Workshop on

PLATFORM DEVELOPMENT AND DEBUGGING ON LINUX

by
Maruthi Seshidhar Inukonda

Document revision history:

2011 March: “Workshop on Platform Development and Debugging on Linux” made for AITAM, Tekkali.
2018 Nov: Renamed as “Linux Platform Development and Debugging”. Added CC BY-NC-SA license

Preface

"A novice was trying to fix a broken Lisp machine by turning the power off and on. Knight, seeing what the student was doing spoke sternly: You cannot fix a machine just by power-cycling it with no understanding of what is going wrong. Knight turned the machine off and on. The machine worked".

This essentially means that a good programmer should know what is going inside an operating system and in a computer.

"To make no mistakes is not in the power of man;
But from their errors and mistakes the wise and good learn wisdom for the future"

Common bugs we see during our daily programming are illegal memory access, segmentation fault, abnormal program termination, stack overflow, out of memory, incorrect results, non deterministic results.

Debugging is process of finding bugs from computer programs. Debugging skills help a programmer know root cause of bugs, and gain more insight into how programs run. It there by helps a programmer write code tight code.

This workshop assumes that participant has done C (or C++) programming. It also assumes participant knows basic Unix & Unix commands. Examples in this workbook are captured from LINUX. Since GCC and GDB are available on all unix flavors (Solaris, AIX, HP-UX), the commands should work without any modifications.

Please let me know if you have any feedback on the workshop / workshop material at my mail-id.

Acknowledgements

During my under-graduation, I got great help from my computer sciences lecturer, S.Rama Krishna, who is a good programmer. He used to inspire me 1%, and at the same time used to keep me perspire 99% by giving lots and lots of daily programming exercises. I am very much grateful to him. From those days, I got motivated to help students in their programming. So I have made a hobby to occasionally take-up programming related courses, workshops and lectures.

Maruthi Seshidhar Inukonda,
M.Tech(CSE) IIT Roorkee,
maruthi.inukonda [at] gmail.com

Contents

1. Platform Development

1.1 Programs

Source Program: Program written in high-level language or Assembly language. Typically we use an editor (vi, emacs) to write source program.

Open a terminal and write the below program in `sample.c` using your preferred editor.

```
   $ vi sample.c
```

```
#include <stdio.h>
int main()
{
  char ch;
  printf("Hello\n");
  scanf("%c", &ch);
  return 0;
}
```

Object Program: Machine language code generated after compiling source program.

To generate object program from source program, use the compiler "gcc" [1] with -c option.

```
  $ gcc -o sample.o -c sample.c
  $ file sample.o
  sample.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV),
            not stripped
```

gcc refers to GNU Project compiler collection for C and C++. More recently g++

Executable Program: Program generated after linking object program file(s) with dependent libraries (eg. libc, libm, librt, …). It contains machine language code. Executable programs are machine dependent. Executable programs are also called binary programs.

Typically, gcc is used to generate executable program from object programs, libraries.

```
  $ gcc  -o sample -lc sample.o

  $ file sample
  sample: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
          for GNU/Linux 2.2.5, dynamically linked (uses shared libs),
          not stripped
```

Here -lc refers to libc. `libc` [2] contains executable code for all C standard library routines.

To directly generate executable program from source programs

```
    $ gcc -o sample sample.c
```

In this case compiler internally calls linker.

Traditionally, linker "ld" [3] is used to generate executable program from object programs, libraries.

1 gcc can be found in /usr/local/bin/

2 libc can be found in /usr/lib/

3 ld can be found in /usr/bin/

1.2 Multi-file Programs

A Source program may be a single file program or a multi-file program.

Write the below code in `fact.h` file using your preferred editor.

```
$ vi fact.h
```

```
int fact(int n);
```

Write the below code in `fact.c` file using your preferred editor.

```
$ vi fact.c
```

```
#include "fact.h"

int fact(int n)
{
  int i, f;

  for(f=1, i=1; i<=n; i++) {
    f = f * i;
  }
  return f;
}
```

Compile the above file.

```
  $ gcc -o fact.o -c fact.c
```

Write the below code in `mainfact.c` using your preferred editor.

```
$ vi mainfact.c
```

```
#include <stdio.h>
#include "fact.h"

int main()
{
  int n, f;
  char ch;

  printf("To calculate factorial, enter n:", &n);
  scanf("%d", &n);

  f = fact(n);
  printf("n!=%d\n", f);
  scanf("%c", &ch);
  return 0;
}
```

Compile the above file.

```
  $ gcc -o mainfact.o -c mainfact.c
```

Link both the object programs `fact.o` , `mainfact.o`, and `libc` to create executable program.

```
  $ gcc -o mainfact -lc mainfact.o fact.o
```

Run the program

```
  $ ./mainfact
  To calculate factorial, enter n:5
  n!=120
```

1.3 Libraries

A set of predefined general purpose functions packaged in a file is called a library. Eg. printf(), scanf(), fopen(), fclose() etc, are standard C functions defined in a library called libc. sin(), cos(), pow() etc are mathematical functions defined in libm. pthread_create(), pthread_destroy(), etc are defined in libpthread.

1.4 Types of Libraries

There are two kinds of binaries as far as linking to libraries are concerned. Correspondingly there are two kinds of libraries.

