# GDB Practice session

## Typographical conventions

We use the following conventions in this guide:

vi The name of a specific command or file

*file*  You should replace *file* with a specific name

(gdb)help quit at the gdb prompt, type the command help quit, then press the <ENTER> key

### **Exit gdb** Output that you see on the screen

## Getting Started

1. Login into the Linux server with your login Ids. Open two terminals, one to work in gdb prompt and the other to work on the shell.
2. Create a new directory called gdb in your home directory <home>

mkdir gdb

1. Go inside the directory you have created in (2) /<home>/gdb

cd gdb

1. Copy the following files from the path as mentioned by the figdb\_trainer:
   1. gdb\_sample\_pointer\_1\_0.c
   2. gdb\_sample\_core\_1\_0.c
   3. bugs\_wall.c
   4. first\_fit\_bugs.c
   5. gdb\_practice\_array.c
2. Although all commands in gdb are full words, you can type any unambiguous prefix. For instance, p instead of print.
3. Use the help command in gdb for a list of topics. Then type help *topic* for information on a specific topic. help *cmd* gives help on the command cmd e.g. help print will give information about the print command. For comprehensive coverage of gdb, read the Info pages. At the shell prompt, type info. When info starts, type mgdb<ENTER>.

(Please note that the info command is not present on the server 10.203.161.9)

## Compilation

1. Compile the file gdb\_sample\_pointer\_1\_0.c and put the output in the executable file called sample

gcc –o sample –g gdb\_sample\_pointer\_1\_0.c

## Execution with gdb

1. Execute the file sample with gdb

gdb sample

(gdb) r

1. Execute the file again, this time redirecting the output of the program to another file

(gdb) r > output

Quit gdb

(gdb) q

View the Output file

cat output

1. To have a look at the source code at any point in time

gdb sample

(gdb) list

By default the number of lines displayed is 10, to change the number of lines to be displayed

(gdb) set listsize 20

(gdb) list

## Breakpoints

1. Before executing the program this time, put a breakpoint

(gdb) b main

(gdb) r

Breakpoints can be put using either:

* the function name or
* the line number or
* source file name: line number

Try putting the breakpoint using all the above ways

1. Information of all the set breakpoints can be seen

(gdb) info b

The first column shows the id of the breakpoint

1. Delete all the breakpoints except the one on function main

(gdb) d *id1 id2* …

1. Now run the program and observe that the execution stops at the first breakpoint that is encountered. Now the control of the program is with gdb and each statement can be executed one at a time using the command:

(gdb) n

Run the n command till the statement before the call to function display is executed

1. Step inside the function display

(gdb) s

1. Let the program continue on its own

(gdb) c

1. Lets now look into the contents of a variable and alter the value

(gdb) p \*zptr

(gdb) p z

(gdb) p z = 10

(gdb) p \*zptr = 25

p command can also be used to compute an expression or a function

(gdb) p strlen(“abc”)

***$4 = 3***

1. Breakpoints can be disabled and enabled using the following commands:

(gdb) dis <id>

(gdb) en <id>

Check the status of the breakpoint after disabling and enabling the breakpoint

1. A watchpoint is a special breakpoint that stops your program when the value of an expression changes. Lets put a watchpoint on variable x in main

(gdb) b main

(gdb) r

(gdb) n

keep running the n command till the variable x is defined

(gdb) watch x

(gdb) c

What do you observe?

1. Commands to explore,

(gdb) help ***cmd***

* 1. ignore command with breakpoints: ignore <id> <number of times to be ignored>
  2. Conditional breakpoints: e.g. b main if x==6, b display if z==5
  3. Until command: Continue running until a source line past the current line, in the current stack frame, is reached.

(gdb) b main

(gdb) r

(gdb) until 16

What code scenarios can you think where the ignore and until commands can be used?

