Performance-optimised computing – Week 3.

# Caches & Virtual Memory

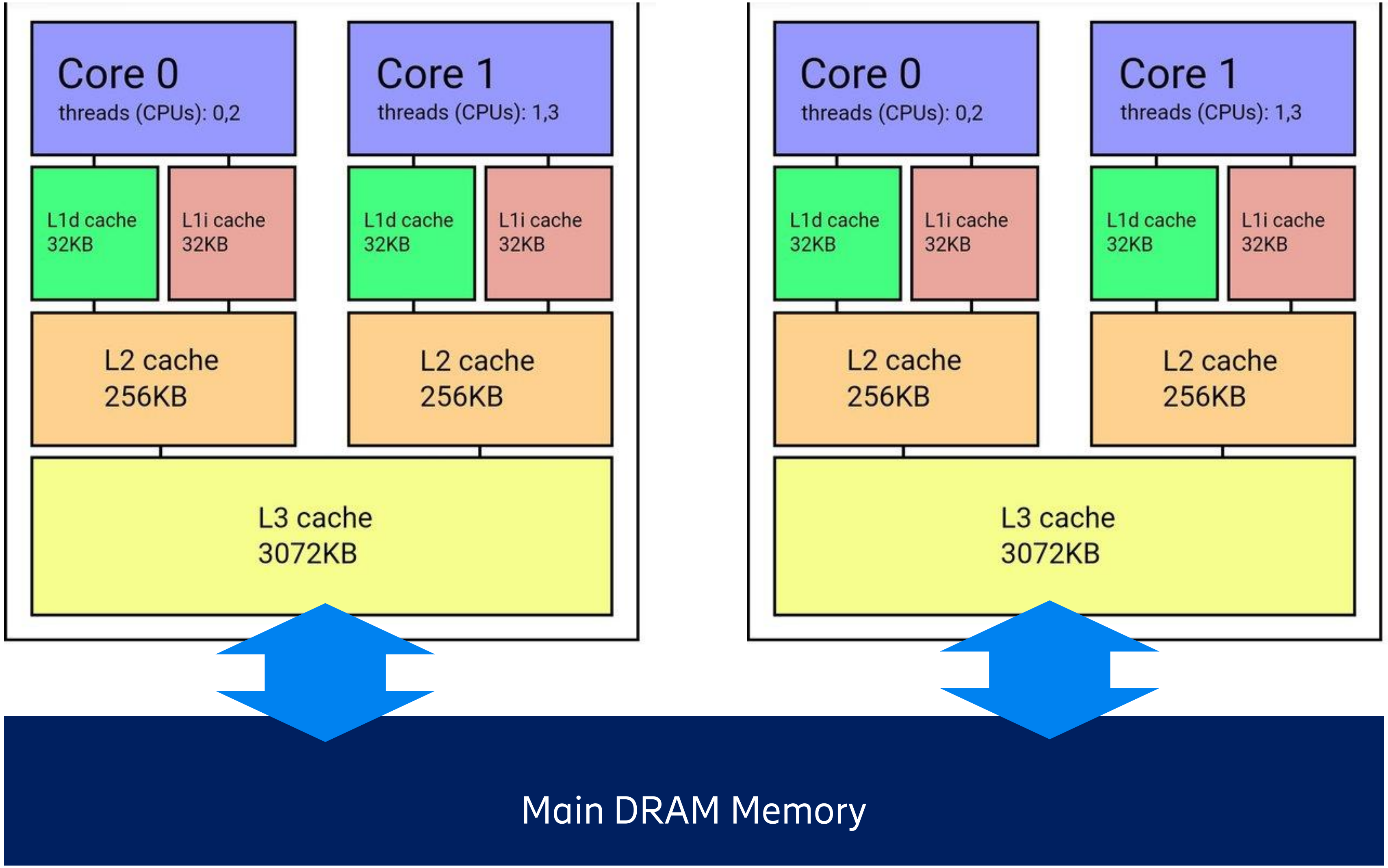
Dr. Bakay Árpád – Ericcson

Fall Semester, 2024

# Announcements

- Write me if you cannot attend the class.
- We can arrange a Teams session if you are sick at home.

# X64 Cache Hierarchy in SMP



# Cache Principal Features

- Width x86/64:    512 bits  (64-bytes)
    - 32 kbyte L1 data cache is only 512 lines!
    - There is no partially loaded cache line -> to read 1 byte from memory,  64 byte reads are required (8 transfers on 64 bit DRAM bus)
- Any data accessible by a CPU Core must pass through ALL levels of caches!
    - And (in most cases) a copy must also be maintained in the lower-level cache
- L1 i / d and L2 are per-core private, L3 (a.k.a Last-Level Cache / LLC) is shared.
- There also exist a few special-purpose caches and other related modules
    - TLB: Translation Lookaside Buffers -> see later
    - IMC: Integrated Memory Controller – not a cache!
    - QPI: Quick Path Interconnect for IO - not a cache!

