ATTACK DEFENSE

by PentesterAcademy

Name	Running Processes Under GDB
URL	https://www.attackdefense.com/challengedetails?cid=2165
Type	Reverse Engineering : GDB Basics

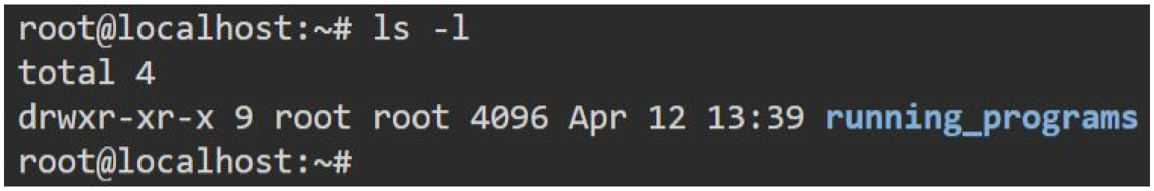
Important Note: This document illustrates all the important steps required to complete this lab. This is by no means a comprehensive step-by-step solution for this exercise. This is only provided as a reference to various commands needed to complete this exercise and for your further research on this topic. Also, note that the IP addresses and domain names might be different in your lab.

Objective: Learn how to run processes under GDB and check out different commands/options/methods.

Solution:

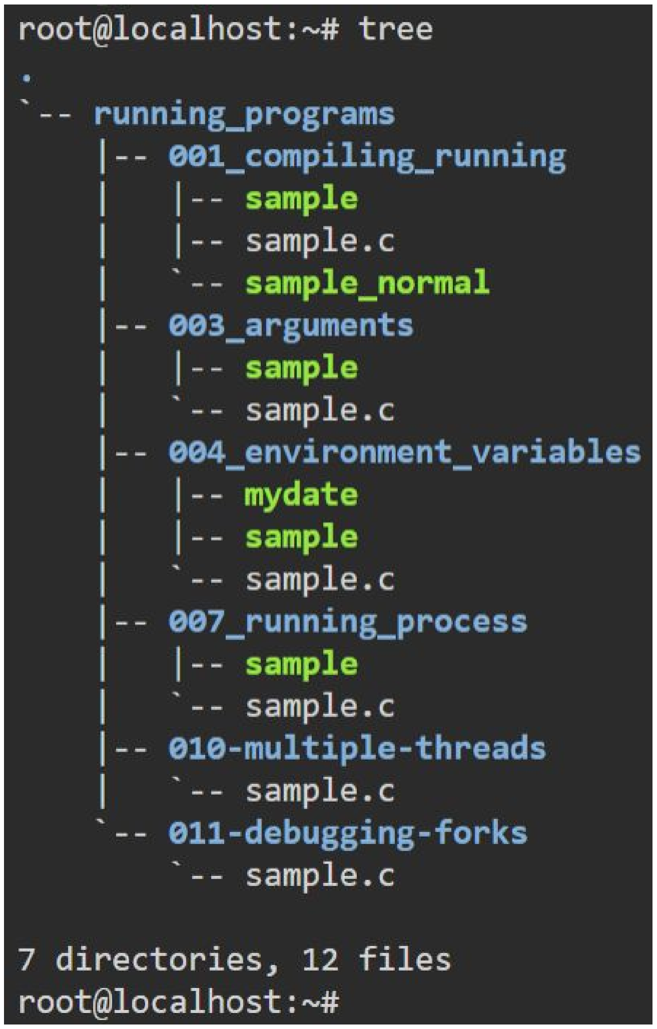
Step 1. Check the contents of the present working directory (i.e. /root/). Also print the directories files present in tree form.

Command: ls -l



```
root@localhost:~# ls -l
total 4
drwxr-xr-x 9 root root 4096 Apr 12 13:39 running_programs
root@localhost:~#
```

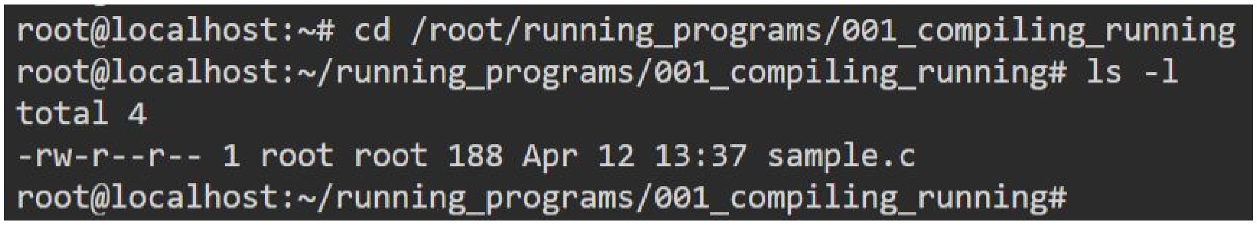
Command: tree



```
root@localhost:~# tree
.
`-- running_programs
    |-- 001_compiling_running
    |   |-- sample
    |   |-- sample.c
    |   `-- sample_normal
    |-- 003_arguments
    |   |-- sample
    |   `-- sample.c
    |-- 004_environment_variables
    |   |-- mydate
    |   |-- sample
    |   `-- sample.c
    |-- 007_running_process
    |   |-- sample
    |   `-- sample.c
    |-- 010-multiple-threads
    |   `-- sample.c
    `-- 011-debugging-forks
        `-- sample.c

7 directories, 12 files
root@localhost:~#
```

Step 2. Switch to /root/running_programs/001_compiling_running directory to start the exercise.

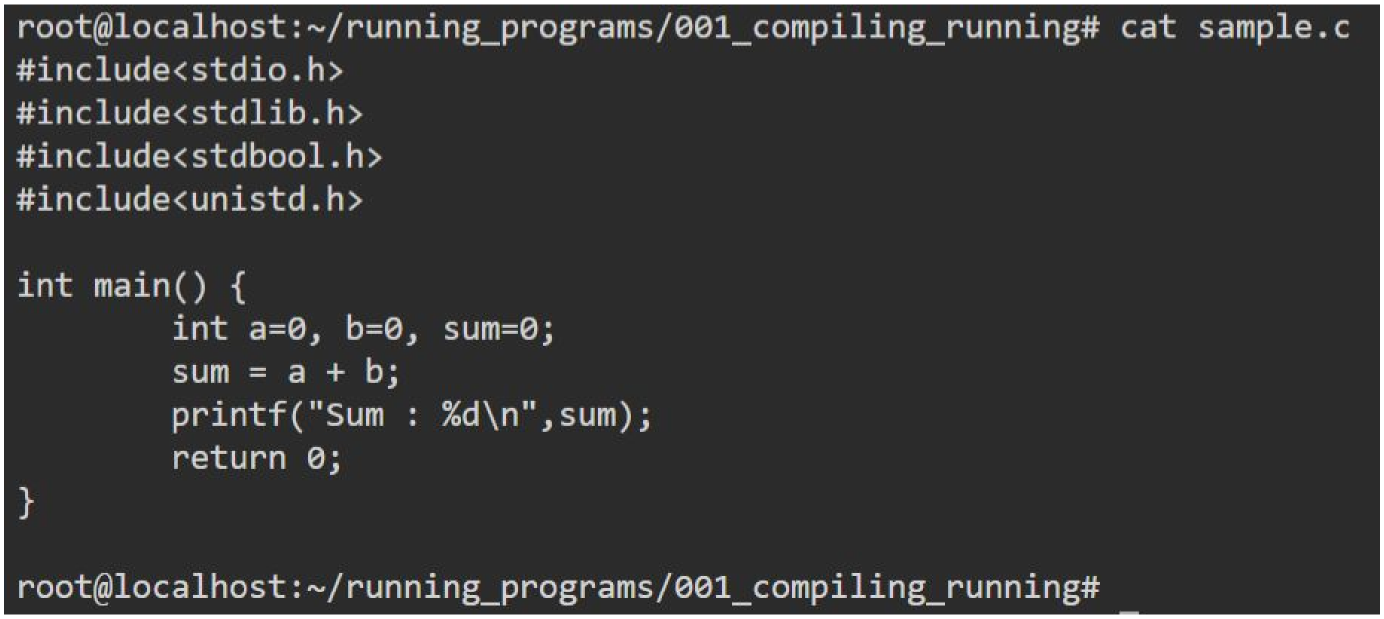
Command: cd /root/running_programs/001_compiling_running



```
root@localhost:~# cd /root/running_programs/001_compiling_running
root@localhost:~/running_programs/001_compiling_running# ls -l
total 4
-rw-r--r-- 1 root root 188 Apr 12 13:37 sample.c
root@localhost:~/running_programs/001_compiling_running#
```

Step 3: Print the contents of the given C file.

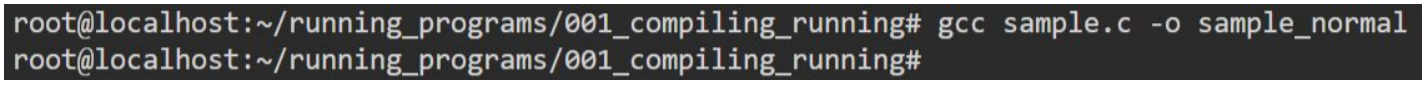
Command: cat sample.c


```
root@localhost:~/running_programs/001_compiling_running# cat sample.c
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<unistd.h>

int main() {
        int a=0, b=0, sum=0;
        sum = a + b;
        printf("Sum : %d\n",sum);
        return 0;
}

root@localhost:~/running_programs/001_compiling_running#
```

Compilation

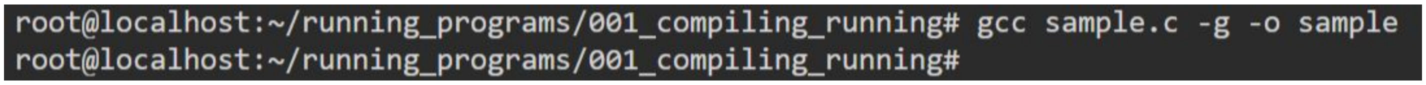
Step 4: Compile the C file with gcc in a normal way. Name the binary sample_normal

Command: gcc sample.c -o sample_normal


```
root@localhost:~/running_programs/001_compiling_running# gcc sample.c -o sample_normal
root@localhost:~/running_programs/001_compiling_running#
```

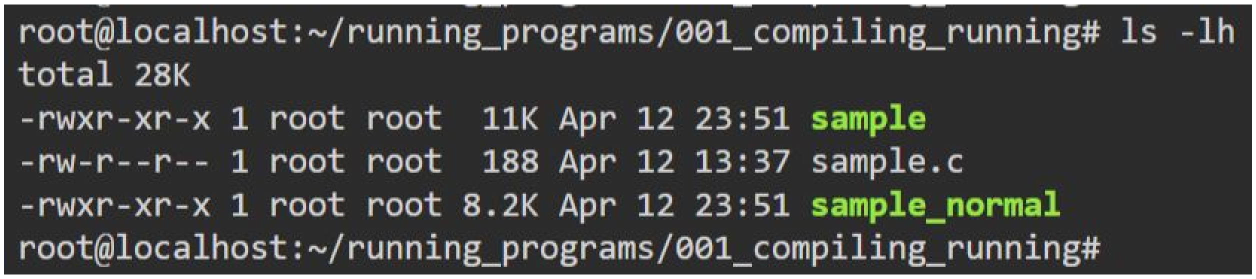
Step 5: Also, compile the C file with gcc while enabling the debug symbols. Name the binary sample

Command: gcc sample.c -d -o sample


```
root@localhost:~/running_programs/001_compiling_running# gcc sample.c -g -o sample
root@localhost:~/running_programs/001_compiling_running#
```

Step 6: Compare the size of both the binaries.

Command: ls -lh

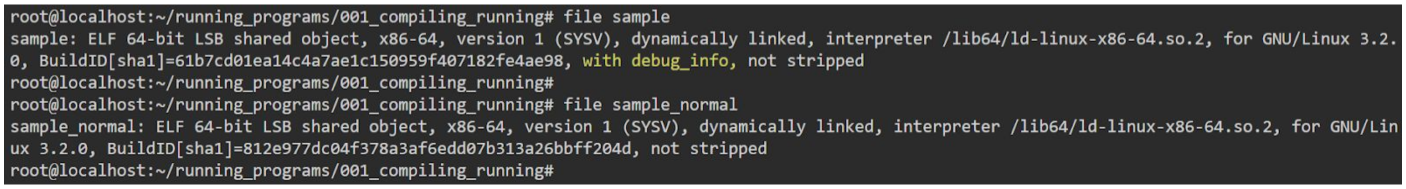


```
root@localhost:~/running_programs/001_compiling_running# ls -lh
total 28K
-rwxr-xr-x 1 root root  11K Apr 12 23:51 sample
-rw-r--r-- 1 root root  188 Apr 12 13:37 sample.c
-rwxr-xr-x 1 root root 8.2K Apr 12 23:51 sample_normal
root@localhost:~/running_programs/001_compiling_running#
```

One can observe that the binary created with debug symbols is bigger in size.

Step 7: Use file command to view details of both binaries.

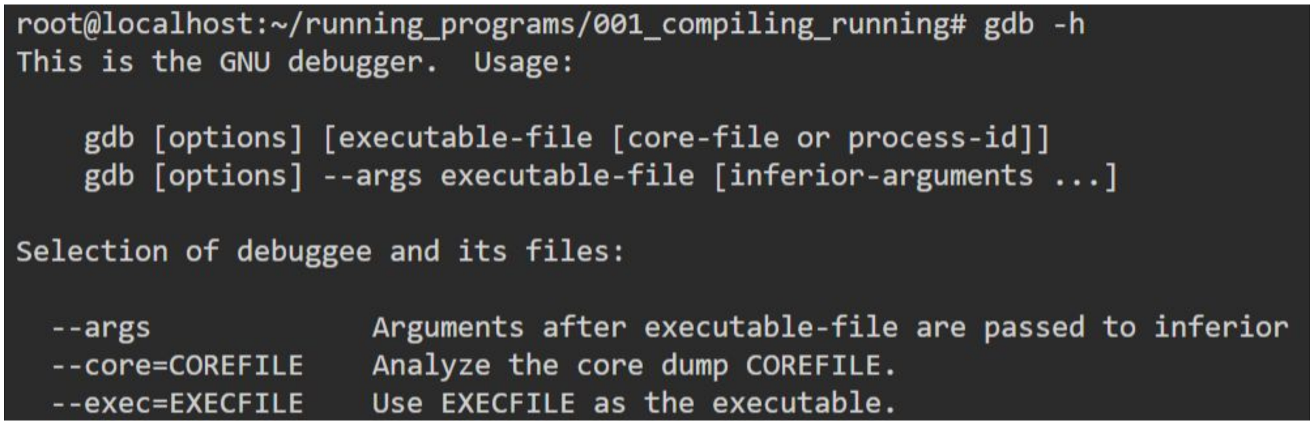
Commands:
file sample_normal
file sample



```
root@localhost:~/running_programs/001_compiling_running# file sample
sample: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.
0, BuildID[sha1]=61b7cd01ea14c4a7ae1c150959f407182fe4ae98, with debug_info, not stripped
root@localhost:~/running_programs/001_compiling_running#
root@localhost:~/running_programs/001_compiling_running# file sample_normal
sample_normal: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Lin
ux 3.2.0, BuildID[sha1]=812e977dc04f378a3af6edd07b313a26bbff204d, not stripped
root@localhost:~/running_programs/001_compiling_running#
```

Step 8: Check the help for GDB.

Command: gdb -h

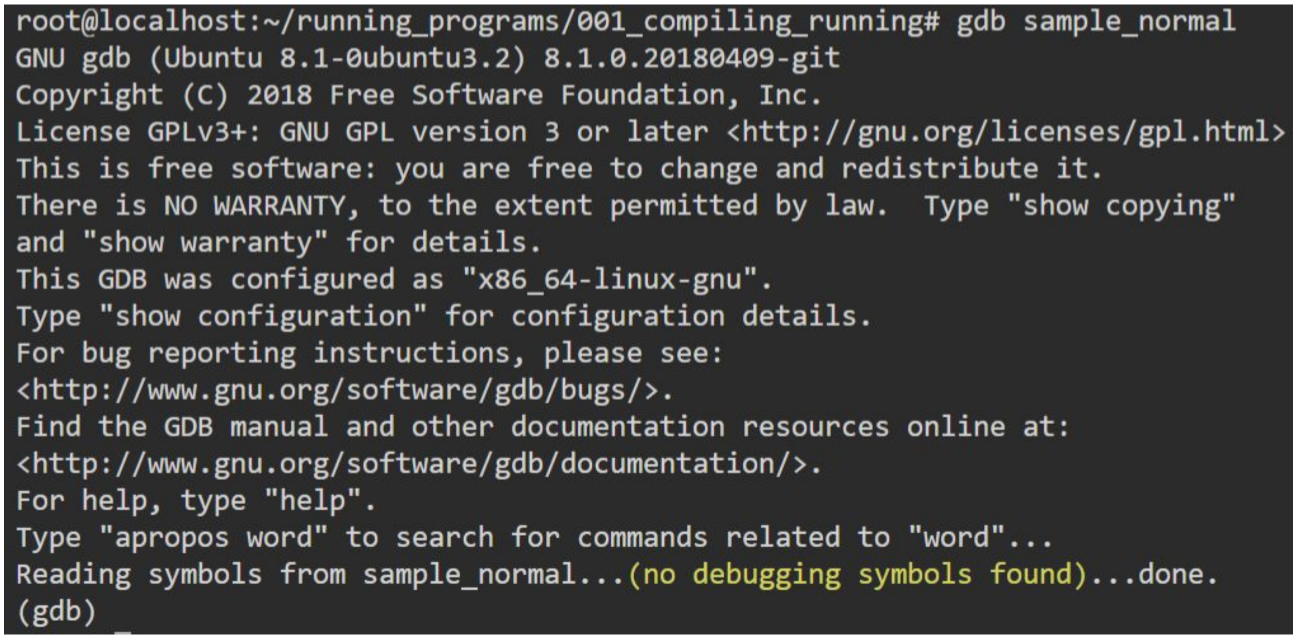


```
root@localhost:~/running_programs/001_compiling_running# gdb -h
This is the GNU debugger.  Usage:

    gdb [options] [executable-file [core-file or process-id]]
    gdb [options] --args executable-file [inferior-arguments ...]

Selection of debuggee and its files:

  --args             Arguments after executable-file are passed to inferior
  --core=COREFILE    Analyze the core dump COREFILE.
  --exec=EXECFILE    Use EXECFILE as the executable.
```

Execution in GDB

Step 9: Start GDB and open sample_normal file in it.

