KERNEL MASTERS

Linux System Programming

GNU Toolchain

(Linux Development Environment)

Authored and Compiled By: **Boddu Kishore Kumar**

Email: kishore@kernelmasters.org

Reach us online: www.kernelmasters.org

Contact: 9949062828

Important Notice

This courseware is both the product of the author and of freely available open source materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of this courseware cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - source code and binaries (where applicable) - that form part of this courseware, and that are present on the participant CD, are released under the GNU GPL v2 license and can therefore be used subject to terms of the afore-mentioned license. If you do use any of them, in any manner, you will also be required to clearly attribute their original source (author of this courseware and/or other copyright/trademark holders).

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant CD are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

2012-2018 Kishore Kumar Boddu
Kernel Masters, Hyderabad - INDIA.

Linux Development Enviornment (GNU Tool Chain)

GNU:

GNU is a recursive acronym for *"**GNU's Not Unix**!"*, chosen because GNU's design is Unix-like, but differs from Unix by being free software and containing no UNIX code.

GNU is an operating system and an extensive collection of computer software. GNU is composed wholly of free software, most of which is licensed under the GNU Project's own GPL.

GNU Tool Chain:

The GNU toolchain is a broad collection of programming tools produced by the GNU Project. These tools form a toolchain used for developing software applications and operating systems.

The GNU toolchain plays a vital role in development of Linux, some BSD systems, and software for embedded systems.

1. GCC (GNU 'C' Compiler (or) GNU Compiler Collection)
2. GNU make - an automation tool for compilation and build
3. GNU Binutils - a suite of tools including linker, assembler and other tools
4. GDB (GNU Debugger) - a code debugging tool
5. GNU Build System – configuraiton tool

1. GCC (GNU 'C' Compiler (or) GNU Compiler Collection) :

GCC Command line options:

-o output
-E Stop after the preprocessing stage.
-S Stop after the compilation stage.
-c Stop after the assembler stage.
-g enable debugging symbol
-I <user defined header file path>
-L <user defined library path>
-l<library_name>
-v Print the commands executed to run the stages of compilation.
-Wall enable all warnings
-Werror Convert warnings into errors
-D[MACRO] Define a MACRO
-U[MACRO] Undefine a MACRO
@file Read command line options from file. Options in file are separated by whitespace.
-funsigned-char char type is treated as unsigned type.
-fsigned-char char type is treated as signed type.

Examples:

+ hello/hello.c - compilation stages (preprocessor, compiler, assembler, linker)
+ addsub/main.c,sub.c,add.c,myinclude.h - How to compile multiple source files.

2. GNU make:-

Compilation of multiple source files is possible in 2 ways.
1. Manual Method.
2. Automation Method: uses GNU Make tool.

```
# simple make file

all: main

main: main.o add.o sub.o
    gcc -o main main.o sub.o add.o

main.o: main.c
    gcc -c main.c

add.o: add.c
    gcc -c add.c

sub.o: sub.c
    gcc -c sub.c

clean:
    rm –rf *.o
```

Makefile Syntax:
Target: Dependencies
<tab> Commands

What is Makefile?

- Makefile help to compile multiple source files.
- Makefile is Dependency tracking utility.
- Make file look in to the current directory for a file by the name Makefile (recommended) or makefile.

Make Advantages:

- Make is a utility for automatically building executable programs from source code.
- Makefile compiles only modified files based on compilation time (i.e., time stamp) because of this compilation time is reduced.

Makefile Examples:
source files: main.c,add.c,sub.c
header files: myinclude.h
Example 1: This is Basic example of Makefile.
$ make (default target is all)
Example 2: This example shows how to use, clean and install targets, in Makefile.
$ make clean (This command cleans object files, binaries and configuration files)
$ make install (This command copy the binaries in /usr/bin directory)
Example 3: This Example shows how to use environment variables in Makefile
Example 4: This Example shows how to create multiple Makefiles in each directory.
Example 5: This Example shows how to give user defined name to Makefile

Static Linker vs Dynamic Linker:

Executale and Linkable Format (ELF) is a standard binary file format.

$ readelf -a first | more (readelf is a details of ELF file)

creation of exec file using Dynamic linker

$ gcc first.c -o first (by default)

creation of exec file using Static linker

$ gcc -static first.c -o first

Static Library vs Dynamic Library:

- Size is Different.
- Static Linkers use static libraries which are appended the executable image at build time.
- Dynamic Linkers use dynamic libraries which carrys symbolic reference in exectable and physically loaded at runtime.

Procedure for Creation of static libraries:

Step1: Implementaion of Source code.

one.c

two.c

Step2: Compile source code up to object file.

$ gcc -c one.c two.c

Step3: Use UNIX archive tools create library image.

