Linux Kernel
Memory Management

Memory Organization
Part 1 of 4

Linux Kernel : Memory Management series by kaiwanTECH, © 2000-2016

Part 1 : Introduction to Virtual Memory, Paging

Part 2 : Kernel and Process Segments

Part 3 : Memory Organization

Part 4 : Page Cache, Watermarks, OOM, VMAs

Important Notice : Courseware - Legal

This courseware is both the product of the author and of freely available opensource and/or public domain materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of the **courseware PDFs are considered proprietary** and thus cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - **source code** and binaries (where applicable) - that form part of this courseware, and that are made available to the participant, are released under the terms of the permissive **MIT license**. Under the terms of the MIT License, you can certainly use the source code provided here; you must just attribute the original source (author of this courseware and/or other copyright/trademark holders).

VERY IMPORTANT :: Before using this source(s) in your project(s), you *MUST* check with your organization's legal staff that it is appropriate to do so.

The courseware PDFs are *not* under the MIT License, they are to be kept confidential, non-distributable without consent, for your private internal use only.

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant VM are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind, and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Table of Contents

Introduction to Virtual Memory

<< Source: "ARM System Developer's Guide – Designing and Optimizing System Software" by Sloss, Symes and Wright, published by Elsevier. >>

When creating a multitasking embedded system, it makes sense to have an easy way to write, load, and run independent application tasks. Many of today's embedded systems use an operating system instead of a custom proprietary control system to simplify this process. More advanced operating systems use a hardware-based memory management unit (MMU).

One of the key services provided by an MMU is the ability to manage tasks as independent programs running in their own private memory space. A task written to run under the control of an operating system with an MMU does not need to know the memory requirements of unrelated tasks. This simplifies the design requirements of individual tasks running under the control of an operating system.

The MMU simplifies the programming of application tasks because it provides the resources needed to enable virtual memory—an additional memory space that is independent of the physical memory attached to the system. The MMU acts as a translator, which converts the addresses of programs and data that are compiled to run in virtual memory to the actual physical addresses where the programs are stored in physical main memory.

This translation process allows programs to run with the same virtual addresses while being held in different locations in physical memory.

This dual view of memory results in two distinct address types: virtual addresses and physical addresses. Virtual addresses are assigned by the compiler and linker when locating a program in memory. Physical addresses are used to access the actual hardware components of main memory where the programs are physically located.

Source

...

Linux processes execute in a virtual environment that makes it appear as if each process had the entire address space of the CPU available to itself.

This virtual address space extends from address 0 all the way to the maximum address.

On a 32-bit platform, such as IA-32, the maximum address is (2^32 - 1) or 0xffffffff.

On a 64-bit platform, such as IA-64, this is (2^64 -1) or 0xffffffffffffffff.

While it is obviously convenient for a process to be able to access such a huge address space, there are really three distinct, but equally important, reasons for using virtual memory.

1. **Resource virtualization.**
   On a system with virtual memory, a process does not have to concern itself with the details of how much physical memory is available or which physical memory locations are already in use by some other process. In other words, virtual memory takes a limited physical resource (physical memory) and turns it into an infinite, or at least an abundant, resource (virtual memory).

2. **Information isolation.**
   Because each process runs in its own address space, it is not possible for one process to read data that belongs to another process. This improves security because it reduces the risk of one process being able to spy on another process and, e.g., steal a password.

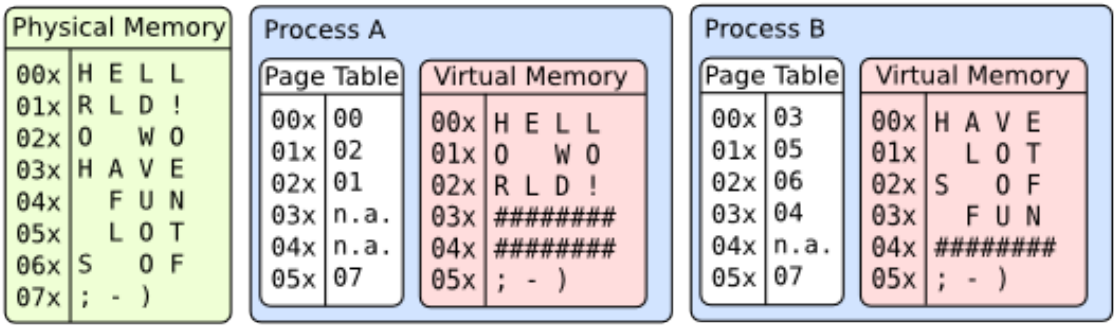
3. **Fault isolation.**
   Processes with their own virtual address spaces cannot overwrite each other's memory. This greatly reduces the risk of a failure in one process triggering a failure in another process. That is, when a process crashes, the problem is generally limited to that process alone and does not cause the entire machine to go down.

   << Consider a modern tab-based multithreaded web browser; if any one thread hits a fatal error, the entire browser must shut down. On the other hand, the Google Chrome open source project architecture is based on the *multi-process* model; see their comic adaptation on why. (From a software design viewpoint, the site is interesting even otherwise!) >>

…

Source

In a (virtual memory based) paged (32-bit) system, each process may execute in its own 4 GB (or whatever) area of memory, without any chance of effecting any other process's memory, or the kernel's.



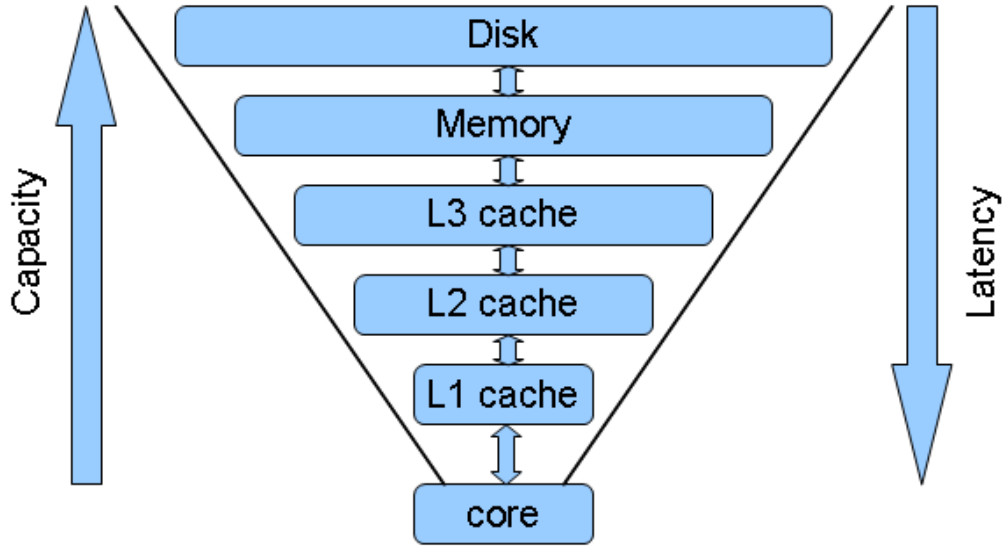
** Page Table : shown holding the virtual address (or page) and the corresponding physical address (or page frame).*

Source – Memory FAQ

```
...
  Virtual memory provides a software-controlled set of memory addresses,
  allowing each process to have its own unique view of a computer's memory.

  Virtual addresses only make sense within a given context, such as a
  specific process.  The same virtual address can simultaneously mean
  different things in different contexts.

  Virtual addresses are the size of a CPU register.  On 32 bit systems each
  process has 4 gigabytes of virtual address space all to itself, which is
  often more memory than the system actually has.

  Virtual addresses are interpreted by a processor's Memory Management Unit
  (mmu), using data structures called page tables which map virtual address
  ranges to associated content.

  Virtual memory is used to implement lazy allocation << demand paging >>,
  swapping, file mapping, copy on write shared memory, defragmentation, and
  more.
...
```

The Memory Hierarchy or Memory Pyramid

Source (diagram below)



CPU Registers :	300 ps
CPU Caches (L1/L2/L3) :	500 ps – 20 ns
RAM :	50 – 100 ns

Disk [swap] : … .... …….. 5 – 10 ms

Another View
Latency Numbers Every Programmer Should Know

Latency Comparison
Numbers

L1 cache reference 0.5 ns
Branch mispredict 5 ns
L2 cache reference 7 ns 14x L1 cache
Mutex lock/unlock 25 ns
Main memory reference 100 ns 20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy 3,000 ns 3 us
Send 1K bytes over 1 Gbps network 10,000 ns 10 us
Read 4K randomly from SSD* 150,000 ns 150 us ~1GB/sec SSD
Read 1 MB sequentially from memory 250,000 ns 250 us
Round trip within same datacenter 500,000 ns 500 us
Read 1 MB sequentially from SSD* 1,000,000 ns 1,000 us 1 ms ~1GB/sec SSD, 4X memory
Disk seek 10,000,000 ns 10,000 us 10 ms 20x datacenter roundtrip
Read 1 MB sequentially from disk 20,000,000 ns 20,000 us 20 ms 80x memory, 20X SSD
Send packet CA->Netherlands->CA 150,000,000 ns 150,000 us 150 ms

Notes
---
1 ns = 10^-9 seconds
1 us = 10^-6 seconds = 1,000 ns
1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns

Credit

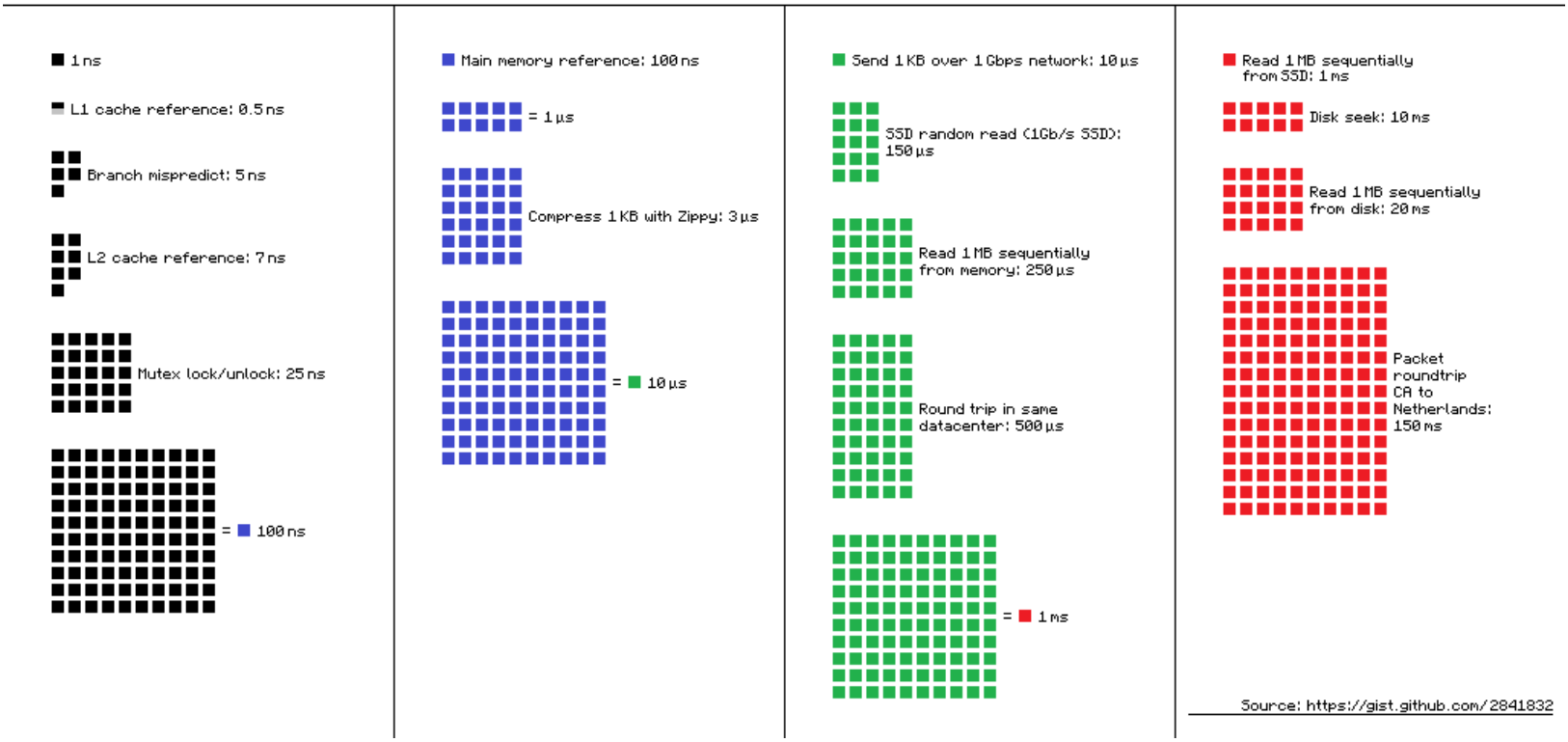
By Jeff Dean: http://research.google.com/people/jeff/
Originally by Peter Norvig: http://norvig.com/21-days.html#answers

Contributions

Some updates from: https://gist.github.com/2843375
'Humanized' comparison: https://gist.github.com/2843375
Visual comparison chart: http://i.imgur.com/k0t1e.png
Animated presentation: http://prezi.com/pdkvgys-r0y6/latency-numbers-for-programmers-web-development/latency.txt

http://i.imgur.com/k0t1e.png

Latency Numbers Every Programmer Should Know



Yet Another View

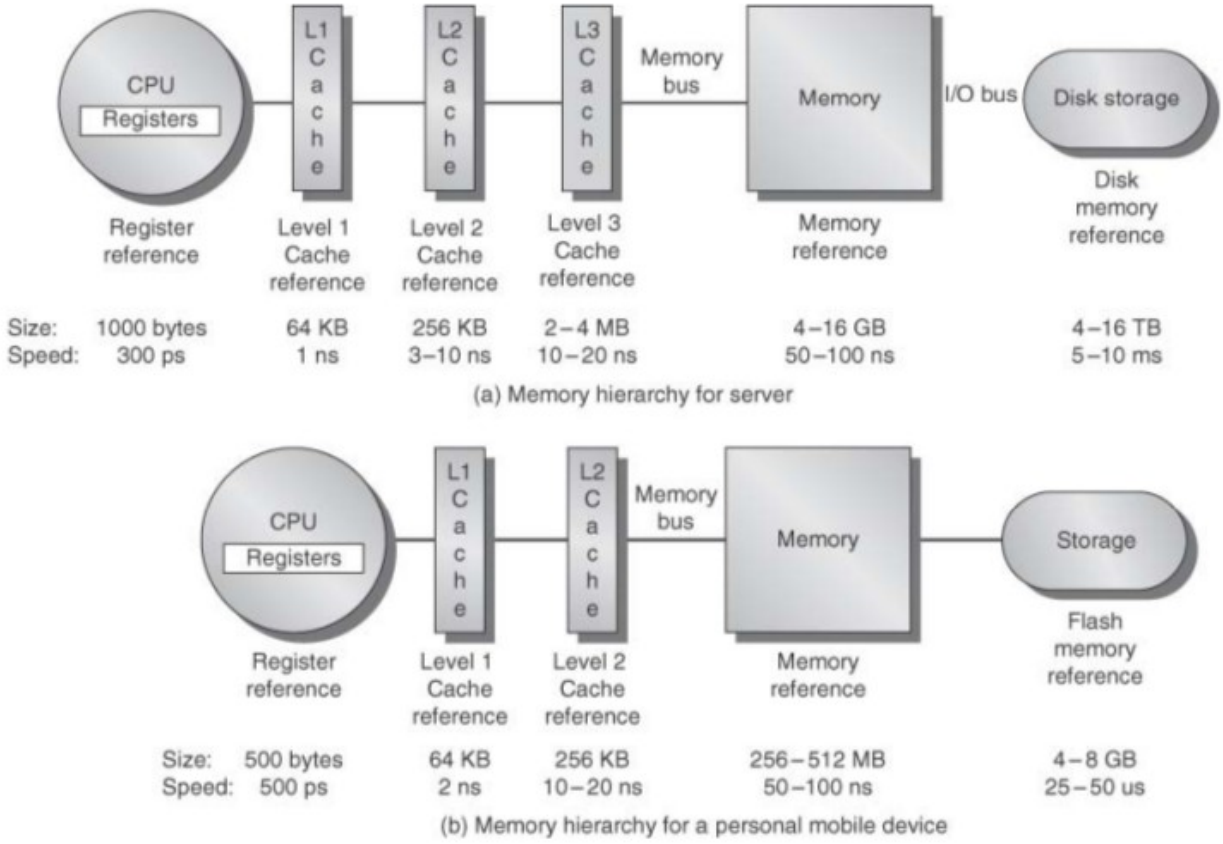


Figure 2.1 The levels in a typical memory hierarchy in a server computer shown on top (a) and in a personal mobile device (PMD) on the bottom (b).

As we move farther away from the processor, the memory in the level below becomes slower and larger. Note that the time units change by a factor of 10^9—from picoseconds to milliseconds—and that the size units change by a factor of 10^12 —from bytes to terabytes. The PMD has a slower clock rate and smaller caches and main memory. A key difference is that servers and desktops use disk storage as the lowest level in the hierarchy while PMDs use Flash, which is built from EEPROM technology.

Source (above diagram): Computer Architecture, A Quantitative Approach, 5th Ed by Hennessy & Patterson.

<< Source (below): http://www.heyrick.co.uk/assembler/memmult.html by Richard Murray. >>

Virtual memory

When you no longer have actual physical memory, you may have virtual memory. A set of memory locations that don't exist, but the operating system tries real hard to convince you they do. And in the centre of the ring is the MMU (Memory Management Unit, inspired name, no?) keeping control.
[*note: you need an MMU anyway when your memory is broken into remappable pages, this just seemed like a good time to introduce it!*]

When the processor is instructed to jump to &8000 to begin executing an application, it passes the address &8000 to the MMU. This translates the address into the correct real address and outputs this on the address lines, say &12FC00. The processor is not aware of this, the application is not aware of this, the computer user is not aware of this.

So we can take this one stage further by mapping onwards into memory that does not exist at all. In this case, the MMU will hiccup and say "Oi! You! *No!*" and the operating system will be called in a panic (correctly known as a "page fault"). The operating system will be calm and collected and think, "Ah, virtual memory". A little-used page of real memory will be shoved out to disc, then the page that the MMU was trying to find will be reloaded in place of the page we just got rid of. The memory map will be updated accordingly, then control will be handed back to the user application at the exact point the page fault occured. It would, unknowing of all of this palaver, perform that instruction again, only this time the MMU will (happily?) output the correct address to the memory system, and all will continue.

Page tables and the MMU

The page table exists to map each page into an address. This allows the operating system to keep track of which memory is pretending to be which. However it is more complex. Some pages cannot be remapped, some pages are doubly mapped, some are not to be touched in user mode code, some aren't to be touched at all. Some are read only. Some just don't exist. All of this must be kept track of.

