Computer Systems

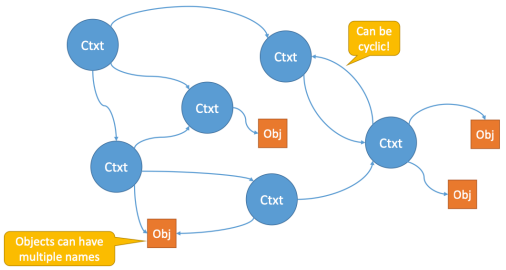
by dcamenisch

1 Introduction

This document is a summary of the 2022 edition of the lecture *Computer Systems* at ETH Zurich. I do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. If you spot any mistakes or find other improvements, feel free to open a pull request at `https://github.com/DannyCamenisch/systems-summary`. This work is published as CC BY-NC-SA.

2 Naming

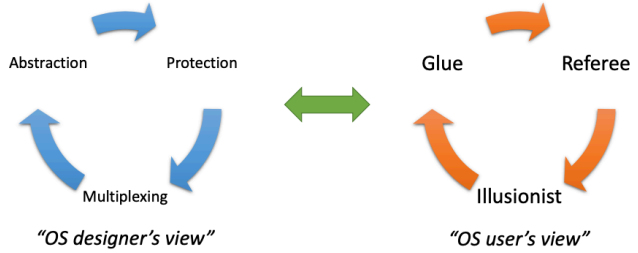
Naming is a fundamental concept, it allows resources to be bound at different times. Names are bound to objects, this is always relative to a context. One example of this would be path names, e.g. */usr/bin/emacs*. Name resolution can be seen as a function from context and name to some object. The resolved object can be a context in itself. This gives us a naming network.



Both synonyms (two names bound to the same object) and homonyms (the same name bound to two different object) can occure.

3 The Kernel

These are the main functions of any operating system. Commonly they are refereed to from a designer's view, but the user's view can be much more helpful to actually understand how it works.



3.1 Bootstraping

The term bootstraping referees to pulling himself up from his own boots. In computer systems it is why we call the process of starting up a computer booting. This boot process looks like this:

1. CPU starts executing code at a fixed address (Boot ROM)
2. Boot ROM code loads 2nd stage boot loader into RAM
3. Boot loader loads kernel and optionally initials file system into RAM
4. Jumps to kernel entry point

The first few lines are always written in assembly, but generally we want to switch to C as soon as possible.

3.2 Mode Switch

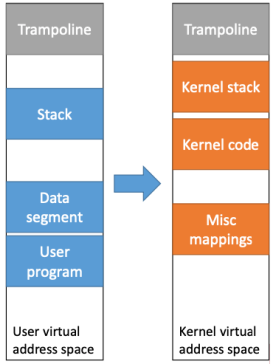
One of our main goals is to protect the OS from applications that could harm it (intentionally or not). For this purpose we introduce two different modes:

- Kernel mode - execution with full privileges, read/write to any memory, access and I/O, etc. code here must be carefully written
- User mode - limited privileges, only those granted by the OS kernel

These two (or more) modes are already implemented in hardware. The main reason for a mode switch is when we encounter a processor exception (mode switch from user to kernel mode). If this is the case, we want the following to happen:

1. Finish executing current instruction
2. Switch mode from user to kernel
3. Look up exception cause in exception vector table
4. Jump to this address

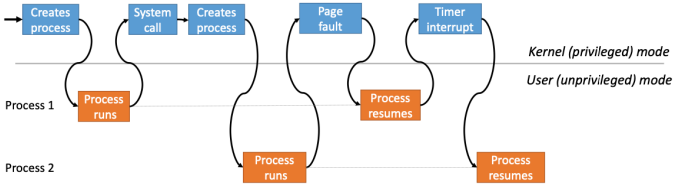
Further we may also want to save the registers and switch page tables. When switching between the modes we also have to change our address space, but we might want to access some informations from the user mode address space. One way of doing this is to use a so called **trampoline**, this is a part of the address space that gets mapped to the same location in user and kernel mode.



Mode switches can also occur the other way around (from kernel mode to user mode). The mains reasons for this are:

- New process / thread start
- Return from exception
- Process / thread context switch
- User-level upcall (UNIX signal)

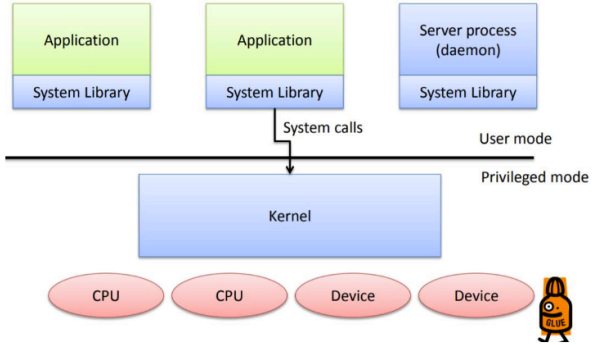
This leads us to the following perspective:



The mode switch is fundamental to modern computers:

- It enables virtualization of the processor
- It creates the illusion of multiple computers
- It referees access to the CPU

3.3 General Model of OS Structure



- Kernel - code that runs in kernel mode
- System Library - interface to kernel, enough for most programs
- Daemon - user space process which provides OS services, varying levels of privileges

We can differentiate between monolithic kernels and microkernels, depending on the amount of code in kernel mode.

