Lab 01: GDB Tutorial

1. Overview

gdb is a debugger for C (and C++). It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line. gdb uses a command line interface.

2. GDB tutorial

This is a brief description of some of the most commonly used features of gdb.

2.1. Compiling

To prepare your program for debugging with gdb, you must compile it with the -g flag. So, if your program is in a source file called test.c and you want to put the executable in the file test.out, then you would compile with the following command:

```
seed@ubuntu:~$ gcc -g -m32 -o test.out test.c
```

Notes:

-m32 option tells the compiler to generate code in 32-bit mode if you are compiling on a 64-bit system. You can skip this option if you are working on a 32 system.

2.2. Invoking and Quitting GDB

To start gdb, just type gdb at the Linux prompt. gdb will give you a prompt that looks like this: (gdb). From that prompt you can run your program, look at variables, etc., using the commands listed below Or, you can start gdb and give it the name of the program executable you want to debug by typing

```
seed@ubuntu:~$ gdb test.out
```

or you can start program after launching gdb:

```
(gdb) file test.out
```

To exit the program just type quit at the (gdb) prompt (actually just typing q is good enough).

Once the debugging symbols from the executable were loaded, you can start executing your program using the run or r command

```
(gdb) run
```

Summary of commonly used commands in gdb

Command	Description	Additional information	Example
disas	Encode memory bytes into assembly code and display (disassemble)	set disassembly-flavor intel	disas main disas func
Start	Start debugging program		start
Run	Execute program until meet breakpoint or till the end		

break	Breakpoints	break function break linenum break *address	break main break 6 break *0xbffff2d3
info break	List breakpoint defined	i b	
delete	remove breakpoints		delete breakpoint 2
step stepi, next, nexti	Execute one line of source code	step – step into next – step over finish – continue the execution until return	
info register	Display a summary of the current registers and flags	i r	
print	Print the value of a specific register or variable	Can use format for suitable output /x – hex, /d – signed decimal, /u – unsigned decimal, /t – binary, /c – character, /f – floating point number	print mem print $esp p/x mem p/s mem
x	Examining memory	x/nfu address n – the repeat count f – the display format (x – hex, I – machine instruction) u – the unit size (b - byte, h – two bytes, w – four bytes, g – eight bytes)	x/20xb x/20xw x/i $eip x/20xb $eip
set	Change the value of a variable or specific memory area		set a=5

2. Objectives

This lab aims to provide students with ability:

a) To compose/edit code in Linux terminal with nano text editor;
b) To compile source code with various gcc options for debugging in gdb;
c) To load, set breakpoints, run program step-by-step, examining CPU registers, memory areas

3. Lab Environment

a) Install VMWare workstation/VirtualBox on your Windows Computer.
b) Download the virtual machine image of a Linux machine for this lab from here.
c) Extract the SEEDUbuntu archive then open it in VMWare, the linux machine will start automatically (login with username: seed, password:dees).

4. Tasks

4.1. Compose s simple C++ program in nano text editor

Start nano on ubuntu by:

```
seed@ubuntu:~$ nano test.c
```

Enter code:

```
#include <stdio.h>
void func(int x, int y)
{
    x = x + y;
    printf("%d\n",x);
}
int main()
{
   int a=5;
   int b=7;
   func(a,b);
   printf("Hello\n");
   return 0;
}
```

Press ^X to quit → Reply with “Y” when being asked then Enter to confirm

4.2. Compile source code

Compile source code with gcc

```
gcc -g -o test.out test.c
```

The object code test.out now is ready for debugging

4.3. Load, debug, disassemble code

Load program in gdb

```
gdb test.out
```

display source code of main function:

```
(gdb) list main
```

display source code of func function:

```
(gdb) list func
```

Disassemble main function:

```
(gdb) disas main
```

Disassemble func function:

```
(gdb) disas func
```

By default, the assembly code follows the AT&T syntax like this:

```
   0x08048414 <+0>:  push   %ebp
   0x08048415 <+1>:  mov    %esp,%ebp
   0x08048417 <+3>:  sub    $0x18,%esp
   0x0804841a <+6>:  mov    0xc(%ebp),%eax
```

```
   0x0804841d <+9>:  add    %eax,0x8(%ebp)
   0x08048420 <+12>: mov    $0x8048550,%eax
   0x08048425 <+17>: mov    0x8(%ebp),%edx
   0x08048428 <+20>: mov    %edx,0x4(%esp)
   0x0804842c <+24>: mov    %eax,(%esp)
   0x0804842f <+27>: call   0x8048320 <printf@plt>
   0x08048434 <+32>: leave
   0x08048435 <+33>: ret
```

To switch to intel assembly syntax, run this command in gdb:

```
(gdb) set disassembly-flavor intel
```

The assembly code will look like:

```
   0x08048414 <+0>:  push   ebp
   0x08048415 <+1>:  mov    ebp,esp
   0x08048417 <+3>:  sub    esp,0x18
   0x0804841a <+6>:  mov    eax,DWORD PTR [ebp+0xc]
   0x0804841d <+9>:  add    DWORD PTR [ebp+0x8],eax
   0x08048420 <+12>: mov    eax,0x8048550
   0x08048425 <+17>: mov    edx,DWORD PTR [ebp+0x8]
   0x08048428 <+20>: mov    DWORD PTR [esp+0x4],edx
   0x0804842c <+24>: mov    DWORD PTR [esp],eax
   0x0804842f <+27>: call   0x8048320 <printf@plt>
   0x08048434 <+32>: leave
   0x08048435 <+33>: ret
```

4.4. Examine the registers and memory

Insert breakpoints

To examine registers, the program needed to be executed and stop at a particular point. We will insert breakpoints to do this. Set breakpoints at the beginning of main and func functions:

```
(gdb) break main
Breakpoint 1 at 0x804843f: file test.c, line 10.
(gdb) break func
Breakpoint 2 at 0x804841a: file test.c, line 4.
```

List all breakpoints defined:

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804843f in main at test.c:10
2       breakpoint     keep y   0x0804841a in func at test.c:4
```

Execute program, the code execution will stop at the first breakpoint which is the first instruction of the main function:

```
(gdb) r
```

Examine the registers (*)

```
(gdb) info reg (or (gdb) i r )
```

All registers and values will be displayed

Try to set new values for some registers then check the result (*)

Examine the memory (*)

Use the x command (try with various unit size b - byte, h – two bytes, w – four bytes, g – eight bytes):

a) To display 24 bytes the code of main function.
b) To display 24 bytes the code from the current instruction.

How do you know that the display result from a) and b) are correct? Justify your reason.

c) To display 32 bytes of the stack memory from the top of stack.

4.5. Debug program (*)

a) Execute the program step by step then examine the values of variables a, b in main()
b) Try to set a, b to values other than 5,7 while executing program then check the final result.