SOFTWARE ENGINEERING LAB

Assignment 2

Group No. : 19

- Alok Ranjan(2022CSB091)
- Ranveer Kumar(2022CSB092)
- Karan Kumar(2022CSB093)
- Rajesh Kumar(2022CSB094)

1. Running the Program

Executing a program inside GDB to analyze its behavior. This helps in debugging by allowing controlled execution, setting breakpoints, and inspecting variables.

```
#include<stdio.h>

Files in(){
        printf("Hello, GDB!\n");

        int x;
        scanf("%d",&x);

        return 0;
}
~
~
~
```

Step 2: Compile with Debugging Information

```
karan@karan-VirtualBox:~/SE_Lab$ gcc -g sample.c -o sample
```

The `-g` flag includes debugging symbols, which GDB uses to analyze the program.

Step 3: Start GDB with the Program

```
karan@karan-VirtualBox:~/SE_Lab$ gdb ./sample
```

Step 4: Run the Program in GDB

```
Reading symbols from ./sample...
(gdb) r
```

This starts execution under GDB's control.

If the program has command-line arguments, pass them as follows:

run arg1 arg2

2. Loading symbol table

- GDB loads the symbol table when a program is compiled with `-g`. It maps variable names, function names, and memory locations, enabling debugging.
- GDB would ordinarily parse a typical file name, like `foo.c', as the three words `foo' `.' `c'. To allow GDB to recognize `foo.c' as a single symbol, enclose it in single quotes; for example,p 'foo.c'::x

Compile with Debugging Information

```
karan@karan-VirtualBox:~/SE_Lab$ gdb ./sample
```

Checking Symbol Table Status

info functions ***# Lists all available functions***
info variables ***# Lists all variables***

```
(gdb) info functions
All defined functions:

File sample.c:
3:      int main();

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001070  __cxa_finalize@plt
0x0000000000001080  puts@plt
0x0000000000001090  __stack_chk_fail@plt
0x00000000000010a0  printf@plt
0x00000000000010b0  __isoc99_scanf@plt
0x00000000000010c0  _start
0x00000000000010f0  deregister_tm_clones
0x0000000000001120  register_tm_clones
0x0000000000001160  __do_global_dtors_aux
0x00000000000011a0  frame_dummy
0x0000000000001224  _fini
```

3. Setting the Breakpoint

- A breakpoint in GDB is a marker that pauses program execution at a specific line, function, or memory location. This allows you to inspect variables, step through code, and debug efficiently.
    1. **Set a breakpoint at the beginning of a function**
        - Example. Set a breakpoint at the beginning of `main`.
        -
```
(gdb) b main
Breakpoint 2 at 0x11b5: file sample.c, line 3.
```
    2. **Set a breakpoint at a line of the current file during debugging.**
        - Example. Set a breakpoint at line 4 while in file sample.c
        -
```
(gdb) b 4
Note: breakpoint 1 also set at pc 0x11c4.
Breakpoint 3 at 0x11c4: file sample.c, line 4.
```
    3. **Set a breakpoint at the beginning of a class member function.**
        - Example. Set a breakpoint at the beginning of *member function* erase of the *class* list.
        - *b list::erase*(I have not made any class so that's why not attaching any screenshots)

    4. **Listing breakpoints.**
        - Example. List all breakpoints which have been set so far in a debugging session.
        -
```
(gdb) info b
Num     Type           Disp Enb Address            What
4       breakpoint     keep y   0x00000000000011b5 in main at sample.c:3
5       breakpoint     keep y   0x00000000000011c4 in main at sample.c:4
```
    5. **Deleting a breakpoint**
        - Example. Delete the breakpoint at line 4.
        -
```
(gdb) delete 4
(gdb) info b
Num     Type           Disp Enb Address            What
5       breakpoint     keep y   0x00000000000011c4 in main at sa
```

4. Listing variables and examining their values

1.Example C Code

```c
#include<stdio.h>

int main(){

        int x = 65,y=3;
        x = x>>1 + y>>1;
        printf("%d\n", x);
        return 0;
}
~
~
```

2.Setting a Breakpoint and Running the Program

```
Breakpoint 1, main () at new.c:5
5               int x = 65,y=3;
```

3.Displaying a Variable's Value

```
(gdb) display x
1: x = -8472
```

4.Stepping Through Execution and Checking Values

```
(gdb) n
6               x = x>>1 + y>>1;
```

Output:

```
1: x = 65
```

5.Executing the Next Step

```
(gdb) n
7               printf("%d\n", x);
```

Output:

```
1: x = 2
```

5. Printing Content of an Array or Contiguous Memory

```c
#include<stdio.h>
int main(){
        int data[5] = {11,2,25,27,45};

        //code to be debugged
        for(int i=0; i<5; i++){
                printf("Element at index %d is %d\n",i,data[i]);
        }

        return 0;
}
```

Arrays in C are stored as contiguous memory blocks, and GDB provides ways to examine these memory regions. GDB allows printing entire arrays, specific elements, and raw memory blocks.

