Linux Development Tools

Faculty: Rajesh Sola, Srinivas.K & Bharath.G

L&T Technology Services

LTTS GLOBAL ENGINEERING ACADEMY

Content

1 GNU Tools

2 Make files

3 Library Linking

4 GNU Debugger

Topic Checklist

- Understanding Tool chain
- GNU Tools
- Makefile
- Static Libraries & Linking
- Dynamic Libraries & Linking
- Static Analysis of Code
- Debugging Tool

Software or Packages Required

- Build essential (GNU tools)
- Valgrind
- gdb
- Make
- git

Tool chain

Set of Software development tools, linked (or chained) together by specific stages

- Preprocessor, Compiler, Assembler, Linker
- Debugger, Symbol Table checker, Object Core dump, header analysis, Size analysis

Native Tool chain

- Translates Program for same Hardware
- It is used to make programs for same hardware & OS it is installed on
- It is dependent on Hardware & OS
- It can generate executable file like exe or elf

Cross Tool chain

- Translates Program for different hardware
- It is used to make programs for other hardware like AVR/ARM
- It is Independent of Hardware & OS

GCC

GNU Compiler Collection includes compilers for C, C++, Objective C, Ada, FORTRAN, Go and java.

gcc : C compiler in GCC

g++ : C++ compiler in GCC

To check the gcc Version

- **gcc -v**
- **man gcc** # More info about gcc

C Program Build Process

1) **Pre-processor :**

```
gcc -E filename.c
cpp hello.c -o hello.i
```

2) **Compilation:**

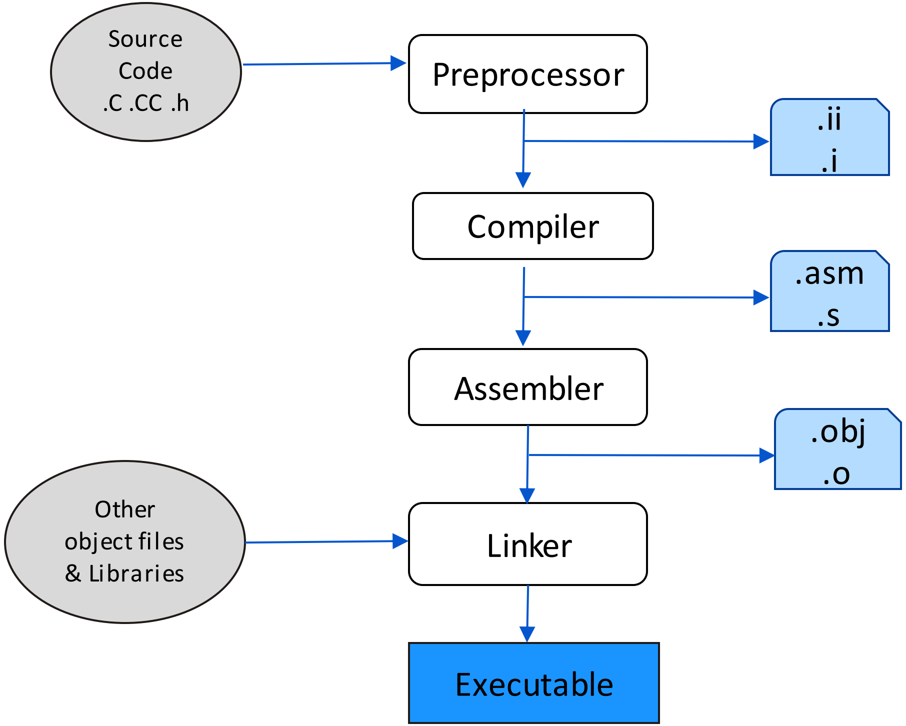
```
gcc -S filename.c
gcc -S hello.i
```

3) **Assembler:**

```
gcc -c  filename.c
as -o hello.o hello.s
```

4) **Linker:**

```
gcc filename.c
ld -o hello.out hello.o ...libraries...
```



Build Using gcc

Build executable:

```
gcc file.c                  # Creates a.out as executable file
gcc file.c -o output             # Creates Output as Executable file
```

Enable all warning:

```
gcc -Wall file.c                 #Enable all Warnings
```

