GDB Tutorial 1

Debugger is a tool to test and debug other programs. A debugger is called a source code Debugger or symbolic debugger if it can show the current point of execution, or the Location of a program error, inside the program's source code.

With a source code debugger, you can step through your code line by line, see what path is taken through the program's conditional and loop statements, show what functions are called, and Where in the function call stack you are. You can inspect the values of variables. You can set breakpoints on individual lines of code, and then let the program run until it reaches this line; this is convenient for navigating through complicated programs.

Getting started with Gdb

Gnu Gdb is a source level debugger, that is widely used on *nix platforms. This part of the tutorial we intend to provide an overview of widely used gdb commands. Gdb kind of Source level debuggers require some debugging information built into the binary application. That means additional symbol and metadata information has to be built into the executable binaries. This can be achieved by instructing the compiler to include symbol, metadata information describing the syntax of our source program.

```
info@techveda.org:~# gcc -g -c sample.c –o sample.o
info@techveda.org:~# gcc sample.o –o sample
```

-g flag instructs gdb to generate and add debug information. Let's run objdump on relocatable object file and find additional debug sections generated for gdb's use.

```
info@techveda.org:~# objdump -h sample.o

sample.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000e8  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000011c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  0000011c  2**2
                  ALLOC
  3 .debug_abbrev 000000a2  00000000  00000000  0000011c  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .debug_info   00000180  00000000  00000000  000001be  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  5 .debug_line   00000050  00000000  00000000  0000033e  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  6 .rodata       00000005  00000000  00000000  0000038e  2**0
```

```
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .debug_loc    000000e0  00000000  00000000  00000393  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_pubnames 00000033  00000000  00000000  00000473  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_pubtypes 00000012  00000000  00000000  000004a6  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
 10 .debug_aranges 00000020  00000000  00000000  000004b8  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
 11 .debug_str    000000b0  00000000  00000000  000004d8  2**0
                  CONTENTS, READONLY, DEBUGGING
 12 .comment      0000002b  00000000  00000000  00000588  2**0
                  CONTENTS, READONLY
 13 .note.GNU-stack 00000000  00000000  00000000  000005b3  2**0
                  CONTENTS, READONLY
 14 .debug_frame  00000094  00000000  00000000  000005b4  2**2
                  CONTENTS, RELOC, READONLY, DEBUGGING
```

All the sections starting with .debug have been generated at compile time and include debug information, the contents of the above sections are arranged/ organized as per Debugging information formats, and these formats are specifications that describe layout of debug info in an object file. There are several debugging formats: stabs, COFF, PE-COFF, OMF, IEEE-695, and three versions of DWARF, to name a few.

Dwarf is the most widely used debug information format, In Linux, it is the default debugging format. Description of Dwarf standard and how the above sections are organized can be found here . Let's conclude for now that additional debug sections identified are dwarf sections, and gdb interprets dwarf information in those debug sections. Let's initiate debug session using gdb with an executable image

```
info@techveda.org:~#  gdb ./sample
GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/sample...done.
(gdb)
```

Gdb starts by reading dwarf debug symbol information from the presented executable and shows up gdb prompt. Using gdb commands we can instruct gdb to run, stop, continue and perform various operations on the executable binary. Lets begin our session with help command

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Gdb supports various operations on the executable and commands have been classified into groups shown above, help command can also be used to print more commands in each category list shown above. Help can also be used to get documentation on specific gdb command. Gdb also supports execution of shell commands from gdb prompt, this facility comes very handy while inspecting multi-process, multi-thread, network communication applications and to run various tools without terminating current debug session.

```
(gdb) shell ls -l /usr/src
total 115188
drwxrwxr-x 25 root root     4096 2012-09-06 12:53 linux-3.2.9
-rw-r--r--  1 root src  78132997 2012-04-08 22:12 linux-3.2.9.tar.bz2
drwxr-xr-x 24 root root     4096 2011-04-26 04:38 linux-headers-2.6.38-8
drwxr-xr-x  7 root root     4096 2011-04-26 04:38 linux-headers-2.6.38-8-generic
-rw-r--r--  1 root src  39797000 2012-04-09 18:46 linux-image-3.2.9.debug_3.2.9.debug-10.00.Custom_i386.deb
drwxr-xr-x  5 root root     4096 2012-03-25 21:17 linux-source-2.6.38
```

```
(gdb)
(gdb) shell objdump -h ./sample.o

./sample.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000e8  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000011c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  0000011c  2**2
                  ALLOC
  3 .debug_abbrev 000000a2  00000000  00000000  0000011c  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .debug_info   00000180  00000000  00000000  000001be  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  5 .debug_line   00000050  00000000  00000000  0000033e  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  6 .rodata       00000005  00000000  00000000  0000038e  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .debug_loc    000000e0  00000000  00000000  00000393  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_pubnames 00000033  00000000  00000000  00000473  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_pubtypes 00000012  00000000  00000000  000004a6  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
 10 .debug_aranges 00000020  00000000  00000000  000004b8  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
 11 .debug_str    000000b0  00000000  00000000  000004d8  2**0
                  CONTENTS, READONLY, DEBUGGING
 12 .comment      0000002b  00000000  00000000  00000588  2**0
                  CONTENTS, READONLY
 13 .note.GNU-stack 00000000  00000000  00000000  000005b3  2**0
                  CONTENTS, READONLY
 14 .debug_frame  00000094  00000000  00000000  000005b4  2**2
                  CONTENTS, RELOC, READONLY, DEBUGGING
```

