Basic 9
Debugging

EECS 201 Winter 2021

Submission Instructions

This assignment will be submitted as a repository on the UMich GitLab server. Create a Project on it with the name/path `eecs201-basic9` and add `brng` as a Reporter.

Preface

In this homework you'll be provided yet another zipped archive containing some starter files.

`https://www.eecs.umich.edu/courses/eecs201/files/assignments/basic9.tar.gz`

Initialize a Git repository in the extracted `basic9` directory. Create a file called `report.txt` in this directory. Add all of the present files and commit them.

Create a **private** project named `eecs201-basic9` on the UMich GitLab (`gitlab.umich.edu`) and add the instructor `brng` as a **Reporter**. Set this UMich GitLab project as your remote: you'll be pushing to it in order to submit.

For this assignment you're going to need GDB and Valgrind. The package names for Ubuntu are `gdb` and `valgrind`. If you haven't already installed `g++`, the GNU C++ compiler, you can install the `g++` package. You may also use course server (peritia.eecs.umich.edu) to complete this assignment.

1 Debugging with GDB

1. `cd` into the `1` directory.
2. Use `make` to build the `seqmean` program. What this program does is take the average of a sequence of integers from 1 to n, where n is an argument.
3. Run `$ ./seqmean 3`. Note that a segfault occurs.
4. Run `$ gdb seqmean`. This will run GDB and it will load the `seqmean` executable.
5. Look at the line above the `(gdb)`: notice how it mentions that there are no debugging symbols. Debugging symbols are what GDB uses to be able to have a sense of what code "lines" and other things are.
6. Let's get some debugging symbols: exit GDB with `(gdb) quit` or `(gdb) q`.
7. Open the `Makefile` in an editor. Set the `CFLAGS` variable so that it contains `-g`. The `-g` flag will cause `gcc` to compile with debugging symbols.
8. Use `make` to clean and rebuild the program, and then open the program in GDB again.
9. To run the program, run `(gdb) run 3` or `(gdb) r 3`. This will run the loaded program with the given arguments: in this case `3`.
10. GDB will catch the program at the segfault, allowing us to investigate the state of the program. It's kind enough to tell us what line of what file it happened at.

11. With `(gdb) backtrace` or `(gdb) bt` (or `(gdb) where`) we can get a backtrace, which shows the function calls and their stack frames that lead up to the current place. Frame #0 represents the current function and frame #1 represents the caller of that function (and frame #2 would be its caller and so on).
12. `(gdb) frame` or `(gdb) f` will show the current frame and the line of code it's at.
13. Segfaults ("segmentation faults") come about due to accessing memory that doesn't "exist" (or unmapped for you OS folks ;)) or if there is a permission access violation (e.g. writing to a read-only spot). In this line, the only operation that performs a risky memory operation is the struct pointer member access operation (`->`).
14. That operation looks risky: let's print out the value of the pointer variable (which is the argument of the function): `(gdb) print d` or `(gdb) p d`. (Note: `(gdb) frame` also shows the arguments and their values that passed into the function).
15. Note that the value of `d` is `0x0`, a null pointer. Null pointer dereferences are some common ways a segfault can happen. But where did it come from? Look at your backtrace again.
16. Let's switch to the frame of the calling function: run `(gdb) frame 1` to investigate it.
17. `frame` conveniently prints out the line of code: we can see that at line 30 of `main.c`, a `NULL` (i.e. null pointer) is passed into the function instead of a pointer to a `data` structure.
18. Now that we've identified the bug, let's fix it. Exit GDB and open `main.c` for editing. Fix line 30 so that `data_calc_stats` takes in a pointer to the `d` variable, which is an instance of the `data` structure.
19. Rebuild the program and run `$ ./seqmean 3` again. There should not be a segfault this time.
20. Add and commit the `main.c` file.

Note how the result is incorrect: $\frac{1+2+3}{3}$ should equal 2, not 1. Let's figure out what's wrong and fix the bug.

