Operating Systems

Timothy Bourke
Notes by Antoine Groudiev

11th March 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.

1.4   VM as a tool for memory management

The key idea is that each process has its own virtual address space. It can view memory as a simple linear array of bytes. Mapping function scatters addresses through physical memory. Well-chosen mappings simplify memory allocation and management. Each virtual page can be mapped to any physical page. A virtual page can be stored in different physical pages at different times. This allows to share code and data among processes, by mapping virtual pages to the same physical page.
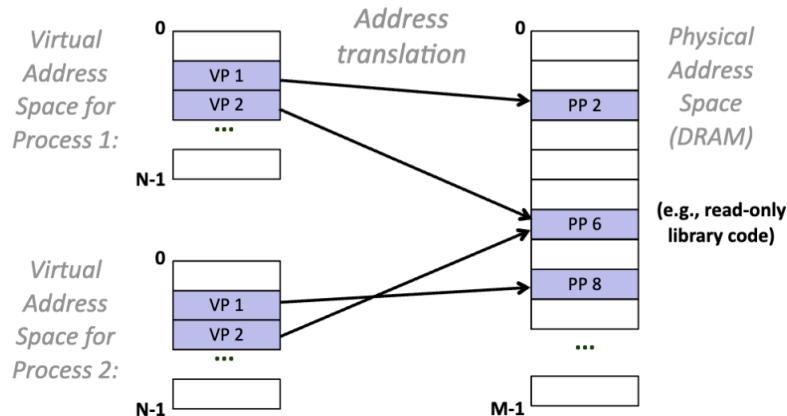

Figure 1: Address translation

1.5   Address translation

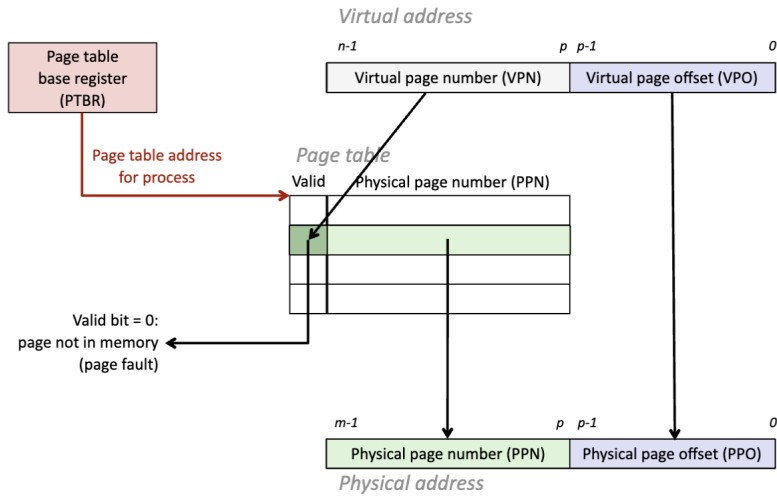The following diagram shows the translation of addresses with a page table:

Figure 2: Use of a page table

The processor stars by sending a virtual address to the MMU, which requests a page table entry address from memory. The page table entry is fetched from page table in memory, and the MMU sends the phydical address to cache or memory. Finally, the cache or memory sends the data word back to the processor. This is summarized in the following diagram:
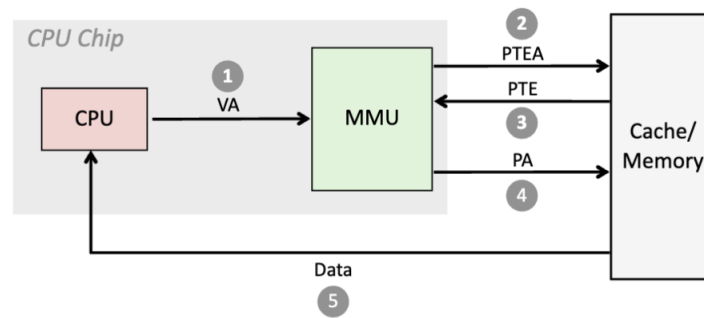