List command can used to view source instructions of the executable attached to debugger, we can access documentation on list command using *help* command sample executable that we have currently attached to gdb, has one source file sample.c.

```
(gdb) help list
```

```
List specified function or line.
With no argument, lists ten more lines after or around previous listing.
"list -" lists the ten lines before a previous ten-line listing.
One argument specifies a line, and ten lines are listed around that line.
Two arguments with comma between specify starting and ending lines to list.
Lines can be specified in these ways:
  LINENUM, to list around that line in current file,
  FILE:LINENUM, to list around that line in that file,
  FUNCTION, to list around beginning of that function,
  FILE:FUNCTION, to distinguish among like-named static functions.
  *ADDRESS, to list around the line containing that address.
With two args if one is empty it stands for ten lines away from the other arg.
(gdb)

(gdb) list 1

1       #include <stdio.h>
2       #include <stdlib.h>
3
4       int add(int x, int y)
5       {
6               return (x + y);
7       }
8
9       int sub(int x, int y)
10      {
(gdb) <Enter key>
11              return (x - y);
12      }
13
14      int mul(int x, int y)
15      {
16              return (x * y);
17      }
18
19      int main()
20      {
(gdb) <Enter key>
21              int a = 10, b = 20;
22              int result;
23              int (*fp) (int, int);
24
25              fp = add;
26              result = (*fp) (a, b);
```

```
27              printf(" %d\n", result);
28
29              fp = sub;
30              result = fp(a, b);

(gdb)
```

To instruct gdb to run current program executable attached use ***run*** command

```
(gdb) run
Starting program: /root/sample
 30
 -10
 200

Program exited normally.
(gdb)
```

Break points

The principal purposes of using a debugger is to inspect elements of a running program, ***Breakpoints*** are strategic instructions in your program where debugger would stop “running program”, and would not allow program to continue until instructed. Break points aid inspection of stack frames, local data values, global data values, processor register dump and so on. A program can be set with any number of break points.

```
(gdb) help break
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
With no LOCATION, uses current execution address of the selected
stack frame.  This is useful for breaking on return to a stack frame.

THREADNUM is the number from "info threads".
CONDITION is a boolean expression.

Multiple breakpoints at one place are permitted, and useful if their
conditions are different.

Do "help breakpoints" for info on other commands dealing with breakpoints.
(gdb)
```

Above Documentation clearly states how break command can be used to set breakpoints at a Particular line no, function, or an address. Break points can be joined with conditional expressions which are often referred as conditional breakpoints.

```
(gdb) break 1
Breakpoint 1 at 0x80483c7: file sample.c, line 1.
(gdb) break 4
Note: breakpoint 1 also set at pc 0x80483c7.
Breakpoint 2 at 0x80483c7: file sample.c, line 4.
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3
4       int add(int x, int y)
5       {
6               return (x + y);
7       }
8
9       int sub(int x, int y)
10      {
(gdb) break add
Note: breakpoints 1 and 2 also set at pc 0x80483c7.
Breakpoint 3 at 0x80483c7: file sample.c, line 6.
(gdb)
```

We have now set three break points, first and second break points have been set using line no parameter and third break point has been set using fuction name. gdb maintains break point information in a table which can be viewed with command ***info breakpoints***

```
(gdb) info breakpoints
Num    Type         Disp Enb Address    What
1      breakpoint   keep y  0x080483c7 in add at sample.c:1
2      breakpoint   keep y  0x080483c7 in add at sample.c:4
3      breakpoint   keep y  0x080483c7 in add at sample.c:6
(gdb)
```

Gdb’s breakpoint table provides attributes for each break point. Each breakpoint has a unique identifier (see “Num” column the above table), break points can be managed using these unique identifiers.

Type field of the table mentions the type of break point, gdb categorizes break points into three types called breakpoints, watchpoints, and catchpoints.