Statically linked binaries: An executable program in which all library functions are part of. Statically linked binaries are bigger in size. Libraries that are used to do static linkage are called static libraries.

```
$ ls -l /usr/lib/libc*.a
-rw-r--r--. 1 root root   20388 2010-04-16 18:52 /usr/lib/libc_nonshared.a
```

Note: Recent linux distributions dont support / recommend statically linked binaries.

Dynamically Linked binaries: An executable program in which all library functions are linked just before the execution. Dynamically linked binaries are smaller in size. Libraries that are used to do dynamic linkage are called dynamic libraries.

```
$ ls -l /usr/lib/libc.*
-rw-r--r--. 1 root root 238 2010-04-16 18:21 /usr/lib/libc.so
```

1.5 Coding Standards & Guidelines

Function Definitions : Return type on the same line as function name, parameters on the same line if they fit. Start brace {, to start in the next line of function definition.

```
Eg.
int main()
{
       …
}
```

Function Declarations and Calls : On one line if it fits; otherwise, wrap arguments at the parenthesis.

Line Length : Each line of text in your code should be at most 80 characters long.

Spaces vs. Tabs : Use only tabs, and indent one tab at a time. Use spaces only if an indentation less than tab space is required.

Conditionals : Prefer no spaces inside parentheses. The else keyword belongs on a new line.

Loops and Switch Statements : Switch statements may use braces for blocks. Empty loop bodies should use {} or continue. Start brace {, to start on the same line of loop statement.

Eg.
```
for(i=0; i<n; i++) {
	sum += i;
}
```

Pointer and Reference Expressions : No spaces around period or arrow. Pointer operators do not have trailing spaces.

Boolean Expressions : When you have a boolean expression that is longer than the standard line length, be consistent in how you break up the lines.

Return Values : Do not surround the return expression with parentheses. Always return a value from main() function. Unix convention is 0 for success, and non-zero for failure.

Variable and Array Initialization : Your choice of = or ().

Preprocessor Directives : Preprocessor directives should not be indented but should instead start at the beginning of the line.

2. Execution Environment

2.1 Process

Process:

- A running instance of executable program.
- process exists only in RAM, where as executable program exists on disk.
- Each running instance of same executable program is a separate & independent process.
- Each process has a unique process id (a +ve number)
- Each process has a parent process (except the init process, which is grandest parent in the system)
- Each process has a separate virtual address space.

Virtual Address Space:
Each process's virtual address space has Four types of segments:

- Text segment
- Initialized Data Segment
- Uninitialized Data Segment
- Stack Segment

2.2 List Processes

Use ps command to see processes.

To see processes run by a user, use ps -f -U <username> , where <username> is the username you used to login.

```
$ ps -f -U imaruthi
UID        PID  PPID  C STIME TTY          TIME CMD
imaruthi  2512  2339  0 20:23 pts/0    00:00:00 ./sample
imaruthi  2513  2336  0 20:23 pts/1    00:00:00 /bin/bash
imaruthi  2544  2513  2 20:23 pts/1    00:00:00 ps -f -U imaruthi
```

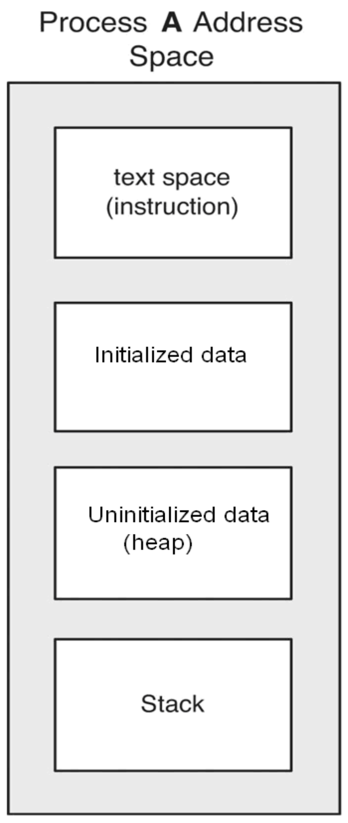
First column "UID" shows user id. Second column "PID" shows process id. Third column "PPID" shows parent process id. "STIME" shows start time. "TTY" shows terminal, "CMD" shows the program name.

2.3 Find a Process

Use ps command with grep to find required process.

```
$ ps -f -e | grep sample
imaruthi  2512  2339  0 20:23 pts/0    00:00:00 ./sample
imaruthi  2555  2513  0 20:25 pts/1    00:00:00 grep sample
```

2.4 Address Space of a Process



Text Segment: Text segment contains program’s binary code and libraries’ binary code. Statically linked executables have bigger text segment. And dynamically linked executables have smaller text segment.

Data Segment: Data segment contains two parts. The Initialized data , which contains global variables. And Uninitialized data which contains dynamically allocated variables (also called heap). Initialized data segment cannot grow and shrink, while a process runs. Where as uninitialized data segment can grow and shrink while a process runs, due to dynamic memory allocations and de-allocations. Initialized data segment is sometimes called Anonymous data segment.

Stack Segment: Stack contains Activation records (also called stack frames) of functions. Stack frame typically contains return address, contents of registers, processor status word, return value pointer, arguments, and local variables in a function. When a function gets executed its stack frame is pushed on the top of the Stack segment. In other words, If a function called from another function, a stack frame for the called function is pushed onto stack segment. Essentially, a stack segment can grow & shrink due to call and return from a function respectively. For eg, Recursion repeatedly pushes same stack frame onto the stack segment.