COMPILE with Debugging Symbols :

- `gcc -g array.c -o array`
- `gcc array.c -o array -g`

Both are same because in GCC (GNU Compiler Collection), the order of options generally does not matter.

`-g` option tells the compiler to include debugging information in the compiled executable.

Load the Program in GDB

- **gdb ./array**

```
Breakpoint 1, main () at array.c:6
6               printf("printing array elements : \n");
(gdb) print arr
$1 = {10, 20, 30, 40, 50}
(gdb) print *arr@3
$2 = {10, 20, 30}
```

```
Breakpoint 1, main () at array.c:9
9                       printf("Element at index %d is %d\n",i,arr[i]);
(gdb) c
Continuing.
Element at index 0 is 10

Breakpoint 1, main () at array.c:9
9                       printf("Element at index %d is %d\n",i,arr[i]);
(gdb) c
Continuing.
Element at index 1 is 20

Breakpoint 1, main () at array.c:9
9                       printf("Element at index %d is %d\n",i,arr[i]);
(gdb) c
Continuing.
Element at index 2 is 30

Breakpoint 1, main () at array.c:9
9                       printf("Element at index %d is %d\n",i,arr[i]);
(gdb) c
Continuing.
Element at index 3 is 40

Breakpoint 1, main () at array.c:9
9                       printf("Element at index %d is %d\n",i,arr[i]);
(gdb) c
Continuing.
Element at index 4 is 50
[Inferior 1 (process 3885) exited normally]
(gdb)
```

Using ‘x’ commands, we can print the contents of a contiguous memory location:

- x/5d arr # Prints 5 integers starting from arr’s memory address
- x/5x arr # Prints 5 elements in hexadecimal format

```
(gdb) x/5d arr
0x7fffffffdda0: 10      20      30      40
0x7fffffffddb0: 50
(gdb) x/5x arr
0x7fffffffdda0: 0x0000000a      0x00000014      0x0000001e      0x00000028
0x7fffffffddb0: 0x00000032
(gdb)
```

6. Printing Function Arguments

- Function arguments determine input values to functions. GDB can display arguments when the function is called.

```c
#include<stdio.h>

void add(int a,int b){
        int sum = a+b;
        printf("sum : %d\n",sum);
}

int main(){
        add(5,10);
        return 0;
}
```

Compile with Debugging Symbols

- `gcc -g args.c -o args`

Load Program in GDB

- `gdb ./args`

Set Breakpoint and Run

- `break add`
- `run`

Print Function Arguments

1. Print Arguments Inside Function
    - `info args`
2. Print Backtrace with Arguments
    - `bt`
3. Print Individual Argument
    - `print a , print b`

7. Next, Continue, Set Command

These commands control program execution flow:

- `next (n)`: Executes the next line without stepping into functions.
- `continue (c)`: Resumes execution until the next breakpoint.
- `set`: Modifies variable values at runtime.

```c
#include<stdio.h>
void insp(){
        int x=10;
        int y=20;
        int sum=x+y;

        printf("sum : %d\n", sum);
}
int main(){
        int arr[5]={12, 3, 43, 5, 60};
        int a=10;

        for(int i=0; i<5; i++){
                printf("Element at index %d is : %d\n", i, arr[i]);
        }
        insp();
        a=28;
        return 0;
}
```

Continue :

```
(gdb) continue
Continuing.
Element at index 0 is : 12
Element at index 1 is : 3
Element at index 2 is : 43
Element at index 3 is : 5
Element at index 4 is : 60

Breakpoint 2, insp () at simple1.c:5
5               int x=10;
(gdb) continue
Continuing.

Breakpoint 1, insp () at simple1.c:6
6               int y=20;
(gdb)
```

next :

```
(gdb) next
7               int sum=x+y;
(gdb) next
9               printf("sum : %d\n", sum);
(gdb) next
sum : 30
10      }
(gdb) next
main () at simple1.c:26
26              return 0;
```

set :

8. Single Stepping into a Function

- To single-step into a function in GDB, you can use the step (s) command. This command allows you to execute the current line of source code and, if it involves a function call, enter that function, stopping at the first line of the called function.