Figure 3: Page hit

Translation can be sped up using a *Translation Lookaside Buffer* (TLB). Page table entries (PTEs) are cached in L1 like any other memory word. PTE hit still requires a small L1 delay, and PTEs may be evicted by other data references. The solution to this is a *Translation Lookaside Buffer* (TLB). It is a small hardware cache in the MMU, which maps virtual page numbers to physical page numbers. It contains complete page table entries for a small number of pages.
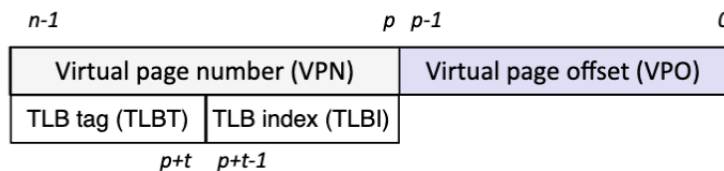

Figure 4: Use of a Translation Lookaside Buffer (TLB)

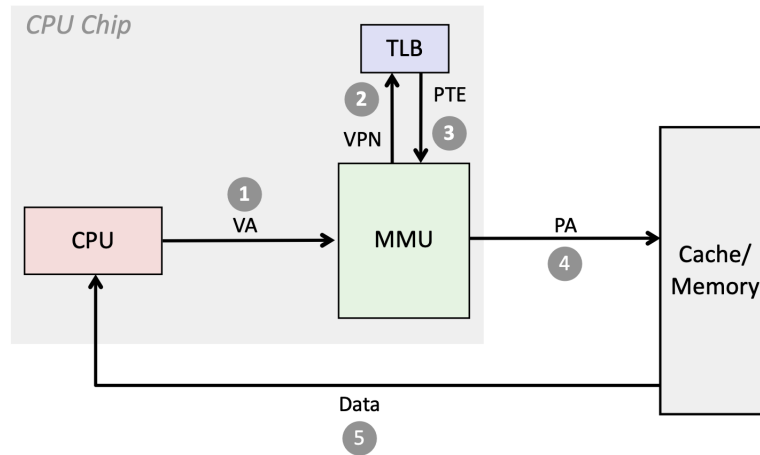The modified hit procedure using TLB is therefore of the form:

Figure 5: TLB hit

Notice that a TLB hit eliminates a memory access. A TLB miss incus an additional memory acces (the PTE). Fortunately, TLB misses are rare.
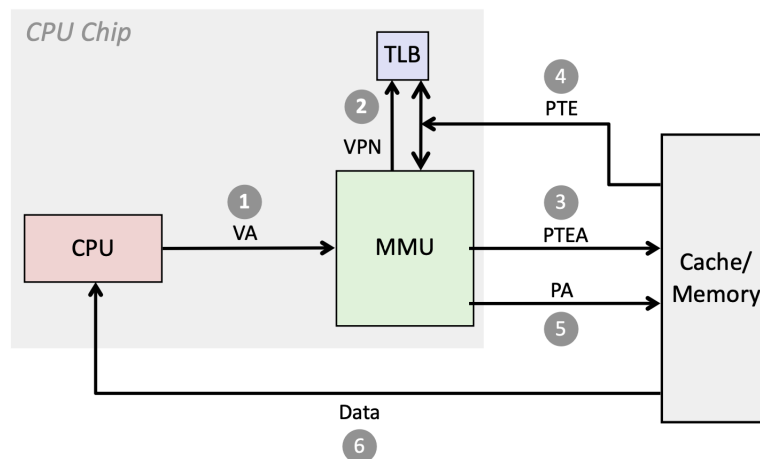

Figure 6: TLB miss

1.6 Kernel and User Mode

Each user process will have its own virtual address space define by a distinct page table. But should user code be allowed to update page tables and virtual memory mappings directly? No! That would not provide any real protection. Therefore, a process has at least 2 operating modes: a *Kernel/Supervisor mode*, in which all instructions and registers are available, and a *User mode*, in which some instructions and registers are restricted. Special instructions or events (interrupts) alloww to switch from one operating mode to the other. Virutal memory mappings can depend on the mode.

In practice, PTEs are extended with permission bits. Page permissions are checked for every move instruction to or from memory. If violated, the OS sends the process the error SIGSEGV (segmentation fault).
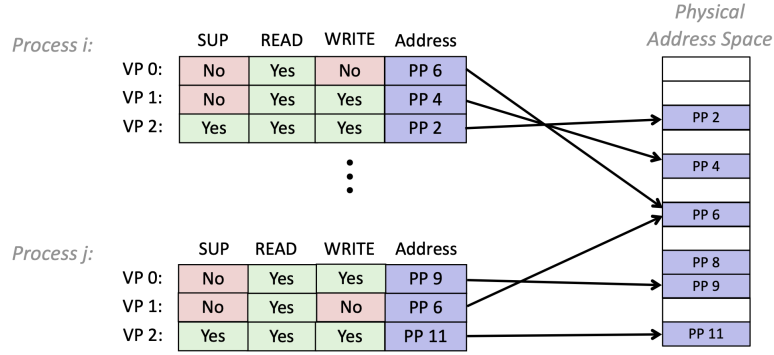
Figure 7: PTEs extended with permission bits