Dispostion field(Disp) indicates what happens to the breakpoint after the next time it causes gdb to pause the programs execution. There are three possible disposition values

Keep: breakpoint will be retained after the next time it is reached. This is default value for all new breakpoints

Del: breakpoint will be deleted after the next time it is reached

Dis: breakpoint will be disabled the next time it is reached.

Enable status(Enb) field indicated whether the breakpoint is currently enabled or disabled

Address field specifies the location in memory where the breakpoint is set.

Location (What) field shows the line number and source filename of the breakpoints

We will run the program with breakpoints set

```
(gdb) run
Starting program: /root/sample

Breakpoint 1, add (x=10, y=20) at sample.c:6
6                     return (x + y);
(gdb)
```

First break point is now hit and program has stopped in the function add at line no 6 of source file. As per breakpoint table record first break point should have been hit at line 1, second break point at line no 4 of the source file , but here we see gdb reports that first break point is hit and line no shown is 6

```
(gdb) list 1
1          #include <stdio.h>
2          #include <stdlib.h>
3
4          int add(int x, int y)
5          {
6                     return (x + y);
7          }
8
9          int sub(int x, int y)
10         {
```

Line no 1 and 4 of the source file at which breakpoints 1 and 2 were set do not contain processor executable instructions as breakpoints are set at machine instruction level, so first breakpoint set at line 1 is automatically moved to first executable instruction after line 1 which happens to be line 5 that contains preamble to add function, program executed preamble instruction and break point was hit. This test clearly shows that when break points are set using line no as parameter, may not break precisely at the specified source line no.

Now since program has hit the first break point lets probe program data variables, function arguments and register values using gdb commands .To print the list of functions executed until this breakpoint we can use gdb command "***backtrace"***

```
(gdb) backtrace
#0  add (x=10, y=20) at sample.c:6
#1  0x08048425 in main () at sample.c:26
(gdb)
```

Above trace shows functions executed so far until this breakpoint and their stack frames. #0 entry is the current function and signifies current stack frame, #1 signifies the caller function. Main function has called add function with 2 integer arguments and we can see values passed

Print command can show status of parameters and local variables of the current frame. It can be used with parameter/variable name or address, it can also be used access address of local variables and parameters in stack frame

Set command lets us alter values of local variables or parameters of current frame, again we can use set command with name of the symbol or address of symbol

```
(gdb) print x
$1 = 10
(gdb) print y
$2 = 20
(gdb) print &x
$3 = (int *) 0xbffff3c0
(gdb) print &y
$4 = (int *) 0xbffff3c4
(gdb) print *(0xbffff3c0)
$5 = 10
(gdb) print *(0xbffff3c4)
$6 = 20
(gdb) set x=40
(gdb) print
```

```
$11 = 40
(gdb) set *(0xbffff3c0)=50
(gdb) print x
$12 = 50
(gdb)
```

We can access processor register values using command ***info registers,*** this command can be used to access all register values or a specific register.

```
(gdb) info registers
eax          0x80483c4          134513604
ecx          0xcca81810         -861399024
edx          0x1        1
ebx          0xb7fcaff4         -1208176652
esp          0xbffff3b8         0xbffff3b8
ebp          0xbffff3b8         0xbffff3b8
esi          0x0        0
edi          0x0        0
eip          0x80483c7          0x80483c7 <add+3>
eflags       0x286    [ PF SF IF ]
cs           0x73       115
ss           0x7b       123
ds           0x7b       123
es           0x7b       123
fs           0x0        0
gs           0x33       51
(gdb) info register
eax          0x80483c4          134513604
ecx          0xcca81810         -861399024
edx          0x1        1
ebx          0xb7fcaff4         -1208176652
esp          0xbffff3b8         0xbffff3b8
ebp          0xbffff3b8         0xbffff3b8
esi          0x0        0
edi          0x0        0
eip          0x80483c7          0x80483c7 <add+3>
eflags       0x286    [ PF SF IF ]
cs           0x73       115
ss           0x7b       123
ds           0x7b       123
es           0x7b       123
fs           0x0        0
gs           0x33       51
(gdb) info register ebp
```

```
ebp          0xbffff3b8               0xbffff3b8
(gdb) info register esp
esp          0xbffff3b8               0xbffff3b8

(gdb) info frame
Stack level 0, frame at 0xbffff3c0:
 eip = 0x80483c7 in add (sample.c:6); saved eip 0x8048425
 called by frame at 0xbffff3f0
 source language c.
 Arglist at 0xbffff3b8, args: x=50, y=20
 Locals at 0xbffff3b8, Previous frame's sp is 0xbffff3c0
 Saved registers:
  ebp at 0xbffff3b8, eip at 0xbffff3bc
(gdb)
```

