CMPT 214: Programming Principles and Practice
Term 1 2016-17
Lab 10 – Environment Variables and Interactive Debugging

Complete each of the tasks below. For all steps involving the use of UNIX/LINUX commands, place the command you used along with the resulting output (i.e. copy-and-paste from your terminal window) into a file called `lab10.txt`. In addition, copy-and-paste a log of the specified interactions with `gdb` into `lab10.txt`. However, do not include extraneous or superfluous commands or output; only include content relevant and essential to the specified task. Then, with a text editor, add to `lab10.txt` identifying information to clearly distinguish which commands/output correspond to each task/question. When done, hand in `lab10.txt`, as well as your modified `power.cc` and `makefile`, through the moodle page for the lab. This lab is out of a total of 22 marks; the number of marks allocated to each question is indicated below. Marks may be docked for extraneous, irrelevant, or superfluous content or for not following directions. Use tuxworld for completing the laboratory exercise. Your submission is due at 11:55 p.m. on Thursday, November 24.

Note that the `makefile` provided for this lab exemplifies how `make` can be used in ways beyond those shown in class.

This lab exercise description is three pages in length.

1. (2 marks) In a previous lab, you set the environment variable `GREP_OPTIONS` to "`-i`" in order to make `grep(1)` case-insensitive without having to manually supply the "`-i`" flag. For this question, first use the command "`echo $GREP_OPTIONS`" to show that there is no setting for the environment variable `GREP_OPTIONS`. Then use the `env` command to set `GREP_OPTIONS` to "`-i`" for just a single invocation of "`grep a <<< A`". The resultant behaviour should be exactly the same as the command "`grep -i a <<< A`". After doing this, execute the commands "`grep a <<< A`" and "`echo $GREP_OPTIONS`" to show that `GREP_OPTIONS` remains unchanged for your shell after the `env` command.

   The `env` command is described on page 471 of the Sobell text. Alternatively, for information about it, consult the `man` page.

2. (2 marks) In question 3a, you will need to produce a core dump file. Whether or not a core dump file will be created for you depends on your resource limit settings. First use a "`ulimit -a`" or `prlimit` command to see the resource limits for your shell and any children it creates. If the limit for core file size is not `unlimited`, then use the "`ulimit -c unlimited`" command to set it to `unlimited`. Confirm the setting with a final "`ulimit -a`" or `prlimit` command.

   For more information about resource allocation limits, see the `man` pages for `prlimit(1)` or `getrlimit(2)`. Usage of `ulimit` is described in the relevant section of the `man` page for `builtins(1)` (on tuxworld).

3. In this question, you will be exploring the use of `gdb` for debugging programs by, for example, utilizing breakpoints and printing out the values of variables. You will be working with the supplementary C++ source file called `power.cc`, a `makefile` used to build this program, and an example input file called `infile.txt`. `power.cc` reads in a list of space-separated pairs of numbers, one pair per line. If you

look at the source code for `power.cc`, you will notice that the program reads either from a file (if a file is specified as a command-line argument) or from the standard input (if not). Therefore, `power` can be run either using "`./power infile.txt`" or "`./power < infile.txt`". For each line of input, the program outputs the value of the first number to the power of the second number, and—on the next line—the value of the second number to the power of the first number. The program assumes that the numbers in the input file are always non-negative integers. Files `power.cc`, `infile.txt`, and `makefile` are in the `tar(1)` saveset `Lab10Files.tar`.

Download the supplementary file for the lab, `Lab10Files.tar`, unpack the `tar` saveset, and then perform the following steps.

(a) (2 marks) Run `make` to build the `power` executable. Then run `power` using `infile.txt` as the command-line argument. You should get a segmentation fault, a problem that can be very difficult to debug. You are going to use `gdb` to make the bug easier to find.

Note that because of your actions in question 2, you should have a core file created in your current working directory. Using an `ls` or `file` command, show that the core file was created.