1.7 Multi-level page tables

Suppose that we maintain a 48-bit virtual address space, with a page size of 4 KiB (2^{12}), with an 8-byte PTE. We would need a 512 GiB page table:

$$2^{48} \times 2^{-12} \times 2^3 = 2^{39} \text{ bytes}$$
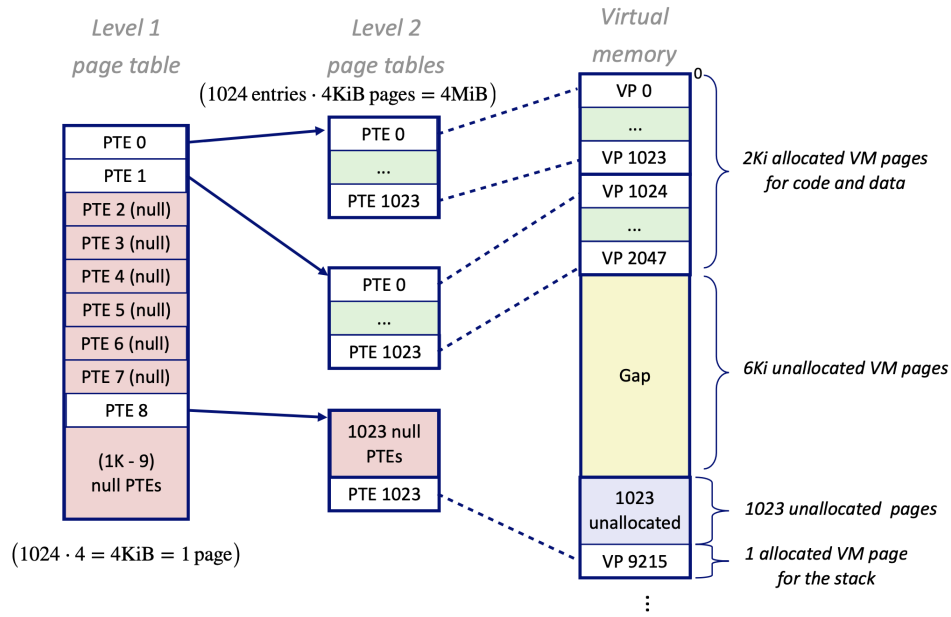
Figure 8: A two-level page table hierarchy

A solution to this is to use multi-level page tables. For example, we could use a 2-levels page table, where in the first level, each PTE points to a page table, and in the second level, each PTE points to a page. This reduces memory requirements. Usually, most of the virtual address space is unallocated. If a level 1 PTE is null, there is no corresponding level 2 page table. The level 2 page tables can be created as needed, reducing pressure on physical memory.

Figure 9: Address translation with a *k*-level page table

This may seem expensive and impractical, but the TLB helps by catching PTEs from the page tables at different levels, and in practice the lookup is not much slower than with a single level.

1.8   x86 real mode and segment:offset addressing

1.8.1   Introduction

The first Intel 16-bit processor, the 8086, was released in 1978. It uses 16-bit words and 20-bit addresses, allowing up to 2^{20}=1 MiB 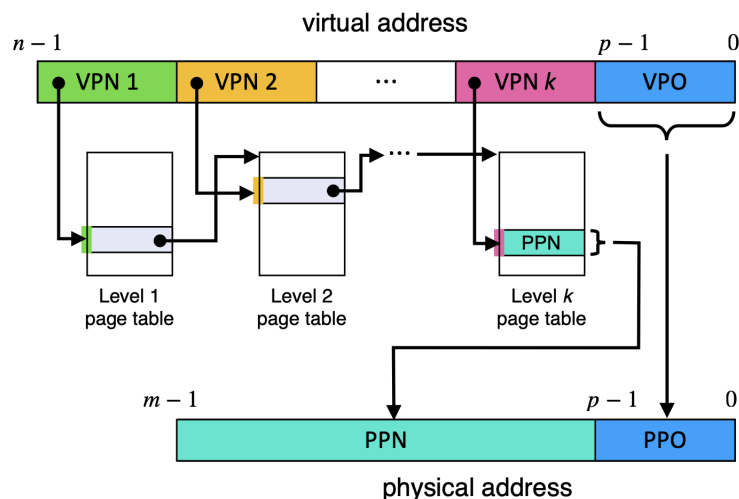of addressable memory. It uses memory segmentation, with no MMU: physical addressing is always used. The following 8088 and 80186 introduces some slight modifications and new instructions. They are the ancestors of the x86 architecture.