  (gdb) step # Step into the next function call

`step` vs. `next`:

- `step`: Steps into functions (i.e., goes into the function call and allows we to debug inside the function).
- `next`: Steps over functions (i.e., runs the entire function and moves to the next line in the current function).

9. Listing All Breakpoints

- To list all breakpoints we have set, use the `info breakpoints` command.

  (gdb) info breakpoints

```
Breakpoint 1 at 0x118f: file test1.c, line 8.
(gdb) break 7
Note: breakpoint 1 also set at pc 0x118f.
Breakpoint 2 at 0x118f: file test1.c, line 8.
(gdb) break 10
Breakpoint 3 at 0x119d: file test1.c, line 10.
(gdb) break greet
Breakpoint 4 at 0x1171: file test1.c, line 4.
(gdb) next
The program is not being run.
(gdb) run
Starting program: /home/rajesh26/software_engineerng/debugtest

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for system-supplied DSO at 0x7ffff7fc3000
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at test1.c:8
8               int a=10 ;
(gdb) next
9                   int b=20;
(gdb) info breakpoint
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000055555555518f in main at test1.c:8
        breakpoint already hit 1 time
2       breakpoint     keep y   0x000055555555518f in main at test1.c:8
        breakpoint already hit 1 time
3       breakpoint     keep y   0x000055555555519d in main at test1.c:10
4       breakpoint     keep y   0x0000555555555171 in greet at test1.c:4
(gdb)
```

10. Ignoring a Breakpoint for N Occurrences

- Use the `ignore` command to ignore a breakpoint for a specific number of times.

  (gdb) ignore <breakpoint_number> <N>

  Example :

  (gdb) ignore 1 3

  This will ignore breakpoint number 1 for 3 times before breaking on it.

11. Enable/Disable a Breakpoint

- Use the `enable` and `disable` commands to control breakpoints

  (gdb) disable <breakpoint_number> # Disable breakpoint

  (gdb) enable <breakpoint_number> # Enable breakpoint

```
(gdb) b 3
Breakpoint 1 at 0x555555555175: file simple1.c, line 4.
(gdb) b 6
Breakpoint 2 at 0x5555555551a7: file simple1.c, line 7.
(gdb) b 8
Breakpoint 3 at 0x5555555551b0: file simple1.c, line 9.
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555555175 in main at simple1.c:4
2       breakpoint     keep y   0x00005555555551a7 in main at simple1.c:7
3       breakpoint     keep y   0x00005555555551b0 in main at simple1.c:9
(gdb) disable 2
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555555175 in main at simple1.c:4
2       breakpoint     keep n   0x00005555555551a7 in main at simple1.c:7
3       breakpoint     keep y   0x00005555555551b0 in main at simple1.c:9
(gdb) enable 2
(gdb) infob
Undefined command: "infob".  Try "help".
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555555175 in main at simple1.c:4
2       breakpoint     keep y   0x00005555555551a7 in main at simple1.c:7
3       breakpoint     keep y   0x00005555555551b0 in main at simple1.c:9
(gdb)
```

12. Break Condition and Command

- we can specify a **condition** for a breakpoint. The program will only break if a certain condition is true. Additionally, you can associate commands with breakpoints, like printing a variable when the breakpoint is hit.

  (gdb) condition <breakpoint_number> <condition>

(gdb) command <breakpoint_number> # Define commands to run when breakpoint hits

Example:

- To set a **condition** on breakpoint number 1, so it only breaks when `a > 10`:

  (gdb) condition 1 a > 10

- To associate a **command** with a breakpoint, for example, printing the value of `a` whenever the breakpoint is hit:

  (gdb) command 1

  Type commands for breakpoint 1, one per line.

  End with a line containing just "end".

  > print a

> end

This will automatically print the value of `a` whenever the breakpoint is triggered.

13. Examining Stack Trace

- A **stack trace** shows the function call hierarchy, which is helpful for identifying where an error or issue occurred. we can use the `backtrace` command to examine the stack trace in GDB.

  (gdb) backtrace # Display the call stack

This will show the current stack frames and how the program arrived at its current position. It's useful for tracing the sequence of function calls leading to an error or unexpected behavior.

14.Examining stack trace for multi-threaded program

- When debugging a multi-threaded program, GDB allows us to inspect the call stack for each thread. This helps identify issues like deadlocks, race conditions, and crashes.

thread.c

```c
#include <stdio.h>
#include <pthread.h>

void *threadFunction1(void *arg) {
    for (int i = 0; i < 5; ++i) {
        printf("Thread 1: %d\n", i);
    }
    return NULL;
}

void *threadFunction2(void *arg) {
    for (int i = 0; i < 5; ++i) {
        printf("Thread 2: %d\n", i);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // Create two threads
    pthread_create(&thread1, NULL, threadFunction1, NULL);
    pthread_create(&thread2, NULL, threadFunction2, NULL);

    // Wait for both threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

When an issue occurs or a breakpoint is hit, use the following commands to examine the stack trace for each thread:

Show all threads:

```
info threads
```

```
thread thread_id
```

Replace thread_id with the thread id you want to visit.

