Linux depelopment environment
Laboration 5

John-Patrik Nilsson
e-mail: jpatrik.nilsson@gmail.com
Skype: j-p.nilsson

March 7, 2010

1 Abstract

This written report is the summary of a laboration in the course "Linux development environment" taken at Umea university of Sweden. The objective of this laboration was to obtain fundamental knowledge about how to use the gcc compiler and make as development tools. In particular, the makefile used by make is an important part of the compilation procedure, and is therefore the focus of the laboration.

Keywords: GCC, compiler, make, makefile.

2 Laboration

The objective of this laboration were two-fold; to write a makefile which included specific rules to handle provided source code, and to write an explanation of specific gcc compiler arguments.

In essence, this report contains three main points of focus; gcc compiler arguments, basic use of makefiles, and the custom makefile written during the laboration.

3 Gcc compiler arguments and example usage

A compiler is used to produce executable binary files (programs) from source code typically written by programmers or software engineers. One such compiler is the GNU project C and C++ compiler called gcc. Gcc is the standard C-type language compiler of almost all modern Unix-based operating systems, including Linux, the BSD-distributions, as well as Mac OS X. The gcc compiler is distributed by the Free Software Foundation under the GNU Public License.

- gcc ... -o outputfile Specify the file in which to place the output of the command.
- gcc ... -E Perform only the preprocessing stage.
- gcc ... -S Stop after compilation proper, before assembling.

- gcc ... -c Compile and assemble source files and exit before the linking stage.
- gcc ... -v Verbose output, print commands executed to run the compilation stages, also writes the version numbers of the compiler and processor programs. Output is sent to standard error.
- gcc ... -ansi Used to control the dialect of C that the compiler accepts. This option turns off certain options of the compiler that are incompatible with the ISO C90 or C++ standards when compiling C or C++ respectively.
- gcc ... -pedantic Issue warnings demanded by the strict ISO standards for C and C++, and rejects those programs in addition to programs that use forbidden extensions.
- gcc ... -D name Predefine name as a macro, with definition 1. This is a preprocessor option, it will be applied on each C source file before actual compilation.
- gcc ... -L dir Add the directory "dir" to the list of directories to be searched for libraries.
- gcc ... -I dir Add the directory "dir" to the list of directories to be searched for header files. Directories named by -I are searched before the standard system include directories. If the directory added by this option is a standard system include directory this option is ignored. Thusly, this option can be used to override system header files, by substituting it with another one.
- gcc ... -Wall This option enables all optional warnings which are desirable for normal code.
- gcc ... -g Produces debugging information in the operating systems native format.
- gcc ... -pg Generate extra code to write profile information suitable for the analysis program gprof.
- gcc ... -O The following options control the level of compiler optimizations. Enabling optimization means that the compiler will try to improve the performance and/or the size of the program at the (possible) cost of compilation time and/or the ability to debug the program. When optimizing with these options the compiler enables "unit-at-a-time" functionaltity, which means that the compiler keeps information it has gained while compiling earlier functions of that program when trying to optimize the current function. This causes the compilation to consume more memory and take longer time to complete. If using multiple -O options, the last such option is the only effective one.
- gcc ... -O0 Default. No optimization. Fastest compilation time and makes debugging produce the expected results.
- gcc ... -O1 Compiler tries to reduce code size and execution time of the program, but (possibly) causes the compilation to consume more memory and/or take longer time.

- gcc ... -O2 Adds nearly all supported GCC optimizations (including -O1) that do not involve gaining speed at the cost of space. In comparison to the -O1 options it increases both performance and the compilation time of the generated code. These optimizations does not include "loop unrolling"[1] or "function inlining"[2]. In Gentoo[3], -D_FORTIFY_SOURCE=2 is set by default, and is activated when -O is set to 2 or higher. This enables additional compile-time and run-time checks for several libc functions.

- gcc ... -Os Optimize for size. Enables all -O2 optimizations that is not known to increase code size, as well as additional optimizations designed to decrease code size.

3.1 Filename suffixes

For completeness, gcc applies certain filename suffixes depending on the type of file it handles. The used suffixes are as follows.

```
.c (source code which should be preprocessed)
.i (source code which should not be preprocessed)
.h (header file to be turned into a precompiled header)
.s (assembly file)
.o (machine code executable file)
.a (machine code library file)
```

4 Make and the makefile

The GNU make development utility is a tool for maintaining software projects. A fundamental part of the make utility is the makefile. A makefile is read by make at run time and is used to organize the different sources of code and how they should be treated.

4.1 Usage

Make is used together with a specified makefile. When make is run, it will look for a file named makefile or Makefile unless otherwise instructed, and will use that file as it's configuration. The makefile should consist of rules (the procedure) to "make" specific targets. Sometimes one piece of software is required to build another one, and that's called a dependency. The rules in the makefile also states the required dependencies to build a target. The basic layout of the rules in the makefile is as follows.

```
target : dependency1 dependency2 ...
[tab]  system command
```

[1] Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. This option makes code larger and may or may not make it run faster.

