CSCI 3308 Software Methods & Tools [Fall 2015]
Instructor: Boese
Lab 9

Dynamic Analysis

Objectives

- Practice using a set of Free dynamic analysis tools: gdb, valgrind, and kcachegrind
- Please pair-program today!

Dynamic Analysis

First, download or clone the source code from https://github.com/asayler/CU-CSCI3308-DynamicPractice
The code consists of a small program that calculates PI using statistics and geometry. The usage of the program is `./pi <iterations>` where iterations is the number of calculations to run. Higher iterations will result in more accurate calculations of PI, but will take longer to run. If no iterations argument is supplied, the program defaults to 1,000,000 iterations.

Then, make sure you have the following software installed on your system (the CU CS Virtual Machine has this all by default):

- gcc
- GNU Make
- gdb
- valgrind
- kcachegrind

Now we'll walk through some analysis and modification of the source code using the tools above. Where questions appear below, please type your answer into a plaintext file called answers.txt. You will submit these answers, as well as the modified code, at the end of this assignment.

Part 1 - GDB:

1. Use make to compile the code.
2. Run ./pi. **What happens?**
3. Run the code again via GDB.
4. Using GDB, determine where the issue is occurring. **On which line of code does the program crash?**
5. Exit GDB.
6. Open the code in an editor. Find the problem line. **What is the problem and how do we fix it?** *Hint: look at the similar working code in zeroDist().*
7. Fix the code, save, and the re-make the code.
8. Run the code again via GDB. Confirm the crash is fixed. If not, iterate until the code does not crash.
9. Set a breakpoint at the zeroDist() function. **What GDB command did you use?**
10. Run the code again.

11. When you reach the breakpoint, print the current x and y values of the 'other_pt' argument. **What GDB commands did you use?**
12. Delete the breakpoint and continue. **What GDB commands did you use?**
13. When the program exists successfully, close GDB.

Part 2 - Valgrind:

1. Now run the code via valgrind
2. **Is the code leaking memory? How much?**
3. Use valgrind to identify where in the code memory is being allocated but not freed. **What are the problematic line numbers?**
4. Open the code in an editor. Find the problem lines. **What is the problem and how do we fix it?**
5. Fix the code, save, and the re-make the code.
6. Run the code again via valgrind. Confirm the memory leak is fixed. If not, iterate until the code does not leak memory.

Part 3 - Profiling:

1. Use '/usr/bin/time' to calculate how long it takes ./pi to run. Play with the number of iteration until you have it taking ~1s of real time. **Record the number of iterations you are using and how long the program takes to run.**
2. Now generate a callgrind profile by running 'valgrind --tool=callgrind ./pi <your iterations>'
3. Open the resulting profile output using kcachegrind: e.g. 'kcachegrind callgrind.out.?????'
4. Explore the information kcachegrind provides. **Starting with main(), what are the top 5 places the program spends the largest percentage if its time? What percentage of time does it spend in each place?**
5. Close kcachegrind.
6. **Using the information from 4, can you make the code faster? How?**
7. Modify and rebuild the code with some of your enhancements.
8. Run the code again via '/usr/bin/time'. **How does the new runtime compare to the runtime from 1?**

To get credit for this lab exercise, show the TA and sign the lab's completion log.