1 Paging

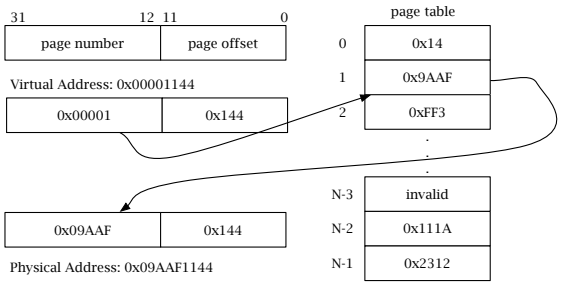
Even with relocation, both fixed partitions and segmentation had problems. Fixed partitions still suffer from internal fragmentation and segments may require many expensive relocations. How do we solve this problem? By shrinking fixed partitions down to a reasonable size, and extending the concept of a virtual address to allow a process to allocate many small partitions. These partitions are called pages.

1.1 Virtual Addressing

If the page size is a power of two, then a virtual address in a paging system can be partitioned into two pieces. The page number and the page offset. The low-order bits (e.g. 12 bits for 4K page size) just specify the byte offset within a particular page, while the high-order bits actually specify which page in memory is being addressed. If a virtual address is interpreted as a physical address, then the page number is just the particular page in memory. In reality, however a page table is used to translate from virtual to physical addresses.

1.2 Page Tables

With simple segmentation or fixed paritioning, a process only needed to carry around one or two extra words of information for the OS and HW to manage its address space. With paging, a process can be reasonably expected to need hundreds or thousands of pages. Therefore a datastructure is needed to hold all the address space information. This data structure is called a page table. Essentially, a page table is an array of entries mapping virtual addresses to physical addresses. The page number from a virtual address is an **index** in the page table. The entry at that index contains the physical address of the page in question. The following diagram illustrates this process:



This diagram is showing virtual to physical address translation on a system with 4K pages. Because each page is 4K, 12 bits are needed to address the bytes within a page. Therefore the lower 12 bits of the virtual address are the page offset. The remaining 20 bits are the page number. For the virtual address `0x00001144`, `0x00001` is the page number. That value is used to index the page table. The value at index 1 is read in (in this case `0x9AAF`), and that is used to construct the physical address `0x09AAF144` (note that the offset remains unchanged).

Notice that the page table decouples virtual and physical addresses. Therefore virtual addresses that are contiguous (such as `0x00001FFF` and `0x00002000`) can map to non-contiguous physical pages (as is the case in the diagram). This means that an OS can grow a process' address space virtually, without having to ensure contiguous free space physically! i.e. no copying necessary. Now, each process on creation, must have a page table allocated to it. And each process must retain a pointer to the beginning of the page table.

1.3 Protection and Swapping with Pages

Memory protection in a paging system is a little different. For a page table, there is a distinguished value (often 0) that signifies an invalid mapping. If the page table entry is invalid, that means that that virtual address is not mapped in this process' address space. Therefore it is an invalid address and the OS should be signalled. In this sense, page tables should all be initialized to invalid entries. Additionally, allocation of memory to processes is all handled in terms of pages. All the OS has to worry about is managing pages, rather than a contiguous sea of bytes.

Page numbers are smaller than full addresses. They have to be, to make room for the page offset. That means that if the page table entries are address sized, there are some leftover bits in each entry that would normally just be 0. For example with 8K pages (13 bits of page offset) each page table entry will have 13 bits unneeded to store the physical page address. These bits can be used to store page meta-information. For instance, three bits could be set aside to store read, write, and execute permission information. These bits then become flags (0 for no, 1 for yes), and now a process can control memory protection at a finer grained level. When the MMU retrieves the page table entry, it also retrieves the permission flags. These flags can then be compared against the kind of memory operation being done to ensure that the access is allowed.

Paging also allows for finer-grained swapping. If the OS reserves an additional bit in the page table entry, the OS can distinguish between resident (1) and swapped (0) pages. Now the OS can swap out processes one page at a time, rather than having to swap out the entire address space. When the OS needs to free up main memory, it chooses a set of pages somehow, swaps them out to disk and marks them as swapped in their process' page tables. Then, later on, if the process attempts to access that page, the OS is signalled by the MMU. The OS then swaps the needed page in, changes the entry's flag to resident and resumes execution of the process.

2 Page Table Performance and Implementation

Paging solves many of the problems associated with partitions and segments. It eliminates external fragmentation, minimizes internal fragmentation, and allows the OS to allocate and swap process' address spaces in a cheaper, incremental fashion. However, page tables are space-hungry and generate extra memory traffic.

