Advanced 9
Debugging

EECS 201 Fall 2020

Submission Instructions

This assignment is an "online assignment" on Gradescope, where you will attach your files and answer some questions.

Preface

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

```
$ wget https://www.eecs.umich.edu/courses/eecs201/files/assignments/adv9.tar.gz
```

1 Attaching to a running process (5)

Occasionally you may want to debug an already running process with GDB. Perhaps it's some sort of daemon process that you want to catch in the act or maybe it's some interesting setup that you have concocted *cough cough*. Regardless, GDB has the ability to attach to already running processes when provided with the process ID ("PID") of a process you want to debug.

In `adv9/gdb-attach` there is a program called `revd` that will reverse strings provided to it and log them. It is intended to run in the background, using a special type of file called a *named pipe* or *FIFO* (like a pipe used to string commands together, but accessible as a file) as input. It uses the FIFO as a form of inter-process communication (IPC): a process that wants to communicate will write to the FIFO file and `revd` will read from it.

The Makefile has targets for building and running the application. `test-producer.sh` is a sample script that communicates with `revd`, forever providing it with a random word every second (until you `^C`). Try running the `run` target and then run `test-producer.sh`. In another terminal run `$ tail -f revd.log` to continuously print out updates to the log file. You should see that the `revd` process is chugging along (for extra fun, run some more instances of `test-producer.sh` or write another program/script that produces more input for `revd`).

Now onto the debugging part: the goal of this exercise is to *attach* GDB to this running process. There's multiple ways of going about this: I'll leave it up to you to take a gander at the GDB manual or manpage or other random resources on the internet.

Notes:

- **If you are using WSL, the Windows filesystem (i.e. stuff under /mnt/c) does not support FIFOs. This can only be done on Unix filesystem side (i.e. everything not /mnt/c).**
- Newer versions of the Linux kernel now by default have security measures in place that prevent the tracing of processes, which GDB does to attach to a process. If you are unable to attach to the process and there is some warning/message about `ptrace` not being permitted,
  run `$ sudo sh -c "echo 0 > /proc/sys/kernel/yama/ptrace_scope"`. This will invoke the `sh` shell under `root` and write a 0 into the `/proc/sys/kernel/yama/ptrace_scope` file which serves as the way to configure the security level for tracing.
- `make run` will run `revd`.
- `make kill` will kill `revd` (since it has been backgrounded it's a tad more involved to stop it).
- `pgrep`, `pidof`, and `ps a` can help you find the PID of `revd`.

- ☐ What command-line commands did you run in the process of attaching GDB to the running `revd` process? Be specific!
- ☐ If you look at the `revd.log` file you'll notice that it isn't reversing the string properly. Fix this issue and submit the fixed version. (By the way, string reversal is a pretty common algorithmic interview question)

2 How do you debug? (5)

Write a paragraph or two about what your process is when you go about debugging your programs. What do you look for? Do you start off with print statements before heading to your debugger? Let me know about your process for it!

3 Pretty printing (5)

GDB is actually extensible via Python scripts! One neat feature is that Python can be used to implement *pretty-printers*, which are a way to print out structs and classes in a more pretty way. Without a pretty-printer registered for a struct/class, you'll get a printout of each individual field, no matter how "user readable" they are. C++ STL is notoriously ugly when not pretty printed; the GNU C++ library implementation bundles pretty-printers for GDB to take advantage of. For instance, if you printed a `std::string` in GDB without a pretty-printer, you'd see a mess of member variables and internal structures that adds a lot of noise (which, to be fair, are useful if you're debugging the `std::string` class itself.) With the pretty-printer, you'd get the string that you want to see.

For this exercise, I want you to write a pretty-printer for a C/C++ struct/class. You probably have something already, like a Euchre project, that has some classes that are noisy when printed. If you want, you can also use the provided `Matrix` class in `adv9/gdb-pretty` to try to pretty print the actual matrix contents.

Helpful hints:

- I've provided a boiler-plate script in the `adv9/gdb-pretty` directory.
- Once in GDB, you can `source` the Python script to get it to run the pretty-printer registration script. (This method of registering a pretty-printer is a bit heavy handed due to its simplicity, but the way to automatically load pretty-printers attached to a library of code a la the GNU C++ library is a fairly involved process)

Some relevant documentation:

- Pretty-printer introduction
- `std::string` without a pretty-printer
- Short guide on writing a pretty-printer in Python.
  This one does go into the more "proper" way of registering the pretty-printer, but you can do it globally a la the provided boiler-plate.

- ☐ Submit the pretty printer script(s) and whatever code you want to pretty print.
- ☐ Explain how to load your pretty-printer script in GDB.
- ☐ Submit a screenshot of GDB's terminal printout of the struct/class.