CS4414 Recitation 8

Debugging with gdb

10/15/2021

Sagar Jha

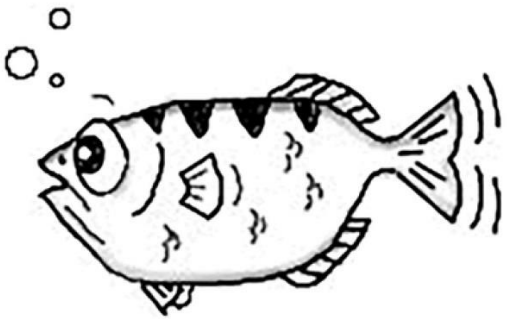

Every programmer’s fantasy
If only my code would work on the first try…

In reality

- You don’t know where to start
- Your code does not compile
- Your code does not run correctly
  - Keeps running forever
  - Segfaults
  - Does not produce the right output

gdb can help debug runtime issues

- What is a debugger? A program that helps debug the behavior of other programs.
- Allows you to pause program execution at any point and examine program state
- Produces a trace in case of segfault
- gdb or the GNU debugger itself is written in C
- Works on Unix and Windows alike



GDB
The GNU Project Debugger

Introduction to the gdb command

- Run **gdb <executable>** from the directory of the executable. This will open the gdb shell. Run **run <args>** to run it.
- To be able to debug properly, you need to supply the "**-g**" option with **g++**: E.g.,

  g++ -g hello_world.cpp –o hello_world
- "**-g**" produces debugging information with the binary. For example, it maps the lines in the machine code binary to lines in the source code

Introduction to the gdb command

- Do not optimize code that is meant for debugging, that is, don’t use -O1, -O2, or –O3 flags
- Optimization strips quite a lot of the program skeleton, even the variable names
- Thought question: What if the bug only shows up with optimized code?

gdb/optimization + cmake

- In my **CMakeLists.txt** file in the project root, I define

```
set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -O0 -Wall -ggdb -gdwarf-3")
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -O3 -Wall")
```

- Aside: The different debug flags for g++ have been listed with some explanation at https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html.

gdb/optimization + cmake

- When I run the **cmake** command to generate the Makefiles, I can specify the variable *CMAKE_BUILD_TYPE* to be either *Release* or *Debug* (*Release* is the default)
- Replace “Release” by “Debug” throughout **build.sh** and then work with the binary in *Debug/bin* for debugging

```
#!/bin/bash

mkdir -p Release
cd Release
cmake -DCMAKE_BUILD_TYPE=Release ..
make

cd ..
cp data/Traffic_Signals_SF.csv Release/bin
mkdir -p Release/bin/files
```

How to debug with gdb?

Let’s start with an example

- Suppose your simulator code for HW 2 is running forever
- What’s the most likely cause?

Let’s start with an example

- Suppose your simulator code for HW 2 is running forever
- What’s the most likely cause? The simulation itself is running forever.
- First step
  - Verify that this is the case

First step: Verifying the problem

- Test with the smallest possible time (-t=1)
- After you pop an event, print its time
- If you see a diverging sequence of non-decreasing numbers, for example, values much greater than 10K, you know this is *a* problem

First step: Verifying the problem

- Test with the smallest possible time (-t=1)
- After you pop an event, print its time
- If you see a diverging sequence of non-decreasing numbers, for example, values much greater than 10K, you know this is *a* problem
- An alternative method - add the following to the code:

```
while(/*termination condition*/) {
    const event current = events.top();
    events.pop();
    if(current.get_time() > 10000) {
      std::cout << "Problem!" << std::endl;
    }...
```

We confirmed the problem. What's next?

- Pause the program execution when it's in a bad state
- Examine the program state
- Take it from there
- We already know the bad state: a value of time > 10K

How to pause execution in a bad state?

- Code breakpoints
- Suppose the body of the if-condition that checks if time > 0 starts at line 108 (that prints "*Problem!*").
- To add a breakpoint at this line, run (inside the gdb shell) **breakpoint simulator.cpp:108**. Then, run the executable with **run –t=1**
- The executable will keep running until time is <= 10000, then enter the if-condition and stop

How do breakpoints work?

- By modifying the binary to call into gdb when the execution reaches a breakpoint
- You can set multiple breakpoints at the same time
- For more on breakpoints, read https://interrupt.memfault.com/blog/cortex-m-breakpoints
- Okay, we are at that line in the execution. What next?

A discussion of termination condition

- For HW 1, it was: when the simulation time is about to exceed the given total time

A discussion of termination condition

- For HW 1, it was: when the simulation time is about to exceed the given total time
- We will process events for as long as the cars have not reached their destination
- Suppose you have a single priority queue, *events*, that stores both car events and intersection events. A car is not reinserted into the queue if it reaches its destination
- In other words, the termination condition is when the queue only has intersection events: **events.size() == controllers.size()**

How do we print the queue size?