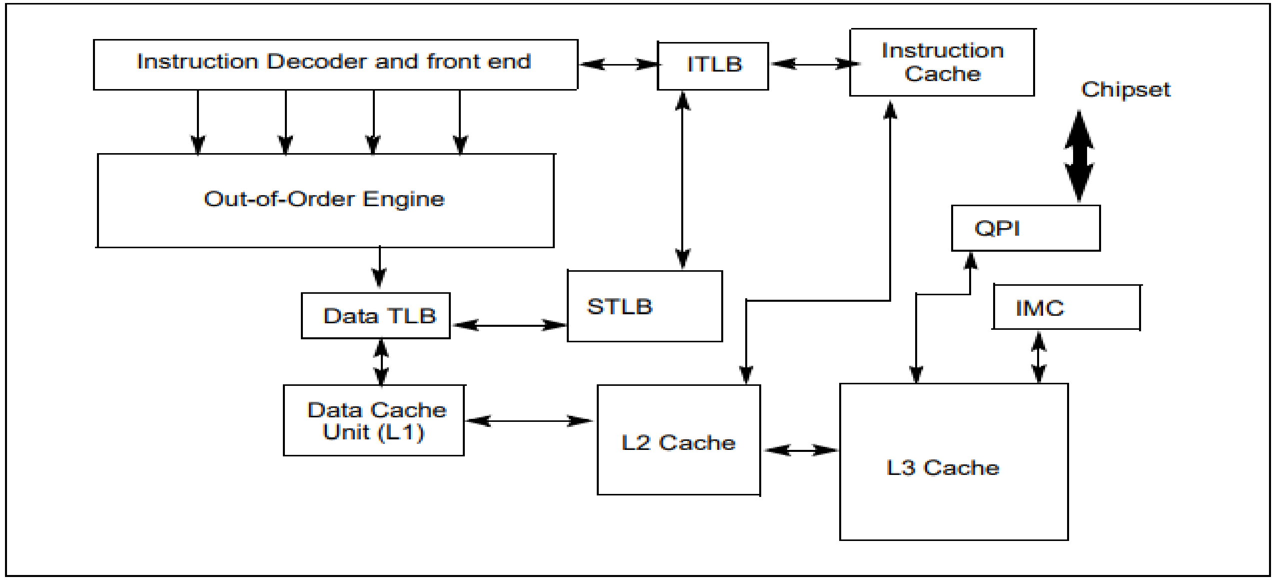
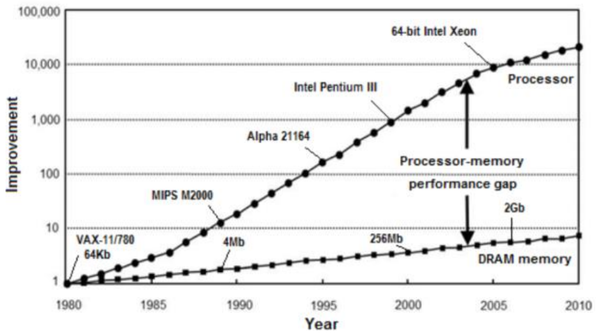



Figure 11-2.  Cache Structure of the Intel Core i7 Processors

# The CPU Caches Compared to other methods of Data Storage




- Modern x64 CPU-s have 3 levels of caches. Why?
- Fast cache memory exists, but expensive & extensive -> only small size is affordable

| Type | Typical size in 2024 | Access time (CPU cycles) | Scope |
|---|---|---|---|
| Generic Register | 16 (300) x64bits | 1 | Hyperthread |
| AVX512 Vector reg | 32 (200) x512 bits | 1 | Hyperthread |
| L1 cache | 2 x 32 kbytes (instr,data) | 3-4 | Core |
| L2 cache | 512 kbytes (instr, data) | 10-12 | Core |
| L3 Cache | 10-30 Mbytes | 50-60 | CPU |
| DRAM Memory | Up to 1 TB | about 100-200 (like 50 ns) | Machine |
| NVMe | Up to 8 TB | 10-20 µs, 2.6 Gbps | Machine |
| SSD Drive (SATA) | Up to 8 TB- (200 kHUF) | 100 µs, 600 MBps | System |
| HDD Drive | Up to 20 TB (120 kHUF) | 5-9 ms, 150-200 MBps | System |

# Cache Size Evolution

- ‚Sandy Bridge‘ **2011**:
  - L1i/d:  32+32 kBytes
  - L2: 256 kBytes  (4-way)
  - L3: 3-20 Mbytes

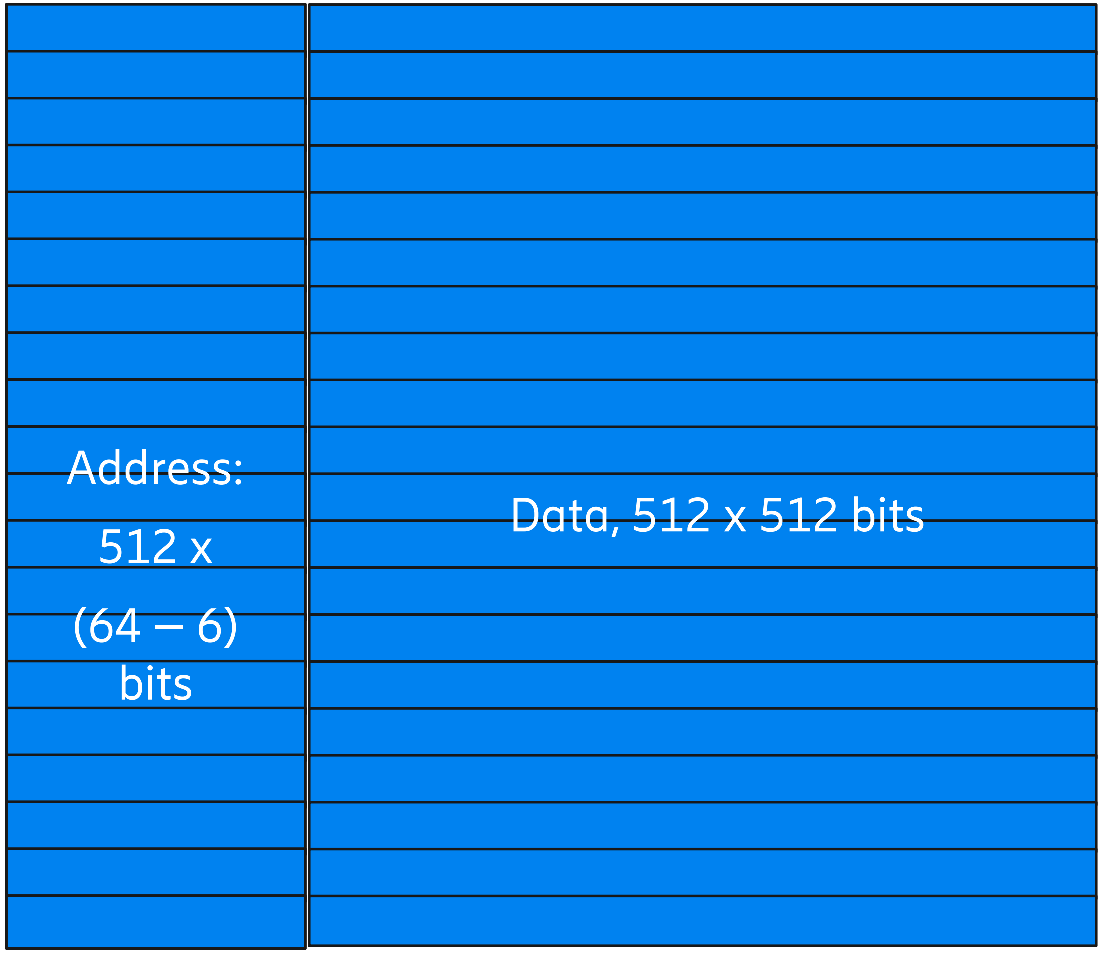
- ‚Raptor Cove‘ **Oct 2022,** and **‚Emerald Rapids‘ Dec 2023**
  - L1i/d:  32+48 kBytes  (with 8-12way associativity)
    - „Redwood Cove“ **2023 dec** -> 64+48 kBytes
  - L2: 2 Mbytes   (16-way)
  - L3: 36 Mbytes  or 5MBytes/Core*64Core