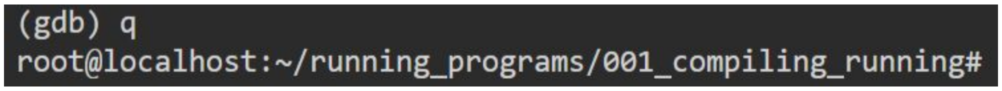
Command: gdb sample_normal



```
root@localhost:~/running_programs/001_compiling_running# gdb sample_normal
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 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 "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sample_normal...(no debugging symbols found)...done.
(gdb)
```

Observe the message “no debugging symbols found”. This is the reason for enabling the debug symbols. GDB based analysis is also possible without debugging symbols, however, that relatively becomes more time and effort-intensive.

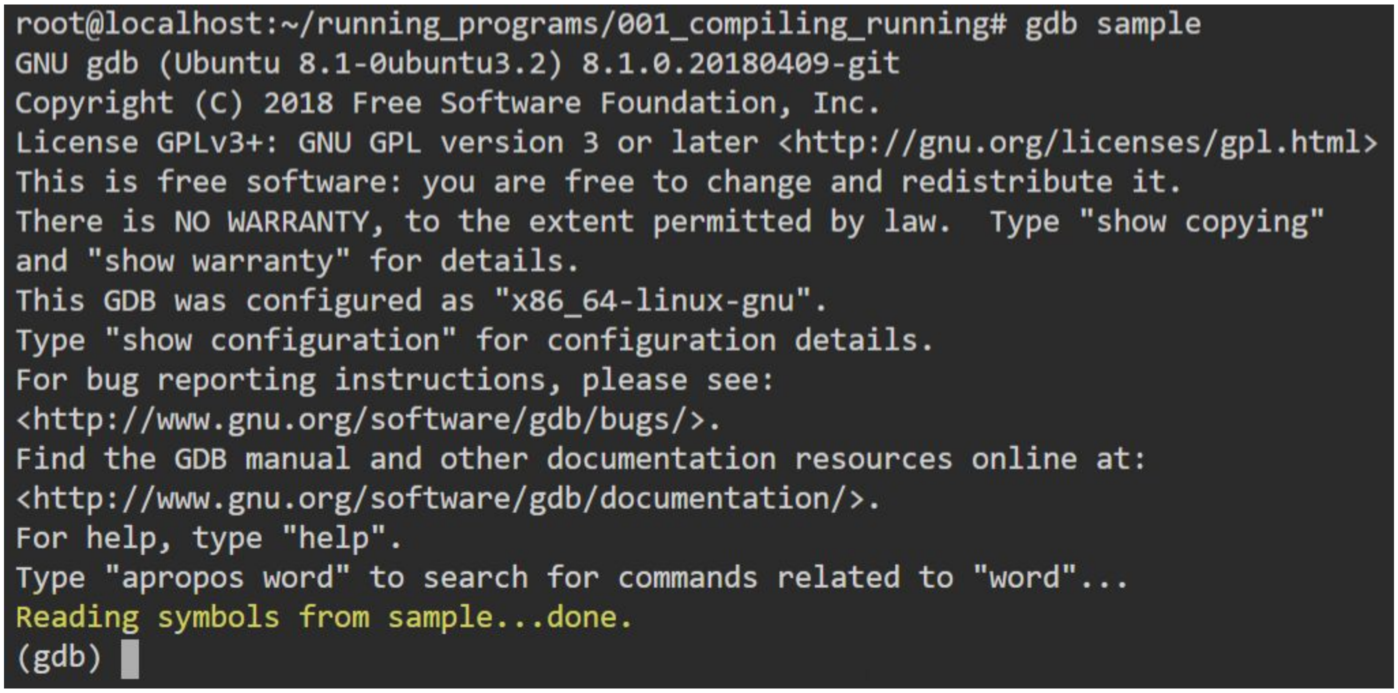
Exit from GDB using “q” key.



```
(gdb) q
root@localhost:~/running_programs/001_compiling_running#
```

Step 10: Start GDB and open sample_normal file in it.

Command: GDB sample

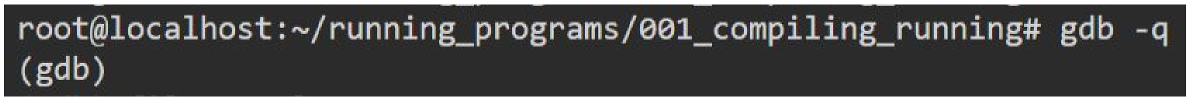


```
root@localhost:~/running_programs/001_compiling_running# gdb sample
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and 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 "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sample...done.
(gdb)
```

Alternatively, the GDB can be started first and then the binary can be opened from inside of GDB.

Start GDB in quiet mode (this skips the information that gets printed on the screen)

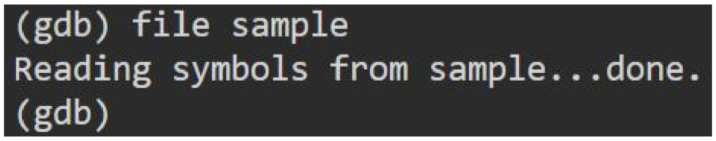
Command: gdb -q



```
root@localhost:~/running_programs/001_compiling_running# gdb -q
(gdb)
```

Step 11: Use file command to load the binary.

Command: file sample

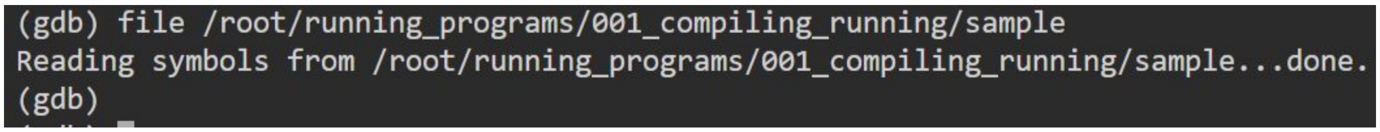


```
(gdb) file sample
Reading symbols from sample...done.
(gdb)
```

This is assuming that the file is present in the current working directory of GDB.

Step 12: The absolute path can also be used.

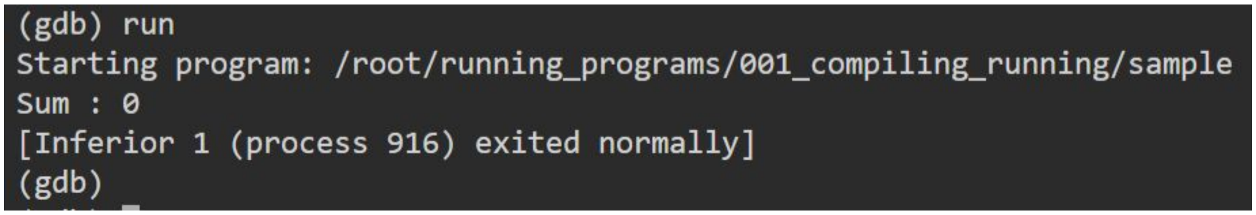
Command: file /root/running_programs/001_compiling_running/sample



```
(gdb) file /root/running_programs/001_compiling_running/sample
Reading symbols from /root/running_programs/001_compiling_running/sample...done.
(gdb)
```

Step 13: Start the loaded program

Command: run

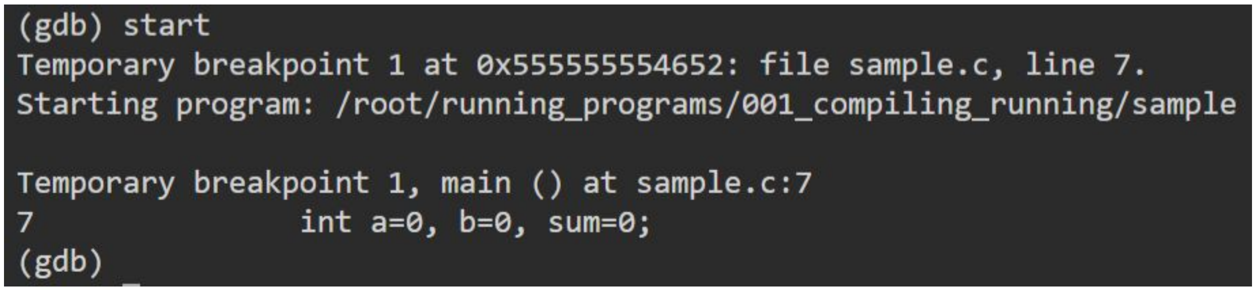


```
(gdb) run
Starting program: /root/running_programs/001_compiling_running/sample
Sum : 0
[Inferior 1 (process 916) exited normally]
(gdb)
```

Program executed successfully and exited.

Alternatively, the binary can also be executed using start command. This command sets a temporary breakpoint at the beginning of the main procedure and then invokes the run command.

Commad: start

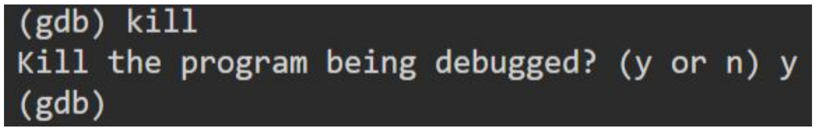


```
(gdb) start
Temporary breakpoint 1 at 0x555555554652: file sample.c, line 7.
Starting program: /root/running_programs/001_compiling_running/sample

Temporary breakpoint 1, main () at sample.c:7
7               int a=0, b=0, sum=0;
(gdb)
```

As one can observe, the breakpoint is set on the first line of main() function.

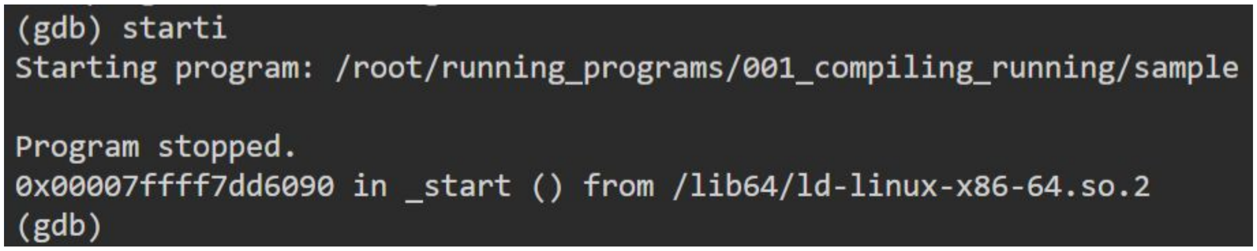
Step 14: The program is stopped at the checkpoint. The kill command can be used to kill this process.

Command: kill


```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb)
```

Another method of running the binary is using starti command. This command sets a temporary breakpoint at the first instruction of program's execution and then invokes the run command

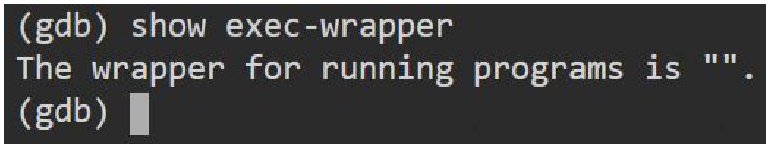
Command: starti


```
(gdb) starti
Starting program: /root/running_programs/001_compiling_running/sample

Program stopped.
0x00007ffff7dd6090 in _start () from /lib64/ld-linux-x86-64.so.2
(gdb)
```

In this case, the breakpoint was set on a library function and not the first line of main() function.

Step 15: Check the execution wrapper. An execution wrapper is used to launch programs for debugging. First the wrapper runs and then it passes the control to program being debugged.

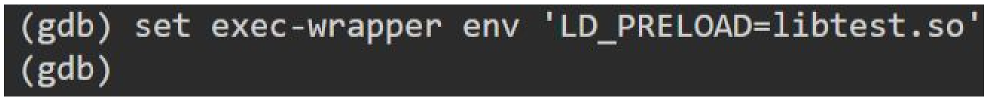
Command: show exec-wrapper


```
(gdb) show exec-wrapper
The wrapper for running programs is "".
(gdb)
```

Currently exec-wrapper is not set.

Step 16: Set the execution wrapper to pass an environment variable to the program. There are also other ways to pass environment variables and we will discuss those later.

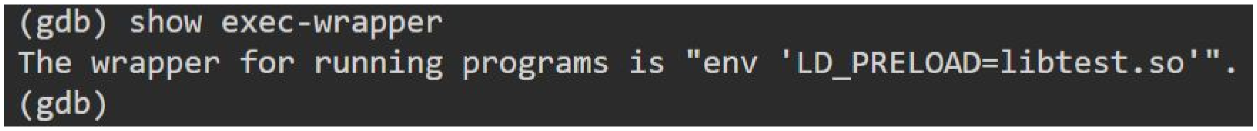
Command: set exec-wrapper env 'LD_PRELOAD=libtest.so'



```
(gdb) set exec-wrapper env 'LD_PRELOAD=libtest.so'
(gdb)
```

Once the wrapper is set, it can be checked

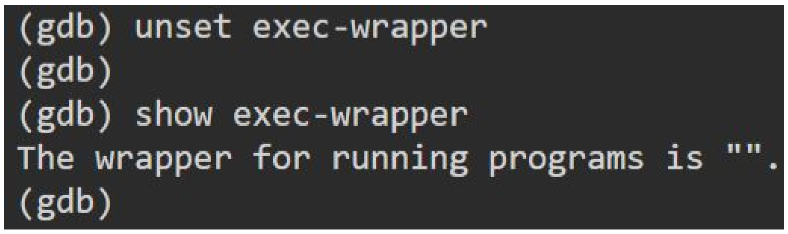
Command: show exec-wrapper



```
(gdb) show exec-wrapper
The wrapper for running programs is "env 'LD_PRELOAD=libtest.so'".
(gdb)
```

Step 17: The exec-mapper can be unset if not needed anymore.

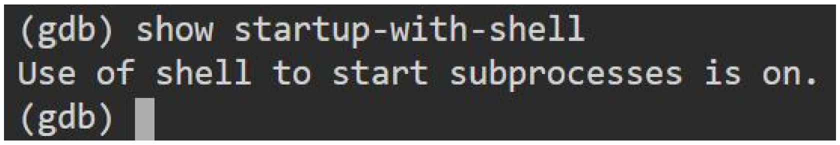
Commands:
unset exec-wrapper
show exec-wrapper



```
(gdb) unset exec-wrapper
(gdb)
(gdb) show exec-wrapper
The wrapper for running programs is "".
(gdb)
```

Step 18: The startup-with-shell option is on by default which makes our program run with the shell.

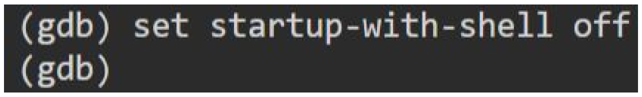
Command: show startup-with-shell



```
(gdb) show startup-with-shell
Use of shell to start subprocesses is on.
(gdb)
```

It can be disabled when the user needs to debug the shell or startup failures of the program.

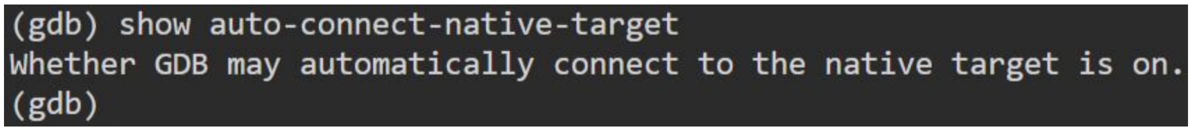
Command: set startup-with-shell off


```
(gdb) set startup-with-shell off
(gdb)
```

Step 19: GDB represents the state of each program execution with an object called an inferior. To run any program GDB has to be either connected to a remote machine or it has to start the program on the local machine as a child process of GDB.

Check the current value of auto-connect-native-target

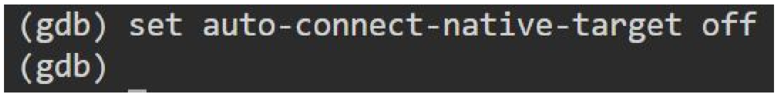
Command: show auto-connect-native-target


```
(gdb) show auto-connect-native-target
Whether GDB may automatically connect to the native target is on.
(gdb)
```

This means any program that is run on GDB will run on a local machine.

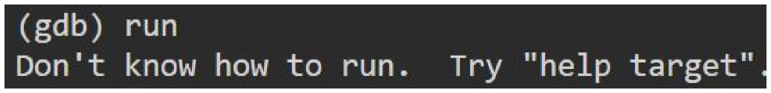
Step 20: This setting can be turned off when the user wants to use a remote machine.

Command: set auto-connect-native-target off


```
(gdb) set auto-connect-native-target off
(gdb) _
```

Now, if the user tries to run the program on it without defining the target machine, it will fail as GDB doesn't know where to run it.

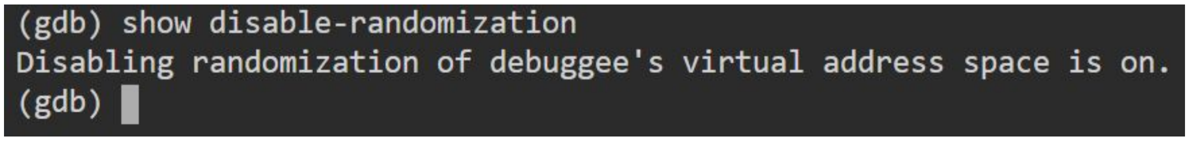
Command: run


```
(gdb) run
Don't know how to run.  Try "help target".
```

Step 21: Address-space layout randomization (ASLR) is a technique to make exploits harder by placing various objects at random rather than at fixed locations. GDB has an option to turn off this randomization to make analysis easy.

