Operating Systems – sprint 2022
Tutorial-Assignment 5

Instructor: Hans P. Reiser

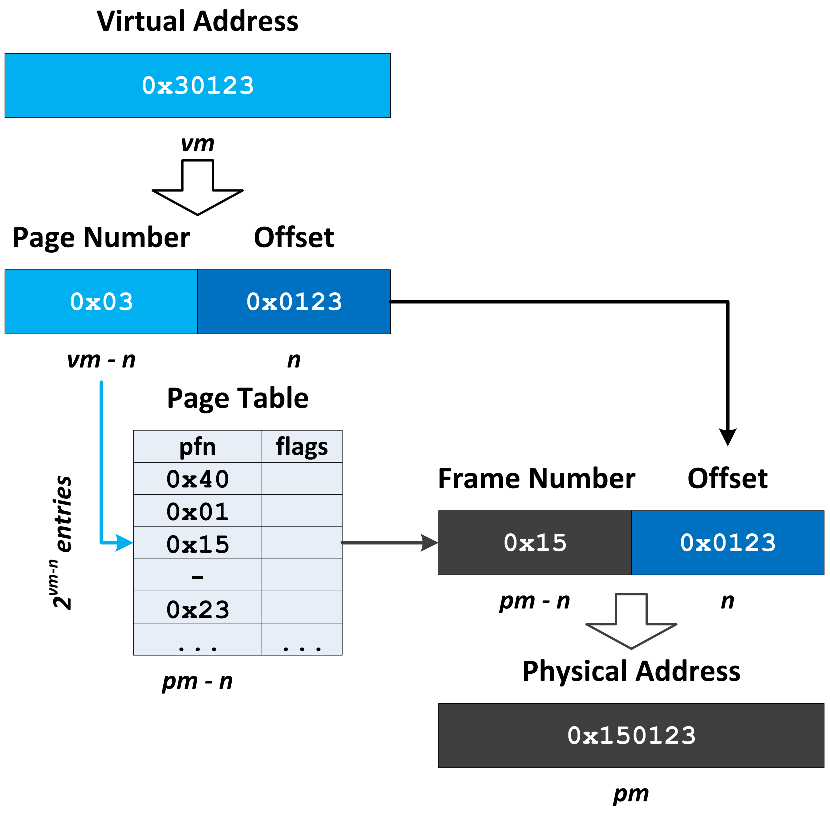
Question 5.1: Paging

a. Explain the basic idea of paging.

Solution:
Paging allows each application to have its own, contiguous (virtual) address space, while the physical space occupied by an application does not need to be contiguous. With paging, virtual memory is broken into fixed-sized blocks, called pages, and physical memory is broken into fixed-sized blocks of the same size, called frames. Each (virtual) page can be mapped to an arbitrary (physical) frame by means of a so-called page table.

b. How is a virtual address translated to a physical address using a single-level page table?

Solution:
Each virtual address is divided into two parts: A (virtual) page number (VPN) and a page offset. The page number is used as an index into a page table. Each element in the page table is called page table entry (PTE) and contains the (physical) frame number (PFN) of the frame to which the corresponding virtual page maps. To get the physical address, the page offset is concatenated to the frame number. This address is passed to the memory controller.



c. What is the disadvantage of using single-level page tables?

Solution:
The size of a page table increases linearly with the size of the virtual address space. A single-level page table can thus become quite large, especially on 64-bit processors.

d. Calculate the space requirements of a single-level page table for (1) a system with 32-bit virtual addresses and (2) a system with 48-bit virtual addresses (current x86-64). Both use a page size of 4 KiB. Assume that 4 bytes are required for a page table entry.

Solution:
32-bit virtual addresses result in a virtual address space size of 2^{32} bytes. As each page is 4 KiB = 2^{12} in size, the virtual address space consists of $2^{32}/2^{12} = 2^{20}$ pages. Each page requires one page table entry (4 bytes), leading to an overall memory consumption of $2^{20} * 4 = 2^{20} * 2^2 = 2^{22}$ bytes (or 4 MiB) per page table.

With 48-bit virtual addresses, there are $2^{48}/2^{12} = 2^{36}$ pages. A single-level page table would require $2^{36} * 4 = 2^{38}$ bytes (256 GiB). Because address spaces are usually only sparsely used, single-level page tables are not feasible for large address spaces.

e. What alternatives to using single-level page tables exist? What are their respective strengths and weaknesses?

Solution:

Multi-level page tables: *Instead of using a large, linear array as a page table, the page table consists of multiple levels. A virtual address is split into one index per level and an offset. The first index is used to obtain the address of the second-level table from the first-level table, the second index is used to obtain the address of the third-level table from the second-level table, and so on. The last table in the hierarchy contains the actual physical frame number.*

As virtual address spaces are normally sparse, not all tables on all levels are needed, so the space requirements for page tables can be reduced. In addition, subtables can also be swapped out to the disk.

