CSE231: Operating Systems

Assignment 0.1

Name: Mihir Chaturvedi
Roll no.: 2019061
Branch: CSE

There are four steps or stages in the entire compilation and executable-generation process that GCC, or any compiler, performs. These are:

1. Preprocessing
2. Compilation
3. Assembly
4. Linking

The final result is an executable file that can be directly run.

GCC Command Line Options

For general purpose use of preprocessing/compiling/assembling/linking input C files, the GCC command must be of the form:

```
gcc [options] <input-files>
```

To get an output in a separate file:

```
gcc [options] <input-files> -o <output-files>
```

If no options are specified (using flags or otherwise), the gcc compiler generates an executable file from the specified input file to the specified output file. If the output is not specified, it by default produces a file with the extension ".out" and same file name.

1. **`-E` flag:** Stop after preprocessing

   This flag halts the compilation/assembly process of the input file right after preprocessing of the input files. The preprocessed file output is either sent to the standard output (`stdout`), or the the output file, specified in the command using the `-o` flag.

2. **-S flag:** Stop after compilation

   This flag compiles the input files (the user-written C code, or the preprocessed files) down to assembly code, but does not assemble them. By default (unless particularly specified), this command replaces the .c/.o/.i/etc. extensions of the input files and replaces them with ".s".

3. **-c flag:** Stop after compilation and assembly

   This flag preprocesses and/or compiles and/or assembles the given input files but does not link them. By default, for each input file, an "object" file is outputted which is a binary that is not human-readable, and has the extension ".o".

4. **-o *file*:** Place output in *file*

   This flag is used to specify the output file of the command, and whatever the command produces as output. In some cases, the primary output stream is the standard output (stdout) which prints out the contents in the terminal itself. We can route the output into a file using this command.

Output File Descriptions

There are five files that we need to deal with:

hello.c	User-written C code
hello.i	Intermediate/preprocessed file
hello.s	Compiled, assembly language file
hello.o	Assembled, object file
hello	Linked executable file

1. Preprocessing stage

   The main aim of this stage in the entire executable-generation process is to convert the rough, user-written C code (high level language, HLL) to its completely pure form, a pure HLL. The output code of the step is still essentially C code, but with a lot of processing done so as to make the compilation process easier for the machine. This stage:

   i. Removes comments and unnecessary whitespace. All comments are replaced with spaces. Long runs of blank lines are discarded.

ii. Expands macros and replaces their instances with what they were defined with.

iii. Includes header files. With this, the contents of the dependency header files of the input files are included in the pure HLL code.

Source file name and line number information in these files is conveyed by lines of the form `# linenum filename flags`. These are called *linemarkers.* Linemarkers indicate that the following line(s) have been originated from this *filename* at this *line number*. There are four types of flags: '1', '2', '3', '4'. As per the GNU GCC documentation:

> '1': This indicates the start of a new file.
>
> '2': This indicates returning to a file (after having included another file).
>
> '3': This indicates that the following text comes from a system header file, so certain warnings should be suppressed.
>
> '4': This indicates that the following text should be treated as being wrapped in an implicit extern "C" block.

Conventionally, these files have the extension '.i' for "intermediate".

2. <u>Compiling Stage</u>

   This stage compiles the preprocessed, intermediate, "pure" HLL (the C code) into a step closer to the final executable file: assembly code. Assembly code is the lowest level of human-readable and human-writable code that we can work with.

   Assembly code directly deals with the memory management, and how different variables need to work with different registers in the memory. It uses mnemonics such as `movl`, `call`, `subq`, along with operands that are either registers, references to registers or immediate values, that refer to opcodes, or the machine language operations that need to be performed.

   Conventionally, and by default, these files have the extension '.s'.

3. <u>Assembling Stage</u>

   Once the code has been compiled down to assembly language, it needs to be converted into "machine code", which is a binary that only machines understand (doesn’t have standard text encoding). The assembly files are assembled and an “object” file is created.

In this step, only the input files are assembled into object files, but their dependencies (or header files, like “stdio.h”) are yet not included. Their definitions need to be “linked” into one cohesive, final, file.

Conventionally, and by default, these object files have the extension ‘.o’.

4. <u>Linking Stage</u>

   This is the final step of the entire compilation process. Multiple object files, that might contain references to each other, need to be “linked” to produce one, united file that can be readily and immediately executed by the system.

   In this process, all references to external functions in either separate header files or libraries are located and brought together. Apart from this, the linker adds additional code to the final executable to handle more intricate system-dependent functions and operations, such as properly running and starting the program once requested to start, gracefully handling an unexpected exit, etc.

   Once again, this is a binary file that cannot be human-readable. By default, gcc generates the executable with ‘.out’ as the extension.