Check the status of the disable-randomization parameter. It is enabled by default.

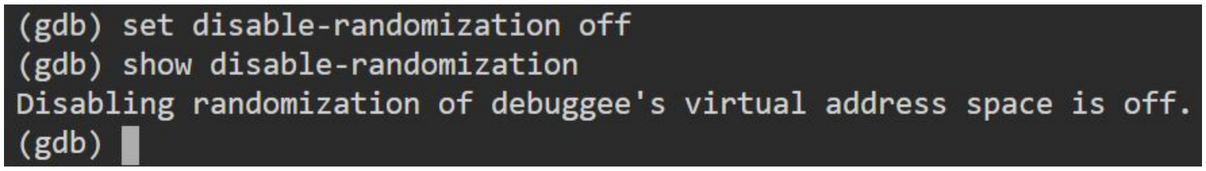
Command: show disable-randomization



```
(gdb) show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
(gdb) 
```

In some cases, the behavior of interest (e.g. error, exploit condition, crash) only works on address randomization. In those cases, address randomization can be disabled.

Commands:
set disable-randomization off
show disable-randomization



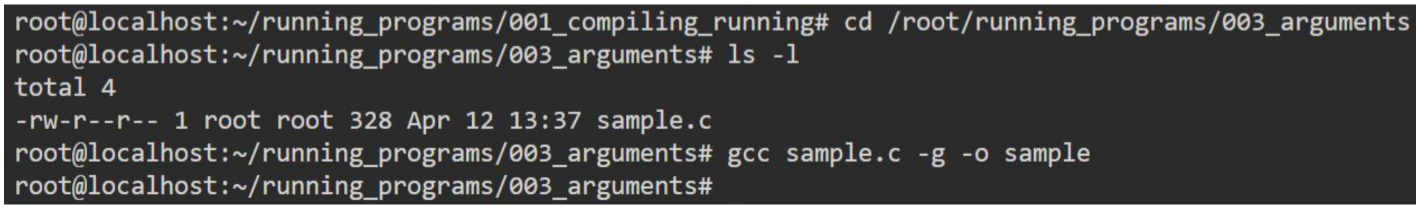
```
(gdb) set disable-randomization off
(gdb) show disable-randomization
Disabling randomization of debuggee's virtual address space is off.
(gdb) 
```

<u>Passing Arguments</u>

Step 22: Some programs take arguments from the command line. GDB supports command passing using various methods.

To try these methods, switch to /root/running_programs/003_arguments directory and compile the file provided in that directory with debugging symbols.

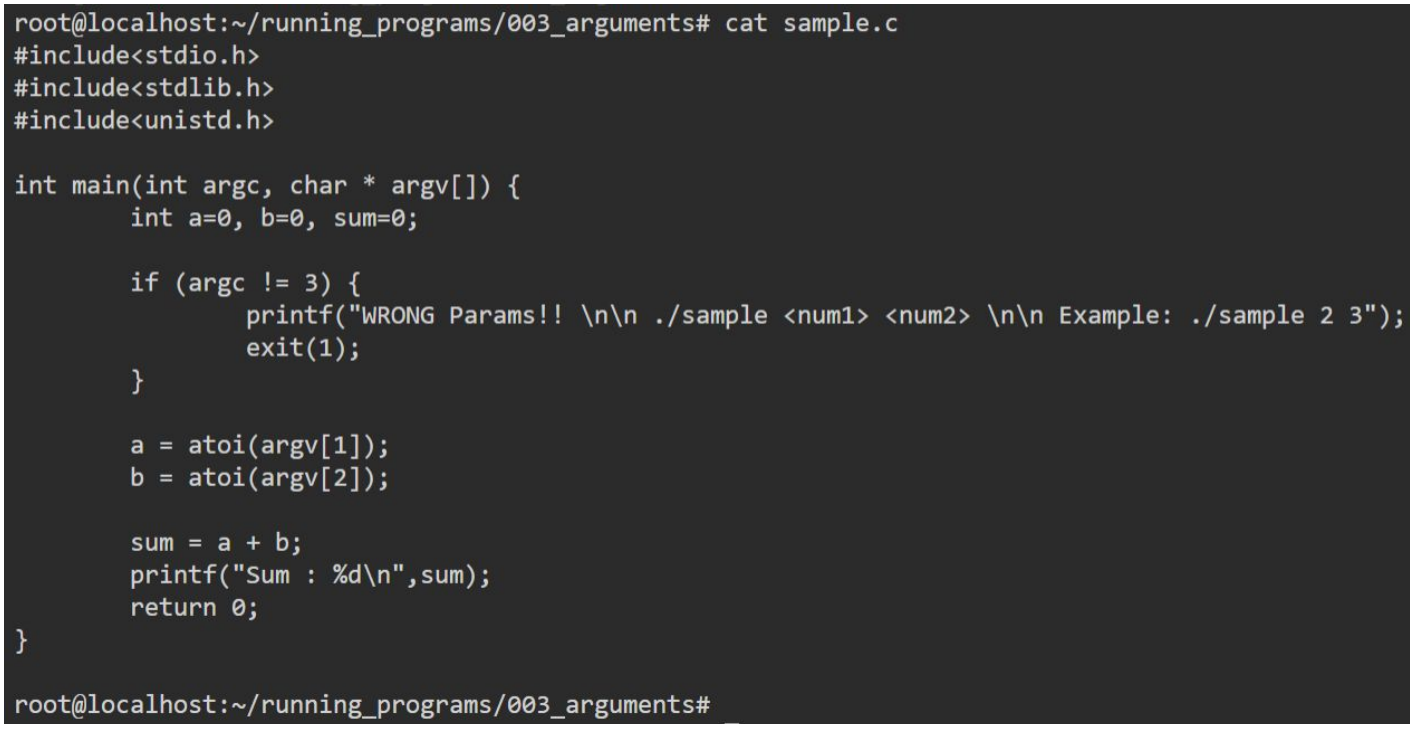
Commands:
cd /root/running_programs/003_arguments
ls -l
gcc sample.c -g -o sample



```
root@localhost:~/running_programs/001_compiling_running# cd /root/running_programs/003_arguments
root@localhost:~/running_programs/003_arguments# ls -l
total 4
-rw-r--r-- 1 root root 328 Apr 12 13:37 sample.c
root@localhost:~/running_programs/003_arguments# gcc sample.c -g -o sample
root@localhost:~/running_programs/003_arguments#
```

Step 23: Check the C file

Command: cat sample.c

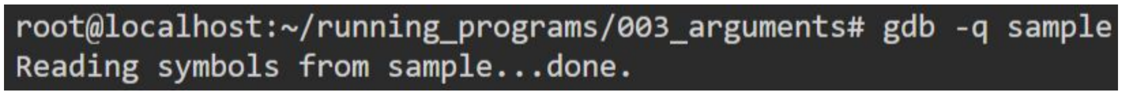


```
root@localhost:~/running_programs/003_arguments# cat sample.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main(int argc, char * argv[]) {
        int a=0, b=0, sum=0;

        if (argc != 3) {
                printf("WRONG Params!! \n\n ./sample <num1> <num2> \n\n Example: ./sample 2 3");
                exit(1);
        }

        a = atoi(argv[1]);
        b = atoi(argv[2]);

        sum = a + b;
        printf("Sum : %d\n",sum);
        return 0;
}

root@localhost:~/running_programs/003_arguments#
```

One can observe that the program takes two arguments and returns the sum of both.

Step 24: Start GDB with this binary.

Command: gdb -q sample

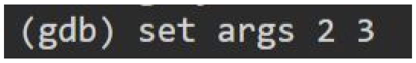


```
root@localhost:~/running_programs/003_arguments# gdb -q sample
Reading symbols from sample...done.
```

Step 25: Set the arguments for to run this program

Method 1: Setting arguments in GDB

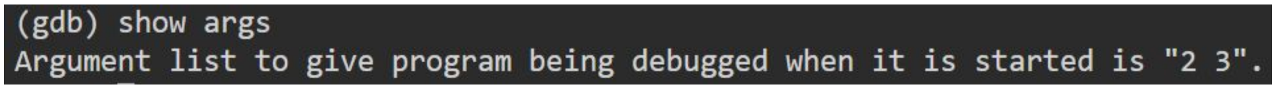
Command: set args 2 3



```
(gdb) set args 2 3
```

The same can be verified

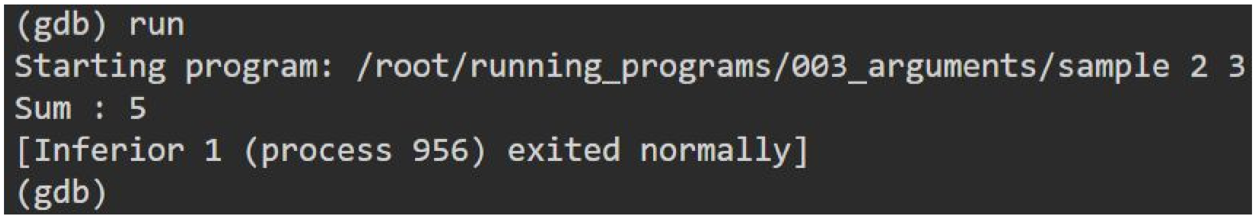
Command: show args



```
(gdb) show args
Argument list to give program being debugged when it is started is "2 3".
```

Run the program

Command: run

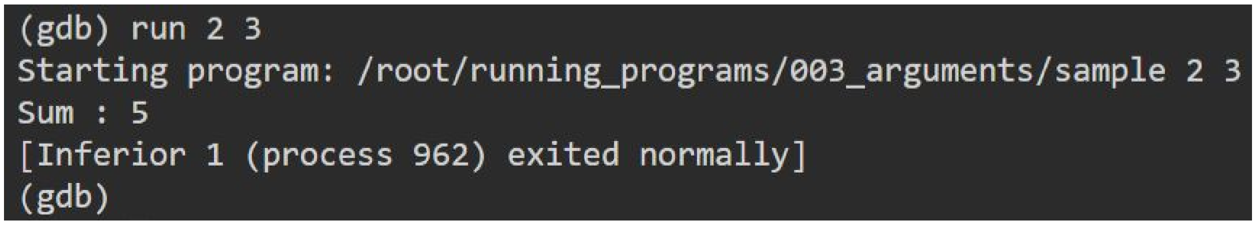


```
(gdb) run
Starting program: /root/running_programs/003_arguments/sample 2 3
Sum : 5
[Inferior 1 (process 956) exited normally]
(gdb)
```

The arguments were passed and the program executed successfully.

Method 2: Arguments can also be passed to run directly

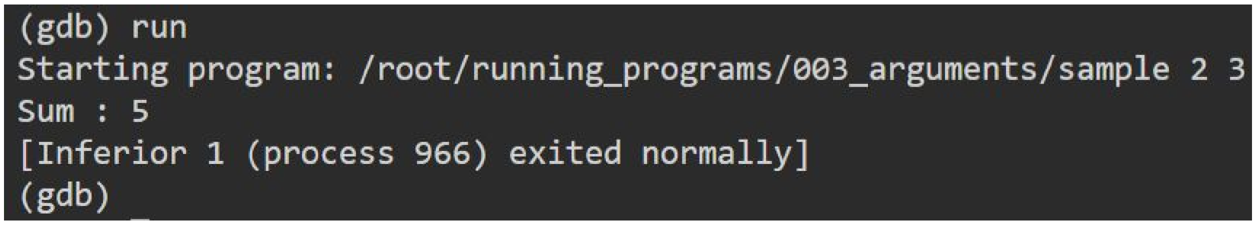
Command: run 2 3



```
(gdb) run 2 3
Starting program: /root/running_programs/003_arguments/sample 2 3
Sum : 5
[Inferior 1 (process 962) exited normally]
(gdb)
```

For all future calls to run automatically uses the same arguments (till the time args are explicitly changed or GDB is restarted)

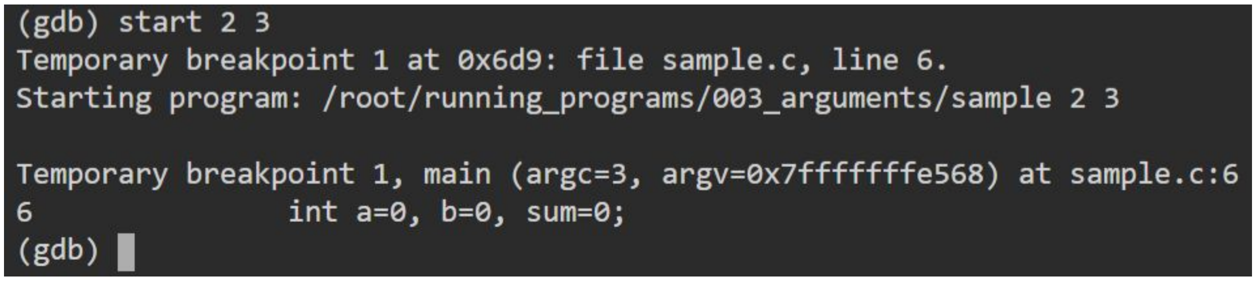
Command: run



```
(gdb) run
Starting program: /root/running_programs/003_arguments/sample 2 3
Sum : 5
[Inferior 1 (process 966) exited normally]
(gdb)
```

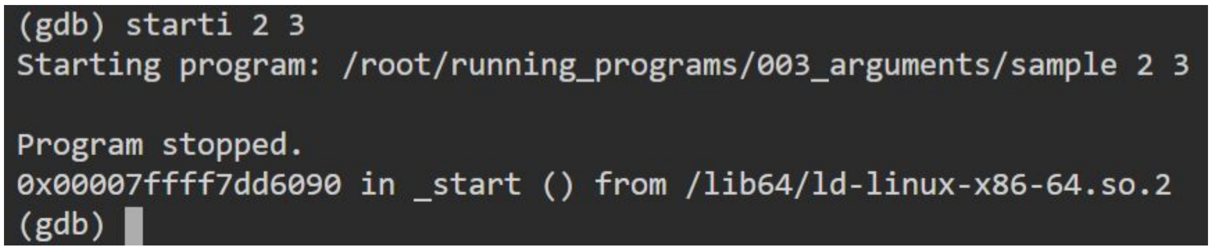
Similarly, the arguments can be passed to start and starti commands.

Command: start 2 3



```
(gdb) start 2 3
Temporary breakpoint 1 at 0x6d9: file sample.c, line 6.
Starting program: /root/running_programs/003_arguments/sample 2 3

Temporary breakpoint 1, main (argc=3, argv=0x7fffffffe568) at sample.c:6
6               int a=0, b=0, sum=0;
(gdb)
```

Command: starti 2 3



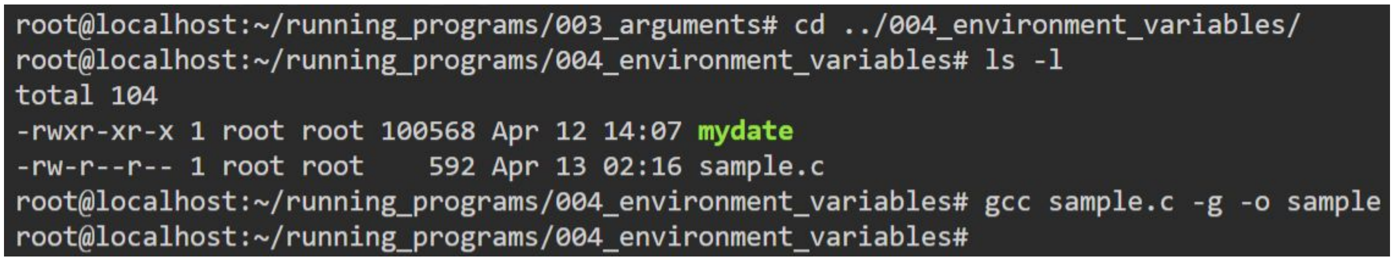
```
(gdb) starti 2 3
Starting program: /root/running_programs/003_arguments/sample 2 3

Program stopped.
0x00007ffff7dd6090 in _start () from /lib64/ld-linux-x86-64.so.2
(gdb)
```

<u>Program's Environment variables</u>

Step 26: Some programs call custom binaries and use environment variables in their operation. And, GDB provides a way to define these for programs under analysis.

To try these methods, switch to /root/running_programs/004_environment_variables directory and compile the file provided in that directory with debugging symbols.