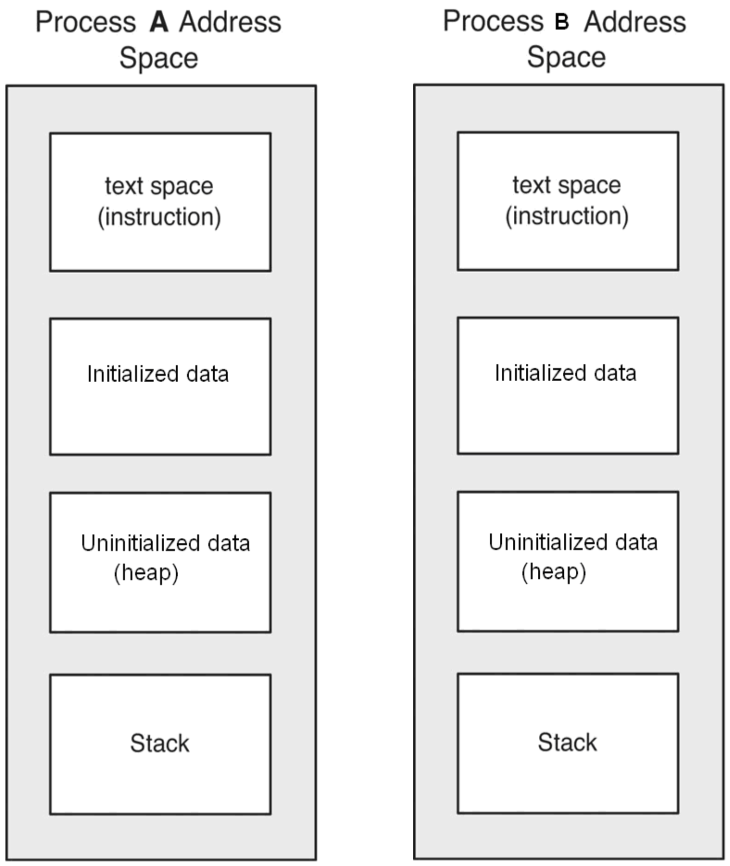
To see virtual address space size of a process, run “`cat /proc/<pid>/status`”.

```
$ cat /proc/2512/status
Name:   sample
State:  S (sleeping)
Tgid:   2512
Pid:    2512
PPid:   2339
TracerPid:      0
Uid:    19774   19774   19774   19774
Gid:    30      30      30      30
Utrace: 0
FDSize: 256
Groups: 30
VmPeak:     1728 kB
VmSize:     1728 kB
VmLck:         0 kB
VmHWM:       356 kB
VmRSS:       356 kB
VmData:       32 kB
VmStk:        84 kB
VmExe:         4 kB
VmLib:      1584 kB
VmPTE:        20 kB
Threads:        1
SigQ:   0/15753
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed:   1
Mems_allowed_list:      0
voluntary_ctxt_switches:        1
nonvoluntary_ctxt_switches:     2
Stack usage:    8 kB
```

To see virtual address space of a process, run "`cat /proc/<pid>/maps`"

```
$ cat /proc/2512/maps
0034f000-00350000 r-xp 00000000 00:00 0          [vdso]
0043c000-0045a000 r-xp 00000000 08:02 912310     /lib/ld-2.11.1.so
0045a000-0045b000 r--p 0001d000 08:02 912310     /lib/ld-2.11.1.so
0045b000-0045c000 rw-p 0001e000 08:02 912310     /lib/ld-2.11.1.so
00462000-005d0000 r-xp 00000000 08:02 912364     /lib/libc-2.11.1.so
005d0000-005d2000 r--p 0016e000 08:02 912364     /lib/libc-2.11.1.so
005d2000-005d3000 rw-p 00170000 08:02 912364     /lib/libc-2.11.1.so
005d3000-005d6000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 08:05 2359327    ./sample
08049000-0804a000 rw-p 00000000 08:05 2359327    ./sample
b779c000-b77a0000 rw-p 00000000 00:00 0
bf904000-bf919000 rw-p 00000000 00:00 0          [stack]
```

2.5 Multiple processes of same program.



When multiple processes of same program are there, operating system creates separate text, data, heap, stack segments in their respective virtual address spaces. Even though at the virtual memory subsystem level, text segments for different process of same program can be mapped to same physical page. But data, heap, stack segments have to be maintained separate in separate physical pages.

In other words, text segments can be mapped[4] in shared mode read-only, but data, heap, stack have to mapped in private writable mode.

2.6 Exercises

Run multiple instances of the sample program given in Section 1.1, and find pid of each of the process. Also see `/proc/<pid>/status`, and `/proc/<pid>/maps` output for each of the processes.

[4] in unix mapping & unmapping is done using mmap() & munmap() system calls respectively.

3. Debugging

3.1 What is Debugging

Debugging is a methodical process of finding and reducing number of defects (aka bugs), in a computer program.

3.2 Types of debugging

There are two types of debugging.
Live debugging : The process of finding bug from a program that is running.
Core debugging : The process of finding bug from a program that was running.

Another classification is
Source-Level debugging : During debugging, debugger shows position in the original source code.
Object-Level debugging : During debugging, debugger shows line in the disassembly.

This document covers Source-Level Live debugging & Source-Level Core dump debugging. It doesn't cover Object-Level debugging. Object-Level debugging requires understanding of Assembly language programming.

