#uni/semester3/Betriebssysteme/chapter2/c2_Introduction

The processor **fetches** instruction from memory, **decodes** it, **executes** it and then moves to **next instruction**.

Crux of the problem is: How does the OS virtualise its resources?
Why? It makes the system easier to use.

Virtualisation

➜ OS takes a physical resource and transforms it into a more general, powerful, and easy to use virtual form of itself. Thus, sometimes the OS is referred as a **virtual machine**.

➜ Virtualisation allows multiple programs to run on the same CPU, share memory or share disk (CPU, memory, disk = physical resources). Therefore the OS is known as a **resource manager**. If two programs run at the same time and want a resource, which one gets it? This question is answered by a **policy** of the OS.

➜ Policy is a decision maker to optimise some workload

➜ Mechanism is low level code that implements the decision

The OS provides a **standard library** (interfaces, APIs) to applications, which allows them to interact with the hardware.

Program in **Figure 2.1** gets a string as user input, then prints it every second in an infinite loop. In **Figure 2.2** program runs again, but as four different processes running at the same time. This illusion of turning a single CPU into an infinite number and thus allowing many programs to run at the same time, is called **virtualising the CPU**.

Memory is just an array of bytes. To read you need an address, to write you also need a data to be written.

In **Figure 2.3** the program allocates some memory, prints process id and the address of the allocated memory. The dereferenced value is init to 0. In an infinite loop this value increases and gets printed. When multiple programs run this code, the address stays the same, but it increases it independently. What is happening is the OS is **virtualising memory**, which means every program has its own private memory (**virtual address space**).

Crux of the problem: How to build correct concurrent programs?

In **Figure 2.5** we have two threads who increment a common variable = 0 in a loop N times. For low loop iterations the final value of the variable would be 2N. With a high N

you don't always get 2N but sometimes lower. The problem is that you need to load the value, increment it and then store it back. All these don't happen at the same time (aka **atomically**), and I suppose that overwriting happens. This is the **problem of concurrency!** Making sure the OS is behaving correctly despite the presence of interrupts is a great challenge.

Crux of the problem: How to store data persistently?

Data can be easily lost, when power goes away or the system crashes. We need hardware and software to store data **persistently**. The software that manages the disk is called the **file system**. Also the OS wants to share information between programs.

In **Figure 2.6** the programs opens and create and file, write to it, and the closes it, because it doesn't want to write to it anymore. These **system calls** go to the **file system** which handles the the requests and returns some kind of error code. How the OS writes to the disk it kind of complicated, because it needs to find the data on the disk. For performance reason, the OS delays the write requests, with the hope to batch them into larger groups. To handle crashes during writes, the file systems use some kind of write protocol, such as **journaling** or **copy-on-write**, which orders writes to disk, and if a crash happens during the write sequence, the system can recover to a reasonable state afterwards.

Goals of designing an OS:
➜ Performance, minimising the overheads if the OS
➜ Protection between applications. Isolating processes from one another is the key to protection
➜ Reliability, the OS must run non-stop; when it fails, all application running also fail
➜ Energy efficiency
➜ Security, against malicious applications
➜ Mobility

Difference between a system and procedure call, is that a system call transfers control into the OS while simultaneously raising the hardware privilege level. (from **user mode** to **kernel mode**)