Valgrind with GDB | Coursera

coursera.org/learn/interacting-system-managing-memory/supplement/lbb6L/valgrind-with-gdb

Valgrind with GDB

Valgrind with GDB

We may have a problem that appears on a particular line of code, but only under specific circumstances that do not manifest the first several times that line of code is encountered. When such a situation occurs, we may find it more difficult to debug the code, as simply placing a breakpoint on the offending line may not give us enough information—we might not know when we reach that breakpoint under the right conditions. What we would like is the ability to run GDB and Valgrind together, and have Valgrind tell GDB when it encounters an error, giving control to GDB.

Fortunately, we can do exactly that. If we run Valgrind with the options **--vgdb=full --vgdb-error=0**, then Valgrind will stop on the first error it encounters and give control to GDB. Some coordination is required to get GDB connected to Valgrind (they run as separate processes); however, when run with those options, Valgrind will give us the information we need to pass to GDB to make this happen:

```
==24099== (action at startup) vgdb me ...
==24099==
==24099== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==24099==   /path/to/gdb ./a.out
==24099== and then give GDB the following command
==24099==   target remote | /usr/lib/valgrind/../../bin/vgdb --pid=24099
==24099== --pid is optional if only one valgrind process is running
```

At this point, Valgrind has started the program, but not yet entered main—it is waiting for you to start GDB and connect it to Valgrind. We can do so by running GDB (in a separate terminal, or Emacs buffer) and then copying and pasting the target command that Valgrind gave us into GDB's command prompt:

```
(gdb) target remote | /usr/lib/valgrind/../../bin/vgdb --pid=24099
Remote debugging using | /usr/lib/valgrind/../../bin/vgdb --pid=24099
relaying data between gdb and process 24099
Reading symbols from /lib64/ld-linux-x86-
64.so.2...Reading symbols from ...
done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x00000000040012d0 in _start () from /lib64/ld-linux-x86-64.so.2
(gdb)
```

At this point, we can give GDB the commands we want. Most often, we will want to give GDB the **continue** command, which will let it run until Valgrind encounters an error. At this point, Valgrind will interrupt the program and return control to GDB. You can now give GDB whatever commands you want to, so that you can investigate the state of your program.

The combination of Valgrind and GDB is quite powerful and gives you the ability to run some new commands, via the **monitor** command. For example, if we are trying to debug pointer-related errors and want to know what variables still point at a particular memory location, we can do so using the**monitor who_points_at** command:

```
gdb) monitor who_points_at 0x51fc040
==24303== Searching for pointers to 0x51fc040
==24303== *0xfff000450 points at 0x51fc040
==24303==  Location 0xfff000450 is 0 bytes inside local var "p"
==24303==  declared at example.c:6, in frame #0 of thread 1
```

There are many other **monitor** commands available for Memcheck. See http://valgrind.org/docs/manual/mc-manual.html#mc-manual.monitor-commands for more information about available **monitor** commands and their arguments.

✓

Completed