3.3 The GNU Debugger

GDB, the GNU[5] Project debugger, GDB allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

The program being debugged can be written in Ada, C, C++, Objective-C, Pascal (and many other languages). Those programs might be executing on the same machine as GDB (native) or on another machine (remote). GDB can run on most popular UNIX and Microsoft Windows variants. It is the most powerful debugger in the world.

3.4 Getting prepared for debugging

Default executable program generated using gcc, does not help debugging. One needs debug executable program

Debug Executable Program: An executable program with debug symbols included in it. Debuggers can only work well with Debug executable programs. Debug information are

- Symbol Table
- Line number and source filename corresponding to each symbols to facilitate source level debugging
- Other debug information for debugger

```
$ gcc -o sample_def sample.c
$ file sample_def
sample_def: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux),
dynamically linked (uses shared libs), for GNU/Linux 2.6.18, not stripped
```

To generate executable program with debugging information, use gcc's "-g" option. The -g option inserts debugging symbols into executable program to facilitate debugging.

```
$ gcc -g -o sample_debug sample.c
```

5 GNU stands for "**G**NU is **N**ot **U**nix". It is a recursive acronym.

```
$ file sample_debug
 sample_debug: ELF 32-bit LSB executable, Intel 80386, version 1
 (GNU/Linux), dynamically linked (uses shared libs), for GNU/Linux
 2.6.18, not stripped

 $ ls -l sample_def sample_debug
 -rwxr-xr-x. 1 imaruthi gopher 6230 2010-07-29 20:38 sample_debug
 -rwxr-xr-x. 1 imaruthi gopher 5058 2010-07-29 20:36 sample_def
```

Debug executables are bigger in file size due to extra debug information. These kind of executables are used before production.

Stripped Executable Program: An executable program from which debug symbols got removed. These kind of executables are used in production environment. Stripping removes

- Symbol Table
- Line number and source filename corresponding to each symbols
- Other debug information

To strip debugging information from an executable program, use strip [6] command.

```
$ cp sample_debug sample_stripped
$ strip sample_stripped
$ file sample_stripped
sample_stripped: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.18, stripped

$ ls -l sample_*
-rwxr-xr-x. 1 imaruthi gopher 6230 2010-07-29 20:38 sample_debug
-rwxr-xr-x. 1 imaruthi gopher 5058 2010-07-29 20:36 sample_def
-rwxr-xr-x. 1 imaruthi gopher 3372 2010-07-29 20:40 sample_stripped
```

Note: “compiling without -g option of gcc” and “compiling with -g option of gcc followed by strip” are not equivalent. gcc without -g produces a non stripped binary.

[6] strip command can be found in /usr/bin/

4. Live Debugging

4.1 Sample program

Consider the following program. The program calculates sum of n numbers. The program is very small one and is just used for demonstration purpose. gdb can help in debugging very big programs (millions of lines of code).

```
$ vi sum_of_n.c
```

```
#include <stdio.h>

unsigned findsum(unsigned n)
{
  unsigned i, sum=0;
  for(i=1; i<=n; i++) {
   sum += i;
  }
  return sum;
}

int main()
{
  int n, sum;

  printf("This program finds sum of n numbers , enter n:");
  scanf("%d", &n);
  sum = findsum(n);
  printf("sum of %d numbers: %d\n", n, sum);
  return 0;
}
```

Compile & run the program.

```
$ gcc -g -o sum_of_n sum_of_n.c
$ ./sum_of_n
This program finds sum of n numbers , enter n:4
sum of 4 numbers: 10
```

There is a bug in program. It hangs for some values of n. (eg. n = -1). Now run the program with n = -1, It hangs. Press “Ctrl C” to abort the program.

```
$ ./sum_of_n
This program finds sum of n numbers , enter n:-1
^C
```

In the next few sections, we will see how to debug this program using gdb.

4.2 Starting and Quitting gdb

To start the gdb [7], run "`gdb <executable>`".

```
$ gdb ./sum_of_n
GNU gdb (GDB) Fedora (7.0.1-45.fc12)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from ./sum_of_n...done.
```

It starts gdb, and loads the program in RAM. It displays copyright information and platform information ("`i686-redhat-linux-gnu`"). After starting it displays gdb prompt "`(gdb)`". This starts live debugging session.

To quit from debugger , enter quit at the gdb prompt.

```
(gdb) quit
```

To run run the program in gdb, use the gdb command "`run`" the gdb prompt. Note: If your program expects command line arguments, use *run* command with arguments "`run  arg1 arg2 ...`"

```
 (gdb) run
 Starting program: ./sum_of_n
 This program finds sum of n numbers , enter n:4
 sum of 4 numbers: 10

 Program exited normally.
```

Now run the program in debugger with n=-1, to troubleshoot it. When the program hangs, press "Ctrl c".

```
$ gdb ./sum_of_n
(gdb) run
Starting program: ./sum_of_n
This program finds sum of n numbers , enter n:-1
^C
Program received signal SIGINT, Interrupt.
0x08048437 in findsum (n=4294967295) at sum_of_n.c:6
6               for(i=1; i<=n; i++) {
```

In the above output, gdb interrupted the program when the program was executing line no 6.