1. Execute shell commands on the gdb prompt

(gdb) shell cat gdb\_sample\_pointer\_1\_0.c

1. Try moving your source file gdb\_sample\_pointer\_1\_0.c to some other directory

(gdb) shell mv gdb\_sample\_pointer\_1\_0.c ../

(gdb) r

***gdb\_sample\_pointer\_1\_0.c: No such file or directory***

(gdb) dir ../

(gdb) n

1. The same commands need not be typed again and again on each invocation of gdb but take the input from a file. Create a file called commands with the following lines :

b main

b display

Now run sample with gdb

gdb sample

(gdb) sou commands

1. You may want to save the output of gdb commands to a file.

(gdb) set logging on

Give a few commands to gdb as discussed earlier

(gdb) shell cat gdb.txt

What do you observe?

(Please note that the set logging on command does not work with the gdb version installed on the 10.203.161.9 server)

## Stack Frames

1. When your program has stopped, the first thing you need to know is where it stopped and how it got there. View the stack frames of the process using the back trace command

(gdb) b display

(gdb) r

(gdb) bt

***#0 display (z=5, zptr=0x7fbffff88c) at gdb\_sample\_pointer\_1\_0.c:27***

***#1 0x000000000040051d in main () at gdb\_sample\_pointer\_1\_0.c:21***

The leftmost column gives the frame id

Now, try to see the value of x (local variable of function main)

(gdb) p x

***No symbol "x" in current context.***

If you do want to see the value of x at this point in time, change the frame to 1 and proceed

(gdb) fr 1

(gdb) p x

What do you observe?

## Analysis of core dumps

1. Login to a solaris server (10.203.161.9) for executing the instructions in this section. This is required because the Linux servers have a default configuration setting due to which a core file is not created. This setting can only be changed by a super user.
2. When your process executes an invalid instruction, the operating system produces a core dump e.g. when the process accesses a memory location at an invalid address, the kernel sends the signal SIGSEGV to the process. If the process does not handle such a signal, the kernel writes the execution context and the contents of the address space in a file (core dump) and terminate the process.

With the help of gdb, this core file can be analyzed to see what was the problem.

Compile the file gdb\_sample\_core\_1\_0.c

gcc –o sample\_core –g gdb\_sample\_core\_1\_0.c

(Do not name the executable as core)

Execute the file sample\_core without gdb

./sample\_core

***Segmentation Fault (coredump)***

Execute the executable with gdb

gdb sample\_core

gdb r

***[ GDB will display the error ]***

Ask gdb to generate the core file

gdb generate-core-file

Quit GDB

{gdb

gdb sample\_core core

***#0 0x10744 in display (z=5, zptr=0x0) at gdb\_sample\_core\_1\_0.c:29***

***29 printf(" zptr points to %p which holds %d.\n", zptr, \*zptr);***

***(gdb)***

The above statement shows the statement that was executed when the segmentation fault occurred. Can you see the problem why the fault occurred?

You can use the following commands while analyzing a core file:

Display values of variables using p

Display stack frames using bt

Display frame information using info fr

Switch between different frames to see the values of variables in different frames using fr

## Attaching and detaching of gdb to/from already running processes

1. Compile the program bugs\_wall.c

gcc –o bugs\_wall –g bugs\_wall.c

Run the executable as a background process

./bugs\_wall &

***[1] 20226***

This is the process id of the process and will be used later to attach this process to gdb.

1. Start gdb on a separate terminal

gdb

(gdb) att *pid*

pid should be replaced by the actual process id

What do you observe?

Now try putting breakpoints and controlling the program flow using gdb

1. After the program is analyzed using gdb, it can be detached as follows:

(gdb) det

## Assignments

1. **Assignment:**  gdb\_practice\_array.c contains a code with bugs. The actual output is supposed to generate a sorted list and then search for 3 strings in that as shown in actual\_output. Find out the bugs in practise.c using gdb.
2. **Assignment:** First Fit Memory Management algorithm is implemented in the file first\_fit\_bugs.c which contains a few bugs. Remove these bugs with the help of gdb so that it gives expected output. Refer link below for algorithm details.

[**https://www.geeksforgeeks.org/program-first-fit-algorithm-memory-management/**](https://www.geeksforgeeks.org/program-first-fit-algorithm-memory-management/)