Consider you average intel PC. It uses 32-bit addresses and defaults to 4K pages. That means 20 bits per page number, or 2^{20} page table entries. Each entry is 32 bits (4 bytes) therefore a flat array of page table entries would consume 4MB (per process!). Now, with our cheap RAM, that may not seem like a lot, but in the early 90s that would set you back $200. For a more modern

example, consider a system with 64-bit addresses and 8K pages. That's 13 bits of offset, 51 bits of page number. A flat page table would need to have 2^{51} entries, which is roughly 2 million trillion entries! They would consume 2 petabytes of memory. No system around today has 2 petabytes of memory, and yet 64-bit systems with page-based virtual memory seem to run just fine. How is this possible?

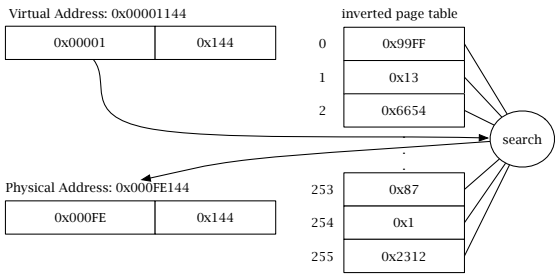
2.1 Page Table Implementation

A simple array of page table entries is obviously infeasable. So what do people do in practice?

2.1.1 Inverted Page Table

One solution is the inverted page table (aka content-addressable memory). A standard page table contains mappings from each virtual page to a physical frame, even if the virtual page is invalid. An inverted page table reduces the space requirements by only storing mappings for valid pages. It does this by inverting the mapping and having a table of frame entries that specify the virtual page being mapped (the index in the table is the physical frame address and the entry is the virtual address). If the number of physical frames is smaller than the virtual address space, you save space.

However, finding the virtual to physical mapping is more complex. Each virtual address must be resolved by searching the IPT, which means that the translation is slower and uses more hardware. In practice IPTs are only feasible for small memories (a few thousand frames at most).



2.1.2 Multi-level Page Table

Inverted page tables are only practical for small memories. If you have a large memory, what do you do? You introduce a bit of hierarchy to reduce the amount of information needed at any one time. A multi-level page table works by introducing some indirection into a flat page table. Instead of partitioning a virtual address into two pieces, it is divided into three or four. Let's consider the two-level version:

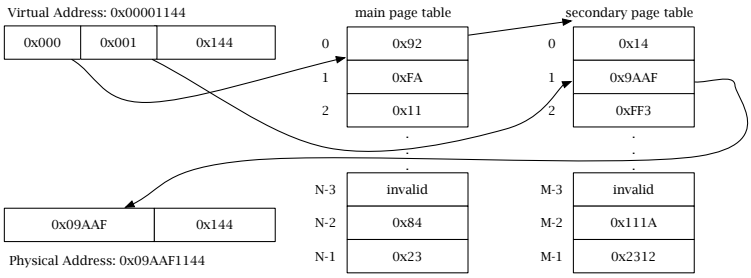
- **High bits:** The highest-order bits are an index into a main page table. Its entries point to *secondary page tables* rather than frames.
- **Middle bits:** The next chunk of bits are an index into the secondary page table referenced by the main page table. These entries contain frame references and act just like the entries in a single-level page table

- **Low bits:** The low-order bits are just an ordinary page offset.

A multi-level page table has several advantages:

1. **Saves Space:** A secondary page table only needs to be allocated if there are any valid mappings in that address range. Therefore, only the main page table "wastes" space on those addresses.

2. **Swappable Page Tables:** Because the main page table now acts as the entry point for address translation, secondary page tables can be swapped out to save space. Of course, this makes address translation more complicated as secondary page tables may have to be swapped in before the translation can complete.



The diagram above illustrates the following example. With 4K pages and a 2 level page table, we can chop the single-level page number into two 10 bit values. To translate the virtual address `0x00001144`, the most significant 10 bits (`0x000`) are used to index into the main page table. That entry contains the physical address of the secondary page table to use (0x92). Then the next 10 bits (0x001) are used as an index into the secondary table. This entry is a normal page table entry, which is then retrieved and used to generate the physical address (0x09AAF144).

Some important things to note. The main page table holds the physical addresses of the secondary page tables. Why? Because if the main page table held virtual addresses, then those addresses would also have to be translated. This could lead to an awful cascade of page-table lookups. So, most systems decide on having the main page table entries refer to physical addresses. Second, the main page table can have invalid entries. At the main table level, an invalid means that there are no valid addresses in the entire range specified by a secondary table. This saves space in that a secondary table doesn't have to be allocated in these cases.