So the MMU takes an address, looks it up in the page table, and spits out the correct address.

Let's do some maths. We'll assume a 4K page size (a la RISC OS in a RiscPC). A 32bit address space has a million pages. With one million pages, you'll need one million entries. In the ARM MMU, each entry takes 7 words. So we are looking at seven megabytes just to index our memory.

It gets better. *Every single memory reference* will be passed through the MMU. So we'll want it to operate in nanoseconds. Faster, if possible.

In reality, it is somewhat easier as most typical machines don't have enough memory to fill the entire addressing space, indeed many are unlikely to get close on technical reasons (the RiscPC can have 258Mb maximum RAM, or 514Mb with Kinetic - the extra 2Mb is the VRAM). Even so, the page tables will get large.

So there are three options:

- Have a huge array of fast registers in the MMU. Costly. Very.
- Hold the page tables in main memory. Slow. Very.
- Compromise. Cache the active pages in the MMU, and store the rest on disc.

An example. A RiscPC, 64Mb of RAM, 2Mb of VRAM, 4Mb of ROM and hardware I/O (double mapped). That's 734000320 bytes, or 17920 pages. It would take 71680 bytes to store each address. But an address on it's own isn't much use. Seven words comprise an entry in the ARM's MMU. So our 17920 pages would require 501760 bytes in order to fully index the memory.

You just can't store that lot in the MMU. So you'll store a snippet, say 16K worth?, and keep the rest in RAM.

The TLB

The Translation Lookaside Buffer is a way to make paging even more responsive. Typically, a program will make heavy use of a few pages and barely touch the rest. Even if you plan to byte read the entire memory map, you will be making four thousand hits in one page before going to the next.

A solution to this is to fit a little bit in the MMU that can map virtual addresses to their physical counterparts without traversing the page table. This is the TLB. It lives within the MMU and contains details of a small number of pages (usually between four and sixty four - the ARM610 MMU TLB has thirty two entries).

Now, when we have a page lookup, we first pass our virtual address to the TLB which will check all of the addresses stored, and the protection level. If a match is found, the TLB will spit out the physical address and the page table isn't touched.

If a miss is encountered, then the TLB will evict one of it's entries and load in the page information looked up in the page table, so the TLB will know the new page requested, so it can quickly satisfy the result for the next memory access, as chances are the next access will be in the page just requested.

So far we have figured on the hardware doing all of this, as in the ARM (and x86) processor. Some RISC processors (such as the Alpha and the MIPS) will pass the TLB miss problem to the operating system. This may allow the OS to use some intelligence to pre-load certain pages into the TLB.

Read an unusual, satiric, fantastic, very brief intro to what VM is really all about here: "The Thing King".

Source – Memory FAQ

...

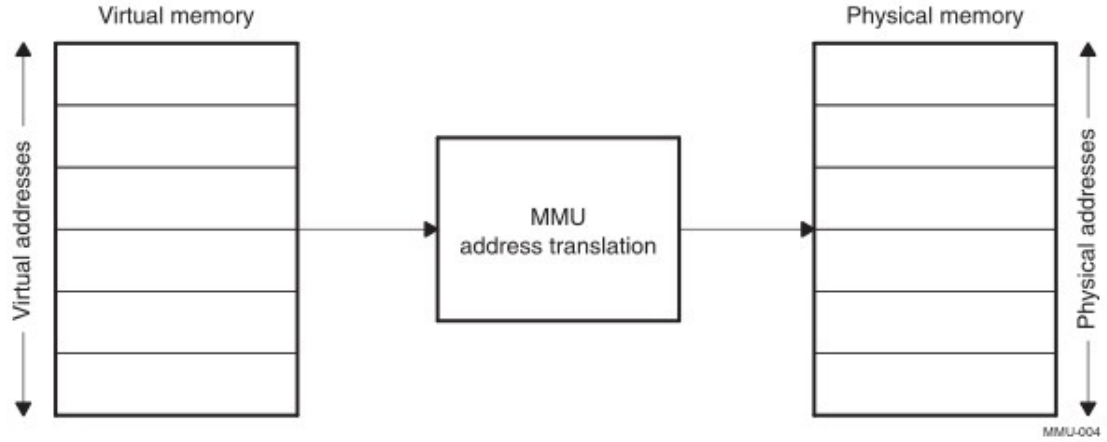
What is a Memory Management Unit (MMU)?

```
  The memory management unit is the part of the CPU that interprets virtual
  addresses.  Attempts to read, write, or execute memory at virtual addresses
  are either translated to corresponding physical addresses, or else
  generate an interrupt (page fault) to allow software to respond to the
  attempted access.

  This gives each process its own virtual memory address range, which is
  limited only by address space (4 gigabytes on most 32-bit system), while
  physical memory is limited by the amount of available storage hardware.

  Physical memory addresses are unique in the system, virtual memory addresses
  are unique per-process.
...
```

<< Source: OMAP35x Application Processor, Technical Reference Manual [SPRUF98T], (c) TI Inc. >>

Memory Management Unit (MMU) Functional Description

MMU Address Translation



MMU Address Translation

MMU Benefits

The MMU offers two major benefits:

• Memory defragmentation: Fragmented physical memory can be translated into contiguous virtual memory without moving data.
• Memory protection: Illegal, that is, non-allowed accesses to memory locations can be detected and prevented.

...

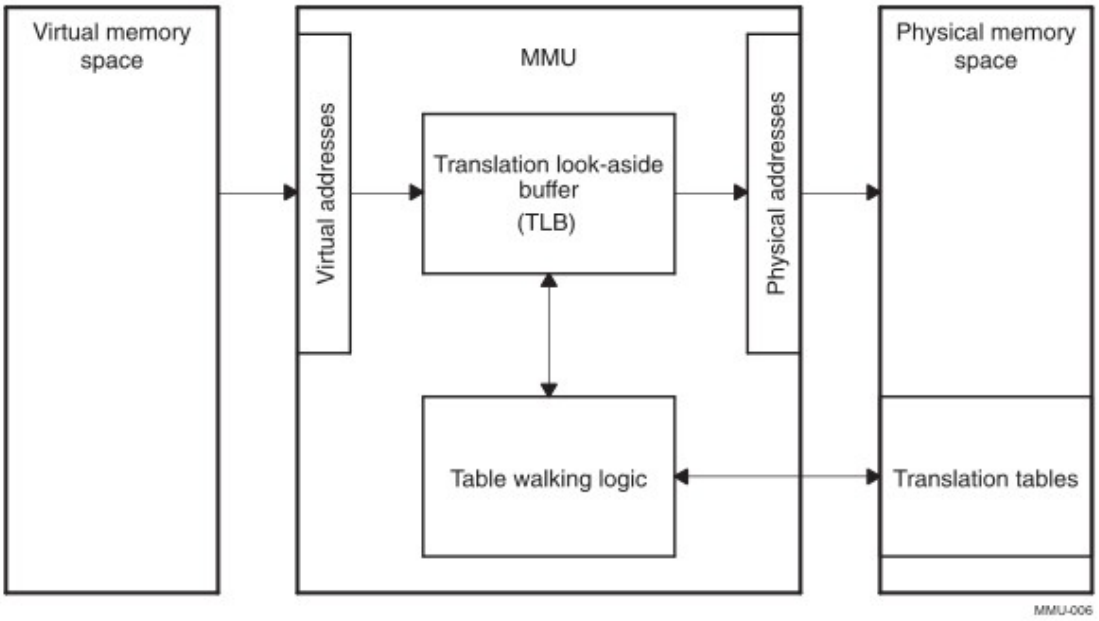
Two major functional units exist in the MMU to provide address translation automatically based on the table entries:

- The table walker automatically retrieves the correct translation table entry for a requested translation. If two-level translation is used (for the translation of small memory pages), the table walker also automatically reads the required second-level translation table entry.

- The TLB stores recently used translation entries, acting like a cache of the translation table.

This basic architecture is outlined in Figure 8-6.

Figure 8-6. MMU Architecture



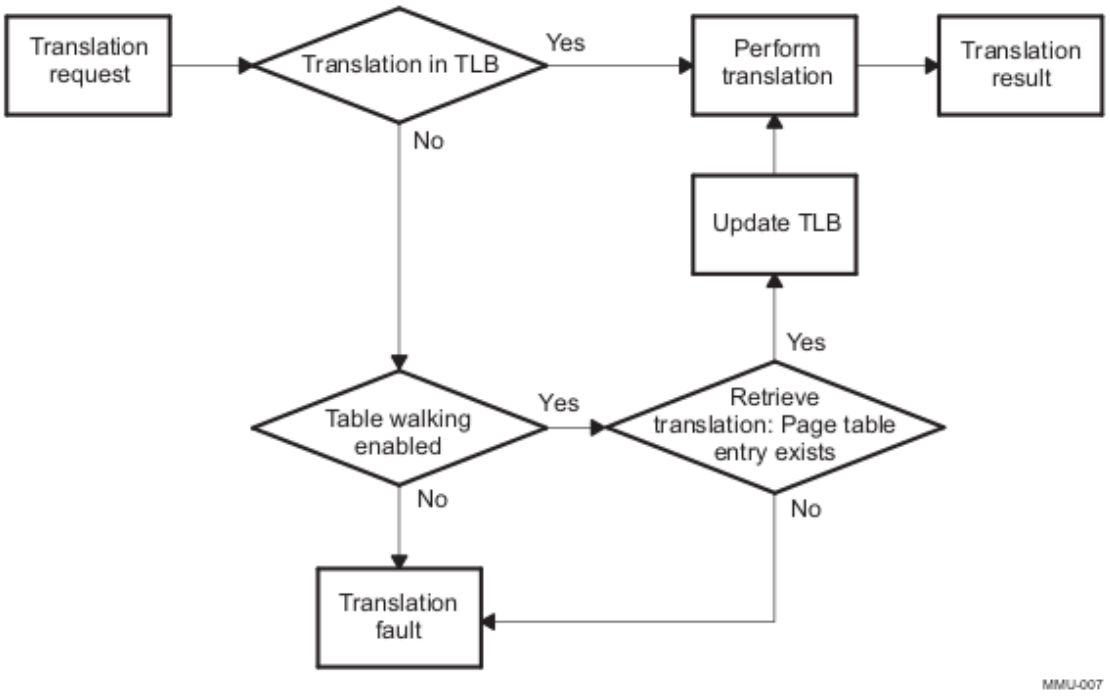
<<
Note that each processor core has an MMU (and therefore TLB); this is required as, at a given point in time, a separate thread (or process) can run on each core.
>>

MMU Address Translation Process

Whenever an address translation is requested (that is, for every access with the MMU enabled), the MMU first checks whether the translation is already contained in the TLB, which acts like a cache storing recent translations. The TLB can also be programmed manually to ensure that time-critical data can be translated without delay.

If the requested translation is not in the TLB, the table-walking logic retrieves this translation from the translation table(s), and then updates the TLB. The address translation is then performed. Figure 8-7 summarizes the process.

Figure 8-7. Translation Process



Source: Virtual Machines: Memory

…

Setting one control register is not an expensive operation, but on an x86 it must be accompanied by a TLB flush, which results in new table walks to fill it again; and that might be quite expensive. This is why both Intel and AMD have recently added Address Space Identifiers (ASIDs) to TLB entries in their newest processors. TLBs with ASIDs don't have to be flushed, the processor will just ignore the entries with the wrong ASIDs.

…

Below Source: Wikipedia page on TLB

…

Typical TLB

These are typical performance levels of a TLB:[15]

size: 12 – 4,096 entries
hit time: 0.5 – 1 clock cycle
miss penalty: 10 – 100 clock cycles
miss rate: 0.01 – 1%

If a TLB hit takes 1 clock cycle, a miss takes 30 clock cycles, and the miss rate is 1%, the effective memory cycle rate is an average of 1 × 0.99 + (1 + 30) × 0.01 = 1.30 (1.30 clock cycles per memory access).

…

Levels of Privilege

Full isolation would be impossible if a user program could directly manipulate page tables. This is something that only the trusted operating system should be able to do. For instance, a user program should not be able to set the value of CR3. This prohibition has to be enforced at the processor level, which designates certain instructions as privileged. Such instructions must be accessible from the kernel of the operating system, but cause a fault when a user process tries to execute them. The x86 must therefore be able to distinguished these two modes of operations: kernel (a.k.a., supervisor) and user. For historical reasons, the x86 offers four levels of protection called rings: ring 0 corresponding to kernel mode and rings 1, 2, and 3 to user mode. In practice, only ring 0 and 3 are used, except for some implementations of virtualization.

When the operating system boots, it starts in kernel mode, so it can set up the page tables and do other kernel-only things. When it starts an application, it creates a ring 3 process for it, complete with separate page tables. The application may drop back into the kernel only through a trap, for instance when a page fault occurs. A trap saves the state of the processor, switches the processor to kernel mode, and executes the handler that was registered by the OS.

Processor Cache

The word cache is a French word meaning “a concealed place for storage.”

The improvement a cache provides is possible because computer programs execute in nonrandom ways. Predictable program execution is the key to the success of cached systems. If a program’s accesses to memory were random, a cache would provide little improvement to overall system performance. The principle of locality of reference explains the performance improvement provided by the addition of a cache memory to a system. This principle states that computer software programs frequently run small loops of code that repeatedly operate on local sections of data memory.

<< Source: “Computer Architecture: A Quantitative Approach”, Hennessy & Patterson, 5th Ed., pg 45:
...
“The most important program property that we regularly exploit is the *principle of locality*: Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code.

An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data access, though not as not as strongly as to code accesses.

Two different types of locality have been observed:
Temporal locality states that recently accessed items are likely to be accessed in the near future.

Spatial locality says that items that are near one another tend to be referenced close together in time.

[Temporal => in time
Spatial => in space]
>>

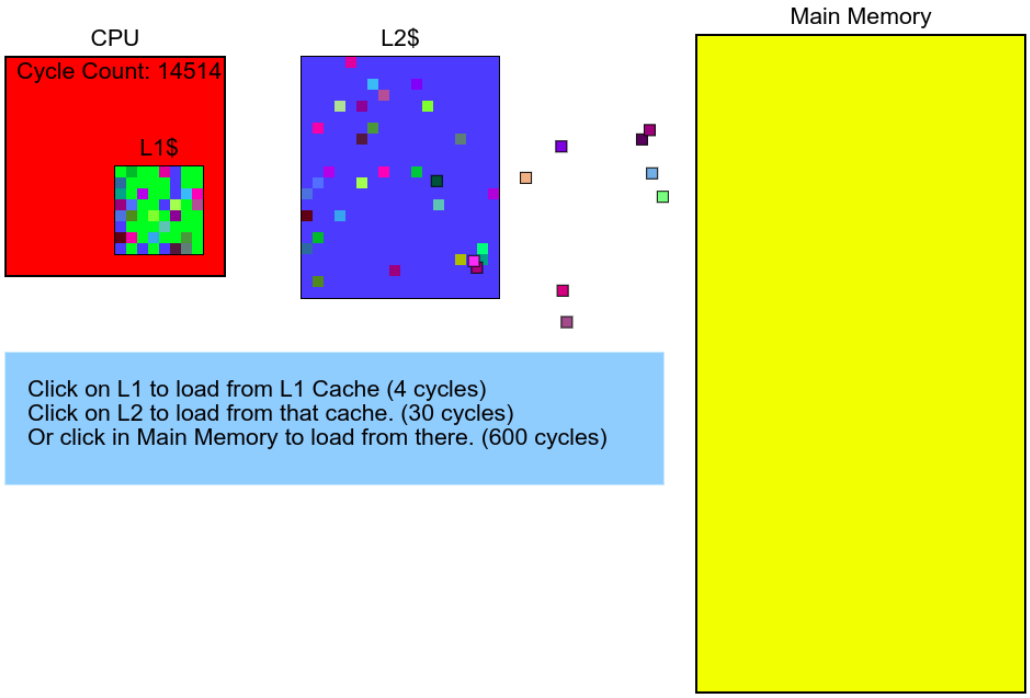
The repeated use of the same code or data in memory, or those very near, is the reason a cache improves performance. By loading the referenced code or data into faster memory when first accessed, each subsequent access will be much faster. It is the repeated access to the faster memory that improves performance.

The cache makes use of this repeated local reference in both time and space. If the reference is in time, it is called temporal locality. If it is by address proximity, then it is called spatial locality.

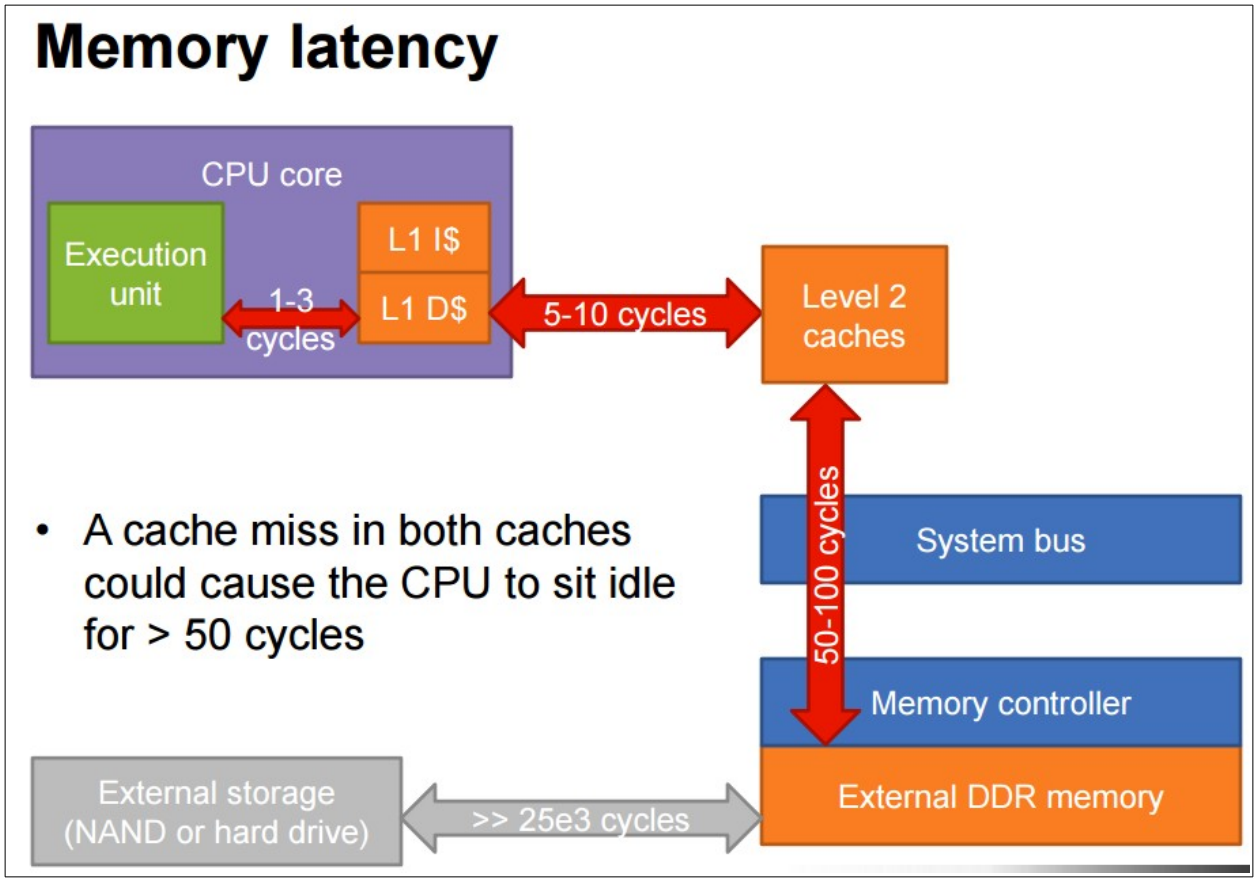
<<

MUST SEE and TRY (it's interactive) !!!

http://www.overbyte.com.au/misc/Lesson3/CacheFun.html



Also, from here:


Memory latency
CPU core
Execution unit
1-3 cycles
L1 I$
L1 D$
5-10 cycles
Level 2 caches
50-100 cycles
System bus
Memory controller
External DDR memory
External storage (NAND or hard drive)
>> 25e3 cycles
• A cache miss in both caches could cause the CPU to sit idle for > 50 cycles

>>

[FYI / OPTIONAL]

Looking up CPU Caches – Some Numbers !