3.4 System Calls

System calls are the only way for user mode programs to enter kernel mode. System calls are a type of exception, but they try to look a lot more like a procedure call. Therefore the kernel system call handler has to first locate arguments, copy these arguments into kernel memory, validate these arguments and the copy the results back into user memory after execution. An example of such a system call would be *write()*.

3.5 Hardware Timers

What happens if a user mode program does not cause any exception and does not give control back to the kernel? Hardware timers are a solution for this problem, the hardware device periodically interrupts the processor and returns control to the kernel handler, which sets the time of the next interrupt.

4 Processes

When you run a program, the OS creates a process to execute the program in. A process is an illusion created by the OS. It is an execution environment for a program. This environment gives the program limited rights (access, name spaces, threads, etc.) and therefore it is both a security and a resource principle.

4.1 Creating a Process

There are two main approaches to creating new processes:

- **Spawn** - constructs a running process from scratch
- **Fork / Exec** - creates a copy of the calling process or replaces the current program with another in the same process

Spawn

- Create and initialize the process control block (PCB) in the kernel
- Create and initialize a new address space
- Load the program into the address space
- Copy arguments into memory in the address space
- Initialize the hardware context to start execution at "start"
- Inform the scheduler that the new process is ready to run

Spawn is very complex, we have to specify everything about the new environment. If we omit a key argument a new process might have insufficient rights or resources or it might fail to function due to a security fault.

Fork

Fork on the other hand is less complex. The child process is almost an exact copy of the parent, with a different PID. We know which process we are in from the return value of the *fork()* call (0 for child, > 0 for parent, < 0 for error). The complete UNIX process management API also includes:

- *exec()* - system call to change the program being run by the current process
- *wait()* - system call to wait for a process to finish
- *signal()* - system call to send a notification to another process

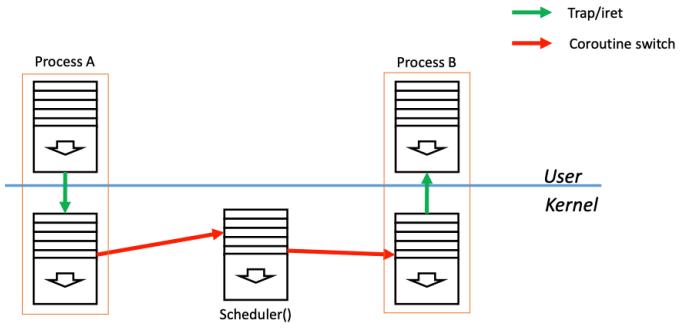
In contrast to *spawn()*, here the child revokes rights and access explicitly before *exec()*, further we can use the full kernel API to customize the execution environment.

4.2 The Process Control Block

The PCB is the main kernel data structure used to represent a process. It has to hold or refer to page table, trap frame, kernel stack, open files, program name, scheduling state, PID, etc.

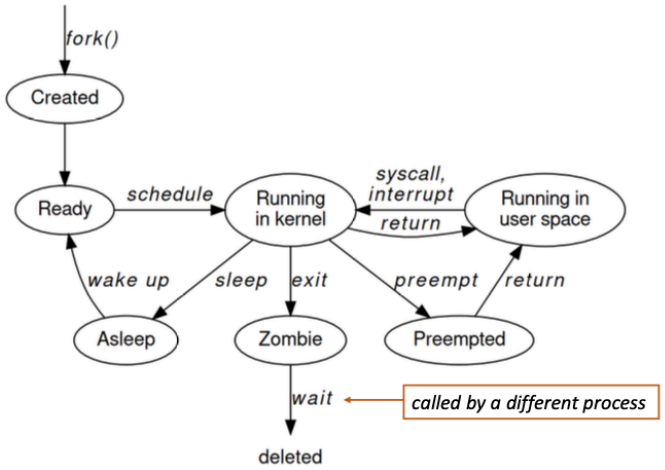
4.3 Process Context Switching

Context switching is the process of switching between different processes running in user mode or kernel mode. It is one of the key elements of the illusion that multiple programs can run in parallel.
First we want to know, when does the kernel switch processes. There are two main reasons for this, first when a process has run for too long and gets interrupted by a hardware timer, secondly when a process blocks. The second case happens when a system call can not complete immediately. The process then often calls *sleep()* and other processes can be executed.



4.4 Process Hierarchy

By forking and spawning new processes we create sort of a hierarchy. If a child process dies, but the parent does not call *wait()*, the child process becomes a zombie - it is dead, but still around since nobody asked for the return code. If a parent dies, but the child does not, the child becomes an orphan and gets reparented to the first process (PID #1, *init*).



The *init* process is basically an infinite loop calling *wait*(NULL), it gets rid of any zombies.