In terms of implementation, often the page size of the machine dictates how large the main page table can be. The main page table cannot be swapped out, therefore if you want to save page table space by using multi-level page tables, you would want to make the main table as small as possible. This is usually a single page. On a system with 4K pages, assuming that each main page table entry is 4B large, a single-page can hold 1024 main page table entries. This would require 10 bits to index into. Therefore the upper 10 bits are an index into the main page table, which is itself a single page. The sizes of the secondary (and possibly tertiary) tables are usually more flexible. Again, for reasons of efficiency, you'd like them to be power-of-two multiples of page size (that means you can chop up addresses easily). For the example above, there are only 10 bits left over for the secondary index, therefore each secondary table is also just a single page.

Because of the indirection of the main table, secondary tables can themselves be swapped out. Swapping proceeds similarly to normal process memory, execept that the entries in the main page must point to the physical location of the secondary page. However, because the page tables are themselves aligned on page boundaries, there are plenty of bits available for marking resident/swapped conditions.

2.2 Memory Traffic

Paging increases memory traffic, unfortunately. Because virtual to physical address translation requires lookups into memory, a single virtual read or write can translate into multiple memory accesses. Consider the running example of page translation we've been using. If the virtual address is 0x00001144. In the old fixed partition or simple segmentation schemes, it could be directly converted into a physical address and then sent on to main memory. Therefore 1 virtual access became one physical access. For a single-level page table, the virtual translation required a lookup into the page table (1 read) to translate the address. Then the physical address could be used (1 access). Therefore, with single-level paging 1 virtual access becomes two physical accesses (1 read + 1 access). Now consider the two-level example: to translate the address required a lookup into the main page table (1 read), and then another lookup into a secondary page table (1 read), then the translated address was used (1 access). Therefore a two level paging system requires 3 physical access for a single virtual access. Consider, now the fact that most modern systems use a three-level paging scheme! These systems could potentially generate 4 physical accesses for a single virtual access. And the gap between memory speed and processor speed keeps growing! How is it that these systems actually run fast?

There is one big reason: caching. First, the actual cache memories onboard the chips can be virtual or physical. That is, the cache can cache values according to their virtual addresses or their physical addresses. If they're cached virtually, then the processor will not have to do address translation if the value is already in cache. However virtual caches have to do translation when they evict values back into main memory. They also have to be aware of swapping decisions made by the OS (or more appropriately, the OS has to make sure that virtual caches do not hold stale values). If the caches are physical, then virtual addresses from the processor will have to be translated first, however actual page table entries are more easily cached. In reality most processors keep the nearest caches (L0 and L1) virtual, and the more distant caches (L3) physical. Caching is extended to the MMU, however. The MMU actually caches the results of translations in order to reduce the number of lookups that need to go out to memory.