Enable Debugger Support:

```
gcc -g file.c                    #Additional info for debugging purpose
```

Enable Verbose during compilation:

```
gcc -v file.c                    #Info will be printed during compilation
```

Optimizations on gcc

Option	Optimization Level	execution time	code size	memory usage	compile time
-O0	Optimization for compilation time(default)	+	+	-	-
-O1 or -O	Optimization for code size and execution time	-	-	+	+
-O2	Optimization for code size and execution time	--		+	++
-O3	Optimization for code size and execution time	---		+	+++
-Os	Optimization for code size		--		++
-Ofast	O3 with fast non accurate math calculations	---		+	+++

Utilities

file: Determines the type of File

```
file hello.c
file hello.o
file hello.out
```

nm: List Symbol Table of Object Files

```
nm hello.o
nm hello.out
```

ldd: List Dynamically Linked Libraries

```
ldd hello.out
```

strip: Remove the symbol table

```
strip hello.out
```

objdump: Information about object files

```
objdump hello.out
```

Libraries

Collection of pre-compiled object files that can be linked into your programs via the linker.
Example: system functions such as printf() and sqrt().

Static Library:

- The machine code of external functions used in program is copied into the executable.
- Has file extension of **.a** (archive file) in Unixes or **.lib** (library) in Windows.
- A static library can be created via the archive program **ar**.

Pros of Static Library linking

- No need to load additional files before running the executable.
- Runtime is faster.

Cons of Static Library linking

- Larger file size of executable
- Updating library code requires rebuild of whole code.

Libraries

Dynamic/Shared Library:

- File extension is **.so** (shared objects) in Unixes or **.dll**(dynamic link library) in Windows.
- Only a small table is created in the executable.
- When the function is called, on demand OS loads machine code of external functions
  - A process known as dynamic linking.
- Executable file is smaller and saves disk space
  - Most operating systems allows one copy of a shared library in memory to be used by all running programs
- Shared library code can be upgraded without the need to recompile your program.

Because of the advantage of dynamic linking, GCC links to the shared library by default if it is available.

Search Path options for GCC (-I, -L and -l)

Include path for header files “-I” (Upper case I)

- gcc file.c -Ipath -o Output

Library Path “-L”

- gcc file.c -Lpath -o Output

Library Name “-l” (Lower case L)

- For libmath

  gcc file.c -o Output -lmath

Default Include-paths, Library-paths and Libraries

- cpp -v
- gcc -v hello.c -o hello.out # Lists the Libraries and Paths while compiling

Define Macro “-D”

- Usage : **-Dname=value**
- Value should be enclosed in double quotes if it contains spaces

Environment Variables used by GCC

PATH:

For searching the executables and run-time shared libraries.

CPATH, C_INCLUDE_PATH or CPLUS_INCLUDE_PATH :

For searching the include-paths for headers.

It is searched after paths specified in -I<dir> options.

LIBRARY_PATH, LD_LIBRARY_PATH:

For searching library-paths for link libraries.

It is searched after paths specified in -L<dir> options.

Variables to select the build tool:

CC, CXX, LD, AS

Flags for the build tools:

CFLAGS, CXXFLAGS, LDFLAGS, ASFLAGS

Static Library Linking

Static Library:

```
gcc sum.c -c
gcc sqr.c -c
ar rc libsimple.a sum.o sqr.o

gcc test.c -c
gcc -L. test.o -o s1.out -lsimple
gcc -L. test.o -o s2.out -lsimple -static
```

fun.h

```
#ifndef __FUN_H
#define __FUN_H
int sum(int x, int y);
int square(int x);
#endif
```

sum.c

```
#include "fun.h"
int sum(int x, int y) {
  return x + y;
}
```

square.c

```
#include "fun.h"
int square(int x) {
return x * x;
}
```

test.c

```
#include "fun.h"
#include <stdio.h>
int main() {
  int c, d;
  c = sum(10, 20);
  d = square(10);
  printf("c=%d,d=%d\n",c,d);
  return 0;
}
```