# Caches are actually bigger then their size

## e.g.: 32k L1 Data Cache

„Content Addressed Memory"

Is address ‚X' present at any line?



# Cache Tricks & Issues

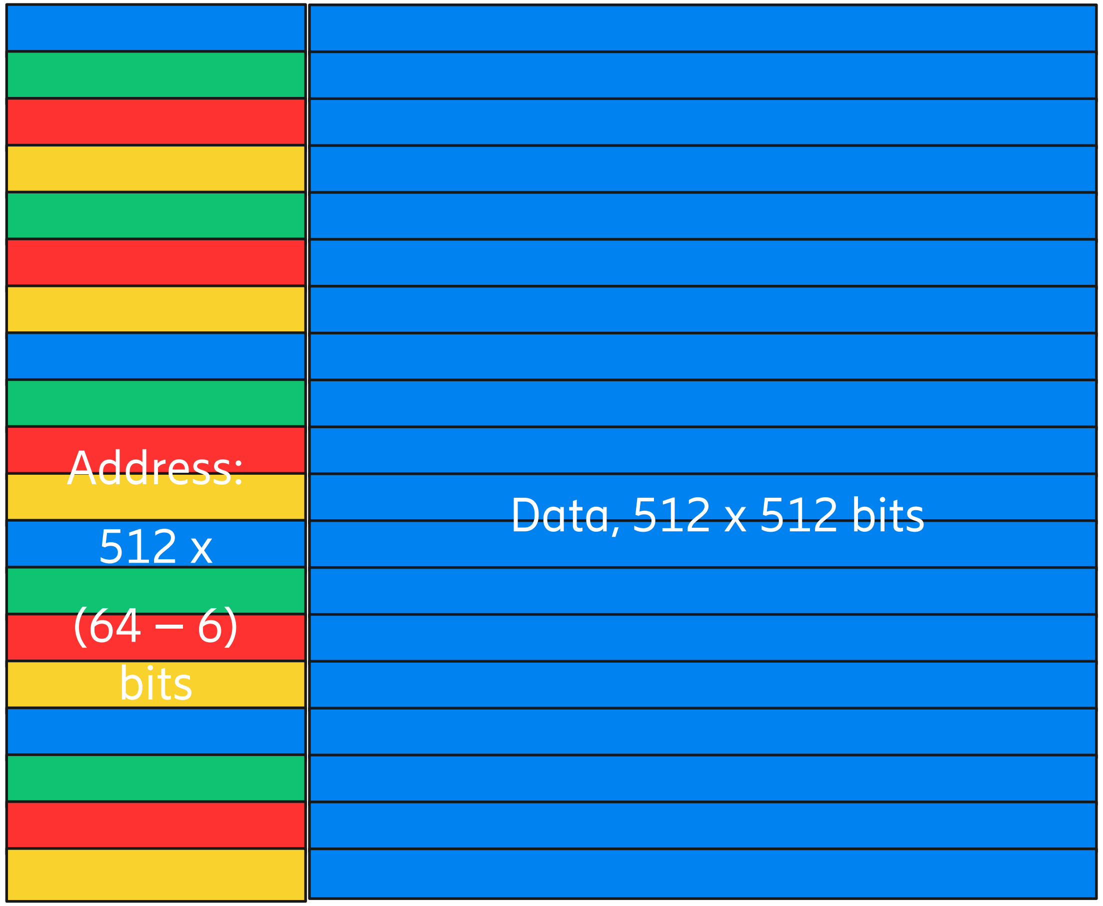
- **Associativity** -> how many slots a certain address can be placed into?
  - 1: „Direct mapped“, >= 2: „N-way associative“, any slot: „Fully associative“
- **Prefetching**: expect data needs in the future
  - Can leverage DRAM „row select“
- **Replacement policies**
  - E.g. LRU or random
- **Upon updates of cached data**: „write through“ immediately (simple) or „write back“ at a later time (better for multiple writes)
- **Inclusive/non-inclusive/exclusive** -> shall lover caches also contain data cached above?
- **Cache coherence**: changing data shall make sure others do not cache the same address.
  - Send update all other caches or broadcast invalidate request
  - Try to colocate processes/threads with shared access