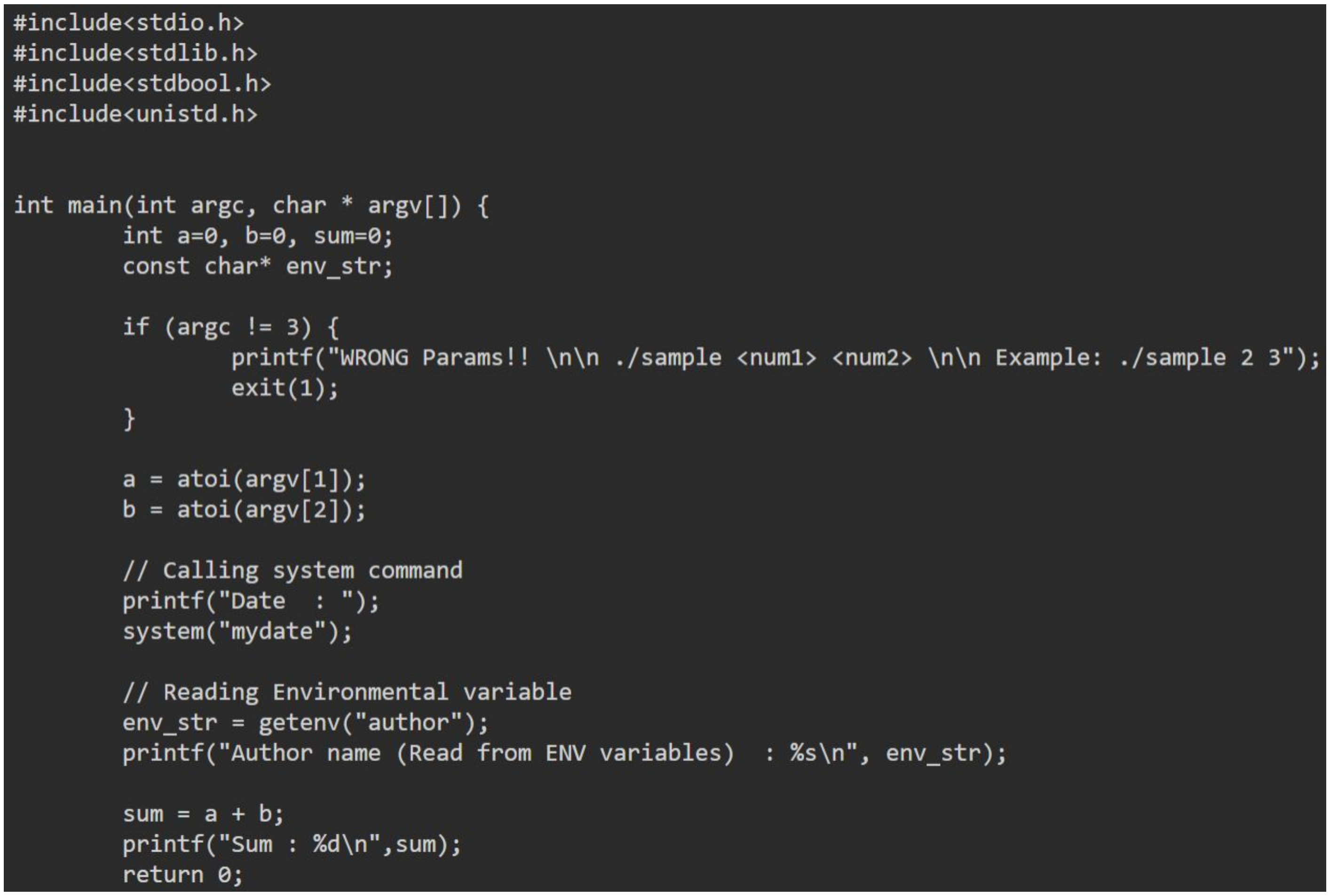
Commands:
cd /root/running_programs/004_environment_variables
ls -l
gcc sample.c -g -o sample



```
root@localhost:~/running_programs/003_arguments# cd ../004_environment_variables/
root@localhost:~/running_programs/004_environment_variables# ls -l
total 104
-rwxr-xr-x 1 root root 100568 Apr 12 14:07 mydate
-rw-r--r-- 1 root root    592 Apr 13 02:16 sample.c
root@localhost:~/running_programs/004_environment_variables# gcc sample.c -g -o sample
root@localhost:~/running_programs/004_environment_variables#
```

Step 27: Check the C file

Command: cat sample.c

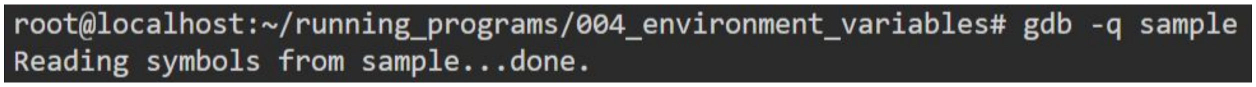


```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<unistd.h>


int main(int argc, char * argv[]) {
        int a=0, b=0, sum=0;
        const char* env_str;

        if (argc != 3) {
                printf("WRONG Params!! \n\n ./sample <num1> <num2> \n\n Example: ./sample 2 3");
                exit(1);
        }

        a = atoi(argv[1]);
        b = atoi(argv[2]);

        // Calling system command
        printf("Date  : ");
        system("mydate");

        // Reading Environmental variable
        env_str = getenv("author");
        printf("Author name (Read from ENV variables)  : %s\n", env_str);

        sum = a + b;
        printf("Sum : %d\n",sum);
        return 0;
```

One can observe that the program takes two arguments and returns the sum of both. It also calls a binary "mydate" (which is present in the same directory and uses environment variable "author".

Step 28: Start GDB with this binary.

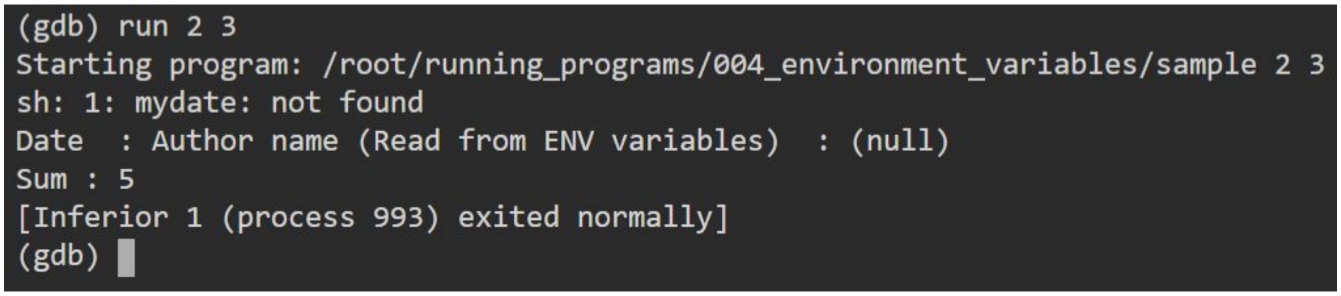
Command: gdb -q sample



```
root@localhost:~/running_programs/004_environment_variables# gdb -q sample
Reading symbols from sample...done.
```

Step 29: Run the binary.

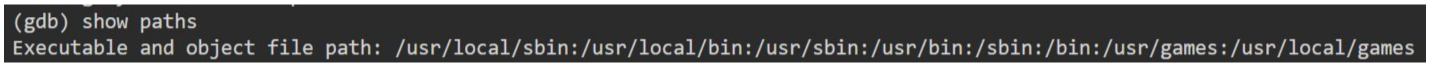
Command: run 2 3


```
(gdb) run 2 3
Starting program: /root/running_programs/004_environment_variables/sample 2 3
sh: 1: mydate: not found
Date  : Author name (Read from ENV variables)  : (null)
Sum : 5
[Inferior 1 (process 993) exited normally]
(gdb) 
```

The program is not able to find mydate binary. To fix this problem, the path of present working directory needs to be added to path variable.

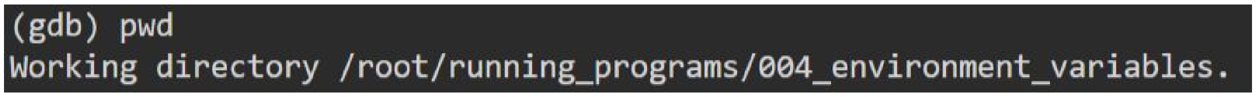
Step 30: Check the path variable (this variable is used to locate all binaries and commands)

Command: show paths


```
(gdb) show paths
Executable and object file path: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

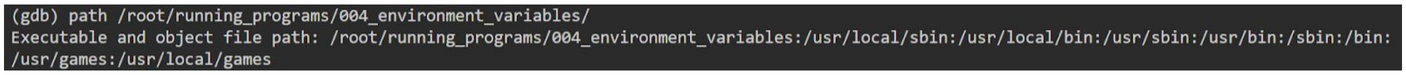
Step 31: Check the path of the present working directory.

Command: pwd


```
(gdb) pwd
Working directory /root/running_programs/004_environment_variables.
```

Step 32: Append the path of the present working directory to the path variable.

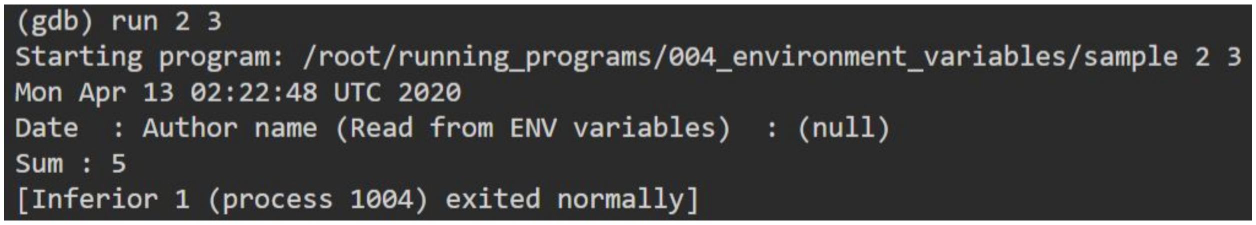
Command: path /root/running_programs/004_environment_variables/


```
(gdb) path /root/running_programs/004_environment_variables/
Executable and object file path: /root/running_programs/004_environment_variables:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
/usr/games:/usr/local/games
```

It is important to note that this change will only affect the running program and NOT the main system.

Step 33: Run the binary again to observe the difference.

Command: run 2 3

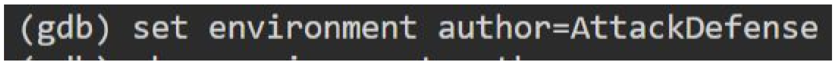


```
(gdb) run 2 3
Starting program: /root/running_programs/004_environment_variables/sample 2 3
Mon Apr 13 02:22:48 UTC 2020
Date   : Author name (Read from ENV variables)  : (null)
Sum : 5
[Inferior 1 (process 1004) exited normally]
```

The program was able to locate mydate now. However, it is still not able to read the environmental variable “author”.

Step 34: Define environment variable “author”.

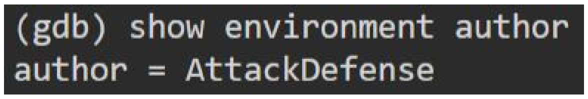
Command: set environment author=AttackDefense



```
(gdb) set environment author=AttackDefense
```

Step 35: Verify the change by checking the value of this variable.

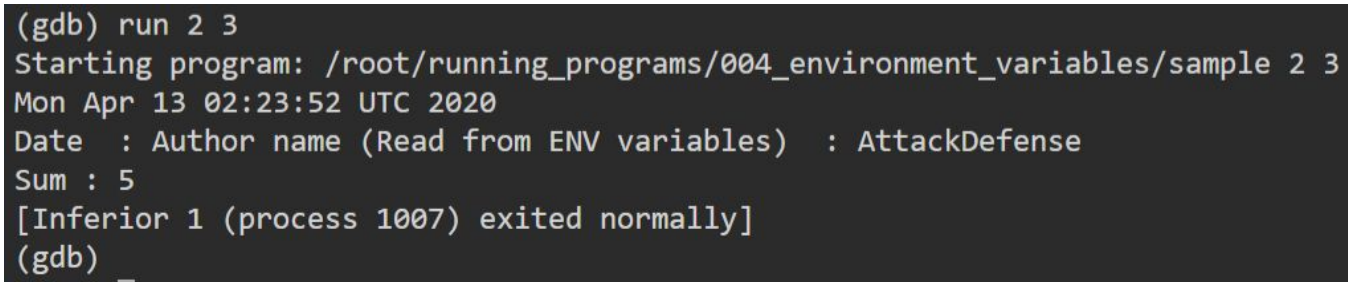
Command: show environment author



```
(gdb) show environment author
author = AttackDefense
```

Step 36: Run the binary again.

Command: run 2 3

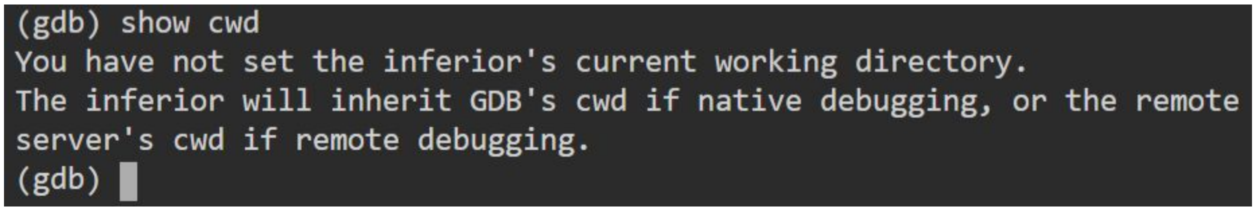


```
(gdb) run 2 3
Starting program: /root/running_programs/004_environment_variables/sample 2 3
Mon Apr 13 02:23:52 UTC 2020
Date   : Author name (Read from ENV variables)  : AttackDefense
Sum : 5
[Inferior 1 (process 1007) exited normally]
(gdb) _
```

This time the program was also able to read the environment variable.

Step 37: Check current working directory

Command: show cwd

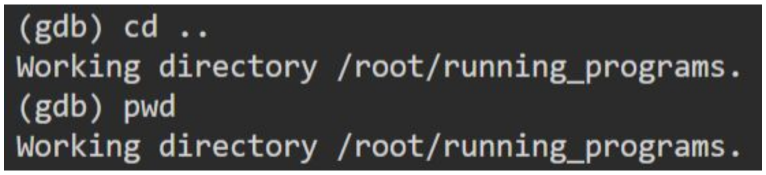


```
(gdb) show cwd
You have not set the inferior's current working directory.
The inferior will inherit GDB's cwd if native debugging, or the remote
server's cwd if remote debugging.
(gdb) 
```

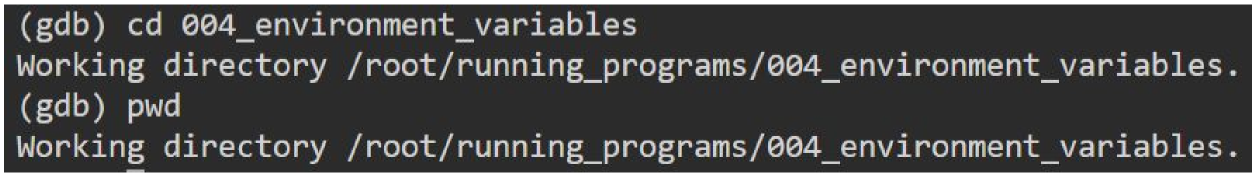
This setting makes more sense when dealing with remote machines for remote debugging. In case of a local machine, it automatically takes pwd.

Step 38: Change directory command works in GDB and can be used to switch directories.

Commands:
cd ..
pwd



```
(gdb) cd ..
Working directory /root/running_programs.
(gdb) pwd
Working directory /root/running_programs.
```



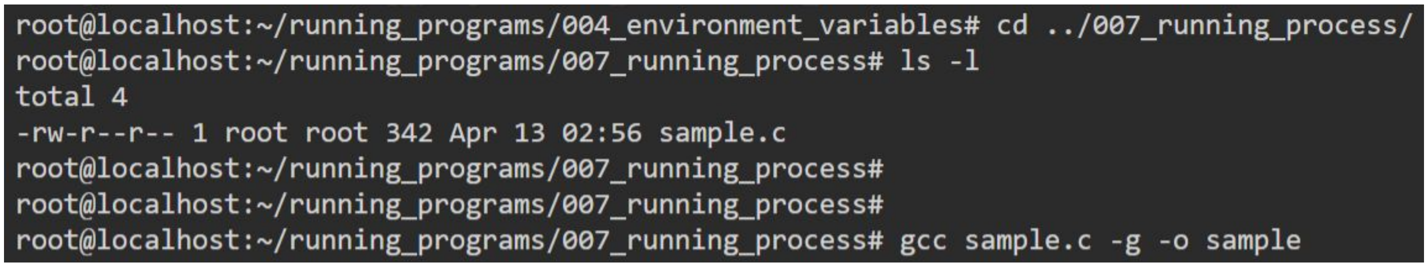
```
(gdb) cd 004_environment_variables
Working directory /root/running_programs/004_environment_variables.
(gdb) pwd
Working directory /root/running_programs/004_environment_variables.
```

Debugging Running Process

Step 39: GDB has capability to attach to the running process and debug it.

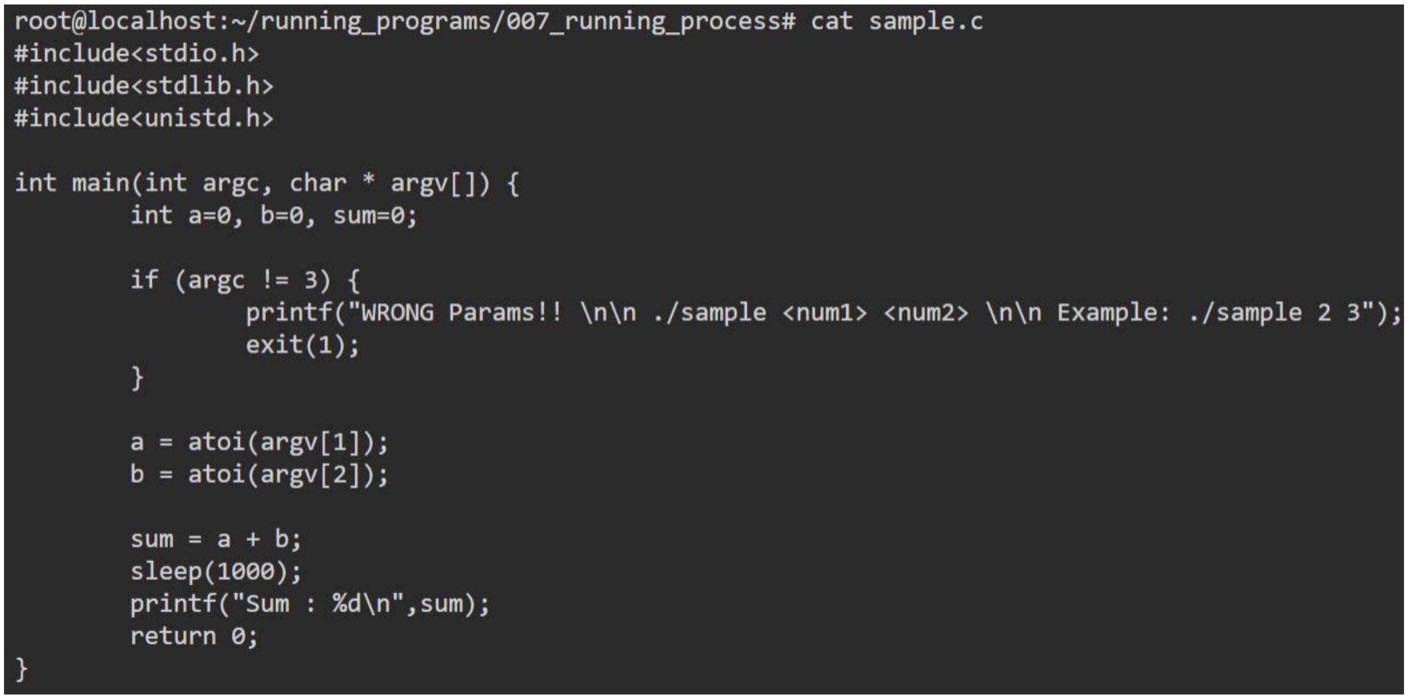
To try these methods, switch to /root/running_programs/007_running_process directory and compile the file provided in that directory with debugging symbols.

Commands:
cd /root/running_programs/007_running_process
ls -l
gcc sample.c -g -o sample


