

# Virtual Memory, Part 1: Introduction to Virtual Memory

Alex Kizer edited this page 2 days ago · 13 revisions

## What is Virtual Memory?

In very simple embedded systems and early computers, processes directly access memory i.e. "Address 1234" corresponds to a particular byte stored in a particular part of physical memory. In modern systems, this is no longer the case. Instead each process is isolated; and there is a translation process between the address of a particular CPU instruction or piece of data of a process and the actual byte of physical memory ("RAM"). Memory addresses are no longer 'real'; the process runs inside virtual memory. Virtual memory not only keeps processes safe (because one process cannot directly read or modify another process's memory) it also allows the system to efficiently allocate and re-allocate portions of memory to different processes.

## What is the MMU?





The Memory Management Unit is part of the CPU. It converts a virtual memory address into a physical address. The MMU may also interrupt the CPU if there is currently no mapping from a particular virtual address to a physical address or if the current CPU instruction attempts to write to location that the process only has read-access.

## So how do we convert a virtual address into a physical address?

Imagine you had a 32 bit machine. Pointers can hold 32 bits i.e. they can address  $2^{32}$  different locations i.e. 4GB of memory (we will following the standard convention of one address can hold one byte).

Imagine we had a large table - here's the clever part - stored in memory! For every possible address (all 4 billion of them) we will store the 'real' i.e. physical address. Each physical address will need 4 bytes (to hold the 32 bits). This scheme would require 16 billion bytes to store all of entries. Oops - our lookup scheme would consume all of the memory that we could possibly buy for our 4GB machine. We need to do better than this. Our lookup table better be smaller than the memory we have otherwise we will have no space left for our actual programs and operating system data. The solution is to chunk memory into small regions called 'pages' and 'frames' and use a lookup table for each page.

## What is a page? How many of them are there?

▼ Pages 51

[Home](#)

[#Example Markdown](#)

[#Informal Glossary](#)

[#Piazza: When And How to Ask For Help](#)

[C Programming, Part 1: Introduction](#)

[C Programming, Part 2: Text Input And Output](#)

[C Programming, Part 3: Common Gotchas](#)

[C Programming, Part 4: Debugging](#)

[Deadlock, Part 1: Resource Allocation Graph](#)

[Deadlock, Part 2: Deadlock Conditions](#)

[File System, Part 1: Introduction](#)

[File System, Part 2: Files are inodes \(everything else is just data...\)](#)


[File System, Part 3: Permissions](#)

[File System, Part 4: Working with directories](#)

[File System, Part 5: Virtual file systems](#)

Show 36 more pages...

Clone this wiki locally

 Clone in Desktop

A page is a block of virtual memory. A typical block size on Linux operating system is 4KB (i.e.  $2^{12}$  addresses), though you can find examples of larger blocks.

So rather than talking about individual bytes we can talk about blocks of 4KBs. Each block is called a page. We can also number our pages ("Page 0" "Page 1" etc)

## EX: How many pages are there in a 32bit machine (assume page size of 4KB)?

Answer:  $2^{32}$  address /  $2^{12}$  =  $2^{20}$  pages.

Remember that  $2^{10}$  is 1024, so  $2^{20}$  is a bit more than one million.

For a 64 bit machine,  $2^{64}$  /  $2^{12}$  =  $2^{52}$ , which is roughly  $10^{15}$  pages.

## What is a frame?

A frame (or sometimes called a 'page frame') is a block of *physical memory* or RAM (=Random Access Memory). This kind of memory is occasionally called 'primary storage' (and contrasted with slower, secondary storage such as spinning disks that have lower access times)

A frame is the same number of bytes as a virtual page. If a 32 bit machine has  $2^{32}$  (4GB) of RAM, then there will be the same number of them in the addressable space of the machine. It's unlikely that a 64 bit machine will ever have  $2^{64}$  bytes of RAM - can you see why?

## What is a page table and how big is it?

A page table is a mapping between a page to the frame. For example Page 1 might be mapped to frame 45, page 2 mapped to frame 30. Other frames might be currently unused or assigned to other running processes, or used internally by the operating system.

A simple page table is just an array, `int frame = table[ page_num ];`

For a 32 bit machine with 4KB pages, each entry needs to hold a frame number - i.e. 20 bits because we calculated there are  $2^{20}$  frames. That's 2.5 bytes per entry! In practice, we'll round that up to 4 bytes per entry and find a use for those spare bits. With 4 bytes per entry x  $2^{20}$  entries = 4 MB of physical memory are required to hold the page table.

For a 64 bit machine with 4KB pages, each entry needs 52 bits. Let's round up to 64 bits (8 bytes) per entry. With  $2^{52}$  entries thats  $2^{55}$  bytes (roughly 40 peta bytes...) Oops our page table is too large.

In 64 bit architectures memory addresses are sparse, so we need a mechanism to reduce the page table size, given that most of the entries will never be used.