Note that actual caches/TLBs being present and their sizes is highly processor model dependant. For typical types and sizes on Intel-32 and Intel-64 processors, see the "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide" "Table 11-1. Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors" page # 11-2.

Some real-world examples follow:

Example 1

Eg. Running 'cpuid' on a generic ***Intel Core-i5*** *(3rd gen) desktop to see processor cache details:*

```
# cpuid
...
CPU 0:
   vendor_id = "GenuineIntel"
   version information (1/eax):
      processor type  = primary processor (0)
      family          = Intel Pentium Pro/II/III/Celeron/Core/Core 2/Atom, AMD Athlon/Duron,
Cyrix M2, VIA C3 (6)
      model           = 0xa (10)
      stepping id     = 0x7 (7)
      extended family = 0x0 (0)
      extended model  = 0x2 (2)
      (simple synth)  = Intel Core i3-2000 / Core i5-2000 / Core i7-2000 / Mobile Core i7-
2000 (Sandy Bridge D2/J1/Q0) / Pentium G500/G600/G800 (Sandy Bridge Q0) / Celeron
G400/G500/700/800/B800 (Sandy Bridge J1/Q0) / Xeon E3-1200 (Sandy Bridge D2/J1/Q0), 32nm
...
cache and TLB information (2):
      0x5a: data TLB: 2M/4M pages, 4-way, 32 entries
      0x03: data TLB: 4K pages, 4-way, 64 entries
      0x76: instruction TLB: 2M/4M pages, fully, 8 entries
      0xff: cache data is in CPUID 4
      0xb2: instruction TLB: 4K, 4-way, 64 entries
      0xf0: 64 byte prefetching
      0xca: L2 TLB: 4K, 4-way, 512 entries
   processor serial number: 0002-06A7-0000-0000-0000-0000
   deterministic cache parameters (4):
      --- cache 0 ---
      cache type                           = data cache (1)
      cache level                          = 0x1 (1)

--snip--
```

Example 2

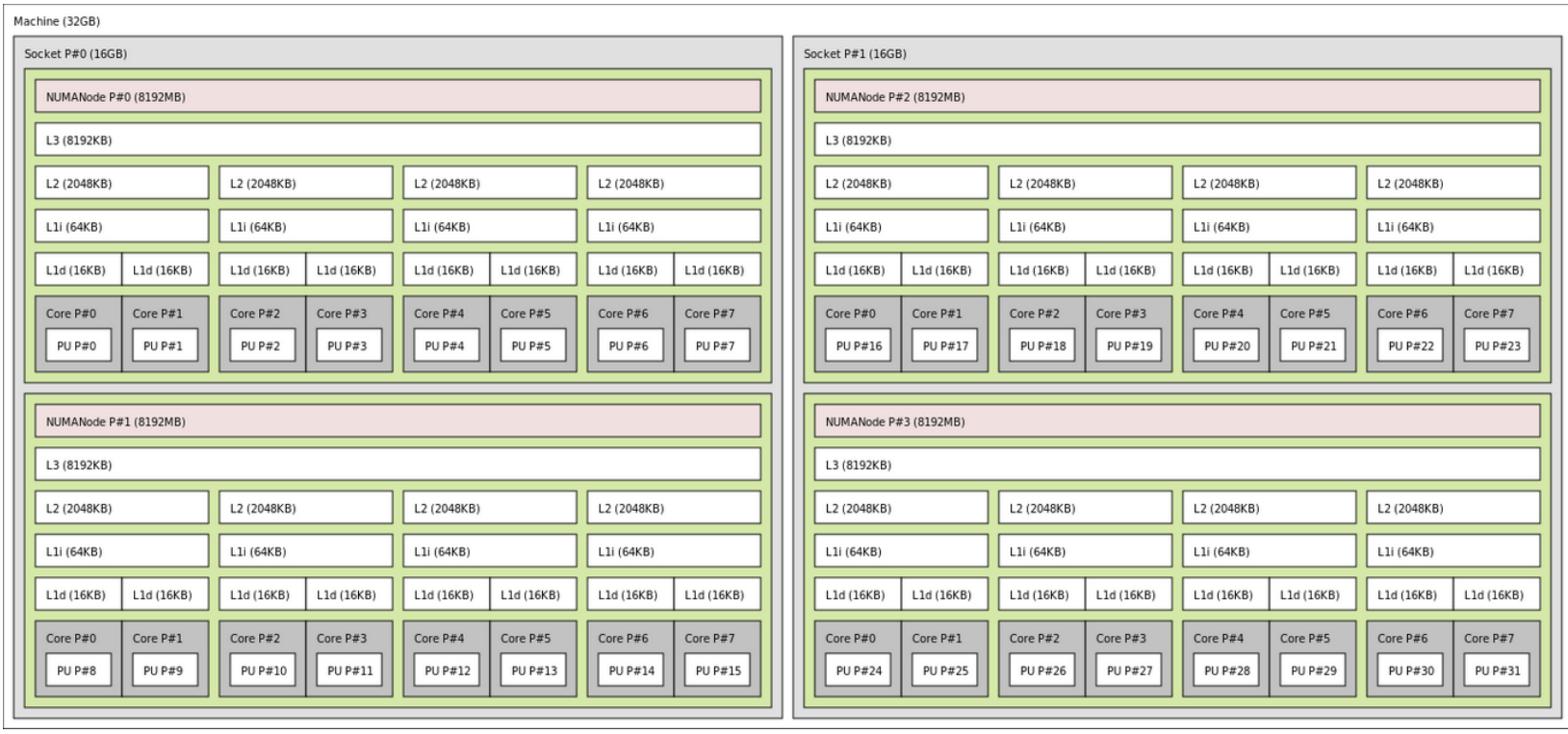
Eg. Running 'lshw' on a Lenovo IdeaPad Y550 laptop ***(Intel Core 2 Duo cpu)*** *to see processor cache details:*

```
# lshw
--snip--
     *-cpu:0
          description: CPU
          product: Intel(R) Core(TM)2 Duo CPU     T6500  @ 2.10GHz
```

```
          vendor: Intel Corp.
--snip--
          configuration: cores=2 enabledcores=2 id=1 threads=2
        *-cache:0
             description: L1 cache
             physical id: 5
             slot: L1 Cache
             size: 64KiB
             capacity: 64KiB
             capabilities: asynchronous internal write-back data
        *-cache:1
             description: L2 cache
             physical id: 6
             slot: L2 Cache
             size: 2MiB
             capacity: 4MiB
             capabilities: burst internal write-back unified
--snip--
#
```

[`$ getconf -a |grep -i cache`
: another way to get more CPU cache details. Also, can try using dmidecode..]
>>

Example 3
An example (pic) from the Wikipedia page on CPU Cache:
Memory hierarchy of an AMD Bulldozer server



Source: https://upload.wikimedia.org/wikipedia/commons/9/95/Hwloc.png
…

[
Good Article: ***How L1 and L2 CPU caches work, and why they're an essential part of modern chips, Joel Hruska, Feb 2016***

…
When AMD's Bulldozer family is compared against Intel's processors, the topic of cache design and performance impact comes up a great deal. It's not clear how much of Bulldozer's lackluster performance can be blamed on its relatively slow cache subsystem — in addition to having relatively high latencies, the Bulldozer family also suffers from a high amount of cache contention. Each Bulldozer/Piledriver/Steamroller module shares its L1 instruction cache,
…
]

Example 4

ARMv8

…

	Cortex-A53	Cortex-A57
L1 Cache size (Instruction)	8KB to 64 KB	48KB
L1 Cache structure (Instruction)	2-way set associative	3-way set associative
L1 Cache size (Data)	8KB to 64KB	32KB
L1 Cache structure (Data)	4-way set associative	2-way set associative
L2 Cache	Optional	Integrated
L2 Cache size	128KB to 2MB	512KB to 2MB
L2 Cache structure	16-way set associative	16-way set associative
Main TLB entries	512	1024
uTLB entries	10	48 I-side 32 D-side

Source

ARM Cortex-A15 MPCore

…

The MMU controls table walk hardware that accesses translation tables in memory. The MMU works with the L1 and L2 memory system to translate virtual addresses to physical addresses. The MMU enables fine-grained memory system control through a set of virtual-to-physical address mappings and memory attributes held in the L1 and L2 Translation Look-aside Buffers (TLBs).

The Cortex-A15 MMU features include the following:

- 32-entry fully-associative L1 instruction TLB.
- Two separate 32-entry fully associative L1 TLBs for data load and store pipelines.
- 4-way set-associative 512-entry L2 TLB in each processor.
- Intermediate table walk caches.
- The TLB entries contain a global indicator or an Address Space Identifier (ASID) to permit context switches without TLB flushes.
- The TLB entries contain a Virtual Machine Identifier (VMID) to permit virtual machine switches without TLB flushes.

…

ARM Cortex-A9 MPCore *:* *Source*

…

The MMU features include the following:

- Instruction side micro TLB
  - 32 fully associative entries
- Data side micro TLB

    - 32 fully associative entries

- Unified main TLB
    - unified, 2-way associative, 2x32 entry TLB
    - support for 4 lockable entries using the lock-by-entry model
    - pseudo round-robin replacement policy
- supports hardware page table walks to perform look-ups in the L1 data cache.

….

ARM11 Microarchitecture (specifically, the Broadcom BCM2835 SoC on the Raspberry Pi's)

Example 5
Source: "White Paper Intel® Next Generation Microarchitecture (Nehalem)" PDF :

…
Next generation Intel microarchitecture (Nehalem) enhances the Intel Smart Cache by adding an inclusive shared L3 (last-level) cache that can be up to 8MB in size. In addition to this cache being shared across all cores, the inclusive shared L3 cache can increase performance while reducing traffic to the processor cores.

Some architectures use exclusive L3 cache, which contains data not stored in other caches. Thus, if a data request misses on the L3 cache, each processor core must still be searched, or snooped, in case their individual caches might contain the requested data. This can increase latency and snoop traffic between cores. With next generation microarchitecture, a miss of its inclusive shared L3 cache guarantees the data is outside the processor and thus is designed to eliminate unnecessary core snoops to reduce latency and improve performance.

The new three-level cache hierarchy for next generation Intel microarchitecture (Nehalem) consists of:

- Same L1 cache as Intel Core microarchitecture (32 KB Instruction Cache, 32 KB Data Cache)
- New L2 cache per core for very low latency (256 KB per core for handling data and instruction)
- New fully inclusive, fully shared 8MB L3 cache (all applications can use entire cache)

A new two-level Translation Lookaside Buffer (TLB) hierarchy is also included in next generation Intel microarchitecture (Nehalem). A TLB is a processor cache that is used by memory management hardware to improve the speed of virtual address translation. The TLB references physical memory addresses in its table.

All current desktop and server processors use a TLB, but next generation Intel microarchitecture (Nehalem) adds a new second level 512 entry TLB to further improve performance.
...

Example 6

<<

Intel mid-to-high-end: *Intel Nehalem microarchitecture*.

From here

... Having said that, these days, our CPUs have much larger caches, and can even have an L3 cache.

- (Intel) 5150: L1i & L1d = 32K each, L2 = 4M
- E5440: L1i & L1d = 32K each, L2 = 6M
- E5520: L1i & L1d = 32K each, L2 = 256K/core, L3 = 8M (same for the X5550)
- L5630: L1i & L1d = 32K each, L2 = 256K/core, L3 = 12M
- E5-2620: L1i & L1d = 64K each, L2 = 256K/core, L3 = 15M

Note that in the case of the E5520/X5550/L5630 (the ones marketed as "i7") as well as the Sandy Bridge E5-2520, the L2 cache is tiny but there's one L2 cache per core (with HT enabled, this gives us 128K per hardware thread). The L3 cache is shared for all cores that are on each physical CPU.

...

>>

Example 7

…

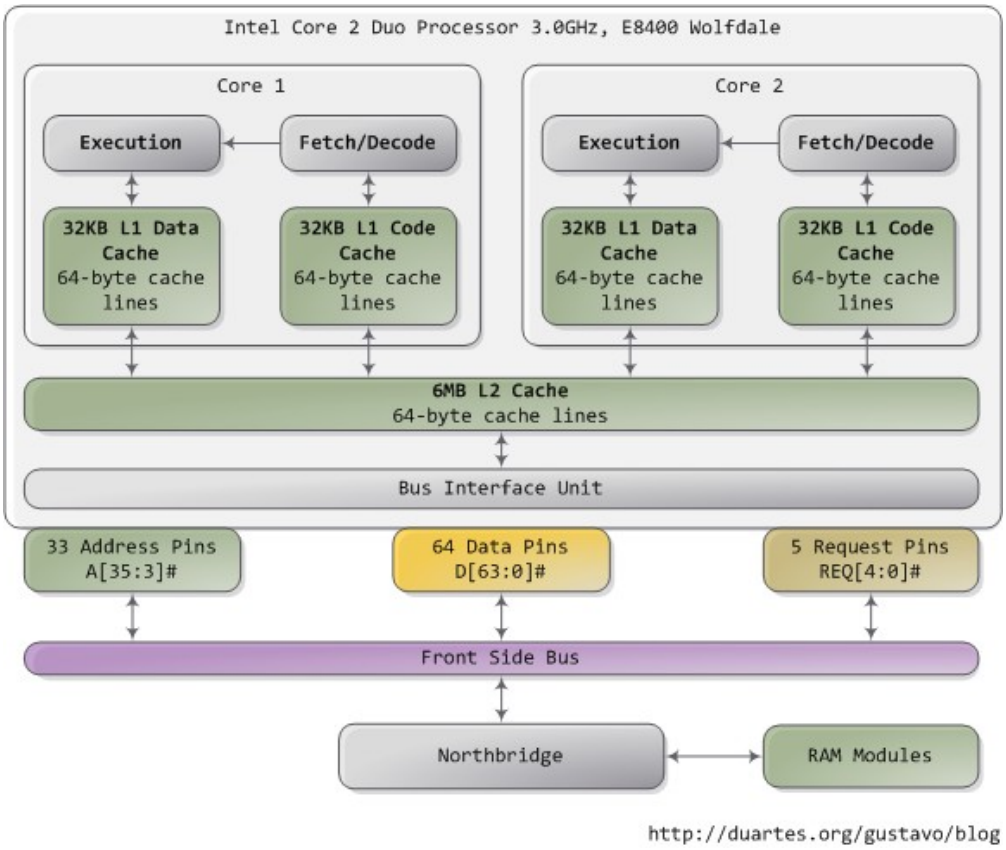
Intel introduced a Level 4 on-package cache with the Haswell microarchitecture. Crystalwell[21] Haswell CPUs, equipped with the GT3e variant of Intel's integrated Iris Pro graphics, effectively feature 128 MB of embedded DRAM (eDRAM) on the same package. This L4 cache is shared dynamically between the on-die GPU and CPU, and serves as a victim cache to the CPU's L3 cache.[22]

…

Getting Physical With Memory, Gustav Duarte

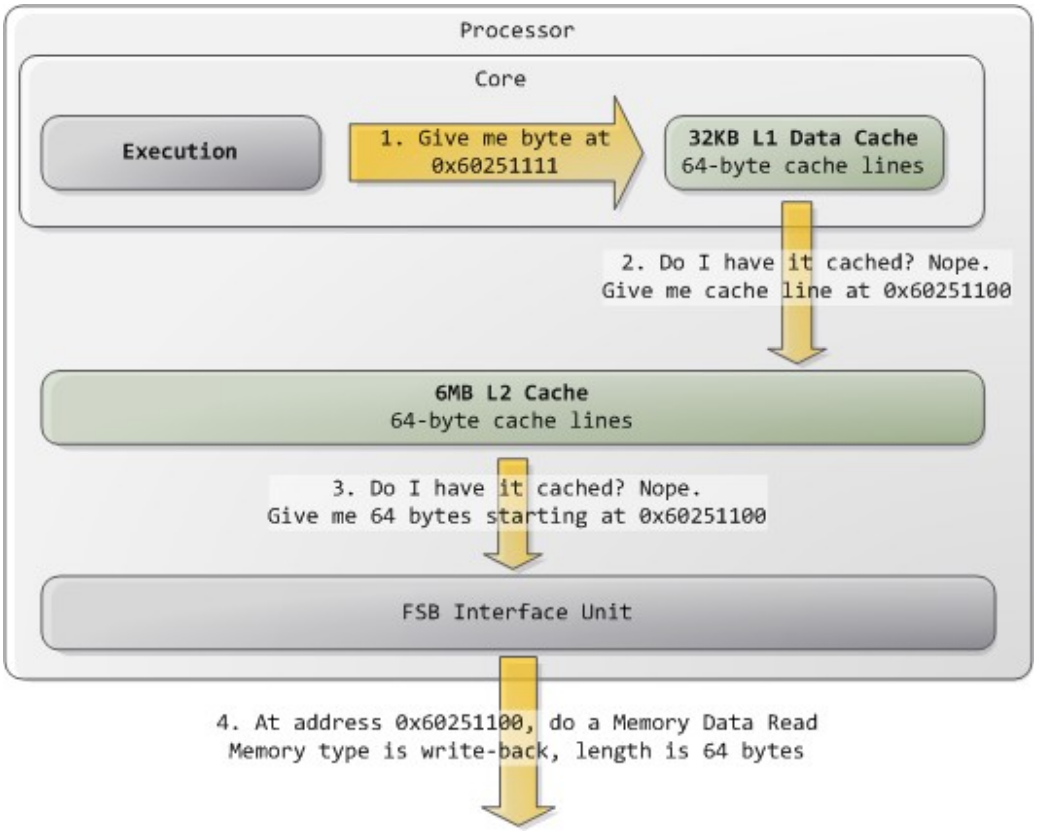
Modern processors do not read/write memory, meaning RAM, directly. Instead, they use a caching scheme to exploit the “read-ahead / write-behind” optimization advantages that a fast cache can provide. Processors usually have several “levels” of fast (single digit nanosecond access speed) caches- usually called L1, L2, L3 caches.

