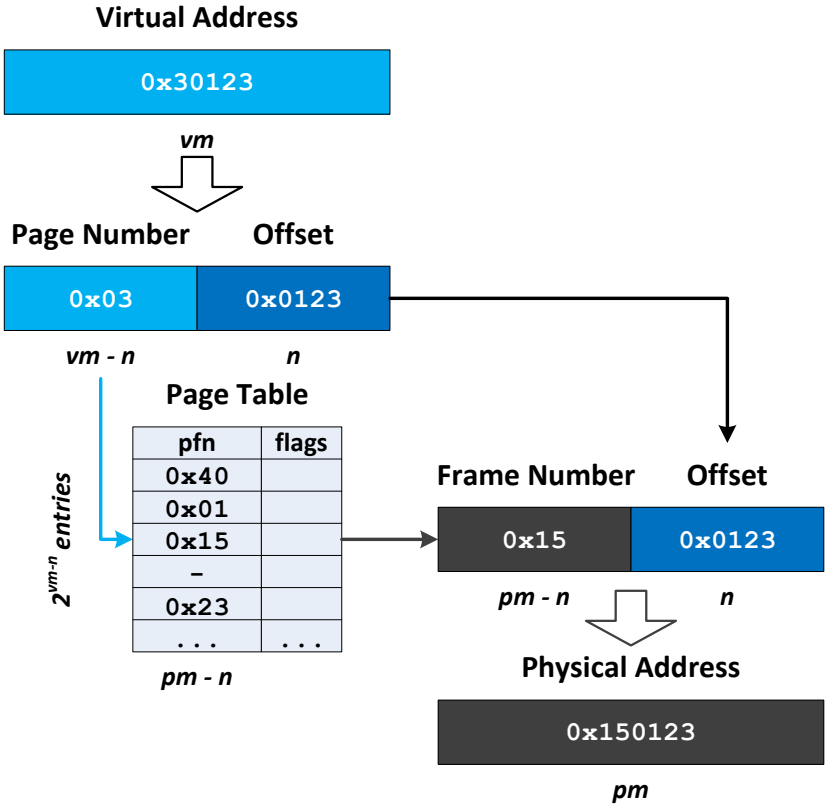
Question 10.1: Paging

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 a system with 32-bit virtual addresses and 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.

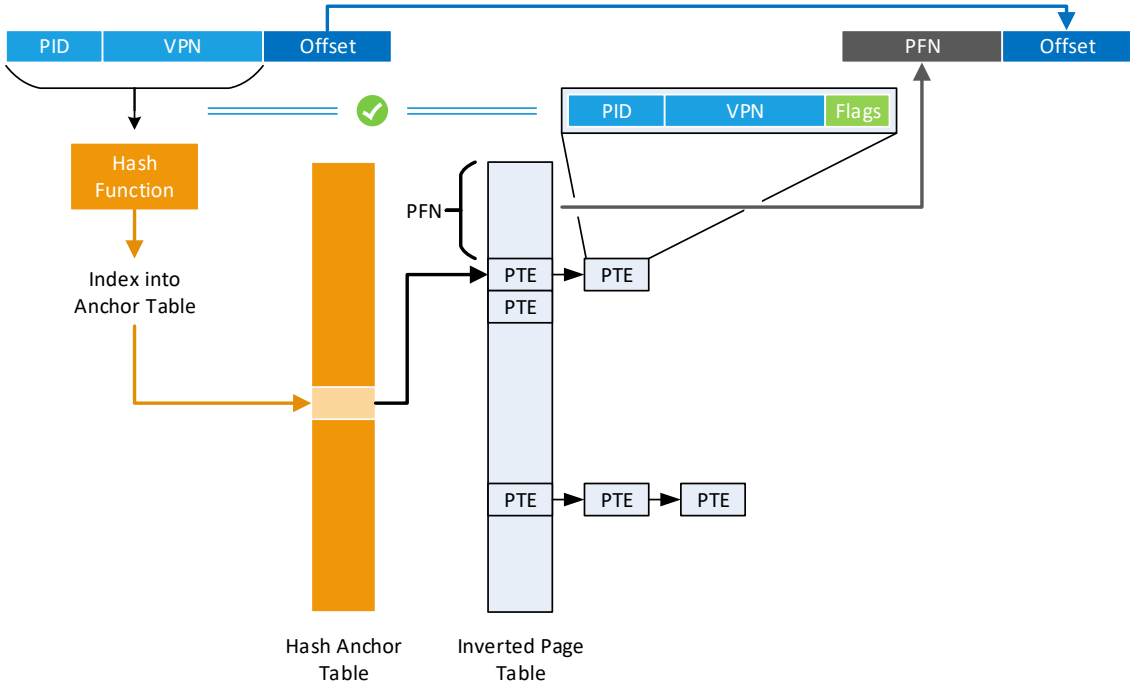
A disadvantage of multi-level page tables is that the number of memory accesses required to translate a virtual address into a physical address increases with each additional level.

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.

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).



f. 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.*

g. 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.

h. Why is reloading the CR3 register (physical address of the top-level page table) expensive? Propose a minor change of the hardware 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 or physically indexed 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.

Question 10.2: Caches

a. Describe the principle and the benefits of a memory hierarchy. How can a memory hierarchy provide both fast access and large capacity? On what typical program behavior does it depend?

