Operating systems. Lecture 6

Michał Goliński

2018-11-06

Introduction

Recall

- Filesystems
- Virtual memory
- Interrupts
- Scheduling

Recall cont

- managing files
- processing text
- pipes
- redirection
- variables
- loops, conditionals etc.
- functions, scripts

Plan for today

- C/C++
- `main` and command line arguments
- compiler on the command line
- using libraries
- `make`

The C language

C

The C language is closely related to UNIX. Although the first UNIX on the PDP-7 machine was written in assembly, when porting the assembly to PDP-11 the authors decide to rewrite UNIX in a higher level language. The language that was available (B) to UNIX authors was

deemed not suitable and Brian Kernighan and Denis Richie created a new one, calling it C (as C follows B). Book by the authors, *The C Programming Language* (1978), is highly regarded and even today is considered a model of technical writing (often called just *K&R*).

C cont.

C was 1989 standardized by the American National Standards Institute, this version is known as C89 or simply ANSI C. The latest version is C18 which is just a clarification a much more extensive revision C11 (from 2011).

The *K&R* has a second edition that was updated for ANSI C, unfortunately not for the newer version.

C++

C++ started its life as "C with classes" – adding object oriented features to the C language. It even used the C compiler: code was *transpiled* to C and only then compiled to machine code. The name translates as "next after C". C++ was first standardized in 1990, the lates version is C++17 with numerous changes/additions with respect to the previous version C++11 and C++14.

C++ cont.

Novadays C++ is a huge language with many features not present in C (in particular namespaces, templates, references and functions overloading). While ANSI C is quite compact (full specification of the language and the standard library is 550 pages, K&R is less then 300 pages), the standard for C++ is over 1400 pages (and often references the C standard). Mastering the subtleties of C++ takes years. Writing a standard-compliant C++ compiler is a huge undertaking.

C and C++

Both languages are closely connected and the most often used compilers for C and C++ are created together. Although technically a C++ compiler is not a C compiler, it comes close.

C and C++ cont.

Both languages are considered mature for demanding jobs: they have excellent compilers and performance of generated code is usually comparable to human-tuned assembly. Compared to other languages there are also drawbacks: both languages are considered verbose (C more than C++) and quite low-level. For example both languages force the programmer to manage memory when newer languages (e.g., Java) do it themselves. Programmer's errors with memory management often lead to security vulnerabilites.

C and C++ cont.

Even though they are higher level than assembly the code written in C/C++ is often not portable between operating systems. Even if a program is portable it needs to be recompiled. Other languages do not have these problems: a Java program can often be run on Windows and Linux without any changes.

C/C++ compilers

Most often used compiler are:

- `gcc`/`g++` (GNU Compiler Collection, free software)
- `clang`/`clang++` (from the LLVM project, free software)
- `icc`/`icpc` (Intel C Compiler, proprietary)
- `cl` (in MS Visual C++, proprietary)

GCC

GCC is a collection of compilers from the GNU Project. A working compiler was one of the first goals of the Project. Today GCC is modern, often the most standard-compliant compiler. It has frontends for multiple languages: C, C++, Objective-C, Fortran, Ada and Go. The Linux kernel is usually compiled with GCC (other compiler may coause ugs or just fail the compilation altogether).

LLVM

LLVM is a novel compiler infrastructure that simplifies writing compilers. The project creates its own C/C++ compiler: `clang`. LLVM simplifies using the compiler inside other tools (e.g., IDEs) and tries to be compatible with GCC (e.g., usually uses the same command line options).

C/C++ IDEs

A compiler should not be confused with an IDE (Integrated Development Environment). This is a program that helps in developing code by integrating many tools used during development (compiler, code editor, debugger, profiler, version control etc.). The team creating an IDE is often not connected to the people behind the compiler, some IDEs allow even to use different compilers.

C/C++ IDEs cont.

The most well regarded IDEs for C++ are:

- Visual Studio (Windows only)
- CLion (by JetBrains, people behind IntelliJ IDEA)
- Code::Blocks
- codelite
- QtCreator

- KDevelop
- Eclipse CDT

IDE features

In an IDE one absolutely wants a working integration with the debugger – the tool that allows to trace program execution line by line and watch the values of variables in memory. If you cannot do that in your IDE you either need to learn that or change the IDE.

IDE features cont.

Other very useful features:

- code completion
- syntax checking (code model)
- integrated documentation viewer
- version control system support
- unit testing support

Programming

To be a successful programmer one needs to learn:

- language syntax
- the standard library of the language
- popular third party libraries
- tools used with the language
- "culture" associated with the language

All of these are important, unfortunately usually tools are not covered in courses, and the last one comes probably with experience only.

Building a C program with `gcc`/`clang`

Source files

C source files usually have the extension `.c` while header files have the extension `.h`.

C++ source files have the extension `.cc`, `.cpp`, or `.cxx`. Headers in the standard library have no extensions while user headers usually also use `.h`.

The `gcc` compiler looks at the extension when deciding how to compile the file.

Stages

Building a C program is carried out in four stages:

- preprocessing
- compiling

- assembling
- linking

In fact there are eight stages (see https://en.cppreference.com/w/c/language/translation_phases), but we will concentrate on these four.

Stages cont.

One needs to know what they do to understand error messages and correct the code. `gcc` allows to stop the process after each stage and look at the resulting files.

Preprocessing

Preprocessor is a simple *macro processor* that works with text. Its commands are called *directives* and start with # (always at the beginning of line). Most often used directives:

- `#define` – defines macros
- `#ifdef, #ifndef` – allows to hide a part of the code if a macro is (not) defined
- `#include` – puts contents of another file (usually a header file) instead of this line (the system keeps multiple headers in `/usr/include`)

Preprocessing cont.