To be more specific, we take the Intel Core 2 Duo processor as a representative example. It has 2 cores: each core has an 32 KB L1 Data Cache (L1-D) and an L1 Code Cache (L1-I(nstruction)), with a cache-line size of 64 bytes. There is a 6 MB Level 2 (L2) Cache with 64-byte with a 64 byte cache-line size.



…
Typically kernels treat **all RAM memory as write-back**, which yields the best performance. In write-back mode the unit of memory access is the cache line, 64 bytes in the Core 2.

If a program reads a single byte in memory, the processor loads the whole cache line that contains that byte into the L2 and L1 caches. When a program *writes* to memory, the processor only modifies the line in the cache, but does *not* update main memory. Later, when it becomes necessary to post the modified line to the bus, the whole cache line is written at once. So most requests have 11 in their length field, for 64 bytes. Here's a read example in which the data is not in the caches:



…
The primitives discussed here have many implications. For example:

1. Performance-sensitive applications should try to pack data that is accessed together into the same cache line. Once the cache line is loaded, further reads are much faster and extra RAM accesses are avoided.

2. Any memory access that falls within a single cache line is guaranteed to be atomic (assuming write-back memory). Such an access is serviced by the processor's L1 cache and the data is read or written all at once; it cannot be affected halfway by other processors or threads. In particular, 32-bit and 64-bit operations that don't cross cache line boundaries are atomic.

3. The front bus is shared by all agents, who must arbitrate for bus ownership before they can start a transaction. Moreover, all agents must listen to all transactions in order to maintain cache coherence. Thus bus contention becomes a severe problem as more cores and processors are added to Intel computers. The Core i7 solves this by having processors attached directly to memory and communicating in a point-to-point rather than broadcast fashion.

<<
A **real-world example (within the kernel network subsystem) of why processor caching times *do* very much matter indeed**:

please read this blog article:
"The calculations: 10Gbit/s wirespeed", Jesper D Brouer.
>>

Resources:
CPU Caches and Why You Care - Scott Meyers (YouTube)
-Slides

<<
Logical and Physical Caches

Logical (or virtual) Cache
– stores data between the processor and MMU.

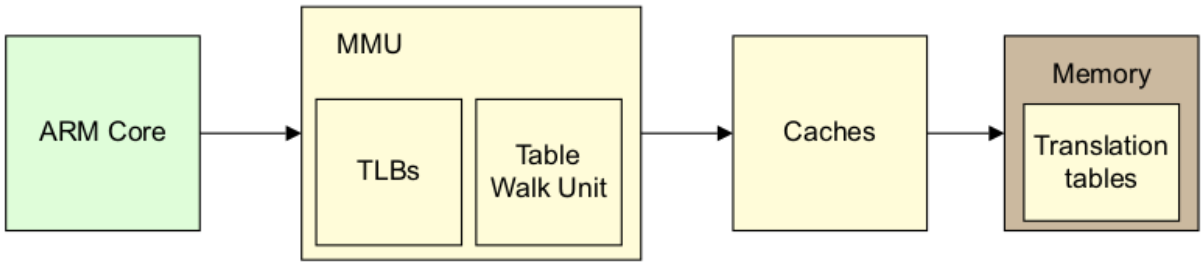
[Processor] <---> [Cache] <---> [MMU] <----------> [Main Memory (RAM)]
va va Addr Translation (TLB) [va→pa]

- ARM7 to ARM10 processor cores (i.e. ARM ISA v4, v5).

Physical Cache
– stores data between the MMU and main memory.

[Processor] <----------> [MMU] <----> [Cache] <----> [Main Memory (RAM)]
Addr Translation (TLB)[va→pa] pa

- ARM11 processor cores (i.e. ARM ISA v6 onward).



Memory System on the ARMv8

Resource:
Logical versus physical caches

A very detailed discussion on the design, implementation and impact of cache and SMP for kernel programmers is exhaustively dealt with in the book "UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers (Addison-Wesley Professional Computing Series)" by Curt Schimmel.

>>

[OPTIONAL / FYI]

Processor / Hardware Components to improve Memory Latencies

Source: Memory Access Patterns are Important by Martin Thompson

…

… Our hardware tries to hide the main-memory latency via a number of techniques. Basically ***three major bets are taken on memory access patterns:***

1. **Temporal**: Memory accessed recently will likely be required again soon.
2. **Spatial**: Adjacent memory is likely to be required soon.
3. **Striding**: Memory access is likely to follow a predictable pattern.

[Temporal => Time
Spatial => Space]

<< The author of this article then writes a Java application that "walks" around memory using the above "patterns" in a programmable fashion, and then computes the results via the perf and likwid (lightweight performance counters) projects. The basic results are shown below; please read the article in full for an in-depth understanding! >>
…

To illustrate these three bets in action let's write some code and measure the results.

1. Walk through memory in a linear fashion being completely predictable.
   << LINEAR_WALK >>
2. Pseudo randomly walk round memory within a restricted area then move on. This restricted area is what is commonly known as an operating system page of memory.
   << RANDOM_PAGE_WALK >>
3. Pseudo randomly walk around a large area of the heap.
   << RANDOM_HEAP_WALK >>

…

What does this mean for algorithms?

The difference between an L1D cache-hit, and a full miss resulting in main-memory access, is 2 orders of magnitude; i.e. <1ns vs. 65-100ns. If algorithms randomly walk around our ever increasing address spaces, then we are less likely to benefit from the hardware support that hides this latency.

Is there anything we can do about this when designing algorithms and data-structures? Yes there is a lot

we can do. If we perform chunks of work on data that is co-located, and we stride around memory in a predictable fashion, then our algorithms can be many times faster. For example rather than using bucket and chain hash tables, like in the JDK, we can employ hash tables using open-addressing with linear-probing. Rather than using linked-lists or trees with single items in each node, we can store an array of many items in each node.

Research is advancing on algorithmic approaches that work in harmony with cache sub-systems. One area I find fascinating is Cache Oblivious Algorithms. The name is a bit misleading but there are some great concepts here for how to improve software performance and better execute in parallel.
This article is a great illustration of the performance benefits that can be gained.

Conclusion

To achieve great performance it is important to have sympathy for the cache sub-systems. We have seen in this article what can be achieved by accessing memory in patterns which work with, rather than against, these caches. When designing algorithms and data structures, it is now vitally important to consider cache-misses, probably even more so than counting steps in the algorithm. This is not what we were taught in algorithm theory when studying computer science. The last decade has seen some fundamental changes in technology. For me the two most significant are the rise of multi-core, and now big-memory systems with 64-bit address spaces.

One thing is certain, if we want software to execute faster and scale better, we need to make better use of the many cores in our CPUs, and pay attention to memory access patterns.

Update: 06-August-2012
Trying to design a random walk algorithm for all processors and memory sizes is tricky. If I use the algorithm below then my Sandy Bridge processor is slower but the Nehalem is faster. The point is performance will be **very**unpredictable when you walk around memory in a random fashion. I've also included the L3 cache counters for more detail in all the tests.

...

Another article along similar lines:

Gallery of Processor Cache Effects

NOTE !!

Going overboard with cache-line sharing is not always a great idea! Caution is required, as incorrect cacheline sharing can result in a detrimental situation known as "False Sharing" (seen later).

A note on Security

Watch out: having a structure of only function pointers:

Source- The status of kernel hardening
…

Perhaps more controversial is struct layout randomization. That cannot be done in a general way without causing all kinds of problems, but there is one place where it is especially useful: structs consisting of only function pointers. Such structs are one of the most prized targets for attackers, and the kernel has a lot of them. A GCC plugin can be used to detect these structures and randomize their order. In general, the kernel shouldn't care about that ordering, and changing it should not have performance effects.

Linus was not entirely convinced; he said that most people are running distributor kernels, so the specific ordering used will always be available to an attacker. The value, Kees responded, is forcing attackers to identify specific kernel builds; that is "excruciating" for them. It greatly expands the number of settings their exploit has to work in.

…

A quick test

Our programs deal with virtual memory addresses that need to be translated to physical memory addresses. Virtual memory systems do this by mapping pages. We need to know the offset for a given page and its size for any memory operation.

Typically page sizes are 4KB and are gradually moving to 2MB and greater. Linux introduced Transparent Huge Pages in the 2.6.38 kernel giving us 2MB pages.

The translation of virtual memory pages to physical pages is maintained by the page table. This translation can require multiple accesses to the page table which is a huge performance penalty. To accelerate this lookup, processors have a small hardware cache at each cache level called the TLB cache. A miss on the TLB cache can be hugely expensive because the page table may not be in a nearby data cache. By moving to larger pages, a TLB cache can cover a larger address range for the same number of entries.

```
$ perf stat -e dTLB-loads,dTLB-load-misses java -Xmx4g TestMemoryAccessPatterns

 Performance counter stats for 'java -Xmx4g TestMemoryAccessPatterns 1':
   << LINEAR_WALK >>
     1,496,128,634 dTLB-loads
           310,901 dTLB-misses
              #    0.02% of all dTLB cache hits

 Performance counter stats for 'java -Xmx4g TestMemoryAccessPatterns 2':
   << RANDOM_PAGE_WALK >>
     1,551,585,263 dTLB-loads
           340,230 dTLB-misses
              #    0.02% of all dTLB cache hits

 Performance counter stats for 'java -Xmx4g TestMemoryAccessPatterns 3':
   << RANDOM_HEAP_WALK >>
     4,031,344,537 dTLB-loads
     1,345,807,418 dTLB-misses
              #   33.38% of all dTLB cache hits
```

Note: We only incur significant TLB misses when randomly walking the whole heap when huge pages are employed.

…

---

include/linux/blkdev.h

```
…
/*
 * Try to put the fields that are referenced together in the same cacheline.
 *
 * If you modify this structure, make sure to update blk_rq_init() and
 * especially blk_mq_rq_ctx_init() to take care of the added fields.
 */
struct request {
        struct request_queue *q;
...
```

[OPTIONAL / FYI]

Source: Professional Linux Kernel Architecture, Mauerer, Wrox Press

Processor Cache and TLB Control on the Linux OS

Caches are crucial in terms of overall system performance, which is why the kernel tries to exploit them as effectively as possible. It does this primarily by skillfully aligning kernel data in memory.

A judicious mix of normal functions, inline definitions, and macros also helps extract greater performance from the processor. The compiler optimizations discussed in Appendix C also make their contribution.

However, the above aspects affect the cache only indirectly. Use of the correct alignment for a data structure does indeed have an effect on the cache but only implicitly — active control of the processor cache is not necessary.

Nevertheless, the kernel features some commands that act directly on the cache and the TLB of the processor. However, they are not intended to boost system efficiency but to maintain the cache contents in a consistent state and to ensure that no entries are incorrect and out-of-date.

For example, when a mapping is removed from the address space of a process, the kernel is responsible for removing the corresponding entries from the TLBs. If it failed to do so and new data were added at the position previously occupied by the mapping, a read or write operation to the virtual address would be redirected to the incorrect location in physical memory.

The hardware implementation of caches and TLBs differs significantly from architecture to architecture. The kernel must therefore create a view on TLBs and caches that takes adequate account of the different approaches without neglecting the specific properties of each architecture.

❑ The meaning of the *translation lookaside buffer* is abstracted to refer to a mechanism that translates a virtual address into a physical address. [38]

[38] Whether TLBs are the only hardware resource for doing this or whether other alternatives (e.g., page tables) are provided is irrelevant.

❑ The kernel regards a cache as a mechanism that provides rapid access to data by reference to a virtual address without the need for a request to RAM memory. There is not always an explicit difference between data and instruction caches. The architecture-specific code is responsible for any differentiation if its caches are split in this manner.

It is not necessary for each processor type to implement every control function defined by the kernel. If a function is not required, its invocation can be replaced with an empty operation `( do {} while (0) )` that is optimized away by the compiler.

This is very frequently the case with cache-related operations because, as above, the kernel assumes that addressing is based on virtual addresses. The resultant problems do not occur in physically organized caches so that it is not usually necessary to implement the cache control functions.

The following functions must be made available (even if only as an empty operation) by each CPU-specific part of the kernel in order to control the TLBs and caches [39] :

[39] The following description is based on the documentation by David Miller [Mil] in the kernel sources.

❑ << Entire >>
flush_tlb_all and *flush_cache_all* flush the entire TLB/cache. This is only required when the page tables of the kernel (and not of a userspace process) are manipulated because a modification of this kind affects not only all processes but also all processors in the system.

❑ << Process-wide >>
*flush_tlb_mm(struct mm_struct *mm)* and *flush_cache_mm* flush all TLB/cache entries belonging to the address space *mm* .

❑ << Range within process >>
*flush_tlb_range(struct vm_area_struct *vma, unsigned long start, unsigned long end)* and *flush_cache_range(vma, start, end)* flush all entries from the TLB/cache between the start and end virtual addresses in the address range *vma->vm_mm* .

❑ << Page >>
*flush_tlb_page(struct vm_area_struct *vma, unsigned long page)* and *flush_cache_page(vma, page)* flush all entries from the TLB/cache whose virtual addresses are in an interval that begins at page and consists of PAGE_SIZE bytes.

<<
See http://stackoverflow.com/questions/3748384/what-is-tlb-shootdown

Invalidating the TLBs of other processor cores because of something that has occurred on one core is called a “TLB shootdown”. See :

```
# grep "TLB shootdowns" /proc/interrupts
TLB:      77644      78707      79922      65568   TLB shootdowns

<< switch to another process, or similar >>

# grep "TLB shootdowns" /proc/interrupts
TLB:      77646      78707      79923      65569   TLB shootdowns
#
```

>>

❑ << After Page Fault >>

*update_mmu_cache(struct vm_area_struct *vma, unsigned long address, pte_t pte)* is invoked after a page fault has been handled. It inserts information in the memory management unit of the processor so that the entry at the virtual address *address* is described by the page table entry *pte* .
This function is needed only if there is an external MMU. Typically, the MMU is integrated into the processor, but MIPS processors, for example, have external MMUs.

The kernel makes no distinction between data and instruction caches. If a distinction is required, the processor-specific code can reference the VM_EXEC flag in *vm_area_struct->flags* to ascertain whether the cache contains data or instructions.

The *flush_cache_* and *flush_tlb_* functions very often occur in pairs; for instance, when the address space of a process is duplicated using fork .

kernel/fork.c

```
flush_cache_mm(oldmm);
...
/* Manipulate page tables */
...
flush_tlb_mm(oldmm);
```

The sequence of operations — cache flushing, memory manipulation, and TLB flushing — is important for two reasons:

❑ If the sequence were reversed, another CPU in a multiprocessor system could take the wrong information from the process table after the TLBs have been flushed but before the correct information is supplied.

❑ Some architectures require the presence of ‘‘virtual-to-physical‘‘ transformation rules in the TLB when the cache is flushed (caches with this property are referred to as strict). *flush_tlb_mm* must execute after *flush_cache_mm* to guarantee that this is the case.

Some control functions apply specifically to data caches (*flush_dcache_ . . .*) or instruction caches (*flush_icache_ . . .*).

❑ *flush_dcache_page(struct page *page)* helps prevent alias problems that arise if a cache may contain several entries (with different virtual addresses) that point to the same page in memory. It is always invoked when the kernel writes to a page in the page cache or when it wants to read data from a page that is also mapped in userspace. This routine gives each architecture in which alias problems can occur an opportunity to prevent such problems.

❑ *flush_icache_range(unsigned long start, unsigned long end)* is invoked when the kernel writes data to kernel memory (between start and end) for subsequent execution. A standard example of this scenario is when a module is loaded into the kernel. The binary data are first copied to RAM and are then executed. *flush_icache_range* ensures that data and instruction caches do not interfere with each other if implemented separately.

❑ *flush_icache_user_range(*vma, *page, addr, len)* is a special function for the ptrace mechanism. It is needed to propagate changes to the address space of a traced process.

<<
Source: How does one write code that best utilizes the CPU cache to improve performance?

…
Improving spatial locality means that you ensure that each cache line is used in full once it has been mapped to a cache. When we have looked at various standard benchmarks, we have seen that a surprising large fraction of those fail to use 100% of the fetched cache lines before the cache lines are evicted.

Improving cache line utilization helps in three respects:

- It tends to fit more useful data in the cache, essentially increasing the effective cache size.
- It tends to fit more useful data in the same cache line, increasing the likelyhood that requested data can be found in the cache.
- It reduces the memory bandwidth requirements, as there will be fewer fetches.

Common techniques are:

- Use smaller data types
- Organize your data to avoid alignment holes (sorting your struct members by decreasing size is one way)

- Beware of the standard dynamic memory allocator, which may introduce holes and spread your data around in memory as it warms up.
- Make sure all adjacent data is actually used in the hot loops. Otherwise, consider breaking up data structures into hot and cold components, so that the hot loops use hot data.
- avoid algorithms and datastructures that exhibit irregular access patterns, and favor linear datastructures.
- We should also note that there are other ways to hide memory latency than using caches.

Modern CPU:s often have one or more *hardware prefetchers*. They train on the misses in a cache and try to spot regularities. For instance, after a few misses to subsequent cache lines, the hw prefetcher will start fetching cache lines into the cache, anticipating the application's needs.

…

To reduce the overall memory bus pressure, you have to start addressing what is called temporal locality. This means that you have to reuse data while it still hasn't been evicted from the cache.

Merging loops that touch the same data (*loop fusion*), and employing rewriting techniques known as tiling or blocking all strive to avoid those extra memory fetches.
…
>>

Paging

• An interesting animation of (the basics) of how paging hardware works can be viewed here!