Solution:
Manufacturing costs and/or technical limitations set an upper bound for the capacity of fast memory types (e.g., L1 cache). In a memory hierarchy multiple levels of gradually faster types of physical memory are installed in front of each other (e.g., L1-, L2-, L3 cache, RAM, hard disk, etc.), with the fastest (and smallest) memory being closest to or even embedded in the component (e.g., CPU) where it is needed. The goal is to create the illusion of a both fast and large memory. As each memory level is smaller than the slower one preceding it, not all data contained in the slow level can be held (or cached) in the faster level. Instead, a policy decides when and which data is migrated from a fast level to a slower level to make room for new data (read or written). Depending on the type of data, the policy may also discard data in that step (e.g., unmodified cache lines).

The two principles of locality *contribute to an efficient hierarchy usage:*

temporal locality *Data words that have been accessed recently are likely to be accessed again (e.g., loop counters). Consequently, it makes sense to spend some time getting them into a cache on first access in order to speed up further accesses.*

spatial locality *Most memory cells that are accessed within a short period of time are close together (clustered) rather than being spread all over the address space. Consequently, caching* lines *rather than mere data* words *makes sense.*

The stack is an example for these principles of locality: For most procedures, a number of local variables residing in the procedures' respective frames on the stack (clustered,

spatial locality) are likely to be accessed, whereas most global variables remain untouched. While executing, the local variables are also likely to be accessed more than once (at least one write access and one read access, otherwise you could have omitted the variable altogether).

b. Cache memory is divided into (and loaded in) blocks, also called cache lines. Why is a cache divided into these cache lines? What might limit the size of a cache line?

Solution:
First of all, each cache is divided into sets of cache lines. Each set is used to cache specific parts of the main memory, which are identified by a part of the virtual or physical address currently being accessed. Depending on the internal cache architecture (fully associative, n-way or direct mapped), each memory location can be mapped to any cache line (i.e., there is only one set), exactly one subset (comprising 2 or more lines), or exactly one cache line. As multiple memory regions map to the same (set of) cache lines, the yet unused part of the address is stored along with the data to identify its location. This extra label is called the tag.

Caches are organized in lines as memory accesses are most efficient if used in bursts: At the DRAM interface, one read operation requesting eight words is much faster than eight read operations for one word each (about seven times slower).

The size of the cache lines depends on the appropriate amount of data that can be easily swapped between the cache and its lower memory level. Current desktop CPUs usually employ 64 byte cache lines.

c. Enumerate and explain the different kinds of cache misses.

Solution:

Compulsory miss: *The accessed data block has not been in the cache before. This type of miss cannot be avoided.*

Capacity miss: *The cache is too small for all required data to fit.*

Conflict miss: *Different data blocks are mapped to the same location in the cache. See the lecture slides for strategies that avoid this type of miss (e.g., page coloring).*

d. Explain the difference between the write-through and write-back strategy.

Solution:
Write-through means that any update to a given memory location is not only applied to the cache, but also to main memory. This implies that the main memory content is always up-to-date (i.e., there are no inconsistencies between the cache and main memory). With write-back, only the cache is updated when data is modified. Only when data is evicted from the cache, it is written back to main memory.

e. How can writes to a data item that is currently not in the cache be handled?

Solution:
There exist two strategies that can be applied in this case: write-allocate and write-to-memory. With write-allocate, the data item to which the write is performed is first loaded from memory into the cache. Afterwards, one of the two strategies from above can be applied to handle the write access. With write-to-memory, the data item is not loaded into the cache, the modification is only applied to main memory.

f. What are the two main problems that can arise with caches that are virtually-indexed and virtually-tagged? What can be done to avoid or solve these problems?

Solution:

Ambiguity (Mehrdeutigkeit): *Two identical virtual addresses point to different physical addresses at two different points in time. Without special precaution, the cache will still contain the old data after the mapping of the virtual address was modified. To avoid this problem, one has to invalidate the appropriate cache line whenever the mapping of a virtual address to a physical address changes. In case of an address space switch, the entire cache has to be invalidated.*

Aliases (Bedeutungsgleichheit): *Two different virtual addresses map to the same physical address. Whenever a modification to one of these virtual addresses is performed, only that cache line will be updated, the cache line that contains the same data item (but addressed via the other virtual address) will remain untouched, thus leading to inconsistencies.*

One "solution" to that problem is to prohibit that a frame is mapped to two (or more) different virtual locations within the same address space, or to disable caching for such shared segments.

Alternatively, one can restrict the virtual addresses that map to the same physical address in a way that all of these addresses are mapped to the same cache line (however, this only works when direct-mapped caches are used).

g. Does using virtually-indexed, physically-tagged caches solve the two problems described in the previous question?

Solution:
The usage of virtually-indexed, physically-tagged caches solves the ambiguity problem: Whenever the mapping of a virtual address to a physical address changes, the appropriate (physical) tag of the cache line will no longer match, and the new data item has to be loaded into the cache. However, that is only true if at least the entire page frame number (i.e., the bits identifying the physical frame) is used as tag. The tag might thus be longer than in virtually-indexed, virtually-tagged caches, where the tag is build from the remaining bits after cutting off the index and offset.

The problems with aliases is not solved: If a physical frame is accessible via two different virtual addresses, it will still be loaded into the cache twice (with the cache lines being derived from the virtual address), but will now be tagged with the physical address (which is the same for both entries).