[7] gdb can be found in /usr/bin/

4.3 specifying source code directory

To do source-level live debugging, you need to specify directory of the source program file(s) using *dir* command. "`dir <source_directory>` ". GDB searches source code file (or files, for a multi-file program) in the `source_directory`

```
(gdb) dir .
Source directories searched: .:$cdir:$cwd
```

It searches the current directory ".". If your source program file(s) are in current directory, gdb picks up the source files automatically without explicit *dir* command.

4.4 listing source code

To List source code, run "list" command. This command lists 10 lines in the vicinity of current line being executed.

```
(gdb) list
1       #include <stdio.h>
2
3       unsigned findsum(unsigned n)
4       {
5         unsigned i, sum=0;
6         for(i=1; i<=n; i++) {
7          sum += i;
8         }
9         return sum;
10      }
```

This is the privilege of source level debugging. It is very helpful, and comfortable to be able to see source code instructions, when a machine language program is running.

4.5 printing variables

To print variables run "`print <variable>`". Where `variable` is a variable visible at the current line of code being executed. The `variable` can be simple data type or composite data type

```
(gdb) print i
$3 = 163896007
(gdb) print n
$4 = 4294967295
(gdb) print sum
$5 = 3623179765
```

Note that, the input n = -1 is stored as 4,294,967,295, which is same as (2^{32} -1).
So the loop will iterate n times, looking like a hang.

4.6 break points

A breakpoint makes your program stop whenever a certain point in the program is reached. Breakpoints are used during Live debugging. You can set a breakpoint at a given location, which can specify a function name, a line number using "`break <function>`" or "`break <file:line>`" respectively.

Start the sum_of_n program in gdb and set a breakpoint at main( ). Note that the breakpoint is set at line 16, which is first executable line of code in main( ).

```
$ gdb ./sum_of_n

(gdb) break main
Breakpoint 1 at 0x804844a: file sum_of_n.c, line 16.
```

Set a breakpoint at findsum( ). Note that the breakpoint is set at line 5, which is first executable line of code in findsum( ).

```
(gdb) break findsum
Breakpoint 2 at 0x804841a: file sum_of_n.c, line 5.
```

Set breakpoint at line number 9, which belongs to function findsum().

```
(gdb) break sum_of_n.c:9
Breakpoint 3 at 0x804843c: file sum_of_n.c, line 9.
```

Set breakpoint at line number 6, which belongs to function findsum().

```
(gdb) break sum_of_n.c:6
Breakpoint 4 at 0x8048421: file sum_of_n.c, line 6.
```

To list all breakpoints use "*info break*" command.

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804844a in main at sum_of_n.c:16
2       breakpoint     keep y   0x0804841a in findsum at sum_of_n.c:5
3       breakpoint     keep y   0x0804843c in findsum at sum_of_n.c:9
4       breakpoint     keep y   0x08048421 in findsum at sum_of_n.c:6
```

To delete a breakpoint use "*delete <number>*"

```
(gdb) delete 4
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804844a in main at sum_of_n.c:16
2       breakpoint     keep y   0x0804841a in findsum at sum_of_n.c:5
3       breakpoint     keep y   0x0804843c in findsum at sum_of_n.c:9
```

The program stops at a breakpoint if the location specified is executed and the breakpoint is enabled. It doesn't stop at disabled breakpoints.

To disable a breakpoint use "*disable <number>*".

```
(gdb) disable 3
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804844a in main at sum_of_n.c:16
2       breakpoint     keep y   0x0804841a in findsum at sum_of_n.c:5
3       breakpoint     keep n   0x0804843c in findsum at sum_of_n.c:9
```

To enable a breakpoint use "*enable <number>*".

```
(gdb) enable 3
```

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804844a in main at sum_of_n.c:16
2       breakpoint     keep y   0x0804841a in findsum at sum_of_n.c:5
3       breakpoint     keep y   0x0804843c in findsum at sum_of_n.c:9
```

Use "`continue`" command to make the program proceed from a break point.

Now run the program, to troubleshoot it. The program starts and jumps to debugger at the breakpoint 1.

```
(gdb) run
Starting program: ./sum_of_n

Breakpoint 1, main () at sum_of_n.c:16
16                printf("This program finds sum of n numbers , enter 
n:");
(gdb) continue
Continuing.
This program finds sum of n numbers , enter n:-1

Breakpoint 2, findsum (n=4294967295) at sum_of_n.c:5
5               unsigned i, sum=0;
```

At the second breakpoint, see the value of function parameter "n", local variable "i" and "sum"

```
(gdb) list
1       #include <stdio.h>
2
3       unsigned findsum(unsigned n)
4       {
5               unsigned i, sum=0;
6               for(i=1; i<=n; i++) {
7                       sum += i;
8               }
9               return sum;
10      }
(gdb) print n
$1 = 4294967295
(gdb) print i
$2 = 0
(gdb) print sum
$3 = 0
```

Run `continue` command now. The program hangs. Press "Ctrl C".

```
(gdb) continue
Continuing.
^C
Program received signal SIGINT, Interrupt.
findsum (n=4294967295) at sum_of_n.c:6
6               for(i=1; i<=n; i++) {
```

See break point list, and notice that the hit count of breakpoints 1 and 2 increased to 1.

```
(gdb) info break
```

```
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804844a in main at sum_of_n.c:16
        breakpoint already hit 1 time
2       breakpoint     keep y   0x0804841a in findsum at sum_of_n.c:5
        breakpoint already hit 1 time
3       breakpoint     keep y   0x0804843c in findsum at sum_of_n.c:9
```

Now print the value of local variables “i" and “sum”.

```
(gdb) print sum
$4 = 3570609206
(gdb) print i
$5 = 274558812
```

4.7 stack-trace

Each function call inserts a stack frame on the stack. Functions called starting from main() to currently running function is called stack-trace. To see stack-trace of the program, use “*backtrace*” command.