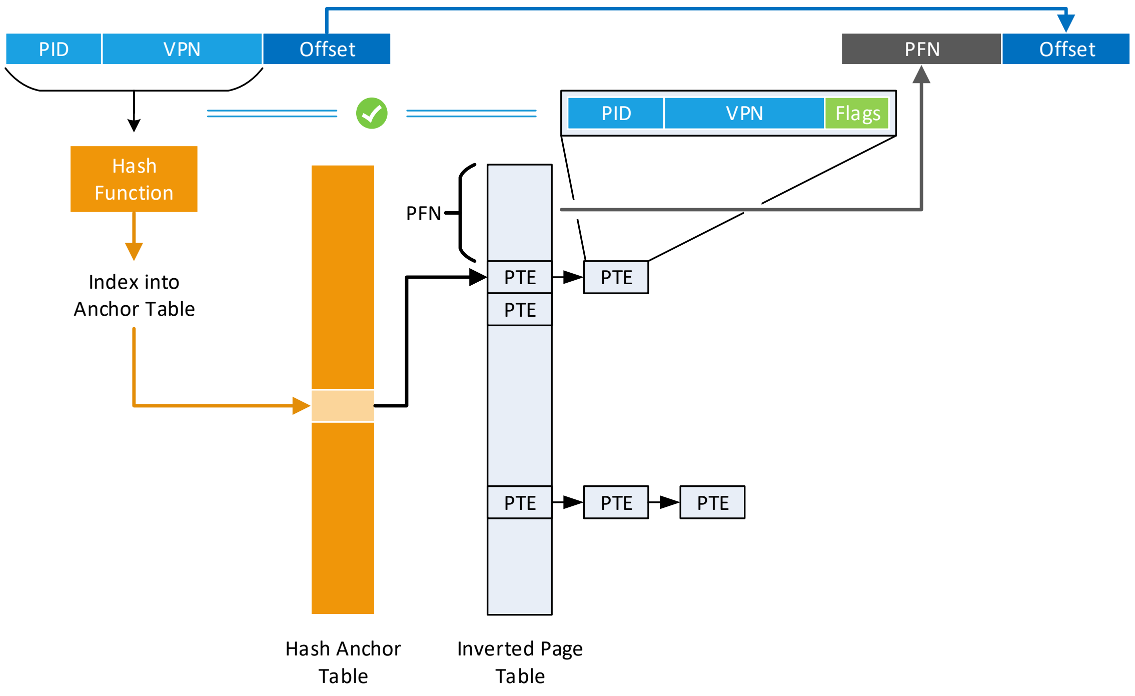
A disadvantage of multi-level page tables is that the number of memory accesses required to translate a virtual address into a physical address increases with each additional level.

Inverted page tables: *The page table does no longer contain one entry per virtual page, but one per physical frame. Each entry contains an identifier for the address space and the number of the virtual page to which the corresponding physical frame is mapped. With this scheme, only one page table is needed (instead of one table per process, as with the forward approach). The size of the page table only depends on the amount of actually available physical memory and is thus expected to be much smaller than a non-inverted page table.*

Hashed inverted page tables: *With the inverted page table, we likely save space compared to (multi-level) forward page tables. However, it is no longer possible to use virtual addresses as an index into the page table. In order to map a virtual address to a physical address, the table must be searched until the correct entry is found.*

A hash table can be used to reduce the lookup costs. Instead of using the virtual page number as an index, the number (and optionally an address space identifier) is hashed first, and the resulting hash value is used as an index into a hash anchor table. An entry in the anchor table points to the entry in the inverted page table that maps the virtual page of interest. However, as with all hash tables, collisions may occur. For that reason, entries in the inverted page table can be linked to build collision chains.

Additional note: Often, the virtual page number is not directly hashed to an index into the inverted page table, because then the hash function would also determine physical memory management (i.e., the placement of virtual pages in physical memory). For example PowerPC and UltraSPARC CPUs use a Hash Anchor Table (HAT). The hash of (PID,vpn) is used as an index into the HAT, which then points to the actual inverted hash table, which includes chaining for handling hash collisions. The size of the HAT can be adjusted to keep the probability of collisions low, without changing the size of the inverted hash table.



f. Let's assume we have a 32 bit architecture with 1 GiB of physical RAM, a page size of 8 KiB, and 20 processes running, each with 256 MiB of allocated virtual memory. Physical addresses, as well as page table entries, are 64 bit in size, For each of the variants linear, two-level, and inverted page table, calculate the total size of memory needed for the page table data structures. For two-level paging, assume that a single page directory with 2^9 entries and 2nd level page tables pages with 2^{10} entries are used, and no entries are shared among multiple page tables For the inverted page table, assume that 64 bit are used for storing PID and virtual address.