[2] Integrates all simple functions into their callers. Decreases overall size of the generated code.

[3] A source based "rolling release" linux meta-distribution which uses the Portage package management system to handle software. Unlike conventional distributions Gentoo Linux typically build the source locally according to chosen configuration. This report and laboration was produced on a PC running Gentoo Linux.

Most makefiles contain some standard targets for make, such as "all", "clean", "install", but since everyone is free to create their own makefile how they want, such rules are never guaranteed to exist. However, when make is issued without any arguments, it will look for the rule to make "all" inside a file named "makefile" or "Makefile", so it makes sense to have one such rule.

Rules are the fundamental part of a makefile, but it could also contain other things, such as variables and paths. Variables are very handy and could be used to create very dynamic and small makefiles. Variables can be created by the author of the makefile but there are also buit-in variables. Variables are created like so.

```
VARIABLE = value
```

The variables can thereafter be reached by the syntax $(VARIABLE).

And here follows a few built-in variables which could be very useful, and their respective explanations.

```
$@ (The filename representing the target)
$% (The filename element of an archive member specification)
$< (The filename of the first dependency)
$? (The names of all dependency that are newer than the target,
    separated by spaces)
$^ (The filenames of all the dependencies, separated by spaces,
    containing no duplicates)
```

By default make searches for targets and dependencies in the current directory, where the makefile resides, but make can be told to search for those resources in other places on the filesystem via special path variables.

```
VPATH = src (look for resources in the src directory)
vpath %.c src  (look for *.c files in the src directory)
vpath %.h include  (look for *.h files in the include directory)
```

4.2 The laboration makefile

See the appendix for the actual code listing.

The laboration contained in total seven files as listed below.

- file.h
- file.c
- gen.h
- gen.c
- stat.h
- stat.c
- main.c

These were sorted by the filename type and ordered into separate directories. The header files (.h) were put into a directory named "include" and the C source files (.c) were put into a directory named "src". The makefile were created in the laboration root directory together with the two directories. To help make locate the different resources the vpath variables were used and directed to point towards the right directory. VPATH was not used because vpath is more effective; it points towards exactly one directory to search in for the appropriate files. However, for the header files this was not enough, gcc had to be invoced with the -I option (include) and direct it to the location of the header files.

The makefile was written with the intent to remove as much duplicate input as possible (less lines), not for readability. This makes the makefile very dynamic and easier to adapt to new projects, but might be harder to work with from a users perspective. To achieve the dynamic nature of the makefile its built-in variables were used to great extent.

In essence, the makefile consists of two targets and their rules, and the rest of the targets are pulled in by those. The two important rules are:

```
$(BINFILE) : $(OFILES)
    gcc $^ $(LDFLAGS) $@
```

Explanation: The binary, executable, file we want to create named "program1" (as per laboration guidelines) will be created with the help of the object files. When the object files are present the compiler will be called to compile and link all the object files and create the execuable file with the same name as the target of the rule. However, the object files (dependencies) does not exist, so make looks for a way to create them, which it finds with the help of the rule stated below.

```
%.o : %.c
    gcc $(CFLAGS) $<
```

Explanation: This rule states that all the targets with the suffix .o can be created from the file with the same file name but with the suffix .c. For example; the target file.o has the dependency file.c.

The laboration required the makefile to include rules for making "all", "install", "uninstall", and "clean". The rule "all" were identical to the rule to make the binary file as stated above, and the other rules were in essence simple shell commands, summarized below.

- **install** Move the binary file to a directory where the system looks for binary files, in this case a directory named bin in the users home directory.
- **uninstall** Remove the file from the directory in which it was installed as per the above rule.
- **clean** Remove all created files.

A References

The gcc man page. The GNU Make Manual, Edition 0.70, 1 April 2006. *http://developer.apple.com/mac/library/documentation/DeveloperTools/gnumake/make.html*

Urtubia, Hector, Makefile by example. *http://mrbook.org/tutorials/make/*

Mecklenburg, Robert, Managing Projects with GNU make, 3rd Edition, O'Reilly, November 2004, ISBN: 0-596-00610-1. *http://www.linuxdriver.net/make3/main.html*

B File: makefile

```
/* File: makefile */
CDIR = src
HDIR = include
INSTALLDIR = ~/bin
CFLAGS = -c -Wall -I $(HDIR)
LDFLAGS = -o
OFILES = main.o file.o gen.o stat.o
BINFILE = program1
vpath %.c $(CDIR)
vpath %.h $(HDIR)

all : $(BINFILE)
$(BINFILE) : $(OFILES)
    gcc $^ $(LDFLAGS) $@
main.o : file.h gen.h stat.h
file.o : file.h
gen.o : gen.h
stat.o : stat.h
%.o : %.c
    gcc $(CFLAGS) $<
install : $(BINFILE)
    mv $< $(INSTALLDIR)
uninstall :
    rm $(INSTALLDIR)/$(BINFILE)
clean :
    rm $(OFILES) $(BINFILE)
```