Info frame command shows details of current stack frame and processors registers like eip, esp and so on. Data shown includes argument addresses, values and caller frames base pointer, and stack pointer. We can move down into caller functions stack frame or any other stack frame show in the back trace using ***select-frame*** command and examine its contents

```
(gdb) select-frame 1
(gdb) info frame
Stack level 1, frame at 0xbffff3f0:
 eip = 0x8048425 in main (sample.c:26); saved eip 0xb7e83e37
 caller of frame at 0xbffff3c0
 source language c.
 Arglist at 0xbffff3e8, args:
 Locals at 0xbffff3e8, Previous frame's sp is 0xbffff3f0
 Saved registers:
  ebp at 0xbffff3e8, eip at 0xbffff3ec
(gdb)
```

Lets examine local data of the main function

```
(gdb) list main
15          {
16                  return (x * y);
17          }
18
19          int main()
20          {
21                  int a = 10, b = 20;
22                  int result;
23                  int (*fp) (int, int);
24
(gdb)
25                      fp = add;
26                      result = (*fp) (a, b);
27                      printf(" %d\n", result);
28
29                      fp = sub;
30                      result = fp(a, b);
31                      printf(" %d\n", result);
32
33                      fp = mul;
34                      result = (*fp) (a, b);
(gdb)
35                      printf(" %d\n", result);
36                      return 0;
37          }
(gdb)
```

Main has symbol a, b, result as local integer variables, lets access their values from stack.

```
(gdb) print a
$13 = 10
(gdb) print b
$14 = 20
(gdb) print result
$15 = -1209414347
(gdb) print *(0xbffff3d0)
$17 = -1209414347
(gdb)
```

Main function also has the function pointer “fp” which should be currently initialized with address of add function, since add is already called and we stepped down from add functions frame into main, we should get add function address in the function pointer

```
(gdb) print fp
$18 = (int (*)(int, int)) 0x80483c4 <add>
```

we can probe the current execution state of the program using ***info program*** command

```
(gdb) info program
            Using the running image of child process 5589.
Program stopped at 0x80483c7.
It stopped at breakpoint 1.
It stopped at breakpoint 2.
It stopped at breakpoint 3.
(gdb)
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x080483c7 in add at sample.c:1
                breakpoint already hit 1 time
2       breakpoint     keep y   0x080483c7 in add at sample.c:4
                breakpoint already hit 1 time
3       breakpoint     keep y   0x080483c7 in add at sample.c:6
                breakpoint already hit 1 time
(gdb)
```

Dump shows we are at break point , it shows all the three breakpoints with hit status, there is no literal effect of breakpoint 1 and 2 for the reasons we have understood earlier. We can delete or disable breakpoints using ***delete*** and ***disable*** commands

```
(gdb) disable 1
(gdb) disable 2
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep n   0x080483c7 in add at sample.c:1
                breakpoint already hit 1 time
2       breakpoint     keep n   0x080483c7 in add at sample.c:4
                breakpoint already hit 1 time
3       breakpoint     keep y   0x080483c7 in add at sample.c:6
                breakpoint already hit 1 time
(gdb) enable 1
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x080483c7 in add at sample.c:1
                breakpoint already hit 1 time
2       breakpoint     keep n   0x080483c7 in add at sample.c:4
                breakpoint already hit 1 time
3       breakpoint     keep y   0x080483c7 in add at sample.c:6
                breakpoint already hit 1 time
(gdb)
```

Disabling a breakpoint will change its Enb disposition value to n, such breakpoints will have no effect on the program until enabled again. Disabled Breakpoints can be enabled back using ***enable*** command. All active breakpoints can be set to disabled

```
(gdb) disable breakpoints
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
```

```
1       breakpoint     keep n   0x080483c7 in add at sample.c:1
                breakpoint already hit 1 time
2       breakpoint     keep n   0x080483c7 in add at sample.c:4
                breakpoint already hit 1 time
3       breakpoint     keep n   0x080483c7 in add at sample.c:6
                breakpoint already hit 1 time
(gdb)
(gdb) delete 1
(gdb) delete 2
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
3       breakpoint     keep n   0x080483c7 in add at sample.c:6
                breakpoint already hit 1 time
(gdb) delete breakpoints
Delete all breakpoints? (y or n) y
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb)
```

Similarly breakpoints can be deleted selectively or all breakpoints can be removed with single command.

Program execution can be resumed from a breakpoint using ***continue*** which runs the program until next breakpoint is hit or we can execute program in single instruction step mode using ***step*** command, or execute next source operation and stop using ***next*** command. We will get acquaintance with next, step and further commands in our next part of this article, for now will close the current debug session issuing ***continue*** command and let the program run to completion.

```
(gdb) continue
Continuing.
 70
 -10
 200

Program exited normally.
(gdb)
```

References

1. Introduction to the DWARF format

2. Gdb quick reference