Tutorial 9 - Paging and memory management

Gidon Rosalki

2025-05-28

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Introduction

We first need to understand some basic units:

- KB = 2^{10} bytes
- MB = 2^{20} bytes
- GB = 2^{30} bytes

For calculations:

$$\frac{4\text{GB}}{8\text{KB}} = \left(\frac{2^2}{2^3}\right) \cdot \left(\frac{2^{30}}{2^{10}}\right) = 2^{20-1} = 2^{19}$$

Understanding binary in depth is left as an exercise to the reader, since we should have all passed either NAND or computer architecture, this should already be understood.

2 Physical or Virtual addresses

2.1 CPU and physical addresses

The CPU can only use data stored in registers or memory. It executes a set of instructions, such as loading from memory to a register, storing from a register to memory, arithmetic operations on registers, and control flow using registers (branch, conditional branch, etc.). Therefore, programs must be brought (from disk) into memory for them to be run.

The **physical address** is the address in the particular storage cell of main memory. The physical address space depends on memory size, if we have 8GB of RAM, then the address length is 33:

$$8\text{GB} = 8 \cdot 2^{30} = 2^{33} \implies \log_2(8\text{GB}) = 33$$

That's why 32 bit OSs can only use 4GB of RAM (excluding methods like Physical Address Extensions (PAE)).
The physical address is also called the real address, or the binary address.
This system has obvious limitations, without further abstraction, we can only run as many programs as we have memory, where each program is given a static amount of the physical memory, and thus we cannot add more programs once we have run out.

2.2 Virtual address spaces

Processes are generally unaware of the memory usage of other processes, and the exact memory usage of their own (either of these would be **horrific** security practices). The OS provides each process with a virtual address space, where the process has (almost) unlimited memory, and the process is only aware of itself, as far as it is concerned, it is the **only** process on this machine. The OS is responsible for mapping the virtual memory of each of the processes into the physical memory. Requests for memory access go through this translation, in a transparent manner to the program. We should note that this virtual address space (also called logical address space) is usually *much larger* than the physical address space.
In a 32 bit system, the virtual address space size is $2^{30} = 4$GB, and in a 64 bit its 2^{64}, which is... a lot. In practice, most 64 bit architectures limit the VAS to 48 bits.
A virtual address space is a series of continuous virtual addresses, though it is not necessarily continuous in the physical address space.

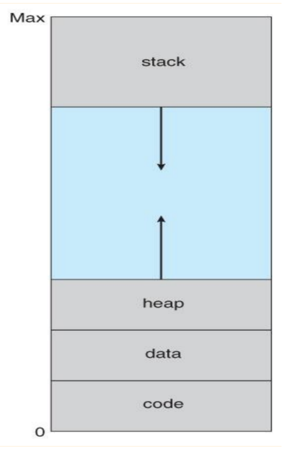

Figure 1: Program virtual address space

2.2.1 MMU

The user program deals with logical addresses, and never sees physical addresses. The MMU is the hardware that maps virtual memory to physical memory. This is done in hardware, because when something is done very frequently, then it is faster (and usually more reliable) to implement it in hardware, than in software.

2.2.2 Paging

The logical address space of process can be non-contiguous; a process is allocated physical memory whenever the latter is available. Physical memory is partitioned into fixed-sized blocks called **frames**. Logical memory is partitioned into blocks of the same size called **pages**. With paging, every time that process 1 runs, some of its memory would be loaded, and other processes would be stored on the disk. The free frames are kept track of, and to run a program of size n pages, need to find n free frames and load program. To do this, one sets up a page table to translate logical to physical addresses. This way, we have avoided external fragmentation of the memory, however there is still likely to be internal fragmentation (within the program that is).

Examples of the model, and frame allocation are available in lecture 8.

The address generated by a CPU is divided into the page number (p), and the page offset (d). The page number is used as an index into a page table, which contains the base address of each page in physical memory, and the page offset is combined with the base address to define the physical memory address that is sent to the memory unit. For a given logical address space of size 2^m words, and page size (normally 4KB) of 2^n words, then our page offset is of maximum size n, and our page number is $m - n$. If n is very large, then this comes at the cost of number of pages, i.e. we cannot have many pages, but they are all very large. However, if n is very small, and m is very big, then we will have a huge number of incredibly small pages.

The page table is ideally kept in the main memory, but the size of the page table is itself usually pretty large, even for a single process. Let us consider a VAS of size 2^{48}, page size of 4KB $= 2^{12}$, and RAM of size 8GB $= 2^{33}$.

$$\begin{aligned} \frac{8\text{GB}}{4\text{KB}} &= 33 - 12 \\ &= 21 \\ &\to 32 \text{ since computers work in powers of 2} \\ 2^{36} \text{ page entries } \cdot 32\text{bits} &= 2^{38}\text{bits} \\ &= 256\text{GB} \end{aligned}$$

That is to say, we need a quarter of a Terabyte in order to store *just the page table.* This is obviously problematic, which we will resolve shortly.

We may add information to the page table:

- Valid/Present bit - this indicates whether the page is assigned to a frame
- Modified/Dirty bit - this shows whether or not the page was modified
- Used/Referenced/Accessed bits - was the page accessed recently (see the clock algorithm for page replacement in lecture 9)
- Access permissions - usually things like read only, read write, etc.