$ ar -rcs libourown.a one.o two.o

Procedure for Creation of Dynamic libraries (Shared Libraries):

Step1: Implementaion of Source code.

one.c

two.c

Step2: Compile source to create position independent relocatale.

$ gcc -c -fpic one.c

$ gcc -c -fpic two.c

Step3:

$ gcc -shared -o libourown.so one.o two.o

Telling GCC where to find the User defiend header file:

-I <USER DEFINED HEADER PATH>

Telling GCC where to find the shared library:

Why -l<library name>, -L and LD_LIBRARY_PATH?

-l<library name> means compiler search the library

-L <path> means gcc compiler checks libraries at compilation time in path location.

Making the library available at runtime:

LD_LIBRARY_PATH=<path> means binary execution time checks the library PATH.

$ file libourown.a
libourown.a: current ar archive
$ file libourown.so
libourown.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, not stripped

Examples:
libraries/addsub_lib: how to create static and dynamic libraries.

3. GNU BinUtils:

ar: Creates static libraries, and inserts, deletes, lists, and extracts members.
strings: Lists all of the printable strings contained in an object file.
strip: Deletes symbol table information from an object file.
nm: Lists the symbols defined in the symbol table of an object file.
size: Lists the names and sizes of the sections in an object file.
readelf: Displays the complete structure of an object file, including all of the information encoded in the ELF header; subsumes the functionality of size and nm.
objdump: The mother of all binary tools. Can display all of the information in an object file. Its most useful function is disassembling the binary instructions in the .text section.
ldd: Lists the shared libraries that an executable needs at run time

4. GDB (GNU Debugger):

- A debugger for several languages, including C and C++
- It allows you to inspect what the program is doing at a certain point during execution.
- Errors like segmentation faults may be easier to find with the help of gdb.
- http://sourceware.org/gdb/current/onlinedocs/gdb toc.html - online manual.

Additional step when compiling program:
Normally, you would compile a program like:
gcc [flags] <source files> -o <output file>
For Example
gcc -Wall -Werror hello.c -o hello.x

Now you add a **-g** option to enable built-in debugging support (which gdb needs):
*gcc [other flags] **-g** <source files> -o <output file>*
For Example:
*gcc -Wall -Werror **-g** hello.c -o hello*

Starting up gdb
Just try “gdb” or “gdb hello” You’ll get a prompt that looks like this:
(gdb)
If you didn’t specify a program to debug, you’ll have to load it in now:
(gdb) file hello
Here, hello is the program you want to load, and “file” is the command to load it.

Gdb help

If you're ever confused about a command or just want more information, use the "help" command, with or without an argument:

(gdb) help [command]

Running the Program

(gdb) run

Settng Breakpoints

Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command "break."

This sets a breakpoint at a specified file-line pair:

(gdb) break file1.c:6

This sets a breakpoint at line 6, of file1.c. Now, if the program ever reaches that location when running, the program will pause and prompt you for another command.

"You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them."

Once you've set a breakpoint, you can try using the run command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).

You can proceed onto the next breakpoint by typing "continue" (Typing run again would restart the program from the beginning, which isn't very useful.)

(gdb) continue

You can single-step (execute just the next line of code) by typing "step." This gives you really fine-grained control over how the program proceeds. You can do this a lot...

(gdb) step

Similar to "step," the "next" command single-steps as well, except this one doesn't execute each line of a sub-routine, it just treats it as one instruction.

(gdb) next

Print Variables:

The print command prints the value of the variable specified, and print/x prints the value in hexadecimal:

(gdb) print my var

(gdb) print/x my var

Setting Watchpoints:

Whereas breakpoints interrupt the program at a particular line or function, watchpoints act on variables. They pause the program whenever a watched variable's value is modified. For example, the following watch command:

(gdb) watch my_var

backtrace - produces a stack trace of the function calls that lead to a seg fault (should remind you of Java exceptions)

where - same as backtrace; you can think of this version as working even when you're still in the middle of the program

finish - runs until the current function is finished

delete - deletes a specified breakpoint

info breakpoints - shows information about all declared breakpoints

GDB Example:
gdb/General_Debug - sample debugging examples.

5. Mini Project :

1. Assume that you are working on a calculator project contained in the directory calculator. under this directory there are several subdirectories with names "lib","doc","include“,"src“ and “bin”.
2. The objective is to create a top level Makefile in the calculator directory and each of the source code directories: lib and src.
3. The include directory contains the header files for the project and does not need a Makefile as these don't have to be compiled separately.
4. When make is invoked in the calculator directory it should automatically invoke the Makefile in each of the src directories automatically and build a library in the "lib" directory and two executables in each of "src" directory.
5. Create any source files with all their header files in the include directory.
6. **GNU Build System supports the following configurations:**
    - DEBUG_BUILD; STATIC_BUILD; ADD ; SUB; MUL ; DIV;