(b) (2 marks) Modify the `makefile` so that debugging information is added to the executable when the program is compiled and linked. You don't need to hand in a log of your editing session—just hand in your modified makefile. Then rebuild the program using your modified makefile and giving the `-W` option to `make`. Note that you need to supply an argument to the `-W` option.

The changes to `makefile` in this step should be minimal—no more than are necessary to fulfil the specification above.

(c) (2 marks) Start `gdb`, specifying the program you want to debug (i.e. `power`). Then begin execution of `power` by issuing the "`run`" command (to `gdb`) with `infile.txt` as the argument. When the program stops after the segmentation fault, execute the `gdb` command "`backtrace full`". Examine the output carefully; there is a wealth of information made available.

(d) (3 marks) Continuing from step 3c, make a note of the line number (in the source code) at which the segmentation fault occurred. Using the `gdb` "`list`" command, output the source code surrounding this line of the program. Then using the `gdb` "`frame`" command, set the frame to frame 0. Now use `gdb`'s "`print`" command to output the values of the memory locations pointed to by `*a` and `*b`. Perform any other `gdb` commands that might provide you with useful information for identifying the source of the "segfault" error.

When you feel you have gathered sufficient information, issue the "`quit`" command to `gdb`. You will probably get a message saying "`A debugging session is active.`" and then asking you if you want to "`Quit anyway`". Answer to the affirmative.

(e) (1 mark) Based on the output from parts 3c and 3d, modify the `swap` function in `power.cc` to fix the error (you do not need to hand in the log of your editing session). Then rebuild the `power` program using `make`. Finally, run the `power` program again outside `gdb`. Your program should no longer "segfault", but the output will not be correct.

The changes to `power.cc` in this step should be minimal — no more than are necessary to fulfil the specification above. Think about whether variable `tmp` should be of type `int*` or just `int`. If you add code to allocate dynamic memory, you are making excessive modifications.

(f) (3 marks) Start `gdb`, again specifying that you want to debug `power`. Set a breakpoint at the `calc_pow` function. Then run the `power` program (within `gdb`) with `infile.txt` as the argument. When execution gets to the breakpoint, instruct `gdb` to display the contents of the variables `num` and `pow` each time the program stops. Repeatedly continue execution to the next breakpoint, noting the values of `num` and `pow`. An occasional `backtrace` will also be useful. Once you have identified the bug, exit `gdb`.

(g) (1 mark) Modify `power.cc` so that `calc_pow` works correctly (note that one other "bug" is that $0^0 = 1$, but we will ignore this). You do not need to hand in the log of your editing session. Run `power` outside of `gdb` to verify that the program now works correctly.

The changes to `power.cc` in this step should be minimal — no more than are necessary to fulfil the specification above.

(h) (2 marks) Now that you've used `gdb` to debug this program, you will explore `gdb` a bit more. Start `gdb` with argument `power`, then list lines 18 through 24 of the (`power`) program. Set a breakpoint on one of the two lines that calculate the value of variable `result` (it does not matter which one). Then run your program within `gdb`, but this time with standard input redirected to come from `infile.txt` (i.e. use input redirection as you would within a UNIX/LINUX shell).

(i) (2 marks) `Gdb` will know about the various C/C++ structured datatypes being used in your program, even the ones defined in system `.h` files. One such datatype is `FILE` as defined in `stdio.h`. Note that variable `in_stream` in `power.cc` is of type `FILE *`.

Continuing from step 3h, when the `power` program stops, output the value of the `in_stream` variable and the size (in bytes) of the `in_stream` variable. Then, with a single command, output the data type of the `struct` pointed to by (the value in) the `in_stream` variable. The output from the single command should include the data types of all the elements within the `struct`. If there is more than one screen full of output, display it all.

(j) (0 marks) Continue using `gdb` to examine the execution of `power`. Explore the functionality of `gdb`. Finally `quit gdb`.

Don't forget to submit your modified `power.cc` and `makefile`. You can also remove any core files and reset your resource limit, if you wish.