## What is the offset and how is it used?

Remember our page table maps pages to frames, but each page is a block of contiguous addresses. How do we calculate which particular byte to use inside a particular frame? The solution is to re-use the lowest bits of the virtual memory address directly. For example, suppose our process is reading the following address- `VirtualAddress = 11110000111100001111000010101010` (binary)

On a machine with page size 256 Bytes, then the lowest 8 bits (10101010) will be used as the offset. The remaining upper bits will be the page number (111100001111000011110000).

## Multi-level page tables

Multi-level pages are one solution to the page table size issue for 64 bit architectures. We'll look at the simplest implementation - a two level page table. Each table is a list of pointers that point to the next level of tables, not all sub-tables need to exist. An example, two level page table for a 32 bit architecture is shown below-

```
VirtualAddress = 11110000111111110000000010101010 (binary)
                |_Index1_|_|          | 10 bit Directory index
                    |_Index2_|_|      | 10 bit Sub-table index
                        |_____| 12 bit offset (passed directly)
```

In the above scheme, determining the frame number requires two memory reads: The topmost 10 bits are used in a directory of page tables. If 2 bytes are used for each entry, we only need 2KB to store this entire directory. Each subtable will point to physical frames (i.e. required 4 bytes to store the 20 bits). However, for processes with only small memory needs, we only need to specify entries for low memory address (for the heap and program code) and high memory addresses (for the stack). Each subtable is 1024 entries x 4 bytes i.e. 4KB for each subtable. Thus the total memory overhead for our multi-level page table has shrunk from 4MB (for the single level) to 10KB!

Do page tables make memory access slower? (And what's a TLB)

Yes - Significantly ! (But thanks to clever hardware, usually no...) Compared to reading or writing memory directly. For a single page table, our machine is now twice as slow! (Two memory accesses are required) For a two-level page table, memory access is now three times as slow. (Three memory accesses are required)

To overcome this overhead the MMU includes an associative cache of recently-used virtual-page-to-frame lookups. This cache is called the TLB ("translation lookaside buffer"). Everytime a virtual address needs to be translated into a physical memory location, the TLB is queried in parallel to the page table. For most memory accesses of most programs, there is a significant chance that the TLB has cached the results. However if a program does not have good cache coherence (for example is reading from random memory locations of many different pages) then the TLB will not have the result cache and now the MMU must use the much slower page table to determine the physical frame.

Can frames be shared between processes? Yes! In addition to storing the frame number, the page table can be used to store whether a process can write or only read a particular frame. Read only frames can then be safely shared between multiple processes. For example, the C-library instruction code can be shared between all processes that

dynamically load the code into the process memory. Each process can only read that memory.

In addition, processes can share a page with a child process using the `mmap` system call.

# What else is stored in the page table and why?

In addition to read-only bit and usage statistics discussed above, its common to store modification and execution information:

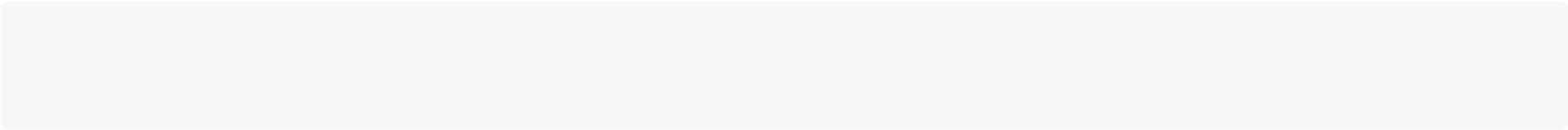
## Dirty bit

[http://en.wikipedia.org/wiki/Page\\_table#Page\\_table\\_data](http://en.wikipedia.org/wiki/Page_table#Page_table_data)

The dirty bit allows for a performance optimization. A page on disk that is paged in to physical memory, then read from, and subsequently paged out again does not need to be written back to disk, since the page hasn't changed. However, if the page was written to after it's paged in, its dirty bit will be set, indicating that the page must be written back to the backing store. This strategy requires that the backing store retain a copy of the page after it is paged in to memory. When a dirty bit is not used, the backing store need only be as large as the instantaneous total size of all paged-out pages at any moment. When a dirty bit is used, at all times some pages will exist in both physical memory and the backing store.

## Execution bit

The execution bit defines whether bytes in a page can be executed as CPU instructions. By disabling a page, it prevents code that is maliciously stored in the process memory (e.g. by stack overflow) from being easily executed. (further reading: [http://en.wikipedia.org/wiki/NX\\_bit#Hardware\\_background](http://en.wikipedia.org/wiki/NX_bit#Hardware_background))



Legal and Licensing information: Unless otherwise specified, submitted content to the wiki must be original work (including text, java code, and media) and you provide this material under a [Creative Commons License](#). If you are not the copyright holder, please give proper attribution and credit to existing content and ensure that you have license to include the materials.