# Caches are actually bigger then their size

e.g.: 32k L1 Data Cache

4-Way associative cache

Only N/4 lines are to be checked



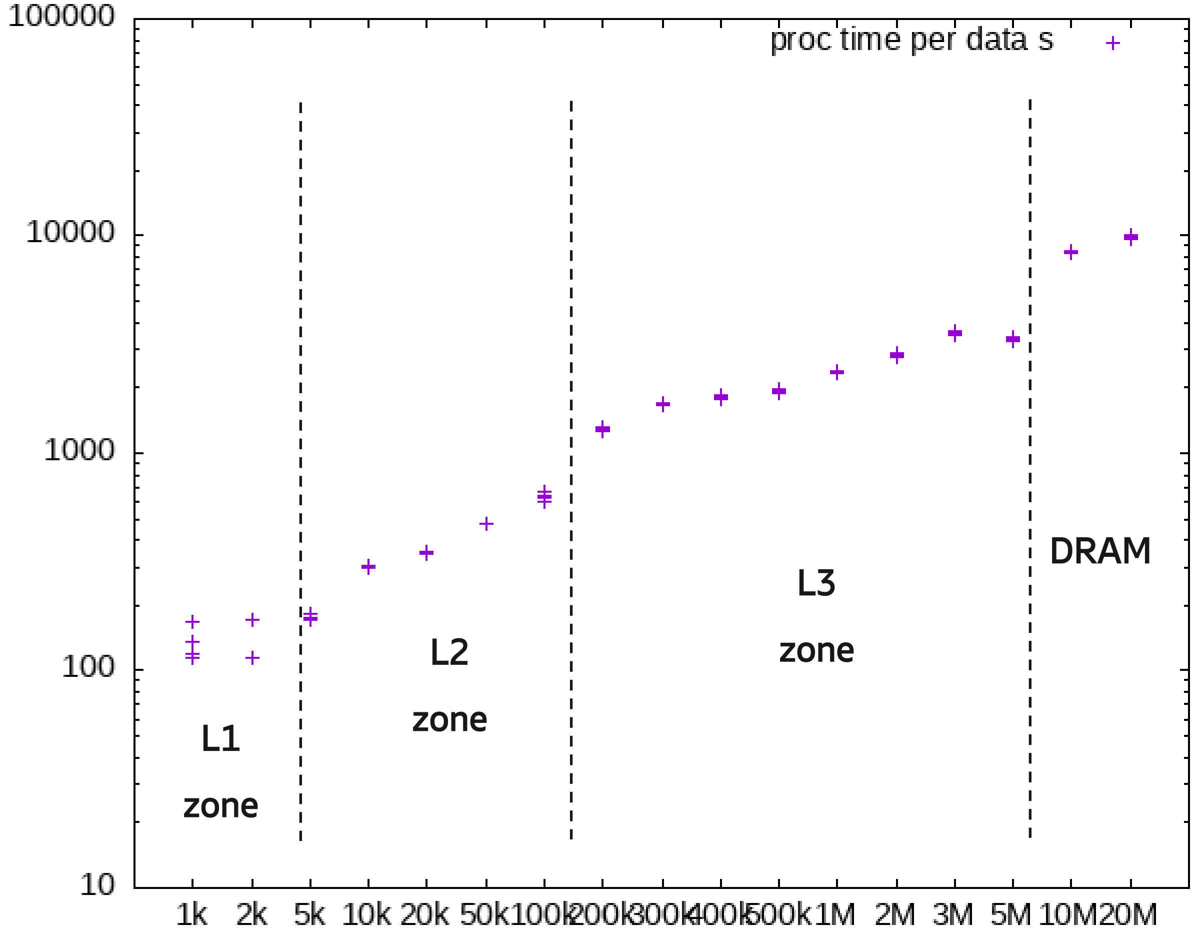
# Cache Performance Test program

- Originally from: [A Survey of CPU Caches (meribold.org)](https://meribold.org) (2017)
- [https://gitlab.inf.elte.hu/-/snippets/31](https://gitlab.inf.elte.hu/-/snippets/31)

- Idea: create a random linked, but circular list where the loop SIZE is changed. Then traverse the elements a fixed (e.g 100M) number of steps.
  - Small circle: mostly uses L1 cache
  - Large circle: uses lower level caches or DRAM

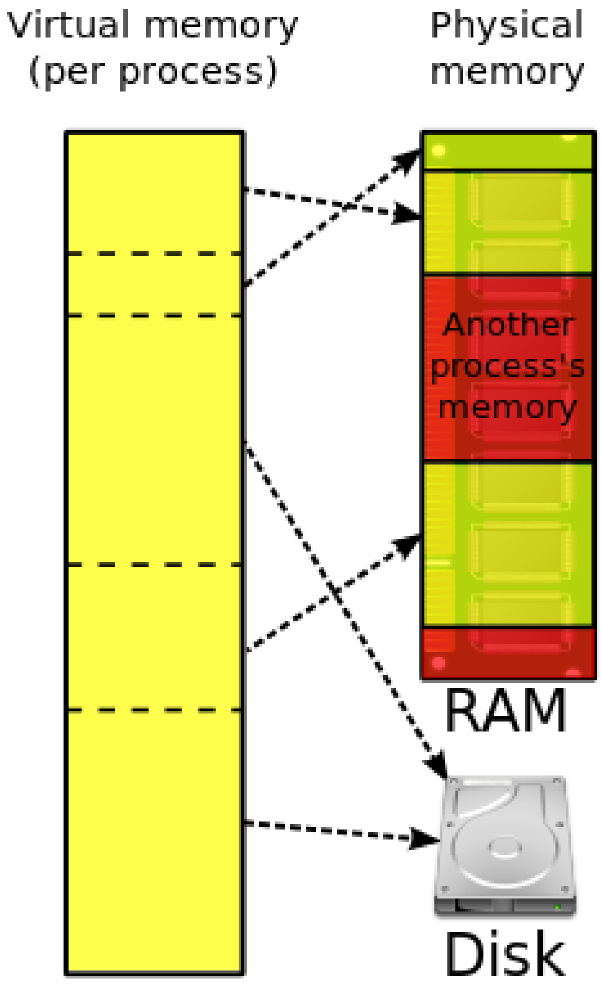
- Enhancements
  - „Wrapper script" to
    - Run multiple tests with various SIZE-s: 1k, 2k, 5k, .... 10M, 20M
    - Use gnuplot to create a „scatter chart„