• All virtual (or linear) addresses are grouped into fixed-size units called a ***page.*** It is the job of the hardware **paging unit** to map virtual pages to their physical memory equivalent; the paging unit resides on the CPU

• Pages in physical RAM are called ***page frames.*** The size of a page and page frame is the same - typically 4096 bytes on 32-bit architectures and 8192 bytes on 64-bit architectures

• Corresponding to contiguous virtual addresses in a page are contiguous physical addresses in the corresponding page frame- this is deliberately arranged so that when the kernel specifies a single physical address (for the page frame) and access rights of a page, it automatically holds for those of all the virtual addresses included within it (an offset is added)

• The kernel sets up & initializes data structures that specify the mapping between virtual pages and physical page frames- these are called the ***page tables***

On the IA32 processor, the 32 bits of a linear address are divided into 3 fields:

- Directory (high 10 bits) => max 2^{10} = 1024 addresses
- Table (middle 10 bits) and => max 2^{10} = 1024 addresses
- Offset (low 12 bits) => max 2^{12} = 4096 addresses

• Address translation is done in 2 steps, each using a separate translation table - the first is called the ***Page Directory*** and the second the ***Page Table***
[See sidebar "Hardware Paging on the x86 Architecture"]

• So, for a process: it's Page Directory can refer to upto 1024 Page Tables, each of which can refer to upto 1024 Page Table Entries - essentially, physical page frames; the Offset of course can span the entire page frame.
Thus the maximum number of locations a process can refer to is:

- 1024 x 1024 x 4096 = 2^{32} = 4 GB; as expected on a 32-bit system

<<
SIDEBAR :: Address Translation
...

Address translation is done using a table. Conceptually it's a table that maps every virtual address to its corresponding physical address. In practice, the address space is divided into larger chunks called pages and the *page table* maps virtual pages into physical pages. It would still be a

lot of data to map the whole virtual address space, so page tables are organized in a tree structure and only some branches of it are present in memory at any given time.

Let’s see how this works on a 32-bit x86. There is a top-level page directory table with entries that point to lower-level page tables. When it comes to translating a virtual address, the x86 takes the top 10 bits of the address and uses them as an offset into the directory table. How does it find the directory table? It stores the pointer to it in one of the special control registers, called CR3. An entry in the directory table contains a pointer to the corresponding page table. The x86 uses the next 10 bits of the virtual address to offset into that table. Finally, the entry in the page table contains the address of the physical page. The lowest 12 bits of the virtual address are then used as the offset into that page.

...

This kind of table walking is relatively expensive, and it involves multiple memory accesses, so a caching mechanism is used to speed it up. The x86 has a special Translation Lookaside Buffer, TLB, in which most recently used page translations are stored. If the upper 20 bits of a given address are present in the TLB, address translation is very fast. In the case of a TLB miss, the processor does the table walk and stores the new translation in the TLB. ...

>>

Page Tables have to be allocated on a **per process** basis - the motivation to use a 2-level scheme is to save RAM- memory need only be allocated for a Page Directory (4x1024=4Kb per process*) and in the Page Tables. Not only that, only those Page Tables that are refer to memory actually in use by a process need be allocated.

- * 1024 entries x 4 bytes (32-bits) for each = 4096 = 4KB

For example, if a process has allocated to it 60 pages of memory, then it's Page Table only requires to map those 60 Page Table Entries - the rest are just nullified. Thus only 1024x4 = 4Kb for the Page Directory plus 60x4 = 240 bytes for the Page Table would be required in kernel memory.

Hardware Paging on the x86 Architecture I

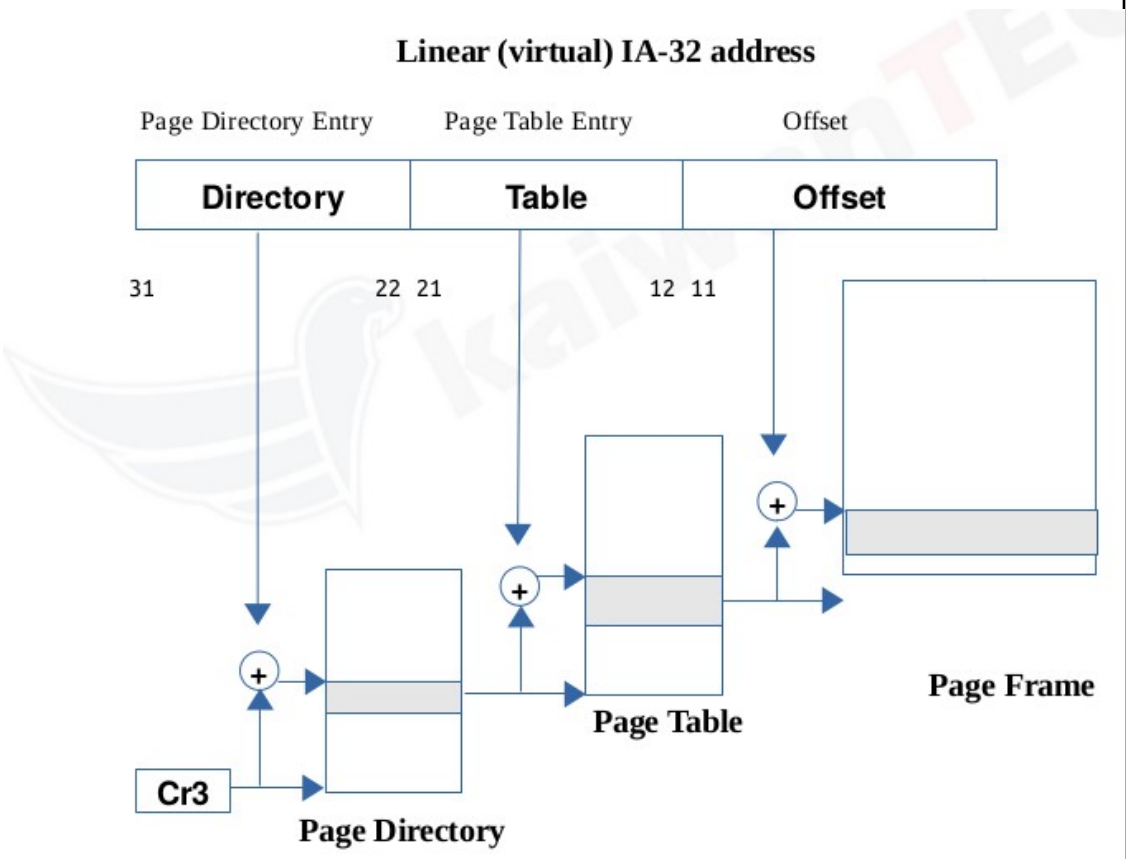


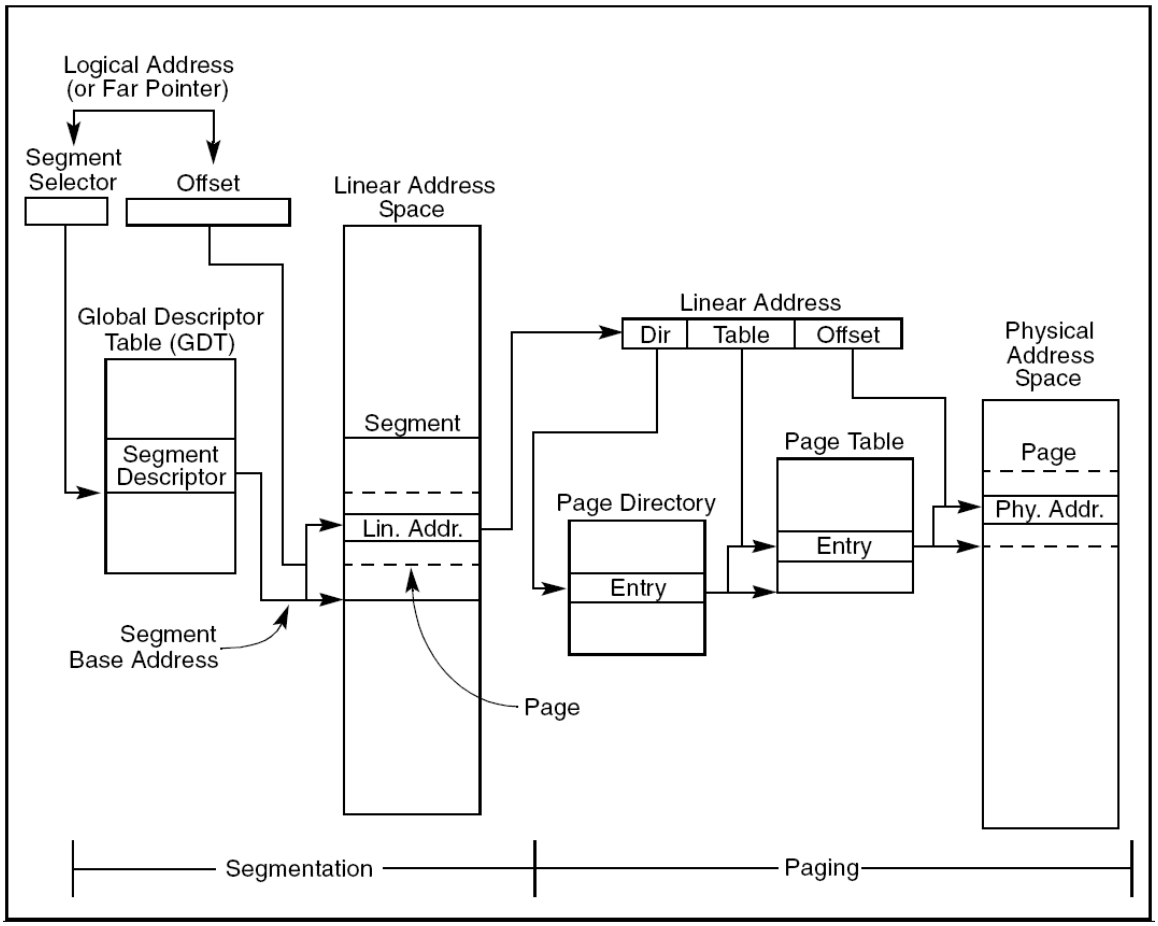
Fig 1: Paging on x86

As seen from the figure above, the Page Directory field in the IA-32 virtual address determines the entry in the Page Directory (by adding the PDE offset to the PGD base address); this points us to the correct Page Table.

The entry in the Page Table (PTE) field of the IA-32 virtual address then determines the entry in the Page Table now referenced (by adding the PTE offset to the Page Table base address). This holds the physical address of the corresponding Page Frame.

Finally, the Offset field of the virtual address determines the actual offset within the physical page frame of the byte being referenced.

Hardware Paging on the x86 Architecture II



[Figure above from Intel's "IA-32 Intel Architecture Software Developers Manual" Vol 3 page 68]

Tip
An interesting animation of the above "segmentation-paged" hardware paging technique can be viewed here! In general, many links to interesting OS animations can be found on William Stallings' page here.

The **CR3** (Control Register 3) on the IA-32 and Intel 64 holds hardware context on the base location of the processes' (current's) paging table (PGD). Hence, during a context-switch, this register must be updated to the "next" process that will subsequently execute on the cpu. Indeed, the context-switch code consists primarily of calling two function's – switch_mm() and switch_to().

switch_mm() updates the CR3 register.
In *arch/x86/include/asm/mmu_context.h :*

```
...
  33static inline void switch_mm(struct mm_struct *prev, struct mm_struct
*next,
  34                             struct task_struct *tsk)
  35{
  36        unsigned cpu = smp_processor_id();
  37
  38        if (likely(prev != next)) {
  39                /* stop flush ipis for the previous mm */
  40                cpumask_clear_cpu(cpu, mm_cpumask(prev));
  41#ifdef CONFIG_SMP
  42                percpu_write(cpu_tlbstate.state, TLBSTATE_OK);
  43                percpu_write(cpu_tlbstate.active_mm, next);
  44#endif
  45                cpumask_set_cpu(cpu, mm_cpumask(next));
  46
  47                /* Re-load page tables */
  48                load_cr3(next->pgd);
  49
...
```

Note that when the CR3 register of the CPU is modified (=> the kernel context-switches to another task), the hardware automatically invalidates all entries of the local Translation Lookaside Buffer (TLB). This will ensure a TLB-flush operation.

<<
NOTE-

1. Setting one control register is not an expensive operation, but on an x86 it must be accompanied by a TLB flush, which results in new table walks to fill it again; and that might be quite expensive. This is why both Intel and AMD have recently added Address Space Identifiers (ASIDs) to TLB entries in their newest processors. TLBs with ASIDs don't have to be flushed, the processor will just ignore the entries with the wrong ASIDs.

2. We should realize by now that *only* virtual addresses can be used; this is as *all address lookups flow across the MMU (and thus get translated to physical addresses)!*

3. The stored CR3 value in the *mm_struct* is a kernel virtual address; BUT here we have a special case: we require to store it into the hardware CR3 register as a physical address (else we'll land up with infinite recursion when attempting to translate a va to a pa).

Thus, the CR3 virtual address is actually first converted to it's physical address via kernel software and then placed into CR3 register. The *load_cr3()* function translates CR3 kernel va to a pa and then writes it!
arch/x86/include/asm/processor.h

```
static inline void load_cr3(pgd_t *pgdir)
{
   write_cr3(__pa(pgdir));
}
```

>>

ARM specific

- See the equivalent discussion for setting next->pgd for the ARM architecture here: ARM Linux: Switching page tables during context switch
- ARM MMU aborts : what exceptions does the ARM MMU raise?

SIDEBAR :: Intel 32 and 64 Control Registers ***[FYI/OPTIONAL]***
Source: "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide" *page 2-18 Vol 3A*
The control registers are summarized below, and each architecturally defined control field in these control registers are described individually. In Figure 2-6, the width of the register in 64-bit mode is indicated in parenthesis (except for CR0). • CR0 — Contains system control flags that control operating mode and states of the processor. • CR1 — Reserved. • CR3 — Contains the physical address of the base of the page directory and two flags (PCD and PWT). This register is also known as the page-directory base register (PDBR). Only the most-significant bits (less the lower 12 bits) of the base address are specified; the lower 12 bits of the address are assumed to be 0. The page directory must thus be aligned to a page (4-KByte) boundary. The PCD and PWT flags control caching of the page directory in the processor's internal data caches (they do not control TLB caching of page-directory information). CR2 — Contains the page-fault linear address (the linear address that caused a page fault). When using the physical address extension (PAE), the CR3 register contains the base address of the page-directory-pointer table In IA-32e mode, the CR3 register contains the base address of the PML4 table. *See also: Section 3.8, "36-Bit Physical Addressing Using the PAE Paging Mechanism."* • CR4 — Contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities. The control registers can be read and loaded (or modified) using the move-to-or-from-control-registers forms of the MOV instruction. In protected mode, the MOV instructions allow the control registers to be read or loaded (at privilege level 0 only). This restriction means that application programs or operating-

system procedures (running at privilege levels 1, 2, or 3) are prevented from reading or loading the control registers.

• CR8 — Provides read and write access to the Task Priority Register (TPR). It specifies the priority threshold value that operating systems use to control the priority class of external interrupts allowed to interrupt the processor. This register is available only in 64-bit mode. However, interrupt filtering continues to apply in compatibility mode.

See figure on following page.

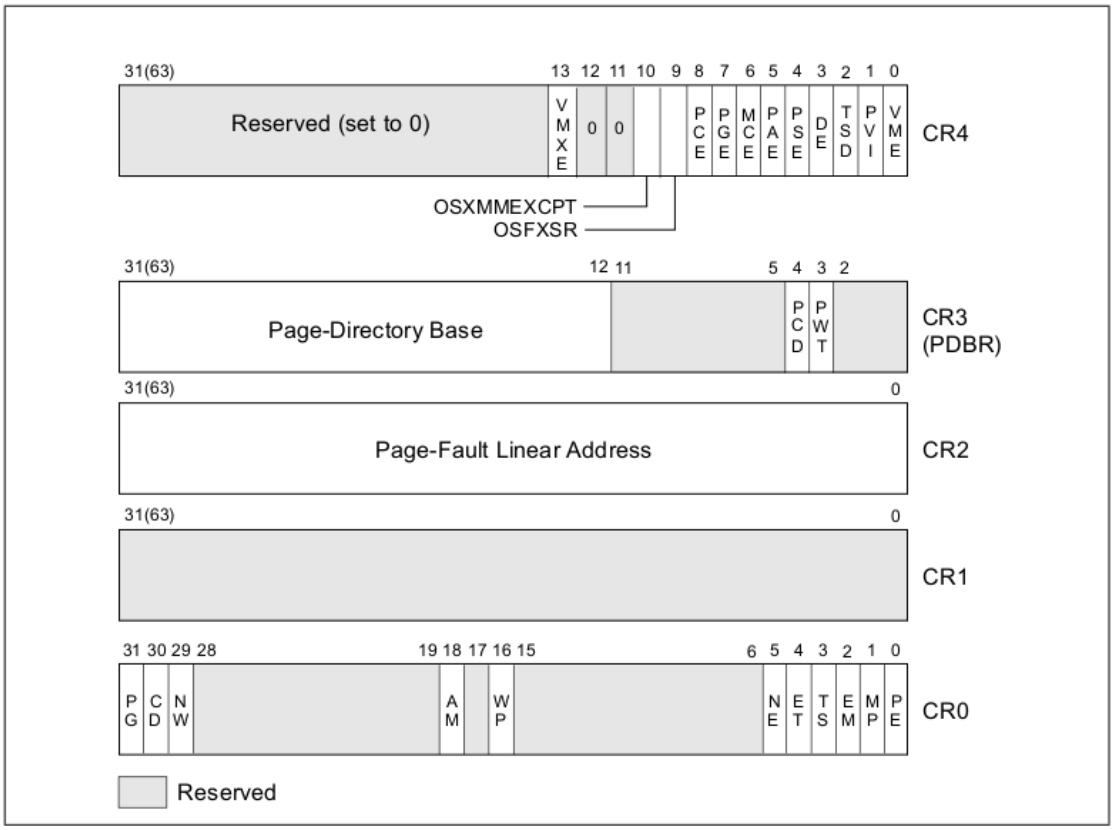


Figure 2-6. Control Registers

FYI

- Excellent article on Page Table Layout (on x86_64) [link]
- see a simple assembly language example of setting up paging on an IA-32 class processor on the OSDev Wiki site.

Source: "Understanding the Linux Virtual Memory Manager" << ULVMM >> by Mel Gorman.

Linux VM Architecture

Linux layers the machine independent/dependent layer in an unusual manner in comparison to other operating systems [CP99]. Other operating systems have objects that manage the underlying physical pages, such as the pmap object in BSD.

Linux instead maintains the concept of a ~~three~~-four-level* page table in the architecture-independent code even if the underlying architecture does not support it. Although this is conceptually easy to understand, it also means that the distinction between different types of pages is very blurry, and page types are identified by their flags or what lists they exist on rather than the objects they belong to.