1.8.2   The segment:offset addressing

How can we access a 20-bit physical address space with 16-bit registers? It uses special segment registers to provide the extra 4 bits. For example, put 0x12F3 in the data segment register `%ds`. This is the base address of the 64K segment starting at 0X12F30. Notice that the physical base address is the segment address shifted 4 bits to the left. Addresses in moste data instructions are implicitly relative to `%ds`. For example, the instruction `movl 7, %ax` loads the word from 0x12F37 into `%ax`. Putting 0X4B27 in the `%bx` register then executing `movl (%bx), %ax` loads the word from 0x12F30+0x4B27 into `%ax`. Therefore, we have the equation:

$$\text{physical address} = (\text{segment address} << 4) + \text{offset}$$

Different segment:offset addresses refer to the same physical address (e.g. both 0020:0010 and 0000:0210 refers to physical address 0x210). Furthermore, some addresses are too big for 20-bits (e.g. FFF:0010 wraps around to the beginning: 0X00000).

1.8.3   Real mode

All x86 and x86-64 start in real mode. They can only access 1 MiB, with no memory protection or virtual memory. They need segment registers to access more than 64K. Nevertheless, they can access BIOS functions to query and control hardware, and can still use 32-bit registers. The code that "boots" a kernel executes in real mode and later switches to protected mode by

updating CPU registers. There exist techniques for running real-mode processes or switching back to real mode to access the BIOS.

The segment registers are the following:

`%cs`	Code segment register	Used implicitly with the instruction pointer `%ip`, and updated on calls, jumps, interrupts, and exceptions
`%ds`	Data segment register	Used implicitly by most data access instructions
`%es`	Extra segment register	Used implicitly in some string instructions
`%fs`	Far segment register	Free for programmer use
`%gs`	Far segment register	Free for programmer use
`%ss`	Stack segment register	Used implicitly with the stack pointer `sp` and base pointer `bp`

Figure 10: Segment registers

Apart from the implicit effects of the segment registers, programmers can also use them explicitly. For example, `movl $4, %fs=200` moves a 32-bit value representing the number 4 to the address given by the segment register `%fs`, plus 200. Similarly, (movl $4, %es:100(%eax, %ebx, 2)) moves a 32-bit value representing the number 4 to the address given by

$$\texttt{\%eax} + \texttt{\%ebx} \times 2 + 100$$

relative to the segment register `%es`.

1.8.4 Low-level real-mode programming

The PC contains a set of firmware routines called the BIOS – Basic Input/Output System. While many modern systems are based on the UEFI – Unified Extensible Firmware Interface, most also support BIOS services.

The BIOS software executes when a computer is powered on. It checks the installed hardware and configures it. It provides features for configuring machine settings, and goes through the attached disks in order, loading their first sector into memory and checking for a special value. If the value is found, then the sector contents loaded into memory are execture in real mode. This small (512 bytes) program is normally responsible for loading and executing the next boot stage and ultimately the kernel software.

The firmware contains routines that can be called from real mode by setting certain registers and triggering an interrupt, e.g.

```
mov $0x0e, %ah
mov '!', %al
int $0x10
```

specifies the teletype function (0x0e) with argument '!', and triggers the "video service" interrupt to display the character. It is time-consuming to experiment with such low-level programs on real hardware, but it is realtively easy to do in an emulator such as `qemu` or `Bochs`.

1.9 x86 protected mode and memory segmentation

1.9.1 Protected mode

The Intel 80286, introduced in 1982, is a 16-bit microprocessor with a 24-bit address bus (2^{24}=16M). It introduced a *protected mode* and was used in the IBM PC/AT. 8°286 added

address bits A20 to A23, but some 8086 software relied on the wraparound "feature". IBM added a logic gate to the motherboard that allows manually forcing the A20 line on the bus to 0. By default, it is enabled (=0) for backward compatibility; the boot software must manually disable this override (=1). It is still a "feature" of modern x86 and x86-64 processors.