# Test results with Meribold test program

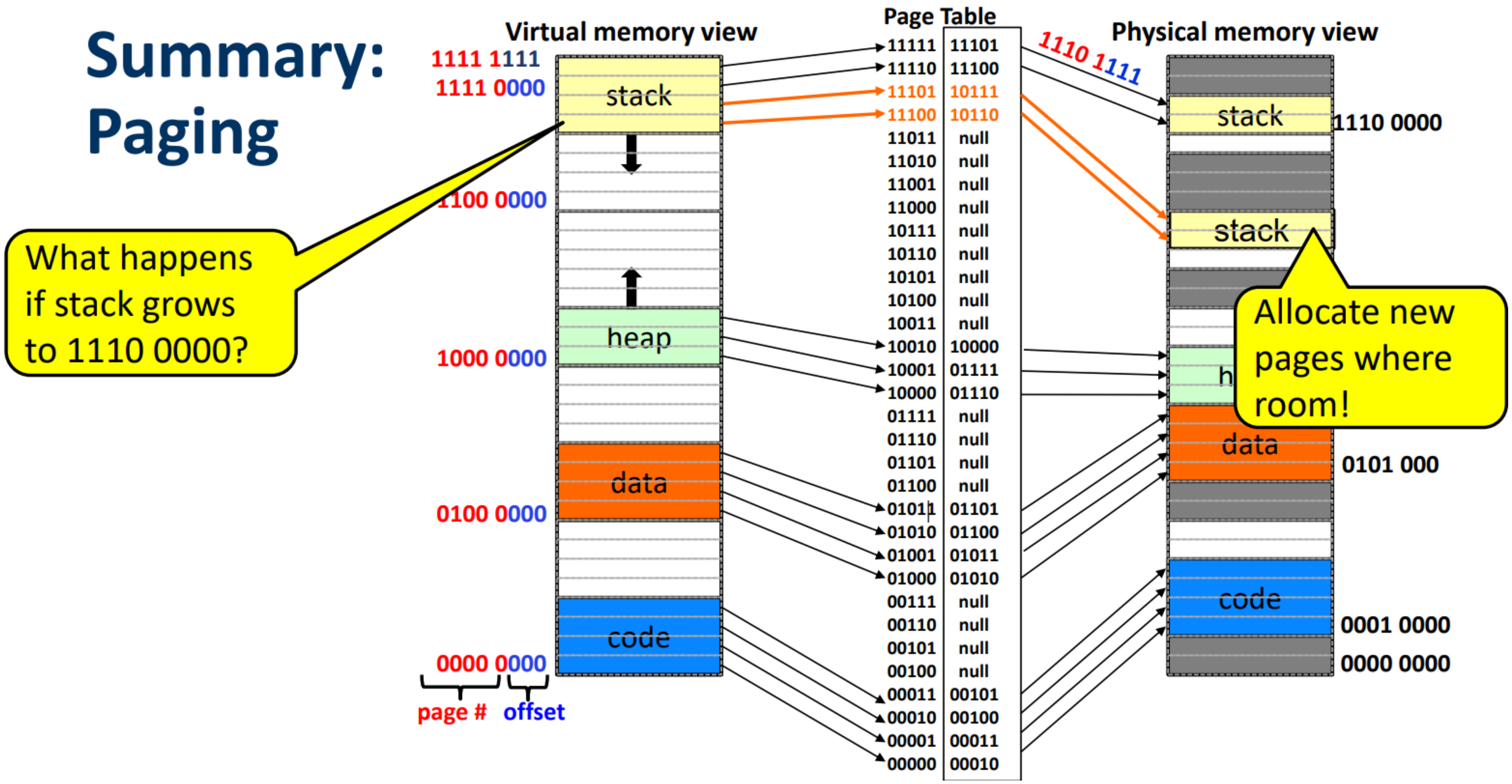


# 2. Virtual memory

- Logical addresses seen by programs differ from physical addrs used on memory bus
- Advantages:
  - Extend limits of address space
    - Extend physical mem: some memory regions „swapped" on disk
    - Or: 32-bit programs on a server with more than 4G physical memory.
  - Provide processess with dedicated/isolated/secure address spaces
  - Or explicitly provide shared memory (visible by multiple processes at the same address), for shared code (dynamic libraries) or data (inter-process communication)
- Implementation requires:
  - **Memory Management Unit - MMU**
  - **Page tables** for virtual-to-physical mapping
    - Separate mappings per process, but shared by threads
    - Basic pages are 4kBytes (12 bits). Last 12 bits of address is the offset within the page, the rest (20 or 52) bits are looked up from the page table.