<<
* From 2.6.11, it's become *four-level* paging tables; this was done to fully support the AMD x86_64 platform.
>>

Architectures that manage their Memory Management Unit (MMU) differently are expected to emulate the four-level page tables. For example, on the x86 without PAE enabled, only two page table levels are available. The Page Middle Directory (PMD) is defined to be of size 1 and "folds back" directly onto the Page Global Directory (PGD), which is optimized out at compile time.

<<
Comment in *arch/x86/include/asm/pgtable_32.h*

Whoops! Outdated: 4-level now

```
...
/*
 * The Linux memory management assumes a three-level page table setup. On
 * the i386, we use that, but "fold" the mid level into the top-level page
 * table, so that we physically have the same two-level page table as the
 * i386 mmu expects.
 *
 * This file contains the functions and defines necessary to modify and use
 * the i386 page table tree.
 */
```
>>

<< Similarly, for the ARM platform, see Source >>

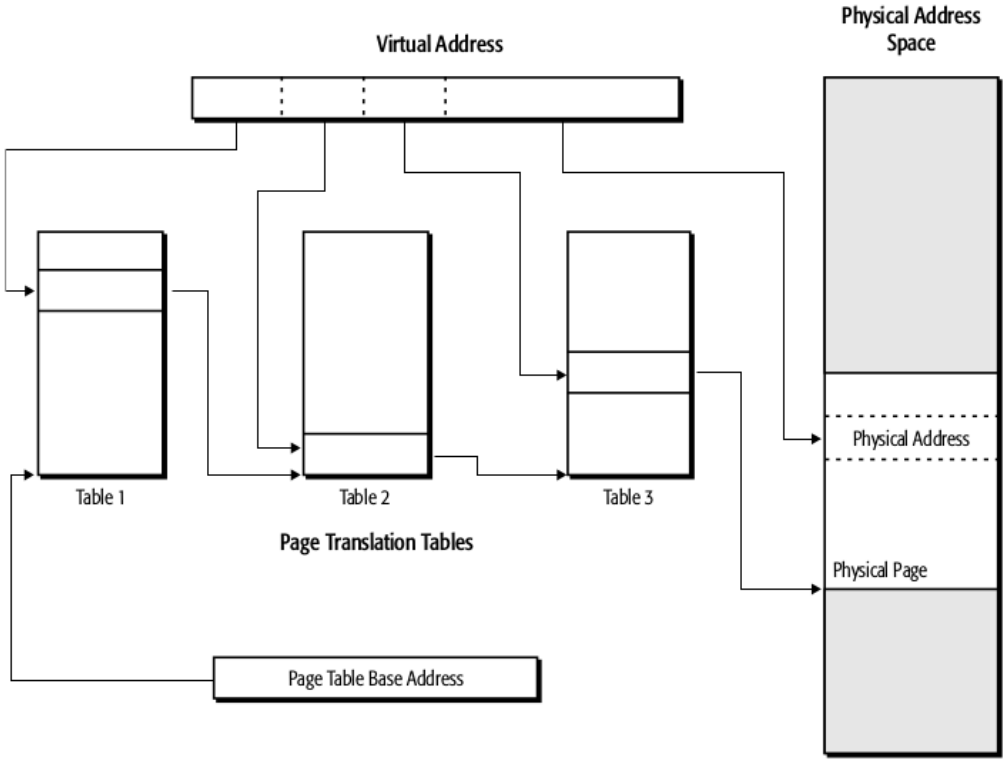
<<
Professional Linux Kernel Architecture by Wolfgang Mauerer, Wrox Press.
…

Architectures with two-level page tables define PTRS_PER_PMD and PTRS_PER_PUD as 1. This persuades the remaining parts of the kernel that they are working with four-level page translation although only two pages are used — the page middle and page upper directories are effectively eliminated because they have only a single entry.

Because only a very few systems use a four-level page table, the kernel uses the header file *include/asm-generic/pgtable-nopud.h* to hold all the declarations needed to simulate the presence of a fourth page table.

The header file *include/asm-generic/pgtable-nopmd.h* is also available to simulate the presence of a third page table level on systems with two-level address translation.
...
>>

3-Level Paging



Paged Memory Model

Above diagram *Source: AMD64 Architecture Programmer's Manual, Vol 2: Systems Programming*

Unfortunately, for architectures that do not manage their cache or Translation Lookaside Buffer (TLB) automatically, hooks that are architecture dependent have to be explicitly left in the code for when the TLB and CPU caches need to be altered and flushed, even if they are null

operations on some architectures like the x86.

Additional Notes

• The Linux OS uses a **4-level** * paging model (even on the 2-level x86 processors) so that paging works on 64-bit architectures as well as 32-bit, all with the same codebase. (When the kernel is being built for 32-bit, the PUD and PMD fields are "folded" to 1 bit, effectively eliminating them and using 2-level paging).

Four-level page tables merged, LWN, 05 Jan 2005, corbet

• The virtual address in the 4-level* model is interpreted by the MMU as a bitmask:

PGD	PUD	PMD	PT	offset

PGD: Page Global Directory
PUD: Page Upper Directory
PMD: Page Middle Directory
PT : Page Table

***** Upto kernel 2.6.10 three level paging was used; from 2.6.11 a *four-level* paging model has been adopted. There is now a Page Upper Directory (PUD) field introduced (between the PGD and PMD). This has been done to fully support the linear address splitting used by the x86_64 platform (see the table below).

4-Level Paging

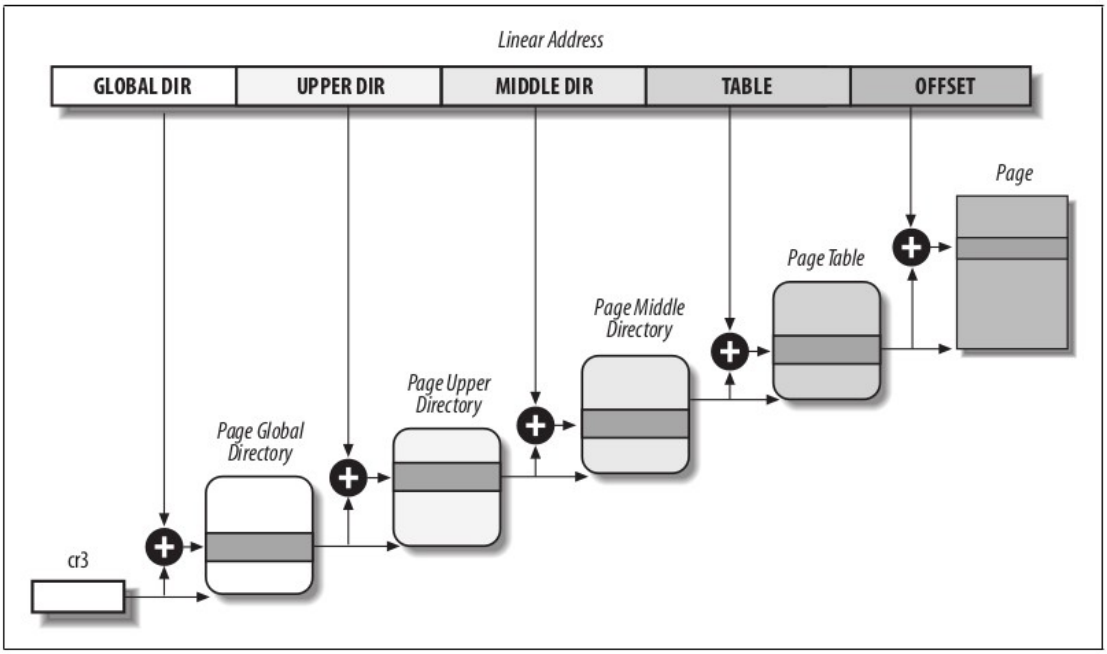


Figure 2-12. The Linux paging model

Diagram Source: *Understanding the Linux Kernel, 3rd Ed, Bovet & Cesati, O'Reilly*

With standard 4 KB Page size

Arch	N-Level	Addr Bits	VM "Split"	Userspace		Kernel-space	
				Start vaddr	End vaddr	Start vaddr	End vaddr
IA-32	2	32	3 GB : 1 GB	0x0	0xbfff ffff	0xc000 0000	0xffff ffff
ARM	2	32	2 GB : 2 GB	0x0	0x7fff ffff	0x8000 0000	0xffff ffff
x86_64	**4**	**48**	**128 TB : 128 TB**	**0x0**	**0x0000 7fff ffff ffff**	**0xffff 8000 0000 0000**	**0xffff ffff ffff ffff**
	5*	56	64 **PB** : 64 **PB**	0x0	0x00ff ffff ffff ffff	0xff00 0000 0000 0000	0xffff ffff ffff ffff
Aarch64	3	39	512 GB : 512 GB	0x0	0x0000 007f ffff ffff	0xffff ff800 0000 000	0xffff ffff ffff ffff
	4	48	256 TB : 256 TB	0x0	0x0000 ffff ffff ffff	0xffff 0000 0000 0000	0xffff ffff ffff ffff

* *>= 4.14 Linux*

IMP Update! Please see the newly updated (as of Linux 4.20) x86_64 process memory layout doc here: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

The number of bits per field varies with the hardware architecture

Platform	Page Size	Number of address bits used	Number of paging levels	Linear address splitting
alpha	8 KB [a]	43	3	10 + 10 + 10 + 13
IA-32	4 KB [a]	32	2	10 + 10 + 12
IA-64	4 KB [a]	51 [b] / 39	4 / 3	9 + 9 + 9 + 12
ppc64	4 KB	41	3	10 + 10 + 9 + 12
sh64	4 KB	41	3	10 + 10 + 9 + 12
x86_64 [e]	4 KB	48	4	9 + 9 + 9 + 9 + 12
x86_64 [f]	4 KB	57	5	9 + 9 + 9 + 9 + 9 + 12
ARM [c]	4 KB	32	2	12 + 8 + 12 [d]

[a] This architecture supports different page sizes; we select a typical page size adopted by Linux.
[b] Update: Newer model Itanium
[c] This entry added on here (not in the original ULK book).
[d] Processor (machine-type)-specific
[e] Naming*
[f] *[New]* 5-level paging support from 4.14 Linux (Nov 2017) onwards; not fully implemented in h/w yet [*see this LWN article*]

* [*Source*] … Various names are used for the instruction set; prior to the launch, x86-64 and x86_64 were used, while upon the release AMD named it AMD64.[3] Intel initially used the names IA-32e and EM64T before finally settling on "Intel 64" for its implementation. Some in the industry, including Apple,[4][5][6] use x86-64 and x86_64, while others, notably Sun Microsystems[7] (now Oracle Corporation) and Microsoft,[8] use x64 while the BSD family of OSs and several Linux distributions[9][10] use AMD64.

• On 32-bit systems, Linux effectively eliminates the PUD and PMD by assigning them to zero bits. However, the position of the PUD/PMD in the sequence of pointers is kept intact - now the **same** kernel codebase can be used on both 32 and 64-bit architectures.

• Every process on Linux has it's own PGD and set of Page Tables. Recall that the **cr3** register holds the base address of the PGD.

• As and when a context switch occurs, Linux saves the cr3 value in the memory descriptor of the process just "leaving" the CPU and loads cr3 with the value stored in the process descriptor for the new process scheduled onto the CPU. So when this process starts referencing it's virtual addresses, automatically the correct PGD and therefore Paging Tables are referred to and address translation occurs (see Note on CR3 update with the switch_mm() inline function above)

• Excellent article on Page Table Layout (on x86_64) [link]

From the Intel Vol-3A Manual:

"…

Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.

In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The FS and GS segments are exceptions. These segment registers (which hold the segment base) can be used as an additional base registers in linear address calculations. They facilitate addressing local data and certain operating system data structures.
Note that the processor does not perform segment limit checks at runtime in 64-bit mode.
…"

[FYI / OPTIONAL]

x86: The Page Directory (PGD) and Page Table (PTE) data structures

Some important members of the Page Directory and Page Table Entry structures are mentioned below:

- **Present flag (resident bit)**
  =1 : The Page Table or page frame that is being referred to is in main memory
  =0 : The Page Table or page frame that is being referred to is not in main memory

(is probably in swap); in this case, the virtual address is stored in register cr2 and Exception 14 - the **Page Fault exception** - is triggered by the paging unit; the page-fault handler that runs ultimately fetches the missing page, stores it in main memory, updates the Page tables accordingly, and hands back control to the paging unit.

- The 20 MSB bits of the physical address
  If this is in the Page Directory entry, it refers to the address of the proper Page Table for the process. In this case, the low-order 12 bits are always 0 anyway.
  If this is in the Page Table, it refers to the physical address of the actual page frame.

  The remaining 12 bits is in the offset part in this latter case.

  20 MSB bits => 2^{20} possible unique addresses
  $2^{20}=2^{10}$ x 2^{10}=1024 x 1024 = 1048576 = 1MB; => 1MB possible Page Tables, each of which can themselves refer 1024 possible page frames (because of the 12-bit offset); so, total addressable memory = 1048576 x 1024 = 2^{20} x 2^{10} = 2^{32} = 4 GB!

- **Dirty flag**
  Applies only to Page Table entries.
  =1 : the corresponding page frame has been written to
  =0 : implies a "clean" page
  (again, used by the virtual page allocation handlers & swapper)

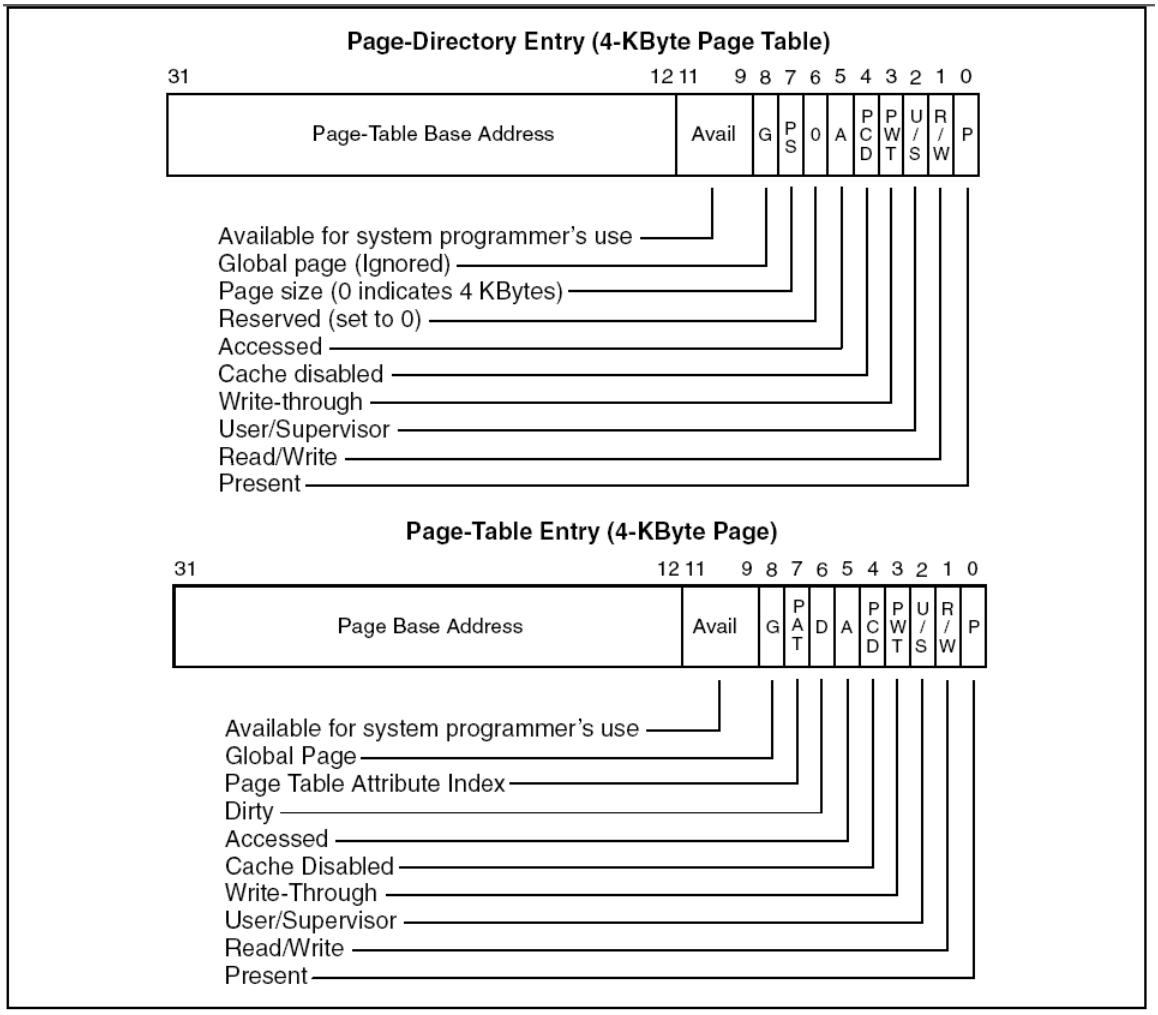


Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

[Figure above from Intel's "IA-32 Intel Architecture Software Developers Manual" Vol 3 page 90]

[Update (more recent ver, Feb 2014): from Intel's "Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Volume 3A" page 4-13 (diagrams) and pages 4-15 and 4-16 : Table 4-5: Format of a 32-Bit Page-Directory Entry that References a Page Table]

…

Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are "not present"; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

--snip--

<<
The second-to-last "PTE: 4KB page" entry in the above table is the one of interest, with details shown on the next page.
>>

[PTO →]

<< ***Level 2 PTE entry:*** *page 4-16 Vol. 3A* >>

Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

…
All the IA-32 processors that have on-chip caches also provide the PCD (page-level cache disable) flag in page table and page directory entries. This flag allows caching to be disabled on a page-by-page basis.

Page-level cache disable (PCD) flag, bit 4
Controls the caching of individual pages or page tables. When the PCD flag is set, caching of the associated page or page table is prevented; when the flag is clear, the page or page table can be cached. This flag permits caching to be disabled for pages that contain memory-mapped I/O ports or that do not provide a performance benefit when
cached. The processor ignores this flag (assumes it is set) if the CD (cache disable) flag in CR0 is set.

Page-level write-through (PWT) flag, bit 3
Controls the write-through or write-back caching policy of individual pages or page tables. When the PWT flag is set, write-through caching is enabled for the associated page or page table; when the flag is clear, write-back caching is enabled for the associated page or page table. The processor ignores this flag if the CD (cache disable) flag in CR0 is set.

On the Linux OS, by default, the PCD, PWT and CD bits are cleared, maximizing performance.

Hardware Paging on the Intel 64 Architecture

Source: "Intel® 64 and IA-32 Architectures - Software Developer's Manual Volume 3A: System Programming Guide, Part 1"

3.10.1 IA-32e Mode Linear Address Translation (4-KByte Pages)

Figure 3-24 shows the PML4, page-directory-pointer, page-directory, and page-table hierarchy when mapping linear addresses to 4-KByte pages in IA-32e mode. This paging method can be used to address up to 2^{36} pages, which spans a linear address space of 2^{48} bytes.