The 32-bit 80386 was released in 1985 as an extension of the 80286. In addition to the real mode, it provides a better protected mode, which protects memory via memory segments, virtual memory and protection via *rings*. This is the main mode of the x86 (IA-32) architecture.

In x86 protected mode, address translation is done in two stages: *segmentation* and *paging*. It translates logical addresses in linear addresses, which are then translated to physical addresses. A *logical addres* is an address used in machine instruction; a *linear address* is simply a virtual address, while a *physical address* is written onto the memory bus to access devices and memory.

Segmentation divides addressable memory into smaller address spaces called *segments*. Compared to the real mode, segments are protected by bounds checking and access rights. Furthermore, a segment is defined by a *descriptor* stored in a *descriptor table*. The DS can be changed with `mov` of the descriptor index, and the CS can be changed with `jmp` or `call`.

1.9.2   Descriptor tables

Descriptor tables can be global (GDT) or local (LDT). An OS can use local descriptor tables for per-process segmentation. For instance, it can use a read-only and executable segment for code, a readable, writable and non-executable segment for data, and another segment for the stack. Or it can just use a single *flat* model together with virtual memory; it is essentially what Linux does, which is simpler, portable and produces less overhead.

Descriptor tables define segmentation. In practice, it is represented by an array of 8-byte *descriptors*, which can be of different types (segment, local descriptor table, task state segment, . . .). Before entering protected mode, a global descriptor table must be installed into the GDTR register using the `lgdt` instruction. A descriptor is specified in 48-bits by a base (virtual address) and a size.

Segment registers index descriptor tables: in protected mode, `%cs`, `%ds`, `%es`, `%fs`, `%gs` and `%ss` hold 16-bit segment selectors. A 12-bit index specifies a descriptor in either the current GDT or LDT, as determined by bit 2. The first two bits specify the privilege level. Instructions implicitly use and set the segment registers similarly to real mode, but linear addresses are calculated indirectly via a descriptor table.

Recap: logical to linear address   A segment register is either specified implicitly (`movl $4, 200`) or explicitly (`movl $4, %fs:100(%eax, %ebx, 2)`). It specifies a descriptor in the GDT or LDT. The (cached) base address of the segment descriptor is added to the offset to determine a linear address in a 4G virtual address space.

1.9.3   Flat segmentation model

The flat model gives access to a contiguous, unsegmented address space. It defines two segment descriptors: one for code (`%cs`), and one for data (via `%ss`, `%ds`, `%es`, `%fs`, and `%gs`). The base address is 0x00000000, and the limit is 4G. X86 Linux uses a variation of this scheme with 4 segments (kernel code and data, and user code and data).

1.9.4   Flat with protection model

Is uses the base addresses and limits to isolate code from data, and also limits access to ranges where physical memory exists. The processor raises a *general protection fault* on illegal accesses. A bit is redundant when using virtual memory.

1.9.5   Multi-segment segmentation model

The model uses all features of x86, with one descriptor table per task. It separates data segments from one another. It is not used in most operating systems, because it is redundant with VM, complicated, and not portable.

1.9.6   The Task State Segment

The Task State Segment (TSS) is yet another data structure for controlling or using the HW. It is specified by the *task register* (TR, instruction `ltr`). It is intended for harware multitasking, but is slow and not portable, and is therefore not used. In protected mode (x86), it stores copies of the segment and data registers. Furthermore, it specifies stack pointers to use on context switches.

1.10   x86-64 long mode

1.11   Summary of x86 processor modes

- Real mode: walks and talks like an 8086

    - How your computer boots up
    - 20-bit address space
    - Direct physical memory access
    - Segmentation available
    - No paging

- Protected mode – 32-bit execution mode

    - Segmentation and paging
    - Privilege levels (user vs. kernel)

- Long mode – 64-bit execution mode

    - Very similar to protected mode
    - Restrict segmentation use
    - Remove deprecated 8086 instructions

1.12   Paging on x86 and x86-64

The basic ideas on address translation and multi-level page tables apply. There is 3 different paging modes: 32-bits paging (x86), PAE paging (Physical address extension), with addresses up to 64G on a 32-bit architecture, and 4-level paging with 64 bit virtual addresses.

Segmentation gives a virtual address in a linear address space. Now, paging translates the virtual address into a physical one.

2   More virtual memory, linking and loading