Continuing.

Thread 2 hit Hardware watchpoint 3: i

Old value = 3
New value = 4
0x0000000100000f20 in main () at debug.c:19
19          for (i = 0; i < n; i++)
(gdb)
```

Figure 5 - Execution stopped on line 19

j) Proceed line by line until the return of the main function. The generated digits should appear on the screen (see Figure 6). Proceed to end the program.
Command: ***next*** (5x) and ***continue***

```
Old value = 4
New value = 5
0x0000000100000f20 in main () at debug.c:19
19          for (i = 0; i < n; i++)
(gdb) next
22          printf("\n");
(gdb) next
812526
23          return 0;
```

Figure 6 - Run the remaining lines of code one by one

Exercise 2

Analyze the following code that generates a random password. The number of characters is 40 by default, but it can be passed as an argument. Save the code to a file called "debug2.c".

```
1.   #include <stdio.h>
2.   #include <stdlib.h>
3.   #include <time.h>
4.   #include <string.h>
5.
6.   char select_char(char * arr, int size)
7.   {
8.     int idx = rand() / size;
9.     return arr[idx];
10.  }
11.
12.  int main(int argc, char* argv[])
13.  {
14.    int i, n, s;
15.    char * alphabet = "abcdefghijklmnopqrstuvwxyz123456789.,;-_";
16.
17.    if (argc < 2)
18.      s = 40;
19.    else
20.      s = atoi(argv[1]);
21.
22.    n = strlen(alphabet);
23.
24.    srand(time(NULL));
25.
26.    for (i = 0; i < s; i++)
27.      printf("%c", select_char(alphabet, n));
28.
29.    printf("\n");
30.    return 0;
31.  }
```
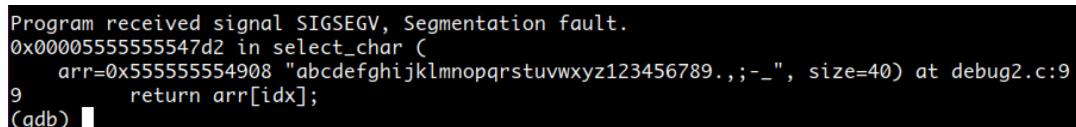
a) Compile the code and start GDB.
   Command: *gcc -Wall -g debug2.c -o debug2* and *gdb debug2*

b) Run the program assuming that the generated password will have 25 characters. It should stop on line 9 with a segmentation fault error associated with an invalid access to the array **arr** (see Figure 7).
   Command: *run 25*


```
Program received signal SIGSEGV, Segmentation fault.
0x00005555555547d2 in select_char (
    arr=0x555555554908 "abcdefghijklmnopqrstuvwxyz123456789.,;-_", size=40) at debug2.c:9
9          return arr[idx];
(gdb)
```

Figure 7 - Segmentation fault on line 9.

c) Display the backtrace to the error line (see Figure 8) and print the values of variables *i*, *size* and *idx* (see Figure 9 and Figure 10). Notice that, in order to print variable *i*, a change of context is required since it is declared inside the *main* function.

Command: **frame 1** and **print i** and **frame 0** and **print idx** and **print size**

Note: the command **x** can be used as an alternative to **print** - i.e. **x /d &idx**.

```
(gdb)
#0  0x00005555555547d2 in select_char (
    arr=0x555555554908 "abcdefghijklmnopqrstuvwxyz123456789.,;-_", size=40) at debug2.c:9
#1  0x0000555555554850 in main (argc=2, argv=0x7fffffffdd98) at debug2.c:27
(gdb)
```

Figure 8 - Backtrace to the error line.

```
(gdb) frame 1
#1  0x0000555555554850 in main (argc=2, argv=0x7fffffffdd98) at debug2.c:27
27              printf("%c", select_char(alphabet, n));
(gdb) print i
$3 = 0
```

Figure 9 - Print variable *i* contents.

```
(gdb) frame 0
#0  0x00005555555547d2 in select_char (
    arr=0x555555554908 "abcdefghijklmnopqrstuvwxyz123456789.,;-_", size=40) at debug2.c:9
9               return arr[idx];
(gdb) print idx
$4 = 51080047
(gdb) x /d &idx
0x7fffffffdc6c: 51080047
(gdb) print size
$5 = 40
```

Figure 10 - Print variables *idx* and *size* contents.

d) The analysis reveals that the variable *idx* has an incorrect value. The bug is a wrong operator on line 8 – it should be *rand() % size* instead of *rand() / size*. Fix the bug and add a break point on line 9. Print the value of the variable *idx* for a few iterations in order to understand if its value is always valid (see Figure 11).

Command: **break 9** and **display idx** and **continue** (3x)

```
Breakpoint 1, select_char (arr=0x555555554908 "abcdefghijklmnopqrstuvwxyz123456789.,;-_",
    size=40) at debug2.c:9
9               return arr[idx];
1: idx = 33
(gdb) continue
Continuing.

Breakpoint 1, select_char (arr=0x555555554908 "abcdefghijklmnopqrstuvwxyz123456789.,;-_",
    size=40) at debug2.c:9
9               return arr[idx];
1: idx = 18
```

Figure 11 - Print variable *idx* for 2 iterations.

e) Remove the breakpoint previously created and proceed to end the program. The generated password should be displayed before the process exits.

Command: **delete 1** and **continue**

Graphical Interface for GDB

Many of the modern IDEs have graphical interfaces for debugging purposes. These GUIs integrate with GDB, performing the operations already described, and can provide a friendlier way to debug a program. CLion and NetBeans are 2 examples, but there are many other IDEs supporting the same integration with similar interfaces.

CLion is a cross-platform IDE for C and C++, developed by JetBrains, and it is widely used for these languages. A tutorial describing how to debug a program using it is available on the IDE official blog (https://blog.jetbrains.com/clion/2015/05/debug-clion/) and official YouTube channel (https://www.youtube.com/watch?v=wUZyoAnPdCY).

NetBeans is another well-known IDE that supports the majority of the languages, including C. The official website includes a small tutorial describing how to debug C/C++ projects using it (https://netbeans.org/kb/docs/cnd/debugging.html).