Solution:
In a 32 bit architecture we have a virtual address space size of of 2^{32} bytes. As each page is 8 KiB $= 2^{13}$ in size, the virtual address space consists of $2^{32}/2^{13} = 2^{19}$ pages. WIth linear page tables, each page requires one page table entry (8 bytes), leading to an overall memory consumption of $2^{19} * 8 = 2^{22}$ bytes (or 4 MiB) per page table. Each process uses a separate page table, thus we have a total of 80 MiB.

Two-level page tables: The virtual address space of each process is 256 MiB, which is 256 MiB / 8 KiB = 32768 pages. A single page table page store 8 KiB / 8 = 1024 entries, so we need 32 page table pages, and a single directory page, i.e. a total of 33 pages or 264 KiB per process, or in total $20 * 264 KiB = 5280 KiB \approx 5.2 MiB$.

*Inverted page tables: With 1 GiB of physical memory and 8 KiB per page frame, we have a total of 1 GiB / 8 KiB = 131072 page frames. There are 1024 (64 bit per entry, thus 8KiB / 8B) page table entries in each page, so we need 131072/1024 = 128 pages or 128*8 KiB = 1 MiB KiB in total for the inverted page table. It is one single table for all processes. (Note that depending on the implementation, this might be a bit more. For example, it is common* *that the hash function does not directly index the page table. Instead it selects an entry in a hash anchor table, which contains an index into the actual page table. Lookup of PID and virtual address as well as collision handling is then done at the page table level.)*

g. What is the purpose of the valid-bit (or present-bit) that is part of each page table entry?

Solution:

The valid-bit indicates whether the corresponding page is currently in memory or not. If the valid-bit is set, the page table entry can be used for deriving the physical address of the page. If the bit is not set, a page fault is raised, and the page fault handler must handle the situation. Note that the operating system must also keep track of whether an access to a given page is legal or not. The fact that the valid-bit is not set does **not** *necessarily mean that access to the page is forbidden. The page may also be temporarily swapped out, or not loaded/initialized yet.*

h. What is a translation lookaside buffer (TLB)?

Solution:
A TLB is a (fully-associative) hardware cache that is used to speed up virtual-to-physical translations. Each TLB entry consists of a key (the page number) and a value (the frame number). When a virtual address shall be translated to a physical address, the page number is presented to the TLB. The hardware (parallely) compares the page number with the keys of all entries. If an entry matches (TLB hit), the TLB can immediately return the corresponding frame number. If there is no match (TLB miss), the page tables must be walked to find the corresponding mapping. The mapping can afterwards be inserted into the TLB.

TLBs can be managed either by the hardware or by software (i.e., the operating system). With hardware-controlled TLBs, the hardware handles TLB misses by walking the page tables and inserting the mapping into the TLB. Only if no valid mapping is found in the page tables, the hardware raises an exception (called a page fault). If the TLB is managed by software, the hardware raises an exception on every TLB miss, and the operating system is responsible for walking the page tables and inserting the mapping into the TLB.

i. Why is reloading the CR3 register (physical address of the top-level page table) expensive? Consider the impact that changing CR3 (and thus the page tables) has on the TLB. Describe an extension to the simple TLB described in part (h) that could reduce this cost.

Solution:
When CR3 is reloaded, a new top-level page table—i.e., a different virtual address space—is installed. As a consequence, all virtual addresses might *be mapped to different physical addresses than before, so that all TLB entries are suddenly invalid. As a consequence, the next accesses to any page cause TLB misses, until the working set has been reloaded into the TLB.*

In addition to flushing the TLB, switching address spaces also requires to flush at least the L1 cache, as it is typically indexed by (parts of the) virtual address. Consequently, the following memory accesses also need to be serviced from (L2 or L3 caches or) main memory instead of the fast L1 cache.

A simple approach to prevent the TLB and L1 flush on each context switch are tagged TLBs and tagged caches: Tagging means to append some address space identifier (ASID) bits to the virtual address between processor and TLB/cache and use different ASIDs for different virtual address spaces. This way, switching from address space A to B and back to A would allow (parts of) A's working set to remain in the TLB and L1 cache across the switch to B.

For caches, using (parts of the) physical addresses as index also prevents having to flush them, but requires to first perform the virtual-to-physical translation, thus slowing down the common case (which is a cache hit). Although L1 caches are usually virtually indexed for speed, L2 caches and beyond are typically physically indexed, as the translation can take place while checking for a hit in the L1 cache.