Systems Programming
LECTURER

Joao Almeida-Domingues[*]

University of Glasgow

September 28th, 2020 – December 10th, 2020

Contents

These lecture notes were collated by me from a mixture of sources , the two main sources being the lecture notes provided by the lecturer and the content presented in-lecture. All other referenced material (if used) can be found in the *Bibliography* and *References* sections.

The primary goal of these notes is to function as a succinct but comprehensive revision aid, hence if you came by them via a search engine , please note that they're not intended to be a reflection of the quality of the materials referenced or the content lectured.

Lastly, with regards to formatting, the pdf doc was typeset in LaTeX, using a modified version of Stefano Maggiolo's class

[*] 2334590D@student.gla.ac.uk

1 Introduction

1.1 *ILOs - Fundamentals*

The goal of the course will be to deepen your understanding of the fundamental principle and techniques in computer systems, such as:

- Memory and computation as fundamental resources of computing
- Representation of data structures in memory and the role of data types
- Techniques for management of computational resources
- Reasoning about concurrent systems

1.2 *Systems vs Application Software*

We contrast system software with application software. System software can be seen as low-level software, i.e it usually concerns itself with interacting with the machine directly or very close to the metal whilst providing abstractions for application software

There are constraints which come with the nature of system software, such as fast execution time, low memory consumption or low energy usage. Because high-level, managed languages are usually highly abstract, it is almost impossible to build software which abides by these constraints, hence one uses system programming languages like `C` which provide the programmer with more fine grained control over how their program will execute

1.3 *History*

In the 1950s the distinction between these two types of software was nonexisting. Machines were very much purposely built to run certain applications and a single executing application would use the entire machine. *Grace Hopper* was responsible for writing one of the first compilers which turned human readable code into machine code.

Until the 1970s system software was always written in a processor specific assembly language, which meant that for each new processor a systems programmer would need to rewrite the same program in a different assembly language. In the 1970s *Dennis Ritchie* and *Ken Thompson* , while trying to port `UNIX` between two machines, invented `C` as a *portable, imperative* language which supported *structured programming.*

In the 1980s, *Stroustroup* came up with `C++` primarily so that he could automate certain tasks which have to be done manually in C.

Until the 2010s the focus very much shifted towards the development of high-level, managed languages however, the emergency of mobile devices rekindled the interested in new systems languages such as `Swift` and `Rust` which permit the devices to not waste their already scarce resources in tasks such as garbage collection

2 C - Introduction

2.1 *A tale of two languages*

In order to illustrate the advantages of C over managed languages take the following programs

```
x = 41
x = x + 1
```

```
int main() {
        int x = 41;
        x = x + 1;
}
```

2.1 remark. Java syntax was heavily borrowed from C. Programs differ in how they are executed and on how memory is organised

The size (in memory) of `x` in Python is dependent on the architecture, in this computer for example is about 28B, whilst in C is just 4B. Why? Given the dynamic nature of python, its C implementation uses a descriptor object which stores the alongside the value which significantly increases the memory requirements. In C integers are usually 4B and that's all you need to represent `x` in memory
What about the number of instructions? That's even harder to know in python, because for example each operation must type-check first, then it must also check that addition is a valid operation and it represents all of that in some structure in the CPU. In C however, we know that only 3 instructions are required, 1 `add` and 2 `mov`.

2.2 remark. This level of fine-grained control allows us to confidently reason about the execution behaviour and performance of a program

2.2 *Compiling*

The compiling of code is the act of transforming the source code of a program into an executable file which can be run by the OS. There are two kinds of source files in C:

- Header Files : a file containing C declarations and macro definitions to be shared between several source files. Its contents can be requested in a program by including it, with the C preprocessing *directive* `#include`. It is conventional to end file names with `.h`
- Compilation Units : These are `.c` files whose directives are replaced by the contents of the header files. Each file is compiled separately to keep compilation times short

The compilation process can be split into 3/4 main steps:

- Preprocessor : During the preprocessing stage the preprocessor will look at the source code and replace all directives and macros with the contents of their corresponding files. For examples `#include <stdio.h>` will copy the contents of the I/O library `stdio.h`

2.3 remark. To see an example of the output run `gcc -E`

- Compiler & Assembler : The compiler transforms the preprocessed code into assembly code which in turn are turned into *object* files `.o` (almost executable) by the assembler
- Linker : The linker will then find, for each name appearing in the object code, the address that was eventually assigned to that name, make the substitution, and produce a true single binary executable in which all names have been replaced by addresses

2.4 remark. You can use constants by using the `#define` compiler directive which will essentially replace every instance of the named variable with the actual value at preprocessing stage

Warnings & Errors

Compiling errors mean that it is impossible to translate the program into an executable. Warnings on the other end indicate that there is something and unusual in the code which will most likely result in a runtime error. Occasionally one might want to keep the piece of code which threw the warning , in that case make sure to make that clear in your code.

For the coursework make sure that to submit code which compiles without warnings

The compiler and the linker will throw different kinds of errors, being able to identify which is which helps with debugging. For example, it is possible to convert a program into object code which has a call to an undefined function this will just throw a warning. However, when running the linker an error will be thrown and an executable won't be created.

2.5 remark. `-Werror` turns all warnings into errors, the `-Wall` flag enables most compiler warnings