```
root@localhost:~/running_programs/004_environment_variables# cd ../007_running_process/
root@localhost:~/running_programs/007_running_process# ls -l
total 4
-rw-r--r-- 1 root root 342 Apr 13 02:56 sample.c
root@localhost:~/running_programs/007_running_process#
root@localhost:~/running_programs/007_running_process#
root@localhost:~/running_programs/007_running_process# gcc sample.c -g -o sample
```

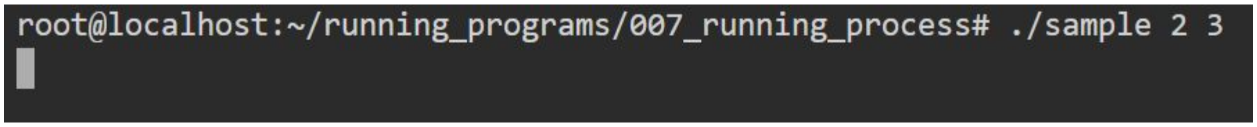
Step 40: Check the C file

Command: cat sample.c


```
root@localhost:~/running_programs/007_running_process# cat sample.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main(int argc, char * argv[]) {
        int a=0, b=0, sum=0;

        if (argc != 3) {
                printf("WRONG Params!! \n\n ./sample <num1> <num2> \n\n Example: ./sample 2 3");
                exit(1);
        }

        a = atoi(argv[1]);
        b = atoi(argv[2]);

        sum = a + b;
        sleep(1000);
        printf("Sum : %d\n",sum);
        return 0;
}
```

Step 41: Run the binary outside of GDB.

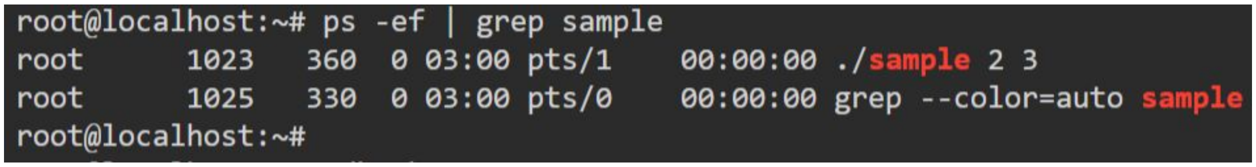
Command: ./sample



```
root@localhost:~/running_programs/007_running_process# ./sample 2 3
```

Step 42: Open a new tab (terminal) and find the PID of this process.

Command: ps -ef | grep sample

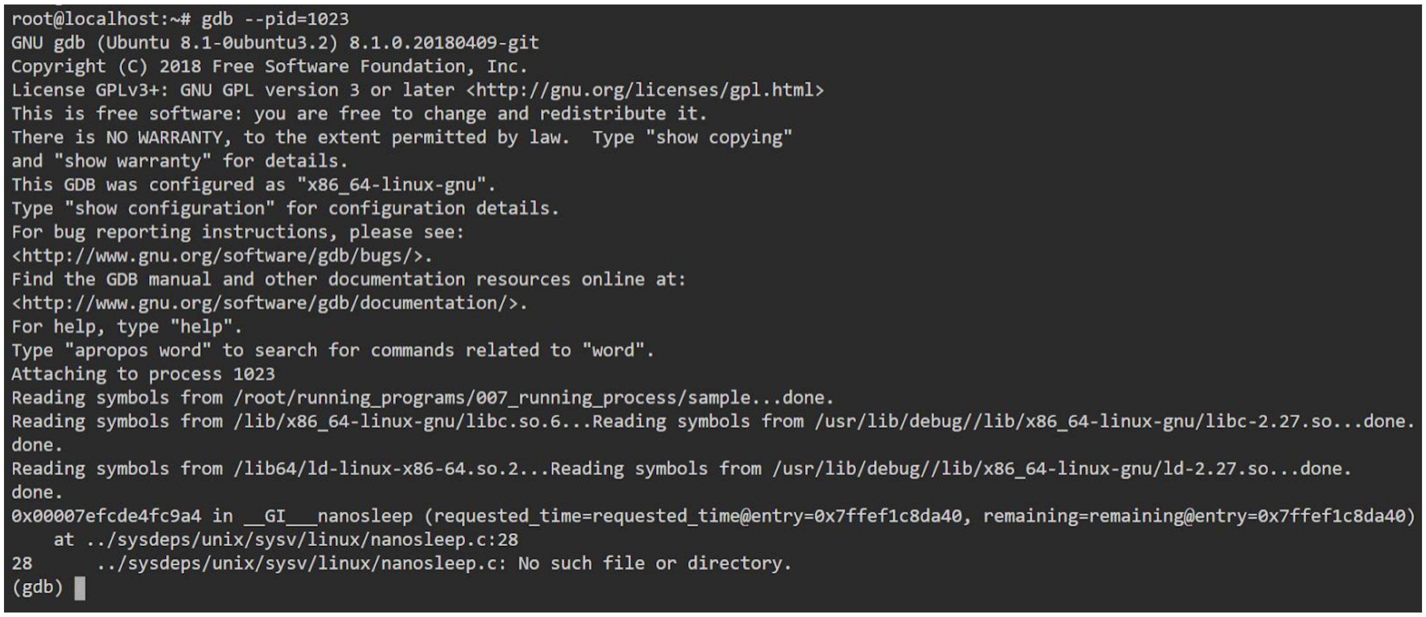


```
root@localhost:~# ps -ef | grep sample
root      1023   360  0 03:00 pts/1    00:00:00 ./sample 2 3
root      1025   330  0 03:00 pts/0    00:00:00 grep --color=auto sample
root@localhost:~#
```

Step 43: There are two ways to attach GDB to a process:

Method 1: PID can be passed as starting (command line) arguments to GDB.

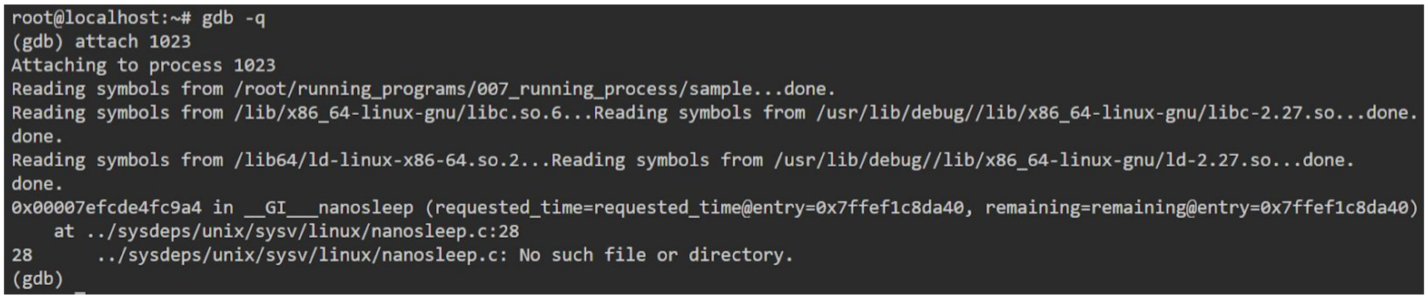
Command: gdb --pid=1023



```
root@localhost:~# gdb --pid=1023
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 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 "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Attaching to process 1023
Reading symbols from /root/running_programs/007_running_process/sample...done.
Reading symbols from /lib/x86_64-linux-gnu/libc.so.6...Reading symbols from /usr/lib/debug//lib/x86_64-linux-gnu/libc-2.27.so...done.
done.
Reading symbols from /lib64/ld-linux-x86-64.so.2...Reading symbols from /usr/lib/debug//lib/x86_64-linux-gnu/ld-2.27.so...done.
done.
0x00007efcde4fc9a4 in __GI___nanosleep (requested_time=requested_time@entry=0x7ffef1c8da40, remaining=remaining@entry=0x7ffef1c8da40)
    at ../sysdeps/unix/sysv/linux/nanosleep.c:28
28      ../sysdeps/unix/sysv/linux/nanosleep.c: No such file or directory.
(gdb) 
```

Method 2: The process can be attached to GDB from inside of GDB.

Command: attach 1023

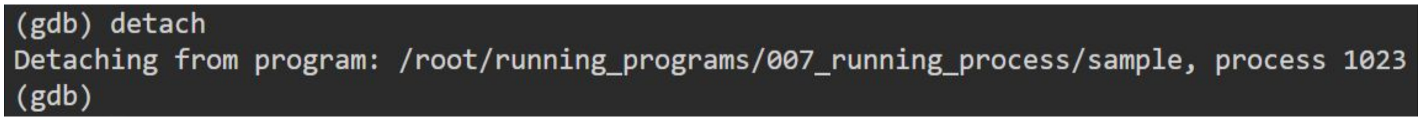


```
root@localhost:~# gdb -q
(gdb) attach 1023
Attaching to process 1023
Reading symbols from /root/running_programs/007_running_process/sample...done.
Reading symbols from /lib/x86_64-linux-gnu/libc.so.6...Reading symbols from /usr/lib/debug//lib/x86_64-linux-gnu/libc-2.27.so...done.
done.
Reading symbols from /lib64/ld-linux-x86-64.so.2...Reading symbols from /usr/lib/debug//lib/x86_64-linux-gnu/ld-2.27.so...done.
done.
0x00007efcde4fc9a4 in __GI___nanosleep (requested_time=requested_time@entry=0x7ffef1c8da40, remaining=remaining@entry=0x7ffef1c8da40)
    at ../sysdeps/unix/sysv/linux/nanosleep.c:28
28      ../sysdeps/unix/sysv/linux/nanosleep.c: No such file or directory.
(gdb) _
```

The process is attached to GDB now.

Step 44: In the same manner, to detach the process, one can simply exit the GDB using the ‘q’ option or can use the detach command.

Command: detach



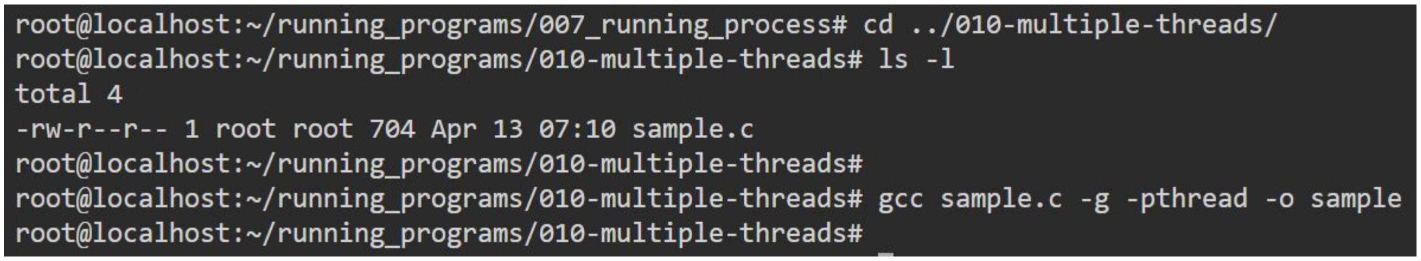
```
(gdb) detach
Detaching from program: /root/running_programs/007_running_process/sample, process 1023
(gdb)
```

<u>**Program with Multiple Threads**</u>

Step 45: GDB has capability to deal with multiple threads of a process.

To try these methods, switch to /root/running_programs/010-multiple-threads directory and compile the file provided in that directory with debugging symbols.

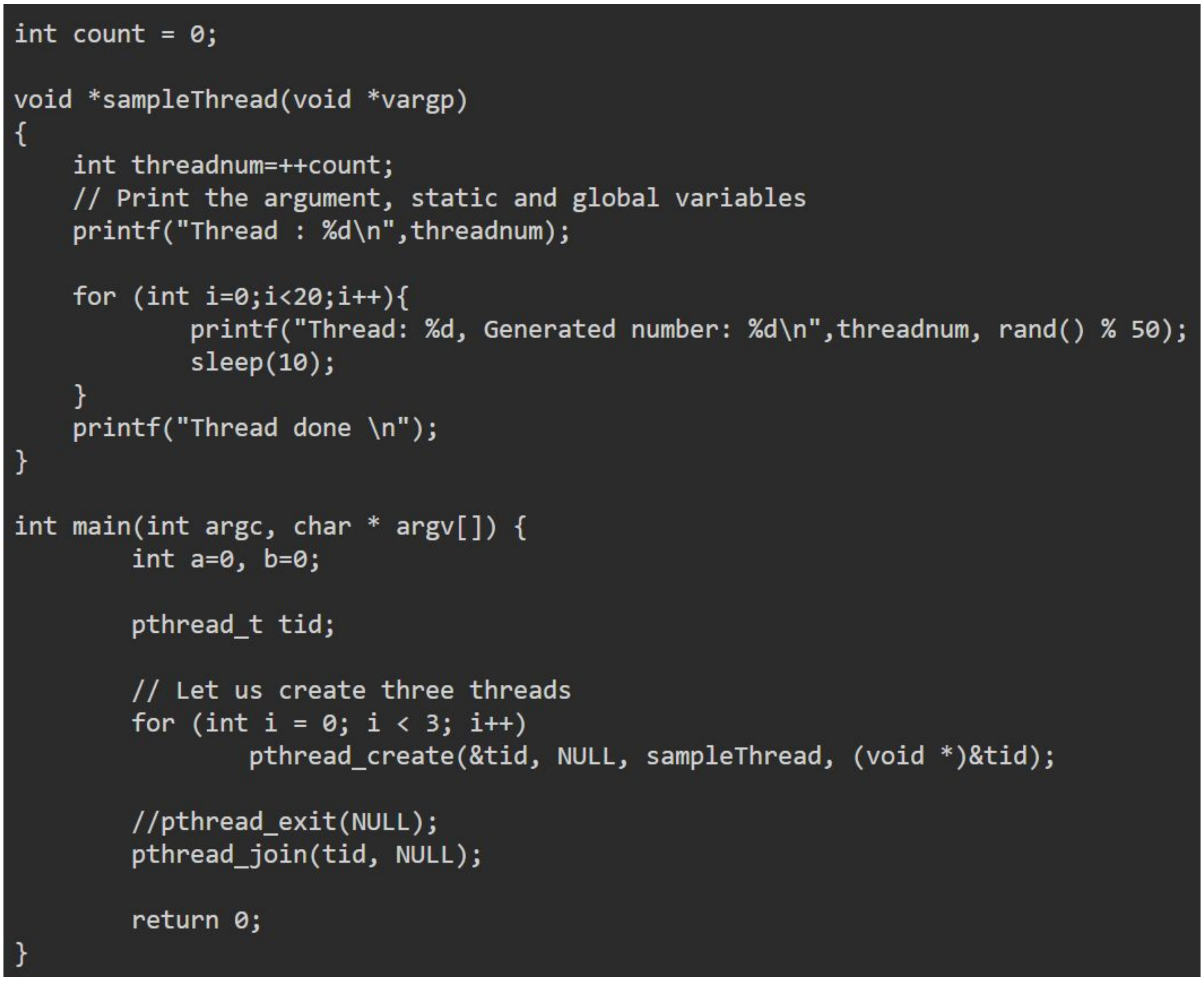
Commands:
cd /root/running_programs/010-multiple-threads
ls -l
gcc sample.c -g -pthread -o sample



```
root@localhost:~/running_programs/007_running_process# cd ../010-multiple-threads/
root@localhost:~/running_programs/010-multiple-threads# ls -l
total 4
-rw-r--r-- 1 root root 704 Apr 13 07:10 sample.c
root@localhost:~/running_programs/010-multiple-threads#
root@localhost:~/running_programs/010-multiple-threads# gcc sample.c -g -pthread -o sample
root@localhost:~/running_programs/010-multiple-threads#
```

Step 46: Check the C file

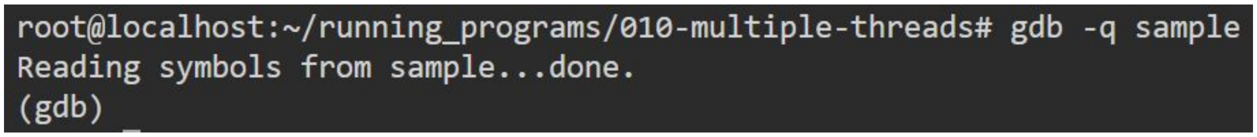
Command: cat sample.c



```
int count = 0;

void *sampleThread(void *vargp)
{
    int threadnum=++count;
    // Print the argument, static and global variables
    printf("Thread : %d\n",threadnum);

    for (int i=0;i<20;i++){
            printf("Thread: %d, Generated number: %d\n",threadnum, rand() % 50);
            sleep(10);
    }
    printf("Thread done \n");
}

int main(int argc, char * argv[]) {
        int a=0, b=0;

        pthread_t tid;

