Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Introduction to Debugging Tools

In this lab, we will learn to use simple debugging tools like `gdb` and `valgrind`. You are provided with a few buggy programs with this lab: `pointers.cpp`, `fibonacci.cpp`, and `memory_bugs.c`. You are expected to debug them and demonstrate your understanding of debugging tools during your evaluation.

Part A: Debugging with GDB

A debugger is a program that runs other programs, with the user being allowed to exercise control over how these programs run. Users can insert breaks in the program, examine variables and so on, during program execution. GNU Debugger, which is also called gdb, is the most popular debugger for UNIX systems to debug C/C++ programs. GDB can only be used to find runtime or logical errors. Please note that compile time errors are detected by the compiler and not a debugger. In this part of the assignment you will use GDB to debug simple logical errors and a segmentation fault runtime error.

How to use GDB

The first step to using GDB is to install it on your computers. Steps for installation can be found online, and an example link is here:

```
https://www.tutorialspoint.com/gnu_debugger/installing_gdb.htm
```

Before using GDB you will have to compile your program using gcc or g++ to generate an executable. The -g flag is particularly important as it generates debugging information that gdb can use.

```
g++ filename -g -o exectable_name
```

GDB is invoked with the shell command `gdb`. Once started, it reads commands from the terminal until you tell it to exit with the GDB command `quit`. You can get online help from gdb itself by using the command `help`. You can run `gdb` with no arguments or options; but the most common way to start GDB is with commandline arguments, specifying an executable program as shown below:

```
gdb executable_name
```

The command above will attach gdb to your executable and open the gdb shell in which you can write commands to debug your program.

Here are some of the most frequently used GDB commands:

`break` [file:]function
Set a breakpoint at function (in file).

`break` filename:linenumber
Set a breakpoint at a line number in a file.

`run` [arglist]
Start your program (with arglist, if specified).

`bt`
Backtrace: display the program stack.

`print` expr
Display the value of an expression.

`c`
Continue running your program (after stopping, e.g. at a breakpoint).

`next`
Execute next program line (after stopping), with stepping over any function calls in the line.

`step`
Execute next program line (after stopping), with stepping into any function calls in the line.

`list` [file:]function
type the text of the program in the vicinity of where it is presently stopped.

`help` [name]
Show information about the GDB command name, or general information about using GDB.

`quit`
Exit from GDB.

***Note**: The above information is sufficient to attempt the exercises in this lab. For more information, please refer to the man page of GDB.*

Exercises

1. Debug the pointers.cpp program given to you using GDB. The program contains some pointer related operations. A bug has been deliberately introduced in the code so that it generates a segmentation fault. Your task is to use GDB to find the line number of the wrong statement. Please make use of the GDB commands provided above in order to find the bug.

2. Debug the fibonacci.cpp program given to you using GDB. This program is supposed to print fibonacci numbers until a certain value of n. However, there is a logical error introduced in the code which causes it to print wrong output. Your task is to use GDB to debug the program. You must insert suitable breakpoints, pause program execution, print intermediate values of variables from GDB, and monitor the execution step by step in order to find the logical error. Even if you can identify the error without stepping through the code, you must be able to demonstrate the process of debugging using GDB during your evaluation.

Part B: Memory Check with Valgrind

Valgrind is a memory mismanagement detector. It helps you identify memory leaks, deallocation errors, use of uninitialized memory, and various other such memory usage errors. In fact, Valgrind is a wrapper around a collection of tools that do many other things, e.g., cache profiling. However, here we focus on the default tool, memcheck.

How to use Valgrind?

The first step to using Valgrind is to install it using a command like the following on Ubuntu:

sudo apt install valgrind

Compile your program using gcc or g++ and generate an executable using the -g flag.

g++ filename -g −−no-warnings -o executable_name

You can now run Valgrind against your executable to figure out memory related bugs in your program using the following command (see the man page to learn more about the various options.)

valgrind −−tool=memcheck −−leak-check=yes −−show-reachable=yes −−num-callers=20 executable-name

Exercise

A program **memory_bugs.c** is provided to you. This program is riddled with memory bugs. You may be able to find some bugs just by looking at the code also. Valgrind can help you find these bugs automatically. Your job is to compile the program using the command provided above and use Valgrind to find the possible issues present in the program. You should first understand the different issues Valgrind can detect, and then use the command given above to find the issues present in the program. You might be asked to provide possible reasons and fixes for those issues during your evaluation.

Submission instructions

- You must submit a text/pdf file briefly describing your debugging procedure for the questions above.
- Place this file and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.