Operating Systems

Timothy Bourke
Notes by Antoine Groudiev

18th February 2024

Contents

Abstract

This document is Antoine Groudiev's class notes while following the class *Systèmes d'exploitation* (Operating Systems) at the Computer Science Department of ENS Ulm. It is freely inspired by Timothy Bourke's classes, and especially its slides.

Introduction

An operating system is made of three main parts: a *kernel*, *libraries*, and *applications*. It is useful for sharing memory, computation time, processors, and devices (keyboards, disks, graphics cards, ...). It constitues an abstraction layer used as a base for building bigger systems. For example, the OS allows hardware independence for applications, and provides common services (such as file systems with access control) and allows concurrency and communication with protection and access control.

The kernel contains lots of data structures and functions used for bookkeeping, as well as abstract modules such as interfaces, data structures, and functions for services. It allows a low-level control of hardware, and manages the concurrency and the events.

1 Virtual Memory

1.1 Introduction

We will start by studying the concept of *virtual memory*. An OS must share finite ressources among multiple users and applications. It provides an abstraction for building such applications, which do not need to worry about the complexity of memory management with other applications. Indeed, physical memory must be shared: but what happens if it runs out, or if one process tries to read or write the memory of another? This is where virtual memory comes in handy: it allows to run each process in a virtual address space, and selectively share memory between processes for them to communicate. Furthermore, it allows processes to use faster physical memory as a cache for files on slower disks.

Virtual memory is an abstraction provided by a sophisticated and elegant mix of hardware and software. Hardware capabilities include exceptions (synchronous interrupts), address translation

(which we will study in this section), main memory and caching of files on disks. Software-wise, we will study the kernel memory system. Hardware is needed to intervene at the lowest-level – each individual `mov` instruction – and for speed. Software is needed to implement sophisticated, flexible algorithms and for pervasive integration within a kernel.

For application programmers, virtual memory is largely invisible. Only very few programmers ever have to deal directly with this low-level hardware and software. Nevertheless, we will study it because it pervades all levels of a computer system; understanding how it works gives a deeper understanding of how the system works. It also provides powerful capabilities that can be exploited in applications. OS programmers cannot avoid knowing about virtual memory.

1.2 Physical and virtual addressing

Physical addressing is used in "simple" systems like embedded microcontrollers, in devices like cars, elevators, digital picture frames, ...

Conversely, *virtual addressing* is used in all modern servers, desktops, laptops and high-end mobile phones. It uses an MMU (Memory Management Unit), which translated the virtual address (VA) to a physical address (PA).

1.3 Virtual memory address translation

A *virtual address space* is a set V of $|V| =: N = 2^n$ virtual addresses used within programs. These addresses are mapped to a *physical address space*, which is a set P of $|P| =: M = 2^m$ physical addresses of DRAM.

The address translation can be formalised by a function $\texttt{VMAP} : V \to P$ `option`. For some virtual address a, we have

$$\texttt{VMAP}(a) = \begin{cases} \texttt{Some } a' & \text{if data at virtual address } a \text{ is at physical address a' in } P \\ \texttt{None} & \text{if data at virtual address } a \text{ is not in physical memory} \end{cases}$$

We can represent this simple model of the function `VMAP`by a data structure accessed by hardware (the MMU), and manipulated by software (the OS).

We could think of individually mapping each byte or word, but it is too complicated in terms of software, and too expensive in terms of hardware and data structure. Thus, the address spaces are divided into numbered *pages*, which are blocks of a specific size, e.g. 4 KiB. They are always aligned on *page size*, i.e. they cannot overlap. The mapping is therefore represented by a *page table*. In first approximation, it is a list of entries, one for each virtual page, specifing either `None`(when there is no mapping), or `Some` p where p is a physical page number.