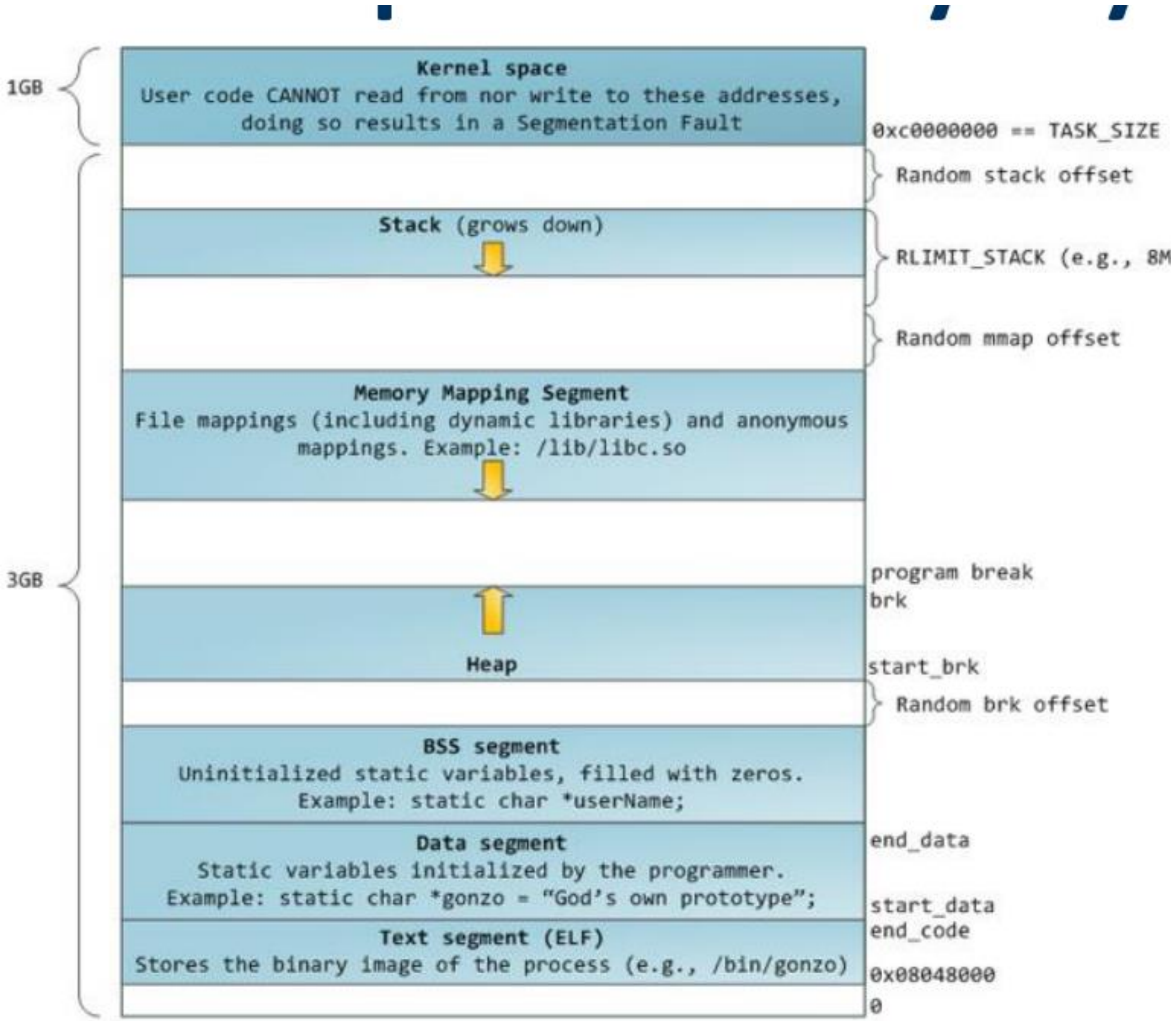


# Overview of virtual memory and paging




Source: https://inst.eecs.berkeley.edu/~cs162/su20/static/lectures/17.pdf

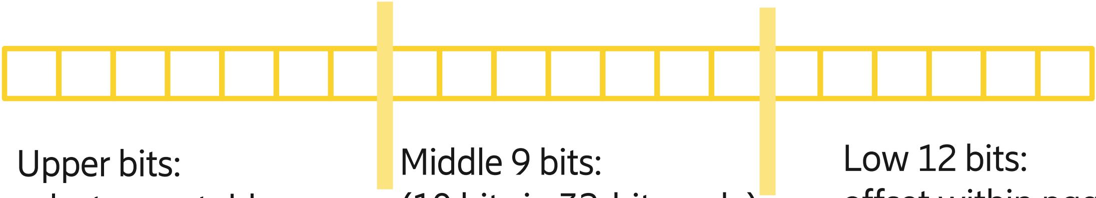
# Linux Practice - Process Address Space




- Example for 32bit - 4 Gbytes
- Stack, Heap, BSS, Data & Text are standard process-private areas
- Memory mapping is used for shared data, including shared libs.
- Most of kernel space is inaccessible for process

# Page Table and Memory Management Unit

Virtual address in memory access request, is split to byt



Upper bits:
select page table
from „PT array"

Middle 9 bits:
(10 bits in 32-bit mode)
select offset in Page Table

Low 12 bits:
offset within page

**MMU Tasks**

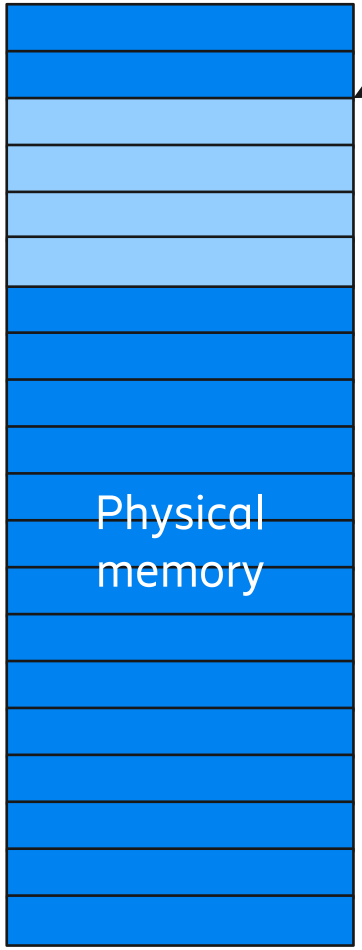
1. Cut address
2. Select right PT
3. Read real address from PT
4. Create physical address by adding offset
5. Read requested memory