```
bt
```

This will display the stack trace for the current thread,

showing the function call hierarchy.

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./thread...
(gdb) break threadFunction1
Breakpoint 1 at 0x11b9: file thread.c, line 5.
(gdb) run
Starting program: /home/tanis/Downloads/GDB Codes/thread
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff7bff640 (LWP 7025)]
[New Thread 0x7ffff73fe640 (LWP 7026)]
Thread 2: 0
Thread 2: 1
Thread 2: 2
Thread 2: 3
Thread 2: 4
[Thread 0x7ffff73fe640 (LWP 7026) exited]
[Switching to Thread 0x7ffff7bff640 (LWP 7025)]

Thread 2 "thread" hit Breakpoint 1, threadFunction1 (arg=0x0) at thread.c:5
5           for (int i = 0; i < 5; ++i) {
(gdb) info threads
  Id   Target Id                                  Frame
  1    Thread 0x7ffff7fa8740 (LWP 7022) "thread" __futex_abstimed_wait_common64
    (private=128, cancel=true, abstime=0x0, op=265, expected=7025,
    futex_word=0x7ffff7bff910) at ./nptl/futex-internal.c:57
* 2    Thread 0x7ffff7bff640 (LWP 7025) "thread" threadFunction1 (arg=0x0)
    at thread.c:5
(gdb) thread 2
[Switching to thread 2 (Thread 0x7ffff7bff640 (LWP 7025))]
#0  threadFunction1 (arg=0x0) at thread.c:5
5           for (int i = 0; i < 5; ++i) {
(gdb) bt
#0  threadFunction1 (arg=0x0) at thread.c:5
#1  0x00007ffff7c94ac3 in start_thread (arg=<optimized out>)
    at ./nptl/pthread_create.c:442
#2  0x00007ffff7d26850 in clone3 ()
    at ../sysdeps/unix/sysv/linux/x86_64/clone3.S:81
(gdb)
```

We can type **thread thread_number** again to return to the original thread.

To see the stack trace of all the threads, we can write -

```
thread apply all bt
```

15. Core File Debugging

A core file is a memory dump generated when a program crashes. GDB can analyze these without rerunning the program.

Step 1: Example C Code (Segmentation Fault)

```
#include <stdio.h>

int main() {
    int *ptr = NULL;  // Null pointer
    *ptr = 42;        // Causes segmentation fault
    return 0;
}
```

Step 2: Enable Core Dump Generation

```
ulimit -c unlimited
```

Step 3: Compile and Run

```
gcc -g crash.c -o crash
./crash  # Causes a segmentation fault, generating a core dump
```

Step 4: Check Core File

```
ls core*  # Confirms core file existence
```

Step 5: Debug Core File in GDB

```
gdb ./crash core
```

Step 6: Analyze the Core Dump

Method 1: Identify Where the Crash Occurred

```
bt  # Shows function calls leading to the crash
```

Method 2: Show Line of Error

```
list
```

Method 3: Print Variables at Crash Point

```
frame 0
```

```
info locals
```

Method 4: Inspect Memory

```
x/10xw $esp  # Examine memory at the stack pointer
```

16. Debugging of an already running program

"Debugging an already running program" refers to the process of attaching a debugger to a program that is currently executing, allowing you to inspect its state, set breakpoints, and analyze variables while it continues to run, essentially troubleshooting issues without restarting the application from scratch; this is typically done using a command like "attach" in most debugging tools, where you need to identify the process ID (PID) of the running program to connect to it

17. Watchpoint

To set a watchpoint on a variable, you can use the `watch` command followed by the name of the variable. GDB will monitor this variable and pause execution as soon as its value changes.

(gdb) watch <variable_name>

References:

- **Debugging with GDB by MIT :** https://web.mit.edu/gnu/doc/html/gdb_toc.html
- **Tutorialspoint :** https://www.tutorialspoint.com/gnu_debugger/gdb_quick_guide.htm
- https://www.isical.ac.in/~dfslab/2019/lectures/2019-day11-tools-I.pdf
- **For x-command:** https://visualgdb.com/gdbreference/commands/x
- Debugging with gdb - Examining Data - Apple Developer