- To print anything that is in present in the local *frame*, you can type **print <variable-name>** in the gdb shell
- When the program stops at the breakpoint, the frame is simulator.cpp, function main. You can just run **print events.size()**
- Let's say we find the queue size is 1470. If there are 1462 intersections, this means 8 cars haven't reached the destination. 15 – 8 = 7 cars (since, t = 1) have reached their destination
- If instead, we had found the queue size to 5000, that would indicate a different issue!

Let’s see where we stand

- Which event should we examine more carefully?

```
while(events.size() > controllers.size()) {
    const event current = events.top();
    events.pop();

    if(current.get_time() > 10000) {
        std::cout << "Problem!" << std::endl;
    }

    if(/* current is an intersection event */) {
        // handle intersection logic, call transition,
        // reinsert...
    } else {
        // find out if the car can move past
        // its current intersection
    }
}
```

Next steps

- Suppose you also print out **current** when stopped at the breakpoint
- **current** happens to be an intersection event
- It is most productive to examine a car event and see why the car doesn’t travel to the next intersection
- We can clear the breakpoint when we are stopped at it by **clear**
- Then we can set a new breakpoint by **breakpoint simulator.cpp:120**
- To make the program begin execution again, we run **continue**

Examining execution line by line

- When we continue from the former breakpoint, the code processes a few intersection events. Then it looks at a car event and stops at the new breakpoint.
- We want to see what's wrong with the car event
  - Is the street light it's at *RED*?
  - Is the next street full of cars (0 leftover capacity)?
- To execute one line and stop again, run **next**
- This will finish any function calls in the previous line before stopping
- Note: When it's stopped at a line, it hasn't executed that line yet!

Stepping into another function

- Suppose you reach a line that calls a function in car.cpp:

  cur_car.move();

- If you want to step into this function, you can run the command **step** when you reach this line
- Note that if that line was written as,

  allCars[car_index].move();

  then step will first take you to the std::vector::[] operator
- You can examine whether the street light is red or the capacity is full by the **print** command. Or by checking which if or else block the code goes to

Suppose we find that the capacity is 0

- We are testing with heavy traffic, so initial capacity is 2
- This is also a problem, since the capacity shouldn’t be 0 this late
- We keep debugging like this and find that our logic for incrementing capacity of the next street is flawed
- We fix the bug and move on to the next

Moral of the story

- Always have a mental image of what the program state is and how it is executing
- Fill gaps in understanding with the help of gdb. Otherwise, you will be lost!
- Add print statements for auxiliary information. Add extra conditions so that the program stops at just the right place

Segmentation faults

- When your program accesses an illegal memory address, you get a segmentation fault
- Two common reasons in this course so far
  - You access an illegal index of a vector
  - You dereference a pointer that does not point to a valid object
- A lot of these issues fall into the **undefined behavior** category

The curse of undefined behavior

- I was debugging a student code for HW 1 that segfaulted in some default object destructor
- All tracing using gdb just proved misleading
- Later, using *valgrind*, I found that there was a memory corruption because a vector was being accessed with index -1
- In a chess program I was writing, two consecutive calls to the same deterministic function produced different results before the program crashed. The cause was the same illegal array access
- **valgrind** is a tool that can help with memory corruption issues

Debugging normal segfault

- Run the program using gdb
- It will stop when segfault occurs
- Then you can investigate which program line caused it
- The frame of the execution will be inside C++ library for handling segfault. You will need to look at the program trace

Printing program trace with **backtrace**

```
gdb$ bt
#0  0xb7fdd424 in __kernel_vsyscall ()
#1  0xb7d514d2 in __lll_lock_wait () at ../nptl/sysdeps/unix/sysv/linux/i386/i686/../i486/lowlevellock
#2  0xb7d4ced4 in _L_lock_776 () from /lib/i386-linux-gnu/libpthread.so.0
#3  0xb7d4cd13 in __GI___pthread_mutex_lock (mutex=0x820c4a8 <bold::AgentState::getInstance()::instance
#4  0x080ab34b in __gthread_mutex_lock (__mutex=0x820c4a8 <bold::AgentState::getInstance()::instance+8
.8/bits/gthr-default.h:748
#5  lock (this=0x820c4a8 <bold::AgentState::getInstance()::instance+8>) at /usr/include/c++/4.8/mutex
#6  lock_guard (__m=..., this=<synthetic pointer>) at /usr/include/c++/4.8/mutex:414
#7  getTrackerState<bold::HardwareState> (this=0x820c4a0 <bold::AgentState::getInstance()::instance>)
AgentState/agentstate.hh:163
#8  get<bold::HardwareState> () at /home/drew/bold-humanoid/./Agent/../AgentState/agentstate.hh:156
#9  bold::Agent::run (this=0x8234468) at /home/drew/bold-humanoid/Agent/run.cc:21
#10 0x0808f5cc in main (argc=0x1, argv=0xbffff834) at /home/drew/bold-humanoid/main.cc:164
```

Maneuvering between frames

- The **backtrace** command prints the program callstack
- Each function on the stack is labeled with a number starting from 0
- Run **frame <frame-num>** to jump to the context of a specific function on the stack. Then you can print local variables of that function