COMPSCI2030 Systems Programming

Bugs

Yehia Elkhatib

Bugs

- Software development is not free of errors
- We call these software errors *bugs*
  - *syntactical* – grammar => compiler
  - *semantic* – meaning => you
- We will catalog a number of tools to help identify and understand bugs
  - **Debuggers:** run a program in a controlled environment to investigate its execution
  - **Static analysis tools:** reason about a program's behaviour without running it
  - **Dynamic analysis tools:** add instructions to a program to detect bugs at runtime

Segmentation fault

- This runtime error message is one of the most common in C systems

```
[1] 83437 segmentation fault ./program
```

- It is raised by the hardware notifying the OS that your program has attempted to access a restricted area of memory
  - The OS will then immediately terminate your program
- The most common causes
  - **Dangling pointer**
  - **Dereferencing a NULL pointer**
  - **Writing to read-only memory**
  - **A buffer overflow**
  - **A stack/heap overflow**

Common Causes of SegFaults

- Dangling pointer
  - a reference to memory which has been de-allocated then allocated to another variable
  - if you `NULL` after `free`ing, it is easier to check if this pointer is "active"
- Derefrencing a `NULL` pointer
  - `NULL` is an invalid memory location
- Buffer overflow
  - accessing memory outside allocated bounds, typically with arrays
  - you *can* read the values in these extra locations, but this data would be corrupted
- Stack overflow
  - trying to write more data than your allocated memory
  - often triggered by a recursion without a base case
- Heap overflow
  - Memory leaks

Debugger

- The two most popular debuggers are GDB and LLDB (very similar)

1. We compile with the -g flag to add debug information into the binary
2. Instead of executing the program normally…
   We start the debugger and then load the program

```
$ ./program arg1 arg2 ; "Run program normally"
$ gdb --arg ./program arg1 arg2 ; "Load program in GDB debugger"
$ lldb -- ./program arg1 arg2 ; "Load program in LLDB debugger"
```

3. Inside the debugger, common commands:

```
(lldb) run b n s bt list quit
```

- A full list of LLDB commands
  https://lldb.llvm.org/lldb-gdb.html
  https://lldb.llvm.org/use/tutorial.html

Where did the segfault come from?

- We load and run the program in the debugger

```
$ lldb -- ./program 12345
(lldb) run
Process 85058 launched: './program' (x86_64)
2018-10-21 20:56:00.106714+0100 program[85058:12827554] detected buffer overflow
Process 85058 stopped
...
```

- The debugger has now stopped the execution
- Using the `bt` (*backtrace*) command we investigate the calls leading to the segfault

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGABRT
* frame #0: 0x00007fff76d1ab86 libsystem_kernel.dylib`__pthread_kill + 10
...
frame #6: 0x00007fff76ca8e84 libsystem_c.dylib`__strcpy_chk + 8
frame #7: 0x0000000100000eaf program overflow(argc=2, argv=0x00007ffeefbff300) at program.c:35
frame #8: 0x0000000100000efb program`main(argc=2, argv=0x00007ffeefbff3c8) at program.c:42
frame #9: 0x00007fff76bdc085 libdyld.dylib`start + 1
frame #10: 0x00007fff76bdc085 libdyld.dylib`start + 1
```

Here *frame 7* shows that the file (`program.c`) and line (35) triggered the segfault

Breakpoints and GUI for debugging

- *Breakpoints* are points at which execution is stopped so we can investigate state
- Setting breakpoints in the command line debugger is possible but could be tedious
- The Visual Studio Code editor (and others, like Atom) provide a GUI for GDB / LLDB
  - Detailed instructions for the configuration:

https://code.visualstudio.com/docs/editor/debugging
https://code.visualstudio.com/docs/cpp/config-linux