        // Let us create three threads
        for (int i = 0; i < 3; i++)
                pthread_create(&tid, NULL, sampleThread, (void *)&tid);

        //pthread_exit(NULL);
        pthread_join(tid, NULL);

        return 0;
}
```

Step 47: Run the GDB with sample binary

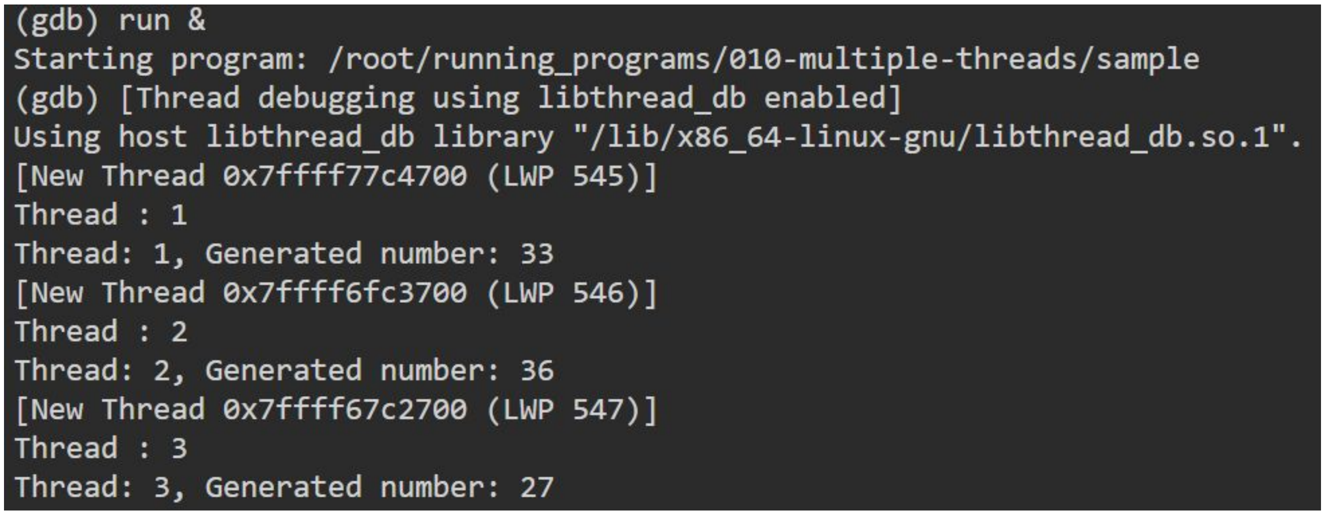
Command: gdb -q sample



```
root@localhost:~/running_programs/010-multiple-threads# gdb -q sample
Reading symbols from sample...done.
(gdb) _
```

Step 48: Run the program in the background to keep interacting with the command prompt. This can be accomplished by using & character.

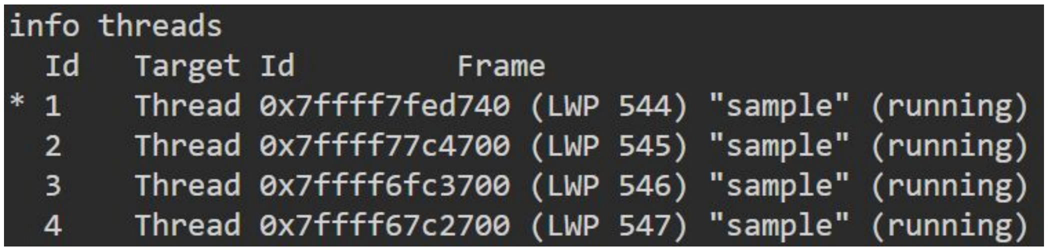
Command: run &



```
(gdb) run &
Starting program: /root/running_programs/010-multiple-threads/sample
(gdb) [Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff77c4700 (LWP 545)]
Thread : 1
Thread: 1, Generated number: 33
[New Thread 0x7ffff6fc3700 (LWP 546)]
Thread : 2
Thread: 2, Generated number: 36
[New Thread 0x7ffff67c2700 (LWP 547)]
Thread : 3
Thread: 3, Generated number: 27
```

Step 49: Total running threads can be checked using info threads command.

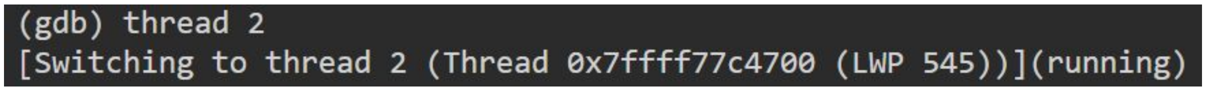
Command: info threads



```
info threads
  Id   Target Id         Frame
* 1    Thread 0x7ffff7fed740 (LWP 544) "sample" (running)
  2    Thread 0x7ffff77c4700 (LWP 545) "sample" (running)
  3    Thread 0x7ffff6fc3700 (LWP 546) "sample" (running)
  4    Thread 0x7ffff67c2700 (LWP 547) "sample" (running)
```

Step 50: The user can also select the thread to interact with by passing the thread id.

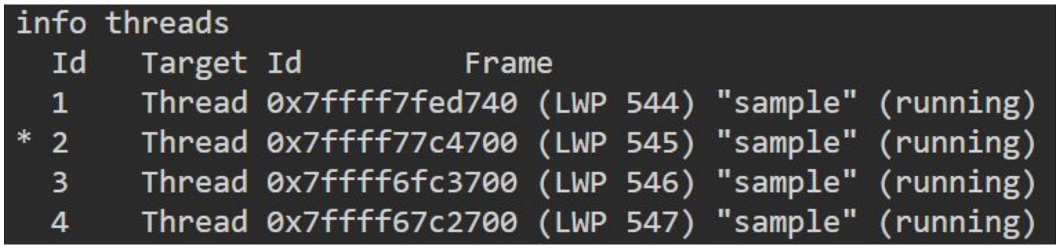
Command: thread 2



```
(gdb) thread 2
[Switching to thread 2 (Thread 0x7ffff77c4700 (LWP 545))](running)
```

Step 51: The currently selected thread is denoted by * (asterisk) character in front of thread entry.

Command: thread 2



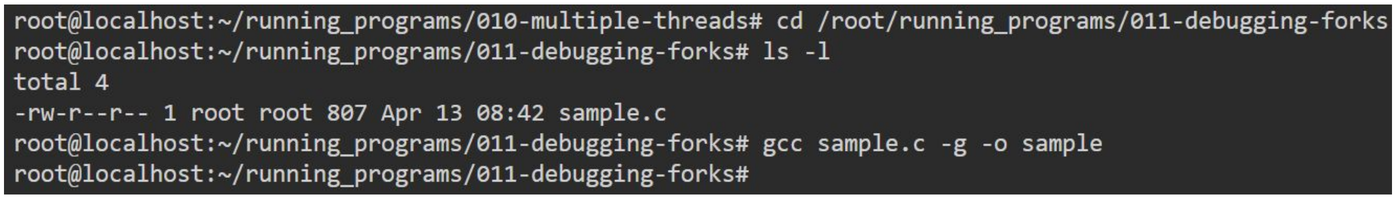
```
info threads
  Id   Target Id         Frame
  1    Thread 0x7ffff7fed740 (LWP 544) "sample" (running)
* 2    Thread 0x7ffff77c4700 (LWP 545) "sample" (running)
  3    Thread 0x7ffff6fc3700 (LWP 546) "sample" (running)
  4    Thread 0x7ffff67c2700 (LWP 547) "sample" (running)
```

Debugging Forks

Step 52: GDB has the capability to attach to the running process and debug it.

To try these methods, switch to /root/running_programs/011-debugging-forks directory and compile the file provided in that directory with debugging symbols.

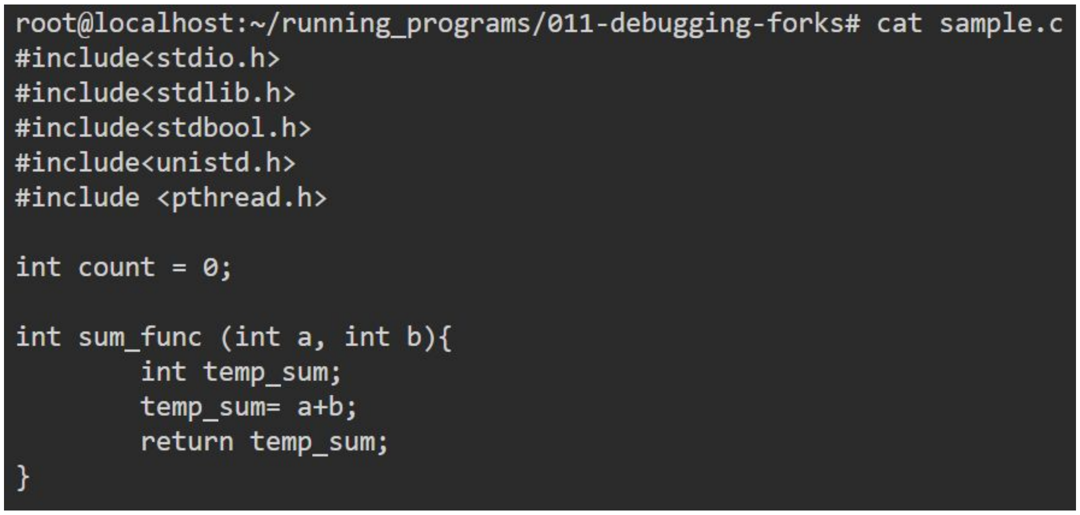
Commands:
cd /root/running_programs/011-debugging-forks
ls -l
gcc sample.c -g -o sample



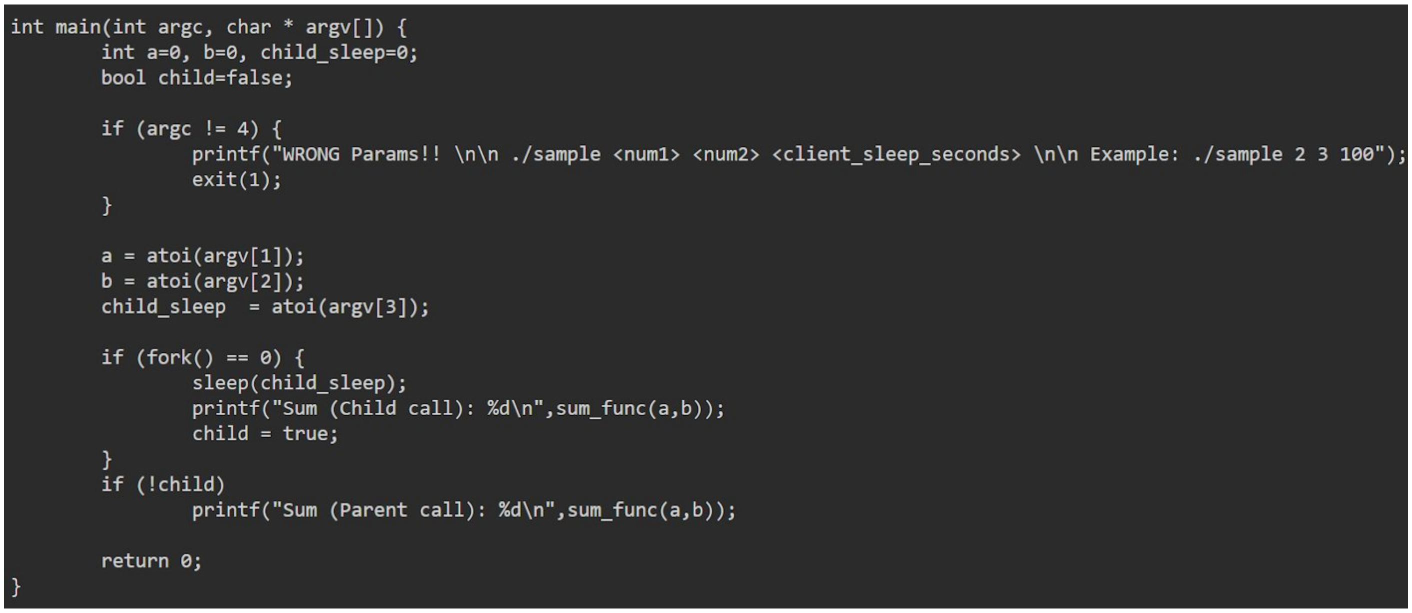
```
root@localhost:~/running_programs/010-multiple-threads# cd /root/running_programs/011-debugging-forks
root@localhost:~/running_programs/011-debugging-forks# ls -l
total 4
-rw-r--r-- 1 root root 807 Apr 13 08:42 sample.c
root@localhost:~/running_programs/011-debugging-forks# gcc sample.c -g -o sample
root@localhost:~/running_programs/011-debugging-forks#
```

Step 53: Check the C file

Command: cat sample.c



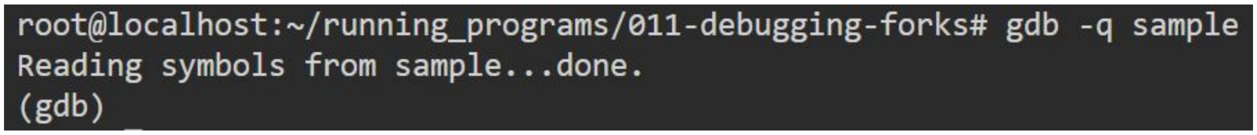
```
root@localhost:~/running_programs/011-debugging-forks# cat sample.c
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<unistd.h>
#include <pthread.h>

int count = 0;

int sum_func (int a, int b){
        int temp_sum;
        temp_sum= a+b;
        return temp_sum;
}
```



```
int main(int argc, char * argv[]) {
        int a=0, b=0, child_sleep=0;
        bool child=false;

        if (argc != 4) {
                printf("WRONG Params!! \n\n ./sample <num1> <num2> <client_sleep_seconds> \n\n Example: ./sample 2 3 100");
                exit(1);
        }

        a = atoi(argv[1]);
        b = atoi(argv[2]);
        child_sleep  = atoi(argv[3]);

        if (fork() == 0) {
                sleep(child_sleep);
                printf("Sum (Child call): %d\n",sum_func(a,b));
                child = true;
        }
        if (!child)
                printf("Sum (Parent call): %d\n",sum_func(a,b));

        return 0;
}
```

Step 54: Run the GDB with sample binary

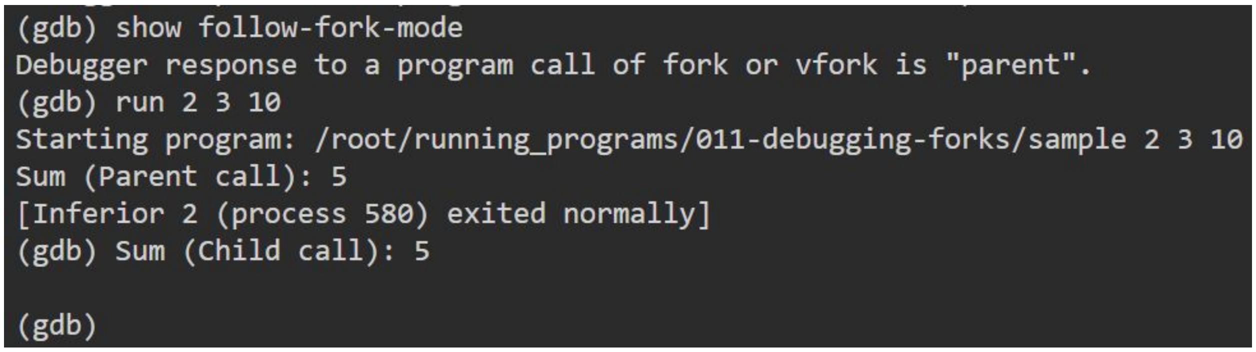
Command: gdb -q sample



```
root@localhost:~/running_programs/011-debugging-forks# gdb -q sample
Reading symbols from sample...done.
(gdb) _
```

Step 55: GDB has an option to choose whether to follow the child process or the parent process. The settings are set to “parent” by default. This program takes three aregiuments i.e. first number, second number and number of second the child process will sleep.

Commands:
show follow-fork-mode
run 2 3 10

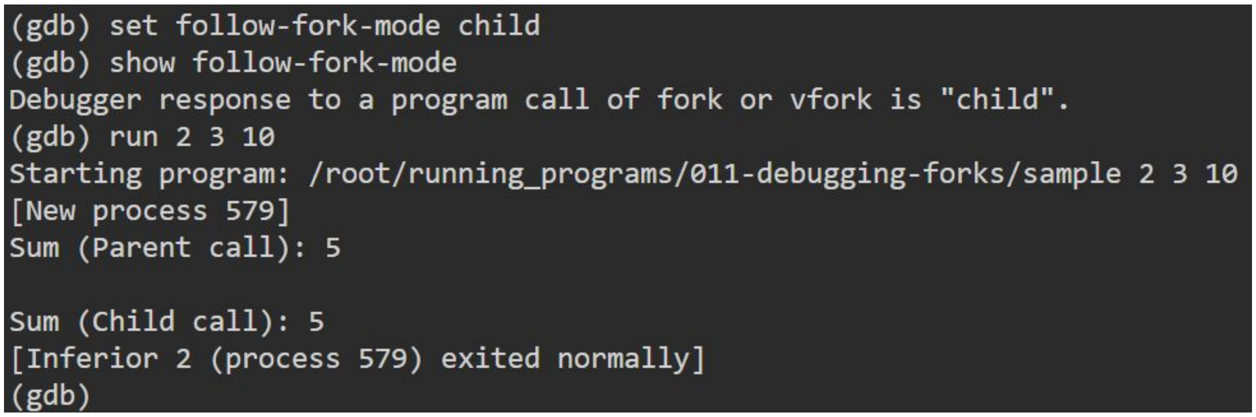


