GDB and Valgrind Tutorial

Carey Pridgeon

October 6, 2016

1 Valgrind

- Gprof is the GNU foundation profiling tool for C and C++ programs.
- However this isn't the easiest of programs to use directly.
- Instead we'll use **valgrind**.
- **valgrind** is part of the **KDE** development environment. It uses **gprof** as a base and extends it to anable the output to be graphically dusplayed easily.
- To do this we need to log in to **nostromo**.
- Profiling is the process of examining the function and method calls of a program to determine which how much time is spent in each, how many times they are called, and the call tree (what calls what and for how long).
- We profile so we know where to best optimise our programs. For example, if the program spends just **2%** of its time in one function, there is little point optimising it.

2 Valgrind Task

- To fetch it from its gitlab repository type the following
- **git clone** `https://gitlab.com/carey.pridgeon/gdb_worksheet_material.git`
- Now cd into the directory directory and open the programs Makefile.

2.1 Edit the Makefile

- We need to add a compiler directive to add profiling symbols to the compiled code.
- To do this find the following lines:

```
$(EXE): $(OBJS)
$(CC) $(OBJS) $(LIB_DIRS) $(LLIBS)$(LDFLAGS) -o $(EXE)
```

- And update it to include **-pg**

```
$(EXE): $(OBJS)
$(CC) $(OBJS) $(LIB_DIRS) $(LLIBS)$(LDFLAGS) -pg -o $(EXE)
```

- and do the same to the following line

```
CFLAGS  = -Wall -O3
```

becomes

```
CFLAGS  = -Wall -O3 -pg
```

- Now we compile the code by typing **make clean && make**

2.2 Running Valgrind

- **valgrind –tool=callgrind ./nmod** (that's two minus signs before tool)
- Run the command
- Once nmod completes, it will have generated a file called
- **callgrind.out.[number]** in the nmod folder.
- This file represents a call graph for nmod, and we need to pass the program and this file to valgrind to generate the kcachegrind readable profiler output.
- Once this command finishes, type **kcachegrind**
- This will launch a graphical program that will parse the output file.

2.3 Reading the output

- You will see a great deal of information on the performance of the program, including call graphs and percentage of runtime/call count occcupied by given functions.
- Find the program function called most often (in which the program spends most of its processing time.
- What function calls it?
- Does it call any other functions from the program?
- Which function is called the smallest number of times (but more than once)?
- Find a function that you think lies roughly half way, in terms of times called and runtime, between these two extremes.

2.4 Submission for this task

- Write the name of these functions, and their call counts/total runtime in your report.
- Include an exported image of the call graph for each function in your submission. Right Clicking on the call graph picture brings up this option.

3 GDB

- GDB is the GNU foundations debugging tool.
- Setting up the program to work with GDB is the same as for Gprof, except instead of **-pg**, we just use **-g**.
- Once the Makefile has been altered, re-compile the program **make clean && make**
- And run it with GDB by typing: **gdb ./nmod**
- This launches the gdb console ready for us to start debuggng the program.

3.1 GDB Commands

3.1.1 tui

to launch gdb with tui (text User Interface) type **gdb -tui**

3.1.2 run

This launches the program. If we run it without setting any breakpoints it will run to completion. If there is a program killing bug it will crash in gdb and produce a useful trace we can use to begin the analysis.

3.1.3 break

- In order to stop the program at a specific point so we can examine it more closely we need to set breakpoints.
- A breakpoint is a point in the running program where execution will halt, allowing us to examine program state and advance through the program step by step to isolate an error.
- To set a breakpoint we must specify either a function/method name, ora sourcefile name and line number.
- So if we want to put a breakpoint for a function **_funcname** we do as follows: **break _funcname**

- So now when we type **run** the program will halt at the break point, allowing us to control the program from there on.
- Alternatively you can specify a location in the source file, but we won't use that:

3.1.4 step

- Using this command we can *step* through the program one line at a time to see if a program works or if and where it crashes. To step in more than once, set a value: e,g step **5**

3.1.5 print

- By typing **print [variablename]** we can inspect the contents of a variable.
- We can also print the contents of a static array, eg: **print array[0]**

3.1.6 until

- This will, if you are in a loop, continue running the program until the loop exits.

3.1.7 list

- This will list a specified number of lines of code forward from the current position in the code:
  - **list n**
  - Where n is the number of lines to display.

3.1.8 quit

- Typing the **quit** command will exit the program.

3.2 task

- Set a breakpoint at _find_gravitational_interaction_between_pair_of_particles
- Step in once, then examine **(local_particles + 4) (the Moon)**.
- To examine a single componant of the particle using print, derefence that element. Look in the file nmod_structures.h for the particle type to find some more elements.
- Display five of them individually, then all of them at once by dereferencing the entire particle.

- Instruct gdb to continue till the for loop is done, then do so again. list the source code, pressing entil until you are out of lines, then type until again to finish the loop operation and move to the calling function, then list again.

3.3 Submission for this task

screenshot your progress as you examine the variable values, and put this in your report.