Build and debug

1 Introduction

This document outlines a course on build and debug of software. This course is based on Gnu tools in a linux environment. The objective of this course is the following.

1. Build awareness of compiler options
2. Familiarize with compiler output like map file, list file, elf file, shared library etc and tools to inspect them
3. Build knowledge on Gnu make and build ability to write simple makefiles
4. Build awareness of Gnu build tools (autotools)
5. Learn basic usage of Gnu Debugger (gdb)
6. Learn basic usage of gcov, gprof and valgrind.

2 Resources

1. "Using the gnu compiler collection" at https://gcc.gnu.org/onlinedocs/gcc/. This can be referred to learn the commonly used command line options. No need to go through the entire documentation. Only specific topics mentioned in course outline below are enough.
2. The open book from Oreilly " Managing Projects with GNU Make, 3rd Edition" at https://www.oreilly.com/library/view/managing-projects-with/0596006101/. Only the "Basic concepts" part needs to be studied.
3. The following tutorial for auto tools. https://www.lrde.epita.fr/~adl/dl/autotools.pdf
4. The following tutorial for auto tools. http://buildsystem-manual.sourceforge.net/. Only specific chapters mentioned in course outline below are required.

3 Course outline

NOTE: It is recommended to complete exercise for each topic described in section 4 before studying next topic.

1. For GCC, study how to exercise the following options/ features
   a) Run only the pre-processor and exit
   b) Run up to the compiler and exit. Do not run assembler.
   c) Run up to the assembler and exit. Do not run linker.
   d) Provide include paths
   e) Define constants to be used by pre-processor
   f) Select libraries to be linked and provide path to search for libraries

g) Provide optimization options

h) Include debug signals in compiler output to enable debugging

i) specify the output file name

j) Enable compiler warnings

k) Treat all warnings as errors

2. For gnu make, Study the “Basic concepts” part in #2 in Resources.
3. For auto tools, the objective is to get a very basic overview. Study the tutorial mentioned in #3 in resources. Go through the following chapters in #4 in resources. Chapters 4, 5 and 6.
4. Study how to use gcov from https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. Understand the basic usage.
5. Study the use following utilities and how to use them

   a) ldd

   b) nm

   c) objdump
6. Study the basic usage of GDB from following reference. http://sourceware.org/gdb/current/onlinedocs/gdb/. You may skip advanced topics.

4 Exercises

4.1 GCC exercises with native compilation

1. Create a simple C program as below let this be test1.c. This takes two characters as input arguments does some operation with it (not anything meaningful). There are no standard library functions used nor there are any standard include files.

```
/*              test1.c                  */
#define DIFFERENCE      1
#define FIND_DIFF(a,b) ((a) - (b))

int main (int argc, char *argv[])
{
        if (FIND_DIFF(argv[1][0], argv[2][0]) == DIFFERENCE)
                return 0;
        return 1;
}
```

Exercise appropriate compiler options for the following and in each case inspect the output

a) Provide output file name

b) Run pre-processor only, do not run compiler proper

c) Compile only do not assemble

d) Assemble only do not link

e) Generate assembly listing along with the executable

2. create a sub directory with name ‘include’. Create a file ‘test1.h’ in that and move the ‘FIND_DIFF’ macro to this include file. Include this file in above code. Remove the definition of ‘DIFFERENCE’ constant from the above code. Instead pass it via the command line while invoking compiler. Give include path as an option to compiler. Get the code compiled again.
3. Create another simple C program as below. Let this be test2.c. This does not do anything meaningful.

```
/*          test2.c                 */

volatile int my_global[8] = {1, 2, 3, 4, 5, 6, 7, 8};

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

        a = argv[1][0] + 13;
        b = argv[2][0] + 17;

        if (a > b)
        b = b + 19;

        while (a--)
        b = my_global[a & 0x7] + 21;

        return (b - 23);
}
```

Compile this with ‘compile only do not assemble’ option. Inspect the assembly code and figure out the assembly code corresponding to each block of code in test2.c. Now compile it again with optimization enabled. First try with optimization level 1. Then try level 2 and 3. In each case inspect how the assembly code changes. No need to worry about assembly code syntax, you should be able to figure out addition, comparison, looping etc from the keywords easily.