```
(gdb) show follow-fork-mode
Debugger response to a program call of fork or vfork is "parent".
(gdb) run 2 3 10
Starting program: /root/running_programs/011-debugging-forks/sample 2 3 10
Sum (Parent call): 5
[Inferior 2 (process 580) exited normally]
(gdb) Sum (Child call): 5

(gdb)
```

One can observe that as the parent process terminated, the prompt was available to user. And, the child process output came after the prompt.

Step 56: Change the setting to follow the child instead.

Commands:
set follow-fork-mode child
show follow-fork-mode
run 2 3 10



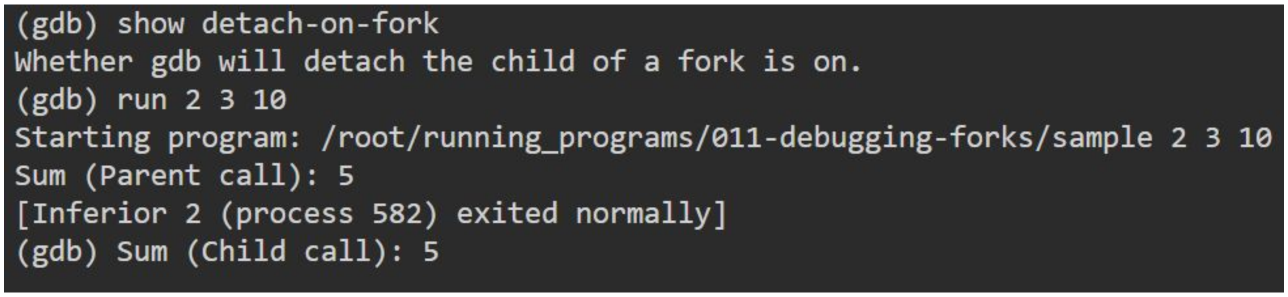
```
(gdb) set follow-fork-mode child
(gdb) show follow-fork-mode
Debugger response to a program call of fork or vfork is "child".
(gdb) run 2 3 10
Starting program: /root/running_programs/011-debugging-forks/sample 2 3 10
[New process 579]
Sum (Parent call): 5

Sum (Child call): 5
[Inferior 2 (process 579) exited normally]
(gdb)
```

This time the control was only returned after the termination of the child process.

Step 57: Another setting “detach-on-fork” which is “on” by default allows the child process to run detached.

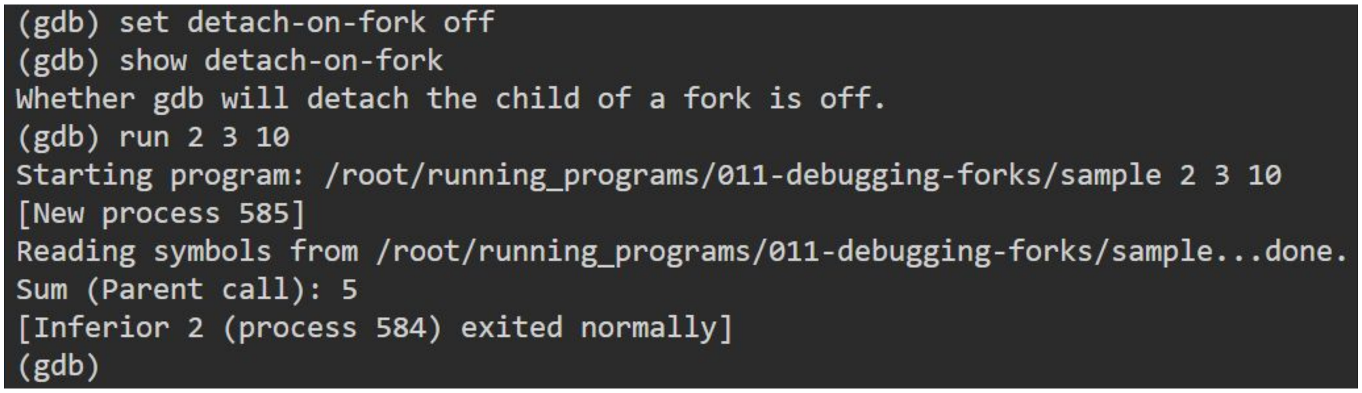
Check the status of the setting and check the difference between on and off state.

Commands:
show detach-on-fork
run 2 3 10



```
(gdb) show detach-on-fork
Whether gdb will detach the child of a fork is on.
(gdb) run 2 3 10
Starting program: /root/running_programs/011-debugging-forks/sample 2 3 10
Sum (Parent call): 5
[Inferior 2 (process 582) exited normally]
(gdb) Sum (Child call): 5
```

Commands:
set detach-on-fork off
show detach-on-fork
run 2 3 10



```
(gdb) set detach-on-fork off
(gdb) show detach-on-fork
Whether gdb will detach the child of a fork is off.
(gdb) run 2 3 10
Starting program: /root/running_programs/011-debugging-forks/sample 2 3 10
[New process 585]
Reading symbols from /root/running_programs/011-debugging-forks/sample...done.
Sum (Parent call): 5
[Inferior 2 (process 584) exited normally]
(gdb)
```

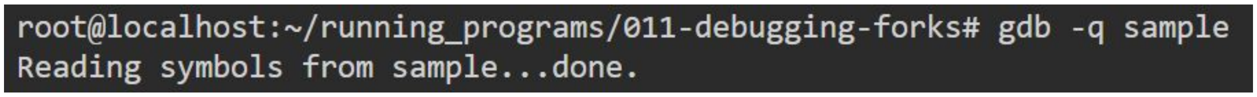
After comparing both outputs, we can observe that the child process was able to continue even after termination of the parent process in former but not in the latter.

Bookmark

GDB has the capability to “bookmark” a point in program execution and when needed it can revert the program execution state to this point. This is also similar to a virtual machine snapshot that can be reverted to by the user when needed.

Step 58: Start the GDB with the same binary.

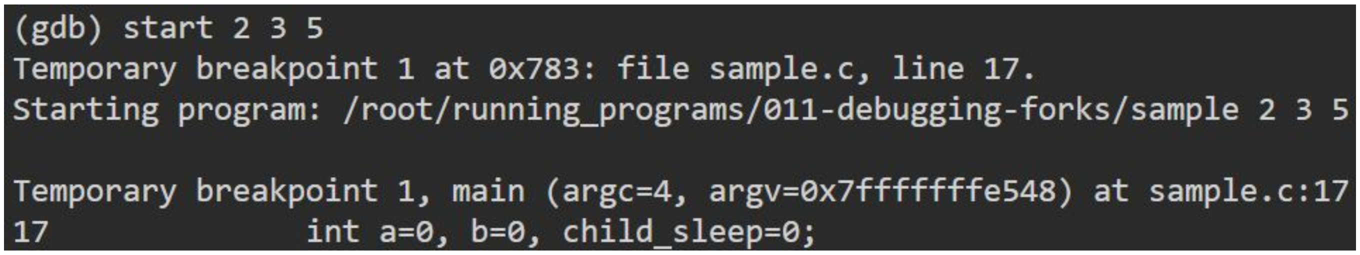
Command: gdb -q sample



```
root@localhost:~/running_programs/011-debugging-forks# gdb -q sample
Reading symbols from sample...done.
```

Step 60: Start the program.

Command: start 2 3 5

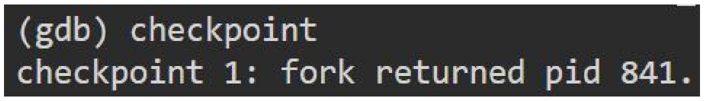


```
(gdb) start 2 3 5
Temporary breakpoint 1 at 0x783: file sample.c, line 17.
Starting program: /root/running_programs/011-debugging-forks/sample 2 3 5

Temporary breakpoint 1, main (argc=4, argv=0x7fffffffe548) at sample.c:17
17              int a=0, b=0, child_sleep=0;
```

As expected the program execution is stopped at the first instruction of the main() function.

Step 61: Set a checkpoint (or bookmark) here.

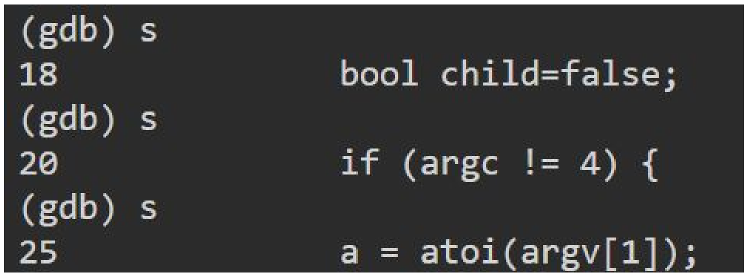
Command: checkpoint



```
(gdb) checkpoint
checkpoint 1: fork returned pid 841.
```

Step 62: Take three steps more using step command.

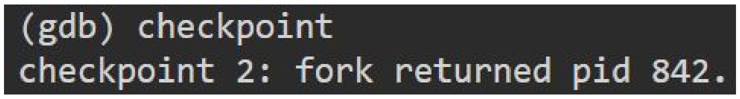
Command: s



```
(gdb) s
18              bool child=false;
(gdb) s
20              if (argc != 4) {
(gdb) s
25              a = atoi(argv[1]);
```

Step 63: Set another checkpoint (or bookmark) here.

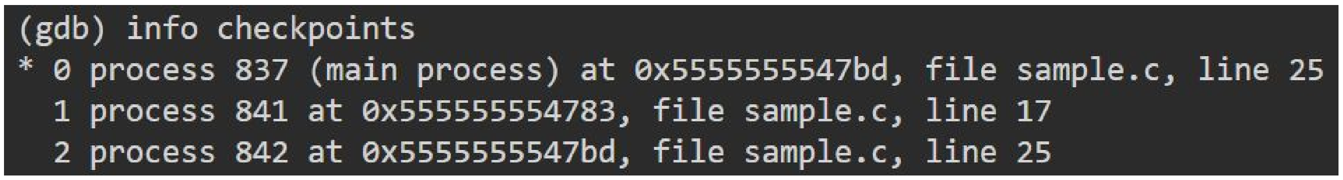
Command: checkpoint



```
(gdb) checkpoint
checkpoint 2: fork returned pid 842.
```

Step 64: List all checkpoints.

Command: info checkpoints

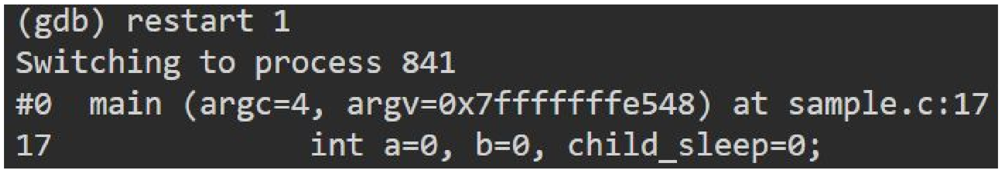


```
(gdb) info checkpoints
* 0 process 837 (main process) at 0x5555555547bd, file sample.c, line 25
  1 process 841 at 0x555555554783, file sample.c, line 17
  2 process 842 at 0x5555555547bd, file sample.c, line 25
```

Both of the checkpoints are listed here.

Step 65: Revert to the first checkpoint i.e. first line of the main function.

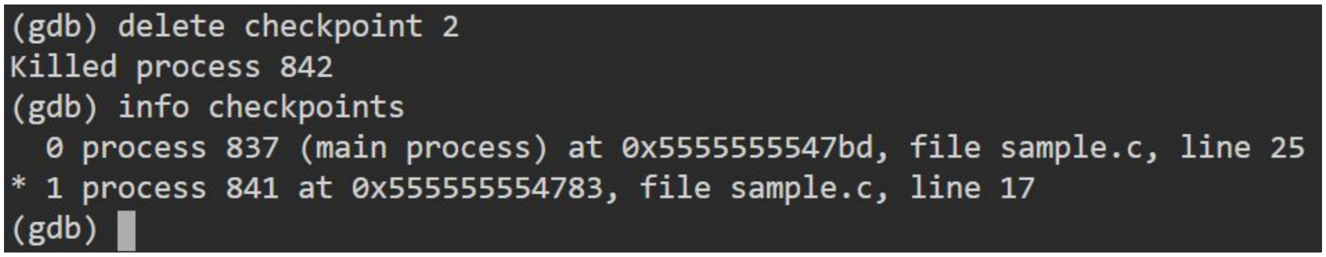
Command: restart 1


```
(gdb) restart 1
Switching to process 841
#0  main (argc=4, argv=0x7fffffffe548) at sample.c:17
17              int a=0, b=0, child_sleep=0;
```

And, the program execution rolled back to checkpoint 1.

Step 66: Checkpoint can be removed once the objective is complete.

Commands:
delete checkpoint 2
info checkpoints

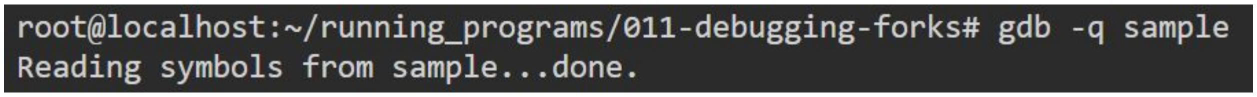

```
(gdb) delete checkpoint 2
Killed process 842
(gdb) info checkpoints
  0 process 837 (main process) at 0x5555555547bd, file sample.c, line 25
* 1 process 841 at 0x555555554783, file sample.c, line 17
(gdb) 
```

Multiple Inferiors Connections and Programs

GDB has the capability to run and debug multiple programs in the same session. GDB represents the state of each program execution with an object called an “inferior”. Each inferior corresponds to a program (or a copy of another inferior) and allows the user to select the inferior of his choice and switch among those.

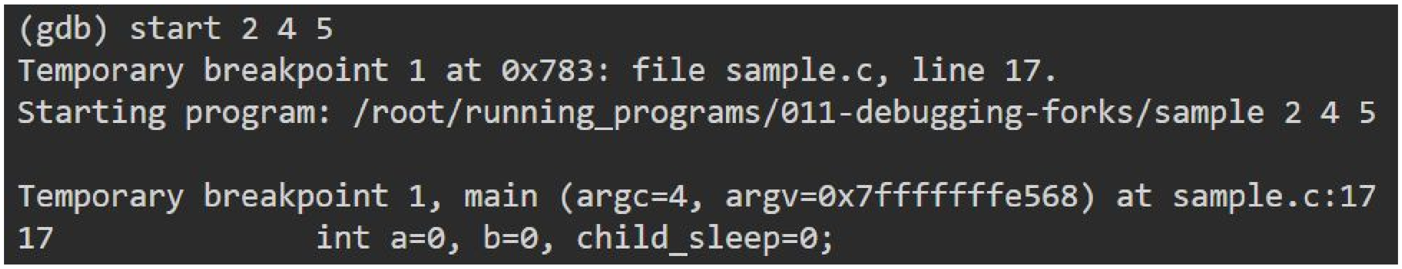
Step 69: Start the GDB with the same binary.

Command: gdb -q sample


```
root@localhost:~/running_programs/011-debugging-forks# gdb -q sample
Reading symbols from sample...done.
```

Step 70: Start the program.

Command: start 2 3 5

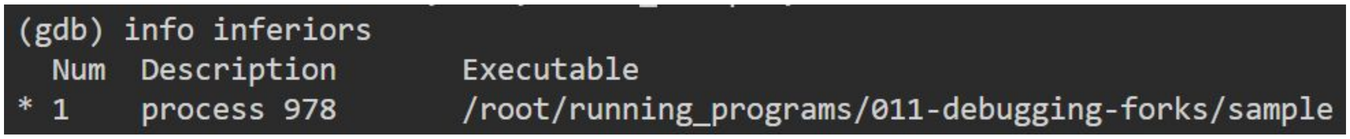


```
(gdb) start 2 4 5
Temporary breakpoint 1 at 0x783: file sample.c, line 17.
Starting program: /root/running_programs/011-debugging-forks/sample 2 4 5

Temporary breakpoint 1, main (argc=4, argv=0x7fffffffe568) at sample.c:17
17              int a=0, b=0, child_sleep=0;
```

As expected the program execution is stopped at the breakpoint i.e. the first line of the main() function.

Step 71: List all inferiors

Command: info inferiors

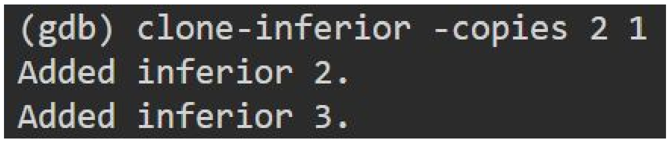


```
(gdb) info inferiors
  Num  Description       Executable
* 1    process 978       /root/running_programs/011-debugging-forks/sample
```

As currently one program is being executed, only one inferior is there.

Step 72: Clone the running inferior with infno 1 and make 2 additional copies of it.

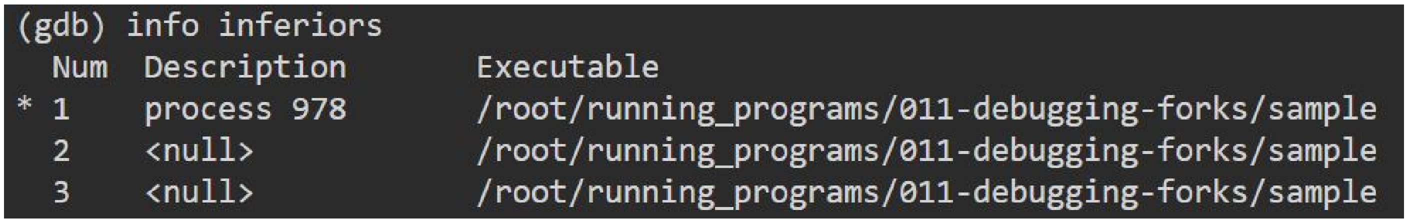
Command: clone-inferior -copies 2 1