2.2.1 Translation Lookaside Buffers

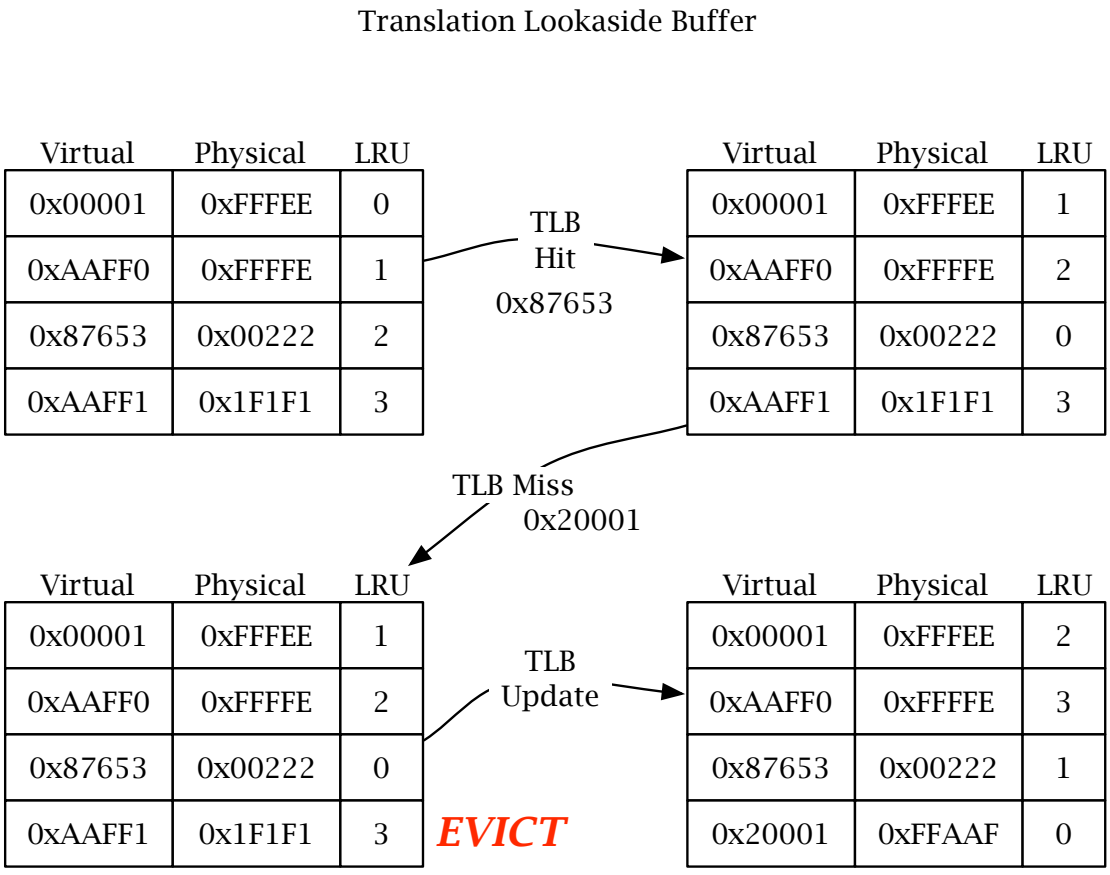
The cache inside the MMU is known as a translation lookaside buffer or TLB. It is basically a list of mappings from virtual to physical page numbers. As virtual addresses are translated, the mapping is stored in the TLB. When the processor asks for a virtual address the following steps occur:

1. The TLB is examined

2. If there is a mapping for the virtual address in the TLB, use it (a TLB hit)
3. If not, perform the normal page table translation (TLB miss)
4. Insert the translation into the TLB

Of course, the TLB is much smaller than a page table, so it gets filled fairly quickly. Like any cache, there is great care taking in choosing a replacement policy in order to reduce the number of TLB misses. In general a least-recently-used (LRU) policy is often implemented. With this policy, the TLB keeps track of when entries were last used. When the TLB is full and must insert a new entry, the entry for the least recently used mapping is evicted and the new entry is written in its place. LRU exploits locality of reference to enhance TLB performance. Locality is the tendency of programs to frequently access the same storage locations. LRU exploits temporal locality, which is the tendency for programs to reference a given storage location frequently over a small duration. If a program exhibits temporal locality, then the most recently used page contains "hot" data and should not be evicted. Conversely the least recently used page is less likely to contain "hot" data, and so can be evicted more easily.

The following diagram illustrates a 4-element TLB serving two address requests (the first hits, and the second misses).



In this example, the TLB has reduced the number of page-table lookups by half. This would indicate a TLB miss rate of 50%. In reality, TLBs have miss rates of 0.1% to 1%. It is because TLBs are so good at their job that OSes can employ sophisticated multi-level paging without imposing a serious performance hit on user-code.

2.2.2 MMU/OS Integration

Historically, there are two ways for an OS and MMU to get along. The first way (and the way currently favored by intel x86 architecture) is to have the MMU basically hardwired. The MMU supports a limited range of page sizes and paging schemes, and all the OS can do is specify which page size it would like and make sure that all the OS memory management behaves in an MMU-compatible way. The advantage of this scheme is that the hardware folks can make the MMU ridiculously fast. The disadvantage is that the OS designers have no control over memory organization, which can lead to inefficiencies and compications (remember that the OS has more knowledge about the processes running on the machine than the MMU does).

The second scheme is a programmable MMU. In this setup, the MMU is controlled by specific supervisor code. The MMU communicates back to the OS by use of interrupts/traps. For example, with a programmed MMU, when a virtual address misses in the TLB, an interrupt is raised and the OS steps in to handle the miss. Similarly, permission bits can be specified to raise interrupts. A programmable MMU has the distinct advantage that the OS can directly control the memory layout for the machine. The OS designer can chose the page-table architecture, and the location and meaning of permission flags. Additionally, OS designers can optimize certain operations (such as context-switching). The disadvantage is that programmable MMUs are slower, software is almost always slower than hardware. Also, the OS will tend to recieve more interrupts from a programmable MMU, and this can degrade performance somewhat.