4. Generate the executable from test2.c. The executable format is known as elf. Run the ‘file’ utility with the executable file and see what it prints. Use the objdump utility and do the following.
    a. List all sections and generate disassembly listing. Inspect the output.
        i. Study what all sections are there and understand what is the use of that.
        ii. Understand what all code sections are run before entering main. Understand the functionality of each of these sections.
        iii. Understand what is being done with stack on entering main and leaving main
    b. Generate a hex file of the program

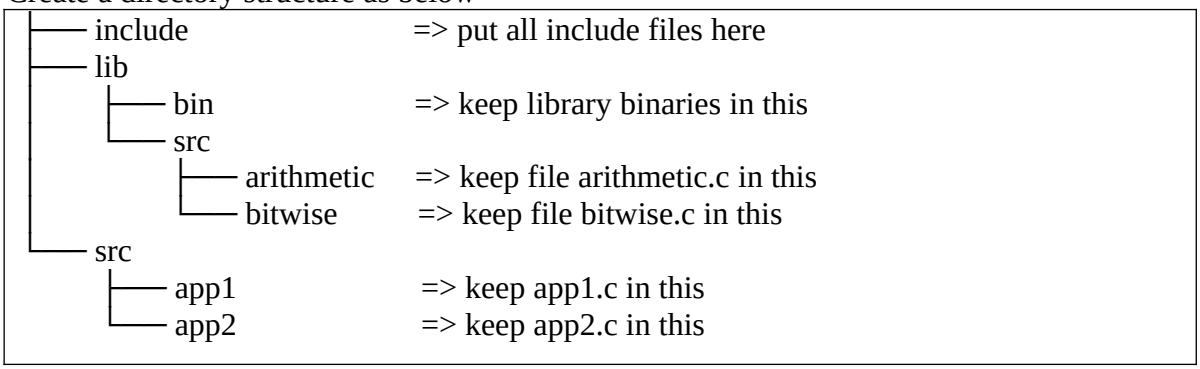
4.2 GDB exercises

1. Compile the program test2.c given in exercise#3 in section 4.1 above. The program expects two characters as arguments. Compile with debug option enabled. Run the program under gdb control giving any two characters as arguments. Explore how to list code lines, how to put breakpoints, how to inspect variables, how to step through code lines, print stack trace etc. Go to main function. Step over each line and inspect the values of variables at each stage.

2. Run the program with not giving any arguments. This will result in segmentation fault. Enable core dump in linux. Run the program again with no arguments and generate a core dump file. Find out how to inspect how to run the core dump file in gdb and take stack trace. Find which line caused the segmentation fault.

4.3 Gnu make exercises

1. Port exercise #1 in section 4.1 to makefile. Take care of the following.
   a. Source file name shall not be mentioned in the Makefile. Shall use wild card.
   b. The root directory name shall be defined as a constant and the include path shall be mentioned relative to this.
   c. The compiler name shall be defined in CC constant
   d. Following targets shall be present. clean, all.
2. Create a directory structure as below



Follow the below instructions

a. create arithmetic.c in lib/src/arithmetic/ implement an ‘add’ function that just adds two integers. Export this function via include/arithmetic.h
b. Similarly create bitwise.c with two functions shift_left and shift_right
c. Implement app1.c as a program that takes two numbers as arguments. It will left shift first number by 2 and then add it to the second number. And then prints the result.
d. Implement app2.c similar way. It will right shift first number by 2 and add to second number. And prints the result

Now create a makefile structure as below.

e. There is common makefile include file in include directory. This will have all common definitions and common compiler flags and include paths.
f. There is a makefile in each app directory that will build each app.
g. There is a makefile in lib/src/ directory that will go to all directory and create corresponding library by name lib<dir_name> in lib/bin directory
h. There shall be a top level makefile in the root directory that will descent into each app directory and lib/src directory and build the corresponding targets. Lib shall be build first.
i. The library shall be built as a static library.

j. The makefile target executable/ library names shall not be hard coded. They shall be created using appropriate makefile built in symbols or wild cards.

3. Repeat the above exercise #3 with the following modifications

a. The libraries are built as shared libraries
b. The libraries are build only if makefile is invoked as ‘make lib’
c. Create a ‘install’ target in top level makefile that will install the shared library in /usr/lib folder
d. run ‘nm’ utility on the shared library and see what symbols get exported.
e. run ‘ldd’ utility on the shared library and see what information is printed.

4.4 Cross compilation exercises

1. Install gcc-arm-none-eabi toolchain from ARM website. Update the shell PATH variable to include the installation path of the tool chain.
2. Find out how to implement custom linker script. Create a simple linker script with separate memory segments for text, read write data and read only data.
3. Get the file mentioned in exercise #1 in section 4.3 cross compiled using make with CC set to the cross compiler.
4. Generate a map file and see what information is available in that

4.5 Auto tools exercise

1. port the exercise #2 in section 4.3 to auto tools

4.6 Other exercises

1. Study how to use gcov tool. Run the test2.c program with gcov.
2. Study how to use gprof tool Run the test2.c program with gprof.
3. Install valgrind software. Run the test2.c program with valgrind. Understand what information is available from valgrind report.