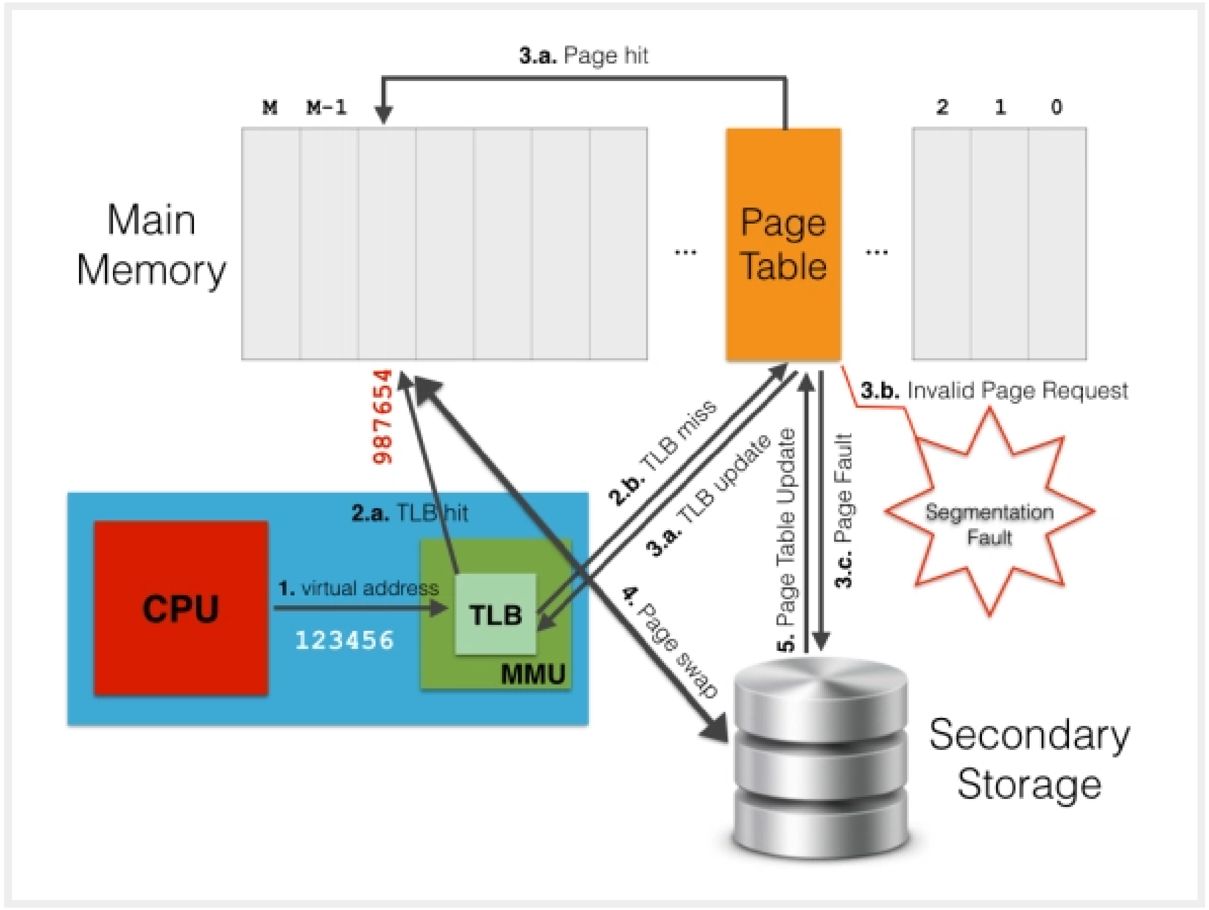
Virtual Memory, Paging, and Swapping

Virtual Memory is a memory management technique that is implemented using both hardware (**MMU**) and software (**operating system**). It abstracts from the real memory available on a system by introducing the concept of **virtual address space**, which allows each process thinking of physical memory as a *contiguous* address space (or collection of contiguous segments).
The goal of virtual memory is to map virtual memory addresses generated by an executing program into physical addresses in computer memory. This concerns two main aspects: *address translation* (from virtual to physical) and *virtual address spaces management*. The former is implemented on the CPU chip by a specific hardware element called **Memory Management Unit** or **MMU**. The latter is instead provided by the operating system, which sets up virtual address spaces (i.e., either a single virtual space for all processes or one for each process) and actually assigns real memory to virtual memory. Furthermore, software within the operating system may provide a virtual address space that can exceed the actual capacity of main memory (i.e., using also secondary memory) and thus reference more memory than is physically present in the system.

The primary benefits of virtual memory include freeing applications (and programmers) from having to manage a shared memory space, increasing security due to memory isolation, and being able to conceptually use more memory than might be physically available, using the technique of **paging**. Indeed, almost every virtual memory implementations divide a virtual address space into blocks of contiguous virtual memory addresses, called **pages**, which are usually **4 KB** in size.
In order to translate virtual addresses of a process into physical memory addresses used by the hardware to actually process instructions, the MMU makes use of so-called **page table**, i.e., a data structure managed by the OS that store mappings between virtual and physical addresses.
Concretely, the MMU stores a *cache* of recently used mappings out of those stored in the whole OS page table, which is called **Translation Lookaside Buffer** (**TLB**).

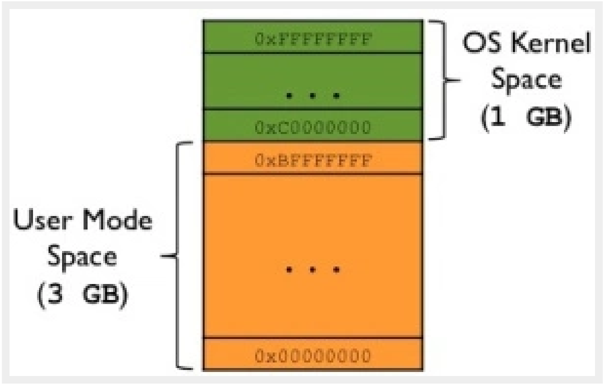
The picture below describes the address translation task as discussed above.