Memory Management Unit

MMU

**Repeated for each request! ... almost**

4096 byte pages




PT start pointer

(One per process)

Page tables pages, each holding 512

8-byte adresses

(or 1024 4-byte addrs for 32 bit mode)

Normal memory pages

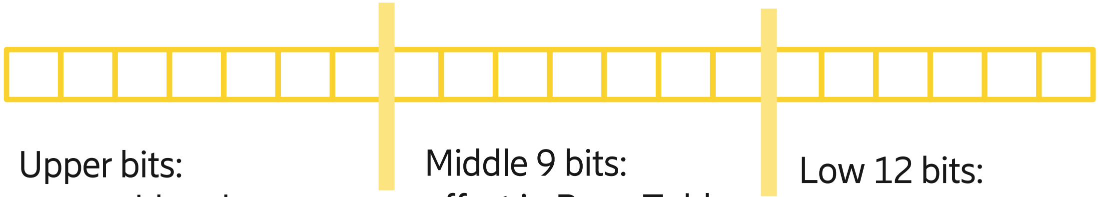
# MMU - Memory Management Unit

- A principal unit of all modern CPU-s.
- Main task is exactly to translate virtual addresses to physical.
- Typically there is a portion of the phys mem (like lower 1G), which is directly addressed
  - I.e. no virtual translation there.
  - This is used by the kernel, unaccessible by processes
- Uses a large number of **Page Tables**, which are <u>also stored in the main memory</u>
  - Page tables are stored in the kernel's private, directly addressed space
  - With 4096-byte pages, this amounts to 0.2-0.3% of total virtual address space in use by any process (8bytes/4096 bytes + some overhead).
  - With swap, shared, mem, etc. this may add up to 1-3% of physical memory.
- MMU cooperates with the kernel:
  - When a process is activated, its page tables are activated in MMU by the kernel through PT base address.
  - When a user program refers to memory not yet mapped (or mapped but swapped), a **page fault interrupt** is handled by the kernel to set up new virt->phys mapping. This may require eviction on an existing mapping (to disk), possibly of another process.
- MMU also provides access control (e.g. read/write/execute bits in page table)

# Page Table and Memory Management Unit

## Caveats

Address in memory access request



Upper bits:
page table selector

Middle 9 bits:
offset in Page Table

Low 12 bits:
offset within page

**,Upper bits' are 10 bits for 32bit mode...**

(1024 pages / 4Mbytes needed for PT per process – hardly tolerable)

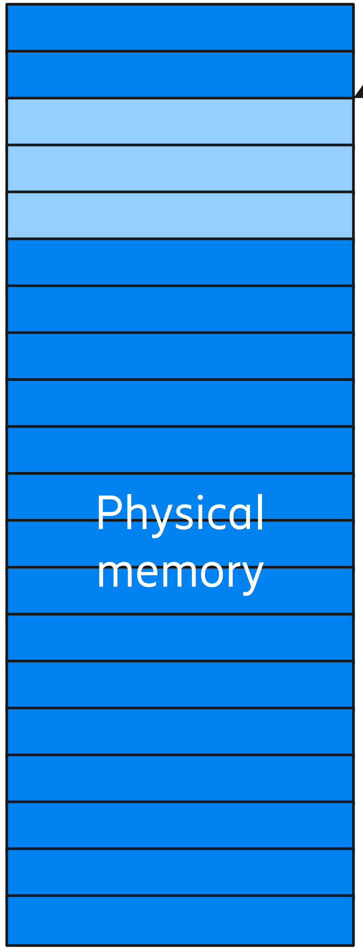
**...and 43 bits for 64 bit!!!**

***(Not tolerable!!!)***

Memory Management Unit

MMU

4096 byte pages




PT start pointer

(One per process)

Page tables pages, each holding 512 8-byte adresses

**Separate page table for each process!!!**

Need to change PT start pointer at each context switch

(This is tolerable)