Start the sum_of_n program in gdb , set a breakpoint at findsum() and run the program.

```
$ gdb ./sum_of_n

(gdb) break findsum
Breakpoint 1 at 0x804841a: file sum_of_n.c, line 5.
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804841a in findsum at sum_of_n.c:5

(gdb) run
Starting program: ./sum_of_n
This program finds sum of n numbers , enter n:-1

Breakpoint 1, findsum (n=4294967295) at sum_of_n.c:5
5               unsigned i, sum=0;

(gdb) backtrace
#0  findsum (n=4294967295) at sum_of_n.c:5
#1  0x08048478 in main () at sum_of_n.c:18
```

#0, #1 represent stack frame numbers. Essentially each frame corresponds to a function call.
The top most frame (i.e., #0) is the current frame in execution.

From the output of backtrace, it is apparent that n=-1 got type-casted to unsigned int and became n=2^{32} – 1 (ie, 4294967295).

To see values of local variables in current stack frame, use “*info locals*” command.

```
(gdb) info locals
i = 0
sum = 0
```

To see function arguments in current stack frame use “*info args*” command.

```
(gdb) info args
n = 4294967295
```

Now issue "`continue`" command to make the program proceed after the first breakpoint. The program hangs. Press "Ctrl C".

```
(gdb) continue
Continuing.
^C
Program received signal SIGINT, Interrupt.
0x08048437 in findsum (n=4294967295) at sum_of_n.c:6
6               for(i=1; i<=n; i++) {
```

Now see the values of local variables in current stack frame (#0)

```
(gdb) info locals
i = 248930482
sum = 2611203465
```

To select another frame other than current frame in execution use "*frame <no>*" command. Note that at any time only one stack frame can be selected.

```
(gdb) frame 1
#1  0x08048478 in main () at sum_of_n.c:18
18              sum = findsum(n);
(gdb) info locals
n = -1
sum = 6103028
(gdb) info args
No arguments.
(gdb) backtrace
#0  0x08048437 in findsum (n=4294967295) at sum_of_n.c:6
#1  0x08048478 in main () at sum_of_n.c:18
(gdb) frame 0
#0  0x08048437 in findsum (n=4294967295) at sum_of_n.c:6
6               for(i=1; i<=n; i++) {
```

To see more details of a stack frame, use "*info frame*" command. You have to use *frame* command prior to *info frame* command, to see stack frame information other than current frame.

```
(gdb) info frame
Stack level 0, frame at 0xbffff0a0:
 eip = 0x8048437 in findsum (sum_of_n.c:6); saved eip 0x8048478
 called by frame at 0xbffff0d0
 source language c.
 Arglist at 0xbffff098, args: n=4294967295
 Locals at 0xbffff098, Previous frame's sp is 0xbffff0a0
 Saved registers:
  ebp at 0xbffff098, eip at 0xbffff09c
(gdb) frame 1
#1  0x08048478 in main () at sum_of_n.c:18
18              sum = findsum(n);
(gdb) info frame
Stack level 1, frame at 0xbffff0d0:
 eip = 0x8048478 in main (sum_of_n.c:18); saved eip 0x478bb6
 caller of frame at 0xbffff0a0
 source language c.
 Arglist at 0xbffff0c8, args:
 Locals at 0xbffff0c8, Previous frame's sp is 0xbffff0d0
 Saved registers:
  ebp at 0xbffff0c8, eip at 0xbffff0cc
```

Now issue "`quit`" command to make the program proceed after the first breakpoint.

```
(gdb) quit
A debugging session is active.

Quit anyway? (y or n) y
```

4.8 Single stepping

The process of executing program line by line under the control of gdb is called single stepping. It helps more fine granular control of a running program.

Start the sum_of_n program in gdb , set a breakpoint at main() and run the program.

```
$ gdb ./sum_of_n

(gdb) break main
Breakpoint 1 at 0x804844a: file sum_of_n.c, line 16.
(gdb) run
Starting program: ./sum_of_n

Breakpoint 1, main () at sum_of_n.c:16
16                 printf("This program finds sum of n numbers , enter
n:");


(gdb) backtrace
#0  main () at sum_of_n.c:16
```

At the breakpoint, run "next" command to run the next statement (printf() in the line no:16).

```
(gdb) next
17              scanf("%d", &n);
(gdb) next
This program finds sum of n numbers , enter n:5
18              sum = findsum(n);
(gdb) next
19              printf("sum of %d numbers: %d\n", n, sum);
(gdb) next
sum of 5 numbers: 15
20              return 0;
(gdb) next
21      }
(gdb) next
0x00478bb6 in __libc_start_main () from /lib/libc.so.6
(gdb) next
Single stepping until exit from function __libc_start_main,
which has no line number information.
```

```
Program exited normally.
(gdb) quit
```

