Operating Systems: Debugging

1 Introduction

In this exercise, we will use **GDB** to debug a tree program. We will also see how the performance debugging tool **gprof** can be used to profile programs.

2 Background

First, we will discuss some background information about GDB and gprof.

2.1 GDB

GDB is a debugging tool that allows for inspecting a program while it is running. If you run a program inside GDB, these are some things you can do:

- Stop the program based on specified conditions. For example, you can put a breakpoint at a certain line number within a certain file. There are other ways to stop the program as well, such as using conditional breakpoints or watchpoints.
- Examine data at the point when the program stopped. You can print the values of variables using the `p` command, you can also use the `x` command to examine memory. There are also helpful commands such as `info locals` or `info args` that will display the local variables or arguments passed into the function, respectively.
- Modify variables. The `print` command can also be used to modify values of variables as the program is running. This can be helpful if you want to test a bug fix quickly from within GDB without recompiling your program.
- Finding more information about segmentation faults. If your program seg faults, you should run it in GDB and look at the backtrace to see where the crash is occurring. From there, you can backtrack to try to determine the root cause of the problem.

For more information about GDB, please see this documentation.

2.2 gprof

gprof is a performance analysis tool used to see how much time a program spends in each function. gprof can produce output in multiple forms, such as:

- Flat Profile: Displays the execution time of each function, which helps to identify performance-heavy functions.
- Call Graph: Shows which functions were called within a particular function, and also which functions called this particular function.

To use gprof, first you must compile your program with the -pg flag (e.g., `gcc -pg ...`). Then, run the program normally, which will generate profiling data in a file called `gmon.out`. This can then be analyzed with gprof. In this exercise, these commands are set up in the Makefile and discussed more below. For more information about gprof, please see this documenation.

3 Programs/Files

3.1 `tree_bad.c`

This program creates a binary tree. Each node contains a value and pointers to its left and right node.

The following helper functions are included:
`create_node`: mallocs space for a new node, then sets the value field to the passed in value. The left and right fields are initialized to NULL, and a pointer to the new node is returned.
`insert`: Recursively determines where the new node will be placed. Once insert is called with NULL as the root, a new node is created by calling `create_node`, then the pointer to this new node will be returned to a previous recursive call to link the new node to the tree (because either `root→left` or `root→right` will be set to the pointer returned from `create_node`).
`in_order`: Prints the tree using in-order format, which is left-root-right order (note that this order also applies to all sub-trees within the tree).
`free_tree`: Frees the memory that was malloced for each node in the tree.

In the main function, we create an array of values, determine how many entries are in the array, and set our root local variable to NULL. Then, we insert nodes with each of the values from the array into the tree. After inserting the nodes, we don't need `n` (the variable to track the number of entries in the array) anymore, so we (attempt to) set it to 0. We then attempt to print the tree's contents using in-order traversal, but nothing is printed.

This unexpected behavior is because of how we try to set `n` to 0 in lines 69 and 70. We cast the address of `n` to an unsigned long long pointer. Unisigned long longs are 8 bytes (and note that n is an integer, which are 4 bytes). We then use array notation to access 8 bytes after `n`'s address, which happens to

be the memory location of the `root` local variable. Thus, line 70 sets `root` to 0 (i.e., NULL), so when `in_order` is called with root in line 74, nothing is printed.

GDB can be helpful to detect this issue. We can use breakpoints at lines 69 or 70, and print the values of `p` and `root` once the program stops there. We can also use the `x` command to display memory before and after line 70 (something like `x/10gx p` can work).

3.1.1 Conclusion

This exercise shows that we must be careful with pointer casting and dereferencing pointers to modify data. When bugs like this show up, GDB can be useful to figure out what is going wrong.

3.2 `tree.c`

The same helper functions from tree_bad.c are used. We do not overwrite the `root` variable with 0 here, and the in-order traversal prints correctly.

3.2.1 Conclusion

The fixed program works as expected.

3.3 `tree_pp.py`

This is a python extension that can be loaded into GDB to pretty print the binary tree. There are multiple ways to do this; one of which is to use the command `source tree-pp.py` inside GDB. After this, you should be able to use a new command in GDB: `print-tree root` (where `root` is an argument to the `print-tree` command), which will pretty print the tree starting at `root`.

The functions within this python program are as follows:
`__init__`: Initialization for the custom print-tree GDB command.
`invoke`: Parses the passed-in argument as a `tree_t` pointer, then calls `pretty_print_tree` to display the nodes of the tree.
`pretty_print_tree`: Recursively prints the nodes in the right subtree, then the current node, then the nodes in the left subtree. The result in the terminal should show the full tree, starting from the root (note that the root will be on the left side, with its children to the right of that, etc.)

3.3.1 Conclusion

GDB extensions can be used to create custom commands to make debugging easier.

3.4 `prof.c`

In the main function of this program, we start by printing a message, indicating that the program is starting. We then call the function `foo`, which contains a for loop which increments the `j` local variable in each iteration. We then call the function `bar`, which also contains a for loop with many iterations in which the `k` local variable is incremented.

3.4.1 Conclusion

This is a simple program that we will profile using gprof.

3.5 gmon.out

This is the output file generated by gprof after running the `prof` program (i.e., `./prof`). We will use gprof to analyze this file to get readable performance data about the program.

3.5.1 Conclusion

This file is output from running a program that was compiled with the `-pg` flag, and input to gprof.

3.6 analysis.txt

This file contains the output from gprof analyzing the `gmon.out` data (in conjunction with the `prof` binary file). Specifically, we can see the Flat Profile, which indicates that about half of the time is spent in the `bar` function, and the other half is spent in the `foo` function. We can also see the Call Graph, which indicates that the main functions calls both `bar` and `foo`.

3.6.1 Conclusion

This file contains useful profiling information about `prof.c`, produced from gprof.

3.7 Makefile

The Makefile contains two targets. First, if you run `make prof` (or just `make`, `prof.c` will be compiled using the `-pg` flag (which is needed to profile the program using gprof). You would then run the prof program (using `./prof`) which will create the `gmon.out` file. Then, you can use `make analysis` to run gprof to analyze the `gmon.out` file, and the readable gprof performance data is redirected to the `analysis.txt` file.

3.7.1 Conclusion

The Makefile contains relevant gprof commands to profile the `prof` program.