If the valid is off, then the page is not mapped to a frame. This causes an exception named a **page fault**. This causes the OS to search for an available frame. If there is one, then the page is loaded from the disk to this frame, and if there isn't, then the OS pages out (swaps out) a page to the disk (only saving it if the dirty bit was on), brings the requested frame to this location in the memory, and updates the page table accordingly. To choose the evicted page, the OS uses Page Replacement Algorithms, such as the clock algorithm (making use of the used bit).

In order to resolve the issue of the page table being huge, we have two main solutions:

- Hierarchical paging
- Inverted page tables (next week)

2.2.3 Hierarchical page tables

This is also known as multilevel page tables. WE break up the logical address space into multiple page tables. We do not need to preserve all the page tables, only those that we are using. This works well since many times, the programs do not use the middle addresses between the stack and the heap (see Figure 1).
A simple technique is a two level page table:

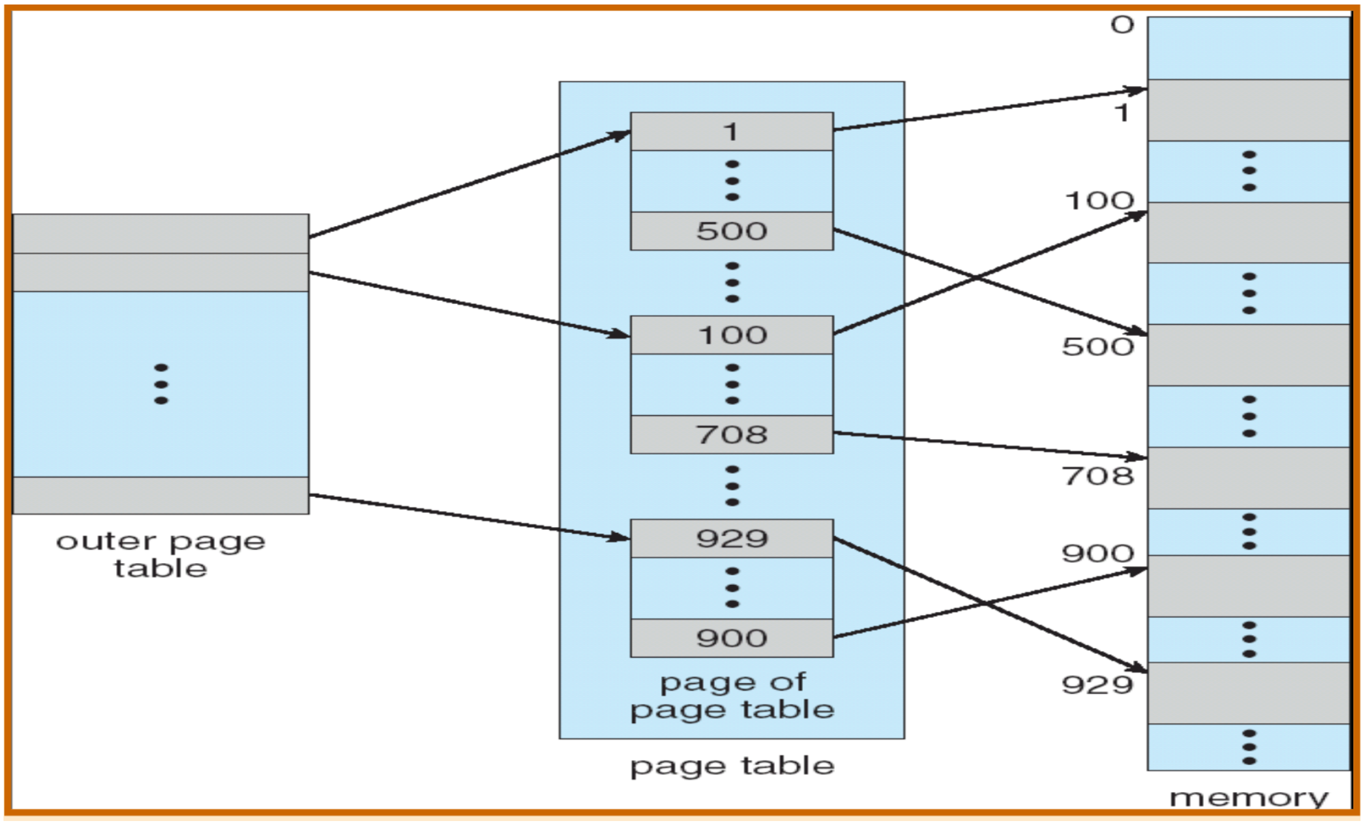

Figure 2: Two level page table scheme

So we can see here, if there are contiguous sections of the memory that are not used, then we need not load the relevant pages in the page table.

Let us consider an example: On a 32 bit machine, with 1K (2^{10}B) page size, a logical address is divided into a page number consisting of 22 bits ($p1 + p2$), and a page offset consisting of 10 bites. Since the page table is itself paged, the page number is divided into a 12 bit page number ($p1$), and a 10 bit page offset ($p2$). Thus, a logical address is comprised of $p1$ $p2$ d, where $p1$ is an index into the outer page table, $p2$ is the displacement within the page of the inner page table, and d is the offset within the page.