Analysis of Static Library

Static Library:

```
file libsimple.a
nm libsimple.a
objdump -d libsimple.a   # -t
readelf -t libsimple.a

file s1.out s2.out
ls -lh s1.out s2.out
size s1.out s2.out
ldd s1.out
ldd s2.out
file s1.out s2.out
objdump -t s2.out
```

```
strip s2.out
objdump -t s2.out
ls -lh s2.out
size s2.out
objdump -t s2.out
strip s2.out
objdump -t s2.out
ls -lh s2.out
size s2.out
```

Dynamic Library Linking

Dynamic Library:

```
gcc sum.c -c
gcc sqr.c -c
gcc -shared -o libsample.so sum.o sqr.o

gcc test.c -c
gcc -L. test.o -o d1.out -lsample
LD_LIBRARY_PATH=. ./d1.out
```

fun.h

```
#ifndef __FUN_H
#define __FUN_H
int sum(int x, int y);
int square(int x);
#endif
```

test.c

```
#include "fun.h"
#include <stdio.h>
int main() {
  int c, d;
  c = sum(10, 20);
  d = square(10);
  printf("c=%d,d=%d\n",c,d);
  return 0;
}
```

sum.c

```
#include "fun.h"
int sum(int x, int y) {
  return x + y;
}
```

square.c

```
#include "fun.h"
int square(int x) {
return x * x;
}
```

Usage of ldconfig to link a dynamic library

What if we want to install our library so everybody on the system can use it?

- Put the library in a standard location - /usr/lib or /usr/local/lib

```
sudo cp libsample.so /usr/lib
sudo chmod 0755 /usr/lib/libsample.so
sudo ldconfig
unset LD_LIBRARY_PATH
gcc -o test.o d1.out -lsample
```

Versioning of shared object files

```
gcc -shared -Wl,-soname,libsample.so -
o /usr/lib/libample.so.1.0.1 sum.o sqr.o
ln -s /usr/lib/libample.so.1.0.1 /usr/lib/libsample
ln -s -f newfile symlink
   #To update the File that symlink points to
ldconfig -n path
```

Building using Makefile

test.c

```
#include "fun.h"
#include <stdio.h>
int main() {
  int c, d;
  c = sum(10, 20);
  d = square(10);
  printf("c=%d,d=%d\n",c,d);
  return 0;
}
```

fun.h

```
#ifndef __FUN_H
#define __FUN_H
int sum(int x, int y);
int square(int x);
#endif
```

square.c

```
#include "fun.h"
int square(int x) {
return x * x;
}
```

sum.c

```
#include "fun.h"
int sum(int x, int y)
{
  return x + y;
}
```

Makefile

```
all.out : test.c sum.c sqr.c
    gcc test.c sum.c sqr.c -o all.out

run:all.out
    ./all.out

clean:
    rm all.out
```

Special Variables and User Variables in Makefile

$@	-Target
$<	-First Dependency
$^	-All Dependencies

Makefile

```
TARGET=all.out
OBJS=test.o sum.o sqr.o

all:${TARGET}:

${TARGET}:${OBJS}
    gcc $^ -o $@

test.o:test.c fun.h
    gcc $< -c
sum.o:sum.c fun.h
    gcc $< -c
sqr.o:sqr.c fun.h
    gcc $< -c

clean:
    rm -rf *.o all.out
```

Rules in Makefile

Pattern Based Rule:

```
%.o:%.c
    gcc $^ -o $@
```

Make already knows how to generate a .o file

Implicit rule

```
%.o:%.c
```

Makefile

```
TARGET=all.out
OBJS=test.o sum.o sqr.o

all:${TARGET}:

${TARGET}:${OBJS}
    gcc $^ -o $@

%.o:%.c fun.h
    gcc $< -c

clean:
    rm -rf *.o all.out
```

GDB

GNU Debugger

Internal commands in gdb shell

- r - run
- c - continue
- q - quit
- s - step (step in)
- n - next (step over)
- f - run up to finish of function (step out)
- b - break point
- Info break – lists the break points
- display x – display value of x after every step or pause.
- bt - back trace of current function
- up
- down
- return – return from current function

Static and Dynamic Code analysis

Static Analysis Tool: **cppCheck**

Usage:

```
cppcheck path_to_src
```

Dynamic Code Analysis tool: **Valgrind**

Usage:

```
valgrind ./a.out
```

Queries?

Thank You !

L&T Technology Services