To select the various table entries, linear addresses are divided into five sections:

- **PML4-table entry** — Bits 47:39 provide an offset to an entry in the PML4 table. The selected entry provides the base physical address of a page directory pointer table.
- **Page-directory-pointer-table entry** — Bits 38:30 provide an offset to an entry in the page-directory-pointer table. The selected entry provides the base physical address of a page directory table.
- **Page-directory entry** — Bits 29:21 provide an offset to an entry in the selected page directory. The selected entry provides the base physical address of a page table.
- **Page-table entry** — Bits 20:12 provide an offset to an entry in the selected page table. This entry provides the base physical address of a page in physical memory.
- **Page offset** — Bits 11:0 provide an offset to a physical address in the page.

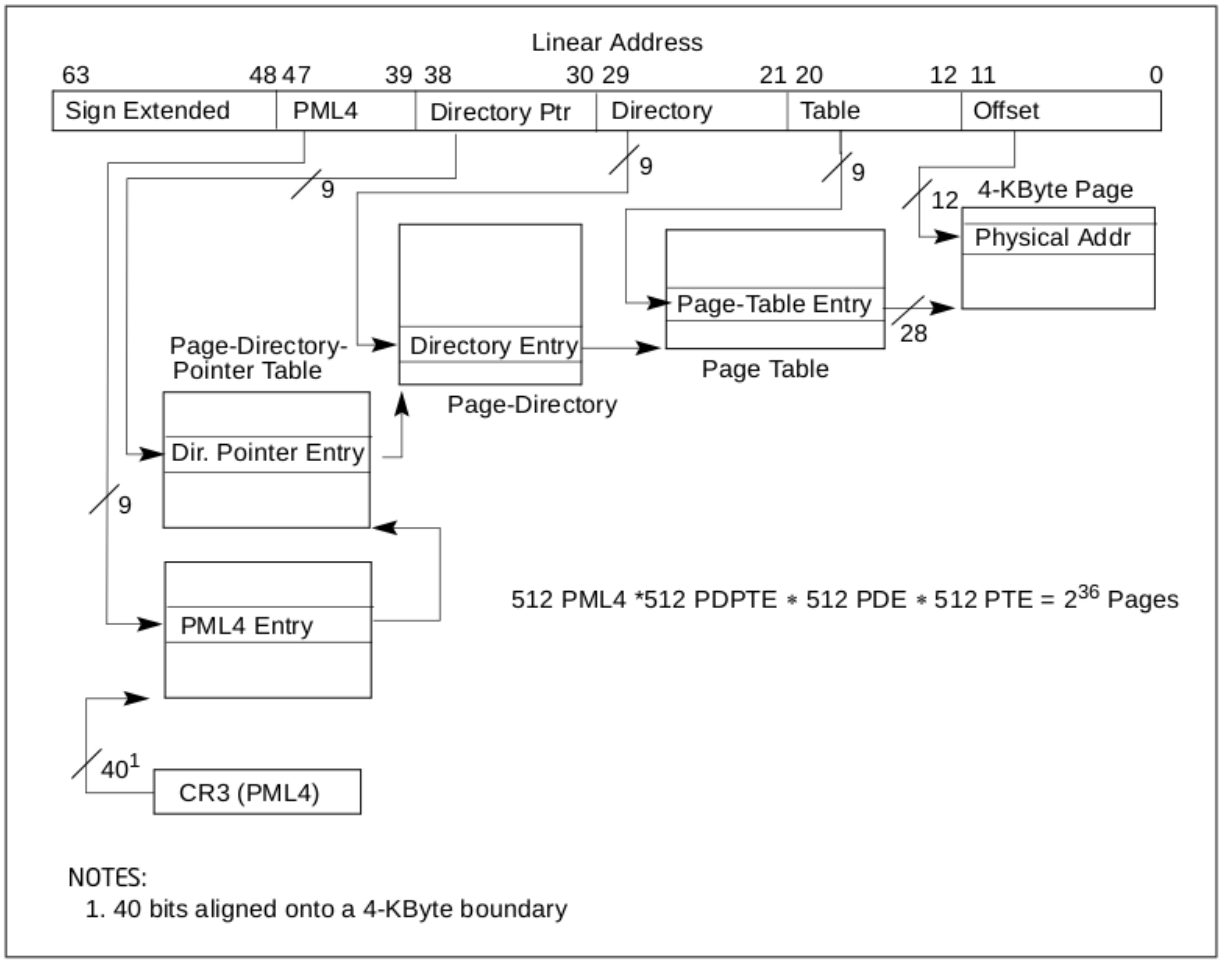


Figure 3-24. IA-32e Mode Paging Structures (4-KByte Pages)

48-bit 4-level paging on the Intel 64.

The Page Table hierarchy consists of 4 levels:

PML4 : Page Map Level 4 offset (level 0) [contents of CR3, physical address]
PDPE : Page Directory Pointer offset (level 1)
PDP : Page Directory offset (level 2)
PTE : Page Table offset (level 3)

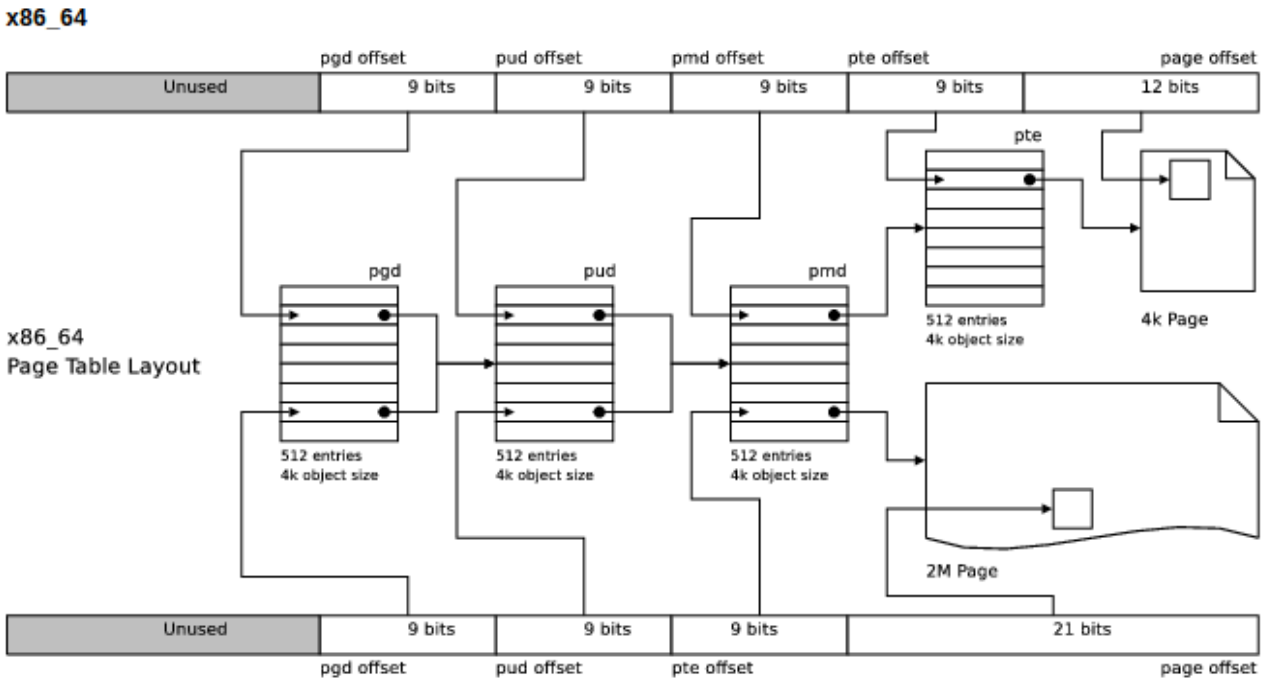
<<
Source : The original implementation of the AMD64 architecture implemented 40-bit physical addresses and so could address up to 1 TB (2^{40} bytes) of RAM.[1](p24)

Current implementations of the AMD64 architecture (starting from AMD 10h microarchitecture) extend this to 48-bit physical addresses[14] and therefore can address up to 256 TB of RAM. The architecture permits extending this to 52 bits in the future[1](p24)[15] (limited by the page table entry format);[1](p131) this would allow addressing of up to 4 PB of RAM.

For comparison, 32-bit x86 processors are limited to 64 GB of RAM in Physical Address Extension (PAE) mode,[16] or 4 GB of RAM without PAE mode.[1](p4)
>>

SIDEBAR :: Above: Why's it called IA-32e Mode and not Intel x86_64 or Intel 64 Mode?
* [*Source*] ... Various names are used for the instruction set; prior to the launch, x86-64 and x86_64 were used, while upon the release AMD named it AMD64.[3] Intel initially used the names IA-32e and EM64T before finally settling on "Intel 64" for its implementation. Some in the industry, including Apple,[4][5][6] use x86-64 and x86_64, while others, notably Sun Microsystems[7] (now Oracle Corporation) and Microsoft,[8] use x64 while the BSD family of OSs and several Linux distributions[9][10] use AMD64.

[Source] Below: Another good diagram of 3 / 4-level paging on the Intel 64, with both 4Kb and 2MB page sizes:



Notice how using a 2 MB page size reduces the number of paging levels (and thus RAM lookups) to 3!

<<
How does the MMU detect that 4K or 2M pages are in use? Intel’s PSE (Page Size Extension) bit set => 2M page, else 4K page.

… *huge pages** can be represented by special entries at the higher levels. For example, a 2MB chunk of memory could be represented by either a single huge-page entry at the PMD level or a full page of single-page PTE entries. …

* Called *Super Pages* on BSD, *Large Pages* on Windows
>>

Additional Resource:

How is a page walk implemented?
Answered wrt x86_64 architecture.

SIDEBAR

4.14 kernel *<< released on 12 Nov 2017 >>*

Bigger memory limits

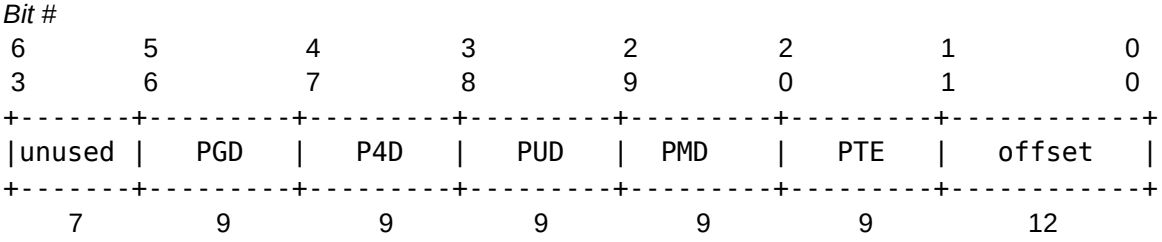
Original x86-64 was limited by 4-level paging to 256 TiB of virtual address space and 64 TiB of physical address space. People are already bumping into this limit: some vendors offers servers with 64 TiB of memory today. To overcome the limitation upcoming hardware will introduce support for 5-level paging. It is a straight-forward extension of the current page table structures adding one more layer of translation. It bumps the limits to 128 PiB of virtual address space and 4 PiB of physical address space. This "ought to be enough for anybody" ©.

On x86, 5-level paging enables 56-bit userspace virtual address space. Not all user space is ready to handle wide addresses. It's known that at least some JIT compilers use higher bits in pointers. It collides with valid pointers with 5-level paging and leads to crashes. To mitigate this, the Linux kernel will not allocate virtual address space above 47-bit by default. Userspace can ask for allocation from full address space by specifying hint address above 47-bits.

Recommended LWN article: Five-level page tables

Code: commit, commit, commit, commit, merge

<<
64-bit virtual-address with 5-level paging enabled:



57 bits (0-56) used for addressing. 2^57 = 128 PiB.

>>

Update: kernel ver 4.6 released on 15 May 2016:
Source
…
1.3. Support for Intel memory protection keys

This release adds support for a memory protection hardware feature that is available in upcoming Intel CPUs: protection keys. Protection keys allow the encoding of user-controllable permission masks in the page table entries (pte). Instead of having a fixed protection mask in the pte (which needs a system call to change and works on a per page basis), the user can map a handful of protection mask variants. User space can then manipulate a new user-accessible, thread-local register, (PKRU) with two separate bits (Access Disable and Write Disable) for each mask. This makes possible to dynamically switch the protection bits of very large amounts of virtual memory by just manipulating a CPU register, without having to change every single page in the affected virtual memory range.

It also allows more precise control of MMU permission bits: for example the executable bit is separate from the read bit. This release adds the infrastructure for that, plus it adds a high level API to make use of protection keys. If a user-space application calls: *mmap(..., PROT_EXEC)* or *mprotect(ptr, sz, PROT_EXEC)* (note PROT_EXEC-only, without PROT_READ/WRITE), the kernel will notice this special case, and will set a special protection key on this memory range. It also sets the appropriate bits in the PKRU register so that the memory becomes unreadable and unwritable. So using protection keys the kernel is able to implement 'true'PROT_EXEC: code that can be executed, but not read, which is a small security advantage (but note that malicious code can manipulate the PKRU register too). In the future, there will further work around protection keys that will offer more high level call APIs to manage protection keys.

Recommended LWN article: Memory protection keys
...

---

* *More details on the ARMv7 Paging architecture below:* **[OPTIONAL]**

ARMv7 MMU Architecture

Source : Memory Management: Paging
Paul Krzyzanowski

...

ARMv7-A MMU Architecture

ARM is a 32-bit reduced instruction set computer (RISC) architecture developed and owned by ARM Holdings. The processors are licensed to manufacturers who may augment them with custom DSPs (digital signal processors), radios, or graphics processors. The processors are used in most cell phones and gaming consoles. The ARMv7-A architecture that we'll be examining is present in the Cortex-A8 and Cortex-A9 processors. The Cortex-A8 is used in Motorola Droid phones, the iPad, and the iPhone (3GS and 4), among numerous other devices. The Cortex-A9 is used in Apple's A5 processor that powers the iPad 2. Recent previous generations of ARM processors are not significantly different in terms of their MMU architecture so much of this discussion can apply to them as well. References to the ARM processor, MMU, or ARM architecture in the rest of this section will refer to the ARMv7-A architecture.

Sections and pages

The ARM MMU supports four page sizes. The largest sizes are called sections and the smaller sizes are called pages:

- Supersections: 16 MB memory blocks (24-bit offsets)
- Sections: 1 MB memory blocks (20-bit offsets)
- Large pages: 64 KB pages (16-bit offsets)
- Small pages: 4 KB pages (12-bit offsets)

<<

An aside:
As seen earlier, the Linux kernel “direct-maps” platform RAM to the kernel virtual address space starting at PAGE_OFFSET (often 2 GB on ARM platforms).

Furthermore, for performance, *ARM Linux will map it as 1 MB sections*.
(The x86 arch does a similar thing: it maps kernel pages as 2 MB “huge pages”).

>>

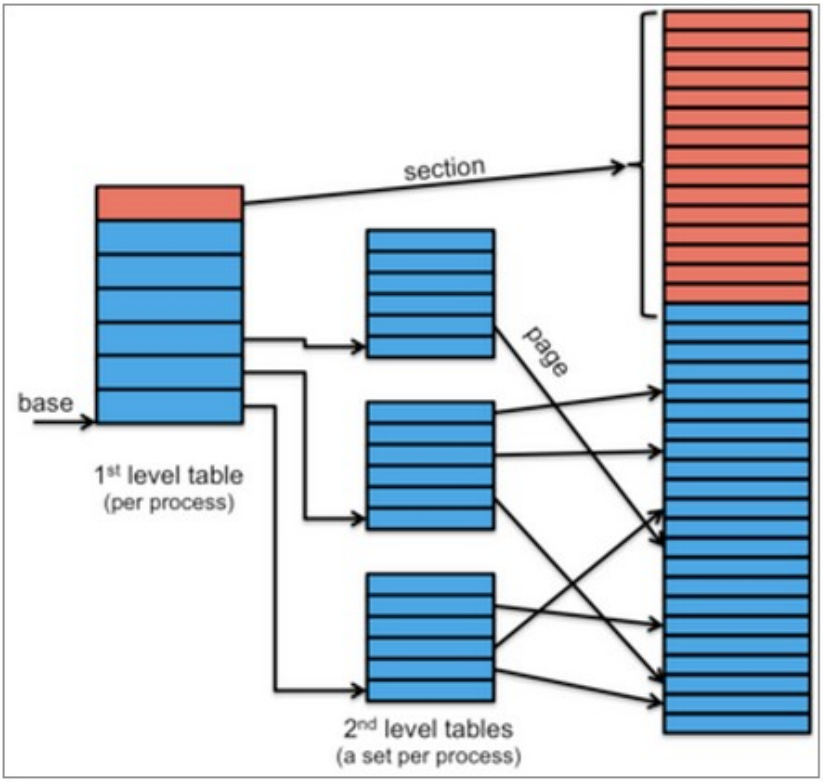


Figure 1. ARM sections and pages

The MMU supports a two-level hierarchy for its page table structure. An entry in the first-level table contains either a pointer to a second-level tables (partial page tables) or a base address of a section or a supersection. Hence, if we use the really big pages — sections and supersections — then we don't have to go through two levels of hierarchy. The benefit of sections and supersections is that you can have a large region of memory, such as the operating system, mapped using just a single entry in the TLB. The MMU can be configured to use either small or large pages as well as sections or supersections. Sections (or supersections) can be mixed together with pages. Just because the architecture supports mapping blocks of several different sizes does not mean that the operating system will use the capability. Doing so introduces the problems associated with variable size partitions that we discussed earlier. However, even if this is not used for general-purpose memory management, it makes a lot of sense for mapping the operating system address space efficiently [1].

Translation Lookaside Buffers (TLB)

The ARM has two levels of TLBs. The smallest and fastest is the MicroTLB. There is a MicroTLB for the instruction and data sides of the CPU (instruction fetches uses one MicroTLB while data read/write operations use the other). The MicroTLB can store 32 entries [2]. The cache is fully associative and can perform a lookup in one clock cycle. Hence, there is no performance penalty for any memory references that are satisfied by the MicroTLB.

The architecture supports the use of an address space identifier (ASID) to allow the operating system to identify one process' address space from another's without flushing the cache. Entries can also be tagged as *global*; so that they are shared among all address spaces. This is, of course, useful for mapping the memory regions used by the operating system.

Each entry contains a number of protection bits and these are checked at each address lookup. If the protections disallow the requested memory operation (e.g., no-execute or read-only) then the MMU will signal a Data Abort, which will cause a trap. On a cache miss, the replacement algorithm may be selected to be either round-robin (the default) or a random replacement.

