1 Contents

Logbook containing all information related to computer systems learnt in this module

1.1 Articles

2 Overview

2.1 Context

Introduces the basics of the course and what we will cover. Expected to:

- Learn C independently
- Maintain logbook

2.2 Embedded Systems

Expected to be incredibly reliable as they are often used in environments that are hard to access and so have to be able to handle any issues. Examples/properties of embedded systems include:

- Robotics
- Mobility
- Sensor Networks
- IoT

When creating operating systems for these systems, we need to consider power consumption as they often run on limited power

2.3 Baremetal Programming

Essentially all about coding without an operating system. There are no systems in place to handle memory or any services

> Note: When doing realtime systems we also need to remember to introduce interrupts

Baremetal programming (bare metal programming) all about interacting directly with hardware. This is often why C is the first language that gets ported to an embedded system.

Used to create operating systems. This also involves and allows for the devlopments of:

- File Systems
- Drives

Security often highly intertwined with reliability at this stage. More reliable a system is, the more secure it is. We also need to look at power efficiency when baremetal programming to ensure we are within limits

2.4 Operating Systems

Provide a uniform interface for users (and more crucially programs). The operating system's job is to abstract away and complexity of changing hardware limitations from programs. For example, as storage moved from internal memory to floppy disks, the programs that ran on these systems stayed the same and it was the job of the OS to handle the change in hardware while still providing a uniform interface for the program.

2.4.1 Resource Management

Other important functions of an OS include resource management in the form of:

- **Device Access**:
  - Serial Ports
  - IO
  - Printers
- **CPU timing and scheduling**: Previously shceduling was done manually with punch-cards, but now we have algorithms that do this. Important when it comes to looking at realtime scheduling where we can give guarantees of when certain tasks will be completed by
- **Memory**: When allocating tasks to memory and moving things around memory

When looking at resource management, a lot of these resources are not real. We make these resources virtual for efficiency reasons and ensuring the OS is fast and is not consuming too much energy. This means we often place hard limits on the sizes of data structures that these resources can handle.

> Note: As soon as we put a hard limit on something, it becomes a resource that needs to be managed. Even if it is virtual, it can cause a deadlock

2.4.2 Interaction Management

The other thing that an OS is expected to do is to handle interaction management. This can be split up into handling desired interactions and undesired interactions:

- **Desired Interactions**: This might include things like interacting with the network or processes or other users in a certain way (eg. sharing files)
- **Undesired Interactions**: This includes protection and security and the OS is expected to handle this (eg. OS should not allow a user of lower access level to view files of a more senior user)

Undesired actions are essentially actions that the OS is expected to stop and have systems in place to ensure it doesn't happen. Desired actions are actions that the OS is expected to support and provide functionality for.

This is normally less of a thing on embedded systems, as they are small and assumed to be collaborative. This means that we don't normally assume that an embedded system will run a malicious program or have malicious policies

2.5 C Programming Language

Created in the early 70s and used in the PDP11 (which had 24KB of RAM). Just for context, early UNIX needed 12KB RAM. C was used for the creation of early Unix

> ***History***: Back when the world wide web was taking off, programming graduates could earn really good money as everyone needed websites. This meant that programmers switched jobs and there were less programmers in Silicon Valley that could understand all the C++ code everything had been written in. Companies were happy when Java came along, as it came with runtime protection

C was made by assembly coders and created with a very different mindset to things like Pascal (used in academia for proving). Everytime new hardware was released, an operating system would have to be remade from scratch on the new hardware to support the new architecture.

2.5.1 Features

The idea was to create a very simple/efficient language, with which code can be written for an OS that could be compiled to different architectures.

When designing C, the memory limitations were very important as in order to run code, we would need a tiny compiler on top of UNIX which was already 12 KB

C had to have these properties:

- **Minimalistic**: There were no resources. The assumption behind C was that the programmer was smarter than the compiler. Back then compilers did not have algorithms to be smart. Additionally, the compiler had to be really small in order to be ran on top of everything else, so could not even store those algorithms that would make it smart. Also means it was easy to learn and implement. Often first language to get implemented on a new architecture.
- **Pragmatic**: Often described as pragmatic, as designed to solve real problems, instead of being pure (like Pascal - academia focused languages). Examples of this include:
  - printf function had a variable number of arguments depending on the fstring passed which means it was not mathematically pure

    > Note: An fstring, is essentially a string that supports special variables to be passed alongside it. eg. `printf("My first number is %d and my second number is %d", 3, 5)`

  - Variables could be of any size, meaning you could load a file of several MB into a variable despite systems of the time only having fixed memory limitations in the KB range.
- **Portable**: It is able to do this as it is so close to the hardware. Can easily be run when switching:
  - Compilers
  - Boards (eg. Arduino to LaFortuna)
  - Architecture (eg. ARM to …)

2.5.2 Limitations

C also does not have IO included in the core language. Also has no dynamic memory management in the core language. Means when program starts, it does not know how much memory it needs.

Dynamic memory management is when you can request more memory and use more memory as the program is running. Often avoided in embedded systems as we might run out of memory. In an embedded system, if it runs out of memory, it would crash in a way that provides no return value (eg. no BSOD - blue screen of death)

C is notorious for having subtle pitfalls as a result of the mindset behind people who programmed it:

- When initalising a local variable in C, you cannot assume that it is set to 0. This is because in the case, where we don't need the initial value to be 0, we would've wasted an instruction just setting it to 0. C does not waste this instruction, so avoids setting this variable

C also has no safety nets at runtime.

Things that typically would've been programmed in C will eventually be programmed in Rust, as Rust is more reliable (which links to security as mentioned above). Rust provides safer memory, and better concurrency support while still compiling to native machine code to get the best performance.

2.5.3 Syntax and Obfuscation

C has a very terse syntax which means it is comprised of very few keywords. This means people can write code that looks like valid code but is actually doing something malicious (called obfuscation)

2.5.4 When is C used?

C is typically used:

- When there is no better alternative
- Low-level hardware access is required Need privileged CPU access or configuring memory management modes
- Runtime resources are critical > C can be run on systems that don't have external memory such as RAM. It does this by being so efficient, it can fit a few function stacks in the CPU registers alone
- When realtime behaviour is required Easy to predict when things happen because there is no dynamic memory allocation.
- For low capability systems
- Software for new hardware
- Also good for short bits of code that could benefit from optimisation
- Often used in embedded systems

2.5.5 Unspecified, undefined and unexpected behaviours

C has some unspecified behaviours. For example, an int can be of any size. It is only guaranteed to be 16 bits. This was done, because it made more sense for the compiler to decide the size. These unspecified behaviours, were often so that the compiler could decide to compile with an emphasis on memory limitations/speed.

> You can check what your compiler does in the limits.h file Allows you to adapt your code to the unspecified limitations made by the compiler.

Undefined behaviours when you do something that isn't defined in the language or the compiler. Not clear what will happen and should be avoided at all times.

Unexpected behaviour is when the program does not know what is supposed to happen. This often happens because the programmer made an incorrect assumption (eg. ignored precedence, forgot quotation marks)