```
(gdb) clone-inferior -copies 2 1
Added inferior 2.
Added inferior 3.
```

Check the inferiors again

Command: info inferiors

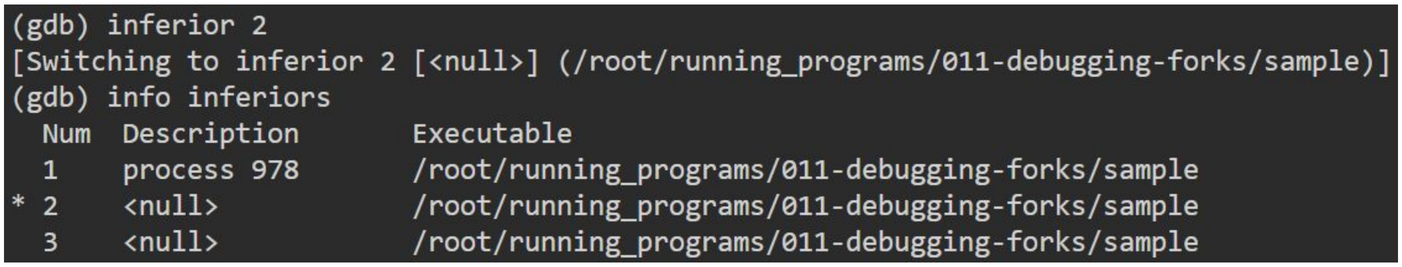


```
(gdb) info inferiors
  Num  Description       Executable
* 1    process 978       /root/running_programs/011-debugging-forks/sample
  2    <null>            /root/running_programs/011-debugging-forks/sample
  3    <null>            /root/running_programs/011-debugging-forks/sample
```

There are 3 inferiors now. One of them is running and the other two newly-cloned inferiors are not running, resulting in <null> in their description.

Step 73: Switch to inferior 2 and check the inferior list again. The asterisk (*) shows the current inferior.

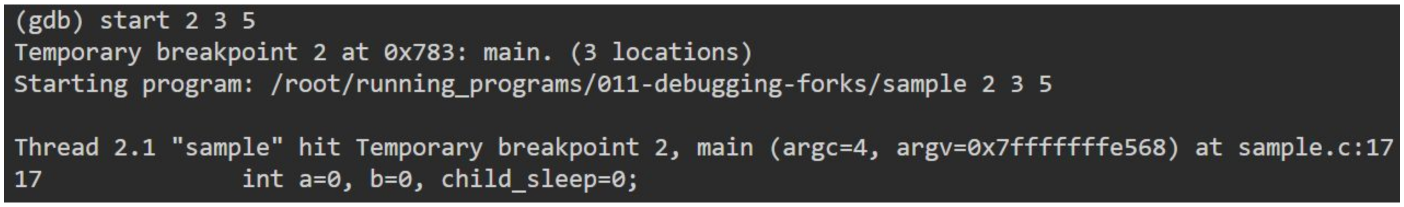
Commands:
inferior 2
info inferiors



```
(gdb) inferior 2
[Switching to inferior 2 [<null>] (/root/running_programs/011-debugging-forks/sample)]
(gdb) info inferiors
  Num  Description       Executable
  1    process 978       /root/running_programs/011-debugging-forks/sample
* 2    <null>            /root/running_programs/011-debugging-forks/sample
  3    <null>            /root/running_programs/011-debugging-forks/sample
```

Step 74: Run the program in this inferior.

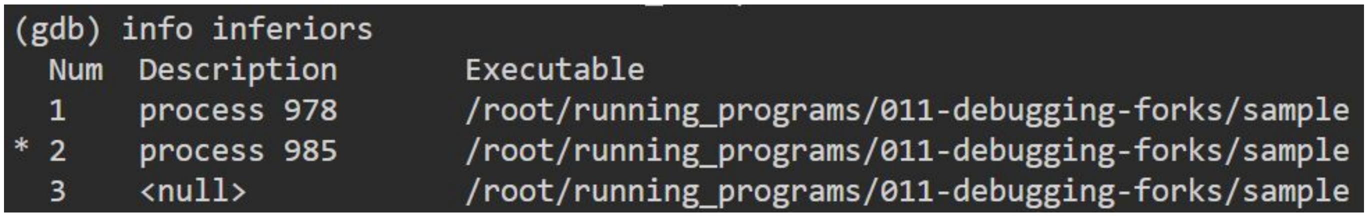
Command: start 2 3 5



```
(gdb) start 2 3 5
Temporary breakpoint 2 at 0x783: main. (3 locations)
Starting program: /root/running_programs/011-debugging-forks/sample 2 3 5

Thread 2.1 "sample" hit Temporary breakpoint 2, main (argc=4, argv=0x7fffffffe568) at sample.c:17
17              int a=0, b=0, child_sleep=0;
```

Check the inferiors again

Command: info inferiors

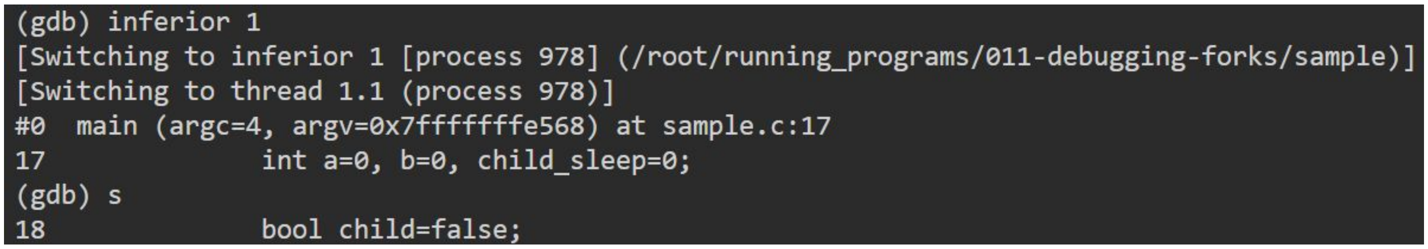


```
(gdb) info inferiors
  Num  Description       Executable
  1    process 978       /root/running_programs/011-debugging-forks/sample
* 2    process 985       /root/running_programs/011-debugging-forks/sample
  3    <null>            /root/running_programs/011-debugging-forks/sample
```

This time the second inferior also has a process ID as it is running.

Step 75: Switch back to inferior 1 and take 1 step in program execution.

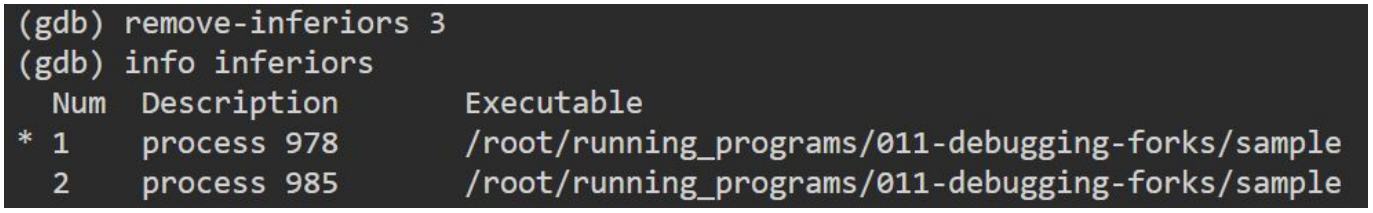
Commands:
inferior 1
s



```
(gdb) inferior 1
[Switching to inferior 1 [process 978] (/root/running_programs/011-debugging-forks/sample)]
[Switching to thread 1.1 (process 978)]
#0  main (argc=4, argv=0x7fffffffe568) at sample.c:17
17              int a=0, b=0, child_sleep=0;
(gdb) s
18              bool child=false;
```

Step 76: Remove inferior 3 as this one is not being used and check the list again.

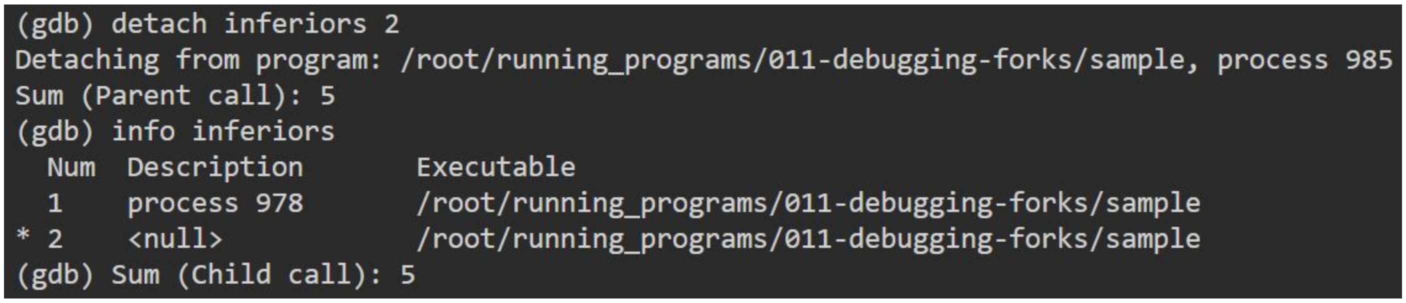
Commands:
remove-inferior 3
info inferiors



```
(gdb) remove-inferiors 3
(gdb) info inferiors
  Num  Description       Executable
* 1    process 978       /root/running_programs/011-debugging-forks/sample
  2    process 985       /root/running_programs/011-debugging-forks/sample
```

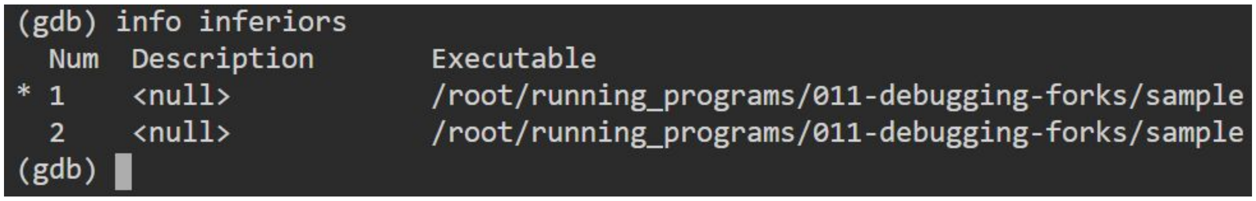
Step 77: Detach the inferior 2. Detaching will only stop the execution of the program but the inferior will still remain in the list. The description, however, will become “null” again.

Commands:
detach inferiors 3
info inferiors


```
(gdb) detach inferiors 2
Detaching from program: /root/running_programs/011-debugging-forks/sample, process 985
Sum (Parent call): 5
(gdb) info inferiors
  Num  Description       Executable
  1    process 978       /root/running_programs/011-debugging-forks/sample
* 2    <null>            /root/running_programs/011-debugging-forks/sample
(gdb) Sum (Child call): 5
```

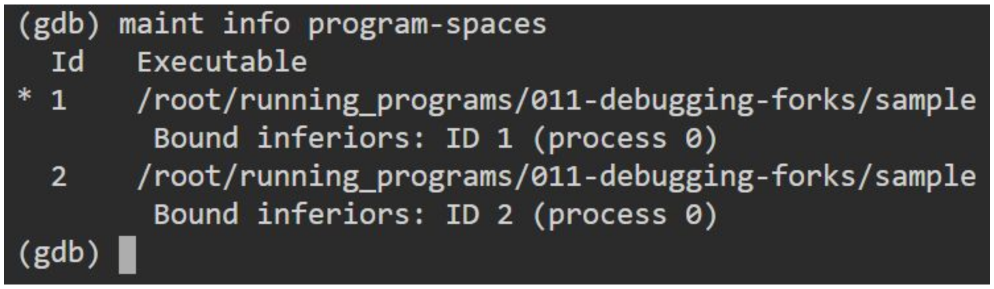
Check the inferiors again

Command: info inferiors


```
(gdb) info inferiors
  Num  Description       Executable
* 1    <null>            /root/running_programs/011-debugging-forks/sample
  2    <null>            /root/running_programs/011-debugging-forks/sample
(gdb)
```

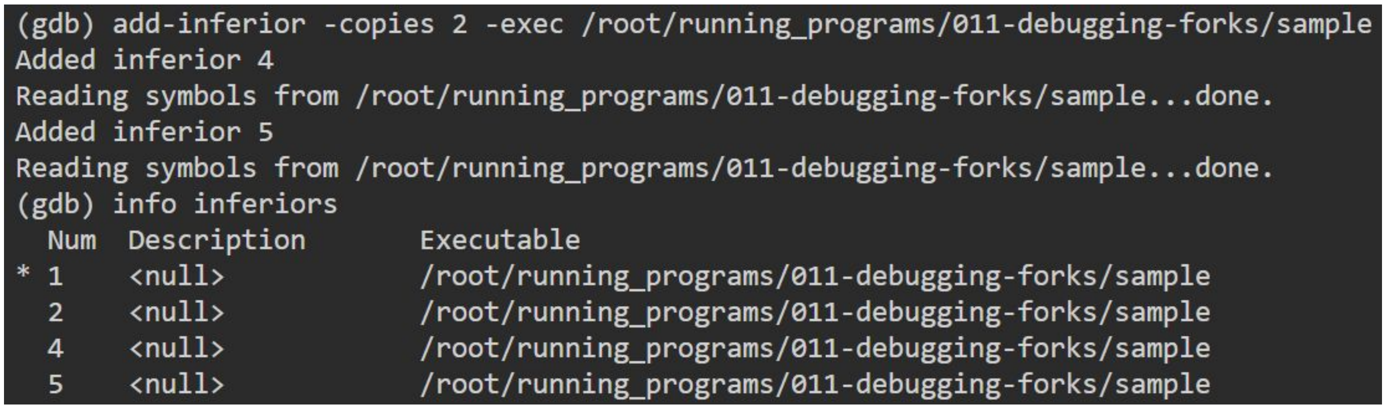
Step 78: List all executables being run in the GDB.

Command: maint info program-spaces


```
(gdb) maint info program-spaces
  Id   Executable
* 1    /root/running_programs/011-debugging-forks/sample
        Bound inferiors: ID 1 (process 0)
  2    /root/running_programs/011-debugging-forks/sample
        Bound inferiors: ID 2 (process 0)
(gdb)
```

Step 79: More inferiors can also be added by specifying the copies to create an executable. Create 2 more copies of inferiors using the same sample binary.

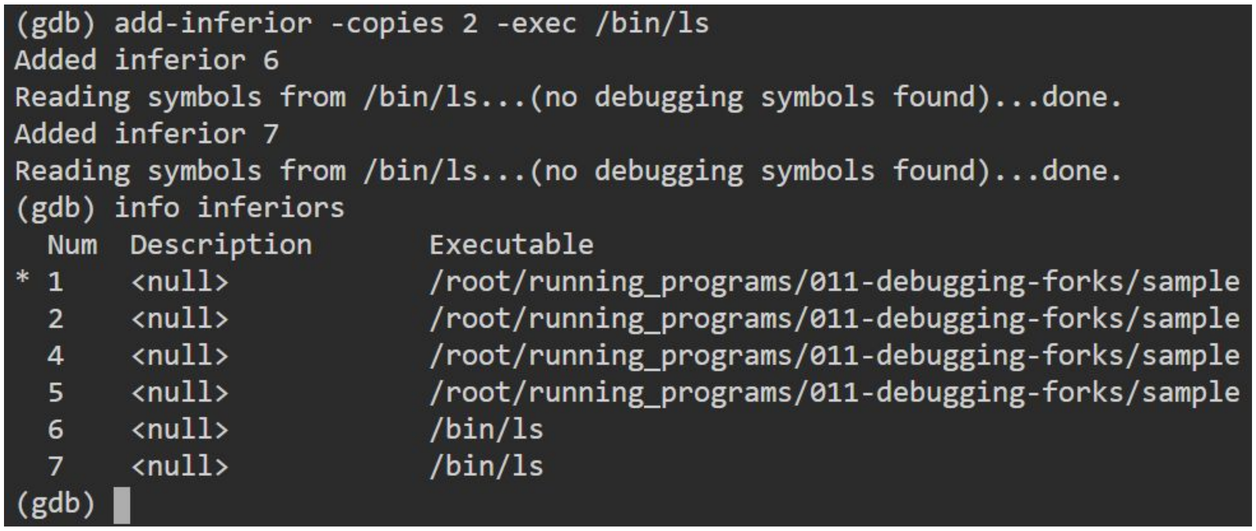
Commands:
add-inferior -copies 2 -exec /root/running_programs/011-debugging-forks/sample
info inferiors



```
(gdb) add-inferior -copies 2 -exec /root/running_programs/011-debugging-forks/sample
Added inferior 4
Reading symbols from /root/running_programs/011-debugging-forks/sample...done.
Added inferior 5
Reading symbols from /root/running_programs/011-debugging-forks/sample...done.
(gdb) info inferiors
  Num  Description       Executable
* 1    <null>            /root/running_programs/011-debugging-forks/sample
  2    <null>            /root/running_programs/011-debugging-forks/sample
  4    <null>            /root/running_programs/011-debugging-forks/sample
  5    <null>            /root/running_programs/011-debugging-forks/sample
```

Step 80: Different binaries can also be added as new inferiors. Create 2 copies of inferiors for binary /bin/ls and check the inferior list.

Commands:
add-inferior -copies 2 -exec /bin/ls
info inferiors



```
(gdb) add-inferior -copies 2 -exec /bin/ls
Added inferior 6
Reading symbols from /bin/ls...(no debugging symbols found)...done.
Added inferior 7
Reading symbols from /bin/ls...(no debugging symbols found)...done.
(gdb) info inferiors
  Num  Description       Executable
* 1    <null>            /root/running_programs/011-debugging-forks/sample
  2    <null>            /root/running_programs/011-debugging-forks/sample
  4    <null>            /root/running_programs/011-debugging-forks/sample
  5    <null>            /root/running_programs/011-debugging-forks/sample
  6    <null>            /bin/ls
  7    <null>            /bin/ls
(gdb) 
```

In this manner, multiple programs can be debugged in the same session.

References:

1. GDB Documentation (https://sourceware.org/gdb/current/onlinedocs/gdb)