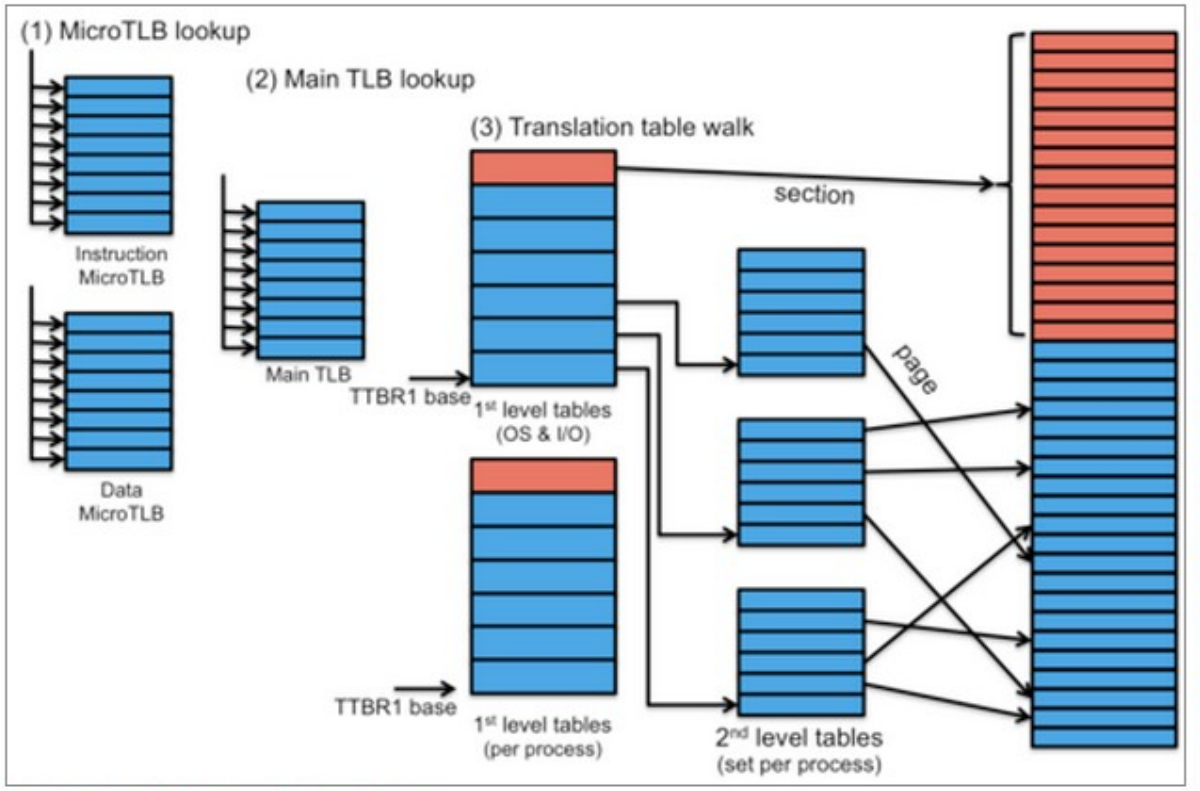


Figure 2. ARM memory translation

<<
On Linux running on ARM, the hardware translation table base registers – TTBRn - are setup such that:

TTBR1 : points to the base address of the kernel page tables
TTBR0 : points to the base address of the usermode process's page tables

Note that this is *not* an architectural requirement by ARM; it's a software feature from the Linux kernel.

[Also Note- the Figure 2 above seems to have a typo: in step (3) the bottom paging table should be marked as "TTBR0 base" (and not 'TTBR1 Base') as it inadvertently seems to have been marked.]

From kernel documentation:
"User addresses have bits 63:48 set to 0 while the kernel addresses have the same bits set to

1. TTBRx selection is given by bit 63 of the virtual address. “
>>

The second-level TLB is called the Main TLB. It catches any cache misses from the microTLBs. There is only one of these per processor, so it handles misses from both the data-side and instruction-side MicroTLBs. The cache comprises eight fully associative entries, which are fast and may also have their contents locked (i.e., they will not be replaced). It also contains 64 low associative entries. These are still much faster than a main memory access but may require a variable amount of clock cycles for lookup. The ability lock eight entries down is crucial for real-time systems. If you need to guarantee rapid real-time response to an event, you may not be able to afford to waste the time to access an in-memory page table to do an address translation. Note that the operating system, in addition to locking the TLB entries, has to ensure that the corresponding page frames stay locked in memory as well.

Figure 2 illustrates the full set elements that come into play for memory translation:

1. The first step is a MicroTLB lookup. An instruction fetch accesses the Instruction MicroTLB and a data read/write operation accesses the Data MicroTLB. If this lookup yields a match then the access permission in the page table entry are validated. If the request does not have proper permissions then a trap is generated (Data Abort signal). Otherwise, the memory access takes place. If the lookup yields no match then we continue to step 2.

2. If the MicroTLB lookup yielded a cache miss then we consult the Main TLB. The process is the same as with the MicroTLB. If there is a match for the requested page and the permissions pass then the memory access is performed. If the lookup yields no match then we continue to step ~~2~~ 3.

<<
A question:
Based on the virtual address arriving at it, the MMU first determines *which* paging table to use (for the OS or for the usermode process), depending on whether it receives a kernel va (virtual address) or usermode va.
How exactly does the MMU differentiate between a kernel va / usermode va?
See the discussion below:
>>

3. The final step is to consult the page table(s) in memory. The first-level table is checked first. The system supports two first-level tables. High-order bits of the virtual address determine which one to use. The base of the table is stored in one of two base registers (TTBR0 or TTBR1), depending on whether the topmost *n* bits of the virtual address are 0 (use TTBR0) or not (use TTBR1). The value for *n* is defined by the Translation Table Base Control Register (TTBCR).

<<
For Linux on ARM (32-bit) with a 3:1 virtual address space split (CONFIG_VM_SPLIT), the PAGE_OFFSET value is 0xc0000000. 0xc = 0b1100 => the value 'n' (TTBR) above would be 2 for Linux! - i.e. the first 2 high-order bits of the virtual address would determine whether it's mapped through TTBR0 or TTRB1.

So:
Kernel virtual-address space (>= 0xc0000000) : first 2 high-order bits of a va are always 1, hence map via TTBR1;
User virtual-address space (< 0xc0000000) : first 2 high-order bits of a va are always 0, hence map via TTBR0.

[Note- the Figure 2 above seems to have a typo: in step (3) the bottom paging table should be marked as "TTBR0 base" (and not 'TTBR1 Base') as it inadvertently seems to have been marked.]
>>

Why this extra complexity? This allows one to have a design where the operating system and memory-mapped I/O are located in the upper part of the address space and managed by the page table in TTBR1 and user processes are in the lower part of memory and managed by the page table in TTB0. On a context switch, the operating system has to change TTBR0 to point to the first-level table for the new process.- ~~TTBR0~~ TTRB1 will still contain the memory map for the operating system and memory-mapped I/O. The page table defined in ~~TTBR0~~ TTBR1 encompasses memory that is common to all processes.

<<
Having said all this, there are counter-arguments to the above; take a look at this discussion thread "Linux kernel ARM Translation table base (TTB0 and TTB1)" on stackoverflow.

- See https://lkml.org/lkml/2013/6/26/544 :
"We don't use TTBR1 because the configurable page table splits between
TTBR0 and TTBR1 are not appropriate for Linux kernels. The common
configuration is to have 3GB of userspace and 1GB of kernel space.
However, the TTBR splits supported are 2GB, 1GB, 512MB etc."
- Russel King, June 2013 *(Russel King has been the ARM (Linux) port maintainer for many years).*

<<<
So perhaps this is why the default VM_SPLIT on many ARM platforms seems to be 2 GB : 2 GB !??

Eg. on the ARM Versatile Express (Cortex-A9):
```
$ grep -i VMSPLIT arch/arm/configs/vexpress_defconfig
CONFIG_VMSPLIT_2G=y
$
```
>>>
>>

4.Looking up an address via in-memory page tables is called a translation table walk since it may involve going through a hierarchy of tables. With the ARM MMU, this is a one-step direct mapping via the first-level page table if sections the entry refers to a section or else a two-step process if the

entry refers to a page. With sections, the physical base address is stored in the page table entry of the first-level table. With pages, the page table entry contains the address of the second-level table.

...

Additional Resources Online

- ARM MMU aborts : what exceptions does the ARM MMU raise?
- what is the right way to update (ARM) MMU translation table

---

ARMv8 (Aarch64 / A64) Addressing

Source: ARM Cortex-A Series : Programmer's Guide to the ARMv8-A pg 12-7

...

The table base addresses are specified in the Translation Table Base Registers (TTBR0_EL1) and (TTBR1_EL1). The translation table pointed to by TTBR0[_EL0] is selected when the upper bits of the VA are all 0 << *i.e. user VAS, EL0* >>. TTBR1[_EL1] is selected when the upper bits of the VA are all set to 1 << *i.e. kernel VAS, EL1* >>. You can enable VA tagging to exclude the top eight bits from the check.

The Virtual Address from the processor of an instruction fetch or data access is 64 bits. However, you must map both of the two regions defined above within a single 48-bit Physical Address memory map.

…

Figure 12-4 shows how the kernel space can be mapped to the most significant area of memory and the Virtual Address space associated with each application mapped to the least significant area of memory. However, both of these are mapped to a much smaller Physical Address space.

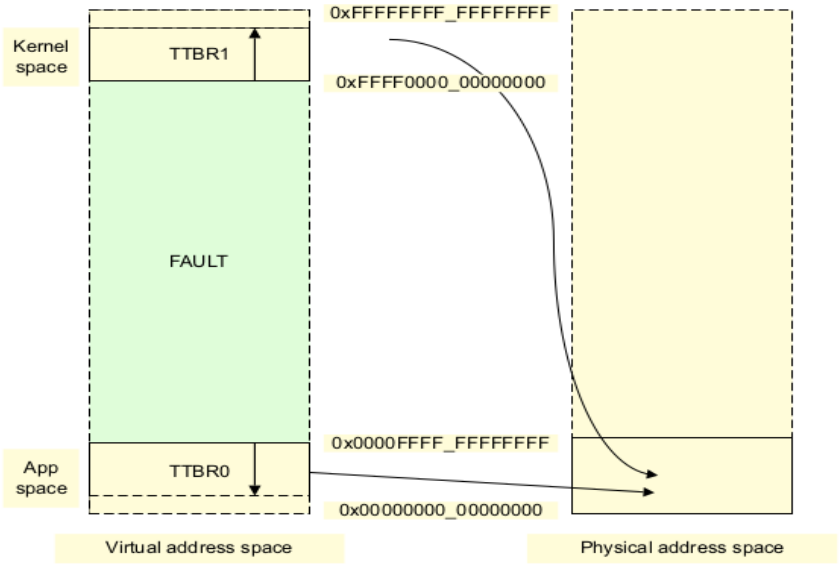


Figure 12-4 Kernel and application memory mapping

SIDEBAR

The above discussion is necessarily platform-specific; using x86 and ARM as the reference hardware (processor).

An excellent resource site on covering the above topics (and more) in great detail with superb diagrams is Gustavo Duarte's blog.

In the same vien, an excellent (though now very old) text on this topic (and in general on the x86) is "Advanced 80386 Programming Techniques" by James L Turley.

Real-world examples of ARM MMU architecture:

- The popular Raspberry Pi SBC uses the Broadcom 8235 SoC based on an ARM11 core (ARM1176jzf-s) (it's an ARMv6 ISA architecture). Details on it's MMU can be found here.
- The TI OMAP3 (ARM Cortex-A8 core): see the "OMAP35x Application Processor : Technical Reference Manual" *Literature Number : SPRUF98T (Rev July 2011)* pages 937-941.

For those interested in low-level architecture and details of **CP-15** (the ARM CoProcessor (logical coprocessor #) 15) **register programming** - which includes MMU protection, cache & TLB management registers, etc - please refer to "ARM system-on-chip Architecture" by Steve Furber, published by Addison-Wesley, Ch 11 "Architectural Support for Operating Systems".

The Intel/HP Itanium-64 (IA-64):

An excellent sample chapter on VM on the IA-64 (from the book " IA-64 Linux Kernel: Design and Implementation" by David Mosberger and Stephanie Eranian) can be found here: "Virtual Memory in the IA-64 Kernel".

Some details on TLB and CPU Cache Management can be found in the Appendices PDF provided.

Architecture-Independent Address Translation on Linux

A similar diagram, showing more detail for the page-table walk.

The diagram below (from the 'ULVMM' book), depicts address-translation for “3-level” paging; today, however, note that the Linux OS supports “4-level” paging tables (mostly to directly support the x86_64 architecture). Thus please visualize an intermediate level of indirection – the “PUD” - between the PGD and the PMD below.

Source: “Understanding the Linux Virtual Memory Manager” << ULVMM >> by Mel Gorman.

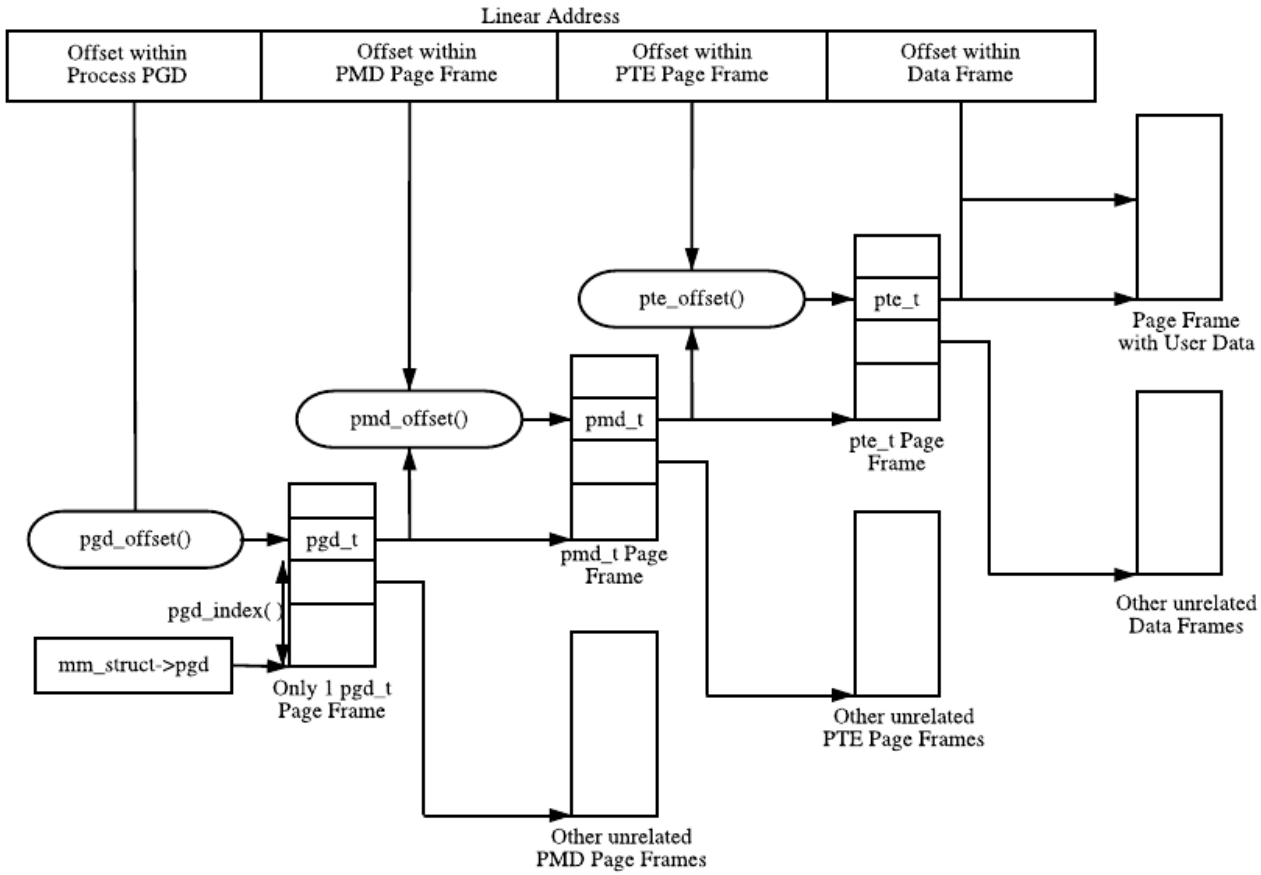


Figure 3.1. Page Table Layout

…

Many parts of the VM are littered with page table walk code, and it is important to recognize it. A very simple example of a page table walk is the function *follow_page()* in *mm/memory.c*. The following is an excerpt from that function. The parts unrelated to the page table walk are omitted.

```
pgd_t *pgd;
pmd_t *pmd;
```

```
pte_t *ptep, pte;

<<----------------
From arch/x86/include/asm/pgtable-2level_types.h:

typedef unsigned long   pteval_t;
typedef unsigned long   pmdval_t;
typedef unsigned long   pudval_t;
typedef unsigned long   pgdval_t;
...

typedef union {
     pteval_t pte;
     pteval_t pte_low;
} pte_t;

--------------->>

pgd = pgd_offset(mm, address);
if (pgd_none(*pgd) || pgd_bad(*pgd))
  goto out;

<<                              Added, taking into account 4-level paging
pud = pud_offset(pgd, address);
if (pud_none(*pud) || pud_bad(*pud))
  goto out;
>>

pmd = pmd_offset(pud, address);
if (pmd_none(*pmd) || pmd_bad(*pmd))
  goto out;

ptep = pte_offset(pmd, address);
if (!ptep)
  goto out;

pte = *ptep;
```

It simply uses the three offset macros to navigate the page tables and the none() and bad() macros to make sure it is looking at a valid page table.

<<

- *Notice this code is architecture-independent*
- *Have a look at actual code here: arch/x86/mm/pageattr.c:*
  `pte_t *lookup_address(unsigned long address, unsigned int *level);`
- One can see similar software address translation techniques in the code of ***crash*** implementing the *vtop (virtual to physical)* command (*in the arch-specific code that implements [k|u]vtop* ***<arch>_[k|u]vtop.c****, like x86_64.c:x86_64_[k|u]vtop, arm.c:arm_[k|u]vtop.c, etc)*
- *Advanced: Security- see* "AnC : Side channeling the MMU for breaking ASLR in the browser."

>>

Sidebar

Examining the process paging table entries (PTEs)

Documentation:
in-kernel: Examining Process Page Tables

Other: *Pagemap Interface of Linux Explained*

Code:

- *GitHub pagemap: Userspace tool to map virtual page addresses to physical addresses.*
  See the README.
- Kernel source: *tools/vm/page-types.c*
  (cd there & do 'make' to build 'em).

==========

<< End of Linux Memory Management, Part 1 >>