One can stop the build and see the file after preprocessing using the `-E` option for `gcc`. Even the simplest C/C++ programs have usually thousands of lines after preprocessing. One can skip preprocessing altogether by using files with the extensions `.i` and `.ii`.

Compiling

This is the main step – translating the source code from C/C++ into assembly. At this step most optimizations are taking place and many options are present to influence the code that is produced. The most important ones are:

- `-std` – version of the language (`gnu17` and `gnu++17` by default)
- `-O` – optimization level (from 0 to 3)
- `-march`, `-mtune` – allow to indicate the target processor for the program
- `-masm=intel` – emits assembly in the Intel syntax, not AT&T

Compiling cont.

One can stop after compilation using the option `-S`. The corresponding file is saved with extension `.s`. One can skip compilation by using files with this extension.

Assembling

Translating the assembly into machine code. There are very little options as the assembler usually just takes assembly instructions and rewrites them in machine code. One can stop

after this step using the option `-c`. The machine code for the program will be stored in the so-called *object file* with extension `.o`.

Linking

Linking is the final creation of an executable (program or a shared library). For many programmers (even seasoned ones) this is a mysterious process that uses binary files (object files and libraries). We will also not delve deeply into what happens and how it works. Linker (`ld` under Linux). Linker's task is to find the code to all the relevant functions and to make sure all the functions are defined just once. Linker can print errors that are very cryptic for beginners.

Usually the only options passed to the linker are the libraries name and the name of the output file. If the output file is not specified by convention it is `a.out` (it does not depend on the names of the source files).

Adding libraries

Libraries are added to linking using the option `-llibname` (usually with no space). The linker will search for a file `liblibname.so` in the system library directory (usually `/usr/lib` or `/usr/lib64`).

Separate compilation

Our basic programs often live in just one file, but this will change. C/C++ allow (promote even) *separate compilation*: each source file should be compiled into its own object file and thses object files should be linked together. This way, if only one source file is changed only it will have to be recompiled and the program relinked. It can drastically decrease the compilation time.

Executable format

It may be a surprise, but executable files have a specific format they have to obey, otherwise the operating system won't be able to run them. Under Linux the standard format for executables is called ELF. Under Windows the executable format is called PE32+. There are numerous tools that are able to get data and info from binaries. Many of them are included in GNU Binutils:

- `ld`, `gold` – linkers
- `as` – assembler
- `nm` – a tool that reads symbols (function names) from a binary
- `objdump` – displays information from object files and executables
- `strip` – discards unnecessary data from executables (e.g., debug info)

make

Introduction

Programs written in C tend to have multiple files with different dependencies. Say we have four source files `a.c`, `b.c`, `c.c` and `main.c` and two header files `ab.h` and `c.h`. The first header is included in both `a.c` and `b.c`, the second in `c.c`. `main.c` uses both headers. If any of the headers is changed, the files including it should probably be recompiled. The compiler does not know which files to recompile beforehand. This is a task for a tool that helps building programs – the basic one is `make`.

Introduction cont. cont.

The `make`'s usefulness is by no means limited to building programs. It is very useful every time where some files (target files) are created from other files (source files) by some commands and have to be recreated if the source files change.

Makefile

The run of `make` is governed by a file that is usually named `Makefile`. The files contains *rules* that describe dependencies between files and commands that can create them. A typical rule looks like this:

```
target : source1 source2
        command1 to create target from sources
        command2 to create target from sources
```

By tradition commands are preceded by a single good-old TAB character (not 8 spaces etc.). Both sources and recipes are optional, although a rule with neither is not very useful.

Processing rules

When `make` encounters a rule it checks whether a file named `target` exists and whether it is not older than sources by looking at the modification times. If `target` does not exist or is too old, the command (or commands) are run, possibly recreating the target from sources, but this is not checked in any way.

Ordering

`Makefile` is purely declarative – the order of rules does not matter, `make` will use the optimal order to recreate the targets. It is also smart enough to recreate intermediate targets.

No commands

If there are no commands, then the rule just marks a dependency between files. The file will have to recreated by a different rule (e.g., by a pattern rule).

Pattern rules

Pattern rules are very often used. The give a template of a rule.

In a separated compilation we may want to compile each `.c` file to a `.o` file. If we have multiple source files it would be tedious and error-prone to list all the rules. We can make a pattern rule like this:

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

This way if `make` encounters a need for an `.o` file it will know how to create it form a `.c` file.

No sources

If there are no source files in a rule, then the commands in the recipe are run if the `target` file does not exist. If `target` file exists no command is run.

Phony rules

If a rule has no prerequisities and the recipes do not create the target it is considered a phony rule. These are often used to write a sequence of commands that can be easily run if needed. Some phony targets names are conventional:

- `all` – build all the targets
- `clean` – delete all the targets leaving just the source files

Sample

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
```

```
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

Invoking `make`

We can give the name of the target to (re)build when invoking `make`. When no target is indicated on the command line, `make` tries to build the first encountered in the `Makefile`.

Other build tools

Rationale

Writing a `Makefile` is not difficult but tedious. Multiple tools were created that create a `Makefile` that is later used for building the project.

Autotools

The venerable GNU Autotools are a collection of programs and macros that create a `Makefile` from a skeleton file `Makefile.in`. The idea is that as systems differ from one another, the tool can find all the necessary libraries and create a suitable `Makefile` and write variables in a header file to be used by the code. The final machine need not have Autotools installed, the user just runs the `./configure` script and then runs `make` and finally `make install`.

CMake

CMake is a relatively new tool that is often considered better than Autotools. One needs to have it installed on the finale machine and presence of a file `CMakeLists.txt` is a sign that a project is built with CMake. CMake prefers out-of-tree builds and is usually run like this:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ make install
```