Note that in the above single stepping session, findsum() is not single-stepped. This is because, `next` command, doesn't detail function calls. To jump into function calls, use "`step`" command. Now, Run the program again.

```
$ gdb ./sum_of_n

(gdb) run
(gdb) break main
Breakpoint 1 at 0x804844a: file sum_of_n.c, line 16.
(gdb) run
Starting program: ./sum_of_n

Breakpoint 1, main () at sum_of_n.c:16
16                printf("This program finds sum of n numbers , enter
n:");

(gdb) next
17              scanf("%d", &n);
(gdb) n
This program finds sum of n numbers , enter n:3
18              sum = findsum(n);
```

After single stepping till line no. 18, the function call to findsum(), run step command.

```
(gdb) step
findsum (n=3) at sum_of_n.c:5
5               unsigned i, sum=0;
(gdb) backtrace
#0  findsum (n=3) at sum_of_n.c:5
#1  0x08048478 in main () at sum_of_n.c:18
(gdb) next
6               for(i=1; i<=n; i++) {
(gdb) next
7                       sum += i;
(gdb) next
6               for(i=1; i<=n; i++) {
(gdb) next
7                       sum += i;
(gdb) next
6               for(i=1; i<=n; i++) {
(gdb) next
7                       sum += i;
(gdb) next
6               for(i=1; i<=n; i++) {
(gdb) next
9               return sum;
(gdb) next
10      }
(gdb) next
main () at sum_of_n.c:19
19              printf("sum of %d numbers: %d\n", n, sum);
(gdb) next
sum of 3 numbers: 6
20              return 0;
(gdb) next
21      }
(gdb) next
0x00478bb6 in __libc_start_main () from /lib/libc.so.6
(gdb) next
Single stepping until exit from function __libc_start_main,
which has no line number information.

Program exited normally.
(gdb)
```

Command "`finish`" is like "`continue`" command, which continues the program till end of current function. Enter the following commands and notice the behavior.

```
$ gdb ./sum_of_n

(gdb) break main
Breakpoint 1 at 0x804844a: file sum_of_n.c, line 16.
(gdb) run
Starting program:
/home/imaruthi/AllPrograms/Seshi/unix/platformdev/sum_of_n

Breakpoint 1, main () at sum_of_n.c:16
16             printf("This program finds sum of n numbers , enter
n:");

(gdb) next
17              scanf("%d", &n);
(gdb) next
This program finds sum of n numbers , enter n:3
18              sum = findsum(n);
(gdb) next
19              printf("sum of %d numbers: %d\n", n, sum);
(gdb) next
sum of 3 numbers: 6
20              return 0;
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: ./sum_of_n

Breakpoint 1, main () at sum_of_n.c:16
16              printf("This program finds sum of n numbers , enter
n:");
(gdb) next
17              scanf("%d", &n);
(gdb) next
This program finds sum of n numbers , enter n:3
18              sum = findsum(n);
(gdb) step
findsum (n=3) at sum_of_n.c:5
5               unsigned i, sum=0;
(gdb) next
6               for(i=1; i<=n; i++) {
(gdb) next
7                       sum += i;
```

```
(gdb) next
6               for(i=1; i<=n; i++) {
(gdb) next
7                       sum += i;
(gdb) next
6               for(i=1; i<=n; i++) {
```

After single stepping in the for loop, for two iterations, run finish command.

```
(gdb) finish
Run till exit from #0  findsum (n=3) at sum_of_n.c:6
0x08048478 in main () at sum_of_n.c:18
18              sum = findsum(n);
Value returned is $1 = 6
```

The finish command completes the function in execution and stops in the calling function.

```
(gdb) next
19              printf("sum of %d numbers: %d\n", n, sum);
(gdb) next
sum of 3 numbers: 6
20              return 0;
(gdb) next
21      }
(gdb) next
0x00478bb6 in __libc_start_main () from /lib/libc.so.6
(gdb) next
Single stepping until exit from function __libc_start_main,
which has no line number information.

Program exited normally.
```

Now execute *quit* command.

```
(gdb) quit
```

4.9 Miscellaneous commands

To run any unix command from within gdb use *shell* command at (gdb) prompt.

```
  (gdb) shell ls
  (gdb) shell time
```

Pressing enter with no command executes the previous command

4.10 Exercises

1. Try the below program and print all variables

```
$ vi composite.c
```

```
#include <stdio.h>

int main()
{
  char ch;
  char my_carray[20] = "hello";

  int  my_iarray[10] = { 100, 101, 102, 103, 104, 105 };

  struct sample_struct {
    int i;
    char c;
  } my_struct = { 534, 'a', };

  union sample_union {
    int i;
    char c;
  } my_union = { 534 };

  printf("Defined & initialized composite variables\n");
  scanf("%c", &ch);
  return 0;
}
```

Compile the program

```
$ gcc -g -o composite composite.c
```

Run the program in gdb. It runs and reaches scanf statement and waits for user input. Press Ctrl+c and see stacktrace of the program.

```
$ gdb ./composite

(gdb) run
Starting program: ./composite
Defined & initialized composite variables
^C
Program received signal SIGINT, Interrupt.
0x00690424 in __kernel_vsyscall ()

(gdb) backtrace
#0  0x00690424 in __kernel_vsyscall ()
#1  0x0052ce63 in __read_nocancel () from /lib/libc.so.6
#2  0x004cc16b in _IO_new_file_underflow () from /lib/libc.so.6
#3  0x004cdd6b in _IO_default_uflow_internal () from /lib/libc.so.6
#4  0x004cf37a in __uflow () from /lib/libc.so.6
#5  0x004b775c in _IO_vfscanf_internal () from /lib/libc.so.6
#6  0x004beac9 in __isoc99_scanf () from /lib/libc.so.6
#7  0x080484c9 in main () at composite.c:21
```