1. Open the program in GDB again, and run `(gdb) run 3` or `(gdb) r 3`.
2. You'll see that GDB will run the program to completion. GDB will remember the arguments you pass along with the run: `(gdb) run` will cause it to run with `3` as the argument again.
3. It's not particularly interesting if it runs to completion. Let's set a breakpoint at `main()` to have it stop there: Run `(gdb) break main` or `(gdb) b main`. You can list out the breakpoints with `(gdb) info breakpoint` or `(gdb) info b`.
4. Run the program again. It should stop at `main()`.
5. Let step through the code: run `(gdb) next` or `(gdb) n` to step over a line.
6. Hit Enter/Return without typing out any command. Entering an empty command will run the previously entered command.
7. Place a breakpoint at line 30 with `(gdb) break 30` or `(gdb) b 30`. Implicitly, without specifying a file, the current file that the current line is in will be referred to when it comes to line numbers and function names. `(gdb) break main.c:30` and `(gdb) break main.c:main` are examples of explicitly referring to a file.
8. Continue execution with `(gdb) continue` or `(gdb) c`.
9. This is line that calls the function that calculates the mean. Step *into* the function with `(gdb) step` or `(gdb) s`.
10. Let's see if the sum is calculated properly: place a breakpoint at line 24. Continue so that this breakpoint gets triggered.
11. Print out the `sum` variable. It's 3! $1 + 2 + 3 \neq 3$!
12. That's curious: maybe the data was bad. Let's investigate the `buffer` member of `d`: try `(gdb) p d->buffer`.
13. Unsurprisingly, GDB prints out the actual pointer value that `buffer` holds. We can dereference it: `(gdb) p *d->buffer`.

14. Well, that gives us only the 0th element. We can get `(gdb) print` to give us elements after it as well: `(gdb) p *d->buffer@3` (or if we're being really fancy `(gdb) p *d->buffer@d->size`).
15. Now we can see the issue. The buffer should be a sequence of 3 integers starting at 1, not 0: the buffer's contents should be `{1, 2, 3}`.
16. I'll let you figure out the fix for this :) Investigate the C source and header files, and take advantage of whatever debugging knowledge you have.
17. When you're done with the fix, add and commit the files associated with the fix. Make sure to remove any debug prints that you make!

2 Finding memory issues with Valgrind

1. `cd` into the `2` directory.
2. Build and run the program. You should see `n = 5` and `0.2 0.4 0.6 0.4 0.2`. (Some people's systems may result in a segfault; we'll deal with it soon :)).
3. While it may be functionally correct, there may be bad memory usage lurking about which may cause issues if this were a more complicated program.
4. We can use Valgrind (more specifically it's Memcheck utility) to analyze the memory usage and access of your program. It's easy: just run `$ valgrind ./conv`.
5. You'll see a lot printed out (which by default goes to `stderr`). There is a lot of good info here: we get info about bad writes and reads and there's a memory leak summary as well.
6. First, let's deal with the memory leaks since those can be fairly easy to deal with. Take a note of the message that says "Rerun with --leak-check=full to see details of leaked memory", and do that.
7. The resulting heap summary will list out where memory is being lost and is nice enough to tell us the function call stack that associated with the allocations. We can see that the buffer allocated by Signal constructor is never getting deallocated.
8. Fix this memory leak by adding the necessary `delete[]` to `Signal`'s destructor.
9. Rebuild and rerun the program with Valgrind. The memory leak should be gone. Now onto the invalid write and read.
10. First, let's look at the invalid write which occurs in `main.cpp`. Note how in the `for` loop, `x[3]` is written to due to the `for` loop condition, writing past the end of the allocated buffer. Change the `for` loop's condition to fix this.
11. Rebuild and rerun with Valgrind. The invalid write should be gone now. The invalid read is still there, however. Let's look at the associated line in `conv.cpp`.
12. The line points to an accumulation: `sum += x[xI] * h[hI]`. The culprits of this invalid read would either be the `x[xI]` or `h[hI]`, as we could be indexing out of the bounds of the buffers for these `Signal`s.
13. If we investigate the code a bit, the `for` loop that immediately contains the `if` statement checks that `xI` is less than `xN`, the size of `x`'s buffer. Since in C/C++ indexing is 0-indexed (unless I'm mean and had overloaded the subscript operator (`[]`) to add 1 to the index) the indexing is correct for `x` since the last element is at index `xN - 1`.
14. If we look at the `if` that controls the accumulation, we can see that it's checking if `hI` is in the bounds of `h`. Notice anything wrong here?
15. Fix that issue in the `if` condition. When you rebuild and rerun, you should get a clean report from Valgrind.
16. Now that it's leak and memory error free, add and commit the files that you made fixes for.

3 Conclusion

1. Add and commit any changes you intend to submit.
2. Fill out the `report.txt` file in the following steps:
3. On the first line provide an integer time in minutes of how long it took for you to complete this assignment. It should just be an integer: no letters or words.
4. On the second line and beyond, write down what you learned while doing this assignment. If you already knew how to do all of this, put down "N/A".
5. Commit your `report.txt` file and push your commits to your remote.