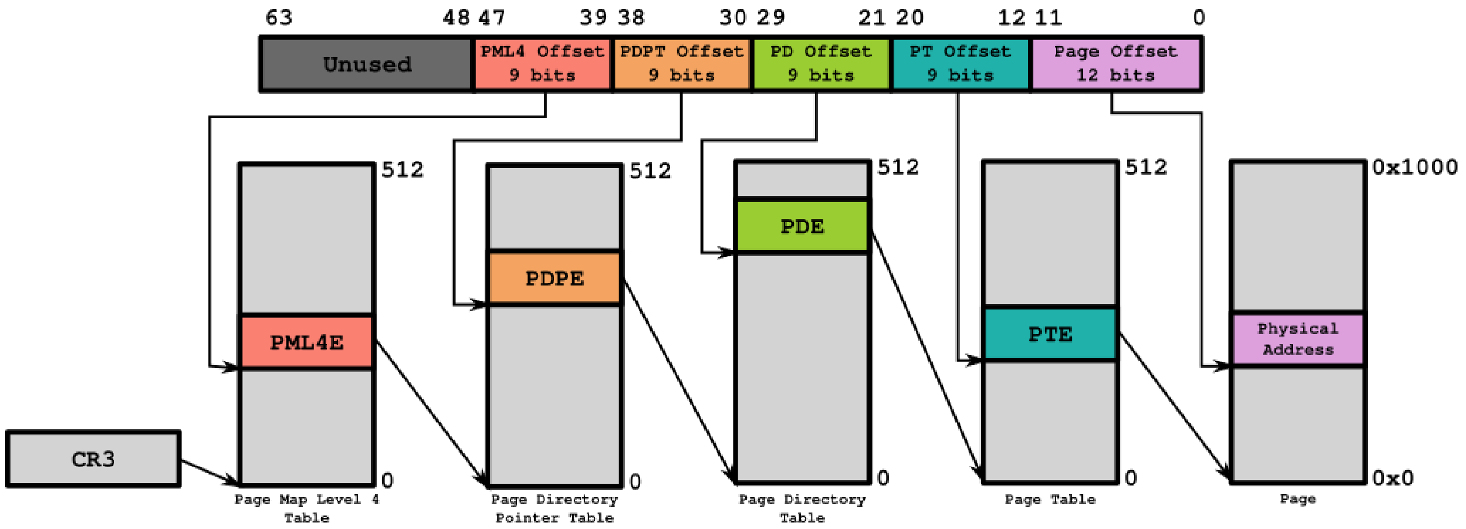
# Solution: Multi-level Page Tables

- Virtual address space is typically **very sparse:** a small process may only have <10 used items
- Full PT coverage would be very big: $2^{52}$ items on 64 bit (per process)
  - **2-level Page Table works fine on 32 bit…**
    - Bits: 31-22: are looked up from the **„page directory"** (1024 4 byte entries), to select a **page table**
    - (PT pointers is are non-zero in PD for only those few of the 1024 items, where needed)
    - Bits 21-12: to look up physical page address from the PT selected above
    - Bits 11-0: direct „offset" address within a page
    - A minimal process needs only 1xPD + a few PT-s (for text, heap, stack etc.)
  - **…but 64-bit requires 4-level Page Tables**
    - 4k page can hold 512 8-bytes entries
    - Page levels for bits 47-39, 38-30, 29-21,20-12 -> 48 bit only! See next slide…

- **Muti-level page translation is a performance pain:** 2-4 memory accesses for each ‚real' access
  - **Solution: „Translation Lookaside Buffer" (TLB):** another „cache" for page tables (again content addressed)

# Virtual address mapping on 64 bit for 4096 byte pages

Source: [Exploring Virtual Memory and Page Structures (xenoscr.net)]




- CR3 is a CPU register, used as „Master Paging Pointer". It is changed at every process switch.
- All PML4, PDP, PD, and Page tables are stored in kernel memory, maintained by the kernel
- „Page" (on the rigth) is the part of phys memory where the process has its real data.
- Upper 16 bits of 64 bit virtual addresses are not used / ignored
  - It is OK, because currently all processes are happy with 2^48bytes -> 260 terabytes

# Example: Calculate the Pages and Page Tables Needed for a Process

| Segment name | virtual address range | size | Pages needed |
|---|---|---|---|
| • Text (code) segment | 0x600000000000 – 0x60000318A512 | ~ 52Mbytes -> | 12682 |
| • Data segment: | 0x400000000000 – 0x40000152143B | ~22Mbytes -> | 5410 |
| • Stack: | 0x300000000000 – 0x30000001A000 | - 104 kbytes -> | 26 |
| • Heap: | 0x800000000000 – 0x8001B3424000 | - 7.3 Gbytes - | 1 782 820 |

**Page Tables needed:**

$\text{roundUp}_{512}(12682) / 512 + \text{roundUp}_{512}(5410) / 512 + \text{roundUp}_{512}(25) / 512 + \text{roundUp}_{512}(1782820) / 512 =$

$25 + 11 + 1 + 3483 =$ **3520 PT-s**

**Page Directories needed:** $1 + 1 + 1 + \text{roundUp}_{512}(3483) / 512$ **= 9 PD-s**

**Page Directory Pointer Tables (PDPT)** needed: **4**

**Page Map Level 4 Table** needed: **1** (always)

**Overhead:** 3534 4k tables -> 14.8 Mbytes **- 0.2 % of the 7.4 Gbytes used**

**Max possible counts:** (for 48bit phys addr): 1.05M PT-s, 260k PD-s, 512 PDPT-s, 1 PML4T

**TLB sizes:** (for latest Golden Cove Core): **256** L1-Instr TLB + **96** L1-Data TLB + **2048** L2 TLB,

# Virtual Memory Mapping – Further Details
## Access Control, Page Fault, Process Swithches

- For efficiency, PT entries are 64 bits on X64 although only 52 bits are needed (as the lower bits select the address within a page)
    - The remaining PT bits are **used for Access Control:** RO/RW/Execute, „dirty”, „available”, etc.
    - Access control is **another important function of the MMU!**

- **Page fault:** when a virtual address without live physical mapping exists
  - This may be A. a page never addressed before, or a page swapped to the disk
  - This is Handled by OS
    - Select a physical page for this virt page
      - If there is no free page, swap a page to storage, and use that one
    - Enter new item into PT page (may require a new PT -> add entry to PD as well)
    - Prepare page: clear (for security reasons), or load swapped data from disk

- Tasks required **process switches** (scheduling is discussed in next class)
  - MPP/CR3 needs to be updated (kernel records this for each process).
  - **Plus:** all TLB-s need to be flushed!!! <- performance loss.

# New Trend: Choice of multiple Page Sizes

- 4kbyte + 2Mbyte + 1Gbyte pages - LARGE / HUGE PAGES
  - 64 bit virtual address space is segmented into regions of different page sizes
  - Direct addressing may also be available for some ranges

- This requires advanced MMU

# That's it for today...