When a virtual address needs to be translated into a physical address, the MMU first searches for it in the TLB cache (**step 1.** in the picture above). If a match is found (i.e., *TLB hit*) then the physical address is returned and the computation simply goes on (**2.a.**). Conversely, if there is no match for the virtual address in the TLB cache (i.e., *TLB miss*), the MMU searches for a match on the whole page table, i.e., *page walk* (**2.b.**). If this match exists on the page table, this is accordingly written to the TLB cache (**3.a.**). Thus, the address translation is restarted so that the MMU is able find a match on the updated TLB (**1** & **2.a.**).
Unfortunately, page table lookup may fail due to two reasons. The first one is when there is no valid translation for the specified virtual address (e.g., when the process tries to access an area of memory which it cannot ask for). Otherwise, it may happen if the requested page is not loaded in main memory at the moment (an apposite flag on the corresponding page table entry indicates this situation). In both cases, the control passes from the MMU (hardware) to the **page supervisor** (a software component of the operating system kernel). In the first case, the page supervisor typically raises a **segmentation fault** exception (**3.b.**). In the second case, instead, a **page fault** occurs (**3.c.**), which means the requested page has to be retrieved

from the secondary storage (i.e., disk) where it is currently stored. Thus, the page supervisor accesses the disk, re-stores in main memory the page corresponding to the virtual address that originated the page fault (**4.**), updates the page table and the TLB with a new mapping between the virtual address and the physical address where the page has been stored (**3.a.**), and finally tells the MMU to start again the request so that a TLB hit will take place (**1** & **2.a.**).
As it turns out, the task of above works until there is enough room in main memory to store pages back from disk. However, when all the physical memory is exhausted, the page supervisor must also free a page in main memory to allow the incoming page from disk to be stored. To fairly determine which page to move from main memory to disk, the paging supervisor may use several **page replacement algorithms**, such as **Least Recently Used** (**LRU**). Generally speaking, moving pages from/to secondary storage to/from main memory is referred to as **swapping** (**4.**), and this is why page faults may occur.

Let us now turn back to the separation between OS kernel and user mode virtual memory spaces on a typical 32-bit Linux OS, as already introduced in the post In-Memory Layout of a Program (Process) and depicted in the following.



This does not mean the kernel uses that much physical memory. Instead, this is just the portion of virtual address space available to map whatever physical memory the OS kernel wishes, and it is orthogonal to the size of available physical memory. Note also that in Linux, kernel space is constantly present and maps the same physical memory in *all* processes, meaning that kernel space doesn't change and is mapped to same physical memory addresses across any process context switch.
Following the above split rule, kernel has 1 GB kernel virtual address space dedicated, and whatever allocation it does, it uses **always** those set of addresses. The actual mapping of virtual addresses to physical memory addresses happens exactly as discussed above through a combination of hardware (i.e., MMU) and software (i.e., OS page supervisor).

Suppose the system X has 512 MB of physical memory, then only those 512 MB out of the entire 1 GB virtual space will be mapped for kernel address space, leaving the remaining 512 MB of virtual addresses unmapped. On the other hand, if X has 2 GB of physical memory, the entire 1 GB of virtual addresses will be mapped to physical addresses.
Having virtual memory could allow OS kernel pages to partly reside on secondary storage (i.e., disk) if the whole kernel does not fit to physical memory.

In practice, this doesn't happen (at least on Linux) since most recent Linux kernels need approximately only **70 MB**, which is significantly below the amount of physical memory nowadays available on modern systems. Moreover, the kernel has data and code that must be kept **always** in main memory for efficiency reasons, and also because a page fault could not be handled otherwise. Think about what we discussed above when a page fault happens: the OS kernel (actually the OS page supervisor) takes the control of the system, enters a specific **Interrupt Service Routine** (**ISR**) to handle the page fault, and gives back the control to the user process that generated the page fault. If the OS kernel was not fit entirely to main memory, it might happen that the kernel itself generates a page fault. In a very bad case, such page fault could, for instance, concern the page with the code for the page fault handling routine, thereby blocking the whole system! That's why kernel code and data are always addressable (i.e., it never generates a page fault), and ready to handle interrupts or system calls at any time.

Share this:

Twitter | Facebook 5 | Google

★ Like

One blogger likes this.

7 Comments

zapy
DECEMBER 23, 2014 AT 7:07 PM
REPLY

2 days that I'm reading my textbook and couldn't have soo much understanding
Thank you

Amol
MAY 31, 2016 AT 6:07 PM
REPLY

Awesome

Pro Bono
NOVEMBER 1, 2016 AT 3:32 AM
REPLY

Hi Dr Tolomei,
Outstanding explanation of Virtual Memory, Paging and Swapping.
Many thanks for this great article.
Cheers
pro bono

---

bowrna
DECEMBER 28, 2016 AT 7:16 AM
REPLY

Thank you so much for such a detailed explanation

---

kanny
MAY 18, 2017 AT 11:57 AM
REPLY

Excellent and very clear explanation

Sanghamesh
JUNE 16, 2017 AT 5:32 PM
REPLY

Precise and wonderful explanation.
Keep up the good work and thank you!

Franco
JUNE 17, 2017 AT 5:50 PM
REPLY

Perfect !
Muchas gracias !

Leave a Reply

Enter your comment here...