Operating Systems Principles (CSE 30341)

Spring 2011 - University of Notre Dame

GDB tutorial

Introduction:

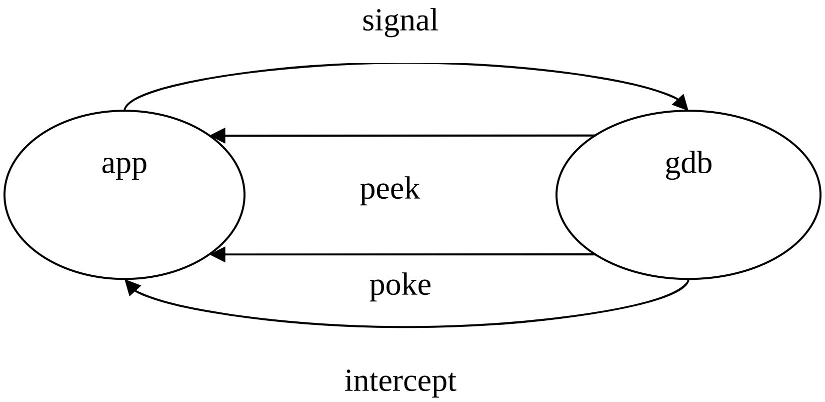
Generally, a debugger is a tool which allows programmers to conveniently test and debug programs. A debugger allows one to keep track of the execution of the program (e.g. how value of certain variable changes during execution) and to see what the program was doing when it crashed (e.g. SEG FAULT).

GDB is a popular debugger for debugging programs written in high-level languages like C. GDB helps you catch bugs in your program.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

GDB is kind of an interceptor!



Debugging with GDB:

Find the file ‘foo.c’ and its Makefile. The file ‘foo.c’ contains a program that implements a simple linked-list with employee name and age for a company.

Compile ‘foo.c’.

Start running this program and insert employee records.

When you try to insert a new employee record, you will get a SEG FAULT. How would you solve this? You want to know what the program was doing when it crashed.

Next we will debug the program using GDB.

Compiling your code with GDB

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

Under Linux, you have to specify the ‘-g’ option when you run the compiler when you want to for debug with gdb. Change your Makefile to look like this:

```
foo:            foo.c
                gcc -g foo.c -o foo
clean:
                rm -f foo
all:            foo
```

NOTE: If you have a larger program with several files, each file must be compiled with the ‘-g’ flag, and it must also be set when you link.

Running your code with GDB

After successful compilation, you can run your code with GDB and track bugs. Start running your program with GDB:

```
>gdb <progname>
```

The above command starts the GDB environment. Next, you can run executable under GDB using the run command:

```
(gdb) run <argument 1, argument 2, …>
```

Again, if you try to insert a new employee record, you will find a SEG FAULT. As I said before, in order to solve this you would want to know what happened to the program when it crashed. The **backtrace** command shows the stack trace (i.e., the chain of function calls that leads to the latest state of the program). Run **backtrace** as:

```
(gdb) backtrace
```

For the code ‘foo.c’, the backtrace command shows that you got a SEG FAULT inside the function *insert_to_list* (the line# where the SEG FAULT happened is also shown). The function *insert_to_list* is called from the function *add_employee_info*, which in turn, is called from the *main* function.

Use the **kill** command in gdb to stop execution of the running program.

```
(gdb)  kill
```

You can get out of GDB environment with **quit** command.

```
(gdb) quit
```

The file ‘foo1.c’ is another version of the same program which does not result in SEG FAULT (try to figure out the difference between the two versions and also why the new version does not crash). Compile the new version and run your code again with gdb.

Getting help on debugger commands

Use the **help** command. Gdb has a description for every command it understand, and there are many, many more then this tutorial covers. The argument to help is the command you want information about. If you just type **help** with no arguments, you will get a list of help topics similar to the following:

```
(gdb) help
```

Type **help** followed by a class name for a list of commands in that class.
Type **help** followed by command name for full documentation.

Tracking logical errors

Breakpoints

GDB is also very convenient for tracking logical errors. GDB allows you to create breakpoints anywhere inside your source using the following command:

```
(gdb) break <progname>:<linenumber>
```

One can also set a breakpoint at the start of the main function:

```
(gdb) break main
```

Afterwards, when you run your program under GDB, the execution will stop at the breakpoint(s) specified. You can check the state of execution to find possible source of logical errors.

You can set a temporary breakpoint by using the **tbreak** command instead of break. A temporary breakpoint only stops the program once, and is then removed. Use the **info breakpoints** command.

```
(gdb) info breakpoints
```

In order to get a list of breakpoints, use the **info breakpoints** command.

```
(gdb) info breakpoints
```

In order to disable breakpoints, Use the **disable** command. Pass the number of the breakpoint you wish to disable as an argument to this command. You can find the breakpoint number in the list of breakpoints.

```
(gdb) disable <breakpointnumber>
```

To skip a breakpoint a certain number of times, use the **ignore** command. The **ignore** command takes two arguments: the breakpoint number to skip, and the number of times to skip it.

```
(gdb) ignore <breakpointnumber> <#timestoskip>
```

Often, you might want to run the code between two breakpoints as a whole. The command **continue** allows you to do that.

```
(gdb) continue
```

Stepping through the execution

Now you can execute the program step-by-step and monitor the changes in different variables. There are two commands for step-by-step execution :

1. **step:** executes the current statement and stops on the next statement to be executed.
2. **next:** works similar to step. However, it the current statement is a function call, it will execute the function call as a whole and stop on the next statement.

These commands will allow you to execute your code step-by-step. While executing instructions step-by-step, you can check values in local variables for logical errors.

Watchpoints

Watchpoints are similar to breakpoints. However, watchpoints are not set for functions or lines of code; instead they are set on variables. When those variables are read or written, the watchpoint is triggered and program execution stops. Use watch command to set a watchpoint for a variable.

```
(gdb) watch <variablename>
```

Thus, GDB allows you to track down the cause of a crash during execution or track down which variable makes it go wrong.

Examining the Data and Source Code

To check the value of a variable, use the **print** command.

```
(gdb) print <variablename>
```

To print lines from a source file, use the ‘list’ command. By default, ten lines are printed.

```
(gdb) list
```

Further readings

The tutorial above should give you an idea on the capabilities of GDB. There are many documents on GDB available online. The original GDB documentation is a good source to find out all the features of GDB. You can find it here:

http://sources.redhat.com/gdb/current/onlinedocs/

GDB also provides some interesting features when used with Emacs. You can find more about GDB under Emacs at the following url:

http://tedlab.mit.edu/~dr/gdbintro.html