Go to main function's stack frame (frame #7), and print the arrays, struct and union.

```
(gdb) frame 7
#7  0x080484c9 in main () at composite.c:21
21        scanf("%c", &ch);
(gdb) print my_carray
$1 = "hello", '\000' <repeats 14 times>
(gdb) print my_iarray
$2 = {100, 101, 102, 103, 104, 105, 0, 0, 0, 0}
(gdb) print my_struct
$3 = {i = 534, c = 97 'a'}
(gdb) print my_union
$4 = {i = 534, c = 22 '\026'}
(gdb) continue
Continuing.

Program exited normally.
(gdb) quit
```

5. Core Dump debugging

5.1 Core Dump

A core file (also called core dump) is created when a program terminates unexpectedly, due to a bug, or a violation of the operating system's or hardware's protection mechanisms. The operating system kills the program and creates a core file that programmers can use to figure out what went wrong. A **core dump** consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has terminated abnormally (crashed). In practice, other key pieces of program state are usually dumped at the same time, including the processor registers, which may include the program counter and stack pointer, memory management information, and other processor and operating system flags and information.

To determine what program a core file came from, use the file command:

```
$ file core.3743
core.3743: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-style, from
'./memaccess'
```

5.2 Core dump debugging

Consider the following program which does null pointer de-referencing.

```
#include<stdio.h>

int main()
{
  char *ptr = (char *)0xfffffff;
  printf("val=%c\n", *ptr);
  return 0;
}
```

Compile the program and run the program.

```
$ gcc –g –o memaccess memaccess.c
```

It receives Segmentation Fault. When run outside debugger, it generates a core dump file.

```
$ ./memaccess
Segmentation fault (core dumped)
```

```
$ file core.3743
core.3743: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-style, from
'./memaccess'
```

To know the reason for the segmentation fault, start gdb with program file and core file. This starts core dump debugging session.

```
$ gdb ./memaccess ./core.3743
Reading symbols from ./memaccess...done.
...
Core was generated by `./memaccess'.
Program terminated with signal 11, Segmentation fault.
#0  0x080483d9 in main () at memaccess.c:6
6         printf("val=%c\n", *ptr);
```

To do source-level core dump debugging, you need to specify directory of the source program file(s) using *dir* command. “dir <source_directory> “

```
(gdb) dir .
Source directories searched: .:$cdir:$cwd
```

See stack trace, and see the local variables in the top most frame.

```
(gdb) backtrace
#0  0x080483d9 in main () at memaccess.c:6
(gdb) frame 0
#0  0x080483d9 in main () at memaccess.c:6
6         printf("val=%c\n", *ptr);
(gdb) info locals
ptr = 0xfffffff <Address 0xfffffff out of bounds>
```

From the “info locals” output, ptr = $2^{31.}$ So the program crashed due to illegal memory access.

```
(gdb) quit
```

5.3 Exercises

Try the following program which calculates factorial of an integer recursively. For a big number, the program gets aborted due to stack overflow.

```
#include <stdio.h>

int fact(int n)
{
  if (n=0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}

int main()
{
  int n, res;
  char ch;

  printf("Enter n to find n! :");
  scanf("%d", &n);

  res = fact(n);
  printf("n!=%d\n", res);

  scanf("%c", &ch);
  return 0;
}
```

Compile and run the program

```
$ gcc -g -o factrec factrec.c

$ ./factrec
Enter n to find n! :2
Segmentation fault (core dumped)
```

```
$ file core.3886
core.3886: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-style, from './factrec'
```

Now do core dump debugging to find out the root cause.

```
$ gdb ./factrec ./core.3886
...
Core was generated by `./factrec'.
Program terminated with signal 11, Segmentation fault.
#0  0x08048434 in fact (n=0) at factrec.c:8
8           return n * fact(n-1);

(gdb) backtrace
```

```
#0  0x08048434 in fact (n=0) at factrec.c:8
#1  0x0804843c in fact (n=0) at factrec.c:8
#2  0x0804843c in fact (n=0) at factrec.c:8
#3  0x0804843c in fact (n=0) at factrec.c:8
…
…
…
---Type <return> to continue, or q <return> to quit---
(gdb)
```

Go to frame # 0 (the top most frame), and see arguments, and frame information.

```
(gdb) frame 0
#0  0x08048434 in fact (n=0) at factrec.c:8
8           return n * fact(n-1);

(gdb) info args
n = 0

(gdb) info frame
Stack level 0, frame at 0xbf2bb010:
 eip = 0x8048434 in fact (factrec.c:8); saved eip 0x804843c
 called by frame at 0xbf2bb030
 source language c.
 Arglist at 0xbf2bb008, args: n=0
 Locals at 0xbf2bb008, Previous frame's sp is 0xbf2bb010
 Saved registers:
  ebp at 0xbf2bb008, eip at 0xbf2bb00c
(gdb)
```

So, the crash was due to stack-overflow. In the fact() function, instead of equality operator ==, we used =, which assigns n with 0, and calls fact(0) again. This lead to indefinite recursion.(aka infinite recursion).