Debugging tools

1 What is a debugger?

A *debugger* is a software tool that is used to locate errors or flaws in a so-called *target* program. It provides a mechanism for controlled execution of the target and has the ability to stop execution when specific conditions are encountered. This makes it possible to inspect or monitor variables, sequences of function calls, program flow, etc. In this note, we will consider two different debuggers, namely the GNU debugger (also known as *GDB*) which may be used with GCC and the LLDB debugger which may be used with Clang.

In order to use these debuggers, we will need to instruct the compiler to embed some debug information in the executable. This is done by including the compiler flag `-g` in the `CFLAGS` variable in our makefile:

```
CFLAGS=-g -Wall -std=c99
```

With the purpose of illustrating some basic debugging principles, we will consider the following program that computes the *n*th Fibonacci number:

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

long fibonacci(long n) {
  if ( n == 0 )
    return 0;
  else if ( n == 1 )
    return 1;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}

int main(int argc, char const *argv[]) {
  long n = 0;
  if (argc != 2) {
    fprintf(stderr,"Usage: %s n\n",argv[0]);
    return EXIT_FAILURE;
  }
  n = atoi(argv[1]);
  printf("f(%ld) = %ld\n",n,fibonacci(n));
  return EXIT_SUCCESS;
}
```

The Fibonacci numbers may be defined recursively as

$$f(n) = f(n-1) - f(n-2), \quad n \geq 2,$$

where $f(0) = f(1) = 0$. The program takes a single command line argument which is converted to an integer using `atoi()` and passed to `fibonacci()`. Here is an example:

```
$ ./myprog 6
f(6) = 8
```

Note that our program does not check the input. It turns out that it will crash if the user enters a negative number:

```
$ ./myprog -1
Segmentation fault
```

We will now use a debugger to find out why the program crashes. To this end, we will compile the program with debug information and run it under our debugger. We will also see how to use breakpoints to halt program execution, allowing us to inspect variables and the stack.

2 The GNU debugger (GCC on Linux/WSL)

2.1 Getting started

We start by checking if GDB is available. This can be done using the `which` command which is used to locate a program file in the user's path:

```
$ which gdb
gdb not found
```

If `gdb` is not found in the path, install it as follows:

```
$ sudo apt-get install gdb
```

2.2 Running your program under GDB

GDB can be started from the terminal by issuing the `gdb` command:

```
$ gdb
```

A prompt with the text `(gdb)` indicates that GDB is ready to read a command. The next step is to specify the program name (which we assume is `myprog`) and arguments:

```
(gdb) file myprog
Reading symbols from myprog...done.
(gdb) set args 6
```

To start execution, issue the `run` command (or simply `r`):

```
(gdb) run
```

The debugger will halt execution if the program crashes, and otherwise your program will exit normally. Finally, use the `quit` command to exit GDB:

```
(gdb) quit
```

Note that the program name and arguments can also be specified when launching GDB as follows:

```
$ gdb --args myprog 6
```

2.3 Breakpoints

Before we start executing our program in GDB, we may instruct GDB to stop execution at specific points in our program. Such points are referred to as breakpoints. For example, we may create a breakpoint on line 5 in `myprog.c` using the `break` command (or simply `br`) as follows:

```
(gdb) break myprog.c:5
Breakpoint 1 at 0x400613: file myprog.c, line 5.
```

It is also possible to set at breakpoint for a function. For example, we may choose to stop our program as soon as `main` is called:

```
(gdb) break main
Breakpoint 2 at 0x40066b: file myprog.c, line 13.
```

A breakpoint can be removed using the `clear` command (or simply `cl`):

```
(gdb) clear main
Deleted breakpoint 2
```

Now issue the `run` command to start the program:

```
(gdb) run
Starting program: /home/user/myprog 6

Breakpoint 1, fibonacci (n=6) at myprog.c:5
```

```
5     if ( n == 0 )
```

GDB informs us that Breakpoint 1 (our only breakpoint since we deleted Breakpoint 2) was reached, causing the program has stopped at this point. We may now do several things such as

- continue execution (`continue` or `c`)
- step through the program (`step` or `s` , `next` or `n`)
- inspect variables (`info args` , `info locals` , `print`)
- inspect the call stack, also known as a backtrace (`backtrace` or `bt`).

2.4 Inspecting the call stack

GDB can be used to inspect the call stack when the program has stopped. Recall that the call stack contains a stack frame for every unfinished function call, and the stack frames contain information such as input arguments and local variables.

We now launch GDB with `myprog` as the target, set a breakpoint on the function `fibonacci` , and start our program:

```
$ gdb --args myprog 6
(gdb) break fibonacci
(gdb) run
```

Once the program stops, use the `backtrace` command to print a backtrace of the entire stack:

```
(gdb) backtrace
#0  fibonacci (n=6) at myprog.c:5
#1  0x004006b5 in main (argc=2, argv=0x7ffd8d698f68) at myprog.c:20
```

At this point, the call stack consists of two stack frames: frame 0 (the currently executing frame) and frame 1 (its caller) which in this case is the call stack entry (the `main` function). A particular stack frame can be selected using the `up` , `down` , and `frame` (or simply `f`) commands. Frame 0 is the current frame, which is selected initially when the program stops. The `up` command selects the next frame:

```
(gdb) up
#1  0x004006b5 in main (argc=2, argv=0x7ffd8d698f68) at myprog.c:20
20    printf("f(%ld) = %ld\n",n,fibonacci(n));
```

A specific frame can be selected with the `frame` command followed by the frame index (e.g, `frame 5`) which is convenient for navigating a stack with many frames.

2.5 Debugging a program crash

We will now use GDB to investigate why our program crashes when the input argument is negative. We will do this by inspecting the call stack. First we start the program with `-1` as input:

```
$ gdb --args myprog -1
(gdb) run
Starting program: /home/user/myprog -1

Program received signal SIGSEGV, Segmentation fault.
0x000000000040062a in fibonacci (n=-174712) at myprog.c:10
10      return fibonacci(n-1) + fibonacci(n-2);
```

GDB informs us that a segmentation fault (i.e., a memory access violation) has caused the program to stop, and this has happened on line 10 in `myprog.c` with `n` begin equal to `-174712`. Issuing the backtrace command `bt 5` shows the innermost five frames the call stack:

```
#0  0x000000000040062a in fibonacci (n=-174712) at myprog.c:10
#1  0x000000000040062f in fibonacci (n=-174711) at myprog.c:10
#2  0x000000000040062f in fibonacci (n=-174710) at myprog.c:10
#3  0x000000000040062f in fibonacci (n=-174709) at myprog.c:10
#4  0x000000000040062f in fibonacci (n=-174708) at myprog.c:10
```

Our program firsts call `fibonacci(-1)`, then `fibonacci(-2)`, `fibonacci(-3)`, and so on due to the recursive nature of `fibonacci()`. However, our program crashes when reaching the call `fibonacci(n-1)` with `n` equal to `-174712`. This may seem puzzling at first, but recall that every function call pushes a stack frame onto the call stack. The program crashes when we run out of stack memory, a phenomenon known as a *stack overflow*.

2.6 Monitoring variables with watchpoints

Watchpoints are similar to breakpoints, but unlike breakpoints they are not set for a function or a line of code. They are set on variables and are used to stop the target program when those variables are read or written. Watchpoints are set using the commands `watch`, `rwatch`, and `awatch`, followed by an expression. The `watch` command will cause GDB to stop execution whenever the value of the expression changes, `rwatch` will stop execution whenever the value of the expression is read, and `awatch` will stop execution whenever the value of the expression is either read or written. This is illustrated in the following example:

```
$ gdb --args myprog 5
(gdb) break main
Breakpoint 1 at 0x40066b: file main.c, line 14.
(gdb) run
Starting program: /home/user/myprog 5

Breakpoint 1, main (argc=2, argv=0x7fffffffecc8) at main.c:14
14        long n = 0
(gdb) watch n
Hardware watchpoint 2: n
(gdb) continue
```

```
Continuing.
Hardware watchpoint 2: n

Old value = 0
New value = 5
main (argc=2, argv=0x7fffffffecc8) at main.c:20
20          printf("f(%ld) = %ld\n",n,fibonacci(n));
```

We end this section by noting that the interested reader may find more information about GDB in the GDB documentation or by writing `help` at the GDB prompt. Moreover, Online GDB is an online compiler and debugger tool that allows you to play around with and get to know GDB without the need to install anything, and Windows/WSL users can find a tutorial here.

3 The LLDB debugger (Clang on macOS)

The LLDB debugger is in many ways similar to GDB, so we only provide a minimal example. To launch LLDB and attach our program with the input argument `6`, issue the following command:

```
$ lldb -- myprog 6
```

We may now set a breakpoint in line 5 and start the program:

```
(lldb) breakpoint set -l 5
(lldb) run
```

Once the program stops, you can do things such as

- continue execution (`continue` or `c`)
- step through program (`step` or `s` , `next` or `n`)
- backtrace (`backtrace` or `bt`)
- select a stack frame (`frame select` or `fr s`)
- show local variables (`frame variable` or `fr v`)
- print variables (`print`)
- set watchpoints (`watchpoint set` or `wa set`)
- get help using the `help` command.

Finally, a table of GDB commands with the LLDB counterparts can be found here, and an official LLDB tutorial is available here.